ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN

COMPUTER SCIENCE AND ENGINEERING

CICLO 35

Settore Concorsuale: 09/H1 - Sistemi di elaborazione delle informazioni
Settore Scientifico Disciplinare: ING-INF/05 - Sistemi di elaborazione delle informazioni

Deploying and Processing Neural Representations of Signals

Presentata da:
Luca DE LUIGI

Coordinatore di Dottorato:
Ilaria BARTOLINI

Supervisore:
Luigi DI STEFANO

ESAME FINALE ANNO 2023

UNIVERSITÀ DI BOLOGNA

Abstract

Facoltà di Ingegneria ed Architettura
Dipartimento di Informatica - Scienza e Ingegneria

Dottorato di Ricerca

**Deploying and Processing
Neural Representations of Signals**

by Luca DE LUIGI

Neural representations (NR) have emerged in the last few years as a powerful tool to represent signals from several domains, such as images, 3D shapes, or audio. Indeed, deep neural networks have been shown capable of approximating continuous functions that describe a given signal with theoretical infinite resolution. This finding allows obtaining representations whose memory footprint is fixed and decoupled from the resolution at which the underlying signal can be sampled, something that is not possible with traditional discrete representations, *e.g.*, grids of pixels for images or voxels for 3D shapes. During the last two years, many techniques have been proposed to improve the capability of NR to approximate high-frequency details and to make the optimization procedures required to obtain NR less demanding both in terms of time and data requirements, motivating many researchers to deploy NR as the main form of data representation for complex pipelines. Following this line of research, we first show that NR can approximate precisely Unsigned Distance Functions, providing an effective way to represent garments that feature open 3D surfaces and unknown topology. Then, we present a pipeline to obtain in a few minutes a compact Neural Twin® for a given object, by exploiting the recent advances in modeling neural radiance fields. Furthermore, we move a step in the direction of adopting NR as a standalone representation, by considering the possibility of performing downstream tasks by processing directly the NR weights. We first show that deep neural networks can be compressed into compact latent codes. Then, we show how this technique can be exploited to perform deep learning on implicit neural representations (INR) of 3D shapes, by only looking at the weights of the networks.

Publications

[1] Luca De Luigi*, Ren Li*, Benoît Guillard, Mathieu Salzmann, and Pascal Fua. *DrapeNet: Generating Garments and Draping them with Self-Supervision*. Accepted at the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) 2023.

[2] Luca De Luigi*, Damiano Bolognini*, Federico Domeniconi*, Daniele De Gregorio, Matteo Poggi, and Luigi Di Stefano. "ScanNeRF: A Scalable Benchmark for Neural Radiance Fields". In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*. Jan. 2023, pp. 816–825.

[3] Gianluca Berardi*, Luca De Luigi*, Samuele Salti, and Luigi Di Stefano. "Learning the Space of Deep Models". In: *2022 26th International Conference on Pattern Recognition (ICPR)*. IEEE. 2022, pp. 2482–2488.

[4] Luca De Luigi*, Adriano Cardace*, Riccardo Spezialetti*, Pierluigi Zama Ramirez, Samuele Salti, and Luigi Di Stefano. *Deep Learning on Implicit Neural Representations of Shapes*. Accepted at the International Conference on Learning Representations (ICLR) 2023.

[5] Pierluigi Zama Ramirez, Claudio Paternesi, Luca De Luigi, Luigi Lella, Daniele De Gregorio, and Luigi Di Stefano. "Shooting labels: 3D semantic labeling by virtual reality". In: *2020 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*. IEEE. 2020, pp. 99–106.

[6] Adriano Cardace, Luca De Luigi, Pierluigi Zama Ramirez, Samuele Salti, and Luigi Di Stefano. "Plugging Self-Supervised Monocular Depth into Unsupervised Domain Adaptation for Semantic Segmentation". In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2022, pp. 1129–1139.

[7] Pierluigi Zama Ramirez*, Adriano Cardace*, Luca De Luigi*, Alessio Tonioni, Samuele Salti, and Luigi Di Stefano. "Learning Good Features to Transfer Across Tasks and Domains". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2023).

[8] Simone Magistri, Marco Boschi, Francesco Sambo, Douglas Coimbra de Andrade, Matteo Simoncini, Luca Kubin, Leonardo Taccari, Luca De Luigi, and Samuele Salti. "Lightweight and Effective Convolutional Neural Networks for Vehicle Viewpoint Estimation From Monocular Images". In: *IEEE Transactions on Intelligent Transportation Systems* (2022).

[9] Marco Boschi, Luca De Luigi, Samuele Salti, Francesco Sambo, Douglas Coimbra de Andrade, Leonardo Taccari, and Alex Quintero Garcia. *Dynamic Bird's Eye View Reconstruction of Driving Accidents*. Under Review at IEEE Transactions on Intelligent Transportation Systems.

Acknowledgments

I would like to start by thanking my supervisors, Prof. Luigi Di Stefano and Prof. Samuele Salti, who gave me the chance to pursue my PhD in the CVLab, a possibility for which I will always be extremely grateful. They were always there for me, supporting me with an inspiring and contagious passion for their work. I couldn't have asked for better supervisors.

I also want to thank all the people in the CVLab, who fought with me through the darkest moments and were ready to laugh together in the lightest ones.

I am truly grateful to Prof. Pascal Fua for having me in his lab at EPFL for six months, and to Mathieu Salzmann, Benoît Guillard and Ren Li for the fruitful collaboration during that period. A warm thank you goes to all the people in EPFL-CVLab: it has been really nice to be part of such a beautiful group, both in the office and outside.

The rest of the acknowledgments will be in Italian, for my family and friends.

Un immenso grazie a tutti i membri della mia famiglia. Il supporto che mi forniscono costantemente mi ha permesso di superare le sfide del dottorato con grande serenità, fiducioso del fatto che sarebbero sempre stati pronti ad affrontare con me gli inevitabili fallimenti ed a festeggiare insieme i traguardi raggiunti.

Ringrazio con grande affetto anche tutti gli amici, quelli che abitano qui con me a Bologna, quelli che vivono sparpagliati per la valle del Reno e tutti gli altri sperduti in giro per il mondo.

Il mio ultimo ringraziamento, di certo non meno importante, va a Chiara, Margherita e Mattia, per aver portato nella mia vita un'ondata di felicità, il carburante di cui avevo bisogno per concludere il dottorato.

Contents

Chapter 1

Introduction

1.1 Neural Representations of Signals

Every computer-based application that interacts with the world surrounding us requires a powerful and efficient way of representing data. As a matter of fact, these applications usually involve the synthesis, estimation, manipulation, display, storage, and transmission of data about objects and scenes across space and time.

To name some examples, we consider that in Computer Vision data such as images, videos, and 3D information are collected from specialized sensors – *e.g.*, cameras and lidars – and are then analyzed to obtain information about the underlying scene. Images can be classified [10] or used to detect and segment the objects in the scene [11, 12]. Video sequences can be used to estimate the trajectory of the camera [13] or to reconstruct the 3D geometry of the environment [14]. 3D data, instead, can be aggregated to recover complete 3D models of objects [15], can be classified and segmented [16], and can be used to complement the visual information from images [17].

Apart from Computer Vision, many other applications require to process and/or generate 2D and 3D data. For instance, we consider Computer Graphics – where we are interested in synthesizing 3D shapes and scenes, and in rendering novel views of them [18] – or Robotics – where the structure of the 3D environment is used to plan and execute actions [19].

The most common structures adopted to represent the input signals typically involve some form of discretization, despite their continuous nature. The obvious example of such discretization is the 2D grid of pixels used to represent images, whose resolution impacts the number of details captured in the image, with a critical trade-off between quality and storage requirements. Things are more complicated when dealing with 3D data, with many different forms of discrete representation coexisting, such as, primarily, voxel grids, point clouds and triangle meshes. Voxel grids can be seen as the generalization of pixels to 3D data. While they enable to process voxels by simple extensions of the 2D machinery, the memory footprint of voxel representations grows cubically with the resolution, hence limiting naive implementations to mainly 32^3 or 64^3 voxel grids. Point clouds avoid wasting memory to represent the empty space and store only the 3D coordinates of surface points within an

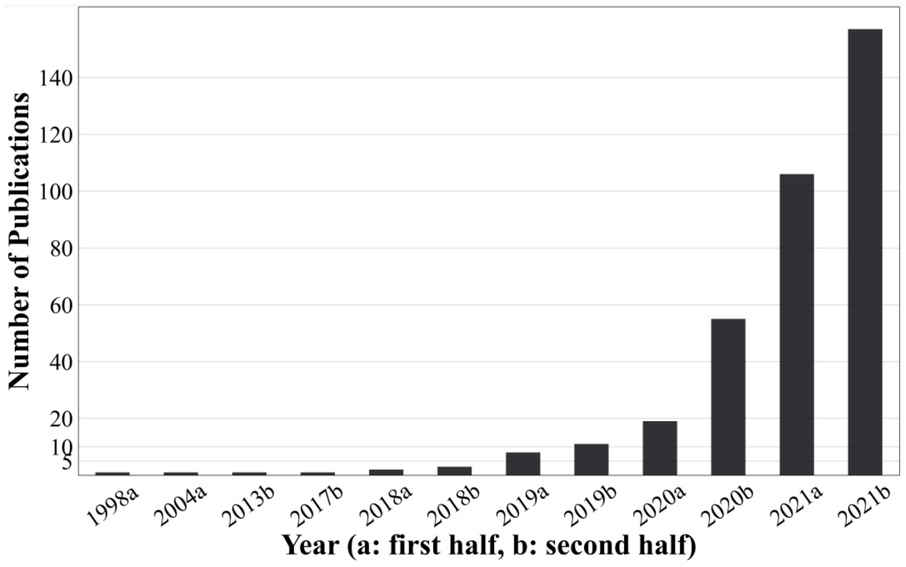


FIGURE 1.1. **Number of publications related to NR (from [20]).** Neural representations were proposed two decades ago [21], yet their growth in visual computing has been concentrated in the last two years with over 250 papers.

unorganized data structure. Even if convenient in terms of memory requirements, this kind of representation lacks the connectivity structure of the underlying surface, resulting in an undesired loss of information. Triangle meshes represent a good trade-off in terms of memory requirements and connectivity information, but are still far from optimal when it comes to ease of processing.

A new form of representation has been proposed in the past few years. Following the many successes of deep learning, it has been shown that, given enough capacity, fully connected neural networks can encode continuous signals of arbitrary dimensions at arbitrary resolution. This leads to the possibility of using a multi-layer perceptron (MLP) to fit a continuous function that represents a signal of interest. Such MLP is trained to predict the value of the function of interest – *e.g.*, the RGB color of a point on the image plane – when queried with the continuous coordinates of a point in the input domain – *e.g.*, the 2D coordinates of the image point. We will refer to this form of representation as Neural Representations (NR).

NR have been successfully deployed with images [22], videos [23], audio [22], radiance fields [24] and 3D shapes represented as *signed distance functions* [25], *unsigned distance functions* [26] and *occupancy fields* [27]. There are several potential advantages when considering NR as a substitute for discrete representations. First of all, the memory footprint of a NR is given by the number of weights of the adopted MLP. Once the capacity of the network has been properly calibrated with respect to the complexity of the target signal, the storage requirement of the NR is fixed and decoupled from the spatial resolution at which the underlying signal can be sampled, which is theoretically infinite. This is true for all the mentioned signals, but perhaps the most convincing example concerns NR fitting radiance fields [24], where a single MLP can be used to generate infinite novel views of a given scene.

Due to their effectiveness and potential advantages over traditional representations, NR are gathering ever-increasing attention from the scientific community, with striking improvements in the NR quality [22, 28, 29, 30, 31, 32] and less demanding data and time requirements [33, 34, 32]. The increasing interest in NR is testified by the concentration of publications on related topics in the last two years, in spite of NR being originally proposed two decades ago [21], as reported in [20] and presented in Fig. 1.1.

In the next sections we present the NR that we adopted throughout the thesis, namely Implicit Neural Representations (INRs) for 3D shapes and Neural Radiance Fields (NeRF).

1.2 Implicit Neural Representations for 3D Shapes

Implicit Neural Representations (INRs) have emerged a few years ago as an effective tool to represent 3D surfaces whose topology is not known a priori.

In its general formulation, an INR works by training a MLP to fit a continuous function $f : \mathbb{R}^{in} \rightarrow \mathbb{R}^{out}$. To do so, a training set composed of N points $\mathbf{x}_i \in \mathbb{R}^{in}$ with $i = 1, 2, ..., N$, paired with values $\mathbf{y}_i = f(\mathbf{x}_i) \in \mathbb{R}^{out}$, is exploited to find the optimal parameters θ^* for the MLP that implements the INR, by solving the optimization problem:

$$\theta^* = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \ell(\mathbf{y}_i, f_\theta(\mathbf{x}_i)), \tag{1.1}$$

where f_θ represents the function f approximated by the MLP with parameters θ and ℓ is a loss function that computes the error between predicted and ground-truth values.

The seminal work DeepSDF [25] proposed to cast the function f to the *signed distance function* (SDF) of a 3D closed surface. The SDF is a continuos function that, for any 3D point, gives its signed distance to the closest point on the surface, with negative sign if the input point is inside the surface or positive sign otherwise. It follows that the surface is *implicitly* described by the level set $\text{SDF}(\cdot) = 0$.

Concurrently to DeepSDF, OccupancyNetworks [27] proposed a slightly different formulation, replacing the SDF with the *Occupancy function*. In this case, the MLP is trained to approximate a function whose output can be interpreted as the probability of the input point to be inside the surface.

Given a 3D surface represented with SDF or Occupancy, an explicit representation of it – *i.e.*, a triangle mesh – can be obtained using Marching Cubes [35] and this can be done while preserving differentiability [36, 37, 38].

The major downside of SDF and Occupancy is that they can be used only to represent closed surfaces, as it is necessary to define which portion of the space is inside the shape. If one needs to represent open surfaces, it is possible to use SDF or Occupancy over an *inflated* version of them, which, however, entails a loss in accuracy. For this reason, there has been a recent push to replace SDFs by *unsigned distance functions* (UDFs) [26, 39, 40], which enable accurate modelling of surfaces with any topology.

One difficulty in adopting UDFs is that Marching Cubes was specifically designed to exploit the signs of SDFs. Obtaining explicit surfaces from UDFs is therefore non-trivial. A first method was proposed in [26], where the UDF gradients are used to project points onto the surface, obtaining an oriented point cloud which can be later meshified with [41]. A recent work [42], instead, proposed a modified version of the Marching Cubes algorithm that uses the UDF gradients to compute pseudo-signs, enabling the recovery of a complete mesh from a given UDF.

Over the time many improvements have been proposed on top of the INR original formulation, with many works focusing on the accuracy of the learned representation [43, 22, 30, 29, 32], and others studying the possibility of reducing the amount of data needed to train the MLPs [33, 44, 45] and the time needed for the training [34, 32].

We refer to Appendix C.1 for a detailed explanation of how INRs are obtained starting from discrete representations of 3D shapes.

1.3 Neural Radiance Fields

Neural Radiance Fields (NeRF) [24] represents nowadays the most popular paradigm for novel view synthesis, rapidly conquering the main stage over explicit approaches exploiting CNNs [46, 47, 48, 49, 50, 51, 52, 53, 54]. Peculiar to NeRF is a continuous volumetric representation encoded by a multilayer perceptron (MLP) – opposed to discrete representations such as voxel grids or multi-plane images – which enables to retrieve color and density of queried 3D points and to render images through differentiable ray casting.

More precisely, NeRF encodes a 3D scene into a function $\boldsymbol{F}_0$ mapping any space position $\boldsymbol{x}$ and viewing direction $\boldsymbol{d}$ pair into density σ and view-dependent color emission $\boldsymbol{c}$:

$$\boldsymbol{F}_0 : (\boldsymbol{x}, \boldsymbol{d}) \rightarrow (\boldsymbol{c}, \sigma). \tag{1.2}$$

Such implicit mapping is learned through a multilayer perceptron (MLP). Specifically, an intermediate $\mathrm{MLP}^{(\mathrm{pos})}$ infers density σ alongside an intermediate embedding $\boldsymbol{e}$, used by a shallower $\mathrm{MLP}^{(\mathrm{rgb})}$ together with viewing direction $\boldsymbol{d}$ to predict color:

$$\begin{aligned} (\sigma, \boldsymbol{e}) &= \mathrm{MLP}^{(\mathrm{pos})}(\boldsymbol{x})\ , \\ \boldsymbol{c} &= \mathrm{MLP}^{(\mathrm{rgb})}(\boldsymbol{e}, \boldsymbol{d}). \end{aligned} \tag{1.3}$$

Before feeding $\boldsymbol{x}$ to MLP$^{(\text{pos})}$, the 3D coordinates are projected into a higher-dimensional space through a positional encoding $\gamma(\boldsymbol{x})$ based on Fourier features [30] which enables to learn to represent more accurately the high-frequencies of the underlying function:

$$\begin{aligned}\gamma(\boldsymbol{x}) = (\,&\sin(2^0\pi\boldsymbol{x}), \cos(2^0\pi\boldsymbol{x}), ..., \\ &\sin(2^{L-1}\pi\boldsymbol{x}), \cos(2^{L-1}\pi\boldsymbol{x})).\end{aligned} \tag{1.4}$$

To render an image, *i.e.*, to get the color of any pixel $\boldsymbol{p}$, a ray $\boldsymbol{r}$ from the camera center through the pixel $\boldsymbol{p}$ is cast through the 3D space. Then, the pixel color $\hat{C}(\boldsymbol{r})$ is obtained through volumetric rendering according to the optical model by Max [55]:

$$\begin{aligned}\hat{C}(\boldsymbol{r}) &= \int_{t_n}^{t_f} T(\boldsymbol{t})\sigma(r(\boldsymbol{t}))c(r(\boldsymbol{t}), \boldsymbol{t})dt \\ T(\boldsymbol{t}) &= \exp\Big(-\int_{t_n}^{t} \sigma(r(\boldsymbol{s}))ds\Big)\end{aligned} \tag{1.5}$$

with $T(\boldsymbol{t})$ being the accumulated transmittance along the ray $\boldsymbol{t}$ from near plane t_n to any specific point $\boldsymbol{t}$. The value of such integral is estimated through quadrature, by sampling N evenly distant 3D points along the ray:

$$\begin{aligned}\hat{C}(\boldsymbol{r}) &= \left(\sum_{i=1}^{K} T_i\alpha_i\boldsymbol{c}_i\right) + T_{K+1}\boldsymbol{c}_{\text{bg}}\,, \\ \alpha_i &= \text{alpha}(\sigma_i, \delta_i) = 1 - \exp(-\sigma_i\delta_i)\,, \\ T_i &= \prod_{j=1}^{i-1}(1-\alpha_j)\,,\end{aligned} \tag{1.6}$$

with α_i being the probability of termination at the point i, δ_i the distance to the adjacent sampled point, and $\boldsymbol{c}_{bg}$ a pre-defined background color.

Given a set of training images with known camera poses, a NeRF model is trained by minimizing the photometric MSE between the pixel color $C(\boldsymbol{r})$ in the training image and the rendered color $\hat{C}(\boldsymbol{r})$:

$$\mathcal{L}_{\text{photo}} = \frac{1}{|\mathcal{R}|}\sum_{r\in\mathcal{R}} \left\|\hat{C}(\boldsymbol{r}) - C(\boldsymbol{r})\right\|_2^2\,, \tag{1.7}$$

with $\mathcal{R}$ with the set of rays in a single batch.

Vanilla NeRF has been rapidly extended to deal with different setups, *e.g.*, relighting [56, 57, 58], deformable objects [59, 60, 61, 62, 63], dynamic scenes [64, 65, 23, 66, 67], multi-resolution images [68] or to implement generative models [69, 70, 71].

Despite the elegant formulation and impressive quality of the synthesised views, the original NeRF suffers of some notable limitations, such as, in particular, the long training process – a few days in its very first implementation [24] – together with the requirement to perform a standalone training from scratch for any new scene and the slow rendering speed – definitely far from real-time.

Faster training. Speeding-up the training procedure represents the main barrier to break in order to deploy NeRF in real applications, as it would soften the limitation of requiring a scene-specific training. The main approaches proposed in literature rely on a pre-training phase [72, 73, 74, 75], deploy additional depth information estimated by means of Multi-View Stereo (MVS) methods [76, 77], use neural rays [76], exploit explicit representations [78] or combine them with implicit ones [79, 32].

Faster rendering. Achieving real-time rendering is highly desirable to improve end-user experience, possibly allowing for interactive visualization of novel viewpoints of a given object. Recent works exploit octree structures [80] to avoid redundant MLP queries in empty space, split a single MLP in thousands of tiny ones [81] or leverage explicit volumetric representations [82, 83, 84, 85].

Next-generation NeRFs. At the time of writing, a few very recent works stand out in terms of both training and inference speed. DirectVoxGo (DVGO) [79] combines implicit and explicit representations, using voxel grids together with a light MLP. Plenoxels [78] gets rid of the MLP and directly optimizes colors over a voxel grid. Instant Neural Graphic Primitives (Instant-NGP) [32] makes use of hash tables and optimized MLP implementations. Any of these frameworks can be easily trained in less than 10 minutes and can achieve good rendering speed without noticeable deterioration of rendering quality.

1.4 Content of the Thesis

The rest of this document is organized in two parts, briefly presented in the following.

1.4.1 Deploying NR of Signals

In Part I we report two works where we deploy NR as the main form of data representation for complex pipelines, motivated by the aforementioned advantages of NR and the continuous improvements in the field.

More specifically, in Chapter 2 we present DrapeNet, a framework for generating and draping garments over human bodies with different shapes and poses, with garments modeled by approximating their Unsigned Distance Function (UDF). This choice leads to a powerful representation for the open surfaces of the garments, that feature different geometries and topologies, and enables sampling new garments, which can be also edited according to specific characteristics. Finally, since adopting UDF for garments makes the whole pipeline fully differentiable, DrapeNet can be used to recover the 3D model of clothed people from partial observations such as images and 3D scans. DrapeNet is presented in a conference paper accepted at the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) 2023.

In Chapter 3, then, we describe ScanNeRF, a pipeline that we designed and implemented to collect a huge amount of images depicting a given object from different points of view in a matter of minutes. We show how these images can be used to build an accurate Neural Twin® of the input object, by taking advantage of neural radiance fields. We further exploit the scan station built in our work to collect a big and varied dataset of images that can be used to benchmark any neural rendering method through our public website. ScanNeRF is presented in a conference paper accepted at the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV) 2023.

1.4.2 Processing NR of Signals

In Part II, we consider an intriguing research question that arises from the above scenario: beyond storage and communication, would it be possible to process directly NR with deep learning pipelines to solve downstream tasks as it is routinely done today with discrete representations? Since NR are neural networks, there is no straightforward way to process them, as a single NR network can easily count hundreds of thousands of parameters. Thus, we settle on investigating whether and how it would be possible to cast the above research question into a representation learning problem, where individual NR are squeezed into compact and meaningful embeddings amenable to pursuing a variety of downstream tasks.

More specifically, in Chapter 4 we show that the weights of deep neural networks form a redundant parametrization of their underlying function. Indeed, we introduce NetSpace, a framework capable of compressing the weights of a given neural network into a low-dimensional embedding and of predicting the weights of a new network that behaves like the input one starting only from such embedding. NetSpace is presented in a conference paper accepted at the International Conference on Pattern Recognition (ICPR) 2022.

In Chapter 5 we move a step further, showing that it is possible to perform deep learning tasks on implicit neural representations (INR) of 3D shapes. We first introduce inr2vec, a framework inspired by NetSpace that allows compressing an input INR into a compact latent code by only looking at its weights. Then, we show that it is possible to use such low-dimensional vector as input and/or output for standard deep learning machinery to perform a great variety of task on the INR underlying 3D shape. inr2vec is presented in a conference paper accepted at the International Conference on Learning Representations (ICLR) 2023.

Part I

Deploying NR of Signals

Chapter 2

Modeling Garments with Unsigned Distance Functions

2.1 Introduction

Draping digital garments over differently-shaped bodies in random poses has been extensively studied due to its many applications such as fashion design, moviemaking, video gaming, virtual try-on and, nowadays, virtual and augmented reality. Physics-based simulation (PBS) [86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96] can produce outstanding results, but at a high computational cost that precludes real-time performance. Algorithms can be sped up by exploiting temporal consistency in pose sequences, but not enough.

Hence, recent years have witnessed the emergence of deep neural networks aiming to achieve the quality of PBS draping while being much faster [97, 98, 99, 100, 101, 102, 103, 104, 105]. These networks are often trained to produce garments that resemble ground-truth ones. While effective, this requires building training datasets, consisting of ground-truth meshes obtained either from computationally expensive simulations [106] or using complex 3D scanning setups [107]. Moreover, to generalize to unseen garments and poses, these supervised approaches require training databases encompassing a great variety of samples depicting many combinations of garments, bodies and poses.

The recent PBNS and SNUG approaches [108, 109] address this by casting the physical models adopted in PBS into constraints used for self-supervision of deep learning models. This makes it possible to train the network on a multitude of body shapes and poses without ground-truth draped garments. Instead, the predicted garments are constrained to obey physics-based rules. However, both PBNS and SNUG, require training a separate network for each garment. They rely on mesh templates for garment representation and feature one output per mesh vertex. Thus, they cannot handle meshes with different topologies, even for the same garment. This makes them very specialized and limits their applicability to large garment collections as a new network must be trained for each new clothing item.

In this chapter, we introduce DrapeNet, an approach that also relies on physics-based constraints to provide self-supervision but can handle generic garments by conditioning a *single* draping network with a latent code describing the garment to be draped. We achieve this by coupling the draping network with a garment *generative* network, composed of an encoder and a decoder. The encoder is trained to compress input garments into compact latent codes that are used as input condition for the draping network. The decoder, instead, reconstructs a 3D garment model from its latent code, thus allowing us to sample and edit new garments from the learned latent space.

Specifically, we model the output of the garment decoder as an unsigned distance function (UDF), which were demonstrated [42] to yield better accuracy and fewer interpenetrations than the inflated signed distance functions often used for this purpose [110, 111]. Moreover, UDFs can be triangulated in a differentiable way [42] to produce explicit surfaces that can easily be post-processed, making our pipeline fully differentiable. Hence, DrapeNet can not only drape garments over given body shapes but can also perform gradient-based optimization to fit garments, along with body shapes and poses, to partial observations of clothed people, such as images or 3D scans.

Our contributions are as follows:

- We introduce a *single* garment draping network conditioned on a latent code to handle generic garments;
- By exploiting physics-based self-supervision, our pipeline only requires a few hundred garment meshes in a canonical pose for training;
- Our framework enables fast draping of new garments with high fidelity, as well as sampling and editing of new garments from the learned latent space;
- Being fully differentiable, our method can be used to recover accurate 3D models of clothed people from images and 3D scans.

2.2 Related Work

Two main classes of draping methods coexist, physics-based algorithms [86, 112, 113, 106, 114] that produce high-quality drapings but at a high computational cost, and data-driven approaches that are faster but often at the cost of realism.

Among the latter, template-based approaches [109, 115, 116, 100, 117, 108, 103, 118] are dominant. Each garment is modeled by a specific triangulated mesh and a draping function is learned for each one. In other words, they do not generalize. There are however a number of exceptions. In [98, 119] the mesh is replaced by 3D point clouds that can represent generic garments. This enables deforming garments with arbitrary topology and geometric complexity, by estimating the deformation separately for each point. [120] goes further and allows differentiable changes in garment topology by sampling a fixed number of points

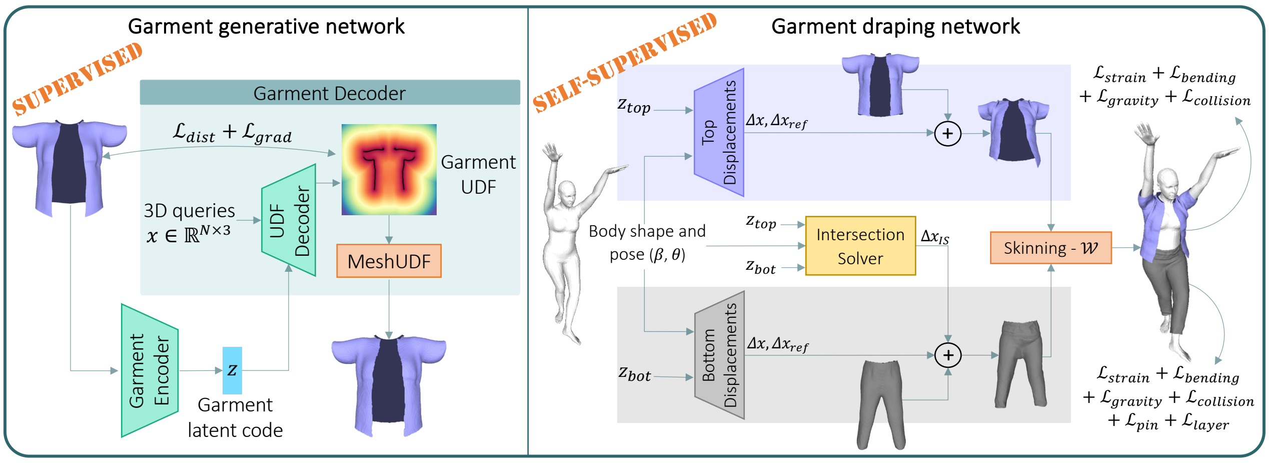

Garment generative network
SUPERVISED
Garment Decoder
$\mathcal{L}_{dist} + \mathcal{L}_{grad}$
Garment UDF
3D queries $x \in \mathbb{R}^{N\times3}$
UDF Decoder
MeshUDF
Garment Encoder
z
Garment latent code
Garment draping network
SELF-SUPERVISED
z_{top}
Top Displacements
$\Delta x, \Delta x_{ref}$
$\mathcal{L}_{strain} + \mathcal{L}_{bending} + \mathcal{L}_{gravity} + \mathcal{L}_{collision}$
Body shape and pose (β, θ)
z_{top}
z_{bot}
Intersection Solver
Δx_{IS}
Skinning - $\mathcal{W}$
z_{bot}
Bottom Displacements
$\Delta x, \Delta x_{ref}$
$\mathcal{L}_{strain} + \mathcal{L}_{bending} + \mathcal{L}_{gravity} + \mathcal{L}_{collision} + \mathcal{L}_{pin} + \mathcal{L}_{layer}$

FIGURE 2.1. **Overview of our framework. Left:** Garment generative network, trained to embed garments into compact latent codes and predict their unsigned distance field (UDF) from such vectors. UDFs are then meshed using [42]. **Right:** Garment draping network, conditioned on the latent codes of the generative network. It is trained in a self-supervised way to predict the displacements Δx and Δx_{ref} to be applied to the vertices of given garments, before skinning them according to body shape and pose (β, θ) with the predicted blending weights $\mathcal{W}$. It includes an Intersection Solver module to prevent intersection between top and bottom garments.

from the body mesh. Unfortunately, this point cloud representation severely limits possible downstream applications, which typically require a complete surface.

In recent approaches [110, 111], a space of garments is learned with clothing items modeled as inflated SDFs and one single shared network to predict their deformations as a 3D displacement field. This makes deployment in real-world scenarios easier and allows the reconstruction of garments from images and 3D scans. However, the inflated SDF scheme reduces realism and precludes post-processing using standard physics-based simulators or other cloth-specific downstream applications. Furthermore, both models are fully supervised and require a dataset of draped garments whose collection is extremely time-consuming.

Alleviating the need for costly ground-truth draped garments is tackled in [108, 109], by introducing physics-based losses to train draping networks in a self-supervised manner. The approach of [108] relies on a mass spring model to enforce the physical consistency of static garments deformed by different body poses. The method of [109] also accounts for variable body shapes and dynamic effects; furthermore, it incorporates a more realistic and expressive material model. Both methods, however, require training one network per garment, a limitation we remove.

2.3 Method

We aim to realistically deform and drape generic garments over human bodies of various shapes and poses. To this end, we introduce the DrapeNet framework, presented in Fig. 2.1.

It comprises a generative network shown on the left and a draping network shown on the right. Only the first is trained in a supervised manner, but using only static unposed garments meshes. This is key to avoiding having to run physics-based simulations to generate ground-truth data. Furthermore, we condition the draping network on latent vectors representing the input garments, which allows us to use the same network for very different garments, something that competing methods [109, 108] cannot do.

The generative network is a decoder trained using an encoder that turns a garment into a latent code $\mathbf{z}$ that can then be decoded to an Unsigned Distance Function (UDF), from which a triangulated mesh can be extracted in a differentiable manner [42]. The UDF representation allows us to accurately represent open surfaces and the many openings that garments typically feature. Since the top and bottom garments – shirts and trousers – have different patterns, we train one generative model for each. Both networks have the same architecture but different weights.

The resulting *garment generative network* is only trained to output garments in a canonical shape, pose, and size that fit a neutral SMPL [121] body. Draping the resulting garments to bodies in non-canonical poses is then entrusted to a *draping network*, again one for the top and one for the bottom. As in [109, 108, 111], this network predicts vertex displacements *w.r.t.* the neutral position. The deformed garment is then skinned onto the articulated body model. To enable generalization to different tops and bottoms, we condition the draping process on the garment latent codes of the generative network, shown as $\mathbf{z}_{top}$ and $\mathbf{z}_{bot}$ in Fig. 2.1.

We use a small database of static unposed garments loosely aligned with bodies in the canonical position to train the two garment generating networks. This being done, we exploit physics-based constraints to train in a fully self-supervised manner the top and bottom draping networks for realism, without interpenetrations with the body and between the garments themselves.

2.3.1 Garment Generative Network

To encode garments into latent codes that can then be decoded into UDFs, we rely on a point cloud encoder that embeds points sampled from the unposed garment surface into a compact vector. This lets us obtain latent codes for previously unseen garments in a single inference pass from points sampled from its surface. This can be done given any arbitrary surface triangulation. Hence, it gives us the flexibility to operate on any given garment mesh.

We use DGCNN [122] as the encoder. It first propagates the features of points within the same local region at multiple scales and then aggregates them into a single global embedding by max pooling. We pair it with a decoder that takes as input a latent vector, along with a point in 3D space, and returns its (unsigned) distance to the garment. The decoder is a multi-layer perceptron (MLP) that relies on Conditional Batch Normalization [123] for conditioning on the input latent vector.

We train the encoder and the decoder by encouraging them to jointly predict distances that are small near the training garments' surface and large elsewhere. Because the algorithm we use to compute triangulated meshes from the predicted distances [42] relies on the gradient vectors of the UDF field, we also want these gradients to be as accurate as possible [45, 39]. We therefore minimize the loss

$$L_{garm} = L_{dist} + \lambda_g L_{grad} \; , \tag{2.1}$$

where L_{dist} encodes our distance requirements, L_{grad} the gradient ones, and λ_g is a weight balancing their influence.

More formally, at training time and given a mini-batch comprising B garments, we sample a fixed number P of points from the surface of each one. For each resulting point cloud $\mathbf{p}_i$ $(1 \leq i \leq B)$, we use the garment encoder E_G to compute the latent code

$$\mathbf{z}_i = E_G(\mathbf{p}_i) \tag{2.2}$$

and use it as input to the decoder D_G. It predicts an UDF field supervised with Eq. (2.1), whose terms we define below.

Distance Loss. Having experimented with many different formulations of this loss, we found the following one both simple and effective. Given N points $\{\mathbf{x}_{ij}\}_{j \leq N}$ sampled from the space surrounding the i-th garment, we pick a distance threshold δ, clip all the ground-truth distance values $\{y_{ij}\}$ to it, and linearly normalize the clipped values to the range $[0, 1]$. This yields normalized ground-truth values $\bar{y}_{ij} = \min(y_{ij}, \delta)/\delta$. Similarly, we pass the output of the final layer of D_G through a sigmoid function $\sigma(\cdot)$ to produce a prediction in the same range for point $\mathbf{x}_{ij}$

$$\widetilde{y}_{ij} = \sigma(D_G(\mathbf{x}_{ij}, \mathbf{z}_i)) \; . \tag{2.3}$$

Finally, we take the loss to be

$$\mathcal{L}_{dist} = \text{BCE}\left[(\bar{y}_{ij})_{j \leq N}^{i \leq B} \, , \; (\widetilde{y}_{ij})_{j \leq N}^{i \leq B} \right] , \tag{2.4}$$

where $\text{BCE}[\cdot, \cdot]$ stands for binary cross-entropy with continuous labels. As observed in [124], the sampling strategy used for points $\mathbf{x}_{ij}$ strongly impacts training effectiveness. We describe ours in Appendix A.1.3. In our experiments, we set $\delta = 0.1$, being the top and bottom garments normalized respectively into the upper and lower halves of the $[-1, 1]^3$ cube.

Gradient Loss. Given the same sample points as before, we take the gradient loss to be

$$\mathcal{L}_{grad} = \frac{1}{BN} \sum_{i,j} \left\| \mathbf{g}_{ij} - \widehat{\mathbf{g}_{ij}} \right\|_2^2 \ , \tag{2.5}$$

where $\mathbf{g}_{ij} = \nabla_{\mathbf{x}} y_{ij} \in \mathbb{R}^3$ is the ground-truth gradient of the i-th garment's UDF at $\mathbf{x}_{ij}$ and $\widehat{\mathbf{g}_{ij}} = \nabla_{\mathbf{x}} D_G(\mathbf{x}_{ij}, \mathbf{z}_i)$ the one of the predicted UDF, computed by backpropagation.

2.3.2 Garment Draping Network

We describe our approach to draping generic garments as opposed to specific ones and our self-supervised scheme.

Draping Generic Garments

We rely on SMPL [121] to parameterize the body in terms of shape (β) and pose (θ) parameters. It uses Linear Blend Skinning to deform a body template. Since garments generally follow the pose of the underlying body, we extend the SMPL skinning procedure to the 3D volume around the body for garment draping. Given a point $\mathbf{x} \in \mathbb{R}^3$ in the garment space, its position $D(\mathbf{x}, \beta, \theta, \mathbf{z})$ after draping becomes

$$\begin{aligned} D(\mathbf{x}, \beta, \theta, \mathbf{z}) &= W(\mathbf{x}_{(\beta,\theta,\mathbf{z})}, \beta, \theta, \mathcal{W}(\mathbf{x})) \ , \\ \mathbf{x}_{(\beta,\theta,\mathbf{z})} &= \mathbf{x} + \Delta x(\mathbf{x}, \beta) + \Delta x_{\text{ref}}(\mathbf{x}, \beta, \theta, \mathbf{z}) \ , \\ \Delta x_{\text{ref}}(\mathbf{x}, \beta, \theta, \mathbf{z}) &= \mathcal{B}(\beta, \theta) \cdot \mathcal{M}(x, \mathbf{z}) \ , \end{aligned} \tag{2.6}$$

where $W(\cdot)$ is the SMPL skinning function, applied with blending weights $\mathcal{W}(\mathbf{x})$, over the point displaced by $\Delta x(\mathbf{x}, \beta)$ and $\Delta x_{\text{ref}}(\mathbf{x}, \beta, \theta, \mathbf{z})$. $\mathcal{W}(\mathbf{x})$ and $\Delta x(\mathbf{x}, \beta)$ are computed as in [117, 111]. However, they only give an initial deformation for garments that roughly fits the underlying body. To refine it, we introduce a new term, $\Delta x_{\text{ref}}(\mathbf{x}, \beta, \theta, \mathbf{z})$. It is a deformation field conditioned on body parameters β and θ, and on the garment latent code $\mathbf{z}$ from the generative network. Following the linear decomposition of displacements in SMPL, it is the composition of an embedding $\mathcal{B}(\beta, \theta) \in \mathbb{R}^{N_\mathcal{B}}$ of body parameters and a displacement matrix $\mathcal{M}(x, \mathbf{z}) \in \mathbb{R}^{N_\mathcal{B} \times 3}$ conditioned on $\mathbf{z}$. Being conditioned on the latent code $\mathbf{z}$, Δx_{ref} can deform different garments differently, unlike the methods of [109, 108].

Since we have distinct encodings for the top and bottom garments, for each one we train two MLPs ($\mathcal{B}$, $\mathcal{M}$) to predict Δx_{ref}. The other MLPs for $\mathcal{W}(\cdot)$ and $\Delta x(\cdot)$ are shared.

Self-Supervised Training

We first learn the weights of $\mathcal{W}(\cdot)$ and $\Delta x(\cdot)$ as in [117, 111], which does not require any annotation or simulation data but only the blending weights and shape displacements of SMPL. We then train our deformation fields Δx_{ref} in a fully self-supervised fashion by minimizing the physics-based losses introduced below. In this way, we completely eliminate the huge cost that extensive simulations would entail.

Top Garments. For upper body garments – shirts, t-shirts, vests, tank tops, etc. – the deformation field is trained using the loss from [108], expressed as

$$\mathcal{L}_{top} = \mathcal{L}_{strain} + \mathcal{L}_{bend} + \mathcal{L}_{gravity} + \mathcal{L}_{col} \ , \tag{2.7}$$

where $\mathcal{L}_{strain}$ is the membrane strain energy of the deformed garment, $\mathcal{L}_{bend}$ the bending energy caused by the folding of adjacent faces, $\mathcal{L}_{gravity}$ the gravitational potential energy, and $\mathcal{L}_{col}$ a penalty for collisions between body and garment. Unlike in [108], we only consider the quasi-static state after draping, that is, without acceleration.

Bottom Garments. Due to gravity, bottom garments, such as trousers, would drop onto the floors if we used only the loss terms of Eq. (2.7). We thus introduce an extra loss term to constrain the deformation of vertices around the waist and hips. The loss becomes

$$\begin{aligned} \mathcal{L}_{bottom} &= \mathcal{L}_{strain} + \mathcal{L}_{bend} + \mathcal{L}_{gravity} + \mathcal{L}_{col} + \mathcal{L}_{pin}, \\ \mathcal{L}_{pin} &= \sum_{v \in V} |\Delta x_y|^2 + \lambda(|\Delta x_x|^2 + |\Delta x_z|^2) \ , \end{aligned} \tag{2.8}$$

where V is the set of garment vertices whose closest body vertices are located in the region of the waist and hips. See Appendix A.2.1 for details. The terms Δx_x, Δx_y and Δx_z are the deformations along the X, Y and Z axes, respectively. λ is a positive value smaller than 1 that penalizes deformations along the vertical direction (Y axis) and produces natural deformations along the other directions.

Top-Bottom Intersection. To ensure that the top and bottom garments do not intersect with each other when we drape them on the same body, we define a loss $\mathcal{L}_{IS}$ that ensures that when the top and the bottom garments overlap, the bottom garment vertices are closer to the body mesh than the top ones, which prevents them from intersecting – this is arbitrary, and the following could be formulated the other way around. To this end, we introduce an Intersection Solver (IS) network. It predicts a displacement correction Δx_{IS}, added only when draping bottom garments as

$$\tilde{\mathbf{x}}_{(\mathbf{z}_{top},\mathbf{z}_{bot})} = \mathbf{x}_{(\mathbf{z}_{bot})} + \Delta x_{IS}(\mathbf{x}, \mathbf{z}_{top}, \mathbf{z}_{bot}) \ , \tag{2.9}$$

where we omit the dependency of $\tilde{\mathbf{x}}$, $\mathbf{x}$ and Δx_{IS} on the body parameters (β, θ) for simplicity.

$\mathbf{z}_{top}$ and $\mathbf{z}_{bot}$ are the latent codes of the top and bottom garments, and $\mathbf{x}_{(\mathbf{z}_{bot})}$ is the input point displaced according to Eq. (2.6). The skinning function of Eq. (2.6) is then applied to $\tilde{\mathbf{x}}_{(\mathbf{z}_{top},\mathbf{z}_{bot})}$ for draping. $\Delta x_{IS}(\cdot)$ is implemented as a simple MLP and trained with

$$\mathcal{L}_{IS} = \mathcal{L}_{bottom} + \mathcal{L}_{layer}, \tag{2.10}$$

where $\mathcal{L}_{layer}$ is a loss whose minimization requires the top and bottom garments to be separated from each other. We formulate it as

$$\mathcal{L}_{layer} = \sum_{v_B \in C} max(d_{bot}(v_B) - \gamma d_{top}(v_B), 0) \ , \tag{2.11}$$

where C is the set of body vertices covered by both the top and bottom garments, $d_{top}(\cdot)$ and $d_{bot}(\cdot)$ the distance to the top and the bottom garments respectively, and γ a positive value smaller than 1 (more details in Appendix A.2.2).

2.4 Experiments

In the following, we describe our experimental setup and we test DrapeNet for the different purposes depicted by Fig. 2.2. We first test our generative network, showing that it allows for reconstructing different kinds of garments and for editing them by manipulating their latent codes. We then gauge the draping network both qualitatively and quantitatively. Finally, we use DrapeNet to reconstruct garments from images and 3D scans.

2.4.1 Datasets, Settings and Metrics

Datasets. Both our generative and draping networks are trained with garments from CLOTH3D [125], a synthetic dataset that contains over 7K sequences of animated 3D humans parametrized used the SMPL model and wearing different garments. Each sequence comprises up to 300 frames and features garments coming from different templates. For training, we randomly selected 600 top garments (t-shirts, shirts, tank tops, etc.) and 300 bottom garments (both long and short trousers). Neither for the generative nor for the draping networks did we use the simulated deformations of the selected garments. Instead, we trained the networks using only garment meshes on average body shapes in T-pose. By contrast, for testing purposes, we selected random clothing items – 30 for top garments and 30 bottom ones – and considered *whole* simulated sequences.

Training. We train two different models for top and bottom garments, both for the generative and for the draping parts of our framework. First, the generative models are trained on the 600/300 neutral garments. Then, with the generative networks weights frozen,

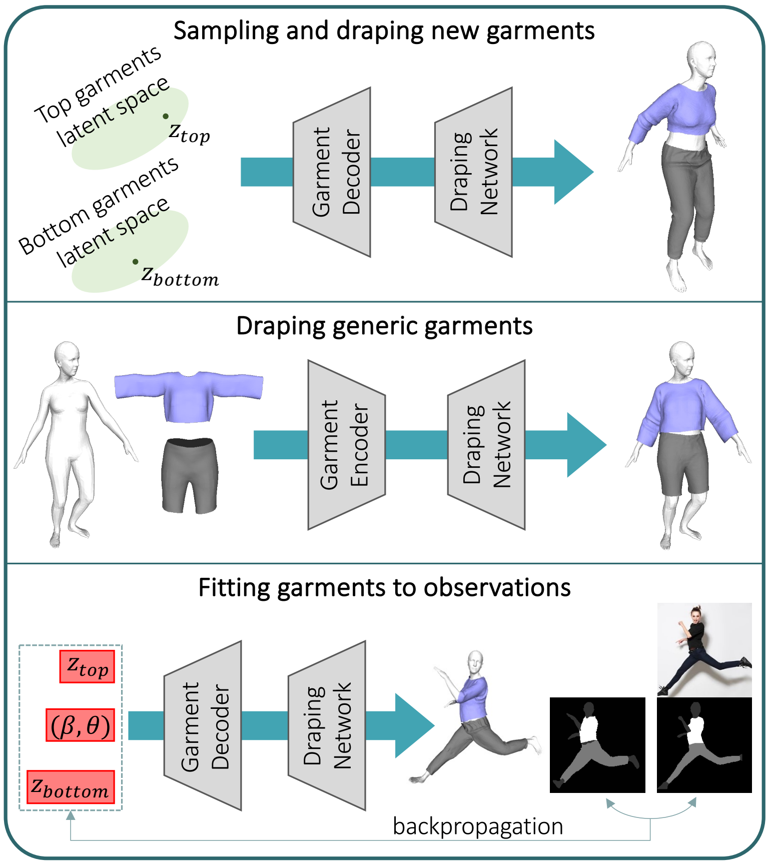


FIGURE 2.2. **Overview of DrapeNet applications. Top:** New garments can be sampled from the latent spaces of the generative networks, and deformed by the draping networks to fit a given body. **Center:** The garment encoders and the draping networks form a general purpose framework to drape any garment with a single forward pass. **Bottom:** Being a differentiable parametric model, our framework can reconstruct 3D garments by fitting observations such as images or 3D scans. The red boxes indicate the parameters optimized in this process.

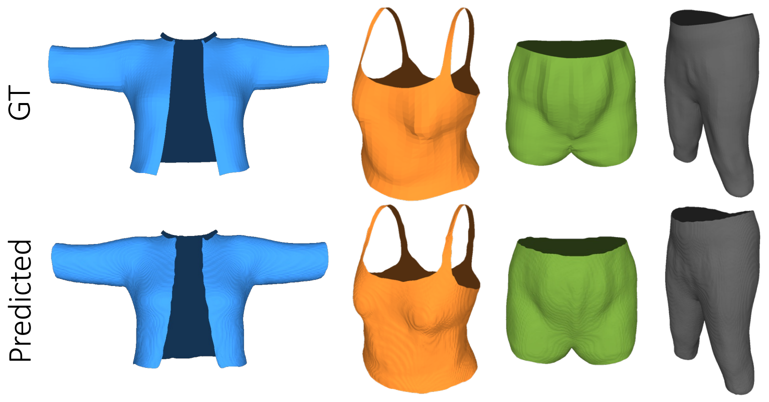


FIGURE 2.3. **Generative network: reconstruction of unseen garments in neutral pose/shape.** The latent codes are obtained with the garment encoder, then decoded into open surface meshes.

we train the draping networks by following [108]: body poses θ are sampled randomly from the AMASS [126] dataset, and shapes β uniformly from $[-3,3]^{10}$ at each step. The other hyperparameters are given in Appendix A.1.5.

Metrics. We report the Euclidean distance (ED), interpenetration ratio between body and garment (B2G), and intersection between top and bottom garments (G2G). ED is computed between corresponding vertices of the considered meshes. B2G is the area ratio between the garment faces inside the body and the whole surface as in [111]. Since CLOTH3D exclusively features pairs of top/bottom garments with the bottom one closer to the body, G2G is computed by detecting faces of the bottom garment that are outside of the top one, and taking the area ratio between those and the overall bottom garment surface.

2.4.2 Garment Parametrization

We first test the encoding-decoding scheme of Sec. 2.3.1.

Encoding-Decoding Previously Unseen Garments. The generative network of Fig. 2.1 is designed to project garments into a latent space and to reconstruct them from the resulting latent vectors. In Fig. 2.3, we visualize reconstructed previously-unseen garments from CLOTH3D. The reconstructions are faithful to the input garments, including fine-grained details such as the shirt collar on the left or the shoulder straps of the tank top. Fig. 2.4 and Fig. 2.5 show additional examples of the encoding-decoding capabilities of our garment generative network for top and bottom *test* garments, respectively. It is possible to notice how the output garments closely match the input ones, both in terms of geometry and topology.

Semantic Manipulation of Latent Codes. Our framework enables us to edit a garment by manipulating its latent code. For the resulting edits to have a semantic meaning, we assigned binary labels corresponding to features of interest to 100 training garments. For instance, we labeled garments as having "short sleeves" (label = 0) or "long sleeves" (label = 1). Then, we fit a linear logistic regressor to the garment latent codes. After training, the

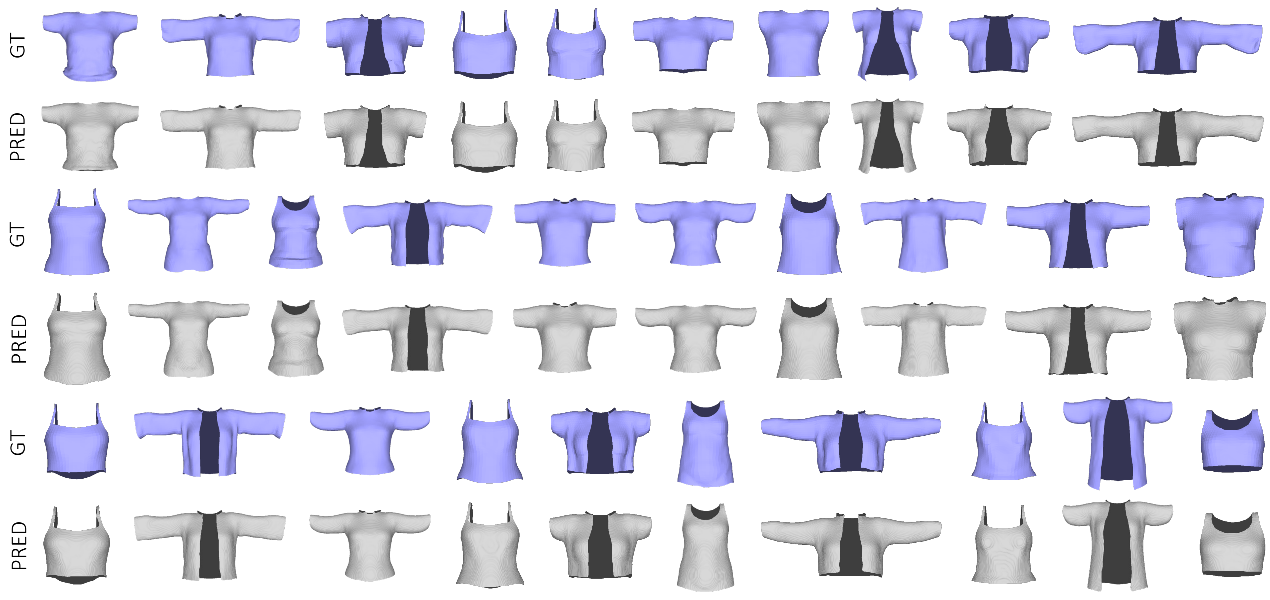


FIGURE 2.4. **Generative network: reconstruction of unseen garments in neutral pose/shape (top garments).** Latent codes for unseen garments can be obtained with our garment encoder. These codes are then used by the garment decoder to reconstruct open surface meshes. Input garments are colored in purple, while the reconstructed meshes are colored in gray.

