jdit Documentation

Release 0.1.5

dingguanglei

Dec 23, 2019

Notes

CHAPTER 1

Quick Start

You can get a quick start by following these setps. After building and installing jdit package, you can make a new directory for a quick test. Assuming that you get a new directory `example`. run this code in `ipython` .(Create a `main.py` file is also acceptable.)

1.1 Fashion-mnist Classification

To start a simple classification task.

```
from jdit.trainer.instances.fashionClassification import start_fashionClassTrainer
start_fashionClassTrainer()
```

Then you will see something like this as following.

```
===> Build dataset
use 8 thread!
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-
↪idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-
↪idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-
↪idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-
↪idx1-ubyte.gz
Processing...
Done!
===> Building model
SimpleModel Total number of parameters: 2776522
ResNet model use CPU!
apply kaiming weight init!
===> Building optimizer
===> Training
using `tensorboard --logdir=log` to see learning curves and net structure.
```

(continues on next page)

(continued from previous page)

```
training and valid_epoch data, configures info and checkpoint were save in `log`␣
↪directory.
  0%|            | 0/10 [00:00<?, ?epoch/s]
0step [00:00, ?step/s]
```

- It will search a fashion mnist dataset.
- Then build a simple network for classification.
- For training process, you can find learning curves in `tensorboard`.
- It will create a `log` directory in `example/`, which saves training processing data and configures.

1.2 Fashion-mnist Generation GAN

To start a simple generation gan task.

```
from jdit.trainer.instances import start_fashionGenerateGanTrainer
start_fashionClassTrainer()
```

Then you will see something like this as following.

```
===> Build dataset
use 2 thread!
===> Building model
Discriminator Total number of parameters: 100865
Discriminator model use GPU(0)!
apply kaiming weight init!
Generator Total number of parameters: 951361
Generator model use GPU(0)!
apply kaiming weight init!
===> Building optimizer
===> Training
  0%|          | 0/200 [00:00<?, ?epoch/s]
0step [00:00, ?step/s]
```

You can get the training processes info from tensorboard and log directory. It contains:

- Learning curves
- Input and output visualization
- The configures of `Model` , `Trainer` , `Optimizer`, `Dataset` and `Performance` in `.csv` .
- Model checkpoint

1.3 Let's build your own task

Although it is just an example, you still can build your own project easily by using jdit framework. Jdit framework can deal with

- Data visualization. (learning curves, images in pilot process)
- CPU, GPU or GPUs. (Training your model on specify devices)
- Intermediate data storage. (Saving training data into a csv file)

- Model checkpoint automatically.
- Flexible templates can be used to integrate and custom overrides.

So, Let's build your own task by using **jdit**.

CHAPTER 2

Build your own trainer

To build your own trainer, you need prepare these sections:

- `dataset` This is the datasets which you want to use.
- `Model` This is a wrapper of your own pytorch `module` .
- `Optimizer` This is a wrapper of pytorch `opt` .
- `trainer` This is a training pipeline which assemble the sections above.

2.1 jdit.dataset

In this section, you should build your own dataset that you want to use following.

2.1.1 Common dataset

For some reasons, many opening dataset are common. So, you can easily build a standard common dataaset. such as :

- Fashion mnist
- Cifar10
- Lsun

Only one parameters you need to set is `batch_shize` . For these common datasets, you only need to reset the batch size.

```
>>> from jdit.dataset import FashionMNIST
>>> fashion_data = FashionMNIST(batch_shize=64)  # now you get a ``dataset``
```

2.1.2 Custom dataset

If you want to build a dataset by your own data, you need to inherit the class

`jdit.dataset.Dataloaders_factory`

and rewrite it's `build_transforms()` and `build_datasets()` (If you want to use default set, rewrite this is not necessary.)

Following these setps:

- Rewrite your own transforms to `self.train_transform_list` and `self.valid_transform_list`. (Not necessary)
- Register your training dataset to `self.dataset_train` by using `self.train_transform_list`
- Register your valid_epoch dataset to `self.dataset_valid` by using `self.valid_transform_list`

Example:

```
class FashionMNIST(DataLoadersFactory):
    def __init__(self, root=r'.\datasets\fashion_data', batch_size=128, num_workers=-
↪1):
        super(FashionMNIST, self).__init__(root, batch_size, num_workers)

    def build_transforms(self, resize=32):
        # This is a default set, you can rewrite it.
        self.train_transform_list = self.valid_transform_list = [
            transforms.Resize(resize),
            transforms.ToTensor(),
            transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])]

    def build_datasets(self):
        self.dataset_train = datasets.CIFAR10(root, train=True, download=True,
            transform=transforms.Compose(self.train_transform_list))
        self.dataset_valid = datasets.CIFAR10(root, train=False, download=True,
            transform=transforms.Compose(self.valid_transform_list))
```

For now, you get your own dataset.

2.2 Model

In this section, you should build your own network.

