Signed Distance Functions

Homework

Dmytro Strus

General overview:

A Signed Distance Function (SDF) represents a 3D object by mapping each point in space to the shortest distance to the surface of the object with a sign indicating whether the point is inside or outside the object. SDFs are particularly useful in 3D modeling as they provide a continuous and differentiable representation of the object's shape.

There are several general methods for representing via SDF with pros and cons:

Explicit Conversion: directly computing SDF values for points in space by evaluating distances from the mesh surface. This method involves sampling points from a 3D space and computing their distances to the nearest point on the mesh surface. Using K-D trees for calculating nearest neighbors for e.g.

- **Advantages:** Straightforward and accurate for capturing object boundaries.
- **Disadvantages:** Computationally expensive, especially for large or complex meshes.

Feature Grid and Neural Network: learning a feature grid that encodes spatial information and using a neural network to map this feature grid to SDF values. The network learns to predict SDF values based on the grid features.

- **Advantages:** Can effectively capture complex shapes and details.
- **Disadvantages:** Requires training a network for each object, leading to high computational and memory costs.

Grid Transformations without Neural Network: Discarding the neural network and directly applying transformations to feature vectors from a grid to model the SDF. This method relies on transforming a fixed grid to approximate the SDF.

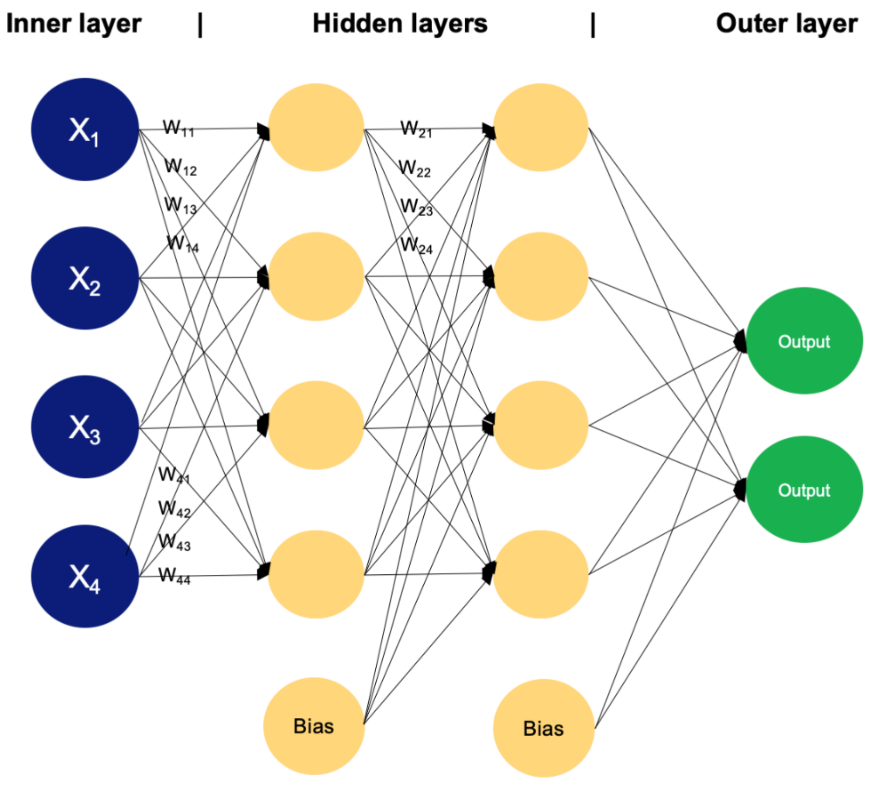
- **Advantages:** Simpler and less computationally expensive than training a full neural network.
- **Disadvantages:** May be less flexible in capturing complex geometries compared to neural network-based methods.

