Introduction to Image Processing HW2 Report

1. 名詞解釋

- **Deep Learning**: Deep learning is a subset of machine learning that involves training artificial neural networks to recognize patterns and make predictions or decisions based on input data. The term "deep" refers to the fact that these neural networks have multiple layers of interconnected nodes (neurons), allowing them to learn increasingly complex features automatically by processing the raw input data through its layers.

- **Multilayer Perceptron**: A multilayer perceptron is a type of artificial neural network that is commonly used in deep learning. It consists of multiple layers of interconnected nodes (neurons), with each layer typically containing several neurons. The first layer of an MLP takes in the input data, which is then passed through the subsequent layers of neurons. Each neuron in a layer computes a weighted sum of the outputs from the previous layer, applies an activation function to this sum, and passes the result to the next layer. The output layer of the MLP produces the final prediction or classification result.

- **Implicit Neural Representation**: An implicit neural representation is a type of neural network that learns to represent a function implicitly, without explicitly computing its output for every input value. In other words, instead of directly producing an output value for a given input, an INR learns to map the input to a high-dimensional feature space, where the function can be represented by the distribution of the features.

- **View Synthesis**: View synthesis is the process of generating novel views of a scene from a given set of input views. The goal of view synthesis is to produce new views of the scene that are visually plausible and consistent with the input views.

- **NeRF (Neural Radiance Fields)**: Neural Radiance Fields (NeRF) is a state-of-the-art method for representing complex 3D scenes using deep neural networks. The main idea behind NeRF is to model a scene as a continuous 3D function that predicts the appearance of a point in space from any viewing direction. To achieve this, NeRF uses a deep neural network to predict the volume density and view-dependent radiance at each point in a 3D scene. The density predicts how much light passes through a point, while the radiance predicts the color of the light that passes through the point. By integrating these predictions along a ray that passes through the 3D space, it is possible to generate a 2D image from any viewpoint.