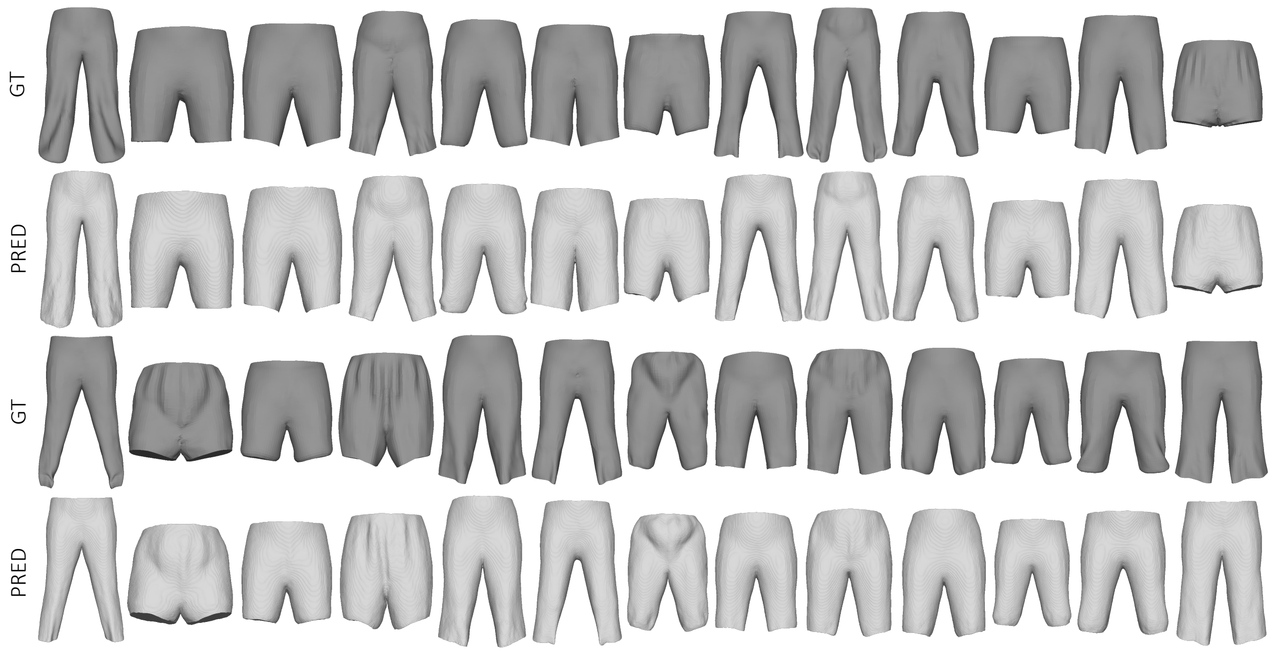


FIGURE 2.5. **Generative network: reconstruction of unseen garments in neutral pose/shape (bottom garments).** Latent codes for unseen garments can be obtained with our garment encoder. These codes are then used by the garment decoder to reconstruct open surface meshes. Input garments are colored in dark gray, while the reconstructed meshes are colored in light gray.

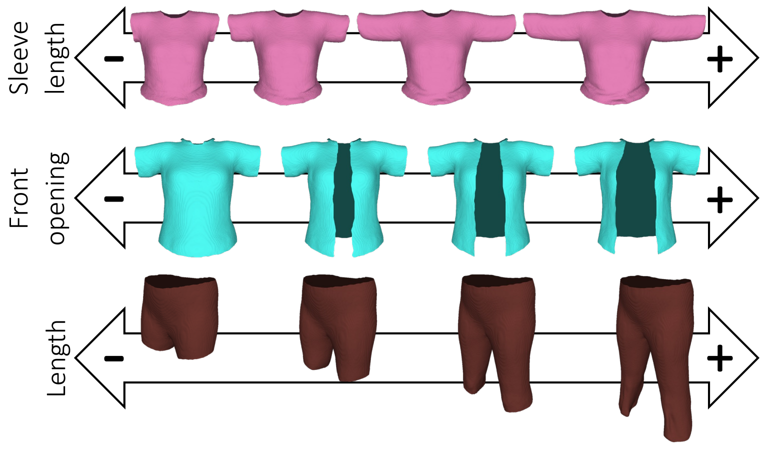


FIGURE 2.6. **Garment editing.** The latent codes produced by the garment encoder can be manipulated to edit specific features of the corresponding garments, without altering the overall geometry.

regressor weights indicate which dimensions of the latent space control the feature of interest. To this end, we first apply min-max normalization to the absolute weight values and then zero out the ones below a certain threshold, empirically set to 0.5. The remaining non-zero weights indicate which dimensions of the latent codes should be increased or decreased to edit the studied feature: we modify all of them, increasing (decreasing) their value if the associated weight is positive (negative), with a step proportional to the weight magnitude. To create Fig. 2.6, we applied this simple procedure to control the sleeve length and the front opening for top garments along with the length for bottom garments. As can be seen from the figure, our latent representations give us the ability to edit a specific garment feature while leaving other aspects of the garment geometry unchanged.

Ablation study for $\mathcal{L}_{garm}$. We report here an ablation study that we conducted to determine the best formulation for $\mathcal{L}_{garm}$, the loss function presented in Eq. (2.1), that we use to train our garment generative network. In particular, we consider three variants for $\mathcal{L}_{dist}$, the term of the supervision signal that guides the network to predict accurate values for the garments UDF. In addition to the binary cross-entropy loss (BCE) presented in Eq. (2.4), we study the possibility of using more traditional regression losses, such as L1 and L2 losses. Adopting the notation introduced in Sec. 2.3.1, the L1 loss is defined as $\frac{1}{BN}\sum_{i,j}|min(y_{ij},\delta) - \tilde{y}_{ij}|$, while the L2 loss is computed as $\frac{1}{BN}\sum_{i,j}(min(y_{ij},\delta) - \tilde{y}_{ij})^2$. On top of the three variants for $\mathcal{L}_{dist}$, we also consider for each one the possibility of removing the gradients supervision from $\mathcal{L}_{garm}$, *i.e.*, setting $\lambda_g = 0$. We trained our generative network for 48 hours with the resulting six loss function variants and then compared the quality of the garments reconstructed with the garment decoder. Fig. 2.7 presents a significant example of what we observed on the test set. Without gradients supervision (top row of the figure), none of the considered loss functions (BCE, L1 or L2) can guide the network to predict smooth surfaces without artifacts or holes. Adding the gradients supervision (bottom row) induces a strong regularization

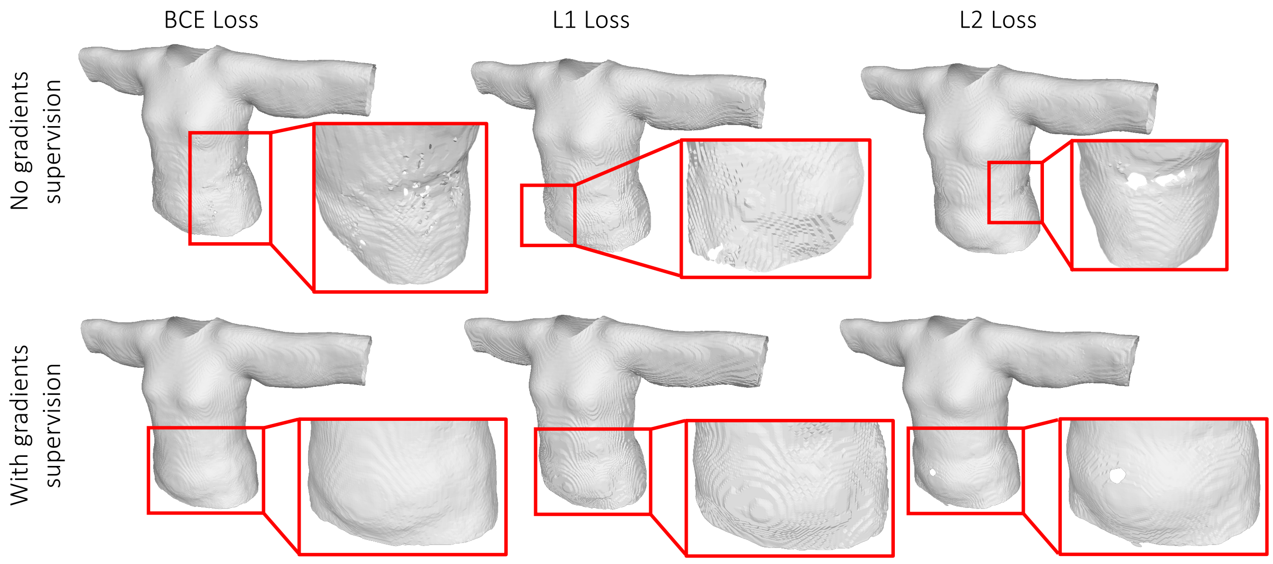


FIGURE 2.7. **Comparison between different loss functions for the garment generative network.** We present the same garment reconstructed by our generative network after being trained for 48 hours with six different alternatives of loss functions.

on the predicted distance fields, helping the network to predict surfaces without holes in most of the cases. However, using the L1 loss leads to rough surfaces, as one can observe in the center column of the bottom row of the figure. The BCE and the L2 losses (first and third columns of the bottom row), instead, produce smooth surfaces that are pleasant to see. We finally opted for the BCE loss over the L2 loss, since the network trained with the latter occasionally predicts surfaces with small holes, as in the example shown in the figure.

2.4.3 Garment Draping

We now turn to the evaluation of the draping network and compare its performance to those of DeePSD [119] or DIG [111], two *fully supervised* learning methods trained on CLOTH3D. DeePSD takes the point cloud of the garment mesh as input and predicts blending weights and pose displacements for each point; DIG drapes garments with a learned skinning field that can be applied to generic 3D points, but is similar for all garments. We chose those because, like DrapeNet, they both can deform garments of arbitrary geometry and topology.

Draping Unseen Meshes. We drape previously unseen garments on different bodies in random poses. We first encode the garments and use the resulting latent codes to condition the draping network, whose inference takes ∼5ms. We provide qualitative results in Fig. 2.8 and report quantitative ones in Tab. 2.1. Despite being completely self-supervised, DrapeNet delivers the lowest ratio of body-garment interpenetrations (B2G) for both top and bottom garments and the least intersections between them (G2G).

However, DrapeNet also yields higher ED values, which makes sense because there is more than one way to satisfy the physical constraints and to achieve realism. Hence, in the

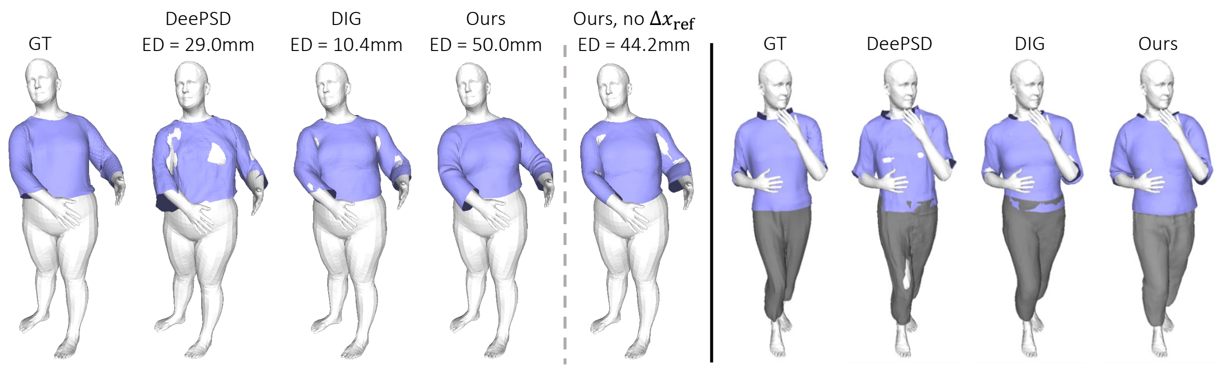


FIGURE 2.8. **Comparison between DeePSD, DIG and our method.** Ours is more realistic despite having the highest Euclidean distance (ED) error (**left**), and has less intersection between garments (**right**). **Left** also shows that Δx_{ref} is necessary for realistic deformations.

	DeePSD	DIG	Ours
ED-top (mm)	28.1	29.6	47.9
ED-bottom (mm)	18.3	20.0	27.3
B2G-top (%) ↓	7.2	1.8	**0.9**
B2G-bottom (%) ↓	3.4	0.8	**0.3**
G2G (%) ↓	2.0	4.0	**0.5**

TABLE 2.1. **Draping unseen garment meshes.** Comparison between DeePSD, DIG and our method, for top and bottom garments: Euclidean distance (ED), intersections with the body (B2G) and between garments (G2G) as ratio of intersection areas.

absence of explicit supervision, there is no reason for the answer picked by DrapeNet to be exactly the same as the one picked by the simulator. In fact, as argued in [109] and illustrated by Fig. 2.8, which is representative in terms of ED, a low ED value does not necessarily correspond to a realistic draping. To confirm this, we conducted a human evaluation study by sharing a link to a website on friends groupchats. We gave no further instructions or details besides those given on the site and reproduced in Appendix A.4. The website displays 3 drapings of the same garment over the same posed body, one computed using our method and the others using the other two. The users were asked to select which one of the three seemed more realistic and more pleasant, with a fourth potential response being “none of them". We obtained feedback from 187 different people. A total of 1258 individual examples were rated and we collected 3738 user opinions. In other words, each user expressed 20 opinions on average. The chart in Fig. 2.9 shows that our method was selected more than 50% of the times, with a large gap over the second best, DIG [111], selected less than 30% of the time. This result confirms that DrapeNet can drape garments with better perceptual quality than the competing methods.

In Fig. 2.10 we show additional qualitative results of garment draping produced by our method, where the garment meshes are generated by our UDF model. It can be seen that

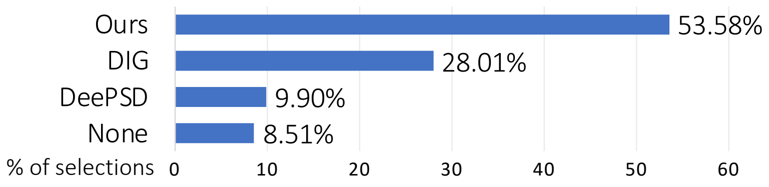


FIGURE 2.9. **Human evaluation of draping results.** When shown draping results of our method, DIG and DeePSD, evaluators selected ours as the most realistic one in more than half of the cases. *None* refers to the case when they had no clear preference.

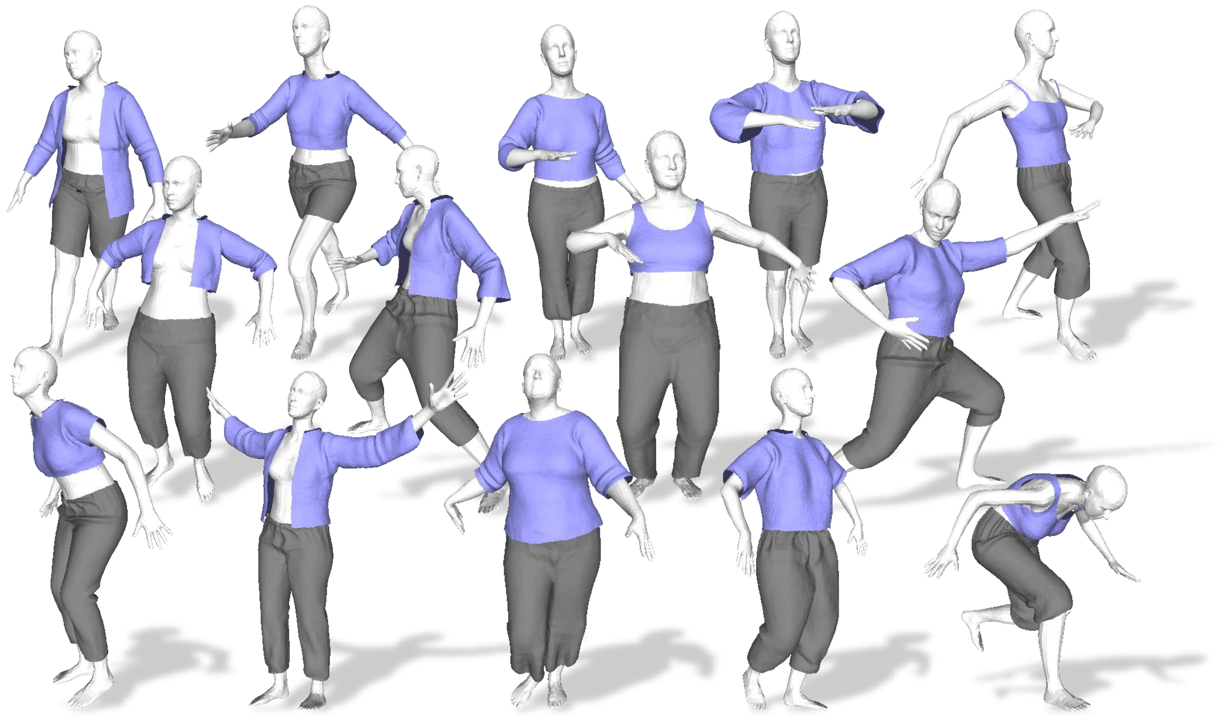


FIGURE 2.10. **Additional draping results.** Draping garments of different topologies over bodies in various shapes and poses with our method.

our method can realistically drape garments with different topologies over bodies of various shapes and poses.

Ablation Study. In Fig. 2.11, we show what happens when the draping network is conditioned with a latent code of a garment that does not match the input one. This creates unnatural deformations on the front when using the code of a shirt with a front opening to deform a shirt without an opening. Similarly, the sleeves penetrate the arms when conditioning with the code of a short sleeves shirt. This demonstrates that the draping network truly exploits the latent codes to predict garment-dependent deformation fields.

In Fig. 2.8 **left** we show that removing our novel displacement term $\Delta x_{\text{ref}}(\cdot)$ from Eq. (2.6) leads to unrealistic results.

We also ablate the influence of our Intersection Solver and observe that G2G increases from 0.5% to 1.1% without it. This demonstrates the effectiveness of this component at

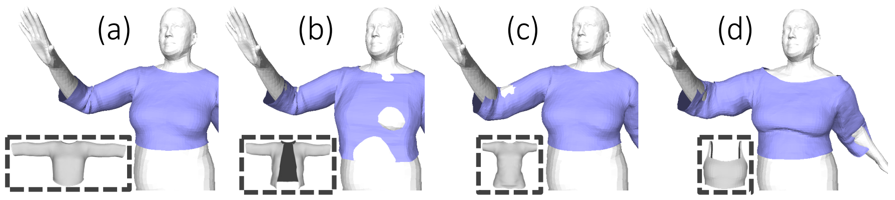


FIGURE 2.11. **Switching input latent codes of the draping network.** Draping the same shirt by conditioning the draping network with **(a)** the corresponding latent code, **(b)** the code of an open vest, **(c)** of a t-shirt and **(d)** of a tank top. Gray meshes in dashed boxed are the garments corresponding to the input latent codes.

reducing collisions between top and bottom garments.

2.4.4 Fitting Observations

Since our method is end-to-end differentiable, it can be used to reconstruct 3D models of people and their garments from partial observations, such as 2D images and 3D scans.

Fitting Images. Given an image of a clothed person, we use the algorithm of [128, 129] to get initial estimates for the body parameters (β, θ) and a segmentation mask $\mathbf{S}$. Then, starting with the mean of the learned codes $\mathbf{z}$, we reconstruct a mesh for the body and its garments by minimizing

$$\begin{aligned} L(\beta, \theta, \mathbf{z}) &= L_{\text{IoU}}(R(D(\mathbf{G}, \beta, \theta, \mathbf{z}), \text{SMPL}(\beta, \theta)),\ \mathbf{S})\ , \\ \mathbf{G} &= \text{MeshUDF}(D_G(\mathbf{z}))\ , \end{aligned} \tag{2.12}$$

w.r.t. $\mathbf{z}$, β and θ, where L_{IoU} is the IoU loss [130] in pixel space penalizing discrepancies between 2D masks, $R(\cdot)$ is a differentiable mesh renderer [131], and $\mathbf{G}$ is the set of vertices of the garment mesh reconstructed with our garment decoder using $\mathbf{z}$. $D(\cdot)$ and $\text{SMPL}(\cdot)$ are the garment and body skinning functions defined in Eq. (2.6) and in [121], respectively. To ensure pose plausibility, θ is constrained by an adversarial pose prior [132].

For simplicity's sake, Eq. (2.12) formulates the reconstruction of a single garment $\mathbf{G}$. In practice, we extend this formulation to both the top and the bottom garments shown in the target image. Fig. 2.12 depicts the results of minimizing this loss. It outperforms the state-of-the-art methods SMPLicit [110], ClothWild [127] and DIG [111]. The garments we recover follow those in the input image with higher fidelity and visual quality, without interpenetration between the body and the garments or between the two garments.

After this optimization, we can further refine the result by minimizing the physics-based objectives of Eq. (2.7) *w.r.t.* the per-vertex displacements of the reconstructed garments, as opposed to *w.r.t.* the latent vectors. We describe this procedure in Appendix A.2.3. As shown in the third column of Fig. 2.12, this further boosts the realism of the reconstructed garments. Note that this refinement is feasible thanks to the open surface representation allowed by our UDF model. Applying these physically inspired losses to an inflated garment, as produced

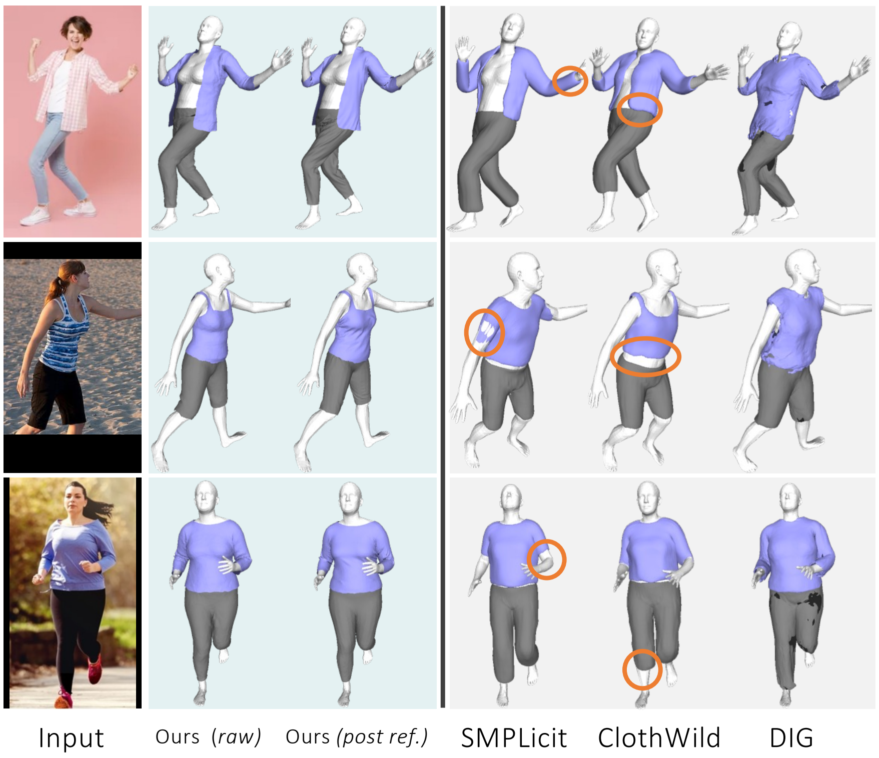


FIGURE 2.12. **Recovering garments and bodies from images.** From left to right we show the input image and the 3D models recovered with our method (without and with post-refinement), and competitors methods: SMPLicit [110], ClothWild [127], DIG [111].

by SMPLicit, ClothWild and DIG, yields poor results with many self-intersections, as shown in Appendix A.2.3.

Fitting 3D scans. Given a 3D scan of a clothed person and segmentation information, we apply a strategy similar to the one presented above and minimize

$$L(\beta,\theta,\mathbf{z}) = d(D(\mathbf{G},\beta,\theta,\mathbf{z}),\ \mathbf{S_G}) + \vec{d}(\text{SMPL}(\beta,\theta),\ \mathbf{S_B}), \tag{2.13}$$

w.r.t. $\mathbf{z}$, β and θ, where $\mathbf{S_G}$ and $\mathbf{S_B}$ denote the segmented garment and body scan points, and $d(a,b)$ and $\vec{d}(a,b)$ are the bidirectional and the one-directional Chamfer distance from b to a. Similarly to Eq. (2.12), we apply Eq. (2.13) to recover both the top and bottom garments. Fig. 2.13 shows our fitting results for some scans of the SIZER dataset [103]. The recovered 3D models closely match the input scans. Moreover, we can also apply a post-refinement procedure similar to the one described above, by minimizing both the physics-based losses from Eq. (2.7) and the Chamfer distance to the input scan *w.r.t.* the 3D coordinates of the vertices of the reconstructed models. This leads to even more realistic results, with fine wrinkles aligning to the input scans.

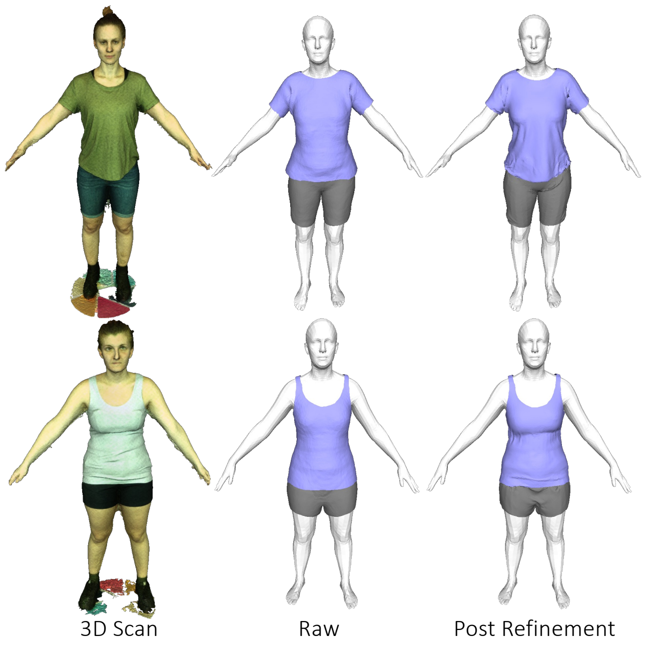


FIGURE 2.13. **Recovering garments and bodies from 3D scans.** We show 3D models recovered with our method from scans of the SIZER dataset [103]. *Raw* indicates the model recovered with Eq. (2.13) from the 3D scan. *Post Refinement* refers to the models further refined with the physics-based losses.

2.5 Conclusion

We have shown that physics-based self-supervision can be leveraged to learn a single parameterization for many different garments to be draped on human bodies in arbitrary poses. Our approach relies on UDFs to represent garment surfaces and on a displacement field to drape them, which enables us to handle a continuous manifold of garments without restrictions on their topology. Our whole pipeline is differentiable, which makes it suitable for solving inverse problems and for modeling clothed people from image data.

One interesting direction for future work deals with modeling dynamic poses instead of only static ones. This is of particular relevance for loose clothes, where our reliance on the SMPL skinning prior should be relaxed. Another future path for this work concerns the possibility of replacing the current global latent code that we used for garments by a set of local codes to yield finer-grained control both for garment editing and draping.

Chapter 3

ScanNeRF: a Scalable Benchmark for Neural Radiance Fields

3.1 Introduction

What is the Metaverse? *Stephenson* coined this portmanteau in his novel *Snow Crash*, hypothesizing that in the 21st century humans, thanks to goggles, would be immersed in virtual worlds mixed with real ones. And here we are! At the time, however, the technology to realize the Metaverse was still hypothetical, but today Cross Reality (XR or Extended Reality) is a fact. XR is made up of a multitude of technologies and variants, such as Virtual Reality and Augmented Reality, but they all share a single paradigm: seamless interaction between virtual environments, digital objects and people. That is the Metaverse! But it does not exist yet, and all that is Digital is often only a virtual representation of the real world. How much will it cost us, then, to transport all our real world into the virtual one?

For Computer Vision and Computer Graphics experts, it is clear what it means to transport an object from the real world to the virtual world: a 3D reconstruction! But 3D reconstructions are expensive, slow, and not all types of objects can be digitized. Yet today, thanks to Deep Learning, we have another way to teleport objects into the Metaverse: Neural Rendering [133]. The basic idea is simple: why reconstructing an object in 3D if we have to render it back in 2D to visualize it by a VR / AR viewer? Neural Rendering (NR) allows us to ask a neural network "render this object from this point of view", et voilà! Moreover, some of the state-of-the-art NR approaches – *e.g.*, Neural Radiance Fields (NeRFs) [24] – allow us to deploy a simple MLP to represent an entire scene (or object), squeezing the spatial cost of a digital object from Gigabytes to a few Kilobytes.

In this chapter, we will focus on one key aspect: the gate to the Metaverse. We have built an effective object scanning station[1], dubbed *ScanNeRF*, which allows for generating ready-to-use data to train and evaluate state-of-the-art Neural Radiance Fields techniques. Using this efficient and simple scanning system, we generated the first real dataset with high

[1]The scan station was built in collaboration with eyecan and won the first prize in the OpenCV Spatial AI Contest 2022.

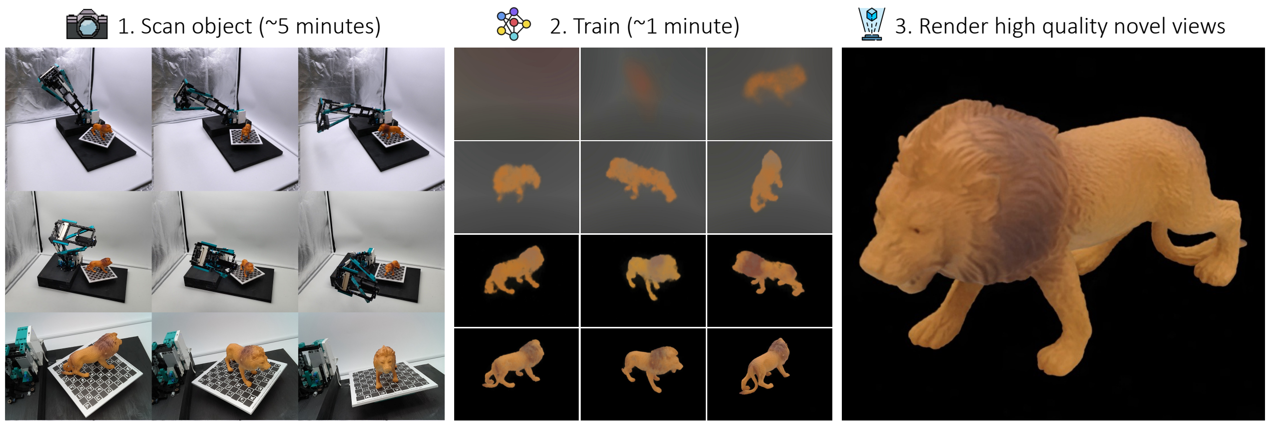

1. Scan object (~5 minutes)
2. Train (~1 minute)
3. Render high quality novel views

FIGURE 3.1. **Overview of the ScanNeRF framework.** Our scan station (left) allows for collecting thousands of images of an object in a few minutes. Then, modern NeRF variants [79, 78, 32] can be trained on them in few minutes (center), producing a digital twin of the object itself and allowing for high-quality, novel view synthesis of it (right).

quality images, pixel masked objects, controlled and repeatable camera poses, specifically designed to evaluate NeRFs. Firstly, this allows us to realize a benchmark for research in the area of Neural Rendering. Secondly, it enables to formally describe which and how many views are best for generating a virtual representation of an object, as well as to unveil some intriguing challenges for the future – *e.g.*, how to fully render an object from any viewpoint, given images mostly collected from a single side of it.

To the best of our knowledge, our work is the first to show that with a simple hardware, made of LEGO, and a low budget – less than 500$ – it is possible to build digital twins of **real** objects, rather than focusing on synthetic ones as in most of NeRF papers [24].

Fig. 3.1 presents an overview of our framework. Fitting a NeRF on the scanned object produces a *digest* of it, ready to be transported into the Metaverse. Actually, this representation is very different from that of a classical Digital Twin, this is indeed a Neural Twin®.

Our contributions are as follows:

- We present a simple, yet effective platform for collecting thousands of images to train NeRFs, or in general, NR frameworks.
- We release a novel benchmark, ScanNeRF, featuring thousands of images depicting real objects collected in inward-facing setting.
- For each object in the benchmark, we define a multitude of train/val/test splits in order to study different properties and stress the performance of NeRF variants. Moreover, we evaluate the performance of three modern NeRFs on these splits, to highlight their strengths and weaknesses under different experimental settings.

Dataset	Type	# Total scenes	# Images per scene	Train splits	Test splits	Withheld images
NeRF Blended [24]	Synth.	8	300	1	1	No
BlendedMVG [134]	Synth.	508	200-4 000	NA	NA	No
LLFF [48]	Real	8	30	1	1	No
DTU [135]	Real	124	49-64	NA	NA	No
CO3D [136]	Real	18 619	100	2	2	Yes[2]
ScanNet [137]	Real	1613	500-5 000	NA	NA	No
Tanks & Temples [138]	Real	14	4 000-20 000	NA	NA	No
ScanNeRF (ours)	**Real**	**35**	**4 000**	**12**	**9**	**Yes**

TABLE 3.1. **Comparison between datasets.** We report properties of existing datasets and our ScanNeRF benchmark.

3.2 Related Work

NeRF and follow-up implementations are usually evaluated on a few, established benchmarks belonging to two acquisition settings, namely forward-facing and inward-facing, The most popular benchmarks are NeRF blender [24], made of 8 synthetic inward-facing scenes with 100 training images and 200 testing images, and LLFF [48], consisting of 8 forward-facing scenes counting about 30 images each. More recently, MVS datasets such as DTU [135], Tanks & Temples [138] and BlendedMVG [134] have been used for this purpose, together with a few more such as CO3D [136] and ScanNet [137] collected through extremely time-consuming practises.

We argue that the aforementioned benchmarks limit the evaluation of NeRF variants under different aspects, since i) some of them [24, 48, 135] provide a few hundred images only, ii) none of them allows for seamlessly scaling the amount of training images or their distribution across the scene and iii) none explicitly defines a testing set – *i.e.*, the evaluation is carried out on images available to the researchers, possibly leading to biased results. In this work, instead, we implement a framework allowing for scalable data collection of a multitude of scenes. For each of them, we explicitly define a testing set, made of frames for which only camera poses are made publicly available, while images are withheld to avoid unfair evaluation. This paves the way towards establishing a next-generation benchmark for research in Neural Radiance Fields and related techniques. Tab. 3.1 shows a comparison between the existing datasets introduced before and the proposed ScanNeRF bechmark.

3.3 The ScanNeRF Benchmark

In this section, we describe both hardware and software components of our ScanNeRF framework. We start by introducing our acquisition platform, then we describe the post-processing steps implemented to select the final images and the masking strategy used to

[2]The possibility to evaluate on withheld images has been added in a second version of the dataset, released concurrently with our work.

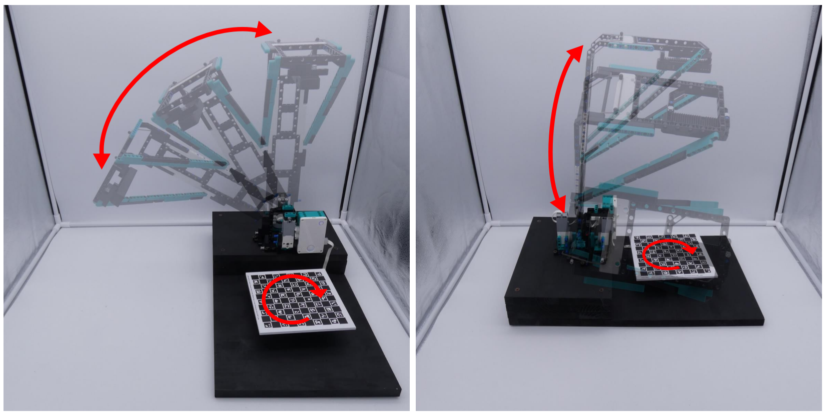


FIGURE 3.2. **The scan station.** Front and side view of our platform, with rotating angles overimposed in red.

extract objects. To conclude, we highlight the overall organization of the produced dataset.

3.3.1 Scan Station Setup

The *scan station* (see Fig. 3.2) that we use to generate the dataset has been built using the Lego Mindstorm toolkit (code 51515)[3] and mounts an OpenCV Oak-D Lite camera[4] to collect images. The system is composed of a rotating base, where the object is placed during scanning, and a robotic arm holding the camera over the base. Acquisitions are carried out inside a light box, in order to minimize effects due to shadows. The base and the arm are fixed on a shared structure, the latter being placed on a higher level with respect to the base, so as to allow for capturing high objects entirely.

The arm has been built using two Lego motors (id: 6299646)[5] connected in series to a gearbox, which holds the arm. We use two motors and a gearbox to deploy more mechanical torque, since the arm and the camera are too heavy for the single motors alone.

The base is driven by a single, additional Lego motor, with a ChArUco board fixed on top of it, which is used to compute the camera pose for each acquired image. This is achieved by calibrating both the intrinsic and extrinsic parameters of the camera based on the ChArUco marker and on the standard algorithm implemented using the functionalities made available by the OpenCV library[6].

To acquire images from poses that are evenly distributed on the hemisphere around the object to be scanned, the arm descends from its initial position, located vertically over the base (zenith angle $\sim$20°), to its final position, located horizontally with respect to the base (zenith angle $\sim$75°), performing sixteen total steps. After each descending step, the arm

[3]`https://www.lego.com/product/robot-inventor-51515`
[4]`https://docs.luxonis.com/projects/hardware/en/latest/pages/DM9095.html`
[5]`https://www.lego.com/en-us/product/medium-angular-motor-88018`
[6]`https://docs.opencv.org/3.4/da/d13/tutorial_aruco_calibration.html`

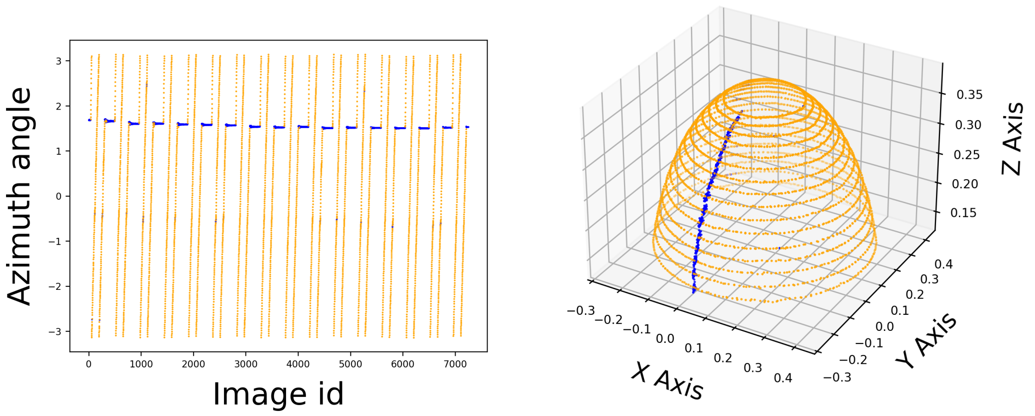


FIGURE 3.3. **Filtering step.** For any collected image, we show the azimuth difference with respect to the previous one (left) and its position on the hemisphere over the object (right). We split the set of collected images into filtered (blue) and remaining ones (orange).

motors are stopped to hold the position, while the base performs two complete rotations (720°), to ensure a dense set of acquisitions. During the whole process, the OAKD-Lite camera records images at 30 FPS frequency and 1440 × 1080 resolution. The scan station has been programmed in python using the API of the Lego Mindstorm Desktop app and is controlled via bluetooth connection.

Combining the two degrees of freedom given by the arm and the rotating table enables to collect images all around the scanned object with very low effort, as well as to implement our scan station with an hardware budget resulting lower than 500$.

3.3.2 Dataset Filtering

After a complete scanning cycle, we obtain roughly 9000 images. As images are acquired throughout the whole cycle, some of them are captured during arm descent, *i.e.*, the step towards the following zenith angle. This causes strong, undesired oscillation of the scan station, with consequent acquisition of several images which are blurred or out from the main trajectory. A first cleaning step consists in removing such images, keeping only those obtained when the arm is not moving and the base is rotating. We observe that the rotation of the base can be detected by computing the azimuth angle of the camera pose in each image and detecting the intervals where the angle between subsequent images is increasing. Thus, we discard every image whose azimuth angle differs from the previous one by less than a fixed threshold, set to 1.15°. Fig. 3.3 shows, for an entire scanning cycle, the filtered (blue) and kept (orange) images. We can notice how selecting the acquisitions with smaller azimuth difference (left) effectively removes the images collected during arm descent (right).

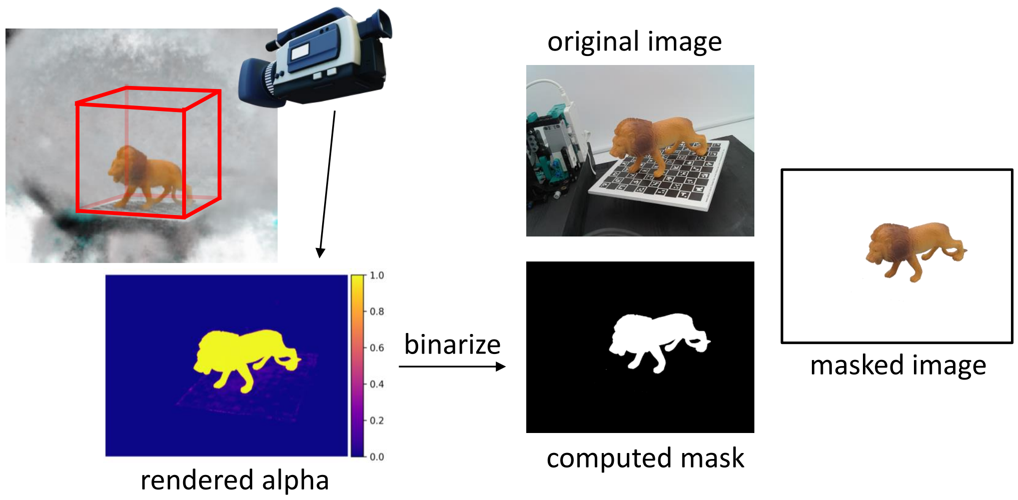


FIGURE 3.4. **Masking procedure.** We train Instant-NGP [32] by placing the rendering bounding-box over the ChArUco board, so as to remove the background and obtain a mask to be applied to the real image.

3.3.3 Background Masking

In our pipeline, we achieve the motion of the camera around the object by properly moving the scan station arm and rotating the base on which the object is placed. This procedure presents a major side effect: the background is not coherent with the computed camera poses, since it remains still during the acquisition of the images. For this reason – and also to obtain more pleasant images featuring only the scanned object – we mask out the background.

Purposely, we exploit a neural rendering framework. First, we train Instant-NGP [32] on the acquired images, which include the background. Then, we use Instant-NGP to render new images from the same poses as the original images, defining the rendering volume to fit the ChArUco marker dimensions in order to crop out the incoherent background (Fig. 3.4, top left). In particular, the rendering volume is placed above the scan station base with a small offset on the Z axis so as to remove the ChArUco marker from the rendered image. This allows us to obtain rendered images featuring the object on a black background. Then, we binarize the rendered images based on the alpha values (*i.e.*, density) of the pixels (Fig. 3.4, bottom left) to generate the desired masks (Fig. 3.4, bottom right). These masks are applied to the original images in order to remove both the background and the scan station base, leaving the object alone in the final images provided by our scan station (rightmost picture in Fig. 3.4).

3.3.4 Dataset Organization and Splitting

Once the undesired frames have been removed and the remaining ones have been properly masked to remove the background and the scan station, we first divide each acquired

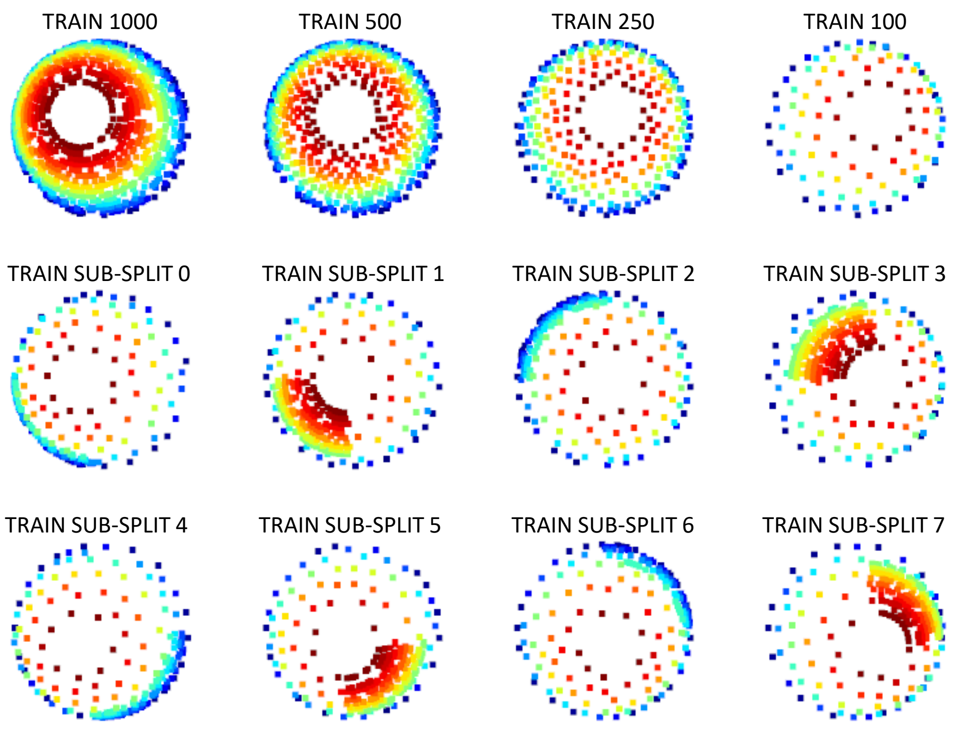


FIGURE 3.5. **Overview of the dataset splits.** On the first row, evenly sampled splits with varying density. On the second and third rows, eight sub-splits with densely localized acquisitions. Point are colored according to the Z coordinate, for a better visualization of their 3D position.

sequence into three macro-splits, namely *Train*, *Val* and *Test*, so that they contain 1000, 500 and 500 images, respectively. We will release the Train and the Val splits publicly, while we will keep private the Test split in order to enable a fair evaluation of the submissions that other researchers would be willing to upload on our website. For each split, we obtain images taken from positions evenly scattered on the hemisphere above the object by applying the Farthest Point Sampling algorithm [139] to the 3D positions from which the images were captured.

From the 1000 images of the Train macro-split, we sample 3 smaller training splits, containing 500, 250 and 100 images, captured uniformly from the whole hemisphere, as shown in Fig. 3.5 first row. These additional training splits are designed to compare the performances of NeRF algorithms when trained on splits with different number of images.

Moreover, every Train/Val/Test macro-split is used to obtain eight additional sub-splits, each containing images acquired more densely in a specific region and only a small portion of images taken from positions scattered across the whole hemisphere (Fig. 3.5, second and third rows). Specifically, we first divide the hemisphere into eight sub-regions, by splitting each range of the X, Y and Z axes in two. Then, sub-splits are sampled from the 1000/500/500 Train/Val/Test images, by retaining all the images collected from viewpoints in the sub-regions ($\sim$ 120/60/60, with small fluctuations depending on the selected region), together with 10% additional frames randomly sampled from the remaining portion of the hemisphere

($\sim$ 80/40/40).

We designed these sub-splits to investigate on the performance of different NeRF proposals when the training set is characterized by an uneven spatial distribution of vantage points and, thus, foster future research in this direction.

3.3.5 Scan Time and Number of Objects

The pipeline sketched so far allows for effortless scanning of a large amount of objects. Specifically, an entire acquisition cycle requires about 5 minutes to collect roughly 9000 images, reduced to about 4000 after the filtering step described in Sec. 3.3.2. At the time of writing, the ScanNeRF dataset counts 35 real objects over which we evaluate the performance of modern NeRF frameworks, as reported in the next Section. Moreover, we plan to scale up our dataset to hundreds (or even thousands!) of objects and distribute the associated Train/Val splits through our benchmark website (`https://eyecan-ai.github.io/scannerf/`).

3.4 Experiments

In this section, we conduct experiments on our novel ScanNeRF dataset. Specifically, we run three modern and efficient NeRF frameworks [79, 78, 32] on the splits we have designed, so as to investigate on how they perform when varying the density and amount of training images, as well as how they behave with images being densely acquired only from a specific region around the scanned object.

3.4.1 Evaluated Frameworks and Settings

We briefly introduce the methods involved in our experiments. The three of them have been selected for our evaluation because of their speed both at training and rendering time. In our opinion, such efficiency makes these methods prominent for future advances in the field.

DVGO [79]. This framework mixes the implicit representation learned by means of MLPs with explicit ones – *i.e.*, voxel grids – to model density and appearance. This allows for training a NeRF in roughly 15 minutes.

Plenoxels [78]. A voxel grid is diretly optimized by this method, getting rid of any neural network. Spherical harmonics are used to model view-dependent RGB values. Training time for a single scene takes about 10 minutes.

Instant-NGP [32]. This framework deploys a multi-resolution hash table of trainable feature vectors, allowing the use of a much smaller neural network and achieving faster convergence. For a single training, approximately 1 minute is enough to reach high-quality renderings.

	1000 training images			500 training images			250 training images			100 training images		
Scene	DVGO*	Plenoxels	Instant-NGP	DVGO*	Plenoxels	Instant-NGP	DVGO*	Plenoxels	Instant-NGP	DVGO*	Plenoxels	Instant-NGP
airplane1	38.90	34.59	37.14	38.97	33.49	36.40	38.41	27.44	37.57	36.69	22.81	37.30
airplane2	39.82	35.21	37.86	39.85	33.69	38.38	39.46	27.21	37.61	37.60	23.36	37.44
brontosaurus	41.56	34.74	39.95	41.46	30.18	39.99	40.76	24.67	39.93	38.62	20.43	39.96
bulldozer1	35.84	32.05	34.99	35.95	29.78	34.72	35.70	23.68	34.90	34.05	19.34	34.72
bulldozer2	39.16	34.21	38.12	38.96	34.33	37.65	37.96	32.45	38.30	36.12	26.40	38.09
cheetah	37.86	33.35	35.68	37.87	32.47	35.24	37.64	29.54	21.82	36.09	23.49	35.59
dumptruck1	37.93	33.90	36.61	37.93	32.41	36.78	37.44	27.14	36.60	35.63	22.01	36.65
dumptruck2	41.34	35.45	39.96	41.01	34.16	39.44	40.00	30.20	38.82	38.01	25.57	39.93
elephant	38.62	32.11	36.49	38.65	25.10	36.21	38.25	21.04	34.65	36.42	18.06	36.01
excavator	40.87	35.23	38.65	40.65	35.33	39.59	39.82	33.74	38.48	37.83	26.90	39.77
forklift	37.95	32.99	37.82	37.71	33.09	38.22	36.63	32.13	37.68	34.59	25.87	37.80
giraffe	36.67	32.38	34.42	36.72	31.25	34.54	36.45	26.61	34.65	34.78	21.97	34.26
helicopter1	39.77	35.52	37.71	39.73	33.35	36.84	39.29	27.55	37.57	37.56	22.81	36.98
helicopter2	38.05	33.68	36.46	38.11	32.30	36.93	37.66	26.96	36.69	35.97	21.67	36.43
lego	34.52	30.42	33.92	34.58	26.32	33.79	34.33	22.15	33.88	32.78	19.44	33.79
lion	39.16	33.50	38.21	39.16	26.41	38.24	38.73	22.20	37.47	36.89	19.33	34.91
plant1	40.31	34.41	37.21	40.34	28.29	37.23	39.72	22.72	37.42	37.44	19.99	37.03
plant2	42.19	36.61	38.86	42.18	34.07	38.98	41.42	27.38	38.38	39.35	23.01	27.53
plant3	33.63	29.33	33.81	33.58	24.17	34.08	33.11	20.49	34.21	30.47	18.46	33.18
plant4	38.08	32.94	36.43	37.97	29.15	36.55	37.71	25.51	36.97	35.86	22.15	36.79
plant5	39.10	34.30	38.11	39.06	28.02	36.64	38.48	24.01	37.18	36.28	20.79	37.99
plant6	36.76	30.87	34.25	36.84	25.30	35.19	36.46	21.12	35.15	34.51	19.13	35.05
plant7	37.15	31.87	35.57	37.16	26.55	35.43	36.64	20.62	35.50	34.85	18.98	35.36
plant8	39.04	33.47	36.68	39.04	28.13	36.74	38.46	22.06	36.61	36.36	19.93	36.34
plant9	40.05	33.79	37.52	40.07	27.44	37.39	39.36	22.03	37.44	37.42	19.57	37.51
roadroller	39.96	34.66	39.18	39.62	34.59	39.66	38.84	33.46	38.94	36.61	27.28	39.37
shark	39.95	32.88	38.33	39.88	25.31	38.44	39.25	19.98	38.15	37.00	17.78	38.28
spinosaurus	40.86	34.96	39.31	40.88	32.73	39.09	40.44	25.81	39.32	38.71	21.74	39.21
stegosaurus	39.07	33.89	38.60	39.25	29.32	37.96	38.82	25.22	38.36	37.37	22.47	38.52
tiger	37.67	32.87	36.41	37.26	30.20	36.38	37.36	24.65	36.39	35.46	20.44	35.95
tractor	34.02	30.55	33.51	34.10	28.67	33.88	33.87	23.34	33.31	32.42	19.32	33.73
trex	37.97	32.99	37.82	38.11	29.12	37.91	37.74	22.46	37.49	35.70	18.88	38.03
triceratops	41.56	35.89	39.31	41.52	32.50	40.04	40.97	25.91	39.74	39.19	22.69	39.80
truck	37.70	33.67	36.36	37.68	32.80	36.64	37.30	27.53	36.66	35.67	22.44	36.50
zebra	35.06	30.32	33.49	35.10	30.32	33.32	34.84	29.71	33.12	33.63	26.39	33.12
avg	38.52	33.42	36.99	38.48	30.30	36.99	37.98	25.68	36.48	36.11	21.74	36.54

TABLE 3.2. **Results on evenly distributed images.** We report results in terms of PSNR on the test splits of 35 scanned objects, when training with varying amount of images (from left to right, 1000, 500, 250 and 100 respectively). * indicates that DVGO has been trained and tested with half resolution images due to memory constraints.