My approach was explicitly convert mesh dirercly from 3D object to get ground truth SDF with module pysdf and trimesh. After this for each object I've trained a separate neural network with hashgrid encoding to learn SDF and implicitly model SDF with points and true explicit SDF. This method is rational because it allows for more accurate and detailed learning of each object's unique geometric features. A single network would need to generalize across different objects potentially leading to poor performance and increased complexity. Separate networks ensure focused resource allocation, higher accuracy and easier optimization making it more effective to capture the concrete SDF of each object.

To this end, I've inspected some neural networks for this task and shortly describe them:

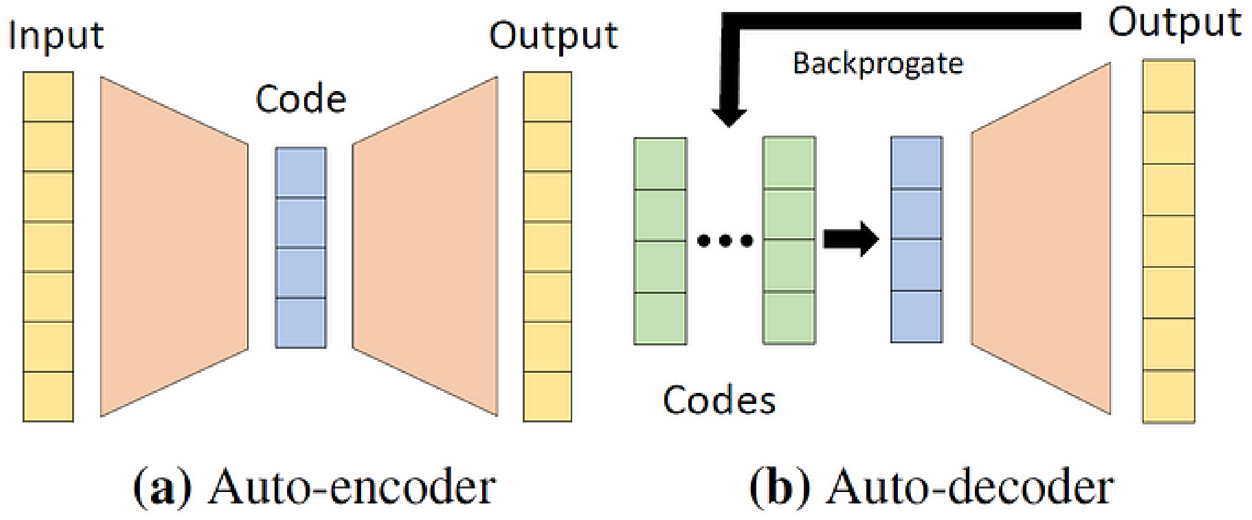
- **Multilayers Perceptrons**

Is a type of feedforward neural network consisting of fully connected neurons with a nonlinear kind of activation function. It is widely used to distinguish data that is not linearly separable. MLPs have been widely used in various fields, including image recognition, natural language processing and speech recognition, among others. Their flexibility in architecture and ability to approximate any function under certain conditions may be suitable for our SDF task. But using only this model without any enhancing was not a good idea and brought both inaccurate F1 scores (near 0.5)



- **DeepSDF**

As described authors of paper with DeepSDF it uses latent vector instead of encoding which representing the specific shape being modeled. Each shape in the dataset has its own unique latent code. In network architecture model uses a multi-layer perceptron (MLP) with fully connected layers. The network takes the concatenated latent code and 3D coordinates as input. This technic which skips default encoding with this vector makes latent space which is continuous allowing interpolation between shapes which enables the generation of new previously unseen shapes by interpolating between latent codes because latent code for each shape is learned jointly with the network weights during training. However, disadvantages are that DeepSDF might not catch complex 3D objects and requires lot of CUDA memory during training.



Implementation:

Important part of my implementation took papers and referenced materials such as Tiny Cuda NN, Torch NGP and DeepSDF article which provides some constants and architectures. According to article I've normalized meshes so it ensures that different meshes have a uniform scale and position and centered with the origin [0,0,0] and from (-1, 1). This also reduce numerical issues when data can lie outside bounded range. Then, as was described in paper, I sampled 500 000 spatial points for near the surface, adding some random normal noise with std 1e-2 for surface perturbation. Also 500 00 points were sampled in bounding volume.