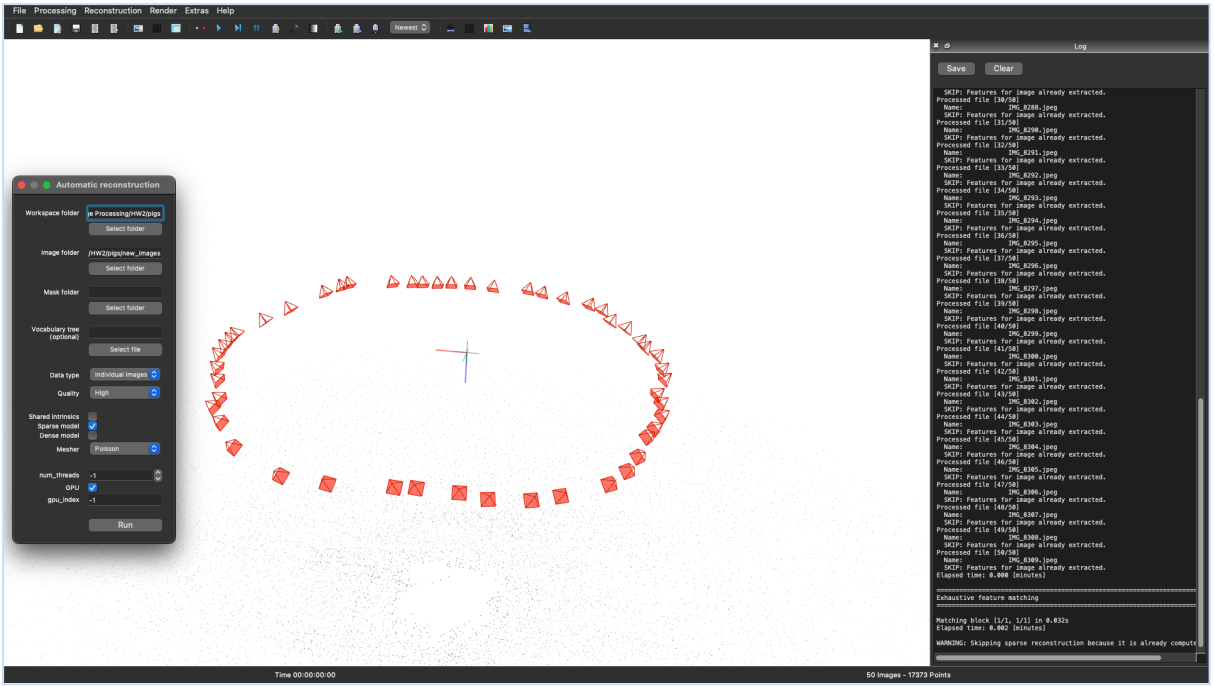
2. 實作方法

(0) GitHub 程式碼連結、成果與資料集連結

(1) 拍攝照片：拍攝 50 張小豬不同角度的照片，成果如連結所示。

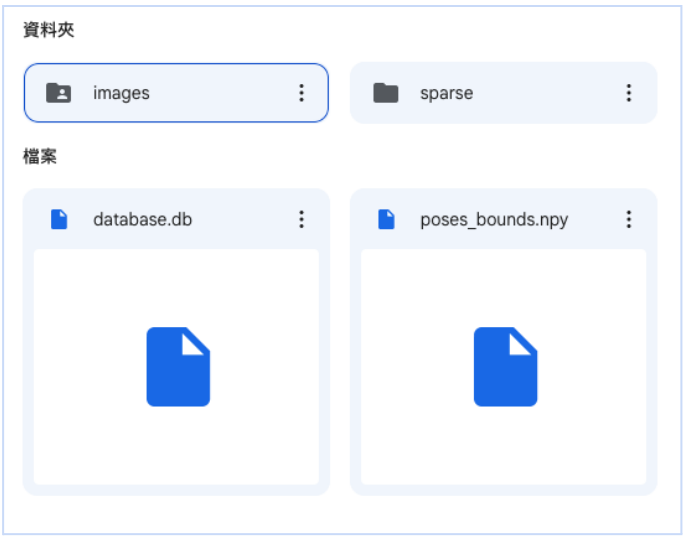
(2) 使用 colmap 做前處理：原本想使用 GitHub 所提供的 colab notebook 來跑 colmap，但試過多種方法都無法成功，最後是採取於本機端安裝程式的方式執行（至官方 GitHub 下載 pre-release 的執行檔），那我原本有遇到記憶體不夠導致 colmap 閃退以及 can not

read image file format 的問題，但將圖片品質降低並裁切後就有解決，最終執行 automatic reconstruction 的成果如下圖所示：



(3) 執行完 reconstruction 後產生 sparse 以及 database.db，接著使用 LLFF 提供的 imgs2poses.py 生成 poses_bound.npy，拿到相機拍攝的位置資訊（執行指令詳見上方連結）

(4) 將 images, sparse, database.db 以及 poses_bound.npy 上傳到雲端資料夾中（需先於雲端開設資料夾 -> root_dir，並將這四個放在該資料夾下）



(5) 使用 GitHub 所提供的 colab 訓練 notebook 訓練資料，因為套件的環境問題我實在一直搞不定，所以我在 requirement.txt 內沒有使用他指定的版本，我全部都不指定版本，直接安裝預設版本，然後對於 train.py 中有與最新版本不相容的地方都有做修正，所以我的 train.py 更改成以下所示

```python
# pytorch-lightning
from pytorch_lightning.callbacks import ModelCheckpoint
from pytorch_lightning import LightningModule, Trainer
from losses import loss_dict
from pytorch_lightning.loggers import TensorBoardLogger
```

PyTorch 2.0 中，以 TensorBoearLogger 取代 TestTubeLogger

```python
class NeRFSystem(LightningModule):
    def __init__(self, hparams):
        super(NeRFSystem, self).__init__()
        # self.hparams = hparams
        self.save_hyperparameters(hparams)
        self.loss = loss_dict[hparams.loss_type]()

        self.embedding_xyz = Embedding(3, 10) # 10 is the default number
        self.embedding_dir = Embedding(3, 4) # 4 is the default number
        self.embeddings = [self.embedding_xyz, self.embedding_dir]

        self.nerf_coarse = NeRF()
        self.models = [self.nerf_coarse]
        if hparams.N_importance > 0:
            self.nerf_fine = NeRF()
            self.models += [self.nerf_fine]
```

