Final Project

Enhanced SimpleNeRF: Improved Performance and Faster Training/Inference

Yizhak Sepiashvilli
207953134
yizhaks@mail.tau.ac.il

Nitzan Monfred
316056126
nitzanm@mail.tau.ac.il

Abstract

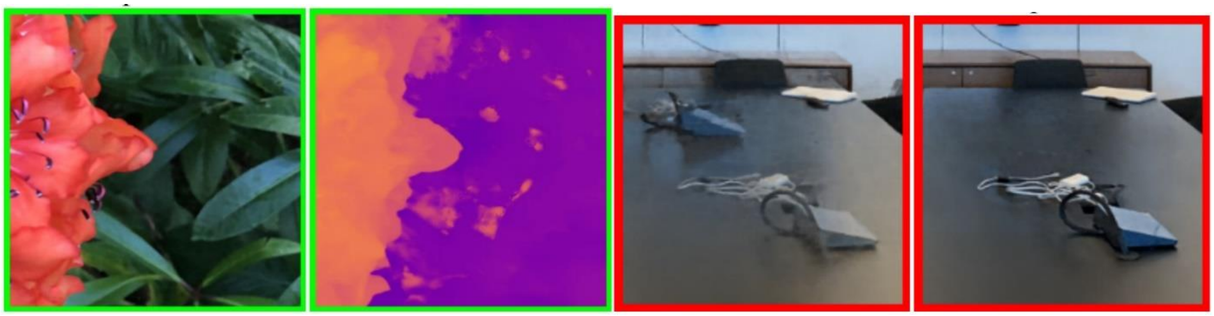


Fig1: “Floaters”(left) in an underconstrained(few-views 3D reconstruction) common few-shot NeRF models at the time introduced unnecessary depth discontinuities because of the ability to fit to the sparse training data. (unnecessary high-frequency space components)
“shape-radiance ambiguity” (right) enforcing the NeRF model to predict view-dependant radiance is counterfeit with sparse input views , the model will not easily distinguish between a dark surface or a poorly lit bright one, and will come up with worse results than what can be achieved for specular objects.

3D reconstruction from few-input views has been a hot research topic since the publication of the original NeRF paper [Mildenhall et al. 2020]. The reason for this is despite the groundbreaking performance of the novelty introduced, there was need of some-times unrealistic amount of views of a given scene and when provided the necessary amount , training for one scene could take hours on end, with top computing power. Plenty of algorithms were developed since that time to combat these issues, PixelNeRF [Yu et al. 2021] was most notable early on with their solution to sparse view rendering solution, which provided much greater results than the original NeRF model for few-view scenarios. However, results were later further improved and so did training time decrease. SimpleNeRF was published in 2023 [Somraj and Soundararajan et al.]
and showed tremendous visual results, as well as high scores on common metrics that are popular for image generation and comparison, for popular datasets LLFF [Mildenhall et al. 2019], RealEstate10K [Zhou et al. 2018]. For the cases of 3 and 4 input views in particular , which is a common real-life scene view data availability. However , it suffered greatly from long tedious training times, mostly through the addition of 2 simpler models and a large computationally heavy depth-loss computation. Training this model for a single RealEstate10K scene with 4 input views on a RTX 5000 machine took us over 2 days for 100000 iterations.
In 2025, there are accesseble options to improve the training time of NeRF models, those are nerfstudio software and instant-NGP improved encoding and algorithm for faster training times and smaller MLPs. Which we’ll make use of.

Introduction

Novel view synthesis and 3D scene reconstruction can help upgrade a lot of real-life scenarios. NeRFs allow photo-realistic scene reconstruction from just a few images, making it easier to render immersive environments in VR/AR without needing complex 3D modeling. For example, if you move your head around, NeRFs would be able to render new views quickly, thanks to quick inference, and if it could be trained on a sparse amount of views and very quickly. Additionaly, novel view synthesis can make a difference in the world of cinematography, rendering new scene views of a scene without having the camera move to that viewing direction, if possibly there is no quick option of getting the camera there. It can also create a "video scene" off of input images from walk through a scene, and turn them to a video. Maybe even in real-time if training efficiency and hardware permit.

All of the above uses rely heavily on the ability to train the model on the given scene in real-time and also for most of them, with few-input views. As stated above, we would like to cut the training time of the SimpleNeRF model significantly.
We will attempt to leverage available advances in the field of NeRF to do so, in the form of nerfstudio software and instant-NGP improved faster model.

The reason for taking up the task to improve SimpleNeRF's training time is that we find the results (Fig2a. and Fig2b.) published by the authors incredible and the approach very sensible. We want to modernize the model in a way that would make it relevant in the current era of 3D reconstruction and novel view synthesis. Additionally, we introduce FreeNeRF's gradually increasing positional encoding frequency band, for enhanced visual performance. We elaborate on this later on in the paper.

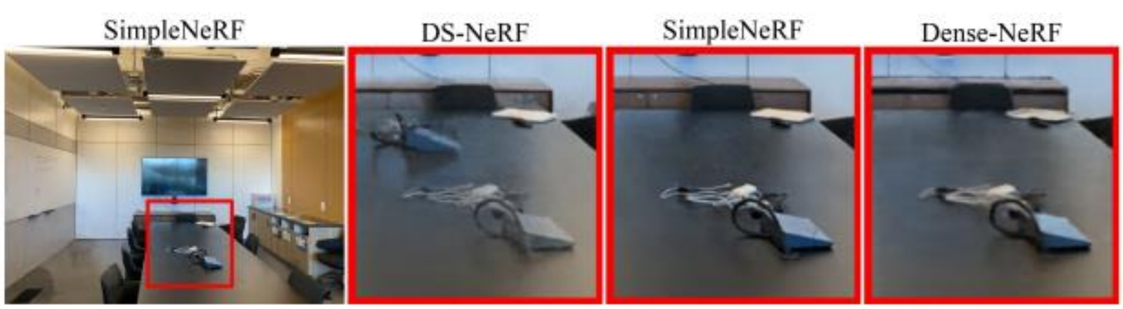


Table 10: Quantitative Results on RealEstate-10K dataset with three input views. The values within parenthesis show unmasked scores.

