DTU Compute
Department of Applied Mathematics and Computer Science

Semantically Guided Volumetric Object Placement

Master Thesis

Lucas Alexander Sørensen

Semantically Guided Volumetric Object Placement

-

Master Thesis
June, 2022

By
Lucas Alexander Sørensen

Cover photo: Vibeke Hempler, 2012
Published by: DTU, Department of Applied Mathematics and Computer Science, Building 324, 2800 Kgs. Lyngby Denmark
www.compute.dtu.dk
ISSN: [0000-0000] (electronic version)
ISBN: [000-00-0000-000-0] (electronic version)

ISSN: [0000-0000] (printed version)
ISBN: [000-00-0000-000-0] (printed version)

Approval

Lucas Alexander Sørensen - s174461

.....

Signature

.....

Date

Abstract

Recent advances in computer vision research have made it possible to produce photo-realistic 3D volumetric reconstructions of complex scenes based on datasets of 2D images - this includes the ability to manipulate specific objects in these scenes. Significant progress has also been made in the field of vision-language models, which have effectively been able to capture many different semantic relationships between text captions and image data, and have paved the way for text-based editing frameworks. This project seeks to bridge the gap between these two areas of research, and proposes a novel method which can perform manipulations of objects in photo-realistic 3D scenes that are semantically consistent with a given text prompt.

The code for the project is available at github.com/lucasalexsorensen/sgvop

Contents

1 Introduction

1.1 Background and motivation

In recent years, there has been some remarkable advances in the field of differentiable volumetric rendering. This has made it possible to faithfully reconstruct complex 3D scenes in a photo-realistic manner. Specifically, a lot of work has been made based on Neural Radiance Fields[1] (also called NeRFs). As such, many extensions of NeRFs have been made already. Quite a few of these extensions revolve around improving the speed and efficiency of NeRFs[2], but there are also extensions that deal with NeRF manipulation. In [3], they present a novel method for disentangling foreground objects from their background scenes, which allows for fine-grained control of how the 3D scenes are composed.

Meanwhile, in a different area of research, there has also been made remarkable strides. Namely, vision-language models have seen significant improvements with the advent of CLIP[4]. The concept of building models that connect text-image pairs has proven to be quite effective, and the models have been performing especially well on zero-shot image classification tasks, and play a significant role in modern text-conditional image generation models (e.g. [5]).

The main idea of this project is to combine the aforementioned NeRF manipulation techniques with CLIP models in order to manipulate objects in 3D volumetric scenes using semantic prompts - to this end, the concept of semantically guided volumetric object placement is established. Given a guiding text prompt, the framework should determine how to perform the most appropriate NeRF manipulation. A concrete example could be to guide a NeRF manipulation to properly relocate a disentangled coffee cup given the text prompt ”a mug on a table”. This is a powerful concept which could be essential for many virtual reality applications.

1.2 Related work

Control-NeRF[6]. By maintaining a discrete feature volume grid, this method is capable of editing and mixing NeRF scenes. Concretely, this is done by manipulating the feature volume grid. However, it can be difficult to perform very precise manipulations, and semantically guided manipulations are not supported.

CLIP-NeRF[7]. This method works by using latent codes to independently represent shapes (geometry) and appearances (color). These latent codes are what drive the manipulation, and can be obtained from CLIP vectors (i.e. guided by either text or image). The technique is incapable of simply transforming objects in the context of scenes (e.g. moving and rotating the object). Also, the paper only presents results from synthetic datasets containing primitive cars and chairs (i.e. not full scenes),

LaTeRF[8]. An object extraction method which uses both pixel annotations and semantic guidance in order to manipulate NeRF scenes. The rendering loss now incorporates the an ”objectness” probability for the points in 3D space, which guides the object disentanglement. The objectness probabilities are determined using the aforementioned pixel-level annotations. Additionally, a CLIP loss term is added to assist in filling occluded parts of the scene after extraction using a provided textual prompt. The method mostly works for extracting objects, and does not appear to extend to object manipulation (i.e. scaling, rotating, translating).

1.3 Structure

The project is structured according to an ”IMRaD” structure - i.e. Introduction, Methods, Results and Discussion (Conclusion).

Introduction (chapter 1) seeks to briefly summarize the current state of research in the field(s), and consequently explain why the research is meaningful.

Methods (chapter 2) aims to succinctly present the most essential theory and specific methods that were used in the experiments of the project.

Results (chapter 3) presents the findings of the project. Chiefly, this chapter is dedicated to presenting experiment results. Also, for brevity, at the end of each subchapter is a small summary of what was obtained in the results.

Conclusion (chapter 4) seeks to make sense of the obtained results, and also seeks to address the limitations of the study, and briefly proposes suggestions for future research.

2 Theory and methods

This section aims to briefly go over the relevant theory and specific methods that were used throughout the project work.

2.1 Neural Radiance Fields

2.1.1 Vanilla NeRF

Neural Radiance Fields[1] (or NeRFs) is a method which achieves state-of-the-arts results within the field of novel view synthesis - i.e. given a set of images of a scene from different viewpoints, it can generate images from previously unseen perspective, which is essentially

At the core of the method lies the *continuous volumetric scene function*:

$$(x, y, z, \theta, \phi) \rightarrow (R, G, B, \sigma) \tag{2.1}$$

This scene function takes a spatial location (x, y, z) and a viewing direction (θ, ϕ) and returns a volume density (σ) along with a color (R, G, B).

The main objective is to most effectively approximate this scene function for a specific scene. In practice, the underlying approximation is done by a fully-connected MLP (denoted F_Θ). Given a set of input images, the first step is to estimate the camera poses using a photogrammetric technique such as structure-from-motion[1]. Then, the idea is to essentially march rays from the estimated cameras and sample points along them (see 2.1). The original pixel values are used to optimize the weights of F_Θ with respect to a *rendering loss* (i.e. the network is fitted such that it most faithfully reconstructs the input images).

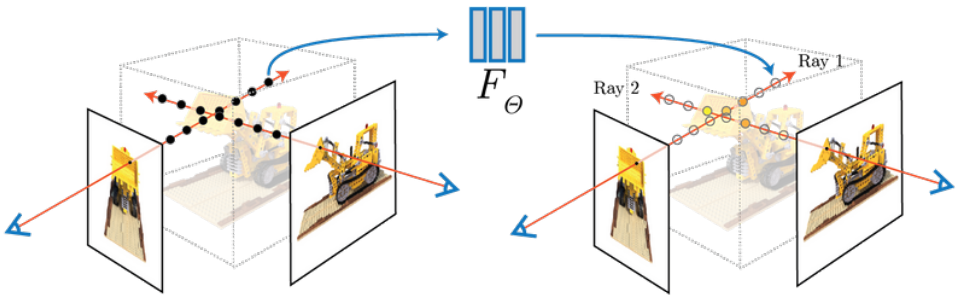

Figure 2.1: Taken from [1]. Points are sampled along the camera rays. F_Θ then returns a density (σ) and color $((R, G, B))$ for every point. The camera rays are obtainable due to the fact that the camera poses for the input images are known.

An essential part of NeRFs is the aforementioned rendering loss. The idea is to take the density and color outputs from F_Θ and reduced it to a single pixel value, such that they can be compared against the ground truth (i.e. the color for the corresponding pixel in the original image data). This concept is illustrated in figure 2.2.

[1] urlhttps://colmap.github.io

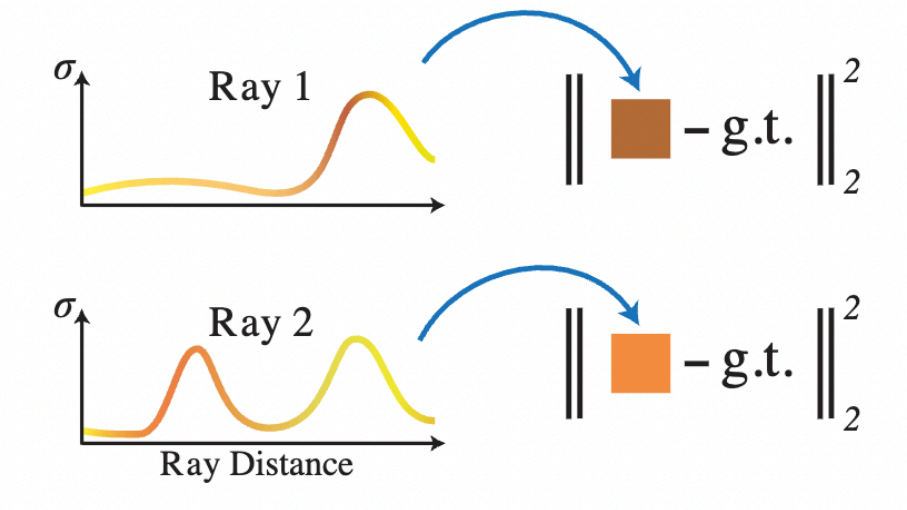

Figure 2.2: Taken from [1]. Rendering loss. Using volume rendering techniques, the density and color estimates (seen on the left) are reduced to a single pixel value, which is compared against the ground truth pixel value.

The mechanism for reducing the ray densities and colors a single color value is inspired by classic volume rendering techniques. Formally, given the ray **r** with points $\mathbf{r}(t) = \mathbf{o}+t\mathbf{d}$, and the near/far bounds t_n and t_f (i.e. where the rays start, and how far the rays should extend), the continuous volume rendering formula can be stated as follows:

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt \tag{2.2}$$

$$T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right) \tag{2.3}$$

The function $T(t)$ is also called the *accumulated transmittance* along the ray from t_n to t, i.e. the probability that the rays travels from t_n to t without hitting any other particle.

In practice, the formula is numerically estimated using quadrature. This yields the following formulation:

$$\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i \left(1 - \exp\left(-\sigma_i\delta_i\right)\right)\mathbf{c}_i \tag{2.4}$$

$$T_i = \exp\left(-\sum_{j=1}^{i-1}\sigma_j\delta_j\right) \tag{2.5}$$

$$\delta_i = t_{i+1} - ti \tag{2.6}$$

The most important detail regarding the volume rendering formulae is the fact that they are trivially differentiable. This is what makes it possible to use them as loss functions for optimizing the weights of F_Θ using stochastic gradient descent.

According to the authors, a big part of NeRF's success is due to the use of "Fourier features"[9]. It turns out that passing the input coordinates $xyz\theta\phi$ through the network F_Θ resulted in poor performance when representing high-frequency variation in color and

geometry, since deep neural networks are biased towards learning lower frequency functions. A solution to this problem is to map the coordinates to a higher dimensional space using some encoding function. In the original NeRF paper, the following encoding is used:

$$\gamma(p) = \big(\sin(2^0\phi p), \cos(2^0\pi p), ..., \sin(2^{L-1}\phi p), \cos(2^{L-1}\pi p)\big) \tag{2.7}$$

Equation 2.7 maps an input coordinate from $\mathbb{R}$ to $\mathbb{R}^{2L}$, and is actually quite similar to how the positional encoding is performed in [10] (although it is used for a different purpose).

To ensure volumetric consistency (e.g. the density σ at point $[x, y, z]$ is the same no matter the viewing direction $[\theta, \phi]$), the network F_Θ is regularized by returning σ before adding the viewing direction as a network input. As such, the density is now estimated by a spatial density function (i.e. depends only on $[x, y, z]$), whereas the the color (R, G, B) is estimated by a spatiodirectional function (i.e. depends on $[x, y, z, \theta, \phi]$. The architecture containing this constraint is shown in figure 2.3

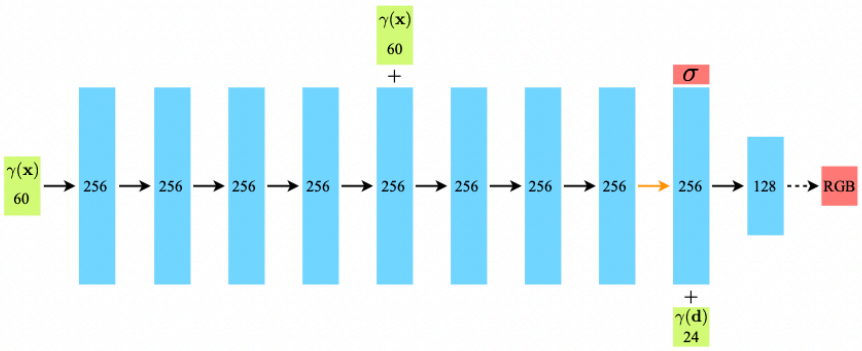

Figure 2.3: Taken from [1]. Architecture of F_Θ. Input vectors are green. Hidden layers are blue. All layers are fully-connected layers. Black arrows indicate ReLU, orange arrows indicate no activation, dashed arrows indicate sigmoid activation. "+" indicates vector concatenation. **x** denotes the 3D position, **d** denotes the viewing direction.

An ablation study was performed in the NeRF paper in order to determine the effectiveness of the different components. This can be seen in figure 2.4.

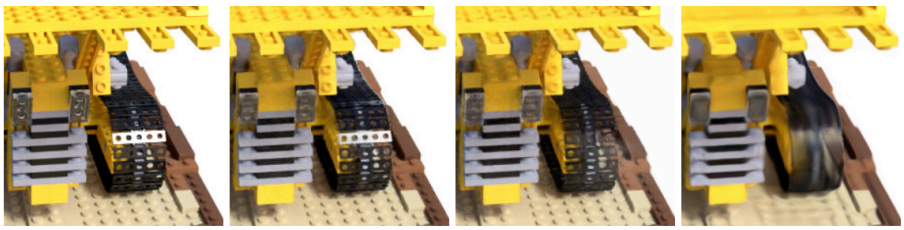

Figure 2.4: Taken from [1]. NeRF ablation study. "View dependence" refers to the notion of incorporating viewing directions (i.e. camera ray directions) in F_Θ. "Positional encoding" refers to the notion of applying the sin / cos-encoding to input coordinates. Removing view dependence prevents the model from recreating the specular reflection on the bulldozer tread. Removing the positional encoding inhibits the model in terms of representing high frequency geometry and colors, resulting in an oversmoothed image.

Recently, a huge leap was made in optimizing the efficiency of NeRFs. The training time for a vanilla NeRF is in the range of 12-24 hours - with the work done in [2], the training time is in the order of minutes instead. The main idea of the paper is the introduction of ”multiresolution hash encoding”, which essentially works by trilinearly interpolating multiple discrete 3D grids of trainable embeddings.

In short, the authors of [2] largely attribute the speed gains to using a smaller MLP (smaller in both width/depth). This downsize is made possible by the fact that the sin / cos-based positional encoding is replaced with the more powerful multiresolution hash encoder. Another contributing factor is the fact that highly efficient CUDA kernels were developed specifically for the project.

2.1.2 NeRF disentanglement

Recently, a framework for volumetrically disentangling objects from the background of NeRF scenes was proposed in [3].

In a standard NeRF setup, a dataset of images (and camera poses) is given. Now, a set of 2D masks for the object of interest should also be provided. The masks are used to apply a *masked* rendering loss. An illustration of this can be seen in figure 2.5.

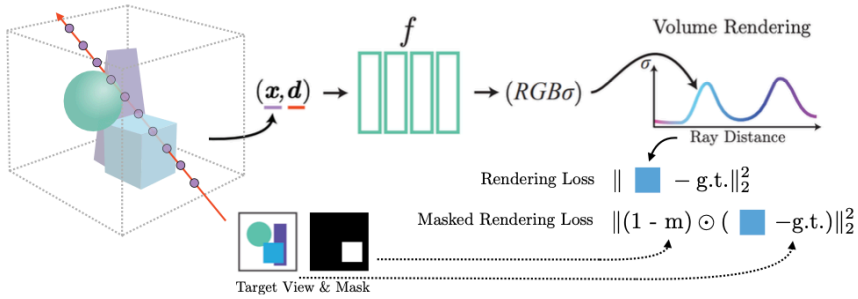

Figure 2.5: Taken from [3]. Masked rendering loss.

Thus, two NeRFs are trained in total - the full scene (i.e. regular rendering loss), and the background scene (i.e. masked rendering loss). The idea is then to obtain the foreground scene by subtracting the background scene from the full scene. This is performed according to the following formulae:

$$c_r^{fg} = \sum_{i=1}^{N} w_{fg}^i \cdot c_{fg}^i \tag{2.8}$$

$$w_{fg}^i = w_{full}^i - w_{bg}^i \tag{2.9}$$

$$c_{fg}^i = c_{full}^i - c_{bg}^i \tag{2.10}$$

In other words, the weights and colors are subtracted from each other separately before being multiplied together to obtain the foreground scene.

An excerpt of the results reported in [3] can be seen in figure 2.6.

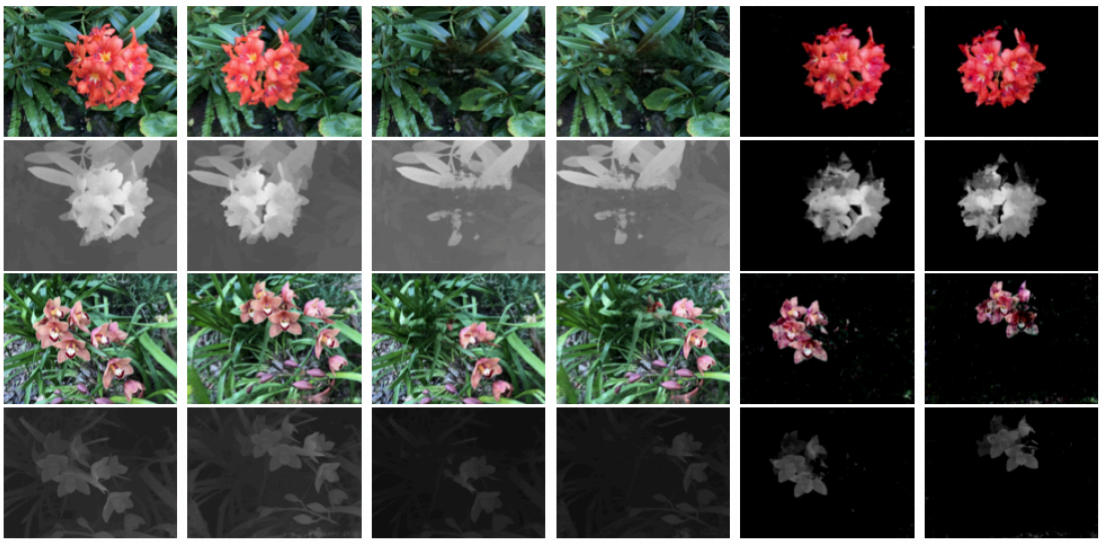

Figure 2.6: Taken from [3]. Example results for foreground object disentanglement.

The work also shows examples of how disentangled foreground objects can be re-inserted into the background scenes after applying some type of transformation. This can be seen in figure 2.7.

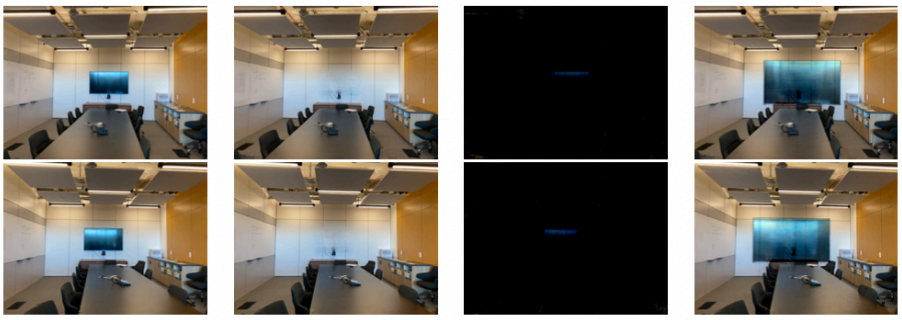

Figure 2.7: Taken from [3]. Re-insertion of a transformed foreground object. The TV has been scaled before being re-inserted into the original scene, and still appears to respect illumination. Thus, the scene itself still retains its photo-realistic appearance.

2.1.3 NeRF manipulation

The goal is to be able to scale, rotate and transform the objects freely in 3D space. As such, additional techniques for transforming disentangled foreground objects are needed. The main idea behind the approach taken for NeRF manipulation in this project is to apply an affine transformation to the camera rays before sending them through NeRF.

Let's start with translation. Let the vector $\mathbf{t} = [\delta_x, \delta_y, \delta_z]$ denote the desired translation. The idea is to add the translation vector to the camera rays. An example translation can be seen in figure 2.8:

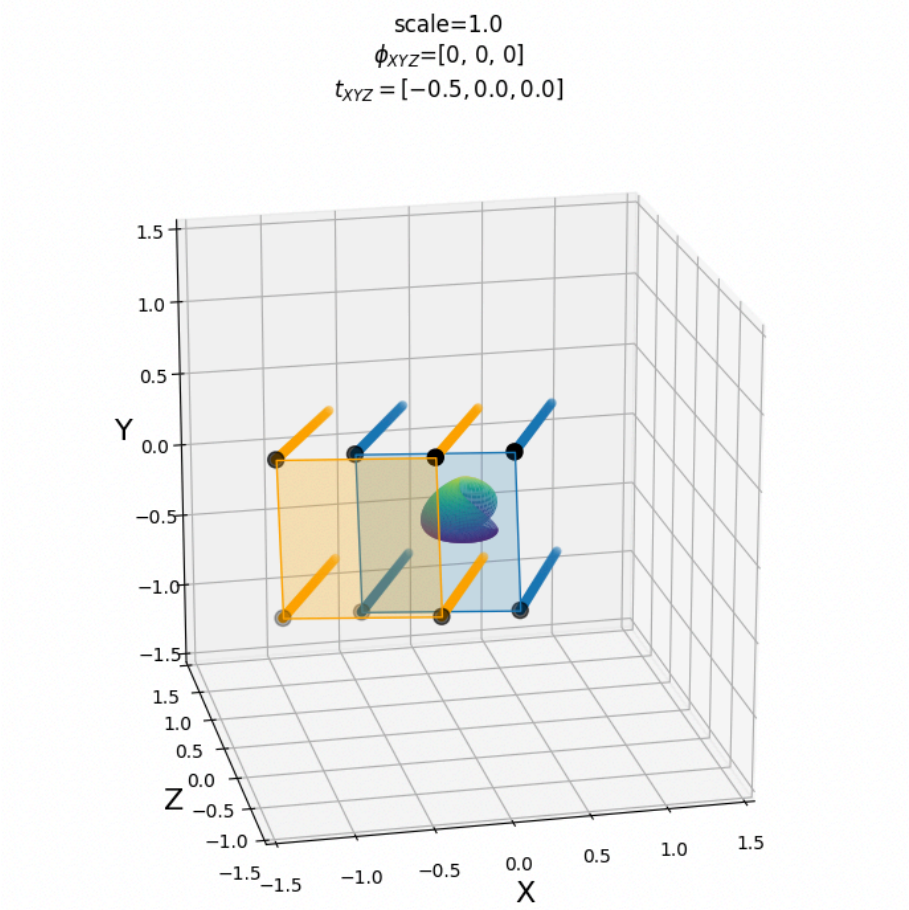

Figure 2.8: Translation along the x-axis. The square polygon faces denotes a camera image, and the lines denote the camera rays. Blue denotes unmodified camera, orange denotes modified camera. This has the effect of moving the object to the right (i.e. further along in the x-direction).

