
PyTorch Template Project

PyTorch deep learning project made easy.

- PyTorch Template Project
 - Requirements
 - Features
 - Folder Structure
 - Usage
 - * Config file format
 - * Using config files
 - * Resuming from checkpoints
 - Using Multiple GPU
 - Customization
 - * Custom CLI options
 - * Data Loader
 - * Trainer
 - * Model
 - * Loss
 - * metrics
 - * Additional logging
 - * Validation data
 - * Checkpoints
 - Tensorboard Visualization
 - Contribution
 - TODOs
 - License
 - Acknowledgements

Requirements

- Python ≥ 3.5 (3.6 recommended)
- PyTorch ≥ 0.4 (1.2 recommended)
- tqdm (Optional for `test.py`)
- tensorboard ≥ 1.14 (see Tensorboard Visualization)

Features

- Clear folder structure which is suitable for many deep learning projects.
- `.json` config file support for convenient parameter tuning.
- Customizable command line options for more convenient parameter tuning.
- Checkpoint saving and resuming.
- Abstract base classes for faster development:
 - `BaseTrainer` handles checkpoint saving/resuming, training process logging, and more.
 - `BaseDataLoader` handles batch generation, data shuffling, and validation data splitting.
 - `BaseModel` provides basic model summary.

Folder Structure

```
1  pytorch-template/| |—
2
3  train.py - main script to start training|—
4  test.py - evaluation of trained model| |—
5
6  config.json - holds configuration for training|—
7  parse_config.py - class to handle config file and cli options| |—
8
9  new_project.py - initialize new project with template files| |—
10
11 base/ - abstract base classes| |—
12     base_data_loader.py| |—
13     base_model.py| |—
14     base_trainer.py| |—
15
16 data_loader/ - anything about data loading goes here| |—
17     data_loaders.py| |—
18
19 data/ - default directory for storing input data| |—
20
21 model/ - models, losses, and metrics| |—
22     model.py| |—
23     metric.py| |—
24     loss.py| |—
25
26 saved/| |—
27     models/ - trained models are saved here| |—
28     log/ - default logdir for tensorboard and logging output| |—
```

```

29
30  trainer/ - trainers|└─
31    trainer.py|└─
32
33  logger/ - module for tensorboard visualization and logging|└─
34    visualization.py|└─
35    logger.py|└─
36    logger_config.json|└─
37
38  utils/ - small utility functions|└─
39    util.py└─
40    ...

```

Usage

The code in this repo is an MNIST example of the template. Try `python train.py -c config.json` to run code.

Config file format

Config files are in `.json` format:

```

1  {
2    "name": "Mnist_LeNet",           // training session name
3    "n_gpu": 1,                     // number of GPUs to use for training.
4
5    "arch": {
6      "type": "MnistModel",         // name of model architecture to train
7      "args": {
8
9      }
10   },
11   "data_loader": {
12     "type": "MnistDataLoader",     // selecting data loader
13     "args": {
14       "data_dir": "data/",         // dataset path
15       "batch_size": 64,            // batch size
16       "shuffle": true,             // shuffle training data before
17                                     splitting
18       "validation_split": 0.1      // size of validation dataset.
19                                     float(portion) or int(number of samples)
20     },
21     "num_workers": 2,              // number of cpu processes to be
22                                     used for data loading
23   }
24 }

```

```

22     "type": "Adam",
23     "args":{
24         "lr": 0.001,                // learning rate
25         "weight_decay": 0,         // (optional) weight decay
26         "amsgrad": true
27     }
28 },
29 "loss": "nll_loss",                // loss
30 "metrics": [
31     "accuracy", "top_k_acc"        // list of metrics to evaluate
32 ],
33 "lr_scheduler": {
34     "type": "StepLR",              // learning rate scheduler
35     "args":{
36         "step_size": 50,
37         "gamma": 0.1
38     }
39 },
40 "trainer": {
41     "epochs": 100,                 // number of training epochs
42     "save_dir": "saved/",          // checkpoints are saved in
43                                     save_dir/models/name
44     "save_freq": 1,                // save checkpoints every
45                                     save_freq epochs
46     "verbosity": 2,                // 0: quiet, 1: per epoch, 2:
47                                     full
48
49     "monitor": "min_val_loss"       // mode and metric for model
50                                     performance monitoring. set 'off' to disable.
51     "early_stop": 10                // number of epochs to wait
52                                     before early stop. set 0 to disable.
53
54     "tensorboard": true,            // enable tensorboard
55                                     visualization
56 }
57 }

