Deep Learning for Computer Vision HW4

contributed by < weihsinyeh (https://github.com/DLCV-Fall-2024/dlcv-fall-2024-hw4-weihsinyeh) >
資訊所 113
姓名：葉惟欣
學號：R13922043
作業說明 (https://docs.google.com/presentation/d/1lTmnEreMgIkmNuWzPnd-2kiRJffvJC2aFfxMhvt3qcM/edit#slide=id.g18f26c43992_0_3)

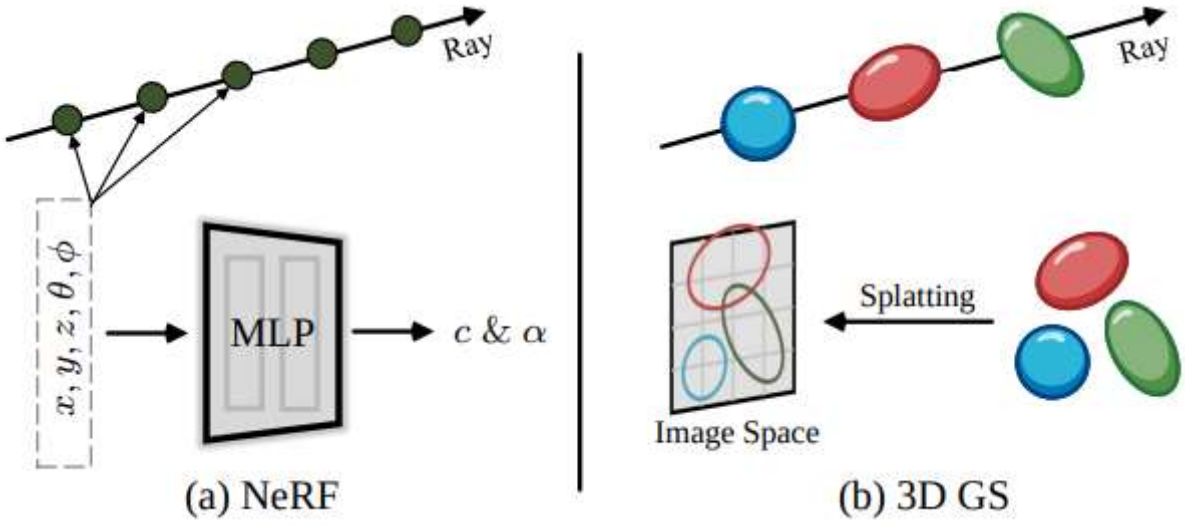
Problem: 3D Novel View Synthesis

Report

(15%) Please explain:

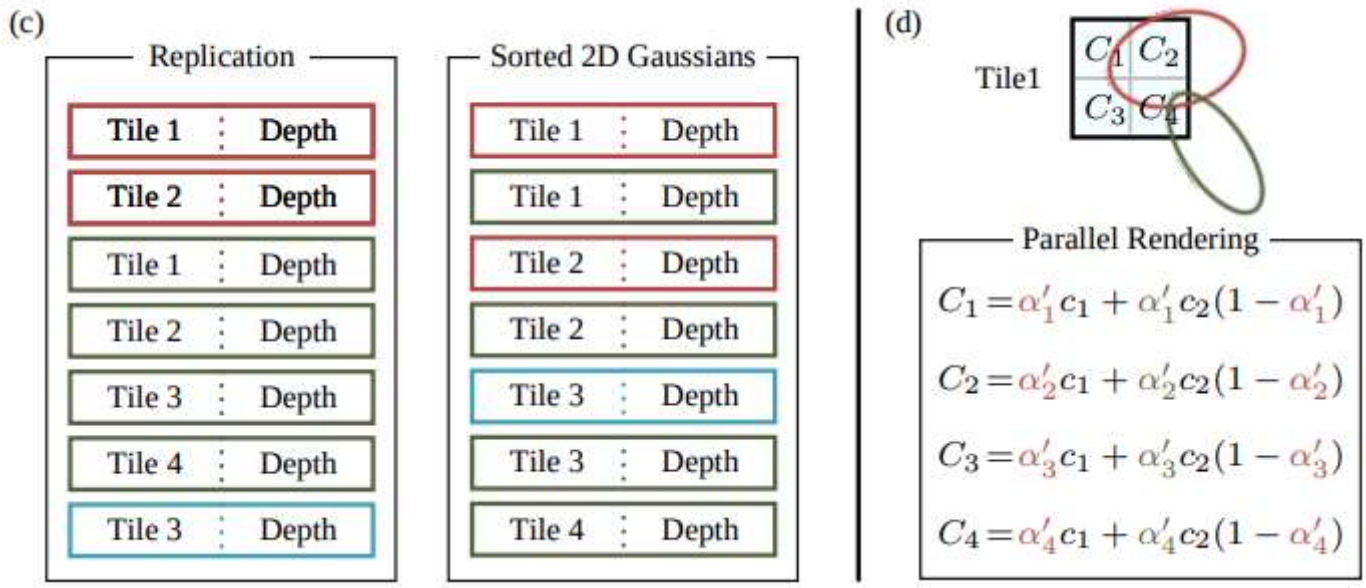
a. Try to explain 3D Gaussian Splatting in your own words