Now, for rotation, the rays are going to be multiplied with a 3×3 rotation matrix, which is defined as follows (where $[\alpha, \beta, \gamma] = [x, y, z]$-rotation):

$$\mathbf{R} = R_z(\gamma)R_y(\beta)R_x(\alpha) = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \tag{2.11}$$

A centre of rotation is also defined using the vector $\mathbf{c} = [c_x, c_y, c_z]$. By default, let $\mathbf{c} = [0, 0, 0]$ (i.e. rotation about the origin). An example rotation can be seen in figure 2.9.

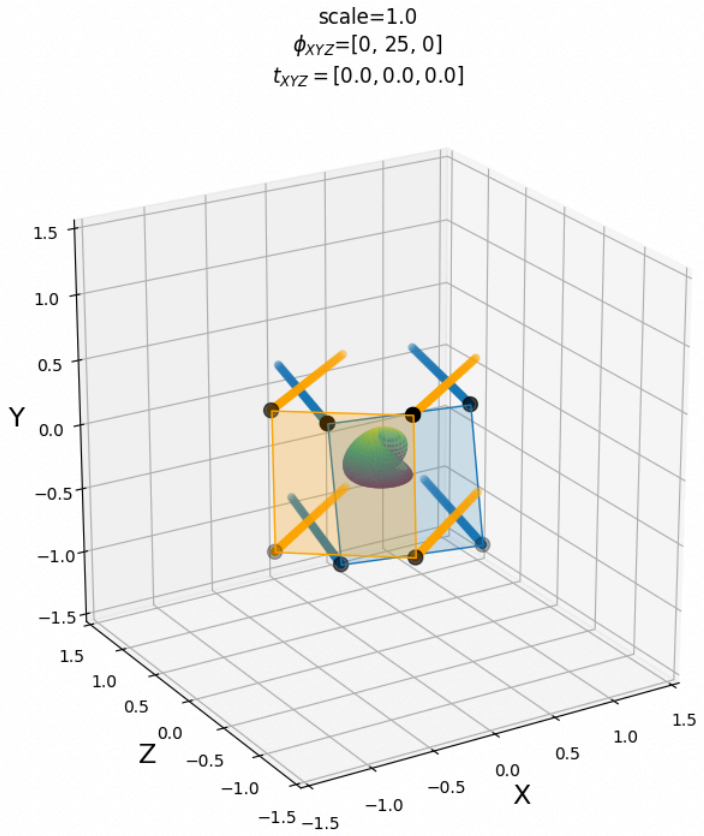

Figure 2.9: Rotation around the z-axis. The square polygon faces denotes a camera image, and the lines denote the camera rays. Blue denotes unmodified camera, yellow denotes modified camera. This has the effect of rotating the object 25 degrees counter-clockwise about y.

Now it's time to define operations for scaling. Let s denote a scalar value which will control scale. Then let $\mathbf{S} = \begin{bmatrix} \frac{1}{s} & 0 & 0 \\ 0 & \frac{1}{s} & 0 \\ 0 & 0 & 1 \end{bmatrix}$ be a scaling matrix which is designed to scale the x and y components of our ray origins (as such, the operation leaves ray directions intact).

Before applying the scaling matrix, the rays must first be aligned with the unit vector $[0, 0, 1]$. To achieve this, we need to determine the matrix **A** which rotates the unit vector $[0, 0, 1]$ such that it is aligned with the ray direction vector. While there likely exists a closed form solution to this, it was easier to just use a Kabsch algorithm (which exists in scipy). Thus, the scaling operation is performed by the chain of matrix products $\mathbf{A}^{-1}\mathbf{SA}$, where $\mathbf{A} = \text{align}([0, 0, 1], \mathbf{d})$. An example of the scaling operation can be seen in figure 2.10.

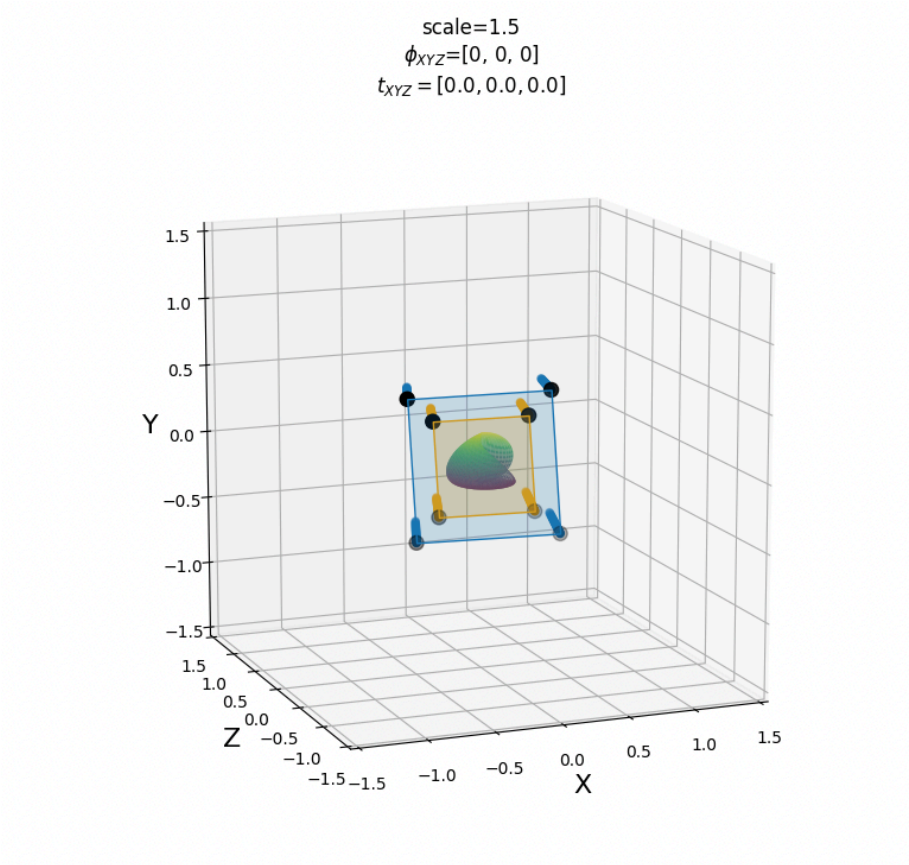

Figure 2.10: Scaling operation. The square polygon faces denotes a camera image, and the lines denote the camera rays. Blue denotes unmodified camera, yellow denotes modified camera. By transforming the ray origins such that they are closer together, the object effectively appears larger.

Recall the ray definition $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$. Now, combining translation, rotation and scaling, the full transformed ray definition can be stated as follows:

$$\hat{\mathbf{r}}(t) = ((\mathbf{o} - \mathbf{c})\mathbf{A}^{-1}\mathbf{S}\mathbf{A})\mathbf{R} + t\,((\mathbf{d} - \mathbf{c})\mathbf{R} + \mathbf{c}) + \mathbf{c} + \mathbf{t} \tag{2.12}$$

An example which performs all transformations at once can be seen in figure 2.11.

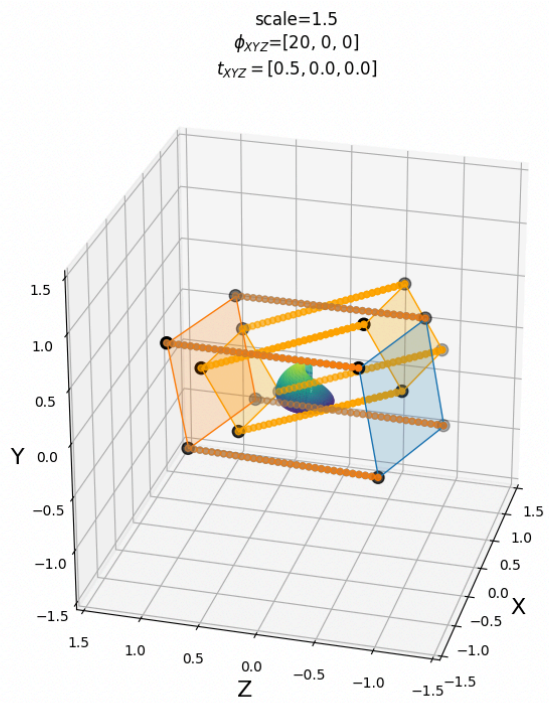

Figure 2.11: Combined transformation. To illustrate the consistency of the transformations for different views, an additional orange camera has been inserted directly opposite of the existing blue camera. The modified cameras are yellow.

There is an additional important detail about re-inserting transformed disentangled objects. The transformation must also be applied to the background scene before performing volumetric subtraction. In order words, given the full scene F_{full} and the background scene $F_{background}$, let G_{full} denote the transformed full scene, and $G_{background}$ the transformed background scene. Now, the volumetric re-insertion is formulated as follows:

$$(G_{full} - G_{background}) + F_{background} \tag{2.13}$$

Recall that $\mathbf{c} = [0, 0, 0]$, which means that the rotation still revolves about the origin. If the transformation should be object-centric, it would require that the centre point of the object is known (or at least estimated). A potential mechanism for this would be to use the marching cubes algorithm (see figure 2.12) to produce a mesh for the foreground object, which can then be used to determine the voxel centroid (i.e. the ”centre of mass” of the mesh).

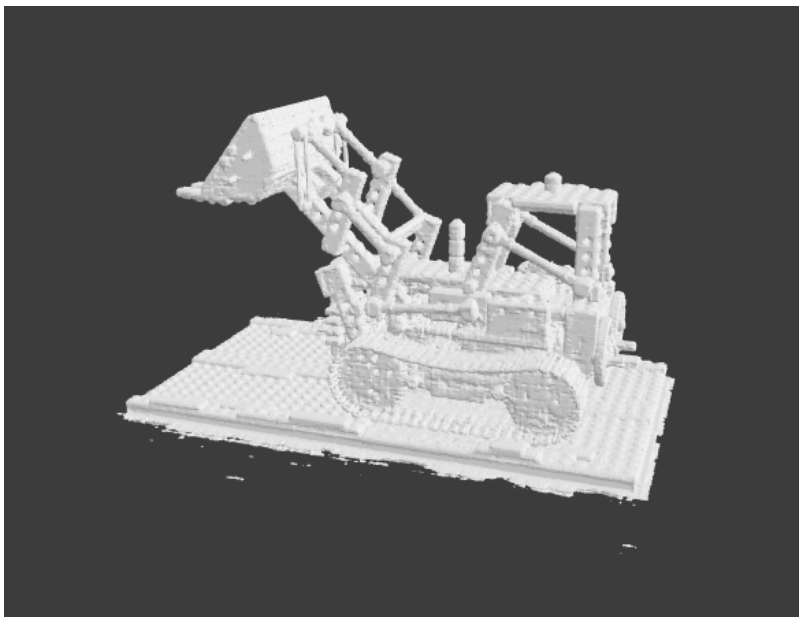

Figure 2.12: Taken from [1]. Marching cubes produces a mesh for the NeRF bulldozer scene. The mesh can be used to calculate a centre point of the foreground object, which can be used to perform object-centric transformations (e.g. rotation about the object itself)

2.2 Semantic guidance

2.2.1 CLIP

Recent advances have been made in the field of vision-language modeling - a very notable one being OpenAI's CLIP[4].

At its core, it contains an image encoder and a text encoder. The image encoder is based on either a ResNet[11] or a Vision Transformer[12] architecture. The text encoder is based on a Transformer[10] architecture. The encoders are configured to produce vectors of the same dimensionality.

A fundamental concept in CLIP is the contrastive pre-training process. Given a dataset of image-text pairs, the encoders are jointly optimized such that they produce vectors that align for the matching image-text pairs, and don't align for the non-matching image-text pairs. In other words, it seeks to maximize the cosine similarity of the embeddings for matching pairs.

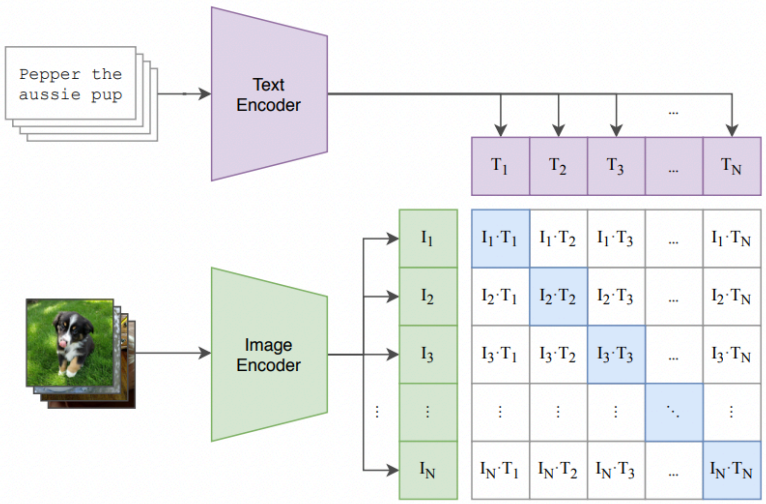

Figure 2.13: Taken from [4]. CLIP contrastive pre-training

The contrastive pre-training is performed in batches of size N. As such, there are N^2 pairs to consider for every batch - of which only N pairs are matches. This is illustrated in figure 2.13, with the diagonal of the matrix indicating the matching pairs for which the similarity should be maximized.

A benefit of pre-training models in this contrastive fashion is the fact that no dataset labels are required. This makes it easier to gather huge datasets by scraping the internet, where a lot of text-image pairs can be found naturally (e.g. Instagram, reddit, etc.). However, despite being trained on a massive dataset, there might still be ”holes” in CLIP’s representation space. For instance, it might not appropriately embed an image due to lighting conditions, image noise, and other visual artifacts. As such, it might be important to heavily augment or somehow regularize the signals obtained from CLIP.

After pre-training, CLIP is capable of embedding instances of different modalities (i.e. text and images) into the same space. This has led to a lot of additional research involving its capabilities as a zero-shot image classifier (zero-shot meaning ”not trained for the specific task”).

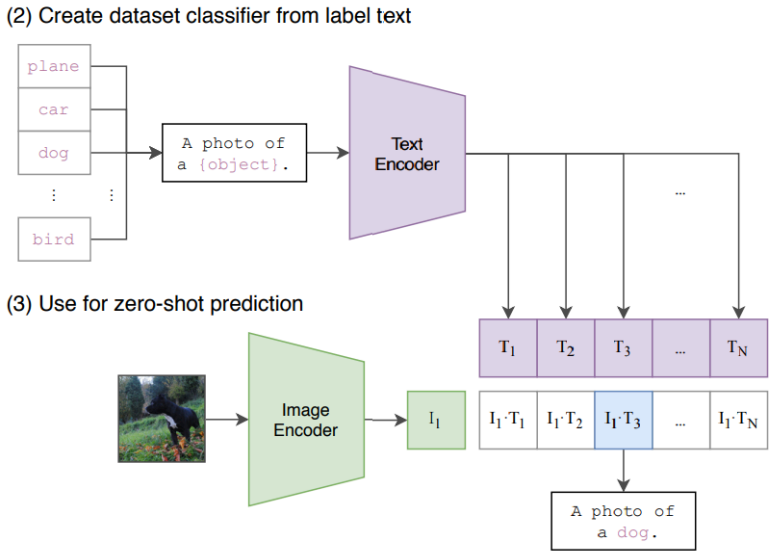

Figure 2.14: Taken from [4]. CLIP zero-shot prediction

Figure 2.14 shows how a zero-shot classifier can be made. The main idea is to produce a set of text prompts based on the classes that are present in the dataset. Then, all the text prompts can be encoded as CLIP vectors. Now, given a new input image, the classifier simply encodes the image to obtain a CLIP vector, and calculates the cosine similarities in order to determine which class yields the highest similarity, which will be the predicted class.

2.2.2 Shattered gradients

The shattered gradients problem occurs when working with deep neural network architectures (such as the ones used in CLIP). The work done in [13] shows that the correlation between gradients in standard feedforward networks decays exponentially with network depth, which ultimately results in gradients that resemble white noise. On the other hand, the gradients in networks that implement skip-connections decay sublinearly, which makes them more resilient to gradient shattering.

In [13], an example neural network can be constructed. It maps from $\mathbb{R} \to \mathbb{R}$ (i.e. scalars to scalar), and contains $N = 200$ rectifier units per hidden layer. The gradient with respect to the input x_i is computed for 256 values of $x_i \in [-2, 2]$. This has been done for multiple network architectures (including ResNet, which has skip-connections). The results can be seen in figure 2.15.

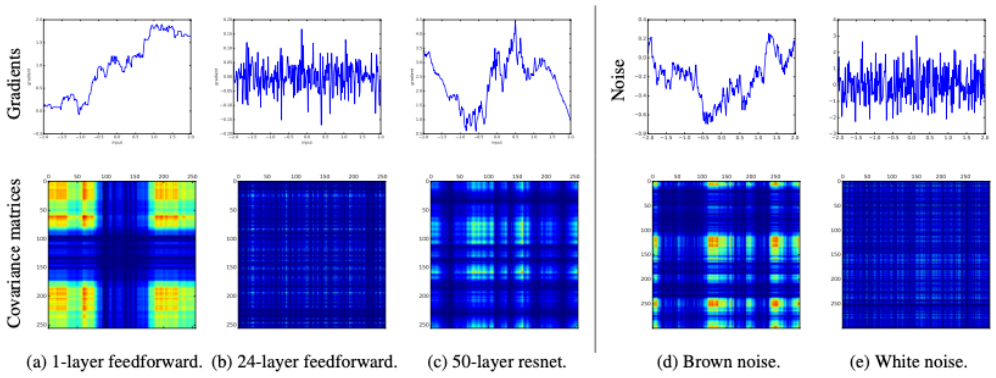

Figure 2.15: Taken from [13]. Comparison between noise and gradients of rectifier nets with 200 neurons per hidden layer.

Figure 2.15 shows that the gradients (and covariance matrices) resemble white noise for the deep feedforward network, and brown noise for the ResNet. This is explored even further in [13], where the phenomenon is illustrated by plotting the autocorrelation function (ACF) for noise compared to the gradients from different network architectures at various depths ranging between $[2, 50]$.

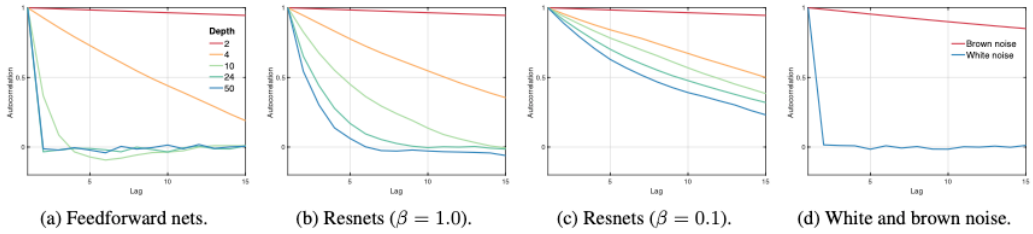

Figure 2.16: Taken from [13]. Autocorrelation functions (ACF) for the gradients from various network architectures compared to white and brown noise.

Figure 2.16 illustrates the impact of network depth on gradient noise. For standard feedforward nets, the gradient ACF already becomes similar to the ACF of white noise at a depth of 10. For ResNets, the gradient ACF seems to be more similar to the ACF of brown noise (depending on the rescaling parameter β).

2.3 Full pipeline

Combining the methods from the previous sections yields the full pipeline for semantically guided volumetric object manipulation. Given a differentiable renderer (in our case NeRF) which supports a set of transformation parameters **s**, the goal is to determine the most optimal parameter values with respect to a given text prompt. The similarity will be driven by CLIP. A diagram of the pipeline can be seen in figure 2.17.

