NeRF + LSA

현재 공개된 버전의 NNCodec은 LSA를 적용 가능한 모델이 Classification Task로 국한되어 있습니다.
(특히 ImageNet에 한정되어 있으나, 이는 내부 코드에 약간의 수정을 거치면 변경 가능합니다).

본 프로젝트에서는 NeRF 모델에 대해서도 LSA를 포함한 압축을 수행할 수 있도록, 기존 NNCodec의 구조를 변경하였습니다.

가이드의 내용은 다음과 같습니다.

코드 실행 방법

NeRF + LSA 코드 실행과 결과 저장, 모델 평가에 필요한 패키지들은 `/home/gbang/jihyoun/NeRF/nerf_lsa` 에 모여 있으며, 이들의 역할은 다음과 같습니다.

1. `/nnc`, `/nnc_core`, `/framework` : NNCodec을 구성하는 세 package 들입니다. NeRF + LSA의 동작을 위해 원본 NNCodec에 비해 상당 부분 수정되었으며, 이에 대한 설명은 부록) 코드 구조 에서 확인 가능합니다.
2. `compress_nerf.py` : 메인 코드입니다. 사용법은 아래에서 확인 가능합니다.
3. `main.sh` : compress_nerf.py를 실행하는 shell script 파일입니다.
4. `/results` : 결과값을 저장하는 백업 폴더입니다. compress_nerf.py의 `base_path_to_save` 변수를 해당 폴더로 지정하는 경우, `/results` 폴더에 자동으로 저장됩니다.
5. `utils.py` : NNCodec에 NeRF 모델을 입력하기 전과 후 과정에, 모델 구조 변환 및 기타 역할을 수행하는 모듈입니다.

추가로 NeRF 폴더에 존재하는 nerf-pytorch 코드는 github에 존재하는 nerf-pytorch 원본 코드이며, nerf_lsa를 통해 생성된 모델을 입력하여 원활하게 동작하는지를 확인하기 위해 사용됩니다.

메인 코드는 `compress_nerf.py` 이며, 이를 실행하는 방법은 다음과 같습니다.

1. `conda activate nnc`를 통해 가상환경을 활성화합니다.
2. `/nerf_lsa`에 저장된 Shell script (`main.sh`)를 통해 `compress_nerf.py`를 실행합니다.

> **주의 : NeRF + LSA를 사용하기 위해 변경된 NNCodec은 NeRF가 아닌 일반적인 모델의 압축 및 복원도 가능하도록 디자인되었으나 동작이 확인되지는 않았으므로, 일반적인 모델의 경우에는 official code를 이용하는 편을 추천드립니다.**

main.sh의 구조는 아래와 같습니다.

```
#!/bin/bash

set -e

CKPT_PATH="model_zoo/blender_paper_lego/lego_200000.tar"
CKPT_NICKNAME='lego_200K'
BASE_PATH_TO_SAVE='.'
DATASET_PATH='~'

CUDA_VISIBLE_DEVICES=0 python compress_nerf.py \
    --ckpt_path $CKPT_PATH \
    --ckpt_nickname $CKPT_NICKNAME \
    --base_path_to_save $BASE_PATH_TO_SAVE \
    --qp -20 \
    --lsa True \
    --epochs 2 \
    --learning_rate 0.0001 \
    --task_type NeRF \
    --dataset_type llff \
    --N_iters 50001 \
    --learning_rate_decay 0.1 \
    --i_save 50000 \
    --dataset_path $DATASET_PATH
```

사용되는 argument들은 다음과 같습니다.

- **CKPT_PATH** : 압축하고자 하는 원본 NeRF checkpoint의 위치입니다.
- **CKPT_NICKNAME** : 압축 결과물들을 저장하는 폴더의 이름을 명명하는 과정에서 사용됩니다. 자유롭게 설정 가능합니다.
- **BASE_PATH_TO_SAVE** : 압축 결과물들을 저장하고자 하는 기본 경로를 의미합니다. 압축 결과는 BASE_PATH_TO_SAVE 내에 자동으로 생성되는 하위 폴더 내에 저장됩니다.
- **DATASET_PATH** : NeRF 압축 과정에서는 사용되지 않습니다.
- **CUDA_VISIBLE_DEVICES** : python 파일을 실행하는 데에 사용되는 GPU device를 강제로 설정하는 parameter입니다. 본 코드는 multi-GPU를 지원하지 않으므로, 사용하고자 하는 GPU 번호 하나만을 지정하여 주면 됩니다.
- **qp** : Video Codec의 QP와 동일합니다. -38이 기본값이며, 커질수록 더 큰 압축률을 의미합니다 (예시) -10 : 큰 압축률, -40 : 작은 압축률)
- **lsa** : LSA를 적용할지의 여부를 의미합니다.