Training setups. For each method, we run experiments using the official code released by the authors, keeping the same default hyper-parameters defined in the source code during training except i) for Instant-NGP, for which we reduced the amount of training step from 100K to 10K without any loss of final rendering quality, and ii) for DVGO, where we train and render half resolution images for the sake of memory constraints. In our evaluation, we trained 420 instances for each model (140 for evenly distributed acquisitions, 280 for densely localized splits). Each training is performed on a single NVIDIA 3090 RTX GPU, requiring a total of about 175 hours/GPU for training.

Evaluation metrics. To assess the quality of the rendered images, we compute the Peak Signal Noise Ratio (PSNR) between the rendered ($\hat{x}$) and real test (x) images:

$$\text{PSNR}(\hat{x}, x) = -10 \log_{10}(x - \hat{x})^2. \tag{3.1}$$

3.4.2 Experiments on Evenly Distributed Acquisitions

We start by training and evaluating the three methods when dealing with evenly distributed images taken from all around the hemisphere over the scanned object. Tab. 3.2 collects experiments over 35 objects scanned by our scan station. From left to right, we report the results on the evenly distributed images of the test split achieved by training on the 1000, 500, 250 and 100 images training splits, respectively. We can notice how all the three NeRF variants excel when trained on 1000 images, always achieving more than 30 PSNR. In general, Instant-NGP yields higher rendering quality compared to Plenoxels, while DVGO produces very good results as well, although not directly comparable with the other methods because of the limiting requirement to work with half resolution images. When gradually reducing the density of the training images to 500, 250 and 100, we can notice different effects on the three frameworks. Instant-NGP achieves almost unaltered quality of the rendered images, DVGO suffers a moderate drop in terms of PSNR (about 2 points when trained on the smallest training set), while Plenoxels seems to suffer the highest drop of render quality, falling to about 20 PSNR when trained with 100 images only.

According to this benchmark, Instant-NGP seems the best choice at the time of writing, thanks to its extremely fast training and rendering speed, its overall high quality and its robustness to decreasing amounts of training images.

Fig. 3.6 shows some renderings obtained by DVGO, Plenoxels and Instant-NGP when trained on 1000 images.

3.4.3 Experiments on Densely Localized Acquisitions

After experimenting on evenly distributed acquisitions, we focus on the densely localized ones. The goal of this experiment is to stress the capability of NeRF algorithms to generate novel views from positions all over the hemisphere, after training on images captured mainly from a localized region of the space, with just few samples evenly distributed on the hemisphere.

We adopt the following protocol: for every object of our dataset, we perform eight trainings for each of the three selected NeRF algorithms (one training for every train sub-split described in Sec. 3.3.4). Then, starting from each training, we test the three algorithms on all the eight test sub-split, performing a total of 64 evaluations for each object.

Tab. 3.3 reports the results of this experiment for each selected NeRF method, averaging them on the 35 objects scanned by our framework. It is possible to observe that, as expected, all the methodologies obtain good PSNR scores (>30) when trained and tested on the same sub-split (*i.e.*, on images acquired from positions with the same distribution over the hemisphere). However, when tested on sub-splits coming from dense acquisition different from the training ones, their behavior is different from case to case. Plenoxels suffers significantly from this setting, with a PSNR drop up to 8 points that leads to poor results (∼22 PSNR). DVGO, instead, appears to be more robust, with a PSNR drop inferior to 4 points. Instant-NGP, finally, seems to be the more resilient to the described stress test, with a PSNR drop of just 1 point in the worst cases.

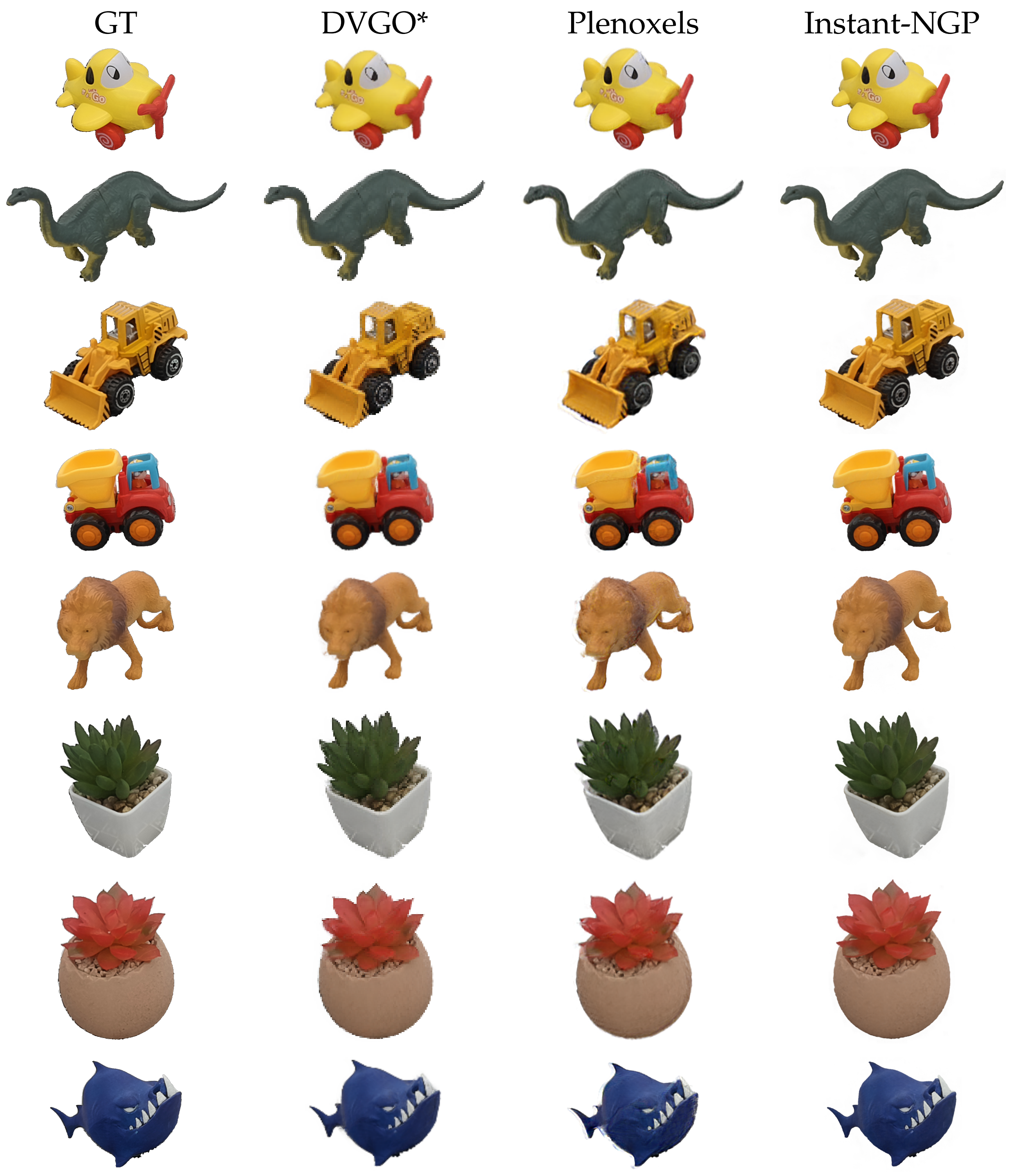


FIGURE 3.6. **Qualitative results obtained by training with 1000 images.** From left to right: ground-truth, image rendered from DVGO (half resolution), image rendered from Plenoxels, image rendered from Instant-NGP.

	Test Split							
Train Split	0	1	2	3	4	5	6	7
0	39.07	36.54	36.45	35.81	36.51	35.76	36.67	35.97
1	37.14	38.36	36.03	35.49	36.04	35.57	36.28	35.57
2	36.74	36.01	38.91	36.37	36.22	35.64	36.86	36.00
3	36.33	35.75	36.91	38.26	35.87	35.31	36.41	35.74
4	36.77	35.95	36.15	35.65	38.78	36.34	36.83	36.07
5	36.26	35.68	35.72	35.23	36.98	38.09	36.46	35.83
6	36.58	35.96	36.42	35.72	36.57	35.85	39.20	36.58
7	36.22	35.61	36.04	35.56	36.15	35.56	37.26	38.43

DVGO*

	Test Split							
Train Split	0	1	2	3	4	5	6	7
0	31.05	24.74	24.68	22.37	24.91	22.55	24.46	22.27
1	27.97	30.10	24.62	23.15	24.85	23.60	24.33	22.45
2	25.10	22.62	31.37	25.01	24.02	21.86	25.47	22.67
3	24.81	23.32	28.09	30.17	23.50	21.72	24.85	23.30
4	25.16	22.56	24.09	22.00	31.17	25.22	25.47	22.84
5	24.83	23.17	23.66	22.06	28.18	30.30	25.16	23.87
6	24.20	22.15	24.96	22.64	24.79	22.35	31.36	25.06
7	23.90	22.31	24.73	23.45	24.72	23.21	28.02	30.10

Plenoxels

	Test Split							
Train Split	0	1	2	3	4	5	6	7
0	36.98	36.10	36.43	36.04	36.34	35.75	36.31	35.92
1	36.23	36.99	36.19	36.24	36.14	35.93	36.07	35.95
2	36.54	36.31	37.40	36.64	36.54	36.17	36.61	36.29
3	36.17	36.18	36.53	37.26	36.19	35.94	36.23	36.21
4	36.39	36.00	36.48	36.12	37.12	36.10	36.46	36.09
5	36.07	36.14	36.20	36.16	36.45	36.94	36.21	36.21
6	36.43	36.25	36.63	36.42	36.58	36.15	37.28	36.48
7	36.17	36.11	36.39	36.36	36.31	36.15	36.50	37.20

Instant-NGP

TABLE 3.3. **Results on densely localized sub-splits.** From top to bottom: DVGO (half-resolution), Plenoxels and Instant-NGP. We show results in terms of PSNR, averaged over the 35 scanned objects, for models trained on one of the eight densely localized sub-splits (rows) and tested on any of the eight sub-splits (columns).

We conjecture that the superior performances achieved by DVGO and Instant-NGP wrt Plenoxels can be explained considering that the former two methods rely on a MLP which is not present in the latter. This component can probably learn strong biases from the few evenly scattered samples, which help DVGO and Instant-NGP to generalize to (almost) unseen regions of the hemisphere.

3.5 Conclusion

In this chapter, we have introduced ScanNeRF, a scalable benchmark for neural radiance fields and, in general, neural rendering frameworks. ScanNeRF consists of a simple, yet effective hardware/software pipeline allowing for collecting thousands images of an object effortlessly and in a few minutes. Our platform results ideal to scan a multitude of different objects, which together build up the ScanNeRF benchmark[7], a novel dataset made of 35 scenes counting thousands of images each. In our experiments, we stressed the potentialities of modern NeRF frameworks [79, 78, 32] under different settings thanks to the peculiar training/validation/testing splits made available by ScanNeRF, highlighting some new challenges for the community to face. We believe ScanNeRF will play a role in fostering the research in neural radiance fields frameworks.

[7] https://eyecan-ai.github.io/scannerf/

Part II

Processing NR of Signals

Chapter 4

Learning the Space of Deep Models

4.1 Introduction

Representation learning has achieved remarkable results in embedding text, sound and images into low dimensional spaces, so as to map semantically close data into points close one to another into the learnt space. In recent years, deep learning has emerged as the most effective machinery to pursue representation learning, many scholars agreeing on representation learning laying at the very core of the deep learning paradigm. On the other hand, the success of network compression and pruning approaches [140] highlight the redundancy of parameters learned by a deep learning model, as in the Lottery Ticket Hypothesis [141], which shows that training as few parameters as 4% of those of the full network (*i.e.*, the *winning tickets*) can attain similar or even higher performance.

Thus, we felt puzzling and worth investigating whether the parameter values of a trained deep model might be squeezed into a semantically meaningful low-dimensional latent space. Two questions arise: is it possible to train a deep learning model to learn to represent other, already trained, deep learning models? And according to which trait should two already trained models lay either close or further away in the latent space? The Lottery Ticket Hypothesis may suggest the existence of a low-dimensional key set of information that is shared by all possible sets of parameters for a predefined architecture that achieve comparable performance on a given task. Hence, it seems reasonable to conjecture that one might pursue learning of an embedding space shaped according to similarity in performance. Moreover, many recent works have demonstrated how small deep networks can be trained to fit accurately complex signals such as images[22], implicit representations of 3D surfaces [25, 27] and even radiance fields [24]. One might then be willing to embed such models into a space amenable to capture the similarity between the underlying signals.

In this chapter we propose a first investigation along this new line of research. In particular, we show that it is possible to deploy a basic encoder-decoder architecture to learn a low-dimensional latent space of deep models and that such a space can be shaped so to exhibit a semantically meaningful structure. We posit that the loss to drive the learning process of our encoder-decoder architecture should entail functional similarity – rather than proximity of parameter values – between the input and output models. Accordingly, we train our architecture by knowledge distillation to drive the output model generated by the decoder to mimic the behaviour of the input model. In our study we address two settings: learning a latent space from a training set of models with the same network architecture and different parameter values as well as based on a training set comprising models with different architectures. In both settings, we show that the learnt latent space does posses a semantic structure as it is possible to sample new trained models with predictable behaviour by simple interpolation operations. Moreover, we show that in the Multi-Architecture setting a latent space trained on a set of architectures can generate already-trained models of architectures never seen instantiated at training time. Finally, we show that in both settings it is possible to train an architecture by performing latent space optimization on the low dimensional embedding space instead of optimizing directly the full set of parameters.

4.2 Related Work

Representations. Representation learning concerns the ability of a machine learning algorithm to transform the information contained in raw data in more accessible form. A common algorithm is the autoencoder [142], a self-supervised solution where the representation is learnt by constraining the output to reconstruct the input. Our architecture is inspired by the autoencoder but aim at producing outputs that behave akin to the input (*e.g.*, similar performance on a certain task). In a recent meta-learning paper, LEO [143], the embedding of the weights of a single layer of a network is learnt for a few shot learning task. Task2Vec [144] learns a task embedding on different visual tasks which enables to predict similarities between them and how well a feature extractor perform on a chosen task. Differently from all these works, we focus on learning a fixed-size embedding for diverse network architectures from which it is possible to draw ready-to-use weights for a specific task, even for networks unseen during training.

Network Parameters Prediction. Many works deploy an auxiliary network to obtain the weights of a target network. Hypernetworks [145] trained a small network (the hypernetwork) to predict weights for a large target network on a given task. The same technique has been extended and applied in many ways: transforming noise into the weights of a target network (Bayesian setting) [146, 147], adapting the weights of a target network to different tasks [148, 149], generating weights corresponding to hyperparameters [150], focusing on the acceleration of the architecture search problem [151]. Moreover, networks that generate their own weights have been proposed and analyzed [152, 153]. While these works and ours share the use of a weights generation module, our novel proposal consists in showing how to learn a fixed-size structured embedding for different architectures and navigate through this

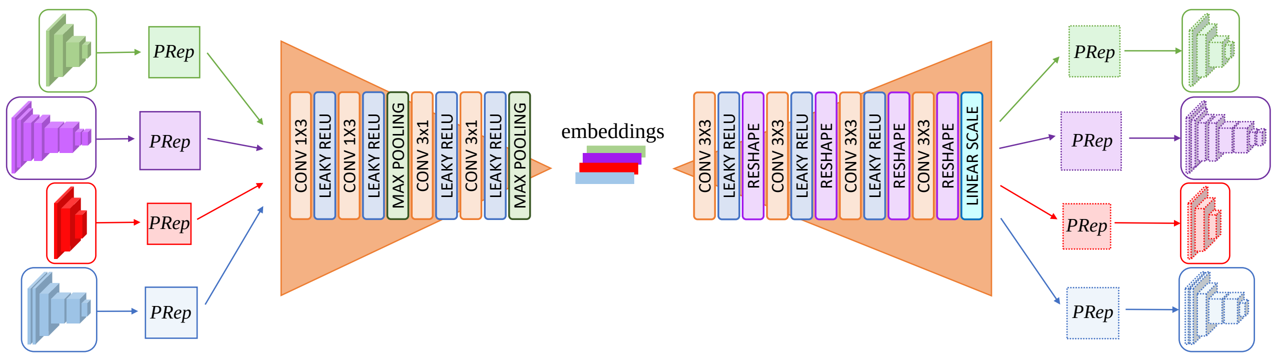


FIGURE 4.1. **Overview of NetSpace.** Our framework takes in input the parameters of a neural network, squeezes them in a compact embedding and predicts the parameters of a new neural network that behaves like the input one starting only from such embedding.

space to obtain new weights for these kinds of architectures as well as for architectures not provided as training examples.

Weight-sharing NAS. In Weight-sharing NAS, optimal architecture search occurs over the space defined by the subnets of a large network, the supernet. Commonly, subnets share weights with the supernet and they are available as ready-to-use networks after training. OFA [154] starts by training the entire supernet and progresses considering subnets of reduced size. After training, desired subnets are selected with an evolutionary algorithm. In NAT [155], many conflicting objectives are considered, training only the weights of promising subnets for every objective. While these works deal with obtaining ready-to-use networks that obey to desired characteristics, we focus on the embeddability of deep models in a latent space organized according to features of interest and on the possibility of explore such latent space by interpolation or optimization.

4.3 Method

In the following, we will use *architecture* to denote the structure of a deep learning model (*i.e.*, number and kind of layers, etc.) and *instance* for an architecture featuring specific parameter values. Of course, given one architecture there can be many instances with different parameter values.

4.3.1 Framework

Our framework, dubbed NetSpace and shown in Fig. 4.1, is able to encode trained instances of different architectures into a fixed-size encoding and to decode this embedding into new instances that behave like the input ones. The parameters of each instance presented in input to our framework are stored into a PRep (parameters representation), a 2D tensor obtained with a simple algorithm exemplified in Fig. 4.2. We designed our framework fixing

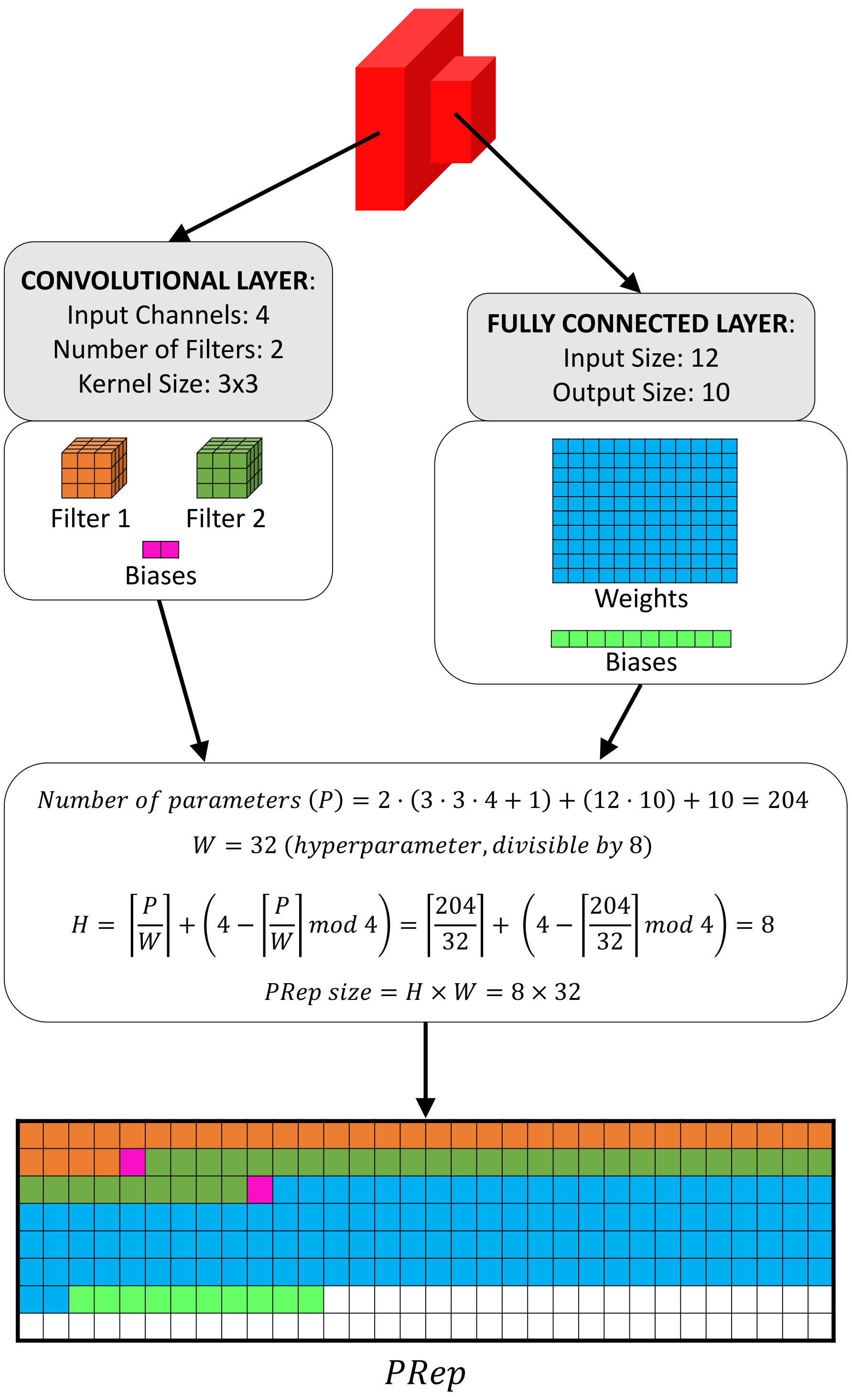

CONVOLUTIONAL LAYER:
Input Channels: 4
Number of Filters: 2
Kernel Size: 3x3
Filter 1
Filter 2
Biases
FULLY CONNECTED LAYER:
Input Size: 12
Output Size: 10
Weights
Biases
$Number\ of\ parameters\ (P) = 2 \cdot (3 \cdot 3 \cdot 4 + 1) + (12 \cdot 10) + 10 = 204$
$W = 32\ (hyperparameter, divisible\ by\ 8)$
$H = \left\lceil \frac{P}{W} \right\rceil + \left(4 - \left\lceil \frac{P}{W} \right\rceil mod\ 4\right) = \left\lceil \frac{204}{32} \right\rceil + \left(4 - \left\lceil \frac{204}{32} \right\rceil mod\ 4\right) = 8$
$PRep\ size = H \times W = 8 \times 32$
$PRep$

FIGURE 4.2. **Algorithm to compute the PRep of a given instance.** We consider here a toy architecture made out of one convolutional layer and one fully connected layer and we fix the PRep width to 32. White cells represent padding with constant value 0.

PReps to be rectangular matrices with high ratio between width and height. Additionally, to favor easy implementation, the height and the width of a PRep are chosen to be multiple of 4 and 8, respectively. Considering an architecture with P parameters and having set the width of the PRep to a number W (divisible by 8), the minimum necessary height of the PRep can be computed as $\lceil \frac{P}{W} \rceil$, adding a padding of $4 - (\lceil \frac{P}{W} \rceil \mod 4)$ rows, if needed, to fulfill divisibility by 4. Given an instance of a neural network and a chosen PRep size, the algorithm to produce the instance PRep is straightforward: parameters from all the layers of the instance are copied in the matrix in sequence one row after the other and final zero-padding is added as needed to match the required size.

NetSpace encoder takes as input the PRep of an instance and produces a small fixed-size embedding, applying first horizontal and then vertical convolutions, alongside with max-pooling. It is worth pointing out that the encoder is designed to produce embeddings of the same size for any input PRep dimension.

The embedding from the encoder is then processed by NetSpace decoder, whose basic block first applies convolutions to increase the depth of the input and then reshapes the intermediate output to grow along spatial dimensions at the cost of depth. Once the required PRep resolution has been reached, an independent linear scaling is applied to every element of the predicted PRep, with weights and biases learnt during the training. We found that this is needed, in particular, for very deep models, probably because convolutions struggle to predict parameters that are close in the PRep but that belong to distant layers of the target architecture. The values of the predicted PRep are loaded into a ready-to-use instance.

The building blocks of the encoder and decoder are specified in more details in Fig. 4.1. In the remainder of the chapter, we will use the term *target* instance to refer to the one in input to NetSpace and the term *predicted* instance to refer to that instantiated with values from the predicted PRep.

4.3.2 Single-Architecture Setting

In the Single-Architecture setting, NetSpace is used to learn an embedding space for the parameters of multiple instances of a single architecture. The first scenario that we consider deals with instances that exhibit different *performance* in solving the same task, such as image classification. Thus, during NetSpace training, the objective is to learn how to predict weights that match the performance of the target instances. Akin to common practice in Knowledge Distillation [156], this can be achieved by minimizing a loss term $\mathcal{L}_{pred}$ that represents the discrepancy between the outputs computed by the target instances and those computed by the corresponding predicted instances. Formally, considering a target instance N_t and training samples x with labels y, we denote by N_p the instance predicted by NetSpace when the input instance is N_t, and by $t = N_t(x)$ and $p = N_p(x)$ the logits computed by the target and predicted instances, respectively. We then realize $\mathcal{L}_{pred}$ as in [156]:

$$\mathcal{L}_{pred} = KL(\text{softmax}(p/T),\ \text{softmax}(t/T)) \cdot T^2 \tag{4.1}$$

where KL denotes the Kullback–Leibler divergence averaged across the samples. As in [156], the softmax functions used in $\mathcal{L}_{pred}$ have inputs divided by a temperature term T.

A second scenario deals with networks sharing the same architecture that are trained to fit different signals. In particular, recent works [25, 27, 22, 24] have shown that it is possible to build neural representations of signals by training MLPs to regress such signals. In this scenario, each instance of the same MLP architecture is trained to represent a different signal. Given one of such instances, the objective of NetSpace is to predict weights capable of regressing the same signal. To achieve this goal, NetSpace can be trained with a loss term that directly compares the outputs of the predicted instance to those computed by the target one, thereby, also in this case, distilling the knowledge of the target instance into the predicted one. In this scenario, then, $\mathcal{L}_{pred}$ becomes simply:

$$\mathcal{L}_{pred} = MSE(y_p,\ y_t) \tag{4.2}$$

i.e., the Mean Squared Error (MSE) between the outputs from the predicted instance (y_p) and those from the target instance (y_t) when queried by the same inputs. In particular, in the experiments we consider MLPs trained to regress the Signed Distance Function (SDF) of a 3D shape (*e.g.*, [25]).

We found that, in both scenarios, using a distillation loss is more effective than using a weights reconstruction loss, as the latter would aim just at mimicking on average the weights of the target instances, which we found not implying similar predictions. Furthermore, a distillation loss allows for using each target instance to create many training examples for NetSpace by simply varying the input data.

4.3.3 Multi-Architecture Setting

In the Multi-Architecture setting, we investigate on how to embed in a common space instances having different architectures. Thus, we consider instances trained to solve an image classification task with the best performances allowed by their architecture. NetSpace is trained to process such instances and to predict weights that reproduce their good performances. In order to ease NetSpace task, we take advantage of the complete Knowledge Distillation described in [156]. Denoting by N^* a teacher network with good performances in the task at hand and by t^* its logits for a batch of images x, we define the loss with respect to it as:

$$\mathcal{L}^*_{pred} = KL(\text{softmax}(p/T),\ \text{softmax}(t^*/T)) \cdot T^2 \tag{4.3}$$

Then, as in [156], we introduce an additional term $\mathcal{L}_{task}$ which, in combination with $\mathcal{L}^*_{pred}$, defines the complete $\mathcal{L}_{kd}$:

$$\mathcal{L}_{task} = CE(\text{softmax}(p),\ y) \tag{4.4}$$

$$\mathcal{L}_{kd} = \alpha \cdot \mathcal{L}^*_{pred} + (1-\alpha) \cdot \mathcal{L}_{task} \tag{4.5}$$

where CE denotes the Cross Entropy loss averaged across the samples of the batch and α is a hyperparameter used to balance the two terms in $\mathcal{L}_{kd}$.

As far as the possibility of handling different architectures is concerned, we identify each architecture uniquely with a categorical ClassId. In this configuration, NetSpace is trained to predict an instance with the same architecture as the target. Even if this information is available at training time from the target instance itself, we would also like to explore by means of interpolation or optimization the latent space learnt by NetSpace after having trained it, without feeding input instances to the framework. Thus, we wish to be able to extract the architecture information directly from the embedding. To achieve this objective, we modify the architecture presented so far by adding a softmax classifier on top of the embedding in order to predict the ClassId of the target instance (details on this variant of the framework are reported in Appendix B.1). Consequently, we complement the learning objective introduced in Eq. (4.5) with an additional $\mathcal{L}_{class}$ term. Given a target instance N_t and the embedding e produced by NetSpace encoder for it, we denote by c_t the ClassId associated to the architecture of N_t and by c_p the logits predicted by the architecture classifier from e. $\mathcal{L}_{class}$ is then defined as the Cross Entropy loss between the predicted and target ClassId:

$$\mathcal{L}_{class} = CE(\text{softmax}(c_p),\ c_t). \tag{4.6}$$

The initial experimental results highlighted that NetSpace was clustering the latent space according to the architecture ClassId only. We judge such organization of the embedding space as not satisfactory, as it would allow, perhaps, to sample new instances within a cluster by proximity or interpolation, but there would be no simple technique to navigate from one cluster to the others. Rather, we aim at endowing the embedding space with a structure enabling exploration along meaningful directions, *i.e.*, directions somehow correlated to a specific characteristic, such as number of parameters or performance. Thus, a more amenable organization would consist in clusters showing up *aligned*, rather than scattered throughout the space, and possibly also *sorted w.r.t.* a given characteristic of interest.

Should such organization of the embedding space be possible, given two *boundary* embeddings (*i.e.*, representing two instances with the smallest and the largest value of the characteristic of interest), it could be possible to move across the aligned clusters by simply interpolating the boundaries and obtain along the way representations of ready-to-use instances with increasing values of the characteristic of interest. To further investigate along

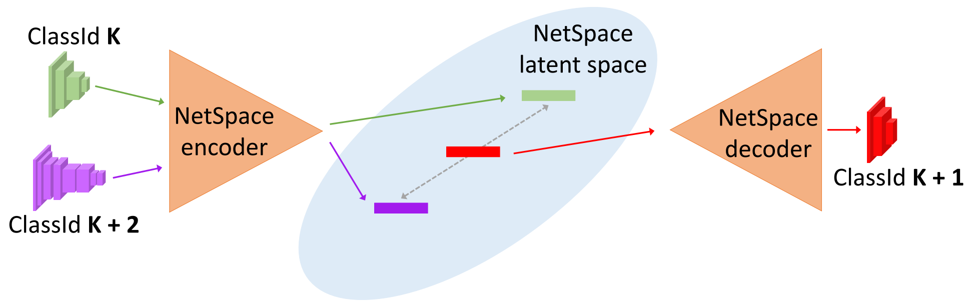


FIGURE 4.3. **NetSpace latent space interpolation.** An example of interpolation in the latent space learnt by NetSpace in the Multi-Architecture scenario.

this path, we shall consider first that it is possible to assign ClassIds to a pool of architectures so as to sort them accordingly to a characteristic of interest. For instance, in our experiments, ClassIds K and $K+1$ will denote two architectures such that the latter has more parameters than the former.

Therefore, we introduce a new loss, denoted as $\mathcal{L}^{\gamma}$ (Interpolation Loss), whose objective is to impose the desired ordered alignment of clusters in the latent space. Given training instances belonging to boundary architectures (*i.e.*, those with the smallest and largest ClassId), we first use NetSpace encoder to obtain their embeddings. Then, we interpolate such embeddings according to a given factor γ, and constrain the interpolated embedding to belong to the architecture whose ClassId is interpolated between the boundaries according to the same factor γ. Fig. 4.3 presents an example of the procedure described in this paragraph.

Formally, given boundary embeddings e^A and e^B of target instances N_t^A and N_t^B with ClassIds c^A and c^B, we define the interpolated embedding $e^{\gamma} = (1-\gamma)\cdot e^A + \gamma \cdot e^B$. Then, considering the logits c_p^{γ} predicted by the ClassId classifier for e^{γ} and the interpolated ClassId $c_t^{\gamma} = (1-\gamma)\cdot c^A + \gamma\cdot c^B$, we define $\mathcal{L}^{\gamma}_{class}$ to impose the consistency of the interpolation factor for ClassId as:

$$\mathcal{L}^{\gamma}_{class} = CE(\text{softmax}(c_p^{\gamma}),\ c_t^{\gamma}). \tag{4.7}$$

Moreover, considering the instance N_p^{γ} predicted by NetSpace from e^{γ}, we denote by p^{γ} the logits predicted by such instance for a batch of images and define $\mathcal{L}^{\gamma}_{kd}$ as:

$$\mathcal{L}^{\gamma}_{pred} = KL(\text{softmax}(p^{\gamma}/T),\, \text{softmax}(t^{*}/T))\cdot T^2 \tag{4.8}$$

$$\mathcal{L}^{\gamma}_{task} = CE(\text{softmax}(p^{\gamma}),\ y) \tag{4.9}$$

$$\mathcal{L}^{\gamma}_{kd} = \alpha\cdot\mathcal{L}^{\gamma}_{pred} + (1-\alpha)\cdot\mathcal{L}^{\gamma}_{task} \tag{4.10}$$

with the objective of distilling the teacher network N^* also in the interpolated instances. Finally, we define the total interpolation $\mathcal{L}^\gamma$ as:

$$\mathcal{L}^\gamma = \mathcal{L}^\gamma_{class} + \mathcal{L}^\gamma_{kd} \tag{4.11}$$

In our framework, we use $\mathcal{L}^\gamma$ with different interpolation factors γ, whose values are computed according to the number of considered architectures. More precisely, considering A architectures, γ can be computed as:

$$\gamma = \frac{i}{A-1} \quad i \in \{1, 2, ..., A-2\}. \tag{4.12}$$

Given a batch of instances, we compute $\mathcal{L}_{kd}$ and $\mathcal{L}_{class}$ on each of them. Then, we apply $\mathcal{L}^\gamma$, with γ values obtained from Eq. (4.12), on all the pairs composed of instances with the minimum and maximum ClassId. The final loss, thus, is the sum of $\mathcal{L}_{kd}$, $\mathcal{L}_{class}$ for each instance of the batch and $\mathcal{L}^\gamma$ for each pair of boundary instances.

4.4 Experiments

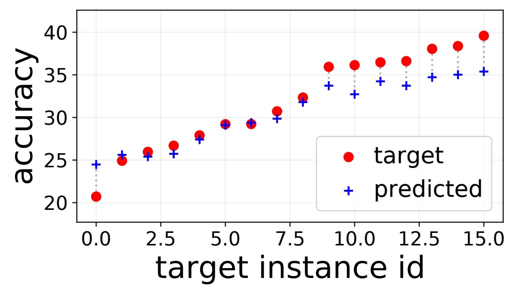
We test NetSpace with networks trained on image classification and 3D SDF regression.

4.4.1 Datasets and Architectures

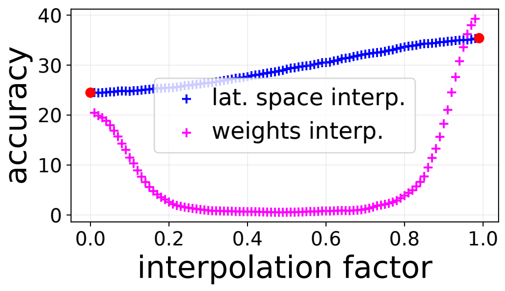
For what concerns image classification, we report results on Tiny-ImageNet (TIN) [157] and CIFAR-10 [158] datasets. The target architectures for our experiments are LeNetLike, a slightly modified version of the lightweight CNN introduced in [159], VanillaCNN, a sequence of standard convolutions followed by a fully connected layer, and two variants of ResNet [160], namely ResNet8, and ResNet32. As far as 3D SDF regression is concerned, we consider MLPs trained to overfit a selection of ~ 1000 *chairs* from the Shapenet dataset [161]. Each MLP has a single hidden layer with 256 nodes and uses periodic activation functions as proposed in [22]. Additional details are available in Appendix B.2.

4.4.2 Single-Architecture Image Classification

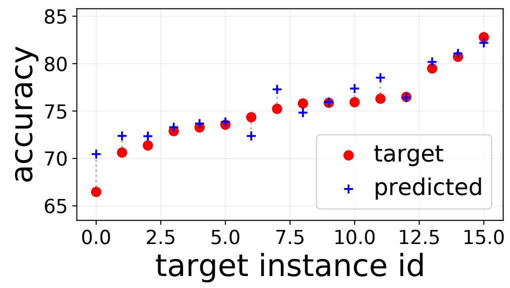
As a first experiment, we test NetSpace in the Single-Architecture setting with the image classification task on CIFAR-10 and TIN. We create a dataset of 132 randomly initialized ResNet8 instances, training them for a different numbers of epochs, to collect instances with different performances. Then we randomly select 100 instances for training, 16 for validation, and 16 for testing. Fig. 4.4a and Fig. 4.4c compare the accuracy achieved on TIN and CIFAR-10 test sets by target and predicted instances. The target instances belong to the test sets and



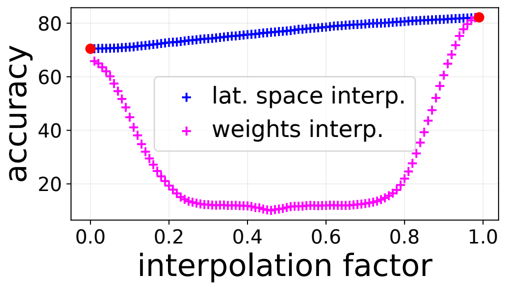
(A) TIN test set



(B) TIN interp.



(C) CIFAR10 test set



(D) CIFAR10 interp.

FIGURE 4.4. **Single-Architecture results for ResNet8.** (a) and (c): Accuracy achieved on the test set by target and predicted instances. Target instances are sorted *w.r.t.* their performances on the test set. (b) and (d): Accuracy achieved on the test set by instances predicted from interpolated embeddings.

were never seen by NetSpace at training time. We can see that, beside few outliers, our framework is effective in predicting new instances that emulate the behavior of the target ones, both on CIFAR10 and on the larger and more varied TIN. It is remarkable that NetSpace is able to reconstruct an instance which follows the input one in terms of performance in spite of the huge compression it introduces. Indeed, the embedding size is a fraction of the number of parameters of the instances it can reconstruct, *e.g.*, it is 4096 for TIN, only $\sim 3.18\%$ of the parameters of ResNet8. The key information about the behaviour of a neural net seems to live in a low dimensional space. Indeed, as shown by the visualization of PReps provided in Appendix B.6, the predicted instance is very different from the target one: NetSpace captures the essential information to reproduce the behaviour of the target network, it does not merely learn to reproduce it.

After training, NetSpace has learnt to map target instances to fixed-sized embeddings. Thus, we can use NetSpace frozen encoder to obtain the embeddings of two anchor instances and linearly interpolate between them in order to study the representations laying in the space between the two anchors. To this aim, we decode every interpolated embedding to generate a new instance with NetSpace frozen decoder and compute the accuracy of this ready-to-use instances on the images in the test sets. As a baseline, we consider the possibility of interpolating directly the weights of the anchor instances. Results are reported in Fig. 4.4b and Fig. 4.4d for TIN and CIFAR-10, respectively. Interestingly, along the multi-dimensional

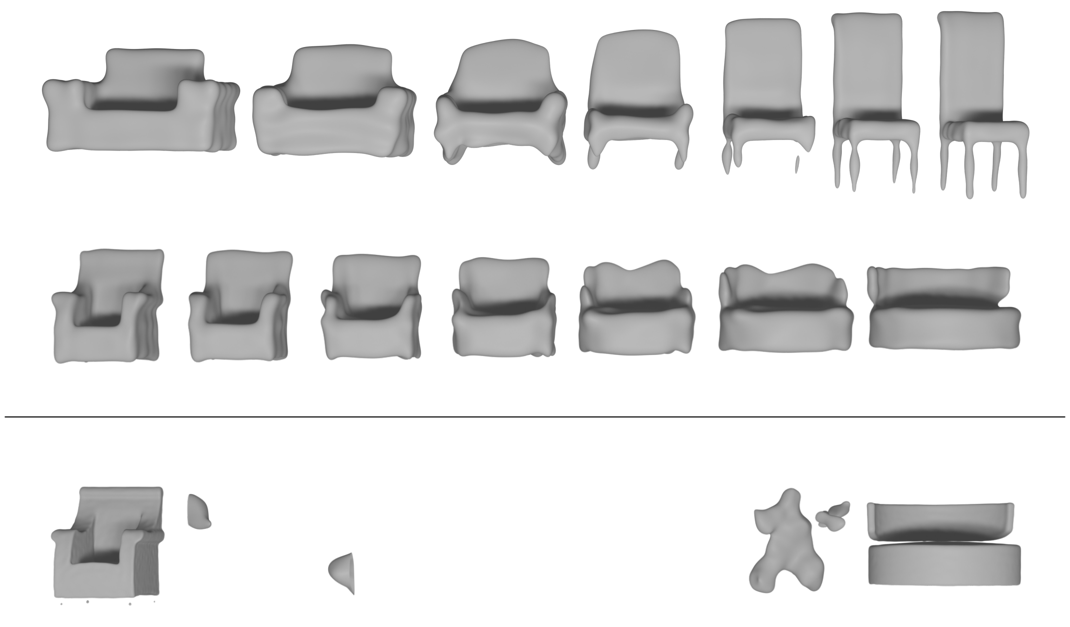


FIGURE 4.5. **Interpolation of 3D shapes.** Top two rows: results obtained by interpolating NetSpace embedding space. Bottom row: the same linear interpolation applied to MLPs weights.

line that connects the anchor embeddings we find representations corresponding to instances whose performances grow almost linearly with the interpolation factor, while interpolating directly the weights of the anchors yields unpredictable performances everywhere. This result suggests that the embedding space learnt by our framework can be organized to have meaningful dimensions, that are not exhibited in the instances weights space. In fact, the loss function used in the Single-Architecture training concerns performance and our framework learns *naturally* a latent space that, at least locally, can be explored along a direction strictly correlated with performance.

4.4.3 Single-Architecture SDF Regression

As a second Single-Architecture experiment, we train NetSpace to learn a latent space of MLPs that represent implicitly the SDF of chairs from the ShapeNet dataset. We train our framework on a dataset of $\sim$ 1000 MLPs: each of them has been trained to overfit a different 3D shape, starting from a different random initialization. The goal of this experiment is to assess if NetSpace is capable of learning a meaningful embedding of 3D shapes, which can then be explored by linear interpolation. Thus, after training NetSpace, we obtain two anchor embeddings by processing two input MLPs with NetSpace frozen encoder. Then, we obtain new embeddings by interpolating the anchors and we predict new MLPs with NetSpace frozen decoder. The results of this experiment are reported in Fig. 4.5. The top two rows show interpolation results obtained from NetSpace latent space, while the bottom row presents results obtained by interpolating directly the weights of the anchor MLPs. We can notice that

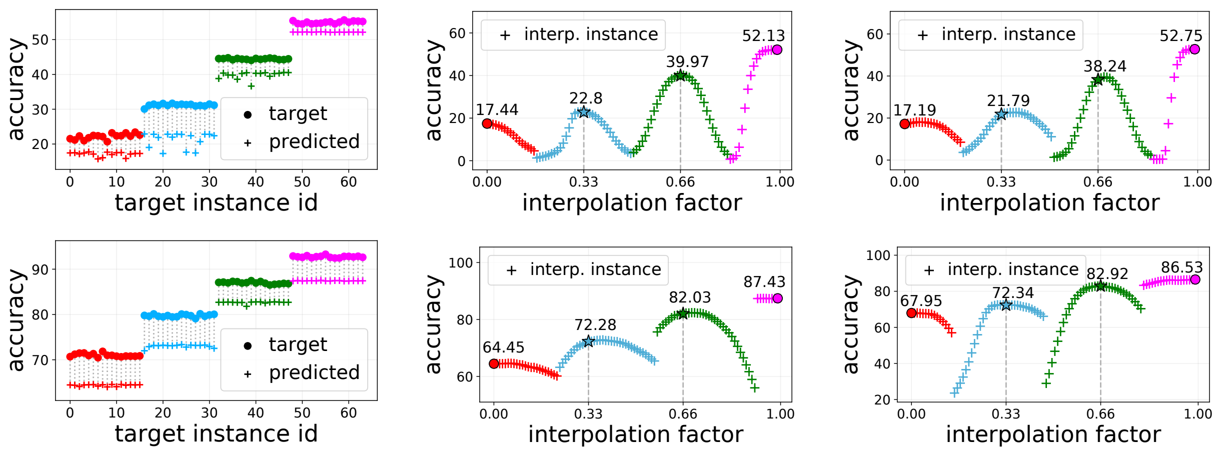


FIGURE 4.6. **Results of the Multi-Architecture setting on TIN (top) and CIFAR10 (bottom).** Left: target instances from test set, instances are sorted *w.r.t.* their ClassId. Center: interpolation with all architectures available at training time. Right: interpolation with only variants of LeNetLike and ResNet32 seen at training time. In all figures, color represent architecture: red-LeNetLike, blue-VanillaCNN, green-ResNet8, fuchsia-ResNet32. Circles correspond to interpolation boundaries. Stars denote instances obtained with the discrete interpolation factors used in $\mathcal{L}^{\gamma}$.

direct interpolation in the MLPs weights space yields catastrophic failures, while NetSpace embedding space enables smooth interpolations between the boundary shapes. This shows its ability to distill the core content of a trained model into a small-size embedding abstracting from the specific values of weights, and also its flexibility: when the loss concerns fitting of shapes, the latent space of models *naturally* organizes to have dimensions correlated with the shapes traits.

4.4.4 Multi-Architecture

In the Multi-Architecture setting we train NetSpace to embed four architectures: LeNetLike, VanillaCNN, ResNet8 and ResNet32. To build the dataset, we train many randomly initialized instances for each architecture, collecting multiple instances with good performances (100 for training, 16 for validation and 16 for testing). We collect in total 400 instances for training, 64 for validation and 64 for testing. We adopt a ResNet56 with high performance as the teacher network in Eq. (4.3) and Eq. (4.11) and set α to 0.9 in Eq. (4.5). We observe that supervision is not the same for different architectures: instances with lower performances receive a stronger signal from the $\mathcal{L}_{kd}$ and $\mathcal{L}^{\gamma}$ provides additional supervision for non-boundary architectures. We alleviate this issue modifing the training set so as to include a different number of instances for each architecture: we include 60 LeNetLike, 50 VanillaCNN, 60 ResNet8 and 100 ResNet32 instances. Fig. 4.6 left shows the results of this experiment: NetSpace successfully embeds instances of multiple architectures in highly compressed representations with all the information needed to a) predict correctly the architecture of the target instance and b) reconstruct an instance of such architecture whose behavior mimics that of the target one.

4.4.5 Multi-Architecture Embedding Interpolation

As we defined the ClassIds of architectures according to their increasing number of parameters, we expect their latent representations to be sorted *w.r.t.* this characteristic thanks to our interpolation loss $\mathcal{L}^{\gamma}$. In this experiment, we explore the latent space by observing the classification accuracy achieved by instances obtained when interpolating one embedding of LeNetLike and one of ResNet32, while moving with smaller steps than those defined in Eq. (4.12). Notably, as shown in Fig. 4.6 center, NetSpace learns an embedding space where architectures vary according to their number of parameters along an hyper-line. Moreover, it organized the space to place best performing embeddings for every class around the positions on which $\mathcal{L}^{\gamma}$ was computed.

4.4.6 Sampling of Unseen Architectures

We perform a new Multi-Architecture experiment by using a training set composed only of LeNetLike and ResNet32 instances. We collect 40 instances for LeNetLike and 80 for ResNet32, with the same balancing strategy discussed above. By not showing to NetSpace encoder any instance of VanillaCNN and ResNet8, we deny it the possibility to learn directly a portion of the embedding space dedicated to them. However, $\mathcal{L}^{\gamma}$ shapes it indirectly: for instance, given two embeddings e_{lenet} and e_{r32} of, respectively, a LeNetLike and ResNet32, it forces the embedding $e^{\gamma} = 0.33 \cdot e_{lenet} + 0.66 \cdot e_{r32}$ to represent an instance of ResNet8 (unseen during training). After training, we perform an interpolation experiment and report results in Fig. 4.6 right: we find that the latent space learnt by NetSpace trained on a reduced set of architectures exhibits the same properties as the space learnt by training with all of them, allowing to draw by interpolation instances with good performance of the *unseen architectures* VanillaCNN and ResNet8.

4.4.7 Latent Space Optimization

Here we consider to obtain NetSpace embeddings by latent space optimization (LSO). In order to do so, we take NetSpace encoder and decoder obtained by a Single-Architecture Image classification experiment, freezing their parameters. Then, we obtain an initial latent code by embedding one ResNet8 instance from the test set with the frozen encoder. We then use the frozen decoder to perform an iterative optimization of the initial embedding. In each step of the optimization, the embedding is processed by the frozen decoder, which predicts a PRep that is loaded into a ResNet8 instance. The resulting network is then used to produce

	Single-Architecture	Multi-Architecture			
	ResNet8	LeNetLike	VanillaCNN	ResNet8	ResNet32
initial	25.73%	17.44%	22.89%	38.81%	52.17%
optimized	35.72%	18.55%	24.17%	42.07%	53.13%

TABLE 4.1. **Accuracy on TIN test set achieved with LSO.** The accuracy obtained by a given neural network can be increased by exploring NetSpace latent space with gradient descent.

predictions on a batch of training images from TIN. We apply $\mathcal{L}_{kd}$ on these predictions using a ResNet56 as teacher network, to guide NetSpace in the search of a high performing instance in the learnt latent space. As the decoder is frozen, we compute the gradient of the loss *w.r.t.* the embedding, so as to explore the latent space by gradient descent. Furthermore, we perform a similar experiment starting from NetSpace encoder and decoder taken from a Multi-Architecture training experiment. Also in this case, we use the frozen encoder to obtain initial embeddings from four instances belonging to different architectures. Then, we process each embeddings with the frozen decoder, which predicts a PRep and a ClassId. We use the predicted ClassId to select the architecture where the predicted PRep is loaded, building an instance which we use to produce predictions on a batch of training images from TIN. In addition to $\mathcal{L}_{kd}$, in this case we apply also $\mathcal{L}_{class}$ on the predicted ClassId, to guide the embedding towards the area of the latent space that corresponds to the desired class. In Tab. 4.1 we report the performances obtained by the optimizations (second row), together with the performances of the instances used to obtain the initial embeddings (first row). Remarkably, the results show that it is actually possible to improve the performances of the input instances by exploring the NetSpace embedding space via latent space optimization.

4.5 Conclusion and Future Work

NetSpace introduces a framework to learn the latent space of deep models. We have shown that the embedding space learnt by NetSpace can be organized according to meaningful traits and ready-to-use instances with predictable properties can be obtained by means of linear interpolation or latent space optimization. Furthermore, our experiments provide evidence that fixed-size embeddings can represent effectively instances of several different architectures.