PyTorch 2.0 中，需使用 save_hyperparameters() 來儲存超參數

```python
def validation_epoch_end(self, outputs):
    mean_loss = torch.stack([x['val_loss'] for x in outputs]).mean()
    mean_psnr = torch.stack([x['val_psnr'] for x in outputs]).mean()

    self.log('val_loss', mean_loss) # log the validation loss using self.log()
    self.log('val_psnr', mean_psnr)

    return {'progress_bar': {'val_loss': mean_loss,
                             'val_psnr': mean_psnr},
            'log': {'val/loss': mean_loss,
                    'val/psnr': mean_psnr}
            }
```

PyTorch 2.0 中，在 val_loss 的 moniter mode 中需使用 self.log() 印出 log

```python
if __name__ == '__main__':
    hparams = get_opts()
    system = NeRFSystem(hparams)
    dirpath = '../new_ckpts/' + hparams.exp_name
    checkpoint_callback = ModelCheckpoint(dirpath=dirpath, filename='{epoch:d}',
                                          monitor='val_loss',
                                          mode='min',
                                          save_top_k=-1,)
```

PyTorch 2.0 中，filepath 這個 argument 改為 dirpath 和 filename

```python
    logger = TensorBoardLogger(
        save_dir="logs",
        name=hparams.exp_name,
        # debug=False,
        # create_git_tag=False
    )
```

logger 改用新的 TensorBoardLogger 並註解掉不支援的 argument

```python
    trainer = Trainer(max_epochs=hparams.num_epochs,
                      enable_checkpointing=True,
                      callbacks=checkpoint_callback,
                      resume_from_checkpoint=hparams.ckpt_path,
                      logger=logger,
                    #   early_stop_callback=None,
                    #   weights_summary=None,
                    #   progress_bar_refresh_rate=1,
                      accelerator='gpu',
                      devices=hparams.num_gpus,
                    #   distributed_backend='ddp' if hparams.num_gpus>1 else None,
                      num_sanity_val_steps=1,
                      benchmark=True,
                    #   profiler=hparams.num_gpus==1
                      )

    trainer.fit(system)
```

將 argument 改為新的名稱並註解掉不支援的 argument

(6) 最後使用 eval.py 生成 GIF

(*) 深度學習網路之函數解釋與參數說明

nerf.py:

a. **class 'Embedding'**: 將一個三維空間點 x 做 positional encoding，使用 sin, cos 將低維資料投影到高維空間後輸出，以便後續的神經網路可以使用此高頻特徵進行建模。此 class 的輸入 in_channels 表示輸入維度，N_freqs 則是影響輸出的頻率波段數量，輸出為輸入資料 x 在高維的投影

b. **class 'NeRF'**: 此 class 定義一個神經網路用於生成 emitted color 跟 volume density，volume density 指的是這個射線會停在這個 3D 位置的機率，而 emmited color

則是由某個視角看過去該 3D 位置的顏色。將 3D 位置與 view direction 輸入此神經網路後會通過 8 層 FC layer，輸出 volume density 與 volume density 以及 256 維的 latent feature。而後與 view direction concatenate，再通過 FC layer 得到 emitted color

train.py:

NeRSystem 定義了 NeRF 模型以及 training, testing 的流程

a. **__init__**: 將執行 command 所輸入之 hyper parameters 存好，並建立 embed position 以及 direction 的 embedding 層，接著根據 N_importance (網格細分時所使用的層數) 生成對應數量的 NeRF

b. **decode_batch**: 從 batch 中提取出訓練所需的光線 rays 以及顏色 rgb

c. **forward**: 將光線 rays 投影到 3D 空間中，並跟根據 chunk 數值決定要將 rays 切成幾個 chunk 來計算，接著根據場景的材質、光源等信息計算出每個像素的rgb值和深度值等信息

d. **prepare_data**: 將 dataset 讀進來

e. **configure_optimizers**: 根據輸入指令參數設定 optimizer 以及 scheduler, optimizer 決定參數更新的方向和大小，它的作用是將模型的損失函數最小化，而 scheduler 則用於動態調整模型訓練中的 learning rate 使訓練更加穩定

f. **train_dataloader**, **val_dataloader**: load 將用於 training 及 validation 的 datasets

g. **training_step**: 定義 train 的過程，包括 loss 以及 psnr 的計算

h. **validation_step**: 定義 validation 的過程，像是 loss 的計算等等

i. **validation_epoch_end**: 在每個 epoch 結束時 log 出 loss

j. **main**: 設定 checkpoints, logger, trainer 等架構，並開始執行訓練 (fit)