First, you need to build a pytorch `module` like this:

```
>>> class SimpleModel(nn.Module):
...     def __init__(self):
...         super(SimpleModel, self).__init__()
...         self.layer1 = nn.Linear(32, 64)
...         self.layer2 = nn.Linear(64, 1)
...
...     def forward(self, input):
...         out = self.layer1(input)
...         out = self.layer2(out)
...         return out
>>> network = SimpleModel()
```

Note: You don't need to convert it to gpu or using data parallel. The `jdit.Model` will do this for you.

Second, wrap your model by using `jdit.Model` . Set which gpus you want to use and the weights init method.

Note: For some reasons, the gpu id in pytorch still start from 0. For this model, it will handel this problem. If you have gpu `[0,1,2,3]` , and you only want to use 2,3. Just set `gpu_ids_abs=[2, 3]` .

```
>>> from jdit import Model
>>> network = SimpleModel()
>>> jdit_model = Model(network, gpu_ids_abs=[2,3], init_method="kaiming")
SimpleModel Total number of parameters: 2177
SimpleModel dataParallel use GPUs[2, 3]!
apply kaiming weight init!
```

For now, you get your own dataset.

2.3 Optimizer

In this section, you should build your an optimizer.

Compare with the optimizer in pytorch. This extend a easy function that can do a learning rate decay and reset.

However, `do_lr_decay()` will be called every epoch or on certain epoch at the end automatically. Actually, you don' need to do anything to apply learning rate decay. If you don't want to decay. Just set `lr_decay = 1.` or set a decay epoch larger than training epoch. I will show you how it works and you can implement something special strategies.

```
   >>> from jdit import Optimizer
   >>> from torch.nn import Linear
   >>> network = Linear(10, 1)
   >>> #set params
   >>> #`optimizer` is equal to pytorch class name (torch.optim.RMSprop).
   >>> hparams = {
   ...     "optimizer" = "RMSprop" ,
   ...     "lr" = 0.001,
   ...     "lr_decay" = 0.5,
   ...     "weight_decay" = 2e-5,
   ...     "momentum" = 0}
   >>> #define optimizer
   >>> opt = Optimizer(network.parameters(),**hparams)
   >>> opt.lr
   0.001
   >>> opt.do_lr_decay()
   >>> opt.lr
   0.0005
   >>> opt.do_lr_decay(reset_lr = 1)
   >>> opt.lr
   1

You can pass a certain name to use it,such "Adam" ,"RMSprop", "SGD".
```

Note: As for spectrum normalization, the optimizer will filter out the differentiable weights. So, you don't need

write something like this `filter(lambda p:  p.requires_grad, params)` Merely pass the `model.parameters()` is enough.

For now, you get an Optimizer.

2.4 trainer

For the final section it is a little complex. It supplies some templates such as `SupTrainer` `GanTrainer` `ClassificationTrainer` and `instances`.

The inherit relation shape is following:

```
SupTrainer

    ClassificationTrainer
        instances.FashionClassTrainer

    SupGanTrainer

        Pix2pixGanTrainer
           instances.CifarPix2pixGanTrainer

        GenerateGanTrainer

           instances.FashionGenerateGanTrainer
```

2.4.1 Top level `SupTrainer`

`SupTrainer` is the top class of these templates.

It defines some tools to record the log, data visualization and so on. Besides, it contain a big loop of epoch, which can be inherited by the second level templates to fill the contents in each opch training.

Something like this:

```
def train():
   for epoch in range(nepochs):
       self._record_configs() # record info
       self.train_epoch()
       self.valid_epoch()
       # do learning rate decay
       self._change_lr()
       # save model check point
       self._check_point()
   self.test()
```

Every method will be rewrite by the second level templates. It only defines a rough framework.

2.4.2 Second level `ClassificationTrainer`

On this level, the task becomes more clear, a classification task. We get one `model`, one `optimizer` and one `dataset` and the data structure is images and labels. So, to init a ClassificationTrainer.

```
class ClassificationTrainer(SupTrainer):
    def __init__(self, logdir, nepochs, gpu_ids, net, opt, datasets, num_class):
        super(ClassificationTrainer, self).__init__(nepochs, logdir, gpu_ids_abs)
        self.net = net
        self.opt = opt
        self.datasets = datasets
        self.num_class = num_class
        self.labels = None
        self.output = None
```

For the next, build a training loop for one epoch. You must using `self.step` to record the training step.

```
def train_epoch(self, subbar_disable=False):
    # display training images every epoch
    self._watch_images(show_imgs_num=3, tag="Train")
    for iteration, batch in tqdm(enumerate(self.datasets.loader_train, 1), unit="step
↪", disable=subbar_disable):
        self.step += 1 # necessary!
        # unzip data from one batch and move to certain device
        self.input, self.ground_truth, self.labels = self.get_data_from_batch(batch,␣
↪self.device)
        self.output = self.net(self.input)
        # this is defined in SupTrainer.
        # using `self.compute_loss` and `self.opt` to do a backward.
        self._train_iteration(self.opt, self.compute_loss, tag="Train")

@abstractmethod
def compute_loss(self):
    """Compute the main loss and observed variables.
    Rewrite by the next templates.
    """

@abstractmethod
def compute_valid(self):
    """Compute the valid_epoch variables for visualization.
    Rewrite by the next templates.
    """
```

The `compute_loss()` and `compute_valid` should be rewrite in the next template.

