Pipeline Documentation

Overview

The pipeline was delivered as part of the S2DS Team CGG project and aims to provide a framework for enabling the development, training, inference and evaluation of deep learning super resolution models. A virtual environment (see requirements.txt) needs to be activated before anything can be performed.

The pipeline can be run in three separate modes. These modes are set by passing in the argument “train”, “inference” or “evaluate” into the app.py script. The configuration file is defined in the github repo, can can be altered as desired.

```
Python app.py --mode train          -- config ./path_to_config_file.txt

Python app.py --mode inference      -- config ./path_to_config_file.txt

Python app.py --mode evaluate       -- config ./path_to_config_file.txt
```

Configuration file

The configuration file is where all the necessary parameters for training/inference/evaluation are set. There are three flags which correspond to each mode. Below is the config file with **all** possible variables filled in. Some of these are optional, meaning they can be deleted from the configuration file. Comments for each parameter are given in green.

[TRAINING]

Path to folder containing low resolution training data
```
low_res_dir = data/path_to_my_low_res_training_data/
```

Path to folder containing low resolution training data
```
ground_truth_dir = data/path_to_my_high_res_training_data/
```

Tell pipeline whether or not to create patches from images, needed for some models such as SRCNN
```
create_patches = false
```

OPTIONAL: If training a model requiring patches of images, give the path to existing high-res patches folder or create folder with the given path name
```
low_res_patches_path = data/path_to_low_res_training_data_as_patches/
```

OPTIONAL: If training a model requiring patches of images, give the path to existing low-res patches folder or create a folder with the given path name
```
high_res_patches_path = data/path_to_low_res_training_data_as_patches/
```

OPTIONAL: if patching images, describe how much smaller the patches should be
```
lr_patches_down_scale = 4
```

Random seed for the separation of the training data into train + validation. Keep the same to compare two models (they will train on the same data)
```
random_seed = 100
```

The proportion of the total files in the training directory to be used for training (e.g. 0.7 = 70%). The remainder go into validation.
```
train_split = 0.7
```

Flag (either true or false - case sensitive). If true, images are normalised using the mean and standard deviation in the training set. The output is saved in the model folder as a JSON.
```
standardisation = true
```

Flag (either true or false - case sensitive). If true, images are flipped randomly in the horizontal axis during training.

```
flip_horizontal = true
```

Flag (either true or false - case sensitive). If true, images are flipped randomly in the vertical axis during training.

```
flip_vertical = false
```

OPTIONAL: Padding for the low-res training images, given as an array [C, H, W] (pytorch convention).

```
pad_training = [3, 107, 107]
```

OPTIONAL: Padding for the high-res training images, given as an array [C, H, W] (pytorch convention).

```
pad_target = [3, 1070, 1070]
```

Model name: i.e. SimpleModel, SRCNN, LapSRN,

```
model = SimpleModel
```

Name of the optimiser to be used: i.e. adam

```
optimizer = adam
```

Learning rate (float) to be set for the optimiser

```
learning_rate = 0.001
```

Number of epochs to train model for

```
num_epochs = 3
```

Number of samples to be loaded into the GPU as a batch

```
batch_size = 10
```

Loss function to be used during the training option (i.e. mse, charb)

```
loss = mse
```

[INFERENCE]

Path to directory where trained model.pth sits in

```
model_dir = models/example_model_dir/
```

OPTIONAL: input directory when performing inference on a new or unseen set. If this is not included, the model will automatically compute inference on the validation set

```
low_res_dir = data/path_to_folder_with_unseen_data/
```

[EVALUATE]

Path to directory where the trained model.pth sits in

```
model_dir = models/example_model_dir/
```

Model training

Takes in data and parameters from a configuration file in order to train a selected model. Saves a folder under the **models/** directory in the main repository. For example, if training a LapSRN model:

- models/
    - LapSRN_datetime/
        - Model.pth
        - Standardisation_rgb_stats.json (if standardisation = true in config)
        - Train_val_split.json
        - Training_config.json
        - tensorboard/
            - Tensorboard event file

The dictionary of the model weights is saved as model.pth. The filenames allocated to training and validation are found in the train_val_split.json. The training_config.json contains the parameters used in the training process for that model. The tensorboard folder contains the event files which show training and validation loss.