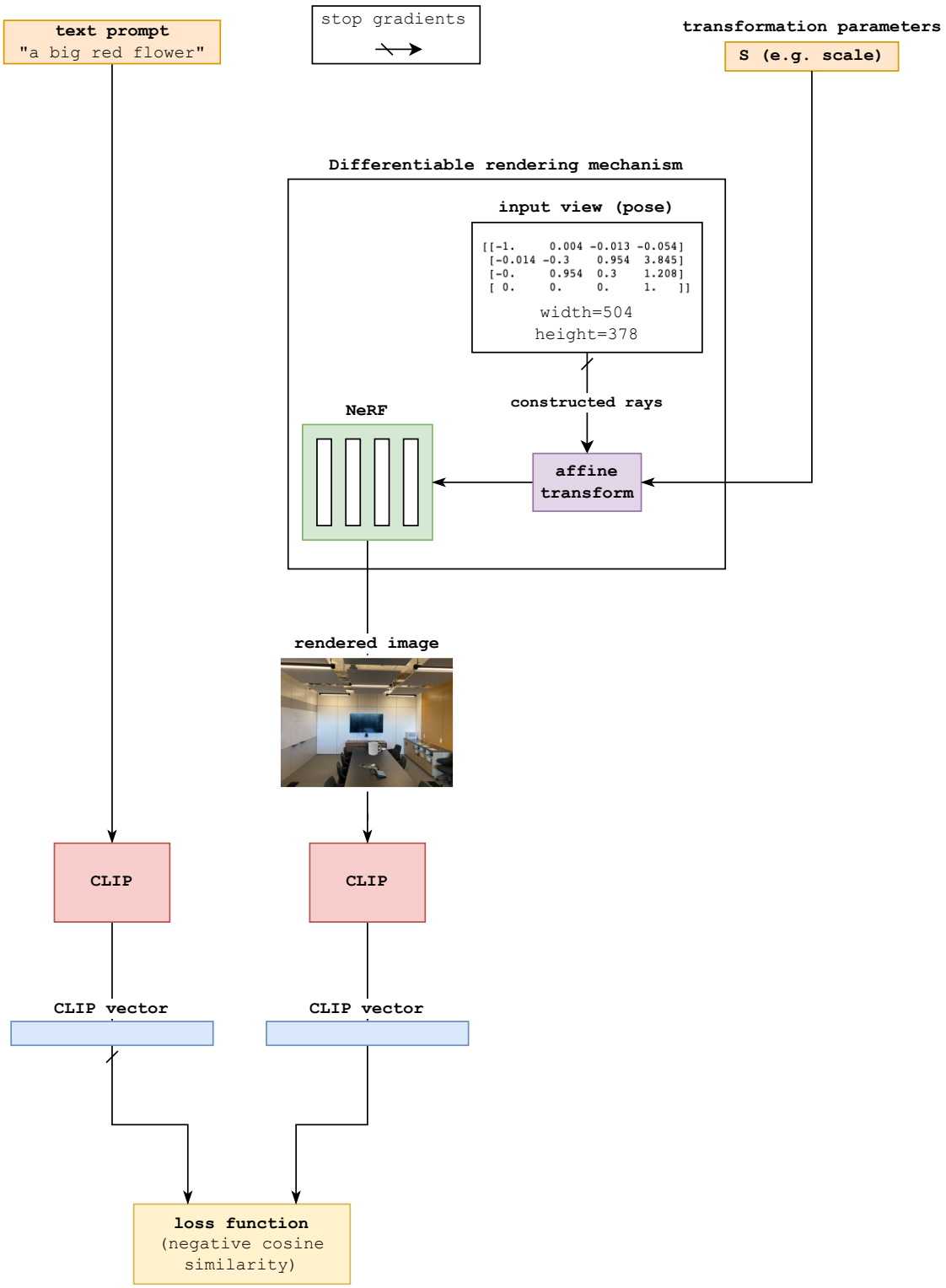

Figure 2.17: Diagram of the full pipeline for semantically guided volumetric object manipulation.

3 Results

Instead of tying everything together at once, the project has been divided into five incremental phases. Being able to establish a working proof-of-concept for each step before adding on the extra complexity of the following step was a good strategy for determining which parts were working, and which parts needed to be worked on more.

The goals for each phase are described as follows:

1. 2D optimization experiments

   - Build a simple, differentiable 2D rendering mechanism.
   - It should support a set of basic parameterized transformations such as scaling, translation and rotation.
   - Given a target image, it should be able to find the optimal transformation parameters to recreate the image using MSE loss.

   Once a simple 2D baseline has been established, it can be extended to incorporate an actual CLIP-based loss function.

2. 2D semantic experiments

   - Replace the MSE loss with a CLIP-based loss.
   - It should be able to adequately optimize these transformation parameters wrt. a given text prompt.

   Next, the 2D operations can be extended to 3D by using a different rendering mechanism. This brings the problem closer to the full NeRF-based pipeline, and introduces additional transformation parameters.

3. 3D semantic experiments

   - Swap out the 2D rendering mechanism with a 3D rendering mechanism.
   - Experiment with novel view augmentation (i.e. pass multiple rendered images from the scene to CLIP).

   Before proceeding, it should be proven that the transformations for disentangled NeRFs work adequately. This is done using fixed transformation parameters (i.e. not semantically guided).

4. Disentangled NeRFs

   - Successfully perform NeRF disentanglement
   - Perform transformations on disentangled NeRFs

3.1 2D optimization experiments

The purpose of this phase is to explore the feasibility of optimizing a set of transformation parameters for a rendering mechanism with respect to an MSE loss function. Given a parameterized rendering mechanism $R(s)$, the loss function is defined as follows:

$$\mathcal{L}(s) = ||R(s) - R(s_{target})||_2^2 \tag{3.1}$$

The loss in eq. 3.1 can also be described as the MSE loss between the image rendered with the parameters s and the target image, which was rendered with the parameters s_{target}. Thus, the objective is to find $s^* = s_{target}$, as this will have the lowest loss value.

The PyTorch library kornia[1] was very useful for implementing the differentiable rendering mechanism - it contains a wide range of differentiable image operations, which made it quick and easy to build the renderers.

Initially, a renderer with a single parameter s for controlling the rotation of a coffee cup was implemented. Instead of just clamping the parameter s to a certain range (e.g. $[-180, 180]$), a more gradient-preserving formulation was used for calculating the effective degrees of rotation: $rot(s) = \tanh(s) * 180$. Thus, the operation is able to rotate the cup by degrees in the range $[-180, 180]$.

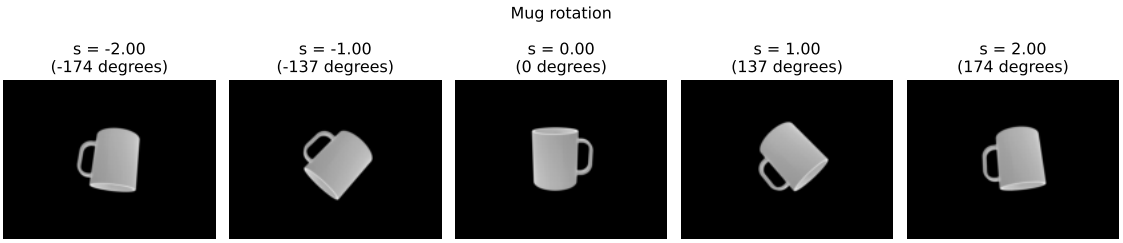

Figure 3.1: A sample of various s-values and the corresponding rendered images.

Setting $s_{target} = 0.0$ and computing the loss from eq. 3.1 for 100 s-values sampled in the range $[-2, 2]$ produced the results seen in figure 3.2.

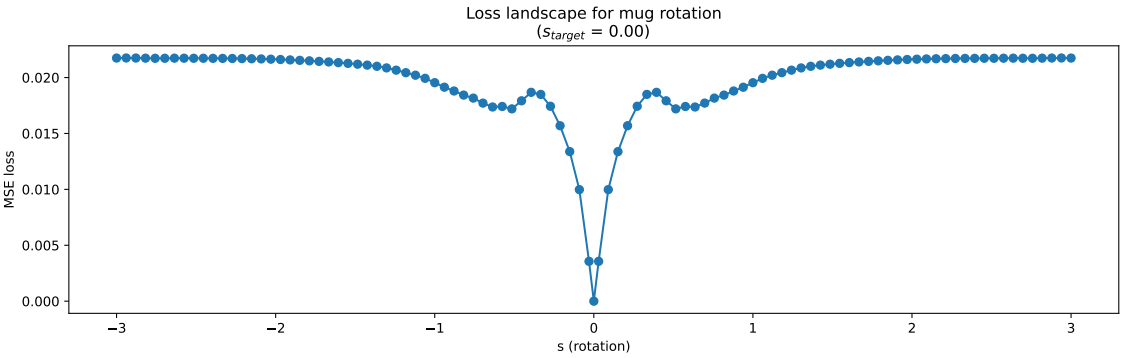

Figure 3.2: MSE loss landscape for mug rotation for 100 sampled s-values in the range $[-3, 3]$

Figure 3.2 shows that the loss landscape is nonconvex. There is a clear global minimum around $s = 0.0$, which is expected. It also shows two local minima on either side of the global minimum. Another observation to make is the fact that the boundary losses will become identical as $s \to \pm\infty$, since rotating by ± 180 degrees will produce identical images.

The approach of evaluating many s-values within a given range (in our case $[-3, 3]$) and simply returning the lowest value observed can be considered a brute force solution to the parameter optimization problem. In the simplified example, the strategy remains feasible, but it is important to note that the brute force search runs in exponential time. Imagine that the parameter count increased to $n = 4$ (rotation, x-translation, y-translation scale), and the resolution (i.e. how many values to sample from each parameter range) remained

[1] https://github.com/kornia/kornia

$r = 100$. The amount of evaluations required then becomes $r^n = 100^4 = 10^8$, which is a rather hefty amount. Time complexity becomes even more of an issue when the rendering mechanism is swapped out for more advanced mechanisms (e.g. NeRF), which are more expensive to run.

Thus, a more elegant and efficient solution to the optimization problem is needed. A viable alternative could be gradient-based optimization methods. Recall that the rendering mechanisms are immediately differentiable thanks to kornia. This means that the gradient $\frac{\partial L}{\partial s}$ is available and can be used for gradient descent optimization.

A training loop for gradient descent was implemented using Adam with a learning rate of 0.05. The loop was run for 50 iterations for two different initial values for s: [-1.0, 2.0].

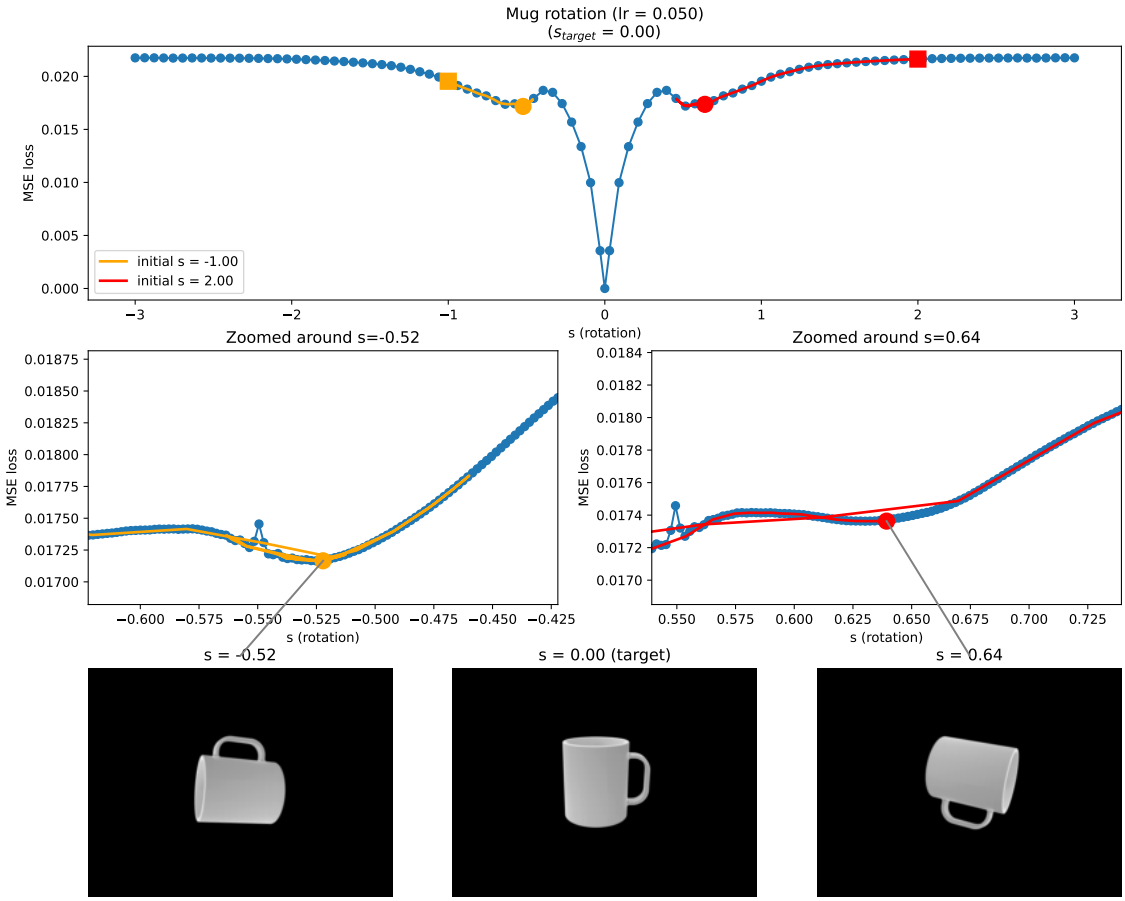

Figure 3.3: The first row shows the previous loss landscape overlaid with the gradient descent paths for two different starting points (the square marker is the starting point, the circle is the final point). The middle row shows two ”zoomed in” views of the plot above. The bottom row shows the final solutions together with the target image.

From figure 3.3 it appears that the gradient descent procedure fails to find the global minimum at $s = s_{target} = 0.0$. For both starting points, the algorithm ends up in a local minimum. Also - it’s not by chance that these two minima exist. They occur when the rotated cup is at a sort of ”equilibrium point” at which the loss is improved equally when rotating the cup in either direction - i.e. a sort of ambiguity is at play here. This is the case when the rotated cup is perpendicular to the target cup.

Similar rendering mechanisms were made, which were respectively able to scale and

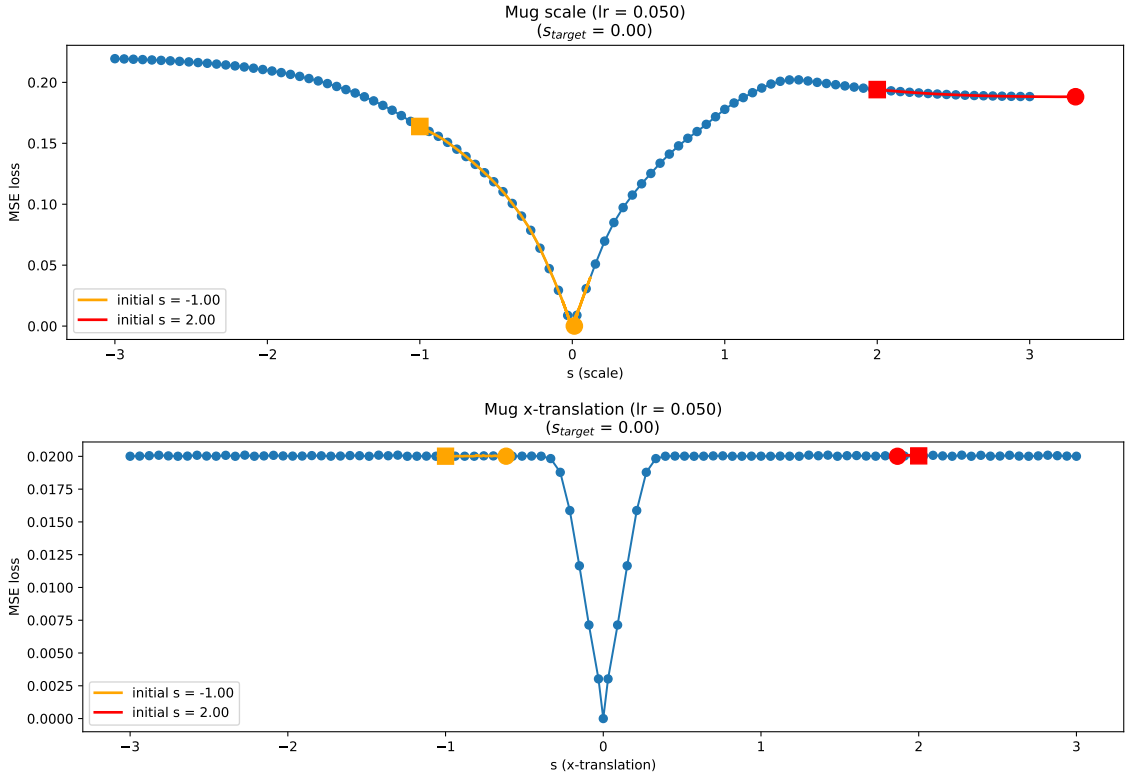

Figure 3.4: Top row shows the loss landscape for the scaling renderer. Bottom row shows the loss landscape for the x-translating renderer.

x-translate the coffee cup instead. This produced the results seen in figure 3.4.

Figure 3.4 shows that the issue with local minima also affected the other types of transformations. If the initial guess was sufficiently close to s_{target}, then gradient descent was able to find its way to the optimal solution. Otherwise, it could still get stuck in a local minimum

Also, it is quite interesting to see the ”plateaus” for the x-translation loss landscape in figure 3.4. The intuition behind this can be explained by a small example: Imagine the target cup being far over on the right side of the image. Now, for a cup on the left side of the image, which has no overlap at all with the target cup on the right, the loss will remain the same when stepping either left or right. This will be the case until the cups start overlapping, at which point the ”valley” in the loss landscape starts to appear. This kind of fundamental ambiguity is what makes the optimization problem difficult to solve with gradients.

Additional experiments were performed in hopes of improving the gradient descent approach. Switching the MSE loss with an L1 loss made no real difference. Switching the optimization algorithm from Adam to RMSprop, Adagrad, SGD did not help much either. The most significant knobs to tweak were the starting points and the learning rate. If adjusted correctly, the learning rate can help gradient descent escape the local minima. However, if the learning rate was too big, the algorithm ended up ”overshooting” and escaping the global minimum. This can be seen in figure 3.5.

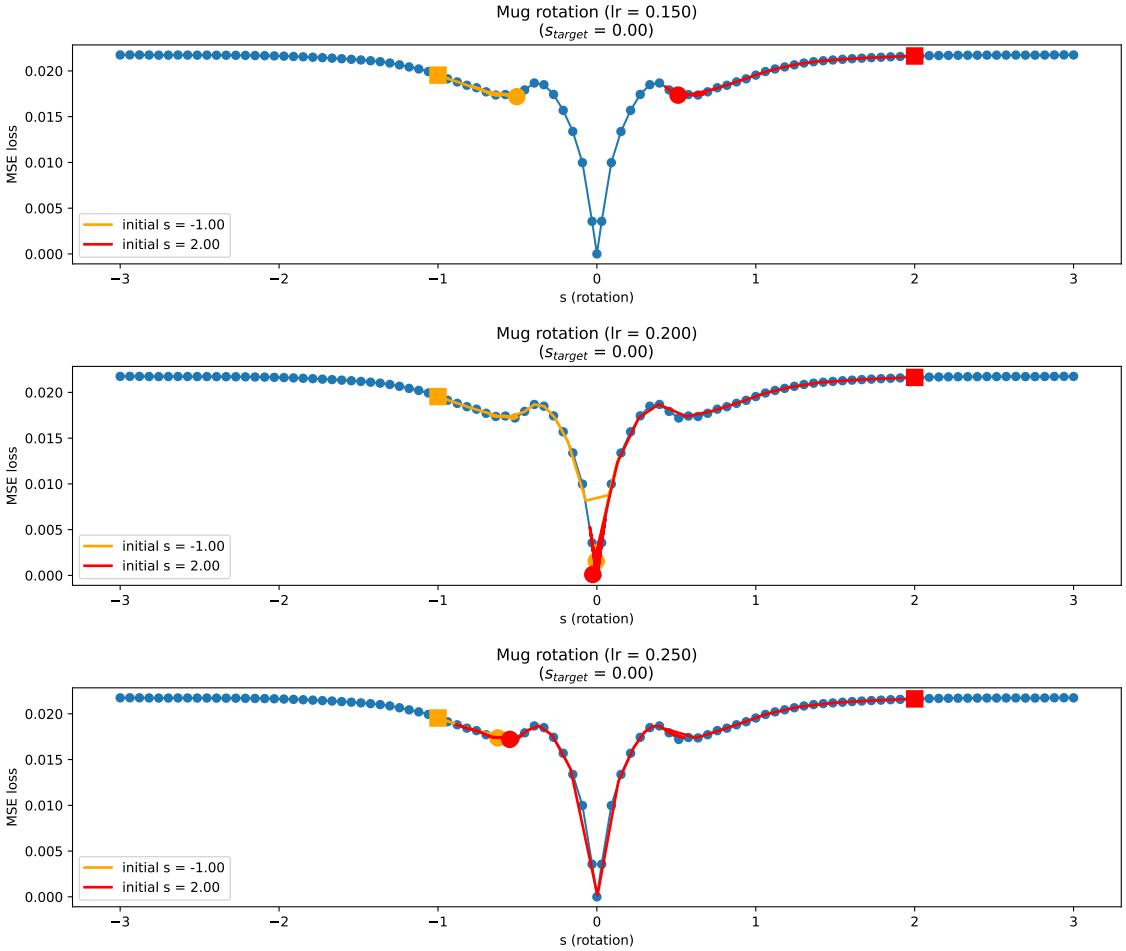

Figure 3.5: Gradient descent solutions for three different learning rates. The optimizer was Adam, and the iteration count was 50. The squares indicate starting points for gradient descent. The circles denote ending points.

Next, it's time to see how gradient-free optimization techniques perform on the task. For this, several algorithms from scipy[2] were experimented with. Specifically, the following collection of methods: [basinhopping, differential_evolution, dual_annealing]. This produced the results seen in figure 3.6.