2.4.3 Third level `FashionClassTrainer`

Up to this level every this is clear. So, inherit the `ClassificationTrainer` and fill the specify methods.

```
class FashionClassTrainer(ClassificationTrainer):
    def __init__(self, logdir, nepochs, gpu_ids, net, opt, dataset):
        super(FashionClassTrainer, self).__init__(logdir, nepochs, gpu_ids, net, opt,␣
↪dataset)
        data, label = self.datasets.samples_train
        # show dataset in tensorboard
        self.watcher.embedding(data, data, label, 1)
```

(continues on next page)

(continued from previous page)

```
    def compute_loss(self):
        var_dic = {}
        var_dic["CEP"] = loss = nn.CrossEntropyLoss()(self.output, self.labels.
↪squeeze().long())
        return loss, var_dic

    def compute_valid(self):
        var_dic = {}
        var_dic["CEP"] = cep = nn.CrossEntropyLoss()(self.output, self.labels.
↪squeeze().long())

        _, predict = torch.max(self.output.detach(), 1)  # 0100=>1  0010=>2
        total = predict.size(0) * 1.0
        labels = self.labels.squeeze().long()
        correct = predict.eq(labels).cpu().sum().float()
        acc = correct / total
        var_dic["ACC"] = acc
        return var_dic
```

`compute_loss()` will be called every training step of backward. It returns two values.

- The first one, `loss` , is **main loss** which will be implemented `loss.backward()` to update model weights.
- The second one, `var_dic` , is a **value dictionary** which will be visualized on tensorboard and depicted as a curve.

In this example, for `compute_loss()` it will use `loss = nn.CrossEntropyLoss()` to do a backward propagation and visualize it on tensorboard named `"CEP"`.

`compute_loss()` will be called every validation step. It returns one value.

- The `var_dic` , is the same thing like `var_dic` in `compute_loss()` .

Note: `compute_loss()` will be called under `torch.no_grad()` . So, grads will not be computed in this method. But if you need to get grads, please use `torch.enable_grad()` to make grads computation available.

Finally, you get a trainer.

You have got everything. Put them together and train it!

```
>>> mnist = FashionMNIST(batch_size)
>>> net = Model(SimpleModel(depth=depth), gpu_ids_abs=gpus, init_method="kaiming")
>>> opt = Optimizer(net.parameters(), **hparams)
>>> Trainer = FashionClassTrainer("log", nepochs, gpus, net, opt, mnist, 10)
>>> Trainer.train()
```

CHAPTER 3

jdit.dataset

3.1 Dataloaders_factory

class jdit.dataset.**DataLoadersFactory**(*root: str*, *batch_size: int*, *num_workers=-1*, *shuffle=True*, *subdata_size=1*)

This is a super class of dataloader.

It defines same basic attributes and methods.

- For training data: `train_dataset`, `loader_train`, `nsteps_train` . Others such as `valid_epoch` and `test` have the same naming format.
- For transform, you can define your own transforms.
- If you don't have test set, it will be replaced by valid_epoch dataset.

It will build dataset following these setps:

1. `build_transforms()` To build transforms for training dataset and valid_epoch. You can rewrite this method for your own transform. It will be used in `build_datasets()`
2. `build_datasets()` You must rewrite this method to load your own dataset by passing datasets to `self.dataset_train` and `self.dataset_valid`. `self.dataset_test` is optional. If you don't pass a test dataset, it will be replaced by `self.dataset_valid`.

   Example:

```
def build_transforms(self, resize=32):
    self.train_transform_list = self.valid_transform_list = [
        transforms.Resize(resize),
        transforms.ToTensor(),
        transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])]
# Inherit this class and write this method.
def build_datasets(self):
    self.dataset_train = datasets.CIFAR10(root, train=True, download=True,
        transform=transforms.Compose(self.train_transform_list))
```

(continues on next page)

(continued from previous page)

```
    self.dataset_valid = datasets.CIFAR10(root, train=False, download=True,
        transform=transforms.Compose(self.valid_transform_list))
```

3. `build_loaders()` It will use dataset, and passed parameters to build dataloaders for `self.loader_train`, `self.loader_valid` and `self.loader_test`.

