SR for CT images of Batteries with Deep Learning

Release 0.1

Haorui Long

Oct 04, 2023

CONTENTS:

This project is about **Super Resolution for CT image of batteries with Deep Learning**. The goal is to improve the resolution of the CT image of batteries. The model from project is based on UNet from paper *U-Net: Convolutional Networks for Biomedical Image Segmentation* <https://arxiv.org/abs/1505.04597>. The model is trained on the dataset from paper *Multimodal Nanoscale Tomographic Imaging for Battery Electrodes* <https://doi.org/10.1002/aenm.201904119>.

CHAPTER

ONE

USAGE

Here is a simple guide to use this project.

1.1 Installation and depandencies

This project is based on Python 3.11. To install the depandencies, you can use the following command:

```
$ pip install -r requirements.txt
```

1.2 Start

To start running the code, you can use the following command:

```
$ python main.py
```

1.3 Configuration

In this project, a configuration file is used to store the parameters of the code. The configuration file is a '.yaml' file, and it is located in the 'configs' folder. The configuration file is divided into different section, and each section contains the parameters of a specific part of code.

split_ratios
: Split ratios of dataset

train
: Training configs

dataset
: Dataset configs

model
: Model configs

pred
: Inference configs

You can change the parameters in the configuration file to change the behavior of the code. Here is an example to change the train epoch.

```
$ python main.py ++train.epochs=50
```

This will change the train epochs from 10 to 50.

CHAPTER

TWO

DATA

2.1 Download

The data can be downloaded from the following link: <https://www.research-collection.ethz.ch/handle/20.500.11850/505938>. And the data is unzipped in the folder 'dataset/'. If there is no such folder, please create one and unzip the data in it.

Create the folder:

```
$ mkdir dataset
```

2.2 Dataset

To get the dataset for model, you can use the `BaseDataset` class.

class source.BaseDataset.**BaseDataset**(*task*, *mode*, *tile_size*, *tile_grid*, *dataset_dir: List[str] | str*, *data_root: str = ''*, *transforms_cfg: DictConfig = None*, *preprocess_cfg: DictConfig = None*)

Bases: `Dataset`

check_dataset_constrains()

To check whether the dataset meets the constrains

get_item_position(*idx*)

To get each tile's information including, which tif file it belongs to, which page it belongs to, which tile in the page it is, and the tiling grid information.

Parameter:
idx : the index of the tile in the dataset

Return:
tifffile_index: the index of the tif file in the dataset page_index: the index of the page in the tif file sequence_number_of_tile_in_page: the sequence number of the tile in the page tiling_grid_info (tuple[int, int]): the tiling grid information

get_pages_count_in_tile(*tiff_file*)

To get the total number of pages in a tif file

Parameter:
tiff_file (str): the path of the tif file

Return:
num_pages: the total number of pages in the tif file

static get_preprocessor(*preprocess_cfg: DictConfig*) → *Preprocessor* | None

Returns a Preprocessor object based on the provided preprocessing configuration.

Args:
preprocess_cfg (DictConfig): A configuration object that specifies the details of the preprocessing pipeline.

Returns:
Preprocessor: A Preprocessor object that can be used to preprocess the data.

Notes:
This method creates a Preprocessor object based on the provided configuration. If no configuration is provided (i.e., if preprocess_cfg is None), None is returned.

get_tile_from_index(*index*)

To get tile from the dataset by the index of the tile

Parameter:
index (int): the index of the tile in the dataset

Return:
tile (tensor): the tile in the dataset

get_tiles_count_in_each_tiff(*tiff_file*)

To get the total number of tiles in a tif file

Parameter:
tiff_file (str): the path of the tif file

Return:
tiles_count_in_each_tiff: the total number of tiles in the tif file

get_tiling_grid_in_each_tiff(*tiff_file*)

To get the tiling grid information in each tif file

Parameter:
tiff_file (str): the path of the tif file

Return:
tiling_grid_info (tuple[int, int]): the tiling grid information in each tif file

static get_transforms(*transforms_cfg: DictConfig*) → *Transforms* | None

Returns a Transform object based on the provided configuration.

