An Autoencoder architecture for Neural Radiance Fields (NeRF)

Advanced Computer Vision and Deep Learning Practical

by **Ulrich Prestel**
ulrich.prestel@protonmail.com

November 2020

Repository `https://github.com/uprestel/AutoNeRF`
Video Samples `https://youtu.be/1ovewzz6u78`
Video Ablation Study `https://youtu.be/lzlAp05DhwQ`

Contents

1 Abstract

The goal of the practical was to come up with a generative model that is able to generate novel views of a 3D scene very quickly and is trained with very few images of that scene. Ultimately, we combined multiple models, namely neural radiance fields (NeRF), a variational autoencoder (VAE) and a conditional invertible neural network (cINN). We carefully combine them to get a generative model that fits our criteria. As a result, we obtain a very quick inference of novel views with only very few training images.
We also demonstrate, that the architecture can easily be modified to estimate the pose of the observer given an image. Finally, we conduct an ablation study where we justify our sampling scheme.

2 Introduction

In this practical we address the problem of view synthesis. Concretely, this means that we are given a sampling of views of a static scene, and try to render this scene from new angles. This problem is closely related to the new field of neural rendering [16], which heavily relies on generative models.
Depending on how dense our view sampling is, and what information we are given, we can apply different techniques.
When our sampling is for example very dense, we can render new images of the scene by light field interpolation techniques [2]. However, when we have very few samples, this problem becomes a lot harder, especially for data-driven methods like neural networks. However, recent work in computer vision and computer graphics has lead to new techniques in this area. A very promising approach is to represent a scene using a MLP that directly maps 3D spatial locations to information of the scene we want to learn, such as color and material properties [13], signed distance function values [9] [12] and so on. A recent architecture, called NeRF (Neural Radiance Fields) does just that. It represents a scene by mapping 3D spatial locations to density and color values using a MLP. During the training process, we can thus cast rays from our given camera poses, sample our scene representation along these rays and accumulate the resulting values to an image using classical volume rendering techniques. This works quite well, and produces high fidelity results and depth maps only from a very small training set. However, rendering an image from a new angle can be very costly.
To mitigate this problem, we employ a Variational Autoencoder to generate new images. Since the Autoencoder needs a lot of image data, we can readily generate it using a trained NeRF model. Now we can generate new images using the VAE, by sampling the latent space and decoding the latent variable. However, since we want to generate new images using the camera pose, we do the following: We use a cINN (Conditional Invertible Neural Network) [17] to act as a normalizing flow between the distribution of the latent variables and some prior distribution, where the condition is our camera pose. That way, we can easily generate new images using our decoder.
Likewise, we can invert this whole process to estimate the pose of the camera.

In summary, (i) our model can be trained on little data, (ii) is able to generate new views quickly and (iii) can be modified to estimate the camera pose given an input image.

2.1 Problem setting & Related Work

Let's say, that we are given a small set of distinct images of a 3D-scene, taken from various positions and orientations. We denote this set as $\mathcal{T} = \{(x_i, c_i)\}_{i=1}^{N}$, where x_i is the RGB-image, and c_i is the corresponding camera pose in homogeneous coordinates, i.e.

$$c_i = \left(\begin{array}{c|c} \mathbf{R}_i & \mathbf{t}_i \\ \hline 0 & 1 \end{array}\right) \in \mathbb{R}^{4\times 4}$$

for some rotation $\mathbf{R}_i \in \mathbb{R}^{3\times 3}$ of the camera and a translation $\mathbf{t}_i \in \mathbb{R}^3$. The goal is now to synthesize new images of the same scene at a new camera pose c. Since the images in our set $\mathcal{T}$ are essentially projections of 3D-data, it makes sense to represent the underlying 3D-scene with some kind of representation we can use later, to generate new views.

Problem definition

Assume that we are given a small sampling of views $\mathcal{T}$ of a 3D-scene, where the views $(x, c) \in \mathcal{T}$ are comprised of a 2D-image x obtained by a camera with a pose c. We are then trying to synthesize new images by solving the following two subproblems:

1. Reconstruct the 3D scene using $\mathcal{T}$ into a representation F.
2. Using the representation F, synthesize new images from a new pose c.

Now we can tackle each aspect of our problem specifically.
To start with the representation F in problem 1, there are of course many such representations, each with their advantages and disadvantages. When we have chosen such a representation, we must choose a way to render new images accordingly.

- ***Mesh representation***: The most popular way to represent 3D scenes are meshes comprised of triangles. Due to the simplicity it has been widely adopted in the computer graphics community, and has a large amounts of data for computer vision research, for example ShapeNet [3]
  To obtain a 3D-scene representation, one could use differentiable renderers to directly optimize the mesh representation by minimizing the loss of the appearance of the mesh compared to the ground truth images via gradient descent.
  However, this is often difficult, since this method requires an initial mesh with fixed topology, which is often unavailable in real-world scenes [6].

- ***Volumetric representation***: Volumetric approaches, such as point clouds and voxels, can yield very impressive results [5]. However, these methods do not scale well as the resolution of the images increases. This is due to the fact, that one needs finer sampling to render 3D images from volumetric representations. This representation can also be very costly to store.

- ***Implicit representation***: In this representation we encode properties of our scene using a function. For example (convolutional) occupancy networks [8][14] treat the surface of an object as a decision boundary, where we discriminate between inside and outside of the object.
  A similar approach is used by neural radiance fields (NeRF) [13]. Here we assign each point in space, given the camera direction, a density value and a RGB value. To obtain a new image, we can simply use classical rendering techniques. For further details, see section 3.

2.2 Datasets

A very important aspect of this project is to get suitable data to train on. Luckily, the authors of NeRF have generated a wide range of synthetic data and provided data of real scenes as well.
We modified the data, such that all images have the resolution 100x100 and stored them in a compressed numpy-array for easy access. Each dataset contains 100 training images and is split into a training set and a test set. For samples see figure 1.

(a) (b) (c)

Figure 1: We will examine three different datasets, all of which contain 100 training images: (a) lego bulldozer (b) chair (c) hotdog.