- `root` is the root path of datasets.
- `batch_shape` is the size of data loader. shape is `(Batchsize, Channel, Height, Width)`
- `num_workers` is the number of threads, using to load data. If you pass -1, it will use the max number of threads, according to your cpu. Default: -1
- `shuffle` is whether shuffle the data. Default: `True`

build_datasets()

You must to rewrite this method to load your own datasets.

- `self.dataset_train` . Assign a training `dataset` to this.
- `self.dataset_valid` . Assign a valid_epoch `dataset` to this.
- `self.dataset_test` is optional. Assign a test `dataset` to this. If not, it will be replaced by `self.dataset_valid` .

Example:

```
self.dataset_train = datasets.CIFAR10(root, train=True, download=True,
                                      transform=transforms.Compose(self.train_
↪transform_list))
self.dataset_valid = datasets.CIFAR10(root, train=False, download=True,
                                      transform=transforms.Compose(self.valid_
↪transform_list))
```

build_loaders()

Build datasets The previous function `self.build_datasets()` has created datasets. Use these datasets to build their's dataloaders

build_transforms(*resize: int = 32*)

This will build transforms for training and valid_epoch.

You can rewrite this method to build your own transforms. Don't forget to register your transforms to `self.train_transform_list` and `self.valid_transform_list`

The following is the default set.

```
self.train_transform_list = self.valid_transform_list = [
    transforms.Resize(resize),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])]
```

3.2 HandMNIST

class jdit.dataset.**HandMNIST**(*root='datasets/hand_data'*, *batch_size=64*, *num_workers=-1*)

Hand writing mnist dataset.

Example:

```
>>> data = HandMNIST(r"../datasets/mnist")
use 8 thread!
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Processing...
Done!
>>> data.dataset_train
Dataset MNIST
Number of datapoints: 60000
Split: train
Root Location: data
Transforms (if any): Compose(
                         Resize(size=32, interpolation=PIL.Image.BILINEAR)
                         ToTensor()
                         Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
                     )
Target Transforms (if any): None
>>> # We don't set test dataset, so they are the same.
>>> data.dataset_valid is data.dataset_test
True
>>> # Number of steps at batch size 128.
>>> data.nsteps_train
469
>>> # Total samples of training datset.
>>> len(data.dataset_train)
60000
>>> # The batch size of sample load is 1. So, we get length of loader is equal to␣
↪samples amount.
>>> len(data.samples_train)
6000
```

build_datasets()
Build datasets by using `datasets.MNIST` in pytorch

build_transforms(*resize: int = 32*)
This will build transforms for training and valid_epoch.

You can rewrite this method to build your own transforms. Don't forget to register your transforms to `self.train_transform_list` and `self.valid_transform_list`

The following is the default set.

```
self.train_transform_list = self.valid_transform_list = [
    transforms.Resize(resize),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])]
```

3.3 FashionMNIST

class jdit.dataset.**FashionMNIST**(*root='datasets/fashion_data'*, *batch_size=64*, *num_workers=-1*)

build_datasets()
You must to rewrite this method to load your own datasets.

- `self.dataset_train` . Assign a training `dataset` to this.
- `self.dataset_valid` . Assign a valid_epoch `dataset` to this.
- `self.dataset_test` is optional. Assign a test `dataset` to this. If not, it will be replaced by `self.dataset_valid` .

Example:

```
self.dataset_train = datasets.CIFAR10(root, train=True, download=True,
                                      transform=transforms.Compose(self.train_
↪transform_list))
self.dataset_valid = datasets.CIFAR10(root, train=False, download=True,
                                      transform=transforms.Compose(self.valid_
↪transform_list))
```

build_transforms(*resize: int = 32*)

This will build transforms for training and valid_epoch.

You can rewrite this method to build your own transforms. Don't forget to register your transforms to `self.train_transform_list` and `self.valid_transform_list`

The following is the default set.

```
self.train_transform_list = self.valid_transform_list = [
    transforms.Resize(resize),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])]
```

3.4 Cifar10

class jdit.dataset.**Cifar10**(*root='datasets/cifar10'*, *batch_size=32*, *num_workers=-1*)

build_datasets()

You must to rewrite this method to load your own datasets.

- `self.dataset_train` . Assign a training `dataset` to this.
- `self.dataset_valid` . Assign a valid_epoch `dataset` to this.
- `self.dataset_test` is optional. Assign a test `dataset` to this. If not, it will be replaced by `self.dataset_valid` .

Example:

```
self.dataset_train = datasets.CIFAR10(root, train=True, download=True,
                                      transform=transforms.Compose(self.train_
↪transform_list))
self.dataset_valid = datasets.CIFAR10(root, train=False, download=True,
                                      transform=transforms.Compose(self.valid_
↪transform_list))
```

3.5 Lsun

class jdit.dataset.**Lsun**(*root*, *batch_size=32*, *num_workers=-1*)

build_datasets()
You must to rewrite this method to load your own datasets.