```

Add additional configurations if you need.

Using config files

Modify the configurations in `.json` config files, then run:

```
1 python train.py --config config.json
```

Resuming from checkpoints

You can resume from a previously saved checkpoint by:

```
1 python train.py --resume path/to/checkpoint
```

Using Multiple GPU

You can enable multi-GPU training by setting `n_gpu` argument of the config file to larger number. If configured to use smaller number of gpu than available, first `n` devices will be used by default. Specify indices of available GPUs by cuda environmental variable. `python train.py --device 2,3 -c config.json` This is equivalent to `CUDA_VISIBLE_DEVICES=2,3 python train.py -c config.py`

Customization

Project initialization

Use the `new_project.py` script to make your new project directory with template files. `python new_project.py ../NewProject` then a new project folder named 'NewProject' will be made. This script will filter out unnecessary files like cache, git files or readme file.

Custom CLI options

Changing values of config file is a clean, safe and easy way of tuning hyperparameters. However, sometimes it is better to have command line options if some values need to be changed too often or quickly.

This template uses the configurations stored in the json file by default, but by registering custom options as follows you can change some of them using CLI flags.

```
1 # simple class-like object having 3 attributes, `flags`, `type`, `
  target`.
2 CustomArgs = collections.namedtuple('CustomArgs', 'flags type target')
3 options = [
4     CustomArgs(['--lr', '--learning_rate'], type=float, target=('
      optimizer', 'args', 'lr')),
5     CustomArgs(['--bs', '--batch_size'], type=int, target=('data_loader
      ', 'args', 'batch_size'))
6     # options added here can be modified by command line flags.
7 ]
```

`target` argument should be sequence of keys, which are used to access that option in the config dict. In this example, `target` for the learning rate option is `('optimizer', 'args', 'lr')` because `config['optimizer']['args']['lr']` points to the learning rate. `python train.py -c config.json --bs 256` runs training with options given in `config.json` except for the `batch size` which is increased to 256 by command line options.

Data Loader

- **Writing your own data loader**

1. **Inherit `BaseDataLoader`**

`BaseDataLoader` is a subclass of `torch.utils.data.DataLoader`, you can use either of them.

`BaseDataLoader` handles:

- Generating next batch
- Data shuffling
- Generating validation data loader by calling `BaseDataLoader.split_validation()`

- **DataLoader Usage**

`BaseDataLoader` is an iterator, to iterate through batches:

```
1 for batch_idx, (x_batch, y_batch) in data_loader:
2     pass
```

- **Example**

Please refer to `data_loader/data_loaders.py` for an MNIST data loading example.

Trainer

- **Writing your own trainer**

1. **Inherit `BaseTrainer`**

`BaseTrainer` handles:

- Training process logging
- Checkpoint saving
- Checkpoint resuming

-
- Reconfigurable performance monitoring for saving current best model, and early stop training.
 - If config `monitor` is set to `max val_accuracy`, which means then the trainer will save a checkpoint `model_best.pth` when `validation accuracy` of epoch replaces current `maximum`.
 - If config `early_stop` is set, training will be automatically terminated when model performance does not improve for given number of epochs. This feature can be turned off by passing 0 to the `early_stop` option, or just deleting the line of config.

2. Implementing abstract methods

You need to implement `_train_epoch()` for your training process, if you need validation then you can implement `_valid_epoch()` as in `trainer/trainer.py`

- **Example**

Please refer to `trainer/trainer.py` for MNIST training.