아래의 parameter들은 모두 **LSA가 True**인 경우에만 사용됩니다.

- **epochs** : epoch의 통상적인 의미와 조금 다르게 사용되었습니다. 단순히 설정된 N_iters 만큼의 iteration을 마치면 1epoch가 끝나게 되며, 각 epoch가 끝날 때마다 scheduler.step()을 적용할 수 있습니다.
- **learning_rate** : LSA parameter 학습 시의 최초 learning rate를 의미합니다.
- **task_type** : 기존의 classification인지, 아니면 NeRF인지를 나누기 위한 parameter입니다. NeRF 모델에 LSA를 적용하고자 하는 경우 "NeRF"로, Classification 모델에 LSA를 적용하고자 하는 경우 "Classification"으로 설정하면 됩니다.
- **dataset_type** : "llff" (fern dataset), "blender" (lego dataset) 의 두 가지 중에서 사용 가능합니다. 압축하고자 하는 모델에 맞추어 선택하시면 되겠습니다.

- **N_iters** : 1epoch를 몇 iteration으로 할 것인지를 결정합니다.
- **learning_rate_decay** : 1epoch 마다 learning rate를 얼마나 줄일 것인지를 결정합니다. 0으로 지정하는 경우 적용되지 않습니다.
- **i_save** : 몇 iteration마다 중간 결과를 저장할 것인지를 선택합니다. 중간 모델, inference 결과가 저장됩니다.

코드 실행 결과물

압축-복원 과정을 통해 수행된 결과는 자동으로 생성된 폴더 내에 저장되며, 아래의 구조를 가집니다.
폴더명의 의미는 다음과 같습니다.

> **{현재 년월일시간분초} _ {CKPT_NICKNAME} _ qp{qp} _ e{epochs} _ lr{learning_rate (. 대신 p)} _ decay {learning_rate_decay} _ N {N_iter} _ {dataset_type}**

아래는 폴더에 저장되는 결과물들의 예시입니다. (fern dataset, i_save = 50000, N_iters = 50000, epochs = 2인 경우)

```
230828181930_fern_200K_qp-20_e2_lr0p0001_decay0.1_N50001_llff
├── bitstream
│   └── 230828181930_fern_200K_qp-
20_e2_lr0p0001_decay0.1_N50001_llff_bitstream.nnc
├── movies
│   ├── step100000_rgb.mp4
│   ├── step1_rgb.mp4
│   └── step50000_rgb.mp4
├── reconstructed
│   ├── 230828181930_fern_200K_qp-
20_e2_lr0p0001_decay0.1_N50001_llff_reconstructed.pt
│   ├── 230828181930_fern_200K_qp-
20_e2_lr0p0001_decay0.1_N50001_llff_reconstructed.tar
│   ├── ckpt_step1.pt
│   ├── ckpt_step100000.pt
│   └── ckpt_step50000.pt
├── result.txt
├── testset_step1
│   ├── 000.png
│   ├── 001.png
│   ...
│   ├── 118.png
│   └── 119.png
├── testset_step100000
│   ├── 000.png
│   ├── 001.png
│   ...
│   ├── 118.png
│   └── 119.png
└── testset_step50000
    ├── 000.png
    ├── 001.png
    ...
    ├── 118.png
    └── 119.png
```

이미지 및 비디오 파일

각 N_iter마다 이미지 및 비디오 파일이 저장되며, 이미지 파일의 경우에는 testset_step000에, 해당 이미지들을 엮은 비디오 파일의 경우에는 /movies 안에 iteration 별로 저장되어 있습니다.
현재는 rgb만 저장되게 설정되어 있으며, disp 정보도 저장하고자 하는 경우에는
`nerf_lsa/framework/nerf_model/run_nerf.pyy` 의 train() 함수 하단에서 코드를 수정해 주시면 됩니다.

Reconstructed

reconstructed는 LSA parameter를 학습하는 도중의 NeRF 모델을, 그리고 최종적으로 복원된 NeRF 모델을 함께 저장하는 폴더입니다.
해당 폴더에 저장되는 모델의 종류는 다음과 같습니다.