We believe that these findings are non-obvious and definitely worth communicating to the community, as they may open up new research directions and foster the identification of new potential applications. As a matter of fact, in the next chapter we build on top of such findings and present a framework where implicit neural representations of 3D shapes are squeezed into compact embeddings, which are then processed to perform several tasks.

Chapter 5

Deep Learning on Implicit Neural Representations of Shapes

5.1 Introduction

Since the early days of computer vision, researchers have been processing images stored as two-dimensional grids of pixels carrying intensity or color measurements. But the world that surrounds us is three dimensional, motivating researchers to try to process also 3D data sensed from surfaces. Unfortunately, representation of 3D surfaces in computers does not enjoy the same uniformity as digital images, with a variety of discrete representations, such as voxel grids, point clouds and meshes, coexisting today. Besides, when it comes to processing by deep neural networks, all these kinds of representations are affected by peculiar shortcomings, requiring complex ad-hoc machinery [162, 122, 163] and/or large memory resources [164]. Hence, no standard way to store and process 3D surfaces has yet emerged.

Recently, a new kind of representation has been proposed, which leverages on the possibility of deploying a Multi-Layer Perceptron (MLP) to fit a continuous function that represents *implicitly* a signal of interest [20]. These representations, usually referred to as Implicit Neural Representations (INRs), have been proven capable of encoding effectively 3D shapes by fitting *signed distance functions (sdf)* [25, 28, 33], *unsigned distance functions (udf)* [26] and *occupancy fields (occ)* [27, 43]. Encoding a 3D shape with a continuous function parameterized as an MLP decouples the memory cost of the representation from the actual spatial resolution, *i.e.*, a surface with arbitrarily fine resolution can be reconstructed from a fixed number of parameters. Moreover, the same neural network architecture can be used to fit different implicit functions, holding the potential to provide a unified framework for 3D shapes.

Due to their effectiveness and potential advantages over traditional representations, INRs are gathering ever-increasing attention from the scientific community, with novel and striking results published more and more frequently [32, 29, 28, 31]. This lead us to conjecture that, in the forthcoming future, INRs might emerge as a standard representation to store and communicate 3D shapes, with repositories hosting digital twins of 3D objects realized only as MLPs becoming commonly available.

An intriguing research question does arise from the above scenario: beyond storage and communication, would it be possible to *process* directly INRs of 3D shapes with deep learning pipelines to solve downstream tasks as it is routinely done today with discrete representations like point clouds or meshes? In other words, would it be possible to process an INR of a 3D shape to solve a downstream task, *e.g.*, shape classification, without reconstructing a discrete representation of the surface?

Since INRs are neural networks, there is no straightforward way to process them. Earlier work in the field, namely OccupancyNetworks [27] and DeepSDF [25], fit the whole dataset with a shared network conditioned on a different embedding for each shape. In such formulation, the natural solution to the above mentioned research problem could be to use such embeddings as representations of the shapes in downstream tasks. This is indeed the approach followed by contemporary work [165], which addresses such research problem by using as embedding a latent modulation vector applied to a shared base network. However, representing a whole dataset by a shared network sets forth a difficult learning task, with the network struggling in fitting accurately the totality of the samples (as we show in Sec. 5.6.1). Conversely, several recent works, like SIREN [22] and others [34, 166, 167, 168, 30] have shown that, by fitting an individual network to each input sample, one can get high quality reconstructions even when dealing with very complex 3D shapes or images. Moreover, constructing an individual INR for each shape is easier to deploy in the wild, as availability of the whole dataset is not required to fit an individual shape. Such works are gaining ever-increasing popularity and we are led to believe that fitting an individual network is likely to become the common practice in learning INRs.

Thus, in this chapter, we investigate how to perform downstream tasks with deep learning pipelines on shapes represented as individual INRs. However, a single INR can easily count hundreds of thousands of parameters, though it is well known that the weights of a deep model provide a vastly redundant parametrization of the underlying function [141, 140]. Hence, we settle on investigating whether and how an answer to the above research question may be provided by a representation learning framework that learns to squeeze individual INRs into compact and meaningful embeddings amenable to pursuing a variety of downstream tasks.

Our framework, dubbed inr2vec and shown in Fig. 5.1, has at its core an encoder designed to produce a task-agnostic embedding representing the input INR by processing only the INR weights. These embeddings can be seamlessly used in downstream deep learning pipelines, as we validate experimentally for a variety of tasks, like classification, retrieval, part segmentation, unconditioned generation, surface reconstruction and completion. Interestingly, since embeddings obtained from INRs live in low-dimensional vector spaces regardless of the underlying implicit function, the last two tasks can be solved by learning a simple mapping between the embeddings produced with our framework, *e.g.*, by transforming the INR of a

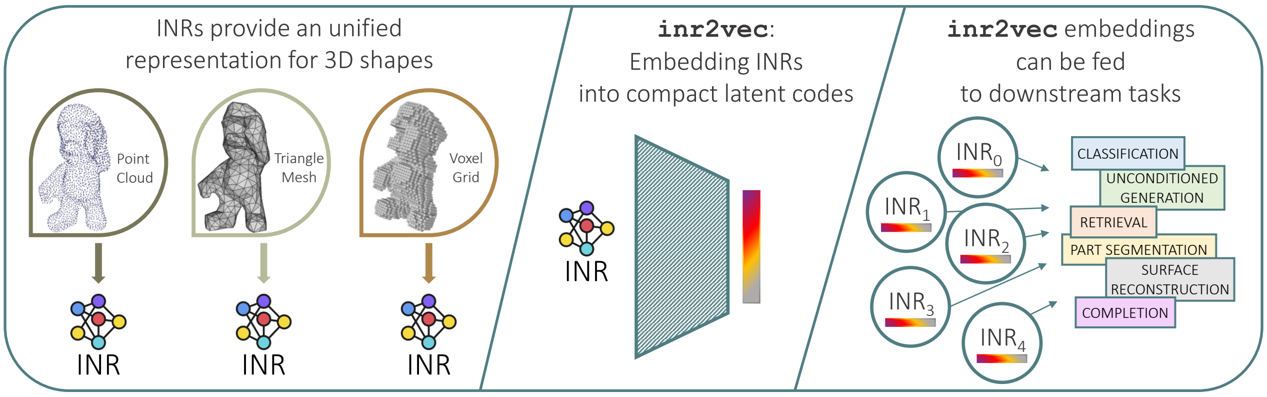


FIGURE 5.1. **Overview of our framework. Left**: INRs hold the potential to provide an unified representation for 3D shapes. **Center**: Our framework, dubbed inr2vec, produces a compact representation for an input INR by looking only at its weights. **Right**: inr2vec embeddings can be used with standard deep learning machinery to solve a variety of downstream tasks.

udf into the INR of an sdf. Moreover, inr2vec can learn a smooth latent space conducive to interpolating INRs representing unseen 3D objects.

Our contributions can be summarised as follows:

- we propose and investigate the novel research problem of applying deep learning directly on individual INRs representing 3D shapes;
- to address the above problem, we introduce inr2vec, a framework that can be used to obtain a meaningful compact representation of an input INR by processing only its weights, without sampling the underlying implicit function;
- we show that a variety of tasks, usually addressed with representation-specific and complex frameworks, can indeed be performed by deploying simple deep learning machinery on INRs embedded by inr2vec, the same machinery regardless of the INRs underlying signal.

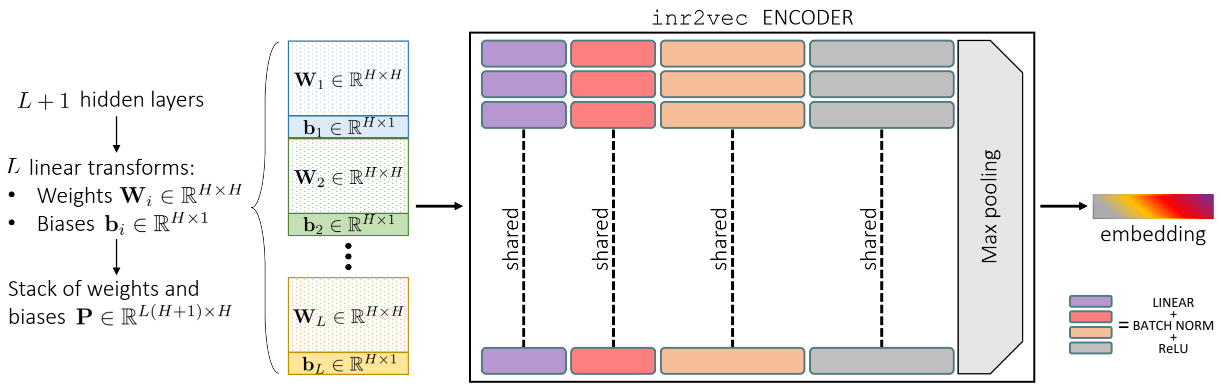
5.2 Related Work

Deep learning on 3D shapes. Due to their regular structure, voxel grids have always been appealing representations for 3D shapes and several works proposed to use 3D convolutions to perform both discriminative [164, 169, 170] and generative [171, 172, 173, 174, 175, 176] tasks. The huge memory requirements of voxel-based representations, though, led researchers to look for less demanding alternatives, such as point clouds. Processing point clouds, however, is far from straightforward because of their unorganized nature. As a possible solution, some works projected the original point clouds to intermediate regular grid structures such as voxels [177] or images [178, 179]. Alternatively, PointNet [16] proposed to operate directly on raw coordinates by means of shared multi-layer perceptrons followed by max pooling to aggregate point features. PointNet++ [162] extended PointNet with a hierarchical feature learning paradigm to capture the local geometric structures. Following PointNet++, many works focused on designing new local aggregation operators [180], resulting in a wide spectrum of specialized methods based on convolution [181, 182, 183, 184, 185, 186, 187], graph [188, 189, 122], and attention [190, 191] operators. Yet another completely unrelated set of deep learning methods have been developed to process surfaces represented as meshes, which differ in the way they exploit vertices, edges and faces as input data [163]. *Vertex-based* approaches leverage the availability of a regular domain to encode the knowledge about points neighborhoods through convolution or kernel functions [192, 193, 194, 195, 196, 197, 198, 199]. *Edge-based* methods take advantages of these connections to define an ordering invariant convolution [200], to construct a graph on the input meshes [201] or to navigate the shape structure [202]. Finally, *Face-based* works extract information from neighboring faces [203, 204, 205, 206]. In this work, we explore INRs as a unified representation for 3D shapes and propose a framework that enables the use of the same standard deep learning machinery to process them, independently of the INRs underlying signal.

Deep learning on neural networks. Few works attempted to process neural networks by means of other neural networks. For instance, [207] takes as input the weights of a network and predicts its classification accuracy. [208] learns a network representation with a self-supervised learning strategy applied on the N-dimensional weight array, and then uses the learned representations to predict various characteristics of the input classifier. [209, 210, 155] represent neural networks as computational graphs, which are processed by a GNN to predict optimal parameters, adversarial examples, or branching strategies for neural network verification. All these works see neural networks as algorithms and focus on predicting properties such as their accuracy. On the contrary, we process networks that represent implicitly 3D shapes to perform a variety of tasks directly from their weights, *i.e.*, we treat neural networks as input/output data. To the best of our knowledge, processing 3D shapes represented as INRs has been attempted only in contemporary work [165]. However, they rely on a shared network conditioned on shape-specific embeddings while we process the weights of individual INRs, that better capture the underlying signal and are easier to deploy in the wild.

5.3 Learning to Represent INRs

The research question we address in this chapter is whether and how can we process directly INRs to perform downstream tasks. For instance, can we classify a 3D shape that is implicitly encoded in an INR? And how? As anticipated in Sec. 5.1, we propose to rely on a representation learning framework to squeeze the redundant information contained in



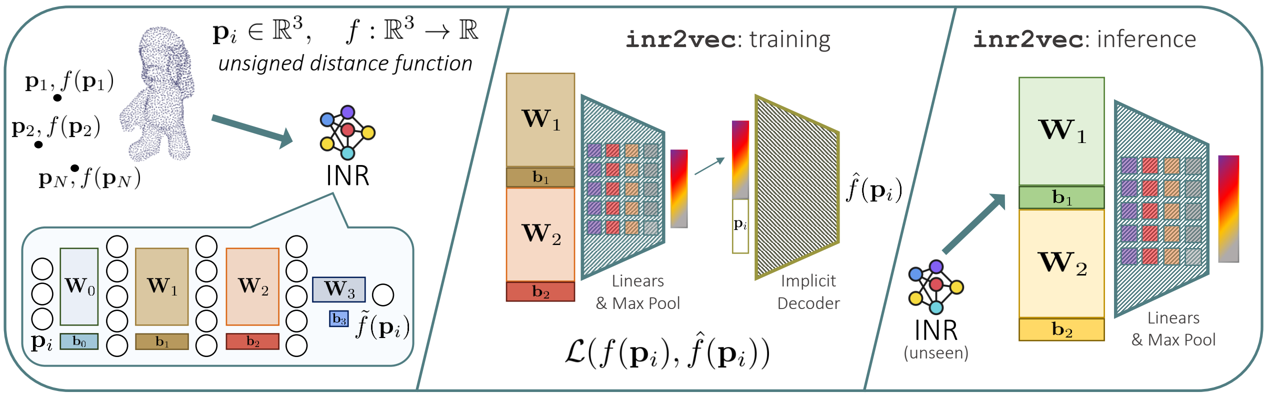
$L+1$ hidden layers
L linear transforms:
• Weights $\mathbf{W}_i \in \mathbb{R}^{H\times H}$
• Biases $\mathbf{b}_i \in \mathbb{R}^{H\times 1}$
Stack of weights and biases $\mathbf{P} \in \mathbb{R}^{L(H+1)\times H}$
$\mathbf{W}_1 \in \mathbb{R}^{H\times H}$
$\mathbf{b}_1 \in \mathbb{R}^{H\times 1}$
$\mathbf{W}_2 \in \mathbb{R}^{H\times H}$
$\mathbf{b}_2 \in \mathbb{R}^{H\times 1}$
$\mathbf{W}_L \in \mathbb{R}^{H\times H}$
$\mathbf{b}_L \in \mathbb{R}^{H\times 1}$
inr2vec ENCODER
shared
shared
shared
shared
Max pooling
embedding
= LINEAR + BATCH NORM + ReLU

FIGURE 5.2. **inr2vec encoder architecture**.

the weights of INRs into compact latent codes that could be conveniently processed with standard deep learning pipelines.

Our framework, dubbed inr2vec, is composed of an encoder and a decoder. The encoder, detailed in Fig. 5.2, is designed to take as input the weights of an INR and produce a compact embedding that encodes all the relevant information of the input INR. A first challenge in designing an encoder for INRs consists in defining how the encoder should ingest the weights as input, since processing naively all the weights would require a huge amount of memory (see Appendix C.4). Following standard practice [22, 34, 166, 167, 168], we consider INRs composed of several hidden layers, each one with H nodes, *i.e.*, the linear transformation between two consecutive layers is parameterized by a matrix of weights $\mathbf{W}_i \in \mathbb{R}^{H\times H}$ and a vector of biases $\mathbf{b}_i \in \mathbb{R}^{H\times 1}$. Thus, stacking $\mathbf{W}_i$ and $\mathbf{b}_i{}^T$, the mapping between two consecutive layers can be represented by a single matrix $\mathbf{P}_i \in \mathbb{R}^{(H+1)\times H}$. For an INR composed of $L+1$ hidden layers, we consider the L linear transformations between them. Hence, stacking all the L matrices $\mathbf{P}_i \in \mathbb{R}^{(H+1)\times H}, i = 1, \dots, L$, between the hidden layers we obtain a single matrix $\mathbf{P} \in \mathbb{R}^{L(H+1)\times H}$, that we use to represent the INR in input to inr2vec encoder. We discard the input and output layers in our formulation as they feature different dimensionality and their use does not change inr2vec performance, as shown in Appendix C.8.

The inr2vec encoder is designed with a simple architecture, consisting of a series of linear layers with batch norm and ReLU non-linearity followed by final max pooling. At each stage, the input matrix is transformed by one linear layer, that applies the same weights to each row of the matrix. The final max pooling compresses all the rows into a single one, obtaining the desired embedding. It is worth observing that the randomness involved in fitting an individual INR (weights initialization, data shuffling, etc.) causes the weights in the same position in the INR architecture not to share the same role across INRs. Thus, inr2vec encoder would have to deal with input vectors whose elements capture different information across the different data samples, making it impossible to train the framework. However, the use of a shared, pre-computed initialization has been advocated as a good practice when fitting INRs, *e.g.*, to reduce training time by means of meta-learned initialization vectors, as done in



$\mathbf{p}_i \in \mathbb{R}^3, \quad f:\mathbb{R}^3 \rightarrow \mathbb{R}$
unsigned distance function
$\mathbf{p}_1, f(\mathbf{p}_1)$
$\mathbf{p}_2, f(\mathbf{p}_2)$
$\mathbf{p}_N, f(\mathbf{p}_N)$
INR
$\mathbf{W}_0$ $\mathbf{W}_1$ $\mathbf{W}_2$ $\mathbf{W}_3$
$\mathbf{b}_0$ $\mathbf{b}_1$ $\mathbf{b}_2$ $\mathbf{b}_3$
$\mathbf{p}_i$
$\tilde{f}(\mathbf{p}_i)$
inr2vec: training
Linears & Max Pool
Implicit Decoder
$\hat{f}(\mathbf{p}_i)$
$\mathcal{L}(f(\mathbf{p}_i), \hat{f}(\mathbf{p}_i))$
inr2vec: inference
INR (unseen)

FIGURE 5.3. **Training and inference of our framework. Left:** We consider shapes represented as INRs. As an example, we show an INR fitting the *udf* of a surface. **Center:** inr2vec encoder is trained together with an implicit decoder to replicate the underlying 3D signal of the input INR. **Right:** At inference time, the learned encoder can be used to obtain a compact embedding from unseen INRs.

MetaSDF [34] and in the contemporary work exploring processing of INRs [165], or to obtain desirable geometric properties [33]. We empirically found that following such a practice, *i.e.*, initializing all INRs with the same random vector, favours alignment of weights across INRs and enables convergence of our framework (see also Sec. 5.4).

In order to guide the inr2vec encoder to produce meaningful embeddings, we first note that we are not interested in encoding the values of the input weights in the embeddings produced by our framework, but, rather, in storing information about the 3D shape represented by the input INR. For this reason, we supervise the decoder to replicate the function approximated by the input INR instead of directly reproducing its weights, as it would be the case in a standard auto-encoder formulation. In particular, during training, we adopt an implicit decoder inspired by [25], which takes in input the embeddings produced by the encoder and decodes the input INRs from them (see Fig. 5.3 center). More specifically, when the inr2vec encoder processes a given INR, we use the underlying signal to create a set of 3D queries $\mathbf{p}_i$, paired with the values $f(\mathbf{p}_i)$ of the function approximated by the input INR at those locations (the type of function depends on the underlying signal modality, it can be *udf* in case of point clouds, *sdf* in case of triangle meshes or *occ* in case of voxel grids). The decoder takes in input the embedding produced by the encoder concatenated with the 3D coordinates of a query $\mathbf{p}_i$ and the whole encoder-decoder is supervised to regress the value $f(\mathbf{p}_i)$. After the overall framework has been trained end to end, the frozen encoder can be used to compute embeddings of unseen INRs with a single forward pass (see Fig. 5.3 right) while the implicit decoder can be used, if needed, to reconstruct the discrete representation given an embedding.

In Fig. 5.4 we compare 3D shapes reconstructed from INRs unseen during training with those reconstructed by the inr2vec decoder starting from the latent codes yielded by the

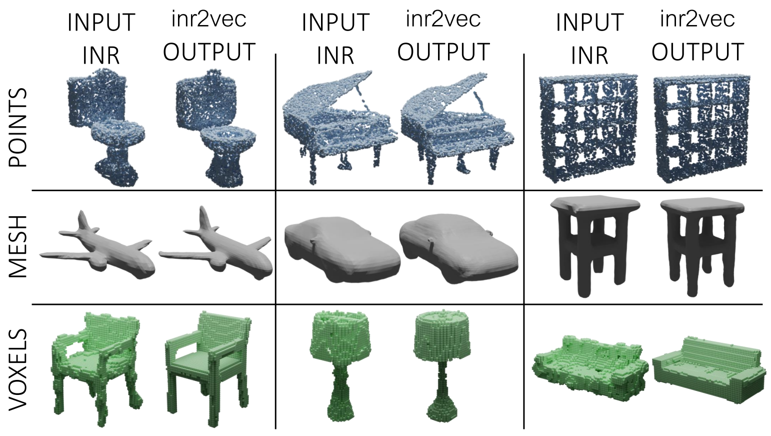


FIGURE 5.4. **inr2vec reconstructions.** Comparison between discrete shapes reconstructed from the INRs presented in input to inr2vec ("INPUT INR") and the ones reconstructed from inr2vec embeddings ("inr2vec OUTPUT").

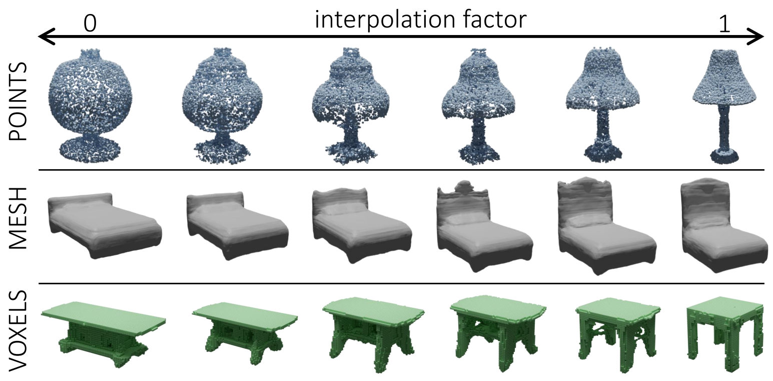


FIGURE 5.5. **inr2vec latent space interpolation.** Given two inr2vec embeddings obtained from two input INRs, it is possible to linearly interpolate between them, producing new embeddings that represent unseen INRs of plausible shapes.

encoder. We visualize point clouds with 8192 points, meshes reconstructed by marching cubes [211] from a grid with resolution 128^3 and voxels with resolution 64^3. We note that, though our embedding is dramatically more compact than the original INR, the reconstructed shape resembles the ground-truth with a good level of detail. Moreover, in Fig. 5.5 we linearly interpolate between the embeddings produced by inr2vec from two input shapes and show the shapes reconstructed from the interpolated embeddings. Results highlight that the latent space learned by inr2vec enables smooth interpolations between shapes represented as INRs. Additional details on inr2vec training and the procedure to reconstruct the discrete representations from the decoder are in the Appendices.

5.4 Using the Same Initialization for INRs

The need to align the multitude of INRs that can approximate a given shape is a challenging research problem that has to be dealt with when using INRs as input data. We empirically

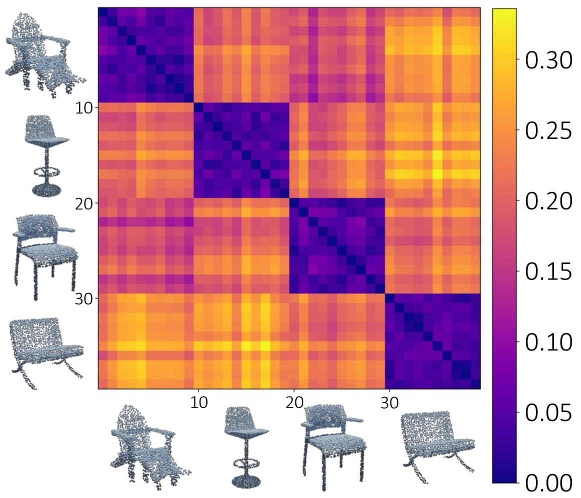


FIGURE 5.6. **L2 distances between inr2vec embeddings.** For each shape, we fit 10 INRs starting from the same weights initialization (40 INRs in total). Then we plot the L2 distances between the embeddings obtained by inr2vec for such INRs.

found that fixing the weights initialization to a shared random vector across INRs is a viable and simple solution to this problem.

We report here an experiment to assess if order of data, or other sources of randomness arising while fitting INRs, do affect the repeatability of the embeddings computed by inr2vec. We fitted 10 INRs on the same discrete shape for 4 different chairs, *i.e.*, 40 INRs in total. Then, we embed all of them with the pretrained inr2vec encoder and compute the L2 distance between all pairs of embeddings. The block structure of the resulting distance matrix (see Fig. 5.6) highlights how, under the assumption of shared initialization and hyperparameters, inr2vec is repeatable across multiple fittings.

Seeking for a proof with a stronger theoretical foundation, we turn our attention to the recent work *git re-basin* [212], where authors show that the loss landscape of neural networks contain (nearly) a single basin after accounting for all possible permutation symmetries of hidden units. The intuition behind this finding is that, given two neural networks that were trained with equivalent architectures but different random initializations, data orders, and potentially different hyperparameters or datasets, it is possible to find a permutation of the networks weights such that when linearly interpolating between their weights, all intermediate models enjoy performance similar to them – a phenomenon denoted as *linear mode connectivity*.

Intrigued by this finding, we conducted a study to assess whether initializing INRs with the same random vector, which we found to be key to inr2vec convergence, also leads to linear mode connectivity. Thus, given one shape, we fitted it with two different INRs and then we interpolated linearly their weights, observing at each interpolation step the loss value obtained by the *interpolated* INR on the same batch of points. For each shape, we repeated the experiment twice, once initializing the INRs with different random vectors and once

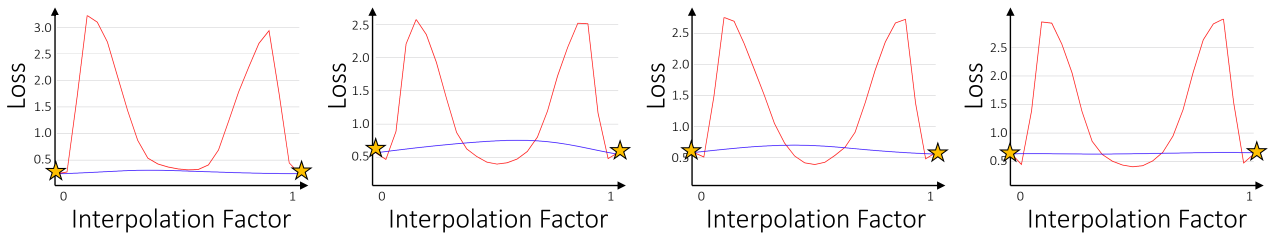


FIGURE 5.7. **Linear mode connectivity study.** Each plot shows variation of the loss function over the same batch of points when interpolating between two INRs representing the same shape. The red line describes the interpolation between INRs initialized differently, while the blue line shows the same interpolation between INRs initialized from the same random vector. The yellow stars represent the loss value of the boundary INRs.

initializing them with the same random vector.

The results of this experiment are reported for four different shapes in Fig. 5.7. It is possible to note that, as shown by the blue curves, when interpolating between INRs obtained from the same weights initialization, the loss value at each interpolation step is nearly identical to those of the boundary INRs. On the contrary, the red curves highlight how there is no linear mode connectivity at all between INRs obtained from different weights initializations.

[212] proposes also different algorithms to estimate the permutation needed to obtain linear mode connectivity between two networks. We applied the algorithm proposed in their paper in Section 3.2 (*Matching Weights*) to our INRs and observed the resulting permutations. Remarkably, when applied to INRs obtained from the same weights initialization, the retrieved permutations are identity matrices, both when the target INRs represent the same shape and when they represent different ones. The permutations obtained for INRs obtained from different initializations, instead, are far from being identity matrices.

All these results favor the hypothesis that our technique of initializing INRs with the same random vector leads to linear mode connectivity between different INRs. We believe that the possibility of performing meaningful linear interpolation between the weights occupying the same positions across different INRs can be interpreted by considering corresponding weights as carrying out the same role in terms of feature detection units, explaining why the inr2vec encoder succeeds in processing the weights of our INRs.

This intuition can be also combined with the finding of another recent work [213], that shows how the expressive power of SIRENs is restricted to functions that can be expressed as a linear combination of certain harmonics of the first layer, which thus serves as basis for the space of learnable functions.

As stated above, the evidence of linear mode connectivity between INRs obtained from the same initialization leads us to believe that the weights of the first layer extract the same features across different INRs. Thus, we can think of using the same random initialization as a way to obtain the same basis of harmonics for all our INRs. We argue that this explains why it is possible to remove the first layer of the INRs presented in input to inr2vec (as empirically proved in Appendix C.8): if the basis is always the same, it is sufficient to process the layers from the second onwards, that represent the coefficients of the basis harmonics combination, as described in [213].

5.5 Deep Learning on INRs

In this section, we first present the set-up of our experiments. Then, we show how several tasks dealing with 3D shapes can be tackled by working only with inr2vec embeddings as input and/or output. Additional details on the architectures and on the experimental settings are in Appendix C.5.

5.5.1 General Settings

In all the reported experiments, we convert 3D discrete representations into INRs featuring 4 hidden layers with 512 nodes each, using the SIREN activation function [22]. We train inr2vec using an encoder composed of four linear layers with respectively 512, 512, 1024 and 1024 features, embeddings with 1024 values and an implicit decoder with 5 hidden layers with 512 features. The baselines are trained using standard data augmentation (random scaling and point-wise jittering), while we train both inr2vec and the downstream task-specific networks on datasets augmented offline with the same transformations.

5.5.2 Point Cloud Retrieval

We first evaluate the feasibility of using inr2vec embeddings of INRs to solve tasks usually tackled by representation learning, and we select 3D retrieval as a benchmark. We follow the procedure introduced in [161], using the euclidean distance to measure the similarity between embeddings of unseen point clouds from the test sets of ModelNet40 [176] and ShapeNet10 (a subset of 10 classes of the popular ShapeNet dataset [161]). For each embedded shape, we select its k-nearest-neighbours and compute a Precision Score comparing the classes of the query and the retrieved shapes, reporting the mean Average Precision for different k (mAP@k). Beside inr2vec, we consider three baselines to embed point clouds, which are obtained by training the PointNet [16], PointNet++ [162] and DGCNN [122] encoders in combination with a fully connected decoder similar to that proposed in [214] to reconstruct the input cloud. Quantitative results, reported in Tab. 5.1, show that, while there is an average gap of 1.8 mAP with PointNet++, inr2vec is able to match, and in some cases even surpass, the performance of the other baselines. Moreover, it is possible to appreciate in Fig. 5.8 that the retrieved shapes not only belong to the same class as the query but present also the same coarse structure. This finding highlights how the pretext task used to learn inr2vec embeddings can summarise relevant shape information effectively.

	ModelNet40			ShapeNet10			ScanNet10		
Method	mAP@1	mAP@5	mAP@10	mAP@1	mAP@5	mAP@10	mAP@1	mAP@5	mAP@10
PointNet [16]	80.1	91.7	94.4	90.6	96.6	98.1	65.7	86.2	92.6
PointNet++ [162]	85.1	93.9	96.0	92.2	97.5	98.6	71.6	89.3	93.7
DGCNN [122]	83.2	92.7	95.1	91.0	96.7	98.2	66.1	88.0	93.1
inr2vec	81.7	92.6	95.1	90.6	96.7	98.1	65.2	87.5	94.0

TABLE 5.1. **Point cloud retrieval quantitative results.**

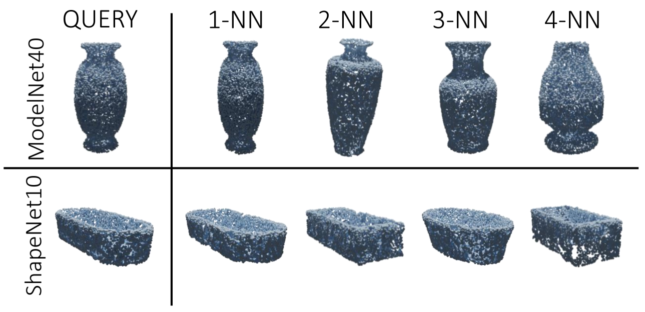


FIGURE 5.8. **Point cloud retrieval qualitative results.** Given the inr2vec embedding of a query shape, we show the shapes reconstructed from the closest embeddings (L2 distance).

5.5.3 Shape Classification

We then address the problem of classifying point clouds, meshes and voxel grids. For point clouds we use three datasets: ShapeNet10, ModelNet40 and ScanNet10 [137]. When dealing with meshes, we conduct our experiments on the Manifold40 dataset [163]. Finally, for voxel grids, we use again ShapeNet10, quantizing clouds to grids with resolution 64^3. Despite the different nature of the discrete representations taken into account, inr2vec allows us to perform shape classification on INRs embeddings, augmented online with E-Stitchup [215], by the very same downstream network architecture, *i.e.*, a simple fully connected classifier consisting of three layers with 1024, 512 and 128 features. We consider as baselines well-known architectures that are optimized to work on the specific input representations of each dataset. For point clouds, we consider PointNet [16], PointNet++ [162] and DGCNN [122]. For meshes, we consider a recent and competitive baseline that processes directly triangle meshes [202]. As for voxel grids, we train a 3D CNN classifier that we implemented following [164] (Conv3DNet from now on). Since only the train and test splits are released for all the datasets, we created validation splits from the training sets in order to follow a proper train/val protocol for both the baselines and inr2vec. As for the test shapes, we evaluated all the baselines on the discrete representations reconstructed from the INRs fitted on the original test sets, as these would be the only data available at test time in a scenario where INRs are used to store and communicate 3D data. We report the results of the baselines tested on the original discrete representations available in the original datasets in Appendix C.7: they are in line with those provided here. The results in Tab. 5.2 show that inr2vec embeddings deliver classification accuracy close to the specialized baselines across all the considered

	Point Cloud			Mesh	Voxels
Method	ModelNet40	ShapeNet10	ScanNet10	Manifold40	ShapeNet10
PointNet [16]	88.8	94.3	72.7	–	–
PointNet++ [162]	89.7	94.6	76.4	–	–
DGCNN [122]	89.9	94.3	76.2	–	–
MeshWalker [202]	–	–	–	90.0	–
Conv3DNet [164]	–	–	–	–	92.1
inr2vec	87.0	93.3	72.1	86.3	93.0

TABLE 5.2. **Results on shape classification across representations.**

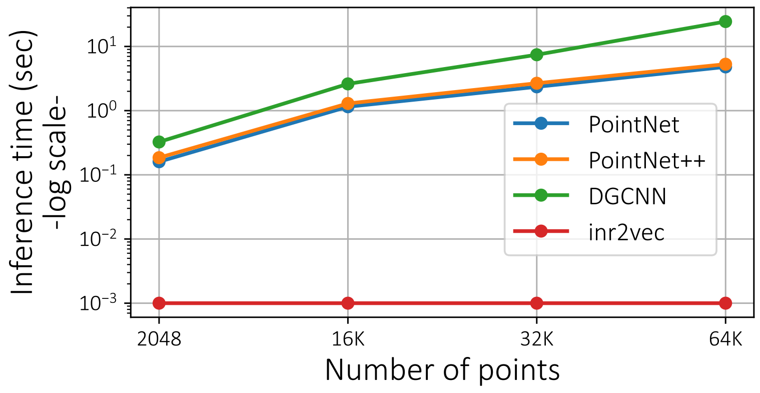
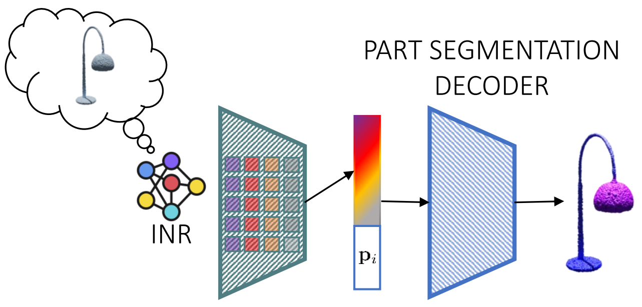


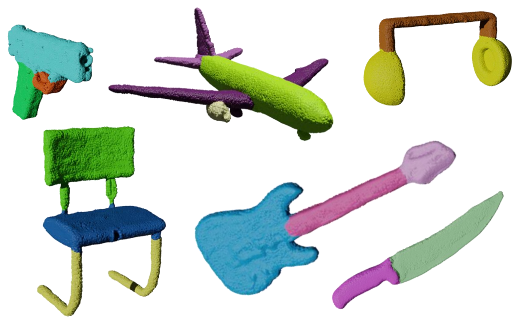
FIGURE 5.9. **Time required to classify INRs encoding udf.** For point cloud classifiers, the time to reconstruct the discrete cloud from the INR is included in the chart.

datasets, regardless of the original discrete representation of the shapes in each dataset. Remarkably, our framework allows us to apply the same simple classification architecture on all the considered input modalities, in stark contrast with all the baselines that are highly specialized for each modality, exploit inductive biases specific to each such modality and cannot be deployed on representations different from those they were designed for.

Furthermore, while presenting a gap of some accuracy points *w.r.t.* the most recent architectures, like DGCNN and MeshWalker, the simple fully connected classifier that we applied on inr2vec embeddings obtains scores comparable to standard baselines like PointNet and Conv3DNet. It is also worth highlighting that, should 3D shapes be stored as INRs, classifying them with the considered specialized baselines would require recovering the original discrete representations by the lengthy procedures described in Appendix C.2. Thus, in Fig. 5.9, we report the inference time of standard point cloud classification networks while including also the time needed to reconstruct the discrete point cloud from the input INR of the underlying *udf* at different resolutions. Even at the coarsest resolution (2048 points), all the baselines yield an inference time which is one order of magnitude higher than that required to classify directly the inr2vec embeddings. Increasing the resolution of the reconstructed clouds makes the inference time of the baselines prohibitive, while inr2vec, not requiring the explicit clouds, delivers a constant inference time of 0.001 seconds.



(A) Method.



(B) Qualitative results.

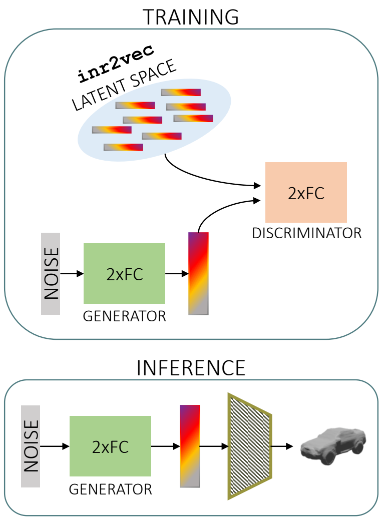
FIGURE 5.10. **Point cloud part segmentation.**

Method	instance mIoU	class mIoU	airplane	bag	cap	car	chair	earphone	guitar	knife	lamp	laptop	motor	mug	pistol	rocket	skateboard	table
PointNet [16]	83.1	78.96	81.3	76.9	79.6	71.4	89.4	67.0	91.2	80.5	80.0	95.1	66.3	91.3	80.6	57.8	73.6	81.5
PointNet++ [162]	84.9	82.73	82.2	88.8	84.0	76.0	90.4	80.6	91.8	84.9	84.4	94.9	72.2	94.7	81.3	61.1	74.1	82.3
DGCNN [122]	83.6	80.86	80.7	84.3	82.8	74.8	89.0	81.2	90.1	86.4	84.0	95.4	59.3	92.8	77.8	62.5	71.6	81.1
inr2vec	81.3	76.91	80.2	76.2	70.3	70.1	88.0	65.0	90.6	82.1	77.4	94.4	61.4	92.7	79.0	56.2	68.6	78.5

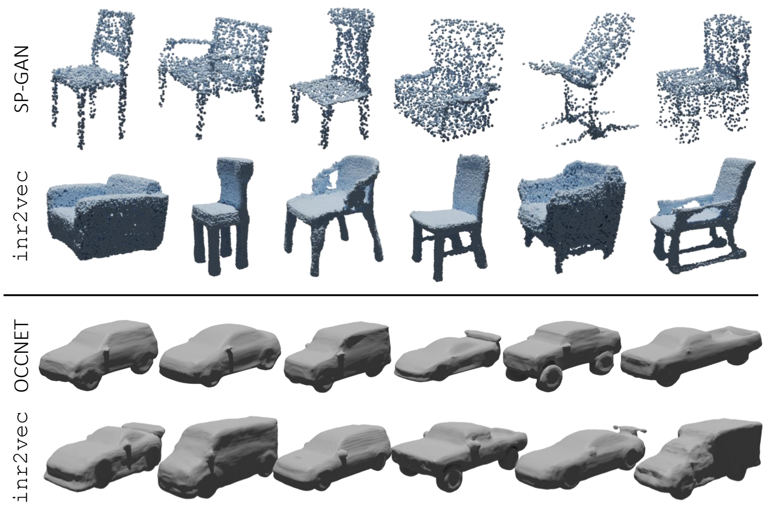
TABLE 5.3. **Part segmentation quantitative results.** We report the IoU for each class, the mean IoU over all the classes (class mIoU) and the mean IoU over all the instances (instance mIoU).

5.5.4 Point Cloud Part Segmentation

While the tasks of classification and retrieval concern the possibility of using inr2vec embeddings as a compact proxy for the global information of the input shapes, with the task of point cloud part segmentation we aim at investigating whether inr2vec embeddings can be used also to assess upon local properties of shapes. The part segmentation task consists in predicting a semantic (*i.e.*, part) label for each point of a given cloud. We tackle this problem by training a decoder similar to that used to train our framework (see Fig. 5.10a). Such decoder is fed with the inr2vec embedding of the INR representing the input cloud, concatenated with the coordinate of a 3D query, and it is trained to predict the label of the query point. We train it, as well as PointNet, PointNet++ and DGCNN, on the ShapeNet Part Segmentation dataset [216] with point clouds of 2048 points, with the same train/val/test of the classification task. Quantitative results reported in Tab. 5.3 show the possibility of performing also a local discriminative task as challenging as part segmentation based on the task-agnostic embeddings produced by inr2vec and, in so doing, to reach performance not far from that of specialized architectures. Additionally, in Fig. 5.10b we show point clouds reconstructed at 100K points from the input INRs and segmented with high precision thanks to our formulation based on a semantic decoder conditioned by the inr2vec embedding.



(A) Method.

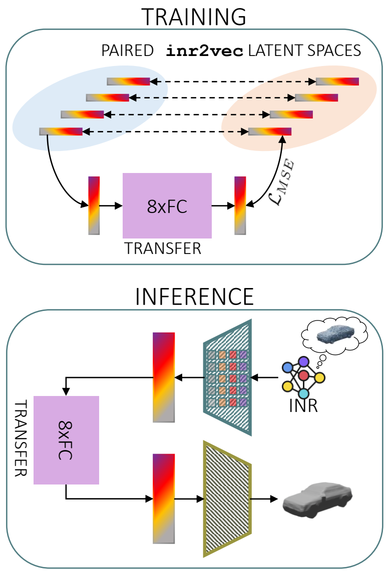


(B) Qualitative results.

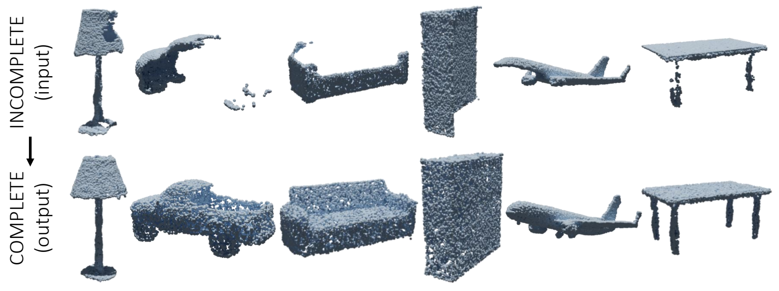
FIGURE 5.11. **Learning to generate shapes from inr2vec latent space.**

5.5.5 Shape Generation

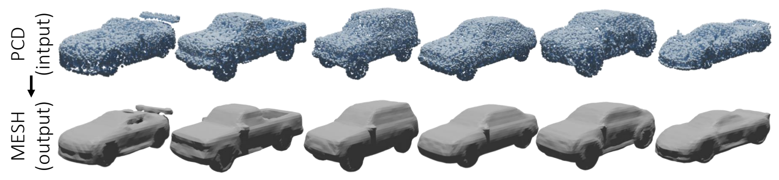
With the experiments reported above we validated that, thanks to inr2vec embeddings, INRs can be used as input in standard deep learning machinery. In this section, we address instead the task of shape generation in an adversarial setting to investigate whether the compact representations produced by our framework can be adopted also as medium for the output of deep learning pipelines. For this purpose, as depicted in Fig. 5.11a, we train a Latent-GAN [217] to generate embeddings indistinguishable from those produced by inr2vec starting from random noise. The generated embeddings can then be decoded into discrete representations with the implicit decoder exploited during inr2vec training. Since our framework is agnostic *w.r.t.* the original discrete representation of shapes used to fit INRs, we can train Latent-GANs with embeddings representing point clouds or meshes based on the same identical protocol and architecture (two simple fully connected networks as generator and discriminator). For point clouds, we train one Latent-GAN on each class of ShapeNet10, while we use models of cars provided by [27] when dealing with meshes. In Fig. 5.11b, we show some samples generated with the described procedure, comparing them with SP-GAN [218] on the *chair* class for what concerns point clouds and Occupancy Networks [27] (VAE formulation) for meshes. Generated examples of other classes for point clouds are shown in Sec. 5.6.5. The shapes generated with our Latent-GAN trained only on inr2vec embeddings seem comparable to those produced by the considered baselines, in terms of both diversity and richness of details. Additionally, by generating embeddings that represent implicit functions, our method enables sampling point clouds at any arbitrary resolution (*e.g.*, 8192 points in Fig. 5.11b) whilst SP-GAN would require a new training for each desired resolution since the number of generated points must be set at training time.



(A) Method.



(B) Point cloud completion.



(C) Surface reconstruction.

FIGURE 5.12. **Learning a mapping between inr2vec latent spaces.**

5.5.6 Learning a Mapping Between inr2vec Embedding Spaces

We showed that inr2vec embeddings can be used as a proxy to feed INRs in input to deep learning pipelines, and that they can also be obtained as output of generative frameworks. In this section we move a step further, considering the possibility of learning a mapping between two distinct latent spaces produced by our framework for two separate datasets of INRs, based on a *transfer* function designed to operate on inr2vec embeddings as both input and output data. Such transfer function can be realized by a simple fully connected network that maps the input embedding into the output one and is trained by a standard MSE loss (see Fig. 5.12a). As inr2vec generates compact embeddings of the same dimension regardless of the input INR modality, the transfer function described here can be applied seamlessly to a great variety of tasks, usually tackled with ad-hoc frameworks tailored to specific input/output modalities. Here, we apply this idea to two tasks. Firstly, we address point cloud completion on the dataset presented in [219] by learning a mapping from inr2vec embeddings of INRs that represent incomplete clouds to embeddings associated with complete clouds. Then, we tackle the task of surface reconstruction on ShapeNet cars, training the transfer function to map inr2vec embeddings representing point clouds into embeddings that can be decoded into meshes. As we can appreciate from the samples in Fig. 5.12b and Fig. 5.12c, the transfer function can learn an effective mapping between inr2vec latent spaces. Indeed, by processing exclusively INRs embedding, we can obtain output shapes that are highly compatible with the input ones while preserving the distinctive details, like the pointy wing of the airplane in Fig. 5.12b or the flap of the first car in Fig. 5.12c.

Method	train set	
	CD (mm) ↓	F-Score ↑
OccupancyNetworks	0.8	0.44
DeepSDF	11.1	0.14
LatentModulatedSiren	0.7	0.37
Individual INRs	**0.3**	**0.50**

(A) Train set.

Method	test set	
	CD (mm) ↓	F-Score ↑
OccupancyNetworks	1.3	0.39
DeepSDF	6.6	0.25
LatentModulatedSiren	1.9	0.28
Individual INRs	**0.3**	**0.49**

(B) Test set.

TABLE 5.4. **Individual INRs vs. shared network frameworks.** Comparison between discrete meshes reconstructed from individual INRs and from shared network frameworks on Manifold40.

5.6 Additional Results and Ablation Studies

5.6.1 Individual INRs vs. Shared Network Frameworks

In this section we aim at comparing the representation power of individual INRs (*i.e.*, one network for each data point) with the one of frameworks adopting a single shared network for the whole dataset, like DeepSDF [25], OccupancyNetworks [27] or [165]. The important difference between such approaches and our method relies in the fact that in shared network frameworks, the shared network and the set of latent codes are the implicit representation, whose reconstruction quality is negatively affected by using a single network to represent the whole dataset. In our framework, instead, we decouple the representations (INRs) from the embeddings used to process them in downstream tasks (yielded by inr2vec). The quality of the representation is then entrusted to the individual INRs and inr2vec does not influence it.

To compare the representation quality of individual INRs with the one of share network frameworks, we fitted the SDF of the meshes in the Manifold40 dataset with OccupancyNetworks [27], DeepSDF [25] and LatentModulatedSiren (*i.e.*, the architecture used by the contemporary work that addresses deep learning on INRs [165]). Then, we reconstructed the training discrete meshes from the three frameworks and we compared them with the ground-truth ones, performing the same comparison using the discrete shapes reconstructed from individual INRs. To perform the comparison, we first reconstructed meshes and then we sampled dense point clouds (16,384 points) from the reconstructed surfaces, doing the same for the ground-truth meshes. We report the quantitative comparisons in Tab. 5.4, using two metrics: the Chamfer Distance as defined in [214] and the F-Score as defined in [220].

The comparison reported in the Tab. 5.4a shows that both OccupancyNetworks and LatentModulatedSiren cannot represent the shapes of the training set with a good fidelity, most likely because of the single shared network that struggles to fit a large number of shapes with high variability (∼10K shapes, 40 classes). At the same time, DeepSDF obtains really poor scores, highlighting the difficulty of training an auto-decoder framework on a large and varied dataset. Individual INRs, instead, produce reconstructions with good quality, even if we adopted a fitting procedure with only 500 steps for each shape.

Moreover, the approaches based on a conditioned shared network tend to fail in representing unseen samples that are out of the training distribution. Hence, in the foreseen scenario where INRs become a standard representation for 3D data hosted in public repositories, leveraging on a single shared network may imply the need to frequently retrain the model upon uploading new samples which, in turn, would change the embeddings of all the previously stored data. On the contrary, uploading the repository with a new shape would not cause any sort of issue with individual INRs, where one learns a network for each data point.

To better support our statements, in Tab. 5.4b we report the comparison between shape reconstructed from OccupancyNetworks, DeepSDF, LatentModulatedSiren and individual INRs, using shapes from the test set of Manifold40, *i.e.*, shapes unseen at training time. The numbers show that both OccupancyNetworks and LatentModulatedSiren present a drop in the quality of the reconstructions, indicating that both frameworks struggle to represent new shapes not available at training time. Surprisingly, DeepSDF produces better scores on the test set *w.r.t.* the results on the train set but still presenting a quite poor performance in comparison with the other methods. Conversely, the problem of representing unseen shapes is inherently inexistent when adopting individual INRs, as shown by the numbers in the last row of Tab. 5.4b, which are almost identical to the ones presented in Tab. 5.4a.

We report in Fig. 5.13 and Fig. 5.14 the comparison described above from a qualitative perspective. It is possible to observe that the visualizations confirm the results posted in Tab. 5.4, with shared network frameworks struggling to represent properly the ground-truth shapes, while individual INRs enable high fidelity reconstructions.

We believe that these results highlight that frameworks based on a single shared network cannot be used as medium to represent shapes as INRs, because of their limited representation power when dealing with large and varied datasets and because of their difficulty in representing new shapes not available at training time.

5.6.2 Ablation on INRs Size

Fig. 5.15 reports a study that we conducted to determine the size of the INRs adopted throughout our experiments. More specifically, we considered three alternatives of SIREN, all featuring 4 hidden layers but different number of hidden features, namely 128, 256 and 512 respectively.

In the figure we show how the three SIREN variants perform in terms of being able of representing properly the underlying signal, which in this example is the orange plane on the left. Since we needed to create datasets comprising a huge number of INRs, we considered both the quality of the representation as well as the number of steps of the fitting procedure, with the goal of finding the best trade-off between quality and fitting time.