Args:
transforms_cfg (DictConfig): The configuration for the transforms.

Returns:
Transforms: The Transform object.

Notes:
This method creates a Transforms object based on the provided configuration. If no configuration is provided (i.e., if transforms_cfg is None), None is returned.

static load_file_paths_from_dir(*data_root: str*, *dataset_dir: List[str]*, *recursive: bool = True*) → List[Path]

Recursively loads all file paths from the given directory or directories.

Args:
data_root (str): The root directory to which the dataset directory paths are relative. dataset_dir (List[str]): A list of directory paths to load file paths from. recursive (bool, optional): Whether to recursively load file paths from all subdirectories. Defaults to True.

Returns:
List[Path]: A sorted list of all file paths found in the given directories.

Raises:
FileNotFoundError: If any of the given directories are not found.

2.3 Preprocessing

To get the preprocess for dataset, you can use the `Preprocessor` class.

`class` preprocess.Preprocessor.**Preprocessor**(*preprocess_cfg: DictConfig*)

Bases: `object`

static binning(*image: Tensor*, *binning_factor: int*) → Tensor

To get binned image, we first downsample the image by a factor of binning_factor using average pooling, and then upsample it to the original size using nearest neighbor interpolation.

Args:
image (torch.Tensor): Input image tensor. binning_factor (int): Binning factor.

Returns:
torch.Tensor: Binned image tensor.

static convert_dtype(*image: Tensor*) → Tensor

Convert the image tensor to the right dytpe. :param image: :return: image

static gamma_correction(*image: Tensor*) → Tensor

static normalize(*image: Tensor*, *mean: list*, *std: list*) → Tensor

Normalize the image tensor with given mean and standard deviation values.

static rescale(*image: Tensor*, *min_val: int*, *max_val: int*) → Tensor

Parameters

- **image** –
- **min_val** –
- **max_val** –

Returns
rescaled image

static resize(*image: Tensor*, *size: list[int, int]*, *padding=True*) → Tensor

Resize a torch.Tensor image to the specified size, while padding it with zeros to make it square, using Kornia library.

Args:
image (torch.Tensor): Input image tensor. size (list[int, int]): Desired output size.

Returns:
torch.Tensor: Resized and padded image tensor.

static to_grayscale(*image: Tensor*) → Tensor

Convert the image tensor to grayscale.

2.4 Transform

To get the transform for dataset, you can use the `Transform` class.

class transform.Transforms.**Transforms**(*transforms_cfg: DictConfig*)

Bases: object

Apply a set of custom transforms on the input image

Example:
transform_dict = { 'rotate': {'angle': 30}, 'adjust_brightness': {'brightness_factor': 0.5} }

static RandomAffine(*param*)

Rotate the given tensor image by angle.

Args:
degrees (float or int): Rotation angle in degrees. etc.

Returns:
torch.Tensor: Rotated tensor image.

static RandomBrightness(*param*)

Adjust the brightness of the given tensor image.

Args:
image (torch.Tensor): Tensor image to be adjusted. brightness_factor (float): Brightness adjustment factor. Must be positive.

Returns:
torch.Tensor: Brightness-adjusted tensor image.

static RandomContrast(*param*)

Adjust the contrast of the given tensor image.

Args:
image (torch.Tensor): Tensor image to be adjusted. contrast_factor (float): Contrast adjustment factor. Must be positive.

Returns:
torch.Tensor: Contrast-adjusted tensor image.

static RandomCrop(*param*)

Crop the given tensor image at a random location.

Args:
image (torch.Tensor): Tensor image to be cropped. output_size (tuple): Expected output size (height, width) of the crop.

Returns:
torch.Tensor: Cropped tensor image.

static RandomGaussianBlur(*param*)

Applies a random Gaussian blur to the given tensor image.

Args:
image (torch.Tensor): Tensor image to be blurred. kernel_size_range (tuple): Range of kernel sizes to randomly sample from. sigma_range (tuple): Range of sigma values to randomly sample from.