[2]https://docs.scipy.org/doc/scipy/reference/optimize.html

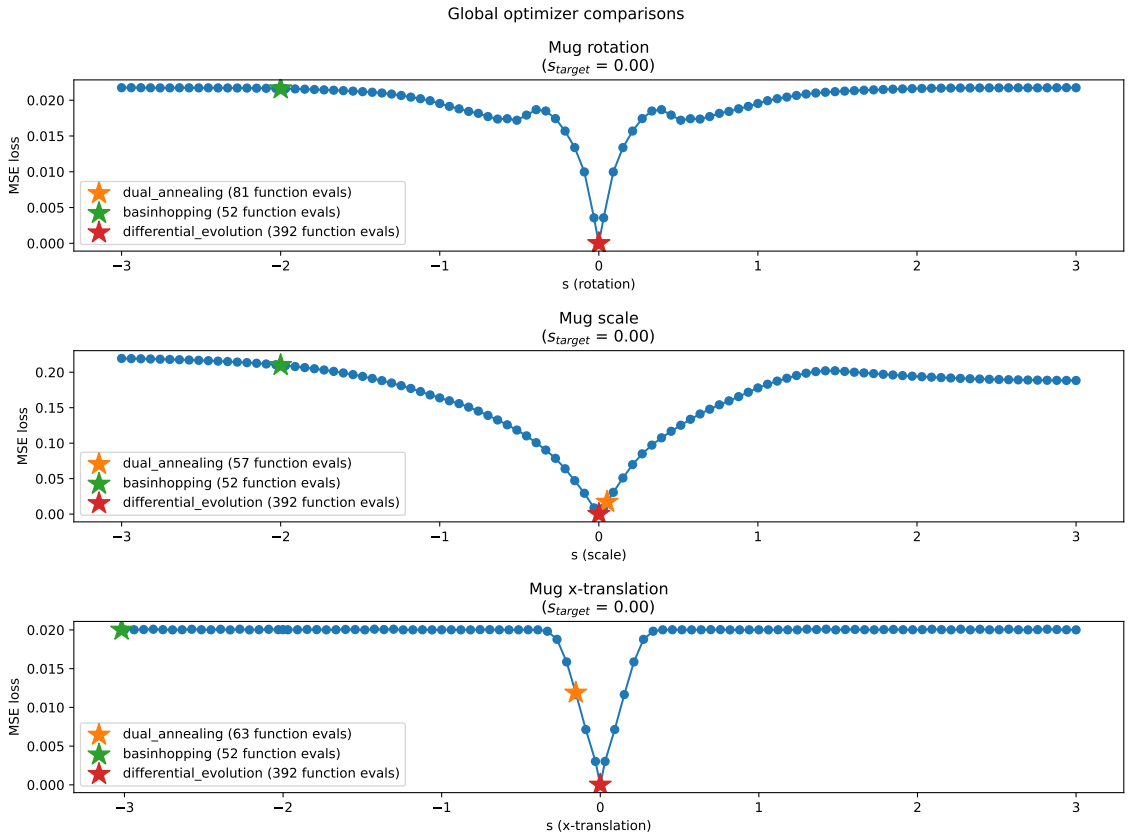

Figure 3.6: Comparison of global optimization methods. Each row corresponds to a loss landscape for a certain transformation (respectively rotation, scale, and x-translation). The colored stars correspond to the final solutions obtained from the methods. The legend also denotes the amount of function evaluations performed during optimization.

Figure 3.6 shows a comparison of the different global optimization methods. When comparing the different optimization methods, it seemed like dual_annealing had a good balance between speed (i.e. function evaluation count) and quality.

To summarize the findings of this subchapter:

- Even for the relatively simple MSE loss, the landscape is still non-convex due to the fundamental ambiguities described. Thus, there will be local minima and plateaus that can be challenging to optimize around.
- While it's not impossible for gradient descent to find the optimal solution, it usually requires a considerable amount of hyperparameter tweaking (e.g. optimizer, learning rate, initial guesses, etc.) to do so reliably.
- Gradient-free global optimization methods can be used as a backup when gradient descent fails. These are usually more expensive than gradient descent in terms of render evaluations, but they aren't quite susceptible to the same issues as gradient descent. Amongst the methods that were experimented with, dual_annealing seemed to perform best.

3.2 2D semantic experiments

To establish the viability of semantically guided volumetric manipulations, it needs to be proven that CLIP is able to understand the visual effects of the transformations (e.g. scale,

translation, rotation). Fortunately, OpenAI has published pre-trained versions of CLIP that are readily available[3]. Initially, the chosen CLIP model version is ”ViT-L/14”. The model is first loaded to the CPU and subsequently moved to the GPU in order to run using 32-bit floating point values (rather than the default optimized 16-bit floating point values).

The image rendering mechanisms from the previous subchapter were reused. However, the loss function was modified. Instead of using the MSE loss between the candidate image and a target image, the loss was now formulated as the negative CLIP similarity between the candidate image and a target text prompt. Formally, it can be formulated as follows:

$$\mathcal{L}(s) = -E(R(s)) \cdot E(text\ prompt) \tag{3.2}$$

In addition to the rendering mechanism $R(s)$ (which was also present in the previous subchapter), there is now $E(...)$, which denotes the CLIP encoder (used synonymously for image and text inputs), and $text\ prompt$, which denotes the given text prompt (i.e. a constant).

First, it might be insightful to see how a few example text prompts relate to the rendered images. For this example, the scaling renderer was used. The following two text prompts were used: [”a tiny coffee mug with no background”, ”a medium coffee mug with no background”, ”a huge coffee mug with no background”]. 100 s-values were sampled in the range $[-2, 2]$. This produced the results seen in figure 3.7.

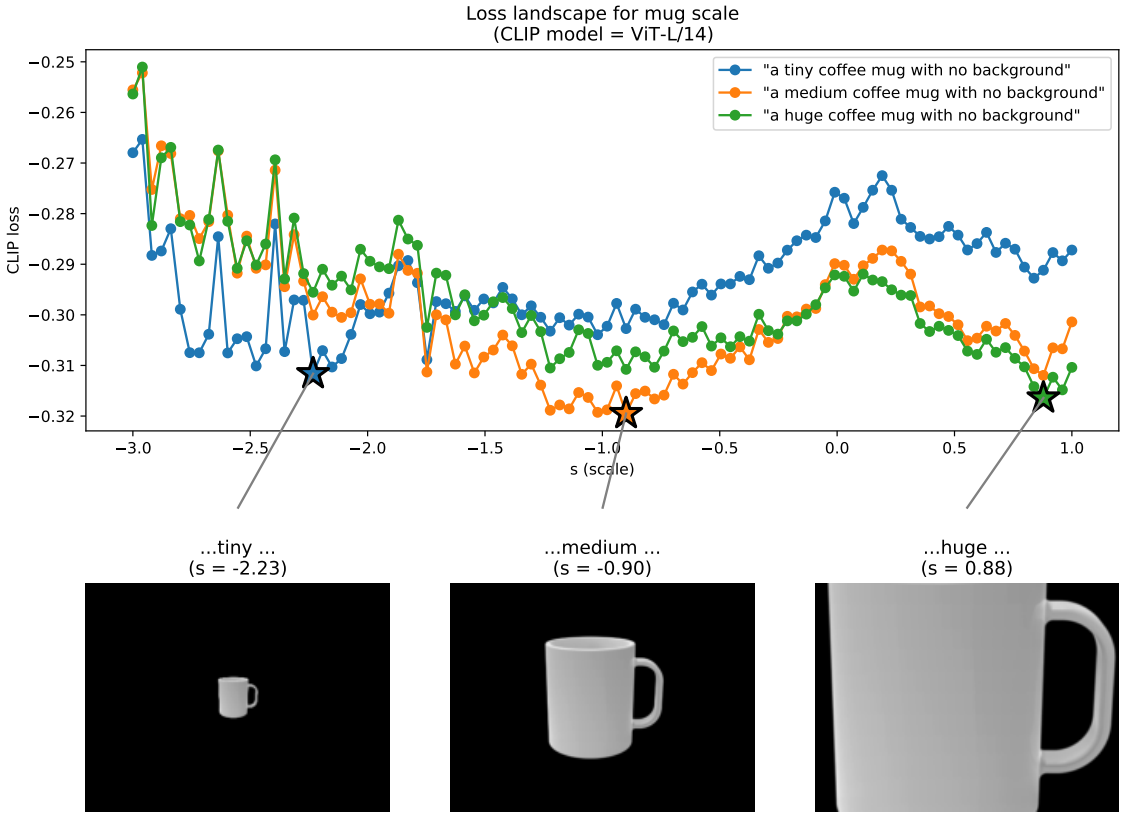

Figure 3.7: Top row shows the CLIP loss landscape for the different text prompts overlaid with star markers that denote the corresponding minimum. Bottom row shows the resulting images rendered at the various minima.

[3] https://github.com/openai/CLIP

Figure 3.7 demonstrates that CLIP has some understanding of scale-related adjectives in the context of scaling a mug with no background. However, the loss landscapes appear to be much noisier than the MSE loss landscapes seen in the previous subchapter, which could make it difficult for gradient descent to work effectively.

The mug x-translation renderer was extended to also include a background image of an office (which is actually from a popular NeRF dataset). The idea is to gauge how well CLIP understands an image of a more complex image with several objects present. The text prompts were also changed to [”a coffee mug on the floor of an office”, ”a coffee mug on a table in an office”]. This produced the results seen in figure 3.8.

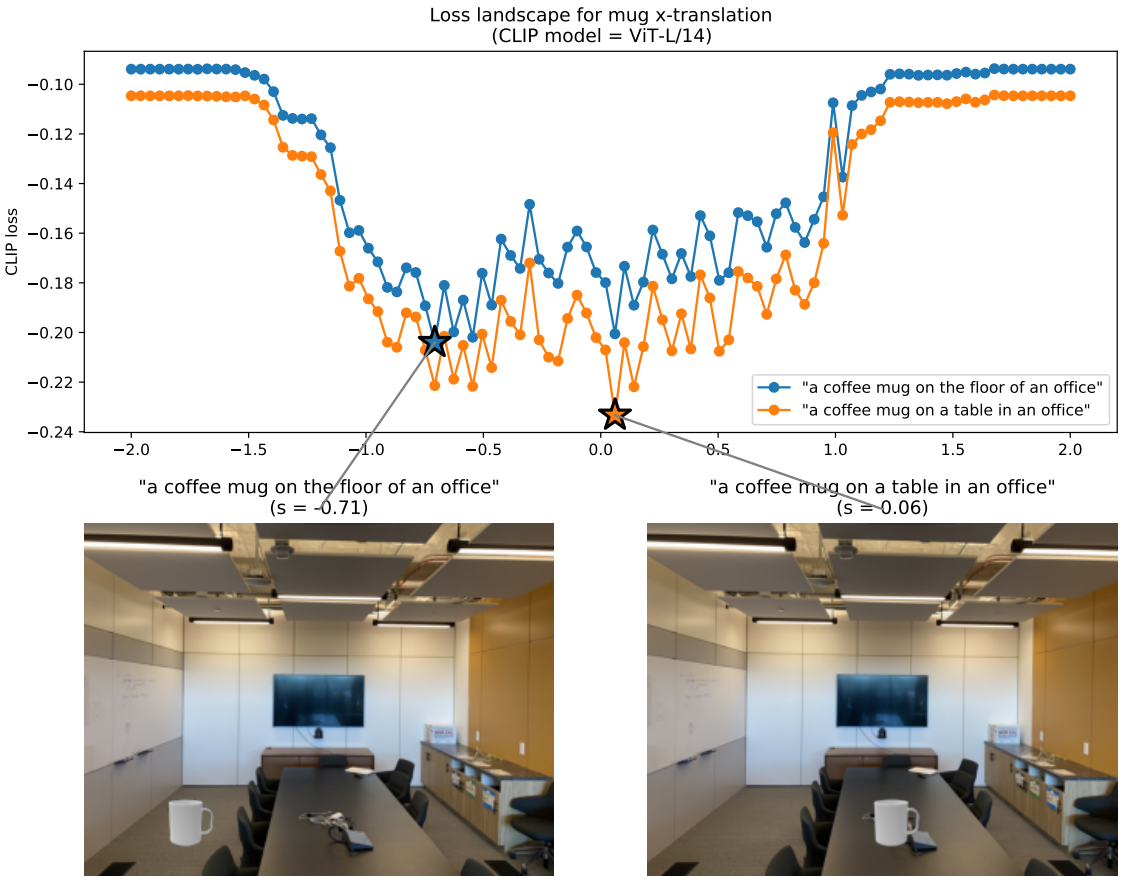

Figure 3.8: Top row shows the CLIP loss landscape for the different text prompts overlaid with star markers that denote the corresponding minimum. Bottom row shows the resulting images for the various minima.

Figure 3.8 shows loss landscapes that are observably even noisier than the ones seen in figure 3.7. Despite the noise, the minimum for the prompt ”... on the floor of an office” correctly places the mug on the floor to the left, and the minimum for the prompt ”... on a table in an office” correctly places the mug center on the table.

Recall that the loss landscapes have been plotted at a granularity of $N = 100$ points in the range $[-2, 2]$ for the parameter s. Perhaps ”zooming in” and plotting the landscape at a different scale (i.e. finer resolution) could provide additional insight on how the landscape behaves. The

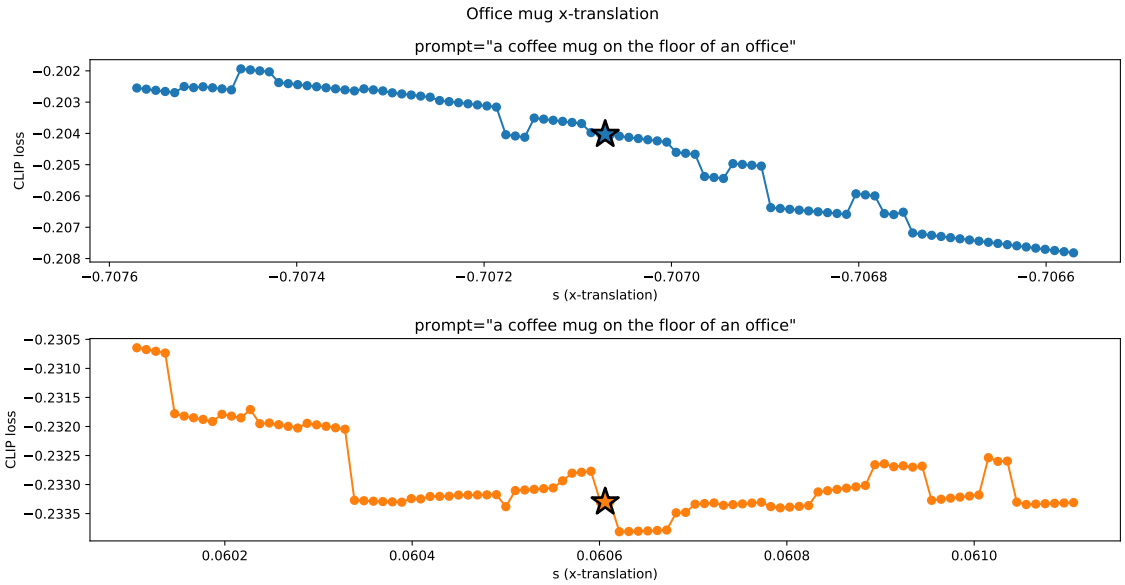

Figure 3.9: Zoomed in loss landscape for the example with mug x-translation in an office environment.

Figure 3.9 shows that the previously marked minima are in fact not the true global minima. Sampling the loss landscape at a higher resolution shows that there exists even better values for s. Another interesting detail is the fact that a minuscule change in s, which a human likely wouldn't notice, leads to a significant change in CLIP loss. This shows that the loss landscape is highly sensitive to s.

Ignoring the noisy loss landscape for now, it's time to see how well gradient descent works for optimization. A training loop was set up together with a cosine annealing learning rate scheduler. Running the loop for 300 iterations with an initial learning rate of 0.1 produced the results seen in figure 3.10.

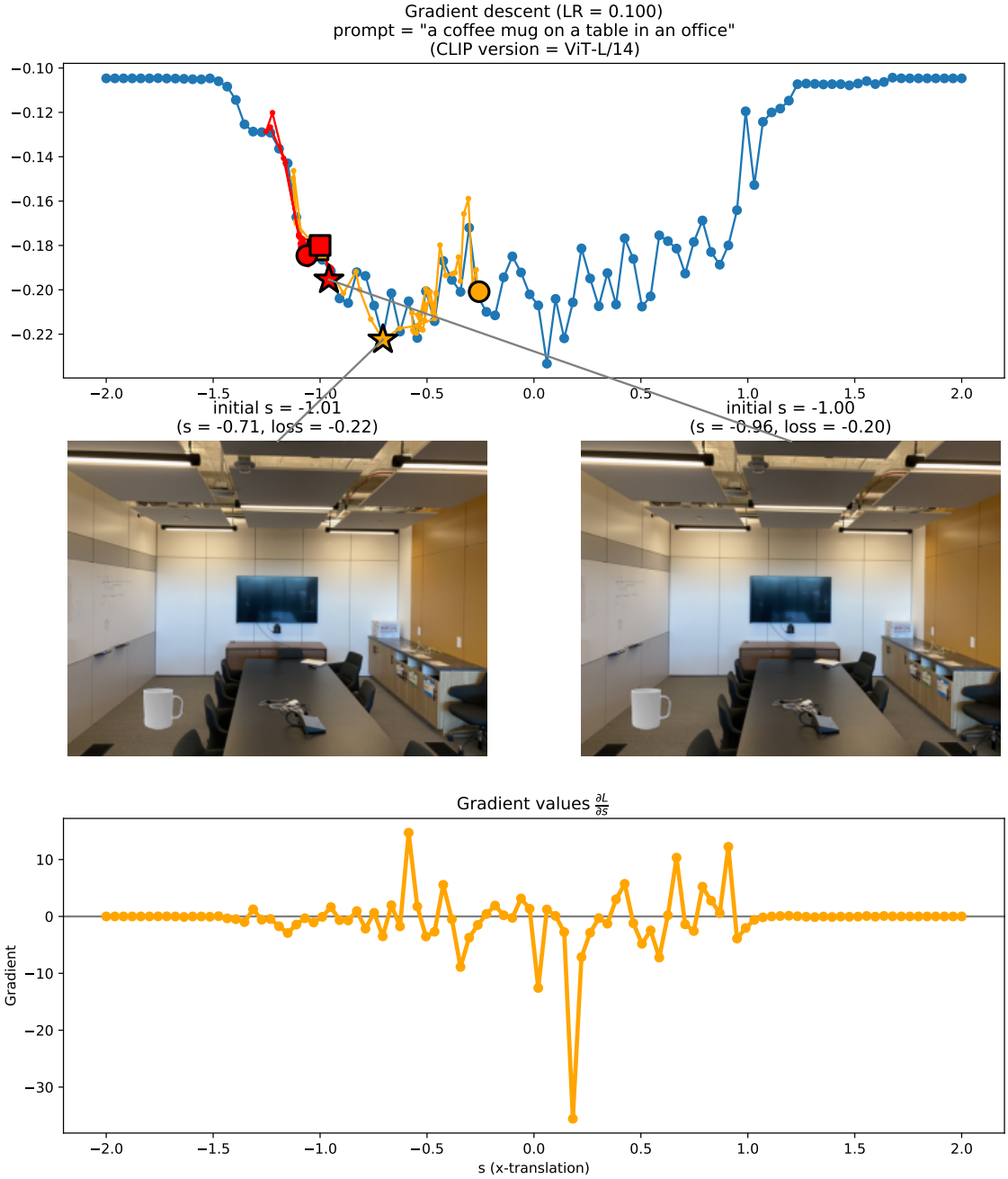

Figure 3.10: The top row shows the CLIP loss landscape overlaid with the gradient descent path (the square is the starting point, the circle is the final point, the star is the minimum value encountered during training). The middle row shows the image corresponding to the minimum found during gradient descent. The bottom row shows the gradient values for the sample s-values.

Figure 3.10 shows that gradient descent fails to locate the minimum. It also shows how two very similar initial conditions (i.e. initial $s = -1.01, -1$) can end up yielding very different paths. The bottom row shows that the gradients are very noisy in the range $[-1, 1]$.

So why does the loss function (and gradients, by extension) become so noisy? Recall what was presented in section 2.2.2 - essentially, it has a lot to do with the depth of the network used. Also recall that the CLIP model version ”ViT-L/14” has been used for the

experiments up until this point. Although skip connections are present both in the vision transformer and ResNet architectures, it might still be beneficial to switch the CLIP model to the shallowest ResNet CLIP architecture. Thus, the model was switched to a RN50 version. The results of this modification are shown in figure 3.11.