..._reconstructed.pt

nnc.decompress_model() 함수의 결과물로, compress_model() 함수에 입력하였던 NeRFWrapper의 instance 형식의 모델 파일입니다. 이 파일은 아직 일반적인 NeRF checkpoint와는 다른 형태이므로, vanilla-NeRF code에 넣어 학습 또는 rendering이 불가능합니다.

..._reconstructed.tar

vanilla-nerf 코드에서 사용하는 checkpoint 형식과 일치하도록 저장된 형태입니다. compress_nerf.py에서 호출되는 마지막 함수인 convert_nerfwrapper_to_nerf_ckpt()를 이용하여 1번의 .pt 파일을 변환한 결과입니다. 해당 파일은 nerf-pytorch코드에 그대로 입력하여 학습 및 랜더링이 가능합니다.

ckpt_step000.pt

LSA parameter 학습 과정에서의 중간 결과가 저장된 파일입니다. 해당 모델은 아직 reconstruct 과정을 거치지 않은 encoder 단에서의 결과물이므로, LSA parameter가 포함되어 있어 다른 nerf-pytorch 모델에 옮겨 사용이 불가능합니다.

result.txt

모델의 매 iteration마다의 성능을 기록한 파일이며, PSNR과 loss를 포함합니다.

해당 파일은 `grapher.ipynb` 를 통해 그래프를 그릴 수 있습니다.

grapher.ipynb

grapher.ipynb의 path 변수를 원하는 result.txt의 파일 경로로 변경한 후, 전체 코드를 실행하면 그래프를 그릴 수 있습니다.
해당 파일에 사용되는 method들은 다음과 같습니다.

Sanity_Checker

저장된 result.txt 파일의 무결성을 검증합니다.
파일 내의 숫자들에 다른 문자가 섞여 있어 plot이 불가능한 경우, 해당 문자와 위치를 출력합니다.

예시 - 모두 정상인 경우

```
Drop last 105 data because length of data (94105) is not divided by
averaging(1000)
Drop last 105 data because length of data (94105) is not divided by
averaging(1000)

Initialize Sanity Check...

Sanity Checking On psnr
No Anomalies Detected.

Sanity Checking On loss
No Anomalies Detected.
```

예시 - 비정상 데이터가 있는 경우

```
Initialize Sanity Check...

Sanity Checking On psnr
1 Anomalies Are Detected. Indexes Are : [52]

Find 52th Number In Data
Data at Index 52 Is:  31.408a86

Sanity Checking On loss
No Anomalies Detected.
```

plot_data

그래프를 그리는 메인 부분입니다. ipynb 파일에서 결과 이미지를 띄움과 동시에, 동일 폴더에 `result.png`라는 이름으로 저장됩니다.

```
plot_data(data_path,
          plot_range_start=0,
          plot_range_end=None,
          average = 1000,
          layout='horizontal',
          num_plot_threshold=0
          )
```

- **plot_range_start, plot_range_end** : 각각 그래프를 그리고자 하는 시작, 끝 iteration을 의미합니다.
- **average** : 몇 개 단위로 평균하여 plot할 것인지를 결정합니다.
- **layout** : PSNR, loss에 대한 그래프를 가로, 또는 세로 방향으로 정렬할지를 선택합니다. 'horizontal', 'vertical'이 가능합니다.
- **num_plot_threshold** : plot 가능한 최대 개수 제한을 의미합니다. plot하고자 하는 대상이 너무 많은 경우, 혹시 모를 렉을 방지하기 위한 parameter입니다.

코드 구조

가장 먼저, NeRF + LSA를 구현하는 과정에서 발생했던 문제와 기본적인 해결 방법은 다음과 같습니다.

> **문제 : NeRF 모델은 model (= coarse model), model_fine (= fine_model) 의 두 모델이 Training 과정에서 동시에 필요한데, NNCodec은 한 번에 하나의 모델만 입력으로 넣는 것을 가정하여 구현되어 있음, LSA를 사용하는 경우 LSA parameter training 과정이 필수적이므로, 두 모델을 함께 입력해야 함**

> **해결 방법 : NeRF를 구성하는 두 가지 모델을 하나의 NeRFWrapper Class 객체로 묶어서, 두 모델이 함께 압축 및 복원되도록 함**

NeRFWrapper의 특징은 다음과 같습니다.