Returns:
torch.Tensor: Blurred tensor image.

static RandomGaussianNoise(*param*)

Adds random Gaussian noise to the given tensor image. If values are out of range, the tensor is clipped.

Args:
image (torch.Tensor): Tensor image to be noised. mean (float): Mean of the Gaussian noise distribution. std (float): Standard deviation of the Gaussian noise distribution.

Returns:
torch.Tensor: Noised tensor image.

static RandomHorizontalFlip(*param*)

Horizontally flip the given tensor image.

Args:
image (torch.Tensor): Tensor image to be flipped.

Returns:
torch.Tensor: Horizontally flipped tensor image.

static RandomHue(*param*)

Adjust the hue of the given tensor image.

Args:
image (torch.Tensor): Tensor image to be adjusted. hue_factor (float): Hue adjustment factor.

Returns:
torch.Tensor: Hue-adjusted tensor image.

static RandomMedianBlur(*param*)

Applies a random median blur to the given tensor image.

Args:
image (torch.Tensor): Tensor image to be blurred. kernel_size_range (tuple): Range of kernel sizes to randomly sample from.

Returns:
torch.Tensor: Blurred tensor image.

static RandomSaturation(*param*)

Adjust the saturation of the given tensor image.

Args:
image (torch.Tensor): Tensor image to be adjusted. saturation_factor (float): Saturation adjustment factor. Must be positive.

Returns:
torch.Tensor: Saturation-adjusted tensor image.

static RandomVerticalFlip(*param*)

Vertically flip the given tensor image.

Args:
image (torch.Tensor): Tensor image to be flipped.

Returns:
torch.Tensor: Vertically flipped tensor image.

CHAPTER

THREE

MODEL

This module contains the implementation of the U-Net model.

3.1 ConvBlock

In this subsection, class `ConvBlock` is a building block of the U-Net model.

class model.unet.blocks.**ConvBlock**(*in_channels: int*, *out_channels: int*, *n_layers: int = 2*, *scale_factor: int = 2*, *kernel_size: int = 3*, *n_groups: int = 32*)

Bases: `Module`

A block of convolutional layers followed by max pooling, group normalization and activation function.

Args:

in_channels (int): Number of input channels. out_channels (int): Number of output channels. n_layers (int): Number of convolutional layers. scale_factor (int): Factor by which to downsample the input. kernel_size (int): Size of the convolutional kernel. n_groups (int): Number of groups to use for group normalization.

Returns:

torch.Tensor: The output tensor.

forward(*x: Tensor*)

Output tensor of the convolutional block.

3.2 SkipConnection

In this subsection, class `SkipConnection` builds a skip connection in the U-Net model.

class model.unet.blocks.**SkipConnection**(*channels*, *scale_factor: int = 2*, *n_groups: int = 32*)

Bases: `Module`

A block that performs a skip connection by max pooling the input tensor and applying group normalization and activation function to it.

Args:

channels (int): Number of input channels. scale_factor (int): Factor by which to downsample the input. n_groups (int): Number of groups to use for group normalization.

Returns:

Tuple[torch.Tensor, torch.Tensor]: A tuple containing the output tensor and the intermediate tensor.

forward(*x: Tensor*)

Skip connection forward pass. Returns a tuple containing the output tensor and the intermediate tensor.

3.3 CombineConnection

In this subsection, class `CombineConnection` combines skip connection in the U-Net model.

class model.unet.blocks.**CombineConnection**(*channels*, *scale_factor: int = 2*, *n_groups: int = 32*)

Bases: `Module`

A block that combines the skip connection and residual block by performing a residual connection between the input tensor and the output of the residual block.

Args:

in_channels (int): Number of input channels. out_channels (int): Number of output channels. scale_factor (int): Factor by which to upsample the input. n_groups (int): Number of groups to use for group normalization. dropout (float): Dropout probability.