Model	LPIPS	SSIM	PSNR	Depth MAE	Depth SROCC
InfoNeRF	0.6561(0.6846)	0.3792(0.3780)	10.57(10.57)	2.2198(2.2830)	0.1929(0.1994)
DietNeRF	0.4636(0.4886)	0.6456(0.6445)	18.01(17.89)	2.0355(2.1023)	0.0240(0.0438)
RegNeRF	0.4171(0.4362)	0.6132(0.6078)	17.86(17.73)	–	0.0574(0.0475)
DS-NeRF	0.2893(0.3211)	0.8004(0.7905)	26.50(25.94)	0.5400(0.6524)	0.8106(0.7910)
DDP-NeRF	0.1518(0.1601)	0.8587(0.8518)	26.67(25.92)	0.4139(0.5222)	0.8612(0.8331)
FreeNeRF	0.5146(0.5414)	0.5708(0.5675)	15.26(15.12)	–	-0.2590(-0.2445)
ViP-NeRF	0.0758(0.0832)	0.8967(0.8852)	31.93(30.27)	0.3365(0.4683)	0.9009(0.8558)
SimpleNeRF	**0.0726(0.0829)**	**0.8984(0.8879)**	**33.21(31.40)**	**0.2770(0.3885)**	**0.9266(0.8931)**

Table 11: Quantitative Results on RealEstate-10K dataset with four input views. The values within parenthesis show unmasked scores.

Model	LPIPS	SSIM	PSNR	Depth MAE	Depth SROCC
InfoNeRF	0.6651(0.6721)	0.3843(0.3830)	10.62(10.59)	2.1874(2.2742)	0.2549(0.2594)
DietNeRF	0.4853(0.4954)	0.6503(0.6475)	18.01(17.89)	2.0398(2.1273)	0.0990(0.1011)
RegNeRF	0.4316(0.4383)	0.6257(0.6198)	18.34(18.25)	–	0.1422(0.1396)
DS-NeRF	0.3103(0.3287)	0.7999(0.7920)	26.65(26.28)	0.5154(0.6171)	0.8145(0.8018)
DDP-NeRF	0.1563(0.1584)	0.8617(0.8557)	27.07(26.48)	0.3832(0.4813)	0.8739(0.8605)
FreeNeRF	0.5226(0.5323)	0.6027(0.5989)	16.31(16.25)	–	-0.2152(-0.2162)
ViP-NeRF	0.0892(0.0909)	0.8968(0.8894)	31.95(30.83)	0.3658(0.4761)	0.8414(0.8080)
SimpleNeRF	**0.0847(0.0891)**	**0.8987(0.8917)**	**32.88(31.73)**	**0.2692(0.3565)**	**0.9209(0.9035)**

Fig2a.

Table 7: Quantitative Results on LLFF dataset with three input views. The values within parenthesis show unmasked scores.

Model	LPIPS	SSIM	PSNR	Depth MAE	Depth SROCC
InfoNeRF	0.6732(0.7679)	0.1953(0.1859)	8.38(8.52)	1.0012(1.1149)	-0.0144(-0.0176)
DietNeRF	0.6120(0.7254)	0.3405(0.3297)	11.76(11.77)	0.9093(1.0242)	-0.0598(-0.0471)
RegNeRF	0.2908(0.3602)	0.6334(0.5677)	20.22(18.65)	–	0.8238(0.7589)
DS-NeRF	0.3031(0.3641)	0.6321(0.5774)	20.20(18.97)	0.1787(0.2699)	0.7852(0.7173)
DDP-NeRF	0.3250(0.3869)	0.6152(0.5628)	18.73(17.71)	0.1941(0.3032)	0.7433(0.6707)
FreeNeRF	0.2754(0.3415)	0.6583(0.5960)	20.93(19.30)	–	0.8379(0.7656)
ViP-NeRF	0.2798(0.3365)	0.6548(0.5907)	20.54(18.89)	0.1721(0.2795)	0.7891(0.7082)
SimpleNeRF	**0.2559(0.3259)**	**0.6940(0.6222)**	**21.37(19.47)**	**0.1199(0.2201)**	**0.8935(0.8153)**

Table 8: Quantitative Results on LLFF dataset with four input views. The values within parenthesis show unmasked scores.

Model	LPIPS	SSIM	PSNR	Depth MAE	Depth SROCC
InfoNeRF	0.6985(0.7701)	0.2270(0.2188)	9.18(9.25)	1.0411(1.1119)	-0.0394(-0.0390)
DietNeRF	0.6506(0.7396)	0.3496(0.3404)	11.86(11.84)	0.9546(1.0259)	-0.0368(-0.0249)
RegNeRF	0.2794(0.3227)	0.6645(0.6159)	21.32(19.89)	–	0.8933(0.8528)
DS-NeRF	0.2979(0.3376)	0.6582(0.6135)	21.23(20.07)	0.1451(0.2097)	0.8506(0.8130)
DDP-NeRF	0.3042(0.3467)	0.6558(0.6121)	20.17(19.19)	0.1704(0.2487)	0.8322(0.7664)
FreeNeRF	0.2848(0.3280)	0.6764(0.6303)	21.91(**20.45**)	–	0.9091(0.8626)
ViP-NeRF	0.2854(0.3203)	0.6675(0.6182)	20.75(19.34)	0.1555(0.2316)	0.8622(0.8070)
SimpleNeRF	**0.2633(0.3083)**	**0.7016(0.6521)**	**21.99**(20.44)	**0.1110(0.1741)**	**0.9355(0.8952)**

Fig2b.

SimpleNeRF's model beats popular few-shot NeRF options in early-mid 2023 on RealEstate-10K and LLFF scenes with 3 or 4 input views. Especially so in depth estimation, which was a particular struggle in Few-shot NeRF options at the time.

Related Work

NeRFs have come a long way throughout the last few years. From the slow vanilla model [Mildenhall et al. 2020] we were introduced to many faster and improved variations. The vanilla model's main drawbacks were slow training time, slow rendering time, large dataset requirement. FastNeRF[Muller et al. 2021] uses two-stage networks to decouple density and geometry rendering from color and view-dependent estimations, in tandem with separation of view variant or invariant surfaces to make the model more efficient at rendering view invariant surfaces. This approach is used in SimpleNeRF to supervise the depth estimation in view invariant areas of a 3D scene , it comes in the form of the view invariant augmented model. With these, FastNeRF achieves multiple times faster rendering than the original NeRF model.