3 Representing scenes using Neural Radiance Fields (NeRF)

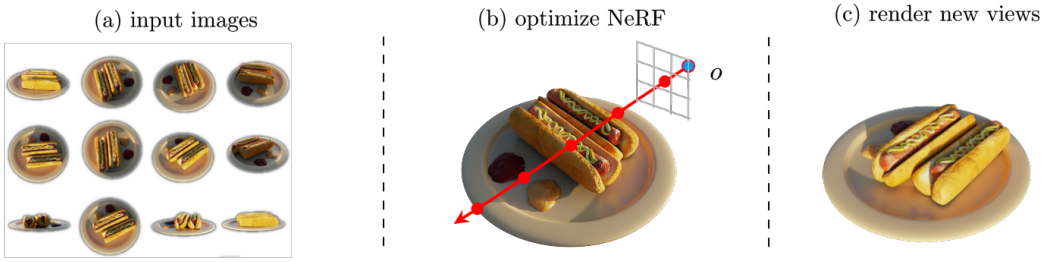

Figure 2: The pipeline of NeRF: (a) We gather 100 training images, (b) for each pixel we generate rays and sample query points from our scene representation. (c) Now we can render the scene from novel views.

The idea of the Neural Radiance Fields (NeRF) is deceptively simple, and solves our previously defined problems 1 and 2 very elegantly. We represent the scene continuously using a MLP F_Θ. The method works as follows:

1. ***Generate rays***: Let's say that we have a camera at the location $\mathbf{o} \in \mathbb{R}^3$ and its orientation in space it described by two angles (θ', ϕ'). From our camera origin $\mathbf{o}$ we generate rays, which are oriented at an angle (θ, ϕ). We can describe the direction of the ray as a cartesian unit vector $\mathbf{d}$, and thus our ray is $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ for $t \in [t_n, t_f]$. We denote the set of these rays as $\mathcal{R}$. Note that we need such a ray for every pixel in our final image.

2. ***Sample query points***: For each ray $r \in \mathcal{R}$, we simply sample N positions along that ray by uniform sampling:

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

   The key idea is now to utilize our MLP-scene representation to obtain the colors and density values at our query points: Assuming we take such a query point $\mathbf{x} = \mathbf{r}(t)$, our MLP is a mapping $F_\Theta : (\mathbf{x}, \mathbf{d}) \longmapsto (\mathbf{c}, \sigma)$ to the RGB-value $\mathbf{c}(\mathbf{x}, \mathbf{d})$ and a density $\sigma(\mathbf{x})$

   Remark. *A few important facts to consider are:*

   - *We encourage the representation F_Θ to be multiview consistent by restricting the network to predict the volume density $\sigma(\boldsymbol{x})$ as a function of only the location $\boldsymbol{x}$, while the color $\boldsymbol{c}(\boldsymbol{x}, \boldsymbol{d})$ is dependent on both the direction of the ray $\boldsymbol{d}$ and the location $\boldsymbol{x}$. This is important, since objects may have a different color depending on the angle of the view.*
   - *The volume density $\sigma(\boldsymbol{x})$ can be interpreted as the probability of a ray terminating at an infinitesimal particle at $\boldsymbol{x}$*

3. ***Volume rendering***: Finally we can use our colors and densities to generate a 2D image using classical volume rendering techniques.
   First we consider the general (continuous) case: Let's say we are given two known functions for our color and density, namely $\mathbf{c}(\mathbf{x}, \mathbf{d})$ and $\sigma(\mathbf{x})$. To compute the expected color for our ray, we compute

$$T(t) = \exp\left(- \int_{t_n}^{t_f} \sigma(\mathbf{r}(s))ds \right) \tag{1}$$

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt \tag{2}$$

   where $T(t)$ denotes the transmittance along the ray from t_n to t, i.e. the probability that the ray travels from t_n to t without hitting any other particle. $C(\mathbf{r})$ is the final accumulated color of a pixel.
   We numerically estimate the integrals above using our sampled points t_i. [1] We get

$$T_i = \exp\left(- \sum_{j=1}^{i-1} \sigma_j \delta_j \right) \tag{3}$$

$$C(\mathbf{r}) = \sum_{i=1}^{N} T_i \Big(1 - \exp(-\sigma_i \delta_i)\Big)\mathbf{c}_i \tag{4}$$

   with $\delta_i = t_{i+1} - t_i$ as the distance between the sample points.

To optimize the weights of the representation F_Θ, we simply compare the final image obtained by steps 1-3 and our ground truth values. As a loss function the MSE is used. All of these steps are visualized in figure 2.
For implementation details, see [13]

4 The Auto-NeRF architecture

Our goal is now to solve our previously defined problem using a generative model. A very prominent instance of generative models are Autoencoders, specifically Variational Autoencoders (VAEs). To successfully use this class of generative models in our context, we need to examine their respective strengths and weaknesses first. Then we can carefully combine them to a final architecture.

4.1 Variational Autoencoders

Assume that we have some training set $\mathcal{D}$, which in our case consists of image data. The standard Autoencoder is a pair of two deterministic functions (E_ϕ, G_θ), an encoder and a decoder respectively. The encoder maps data $x \in \mathcal{D}$ to some low-dimensional representation vector z, which is known as a latent variable or latent vector. The decoder then tries to reconstruct the input image as faithfully as possible, by utilizing the latent variable via $x \approx G_\theta(z)$. During training we are therefore trying to minimize the following problem

$$\theta^*, \phi^* = \arg\min_{\theta,\phi} \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} \mathcal{L}\Big(x, G_\theta \phi(x))\Big) \tag{5}$$

for some loss function $\mathcal{L}$, which is typically the MSE. The idea is then, that during training, the autoencoder learns to extract useful features from the training set due to its small bottleneck z. However, this formulation has the following problems:

1. ***We can't control the structure of the latent space***. This means, that there are regions in the latent space, which don't correspond to valid forms of data when decoded. It may also be true that data $x, x' \in \mathcal{D}$ which is close together in $\mathcal{D}$ is far apart in the latent space.

2. ***No easy way to generate data***. Since we have no knowledge of the latent space itself, we can't generate valid latent space vectors to generate new data using G_{θ^*}.