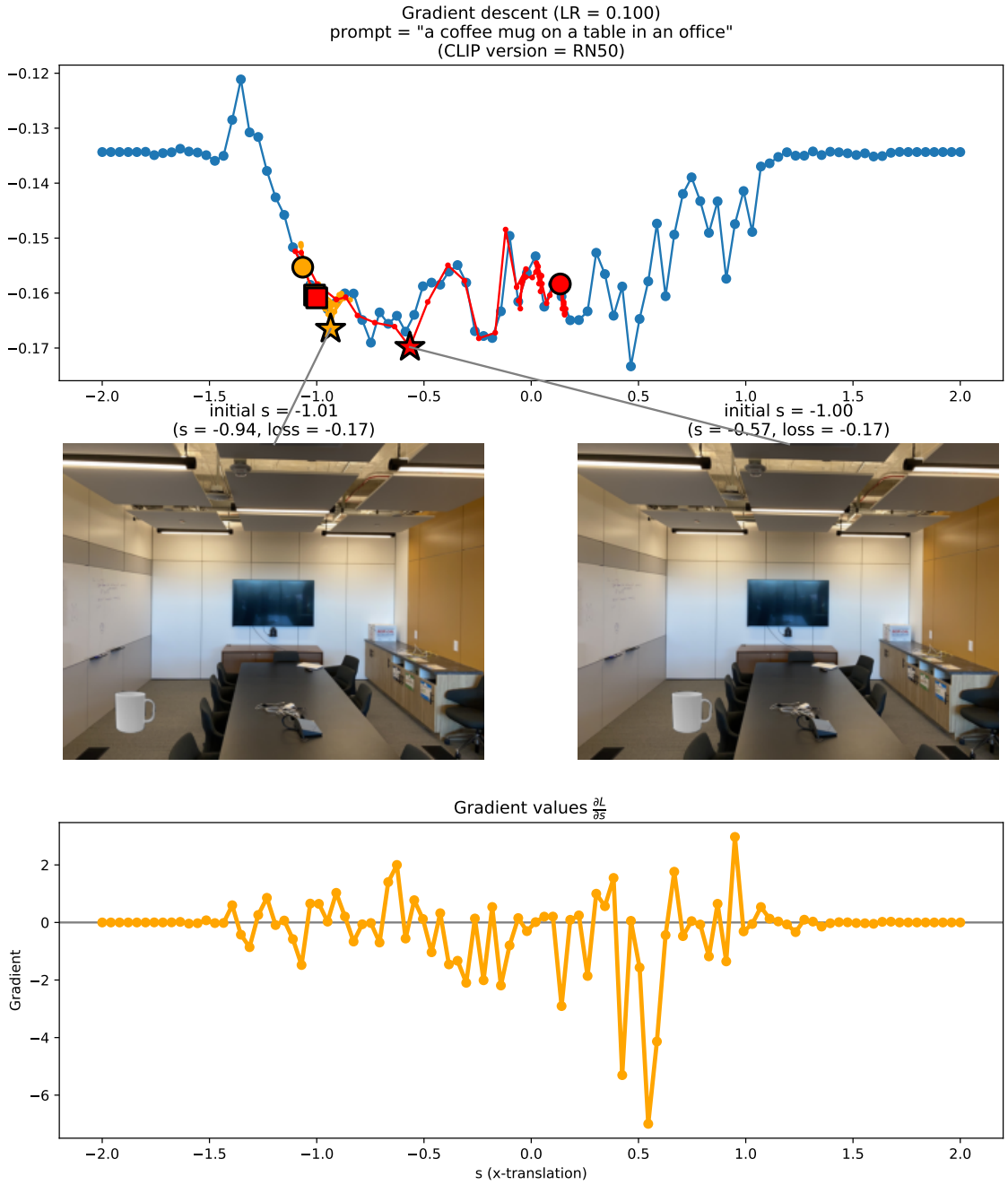

Figure 3.11: The top row shows the CLIP loss landscape overlaid with the gradient descent path (the square is the starting point, the circle is the final point, the star is the minimum value encountered during training). The middle row shows the image corresponding to the minimum found during gradient descent. The bottom row shows the gradient values for the sample s-values.

Figure 3.11 shows that the loss landscape is still rather noisy. The magnitude of the gra-

dient extremes seems to have decreased. However, gradient descent still didn't manage to find the minimum. Alas, choosing a shallower CLIP model did not seem to improve things much.

There exists other techniques which could potentially improve the performance of gradient descent:

- Gradient clipping/scaling
- Provide an ensemble of different text prompts
- Random image augmentations
- Using an ensemble of CLIP models

An experiment was performed where all modifications except the last one were implemented. The gradients were clipped at ± 3. An additional two text prompts were provided (replacing ”coffee mug” with ”cup of coffee” and ”white mug”). 8 random image augmentations were applied at every render. The image augmentations consisted of different operations from *kornia*: [RandomElasticTransform, ColorJitter, RandomHorizontalFlip, RandomGaussianBlur, RandomGrayscale]. Again, the same learning rate of 0.1 and iteration count of 300 was used. The results can be seen in figure 3.12.

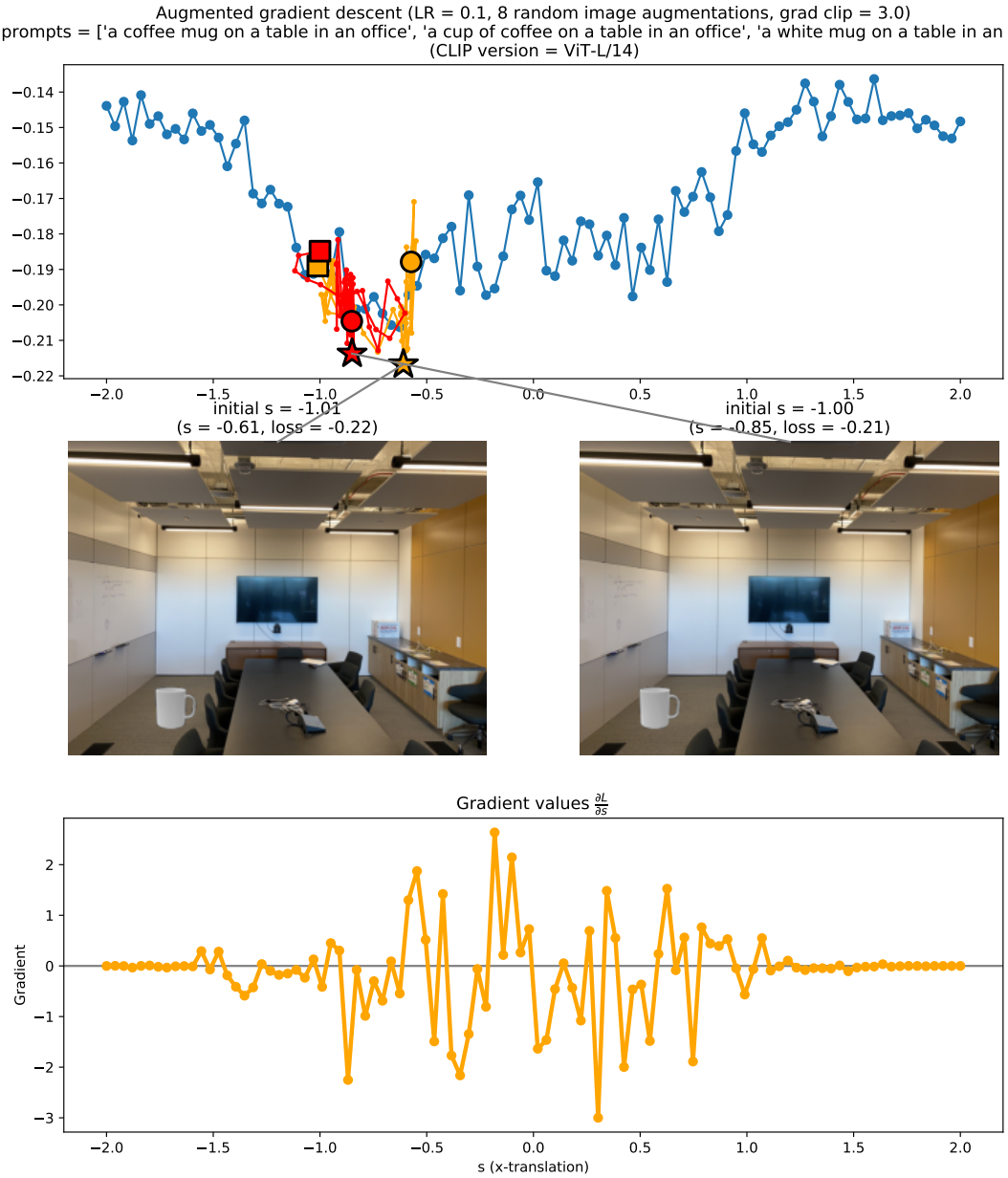

Figure 3.12: Augmented gradient descent. The top row shows the CLIP loss landscape overlaid with the gradient descent path (the square is the starting point, the circle is the final point, the star is the minimum value encountered during training). The middle row shows the image corresponding to the minimum found during gradient descent. The bottom row shows the gradient values for the sample s-values.

Figure 3.12 shows that the loss landscape is not dramatically improved. In fact, the minimum seems to have moved away from around $s = 0.0$ - it now appears to be between -1.0 and -0.5. However, the gradient descent final solutions do seem to end up slightly closer this time.

Taking a step back, one might notice that when the example was changed from the backgroundless mug scaling renderer to the office mug translation renderer, the loss landscape became even noisier. Perhaps the background is contributing a lot to the noise? An effective augmentation might be to blur the background using something like a Gaussian

blur at multiple σ-values. Examples of this augmentation can be seen in figure 3.13.

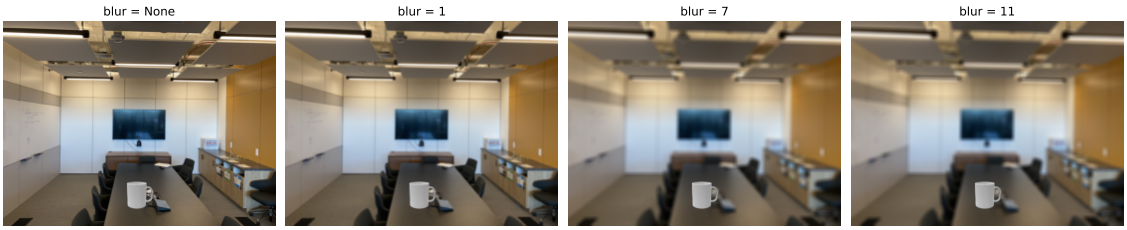

Figure 3.13: Examples of the described background blur augmentation

The augmentation procedure produced 4 images at every render evaluation using $\sigma = [None, 3, 7, 11]$. The same gradient descent training loop is used, except the learning rate is increased to 0.2. This produced the results seen in figure 3.14. Figure 3.14 shows a less noisy loss landscape. The two different initial conditions also seem to lead to more similar paths than before (i.e. the landscape is less chaotic now). Alas, the augmentation had an arguably positive impact.

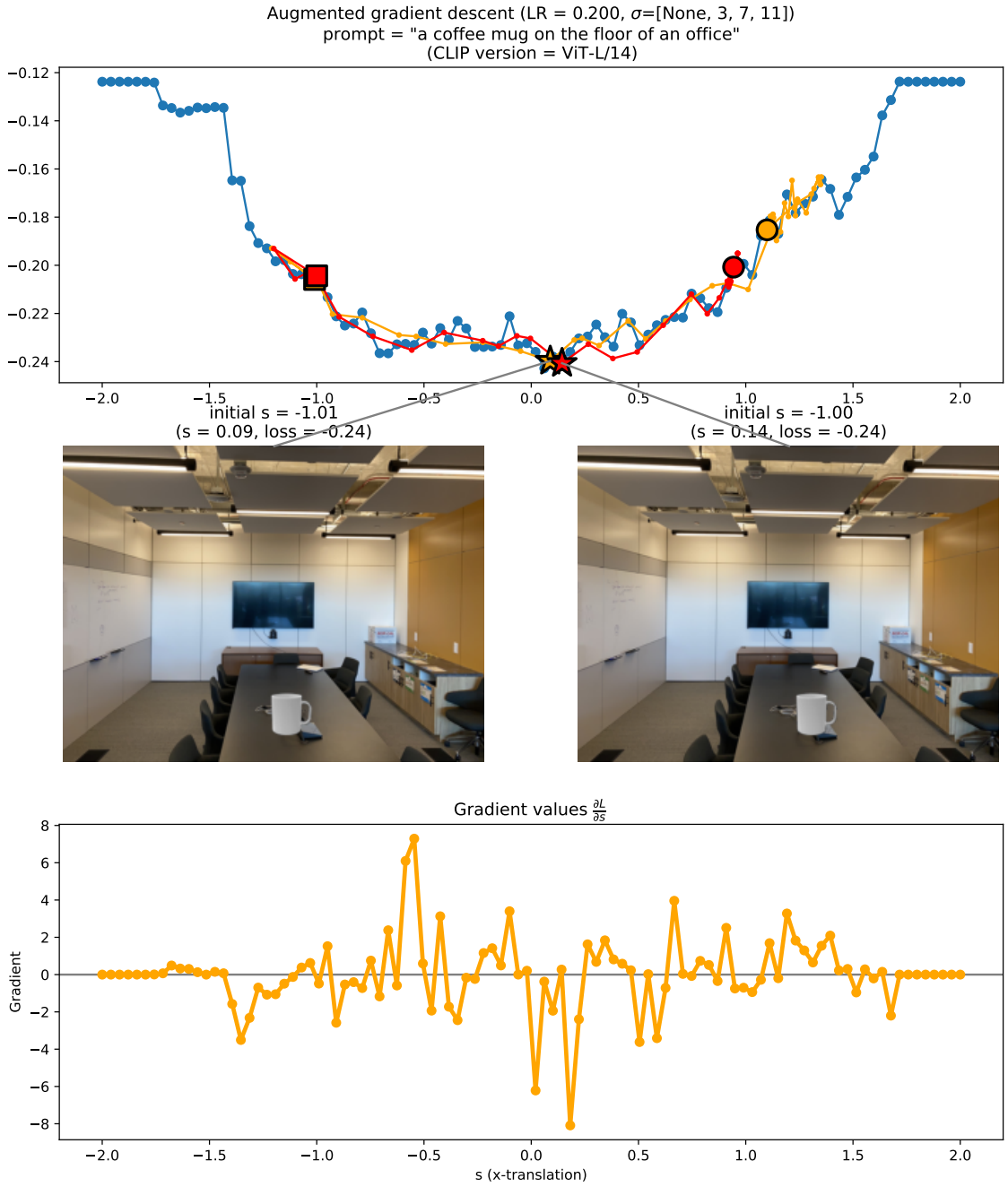

Figure 3.14: Background-blur augmentation gradient descent. The top row shows the CLIP loss landscape overlaid with the gradient descent path (the square is the starting point, the circle is the final point, the star is the minimum value encountered during training). The middle row shows the image corresponding to the minimum found during gradient descent. The bottom row shows the gradient values for the sample s-values.

A comparison of the gradients was made in order to validate if the gradients differ in terms of noise. The gradients from the bottom rows of figures 3.10, 3.11, 3.14 were gathered at a higher resolution this time ($N = 100 \rightarrow N = 300$ values of s). The collected gradients (along with their empirical mean and standard deviation) can be seen in the first row of 3.15. As explained in section2.2.2, the autocorrelation function (ACF) for the gradients can be plotted in order to visually validate noisiness. This can be seen in the bottom row

of 3.15.

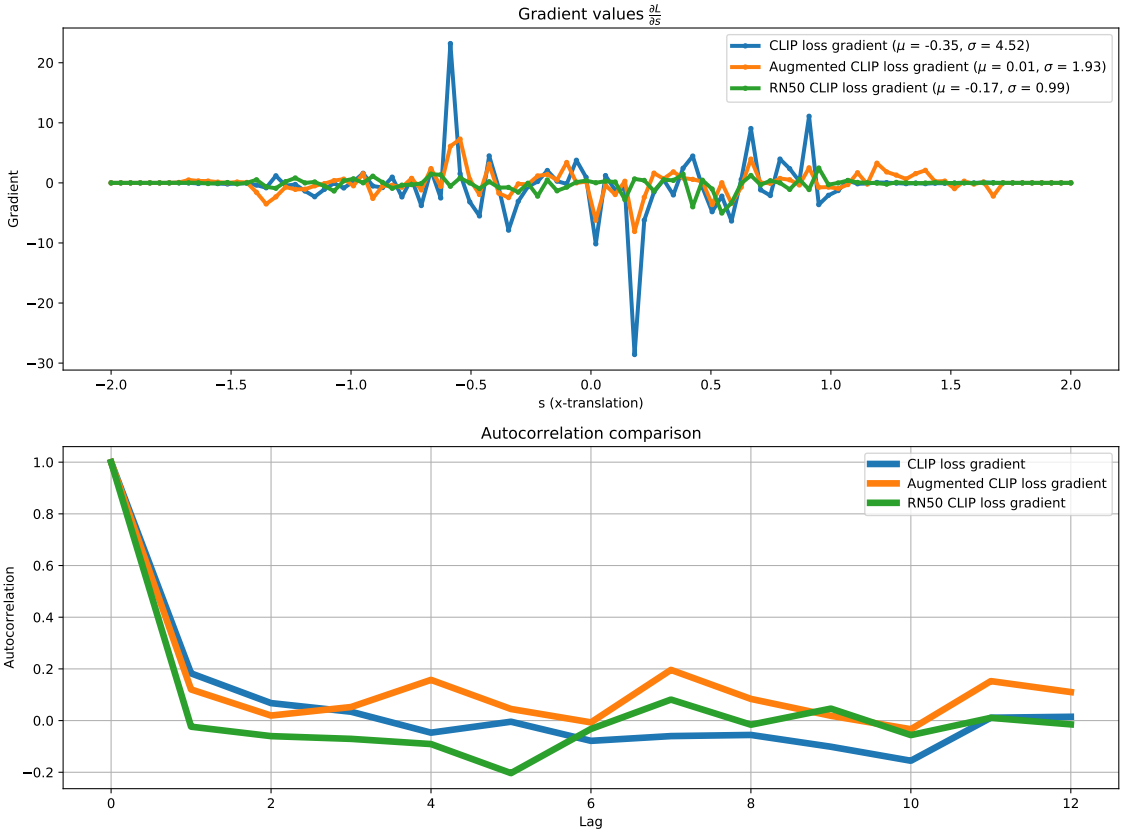

Figure 3.15: Top row shows the gradients plotted with each other, with the calculated statistics in the legend. Bottom row shows the autocorrelation function for the various gradients.

The top row of 3.15 shows that there are quite a few drastic spikes of significant magnitudes in the unmodified gradients. The RN50 and augmented gradients don't exhibit the same spikes. The statistics in the legend also state that the mean values for the gradients are pretty similar (spans from 0.01 to -0.35), but the standard deviation varies significantly between the different gradients (spans from 0.99 to 4.52). Alas, it seems that the RN50 gradients and the augmented gradients are pretty close to each other when compared to the unmodified gradients.

The bottom row of 3.15 shows a comparison of the autocorrelation functions. For context, the ACF for a white noise process will quickly tend towards 0. With that in mind, it seems that every gradient is less random than a white noise process. While there isn't an overwhelming difference between the gradients, it does arguably look like the augmented gradients differ from 0 the most.

A multiparameter renderer was constructed. In addition to x-translation, the renderer also had 3 additional parameters for controlling scale, rotation and y-translation - thus, 4 parameters in total. An analogous gradient descent training loop was set up. The blur augmentation is applied, and the learning rate is decreased to 0.001. The loop is run for 300 iterations, and the results can be seen in figure 3.16.

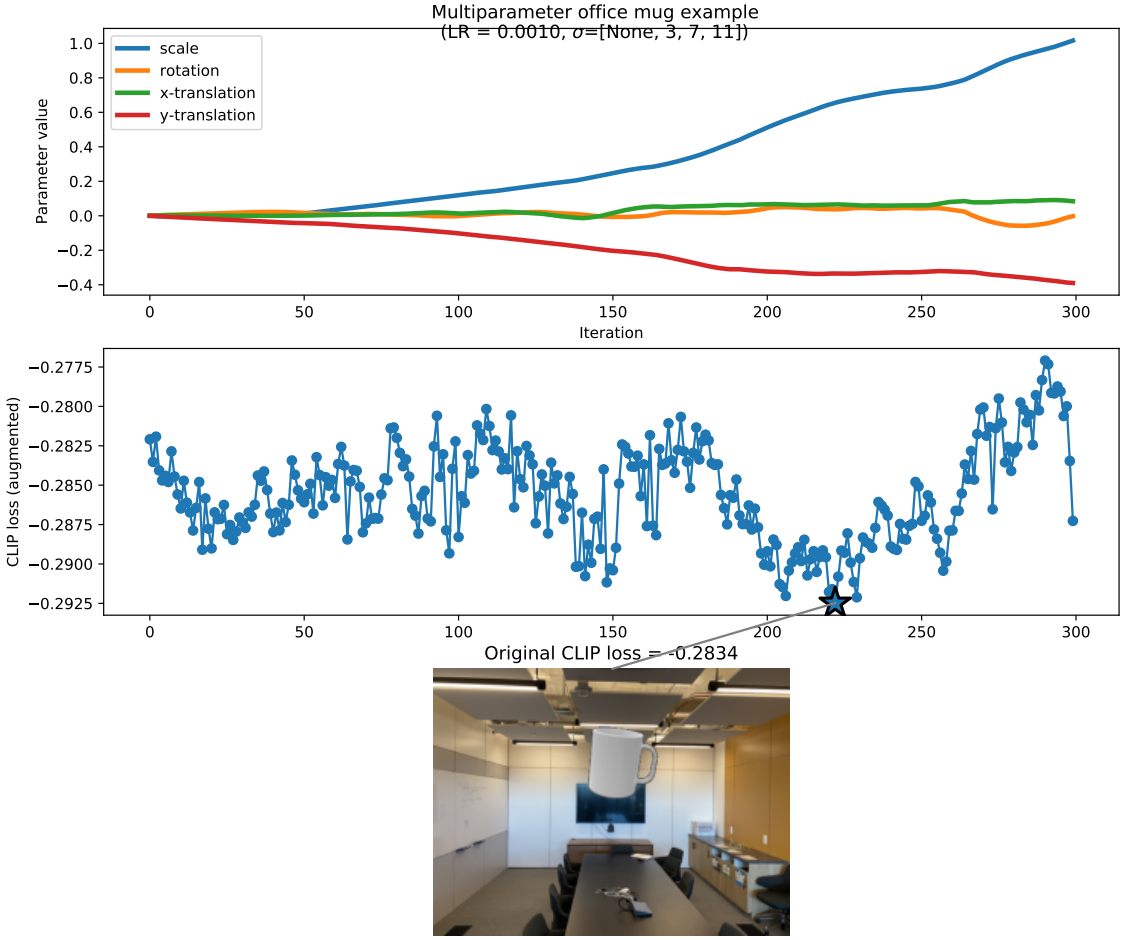

Figure 3.16: Top row shows how the renderer parameters evolve over time. The middle row shows how the augmented CLIP loss evolves over time. The bottom row shows the image for the minimum encountered augmented CLIP loss, with the original CLIP loss written on top.