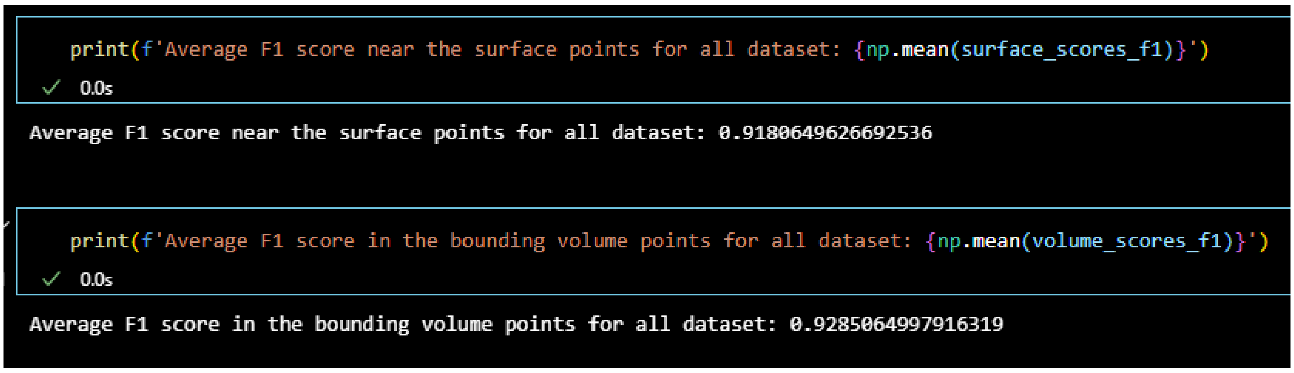
Having this points and normalized mesh, I had to define my neural network. My choice fell on the TinyCudaNN SDFModel which was described in their github (torch ngp). It consists of two main parts: encoding and backbone. The encoding process uses HashGrid which encodes points into embeddings.

This HashGrid has complex configuration so I've decided to use default config for this grid which was stated in Torch NGP. After this these embeddings processed in backbone with default fully fused multilayers perceptron and I also used default config via tinycudann library. There was also option to clip SDF but I already had normalized meshes before so there was no need to clip them because they were from -1 to 1. It was suggested to use L1Loss or MAPE loss and I've decided to use MAPE loss function. The optimizer was Adam with LR 1e-3 and some other default config which were suggested in torch ngp. The model return minimal distance according to SDF rules in train process.

With this neural network, I've started training loop. For each 3D mesh object I have initialized a new model and set 20 epochs for train. For each normalized 3D mesh I was concatenating half of surface points (250 000) and half of volume points (250 000) to make points for train. With module pysdf I was making SDF from points to train which represented ground truth.

After training every separate model, I was passing 500 000 surface points and calculating ground truth SDF for them and F1 score with passing 500 000 volume points and doing same action to get ground truth SDF. For each set of points were calculated accordingly their F1 scores: near the surface and in bounding volume.

It maybe will be questionable, why I was evaluating object for test in train loop and the answer is that in this case I'll see if all is correctly calculated and if my F1 scores are high. I've got 91.8% F1 average score near the surface points and 92.9% F1 average score in the bounding volume points. In other words, there it doesn't matter, results will be same