With Variational Autoencoders (VAEs) we address these problems, by regularizing the latent space. That way, we get a more structured latent space, and we are able to sample latent vectors from some prior distribution.
VAEs are rooted in Bayesian inference, and their goal is to model the underlying distribution of the data $p(x)$, so we can sample new data from that distribution. This means that instead of mapping an input $x \in \mathcal{D}$ to a deterministic latent variable z, we map it to a distribution $p(z|x)$.
Thus, the objective is to model the data in our trainig set $\mathcal{D}$, and we want to find

$$p(x) = \int_z p(x, z) dz = \int_z p(x|z) p(z) dz \tag{6}$$

The idea is now to infer $p(z)$ using $p(z|x)$. However, the expression above is intractable in practice, so instead we try to approximate $p(z|x)$ using a parametric function

$q_\phi(z|x)$, which can be understood as our encoder, while $p_\theta(x|z)$ is the decoder, for some parameters ϕ, θ.

During training we are trying to maximize the log-likelihood, which is however intractable. Instead, we are optimizing the Evidence Lower Bound (ELBO) instead:

$$\begin{aligned}
\log p(x) &= \mathbb{E}_{z\sim q_\phi(z|x)}[\log p(x,z) - \log q_\phi(z|x)] + \mathrm{KL}(q_\phi(z|x)||p(z|x)) && (7)\\
&\geq \mathbb{E}_{z\sim q_\phi(z|x)}[\log p(x,z) - \log q_\phi(z|x)] && (8)\\
&= \mathbb{E}_{z\sim q_\phi(z|x)}[\log p(x|z)] - \mathrm{KL}(q_\phi(z|x)||p(z)) && (9)\\
&=: \mathrm{ELBO} && (10)
\end{aligned}$$

The ELBO consists of two terms, the Kullback-Leibler (KL) divergence between the variational distribution $q_\phi(z|x)$ and the prior $p(z)$ and the expected conditional log-likelihood. The KL divergence forces the two input distributions to be the same, namely $q_\phi(z|x)$ and $p(x)$, which we assume to be a standard Gaussian $\mathcal{N}(0,1)$. This essentially regularizes the latent space.

4.1.1 Posterior Collapse

As mentioned in [7], the most consistent issue with VAE training is posterior collapse, in which the posterior distribution collapses torwards the uninformative prior.
Concretely, this means that there are components in the latent vector that get completely ignored in the worst-case, i.e.

$$\exists i : \forall x \in \mathcal{D} : q_\phi(z_i|x) \approx p(z_i) \tag{11}$$

The causes for this issue are well-understood for linear VAEs [7]. For non-linear VAEs, there are some recipes to counteract the effects of posterior collapse, and the most prominent one is KL-annealing.

To avoid this issue altogether, we chose a very small regularization-constant $\beta \approx 10^{-5}$ for the KL-term in the ELBO. This has the following consequences:

1. We lose all the nice regularization benefits that we get from VAEs. Our autoencoder can now focus on minimizing the reconstruction error, instead of trying to match the posterior $q_\phi(z|x)$ and the prior $p(z)$. After training, we get an aggregated posterior which may not be close to the prior.

2. We can't use our prior $p(z)$ anymore to generate new samples, so we can't use our autoencoder anymore as a generative model.

Luckily, a new and very powerful class of neural networks, called invertible neural networks, come to the rescue. We will use this class to address both of these problems. This invertible neural network will be used, to find a bijective mapping between samples from $q_\phi(z|x)$ and a prior $p(z)$.

4.2 Invertible neural networks (INNs) and Normalizing Flows

In this project we use the concept of Normalizing Flows to tackle the above problem, and implement it using a modified version of the RealNVP model [4].

4.2.1 Normalizing Flows

The idea of Normalizing Flows is to approximate some density by transforming a very simple one. These transformations are invertible and applied successively, and to transform one distribution to another, we use the change of variable formula.
We denote these transformations by f_i. These transformations have to be invertible, differentiable, and ideally simple.

Let's say that we have two distributions p_i, p_{i+1} and a transform t_i that transforms the distribution p_i to p_{i+1}, i.e. $z_i \sim p_i(z_i), z_{i+1} \sim p_{i+1}(z_{i+1})$ and $z_{i+1} = f_{i+1}(z_i), z_i = f_{i+1}^{-1}(z_{i+1})$. The change of variable formula then says

$$p_{i+1}(z_{i+1}) = p_i(f_{i+1}^{-1}(z_{i+1}))\left|\det \frac{\mathrm{d}f_{i+1}^{-1}(z_{i+1})}{\mathrm{d}z_{i+1}}\right| \tag{12}$$

$$= p_i(z_i)\left|\det \frac{\mathrm{d}f_{i+1}^{-1}(z_{i+1})}{\mathrm{d}z_{i+1}}\right| \tag{13}$$

$$= p_i(z_i)\left|\det \frac{\mathrm{d}f_{i+1}(z_i)}{\mathrm{d}z_i}\right|^{-1} \tag{14}$$

We take this scenario further, to K such transformations $t_1, ..., t_K$ which transform our initial distribution p_0, i.e. for $z_0 \sim p_0(z_0)$ we say $f_K \circ f_{K-1} \circ ... \circ f_1(z_0) = z_K := x$. To express our log-likelihood of the distribution of x, we can now apply this formula iteratively:

$$\log p(x) = \log p_k(z_k) = \log p_o(z_0) - \sum_{i=1}^{K} \log\left|\det \frac{\mathrm{d}f_i(z_{i-1})}{\mathrm{d}z_{i-1}}\right| \tag{15}$$

To make this computation as efficient as possible, the transformations f_i should satisfy two properties:

1. *The inverse function is easy to find*
2. *The Jacobian is easy to compute*

4.2.2 RealNVP Coupling layers

The Real-valued Non-Volume Preserving (RealNVP) model [4] satisfies these conditions. the transformations we use are so-called affine coupling layers. An affine coupling layer consists of three operations: splitting the input $z \in \mathbb{R}^D$ into two parts, one with the first d elements, called $z_{1:d}$ and one with the next $D-d$ elements, called $z_{d+1:D}$. We then apply a scale-transform $s : \mathbb{R}^d \to \mathbb{R}^{D-d}$ and a translation $t : \mathbb{R}^d \to \mathbb{R}^{D-d}$ on the first split, and finally combine them:

$$f(z) := \begin{cases} z_{1:d}, z_{d+1:D} = \text{split}(z) \\ z'_{d+1:D} = z_{d+1:D} \odot \exp\big(s(z_{1:d})\big) + t(z_{1:d}) \\ z' = [z_{1:d} \mid z'_{d+1:D}] \end{cases} \tag{16}$$

where $\odot$ means Hadamard-product or element-wise product, and $[\cdot \mid \cdot]$ simply concatenates the input vectors. We will now show that this transform fits our criteria.