DS-NeRF [Depth-supervised NeRF: Deng et al. 2021] uses depth information to help train the model , since depth information is basically with, with software applications like COLMAP, given a set of images, SFM can generate sparse pseudo-gt depths for features of an image, and using a depth mask, you can train the model's depth estimation of a scene using the sparse set of points with known depths. They also introduce a depth loss function which we use in our implementation of SimpleNeRF as a nerfstudio model. By using this "basically free" information of sparse depth points, the NeRF model is able to generalize much better to fewer views, providing much better results for few-shot scenes, in addition to improving overall quality of rendering when larger datasets of a scene are available.

Instant-NGP [Nvidia. 2022] was a large step forward for NeRF research and applcations, as it could train in **seconds** while producing high quality renderings. Their novelty was Instead of using a dense voxel grid or an MLP input, they encode 3D positions into a hash table of features at multiple resolutions — which turns out to be extremely fast and memory-efficient. This allowed them to use a smaller MLP and take advantage of other NeRF optimization frame-works that popped up at the time. Instant-NGP single-handily launched the research subject of NeRFs into real-time capability. We propose an enhanced version of our improved model using instant-NGP frameworks.

FreeNeRF [Yang et al. 2023] proposes increasing the positional encoding degree of the MLP inputs as training progresses, such that the model learns smooth surfaces early on, and high frequency discontinuities later in training, which allowed for training for a scene with only a few input images. We make use of this in our work.

Data

Since we implement our proposed improvements with nerfstudio, we use nerfstudio's data-processing utilities, for our specific model , the preprocessing steps are similar to most other nerfstudio models and go as follows:

- ns-process-data recieves a path to a directory of any set of captured images of a scene we want to render new-views of.
- The images are rescaled to the same dimensions, in case the were not all captured by the same camera
- Retrieve camera poses and dense depth maps with a SFM/MVS software package.

In order to be able to generate the rays in the right direction and from the right starting point, relative camera poses for each image are required after preprocessing. nerfstudio generates camera poses from the directory of images using classic SFM algorithm (MVS, we're given more than 2 input images), which comes with a package called COLMAP. With COLMAP, we receive not only camera translation and rotation for each of the views, but also sparse depth maps which will come in handy when we train our models with DS-NeRF depth supervision. (Which will serve as gt-labels for depth estimation)
this is the process we use COLMAP for:

- feature detection in each image.
- matching the common features in images.
- pose estimation, camera intrinsics estimation and 3D point cloud construction(Epipolar constraints, fundemental matrix with RANSAC) + bundle adjustment.
- depth estimation for matched features.

Following this data preprocessing we have all the necessary tools to train our model using ground truth depth labels and rgb pixel value labels.

Preliminaries:

Rendering

To render a pixel **p**, we:

- Shoot a ray from the camera origin $\boldsymbol{o}$ in a certain direction $\boldsymbol{d}$ through the scene
- Sample points along that ray between a near and far depth (t: t_{near} to t_{far}).
- Estimate the final rgb pixel value by discretely integrating color and density of each of those sample points(σ, c)

Each sampled point along the ray enters the MLP along with its positional encodings, which outputs estimated color and density value of the the 3D sampled point:

$f_\theta(\mathbf{x}, \mathbf{d}) = (\sigma, \mathbf{c})$

where **x** = **o** + t * **d**

Using all the sampled points along a ray we calculate the predicted rgb value of the pixel **p** as follows:

$$\hat{\mathbf{c}}(\mathbf{r}; \theta, \mathbf{t}_K) = \sum_K T_k(1 - \exp(-\sigma_k(t_{k+1} - t_k)))\mathbf{c_k}$$

$$\text{with} \quad T_k = \exp\left(-\sum_{k'<k} \sigma'_k \left(t_{k'+1} - t_{k'}\right)\right)$$

θ being the MLP's parameters

Positional Encoding

MLPs tend to try to learn low-frequency components of given data, to combat this, and introduce rapidly changing colored surfaces and depth discontinuities, we use positional encoding, which maps the input 3D position and viewing direction to higher dimensions:

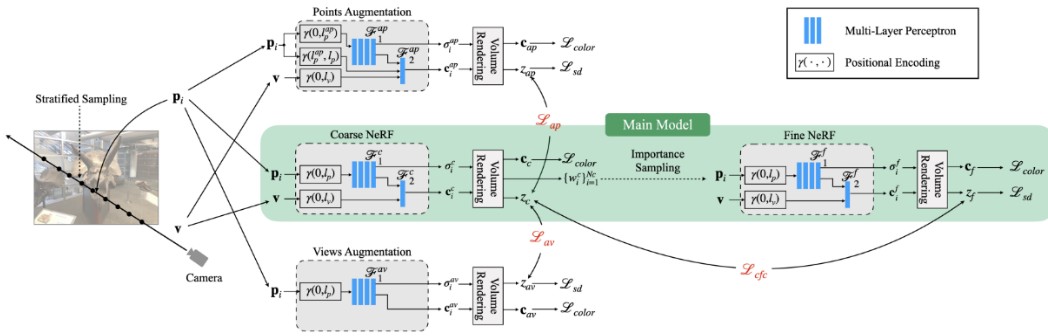
$$\gamma_L(\mathbf{x}) = \left[\sin(\mathbf{x}), \cos(\mathbf{x}), ..., \sin(2^{L-1}\mathbf{x}), \cos(2^{L-1}\mathbf{x})\right]$$

in our implementation, L will increase as the training progress, in accordance with FreeNeRF's proposed training methodology.

eventually the input to the MLP is the 3D point along with its encodings:

$$\mathbf{x}' = [\mathbf{x}, \gamma_L(\mathbf{x})]$$

Methods



Our first method has to do with tackling the long training time issue, and it is to introduce the SimpleNeRF model to nerfstudio as a selectable NeRF model.
nerfstudio has several things going for it , such that just by implementing the model there will show significant speed-up in training and inference:

- nerfstudio utilizes parallel data-loaders to reduce data transfer overhead during training.
- It uses strategic ray samplers to improve performance of rendering
- Smarter ray batching method, replacing classic pre-image batching to reduce data overhead
. and increase GPU uptime (Less CPU bottleneck)

To further decrease training and inference time we considered using a depth-supervised Instant-NGP model to replace the course-fine models of in the SimpleNeRF layout, since Instant-NGP implementation is readily available in nerfstudio source code, we could easily swap out the vanilla course & fine networks and replace them with an Instant-NGP model instead, without much coding. Additionally, one can modify the existing Instant-NGP nerfstudio implementation and make it so we could replace our vanilla view-independent network with a view-independent depth-supervised Instant-NGP network for faster more accurate supervision. Turning the model view-independent in the original NeRF network is as simple as inputing only the 3D position along with its positional encodings and omitting the viewing-directions with their positional encoding. For Instant-NGP, we would have to rid of the auxilary input vector $\mu \in R^E$ and input the small MLP only the interpolated feature vectors.