Model inference

Takes in the model directory (as essential). Performs inference using the specified model.pth on the validation set, if low_res_dir is not given. Creates a folder and saves results. For example, after performing inference (without low_res_dir), and after performing inference (with low_res_dir=my_unseen_data/), the folder structure should appear as follows:

- Models
    - LapSRN/
        - Model.pth
        - Standardisation_rgb_stats.json (if standardisation = true in config)
        - Train_val_split.json
        - Training_config.json
        - tensorboard/
            - Tensorboard event file
        - **inference_my_validation_data/**
        - **inference_my_unseen_data/**

Model evaluation

Requires only model_dir as a parameter in the config. Currently functions by doing a direct comparison of images in the validation inference set (prediction), and in the high-res inference folder (ground truth). **Note:** *Will not work if the user changes the paths of either the validation inference folder, or the training data folder.* In the previous inference example, running evaluation would read the images from the inference_my_validation_data/ directory and compare them against the original GT images in the training directory. The function computes mean squared error, structural similarity index and peak signal to noise ratios for this data, saving results in an errors_validation_set.json.

- Models
    - LapSRN/
        - Model.pth
        - Standardisation_rgb_stats.json (if standardisation = true in config)
        - Train_val_split.json
        - Training_config.json
        - tensorboard/
            - Tensorboard event file
        - inference_my_training_and_validation_data/
        - inference_my_unseen_data/
        - **errors_validation_set.json**

Source Code

src/data/datasets.py

Contains custom dataset classes for the pipeline

- SRDataset(Dataset): Creates a super resolution dataset compatible with torch dataloaders
- New custom classes should inherit torch.utils.data import Dataset and override the following methods:
    - __len__ so that len(dataset) returns the size of the dataset.
    - __getitem__ to support the indexing such that dataset[i] can be used to get ith sample.

src/data/transforms.py

Contains custom transformation classes to be used in composed transforms list

- Pad(): Applies padding to a torch tensor

src/models/models.py

Contains the different model architecture classes and custom loss classes:

- SimpleModel(): dummy model for testing pipeline with chosen scale_factor
- SRCNN(), LapSRN(), CharbonnierLoss(), SRGAN related classes

src/trainer/trainer.py

Contains the code for training a model:

- Contains several functions which return the relevant object given information from the config: get_model_class(), get_loss_function(), get_optimizer(), get_metric()
- train_loop(): training loop for a CNN network
- evaluate(): computes metric_fn between batch of model outputs and targets
- model_predict(): function for performing inference and computes model errors
- run_model(): helper function for performing inference - runs on the GPU
- train_loop_gan(): training loop for a GAN architecture *(in development)*

src/utils/np_utils.py

Contains helper functions mainly for preprocessing (not in pytorch functions such as training)

- load_image_as_np(): loads an image from path as a numpy array (H, W, C)
- normalise_image(): divides all pixels in an image by 255 (normalises to 0-1)
- pad_image(): adds padding to an image to get it to a desired size
- generate_low_resolution_image(): scales an image given a scale factor
- resize_image(): resize image to a target size (with interpolation)
- standardise_image(): normalise image using mean and standard deviation
- reverse_image_standardisation(): return image to original scale with mean and stdev
- compute_stats_channel_dim(): computes mean and standard deviation for rgb channels in a batch of images (by concat each image)
- compute_percentage_green(): calculate percentage of image which contains green pixels

- compute_percentage_edges(): compute percentage of image which contains Canny edges (binary mask of edges)
- create_patches(): compute and save patches of images for a given size in a new folder, needed for some models such as SRCNN
- crop_img(): perform center-crop on an image

src/utils/crop_and_downsample.py

Executable python script to batch crop and downsample ground truth (gt) images according to scale factor

- When running, will be given the following prompts:
  ```
  Enter path to gt dir e.g. data/raw/CGG_data/train/gt:
  Enter path to output dir e.g. data/processed/CGG_data/train:
  Enter downsampling scaling factor (e.g. 2 for 2x downscaling):
  Do you want to save cropped gt? (y or n):
  ```
  - For this prompt, if cropped gt images have already been created then enter 'n', otherwise the cropped gt will be overwritten.
- Uses the following functions from np_utils.py:
  - load_image_as_np()
  - crop_img()
  - generate_low_resolution_image()

src/utils/downsampler.py

Executable python script to downsample images for a given scale factor (same as above, without cropping)
Note: imports package "readline" which may not work on non-linux OS. Lines 10 and 11 related to package readline can be freely removed.

-
  ```
  Enter path to gt dir e.g. data/raw/CGG_data/train/gt:
  Enter path to output dir e.g. data/processed/CGG_data/train:
  Enter downsampling scaling factor (e.g. 2 for 2x downscaling):
  ```
- Uses the following functions from np_utils.py:
  - generate_low_resolution_image()

src/utils/preprocessing_pipeline.py

Executable python script to preprocess images, determine the train validation test split and compute mean + std on the training set. Saves a json of the data. Is not required anymore in the main app.py pipeline, as these steps are performed within the train() function (in app.py).

src/utils/torch_utils.py

Contains helper functions mainly pytorch tensors

- compute_psnr(): computes psnr between tensor image and reference
- pad_image(): pads a tensor image to desired size
- resize_image(): resizes a tensor image to desired size
- standardise_image(): standardize tensor image - subtracts given population mean and divides by given standard deviation
- reverse_image_standardisation():reverses standardization of given tensor image by multiplying by population mean and adding standard deviation
- compose_transforms_dict(): generates dictionary of transforms to be applied to the SRDataset pytorch class

src/test.py

Contains helper functions for the evaluate() loop in app.py.