- **Iteration-based training**

`Trainer.__init__` takes an optional argument, `len_epoch` which controls number of batches(steps) in each epoch.

Model

- **Writing your own model**

1. Inherit `BaseModel`

`BaseModel` handles:

- Inherited from `torch.nn.Module`
- `__str__`: Modify native `print` function to prints the number of trainable parameters.

2. Implementing abstract methods

Implement the forward pass method `forward()`

- **Example**

Please refer to `model/model.py` for a LeNet example.

Loss

Custom loss functions can be implemented in 'model/loss.py'. Use them by changing the name given in "loss" in config file, to corresponding name.

Metrics

Metric functions are located in 'model/metric.py'.

You can monitor multiple metrics by providing a list in the configuration file, e.g.: `json "metrics": ["accuracy", "top_k_acc"],`

Additional logging

If you have additional information to be logged, in `_train_epoch()` of your trainer class, merge them with `log` as shown below before returning:

```
1 additional_log = {"gradient_norm": g, "sensitivity": s}
2 log.update(additional_log)
3 return log
```

Testing

You can test trained model by running `test.py` passing path to the trained checkpoint by `--resume` argument.

Validation data

To split validation data from a data loader, call `BaseDataLoader.split_validation()`, then it will return a data loader for validation of size specified in your config file. The `validation_split` can be a ratio of validation set per total data ($0.0 \leq \text{float} < 1.0$), or the number of samples ($0 \leq \text{int} < n_{\text{total_samples}}$).

Note: the `split_validation()` method will modify the original data loader **Note:** `split_validation()` will return `None` if `"validation_split"` is set to 0

Checkpoints

You can specify the name of the training session in config files: `json "name": "MNIST_LeNet"`,

The checkpoints will be saved in `save_dir/name/timestamp/checkpoint_epoch_n`, with timestamp in `mmdd_HHMMSS` format.

A copy of config file will be saved in the same folder.

Note: checkpoints contain: `python { 'arch': arch, 'epoch': epoch, 'state_dict': self.model.state_dict(), 'optimizer': self.optimizer.state_dict(), 'monitor_best': self.mnt_best, 'config': self.config }`

Tensorboard Visualization

This template supports Tensorboard visualization by using either `torch.utils.tensorboard` or TensorboardX.

1. Install

If you are using pytorch 1.1 or higher, install tensorboard by 'pip install tensorboard>=1.14.0'. Otherwise, you should install tensorboardx. Follow installation guide in TensorboardX.

2. Run training

Make sure that `tensorboard` option in the config file is turned on.

```
1  "tensorboard" : true
```

3. Open Tensorboard server

Type `tensorboard --logdir saved/log/` at the project root, then server will open at `http://localhost:6006`

By default, values of loss and metrics specified in config file, input images, and histogram of model parameters will be logged. If you need more visualizations, use `add_scalar('tag', data)`, `add_image('tag', image)`, etc in the `trainer._train_epoch` method. `add_something()` methods in this template are basically wrappers for those of `tensorboardX.SummaryWriter` and `torch.utils.tensorboard.SummaryWriter` modules.

Note: You don't have to specify current steps, since `WriterTensorboard` class defined at `logger/visualization.py` will track current steps.

Contribution

Feel free to contribute any kind of function or enhancement, here the coding style follows PEP8

Code should pass the Flake8 check before committing.

TODOs

- ☐ Multiple optimizers

-
- ☐ Support more tensorboard functions
 - ☒ Using fixed random seed
 - ☒ Support pytorch native tensorboard
 - ☒ [tensorboardX](#) logger support
 - ☒ Configurable logging layout, checkpoint naming
 - ☒ Iteration-based training (instead of epoch-based)
 - ☒ Adding command line option for fine-tuning

License

This project is licensed under the MIT License. See LICENSE for more details

Acknowledgements

This project is inspired by the project Tensorflow-Project-Template by Mahmoud Gemy