1. NNCodec의 입장에서 pytorch의 model로 인식되어야 하므로, nn.Module을 상속합니다. 다만 forward 함수를 따로 overwrite하지는 않습니다.
2. self.tuning_optimizer를 통해 train 함수를 여러 번 실행하여도 optimizer 상태를 유지합니다.
3. self.global_step을 통해 train 함수를 여러 번 실행하여도 총 step(=iteration)을 저장합니다.

```
class NeRFWrapper(nn.Module):

    def __init__(self, D=8, W=256,
                 input_ch=63, input_ch_views=27,
                 output_ch=4, skips=[4],
                 use_viewdirs=True):

        super(NeRFWrapper, self).__init__()

        self.model = NeRF(D=D, W=W,
                          input_ch=input_ch, input_ch_views=input_ch_views,
                          output_ch=output_ch, skips=skips,
                          use_viewdirs=use_viewdirs, is_fine=False)

        self.model_fine = NeRF(D=D, W=W,
                               input_ch=input_ch,
input_ch_views=input_ch_views,
                               output_ch=output_ch, skips=skips,
                               use_viewdirs=use_viewdirs, is_fine=True)

        # Will automatically assigned at
        # pytorch_model/__init__.BlenderNeRFModelExecuter.tune_model part
```

```
        self.tuning_optimizer = None
        self.global_step = 0
```

우선 compress_nerf.py에서 nnc/compression.py의 compress_model() 함수를 호출합니다.
해당 함수는 LSA가 True인 경우, LSA 적용을 위해 **ModelExecuter** 객체를 생성합니다
(ModelExecuter는 모델의 구조, 학습 방법 등을 포함하는 class입니다).
이후 compress() 함수를 호출하여 실질적으로 LSA를 포함한 압축 과정을 수행합니다.

LSA를 적용하여 NNCodec을 수행하기 위해서는, 해당 모델에 대한 ModelExecuter 객체가 생성되어야 합니다.
기본적으로 ImageNetModelExecuter만 존재하므로, 코드를 적절히 변경하여 NeRFModelExecuter 코드를 추가하였습니다.
아래의 create_NNC_model_instance_from_object() 를 수정하여 NeRFModelExecuter 생성이 가능하도록 하였습니다 (create_NNC_model_instance_from_file() 함수에 대해서는 구현하지 않았습니다).

```
    # ...

    # Check if the model is a PyTorch model
    elif pytorch_model.is_pyt_model(model_path_or_object):
        is_pyt_model = True

        # Create an NNC model instance from the PyTorch model object or
path
        if model_executer:
                nnc_mdl, _, model_parameters =
pytorch_model.create_NNC_model_instance_from_object(
                 model_path_or_object,
                )
        else:
            # CURRENT IMPLEMENTED NERF-LSA ONLY USE THIS TYPE OF NNC-MODEL
INITIALIZATION
            nnc_mdl, nnc_mdl_executer, model_parameters =
pytorch_model.create_NNC_model_instance_from_object(
                model_path_or_object,
                dataset_path=dataset_path,
                lr=learning_rate,
                batch_size=batch_size,
                num_workers=num_workers,
                model_struct=model_struct,
                lsa=lsa,
                epochs=epochs,
                max_batches=max_batches,
                task_type = task_type,
                dataset_type = dataset_type,
                N_iters = N_iters,
                learning_rate_decay = learning_rate_decay,
                i_save = i_save
                )
```

다음은 변경된 pytorch_model.create_NNC_model_instacne_from_object() 의 구조입니다.