- `self.dataset_train` . Assign a training `dataset` to this.
- `self.dataset_valid` . Assign a valid_epoch `dataset` to this.
- `self.dataset_test` is optional. Assign a test `dataset` to this. If not, it will be replaced by `self.dataset_valid` .

Example:

```
self.dataset_train = datasets.CIFAR10(root, train=True, download=True,
                                      transform=transforms.Compose(self.train_
↪transform_list))
self.dataset_valid = datasets.CIFAR10(root, train=False, download=True,
                                      transform=transforms.Compose(self.valid_
↪transform_list))
```

build_transforms(*resize: int = 32*)
This will build transforms for training and valid_epoch.

You can rewrite this method to build your own transforms. Don't forget to register your transforms to `self.train_transform_list` and `self.valid_transform_list`

The following is the default set.

```
self.train_transform_list = self.valid_transform_list = [
    transforms.Resize(resize),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])]
```

3.6 get_mnist_dataloaders

jdit.dataset.**get_mnist_dataloaders**(*root='..\\data'*, *batch_size=128*)
MNIST dataloader with (32, 32) sized images.

3.7 get_fashion_mnist_dataloaders

jdit.dataset.**get_fashion_mnist_dataloaders**(*root='.\\dataset\\fashion_data'*, *batch_size=128*, *resize=32*, *transform_list=None*, *num_workers=-1*)
Fashion MNIST dataloader with (32, 32) sized images.

3.8 get_lsun_dataloader

jdit.dataset.**get_lsun_dataloader**(*path_to_data='/data/dgl/LSUN'*, *dataset='bedroom_train'*, *batch_size=64*)
LSUN dataloader with (128, 128) sized images.

path_to_data [str] One of 'bedroom_val' or 'bedroom_train'

CHAPTER 4

jdit.model

4.1 Model

class jdit.**Model**(*proto_model: <Mock name='mock.Module' id='140594783391360'>*, *gpu_ids_abs: Union[list, tuple] = ()*, *init_method: Union[str, function, None] = 'kaiming'*, *show_structure=False*, *check_point_pos=None*, *verbose=True*)

A warapper of pytorch `module` .

In the simplest case, we use a raw pytorch `module` to assemble a `Model` of this class. It can be more convenient to use some feather method, such `_check_point` , `load_weights` and so on.

- `proto_model` is the core model in this class. It is no necessary to passing a `module` when you init a `Model` . You can build a model later by using `Model.define(module)` or load a model from a file.
- `gpu_ids_abs` controls the gpus which you want to use. you should use a absolute id of gpus.
- `init_method` controls the weights init method.
  - At init_method="xavier", it will use `init.xavier_normal_` , in `pytorch.nn.init` , to init the Conv layers of model.
  - At init_method="kaiming", it will use `init.kaiming_normal_` , in `pytorch.nn.init` , to init the Conv layers of model.
  - At init_method=your_own_method, it will be used on weights, just like what `pytorch.nn.init` method does.
- `show_structure` controls whether to show your network structure.

Note: Don't try to pass a `DataParallel` model. Only `module` is accessible. It will change to `DataParallel` class automatically by passing a muti-gpus ids, like `[0, 1]` .

Note: `gpu_ids_abs` must be a tuple or list. If you want to use cpu, just passing an ampty list like `[]` .

Args: proto_model (module): A pytroch module. Default: `None`

gpu_ids_abs (tuple or list): The absolute id of gpus. if [] using cpu. Default: `()`

init_method (str or def): Weights init method. Default: `"Kaiming"`

show_structure (bool): Is the structure shown. Default: `False`

Attributes: num_params (int): The totals amount of weights in this model.

gpu_ids_abs (list or tuple): Which device is this model on.

Examples:

```
>>> from torch.nn import Sequential, Conv3d
>>> # using a square kernels and equal stride
>>> module = Sequential(Conv3d(16, 33, (3, 5, 2), stride=(2, 1, 1), padding=(4, 2,
↪ 0)))
>>> # using cpu to init a Model by module.
>>> net = Model(module, [], show_structure=False)
Sequential Total number of parameters: 15873
Sequential model use CPU!
apply kaiming weight init!
>>> input_tensor = torch.randn(20, 16, 10, 50, 100)
>>> output = net(input_tensor)
```

convert_to_distributed(*device_ids=None*, *output_device=None*, *dim=0*, *broadcast_buffers=True*, *process_group=None*, *bucket_cap_mb=25*, *find_unused_parameters=False*, *check_reduction=False*)