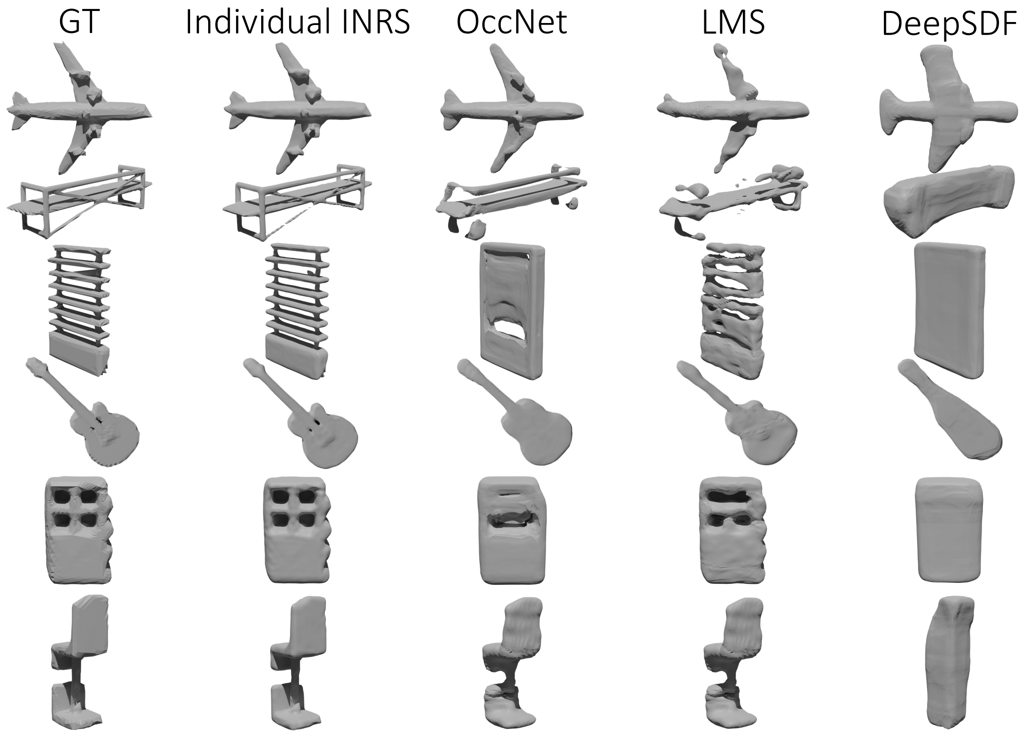


FIGURE 5.13. **Individual INRs vs. shared network frameworks (train shapes).** Qualitative comparison of meshes from Manifold40 reconstructed from individual INRs or from shared network frameworks, when dealing with training shapes. OccNet stands for OccupancyNetworks, LMS stands for LatentModulatedSiren.

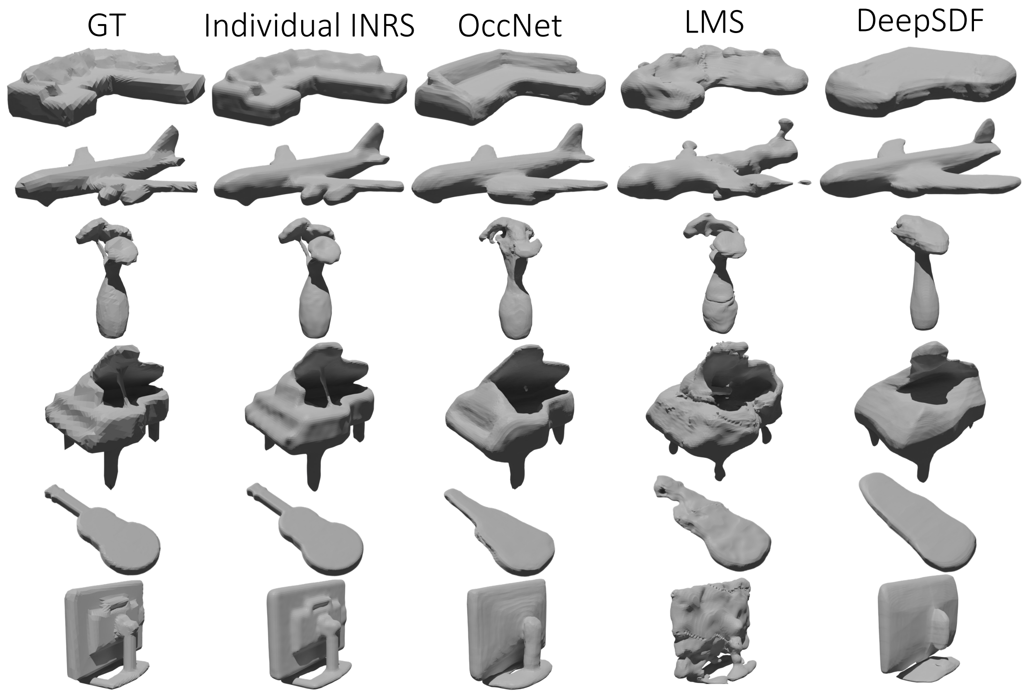


FIGURE 5.14. **Individual INRs vs. shared network frameworks (test shapes).** Qualitative comparison of meshes from Manifold40 reconstructed from individual INRs or from shared network frameworks, when dealing with shapes unseed during training. OccNet stands for OccupancyNetworks, LMS stands for LatentModulatedSiren.

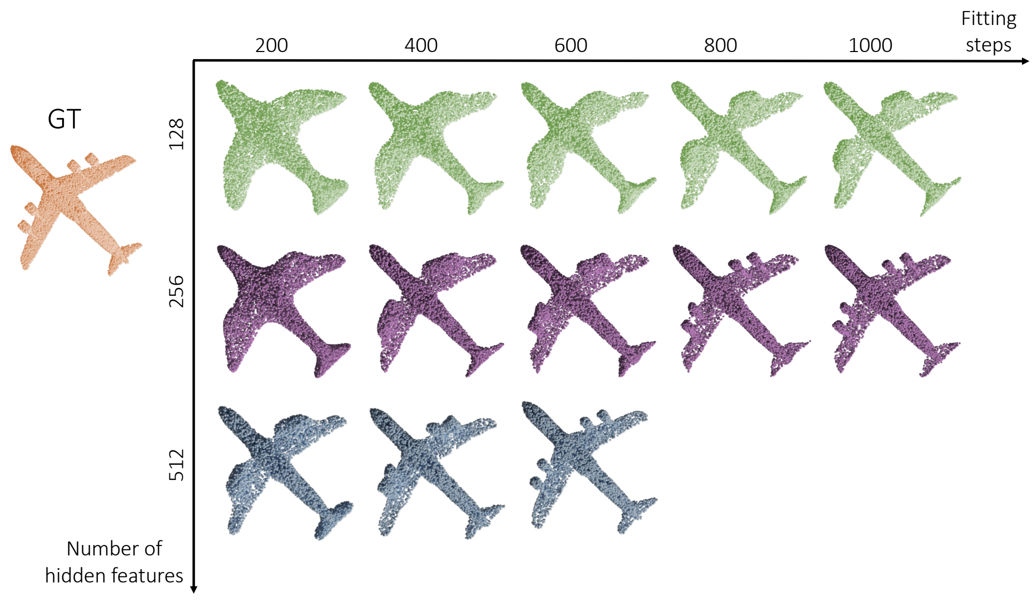


FIGURE 5.15. **Comparison between different hidden sizes for INRs.** We report the fitting steps needed to obtain a good representation for INRs featuring different number of hidden features.

The results presented in the figure highlight how a SIREN with 512 hidden features can learn to represent properly the input shape in just 600 steps, while the other variants either take longer (as in the case of 256 hidden features) or not obtain at all the same quality (when using 128 hidden features).

This experiment enabled us to draw the conclusion that a SIREN with 4 hidden layers and 512 hidden features is the proper tool to obtain a good quality INR in short time.

5.6.3 Deep Learning on DeepSDF Latent Codes

In Sec. 5.6.1 we show that frameworks that adopt a shared network to produce INRs struggle to obtain a good representation quality, while individual INRs do not suffer of this problem by design. In this section we goes one step further and consider the possibility of peforming downstream tasks on the latent codes produced by DeepSDF [25].

In particular, we trained DeepSDF to fit the UDFs of our augmented version of ModelNet40, composed of ~100K point clouds. For a fair comparison, we set the dimension of DeepSDF latent codes to 1024 – *i.e.*, the same used for inr2vec embeddings. Then we performed the experiments of shape retrieval and classification using DeepSDF latent codes, with the same settings presented in Sec. 5.5 for our framework.

The results reported in Tab. 5.5 highlight that the poor representation quality obtained with DeepSDF – and shown to be an intrinsic problem with shared network frameworks in Sec. 5.6.1 – has a detrimental effect on the quantitative results, proving once again that INR frameworks based on a shared network cannot be deployed as tool to obtain and process INRs effectively.

	ModelNet40		
Method	mAP@1	mAP@5	mAP@10
PointNet [16]	80.1	91.7	94.4
PointNet++ [162]	85.1	93.9	96.0
DGCNN [122]	83.2	92.7	95.1
inr2vec	81.7	92.6	95.1
DeepSDF	69.8	85.4	89.8

	ModelNet40
Method	Accuracy
PointNet [16]	88.8
PointNet++ [162]	89.7
DGCNN [122]	89.9
inr2vec	87.0
DeepSDF	64.9

TABLE 5.5. **Comparison between inr2vec and DeepSDF.** We report results in shape retrieval (left) and shape classification (right) when using standard baselines, inr2vec embeddings or DeepSDF latent codes.

5.6.4 Shape Generation: Additional Comparison

In Fig. 5.11b we show a qualitative comparison between shapes generated with our framework (see Sec. 5.5) and with competing methods, *i.e.*, SP-GAN [218] for point clouds and OccupancyNetworks [27] for meshes.

In Fig. 5.16 we extend this comparison, by presenting samples obtained with our formulation applied to the voxelized chairs of ShapeNet10 and comparing them with samples produced by two additional methods that learn a manifold of individual INRs, namely GEM [221] and GASP [222], for which we used the original source code released by the authors. To generate the figure, despite the sampled shapes being voxel grids, we adopt the same procedure used by GEM and GASP and reconstruct meshes by applying Marching Cubes to extract the 0.5 isosurface.

Fig. 5.16 show that all the considered methods can generate samples with a good variety in terms of geometry. However, it is possible to observe how the qualitative comparison favors the shape generated with inr2vec, that appear smoother than the ones generated by GASP and less noisy that the samples produced by GEM.

5.6.5 Additional Qualititative Results

We report here additional qualitative results. In Fig. 5.17, Fig. 5.18 and Fig. 5.19 we show some comparisons between the discrete shapes reconstructed from input INRs and the ones reconstructed from inr2vec embeddings. Fig. 5.20, Fig. 5.21 and Fig. 5.22 present smooth interpolations between inr2vec embeddings. In Fig. 5.23 and Fig. 5.24 we propose additional qualitative results for the point cloud retrieval experiments. Fig. 5.25 shows qualitative results for point cloud part segmentation for all the classes of the ShapeNet Part Segmentation dataset. Fig. 5.26, Fig. 5.27 and Fig. 5.28 report shapes generated with Latent-GANs [217] trained on inr2vec embeddings. Finally, Fig. 5.29 and Fig. 5.30 present additional qualitative results for the experiments of point cloud completion and surface reconstruction, tackled by learning a mapping between inr2vec latent spaces.

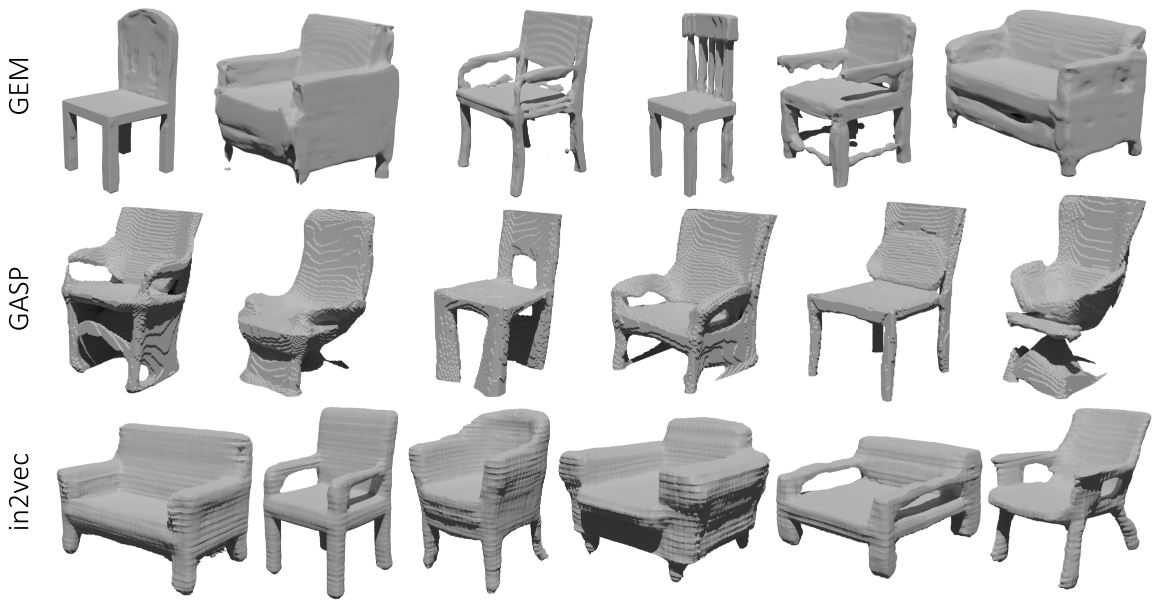


FIGURE 5.16. **Shape generation: qualitative comparison.** We show samples generated with GEM [221], GASP [222] and with our method.

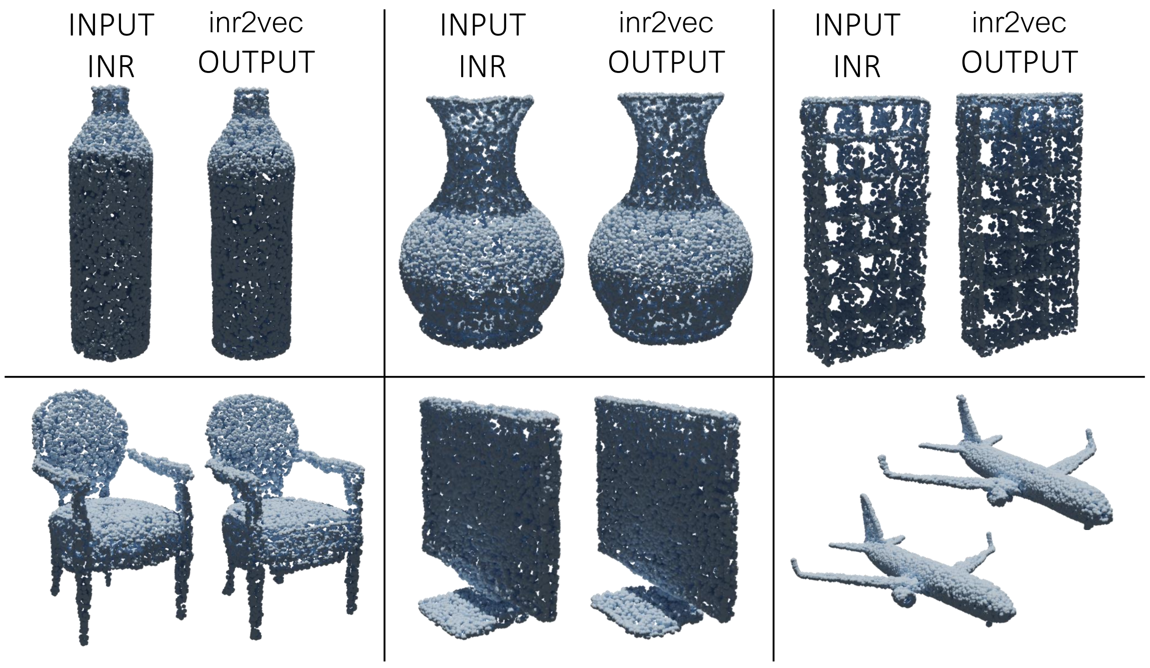


FIGURE 5.17. **inr2vec reconstructions when dealing with INRs fitted on point clouds.** Comparison between discrete shapes reconstructed from the INRs presented in input to inr2vec ("INPUT INR") and the ones reconstructed from inr2vec embeddings ("inr2vec OUTPUT").

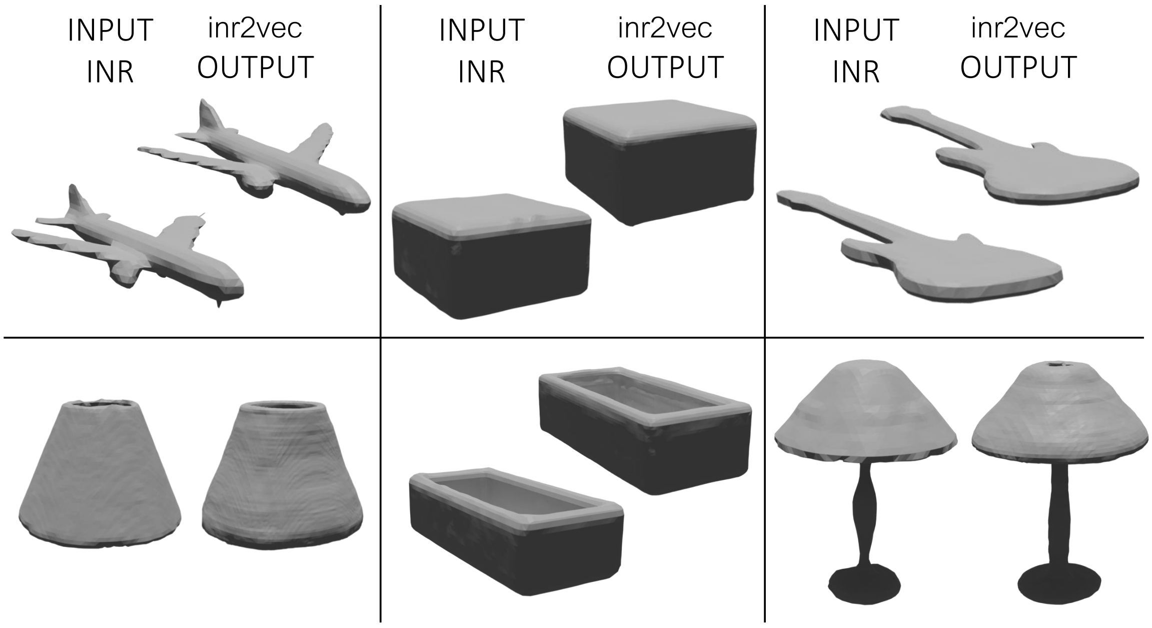


FIGURE 5.18. **inr2vec reconstructions when dealing with INRs fitted on meshes.** Comparison between discrete shapes reconstructed from the INRs presented in input to inr2vec ("INPUT INR") and the ones reconstructed from inr2vec embeddings ("inr2vec OUTPUT").

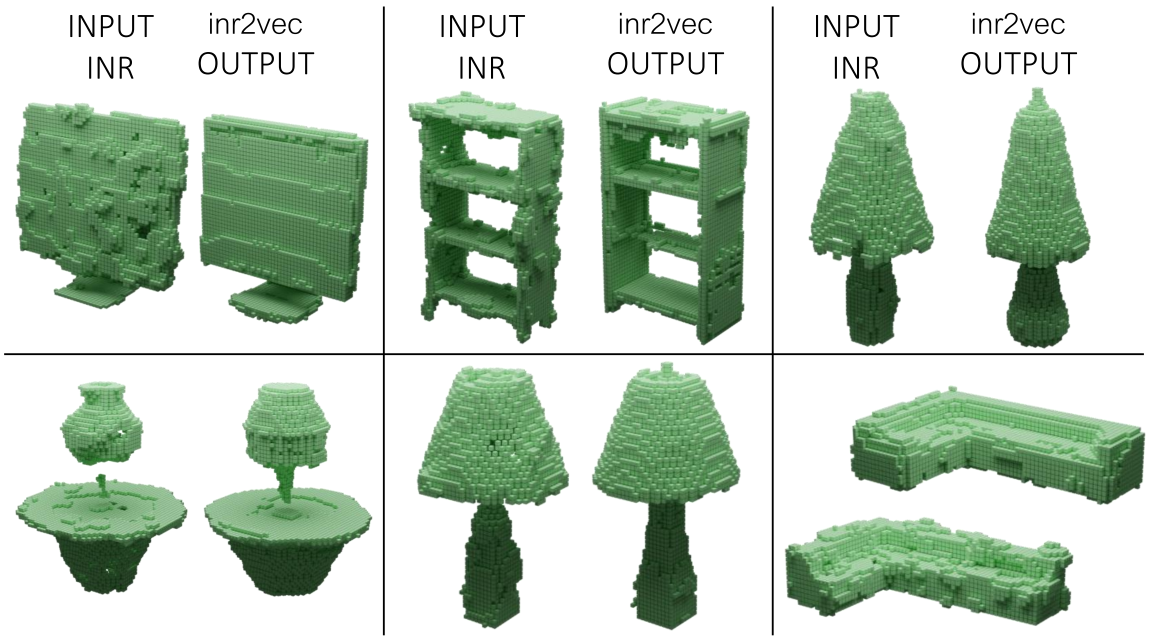


FIGURE 5.19. **inr2vec reconstructions when dealing with INRs fitted on voxel grids.** Comparison between discrete shapes reconstructed from the INRs presented in input to inr2vec ("INPUT INR") and the ones reconstructed from inr2vec embeddings ("inr2vec OUTPUT").

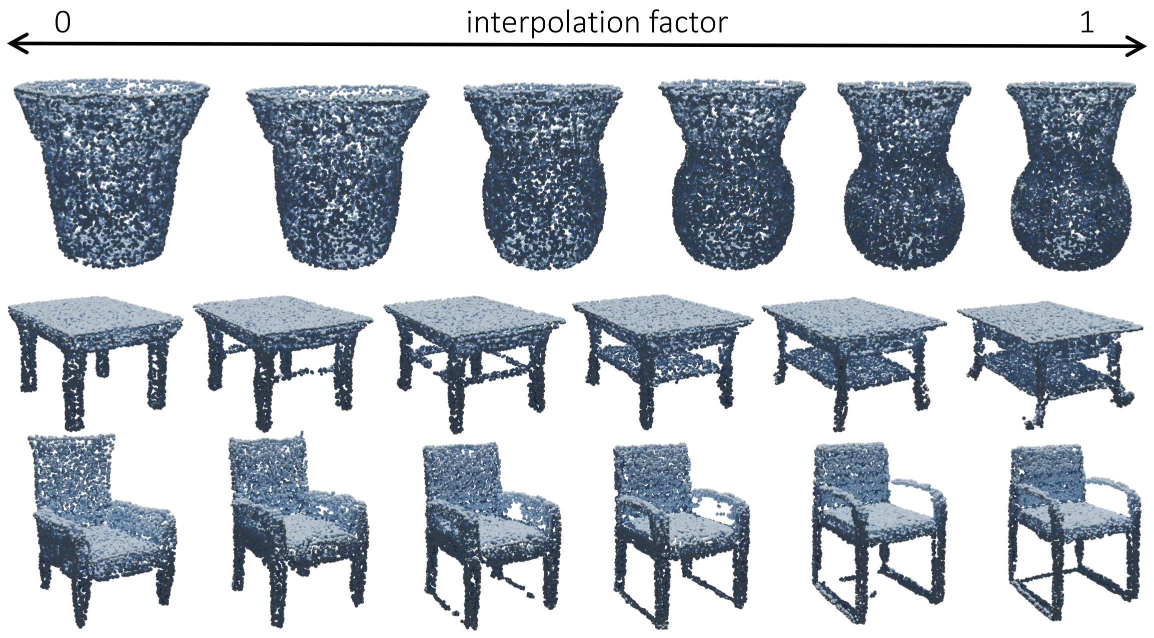


FIGURE 5.20. **inr2vec latent space interpolation.** Given two inr2vec embeddings obtained from INRs fitted on point clouds, it is possible to linearly interpolate between them, producing new embeddings that represent unseen INRs of plausible shapes.

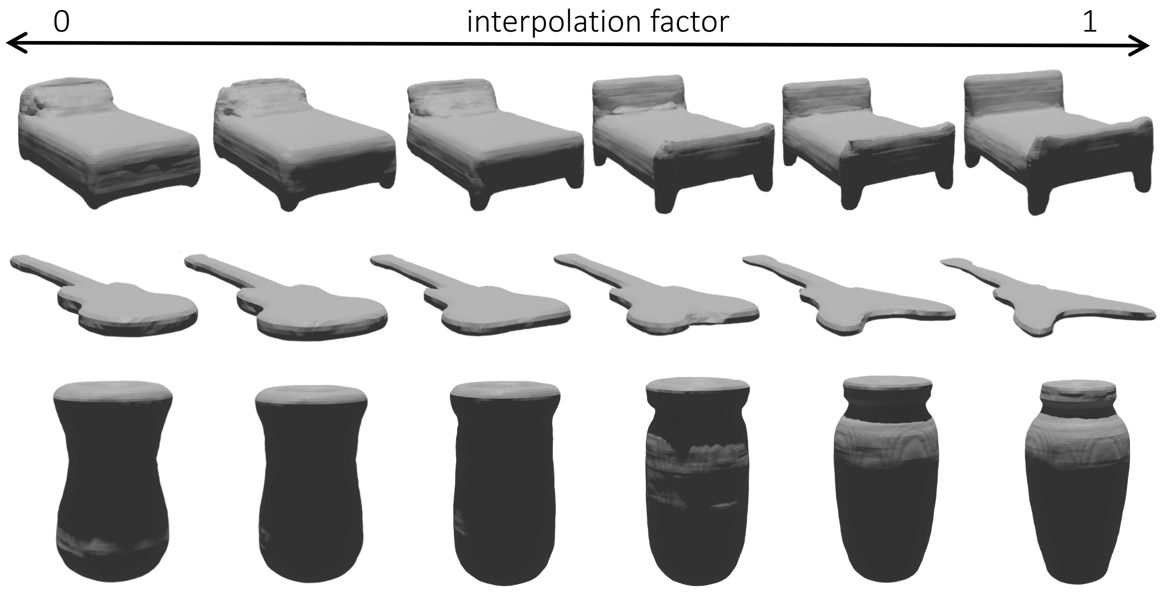


FIGURE 5.21. **inr2vec latent space interpolation.** Given two inr2vec embeddings obtained from INRs fitted on meshes, it is possible to linearly interpolate between them, producing new embeddings that represent unseen INRs of plausible shapes.

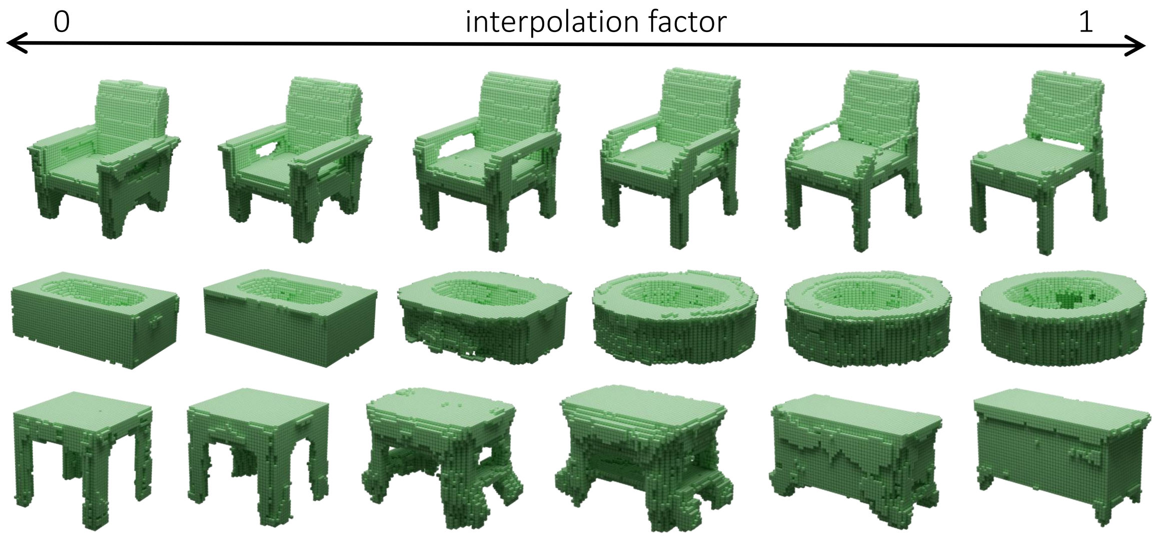


FIGURE 5.22. **inr2vec latent space interpolation.** Given two inr2vec embeddings obtained from INRs fitted on voxel grids, it is possible to linearly interpolate between them, producing new embeddings that represent unseen INRs of plausible shapes.

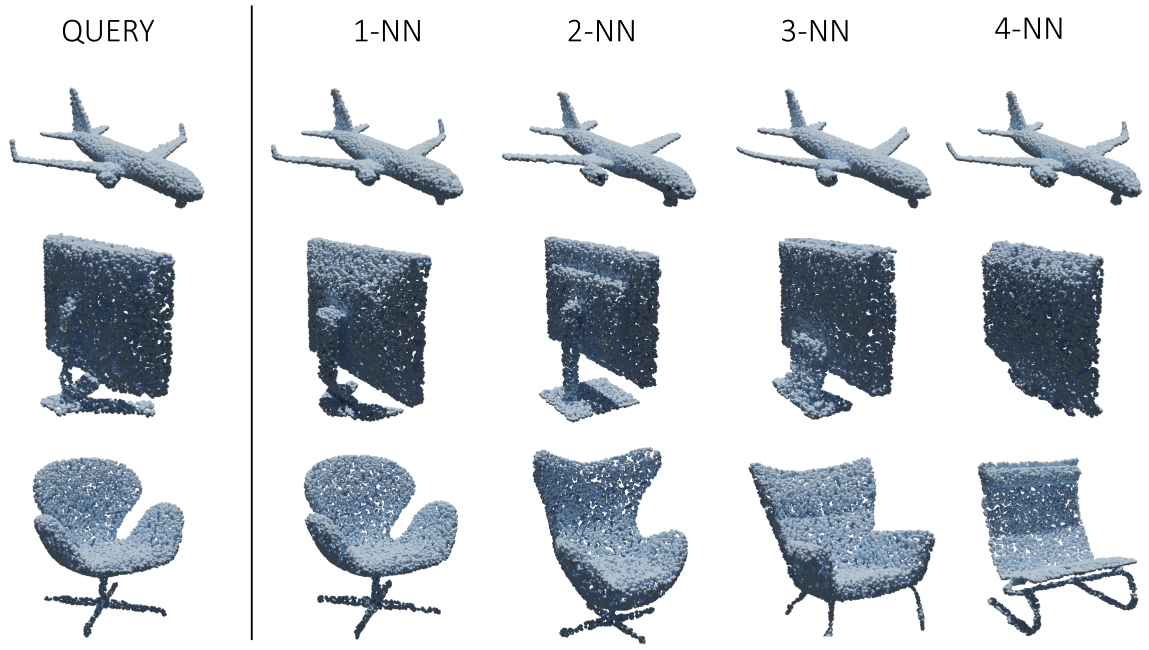


FIGURE 5.23. **Point cloud retrieval (ModelNet40).** Qualitative results of the point cloud retrieval experiment conducted on inr2vec latent space. We show the discrete shape reconstructed from the query INR on the left and the discrete clouds reconstructed from the closest inr2vec embeddings in the columns 2-5.

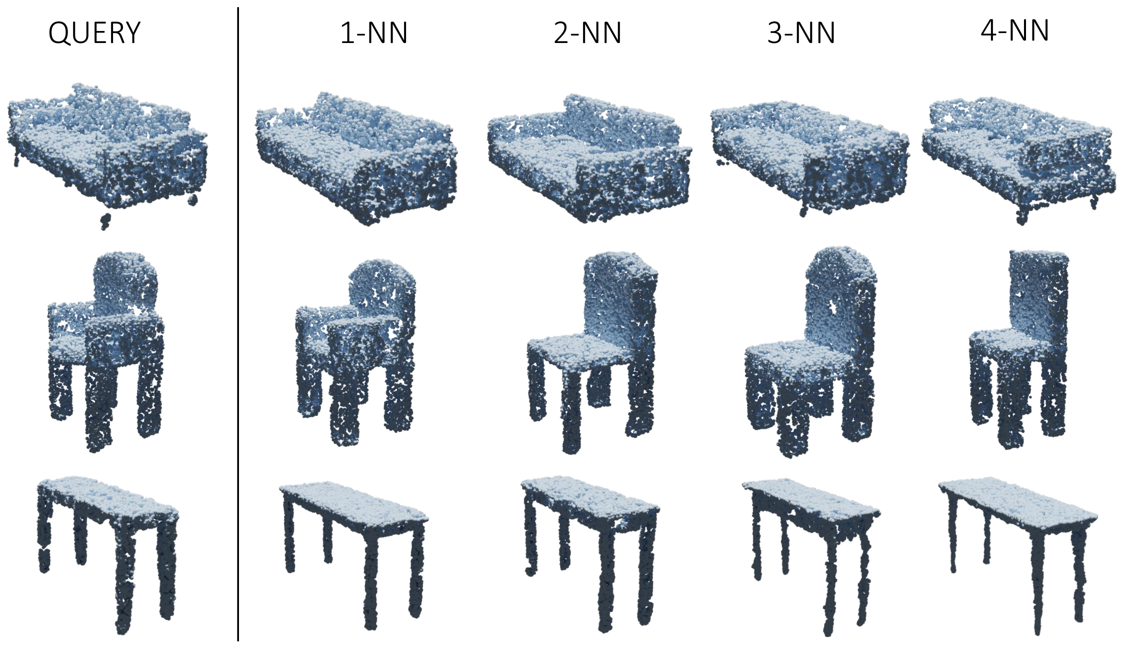


FIGURE 5.24. **Point cloud retrieval (ShapeNet10).** Qualitative results of the point cloud retrieval experiment conducted on inr2vec latent space. We show the discrete shape reconstructed from the query INR on the left and the discrete clouds reconstructed from the closest inr2vec embeddings in the columns 2-5.

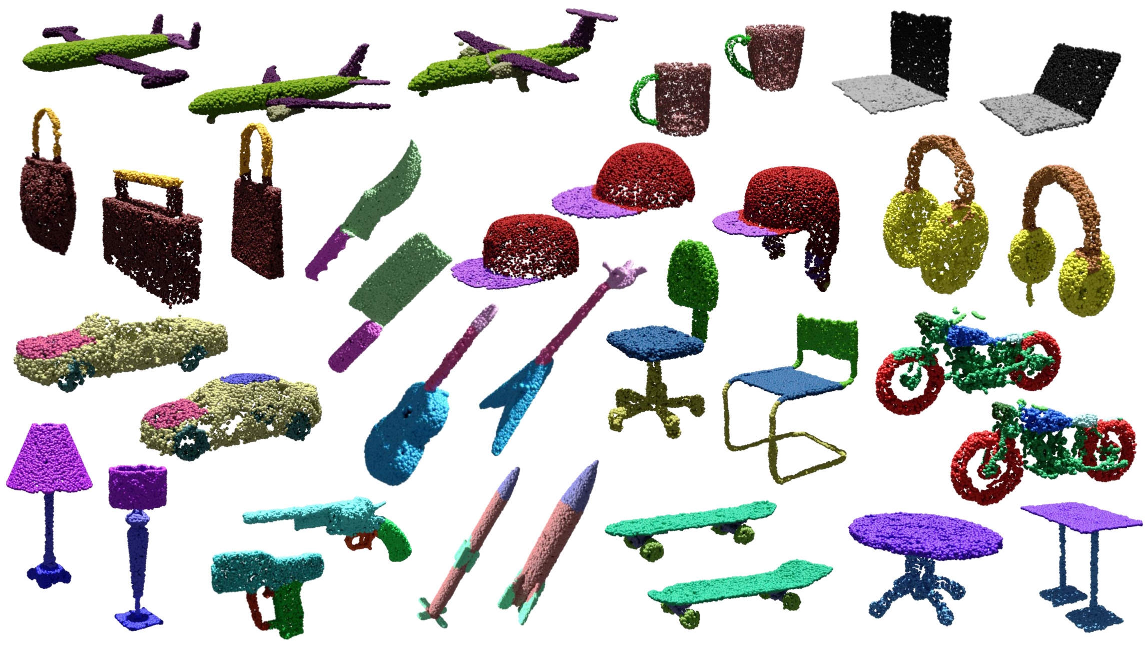


FIGURE 5.25. **Point cloud part segmentation.** Qualitative results of the part segmentation experiment conducted with our segmentation decoder conditioned on inr2vec embeddings.

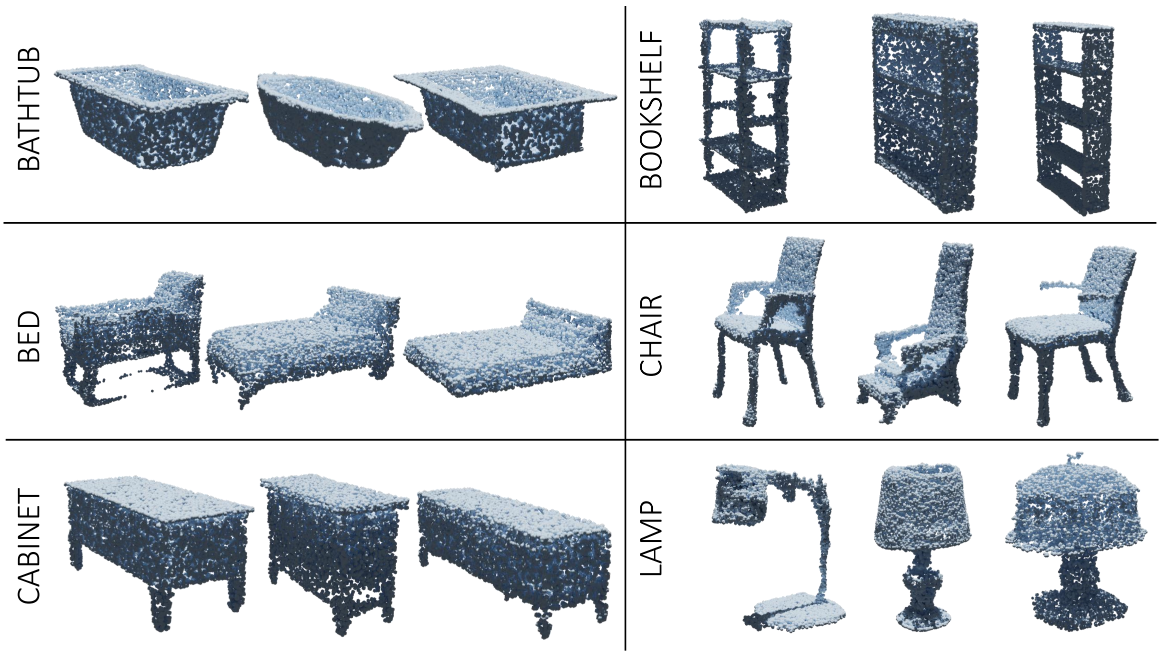


FIGURE 5.26. **Shape generation (point clouds).** We show point clouds reconstructed from embeddings generated by a Latent-GAN trained on inr2vec embeddings (one model for each class).

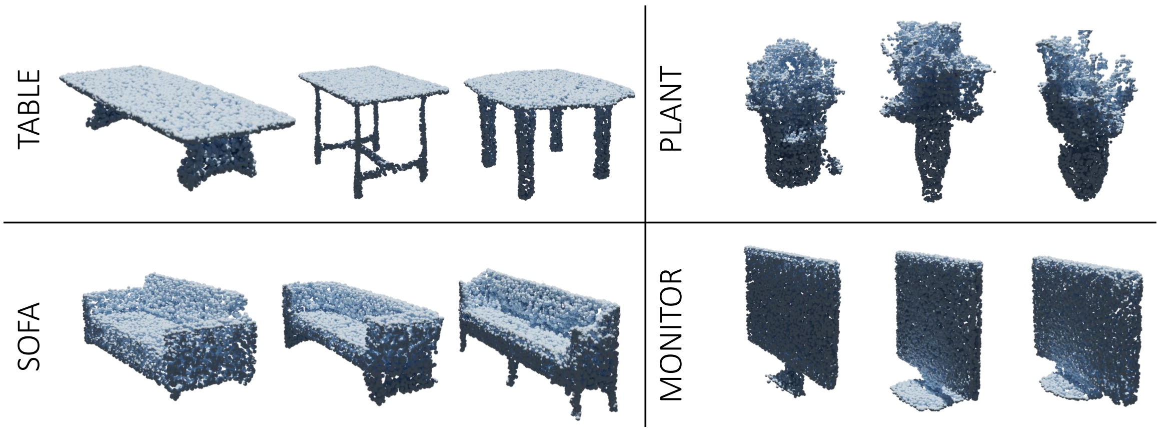


FIGURE 5.27. **Shape generation (point clouds).** We show point clouds reconstructed from embeddings generated by a Latent-GAN trained on inr2vec embeddings (one model for each class).

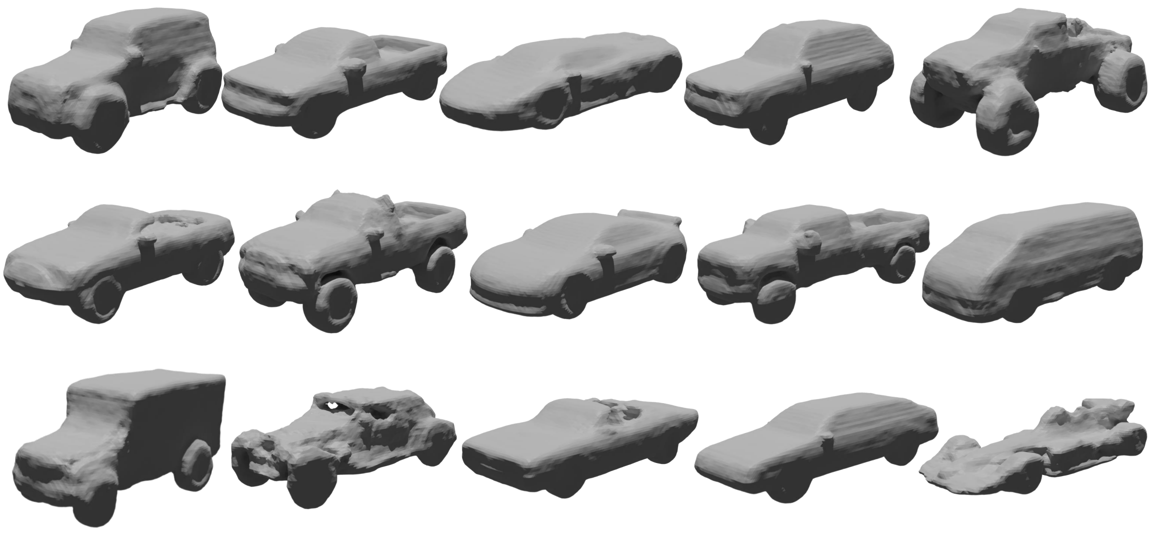


FIGURE 5.28. **Shape generation (meshes).** We show meshes reconstructed from embeddings generated by a Latent-GAN trained on inr2vec embeddings.

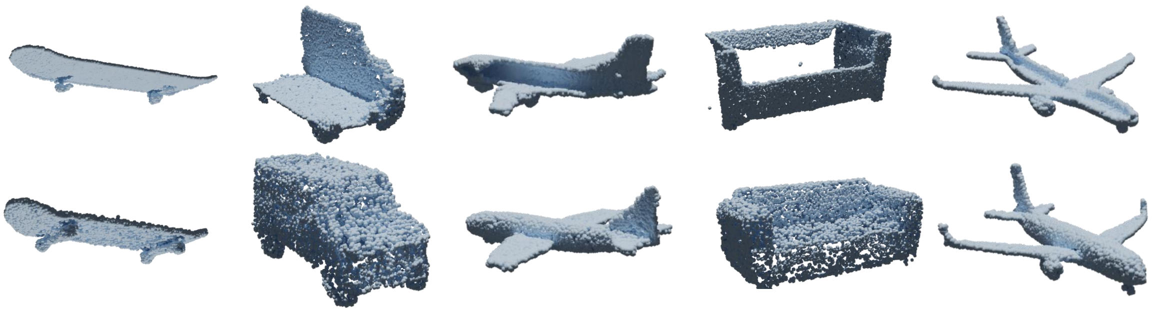


FIGURE 5.29. **Point cloud completion.** Qualitative results of the point cloud completion experiment conducted with a transfer network that learns a mapping between inr2vec latent spaces.

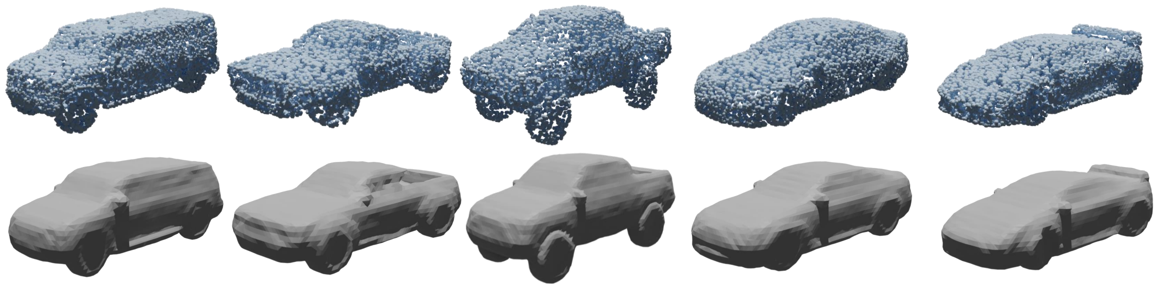


FIGURE 5.30. **Surface reconstruction.** Qualitative results of the surface reconstruction experiment conducted with a transfer network that learns a mapping between inr2vec latent spaces.

5.7 Conclusion

We have shown that it is possible to apply deep learning directly on individual INRs representing 3D shapes. Our formulation of this novel research problem leverages on a task-agnostic encoder which embeds INRs into compact and meaningful latent codes without accessing the underlying implicit function. Our framework ingests INRs obtained from different 3D discrete representations and performs various tasks through standard machinery. However, we point out two main limitations: i) Although INRs capture continuous geometric cues, inr2vec embeddings achieve results inferior to state-of-the-art solutions ii) There is no obvious way to perform online data augmentation on shapes represented as INRs by directly altering their weights.

Future works may investigate these shortcomings as well as apply inr2vec to other input modalities like images, audio or radiance fields. Another interesting direction for future work concerns exploring weight-space symmetries [223] as a different path to favour alignment of weights across INRs despite the randomness of training.

We reckon that our work may foster the adoption of INRs as a unified and standardized 3D representation, thereby overcoming the current fragmentation of discrete 3D data and associated processing architectures.

Chapter 6

Final Remarks

In this thesis, we have investigated the possibility of adopting neural representations (NR) as a way of representing continuous signals in different scenarios. The first part of this manuscript deals with *deploying* NR as the only medium to represent data in two different contexts.

In Chapter 2 we showed extensively how NR can model accurately the open surfaces of garments, by approximating their unsigned distance function (UDF). The use of UDF allows for representing open surfaces of any geometry and topology in a fully differentiable manner. Moreover, our approach leads to a continuous latent space of garments, that could be used to sample brand-new items and explored by means of gradient descent.

The main problem in our method is the reliance on a single neural network to represent a whole dataset of garments. The limited capacity of the network could entail a loss in accuracy, especially for clothing items that deviate from the dataset geometric priors. A possible solution for this would be representing each garment with a separate (small) neural network, a solution that we showed being superior to shared network frameworks in Sec. 5.6.1. However, this solution would also make it more difficult to obtain the latent space of garments described above. An alternative solution could feature a single shared network trained to represent local patches of the garments surfaces, instead of their whole surfaces. This approach has been shown to be effective in improving the representation accuracy [224] and it would preserve the possibility of obtaining an explorable latent space.

In Chapter 3 we presented a simple yet effective scan station that enables the creation of a Neural Twin® of a given object in a few minutes, by training a fast variant of NeRF [24] on the collected images. This scenario highlights a clear advantage in adopting NR: the Neural Twin® obtained with ScanNeRF provides an extremely compact representation of the object, entailing the possibility of rendering an infinite number of novel views without the need of storing such images.

One issue in using NeRF concerns the slow training process, which however has been already drastically reduced by works such as [32, 79, 78] and is actively tackled by many researchers. Another limitation in adopting NeRF concerns the difficulty in modeling reflectant and transparent surfaces, a problem that is still open at the time of writing. Finally, being able of recovering an accurate 3D model of the underlying object from a NeRF model would be of interest to many applications, but doing that is still far from being trivial.

The second part of the thesis is focused on *processing* NR, *i.e.*, on using NR as input/output data to solve given tasks that are usually tackled by processing discrete representations of the signals – *e.g.*, grid of pixels for images. Since sampling the underlying signal from a NR typically involves cumbersome procedures, we aim at the possibility of processing directly the weights of the NR, to exclude completely the sampling overhead.

Thus, we first presented NetSpace in Chapter 4, a framework that we designed to prove that the weights of a neural network form a redundant parametrization of the function approximated by the network. Indeed, with NetSpace we show that it is possible to squeeze the weights of a neural network into a low-dimensional embedding and that such embedding contains all the information needed to restore the input network.

In Chapter 5 we build on top of the findings from NetSpace and present inr2vec, a framework that allows for processing implicit neural representations (INR) of 3D shapes by only looking at the networks weights. More specifically, inr2vec has at its core an encoder which squeezes the weights of an input INR into a compact embedding, that can be processed with standard deep learning machinery to tackle a great variety of tasks.

While we believe the results that we presented show that it is possible to consider a future where NR are used as standalone representations for continuous signals, several shortcomings still need to be addressed before such future could become a reality. First, NR are typically obtained with slow fitting procedures starting from discrete samplings of the target signal, an obstacle that could discourage the deployment of NR at a large scale. Then, one must consider that, given the NR of a signal, it is extremely difficult or even impossible to perform some kind of manipulation of the signal by acting exclusively on the NR weights. For instance, given the INR of a 3D point cloud, it is not possible to apply a certain rotation by modifying directly the weights of the INR. Furthermore, while we consider the results from inr2vec really promising, our solution to perform deep learning directly on the INR weights shows inferior performance to the algorithms developed in the past decades, that have been carefully designed to work only on certain discrete representations of signals. Finally, it must be remembered that NR are an approximation of the represented signals and that obtaining an accurate approximation is not always trivial, especially when dealing with signals that feature high-frequency details. In such cases, more than often, the accuracy of NR is still not satisfying.

However, we are thrilled to see that at the time of writing many researchers are exploring new ways to overcome the shortcomings listed above [225, 226, 227, 228, 229] and we still argue that NR have the potential to become a standalone representation for continuous signal.

In conclusion, we believe that this thesis provides useful insights on the possibility of deploying and processing NR of signals, considering them as first-class representations.

Part III

Appendices

Appendix A

Modeling Garments with Unsigned Distance Functions

A.1 Network Architectures and Training

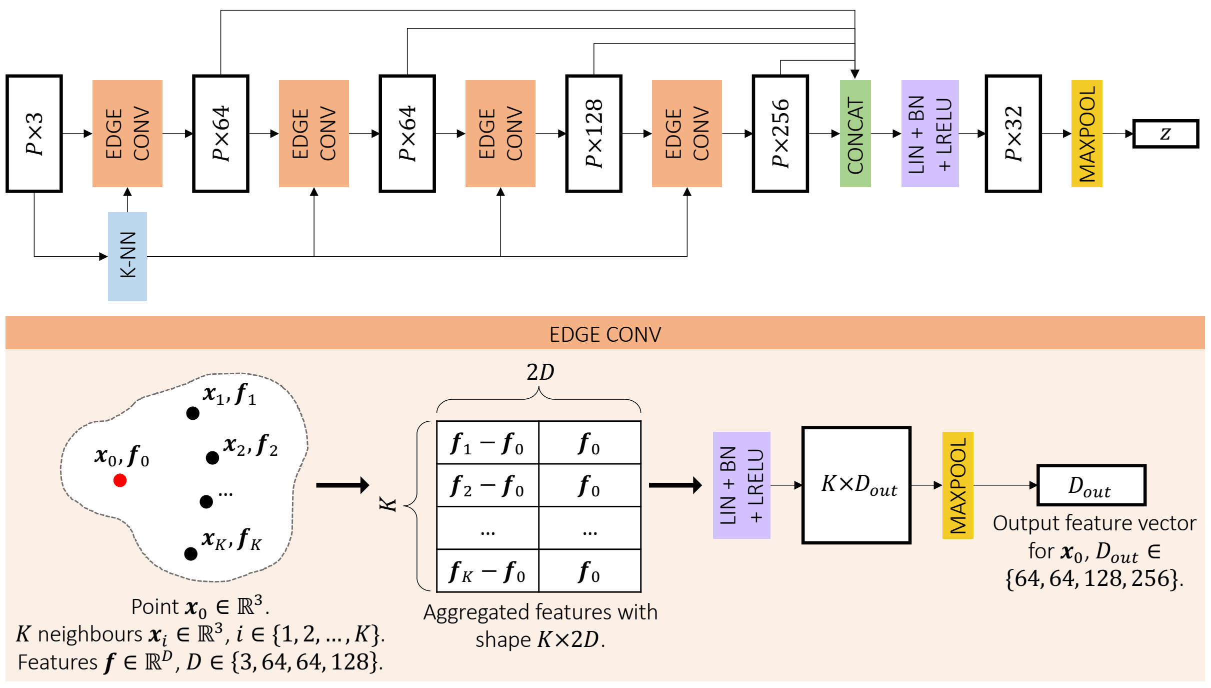
A.1.1 Garment Generative Network: Encoder

To encode a given garment into a compact latent code, we first sample P points from its surface and then we feed them to a DGCNN [122] encoder, detailed in Fig. A.1. The input point cloud is processed by four *edge convolution* layers, which project the input 3D points into features with increasing dimensionality – *i.e.*, 64, 64, 128 and finally 256.