3D Gaussian Splatting 有 differentiable pipelines 還有 point-based 的算繪技術。用可學習 3D Gaussians 去表示場景。continuous volumetric radiance fields 可以表示為影像合成。3D GS 將所有物體其實都是 3D Gaussians 的表示投影(splatting)到一個 image space，執行 parallel render。



3D Gaussian 的特色是有一個 center (position) μ, opacity (不透明度) α, 3D covariance matrix Σ, and color c. c is represented by spherical harmonics for view-dependent appearance. 所有參數都是可以被學的透過 back propagation 去學習跟優化。**Frustum Culling:** 接著有 frustum culling，給定一個特定的相機位置，只計算出現在相機視錐體以內的 3D Gaussian。**Splatting:** 接著 3D Gaussian 被投影到 2D image space 去算繪。**Rendering**

by Pixels: 不同的 3D Gaussian 要疊到同個 pixel 就用 alpha compositing 。隨然要在有不同深度也就是 render 的順序下，要平行計算很困難。**Tiles (Patches)** 為了避免每個像素都需計算高斯分布，造成計算成本大，3D GS 將精度從 pixel level 提升到 patch level。具體而言，3D GS 起初將影像劃分為多個不重疊的 patch 圖塊（tiles）。由 16×16 像素構成。3D GS 接著確定哪些圖塊與投影的高斯分布相交。由於投影的高斯分布可能覆蓋多個圖塊，一種合理的方法是複製該高斯分布，並為每個複製體分配一個識別符（即圖塊 ID），以標識其相關的圖塊。



Parallel Rendering接著其去計算每個tile 會用到哪幾個 gaussian 分佈，每個高斯分佈由視角變換得到的 depth value （低位元）與對應的 tile ID （高位元）組合產生一個未排序的位元組列表。如此一來排序就會先照高位元排序，排序後就可以直接做 alpha compositing。由於 tile 的算繪是獨立進行，適合 GPU parallel computing。

b. Compare 3D Gaussian Splatting with NeRF (pros & cons)

原先的技術 NeRF 為計算量大 ray tracing 方式去捕捉場景中每個物件的像素資訊，然後繪製在最後的屏幕上面，這個方式可以捕捉光線的折射、反射、陰影可以較模擬真實的世界，這個缺點就是 ray tracing (marching) 的計算量很大 （Computational Intensity）同時還會對 empty space 計算 。另外一個缺點就是 NeRF 的 Editability 可編輯性不高，因為在場景中編輯一個場景相當於要直接修改整個 neural network 的參數權重。而 3D Gaussian Splatting 用 Raster 的技術由每個物件自己發光然後拍打在頻幕上。3D Gaussian Splatting 重新定義場景重建與算繪 (rendering) 的方式。同時由於 raster 的方式可以高度平行，因此有 efficient computation and rendering 的優點。

c. Which part of 3D Gaussian Splatting is the most important you think? Why?