Returns:

torch.Tensor: The output tensor.

forward(*x: Tensor*, *x_e: Tensor*)

Combine connection forward pass. Element wise addition of two same spatial dimension tensor.

3.4 ConvUNet

In this subsection, class `ConvUNet` builds a U-Net model.

class model.unet.ConvUNet.**ConvUNet**(*model_cfg: DictConfig*, *image_channels: int = 1*, *output_channels: int = 1*, *c_factor: int = 6*, *ch_mults: Tuple[int, ...] | List[int] = (1, 2, 3, 4)*, *n_blocks: int = 2*, *n_layers: int = 2*, *scale_factor: int = 2*, *kernel_size: int = 3*, *n_groups: int = 32*, *verbose: bool = True*)

Bases: `LightningModule`

A simple Conv U-Net.

Args:

model_cfg (DictConfig): The configuration dictionary for the model. image_channels (int): The number of channels in the input image. output_channels (int): The number of channels in the output image. c_factor (int, optional): The channel factor. Defaults to 6. ch_mults (Union[Tuple[int, ...], List[int]], optional): The channel multipliers. Defaults to (1, 2, 3, 4). n_blocks (int, optional): The number of blocks. Defaults to 2. n_layers (int, optional): The number of layers. Defaults to 2. scale_factor (int, optional): The scale factor. Defaults to 2. kernel_size (int, optional): The kernel size. Defaults to 3. n_groups (int, optional): The number of groups in GroupNorm. Defaults to 32. verbose (bool, optional): Whether to print verbose logs. Defaults to True.

Attributes:

image_proj (nn.Conv2d): The image projection layer. down (nn.ModuleList): The down-sampling layers. middle (ConvBlock): The middle layer. up (nn.ModuleList): The up-sampling layers. norm (nn.GroupNorm): The group normalization layer. act (Swish): The activation function. final (nn.Conv2d): The final convolutional layer. sig (nn.Sigmoid): The sigmoid activation function.

configure_optimizers() → Tuple[Optimizer | Sequence[Optimizer], Scheduler | Sequence[Scheduler]]

Configure the optimizer and learning rate scheduler for the Lightning module.

Returns:

A tuple containing one or two elements, depending on whether a learning rate scheduler is used or not. The first element is an optimizer or a list of optimizers. The second element is a learning rate scheduler or a list of learning rate schedulers.

forward(*x: Tensor*)

The forward method of the ConvUNet class defines the forward pass of the convolutional U-Net model. It takes an input tensor x and passes it through the layers of the model to produce an output tensor.

The method starts by passing the input tensor through the image projection layer. It then performs downsampling using skip and residual connections, saving the output tensor from each skip connection in a list h for later combination.

After the downsampling layers, the method passes the output tensor through the middle layer of the model. It then performs upsampling using the saved skip connection tensors and residual connections, combining them with the output tensor from each upsampling layer.

Finally, the method passes the output tensor through the final layer of the model, applies an activation function, and returns the resulting tensor.

get_loss(*y_hat*, *y*)

To get the loss from the loss function

Returns:

torch.Tensor: The loss for the current step.

predict_step(*batch*, *batch_idx*, *dataloader_idx: int = 0*)

Prediction for the lightning module.

Args:

batch (torch.Tensor): The input batch. batch_idx (int): The index of the batch. dataloader_idx (int): The index of the dataloader.

Returns:

torch.Tensor: The prediction of current batch.

classmethod restore(*ckpt: str*, *map_location: device | str | int = 'mps'*)

Restores a PyTorch Lightning model from a checkpoint file.

Args:

cls (LightningModule): The class of the PyTorch Lightning model to restore. ckpt (str): The path to the checkpoint file to restore from. map_location (Union[torch.device, str, int]): Device to map the restored model to.

Returns:

Tuple[LightningModule, DictConfig]: A tuple of the restored model and its configuration.

Raises:

RuntimeError: If the checkpoint file does not contain hyperparameters.

Example:

Restore a PLModel from a checkpoint file model, config = PLModel.restore(ckpt='path/to/checkpoint.ckpt')

save_model(*model_name*, *save_dir*)

Save a PyTorch model to a given directory.