1. ***The determinant of the Jacobian is cheap to compute***: Indeed, if we plug in our definition in equation 16, we get the lower triangular matrix

$$\mathbf{J} = \frac{\partial f(z)}{\partial z^T} = \frac{\partial z'}{\partial z^T} = \begin{bmatrix} \mathbb{I}_d & 0_{d\times(D-d)} \\ \frac{\partial z'_{d+1:D}}{\partial x_{1:d}^T} & \text{diag}(\exp(s(z_{1:d}))) \end{bmatrix} \tag{17}$$

   This makes the computation of the Jacobian very efficient.

$$\det(\mathbf{J}) = \prod_{j=1}^{D-d} \exp(s(z_{1:d}))_j = \exp\Big(\sum_{j=1}^{D-d} s(z_{1:d})_j\Big) \tag{18}$$

   Since the determinant of the Jacobian does not require any Jacobian of both s and t, both of these functions can be arbitrarily complex, and they don't need to be invertible themselves.

2. ***The mapping can be inverted easily***: We invert the operation in equation 16 by computing

$$f^{-1}(z') := \begin{cases} z'_{1:d}, z'_{d+1:D} = \text{split}(z') \\ z_{d+1:D} = \big(z'_{d+1:D} - t(z'_{1:d})\big) \odot \exp\big(-s(z'_{1:d})\big) \\ z = [z'_{1:d} \mid z_{d+1:D}] \end{cases} \tag{19}$$

   Note that to compute the inverse operation, we don't need the inverse of s and t. This means, that these two functions don't need to be invertible themselves.

4.2.3 Alternating Coupling Layers

As we can see though, some dimensions of the input remain unchanged, specifically $z_{1:d}$ of our input vector z. A trick we can use is to add another affine coupling layer which acts on $z_{1:d}$. Denote our first affine coupling vector by f_a and the second one by f_b. As it turns out, our nice properties from above survive this arrangement: Assuming that $f := f_b \circ f_a$ and that $f_a(z_a) = z_b$, we see that the Jacobian of the concatenation $f := f_b \circ f_a$ remains tractable:

$$\frac{\partial f}{\partial z_a^T}(z_a) = \frac{\partial (f_b \circ f_a)}{\partial z_a^T}(z) = \frac{\partial f_a}{\partial z_a^T}(x_a) \cdot \frac{\partial f_b}{\partial z_b^T}(z_b) \tag{20}$$

Using the fact that $\det(A \cdot B) = \det(A) \cdot \det(B)$, we see that the final determinant is just the product of the individual determinants.
Similarly, the concatenation is easily invertible:

$$f^{-1} = (f_b \circ f_a)^{-1} = f_a^{-1} \circ f_b^{-1} \tag{21}$$

We call these concatenated coupling blocks an alternating coupling layer. In figure 3 we see the conditional version of this function f.

4.3 Conditional Invertible Neural Networks (cINNs)

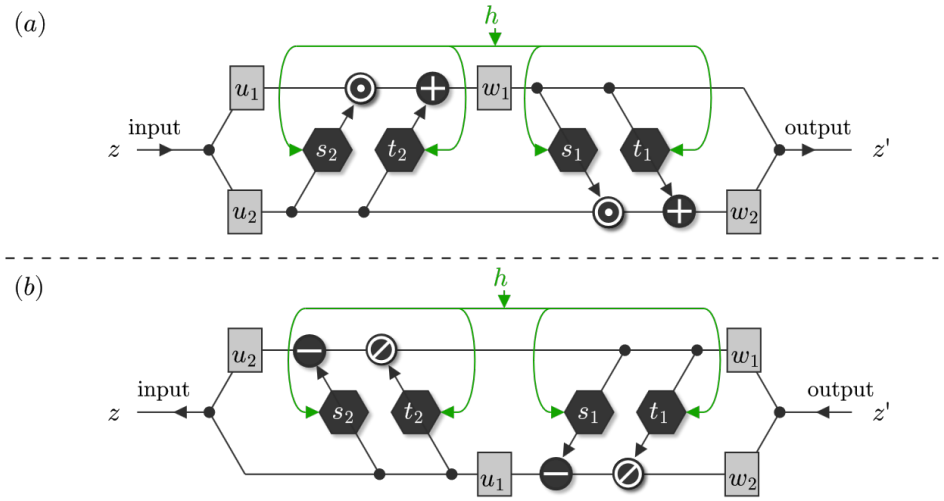

Figure 3: The depiction of the forward-pass (a) and the backward-pass (b) of the conditional alternating coupling layer. The function $H(c) = h$ produces a conditional code, which is concatenated with the inputs of the scale transforms s_i and the translations t_i. The final result is then produced according to equation 22. Conversely, the backward pass corresponds to equation 23.

In this project we use an extension of the just introduced alternating coupling layer f_i, to realize a conditional mapping $f_i(\cdot|\cdot)$. From now on we also assume, that the split-function partitions the input $z \in \mathbb{R}^D$ into two equal parts, i.e. $d = D/2$.

To make this mapping conditional, we also need a conditional input, which is produced by a conditioning function $H(c) = h$. The job of this function is to pre-process the conditional input c, for example to reduce its dimension or to disentangle the components. Finally, this
We call $z_{1:d} = u_1$ and $z_{d+1:D} = u_2$ to make operations more readable. The forward-pass is then

$$f(z|h) := \begin{cases} u_1, u_2 = \text{split}(z) \\ w_1 = u_1 \odot \exp\big(s_2([u_2 \mid h])\big) + t_2([u_2 \mid h]) \\ w_2 = u_2 \odot \exp\big(s_1([w_1 \mid h])\big) + t_1([]w_1 \mid h]) \\ z' = [w_1 \mid w_2] \end{cases} \tag{22}$$

The backward-pass is similarly

$$f^{-1}(z'|h) := \begin{cases} w_1, w_2 = \text{split}(z') \\ u_2 = \big(w_2 - t_1([w_1 \mid h])\big) \odot \exp\big(-s_1([w_1 \mid h])\big) \\ u_1 = \big(w_1 - t_2([u_2 \mid h])\big) \odot \exp\big(-s_2([u_2 \mid h])\big) \\ z = [u_1 \mid u_2] \end{cases} \tag{23}$$

These operations are depicted in Figure 3.