Each edge convolution layer works as follows. For each input point, the features from its K neighbours are collected and used to prepare a matrix with K rows. Each row is the concatenation of two vectors: $\mathbf{f}_i - \mathbf{f}_0$ and $\mathbf{f}_0$, $\mathbf{f}_i$ and $\mathbf{f}_0$ being respectively the feature vector of the i-th neighbour and the feature vector of the considered point. Each row of the resulting matrix is then transformed independently to the desired output dimension. The output feature vector for the considered point is finally obtained by applying max pooling along the rows of the produced matrix.

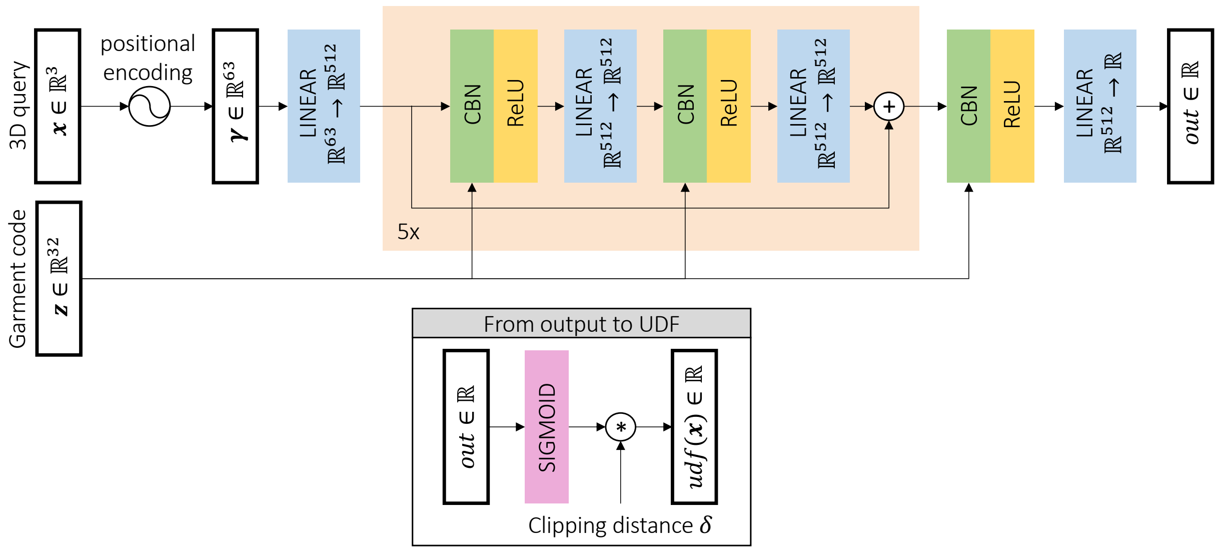
The original DGCNN implementation recomputes the neighborhoods in each edge convolution layer, using the distance between the feature vectors as metric. This can be explained by the original purposes of DGCNN, *i.e.*, point cloud classification and part segmentation. Since we are interested in encoding the geometric details of the input point cloud, we compute neighborhoods only once based on the euclidean distance of the points in the 3D space and reuse this information in every edge convolution layer. We set $K = 16$ in our experiments.

The feature vectors from the four edge convolutions are then concatenated to form a single vector with 512 elements, that is fed to a final linear layer paired with batch normalization and leaky ReLU. Such layer projects the 512 sized vectors into the final desired dimension, which is 32 in our case. The final latent code is obtained by compressing the feature matrix with shape $P \times 32$ along the first dimension with max pooling.



$P\times3$
EDGE CONV
$P\times64$
$P\times64$
$P\times128$
$P\times256$
CONCAT
LIN + BN + LRELU
$P\times32$
MAXPOOL
z
K-NN
EDGE CONV
2D
x_1, f_1
x_0, f_0
x_2, f_2
x_K, f_K
K
$f_1 - f_0$ f_0
$f_2 - f_0$ f_0
$f_K - f_0$ f_0
$K\times D_{out}$
D_{out}
Output feature vector for x_0, $D_{out} \in \{64, 64, 128, 256\}$.
Point $x_0 \in \mathbb{R}^3$.
K neighbours $x_i \in \mathbb{R}^3$, $i \in \{1, 2, \ldots, K\}$.
Features $f \in \mathbb{R}^D$, $D \in \{3, 64, 64, 128\}$.
Aggregated features with shape $K\times2D$.

FIGURE A.1. **DGCNN point cloud encoder.** We adopt DGCNN [122] as the point cloud encoder of our garment generative network. The input cloud is passed through four *edge convolutions*, which gather features of local neighborhoods of points to project them into higher dimensional spaces. The features from all the layers are then concatenated and projected to the final desired dimension. Max pooling is finally used to obtain the latent code **z** for the input cloud. *CONCAT* stands for features concatenation, while *LIN + BN + LRELU* represents a linear layer followed by batch normalization and leaky ReLU.



3D query
$x \in \mathbb{R}^3$
positional encoding
$\gamma \in \mathbb{R}^{63}$
LINEAR $\mathbb{R}^{63} \to \mathbb{R}^{512}$
CBN
ReLU
LINEAR $\mathbb{R}^{512} \to \mathbb{R}^{512}$
5x
LINEAR $\mathbb{R}^{512} \to \mathbb{R}$
$out \in \mathbb{R}$
Garment code
$z \in \mathbb{R}^{32}$
From output to UDF
SIGMOID
$udf(x) \in \mathbb{R}$
Clipping distance δ

FIGURE A.2. **UDF decoder.** Given a 3D query and a garment latent code, the decoder of our garment generative network is trained to predict the UDF of the input query *w.r.t.* the surface of the garment. The latent code is used to condition the prediction by the means of Conditional Batch Normalization (CBN) [123]. Since we trained the decoder with the binary cross-entropy loss, its outputs need to be converted to UDF values by applying the sigmoid function and then scaling the result with the UDF clipping distance δ.

A.1.2 Garment Generative Network: Decoder

The garment generative network features an implicit decoder that can predict the unsigned distance field of a garment starting from its latent code. More specifically, the decoder is a coordinate-based MLP that takes as inputs the garment latent code and a 3D query. Using the latent code as condition, the decoder predicts the unsigned distance from the query to the garment surface.

Our UDF decoder, shown in Fig. A.2, is inspired by [27]. The input 3D query is first mapped to a higher dimensional space ($\mathbb{R}^{63}$) with the positional encoding proposed in [24], which is known to improve the capability of the network to approximate high frequency functions. The encoded query is then mapped with a linear layer to $\mathbb{R}^{512}$ and then goes through 5 residual blocks. The output of each block is computed as $\mathbf{f}_{out} = \mathbf{f}_{in} + \Delta\mathbf{f}$, where $\mathbf{f}_{in}$ is the input vector and $\Delta\mathbf{f}$ is a residual term predicted by two consecutive linear layers starting from $\mathbf{f}_{in}$. The size of the feature vector is 512 across the whole sequence of residual blocks. The output of the last block is mapped to the scalar output $out \in \mathbb{R}$ with a final linear layer.

All the linear layers but the output one are paired with Conditional Batch Normalization (CBN) [123] and ReLU activation function. CBN is used to condition the MLP with the input latent code $\mathbf{z}$. In more details, each CBN module applies standard batch normalization [230] to the input vectors, with the difference that the parameters of the final affine transformation are not learned during the training but are instead predicted at each inference step by dedicated linear layers starting from $\mathbf{z}$.

As a final remark, we recall that our generative network is trained with the binary cross-entropy loss. Thus, the output of the decoder must be converted to the corresponding UDF value by first applying the sigmoid function and then scaling the result with the UDF clipping distance δ, which we set to 0.1 in our experiments. Such procedure is indeed the dual of the one applied on the UDF ground-truth labels during training to normalize them in the range $[0, 1]$.

A.1.3 Garment Generative Network: Surface Sampling

We sample supervision points with a probability inversely proportional to the distance to the surface: 30% of the points are sampled directly on the input surface, 30% are sampled by adding gaussian noise with ϵ variance to surface points, 30% are obtained with gaussian noise with 3ϵ variance, and the remaining ones are gathered by sampling uniformly the bounding box in which the garment is contained. Since in our experiments, the top and bottom garments are normalized respectively into the upper and lower halves of the $[-1, 1]^3$ cube, we set $\epsilon = 0.003$.

A.1.4 Draping Network

The networks $\mathcal{W}(\mathbf{x}) \in \mathbb{R}^{24}$ and $\Delta x(\mathbf{x}, \beta) \in \mathbb{R}^3$ that predict blending weights and coarse displacements are implemented by a 9-layer multilayer perceptron (MLP) with a skip connection from the input layer to the middle. Each layer has 256 nodes except the middle and the last ones. ReLU is used as the activation function. The body-parameter-embedding module $\mathcal{B}(\beta, \theta) \in \mathbb{R}^{128}$ and the displacement-matrix module $\mathcal{M}(x, \mathbf{z}) \in \mathbb{R}^{128\times 3}$ for Δx_{ref} are implemented by a 5-layer MLP with LeakyReLU activation in-between. Each layer has 512 nodes except the last one. Δx_{IS} uses the same architecture as Δx_{ref}.

A.1.5 Training Hyperparameters

The generative models (top/bottom ones) are trained on the 600/300 neutral garments for 4000 epochs, using mini-batches of size $B = 4$. Each item of the mini-batch contains an input point cloud with $P = 10,000$ points and $N = 20,000$ random UDF 3D queries. The dimension of the latent codes is set to 32 for both top and bottom garments, and we set $\lambda_g = 0.1$ in

$$\mathcal{L}_{garm} = \mathcal{L}_{dist} + \lambda_g \mathcal{L}_{grad} \ . \tag{A.1}$$

The draping networks are trained for 250K iterations with mini-batches of size 18, where each item is composed of the vertices of one garment paired with one body shape and pose. We set $\lambda = 0.1$ for $\mathcal{L}_{pin}$ and $\gamma = 0.5$ for $\mathcal{L}_{layer}$.

Both the generative and the draping networks are trained with Adam optimizer [231] and learning rates set to 0.0001 and 0.001 respectively.

A.2 Loss Terms and Ablation Studies

A.2.1 $\mathcal{L}_{pin}$ for Bottom Garments

To determine V, the set of bottom garment vertices that need to be constrained by $\mathcal{L}_{pin}$, we first find the closest body vertex v_B for each bottom garment vertex v. If v_B locates in the body trunk (cyan region as shown in Fig. A.3), v is added to V.

In Fig. A.4, we show the draping results of bottom garments by using different values for λ in $\mathcal{L}_{pin}$. When λ equals 0 or 1, the deformations along the X and Z axes are not natural because no constraints or too strong constraints are applied, while it is not the case when $\lambda = 0.1$, which is our setting.

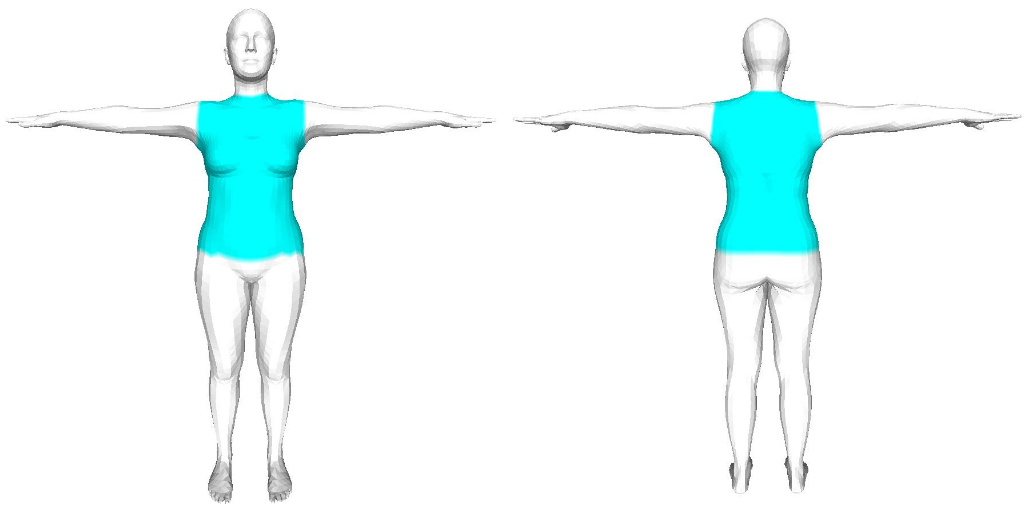


FIGURE A.3. Body region (marked in cyan) used to compute $\mathcal{L}_{pin}$.

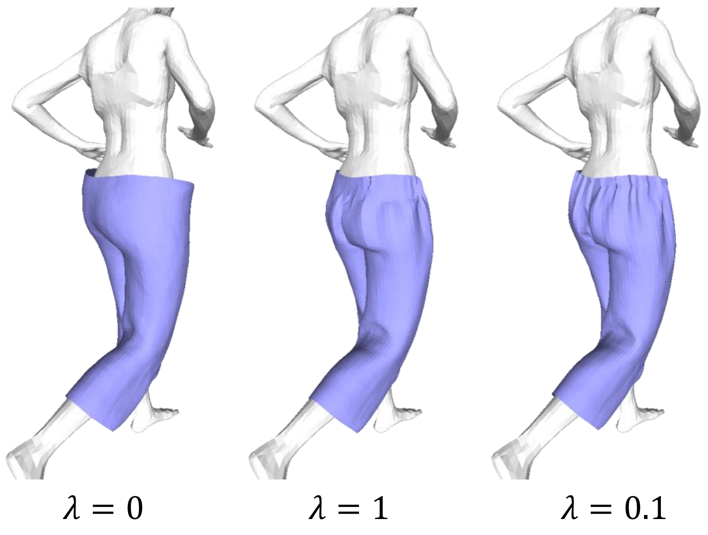


FIGURE A.4. **Comparison between different values for λ of $\mathcal{L}_{pin}$.** To restrict the deformation mainly along the vertical direction (Y axis) and produce natural deformations along other directions, λ has to be a positive value smaller than 1. We use $\lambda = 0.1$ for our training.

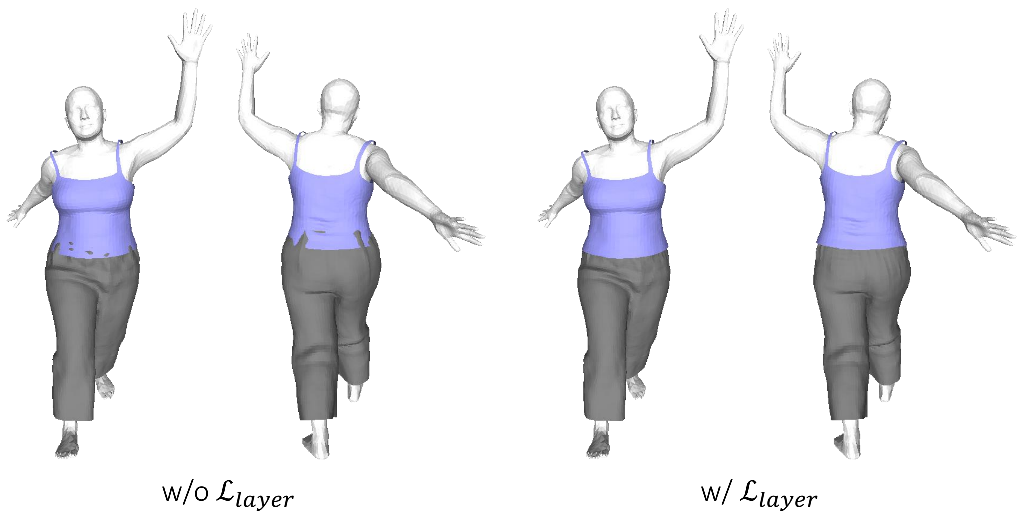


FIGURE A.5. **Comparison: draping without and with $\mathcal{L}_{layer}$.** Without it, the top and bottom garments intersect with each other.

A.2.2 $\mathcal{L}_{layer}$ for Top-bottom Intersection

To determine C, the set of body vertices covered by both the top and bottom garments, we first subdivide the SMPL body mesh for a higher resolution, and then we compute C_{top} the set of closest body vertices for the given top garment, and C_{bottom} the set of closest body vertices for the bottom. C is derived as the intersection of C_{top} and C_{bottom}.

In Fig. A.5 we compare the results of models trained without and with $\mathcal{L}_{layer}$. We can observe that without $\mathcal{L}_{layer}$, the top tank can intersect with the bottom trousers, while it is not the case when using $\mathcal{L}_{layer}$. This indicates the efficacy of $\mathcal{L}_{layer}$ to avoid intersections between garments.

A.2.3 Physics-based Refinement

After recovering the draped garment $\mathbf{G}_D$ from images by the optimization of Eq. (2.12), we can apply the physics-based objectives of Eq. (2.7) to increase its level of realism

$$\begin{aligned} L(\Delta_\mathbf{G}) =& \mathcal{L}_{strain}(\mathbf{G}_D + \Delta_\mathbf{G}) + \mathcal{L}_{bend}(\mathbf{G}_D + \Delta_\mathbf{G}) \\ &+ \mathcal{L}_{gravity}(\mathbf{G}_D + \Delta_\mathbf{G}) + \mathcal{L}_{col}(\mathbf{G}_D + \Delta_\mathbf{G}) \,, \end{aligned} \tag{A.2}$$

where $\Delta_\mathbf{G}$ is the per-vertex-displacement initialized to zero. For the recovery from 3D scans, we apply the following optimization which minimizes both the above physics-based

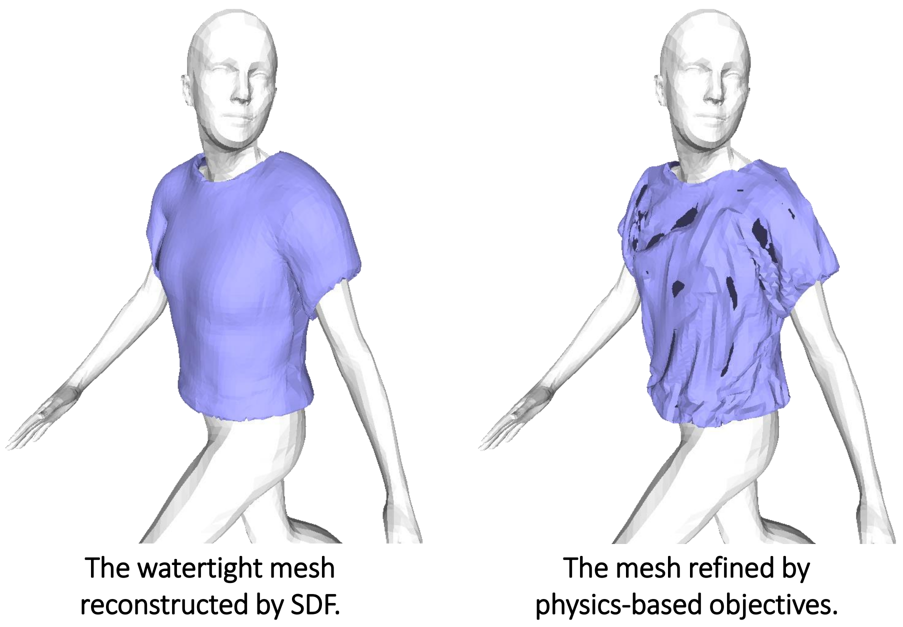


FIGURE A.6. **Applying post-refinement procedure to watertight mesh**. Left: the watertight mesh reconstructed by DIG [111]. Right: the same mesh after being refined with physics-based objectives (Eq. (A.2)). Physics-based refinement is not compatible with inflated garment meshes, and leads to many self-intersections.

objectives and the Chamfer Distance $d(\cdot)$ to the input scan $\mathbf{S_G}$

$$\begin{aligned} L(\Delta_\mathbf{G}) = &\mathcal{L}_{strain}(\mathbf{G}_D + \Delta_\mathbf{G}) + \mathcal{L}_{bend}(\mathbf{G}_D + \Delta_\mathbf{G}) \\ &+ \mathcal{L}_{gravity}(\mathbf{G}_D + \Delta_\mathbf{G}) + \mathcal{L}_{col}(\mathbf{G}_D + \Delta_\mathbf{G}) \\ &+ d(\mathbf{G}_D + \Delta_\mathbf{G}, \mathbf{S_G}) \,. \end{aligned} \tag{A.3}$$

This refinement procedure is only applicable to open surface meshes, and our UDF model is thus key to enabling it. Applying Eq. (A.2) or Eq. (A.3) to an inflated garment (as recovered by SMPLicit [110], ClothWild [127] and DIG [111]) indeed yields poor results with many self-intersections as illustrated in Fig. A.6. This is because inflated garments modelled as SDFs have a non-zero thickness, with distinct inner and outer surfaces whose interactions are not taken into account in this fabric model. Note that this is the case for most garment draping softwares [106, 114, 232, 233, 98] to expect single layer garments. Modeling garment with UDFs is thus a key feature of our pipeline for its integration in downstream tasks.

Both the optimizations of Eqs. (2.12) and (2.13) and Eqs. (A.2) and (A.3) are done with Adam [214] but with different learning rates set to 0.01 and 0.001 respectively.

A.3 Additional Results and Considerations

A.3.1 Garment Encoder/Decoder Latent Space Optimization (LSO)

After training the garment generative network, we obtain a latent space that allows us to sample a garment latent code and to feed it to the implicit decoder to reconstruct the explicit surface. We study here the possibility of exploring the garment latent space by the means of LSO. To do that, given a target 2D silhouette or a sparse 3D point cloud of a garment, we optimize with gradient descent a latent code – initialized to the training codes average – so that the frozen decoder conditioned on it can produce a garment which fits the target image or point cloud.

Given the silhouette $\mathcal{S}$ of a target garment, we can retrieve its latent code $\mathbf{z}$ by minimizing

$$\begin{aligned} L(\mathbf{z}) &= L_{\text{IoU}}(R(\mathbf{G}), \mathcal{S}) \ , \\ \mathbf{G} &= \text{MeshUDF}(D_G(\cdot, \mathbf{z})) \ , \end{aligned} \tag{A.4}$$

where L_{IoU} is the IoU loss [130] in pixel space measuring the difference between 2D silhouettes , $R(\cdot)$ is a differentiable silhouette renderer for meshes [131], and $\mathbf{G}$ is the garment mesh reconstructed with our garment decoder using $\mathbf{z}$.

In the case of a target garment provided as a point cloud $\mathcal{P}$, the garment latent code $\mathbf{z}$ can be obtained by minimizing

$$\begin{aligned} L(\mathbf{z}) &= d(ps(\mathbf{G}), \mathcal{P}) \ , \\ \mathbf{G} &= \text{MeshUDF}(D_G(\cdot, \mathbf{z})) \ , \end{aligned} \tag{A.5}$$

where $d(a, b)$ is the Chamfer distance [214] between point clouds a and b, and $ps(\cdot)$ represents a differentiable procedure to sample points from a given mesh [131].

In both cases, we run the optimization for 1000 steps, with Adam optimizer [231] and learning rate set to 0.01.

In Fig. A.7 and Fig. A.8 we present some results of the LSO procedures here described, showing that the latent space learned by the garment generative network can be explored effectively with gradient descent to recover the codes associated with the target garments.

A.3.2 Draping Network: Euclidean Distance is not a Good Metric

In Fig. A.9, we show an example of bottom garment where our result is more realistic than the competitors DeePSD [119] and DIG [111] despite having the highest Euclidean distance. This demonstrates again that Euclidean distance is not able to measure the draping quality.

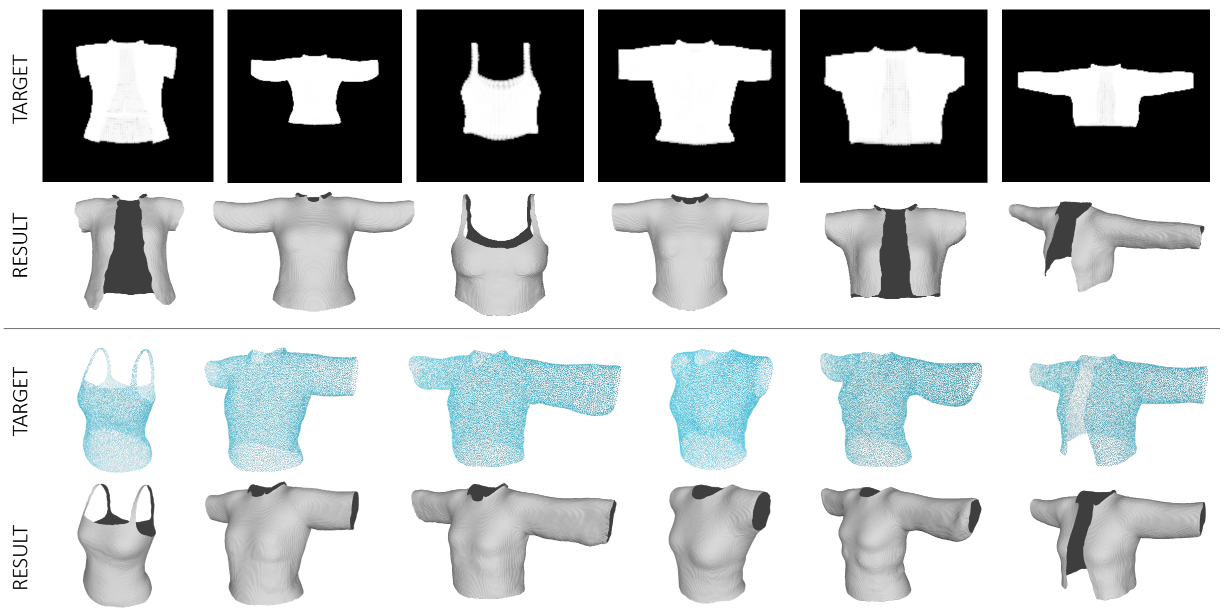


FIGURE A.7. **Generative network: latent space optimization (top garments).** After training, we can explore the latent space learned by the garment generative network with gradient descent, to recover target garments from 2D silhouettes (top) or 3D point clouds (bottom).

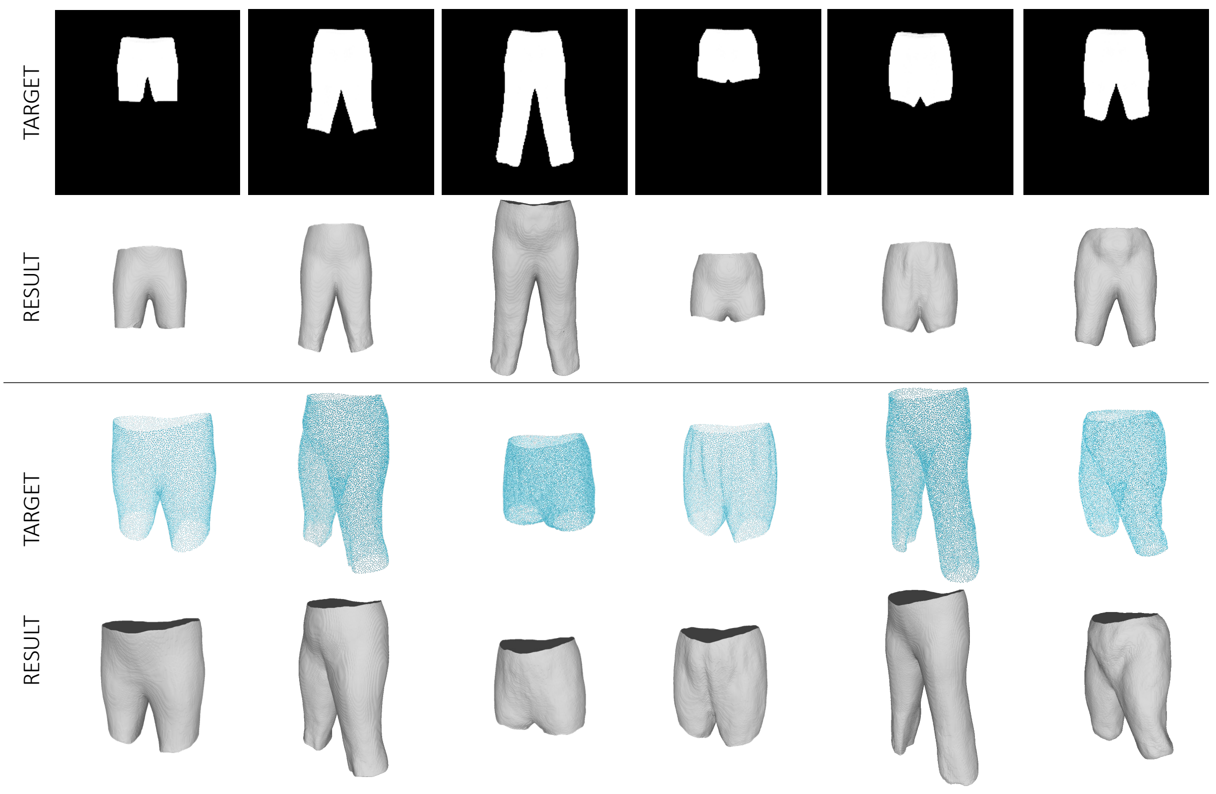


FIGURE A.8. **Generative network: latent space optimization (bottom garments).** After training, we can explore the latent space learned by the garment generative network with gradient descent, to recover garments from 2D silhouettes (top) or 3D point clouds (bottom).

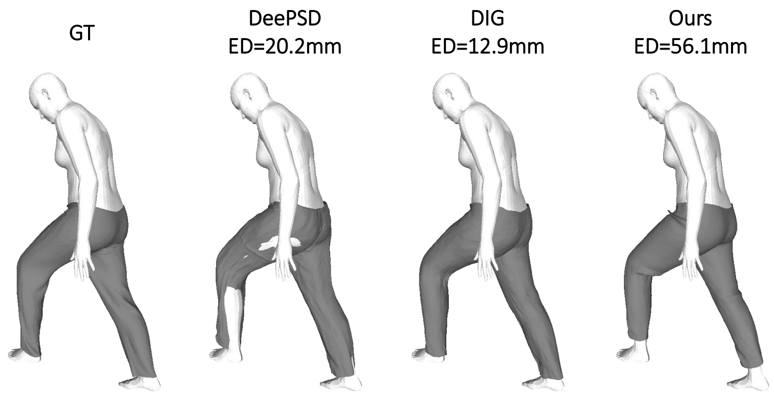


FIGURE A.9. **Comparison between DeePSD, DIG and our method.** Our result is more realistic than the others despite having the highest Euclidean distance (ED) error.

Top	Strain ↓	Bending ↓	Gravity ↓	Total ↓
DeePSD	7.22	**0.01**	**0.98**	8.21
DIG	6.32	**0.01**	1.05	7.38
Ours	**0.43**	**0.01**	1.05	**1.81**

Bottom	Strain ↓	Bending ↓	Gravity ↓	Total ↓
DeePSD	8.46	0.02	0.90	9.38
DIG	7.48	**0.01**	0.90	8.39
Ours	**0.41**	**0.01**	**0.86**	**1.28**

TABLE A.1. **Draping unseen garment meshes.** Quantitative comparison in physics-based energy between DeePSD, DIG and our method. "Strain", "Bending" and "Gravity" denote the membrane strain energy, the bending energy and the gravitational potential energy, respectively.

A.3.3 Draping Network: Physics-based Energy Evaluation

In Tab. A.1, we report the physics-based energy of *Strain*, *Bending* and *Gravity* as proposed by [108] on test garment meshes when draped by DeePSD, DIG and our method. These energy terms are used as training losses for our garment network (Eqs. (2.7) and (2.8)). For the gravitational potential energy, we choose the lowest body vertex as the 0 level. Generally, our results have the lowest energies, especially for the *Strain* component. Since DeePSD and DIG do not apply constraints on mesh faces, their results exhibit much higher *Strain* energy. This indicates that our method can produce results that have more realistic physical properties.

A.3.4 Inference Times

We report inference times for the components of our framework, computed on an NVIDIA Tesla V100 GPU. The garment encoder, which needs to be run only once for each garment,

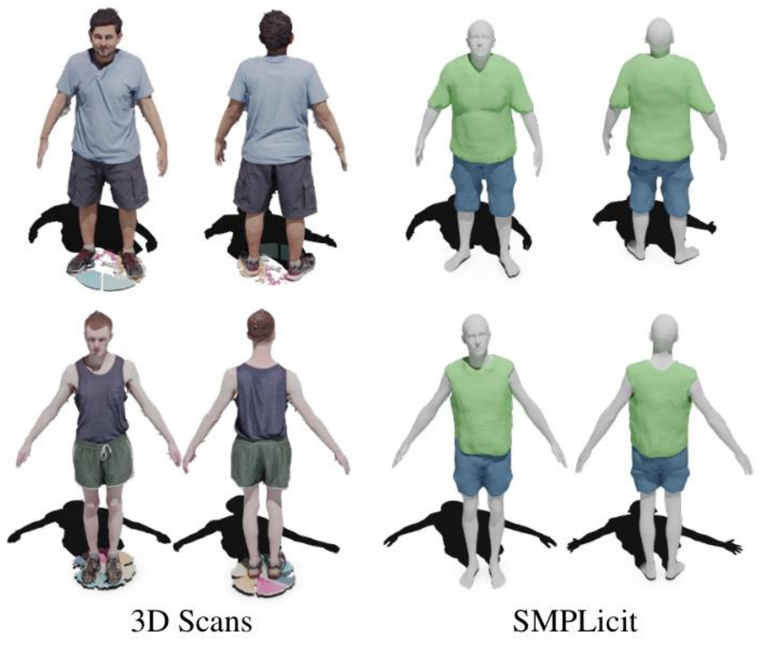


FIGURE A.10. **Recovered garments of SMPLicit from 3D scans**. Figures are extracted from [110].

takes ∼25 milliseconds. The decoder takes ∼2 seconds to reconstruct an explicit garment mesh from a given latent code, including the modified Marching Cubes from [42] at resolution 256^3.

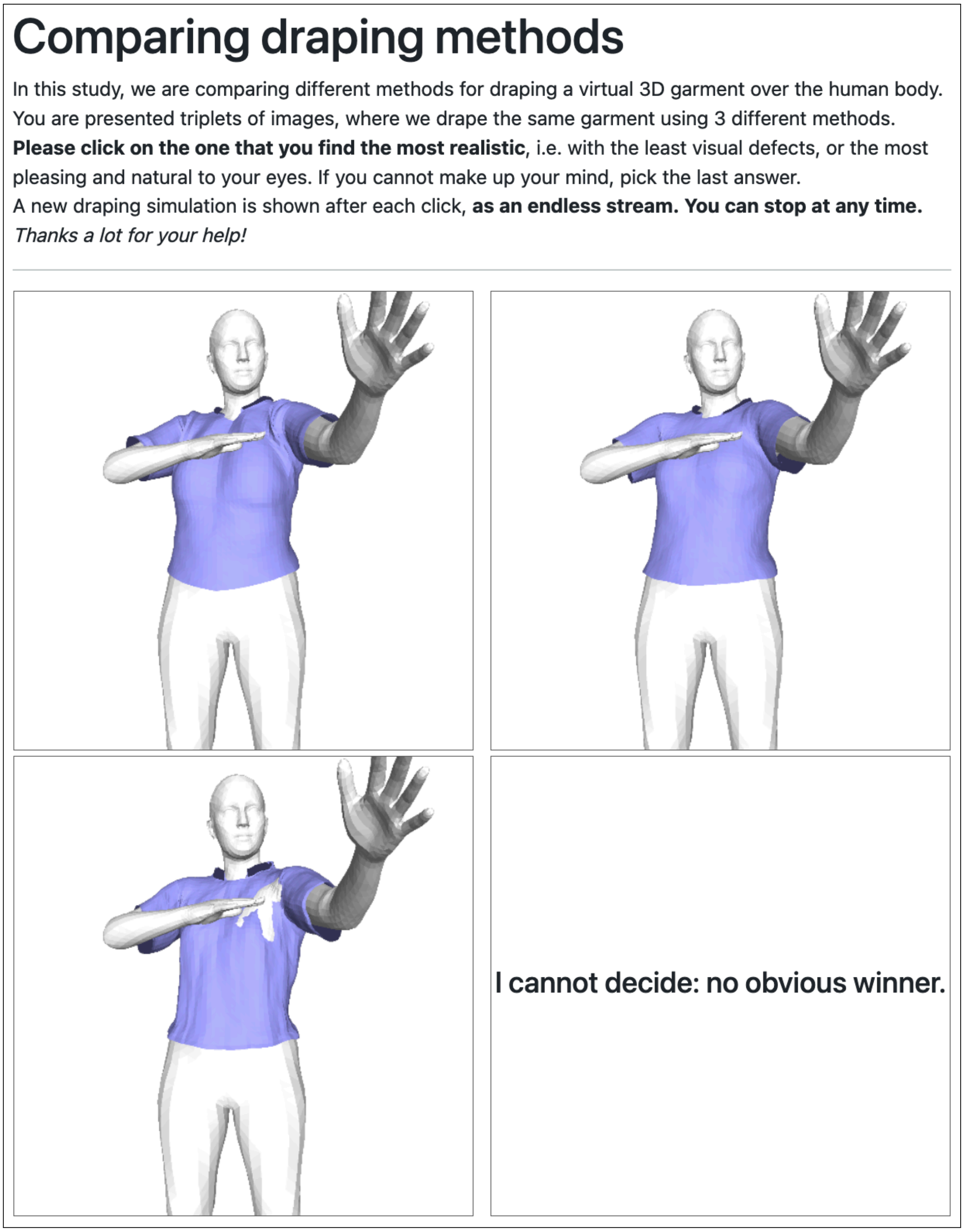
The draping network takes ∼5 ms to deform a garment mesh composed of 5K vertices. Since it is formulated in an implicit manner and is queried at each vertex, its inference time increases to ∼8 ms for a mesh with 8K vertices, or ∼53 ms with 100K vertices.

A.3.5 Fitting SMPLicit to 3D Scans

In Fig. A.10 we show results of fitting the concurrent approach SMPLicit [110] to 3D scans of the SIZER dataset [103]. We can observe that they are not as realistic as ours shown in Fig. 2.13. Since we have no access to their code and not enough information for a re-implementation, we directly extract this figure from [110].

A.4 Human Evaluation

In Fig. A.11 we show the interface and instructions that were presented to the 187 respondents of our survey. These evaluators were volunteers with various backgrounds from the authors respective social circles, which were purposely not given any further detail or instruction. We collected collected 3738 user opinions in total, each user expressing 20 opinions on average.



Comparing draping methods

In this study, we are comparing different methods for draping a virtual 3D garment over the human body. You are presented triplets of images, where we drape the same garment using 3 different methods. **Please click on the one that you find the most realistic**, i.e. with the least visual defects, or the most pleasing and natural to your eyes. If you cannot make up your mind, pick the last answer.
A new draping simulation is shown after each click, **as an endless stream. You can stop at any time.**
Thanks a lot for your help!

I cannot decide: no obvious winner.

FIGURE A.11. **Interface of our qualitative survey.** The garment is draped with our method, DIG, and DeePSD, in a random order.

Appendix B

Learning the Space of Deep Models

B.1 ClassId Classifier

In Sec. 3 (*Multi-Architecture Setting*) of the main paper we introduce the need to extend NetSpace architecture to the Multi-Architecture setting. In this setting, in fact, we ask our framework to extract the ClassId of the predicted instances from the embeddings. To achieve this goal, we extend the architecture of our framework with a softmax classifier, which takes in input the embeddings generated by NetSpace encoder and is trained to predict the correct ClassId with $\mathcal{L}_{class}$. The classifier is a lightweight neural network, composed of one convolutional layer and one fully connected layer, interleaved by the LeakyReLU activation function. The outputs of the classifier are transformed into probabilities by the softmax function. Fig. B.1 presents an overview of the version of NetSpace used in the Multi-Architecture case, including the ClassId classifier.

B.2 Network Architectures

Image Classification. Fig. B.2 and Fig. B.3 present the architecture of the neural networks used in our experiments dealing with image classification, *i.e.*, LeNetLike [159], VanillaCNN

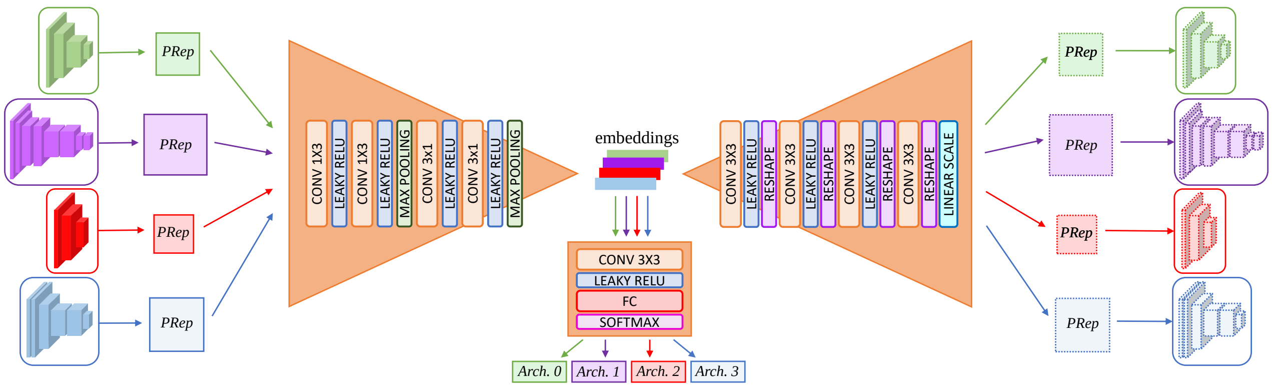


FIGURE B.1. **NetSpace architecture for the Multi-Architecture experiments.** An additional component is trained to predict the ClassId of the input instance from the embedding.

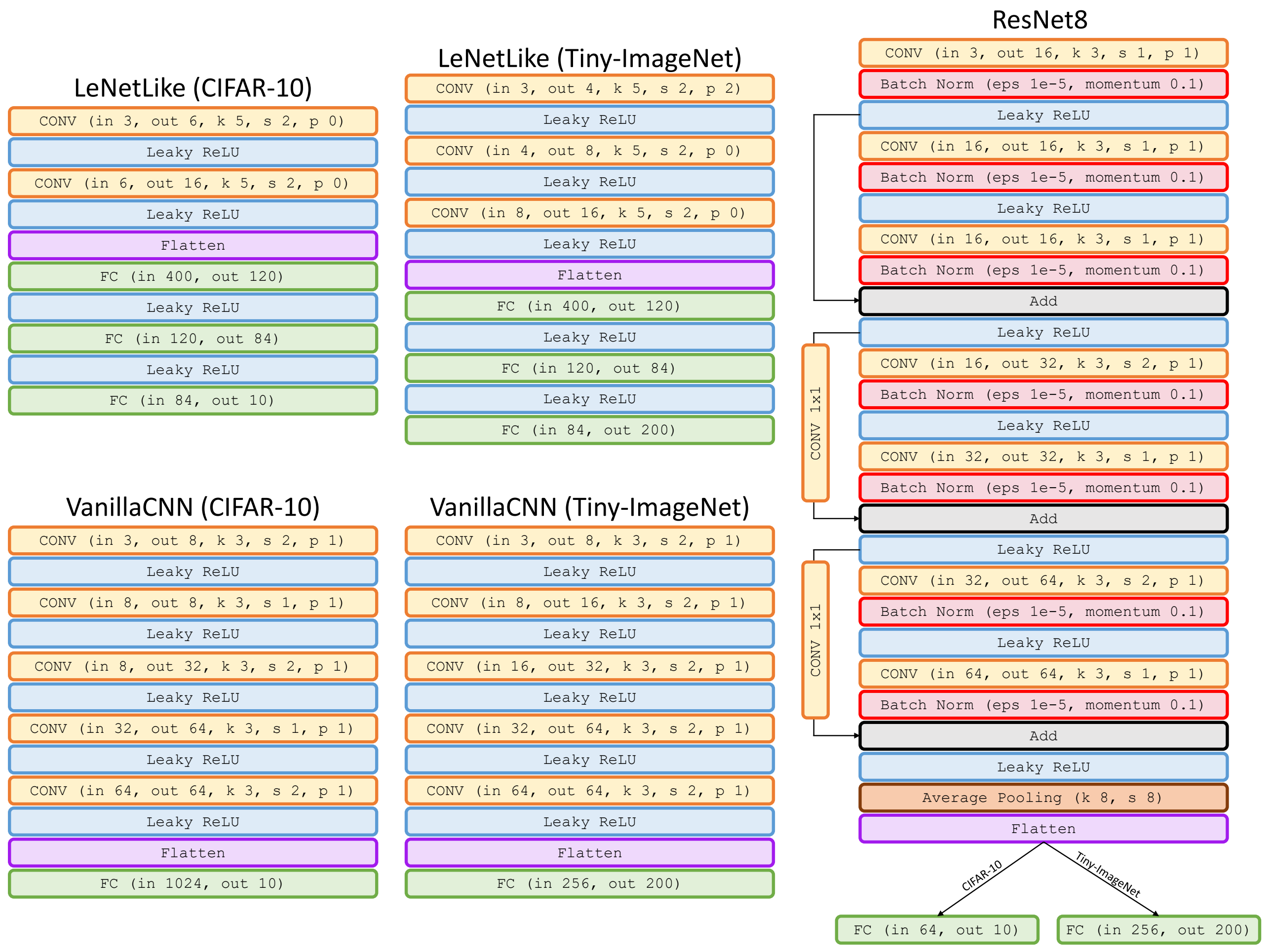

LeNetLike (CIFAR-10)
CONV (in 3, out 6, k 5, s 2, p 0)
Leaky ReLU
CONV (in 6, out 16, k 5, s 2, p 0)
Leaky ReLU
Flatten
FC (in 400, out 120)
Leaky ReLU
FC (in 120, out 84)
Leaky ReLU
FC (in 84, out 10)
LeNetLike (Tiny-ImageNet)
CONV (in 3, out 4, k 5, s 2, p 2)
Leaky ReLU
CONV (in 4, out 8, k 5, s 2, p 0)
Leaky ReLU
CONV (in 8, out 16, k 5, s 2, p 0)
Leaky ReLU
Flatten
FC (in 400, out 120)
Leaky ReLU
FC (in 120, out 84)
Leaky ReLU
FC (in 84, out 200)
VanillaCNN (CIFAR-10)
CONV (in 3, out 8, k 3, s 2, p 1)
Leaky ReLU
CONV (in 8, out 8, k 3, s 1, p 1)
Leaky ReLU
CONV (in 8, out 32, k 3, s 2, p 1)
Leaky ReLU
CONV (in 32, out 64, k 3, s 1, p 1)
Leaky ReLU
CONV (in 64, out 64, k 3, s 2, p 1)
Leaky ReLU
Flatten
FC (in 1024, out 10)
VanillaCNN (Tiny-ImageNet)
CONV (in 3, out 8, k 3, s 2, p 1)
Leaky ReLU
CONV (in 8, out 16, k 3, s 2, p 1)
Leaky ReLU
CONV (in 16, out 32, k 3, s 2, p 1)
Leaky ReLU
CONV (in 32, out 64, k 3, s 2, p 1)
Leaky ReLU
CONV (in 64, out 64, k 3, s 2, p 1)
Leaky ReLU
Flatten
FC (in 256, out 200)
ResNet8
CONV (in 3, out 16, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 16, out 16, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 16, out 16, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Add
Leaky ReLU
CONV (in 16, out 32, k 3, s 2, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 32, out 32, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Add
CONV 1x1
Leaky ReLU
CONV (in 32, out 64, k 3, s 2, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 64, out 64, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Add
CONV 1x1
Leaky ReLU
Average Pooling (k 8, s 8)
Flatten
CIFAR-10
Tiny-ImageNet
FC (in 64, out 10)
FC (in 256, out 200)

FIGURE B.2. **Architectures used in our experiments dealing with image classification.** Top left: LeNetLike, bottom left: VanillaCNN, right: ResNet8.

and ResNet8/32/56 [160]. Each convolutional layer is presented in the form CONV (in I, out O, k K, s S, p P), where I, O, K, S and P represent input channels, number of filters, kernel size, stride and padding, respectively. Fully Connected layers, instead, are reported in the form FC (in I, out O), where I and O stand for input and output units, respectively . Average pooling is shown as Average Pooling (k K, s S), where K is kernel size and S is stride. Finally, CONV 1×1 represents a convolutional layer with kernel size 1 used to adapt feature maps in residual connections.

3D SDF Regression. As far as 3D SDF regression is concerned, we use simple Multilayer Perceptrons (MLPs) composed of a single hidden layer with 256 nodes. Following [22], we use a periodic activation function between the input layer and the hidden layer and between the hidden layer and the output layer. No activation function is applied instead on the final outputs.

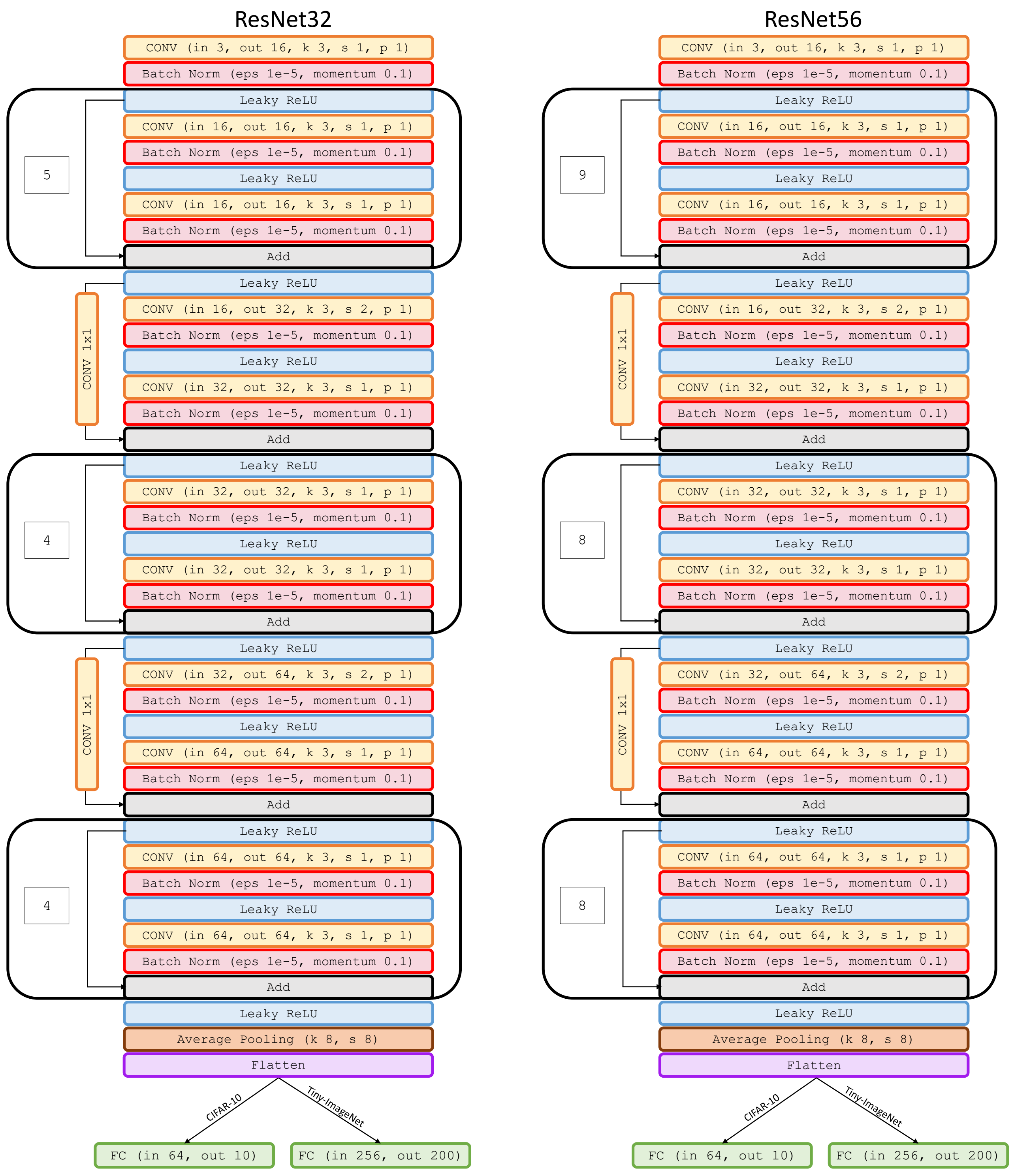

ResNet32
CONV (in 3, out 16, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
5
Leaky ReLU
CONV (in 16, out 16, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 16, out 16, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Add
CONV 1x1
Leaky ReLU
CONV (in 16, out 32, k 3, s 2, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 32, out 32, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Add
4
Leaky ReLU
CONV (in 32, out 32, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 32, out 32, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Add
CONV 1x1
Leaky ReLU
CONV (in 32, out 64, k 3, s 2, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 64, out 64, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Add
4
Leaky ReLU
CONV (in 64, out 64, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 64, out 64, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Add
Leaky ReLU
Average Pooling (k 8, s 8)
Flatten
CIFAR-10
Tiny-ImageNet
FC (in 64, out 10)
FC (in 256, out 200)
ResNet56
CONV (in 3, out 16, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
9
Leaky ReLU
CONV (in 16, out 16, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 16, out 16, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Add
CONV 1x1
Leaky ReLU
CONV (in 16, out 32, k 3, s 2, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 32, out 32, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Add
8
Leaky ReLU
CONV (in 32, out 32, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 32, out 32, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Add
CONV 1x1
Leaky ReLU
CONV (in 32, out 64, k 3, s 2, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 64, out 64, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Add
8
Leaky ReLU
CONV (in 64, out 64, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Leaky ReLU
CONV (in 64, out 64, k 3, s 1, p 1)
Batch Norm (eps 1e-5, momentum 0.1)
Add
Leaky ReLU
Average Pooling (k 8, s 8)
Flatten
CIFAR-10
Tiny-ImageNet
FC (in 64, out 10)
FC (in 256, out 200)

FIGURE B.3. **Architectures used in our experiments dealing with image classification.** Left: ResNet32, right: ResNet56.