However, we couldn't find a method to replace the lower-degree positional encoding augmented model and therefore we decided to leave it up to future research for now on this instant-NGP route, and instead we implemented Classic SimpleNeRF with depth-loss supervision changes and FreeNeRF encoding, using the original NeRF models as the networks.

In an attempt to improve upon the original SimpleNeRF model proposed, we introduce FreeNeRF positional encoding, which increases the degree and the number of frequencies used for positional encoding as the model undergoes further and further training. The main idea is to help each of the networks learn the 3D scene representation of the smooth areas first, and introduce the higher-frequency surfaces, along with frequent depth discontinuities later on in training so its getting targeted more effieciently.
At each step in the training loop, the 3D positions entered into each network undergoes this transformation + concatenation and for all networks except the view-invariant augmented one we do this also for the viewing directions:

$$\gamma'_L(t, T; \mathbf{x}) = \gamma_L(\mathbf{x}) \odot \boldsymbol{\alpha}(t, T, L),$$

$$\text{with } \boldsymbol{\alpha}_i(t, T, L) = \begin{cases} 1 & \text{if } i \leq \frac{t \cdot L}{T} + 3 \\ \dfrac{t \cdot L}{T} - \lfloor \dfrac{t \cdot L}{T} \rfloor & \text{if } \frac{t \cdot L}{T} + 3 < i \leq \frac{t \cdot L}{T} + 6 \\ 0 & \text{if } i > \frac{t \cdot L}{T} + 6 \end{cases}$$

Where t is the current step/iteration in training and T is the number of total iterations of training. L is an hyperparameter as mentioned in the preliminaries.

We switched up the depth-loss used in the SimpleNeRF paper, with that used in DS-NeRF paper[Deng et al. 2021]
The depth-loss in the SimpleNeRF paper is calculated by reprojecting a patch of K by K pixels around the pixel q with a gt-depth label, the reprojection is using the estimated depth values outputed by the different networks (course/fine/views-invariant/lower-degree-pos).
and than calculating the MSE loss with the K neighborhood of the pixel from the training image.
We saw that DS-NeRF loss that was proposed can work well and decided to switch, as it requires less computation while offering good results.
DS-NeRF loss function is such:

$$\mathcal{L}_{Depth} \approx \mathbb{E}_{x_i \in X_j}\left[-\sum_k \log h_k \exp\left(-\frac{(t_k - \mathbf{D}_{ij})^2}{2\hat{\sigma}_i^2}\right)\Delta t_k\right]$$

and is already implemented in nerfstudio's losses.py

The final Loss function is

$$\mathcal{L} = \lambda_1 \mathcal{L}_{\text{color}} + \lambda_2 \mathcal{L}_{\text{sd}} + \lambda_3 \mathcal{L}_{\text{ap}} + \lambda_4 \mathcal{L}_{\text{av}} + \lambda_5 \mathcal{L}_{\text{cfc}}$$

where 3 4 and 5 are MSE losses between predicted depth values between the different networks (course/view/pos/fine) and:

$$\mathcal{L}_{\text{color}} = \|\mathbf{c}_c - \hat{\mathbf{c}}\|^2 + \|\mathbf{c}_f - \hat{\mathbf{c}}\|^2 + \|\mathbf{c}_{\text{ap}} - \hat{\mathbf{c}}\|^2 + \|\mathbf{c}_{\text{av}} - \hat{\mathbf{c}}\|^2$$

$$\mathcal{L}_{\text{sd}} = \|z_f - \hat{z}\|^2 + \|z_{\text{ap}} - \hat{z}\|^2 + \|z_{\text{av}} - \hat{z}\|^2$$

where z is a pixel's depth-value, **ap** is for the positional-encoding network and **av** is for the view-invariant network

Conclusions

Since we've found out about nerfstudio in the later stage of our research for this work, we realized how necessary this software package is. After attempting to make sense of the code provided by the authors of SimpleNeRF (Our initial direction was to improve upon it) we experienced first hand how disorganized and unclear collecting the right data, at the right format and putting it at the right directories as well as having to understand how the data-loading for depth and features data we generated with colmap is gonna be handled in their code. There was little to no guidance and plenty of misinformation about these procedures (As well as provided datasets which are now outdated and no longer available in their respective links). We also couldn't find where to place the camera intrinsics and extrinsic files outputted by COLMAP and organized by us. To this end, we found nerfstudio where all of this is handled with a lot of detail and is constatly up-to-date. We could easily just inset our folder of images and nerfstudio handled all of the data preprocessing, which is why we decided to switch directions and implement the enhanced model in nerfstudio.

Although we added a lot of functionality and implemented the model in its entirety as a trainble nerfstudio model (Coding is in the Appendix). We weren't able to fully integrate the model and make it trainable as we still have more understading and debugging to do for the code, things such as input and output dimensions and values for using existing nerfstudio python functions and utilities were not always trivial and took a lot of testing to understand and we still have uncovered grounds, however, our additions and implementations provide solid starting ground for future work.