기존에는 task_type이라는 parameter가 없이 항상 create_imagenet_model_executer()를 호출하였으나,
task_type parameter를 추가하여 imagenet과 NeRF의 두 branch로 나뉘어지도록 설정하였습니다.
추가로 create_nerf_model_executer() 함수에 dataset_type parameter도 추가하여, fern인지 lego인지를 선택 가능하도록 하였습니다.

```
############################## CUSTOM MODIFIED ##############################
"""
Add funtionality to check task_type and initialize PYTModelExecuter
Adaptively
Using create_imagenet_model_executer or create_blender_model_executer
"""
def create_NNC_model_instance_from_object(
                model_object,
                dataset_path,
                lsa,
                lr,
                epochs,
                task_type,
                dataset_type,
                N_iters,
                learning_rate_decay,
                i_save,
                max_batches=None,
                batch_size=64, # Only Applied for Classification Task
                num_workers=1,
                model_struct=None
                ):

    PYTModel = PytorchModel()

    # Returns parameter of PYTModel, and model object (loaded_model_struct
== model_object)
    # and internally, it changes private member variable
PYTModel.__model_info
    model_parameters, loaded_model_struct =
PYTModel.init_model_from_model_object(model_object)

    if model_struct == None and loaded_model_struct != None:
        model_struct = loaded_model_struct

    if dataset_path and model_struct:

        # Classification : Official supported task (ImageNet +
Classification)
        if task_type == 'Classification':
            PYTModelExecuter =
```

```
create_imagenet_model_executer(model_struct=model_struct,

dataset_path=dataset_path,
                                                                lr=lr,
                                                                epochs=epochs,

max_batches=max_batches,

batch_size=batch_size,

num_workers=num_workers,
                                                                lsa=lsa
                                                                )
        # Custom added task type
        elif task_type == 'NeRF':

            assert dataset_type in ['blender', 'llff']

            PYTModelExecuter = create_nerf_model_executer(
model_struct=model_struct,

dataset_type=dataset_type,
                                                            lr=lr,
                                                            epochs=epochs,

max_batches=max_batches,
                                                            lsa=lsa,
                                                            N_iters =
N_iters,

learning_rate_decay = learning_rate_decay,
                                                            i_save = i_save
                                                            )
        else:
            # If task_type is neither 'Classification' nor 'NeRF', raise an
error
            raise ValueError("Invalid task_type. Supported values are
'Classification' and 'NeRF'.")

        if lsa:
            # if LSA enabled
            # Model parameters have to include additional LSA parameter
            # So, model_parameters should changed to lsa-added ones
            model_parameters =
PYTModel.init_model_from_dict(PYTModelExecuter.model.state_dict())

    else:
        PYTModelExecuter = None

    return PYTModel, PYTModelExecuter, model_parameters
```

create_nerf_model_executer()는 NeRFModelExecuter class의 instance를 생성합니다.
이때 handler라는 변수가 등장하는데, 이는 해당 모델의 train / val / test function을 가지는 ModelSetting Class의 instance입니다.

다만 NeRF의 경우에는 Classification과 달리 train / val / test 가 명확히 나뉘어져 있지 않은데,
구현의 용이성을 위해 NeRF의 경우에는 handler의 train 부분을 제외하고는 단순히 이후 코드와의 일관성을 위한 dummy로 구현되어 있습니다.

```
############################## CUSTOM ADDED ################################

def create_nerf_model_executer(model_struct,
                           dataset_type,
                           lr,
                           epochs,
                           lsa,
                           N_iters,
                           learning_rate_decay,
                           i_save,
                           max_batches = False
                           ):

    handler = use_cases['NERF_PYT'] # handler : instance of
NeRFModelSetting (includes train / val / test function)

    nerf_modelexecuter = NeRFModelExecuter( handler = handler,
                                            model_struct = model_struct,
                                            dataset_type = dataset_type,
                                            lsa = lsa,
                                            lr = lr,
                                            epochs = epochs,
                                            max_batches = max_batches,
                                            N_iters = N_iters,
                                            learning_rate_decay =
learning_rate_decay,
                                            i_save = i_save
                                            )

    return nerf_modelexecuter
```