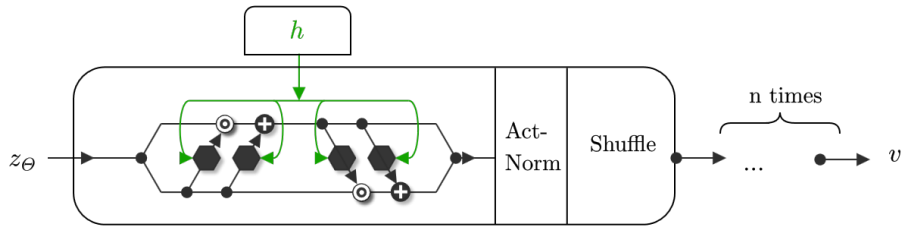

Figure 4: A conditional block, which we use to build the cINN. A block consists of a conditional coupling layer, an activation normalization layer and a shuffling layer. The cINN is simply n consecutive blocks.

Finally, we define a conditional block by first adding our just defined conditioned alternating coupling layer, then adding an ActNorm layer and a Shuffle layer.

- The ActNorm- layer acts very similar to a BatchNorm, however, instead of normalizing the input w.r.t the mean and the variance of the input itself, we normalize w.r.t to the first input-batch.
- The Shuffle-layer simply shuffles the input by a given permutation.

This setup is seen in figure 4.
A conditional invertible neural network is simply the serial concatenation of such conditional coupling blocks: Let $f_1, ..., f_K$ be conditional coupling blocks, and assume that we have a conditional input $H(c) = h$. A conditional invertible neural network is then

$$\tau(x|h) = f_K \circ f_{K-1} \circ ... \circ f_1(x|h) \tag{24}$$

4.4 Final Architecture

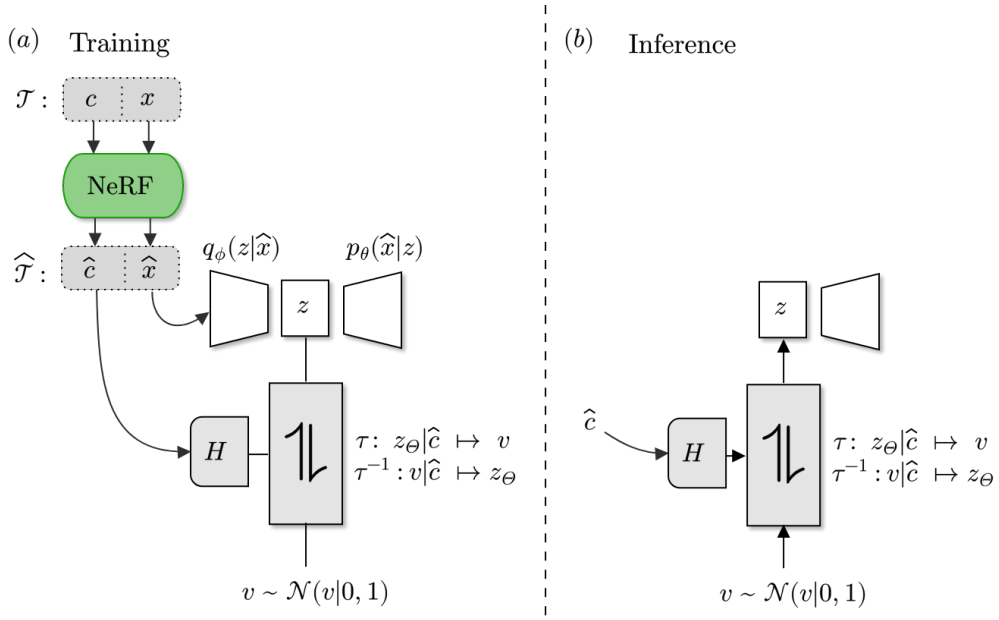

Figure 5: The architecture of our model. (a): Initially we have a small dataset $\mathcal{T}$, which we enlarge to a dataset $\hat{\mathcal{T}}$ using a trained NeRF model. Using this new training set, we can train our VAE and our cINN. (b): To infer new latent codes for the VAE, we sample from a prior distribution $\mathcal{N}(0,1)$ and generate a camera pose $\hat{c}$ we want. This gets translated to a new latent code by the cINN. Finally, we can decode the generated latent code.

Let's get back to solving our problem. Assume that we have a small training set of images $\mathcal{T}$ as described in 2.1. The models we have introduced until now will come into play as follows:

- As we have seen, NeRF can learn a scene representation with very little training data. A big drawback however, is the time to render a new image using this representation.
  However, we can use it to generate new, high quality data for the new training set $\hat{\mathcal{T}}$.

- We can train a VAE on this new training set $\hat{\mathcal{T}}$, since autoencoders need a lot of training data. To circumvent the problem of posterior collapse in the VAE training, we train with a small KL-weight as described in 4.1.1.

- To generate new latent vectors, we train a cINN to translate between samples from $q_\phi(z|x)$ and a simple distribution, namely $\mathcal{N}(0,1)$. The conditional code is in this case the camera pose.
  We will elaborate this step in the next subsection.

4.4.1 Training the cINN

In an ideal world, we could just train a MLP to map a given camera pose c to a latent code z of our VAE. Unfortunately, this is not possible, since the latent code contains much more information than our camera pose.
However, not all hope is lost when we approach this problem with a probabilistic framework: Obtaining a plausible latent code for a given camera pose c is best described as sampling from $p(z|c)$. Our goal is now to model this process with a normalizing flow τ, which is implemented as a cINN $\tau(\cdot|\cdot)$. We are now transforming samples from $z \sim q_\phi(z|x)$ to $v \sim \mathcal{N}(0,1)$ by

$$z = \tau(v|c) \tag{25}$$

By construction this mapping is invertible, so the value $v = \tau^{-1}(z|c)$ exists. Now it remains to derive a learning task which ensures that information of c is discarded in v, which can be formalized as follows: We can consider the residual v as a result of the followig stochastic process: $v = \tau^{-1}(z|c)$ with $z, c \sim p(z, c)$. Then v discards all information of c, if v and c are independent. Therefore, we are trying to minimize the distance between the distribution $p(v|c)$ and our prior $q(v) = \mathcal{N}(0,1)$.
Using the invertibility of τ, we can write out our objective function

$$\mathbb{E}_c \mathrm{KL}(p(v|c)||q(v)) = \mathbb{E}_{z,c}\Big[-\log q\big(\tau^{-1}(z|c)\big) - \log \Big| \det \mathbf{J}_{\tau^{-1}}(z|c) \Big| \Big] - H(z|c) \tag{26}$$

Note that the conditional entropy term on the right can be ignored during training, because it is constant. As we have seen in the previous sections, the log-det term in (26) is simply the sum of the intermediate couplings. The architecture and the training is visualized in figure 5 (a).
Remark that our conditional input of our cINN is not conditioned, i.e. $H(c) = c$. This is due to the fact, that the camera pose is already in a disentangled form and the dimension is small enough.