3D Gaussian Splatting 最重要的是 Optimization， 因為要準確的重建場景，需要產生大量的 3D Gaussian Distribution，這樣才能在不同的 view point 都可以算繪 (render) 成功。
通過可微分的渲染技術，需要優化 3D 高斯分布的屬性以貼合給定場景的紋理。另一方面，能夠良好表示給定場景的 3D 高斯分布數量在事前是未知的。一個有前景的解決途徑是讓神經網絡自動學習 3D 高斯分布的密度。

如何優化每個 Gaussian Distribution 的屬性：

Prameter update 要優化每個 3D 高斯分布的屬性，特別是其 covarian matrix Σ，其控制了高斯分布的形狀和方向。然而，covarian matrix Σ 必須為 positive-definite matrix ，用 stocatstic gradient descent 直接更新參數可能產生的矩陣不是 positive-definite matrix ，而這種矩陣無法用來解釋 Gaussian Distribution 的數學性質。因此使用旋轉和縮放來間接表示 covarian matrix Σ ，用 quaternion q 表示旋轉矩陣（**R**）。 3D vector 表示縮放矩陣 （**S**）。用 **R** 與 **S** 去表示 covarian matrix 。

$$\Sigma = RSS^\top R^\top,$$

這種方法使得 covarian matrix Σ 每次被參數更新都是符合 Gaussian Distribution 的數學性質。

如何控制 Gaussian Distribution 的密度：

初始：

3D Gaussian Splatting 從一個由 random initialization 或是 SFM 的 sparse points 來的。好的 initialization 對 covergence 和場景重建品質影響很大。

過程：

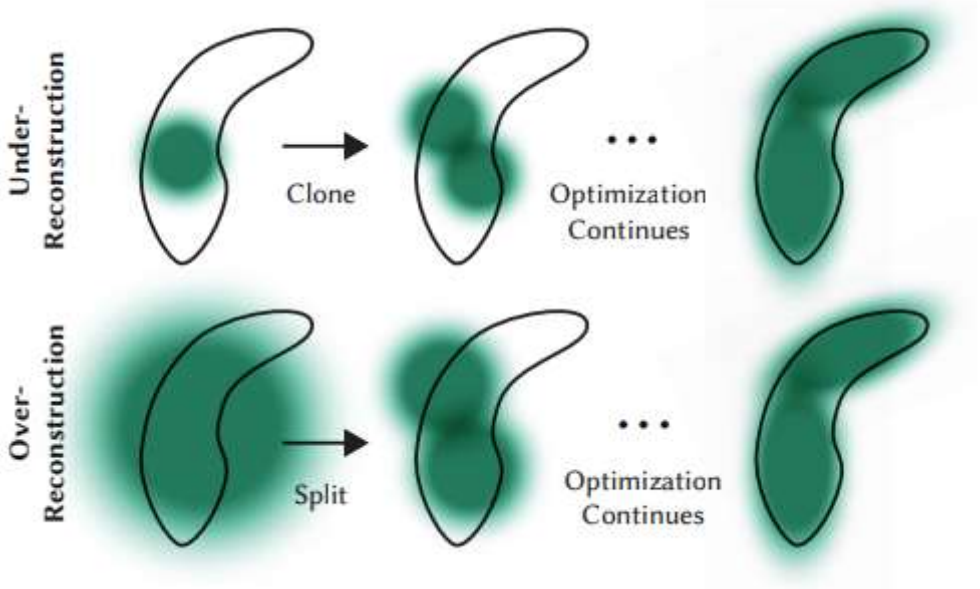
透過 point densification 和 pruning 去控制 3D Gaussians 的密度。

point densification

這個方法是來找到 Gaussian 在場景最佳的分佈方式，讓重建的效果更好。

主要處理缺失 geometric features 的區域或是 Gaussians 很稀疏的區域。 Densification procedure 會定期增加密度。

densification procedure 有 Cloning 與 Splitting 兩種方法，其都是針對有 large view-space positional gradients 的 Gaussians 處理