handler의 선언 부분을 조금 더 살펴보자면, 다음과 같습니다.
NeRF의 경우에는 ModelSetting이 아닌, 추가된 NeRFModelSetting을 사용합니다.

```
# Original ModelSetting Class Definition of Classification Model #
class ModelSetting:

    def __init__(self, model_transform, evaluate, train, dataset,
criterion):
```

```
        self.model_transform = model_transform
        self.evaluate = evaluate
        self.train = train
        self.dataset = dataset
        self.criterion = criterion
        self.train_cf = False

    def init_training(self, dataset_path, batch_size, num_workers):
        # ...
        return train_loader

    def init_test(self, dataset_path, batch_size, num_workers):
        # ...
        return test_set, test_loader

    def init_validation(self, dataset_path, batch_size, num_workers): # 
Only for Tensorflow
        # ...
        return val_set, val_loader

    def init_test_tef(self, dataset_path, batch_size, num_workers, 
model_name): # Only for Tensorflow
        # ...
        return test_set, test_loader

    def init_validation_tef(self, dataset_path, batch_size, num_workers, 
model_name): # Only for Tensorflow
        # ...
        return val_set, val_loader

    def preprocess( self, image, label ): # Not Used in NeRF


################################## CUSTOM ADDED ##############################
""" Added List
Class DummyModelSetting
Class DummyDataset
Class DummyDataLoader
"""

# Define a DummyDataset class that inherits from Dataset
class DummyDataset(Dataset):
    def __init__(self):
        # Initialize any necessary variables or data for the dummy dataset
        pass

    def __len__(self):
        return 1000  # Adjust the number of samples as needed

    def __getitem__(self, index):
        sample = torch.randn(3, 224, 224)  # Dummy data with shape (3, 224, 
224)
        label = torch.randint(0, 10, (1,))  # Dummy label as integer
        return sample, label
```

```
class DummyDataLoader(DataLoader):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

class NeRFModelSetting:

    def __init__(self, train):

        self.train = train # Means run_nerf.train()

    def init_training(self):
        train_set, train_loader = DummyDataset(), DummyDataLoader()
        return train_set, train_loader

    def init_validation(self):
        val_set, val_loader = DummyDataset(), DummyDataLoader()
        return val_set, val_loader

    def init_test(self):
        test_set, test_loader = DummyDataset(), DummyDataLoader()
        return test_set, test_loader
################################################################################

# supported use cases
use_cases = {
    # NNR_PyTorch
    "NNR_PYT":  ModelSetting(None,
                             evaluation.evaluate_classification_model, #
Evaluate
                             train.train_classification_model, # You can
find it in framework/applications/utils/train.py
                             imagenet.imagenet_dataloaders, # Dataset
                             torch.nn.CrossEntropyLoss() # Criterion
                             ),
    # NNR_TensorFlow
    "NNR_TEF": ModelSetting(transforms.transforms_tef_model_zoo,
                            evaluation.evaluate_classification_model_TEF,
                            None,
                            imagenet.imagenet_dataloaders,
                            torch.nn.CrossEntropyLoss
                            ),
    # CUSTOM : NeRF_PyTorch
    "NERF_PYT" : NeRFModelSetting(train = train_nerf.train_nerf_model) #
Only train_nerf_model is Used
}
```

위와 같이 NeRFModelSetting Class에서는 train_nerf_model() 함수를 제외하고는 모두 dummy인 것을 확인할 수 있습니다.

이제 본격적으로, NeRFModelExecuter Class의 구조에 대해 보겠습니다.
여기에서도 마찬가지로, tune_model (= lsa parameter 학습)을 제외한 val_model, test_model은 dummy입니다.

코드의 대략적인 흐름은 다음과 같습니다.