Args: module (Module): module to be parallelized device_ids (list of int or torch.device): CUDA devices. This should

only be provided when the input module resides on a single CUDA device. For single-device modules, the i``th :attr:`module` replica is placed on ``device_ids[i]. For multi-device modules and CPU modules, device_ids must be None or an empty list, and input data for the forward pass must be placed on the correct device. (default: all devices for single-device modules)

output_device (int or torch.device): device location of output for single-device CUDA modules. For multi-device modules and CPU modules, it must be None, and the module itself dictates the output location. (default: device_ids[0] for single-device modules)

broadcast_buffers (bool): flag that enables syncing (broadcasting) buffers of the module at beginning of the forward function. (default: `True`)

process_group: the process group to be used for distributed data all-reduction. If `None`, the default process group, which is created by `torch.distributed.init_process_group`, will be used. (default: `None`)

bucket_cap_mb: DistributedDataParallel will bucket parameters into multiple buckets so that gradient reduction of each bucket can potentially overlap with backward computation. `bucket_cap_mb` controls the bucket size in MegaBytes (MB) (default: 25)

find_unused_parameters (bool): Traverse the autograd graph of all tensors contained in the return value of the wrapped module's `forward` function. Parameters that don't receive gradients as part of this graph are preemptively marked as being ready to be reduced. (default: `False`)

check_reduction: when setting to `True`, it enables DistributedDataParallel to automatically check if the previous iteration's backward reductions were successfully issued at the beginning of every iteration's forward function. You normally don't need this option enabled unless you are observing

weird behaviors such as different ranks are getting different gradients, which should not happen if DistributedDataParallel is correctly used. (default: `False`)

Attributes: module (Module): the module to be parallelized

Example:

```
>>> torch.distributed.init_process_group(backend='nccl', world_size=4, init_
↪method='...')
>>> net.convert_to_distributed(pg)
>>> # same thing
>>> net.model = torch.nn.DistributedDataParallel(net.model, pg)
```

static count_params(*proto_model: <Mock name='mock.Module' id='140594783391360'>*)
count the total parameters of model.

Parameters **proto_model** – pytorch module

Returns number of parameters

define(*proto_model: <Mock name='mock.Module' id='140594783391360'>, gpu_ids_abs: Union[list, tuple], init_method: Union[str, function, None], show_structure: bool*)
Define and wrap a pytorch module, according to CPU, GPU and multi-GPUs.

- Print the module's info.
- Move this module to specify device.
- Apply weight init method.

Parameters

- **proto_model** – Network, type of `module`.
- **gpu_ids_abs** – Be used GPUs' id, type of `tuple` or `list`. If not use GPU, pass `()`.
- **init_method** – init weights method("kaiming") or `False` don't use any init.
- **show_structure** – If print structure of model.

load_point(*model_name: str*, *epoch: int*, *logdir='log'*)
load model and weights from a certain checkpoint.

this method is cooperate with method *self.chechPoint()*

load_weights(*weights: Union[dict, str], strict=True*)
Assemble a model and weights from paths or passing parameters.

You can load a model from a file, passing parameters or both.

Parameters

- **weights** – Pytorch weights or weights file path.
- **strict** – The same function in pytorch `model.load_state_dict(weights, strict = strict)` . default:`True`

Returns `module`

Example:

```
>>> from torchvision.models.resnet import resnet18
>>> model = Model(resnet18())
ResNet Total number of parameters: 11689512
ResNet model use CPU!
apply kaiming weight init!
>>> model.save_weights("model.pth",)
try to remove 'module.' in keys of weights dict...
>>> model.load_weights("model.pth", True)
Try to remove `moudle.` to keys of weights dict
```

print_network(*proto_model: <Mock name='mock.Module' id='140594783391360'>*, *show_structure=False*)

Print total number of parameters and structure of network

Parameters

- **proto_model** – Pytorch module
- **show_structure** – If show network's structure. default: `False`

Returns Total number of parameters

save_weights(*weights_path: str*, *fix_weights=True*)

Save a model and weights to files.

You can save a model, weights or both to file.

Note: This method deal well with different devices on model saving. You don' need to care about which devices your model have saved.

Parameters

- **weights_path** – Pytorch weights or weights file path.
- **fix_weights** – If this is true, it will remove the '.module' in keys, when you save a `DataParallel`. without any moving operation. Otherwise, it will move to cpu, especially in `DataParallel`. default:`False`

Example:

```
>>> from torch.nn import Linear
>>> model = Model(Linear(10,1))
Linear Total number of parameters: 11
Linear model use CPU!
apply kaiming weight init!
>>> model.save_weights("weights.pth")
try to remove 'module.' in keys of weights dict...
>>> model.load_weights("weights.pth")
Try to remove `moudle.` to keys of weights dict
```

CHAPTER 5

jdit.optimizer

5.1 Optimizer

class jdit.**Optimizer**(*params: parameters of model, optimizer: [Adam,RMSprop,SGD...], lr_decay: float = 1.0, decay_position: Union[int, tuple, list] = -1, lr_reset: Dict[int, float] = None, position_type: ('epoch','step') = 'epoch', **kwargs*)

This is a wrapper of `optimizer` class in pytorch.

We add something new features in order to feather control the optimizer.