對於 under-reconstructed 的區域，會用 Cloning 小 Gaussian 且朝向 positional gradients 。對於 over-reconstructed 的區域，會用 Splitting 將大 Gaussian 替換為兩個小 Gaussian。

point pruning

Regularization process : 刪除多餘或影響較小的高斯分布。包括刪除幾乎透明 (α 值低於指定閾值) 的 Gaussian 或在 world space 或 view space 過大的 Gaussian。還會在每幾個迭代後將靠近相機的 Gaussian 的 alpha value 靠近 0 ，讓他們變幾乎透明，以防止 Gaussians 的密度在相機附近不斷增加。如此一來，Gaussians 的密度是受控地增加，同時還能剔除多餘的 Gaussians 來節省計算資源。

增加密度的方法 point densification 與降低密度的方法 point pruning 交替進行，讓 gaussain splatting 的 Optimization 是可運作的。

(15%) Describe the implementation details of your 3D Gaussian Spaltting for the given dataset. You need to explain your ideas completely.

position_lr_final

我將位置的學習率調低 position_lr_final=0.0000008 讓他可以做更細緻的優化。

曝光參數

沒有使用曝光參數調整 rendered_image 的色彩和亮度，曝光參數是用於模擬攝影機的曝光效果、渲染的後期處理，或校正影像的顯示效果。

```
# Apply exposure to rendered image (training only)
    if use_trained_exp:
        exposure = pc.get_exposure_from_name(viewpoint_camera.image_name)
        rendered_image = torch.matmul(rendered_image.permute(1, 2, 0),
                                      exposure[:3, :3]).permute(2, 0, 1)
                                    + exposure[:3, 3,   None, None]
```

因為我發現使用曝光參數，我的整張圖會變得亮度過暗，所以我就都沒有使用了。

這是分數較高的照片

使用曝光參數後的照片

lambda_dssim

在 `train.py`

我調整到 0.35

```
loss = (1.0 - opt.lambda_dssim) * Ll1 + opt.lambda_dssim * (1.0 - ssim_value)
```

`(1.0 - opt.lambda_dssim)` ：控制 L1 loss 的影響力。

`opt.lambda_dssim` ：控制 SSIM 損失的影響力。

我增加了 SSIM 損失 的權重。

percent_dense

調整 percent dense ，這個參數會被用在 `densify_and_split()` `densify_and_clone()` 與 `densify_and_prune()` 用在 Gaussian Model 優化的時候。

調大於 0.01 像是 0.25

圖像會變得模糊如下圖。

如果設置會 0.08，圖像則會有霧霧的感覺

(15%) Given novel view camera pose, your 3D gaussians should be able to render novel view images. Please evaluate your generated images and ground truth images with the following three metrics (mentioned in the 3DGS paper). Try to use at least three different hyperparameter settings and discuss/analyze the results.

- You also need to explain the meaning of these metrics.

PSNR

SNR (Signal to Noise Ratio)為訊號雜訊比。

Signal to Noise Ratio (SNR)

$$\text{SNR (dB)} = 10 \cdot \log_{10} \left[\frac{\sum_{x=1}^{N_x} \sum_{y=1}^{N_y} \left(f(x,y) \right)^2}{\sum_{x=1}^{N_x} \sum_{y=1}^{N_y} \left(f(x,y) - \hat{f}(x,y) \right)^2} \right]$$

PSNR (Peak Signal to Noise Ratio)也是訊雜比，只是訊號部分的值都改用該訊號度量的最大值。以RGB為例子，訊號度量範圍為 0 到 255 當作例子來計算 PSNR 時，訊號部分都當成是其能夠度量的最大值，也就是 255，而不是原來的訊號。

Peak Signal to Noise Ratio (PSNR)

$$\text{PSNR (dB)} = 10 \cdot \log_{10} \left[\frac{\sum_{x=1}^{N_x} \sum_{y=1}^{N_y} (255)^2}{\sum_{x=1}^{N_x} \sum_{y=1}^{N_y} \left(f(x,y) - \hat{f}(x,y) \right)^2} \right]$$

$$= 10 \cdot \log_{10} \left[\frac{255^2}{MSE} \right]$$

PSNR 與 SNR 的關係式：

$$\text{PSNR(dB)} = \text{SNR(dB)} + 10 \cdot \log_{10} \left[\frac{\sum_{x=1}^{N_x} \sum_{y=1}^{N_y} (255)^2}{\sum_{x=1}^{N_x} \sum_{y=1}^{N_y} \left(f(x,y) \right)^2} \right]$$