Args:
model (torch.nn.Module): The PyTorch model to save. save_dir (str): The directory to save the model to.

`test_model`(*input_data*)

`training_step`(*batch*, *batch_idx*)

Training step for the lightning module.

Args:
batch (torch.Tensor): The input batch. batch_idx (int): The index of the batch.

Returns:
torch.Tensor: The loss for the current training step.

`validation_step`(*batch*, *batch_idx*)

Validation step for the lightning module.

Args:
batch (torch.Tensor): The input batch. batch_idx (int): The index of the batch.

Returns:
torch.Tensor: The loss for the current validation step.

CHAPTER

FOUR

TRAIN

To use configuration file to start training model.

4.1 Training configuration

Here is an example of training configuration file:

train.**custom_collate**(*batch*)

Custom collate function to drop none informative tiles on edge to prevent model params pollution

train.**init_model**(*cfg: DictConfig*)

Create model using parameters from config file.

Parameters:
cfg: DictConfig

Returns:
model: ConvUNet

train.**setup_dataloaders**(*cfg: DictConfig*)

Create dataloaders using parameters from config file.

Parameters:
cfg: DictConfig

Returns:
train_dataloader: DataLoader. validation_dataloader: DataLoader. test_dataloader: DataLoader.

train.**setup_logger**(*cfg: DictConfig*)

Create log to visualize training process.

Parameters:
cfg: DictConfig

Returns:
logger: TensorBoardLogger

train.**start_training**(*cfg: DictConfig*)

Training process.

Parameters:
cfg: DictConfig

CHAPTER

FIVE

PREDICTION

This section introduce how to predict from the trained model.

5.1 Predicting configuration

Here is the function to set the configuration for prediction.

pred.**test_slice**(*cfg: DictConfig*)

To get specific slice from test set. And applying it with preprocessing and transforms in order to get the input and label.

Parameters:
cfg: DictConfig

Returns:
data: torch.Tensor label: torch.Tensor

pred.**setup_test_set**(*cfg: DictConfig*)

To get the test set from the dataset. Using dataset splitting ratio.

Returns:
test_set: torch.utils.data.Dataset

pred.**setup_input_label**(*cfg: DictConfig*)

To set up the input and label from the test set for thesis used.

Returns:
input: torch.Tensor label: torch.Tensor

pred.**splitting**(*image: Tensor*, *cfg: DictConfig*)

To split image into specific size of tiles.

Parameters:
image: torch.Tensor cfg: DictConfig

Returns:
tiles: list

pred.**load_model**(*cfg: DictConfig*)

Load model from checkpoint.

Returns:
model: ConvUNet

pred.**prediction**(*model*, *tile*)

Predicting image from trained model.

Returns:
output: torch.Tensor

pred.**reassemble**(*tiles: list*, *cfg: DictConfig*)

To reassemble the tiles into a image, which is one of the slice in the test set.

Parameters:
tiles: list cfg: DictConfig

Returns:
reassembled_image: torch.Tensor

pred.**inference**(*cfg: DictConfig*)

Start inference.

pred.**calc_average_psnr_in_testset**(*cfg: DictConfig*)

To calculate the average PSNR in the test set.

Returns:
average_psnr: float

pred.**convert_iter_to_epoch**(*data: list*, *epoch_size: int*)

In loss curves, we want to show the loss in each epoch. But the loss in log is save in each iteration. So we need to convert the loss in iteration to loss in epoch.

Parameters:
data: list epoch_size: int

Returns:
averages: list

pred.**get_train_loss_val_loss**(*cfg: DictConfig*)

Load the train loss and validation loss from csv file.

Returns:
train_loss_values: list iter: list val_loss_values: list val_iter: list

pred.**plot_loss_in_iter**(*cfg: DictConfig*)

To plot the loss curve in iteration.

pred.**plot_loss_in_epoch**(*cfg: DictConfig*)

To plot the loss curve in epoch.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

R

S

T

V