A brute force search grid was constructed for the multiparameter renderer with a granularity of 20 for each parameter. Thus, the entire search space consists of $20^4 = 160.000$ parameter combinations. The brute force search took just under 2 hours to finish. The top three resulting images from the brute force search can be seen in 3.17

From figure 3.17, it appears that best results from the brute force search yield losses of $-0.2753, -0.2751, -0.2751$. These losses are all higher than the minimum loss found during gradient descent (0.2834). Alas, the brute force fails to find a more optimal solution than gradient descent for the multiparameter case.

To summarize the findings of this subchapter:

- CLIP loss is very noisy and non-convex - even for simple images such as the mug with no background.
- CLIP loss is very sensitive to the rendering parameters s - i.e. very minuscule changes in s can result in significant changes in CLIP loss.
- Gradient descent is very sensitive to initial conditions (e.g. initial s, learning rate, etc.)
- The most effective form of augmentation appeared to be the background blur aug-

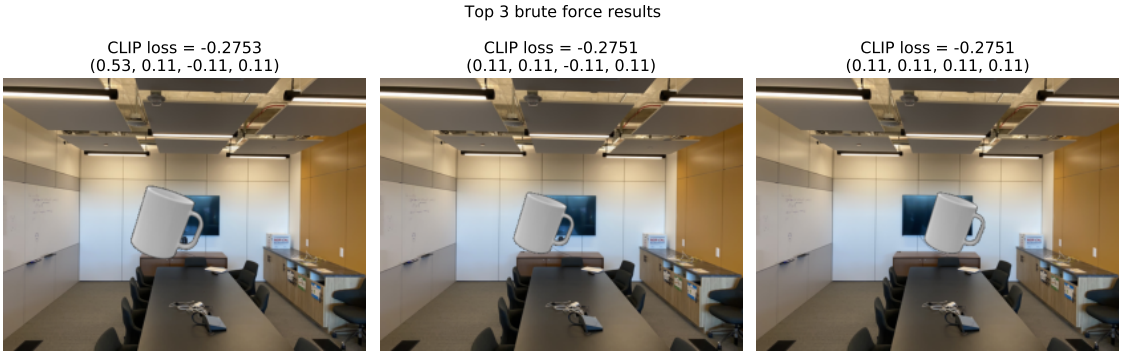

Figure 3.17: Top three results from the brute force search. The corresponding CLIP losses have been written on top of each image.

mentation. By observing the autocorrelation functions (ACF), it appears to slightly improve gradient noise.

- Determining the true global minimum for the CLIP loss is very difficult even for a brute force search. For the example with just 4 transformation parameters, gradient descent seems to win over brute force, both in regard to time complexity and loss minimization.

3.3 3D semantic experiments

Now it's time to move on to using a 3D rendering mechanism. There are many solutions that could be used here - NeRFs are just one possibility in this regard. Due to time constraints, an alternative differentiable rendering mechanism was used. The library pytorch3d[4] provides a differentiable renderer which works using meshes and textures. For the experiments, I found meshes and textures for a small toy cow and a wooden table.

Initially, a single-parameter 3D renderer was set up such that the parameter controls the y-translation of the cow (i.e. moves the cow vertically). The parameter is "bounded" using a tanh function, such that the cow never leaves the rendered image, even as $s \to \pm\infty$. Similar to the 2D mug translation example, a "sanity check" to see how a couple of example text prompts relate to the rendered image in terms of CLIP loss. Again, for the s-values, 100 linearly spaced samples from $[-2, 2]$ are taken. The results can be seen in figure 3.18.

[4] https://pytorch3d.org/

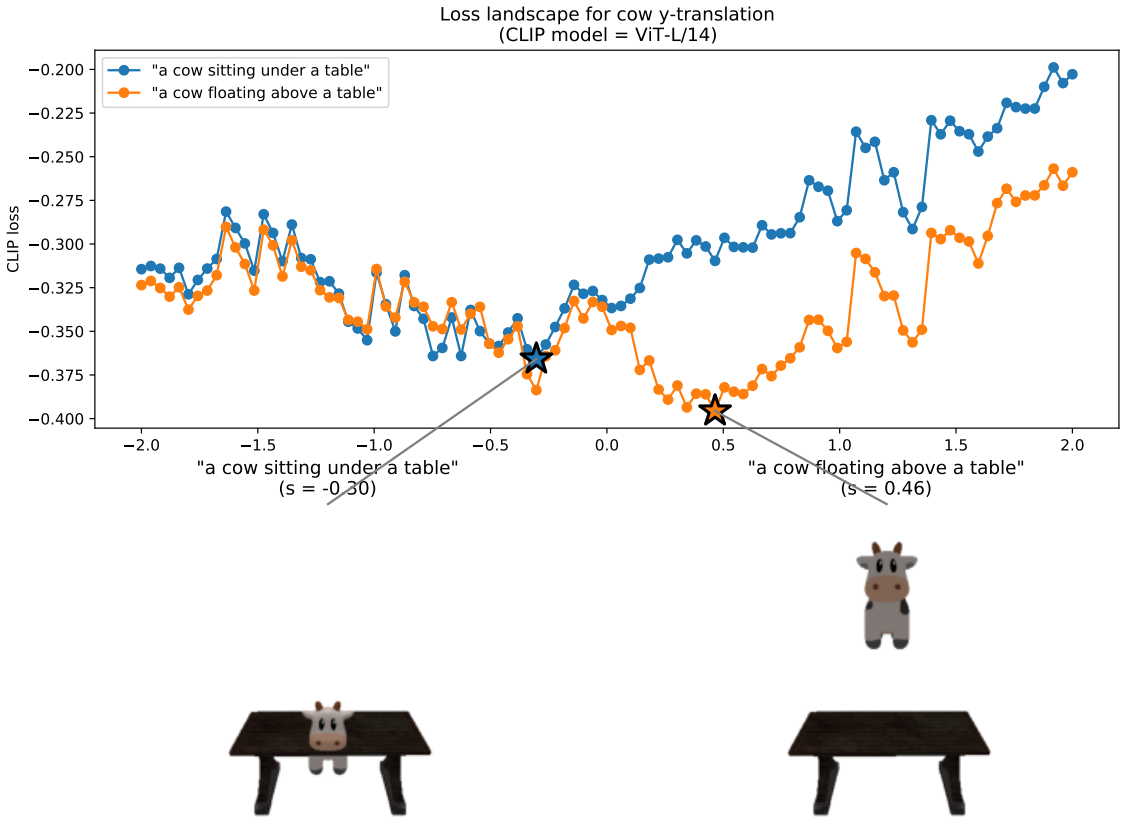

Figure 3.18: Top row shows the CLIP loss landscape for the different text prompts (star denotes the lowest encountered loss value for the corresponding text prompt). Bottom row shows how the rendered image looks for the corresponding parameter value.

Remarkably, the losses obtained in figure 3.18 are significantly lower than those for the mug examples from last subchapter (e.g. fig 3.8). The cow example reaches losses of almost -0.40, whereas the mug examples merely reached losses of about -0.23.

Now, the transformation is changed to z-translation (instead of y-translation). As the parameter value decreases, the z-translation will move the cow towards the camera, and, inversely, further away from the camera as the value increases.

An experiment is set up to see how well gradient descent performs for the single-parameter cow z-translation example. A training loop was constructed using Adam with initial LR = 0.05 and a cosine annealing LR schedule. The loop is run for 300 iterations for each of the two starting points. The results can be seen in figure 3.19.

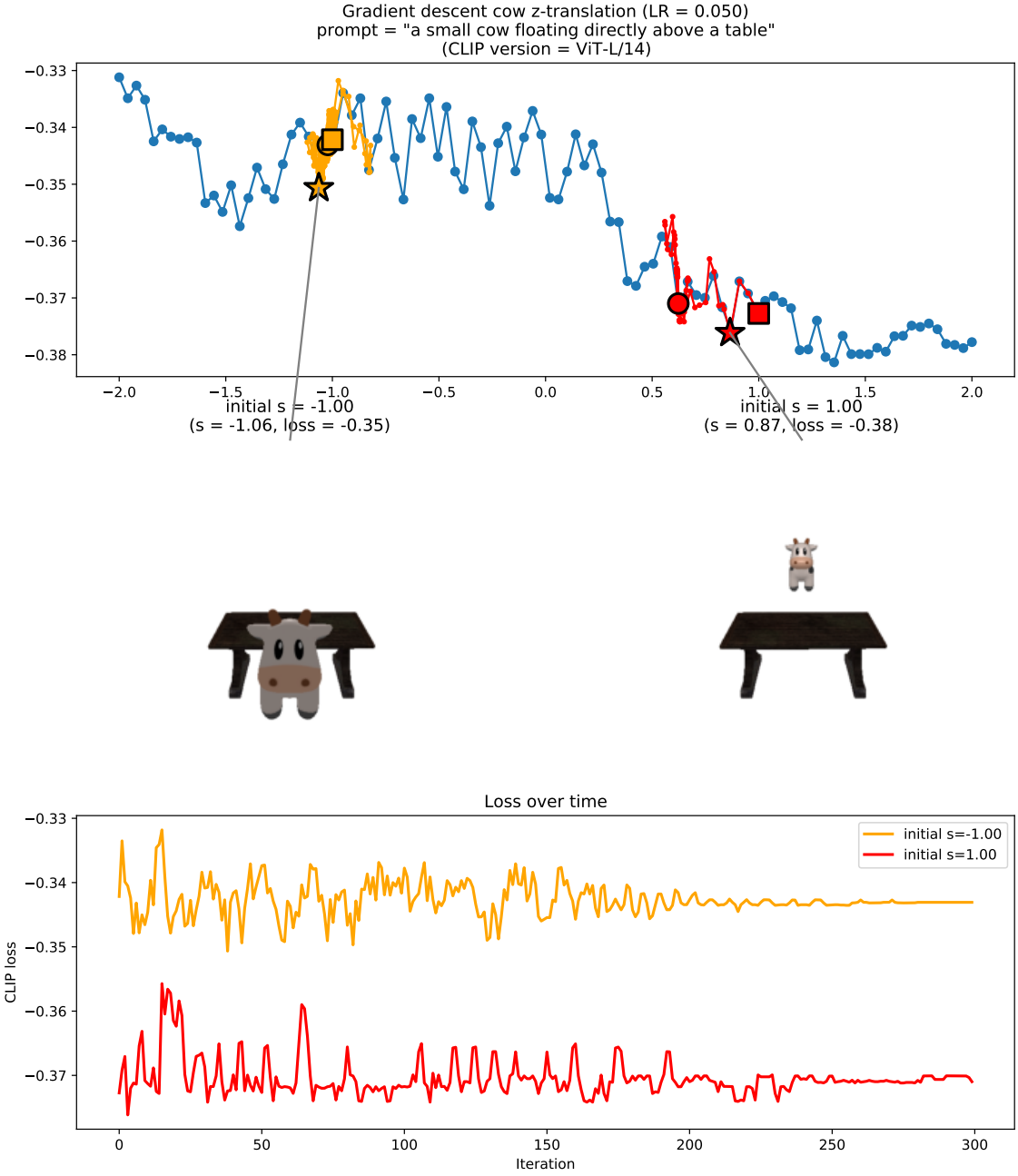

Figure 3.19: Gradient descent for cow z-translation. Top row shows the loss landscape overlaid with the two gradient descent paths (square denotes starting point, circle denotes final iteration, star denotes lowest loss encountered). Middle row shows the produced images for the encountered minima. Bottom row shows the learning curve (i.e. loss over time) for the two runs.

Figure 3.19 shows that the gradient descent solutions don't end up in the global minimum. An important detail here is the fact that the loss landscape shows the most optimal s-value being around $s \approx 1.5$. However, the value which actually places the cow directly above the table is $s = 0$. This illustrates an underlying issue that has to do with the perspective from which the scene is observed. It appears that the scene needs to be rendered from multiple perspectives in order to obtain sufficient information about the cow's z-translation. The issue is illustrated in figure 3.20.

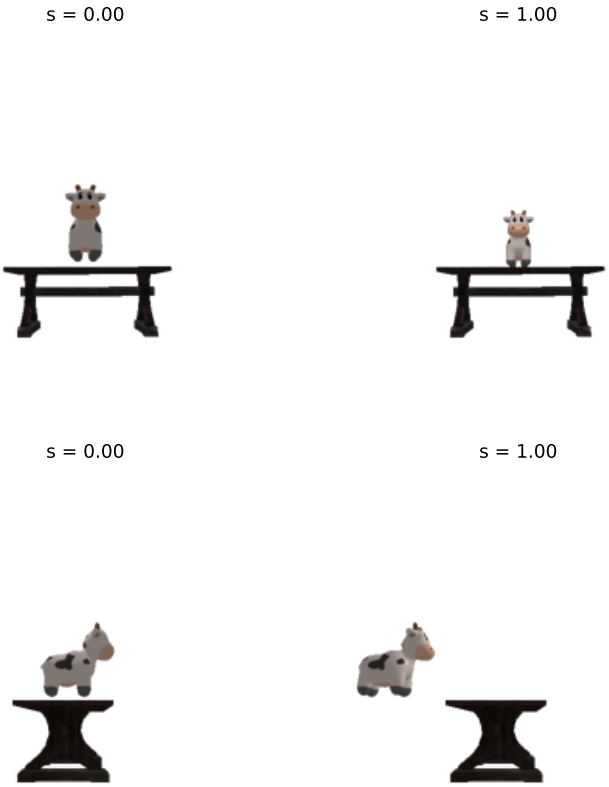

Figure 3.20: A demonstration of how perspectives can be confusing in 3D. The extent of the z-translation is not really clear from the perspective in the first row. The alternate perspective in the second makes it much easier to judge how far back the cow was moved.

As a means of mitigating the issue of confusing perspectives in 3D, the next experiment attempts to explore the capabilities of novel view augmentation. The idea is to render multiple images of the scene seen from different perspectives, which can effectively be used as image augmentations. The exact process is illustrated in figure 3.21.

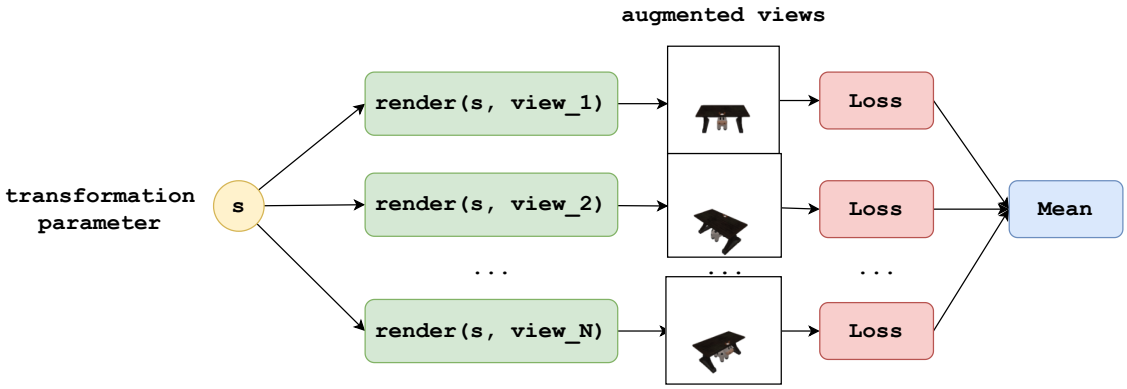

Figure 3.21: Overview of the novel view augmentation process

The training loop was replicated from the previous experiment (i.e. the experiment that produced fig 3.19), but it has the novel view augmentation this time. The alternate per-

spectives all had the same distance to the origin and the same elevation angle. The azimuth angle, however, was modulated by adding to it the degrees $[45, 90, 135, 180, 225, 270, 315]$ - this yielded a total of 8 rendered images per iteration. The results from running gradient descent are shown in figure 3.22.

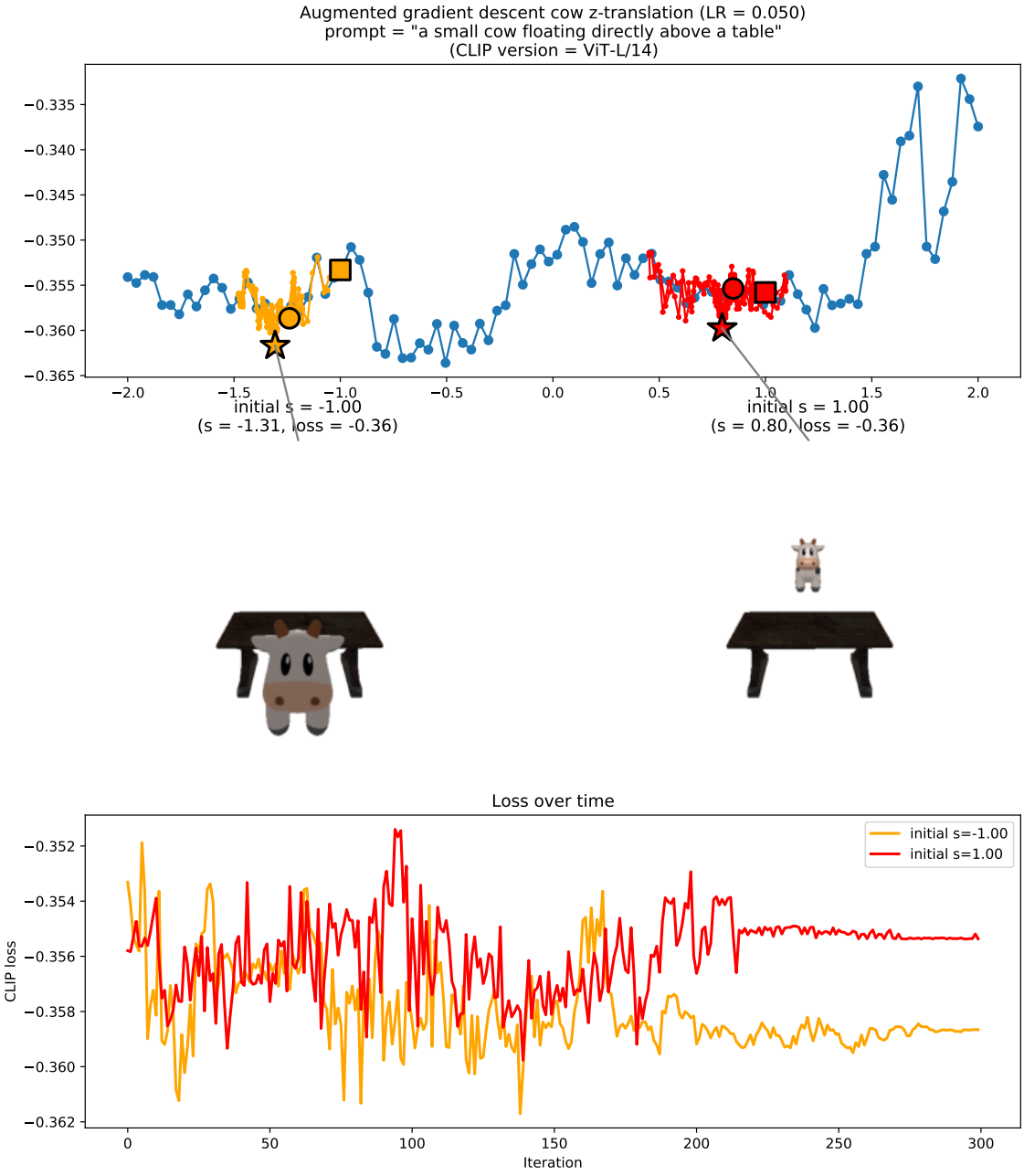

Figure 3.22: Augmented gradient descent for cow z-translation. Top row shows the loss landscape overlaid with the two gradient descent paths (square denotes starting point, circle denotes final iteration, star denotes lowest loss encountered). Middle row shows the produced images for the encountered minima. Bottom row shows the learning curve (i.e. loss over time) for the two runs.

Figure 3.22 shows that the loss landscape has changed quite drastically compared to the non-augmented results in fig 3.19. The minimum is no longer located around $s = 1.5$, but

has instead shifted towards $s = -0.5$. This isn't exactly $s = 0$, but it's a step in the right direction. As for the quality of the gradient descent solutions, not much has improved. Neither of the two starting points end up yielding a solution near the global minimum. In fact, the loss over time (i.e. training curve) appears seems a bit more noisy this time.

Now it's time to introduce additional transformation parameters. Respectively, the new parameters are: [scale, x-rotation, y-rotation, z-rotation, x-translation, y-translation, z-translation]. Similar to the 2D example, all the parameters are "bounded" using the sigmoid function for scale, and tanh for all other parameters. Examples of rendered images using various parameters can be seen in figure A.1.

The problem can effectively be treated the same as the 2D multiparameter mug renderer from the previous subchapter, where instead of 4 parameters, there is now 7 in total. A training loop was set up for 300 iterations with Adam, initial LR = 0.01, a cosine annealing learning rate scheduler, and the starting point $\mathbf{s} = \mathbf{0}$. The novel view augmentation technique is *not* included for this experiment. The results can be seen in figure 3.23.