1. 원래의 모델을 deepcopy 하여 저장
2. tune_model()을 통해 모델의 lsa parameter를 학습
3. 학습이 완료된 모델에서 lsa parameter를 가져온 후, 원래의 모델에 덮어쓰기

```
class NeRFModelExecuter(nnc_core.nnr_model.ModelExecute):

    def __init__(self,
                 handler, # train, val, test dataset & dataloader
                 model_struct,
                 dataset_type : str,
                 lsa : bool,
                 lr : float,
                 epochs : int,
                 max_batches : int,
                 N_iters : int,
                 learning_rate_decay : float,
                 i_save : int
                 ):

        # ...

        if model_struct:
            self.original_model = copy.deepcopy(model_struct)

            if lsa:
                lsa_gen = transforms.LSA(model_struct) # create instance of
LSA class
                model_struct = lsa_gen.add_lsa_params() # add_lsa_params :
adds LSA scaling parameters linear layers

            self.model = model_struct
        else:
            raise ValueError("model_struct is not properly set!")

    def test_model(self,
                   parameters,
                   verbose=False
                   ):

        # ...

        # MOCK VERSION
        acc = self.handle.evaluate(
            model = Model,
            device = self.device
        )
```

```
        del Model
        return acc

    def eval_model(self,
                   parameters,
                   verbose=False
                   ):

        # ...

        # MOCK VERSION
        acc = self.handle.evaluate(
            Model,
            device = self.device
        )

        del Model
        return acc

    def tune_model(self, bitstream_path, parameters, param_types,
lsa_flag=True, ft_flag=False, verbose=False):

        # 원래 모델의 구조를 model_dict라는 새로운 OrderDict 형태로 저장
        torch.set_num_threads(1)
        verbose = 1 if (verbose & 1) else 0

        """ base_model_arch : Represents Initial State of self.model before
fine_tuning """
        base_model_arch = self.model.state_dict() # use model_architecture
as state_dict()
        model_dict = OrderedDict()

        for module_name in base_model_arch:

            if module_name in parameters:

                model_dict[module_name] = parameters[module_name] if
torch.is_tensor(parameters[module_name]) else \
                    torch.tensor(parameters[module_name])

                if "weight_scaling" in module_name:
                    model_dict[module_name] =
model_dict[module_name].reshape(base_model_arch[module_name].shape)

        # Load the copied model_dict into the model to update its
parameters
        self.model.load_state_dict(model_dict)

        # parameters : training (=tuning)이 완료된 LSA Parameter를 저장하는 역할
        for param in parameters: # iterate key of parameter(dict)
            parameters[param] = copy.deepcopy(self.model.state_dict()
[param])

        # Optimizer에게 넘겨주기 위한 LSA Parameter들을 저장하는 List
```

```
        tuning_params = []

        """ param_types May Looks Like :

            param_types = {
                'module_name1.weight': 'weight.ls',  # Parameter name and
its type
                'module_name1.bias': 'bias',         # Another parameter
name and its type
                'module_name2.weight': 'weight.conv',
                'module_name2.bias': 'bias',
                # Add more parameter names and their corresponding types
here
            }
        """

        for name, param in self.model.named_parameters():

            # O_TYPES =
["weight.ls","bias","bn.beta","bn.gamma","bn.mean","bn.var","unspecified"]

            if lsa_flag and ft_flag and param_types[name] in
nnc_core.nnr_model.O_TYPES:
                param.requires_grad = True
                tuning_params.append(param)
            elif lsa_flag and param_types[name] == 'weight.ls': # LSA만 사용
하는 경우, 여기에 해당
                # <=> name : ~~~~~.weight_scaling
                # ~~~~.weight_scaling에 해당하는 parameter에 대해서만
requires_grad = True (다른 parameter들은 fixed)
                # self.model.named_parameters()에 model과 model_fine이 모두
포함되어 있으므로,
                # model, model_fine의 lsa parameter들에 대해 모두 학습이 가능해집
니다.
                param.requires_grad = True
                tuning_params.append(param)
            elif ft_flag and param_types[name] != 'weight.ls' and
param_types[name] in nnc_core.nnr_model.O_TYPES:
                param.requires_grad = True
                tuning_params.append(param)
            else:
                param.requires_grad = False


        ########################################## CUSTOM MODIFIED
################################################
        self.model.global_step = 0
        self.model.tuning_optimizer = torch.optim.Adam(tuning_params,
lr=self.learning_rate)

        if self.learning_rate_decay != 0:
            scheduler =
torch.optim.lr_scheduler.StepLR(optimizer=self.model.tuning_optimizer,
                                                        step_size=1,
```

```
gamma=self.learning_rate_decay)

        for e in range(self.epochs):

            # self.handle.train는 utils/train_nerf.py를 의미하며,
            # train_nerf.py는 dataset_type에 따라 적절한 config를 통해
run_nerf.train()을 호출합니다.
            train_acc, loss = self.handle.train(
                nerf_wrapper = self.model,
                dataset_type = self.dataset_type,
                freeze_batch_norm = True,
                basedir_save =
os.path.dirname(os.path.dirname(bitstream_path)),
                N_iters = self.N_iters,
                i_save = self.i_save
            )

            if self.learning_rate_decay != 0:
                scheduler.step() # Learning rate decay to 0.1 every epoch
(Every N_iter times of iterations)


            print(f'Epoch {e+1} done. Train accuracy: {train_acc}, Loss:
{loss}')

            """
            IN THE CASE OF NERF MODEL, THERE IS NO PROCESS OF EARLY-
STOPPING
            """

            for param in parameters:
                parameters[param] = copy.deepcopy(self.model.state_dict()
[param])

            best_params = copy.deepcopy(parameters)


##############################################################################
#########################

        lsa_params, ft_params = {}, {}

        for name in best_params:
            if lsa_flag and param_types[name] == 'weight.ls':
                lsa_params[name] =
best_params[name].cpu().numpy().flatten()
            elif ft_flag and param_types[name] != 'weight.ls' and
param_types[name] in nnc_core.nnr_model.O_TYPES:
                ft_params[name] = best_params[name].cpu().numpy()

        return (lsa_params, ft_params)

    def has_eval(self):
```

```
        return True

    def has_test(self):
        return True

    def has_tune_ft(self):
        return True

    def has_tune_lsa(self):
        return True
```

NeRF 학습과 관련된 모든 코드는 framework/nerf_model에 통합되어 있습니다. classification model의 학습과는 달리 학습에 필요한 코드의 양이 많아, pytorch_model과는 분리된 별도의 폴더를 만들었습니다.

run_nerf.train()는 NeRF 학습의 메인 함수로, NNCodec의 흐름에 맞도록 여러 부분들이 변경되어 있습니다.