B.3 Image Classification: Experiment Details

Images Datasets. To test our general framework in image classification, we make use of the CIFAR-10 [158] and Tiny-ImageNet [157] datasets. CIFAR-10 is composed of 60K 32×32 colour images, 50K for training and 10K for test, categorized in 10 classes. For our experiments we obtain a validation set by splitting the training set in 40K images for training and 10K for validation, while we keep unchanged the test set. Tiny-ImageNet consists of 100K colour images with resolution 64×64, categorized in 200 different classes. We split the training set in 80K images for training and 20K images for validation and use the 10K images of the provided validation set for testing. With both datasets, we follow a standard data augmentation regime [234] and use a batch size of 128.

Nets Datasets. In the Single-Architecture setting, we train the instances in input to NetSpace with the Adam optimizer [231] and constant learning rate set to 0.0001 for a number of epochs that vary from 1 to 600 epochs, obtaining instances with weights and performances varying smoothly across the training iterations. Then we use 100 instances for training, 16 for validation and 16 for test. In the Multi-Architecture setting, we aim at embedding only instances with high performances. Accordingly, we found it more effective to train models with the SGD optimizer for 300 epochs and setting momentum and weight decay to 0.9 and 5e-4, respectively. We set the initial learning rate to 0.05 and decay it by 0.1 at epochs 150, 180 and 210. As discussed in the main paper in Sec. 4, for the Multi-Architecture setting we define a training dataset with 60 LeNetLike instances, 50 VanillaCNN instances, 60 ResNet8 instances and 100 ResNet32 instances, while for the experiment "Sampling of Unseen Architectures" we use a training set composed of 40 LeNetLike instances and 80 ResNet32 instances. In both Multi-Architecture settings, the validation and test sets are composed of 16 instances of the architectures available during training. In Tab. B.1 we report the accuracy achieved by the trained models on the CIFAR-10 and the Tiny-ImageNet test sets, alongside with the number of parameters of each architecture.

Framework Training. To train NetSpace in the Single-Architecture and in the Multi-Architecture settings, we use the Adam optimizer and a learning rate value of 0.0001, we train for around 1K epochs, and then we use the model with the highest performance on the validation set. The temperature term used in $\mathcal{L}_{kd}$ and in $\mathcal{L}^{\gamma}$ is set to 4, while the size of the meta-batch of instances is set to 8 and to 2 in the Single-Architecture and in the Multi-Architecture settings, respectively. The Latent Space Optimization experiments are conducted by training NetSpace with the Adam optimizer and constant learning rate 0.0001, stopping the training when the accuracy achieved by the optimized network doesn't show any additional improvement.

Computational Time and Resources. For our experiments we used several Nvidia RTX 3090 GPUs. Training the networks to populate the datasets requires approximately 600

CIFAR-10

Net	ClassId	Acc. (Adam)	Acc. (SGD)	# Params
LeNetLike	0	-	70.87 %	62,006
VanillaCNN	1	-	79.77 %	68,818
ResNet8	2	82.79 %	87.13 %	78,042
ResNet32	3	-	92.70 %	466,906
ResNet56	-	-	92.85 %	855,770

Tiny-ImageNet

Net	ClassId	Acc. (Adam)	Acc. (SGD)	# Params
LeNetLike	0	-	22.13 %	79,612
VanillaCNN	1	-	31.32 %	112,856
ResNet8	2	40.51 %	44.38 %	128,792
ResNet32	3	-	54.81 %	517,656
ResNet56	-	-	56.93 %	906,520

TABLE B.1. **Models used in our experiments.** We show classification accuracies on the CIFAR-10 (top) and Tiny-ImageNet (bottom) test sets alongside the number of parameters for each architecture. For the Single-Architecture setting (Adam optimizer), we report the accuracy achieved by the best performing network between the trained ones, while for the Multi-Architecture setting (SGD optimizer), we report the average accuracy of the networks that compose the training set.

GPU hours, while training NetSpace with Tiny-ImageNet requires around 84 GPU hours in the Multi-Architecture setting and around 48 GPU hours in the Single-Architecture setting. Training over CIFAR-10, instead, requires less time, with less then 48 and 36 GPU hours respectively in the Multi-Architecture and in the Single-Architecture settings. Finally, the Latent Space Optimization experiments require few GPU hours, but it's possible to observe good improvements over the initial performance already after few minutes of training.

B.4 3D SDF Regression: Experiment Details

3D Shape Dataset. To test NetSpace with networks dealing with 3D SDF regression, we use the ShapeNet dataset [161]. In particular, we use ShapeNetCore, a subset of the full ShapeNet dataset which covers 55 common object categories with about 51,300 unique 3D models. We conduct our experiments on a small subset of ShapeNetCore, consisting of 1000 3D objects from the *chair* category.

MLP Dataset. The MLP dataset used in our experiments contains 1000 MLP. Each of them is obtained by training a randomly initialized MLP to fit the SDF of a single *chair*, whose ground-truth is computed with the code provided with [25]. The fitting procedure consists in 10,000 gradient descent steps: at each step the MLP is queried on 20,000 random 3D coordinates and is asked to regress the SDF value for each of them. Then, the parameters of the MLP are optimized by Adam [231], using as loss function the mean squared error between the predictions and the ground-truth. The learning rate is initially set to 0.0001 and multiplied by 0.9 every 1000 steps.

Framework Training. NetSpace is trained on the 1000 MLPs with the protocol described in Sec. 3 of the main paper, using Adam [231] with learning rate set to 0.0001. During training, we evaluate the performance of NetSpace by comparing directly the predictions of the output MLPs with those of the input MLPs: by querying input and output MLPs with the same random coordinates, we can compute the percentage of predictions of the output MLPs that are sufficiently close to the values predicted by the input MLPs. We monitor this metric and stop the training when it reaches the value 0.8. The results reported in the main paper are obtained by training for 4000 epochs.

Computational Time and Resources. We adopted Nvidia RTX 3090 GPUs also in the experiments involving SDF regression. The creation of the MLP dataset requires approximatively 20 GPU hours, while training NetSpace requires around 48 GPU hours.

B.5 Fusing Batch Norm and Convolutions

To be able to process architectures including batch norm layers, *e.g.*, ResNet in our experiments, without changing the PRep structure, we decided to fuse batch norm layers with convolutional layers, which is always possible for a trained model since batch norm becomes a frozen affine transformation at test time. Therefore, when processing ResNet instances in our experiments, we first prepared a dataset of instances trained with batch norm to achieve the best performances and then, by the process described below, we transformed such instances into equivalent ones featuring only plain convolutional layers without batch norm.

If we consider a feature map F with shape $C \times H \times W$, at inference time its batch normalized version $\widehat{F}$ is obtained by computing at each spatial location i, j:

$$\begin{pmatrix} \widehat{F}_{1,i,j} \\ \widehat{F}_{2,i,j} \\ \vdots \\ \widehat{F}_{C,i,j} \end{pmatrix} = W_{BN} \cdot \begin{pmatrix} F_{1,i,j} \\ F_{2,i,j} \\ \vdots \\ F_{C,i,j} \end{pmatrix} + b_{BN} \tag{B.1}$$

with

$$W_{BN} = \begin{pmatrix} \frac{\gamma_1}{\sqrt{\widehat{\sigma}_1^2+\epsilon}} & & & \\ & \frac{\gamma_2}{\sqrt{\widehat{\sigma}_2^2+\epsilon}} & & \\ & & \ddots & \\ & & & \frac{\gamma_C}{\sqrt{\widehat{\sigma}_C^2+\epsilon}} \end{pmatrix} \tag{B.2}$$

$$b_{BN} = \begin{pmatrix} \beta_1 - \gamma_1 \frac{\widehat{\mu}_1}{\sqrt{\widehat{\sigma}_1^2 + \epsilon}} \\ \beta_2 - \gamma_2 \frac{\widehat{\mu}_2}{\sqrt{\widehat{\sigma}_2^2 + \epsilon}} \\ \vdots \\ \beta_C - \gamma_C \frac{\widehat{\mu}_C}{\sqrt{\widehat{\sigma}_C^2 + \epsilon}} \end{pmatrix} \tag{B.3}$$

where $\widehat{\mu}_c$, $\widehat{\sigma}_c^2$, β_c and γ_c ($c = 1, 2, \ldots, C$) are respectively the mean, variance and batch norm parameters computed during training for the channel c of the feature map. From this formulation, we can see that batch norm can be implemented as a 1×1 convolution and therefore, when batch norm comes after another convolution as in ResNet, we can fuse these two convolutions into a single one.

We can express a convolutional layer with kernel size k processing the $C_{prev} \times k \times k$ volume at the spatial location (i, j) of a feature map F_{prev} with C_{prev} channels to produce the feature map $\tilde{F}$ with C output channels as an affine transfomation

$$\tilde{f}_{i,j} = W_{conv}\, f_{i,j} + b_{conv}, \tag{B.4}$$

where $W_{conv} \in \mathbb{R}^{C \times (C_{prev}\, k^2)}$, $b_{conv} \in \mathbb{R}^C$ and $f_{i,j}$ represents the area of size $C_{prev} \times k \times k$ around cell (i, j) reshaped as a $(C_{prev}\, k^2)$-dimensional vector.

If the batch norm defined by $W_{BN} \in \mathbb{R}^{C \times C}$ and $b_{BN} \in \mathbb{R}^C$ presented in Eq. (B.2) and Eq. (B.3) comes after such convolutional layer, the normalized values $\widehat{f}_{i,j} \in \mathbb{R}^C$ at cell (i, j) of its output feature map can be computed as

$$\begin{aligned} \widehat{f}_{i,j} &= W_{BN}\, \tilde{f}_{i,j} + b_{BN} \\ &= W_{BN}\, (W_{conv}\, f_{i,j} + b_{conv}) + b_{BN}. \end{aligned} \tag{B.5}$$

Hence, it is possible to replace every convolutional layer (with weights W_{conv} and b_{conv}) followed by a batch norm layer (whose weights can be shaped in W_{BN} and b_{BN} as described above) with a single convolutional layer whose parameters W and b can be computed as:

$$W = W_{BN}\, W_{conv} \tag{B.6}$$

$$b = W_{BN}\, b_{conv} + b_{BN} \tag{B.7}$$

B.6 Visualizing Networks as Images

As in our framework network instances are represented as PRep tensors, they can be visualized as images. Thus, in this section we highlight some properties of NetSpace by visualizing input and output instances as images. In particular, following the ResNet8 Single-Architecture (Image classification) and ResNet32 Multi-Architecture trainings, we take some instances from the test set and obtain their PReps along with those of the corresponding instances predicted by NetSpace. Such representations are 2D matrices, that we reshaped to obtain images of form factors amenable to clear visualization. Then, as shown in Fig. B.4 and Fig. B.5, we represent parameters according to a standard colormap.

From these results we can highlight some interesting properties about our framework. Firstly, we observe that PReps predicted by NetSpace are significantly different *w.r.t.* the input ones: as we did not use any reconstruction loss in the learning objective, NetSpace learnt to predict instances which behave like the input ones but that are different in terms of parameter values. Secondly, we can see that PReps corresponding to different instances can indeed be visualized as different images, which suggests that, perhaps, in future work these images may be used as proxies for neural networks instances, so that training or fine-tuning or distillation may be realized by learning to generate images.

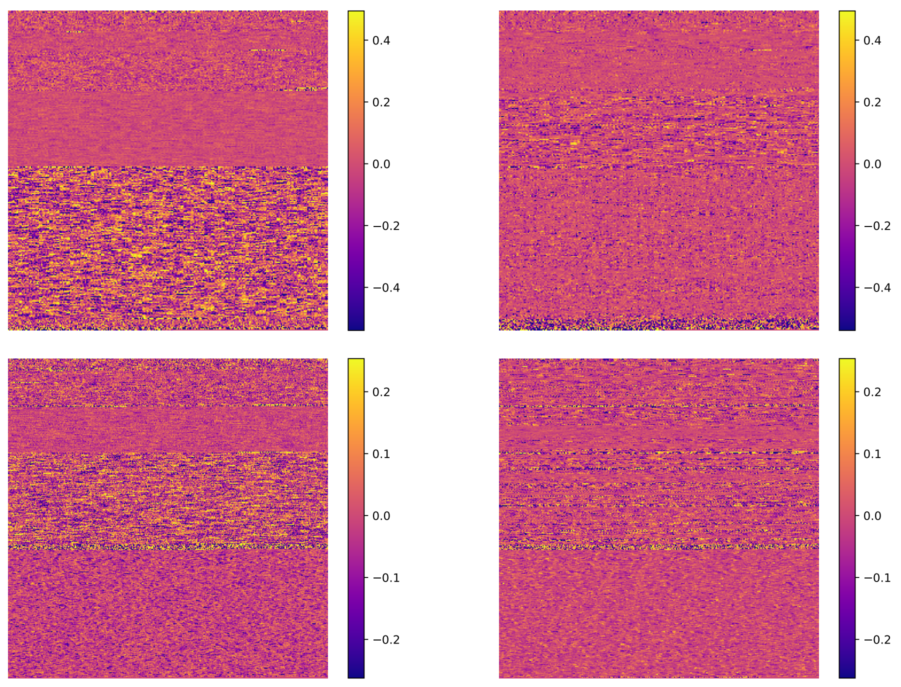


FIGURE B.4. **Networks as images.** Visualization of the PRep of a ResNet8 instance in the Single-Architecture setting (Image classification): top, CIFAR-10, bottom, Tiny-ImageNet. The target PRep on the left is given in input to NetSpace, that produces the predicted PRep on the right.

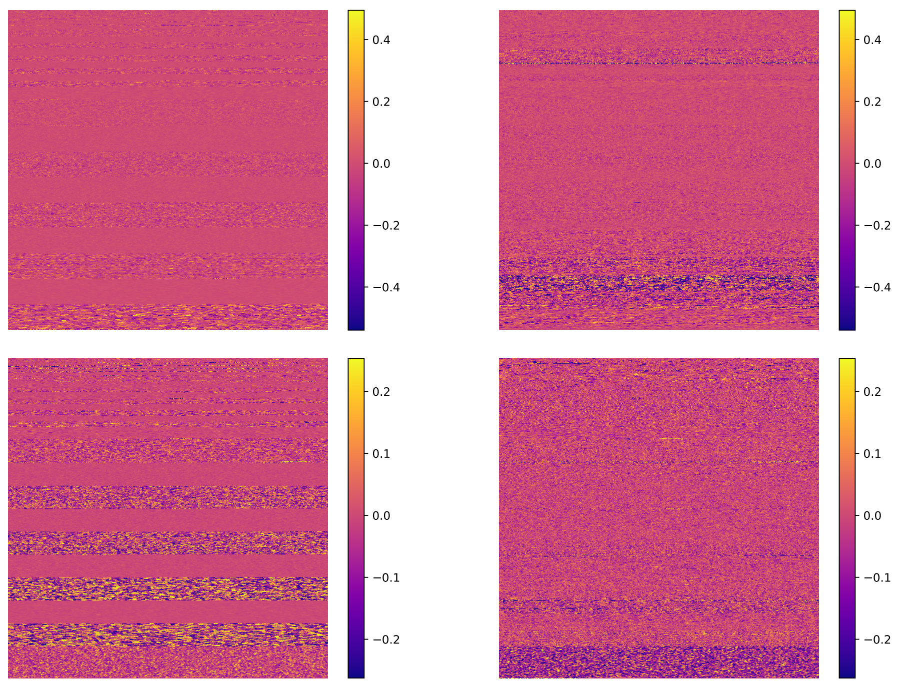


FIGURE B.5. **Networks as images.** Visualization of the PRep of a ResNet32 instance in the Multi-Architecture setting: top, CIFAR-10, bottom, Tiny-ImageNet. The target PRep on the left is given in input to NetSpace, that produces the predicted PRep on the right.

Appendix C

Deep Learning on Implicit Neural Representations of Shapes

C.1 Obtaining INRs from 3D Discrete Representations

In this section, we detail the procedure used when fitting INRs to create the datasets used in this work. Given a dataset of 3D shapes we train a set of the same number of MLPs, fitting each one on a single 3D shape. Every MLP is thus trained to approximate a continuous function that describes the represented shape, the nature of the function being chosen according to the discrete representation in which the shape is provided. We adopt MLPs with multiple hidden layers of the same dimension as done in [22, 34, 166, 167, 168], interleaved by the sine activation function, as proposed in [22], to enhance the capability of the MLPs to fit the high frequency details of the input signal.

In its general formulation, an INR can be used to fit a continuous function $f : \mathbb{R}^{in} \rightarrow \mathbb{R}^{out}$. To do so, a training set composed of N points $\mathbf{x}_i \in \mathbb{R}^{in}$ with $i = 1, 2, ..., N$, paired with values $\mathbf{y}_i = f(\mathbf{x}_i) \in \mathbb{R}^{out}$, is exploited to find the optimal parameters θ^* for the MLP that implements the INR, by solving the optimization problem:

$$\theta^* = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \ell(\mathbf{y}_i, f_\theta(\mathbf{x}_i)), \tag{C.1}$$

where f_θ represents the function f approximated by the MLP with parameters θ and ℓ is a loss function that computes the error between predicted and ground-truth values.

The output value $f_\theta(\mathbf{x}_i)$ is computed as a series of linear transformations, each one followed by a non-linear activation function (*i.e.*, the sine function in our case), except the last one. Considering a MLP m, the mapping between its layers $L-1$ and L consists in a linear transformation that maps the values $\mathbf{h}_m^{L-1} \in \mathbb{R}^{D_{L-1}}$ from the layer $L-1$ into the values $\mathbf{h}_m^L = \phi(\mathbf{W}_m^L \mathbf{h}_m^{L-1} + \mathbf{b}_m^L) \in \mathbb{R}^{D_L}$ of the layer L, with $\mathbf{W}_m^L$ being the matrix of weights $\in \mathbb{R}^{D_L \times D_{L-1}}$, $\mathbf{b}_m^L$ being the biases vector $\in \mathbb{R}^{D_L}$, and $\phi(\cdot)$ the non-linearity [22]. If we now consider M MLPs used to fit M different INRs and the mapping between the layers

$L-1$ and L, we can easily compute such mapping simultaneously for all the MLPs on modern GPUs thanks to tensor programming frameworks. The mapping consists indeed in a straightforward tensor contraction operation, where the values $\mathbf{h}_{L-1} \in \mathbb{R}^{M \times D_{L-1}}$ of the layer $L-1$ are mapped to the values $\mathbf{h}_L = \mathbf{W}_L \mathbf{h}_{L-1} + \mathbf{b}_L \in \mathbb{R}^{M \times D_L}$ of the layer L, with $\mathbf{W}^L \in \mathbb{R}^{M \times D_L \times D_{L-1}}$ and $\mathbf{b}^L \in \mathbb{R}^{M \times D_L}$. Extending this formulation to all the layers of the chosen MLP architecture allows to fit multiple INRs in parallel.

In the following, we describe how we train MLPs to obtain INRs starting from point clouds, triangle meshes and voxel grids.

Point clouds. The INR for a 3D shape represented by a point cloud $\mathcal{P}$ encodes the *unsigned distance function* (udf) of the point cloud $\mathcal{P}$. Given a point $\mathbf{p} \in \mathbb{R}^3$, the value $udf(\mathbf{p})$ is defined as $\min_{q \in \mathcal{P}} \|\mathbf{p} - \mathbf{q}\|_2$, *i.e.*, the euclidean distance from $\mathbf{p}$ to the closest point $\mathbf{q}$ of the point cloud. After preparing a training set of N points $\mathbf{x}_i \in \mathbb{R}^3$ with $i = 1, 2, ..., N$, coupled with their udf values $y_i \in \mathbb{R}$, the INR of the underlying 3D shape is obtained by training a MLP to regress correctly the udf values, with the learning objective:

$$\mathcal{L}_{mse} = \frac{1}{N} \sum_{i=1}^{N} (y_i - f_\theta(\mathbf{x}_i))^2, \tag{C.2}$$

that consists in the mean squared error between ground-truth values y_i and the predictions by the MLP $f_\theta(\mathbf{x}_i)$. An alternative objective is converting the udf values y_i into values y_i^{bce} continuously spanned in the range $[0, 1]$, with 0 and 1 representing respectively the predefined maximum distance from the surface and the surface level set (*i.e.*, distance equal to zero). Then, the MLP optimizes the binary cross entropy between such labels and the predicted values, defined as:

$$\mathcal{L}_{bce} = -\frac{1}{N} \sum_{i=1}^{N} y_i^{bce} log(\hat{y}_i) + (1 - y_i^{bce}) log(1 - \hat{y}_i), \tag{C.3}$$

where $\hat{y}_i = \sigma(f_\theta(\mathbf{x}_i))$, with σ representing the sigmoid function. In our experiments, we found empirically that this second learning objective leads to faster convergence and more accurate INRs, and we decided to adopt this formulation when producing INRs from point clouds.

Triangle meshes. Triangle meshes are usually adopted to represent closed surfaces. This provides an additional information compared to the point clouds case, since the 3D space can be divided into the portion contained *inside* and *outside* the closed surface. Thus, the INR of a closed 3D surface represented by a triangle mesh can be obtained by fitting the *signed distance function* (sdf) to the surface defined by the mesh. Given a point $\mathbf{p} \in \mathbb{R}^3$, the value $sdf(\mathbf{p})$ is defined as the euclidean distance from $\mathbf{p}$ to the closest point of the surface, with positive sign if $\mathbf{p}$ is outside the shape and negative sign otherwise. Similarly to the point clouds case, an

INR for a 3D shape represented by a triangle mesh can be obtained by pursuing the learning objective presented in Eq. (C.2), using a training set composed of 3D points paired with their sdf values. However, it possible to adopt a learning objective based on the binary cross entropy loss reported in Eq. (C.3) also for triangle meshes, and we empirically observed the same benefits. Hence, we adopt it also when fitting INRs on meshes. In this case, the sdf values y_i are converted into values $y_i^{bce} \in [0, 1]$, with 0 and 1 representing respectively the predefined maximum distance *inside* and *outside* the shape, *i.e.*, 0.5 represents the surface level set.

Voxel grids. A voxel grid is a 3D grid of V^3 cubes marked with label 1, if the cube is occupied, and label 0 otherwise. In order to fit an INR on voxels, it is possible to learn to regress the *occupancy function* (occ) of the grid itself. The training set, in this case, contains V^3 3D points that corresponds to the centroids of the cubes that compose the voxel grid. Being each of such points $\mathbf{x}_i$ associated to a 0-1 label y_i, it is straightforward to use a binary classification objective to train the MLP that implement the desired INR. More specifically, we adopt the learning objective defined as:

$$\mathcal{L}_{focal} = -\frac{1}{N}\sum_{i=1}^{N} \alpha(1-\hat{y}_i)^{\gamma} y_i log(\hat{y}_i) + (1-\alpha)\hat{y}_i^{\gamma}(1-y_i) log(1-\hat{y}_i), \tag{C.4}$$

where $\hat{y}_i = \sigma(f_\theta(\mathbf{x}_i))$, while α and γ are respectively the balancing parameter and the focusing parameter of the focal loss proposed in [235]. We deploy a focal loss to alleviate the imbalance between the number of occupied and empty voxels.

C.2 Reconstructing Discrete Representations from INRs

In this section we discuss how it is possible to sample 3D discrete representations from INRs, which could be necessary to process the underlying shapes with algorithms that need an explicit surface (*e.g.*, Computational Fluid Dynamics [236, 237, 238]) or simply for visualization purposes.

Point clouds from udf. To sample a dense point cloud from an INR fitted on its udf, we use a slightly modified version of the algorithm proposed in [26]. The basic idea is to query the udf with points scattered all over the considered portion of the 3D space, *projecting* such points onto the isosurface according to the predicted udf values. In order to do that, let us define f_θ as the udf approximated by the INR with parameters θ. Given a point $\mathbf{p} \in \mathbb{R}^3$, it can be projected onto the isosurface by computing its updated position $\mathbf{p_s}$ as:

$$\mathbf{p_s} = \mathbf{p} - f_\theta(\mathbf{p}) \cdot \frac{\nabla_p f_\theta(\mathbf{p})}{\|\nabla_p f_\theta(\mathbf{p})\|}. \tag{C.5}$$

This can be intuitively understood by considering that the negative gradient of the udf indicates the direction of maximum decrease of the distance from the surface, pointing towards the closest point on it. Eq. (C.5), thus, can be interpreted as moving $\mathbf{p}$ along the direction of maximum decrease of the udf of a quantity defined by the value of the udf itself in $\mathbf{p}$, reaching the point $\mathbf{p}_s$ on the surface. One must consider, though, that f_θ is only an approximation of the real udf, which leads to two considerations. On a first note, the gradient of f_θ must be normalized (as done in Eq. (C.5)), while the gradient of the real udf has norm equal to 1 everywhere except on the surface. Secondly, the predicted udf value can be imprecise, implying that $\mathbf{p}$ can still be distant from the surface after moving it according Eq. (C.5). To address the second issue, the 3D position of p_s is refined repeating the update described in Eq. (C.5) several times. Indeed, after each update, the point gets closer and closer to the surface, where the values approximated by f_θ are more accurate, implying that the last updates should successfully place the point on the isosurface. Given an INR fitted on the udf of a point cloud, the overall algorithm to sample a dense point cloud from it is composed of the following steps. Firstly, we prepare a set of points scattered uniformly in the considered portion of the 3D space and we predict their udf value with the given INR. Then we filter out points whose predicted udf is greater than a fixed threshold (0.05 in our experiments). For the remaining points, we update their coordinates iteratively with Eq. (C.5) (we found 5 updates to be enough). Finally, we repeat the whole procedure until the reconstructed point cloud counts the desired number of points.

Triangle meshes from sdf. An INR fitted on the sdf computed from a triangle mesh allows to reconstruct the mesh by means of the Marching Cubes algorithm [211]. We refer the reader to the original paper for a detailed description of the method, but we report here a short presentation of the main steps, for the sake of completeness. Marching Cubes explores the considered 3D space by querying the sdf with 8 locations at a time, that are the vertices of an arbitrarily small imaginary cube. The whole procedure involves *marching* from one cube to the other, until the whole desired portion of the 3D space has been covered. For each cube, the algorithm determines the triangles needed to model the portion of the isosurface that passes through it. Then, the triangles defined for all the cubes are fused together to obtain the reconstructed surface. In order to determine how many triangles are needed for a single cube and how to place them, for each pair of neighbouring vertices of the cube, their sdf values are computed and one triangle vertex is placed between them if such values have opposite sign. Considering that the number of possible combinations of the sdf signs at the cube vertices is limited, it is possible to build a look-up table to retrieve the triangles configuration for the cube starting from the sdf signs at its eight vertices, combined in a 8-bit integer and used as key for the look-up table. After the triangles configuration for a cube has been retrieved, the vertices of the triangles are placed on the edges connecting the cube vertices, computing their exact position by linearly interpolating the two sdf values that are connected by each edge.

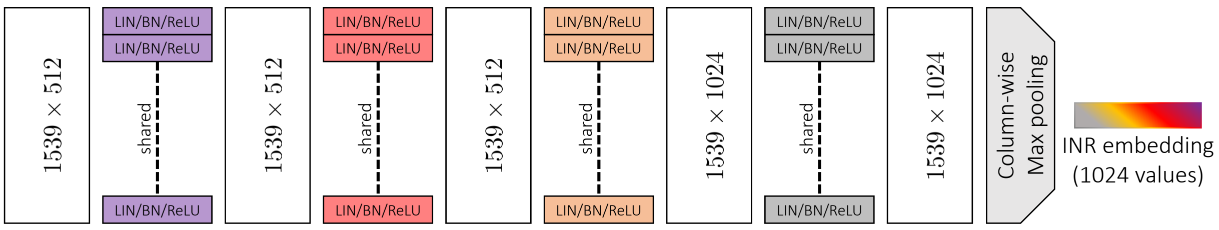


FIGURE C.1. **inr2vec encoder.** With a series of linear transformations and a final column-wise max pooling, the encoder maps the input weights matrix into a compact embedding. LIN/BN/ReLU stands for a linear transformation, followed by batch normalization and ReLU activation function.

Voxel grids from *occ***.** In order to reconstruct voxel grids from INRs, we adopt a straightforward procedure. Each INR has been trained to predict the probability of a certain voxel to be occupied, when queried with the 3D coordinates of the voxel centroid. Thus, a first step to reconstruct the fitted voxels consists in creating a grid of the desired resolution V. Then, the INR is queried with the V^3 centroids of the grid and predicts an occupancy probability for each of them. Finally, we consider as occupied only voxels whose predicted probability is greater than a fixed threshold, which we set to 0.4, as we found empirically that it allows for a good trade-off between scattered and over-filled reconstructions.

C.3 inr2vec Encoder and Decoder Architectures

In this section, we describe the architecture of inr2vec encoder, along with the one of the implicit decoder used to train it (see Sec. 5.3).

Encoder. inr2vec encoder, detailed in Fig. C.1, consists in a series of linear transformations, that maps the input INR weights into features with higher dimensionality, before applying max pooling to obtain a compact embedding. More specifically, the input weights are rearranged in a matrix with shape $L(H+1) \times H$, where H is the number of nodes in the hidden layers of the MLP that implements the input INR and L is the number of linear transformations between such hidden layers (*i.e.*, the MLP has $L+1$ hidden layers). The matrix is obtained by stacking L matrices (one for each linear transformation), each one with shape $(H+1) \times H$, being composed of a matrix of weights with shape $H \times H$ and a row of H biases. In our setting, each MLP has 4 hidden layers with 512 nodes: the final matrix in input to inr2vec encoder has shape $3 \cdot (512+1) \times 512 = 1539 \times 512$. In the current implementation, the four linear mappings of the encoder transform each row of the input matrix into features with size 512, 512, 1024 and 1024, obtaining, at each step, features matrices with shape 1539×512, 1539×512, 1539×1024 and 1539×1024. Finally, the encoder applies column-wise max pooling to compress the final matrix into a single compact embedding composed of 1024 values. Between the linear mappings of the encoder, we adopt 1D batch normalization and ReLU activation functions.

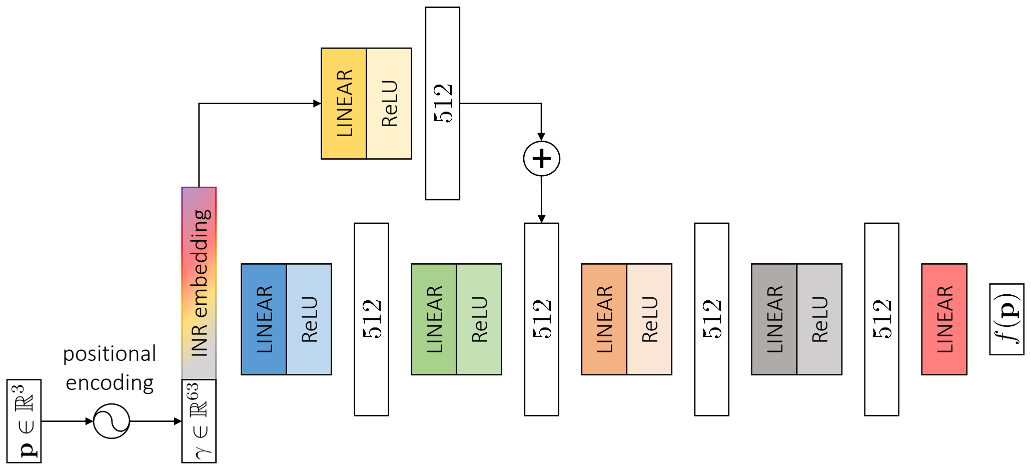


FIGURE C.2. **inr2vec decoder.** Our framework is trained with an implicit decoder, that maps an INR embedding concatenated with a 3D query into the value of the implicit function at the query coordinates.

Decoder. The implicit decoder that we adopt to train inr2vec is presented in Fig. C.2. We designed it taking inspiration from [25], since we need a decoder capable of reproducing the implicit function of input INR when conditioned on the embedding obtained by the encoder. Thus, the decoder takes in input the concatenation of the INR embedding with the coordinates of a given 3D query. We adopt the positional encoding proposed in [24] to embed the input 3D coordinates into a higher dimensional space to enhance the capability of the decoder to capture the high frequency variations of the input data. The query 3D coordinates are mapped into 63 values that, concatenated with the 1024 values that compose the INR embedding, result in a vector with 1087 values as input for inr2vec decoder. Internally, the implicit decoder is composed of 4 hidden layers with 512 nodes and of a skip connection that projects the input 1087 values into a vector of 512 elements, that are summed to the features of the second hidden layer before being fed to the transformation that bridges the second and the third hidden layers. Finally, the features of the last hidden layer are mapped to a single output, which is compared to the ground-truth associated with the input 3D query to compute the loss. Each linear transformation of the decoder, except the output one, is paired with the ReLU activation function.

C.4 Motivation Behind inr2vec Encoder Design

We designed inr2vec encoder with the goal of obtaining a good scalability in terms of memory occupation. Indeed, a naive solution to process the weights of an input INR would consist in an MLP encoder mapping the flattened vector of weights to the embedding of the desired dimension. However, such approach would require a huge amount of memory resources, since an input INR of 4 layers of 512 neurons would have approximately 800K

INR hidden dim.	INR #layers	INR #params	#params inr2vec encoder	#params MLP encoder
512	4	∼800K	∼3M	∼800M
512	8	∼2M	∼3M	∼2B
512	12	∼3M	∼3M	∼3B
512	16	∼4M	∼3M	∼4B
1024	4	∼3M	∼3.5M	∼3B
1024	8	∼7M	∼3.5M	∼7.5B
1024	12	∼11M	∼3.5M	∼12B
1024	16	∼15M	∼3.5M	∼16B

TABLE C.1. **Number of parameters of inr2vec encoder.** Comparison between the number of parameters of inr2vec encoder and the number of parameters of a generic MLP encoder.

parameters. Thus, an MLP encoder going from 800K parameters to an embedding space of size 1024 would already have a totality of ∼800M parameters. We report in Tab. C.1 a detailed analysis of the parameters of our encoder *w.r.t.* the ones of an MLP encoder by varying the input INR dimension. As we can notice the MLP encoder does not scale well, making this kind of approach very expensive in practice, while inr2vec encoder scales gracefully to bigger input INRs.

C.5 Experimental Settings

We report here a detailed description of the settings adopted in our experiments.

INRs fitting. In every experiment, we fit INRs on 3D discrete representations using MLPs having 4 hidden layers with 512 nodes each. We implement MLPs using sine as a periodic activation function, as proposed in [22]. The procedure adopted to fit a single MLP consists in querying it with 3D points sampled properly in the space surrounding the underlying shape. The MLP predicts a value for each query and it's trained by computing a loss function between the predicted value and the ground-truth value of the fitted implicit function (*i.e.*, udf for point clouds, sdf for meshes and occ for voxels). The set of training queries is prepared according to different strategies, depending on the nature of the discrete representation being fitted. For voxel grids, the set of possible queries consists of the 3D coordinates of all the centroids of the grid. For point clouds and meshes, instead, queries are sampled with different densities in the volume containing the fitted shape: indeed, for each shape, we prepare 500K queries by taking 250K points close to the surface, 200K points at a medium-far distance for the surface, 25K far from the surface and other 25K scattered uniformly in the volume. The queries coordinates are computed by adding gaussian noise to the points of the fitted point cloud or to points sampled uniformly from the fitted mesh surface. More precisely, close queries are computed with noise sampled from the normal distribution $\mathcal{N}(0, 0.001)$, medium-far queries with noise from $\mathcal{N}(0, 0.01)$, far queries with noise from $\mathcal{N}(0, 0.1)$. The uniformly scattered queries are just computed by sampling each of their coordinates from the uniform distribution $\mathcal{U}(-1, 1)$, being the considered shapes normalized in such volume. As for the ground-truth values, for voxels they consist simply in the occupied/empty label of the voxel associated to the query. For point clouds, for each query we compute its udf value by building a KDTree on the fitted point cloud and looking for the closest point to the considered query (we used the Pytorch3D [131] implementation of the KDTree algorithm). For meshes, finally, we compute the sdf of queries with the functions provided in the Open3D library [239][1]. For each of the considered modalities, at each step of the fitting procedure, we randomly sample 10K pairs of queries/ground-truth values from the precomputed ones, performing a total of 500 steps for each shape. Thanks to the procedure detailed in Appendix C.1, we are able to fit up to 16 multiple MLPs in parallel, using Adam optimizer [231] with learning rate set to 1e-4. On a final note, we fixed the weights initialization of the MLPs to be always the same, as we observed empirically this to be key to convergence of inr2vec. This choice poses no limitation to the practical use of our framework and has also been adopted in recent works [33, 34].

inr2vec training. According to what is described in Sec. 5.3, during training our framework takes in input the weights of a given INR and is asked to reproduce the implicit function fitted by the INR on a set of predefined 3D queries. Such queries are prepared with the same strategies described in the previous paragraph and, similarly to what is done while fitting INRs, at each step the training loss for inr2vec is computed on 10K queries randomly sampled from a set of precomputed ones. In every experiment, we train inr2vec with AdamW optimizer [240], learning rate 1e-4 and weight decay 1e-2 for 300 epochs, one epoch corresponding to processing all the INRs that compose the considered dataset, processing at each training step a mini-batch of 16 INRs. During training, we select the best model by evaluating its reconstruction capability on a validation set of INRs. When training on INRs obtained from point clouds, we compare the ground-truth set of points with the ones reconstructed by inr2vec decoder. For voxels, we compare the input and the output grid by comparing the point clouds composed by the centroids corresponding to occupied voxels. As for what concerns meshes, we compare the clouds containing input and output vertices. In all cases, the reconstruction quality is evaluated by computing the Chamfer Distance between ground-truth and output point clouds, as defined in [214]. See Appendix C.2 of this document for details on how to sample discrete 3D representations from the implicit functions fitted by INRs and that inr2vec is trained to reproduce.

Shape classification. The classifier that we deploy on inr2vec embeddings is composed of three linear transformations, mapping sequentially the input embedding with 1024 features to vectors of size 512, 256 and, finally, to a vector with a number of values corresponding to the number of classes of the considered dataset. The final vector is then transformed to a probability distribution with the softmax function. We use 1D batch normalization and the ReLU activation function between the classifier linear transformations. In all experiments,

[1] `http://www.open3d.org/docs/latest/tutorial/geometry/distance_queries.html`

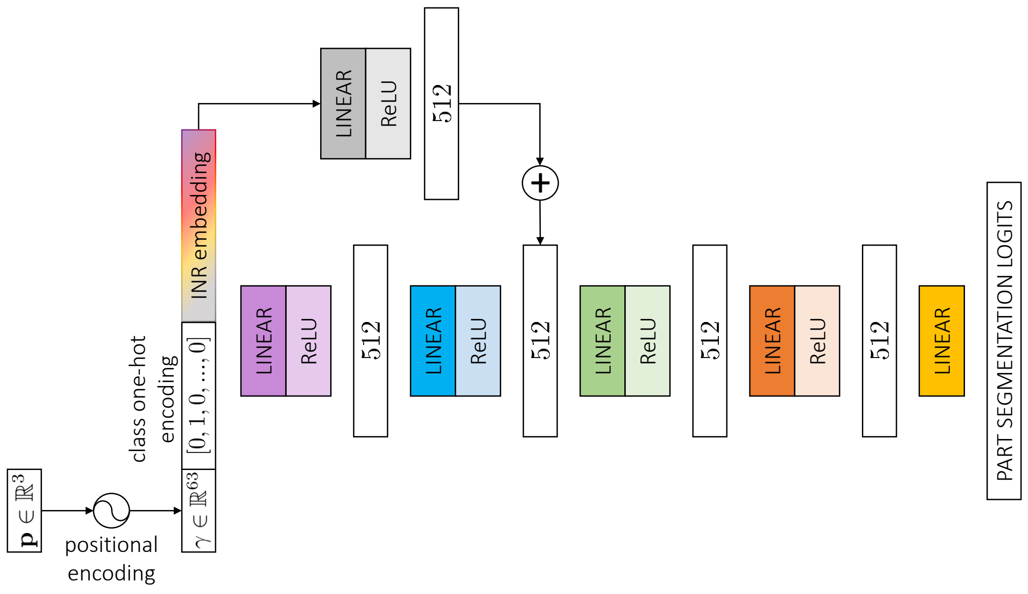


FIGURE C.3. **Part segmentation decoder.** We train a decoder to predict the part segmentation label of a given 3D query when conditioned on the embedding of the input INR and on the one-hot encoding of the INR class.

the classifier is trained for 150 epochs, with AdamW optimizer [240] and weight decay 1e-2. The learning rate is scheduled according to the OneCycle strategy [241], with maximum learning rate set to 1e-4. At each training step, the classifier processes a mini-batch counting 256 embeddings. During training, we select the best model by computing the classification accuracy on a validation set of embeddings. The best model is used after training to compute the classification accuracy on the test set, obtaining the numbers reported in the tables.

Point cloud part segmentation. In order to tackle point cloud part segmentation starting from inr2vec embeddings, we adopt a decoder similar to the one that we use for reconstruction during inr2vec training. The part segmentation decoder, depicted in Fig. C.3, is fed with the positional encoding of a 3D query together with the embedding of an input INR and predicts a K-dimensional vector of segmentation logits for the given query, with K representing the total number of parts of all the C available categories. Moreover, as done in previous work [16, 162, 122], we concatenate an additional C-dimensional vector to the input of the part segmentation decoder, conditioning the output of our decoder with the one-hot encoding of the input INR class. We conduct our experiments on the ShapeNet Part Segmentation dataset [216], that presents 16 categories labeled with two to five parts, for a total of 50 parts (*i.e.*, C=16 and K=50). According to a standard protocol [16, 162, 122], during training we compute the cross-entropy loss function on all the K logits predicted by our decoder, while, at test time, the final prediction is obtained considering only the subset of parts belonging to the specific class of the input INR. The part segmentation decoder is trained with the original point clouds available in the ShapeNet Part Segmentation dataset, where part labels are provided for each point of each cloud. At test time, though, we test both our decoder and the considered competitors on the point clouds reconstructed from the input INRs, since we want to simulate the scenario of 3D shapes being available exclusively in the form of INRs. Thus, the protocol to obtain a segmented point cloud starting from an input INR consists in reconstructing the cloud first (see Appendix C.2) and then in assigning a part label to each point of the reconstructed shape with our part segmentation decoder. When ground-truth labels are required to compute quantitative results, we obtain them by comparing the reconstructed cloud with the original one and assigning to each point of the reconstructed shape the part label of the closest point in the original cloud. Our part segmentation decoder is trained for 250 epochs with AdamW optimizer, OneCycle learning rate scheduling with maximum value set to 1e-4, weight decay equal to 1e-2 and mini-batches composed of 256 embeddings, each one paired with 3D queries from the original point clouds during training and from the ones reconstructed from the input INRs at test time. During training, we compute the class mIoU on the validation split and save the best model in order to compute the final metrics on the test set.

Shape generation. We perform unconditional shape generation by training Latent-GAN [217] to generate embeddings indistinguishable from the ones produced by inr2vec on a given dataset. This approach allows us to train a shape generation framework with the very same architecture to generate embeddings representing INRs with different underlying implicit functions, such as udf for the point clouds of ShapeNet10 and sdf for the models of cars provided by [27]. We conducted our experiments using the official implementation[2], setting all the hyperparameters to default. The generator network is implemented as a fully connected network with two layers and ReLU non linearity, that map an input noise vector with 128 values sampled from the normal distribution $\mathcal{N}(0, 0.2)$ to an intermediate hidden vector of the same dimension and then to the predicted embedding with 1024 values (we removed the final ReLU present in the original implementation). The discriminator is also a fully connected network, with three layers and ReLU non linearity. The first layer maps the embedding produced by the generator to a hidden vector with 256 values, which are then transformed by the second layer into a hidden vector with 512 values, that are finally used by the third layer, together with the sigmoid function, to predict the final score. According to the original implementation, we trained one separate Latent-GAN for each class of the considered datasets, using the Wasserstein objective with gradient penalty proposed in [242] and training each model for 2000 epochs.

Learning a mapping between inr2vec embedding spaces. The transfer function between inr2vec embedding spaces is implemented as a simple fully connected network, with 8 linear layers interleaved by 1D batch norm and ReLU activation functions. All the hidden features produced by the linear transformations present the same dimension of the input embedding, *i.e.*, 1024 values. The final linear layer predicts the output embedding, which is compared

[2]`https://github.com/optas/latent_3d_points`

with the target one with a standard L2 loss. We train the transfer network with AdamW optimizer, constant learning rate and weight decay both set to 1e-4, stopping the training upon convergence, which we measure by comparing the shapes reconstructed by the predicted embeddings with the ground-truth ones on a predetermined validation split. Such validation metrics are used also to save the best model during training, which is finally evaluated on the test set.

C.6 Implementation, Hardware and Timings

We implemented our framework with the PyTorch library, performing all the experiments on a single NVIDIA 3090 RTX GPU. We created an augmented version of each considered dataset, in order to obtain roughly ∼100K INRs, whose fitting requires around 4 days in the current implementation. Training inr2vec requires another 48 hours, while all the networks adopted to perform the downstream tasks on inr2vec embeddings can be trained in few hours.

C.7 Testing on Original Discrete 3D Representations

In the experiments "Shape classification" and "Point cloud part segmentation", we evaluated the competitors on the 3D discrete representations reconstructed from the INRs fitted on the test sets of the considered datasets, since these would be the only data available at test time in a scenario where INRs are used to store and communicate 3D data. For completeness, we report here the scores achieved by the baselines when tested on the original discrete representations, without reconstructing them from the input INRs. Such results are presented in Tab. C.2 for shape classification and in Tab. C.3 for part segmentation. We report in the tables also the results obtained with our framework: they are the same reported in Tab. 5.2 for what concerns shape classification, since our classifier processes exclusively inr2vec embeddings, while they are different for part segmentation, as we use as query points for our segmentation decoder those from the discrete point clouds reconstructed from input INRs in Tab. 5.3 while we use those from the original point clouds in the experiment reported here, as done for the competitors. The results reported in the tables show limited differences, either positive or negative, with the ones presented in Sec. 5.5, mostly within the range of variations due to the inherent stochasticity of training. There are few larger differences, like DGCNN on ModelNet40 (+1.7 when tested on the original discrete representations) or on ScanNet (-1.1 when tested on the original discrete representations), whose difference in sign however suggests neither of the two settings is clearly superior to the other.

	Point Cloud			Mesh	Voxels
Method	ModelNet40	ShapeNet10	ScanNet10	Manifold40	ShapeNet10
PointNet [16]	88.8	94.7	72.8	–	–
PointNet++ [162]	91.0	95.2	76.3	–	–
DGCNN [122]	91.6	94.0	75.1	–	–
MeshWalker [202]	–	–	–	90.6	–
Conv3DNet [164]	–	–	–	–	92.5
inr2vec	87.0	93.3	72.1	86.3	93.0

TABLE C.2. **Shape classification results.** We report here shape classification results when testing on the original discrete representations of the test sets instead of reconstructing them from the input INRs.

Method	instance mIoU	class mIoU	airplane	bag	cap	car	chair	earphone	guitar	knife	lamp	laptop	motor	mug	pistol	rocket	skateboard	table
PointNet [16]	83.0	78.8	80.5	77.9	78.3	74.4	89.0	68.3	90.1	82.2	80.7	94.7	63.1	91.7	79.3	58.2	72.7	81.0
PointNet++ [162]	84.4	82.8	81.7	86.5	85.2	78.6	90.2	77.9	91.2	84.4	83.2	95.4	72.0	94.6	83.3	64.2	75.6	80.9
DGCNN [122]	84.3	81.4	81.6	82.2	80.9	75.7	90.7	80.9	90.2	86.9	82.6	94.8	64.8	92.8	81.0	60.6	74.7	81.8
inr2vec	80.5	71.1	79.5	72.9	72.3	70.7	87.4	64.1	89.4	81.6	76.5	94.5	59.3	92.4	78.4	53.5	67.5	77.3

TABLE C.3. **Part segmentation results.** In this table we present part segmentation results when testing on the original discrete representations of the test sets instead of reconstructing them from the input INRs. We report the IoU for each class, the mean IoU over all the classes (class mIoU) and the mean IoU over all the instances (instance mIoU).

C.8 Alternative Architecture for inr2vec

As reported in Sec. 5.3, inr2vec encoder takes in input the weights of an INR reshaped in a suitable way, discarding the parameters of the first and of the last layers. In this section we consider the possibility of processing all the weights of the input INR, including the input/output ones. To this end, one must properly arrange the input/output parameters since they feature different dimensionality from the ones of the hidden layers and cannot be seamlessly stacked together with them. More specifically, the first layer of an INR consists in a matrix of weights $\mathbf{W}_{in} \in \mathbb{R}^{H \times D}$ and in a vector of biases $\mathbf{b}_{in} \in \mathbb{R}^{H \times 1}$, with H being the dimension of the hidden features of the INR and D being the dimension of the inputs (*i.e.*, 3 in our case of 3D coordinates). The output layer, instead, is responsible of transforming the final vector of hidden features to the predicted output, which is always a single value in the cases considered in our experiments (*i.e.*, udf, sdf and occ). Thus, the last layer presents a matrix of weights $\mathbf{W}_{out} \in \mathbb{R}^{1 \times H}$ and single bias $\mathbf{b}_{out}$. In order to include the input/output parameters in the matrix $\mathbf{P}$ presented in input to inr2vec encoder (see Sec. 5.3), $\mathbf{W}_{in}$ needs to be transposed, obtaining a matrix with shape $3 \times H$, $\mathbf{b}_{in}$ is transposed as done also for the biases of all the other layers, $\mathbf{W}_{out}$ doesn't need any manipulation and we decided to repeat the single-valued $\mathbf{b}_{out}$ H times. In this section, we compare the formulation presented in Sec. 5.3 (reported as "hidden layers") with the one proposed here (reported as "all layers"), looking at the reconstruction capabilities of the two variants of our framework when trained

Architecture	F-Score ↑	CD (mm) ↓
hidden layers	57.41	3.1
all layers	56.76	3.1

TABLE C.4. **Quantitative comparison between alternative inr2vec architectures**. We compare the reconstruction capability of inr2vec when processing only the weights of the hidden layers ("hidden layers") or all the weights ("all layers") of the input INRs.

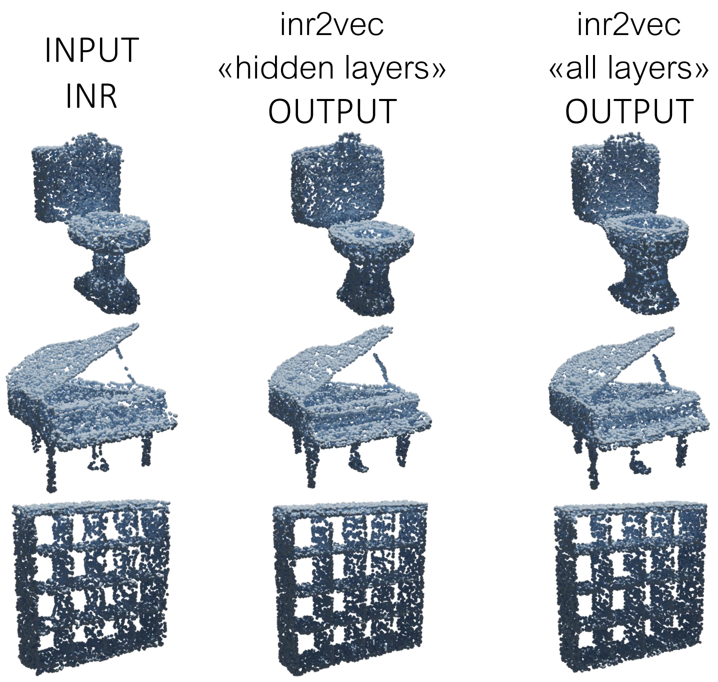
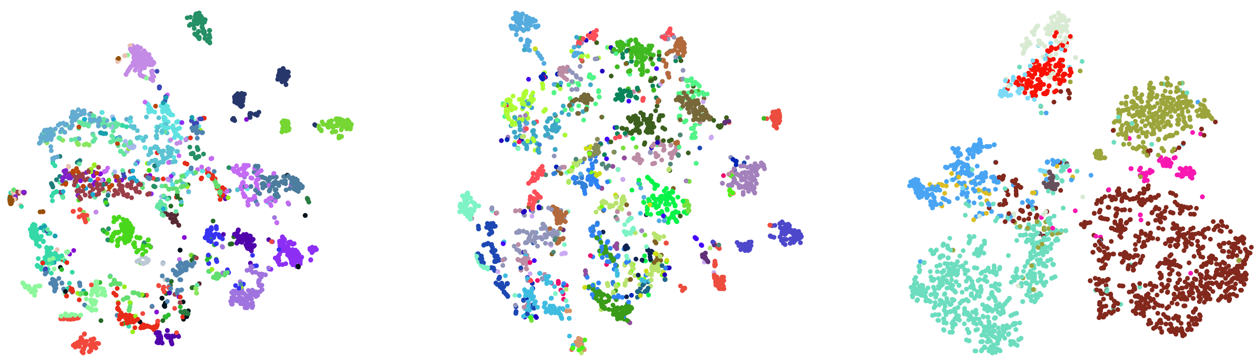


FIGURE C.4. **Qualitative comparison between alternative inr2vec architectures**. We compare the reconstruction capability of inr2vec when processing only the weights of the hidden layers ("hidden layers") or all the weights ("all layers") of the input INRs.

on ModelNet40. In Tab. C.4, we report both the F-score [220] and the Chamfer Distance (CD) [214] between the clouds used to obtain the INRs presented in input to inr2vec and the ones reconstructed from inr2vec embeddings, while in Fig. C.4 we show the same comparison from a qualitative perspective. Results show that processing all the INR weights doesn't produce any significant difference *w.r.t.* ingesting only the weights of the hidden layers. However, the latter variant provides a slight advantage in terms of F-score, simplicity and processing time, motivating our choice to adopt it as formulation for inr2vec.