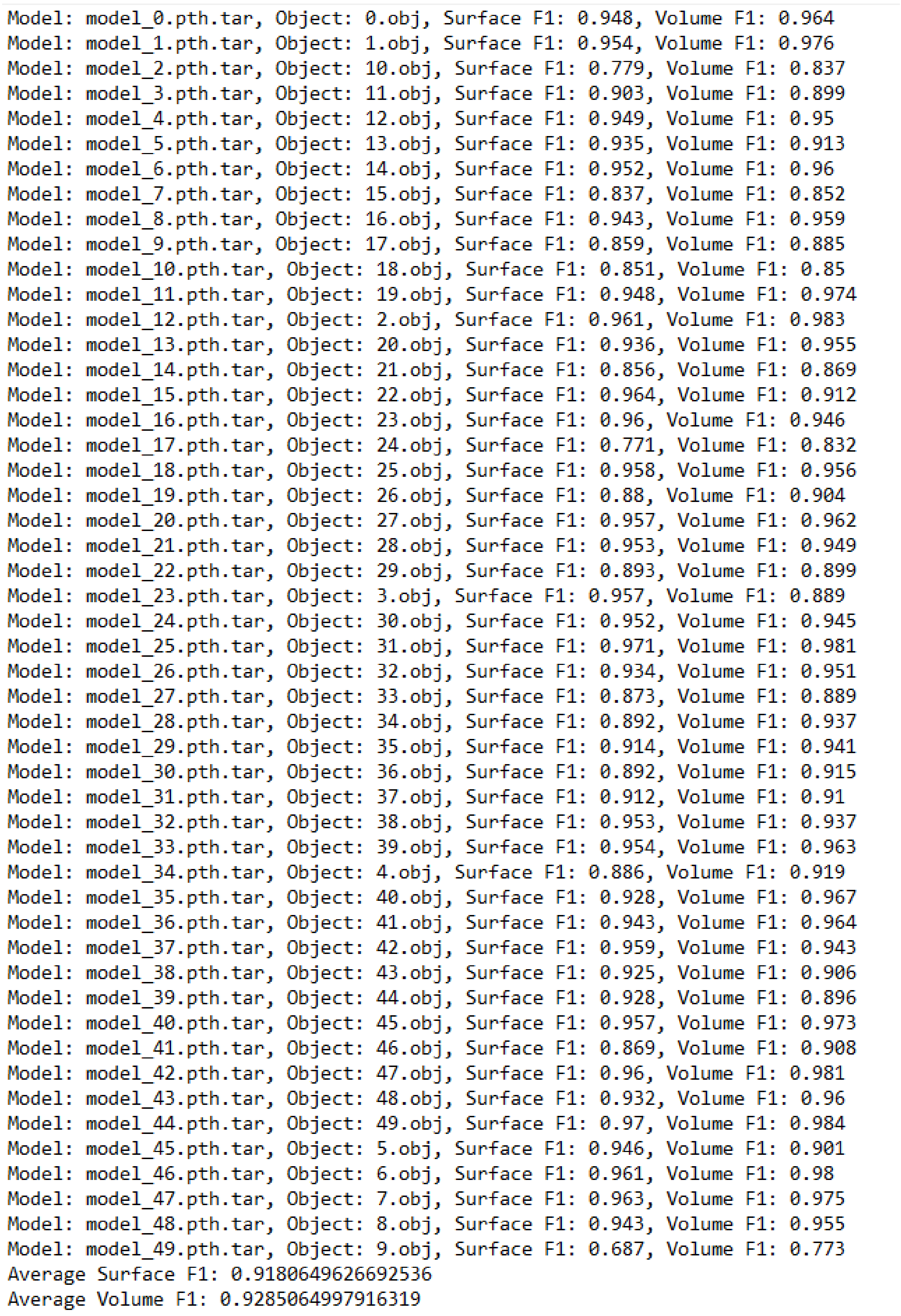
```
print(f'Average F1 score near the surface points for all dataset: {np.mean(surface_scores_f1)}')
✓ 0.0s

Average F1 score near the surface points for all dataset: 0.9180649626692536

print(f'Average F1 score in the bounding volume points for all dataset: {np.mean(volume_scores_f1)}')
✓ 0.0s

Average F1 score in the bounding volume points for all dataset: 0.9285064997916319
```

Additionally, I saved result in txt file for every separate model after they were trained