1. run_nerf.train()은 NeRF 학습의 main 함수에 해당합니다. 원래는 .sh파일에서 argument를 받아 실행되도록 되어 있으나, NNCodec에서는 NNCodec 과정에서의 한 부분으로 속해 있으므로 argument가 아닌 parameter만을 받아 동작하도록 변경되었습니다.
2. optimizer나 step에서 nerf_wrapper의 것을 사용합니다.
3. precrop_iter 변수가 0으로 통일되어 있습니다 (학습 초기 단계에서의 안정성을 위한 방법이지만, LSA tuning의 경우에는 이미 어느정도 학습된 모델에 대해 적용하는 상황이므로 해당 기법을 사용하지 않는 편이 더 낫다는 판단에 의해서입니다).
4. i_save, i_video 등의 변수들이 i_save 하나로 통일되었습니다.

```
def train(nerf_wrapper, basedir_save, basedir, datadir, i_save, N_iters,
          N_rand=32*32*4, chunk=1024*32, netchunk=1024*64,
no_batching=False,
          N_samples=64, N_importance=0, perturb=1., use_viewdirs=False,
i_embed=0,
          multires=10, multires_views=4, raw_noise_std=0.,
          render_only=False, render_test=False, render_factor=0,
          precrop_iters=0, precrop_frac=0.5, dataset_type='llff',
testskip=8,
          shape='greek', white_bkgd=False, half_res=False, factor=8,
          no_ndc=False, lindisp=False, spherify=False, llffhold=8):

        # ...

        for i in tqdm(range(0, N_iters)):

            nerf_wrapper.global_step += 1

            # ...

            loss.backward()
            nerf_wrapper.tuning_optimizer.step()
            nerf_wrapper.tuning_optimizer.zero_grad()
```

마지막으로 framework/applications/utils/transforms.py 의 ScaledLinear.forward() 함수가 수정되어 있습니다.
해당 함수는 기존 모델에 존재하는 nn.Linear 모듈에 LSA parameter를 추가하여 ScaledLinear라는 custom class로 변환하는 역할을 합니다.
이때 기존 코드는 forward() 함수가 호출될 때 weight, bias, weight_scaling parameter를 nn.Parameter의 instance로 선언하는데, NeRF에서 model forward를 여러번 호출하는 경우 각 parameter들이 nn.Parameter로 여러 번 wrapping되어 학습이 제대로 되지 않는 문제가 발생합니다. (nn.Parameter(nn.Parameter(...)) 처럼 되어버리는 문제).
따라서 해당 부분을 주석처리하였습니다.

```
# Inner class of LSA.update_linear
class ScaledLinear(nn.Linear):
    def __init__(self, in_features, out_features, *args, **kwargs):
        super().__init__(in_features, out_features, *args, **kwargs)

        """ < weight_scaling >
        Initialize weight parameter which sized out_features x 1
        Which means, there is one weight_scaling parameter per each output
node.
        i.e. if nn.Linear has 10 inputs and 5 outputs, then there are 5
weight_scaling parameters
        which mean, each 10 weights share single weight_scaling parameter.
        """
        self.weight_scaling =
nn.Parameter(torch.ones_like(torch.Tensor(out_features, 1)))
        self.reset_parameters()

    def reset_parameters(self):
        # The if condition is added so that the super call in init does not
reset_parameters as well.
            if hasattr(self, 'weight_scaling'):
                nn.init.normal_(self.weight_scaling, 1, 1e-5)
                super().reset_parameters()

    # nn.Linear automatically returns result of forward() when called
    def forward(self, input):

        # self.weight = nn.Parameter(self.weight)
        # self.bias = nn.Parameter(self.bias)
        # self.weight_scaling = nn.Parameter(self.weight_scaling)

        return F.linear(input, self.weight_scaling * self.weight,
self.bias)
```