參考：https://cg2010studio.com/2014/12/10/opencv-snr-與-psnr/ (https://cg2010studio.com/2014/12/10/opencv-snr-%E8%88%87-psnr/)

可以看到 PSNR 的分母都固定，因此比較在意分子以也就是 pixel value 的最大差異，因此對影像特性更敏感，而因為 SNR 的分母為每個訊號的平方加總，因此算出來的值會被每個 pixel value 主導，不能很好地捕捉局部區域的細微變化。

- Please report the PSNR/SSIM/LPIPS on the public testing set.
- Also report the number of 3D gaussians.
- Different settings such as learning rate and densification interval, etc.

SSIM

參考 Structural_similarity_index_measure

(https://en.wikipedia.org/wiki/Structural_similarity_index_measure)

$$l(\mathbf{x}, \mathbf{y}) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}$$

SSIM 考慮亮度，對比度與結構(structure)

亮度是整張 image 的像素每個通道 Red, Green, Blue 分別去計算後再取平均的平均值。得到 μx 與 μy 。來確保兩張圖的亮度。因為人類對整體亮度改變比較明感，影像的過量或過暗都會直接影響視覺感知。所以 SSIM 的亮度是拿平均去計算。

$$c(x, y) = \frac{2\sigma_x\sigma_y + c_2}{\sigma_x^2 + \sigma_y^2 + c_2}$$

σx 和 σy 是影像的某部份像素值的標準差。因為標準差為像素值變化的指標，如果標準差大代表影像的該區域對比度很大，標準差小代表影像的該區域對比度很小。這個標準差是為了要平衡剛剛計算亮度直接用平均，因為有可能平均亮度一樣，也就是總像素值相加相同，但其實像素值大小的分佈差很大。但如果 image 中有局部過亮或過暗區域，對比可能會被局部亮度影響，而忽略了整體結構。且因為 Red, Green, Blue 分別去計算，對比度的計算也會隨著色彩分布的不同而不準確。

$$s(x, y) = \frac{\sigma_{xy} + c_3}{\sigma_x\sigma_y + c_3}$$

結構性也就是相關係數 如果 x 和 y 在局部區域內的像素值變化方向一致（如邊緣線條、紋理的分佈相似），則 σxy 接近 σxσy，此時 s(x,y)≈1。 如果 x 和 y 的變化方向不一致（如紋理、邊緣方向不同），σxy的值會較小甚至為負，此時s(x,y) 接近 0 或負值（在實際計算中會歸一化為非負數）。σxy 測量兩張影像在局部區域的結構變化是否一致。

最後將這三項相乘得到結果：

$$\mathrm{SSIM}(x, y) = l(x, y)^{\alpha} \cdot c(x, y)^{\beta} \cdot s(x, y)^{\gamma}$$

LPIPS

原始程式碼:

https://github.com/richzhang/PerceptualSimilarity/blob/31bc1271ae6f13b7e281b9959ac24a5e8f2ed522/lpips/lpips.py#L112

(https://github.com/richzhang/PerceptualSimilarity/blob/31bc1271ae6f13b7e281b9959ac24a5e8f2ed522/lpips/lpips.py#L112)

$$d(x, x_0) = \sum_l \frac{1}{H_l W_l} \sum_{h,w} ||w_l \odot (\hat{y}^l_{hw} - \hat{y}^l_{0hw})||_2^2$$

LPIPS的值越低代表兩張圖片越相似。

將兩個image作為 VGG　的輸入後，經過 VGG 的每一層都要做 normalize。

```
def normalize_tensor(in_feat,eps=1e-10):
    norm_factor = torch.sqrt(torch.sum(in_feat**2,dim=1,keepdim=True))
    return in_feat/(norm_factor+eps)
```