4.4.2 Generating new latent codes

Now we can generate new latent codes easily. Let's say, that we have chosen a camera pose and want to render it. We call this pose $\hat{c}$. Now we sample $v \sim \mathcal{N}(0,1)$ and compute $z = \tau(v|\hat{c})$. Now we can simply decode the new latent code using our decoder, as shown in figure 5 (b).

4.4.3 Experiment: pose estimation

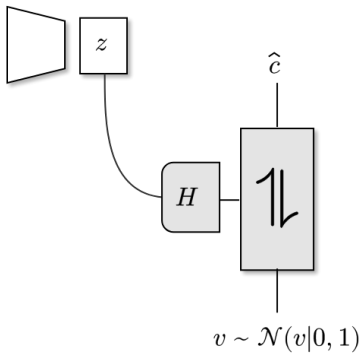

Figure 6: Comparison of NeRF and AutoNeRF over various datasets. As we can see, NeRF is better at encoding high frequency details. As we can see, AutoNeRF is able to generate plausible novel views of our scene.

A nice feature of our architecture is that we can slightly modfiy it to get an estimator for our camera pose. Given a trained VAE, we encode it to get a latent vector $z \in \mathbb{R}^{64}$. Our approach is very similar to the previous derivation: In our case, we want to estimate the camera pose $\hat{c}$ for a given input image. This can also be realized as a normalizing flow, where we have the translation $\hat{c} = \tau(v|z) \iff v = \tau^{-1}(\hat{c}|z)$, where $v \sim \mathcal{N}(0,1)$. Our training objective is then

$$\mathbb{E}_z \mathrm{KL}(p(v|z)||q(v)) = \mathbb{E}_{\hat{c},z}\Big[-\log q\big(\tau^{-1}(\hat{c}|z)\big) - \log \Big| \det \mathbf{J}_{\tau^{-1}}(\hat{c}|z) \Big| \Big] - H(\hat{c}|z) \quad (27)$$

To then estimate a camera pose, we simply encode the image to get a latent vector z, then we sample $v \sim \mathcal{N}(0,1)$ and compute $\hat{c} = \tau(v|z)$

5 Ablation study

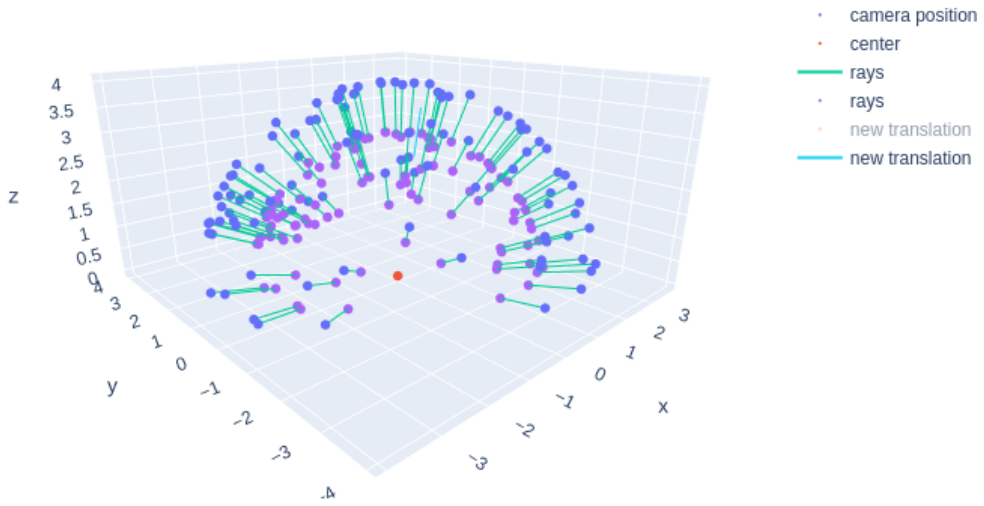

Figure 7: The distribution of camera positions in the lego-bulldozer dataset. Note that the cameras are all focused on the center point (marked red), and are also all lying on a sphere. Even though this distribution works well for NeRF, it doesn't work for our model.

In this section we will take a closer look at the sampling scheme when we generate the new dataset $\hat{\mathcal{T}}$ as described in figure 5. To do this, we conduct two experiments:

1. ***Keep the radius constant*** When we generate our dataset with the setup in figure 7, i.e. keeping the distance to the center constant, our model becomes too 'stiff'. Our model then only works well when we stay on this sphere, but when we want to increase the radius, the model simply ignores the changed radius and stays on the spere.

2. ***Vary the radius*** To mitigate the problem above, we vary the radius by sampling from $\mathcal{U}[a, b]$, for some $a, b > 0$ depending on the dataset. For the lego-dataset we chose $\mathcal{U}[a, b] = \mathcal{U}[r - 1, r + 1]$ where $r \approx 4$ is some offset radius. This makes the model more flexible, and we can even generate plausible frames even when we are outside $[r - 1, r + 1]$.

Both of these scenarios are presented in our video [1]. The key point here is the following: We have to do all of this, because our model does not exploit the underlying 3D-structure of our data. This is further discussed in the conclusion.

[1] video-link for the ablation study `https://youtu.be/lzlAp05DhwQ`

6 Conclusion

In this practical, we addressed the problem of quick view synthesis based on little training data. Our architecture is able to render novel views in real-time. Even though the results of our architecture are very close to the state-of-the-art models, there are many aspects of the architecture which must be improved in order to make it more robust. We have outlined these areas below.

6.1 Further research

- ***Stability***: As we can see in the video sample, is that the movement in space is a bit shaky. This is due to the fact that we are sampling from a conditional distribution per frame, and we don't enforce consistency between frames.
  A very important point is the stability of NeRF. A recent paper [11] showed that deep learning is typically unstable for inverse problems. However, it is unclear if NeRF is affected by these instabilities.