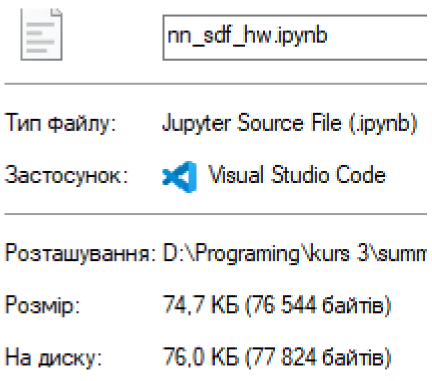


```
Model: model_0.pth.tar, Object: 0.obj, Surface F1: 0.948, Volume F1: 0.964
Model: model_1.pth.tar, Object: 1.obj, Surface F1: 0.954, Volume F1: 0.976
Model: model_2.pth.tar, Object: 10.obj, Surface F1: 0.779, Volume F1: 0.837
Model: model_3.pth.tar, Object: 11.obj, Surface F1: 0.903, Volume F1: 0.899
Model: model_4.pth.tar, Object: 12.obj, Surface F1: 0.949, Volume F1: 0.95
Model: model_5.pth.tar, Object: 13.obj, Surface F1: 0.935, Volume F1: 0.913
Model: model_6.pth.tar, Object: 14.obj, Surface F1: 0.952, Volume F1: 0.96
Model: model_7.pth.tar, Object: 15.obj, Surface F1: 0.837, Volume F1: 0.852
Model: model_8.pth.tar, Object: 16.obj, Surface F1: 0.943, Volume F1: 0.959
Model: model_9.pth.tar, Object: 17.obj, Surface F1: 0.859, Volume F1: 0.885
Model: model_10.pth.tar, Object: 18.obj, Surface F1: 0.851, Volume F1: 0.85
Model: model_11.pth.tar, Object: 19.obj, Surface F1: 0.948, Volume F1: 0.974
Model: model_12.pth.tar, Object: 2.obj, Surface F1: 0.961, Volume F1: 0.983
Model: model_13.pth.tar, Object: 20.obj, Surface F1: 0.936, Volume F1: 0.955
Model: model_14.pth.tar, Object: 21.obj, Surface F1: 0.856, Volume F1: 0.869
Model: model_15.pth.tar, Object: 22.obj, Surface F1: 0.964, Volume F1: 0.912
Model: model_16.pth.tar, Object: 23.obj, Surface F1: 0.96, Volume F1: 0.946
Model: model_17.pth.tar, Object: 24.obj, Surface F1: 0.771, Volume F1: 0.832
Model: model_18.pth.tar, Object: 25.obj, Surface F1: 0.958, Volume F1: 0.956
Model: model_19.pth.tar, Object: 26.obj, Surface F1: 0.88, Volume F1: 0.904
Model: model_20.pth.tar, Object: 27.obj, Surface F1: 0.957, Volume F1: 0.962
Model: model_21.pth.tar, Object: 28.obj, Surface F1: 0.953, Volume F1: 0.949
Model: model_22.pth.tar, Object: 29.obj, Surface F1: 0.893, Volume F1: 0.899
Model: model_23.pth.tar, Object: 3.obj, Surface F1: 0.957, Volume F1: 0.889
Model: model_24.pth.tar, Object: 30.obj, Surface F1: 0.952, Volume F1: 0.945