再算兩個 feature 相減的平方。

```
for kk in range(self.L):
    feats0[kk], feats1[kk] = lpips.normalize_tensor(outs0[kk]), lpips.normalize_tensor(outs1[kk])
    diffs[kk] = (feats0[kk]-feats1[kk])**2
```

再將全部相加起來，取平均。或是如果有用 spatial 的話則是對每層特徵的差異經過線性變換（self.lins[kk]）後 unsample 回原始影像的尺寸（由 in0.shape[2:] 決定）。

```
if(self.lpips):
    if(self.spatial):
        res = [upsample(self.lins[kk](diffs[kk]), out_HW=in0.shape[2:]) for kk in range(self.L)]
    else:
        res = [spatial_average(self.lins[kk](diffs[kk]), keepdim=True) for kk in range(self.L)]
else:
    if(self.spatial):
        res = [upsample(diffs[kk].sum(dim=1,keepdim=True), out_HW=in0.shape[2:]) for kk in range(self.L)]
    else:
        res = [spatial_average(diffs[kk].sum(dim=1,keepdim=True), keepdim=True) for kk in range(self.L)]

val = 0
for l in range(self.L):
    val += res[l]

if(retPerLayer):
    return (val, res)
else:
    return val
```

這個指標是通過 VGG 神經網路，來提取特徵，可前兩者單純用 RGB 處理，能比較兩張圖片潛在的特徵的差異，這些潛在的特徵是由圖片的每個像素不同的 RGB 值複雜的關係組成，而不只是單純靠圖片的 RGB 值用簡單的加減乘除(加總、平均、標準差)計算』′來。

我的 setting 1 是，setting 2 則是換 percent_dense 換成 0.025 。 setting 3 則是換 percent_dense 換成 0.008 。

```
self.iterations = 30_000
self.position_lr_init = 0.00016
self.position_lr_final = 0.0000008
self.position_lr_delay_mult = 0.01
self.position_lr_max_steps = 30_000
self.feature_lr = 0.0025
self.opacity_lr = 0.05
self.scaling_lr = 0.005
self.rotation_lr = 0.001
self.exposure_lr_init = 0.01
self.exposure_lr_final = 0.001
self.exposure_lr_delay_steps = 0
self.exposure_lr_delay_mult = 0.0
self.percent_dense = 0.01
self.lambda_dssim = 0.35
self.densification_interval = 100
self.opacity_reset_interval = 3000
self.densify_from_iter = 500
self.densify_until_iter = 15_000
self.densify_grad_threshold = 0.0002
self.depth_l1_weight_init = 1.0
self.depth_l1_weight_final = 0.01
self.random_background = False
self.optimizer_type = "default"
```

percent_dense	**PSNR**	**SSIM**	**LPIPS (vgg)**	**Number of 3D gaussians**
Setting 1 : 0.01	36.4517593 (avg)	0.9772456 (avg)	0.0475243	573729
Setting 2 : 0.025	35.1542625	0.9719375	0.0604806	469995
Setting 3 : 0.008	33.6018791	0.9655844	0.0732194	608115
Setting 2 (You need to write your setting)	11.8842154	0.7187936	0.1298615	491500

可以看到 percent_dense 變大其實 number of 3D gaussians 會變少。這是由於點的散步更均勻了，讓較少的點需要做密化操作，也就是 point densification 。所以最後的 3D gaussians 數量變少 。

`percent_dense` 是一個比例值，表示允許的最大密集程度 (density)，在 `densify_and_clone()` 中

```
torch.max(self.get_scaling, dim=1).values <= self.percent_dense * scene_extent
```

`scene_extent` 與 `percent_dense` 相乘，定義了一個相對於場景大小的密集度上限。
篩選條件中，檢查 self.get_scaling（點的縮放屬性）是否小於這個上限，確保選出的點不會超過指定的密集程度。如果某些點的值超過 `self.percent_dense * scene_extent` ，這些點都不會被包含到新的資料中。只有小於這個值的 3D gaussian 才會被複製。

```
selected_pts_mask = torch.logical_and(selected_pts_mask,
                                      torch.max(self.get_scaling, dim=1).values
                                      <= self.percent_dense * scene_extent)
```

這段程式碼用 `torch.max(self.get_scaling, dim=1).values <= self.percent_dense * scene_extent` 控制了點的篩選標準，影響哪些點可以被保留或進行密化操作。

(15%) Instead of initializing from SFM points [dataset/sparse/points3D.ply], please try to train your 3D gaussians with random initializing points.