- ***Image quality***: A common phenomenon in VAEs is the lack of sharpness in the final image. Many new architectures have been proposed to tackle this problem, which most of them are hierarchical VAEs such as VQ-VAE2 [10].

- ***Inductive bias***: A nice feature of the NeRF architecture is that it utilizes the underlying structure of the data. It uses the fact that the images in our training set are simply projections of a 3D scene, which is reflected in the training process. This inductive bias makes the model very powerful and efficient. However, such an inductive bias is hard to construct for our VAE, which obviously makes no such assumptions about our data.
  Our ablation study shows, that the model is dependent on good data be able to generalize to new viewpoints. A model with a strong inductive bias, such as NeRF wouldn't be affected too much.

- ***Performance***: It is worth mentioning, that this model is 10 times quicker than NeRF. Even though we can produce novel views very rapidly using this architecture, the training process takes a long time. A more unified architecture could address this problem.

7 Appendix

7.1 Appendix 0 - Useful results from probability theory and statistics

Definition 0.1 (Kullback-Leibler Divergence)**.** *Let P, Q be two continuous probability distributions, with probability densities p, q respectively. The Kullback-Leibler Divergence is defined as*

$$KL(P||Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx \tag{28}$$

Analogously, if our distributions P, Q are discrete and defined over a probability space $\mathcal{X}$, we define it as

$$KL(P||Q) = \sum_{x \in \mathcal{X}} p(x) \log \frac{p(x)}{q(x)} dx \tag{29}$$

Definition 0.2 (Conditional Entropy)**.** *Let X and Y be random variables with supports $\mathcal{X}, \mathcal{Y}$ respectively. The conditional entropy of X given Y is given by*

$$H(X|Y) = -\sum_{x \in \mathcal{X}, y \in \mathcal{Y}} p(x, y) \log \frac{p(x, y)}{p(x)} \tag{30}$$

Theorem 1 (Change of variable formula)**.** *Let P, Q be two continuous distributions with densities p, q respectively. Let $g : y \mapsto x$ be a bijective function. Then the densities are related by*

$$p(x) = q(g^{-1}(x)) \det \left| \frac{\partial g^{-1}(x)}{\partial y} \right| \tag{31}$$

Remark. *Note the following facts:*

- *Note that the same is true for conditional densities, which is used heavily in normalizing flows.*
- *The inverse function theorem is a special case of this formula*

7.2 Appendix A - Implementation details

7.2.1 A.3 VAE architecture

Conv2d(in, out, kernel=(3,3), stride=(2,2), padding=(1,1))
BatchNorm2d(in)
LeakyReLU(negative_slope=0.01)
EncoderBlock

Table 1: EncoderBlock

ConvTranspose2d(in, out, kernel=(3,3), stride=(2,2), padding=(1,1))
BatchNorm2d(in)
LeakyReLU(negative_slope=0.01)
DecoderBlock

Table 2: DecoderBlock

Linear(64, 4096)
DecoderBlock(1024, 512)
DecoderBlock(512, 256)
DecoderBlock(256, 128)
DecoderBlock(128, 64)
DecoderBlock(64, 32)
DecoderBlock(32,32)
Conv2d(32,3)
Tanh()
Decoder

EncoderBlock(3,32)
EncoderBlock(32,64)
EncoderBlock(64, 128)
EncoderBlock(128, 256)
EncoderBlock(256, 512)
EncoderBlock(512, 1024)
2x Linear(4096, 64)
Encoder

Table 3: Encoder and Decoder

7.3 Appendix B - Derivation of the cINN loss

For the sake of completeness, we show the derivation of the training objective. The derivation is completely analogous to [15].

$$\begin{aligned}
\mathrm{KL}(p(v|c)||q(v)) &= \int_v p(v|c) \log \frac{p(v|c)}{q(v)} dv \\
&= \int_z p\big(\tau^{-1}(z|c)|c\big) \big| \det \mathbf{J}_{\tau^{-1}}(z|c) \big| \log \frac{p\big(\tau^{-1}(z|c)|c\big) \big| \det \mathbf{J}_{\tau^{-1}}(z|c) \big|}{q\big(\tau^{-1}(z|c)\big) \big| \det \mathbf{J}_{\tau^{-1}}(z|c) \big|} dz \\
&= \int_z p\big(\tau^{-1}(z|c)|c\big) \big| \det \mathbf{J}_{\tau^{-1}}(z|c) \big| \log \frac{p\big(\tau^{-1}(z|c)|c\big)}{q\big(\tau^{-1}(z|c)\big)} dz \qquad (32)
\end{aligned}$$

Now we can express $p(v|c)$ in terms of $p(z|v)$ by utilizing the invertibility of τ and the fact that $v = \tau^{-1}(z|c) \iff z = \tau(v|c)$, we get

$$p(v|c) = p(\tau(v|c)) \big| \det \mathbf{J}_{\tau}(v|c) \big| \qquad (33)$$

By the inverse function theorem, we get

$$p(\tau^{-1}(z|c)|c) = p(z|c) \big| \det \mathbf{J}_{\tau^{-1}}(z|c) \big|^{-1} \qquad (34)$$

By plugging (34) into (32) we obtian

$$\begin{aligned}
\mathrm{KL}(p(v|c)||q(v)) &= \int_z p\big(z|c\big) \log \frac{p(z|c)}{q\big(\tau^{-1}(z|c)\big) \big| \det \mathbf{J}_{\tau^{-1}}(z|c) \big|} dz \qquad (35) \\
&= \mathbb{E}_{z \sim q(z|c)} \Big[- \log q(\tau^{-1}(z|c)) - \log \big| \det \mathrm{J}_{\tau^{-1}}(z|c) \big| + \log p(z|c) \Big] \qquad (36)
\end{aligned}$$