Model: model_25.pth.tar, Object: 31.obj, Surface F1: 0.971, Volume F1: 0.981
Model: model_26.pth.tar, Object: 32.obj, Surface F1: 0.934, Volume F1: 0.951
Model: model_27.pth.tar, Object: 33.obj, Surface F1: 0.873, Volume F1: 0.889
Model: model_28.pth.tar, Object: 34.obj, Surface F1: 0.892, Volume F1: 0.937
Model: model_29.pth.tar, Object: 35.obj, Surface F1: 0.914, Volume F1: 0.941
Model: model_30.pth.tar, Object: 36.obj, Surface F1: 0.892, Volume F1: 0.915
Model: model_31.pth.tar, Object: 37.obj, Surface F1: 0.912, Volume F1: 0.91
Model: model_32.pth.tar, Object: 38.obj, Surface F1: 0.953, Volume F1: 0.937
Model: model_33.pth.tar, Object: 39.obj, Surface F1: 0.954, Volume F1: 0.963
Model: model_34.pth.tar, Object: 4.obj, Surface F1: 0.886, Volume F1: 0.919
Model: model_35.pth.tar, Object: 40.obj, Surface F1: 0.928, Volume F1: 0.967
Model: model_36.pth.tar, Object: 41.obj, Surface F1: 0.943, Volume F1: 0.964
Model: model_37.pth.tar, Object: 42.obj, Surface F1: 0.959, Volume F1: 0.943
Model: model_38.pth.tar, Object: 43.obj, Surface F1: 0.925, Volume F1: 0.906
Model: model_39.pth.tar, Object: 44.obj, Surface F1: 0.928, Volume F1: 0.896
Model: model_40.pth.tar, Object: 45.obj, Surface F1: 0.957, Volume F1: 0.973
Model: model_41.pth.tar, Object: 46.obj, Surface F1: 0.869, Volume F1: 0.908
Model: model_42.pth.tar, Object: 47.obj, Surface F1: 0.96, Volume F1: 0.981
Model: model_43.pth.tar, Object: 48.obj, Surface F1: 0.932, Volume F1: 0.96
Model: model_44.pth.tar, Object: 49.obj, Surface F1: 0.97, Volume F1: 0.984
Model: model_45.pth.tar, Object: 5.obj, Surface F1: 0.946, Volume F1: 0.901
Model: model_46.pth.tar, Object: 6.obj, Surface F1: 0.961, Volume F1: 0.98
Model: model_47.pth.tar, Object: 7.obj, Surface F1: 0.963, Volume F1: 0.975
Model: model_48.pth.tar, Object: 8.obj, Surface F1: 0.943, Volume F1: 0.955
Model: model_49.pth.tar, Object: 9.obj, Surface F1: 0.687, Volume F1: 0.773
Average Surface F1: 0.9180649626692536
Average Volume F1: 0.9285064997916319
```