用這組設定換成　random points

```
self.iterations = 30_000
self.position_lr_init = 0.00016
self.position_lr_final = 0.0000008
self.position_lr_delay_mult = 0.01
self.position_lr_max_steps = 30_000
self.feature_lr = 0.0025
self.opacity_lr = 0.05
self.scaling_lr = 0.005
self.rotation_lr = 0.001
self.exposure_lr_init = 0.01
self.exposure_lr_final = 0.001
self.exposure_lr_delay_steps = 0
self.exposure_lr_delay_mult = 0.0
self.percent_dense = 0.01
self.lambda_dssim = 0.35
self.densification_interval = 100
self.opacity_reset_interval = 3000
self.densify_from_iter = 500
self.densify_until_iter = 15_000
self.densify_grad_threshold = 0.0002
self.depth_l1_weight_init = 1.0
self.depth_l1_weight_final = 0.01
self.random_background = False
self.optimizer_type = "default"
```

- Describe how you initialize 3D gaussians
  先獲取原始點的坐標和邊界範圍，每個維度的最小與最大值

```
points = np.asarray(pcd.points)
min_bound = points.min(axis=0)
max_bound = points.max(axis=0)
```

在依據相同的點的數量，在邊界範圍生成點。

```
random_points = np.random.uniform(low=min_bound, high=max_bound, size=points.shape)
```

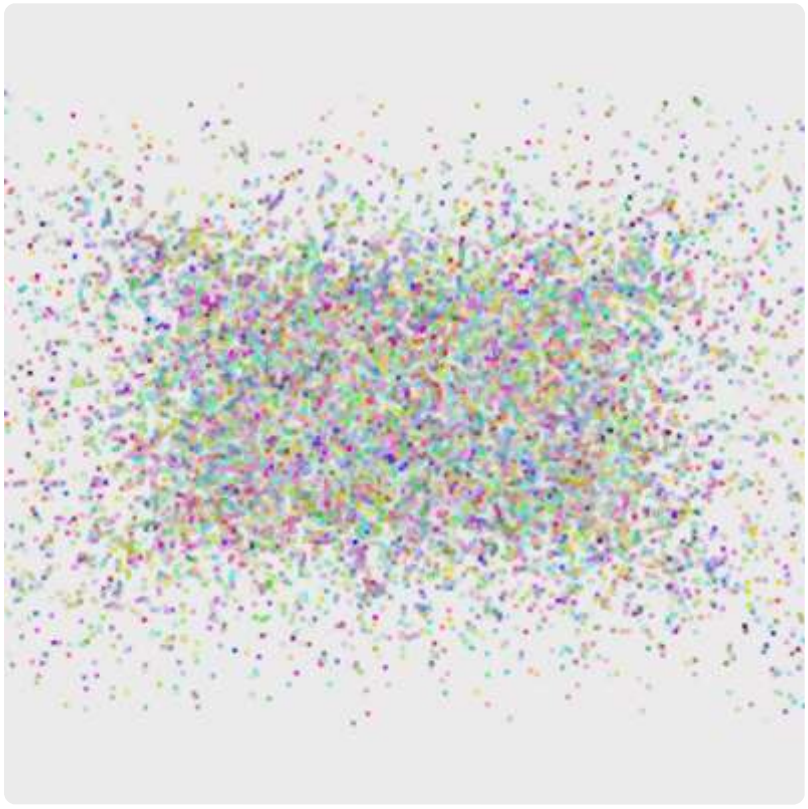
點的顏色也是隨機的

```
random_colors = np.random.uniform(low=0, high=1, size=(len(random_points), 3))
```

```
pcd.points = o3d.utility.Vector3dVector(random_points)
pcd.colors = o3d.utility.Vector3dVector(random_colors)
```