Appendix

We will show our code implementation of our enhanced SimpleNeRF inside of nerfstudio's python project(open-source, github)

we add a new method in method_configs.py:

```
53 from nerfstudio.models.simple_nerf import SimpleNeRFModelConfig
```

```
method_configs: Dict[str, Union[TrainerConfig,
ExternalMethodDummyTrainerConfig]] = {}
descriptions = {
…..
"simple-nerf": "Simple NeRF model with augmented networks depth-
supervision (& DS-NeRF depth-supervision). (slow)",
…..
```

```
}
method_configs["simple-nerf"] = TrainerConfig(
    method_name="simple-nerf",
    steps_per_eval_batch=500,
    steps_per_save=2000,
    max_num_iterations=30000,
    mixed_precision=True,
    pipeline=VanillaPipelineConfig(
        datamanager=VanillaDataManagerConfig(
            _target=ParallelDataManager[DepthDataset],
            dataparser=NerfstudioDataParserConfig(),
            train_num_rays_per_batch=4096,
            eval_num_rays_per_batch=4096,
        ),
        model=DepthNerfactoModelConfig(
            eval_num_rays_per_chunk=1 << 15,
            camera_optimizer=CameraOptimizerConfig(mode="SO3xR3"),
        ),
    ),
    optimizers={
        "proposal_networks": {
            "optimizer": AdamOptimizerConfig(lr=1e-2, eps=1e-15),
            "scheduler": None,
        },
        "fields": {
            "optimizer": AdamOptimizerConfig(lr=1e-2, eps=1e-15),
            "scheduler": None,
        },
        "camera_opt": {
            "optimizer": AdamOptimizerConfig(lr=1e-3, eps=1e-15),
            "scheduler": ExponentialDecaySchedulerConfig(lr_final=1e-4, max_steps=5000),
        },
    },
    viewer=ViewerConfig(num_rays_per_chunk=1 << 15),
    vis="viewer",
)
```

this would eventually allow us, given our dataset, to train our model on it: ns-train simple-nerf --data data/nerfstudio/images_diirectory

We implement the method similarly to how other NeRF methods are implemented in a standalone python script simple_nerf.py:

```python
"""
Implementation of simple nerf.
"""

from __future__ import annotations

from dataclasses import dataclass, field
from typing import Any, Dict, List, Literal, Tuple, Type

import torch
from torch.nn import Parameter

from nerfstudio.cameras.rays import RayBundle
from nerfstudio.configs.config_utils import to_immutable_dict
from nerfstudio.field_components.encodings import FreeNeRFEncoding
from nerfstudio.field_components.field_heads import FieldHeadNames
from nerfstudio.field_components.temporal_distortions import TemporalDistortionKind
from nerfstudio.fields.vanilla_nerf_field import NeRFField
from nerfstudio.model_components.losses import MSELoss, scale_gradients_by_distance_squared, DepthLossType, depth_loss, depth_ranking_loss
from nerfstudio.model_components.losses import course_fine_consistensy_loss, points_course_loss, view_course_loss
from nerfstudio.model_components.ray_samplers import PDFSampler, UniformSampler
from nerfstudio.model_components.renderers import AccumulationRenderer, DepthRenderer, RGBRenderer
from nerfstudio.models.base_model import Model, ModelConfig
from nerfstudio.utils import colormaps, misc
```

This up here is including the losses we defined for course-fine-consistensy, augmented-course losses and DS-NeRF loss. We also include here the Encoding-class we created for FreeNeRF positional encoding.

```python
@dataclass
class SimpleNeRFModelConfig(ModelConfig):
    """SimpleNeRF Model Config"""

    _target: Type = field(default_factory=lambda: SimpleNeRFModel)
    num_coarse_samples: int = 64
    """Number of samples in coarse field evaluation"""
    num_importance_samples: int = 128
    """Number of samples in fine field evaluation"""

    enable_temporal_distortion: bool = False
    """Specifies whether or not to include ray warping based on time."""
    temporal_distortion_params: Dict[str, Any] = to_immutable_dict({"kind": TemporalDistortionKind.DNERF})
    """Parameters to instantiate temporal distortion with"""
    use_gradient_scaling: bool = False
    """Use gradient scaler where the gradients are lower for points closer to the camera."""
    background_color: Literal["random", "last_sample", "black", "white"] = "white"
    """Whether to randomize the background color."""
```

Parameters config class inhering from the abstract parameters config class defined by nerfstudio.

```python
class SimpleNeRFModel(Model):
    """SimpleNeRF NeRF model

    Args:
        config: SimpleNeRF configuration to instantiate model
    """

    config: SimpleNeRFModelConfig

    def __init__(
        self,
        config: SimpleNeRFModelConfig,
        **kwargs,
    ) -> None:
        self.field_coarse = None
        self.field_fine = None
        self.temporal_distortion = None

        super().__init__(
            config=config,
            **kwargs,
        )

```

```python
    def populate_modules(self):
        """Set the fields and modules"""
        super().populate_modules()

        # fields
        position_encoding = FreeNeRFEncoding(
            in_dim=3, max_iters=100000,num_frequencies=10, min_freq_exp=0.0, max_freq_exp=8.0, include_input=True
        )
        augmented_position_encoding = FreeNeRFEncoding(
            in_dim=3,max_iters=100000, num_frequencies=4, min_freq_exp=0.0, max_freq_exp=4.0, include_input=True
        )
        direction_encoding = FreeNeRFEncoding(
            in_dim=3,max_iters=100000, num_frequencies=4, min_freq_exp=0.0, max_freq_exp=4.0, include_input=True
        )

        view_invariant_direction_encoding = FreeNeRFEncoding(
            in_dim=3,max_iters=100000, num_frequencies=0, min_freq_exp=0.0, max_freq_exp=0.0, include_input=False
        )

        self.field_coarse = NeRFField(
            position_encoding=position_encoding,
            direction_encoding=direction_encoding,
        )

        self.field_fine = NeRFField(
            position_encoding=position_encoding,
            direction_encoding=direction_encoding,
        )

        self.field_pos = NeRFField(
            position_encoding=augmented_position_encoding,
            direction_encoding=direction_encoding,
        )

        self.field_view = NeR
        FField(
            position_encoding=position_encoding,
            direction_encoding=view_invariant_direction_encoding,
        )
```

```python
        # samplers
        self.sampler_uniform = UniformSampler(num_samples=self.config.num_coarse_samples)
        self.sampler_pdf = PDFSampler(num_samples=self.config.num_importance_samples)

        # renderers
        self.renderer_rgb = RGBRenderer(background_color=self.config.background_color)
        self.renderer_accumulation = AccumulationRenderer()
        self.renderer_depth = DepthRenderer()
        self.renderer_median_depth = DepthRenderer(method="median")

        # losses
        self.rgb_loss = MSELoss()

        # metrics
        from torchmetrics.functional import structural_similarity_index_measure
        from torchmetrics.image import PeakSignalNoiseRatio
        from torchmetrics.image.lpip import LearnedPerceptualImagePatchSimilarity

        self.psnr = PeakSignalNoiseRatio(data_range=1.0)
        self.ssim = structural_similarity_index_measure
        self.lpips = LearnedPerceptualImagePatchSimilarity(normalize=True)

        if getattr(self.config, "enable_temporal_distortion", False):
            params = self.config.temporal_distortion_params
            kind = params.pop("kind")
            self.temporal_distortion = kind.to_temporal_distortion(params)

```

In the populate_modules function we define all the necessary components that build up the model. We declare the Encoding functions for every different network in our model, choose the ray-sampling methods from those available in nerfstudio (Uniform sampling for course/view/pos and PDFSampling for fine network)

We also define the rendering utilities, such as the rgb pixel value renderer (sum of weights*color) and the density renderer(integral over opacities of sampled rays), where the weights are just the densities*accumulated transmitance.