Then I evaluated my 50 separate models for time on random 500 000 points and saved it to txt file.

```
Model: model_0.pth.tar - 0.3153 milliseconds or 315300.0 nanoseconds for 500000 points
Model: model_1.pth.tar - 0.3144 milliseconds or 314400.0 nanoseconds for 500000 points
Model: model_10.pth.tar - 0.3141 milliseconds or 314100.0 nanoseconds for 500000 points
Model: model_11.pth.tar - 0.3147 milliseconds or 314700.0 nanoseconds for 500000 points
Model: model_12.pth.tar - 0.313 milliseconds or 313000.0 nanoseconds for 500000 points
Model: model_13.pth.tar - 0.3145 milliseconds or 314500.0 nanoseconds for 500000 points
Model: model_14.pth.tar - 0.3147 milliseconds or 314700.0 nanoseconds for 500000 points
Model: model_15.pth.tar - 0.3157 milliseconds or 315700.0 nanoseconds for 500000 points
Model: model_16.pth.tar - 0.3142 milliseconds or 314200.0 nanoseconds for 500000 points
Model: model_17.pth.tar - 0.3158 milliseconds or 315800.0 nanoseconds for 500000 points
Model: model_18.pth.tar - 0.3157 milliseconds or 315700.0 nanoseconds for 500000 points
Model: model_19.pth.tar - 0.3182 milliseconds or 318200.0 nanoseconds for 500000 points
Model: model_2.pth.tar - 0.3162 milliseconds or 316200.0 nanoseconds for 500000 points
Model: model_20.pth.tar - 0.3174 milliseconds or 317400.0 nanoseconds for 500000 points
Model: model_21.pth.tar - 0.316 milliseconds or 316000.0 nanoseconds for 500000 points
Model: model_22.pth.tar - 0.3157 milliseconds or 315700.0 nanoseconds for 500000 points
Model: model_23.pth.tar - 0.3148 milliseconds or 314800.0 nanoseconds for 500000 points
Model: model_24.pth.tar - 0.3157 milliseconds or 315700.0 nanoseconds for 500000 points
Model: model_25.pth.tar - 0.3134 milliseconds or 313400.0 nanoseconds for 500000 points
Model: model_26.pth.tar - 0.3167 milliseconds or 316700.0 nanoseconds for 500000 points
Model: model_27.pth.tar - 0.3148 milliseconds or 314800.0 nanoseconds for 500000 points
Model: model_28.pth.tar - 0.3153 milliseconds or 315300.0 nanoseconds for 500000 points
Model: model_29.pth.tar - 0.3152 milliseconds or 315200.0 nanoseconds for 500000 points
Model: model_3.pth.tar - 0.3145 milliseconds or 314500.0 nanoseconds for 500000 points
Model: model_30.pth.tar - 0.3168 milliseconds or 316800.0 nanoseconds for 500000 points
Model: model_31.pth.tar - 0.3162 milliseconds or 316200.0 nanoseconds for 500000 points
Model: model_32.pth.tar - 0.3137 milliseconds or 313700.0 nanoseconds for 500000 points
Model: model_33.pth.tar - 0.3143 milliseconds or 314300.0 nanoseconds for 500000 points
Model: model_34.pth.tar - 0.3145 milliseconds or 314500.0 nanoseconds for 500000 points
Model: model_35.pth.tar - 0.315 milliseconds or 315000.0 nanoseconds for 500000 points
Model: model_36.pth.tar - 0.3137 milliseconds or 313700.0 nanoseconds for 500000 points
Model: model_37.pth.tar - 0.3152 milliseconds or 315200.0 nanoseconds for 500000 points
Model: model_38.pth.tar - 0.3152 milliseconds or 315200.0 nanoseconds for 500000 points
Model: model_39.pth.tar - 0.3152 milliseconds or 315200.0 nanoseconds for 500000 points
Model: model_4.pth.tar - 0.3151 milliseconds or 315100.0 nanoseconds for 500000 points
Model: model_40.pth.tar - 0.3146 milliseconds or 314600.0 nanoseconds for 500000 points
Model: model_41.pth.tar - 0.3146 milliseconds or 314600.0 nanoseconds for 500000 points
Model: model_42.pth.tar - 0.3145 milliseconds or 314500.0 nanoseconds for 500000 points
Model: model_43.pth.tar - 0.316 milliseconds or 316000.0 nanoseconds for 500000 points
Model: model_44.pth.tar - 0.3155 milliseconds or 315500.0 nanoseconds for 500000 points
Model: model_45.pth.tar - 0.3153 milliseconds or 315300.0 nanoseconds for 500000 points
Model: model_46.pth.tar - 0.3139 milliseconds or 313900.0 nanoseconds for 500000 points
Model: model_47.pth.tar - 0.3158 milliseconds or 315800.0 nanoseconds for 500000 points
Model: model_48.pth.tar - 0.3139 milliseconds or 313900.0 nanoseconds for 500000 points
Model: model_49.pth.tar - 0.3169 milliseconds or 316900.0 nanoseconds for 500000 points
Model: model_5.pth.tar - 0.3192 milliseconds or 319200.0 nanoseconds for 500000 points
Model: model_6.pth.tar - 0.3171 milliseconds or 317100.0 nanoseconds for 500000 points
Model: model_7.pth.tar - 0.3156 milliseconds or 315600.0 nanoseconds for 500000 points
Model: model_8.pth.tar - 0.3142 milliseconds or 314200.0 nanoseconds for 500000 points
Model: model_9.pth.tar - 0.3129 milliseconds or 312900.0 nanoseconds for 500000 points
```

And memory is



nn_sdf_hw.ipynb

Тип файлу:	Jupyter Source File (.ipynb)
Застосунок:	Visual Studio Code
Розташування:	D:\Programing\kurs 3\summ
Розмір:	74,7 КБ (76 544 байтів)
На диску:	76,0 КБ (77 824 байтів)

Links:

Torch NGP

TinyCudaNN

DeepSDF

Important note

I will provide my code in jupyter notebook and colab but to run my code you have to install tinycudann from their github

My code on colab - link