7.4 Appendix C - Samples

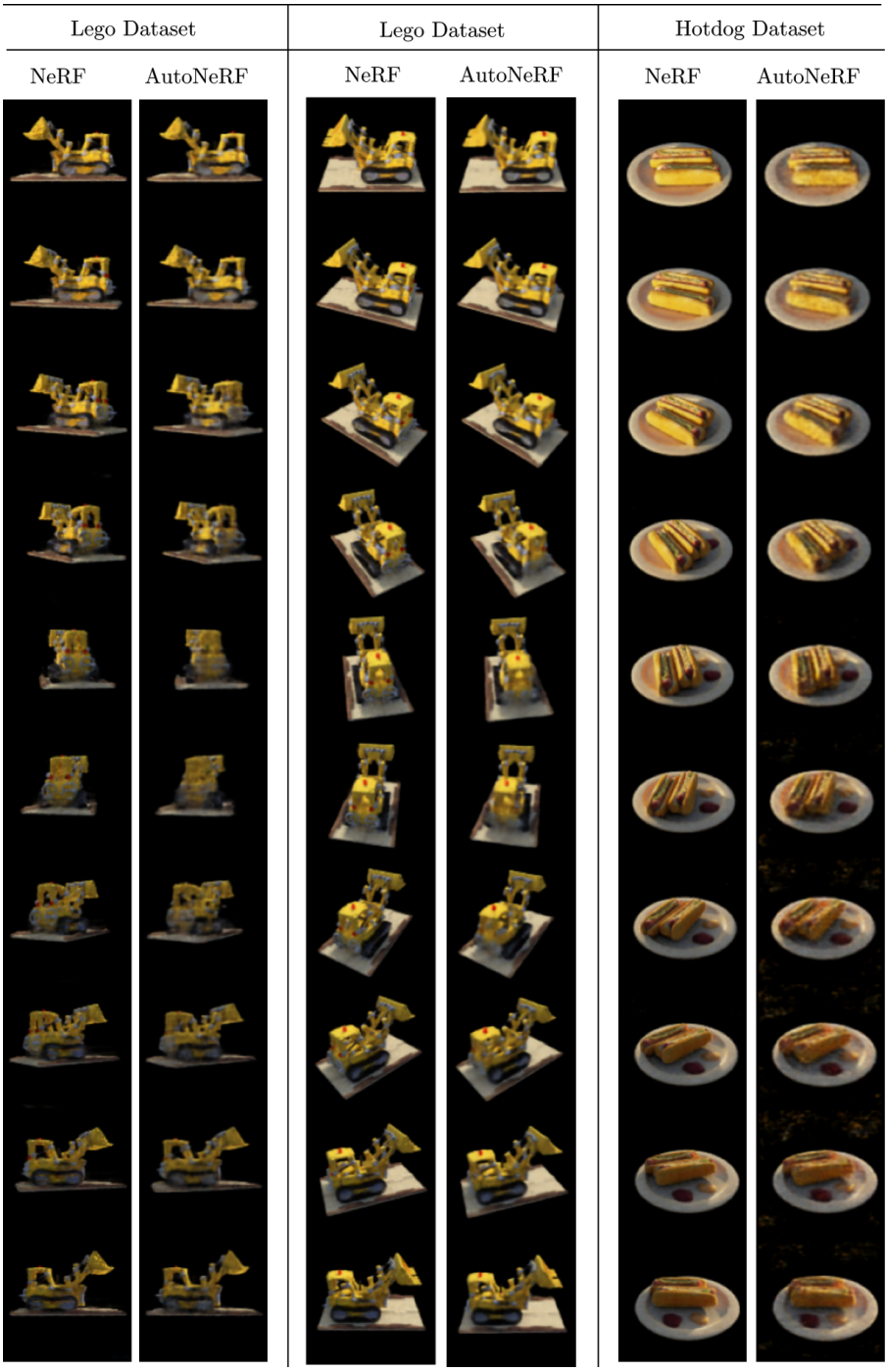

Figure 8: Comparison of NeRF and AutoNeRF over various datasets. As we can see, NeRF is better at encoding high frequency details. As we can see, AutoNeRF is able to generate plausible novel views of our scene.
For a animated sample, see `https://youtu.be/1ovewzz6u78`

References

[1] N. Max. "Optical models for direct volume rendering". In: *IEEE Transactions on Visualization and Computer Graphics* 1.2 (1995), pp. 99–108. DOI: 10.1109/2945.468400.

[2] Levoy M. and Hanrahan P. "Light field rendering". In: 1996.

[3] Angel X. Chang et al. *ShapeNet: An Information-Rich 3D Model Repository.* 2015. arXiv: 1512.03012 [cs.GR].

[4] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. "Density estimation using Real NVP". In: 2017. URL: https://arxiv.org/abs/1605.08803.

[5] Philipp Henzler et al. "Single-image Tomography: 3D Volumes from 2D X-Rays". In: *Computer Graphics Forum* 37 (Oct. 2017). DOI: 10.1111/cgf.13369.

[6] Tzu-Mao Li et al. "Differentiable Monte Carlo Ray Tracing through Edge Sampling". In: *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37.6 (2018), 222:1–222:11.

[7] James Lucas et al. "Understanding Posterior Collapse in Generative Latent Variable Models". In: *Deep Generative Models for Highly Structured Data, ICLR 2019 Workshop, New Orleans, Louisiana, United States, May 6, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=r1xaVLUYuE.

[8] Lars Mescheder et al. "Occupancy Networks: Learning 3D Reconstruction in Function Space". In: *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*. 2019.

[9] Jeong Joon Park et al. *DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation.* 2019. arXiv: 1901.05103 [cs.CV].

[10] Ali Razavi, Aaron van den Oord, and Oriol Vinyals. *Generating Diverse High-Fidelity Images with VQ-VAE-2.* 2019. arXiv: 1906.00446 [cs.LG].

[11] Nina M. Gottschling et al. *The troublesome kernel: why deep learning for inverse problems is typically unstable.* 2020. arXiv: 2001.01258 [cs.LG].

[12] Chiyu Max Jiang et al. *Local Implicit Grid Representations for 3D Scenes.* 2020. arXiv: 2003.08981 [cs.CV].

[13] Ben Mildenhall et al. "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". In: *ECCV*. 2020.

[14] Songyou Peng et al. "Convolutional Occupancy Networks". In: *European Conference on Computer Vision (ECCV)*. Cham: Springer International Publishing, Aug. 2020.

[15] Robin Rombach, Patrick Esser, and Björn Ommer. "Network-to-Network Translation with Conditional Invertible Neural Networks". In: 2020.

[16] Ayush Tewari et al. *State of the Art on Neural Rendering.* 2020. arXiv: 2004.03805 [cs.CV].

[17] Patrick Esser, Robin Rombach, and Björn Ommer. In: URL: https://compvis.github.io/iin/.