We provide estimated depth rendering using median-depth-rendering method, meaning the depth value will be the crossing point of half the final transmittance.

```python
    def get_param_groups(self) -> Dict[str, List[Parameter]]:
        param_groups = {}
        if self.field_coarse is None or self.field_fine is None:
            raise ValueError("populate_fields() must be called before get_param_groups")
        param_groups["fields"] = list(self.field_coarse.parameters()) + list(self.field_fine.parameters()) + list(self.field_pos.parameters()) + list(self.field_view.parameters())
        if self.temporal_distortion is not None:
            param_groups["temporal_distortion"] = list(self.temporal_distortion.parameters())
        return param_groups

    def get_outputs(self, ray_bundle: RayBundle):
        if self.field_coarse is None or self.field_fine is None:
            raise ValueError("populate_fields() must be called before get_outputs")

        # uniform sampling
        ray_samples_uniform = self.sampler_uniform(ray_bundle)
        if self.temporal_distortion is not None:
            offsets = None
            if ray_samples_uniform.times is not None:
                offsets = self.temporal_distortion(
                    ray_samples_uniform.frustums.get_positions(), ray_samples_uniform.times
                )
            ray_samples_uniform.frustums.set_offsets(offsets)
```

```python
        # coarse field:
        field_outputs_coarse = self.field_coarse.forward(ray_samples_uniform)
        if self.config.use_gradient_scaling:
            field_outputs_coarse = scale_gradients_by_distance_squared(field_outputs_coarse, ray_samples_uniform)
        weights_coarse = ray_samples_uniform.get_weights(field_outputs_coarse[FieldHeadNames.DENSITY])
        rgb_coarse = self.renderer_rgb(
            rgb=field_outputs_coarse[FieldHeadNames.RGB],
            weights=weights_coarse,
        )
        accumulation_coarse = self.renderer_accumulation(weights_coarse)
        depth_coarse = self.renderer_median_depth(weights_coarse, ray_samples_uniform)

        # positional_encoding field:
        field_outputs_pos = self.field_pos.forward(ray_samples_uniform)
        if self.config.use_gradient_scaling:
            field_outputs_coarse = scale_gradients_by_distance_squared(field_outputs_coarse, ray_samples_uniform)
        weights_pos = ray_samples_uniform.get_weights(field_outputs_pos[FieldHeadNames.DENSITY])
        rgb_pos = self.renderer_rgb(
            rgb=field_outputs_pos[FieldHeadNames.RGB],
            weights=weights_pos,
        )
        accumulation_pos = self.renderer_accumulation(weights_pos)
        depth_pos = self.renderer_median_depth(weights_pos, ray_samples_uniform)

        # view invariant field:
        field_outputs_view = self.field_view.forward(ray_samples_uniform)
        if self.config.use_gradient_scaling:
            field_outputs_view = scale_gradients_by_distance_squared(field_outputs_view, ray_samples_uniform)
        weights_view = ray_samples_uniform.get_weights(field_outputs_view[FieldHeadNames.DENSITY])
        rgb_view = self.renderer_rgb(
            rgb=field_outputs_view[FieldHeadNames.RGB],
            weights=weights_view,
        )
        accumulation_view = self.renderer_accumulation(weights_view)
        depth_view = self.renderer_median_depth(weights_view, ray_samples_uniform)


        # pdf sampling
        ray_samples_pdf = self.sampler_pdf(ray_bundle, ray_samples_uniform, weights_coarse)
        if self.temporal_distortion is not None:
            offsets = None
            if ray_samples_pdf.times is not None:
                offsets = self.temporal_distortion(ray_samples_pdf.frustums.get_positions(), ray_samples_pdf.times)
            ray_samples_pdf.frustums.set_offsets(offsets)

        # fine field:
        field_outputs_fine = self.field_fine.forward(ray_samples_pdf)
        if self.config.use_gradient_scaling:
            field_outputs_fine = scale_gradients_by_distance_squared(field_outputs_fine, ray_samples_pdf)
        weights_fine = ray_samples_pdf.get_weights(field_outputs_fine[FieldHeadNames.DENSITY])
        rgb_fine = self.renderer_rgb(
            rgb=field_outputs_fine[FieldHeadNames.RGB],
            weights=weights_fine,
        )
        accumulation_fine = self.renderer_accumulation(weights_fine)
        depth_fine = self.renderer_median_depth(weights_fine, ray_samples_pdf)

        outputs = {
            "rgb_coarse": rgb_coarse,
            "rgb_fine": rgb_fine,
            "rgb_pos": rgb_pos,
            "rgb_view": rgb_view,
            "accumulation_coarse": accumulation_coarse,
            "accumulation_fine": accumulation_fine,
            "accumulation_pos": accumulation_pos,
            "accumulation_view": accumulation_view,
            "depth_coarse": depth_coarse,
            "depth_fine": depth_fine,
            "depth_pos": depth_pos,
            "depth_view": depth_view,
            "weights_coarse": weights_coarse,
            "weights_fine": weights_fine,
            "weights_pos": weights_pos,
            "weights_view": weights_view,
            "ray_samples_uniform": ray_samples_uniform,
            "ray_samples_pdf": ray_samples_pdf,
        }
        return outputs
```