- Compare the performance with that in previous question.
  助教初始給的 SFM points 有 13414 個點
  

隨機初始化的 random points 有 13414 個點



percent_dense	PSNR	SSIM	LPIPS (vgg)	Number of 3D gaussians	3D points image
助教初始給的 SFM points	36.4517593	0.9772456	0.0475243	573729	
randomlize SFM points	34.2301407	0.9654032	0.0710974	673141	

隨機的話會在同樣的 30000 迭代後仍然有些霧霧的前景，而且 Number of 3D gaussians 也變得很多。

我同樣的設定，用不是初始化的 3D points。

Performance (40%)

PSNR and SSIM scores should both be above the baseline scores to get points

- Public baseline (on the public testing set):
  - Baseline (20%):
    - PSNR: 35

    - SSIM: 0.97
- Private baseline (on the private testing set):
    - Baseline (20%):
        - PSNR: TBD
        - SSIM: TBD

PSNR	SSIM
Testing psnr 36.45175833702088 (avg)	Testing ssim 0.9767982854172302 (avg)

install

```
$ conda install pytorch==1.12.1 torchvision==0.13.1 torchaudio==0.12.1 cudatoolkit=11.6
```

Reference

A Survey on 3D Gaussian Splatting (https://arxiv.org/pdf/2401.03890)

3D Gaussian Splatting for Real-Time Radiance Field Rendering (https://arxiv.org/pdf/2308.04079)

NeRF: Representing Scene as Neural Radiance Fields for View Synthesis (https://arxiv.org/pdf/2003.08934)

3D 重建技術 (三): 3D Gaussian Splatting，讓 3D 重建從學術走向實際應用的革新技術！用彩色橢圓球球表示場景！ (https://youtu.be/UxP1ruyFOAQ)

code 解釋 (https://www.ewbang.com/community/article/details/1000101327.html)

開發過程

安裝

方法 1: 安裝完整的 CUDA 開發工具 (cudatoolkit-dev)
在 Conda 虛擬環境中安裝 cudatoolkit-dev，這會包含 nvcc：

```
$ conda install -c conda-forge cudatoolkit-dev
```

安裝完成後，檢查 Conda 虛擬環境的 bin 目錄：

```
$ ls $CONDA_PREFIX/bin/nvcc
```

如果存在，則表示 nvcc 已安裝成功。

方法 2: 更新 PATH
激活虛擬環境後，手動覆蓋 PATH，將 Conda 的 CUDA 工具包放在 PATH 的最前面：

```
$ export PATH=$CONDA_PREFIX/bin:$PATH
$ export LD_LIBRARY_PATH=$CONDA_PREFIX/lib:$LD_LIBRARY_PATH
```

再次檢查 nvcc 的路徑：

```
$ which nvcc
```

應該返回：

```
/home/master/13/weihsinyeh/miniconda3/envs/<your_env>/bin/nvcc
```

```
$ conda install -c conda-forge cudatoolkit-dev
$ ls $CONDA_PREFIX/bin/nvcc
$ export PATH=$CONDA_PREFIX/bin:$PATH
$ export LD_LIBRARY_PATH=$CONDA_PREFIX/lib:$LD_LIBRARY_PATH
$ which nvcc
$ pip install submodules/simple-knn/
$ pip install submodules/fused-ssim/
$ pip install submodules/diff-gaussian-rasterization/
```

操作命令

```
$ python train.py -s /project/g/r13922043/hw4_dataset/dataset/train/ > log/1126
$ python render.py -m ./output/1b4e97e7-a  -s /project/g/r13922043/hw4_dataset/dataset/t
$ python grade.py ./output/1b4e97e7-a/train/ours_30000/renders ./output/1b4e97e7-a/train
```

```
$ bash hw4.sh /project/g/r13922043/hw4_dataset/dataset/public_test ./final_test
$ python gaussian-splatting/evaluation.py --source_path /project/g/r13922043/hw4_dataset
$ python grade.py ./final_test /project/g/r13922043/hw4_dataset/dataset/public_test/imag
```