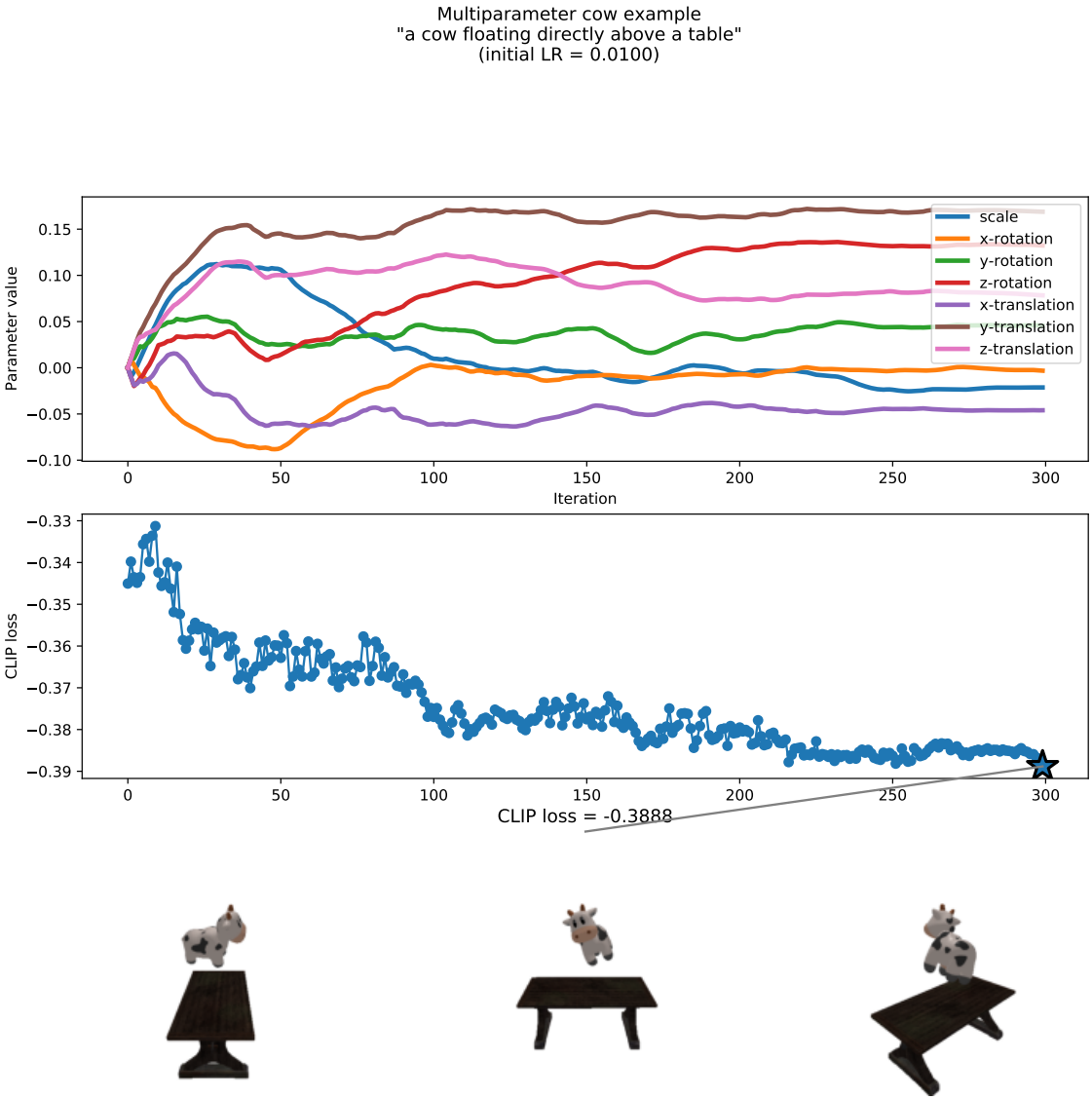

Figure 3.23: Non-augmented gradient descent for the multiparameter cow renderer. Top row shows how the transformation parameter evolve over time. Middle row shows how the CLIP loss evolves over time. Bottom row shows three rendered perspectives of the optimal solution.

Figure 3.23 shows that the loss seems to converge to a minimum over time. Also, the values of the parameters **s** appear to lie in the range of $[-0.10, 0.15]$. Despite their magnitude, these small parameter values are apparently capable of reducing the loss from the initial -0.345 to about -0.39. Also, the fact that the y-translation parameter goes from 0 to a bit over 0.15 is interesting, since this is what actually makes the cow float above the table. It also seems to slightly rotate the cow.

For the next experiment, the same training loop for the multiparameter cow renderer is used. However, this time, the novel view augmentations are added. This produced the results seen in figure 3.24

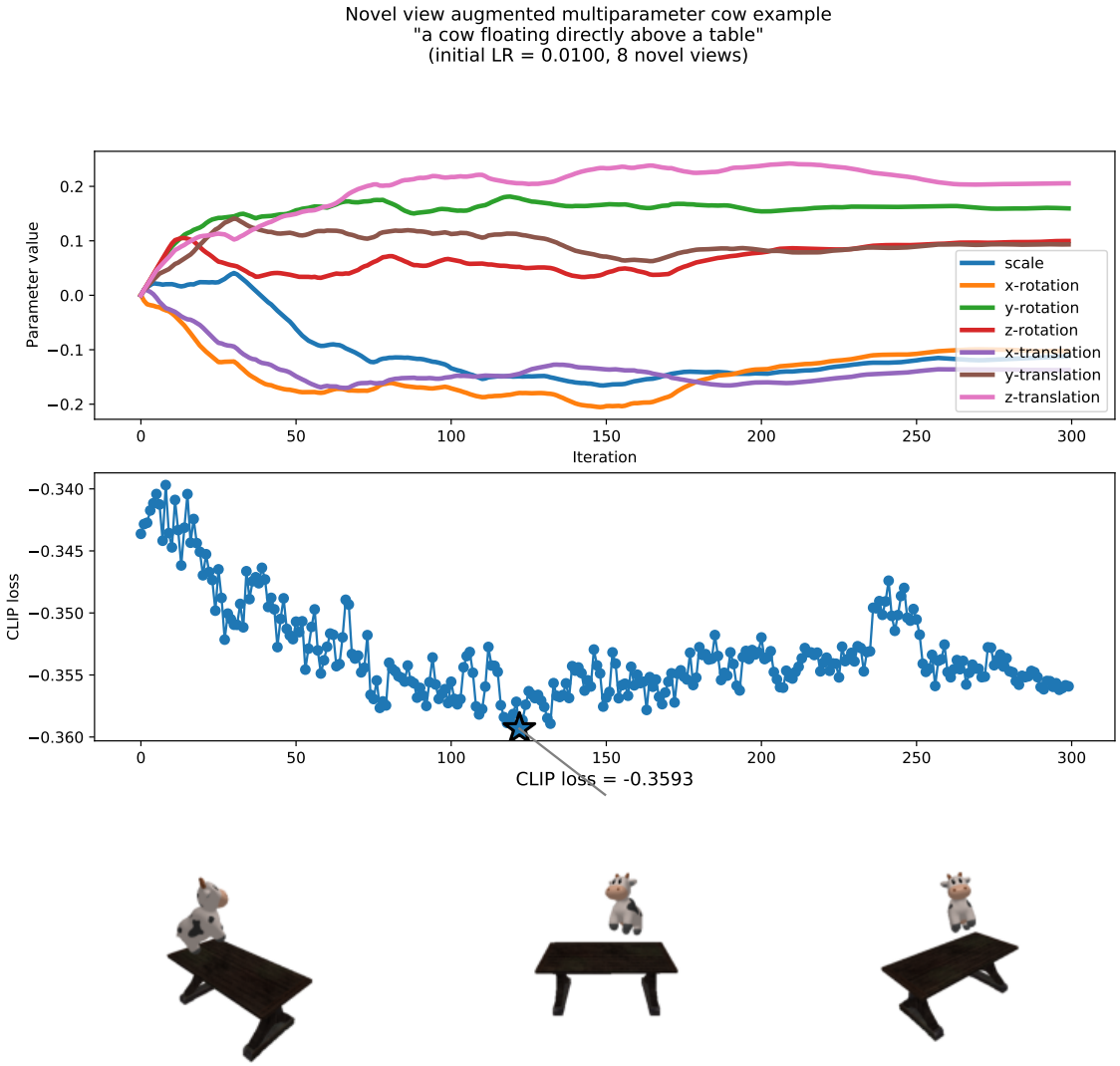

Figure 3.24: Novel view augmented gradient descent for the multiparameter cow renderer. Top row shows how the transformation parameter evolve over time. Middle row shows how the CLIP loss evolves over time. Bottom row shows three rendered perspectives of the optimal solution.

Figure 3.24 shows that the transformation still puts the cow slightly above the table (i.e. increases y-translation). The loss does not seem to converge as nicely as in the previous experiment. The cow seems to be translated to the side of the table, and, again, is slightly rotated. Thus, it's hard to say that the novel view augmentations improved things much for the multiparameter cow renderer.

Recall the brute force example from the previous subchapter. The search space consisted of $20^4 = 160.000$ parameter combinations, which took just under 2 hours to search through. Since we now have 7 parameters, a grid of the same resolution would result in $20^7 = 1.280.000.000$ combinations, which, by extrapolation, would take around 16.000 hours to search through. Considering the fact that brute force searching a grid of similar granularity is pretty much computationally infeasible, a brute force experiment was not carried out.

To summarize the findings of this subchapter:

- When performing manipulations in 3D, sometimes rendered images from additional

perspectives are required in order to gain more visual information of the effect of the manipulation (see figure 3.20).
- Novel view augmentation had the effect of drastically changing the shape of the loss landscape. When applying the novel view augmentations to the multiparameter cow example, it did not seem to have a significantly positive impact.
- Even though the image and text embeddings are very similar for the cow renderer example (i.e. losses of almost -0.40), it's still difficult for gradient descent to reach the global minimum.

3.4 Disentangled NeRFs

Neural Radiance Fields (NeRFs) are capable of rendering photorealistic images from arbitrary perspectives. An essential part of the full pipeline is being able to sufficiently disentangle a foreground object from a NeRF scene. This is where [3] will be applied. Once this disentanglement has been obtained, it becomes possible to perform 3D manipulations on the foreground object before re-inserting the object into the original scene.

At the beginning of the project, a lot of effort was put into finding a faster way to train new NeRFs. Traditionally, the training time is ≈ 12 hours for a single scene, and the training has to be done twice per scene (once masked, once unmasked).

After trying out a couple of different implementations, a fork was made of https://github.com/ashawkey/torch-ngp, which is based on [2]. It is written in PyTorch with bindings to some CUDA kernels. The goal was to modify the code to replicate the example from [3] which disentangles the TV from the back wall of the office. To this end, two NeRFs were trained (with/without masking the TV). The results from training can be seen in figure 3.25.

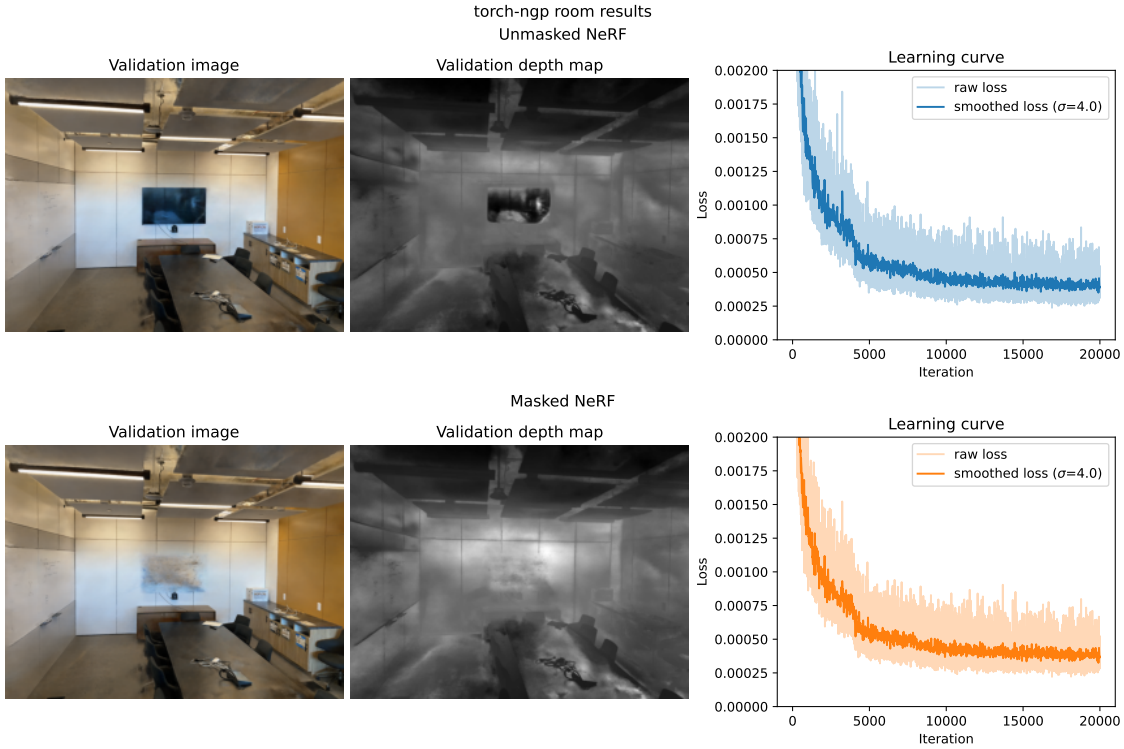

Figure 3.25: Results from training two NeRFs. Top row shows results from the unmasked example. Bottom row shows results from the masked example. First column shows an image rendered for a validation pose (i.e. not seen in the training data). Second column shows the associated depth map generated for the same validation pose. The last row shows the learning curves obtained from the training process.

Figure 3.25 shows that the loss curves have somewhat converged after 20k iterations. However, the depth maps are not looking very clear considering what a traditional NeRF is usually capable of producing. Disregarding the noisy depth maps for now, a couple of disentanglement operations were performed for the two NeRFs. The first operation should only show the disentangled foreground object. The second operation should re-insert the disentangled foreground object into the background scene. The results can be seen in figure 3.26.

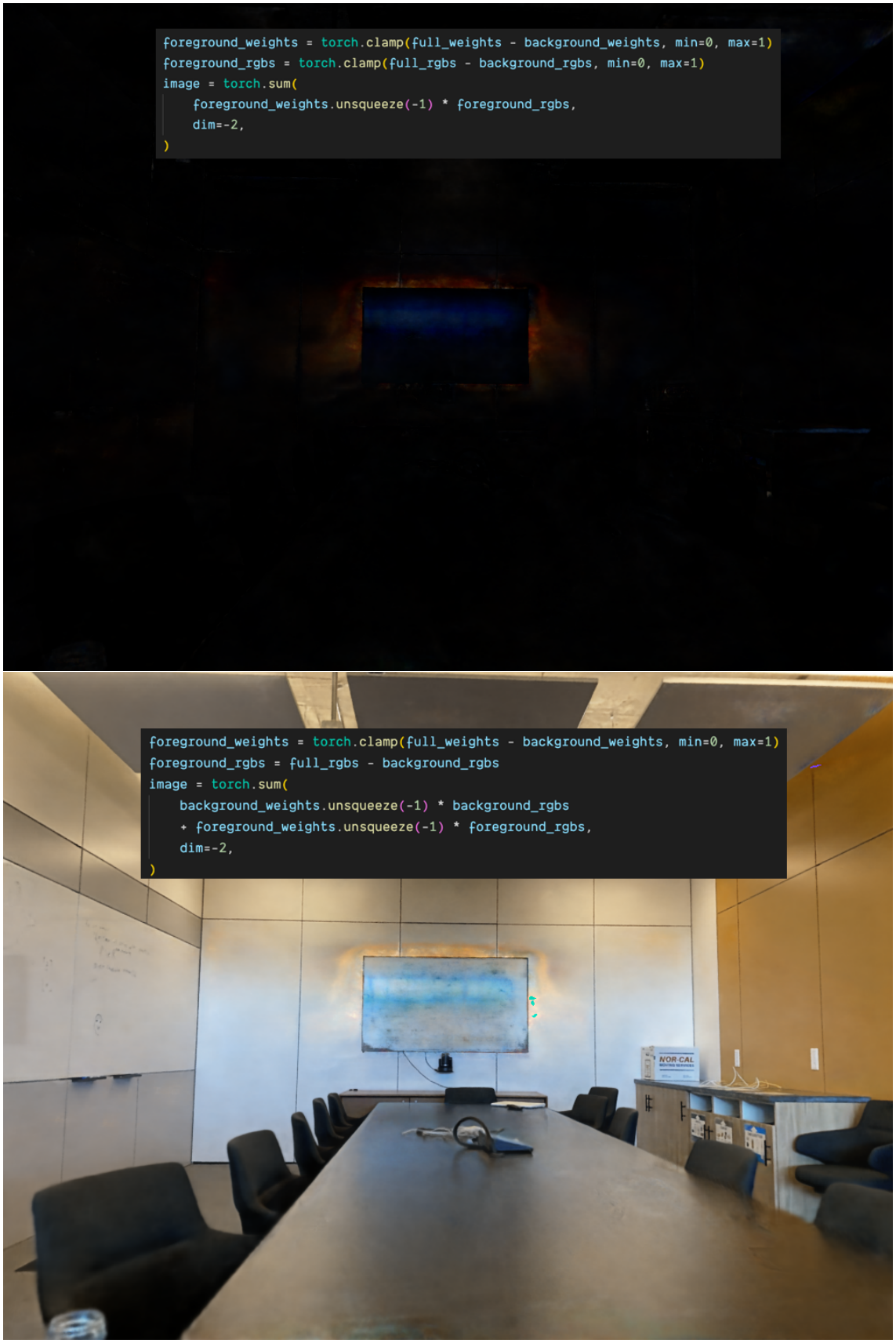

Figure 3.26: TV disentanglement results. The exact equation used (in code form) for making each image is shown at the top.

Figure 3.26 shows some rather underwhelming results for the disentanglement operations. It was a bit unclear why this was happening - below is a list of suspected reasons:

- Code error (pretty likely, considering the fact that an entirely different codebase is used)
- Scenes weren't trained for long enough (poor depth maps could be a symptom of this)
- Too many collisions are happening with the multiresolution hash encoding, which makes disentanglement difficult (again, poor depth maps could be a symptom of this)

Instead of going too far down the debugging rabbit hole, a different set of pre-trained vanilla NeRFs was obtained. These NeRFs were trained on images of a red flower. The results from training these vanilla NeRFs can be seen in figure 3.27.

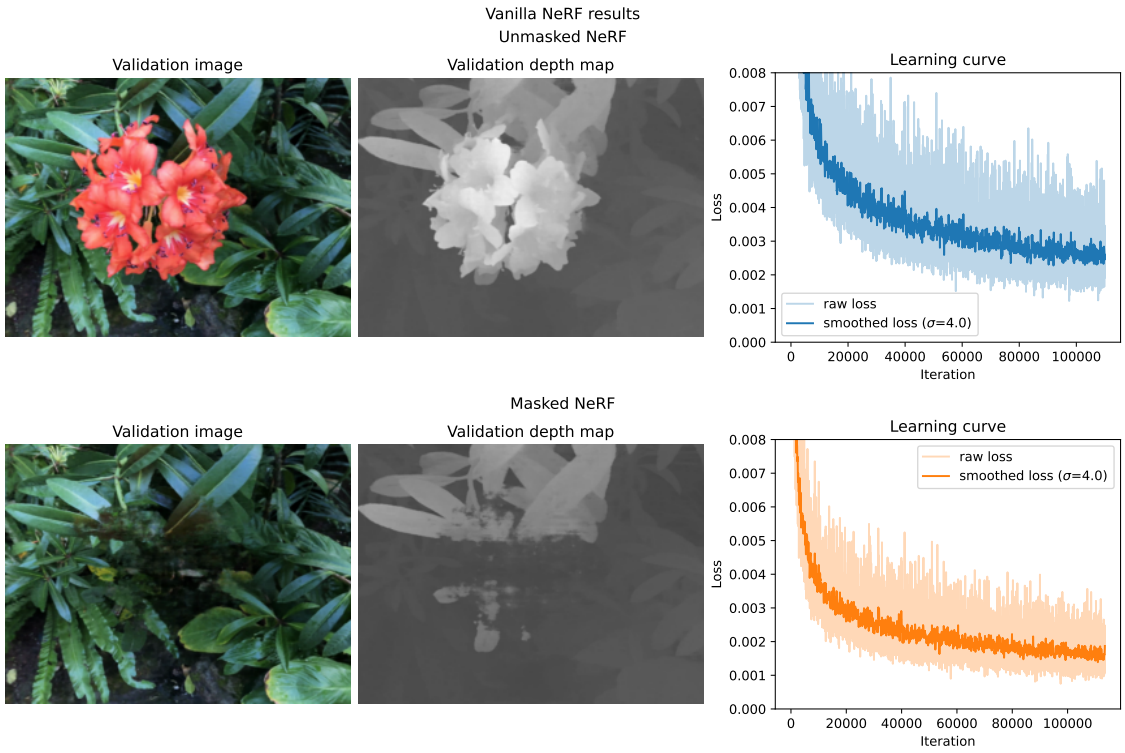

Figure 3.27: Results from training two vanilla NeRFs. Top row shows results from the unmasked example. Bottom row shows results from the masked example. First column shows an image rendered for a validation pose (i.e. not seen in the training data). Second column shows the associated depth map generated for the same validation pose. The last row shows the learning curves obtained from the training process.

Figure 3.27 shows much clearer depth maps compared to those seen in figure 3.25. The learning curves seem to be a bit more noisy this time, as well - in fact, it seems that the previous learning curves were a bit more stable towards the end.

The same two disentanglement operations were performed for the flower example - i.e. first operation only shows the disentangled object, and the second operation shows the foreground object re-inserted into the background scene. The results can be seen in figure 3.28.

Figure 3.28: Flower disentanglement results. The exact equation used (in code form) for making each image is shown at the top.

The results from figure 3.28 are significantly better than those obtained for the TV disentanglement. However, some visual artifacts can still be observed - especially for the re-insertion example on the bottom row. The flower becomes a bit ”translucent” in its appearance, as if it has lost some of its density.

The exact reason behind the artifacts was never quite understood. Trying different formulas for the disentanglement (e.g. adding/removing clamp operations, adding expanded terms, etc.) did not improve the results either. Again, instead of going too far down the debugging rabbit hole, the artifacts were ignored in order to continue with the transformations.

The methodology for disentangled NeRF manipulations (described in section 2.1.3) can now be applied to the disentangled flower. This produced the results seen in figure 3.29.