- `params` is the parameters of model which need to be updated. It will use a filter to get all the parameters that required grad automatically. Like this

  `filter(lambda p:  p.requires_grad, params)`

  So, you can passing `model.all_params()` without any filters.

- `learning rate decay` When calling `do_lr_decay()`, it will do a learning rate decay. like:

$$lr = lr * decay$$

- `learning rate reset` . Reset learning rate, it can change learning rate and decay directly.

Parameters

- **params** – parameters of model, which need to be updated.
- **optimizer** – An optimizer classin pytorch, such as `torch.optim.Adam`.
- **lr_decay** – learning rate decay. Default: 0.92.
- **decay_at_epoch** – The position of applying lr decay. Default: None.
- **decay_at_step** – learning rate decay. Default: None
- **kwargs** – pass hyper-parameters to optimizer, such as `lr` , `betas` , `weight_decay` .

Returns

Args:

params (dict): parameters of model, which need to be updated.

optimizer (torch.optim.Optimizer): An optimizer classin pytorch, such as `torch.optim.Adam`

lr_decay (float, optional): learning rate decay. Default: 0.92

decay_position (int, list, optional): The decaly position of lr. Default: None

lr_reset (Dict[position(int), lr(float)]): Reset learning at a certain position. Default: None

position_type ('epoch','step'): Position type. Default: None

****kwargs** : pass hyper-parameters to optimizer, such as `lr` , `betas` , `weight_decay` .

Example:

```
>>> from torch.nn import Sequential, Conv3d
>>> from torch.optim import Adam
>>> module = Sequential(Conv3d(16, 33, (3, 5, 2), stride=(2, 1, 1), padding=(4, 2,
↪ 0)))
>>> opt = Optimizer(module.parameters() ,"Adam", 0.5, 10, {4:0.99},"epoch", lr=1.
↪0, betas=(0.9, 0.999),
weight_decay=1e-5)
>>> print(opt)
(Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    eps: 1e-08
    lr: 1.0
    weight_decay: 1e-05
)
    lr_decay:0.5
    decay_position:10
    lr_reset:{4: 0.99}
    position_type:epoch
))
>>> opt.lr
1.0
>>> opt.lr_decay
0.5
>>> opt.do_lr_decay()
>>> opt.lr
0.5
>>> opt.do_lr_decay(reset_lr=1)
>>> opt.lr
1
>>> opt.opt
Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    eps: 1e-08
    lr: 1
    weight_decay: 1e-05
)
>>> opt.is_decay_lr(1)
False
>>> opt.is_decay_lr(10)
```

(continues on next page)

(continued from previous page)

```
True
>>> opt.is_decay_lr(20)
True
>>> opt.is_reset_lr(4)
0.99
>>> opt.is_reset_lr(5)
False
```

do_lr_decay(*reset_lr_decay: float = None*, *reset_lr: float = None*)
Do learning rate decay, or reset them.

Passing parameters both None: Do a learning rate decay by `self.lr = self.lr * self.lr_decay`.

Passing parameters reset_lr_decay or reset_lr: Do a learning rate or decay reset. by `self.lr = reset_lr` `self.lr_decay = reset_lr_decay`

Parameters

- **reset_lr_decay** – if not None, use this value to reset *self.lr_decay*. Default: None.
- **reset_lr** – if not None, use this value to reset *self.lr*. Default: None.

Returns

is_decay_lr(*position: Optional[int]*) → bool
Judge if use learning decay on this position.

Parameters **position** – (int) A position of step or epoch.

Returns bool

is_reset_lr(*position: Optional[int]*) → bool
Judge if use learning decay on this position.

Parameters **position** – (int) A position of step or epoch.

Returns bool

CHAPTER 6

jdit.trainer

6.1 SupTrainer

class jdit.trainer.**SupTrainer**(*nepochs: int*, *logdir: str*, *gpu_ids_abs: Union[list*, *tuple] = ()*)

this is a super class of all trainers

It defines: * The basic tools, `Performance()`, `Watcher()`, `Loger()`. * The basic loop of epochs. * Learning rate decay and model check point.

debug()

Debug the trainer.

It will check the function

- `self._record_configs()` save all module's configures.
- `self.train_epoch()` train one epoch with several samples. So, it is vary fast.
- `self.valid_epoch()` valid one epoch using dataset_valid.
- `self._change_lr()` do learning rate change.
- `self._check_point()` do model check point.
- `self.test()` do test by using dataset_test.

Before debug, it will reset the `datasets` and only pick up several samples to do fast test. For test, it build a `log_debug` directory to save the log.

Returns bool. It will return `True`, if passes all the tests.

dist_train(*process_bar_header: str = None*, *process_bar_position: int = None*, *sub-bar_disable=False*, *record_configs=True*, *show_network=False*, ***kwargs*)

The main training loop of epochs.