get_outputs returns estimated rgb pixel value, and depth value for loss calculations. We follow nerfstudio convention of grouping them in a Dictionary. We run inference during training/rendering using this function and use the results for optimization if gradient calculation is on.

```python
    def get_loss_dict(self, outputs, batch, metrics_dict=None) -> Dict[str, torch.Tensor]:
        # Scaling metrics by coefficients to create the losses.
        device = outputs["rgb_coarse"].device
        image = batch["image"].to(device)
        coarse_pred, coarse_image = self.renderer_rgb.blend_background_for_loss_computation(
            pred_image=outputs["rgb_coarse"],
            pred_accumulation=outputs["accumulation_coarse"],
            gt_image=image,
        )
        fine_pred, fine_image = self.renderer_rgb.blend_background_for_loss_computation(
            pred_image=outputs["rgb_fine"],
            pred_accumulation=outputs["accumulation_fine"],
            gt_image=image,
        )

        pos_pred, pos_image = self.renderer_rgb.blend_background_for_loss_computation(
            pred_image=outputs["rgb_pos"],
            pred_accumulation=outputs["accumulation_pos"],
            gt_image=image,
        )

        view_pred, view_image = self.renderer_rgb.blend_background_for_loss_computation(
            pred_image=outputs["rgb_view"],
            pred_accumulation=outputs["accumulation_view"],
            gt_image=image,
        )

        rgb_loss_coarse = self.rgb_loss(coarse_image, coarse_pred)
        rgb_loss_fine = self.rgb_loss(fine_image, fine_pred)
        rgb_loss_pos = self.rgb_loss(pos_image, pos_pred)
        rgb_loss_view = self.rgb_loss(view_image, view_pred)

```

```python
        loss_dict = {"rgb_loss_coarse": rgb_loss_coarse, "rgb_loss_fine": rgb_loss_fine, "rgb_loss_pos": rgb_loss_pos, "rgb_loss_view": rgb_loss_view

        termination_depth = batch["depth_image"].to(self.device)
        sigma = 0.01
        for i in range(len(outputs["weights_coarse"].shape[0])):
            loss_dict["depth_loss_course"] += depth_loss(
                weights=outputs["weights_course"][i],
                ray_samples=outputs["ray_samples_uniform"][i],
                termination_depth=termination_depth,
                sigma=sigma,
                depth_loss_type=DepthLossType.DS_NERF,
            ) / len(outputs["weights_coarse"].shape[0])

        for i in range(len(outputs["weights_fine"].shape[0])):
            loss_dict["depth_loss_fine"] += depth_loss(
                weights=outputs["weights_fine"][i],
                ray_samples=outputs["ray_samples_pdf"][i],
                termination_depth=termination_depth,
                sigma=sigma,
                depth_loss_type=DepthLossType.DS_NERF,
            ) / len(outputs["weights_fine"].shape[0])

        for i in range(len(outputs["weights_pos"].shape[0])):
            loss_dict["depth_loss_pos"] += depth_loss(
                weights=outputs["weights_pos"][i],
                ray_samples=outputs["ray_samples_uniform"][i],
                termination_depth=termination_depth,
                sigma=sigma,
                depth_loss_type=DepthLossType.DS_NERF,
            ) / len(outputs["weights_pos"].shape[0])

        for i in range(len(outputs["weights_view"].shape[0])):
            loss_dict["depth_loss_view"] += depth_loss(
                weights=outputs["weights_view"][i],
                ray_samples=outputs["ray_samples_uniform"][i],
                termination_depth=termination_depth,
                sigma=sigma,
                depth_loss_type=DepthLossType.DS_NERF,
            ) / len(outputs["weights_view"].shape[0])



        loss_dict["depth_loss_course"] = loss_dict["depth_loss_course"] * 1e-3
        loss_dict["depth_loss_fine"] = loss_dict["depth_loss_fine"] * 1e-3
        loss_dict["depth_loss_pos"] = loss_dict["depth_loss_pos"] * 1e-3
        loss_dict["depth_loss_view"] = loss_dict["depth_loss_view"] * 1e-3

        loss_dict["course_fine_consistency_loss"] = course_fine_consistensy_loss(loss_dict["depth_loss_fine"],loss_dict["depth_loss_course"])
        loss_dict["points_course_loss"] = points_course_loss(loss_dict["depth_loss_pos"],loss_dict["depth_loss_course"])
        loss_dict["view_course_loss"] = view_course_loss(loss_dict["depth_loss_view"],loss_dict["depth_loss_course"])
        return loss_dict
```

get_loss_dict returns all the calculated losses using the outputs outputted by get_outputs() function. Here we make use of our cfc and augmented-course networks losses we defined in losses.py

```python
    def get_image_metrics_and_images(
        self, outputs: Dict[str, torch.Tensor], batch: Dict[str, torch.Tensor]
    ) -> Tuple[Dict[str, float], Dict[str, torch.Tensor]]:
        image = batch["image"].to(outputs["rgb_coarse"].device)
        image = self.renderer_rgb.blend_background(image)
        rgb_coarse = outputs["rgb_coarse"]
        rgb_fine = outputs["rgb_fine"]
        rgb_pos = outputs["rgb_pos"]
        rgb_view = outputs["rgb_view"]

        acc_coarse = colormaps.apply_colormap(outputs["accumulation_coarse"])
        acc_fine = colormaps.apply_colormap(outputs["accumulation_fine"])
        acc_pos = colormaps.apply_colormap(outputs["accumulation_pos"])
        acc_view = colormaps.apply_colormap(outputs["accumulation_view"])

        assert self.config.collider_params is not None
        depth_coarse = colormaps.apply_depth_colormap(
            outputs["depth_coarse"],
            accumulation=outputs["accumulation_coarse"],
            near_plane=self.config.collider_params["near_plane"],
            far_plane=self.config.collider_params["far_plane"],
        )
        depth_fine = colormaps.apply_depth_colormap(
            outputs["depth_fine"],
            accumulation=outputs["accumulation_fine"],
            near_plane=self.config.collider_params["near_plane"],
            far_plane=self.config.collider_params["far_plane"],
        )

        depth_pos = colormaps.apply_depth_colormap(
            outputs["depth_pos"],
            accumulation=outputs["accumulation_pos"],
            near_plane=self.config.collider_params["near_plane"],
            far_plane=self.config.collider_params["far_plane"],
        )

        depth_view = colormaps.apply_depth_colormap(
            outputs["depth_view"],
            accumulation=outputs["accumulation_view"],
            near_plane=self.config.collider_params["near_plane"],
            far_plane=self.config.collider_params["far_plane"],
        )
```

```python
        combined_rgb = torch.cat([image, rgb_coarse, rgb_fine, rgb_pos, rgb_view], dim=1)
        combined_acc = torch.cat([acc_coarse, acc_fine, acc_pos, acc_view], dim=1)
        combined_depth = torch.cat([depth_coarse, depth_fine, depth_pos, depth_view], dim=1)

        # Switch images from [H, W, C] to [1, C, H, W] for metrics computations
        image = torch.moveaxis(image, -1, 0)[None, ...]
        rgb_coarse = torch.moveaxis(rgb_coarse, -1, 0)[None, ...]
        rgb_fine = torch.moveaxis(rgb_fine, -1, 0)[None, ...]

        coarse_psnr = self.psnr(image, rgb_coarse)
        fine_psnr = self.psnr(image, rgb_fine)
        fine_ssim = self.ssim(image, rgb_fine)
        fine_lpips = self.lpips(image, rgb_fine)
        assert isinstance(fine_ssim, torch.Tensor)

        metrics_dict = {
            "psnr": float(fine_psnr.item()),
            "coarse_psnr": float(coarse_psnr),
            "fine_psnr": float(fine_psnr),
            "fine_ssim": float(fine_ssim),
            "fine_lpips": float(fine_lpips),
        }
        images_dict = {"img": combined_rgb, "accumulation": combined_acc, "depth": combined_depth}
        return metrics_dict, images_dict
```