Common Hyperparameters:

a. **batch size**: 一次訓練中使用的樣本數量，也就是在一次 back propagation 中，用於更新參數的樣本數量

b. **optimizer**: 決定參數更新的方向和大小，它的作用是將模型的損失函數最小化，常見的 optimizer 包括 SGD、Adam、Adagrad

c. **lr**: learning rate 的縮寫，用於控制模型參數更新速度

d. **weight_decay**: Weight decay 是一種正規化技術，用於降低模型的過度擬合。在訓練過程中，模型的權重會不斷更新，為了避免模型過度擬合訓練數據，就需要控制權重的值，以免權重值過大，它通過將權重值添加到損失函數中，限制了模型權重的增長

e. **lr_scheduler**: 用於動態調整模型訓練中的 learning rate 使訓練更加穩定

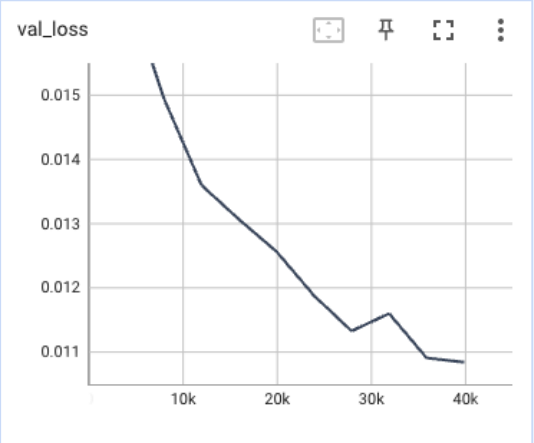
f. **decay_gamma**: 用於控制 learning rate 的衰減速率

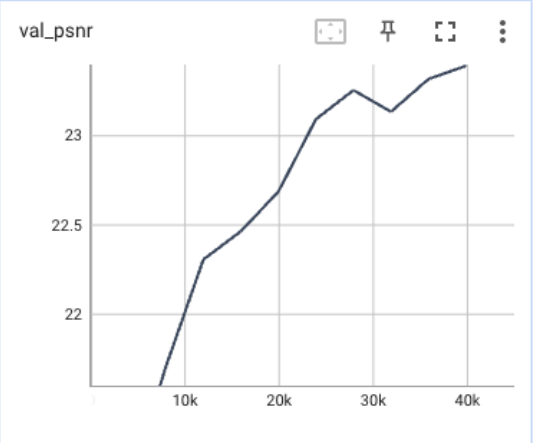
其他的 hyperparameters 可以在 opt.py 中查看相關的解釋

3. 深度學習模型訓練及評估結果

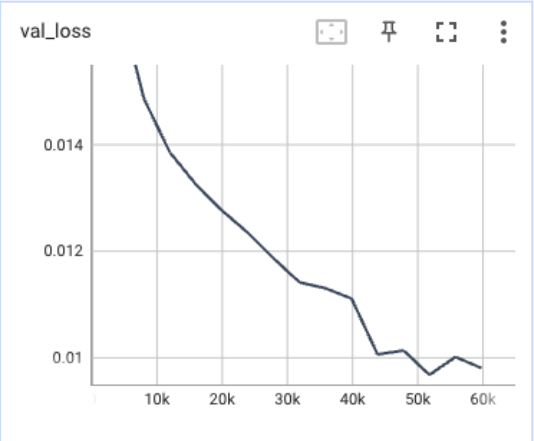
(1) **training**:

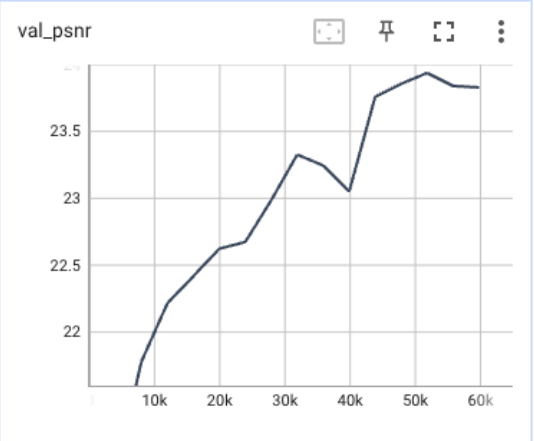
10 epochs:



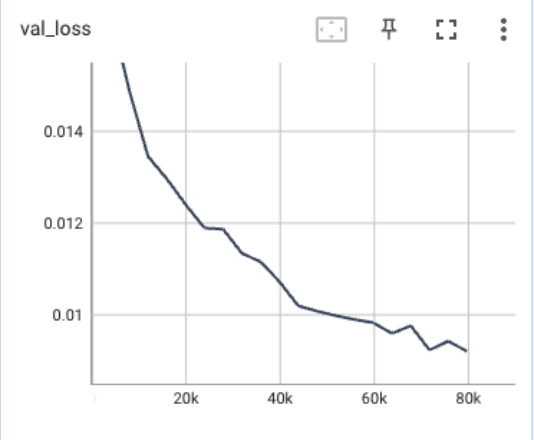


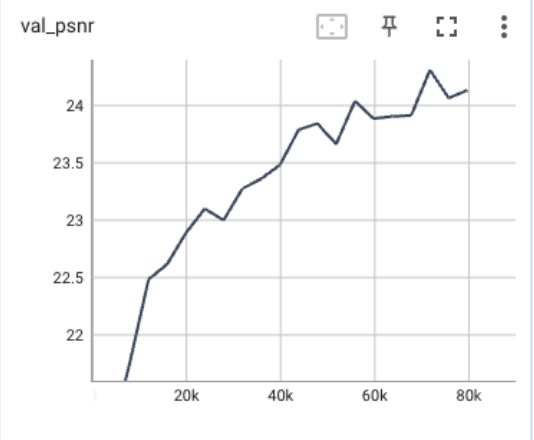
15 epochs:





20 epochs:





根據上方的圖片可以觀察到，PSNR 隨著 epoch 的增加而增加，loss 也越來越小，表示模型逐漸收斂，而使用 20 epoch 的 psnr 也高過 15, 10 epoch。因為 PSNR (Peak Signal-to-Noise Ratio) 是一種用來測量兩個信號之間質量差異的指標，所以如果生成圖像與目標圖像間的 PSNR 越大，表示生成圖片的品質越好，由此可知 20 epoch 所產生的圖片品質最高

(2) **testing**:

10 epochs	**15 epochs**	**20 epochs**

上方從左到右分別是 10, 15, 20 epochs 生成的圖片，可以發現在 10 epochs 的時候物體都還是有一些奇怪的顏色，而隨著 epoch 增加這些奇怪的顏色都在慢慢減少，成像越來越清晰，但圖片仍然有背影雜訊

訓練生成的圖片檔連結

4. 結果分析與討論

觀察生成的 GIF 及圖片可以發現，不管是在 10 epochs 還是 20 epochs，生成的圖片幾乎都還是含有許多雜訊，我猜測可能是因為 NeRF 在做 ray tracing 的過程中假設只有一個 light source，而我的照片是在白天的室內拍攝，所以光源有室內的日光燈以及室外的自然光，光源的方向很不一樣，所以可能導致 NeRF 在做 ray tracing 的效果不太好。那根據這個推斷，我有重新拍攝照片做訓練，成果如下圖所示，可以看到我就算只跑 10 epochs，生成出來的圖片品質也比之前訓練 20 epochs 的效果都好（雜訊很明顯消失），跑到 20 epochs 的時候成像就非常清晰了

Original 10 epochs	**New 10 epochs**	**New 20 epochs**

改善之後生成的圖片檔連結

NeRF 的優點:

a. 精度高:因為加入了 positional encoding 以及 hirachical volume sampling 所以使得 NeRF 可在相對較短時間內實現高質量的三維重建，同時在表面細節與光照效果表現出色

b. 靈活度高:因為使用了 Implicit Neural Representation 來學習一個隱式函數來表示複雜的幾何形狀和物體表面，所以 NeRF 可以適應各種不同類型包括具有負責幾何形狀與紋理的物體

NeRF 的缺點:

a. 計算成本高:NeRF 的訓練及 render 都需要大量的運算資源，這對於即時的運用以一定的限制

b. 訓練圖片品質要求高:根據我上方重做的實現以及結合週遭同學們遇到的困難，可以發現 NeRF 對於訓練圖片品質的要求很高，只要背景過於雜亂甚至是有多個光源都會造成訓練結果不佳

c. 難以處理動態場景:NeRF 對於動態物體或場景的處理會有困難，因為隨著物體的移動或姿態的變化，隱式函數都會需要重新計算和渲染