C.9 t-SNE Visualization of inr2vec Latent Space

We provide in Fig. C.5 the t-SNE visualization of the embeddings produced by inr2vec when presented with the test set INRs of three different datasets. Fig. C.5a shows this visualization for INRs representing the point clouds from ModelNet40, Fig. C.5b for INRs



(A) ModelNet40 (points) (B) Manifold40 (meshes) (C) ShapeNet10 (voxels)

FIGURE C.5. **t-SNE visualizations of inr2vec latent spaces.** We plot the t-SNE components of the embeddings produced by inr2vec on the test sets of three datasets, ModelNet40 (left), Manifold40 (center) and Shapenet10 (right). Colors represent the different classes of the datasets.

representing meshes from Manifold 40, and Fig. C.5c for INRs obtained from the voxelized shapes in ShapeNet10.

The supervision signal adopted during the training of our framework does not entail any kind of constraints *w.r.t.* the organization of the learned latent space. Indeed, this was not necessary for our ultimate goal – *i.e.*, performing downstream tasks on the produced embeddings. However, it is interesting to observe from the t-SNE plots that inr2vec favors spontaneously a semantic arrangement of the embeddings in the learned latent space, with INRs representing objects of the same category being mapped into close positions – as shown by the colors representing the different classes of the considered datasets.

C.10 INR Classification Time: Extended Analysis

We report here the extended analysis of the inference times reported in Fig. 5.9, where we present the classification inference time needed to process udf INRs by standard point cloud baselines – PointNet [16], PointNet++ [162] and DGCNN [122] – and by inr2vec encoder paired with the fully-connected network that we adopt to classify the embeddings (see Sec. 5.5).

The scenario that we had in mind while designing inr2vec is the one where INRs are the only medium to represent 3D shapes, with discrete point clouds not being available. Thus, in Fig. 5.9 for PointNet, PointNet++ and DGCNN we report the inference time including the time spent to reconstruct the discrete cloud from the input INR. In Fig. C.6 and Tab. C.5, for the sake of completeness, we report also the baselines inference times assuming the availability of discrete point clouds, stressing however that this is unlikely if INRs become a standalone format to represent 3D shapes.

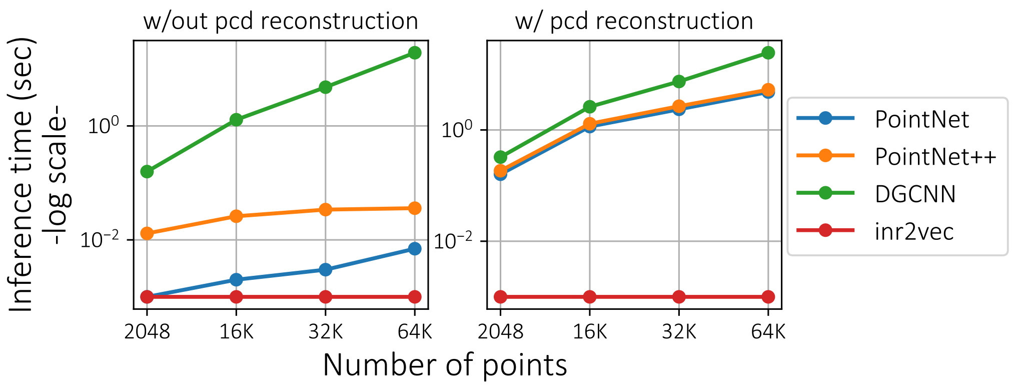


FIGURE C.6. **Time required to classify INRs encoding udf.** We plot the inference time of standard baselines and of our method, both considering the case in which discrete point clouds are available (left) and the one where point clouds must be reconstructed from the input INRs (right).

	Inference Time (seconds)			
Method	2048 pts	16K pts	32K pts	64K pts
PointNet	**0.001**	0.002	0.003	0.007
PointNet*	0.171	1.315	2.609	5.230
PointNet++	0.013	0.026	0.034	0.036
PointNet++*	0.185	1.293	2.672	5.287
DGCNN	0.158	1.285	4.788	19.26
DGCNN*	0.325	2.612	7.426	24.436
inr2vec	**0.001**	**0.001**	**0.001**	**0.001**

TABLE C.5. **Time required to classify INRs encoding udf.** All the times are computed on a gpu NVidia RTX 2080 Ti. * indicates that the time to reconstruct the discrete point cloud from the INR is included.

The numbers plotted in Fig. C.6 and reported in Tab. C.5 show clearly that our framework presents a big advantage *w.r.t.* the competitors. Indeed, by processing directly INRs – where the resolution of the underlying signal is theoretically infinite – inr2vec can classify INRs representing point clouds with different number of points with a constant inference time of 0.001 seconds.

The considered baselines, instead, are negatively affected by the increasing resolution of the input point clouds. While the inference time of PointNet and PointNet++ is still affordable even when processing 64K points, DGCNN gets drastically slow already at 16K points. Furthermore, if point clouds need to be reconstructed from the input INRs, the resulting inference time become prohibitive for all the three baselines.

Bibliography

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. "An image is worth 16x16 words: Transformers for image recognition at scale". In: *arXiv preprint arXiv:2010.11929* (2020).

[11] J. Redmon and A. Farhadi. "YOLOv3: An Incremental Improvement". In: *arXiv Preprint*. 2018.

[12] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. "Encoder-decoder with atrous separable convolution for semantic image segmentation". In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 801–818.

[13] Sen Wang, Ronald Clark, Hongkai Wen, and Niki Trigoni. "Deepvo: Towards end-to-end visual odometry with deep recurrent convolutional neural networks". In: *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2017, pp. 2043–2050.

[14] Jakob Engel, Vladlen Koltun, and Daniel Cremers. "Direct sparse odometry". In: *IEEE transactions on pattern analysis and machine intelligence* 40.3 (2017), pp. 611–625.

[15] Zhiyuan Zhang, Yuchao Dai, and Jiadai Sun. "Deep learning based point cloud registration: an overview". In: *Virtual Reality & Intelligent Hardware* 2.3 (2020). 3D Visual Processing and Reconstruction Special Issue, pp. 222–246. ISSN: 2096-5796. DOI: https://doi.org/10.1016/j.vrih.2020.05.002. URL: https://www.sciencedirect.com/science/article/pii/S2096579620300383.

[16] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. "Pointnet: Deep learning on point sets for 3d classification and segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 652–660.

[17] Caner Hazirbas, Lingni Ma, Csaba Domokos, and Daniel Cremers. "Fusenet: Incorporating depth into semantic segmentation via fusion-based cnn architecture". In: *Asian conference on computer vision*. Springer. 2016, pp. 213–228.

[18] Houssam Halmaoui and Abdelkrim Haqiq. "Computer graphics rendering survey: From rasterization and ray tracing to deep learning". In: *International Conference on Innovations in Bio-Inspired Computing and Applications*. Springer. 2021, pp. 537–548.

[19] Kilian Kleeberger, Richard Bormann, Werner Kraus, and Marco F Huber. “A survey on learning-based robotic grasping”. In: *Current Robotics Reports* 1.4 (2020), pp. 239–249.

[20] Yiheng Xie, Towaki Takikawa, Shunsuke Saito, Or Litany, Shiqin Yan, Numair Khan, Federico Tombari, James Tompkin, Vincent Sitzmann, and Srinath Sridhar. “Neural Fields in Visual Computing and Beyond”. In: *arXiv preprint arXiv:2111.11426* (2021).

[21] David Gargan and Francis Neelamkavil. “Approximating reflectance functions using neural networks”. In: *Eurographics Workshop on Rendering Techniques*. Springer. 1998, pp. 23–34.

[22] Vincent Sitzmann, Julien Martel, Alexander Bergman, David Lindell, and Gordon Wetzstein. “Implicit neural representations with periodic activation functions”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 7462–7473.

[23] Zhengqi Li, Simon Niklaus, Noah Snavely, and Oliver Wang. “Neural scene flow fields for space-time view synthesis of dynamic scenes”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 6498–6508.

[24] 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: *European conference on computer vision*. Springer. 2020, pp. 405–421.

[25] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. “Deepsdf: Learning continuous signed distance functions for shape representation”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 165–174.

[26] Julian Chibane, Gerard Pons-Moll, et al. “Neural unsigned distance fields for implicit function learning”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 21638–21652.

[27] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. “Occupancy networks: Learning 3d reconstruction in function space”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 4460–4470.

[28] Towaki Takikawa, Joey Litalien, Kangxue Yin, Karsten Kreis, Charles Loop, Derek Nowrouzezahrai, Alec Jacobson, Morgan McGuire, and Sanja Fidler. “Neural geometric level of detail: Real-time rendering with implicit 3D shapes”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 11358–11367.

[29] Julien N. P. Martel, David B. Lindell, Connor Z. Lin, Eric R. Chan, Marco Monteiro, and Gordon Wetzstein. “ACORN: Adaptive coordinate networks for neural scene representation”. In: *ACM Trans. Graph. (SIGGRAPH)* 40.4 (2021).

[30] Matthew Tancik, Pratul Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan Barron, and Ren Ng. "Fourier features let networks learn high frequency functions in low dimensional domains". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 7537–7547.

[31] Hsueh-Ti Derek Liu, Francis Williams, Alec Jacobson, Sanja Fidler, and Or Litany. "Learning Smooth Neural Functions via Lipschitz Regularization". In: *arXiv preprint arXiv:2202.08345* (2022).

[32] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding". In: *ACM Trans. Graph.* 41.4 (July 2022), 102:1–102:15. DOI: `10.1145/3528223.3530127`. URL: `https://doi.org/10.1145/3528223.3530127`.

[33] Amos Gropp, Lior Yariv, Niv Haim, Matan Atzmon, and Yaron Lipman. "Implicit Geometric Regularization for Learning Shapes". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 3789–3799.

[34] Vincent Sitzmann, Eric Chan, Richard Tucker, Noah Snavely, and Gordon Wetzstein. "Metasdf: Meta-learning signed distance functions". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 10136–10147.

[35] T. Lewiner, H. Lopes, A. W. Vieira, and G. Tavares. "Efficient Implementation of Marching Cubes' Cases with Topological Guarantees". In: *Journal of Graphics Tools*. 2003.

[36] E. Remelli, A. Lukoianov, S. Richter, B. Guillard, T. Bagautdinov, P. Baque, and P. Fua. "Meshsdf: Differentiable Iso-Surface Extraction". In: *Advances in Neural Information Processing Systems*. 2020.

[37] M. Atzmon, N. Haim, L. Yariv, O. Israelov, H. Maron, and Y. Lipman. "Controlling Neural Level Sets". In: *Advances in Neural Information Processing Systems*. 2019.

[38] I. Mehta, M. Chandraker, and R. Ramamoorthi. "A Level Set Theory for Neural Implicit Evolution under Explicit Flows". In: 2022.

[39] F. Zhao, W. Wang, S. Liao, and L. Shao. "Learning Anchored Unsigned Distance Functions with Gradient Direction Alignment for Single-View Garment Reconstruction". In: *Conference on Computer Vision and Pattern Recognition*. 2021.

[40] R. Venkatesh, T. Karmali, S. Sharma, A. Ghosh, R. V. Babu, L. A. Jeni, and M. Singh. "Deep Implicit Surface Point Prediction Networks". In: *International Conference on Computer Vision*. 2021.

[41] M. Kazhdan and H. Hoppe. "Screened Poisson Surface Reconstruction". In: *ACM Transactions on Graphics* (2013).

[42] B. Guillard, F. Stella, and P. Fua. “Meshudf: Fast and Differentiable Meshing of Unsigned Distance Field Networks”. In: *European Conference on Computer Vision*. 2022.

[43] Songyou Peng, Michael Niemeyer, Lars Mescheder, Marc Pollefeys, and Andreas Geiger. “Convolutional occupancy networks”. In: *European Conference on Computer Vision*. Springer. 2020, pp. 523–540.

[44] Matan Atzmon and Yaron Lipman. “Sal: Sign agnostic learning of shapes from raw data”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 2565–2574.

[45] M. Atzmon and Y. Lipman. “SALD: Sign Agnostic Learning with Derivatives”. In: *International Conference on Learning Representations*. 2020.

[46] Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. “Stereo magnification: learning view synthesis using multiplane images”. In: *ACM Trans. Graph.* (2018).

[47] John Flynn, Michael Broxton, Paul E. Debevec, Matthew DuVall, Graham Fyffe, Ryan S. Overbeck, Noah Snavely, and Richard Tucker. “DeepView: View Synthesis With Learned Gradient Descent”. In: *CVPR*. 2019.

[48] Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. “Local light field fusion: practical view synthesis with prescriptive sampling guidelines”. In: *ACM Trans. Graph.* (2019).

[49] Pratul P. Srinivasan, Richard Tucker, Jonathan T. Barron, Ravi Ramamoorthi, Ren Ng, and Noah Snavely. “Pushing the Boundaries of View Extrapolation With Multiplane Images”. In: *CVPR*. 2019.

[50] Zhengqi Li, Wenqi Xian, Abe Davis, and Noah Snavely. “Crowdsampling the Plenoptic Function”. In: *ECCV*. 2020.

[51] Richard Tucker and Noah Snavely. “Single-View View Synthesis With Multiplane Images”. In: *CVPR*. 2020.

[52] Stephen Lombardi, Tomas Simon, Jason M. Saragih, Gabriel Schwartz, Andreas M. Lehrmann, and Yaser Sheikh. “Neural volumes: learning dynamic renderable volumes from images”. In: *ACM Trans. Graph.* (2019).

[53] Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Nießner, Gordon Wetzstein, and Michael Zollhöfer. “DeepVoxels: Learning Persistent 3D Feature Embeddings”. In: *CVPR*. 2019.

[54] Tong He, John P. Collomosse, Hailin Jin, and Stefano Soatto. “DeepVoxels++: Enhancing the Fidelity of Novel View Synthesis from 3D Voxel Embeddings”. In: *ACCV*. Ed. by Hiroshi Ishikawa, Cheng-Lin Liu, Tomás Pajdla, and Jianbo Shi. 2020.

[55] Nelson L. Max. “Optical Models for Direct Volume Rendering”. In: *IEEE Trans. Vis. Comput. Graph.* (1995).

[56] Pratul P. Srinivasan, Boyang Deng, Xiuming Zhang, Matthew Tancik, Ben Mildenhall, and Jonathan T. Barron. “NeRV: Neural Reflectance and Visibility Fields for Relighting and View Synthesis”. In: *CVPR*. 2021.

[57] Xiuming Zhang, Pratul P. Srinivasan, Boyang Deng, Paul E. Debevec, William T. Freeman, and Jonathan T. Barron. “NeRFactor: Neural Factorization of Shape and Reflectance Under an Unknown Illumination”. In: *arxiv CS.CV 2106.01970* (2021).

[58] Mark Boss, Raphael Braun, Varun Jampani, Jonathan T. Barron, Ce Liu, and Hendrik P. A. Lensch. “NeRD: Neural Reflectance Decomposition from Image Collections”. In: *ICCV*. 2021.

[59] Keunhong Park, Utkarsh Sinha, Jonathan T. Barron, Sofien Bouaziz, Dan B. Goldman, Steven M. Seitz, and Ricardo Martin-Brualla. “Deformable Neural Radiance Fields”. In: *ICCV*. 2021.

[60] Edgar Tretschk, Ayush Tewari, Vladislav Golyanik, Michael Zollhöfer, Christoph Lassner, and Christian Theobalt. “Non-Rigid Neural Radiance Fields: Reconstruction and Novel View Synthesis of a Deforming Scene from Monocular Video”. In: *ICCV*. 2021.

[61] Guy Gafni, Justus Thies, Michael Zollhöfer, and Matthias Nießner. “Dynamic Neural Radiance Fields for Monocular 4D Facial Avatar Reconstruction”. In: *CVPR*. 2021.

[62] Atsuhiro Noguchi, Xiao Sun, Stephen Lin, and Tatsuya Harada. “Neural Articulated Radiance Field”. In: *ICCV*. 2021.

[63] Keunhong Park, Utkarsh Sinha, Peter Hedman, Jonathan T. Barron, Sofien Bouaziz, Dan B. Goldman, Ricardo Martin-Brualla, and Steven M. Seitz. “HyperNeRF: A Higher-Dimensional Representation for Topologically Varying Neural Radiance Fields”. In: *arxiv CS.CV 2106.13228* (2021).

[64] Ricardo Martin-Brualla, Noha Radwan, Mehdi S. M. Sajjadi, Jonathan T. Barron, Alexey Dosovitskiy, and Daniel Duckworth. “NeRF in the Wild: Neural Radiance Fields for Unconstrained Photo Collections”. In: *CVPR*. 2021.

[65] Albert Pumarola, Enric Corona, Gerard Pons-Moll, and Francesc Moreno-Noguer. “D-NeRF: Neural Radiance Fields for Dynamic Scenes”. In: *CVPR*. 2021.

[66] Wenqi Xian, Jia-Bin Huang, Johannes Kopf, and Changil Kim. “Space-Time Neural Irradiance Fields for Free-Viewpoint Video”. In: *CVPR*. 2021.

[67] Chen Gao, Ayush Saraf, Johannes Kopf, and Jia-Bin Huang. “Dynamic View Synthesis from Dynamic Monocular Video”. In: *ICCV*. 2021.

[68] Jonathan T. Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P. Srinivasan. "Mip-NeRF: A Multiscale Representation for Anti-Aliasing Neural Radiance Fields". In: *ICCV*. 2021.

[69] Katja Schwarz, Yiyi Liao, Michael Niemeyer, and Andreas Geiger. "GRAF: Generative Radiance Fields for 3D-Aware Image Synthesis". In: *NeurIPS*. 2020.

[70] Eric R. Chan, Marco Monteiro, Petr Kellnhofer, Jiajun Wu, and Gordon Wetzstein. "Pi-GAN: Periodic Implicit Generative Adversarial Networks for 3D-Aware Image Synthesis". In: *CVPR*. 2021.

[71] Adam R. Kosiorek, Heiko Strathmann, Daniel Zoran, Pol Moreno, Rosalia Schneider, Sona Mokrá, and Danilo Jimenez Rezende. "NeRF-VAE: A Geometry Aware 3D Scene Generative Model". In: *ICML*. 2021.

[72] Alex Yu, Vickie Ye, Matthew Tancik, and Angjoo Kanazawa. "pixelNeRF: Neural Radiance Fields From One or Few Images". In: *CVPR*. 2021.

[73] Julian Chibane, Aayush Bansal, Verica Lazova, and Gerard Pons-Moll. "Stereo Radiance Fields (SRF): Learning View Synthesis from Sparse Views of Novel Scenes". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. June 2021.

[74] Anpei Chen, Zexiang Xu, Fuqiang Zhao, Xiaoshuai Zhang, Fanbo Xiang, Jingyi Yu, and Hao Su. "MVSNeRF: Fast Generalizable Radiance Field Reconstruction from Multi-View Stereo". In: *ICCV*. 2021.

[75] Qianqian Wang, Zhicheng Wang, Kyle Genova, Pratul P. Srinivasan, Howard Zhou, Jonathan T. Barron, Ricardo Martin-Brualla, Noah Snavely, and Thomas A. Funkhouser. "IBRNet: Learning Multi-View Image-Based Rendering". In: *CVPR*. 2021.

[76] Yuan Liu, Sida Peng, Lingjie Liu, Qianqian Wang, Peng Wang, Christian Theobalt, Xiaowei Zhou, and Wenping Wang. "Neural Rays for Occlusion-aware Image-based Rendering". In: *arxiv CS.CV 2107.13421* (2021).

[77] Kangle Deng, Andrew Liu, Jun-Yan Zhu, and Deva Ramanan. "Depth-supervised NeRF: Fewer Views and Faster Training for Free". In: *arxiv CS.CV 2107.02791* (2021).

[78] Alex Yu and Sara Fridovich-Keil, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. *Plenoxels: Radiance Fields without Neural Networks*. 2021. arXiv: 2112.05131 [cs.CV].

[79] Cheng Sun, Min Sun, and Hwann-Tzong Chen. "Direct Voxel Grid Optimization: Super-fast Convergence for Radiance Fields Reconstruction". In: *arXiv preprint arXiv:2111.11215* (2021).

[80] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. "Neural Sparse Voxel Fields". In: *NeurIPS*. 2020.

[81] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. “KiloNeRF: Speeding up Neural Radiance Fields with Thousands of Tiny MLPs”. In: *ICCV*. 2021.

[82] Suttisak Wizadwongsa, Pakkapon Phongthawee, Jiraphon Yenphraphai, and Supasorn Suwajanakorn. “NeX: Real-time View Synthesis with Neural Basis Expansion”. In: *CVPR*. 2021.

[83] Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, and Angjoo Kanazawa. “PlenOctrees for Real-time Rendering of Neural Radiance Fields”. In: *ICCV*. 2021.

[84] Stephan J. Garbin, Marek Kowalski, Matthew Johnson, Jamie Shotton, and Julien P. C. Valentin. “FastNeRF: High-Fidelity Neural Rendering at 200FPS”. In: *ICCV*. 2021.

[85] Peter Hedman, Pratul P. Srinivasan, Ben Mildenhall, Jonathan T. Barron, and Paul E. Debevec. “Baking Neural Radiance Fields for Real-Time View Synthesis”. In: *ICCV*. 2021.

[86] D. Baraff and A. Witkin. “Large Steps in Cloth Simulation”. In: *ACM SIGGRAPH*. 1998, pp. 43–54.

[87] T. Liu, S. Bouaziz, and L. Kavan. “Quasi-newton methods for real-time simulation of hyperelastic materials”. In: *ACM Transactions on Graphics* (2017).

[88] Xavier Provot et al. “Deformation constraints in a mass-spring model to describe rigid cloth behaviour”. In: *Graphics interface*. 1995.

[89] X. Provot. “Collision and self-collision handling in cloth model dedicated to design garments”. In: *Computer Animation and Simulation*. 1997.

[90] M. Tang, R. Tong, R. Narain, C. Meng, and D. Manocha. “A GPU-based streaming algorithm for high-resolution cloth simulation”. In: 2013.

[91] T. Vassilev, B. Spanlang, and Y. Chrysanthou. “Fast cloth animation on walking avatars”. In: *Computer Graphics Forum*. 2001.

[92] C. Zeller. “Cloth simulation on the gpu”. In: *ACM SIGGRAPH*. 2005.

[93] Nvidia. *Nvcloth*. 2018.

[94] Optitext Fashion Design Software. `https://optitex.com/`. 2018.

[95] Nvidia. *NVIDIA Flex*. `https://developer.nvidia.com/flex`. 2018.

[96] M. Designer. `https://www.marvelousdesigner.com`. 2018.

[97] E. Gundogdu, V. Constantin, A. Seifoddini, M. Dang, M. Salzmann, and P. Fua. “Garnet: A Two-Stream Network for Fast and Accurate 3D Cloth Draping”. In: *International Conference on Computer Vision*. 2019.

[98] E. Gundogdu, V. Constantin, S. Parashar, A. Seifoddini, M. Dang, M. Salzmann, and P. Fua. “Garnet++: Improving Fast and Accurate Static 3D Cloth Draping by Curvature Loss”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22.1 (2022), pp. 181–195.

[99] Q. Ma, J. Yang, A. Ranjan, S. Pujades, G. Pons-Moll, S. Tang, and M. J. Black. “Learning to Dress 3D People in Generative Clothing”. In: *Conference on Computer Vision and Pattern Recognition*. 2020.

[100] C. Patel, Z. Liao, and G. Pons-Moll. “Tailornet: Predicting clothing in 3d as a function of human pose, shape and garment style”. In: *Conference on Computer Vision and Pattern Recognition*. 2020.

[101] I. Santesteban, M. A. Otaduy, and D. Casas. “Learning-Based Animation of Clothing for Virtual Try-On”. In: *Computer Graphics Forum (Proc. of Eurographics)* 33.2 (2019).

[102] Y. Shen, J. Liang, and M.C. Lin. “Gan-based garment generation using sewing pattern images”. In: *European Conference on Computer Vision*. 2020.

[103] G. Tiwari, B. L. Bhatnagar, T. Tung, and G. Pons-Moll. “Sizer: A Dataset and Model for Parsing 3D Clothing and Learning Size Sensitive 3D Clothing”. In: *European Conference on Computer Vision*. 2020.

[104] R. Vidaurre, I. Santesteban, E. Garces, and D. Casas. “Fully Convolutional Graph Neural Networks for Parametric Virtual Try-On”. In: *Computer Graphics Forum*. 2020.

[105] T. Y. Wang, D. Ceylan, J. Popovic, and N. J. Mitra. “Learning a Shared Shape Space for Multimodal Garment Design”. In: *ACM SIGGRAPH Asia*. 2018.

[106] R. Narain, A. Samii, and J.F. O’brien. “Adaptive anisotropic remeshing for cloth simulation”. In: *ACM Transactions on Graphics* (2012).

[107] G. Pons-Moll, S. Pujades, S. Hu, and M.J. Black. “Clothcap: Seamless 4D Clothing Capture and Retargeting”. In: *ACM SIGGRAPH* 36.4 (2017), pp. 731–7315.

[108] I. Santesteban, M.A. Otaduy, and D. Casas. “SNUG: Self-Supervised Neural Dynamic Garments”. In: *Conference on Computer Vision and Pattern Recognition*. 2022.

[109] H. Bertiche, M. Madadi, and S. Escalera. “PBNS: Physically Based Neural Simulation for Unsupervised Garment Pose Space Deformation”. In: *ACM Transactions on Graphics* (2021).

[110] E. Corona, A. Pumarola, G. Alenya, G. Pons-Moll, and F. Moreno-Noguer. “Smplicit: Topology-Aware Generative Model for Clothed People”. In: *Conference on Computer Vision and Pattern Recognition*. 2021.

[111] R. Li, B. Guillard, E. Remelli, and P. Fua. “DIG: Draping Implicit Garment over the Human Body”. In: *Asian Conference on Computer Vision*. 2022.

[112] Y. Li, M. Habermann, B. Thomaszewski, S. Coros, T. Beeler, and C. Theobalt. “Deep physics-aware inference of cloth deformation for monocular human performance capture”. In: *International Conference on 3D Vision*. 2021.

[113] J. Liang, M. Lin, and V. Koltun. “Differentiable Cloth Simulation for Inverse Problems”. In: *Advances in Neural Information Processing Systems*. 2019.

[114] R. Narain, T. Pfaff, and J.F. O’Brien. “Folding and crumpling adaptive sheets”. In: *ACM Transactions on Graphics* (2013).

[115] B. L. Bhatnagar, G. Tiwari, C. Theobalt, and G. Pons-Moll. “Multi-Garment Net: Learning to Dress 3D People from Images”. In: *International Conference on Computer Vision*. 2019.

[116] B. Jiang, J. Zhang, Y. Hong, J. Luo, L. Liu, and H. Bao. “Bcnet: Learning body and cloth shape from a single image”. In: *European Conference on Computer Vision*. 2020.

[117] I. Santesteban, N. Thuerey, M. A. Otaduy, and D. Casas. “Self-Supervised Collision Handling via Generative 3D Garment Models for Virtual Try-On”. In: *Conference on Computer Vision and Pattern Recognition*. 2021.

[118] X. Pan, J. Mai, X. Jiang, D. Tang, J. Li, T. Shao, K. Zhou, X. Jin, and D. Manocha. “Predicting loose-fitting garment deformations using bone-driven motion networks”. In: *ACM SIGGRAPH*. 2022.

[119] H. Bertiche, M. Madadi, E. Tylson, and S. Escalera. “DeePSD: Automatic Deep Skinning and Pose Space Deformation for 3D Garment Animation”. In: *International Conference on Computer Vision*. 2021.

[120] I. Zakharkin, K. Mazur, A. Grigorev, and V. Lempitsky. “Point-based modeling of human clothing”. In: *International Conference on Computer Vision*. 2021.

[121] M. Loper, N. Mahmood, J. Romero, G. Pons-Moll, and M.J. Black. “SMPL: A Skinned Multi-Person Linear Model”. In: *ACM SIGGRAPH Asia* 34.6 (2015).

[122] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. “Dynamic graph cnn for learning on point clouds”. In: *Acm Transactions On Graphics (tog)* 38.5 (2019), pp. 1–12.

[123] H. De Vries, F. Strub, J. Mary, H. Larochelle, O. Pietquin, and A.C. Courville. “Modulating Early Visual Processing by Language”. In: *Advances in Neural Information Processing Systems*. 2017.

[124] Y. Duan, H. Zhu, H. Wang, L. Yi, R. Nevatia, and L. J. Guibas. “Curriculum DeepSDF”. In: *European Conference on Computer Vision*. 2020.

[125] H. Bertiche, M. Madadi, and S. Escalera. “CLOTH3D: Clothed 3D Humans”. In: 2020, pp. 344–359.

[126] N. Mahmood, N. Ghorbani, N. F. Troje, G. Pons-Moll, and M. J. Black. "AMASS: Archive of Motion Capture as Surface Shapes". In: *International Conference on Computer Vision*. 2019, pp. 5442–5451.

[127] G. Moon, H. Nam, T. Shiratori, and K.M. Lee. "3D Clothed Human Reconstruction in the Wild". In: *European Conference on Computer Vision*. 2022.

[128] Yu Y. Rong, T. Shiratori, and H. Joo. "Frankmocap: Fast monocular 3d hand and body motion capture by regression and integration". In: *International Conference on Computer Vision Workshops*. 2021.

[129] L. Yang, Q. Song, Z. Wang, M. Hu, C. Liu, X. Xin, W. Jia, and S. Xu. "Renovating parsing R-CNN for accurate multiple human parsing". In: *European Conference on Computer Vision*. 2020.

[130] R. Li, M. Zheng, S. Karanam, T. Chen, and Z. Wu. "Everybody Is Unique: Towards Unbiased Human Mesh Recovery". In: 2021.

[131] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. *PyTorch3D*. `https://github.com/facebookresearch/pytorch3d`. 2020.

[132] A. Davydov, A. Remizova, V. Constantin, S. Honari, M. Salzmann, and P. Fua. "Adversarial Parametric Pose Prior". In: *Conference on Computer Vision and Pattern Recognition*. 2022.

[133] Ayush Tewari, O Fried, J Thies, V Sitzmann, S Lombardi, Z Xu, T Simon, M Nießner, E Tretschk, L Liu, et al. "Advances in Neural Rendering". In: *ACM SIGGRAPH 2021*. ACM. 2021, pp. 1–320.

[134] Yao Yao, Zixin Luo, Shiwei Li, Jingyang Zhang, Yufan Ren, Lei Zhou, Tian Fang, and Long Quan. "BlendedMVS: A Large-Scale Dataset for Generalized Multi-View Stereo Networks". In: *CVPR*. 2020.

[135] Henrik Aanæs, Rasmus Ramsbøl Jensen, George Vogiatzis, Engin Tola, and Anders Bjorholm Dahl. "Large-scale data for multiple-view stereopsis". In: *International Journal of Computer Vision* 120.2 (2016), pp. 153–168.

[136] Jeremy Reizenstein, Roman Shapovalov, Philipp Henzler, Luca Sbordone, Patrick Labatut, and David Novotny. "Common objects in 3d: Large-scale learning and evaluation of real-life 3d category reconstruction". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 10901–10911.

[137] Angela Dai, Angel X Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. "Scannet: Richly-annotated 3d reconstructions of indoor scenes". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 5828–5839.

[138] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. "Tanks and temples: Benchmarking large-scale scene reconstruction". In: *ACM Transactions on Graphics (ToG)* 36.4 (2017), pp. 1–13.

[139] Carsten Moenning and Neil A Dodgson. "Fast marching farthest point sampling for implicit surfaces and point clouds". In: *Computer Laboratory Technical Report* 565 (2003), pp. 1–12.

[140] Tejalal Choudhary, Vipul Mishra, Anurag Goswami, and Jagannathan Sarangapani. "A comprehensive survey on model compression and acceleration". In: *Artificial Intelligence Review* 53.7 (2020), pp. 5113–5155.

[141] Jonathan Frankle and Michael Carbin. "The lottery ticket hypothesis: Finding sparse, trainable neural networks". In: *arXiv preprint arXiv:1803.03635* (2018).

[142] G. E. Hinton and R. R. Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks". In: *Science* 313.5786 (2006), pp. 504–507. ISSN: 0036-8075. DOI: 10.1126/science.1127647. eprint: https://science.sciencemag.org/content/313/5786/504.full.pdf. URL: https://science.sciencemag.org/content/313/5786/504.

[143] Andrei A Rusu, Dushyant Rao, Jakub Sygnowski, Oriol Vinyals, Razvan Pascanu, Simon Osindero, and Raia Hadsell. "Meta-learning with latent embedding optimization". In: *arXiv preprint arXiv:1807.05960* (2018).

[144] Alessandro Achille, Michael Lam, Rahul Tewari, Avinash Ravichandran, Subhransu Maji, Charless C Fowlkes, Stefano Soatto, and Pietro Perona. "Task2vec: Task embedding for meta-learning". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 6430–6439.

[145] David Ha, Andrew Dai, and Quoc V Le. "Hypernetworks". In: *arXiv preprint arXiv:1609.09106* (2016).

[146] David Krueger, Chin-Wei Huang, Riashat Islam, Ryan Turner, Alexandre Lacoste, and Aaron Courville. "Bayesian hypernetworks". In: *arXiv preprint arXiv:1710.04759* (2017).

[147] Christos Louizos and Max Welling. "Multiplicative normalizing flows for variational bayesian neural networks". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 2218–2227.

[148] Luca Bertinetto, João F Henriques, Jack Valmadre, Philip Torr, and Andrea Vedaldi. "Learning feed-forward one-shot learners". In: *Advances in neural information processing systems*. 2016, pp. 523–531.

[149] Xu Jia, Bert De Brabandere, Tinne Tuytelaars, and Luc V Gool. "Dynamic filter networks". In: *Advances in Neural Information Processing Systems*. 2016, pp. 667–675.

[150] Jonathan Lorraine and David Duvenaud. “Stochastic hyperparameter optimization through hypernetworks”. In: *arXiv preprint arXiv:1802.09419* (2018).

[151] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. “Smash: one-shot model architecture search through hypernetworks”. In: *arXiv preprint arXiv:1708.05344* (2017).

[152] Seung Hyun Lee, Dae Ha Kim, and Byung Cheol Song. “Self-supervised Knowledge Distillation Using Singular Value Decomposition”. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 335–350.

[153] Oscar Chang and Hod Lipson. “Neural network quine”. In: *Artificial Life Conference Proceedings*. MIT Press. 2018, pp. 234–241.

[154] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. “Once-for-all: Train one network and specialize it for efficient deployment”. In: *arXiv preprint arXiv:1908.09791* (2019).

[155] Jingyue Lu and M. Pawan Kumar. “Neural Network Branching for Neural Network Verification”. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=B1evfa4tPB.

[156] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. “Distilling the knowledge in a neural network”. In: *arXiv preprint arXiv:1503.02531* (2015).

[157] Ya Le and Xuan Yang. “Tiny imagenet visual recognition challenge”. In: *CS 231N* 7 (2015), p. 7.

[158] Alex Krizhevsky, Geoffrey Hinton, et al. “Learning multiple layers of features from tiny images”. In: *Tech Report* (2009).

[159] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[160] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[161] Angel X Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, et al. “Shapenet: An information-rich 3d model repository”. In: *arXiv preprint arXiv:1512.03012* (2015).

[162] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. “Pointnet++: Deep hierarchical feature learning on point sets in a metric space”. In: *Advances in neural information processing systems* 30 (2017).

[163] Shi-Min Hu, Zheng-Ning Liu, Meng-Hao Guo, Jun-Xiong Cai, Jiahui Huang, Tai-Jiang Mu, and Ralph R Martin. "Subdivision-based mesh convolution networks". In: *ACM Transactions on Graphics (TOG)* 41.3 (2022), pp. 1–16.

[164] Daniel Maturana and Sebastian Scherer. "Voxnet: A 3d convolutional neural network for real-time object recognition". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2015, pp. 922–928.

[165] Emilien Dupont, Hyunjik Kim, SM Ali Eslami, Danilo Jimenez Rezende, and Dan Rosenbaum. "From data to functa: Your data point is a function and you can treat it like one". In: *International Conference on Machine Learning*. PMLR. 2022, pp. 5694–5725.

[166] Emilien Dupont, Adam Goliński, Milad Alizadeh, Yee Whye Teh, and Arnaud Doucet. "Coin: Compression with implicit neural representations". In: *arXiv preprint arXiv:2103.03123* (2021).

[167] Yannick Strümpler, Janis Postels, Ren Yang, Luc Van Gool, and Federico Tombari. "Implicit Neural Representations for Image Compression". In: *arXiv preprint arXiv:2112.04267* (2021).

[168] Yunfan Zhang, Ties van Rozendaal, Johann Brehmer, Markus Nagel, and Taco Cohen. "Implicit Neural Video Compression". In: *arXiv preprint arXiv:2112.11312* (2021).

[169] Charles R Qi, Hao Su, Matthias Nießner, Angela Dai, Mengyuan Yan, and Leonidas J Guibas. "Volumetric and multi-view cnns for object classification on 3d data". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 5648–5656.

[170] Shuran Song and Jianxiong Xiao. "Deep sliding shapes for amodal 3d object detection in rgb-d images". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 808–816.

[171] Christopher B Choy, Danfei Xu, JunYoung Gwak, Kevin Chen, and Silvio Savarese. "3d-r2n2: A unified approach for single and multi-view 3d object reconstruction". In: *European conference on computer vision*. Springer. 2016, pp. 628–644.

[172] Rohit Girdhar, David F Fouhey, Mikel Rodriguez, and Abhinav Gupta. "Learning a predictable and generative vector representation for objects". In: *European Conference on Computer Vision*. Springer. 2016, pp. 484–499.

[173] Danilo Jimenez Rezende, SM Eslami, Shakir Mohamed, Peter Battaglia, Max Jaderberg, and Nicolas Heess. "Unsupervised learning of 3d structure from images". In: *Advances in neural information processing systems* 29 (2016).

[174] David Stutz and Andreas Geiger. "Learning 3d shape completion from laser scan data with weak supervision". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 1955–1964.

[175] Jiajun Wu, Chengkai Zhang, Tianfan Xue, Bill Freeman, and Josh Tenenbaum. "Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling". In: *Advances in neural information processing systems* 29 (2016).

[176] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. "3D ShapeNets: A deep representation for volumetric shapes". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1912–1920.

[177] Shaoshuai Shi, Chaoxu Guo, Li Jiang, Zhe Wang, Jianping Shi, Xiaogang Wang, and Hongsheng Li. "Pv-rcnn: Point-voxel feature set abstraction for 3d object detection". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 10529–10538.

[178] Haoxuan You, Yifan Feng, Rongrong Ji, and Yue Gao. "Pvnet: A joint convolutional network of point cloud and multi-view for 3d shape recognition". In: *Proceedings of the 26th ACM international conference on Multimedia*. 2018, pp. 1310–1318.

[179] Lei Li, Siyu Zhu, Hongbo Fu, Ping Tan, and Chiew-Lan Tai. "End-to-end learning local multi-view descriptors for 3d point clouds". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 1919–1928.

[180] Ze Liu, Han Hu, Yue Cao, Zheng Zhang, and Xin Tong. "A closer look at local aggregation operators in point cloud analysis". In: *European Conference on Computer Vision*. Springer. 2020, pp. 326–342.

[181] Binh-Son Hua, Minh-Khoi Tran, and Sai-Kit Yeung. "Pointwise convolutional neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 984–993.

[182] Yifan Xu, Tianqi Fan, Mingye Xu, Long Zeng, and Yu Qiao. "Spidercnn: Deep learning on point sets with parameterized convolutional filters". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 87–102.

[183] Matan Atzmon, Haggai Maron, and Yaron Lipman. "Point Convolutional Neural Networks by Extension Operators". In: *ACM Transactions on Graphics* 37.4 (2018).

[184] Wenxuan Wu, Zhongang Qi, and Li Fuxin. "Pointconv: Deep convolutional networks on 3d point clouds". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 9621–9630.

[185] Siqi Fan, Qiulei Dong, Fenghua Zhu, Yisheng Lv, Peijun Ye, and Fei-Yue Wang. "SCF-Net: Learning spatial contextual features for large-scale point cloud segmentation". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 14504–14513.

[186] Mutian Xu, Runyu Ding, Hengshuang Zhao, and Xiaojuan Qi. “Paconv: Position adaptive convolution with dynamic kernel assembling on point clouds”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 3173–3182.

[187] Hugues Thomas, Charles R Qi, Jean-Emmanuel Deschaud, Beatriz Marcotegui, François Goulette, and Leonidas J Guibas. “Kpconv: Flexible and deformable convolution for point clouds”. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2019, pp. 6411–6420.

[188] Guohao Li, Matthias Muller, Ali Thabet, and Bernard Ghanem. “Deepgcns: Can gcns go as deep as cnns?” In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2019, pp. 9267–9276.

[189] Lei Wang, Yuchun Huang, Yaolin Hou, Shenman Zhang, and Jie Shan. “Graph attention convolution for point cloud semantic segmentation”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 10296–10305.

[190] Meng-Hao Guo, Jun-Xiong Cai, Zheng-Ning Liu, Tai-Jiang Mu, Ralph R Martin, and Shi-Min Hu. “Pct: Point cloud transformer”. In: *Computational Visual Media* 7.2 (2021), pp. 187–199.

[191] Hengshuang Zhao, Li Jiang, Jiaya Jia, Philip HS Torr, and Vladlen Koltun. “Point transformer”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 16259–16268.

[192] Jonathan Masci, Davide Boscaini, Michael Bronstein, and Pierre Vandergheynst. “Geodesic convolutional neural networks on riemannian manifolds”. In: *Proceedings of the IEEE international conference on computer vision workshops*. 2015, pp. 37–45.

[193] Davide Boscaini, Jonathan Masci, Emanuele Rodolà, and Michael Bronstein. “Learning shape correspondence with anisotropic convolutional neural networks”. In: *Advances in neural information processing systems* 29 (2016).

[194] Jingwei Huang, Haotian Zhang, Li Yi, Thomas Funkhouser, Matthias Nießner, and Leonidas J Guibas. “Texturenet: Consistent local parametrizations for learning from high-resolution signals on meshes”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 4440–4449.

[195] Yuqi Yang, Shilin Liu, Hao Pan, Yang Liu, and Xin Tong. “PFCNN: Convolutional neural networks on 3D surfaces using parallel frames”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 13578–13587.

[196] Niv Haim, Nimrod Segol, Heli Ben-Hamu, Haggai Maron, and Yaron Lipman. “Surface networks via general covers”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 632–641.

[197] Jonas Schult, Francis Engelmann, Theodora Kontogianni, and Bastian Leibe. "Dualconvmesh-net: Joint geodesic and euclidean convolutions on 3d meshes". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 8612–8622.

[198] Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M Bronstein. "Geometric deep learning on graphs and manifolds using mixture model cnns". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 5115–5124.

[199] Dmitriy Smirnov and Justin Solomon. "HodgeNet: learning spectral geometry on triangle meshes". In: *ACM Transactions on Graphics (TOG)* 40.4 (2021), pp. 1–11.

[200] Rana Hanocka, Amir Hertz, Noa Fish, Raja Giryes, Shachar Fleishman, and Daniel Cohen-Or. "Meshcnn: a network with an edge". In: *ACM Transactions on Graphics (TOG)* 38.4 (2019), pp. 1–12.

[201] Francesco Milano, Antonio Loquercio, Antoni Rosinol, Davide Scaramuzza, and Luca Carlone. "Primal-dual mesh convolutional neural networks". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 952–963.

[202] Alon Lahav and Ayellet Tal. "Meshwalker: Deep mesh understanding by random walks". In: *ACM Transactions on Graphics (TOG)* 39.6 (2020), pp. 1–13.

[203] Chunfeng Lian, Li Wang, Tai-Hsien Wu, Mingxia Liu, Francisca Durán, Ching-Chang Ko, and Dinggang Shen. "Meshsnet: Deep multi-scale mesh feature learning for end-to-end tooth labeling on 3d dental surfaces". In: *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer. 2019, pp. 837–845.

[204] Yutong Feng, Yifan Feng, Haoxuan You, Xibin Zhao, and Yue Gao. "Meshnet: Mesh neural network for 3d shape representation". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 8279–8286.

[205] Xianzhi Li, Ruihui Li, Lei Zhu, Chi-Wing Fu, and Pheng-Ann Heng. "DNF-Net: A deep normal filtering network for mesh denoising". In: *IEEE Transactions on Visualization and Computer Graphics* 27.10 (2020), pp. 4060–4072.

[206] Amir Hertz, Rana Hanocka, Raja Giryes, and Daniel Cohen-Or. "Deep Geometric Texture Synthesis". In: *ACM Trans. Graph.* 39.4 (2020). ISSN: 0730-0301. DOI: `10.1145/3386569.3392471`. URL: `https://doi.org/10.1145/3386569.3392471`.

[207] Thomas Unterthiner, Daniel Keysers, Sylvain Gelly, Olivier Bousquet, and Ilya O. Tolstikhin. "Predicting Neural Network Accuracy from Weights". In: *arXiv* abs/2002.11448 (2020).

[208] Konstantin Schürholt, Dimche Kostadinov, and Damian Borth. "Self-Supervised Representation Learning on Neural Network Weights for Model Characteristic Prediction". In: *Advances in Neural Information Processing Systems*. Ed. by A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan. 2021. URL: https://openreview.net/forum?id=F1D8buayXQT.

[209] Boris Knyazev, Michal Drozdzal, Graham W. Taylor, and Adriana Romero. "Parameter Prediction for Unseen Deep Architectures". In: *Advances in Neural Information Processing Systems*. Ed. by A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan. 2021. URL: https://openreview.net/forum?id=vqHak8NLk25.

[210] Florian Jaeckle and M Pawan Kumar. "Generating adversarial examples with graph neural networks". In: *Uncertainty in Artificial Intelligence*. PMLR. 2021, pp. 1556–1564.

[211] William E Lorensen and Harvey E Cline. "Marching cubes: A high resolution 3D surface construction algorithm". In: *ACM siggraph computer graphics* 21.4 (1987), pp. 163–169.

[212] Samuel K Ainsworth, Jonathan Hayase, and Siddhartha Srinivasa. "Git re-basin: Merging models modulo permutation symmetries". In: *arXiv preprint arXiv:2209.04836* (2022).

[213] Gizem Yüce, Guillermo Ortiz-Jiménez, Beril Besbinar, and Pascal Frossard. "A Structured Dictionary Perspective on Implicit Neural Representations". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 19228–19238.

[214] Haoqiang Fan, Hao Su, and Leonidas J Guibas. "A point set generation network for 3d object reconstruction from a single image". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 605–613.

[215] Cameron R Wolfe and Keld T Lundgaard. "E-Stitchup: Data Augmentation for Pre-Trained Embeddings". In: *arXiv preprint arXiv:1912.00772* (2019).

[216] Li Yi, Vladimir G. Kim, Duygu Ceylan, I-Chao Shen, Mengyan Yan, Hao Su, Cewu Lu, Qixing Huang, Alla Sheffer, and Leonidas Guibas. "A Scalable Active Framework for Region Annotation in 3D Shape Collections". In: *SIGGRAPH Asia* (2016).

[217] Panos Achlioptas, Olga Diamanti, Ioannis Mitliagkas, and Leonidas Guibas. "Learning representations and generative models for 3d point clouds". In: *International conference on machine learning*. PMLR. 2018, pp. 40–49.

[218] Ruihui Li, Xianzhi Li, Ka-Hei Hui, and Chi-Wing Fu. "SP-GAN: Sphere-guided 3D shape generation and manipulation". In: *ACM Transactions on Graphics (TOG)* 40.4 (2021), pp. 1–12.

[219] Liang Pan, Xinyi Chen, Zhongang Cai, Junzhe Zhang, Haiyu Zhao, Shuai Yi, and Ziwei Liu. “Variational relational point completion network”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 8524–8533.

[220] Maxim Tatarchenko, Stephan R Richter, René Ranftl, Zhuwen Li, Vladlen Koltun, and Thomas Brox. “What do single-view 3d reconstruction networks learn?” In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 3405–3414.

[221] Yilun Du, Katie Collins, Josh Tenenbaum, and Vincent Sitzmann. “Learning signal-agnostic manifolds of neural fields”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 8320–8331.

[222] Emilien Dupont, Yee Whye Teh, and Arnaud Doucet. “Generative models as distributions of functions”. In: *arXiv preprint arXiv:2102.04776* (2021).

[223] Rahim Entezari, Hanie Sedghi, Olga Saukh, and Behnam Neyshabur. “The Role of Permutation Invariance in Linear Mode Connectivity of Neural Networks”. In: *International Conference on Learning Representations*. 2021.

[224] Tianyang Li, Xin Wen, Yu-Shen Liu, Hua Su, and Zhizhong Han. “Learning deep implicit functions for 3D shapes with dynamic code clouds”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 12840–12850.

[225] Vishwanath Saragadam, Daniel LeJeune, Jasper Tan, Guha Balakrishnan, Ashok Veeraraghavan, and Richard G Baraniuk. “WIRE: Wavelet Implicit Neural Representations”. In: *arXiv preprint arXiv:2301.05187* (2023).

[226] Chenxi Lola Deng and Enzo Tartaglione. “Compressing explicit voxel grid representations: fast nerfs become also small”. In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2023, pp. 1236–1245.

[227] Verica Lazova, Vladimir Guzov, Kyle Olszewski, Sergey Tulyakov, and Gerard Pons-Moll. “Control-nerf: Editable feature volumes for scene rendering and manipulation”. In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2023, pp. 4340–4350.

[228] Decai Chen, Peng Zhang, Ingo Feldmann, Oliver Schreer, and Peter Eisert. “Recovering Fine Details for Neural Implicit Surface Reconstruction”. In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2023, pp. 4330–4339.

[229] Petros Tzathas, Petros Maragos, and Anastasios Roussos. “3D Neural Sculpting (3DNS): Editing Neural Signed Distance Functions”. In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2023, pp. 4521–4530.

[230] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *International Conference on Machine Learning*. 2015.

[231] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *ICLR*. 2015.

[232] T. Pfaff, R. Narain, J.M. De Joya, and J.F. O'Brien. "Adaptive tearing and cracking of thin sheets". In: *ACM Transactions on Graphics* 33.4 (2014), pp. 1–9.

[233] M. Tang, T. Wang, Z. Liu, R. Tong, and D. Manocha. "I-Cloth: Incremental Collision Handling for Gpu-Based Interactive Cloth Simulation". In: *ACM Transactions on Graphics*. 2018.

[234] Yonglong Tian, Dilip Krishnan, and Phillip Isola. "Contrastive representation distillation". In: *arXiv preprint arXiv:1910.10699* (2019).

[235] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. "Focal loss for dense object detection". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2980–2988.

[236] Pierre Baque, Edoardo Remelli, Francois Fleuret, and Pascal Fua. "Geodesic convolutional shape optimization". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 472–481.

[237] David JJ Toal and Andy J Keane. "Efficient multipoint aerodynamic design optimization via cokriging". In: *Journal of Aircraft* 48.5 (2011), pp. 1685–1695.

[238] Nobuyuki Umetani and Bernd Bickel. "Learning three-dimensional flow for interactive aerodynamic design". In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), pp. 1–10.

[239] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. "Open3D: A Modern Library for 3D Data Processing". In: *arXiv:1801.09847* (2018).

[240] Ilya Loshchilov and Frank Hutter. "Decoupled weight decay regularization". In: *arXiv preprint arXiv:1711.05101* (2017).

[241] Leslie N Smith and Nicholay Topin. "Super-convergence: Very fast training of neural networks using large learning rates". In: *Artificial intelligence and machine learning for multi-domain operations applications*. Vol. 11006. International Society for Optics and Photonics. 2019, p. 1100612.

[242] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. "Improved training of wasserstein gans". In: *Advances in neural information processing systems* 30 (2017).