- compute_mse_errors(): returns a dictionary of stats related to the mse errors between the validation prediction/ground truth set.
- compute_psnr_errors(): returns a dictionary of stats related to the psnr errors between the validation prediction/ground truth set.
- compute_ssim_errors(): returns a dictionary of stats related to the ssim errors between the validation prediction/ground truth set.

Experiments:

LapSRN: https://arxiv.org/abs/1704.03915

- We implemented the architecture class and Charbonnier loss class from https://github.com/Lornatang/LapSRN-PyTorch in models.py
- We used standardization for LapSRN model runs
- Used src/utils/crop_and_downsample.py to generate training data:
  - Generated 1156 center cropped ground truth (GT) images of size 1024x1024
  - Discarded 72 images with size <1024
  - Generated 8x downsampled images from these cropped GT images
- We trained this model on 80% of the training data with random augmentation (flips) - where 20% was kept for computing validation metrics which are included in tensorboard data
- We experimented with different learning rates:
  - 0.001 - produced caused loss to go to infinity
  - 0.0001 - produced good results (this is used for both trained models sent over)
- Specific model runs are documented in the config file inside the trained model folder
- We trained 2 model runs using:
  - Charbonnier loss (inside LapSRN_31_03_2023_2022 folder):

  "mean_mse": 56.78293196360271, "mean_psnr": 23.686205486678986, "mean_ssim": 0.8024150890955879
  - MSE loss (inside LapSRN_01_04_2023_0838 folder)

  "mean_mse": 57.354437703671664, "mean_psnr": 23.727110166192602, "mean_ssim": 0.8016701846337162
- The saved trained models were run for 200 epochs
  - Qualitatively we found that the network achieved good results by epoch 80
  - But more detail is retrieved when trained for at least epoch 150
  - Results converge around 200 but decrease slightly after
  - When trained for 240, there was some artifacting in the sentinel testing inference despite the fact that the loss continued to have a slight decrease

SRCNN: https://arxiv.org/pdf/1501.00092.pdf

- We created low res (4x) and high res patches (300 X 300) of the ground truth images with a stride of 250. This resulted in ~10000 patches which are used as input to the network. The low res patches are again upsampled to match the size of high res patches using bicubic interpolation.

- We need to make sure the LR images need to be same size as SR images only for this network
- Standardization was not used for this model
- We trained this model for 200 epochs with a batch size of 130 and learning rate of 0.001
- This model can be used to input random size image to get 4x upscaling

More models with preliminary results:

1. **SRResNet: https://arxiv.org/abs/1609.04802**

    - This model is the generator part of the SRGAN. So in order to use the SRGAN model, we need to train this network first. The weights of the trained model can be used as input to start training the SRGAN.
    - We ran the model using https://github.com/Lornatang/SRGAN-PyTorch.
        - We did not use standardization for SRResNet model runs
        - We need to pass the input images with a size (size(0) % 8 =0). We created patches of size 296x296 as input to the SRResNet model
        - This model performs 8x upscaling
        - Learning rate was constant, set as 0.0001
        - We trained this model for 130 epochs with a batch size of 60
        - The results of this model training are inside SRRESNET_05_04_2139 folder. It is recommended that further training is necessary to reach optimal performance

2. **SRGAN: https://arxiv.org/abs/1609.04802v5**

    - This model has two distinct networks that are trained simultaneously. The generator part of the network has the same architecture as SRResNet.
    - The trained SRResNet model is used to start training the SRGAN
    - We ran the model using https://github.com/Lornatang/SRGAN-PyTorch
        - We need to pass the input images with a size (96x96) only, as the discriminator architecture does not take other sizes as input. We created patches of 96x96 from the training set.
        - This model also does 8x upscaling but with improved texture details in the SR images compared with all the previous models. Although the images are a bit grainy and noisy, this may be reduced with hyperparameter tuning.
        - The learning rate is fixed as 0.0001, but some implementations use LR scheduler to decrease learning rate with epochs
        - We found that it is better not to train SRGAN for many epochs as with other models due to emergence of artifacts during the larger epochs.

- We found for example, the appearance of white diffuse patches in areas of roof tops which are white and sufficiently large in size.