Figure 3.29: Flower transformations. First row shows scaling operations. Second row shows translation operations stated as [x,y,z]. Third row shows rotation operations stated as [x,y,z] (in degrees)

Figure 3.29 shows that the transformation parameters work to a certain extent. It seems to be possible to manipulate the foreground object while keeping the background intact.

There is some visible noise that can be seen on the background - the intensity of which depends on the extent of the manipulation. Also, the z-rotation seems to skew the flower, which is not ideal.

However, there is a limit to how much the foreground object can be transformed without starting to degenerate. This is due to the fact that NeRFs are only able to interpolate, and are unable to extrapolate to wildly unseen perspectives - i.e. NeRFs are ”bounded” to the given input views, and trying to escape these bounds will yield weird images. Trying a wider range of transformation parameters produced the results seen in figure 3.30.

Figure 3.30: Flower transformations. First row shows scaling operations. Second row shows translation operations stated as [x,y,z]. Third row shows rotation operations stated as [x,y,z] (and in degrees)

The first row in figure 3.30 shows a type of ”mosaic” visual artifacts appearing when the flower scale becomes very small. This could likely be the same artifact observed in the extreme translations in the second row. The third row shows that the flower starts disappearing for extreme rotations about x and y. However, the extreme z-translation does not appear to exhibit the same type of artifact.

- Using the *torch-ngp* implementation (i.e. multiresolution hash encoding) of NeRF wasn't immediately well-suited for disentangled NeRFs. The reason behind this is a bit unclear and requires further debugging.
- Using vanilla NeRFs, it appears to be possible to perform transformations in 3D space - within the limitations imposed by the NeRFs.
- There are still a few issues with the implementation that need to be resolved - e.g. the ”translucency” visual artifact, and the skewed z-rotation.

4 Conclusion

4.1 Main conclusions

In summary, the results obtained in the previous subchapters indicate that it is possible to perform semantically guided volumetric object manipulations using the chosen methodologies.

The experiments with the synthetic renderers (both 2D and 3D) demonstrate successes and failures when optimizing a differentiable renderer with respect to CLIP loss. If the optimization is only for a single parameter, there are viable alternatives to gradient descent. However, the benefit of using gradient descent becomes more pronounced when optimizing for multiple parameters - especially since the alternatives begin to perform progressively much worse as the parameter count increases. Many attempts were made to improve the stability of gradient descent, e.g. by augmenting the text prompts and rendered images. This had a measurable, but not entirely significant impact, and as such, the most deciding factor for the success of gradient descent remains to be the learning rate and starting point.

The NeRF manipulations appear to work to a certain degree, although the extent of how freely the object can be manipulated is limited by the bounds of the NeRF. As such, the manipulations will likely work better for NeRFs of a more unbounded nature. Also, there seems to be a few issues with the implementation - e.g. the ”translucent” re-inserted flowers and the skewed z-rotation. However, these issues seem to be mostly code-related, since the re-insertion technique has been proven to work in [3].

4.2 Future work

4.2.1 Text prompt augmentation

A lot of work is currently being made in order to ”unlock” the full potential of CLIP. In that regard, many are exploring the idea of *prompt engineering*, which essentially revolves around rephrasing the text prompt to elicit different results from the model. Prompt engineering was explored a bit in 3.2, but it probably makes sense to perform these experiments in a more structured manner.

On CLIP’s official repository[1], there is a notebook demonstrating how using various formulations involving the class name can improve the zero-shot classification capabilities of CLIP on ImageNet. For example, the formulations could be: ”a bad photo of a .”, ”a cartoon .”, ”a jpeg corrupted photo of the .”. These various formulations try to actuate some of the context regarding the text captions for the images in the dataset. A lot of these images are scraped from the internet, which is why they experiment with many prompts related to this origin (e.g. ”a low resolution photo of .”, ”itap of .” (itap is a subreddit), etc.). After compiling 80 of such templates, they proceeded to perform a sequential forward selection to determine which prompts performed the best on ImageNet. The search terminated after ensembling 7 templates. These templates are tried out for the 3D cow rendering example, producing the results seen in figure 4.1.

[1] https://github.com/openai/CLIP

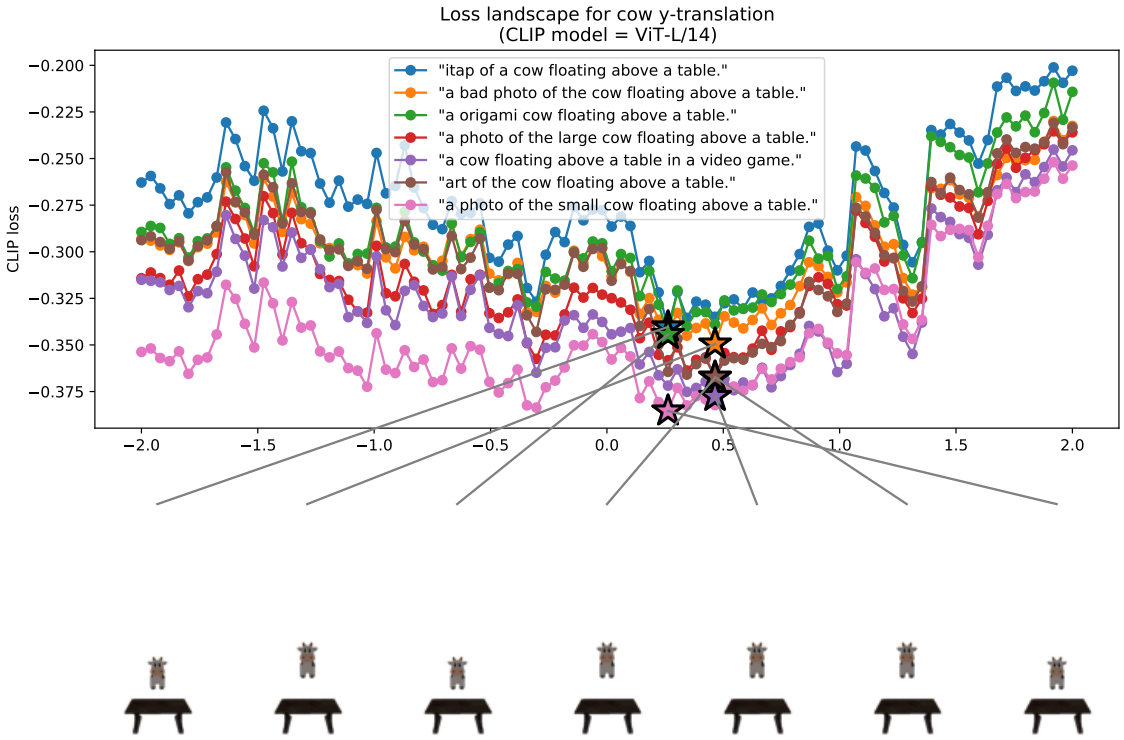

Figure 4.1: Text prompt experiment for the 3D cow y-translation renderer. It seems like the shape of the landscape doesn't change much, but is instead uniformly shifted up/down.

As mentioned in 3.2, it might also be worthwhile to try using an ensemble of CLIP models at once rather than using only a single version.

4.2.2 Image augmentation

While many different variations of 2D image augmentation were experimented with in section 3.2, there was a method in particular which seemed promising (which wasn't mentioned in the section). It can be described as a sort of "Gaussian sampling" technique for obtaining augmented images. Given the transformation parameter(s) **s**, a fixed amount of perturbations **N** are made in accordance to I.I.D samples from $\mathcal{N}(0, \sigma)$ (the reparameterization trick is applied here). The loss is then calculated as the mean of the losses for all the images. The mechanism is illustrated in figure 4.2, and the results from running gradient descent using the augmentation can be seen in figure 4.3.

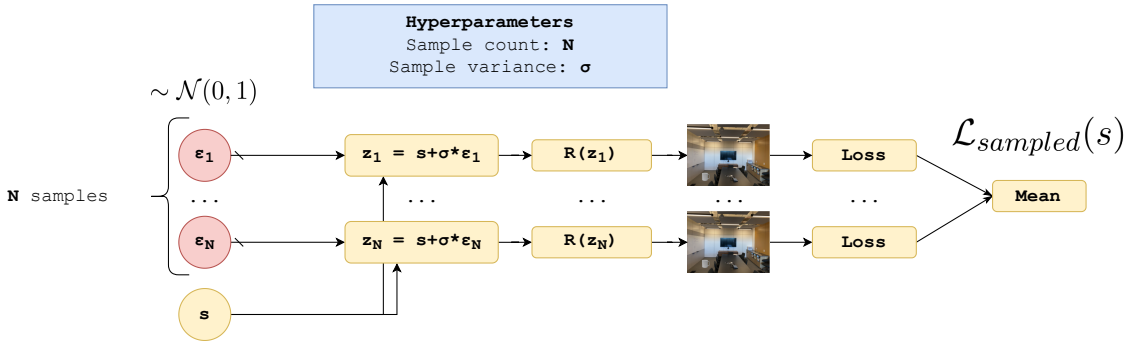

Figure 4.2: "Gaussian sampling" method for image augmentation. Red nodes are stochastic (i.e. non-differentiable).

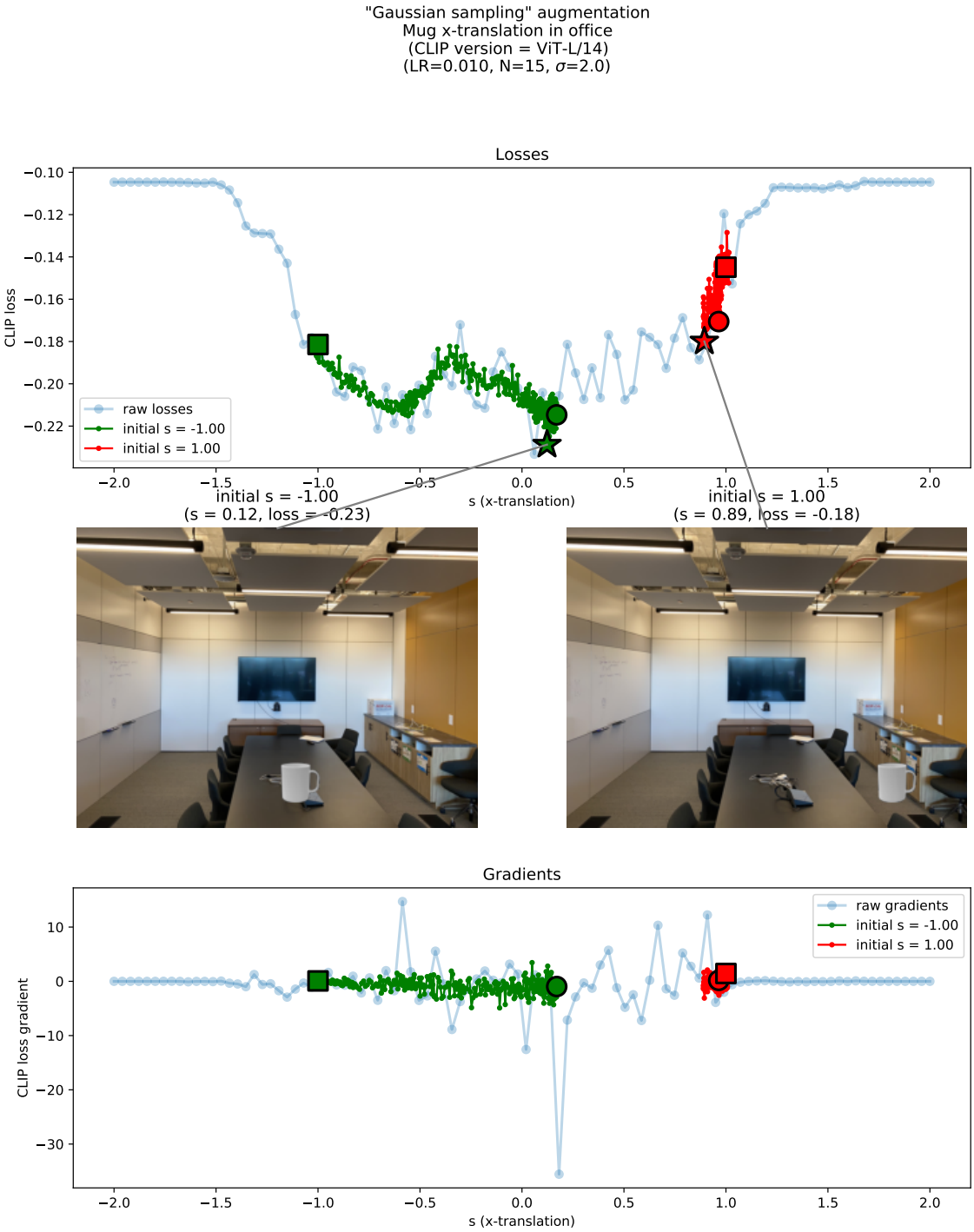

Figure 4.3: Gradient descent using the ”Gaussian sampling” augmentation method. The top row shows the CLIP loss landscape overlaid with the gradient descent paths (the square is the starting point, the circle is the final point, the star is the minimum value encountered during training). The middle row shows the image corresponding to the minimum found during gradient descent. The bottom row shows the gradient values for the sample s-values.

In short, the results in figure 4.3 seem rather promising (it successfully descends into a good minimum for one of the starting points, and the gradient look much more smooth).

Thus, for future work, it might be fruitful to perform further experiments with this particular type of augmentation.

4.2.3 Hybrid optimization

Even after many tricks, gradient descent is still rather imperfect, and the quality of the obtained solutions still depends a lot on the configuration (e.g. learning rate, schedule for learning rate, starting points, augmentation hyperparameters, etc.). It might make sense to draw inspiration from some of the gradient-free global optimizers that were experimented with in section 3.1. Maybe trying an ensemble of configurations and performing gradient descent multiple times could lead to finding better solutions.

4.2.4 NeRF improvements

There is definitely some room for improvement in regard to the *torch-ngp* NeRF results. It could be worthwhile to investigate why the disentanglement did not work as expected. It might be important to uncover if the issue is code-related, or if disentanglement is fundamentally incompatible with that type of NeRF.

As for the flower NeRF results, there seemed to be some visual artifacts upon re-inserting the disentangled foreground object into the scene - even without performing any transformations. It's quite likely that the issue is code-related, since the methodology has been proven to work in [3]. In my opinion, it definitely makes a lot of sense to figure out the reason behind this issue.

Also, given more time, it would have been very insightful to apply the disentanglement and manipulation to several other NeRF datasets. Currently, it cannot be entirely ruled out that the types of observed visual artifacts (e.g. the mosaic artifact) are specific to just the flower scene. Finding a dataset which is less bounded allow for greater manipulations in 3D space (e.g. rotate more freely, etc.).

4.2.5 Object-centric transformations

The NeRF transformations (specifically rotations) experimented with were all centered about the origin (i.e. [0,0,0]). If the goal is to rotate the object around itself, it requires that the 3D position of the foreground object be known (or at least estimated). The idea was to research how a "centre-of-mass" method (such as the one proposed in the end of section 2.1.3) could be implemented. However, this was deemed out-of-scope since there were issues with the regular NeRF transformations (i.e. visual artifacts appearing). In the future, this could be an interesting direction to take, since it allows for more granular manipulation of the object in the scene.

4.2.6 Complete entire pipeline

The main idea of this project was to combine CLIP-guided optimization and NeRF manipulations into a single pipeline. In practice, combining it all requires introduces a wide range of technical challenges in regard to code and hardware (specifically VRAM), which are beyond the scope of the project. Also, there are still existing issues which need to be resolved first (e.g. visual artifacts from transformations).

Bibliography

[1] Ben Mildenhall et al. “NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis”. In: *CoRR* abs/2003.08934 (2020). arXiv: 2003.08934. URL: https://arxiv.org/abs/2003.08934.

[2] Thomas Müller et al. “Instant Neural Graphics Primitives with a Multiresolution Hash Encoding”. In: *CoRR* abs/2201.05989 (2022). arXiv: 2201.05989. URL: https://arxiv.org/abs/2201.05989.

[3] Sagie Benaim et al. *Volumetric Disentanglement for 3D Scene Manipulation*. 2022. DOI: 10.48550/ARXIV.2206.02776. URL: https://arxiv.org/abs/2206.02776.

[4] Alec Radford et al. “Learning Transferable Visual Models From Natural Language Supervision”. In: *CoRR* abs/2103.00020 (2021). arXiv: 2103.00020. URL: https://arxiv.org/abs/2103.00020.

[5] Aditya Ramesh et al. *Hierarchical Text-Conditional Image Generation with CLIP Latents*. 2022. DOI: 10.48550/ARXIV.2204.06125. URL: https://arxiv.org/abs/2204.06125.

[6] Verica Lazova et al. *Control-NeRF: Editable Feature Volumes for Scene Rendering and Manipulation*. 2022. DOI: 10.48550/ARXIV.2204.10850. URL: https://arxiv.org/abs/2204.10850.

[7] Can Wang et al. *CLIP-NeRF: Text-and-Image Driven Manipulation of Neural Radiance Fields*. 2021. DOI: 10.48550/ARXIV.2112.05139. URL: https://arxiv.org/abs/2112.05139.

[8] Ashkan Mirzaei et al. *LaTeRF: Label and Text Driven Object Radiance Fields*. 2022. DOI: 10.48550/ARXIV.2207.01583. URL: https://arxiv.org/abs/2207.01583.

[9] Matthew Tancik et al. *Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains*. 2020. DOI: 10.48550/ARXIV.2006.10739. URL: https://arxiv.org/abs/2006.10739.

[10] Ashish Vaswani et al. “Attention Is All You Need”. In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[11] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[12] Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *CoRR* abs/2010.11929 (2020). arXiv: 2010.11929. URL: https://arxiv.org/abs/2010.11929.

[13] David Balduzzi et al. “The Shattered Gradients Problem: If resnets are the answer, then what is the question?” In: (2017). DOI: 10.48550/ARXIV.1702.08591. URL: https://arxiv.org/abs/1702.08591.

A Appendix

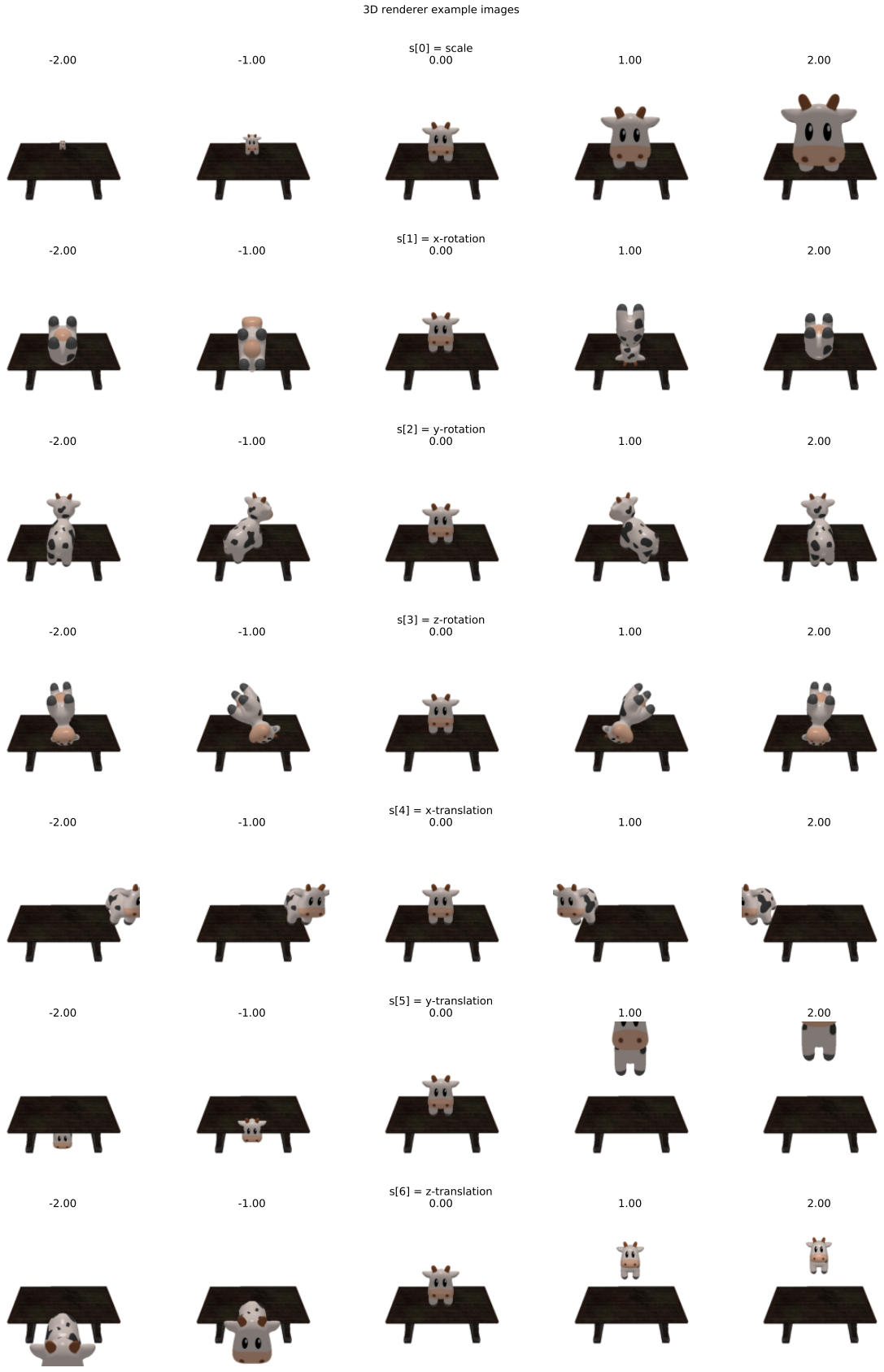

Figure A.1: Example images from the 3D renderer

Technical
University of
Denmark

Building 324
2800 Kgs. Lyngby
Tlf. 4525 1700

www.compute.dtu.dk