get_image_matrics_and_images is just a function we modified to register the depths outputted by the networks, however, we don't calculate depth metrics in our code.

Our code relies on additional changes and implementations in losses.py and encodings.py.
in losses.py we introduce the losses between the augmented models and the course one, as well as the one between the course and the fine.
in encodings.py we create the FreeNeRF encoding class as follows:

```python
class FreeNeRFEncoding(Encoding):
    """Multi-scale progressively enhanced sinusoidal encodings. Support ``integrated positional encodings`` if covariances are provided.
    Each axis is encoded with frequencies ranging from 1 to 2^(int(current_iter/max_iters * max_freq_exp)).

    Args:
        in_dim: Input dimension of tensor
        max_num_frequencies: Number of encoded frequencies per axis
        max_iters: maximum number of iterations
        min_freq_exp: Minimum frequency exponent
        max_freq_exp: Maximum frequency exponent
        include_input: Append the input coordinate to the encoding
    """

    def __init__(
        self,
        in_dim: int,
        max_iters: int,
        max_num_frequencies: int,
        min_freq_exp: float,
        max_freq_exp: float,
        include_input: bool = False,
        implementation: Literal["tcnn", "torch"] = "torch",
    ) -> None:
        super().__init__(in_dim)

        self.max_num_frequencies = max_num_frequencies
        self.min_freq = min_freq_exp
        self.max_freq = max_freq_exp
        self.include_input = include_input
```

we take the regular Positional encoding class but also require the total number of training iterations as input.

```
self.tcnn_encoding = None
if implementation == "tcnn" and not TCNN_EXISTS:
    print_tcnn_speed_warning("NeRFEncoding")
elif implementation == "tcnn":
    assert min_freq_exp == 0, "tcnn only supports min_freq_exp = 0"
    assert max_freq_exp == max_num_frequencies - 1, "tcnn only supports max_freq_exp = num_frequencies - 1"
    encoding_config = self.get_tcnn_encoding_config(max_num_frequencies=self.max_num_frequencies)
    self.tcnn_encoding = tcnn.Encoding(
        n_input_dims=in_dim,
        encoding_config=encoding_config,
    )
```

```
def get_tcnn_encoding_config(cls, max_num_frequencies , step) -> dict:
    """Get the encoding configuration for tcnn if implemented"""
    encoding_config = {"otype": "Frequency", "n_frequencies": int(step/self.max_iters*max_num_frequencies)}
    return encoding_config

def get_out_dim(self) -> int:
    if self.in_dim is None:
        raise ValueError("Input dimension has not been set")
    out_dim = self.in_dim * self.max_num_frequencies * 2
    if self.include_input:
        out_dim += self.in_dim
    return out_dim

def pytorch_fwd(
    self,
    in_tensor: Float[Tensor, "*bs input_dim"],
    covs: Optional[Float[Tensor, "*bs input_dim input_dim"]] = None,
    step: int
) -> Float[Tensor, "*bs output_dim"]:
    """Calculates NeRF encoding. If covariances are provided the encodings will be integrated as proposed
        in mip-NeRF.

    Args:
        in_tensor: For best performance, the input tensor should be between 0 and 1.
        covs: Covariances of input points.
    Returns:
        Output values will be between -1 and 1
    """
    scaled_in_tensor = 2 * torch.pi * in_tensor  # scale to [0, 2pi]
    freqs = 2 ** torch.linspace(0, int(step/self.max_iters*self.max_freq), int(step/self.max_iters*self.max_num_frequencies), device=in_tensor.device)
    scaled_inputs = scaled_in_tensor[..., None] * freqs  # [..., "input_dim", "num_scales"]
    scaled_inputs = scaled_inputs.view(*scaled_inputs.shape[:-2], -1)  # [..., "input_dim" * "num_scales"]

    if covs is None:
        encoded_inputs = torch.sin(torch.cat([scaled_inputs, scaled_inputs + torch.pi / 2.0], dim=-1))
    else:
        input_var = torch.diagonal(covs, dim1=-2, dim2=-1)[..., :, None] * freqs[None, :] ** 2
        input_var = input_var.reshape((*input_var.shape[:-2], -1))
        encoded_inputs = expected_sin(
            torch.cat([scaled_inputs, scaled_inputs + torch.pi / 2.0], dim=-1), torch.cat(2 * [input_var], dim=-1)
        )
    return encoded_inputs

 def forward(
    self, in_tensor: Float[Tensor, "*bs input_dim"], covs: Optional[Float[Tensor, "*bs input_dim input_dim"]] = None, step: int
 ) -> Float[Tensor, "*bs output_dim"]:
    if self.tcnn_encoding is not None:
        encoded_inputs = self.tcnn_encoding(in_tensor)
    else:
        encoded_inputs = self.pytorch_fwd(in_tensor, covs,step)
    if self.include_input:
        encoded_inputs = torch.cat([encoded_inputs, in_tensor], dim=-1)
    return encoded_inputs
```

We simply take as input into all the regular encoding functions the current iteration out of the total number of iterations and scale the number of frequencies and the maximum frequency linearly to it.