Parameters

- **process_bar_header** – The tag name of process bar header, which is used in `tqdm(desc=process_bar_header)`

- **process_bar_position** – The process bar's position. It is useful in multitask, which is used in `tqdm(position=process_bar_position)`
- **subbar_disable** – If show the info of every training set,
- **record_configs** – If record the training processing data.
- **show_network** – If show the structure of network. It will cost extra memory,
- **kwargs** – Any other parameters that passing to `tqdm()` to control the behavior of process bar.

get_data_from_batch(*batch_data: list*, *device: <Mock name='mock.device' id='140594778247688'>*)

Split your data from one batch data to specify . If your dataset return something like

`return input_data, label.`

It means that two values need unpack. So, you need to split the batch data into two parts, like this

`input, ground_truth = batch_data[0], batch_data[1]`

> **Caution:** Don't forget to move these data to device, by using `input.to(device)` .

Parameters

- **batch_data** – One batch data from dataloader.
- **device** – the device that data will be located.

Returns The certain variable with correct device location.

Example:

```
# load and unzip the data from one batch tuple (input, ground_truth)
input, ground_truth = batch_data[0], batch_data[1]
# move these data to device
return input.to(device), ground_truth.to(device)
```

plot_graphs_lazy()

Plot model graph on tensorboard. To plot all models graphs in trainer, by using variable name as model name.

Returns

train(*process_bar_header: str = None*, *process_bar_position: int = None*, *subbar_disable=False*, *record_configs=True*, *show_network=False*, ***kwargs*)

The main training loop of epochs.

Parameters

- **process_bar_header** – The tag name of process bar header, which is used in `tqdm(desc=process_bar_header)`
- **process_bar_position** – The process bar's position. It is useful in multitask, which is used in `tqdm(position=process_bar_position)`
- **subbar_disable** – If show the info of every training set,
- **record_configs** – If record the training processing data.
- **show_network** – If show the structure of network. It will cost extra memory,

- **kwargs** – Any other parameters that passing to `tqdm()` to control the behavior of process bar.

train_epoch(*subbar_disable=False*)

You get train loader and do a loop to deal with data.

> **Caution:** You must record your training step on `self.step` in your loop by doing things like this `self.step += 1`.

Example:

```
for iteration, batch in tqdm(enumerate(self.datasets.loader_train, 1)):
    self.step += 1
    self.input_cpu, self.ground_truth_cpu = self.get_data_from_batch(batch,␣
↪self.device)
    self._train_iteration(self.opt, self.compute_loss, tag="Train")
```

Returns

6.2 Single Model Trainer

6.2.1 SupSingleModelTrainer

class jdit.trainer.**SupSingleModelTrainer**(*logdir*, *nepochs*, *gpu_ids_abs*, *net: jdit.model.Model*, *opt: jdit.optimizer.Optimizer*, *datasets: jdit.dataset.DataLoadersFactory*)

This is a Single Model Trainer. It means you only have one model.

input, gound_truth output = model(input) loss(output, gound_truth)

compute_loss() -> (*<Mock name='mock.Tensor' id='140594777874840'>*, *<class 'dict'>*)

Rewrite this method to compute your own loss Discriminator. Use self.input, self.output and self.ground_truth to compute loss. You should return a **loss** for the first position. You can return a `dict` of loss that you want to visualize on the second position.like

Example:

```
var_dic = {}
var_dic["LOSS"] = loss_d = (self.output ** 2 - self.groundtruth ** 2) ** 0.5
return: loss, var_dic
```

compute_valid() → dict

Rewrite this method to compute your validation values. Use self.input, self.output and self.ground_truth to compute valid loss. You can return a `dict` of validation values that you want to visualize.

Example:

```
# It will do the same thing as ``compute_loss()``
var_dic, _ = self.compute_loss()
return var_dic
```

get_data_from_batch(*batch_data: list*, *device: <Mock name='mock.device' id='140594778247688'>*)

Load and wrap data from the data lodaer.

Split your one batch data to specify variable.

Example:

```
# batch_data like this [input_Data, ground_truth_Data]
input_cpu, ground_truth_cpu = batch_data[0], batch_data[1]
# then move them to device and return them
return input_cpu.to(self.device), ground_truth_cpu.to(self.device)
```

Parameters

- **batch_data** – one batch data load from `DataLoader`
- **device** – A device variable. `torch.device`

Returns input Tensor, ground_truth Tensor

train_epoch(*subbar_disable=False*)

You get train loader and do a loop to deal with data.

> **Caution:** You must record your training step on `self.step` in your loop by doing things like this `self.step += 1`.

Example:

```
for iteration, batch in tqdm(enumerate(self.datasets.loader_train, 1)):
    self.step += 1
    self.input_cpu, self.ground_truth_cpu = self.get_data_from_batch(batch,␣
↪self.device)
    self._train_iteration(self.opt, self.compute_loss, tag="Train")
```

Returns

valid_epoch()

Validate model each epoch.

It will be called each epoch, when training finish. So, do same verification here.

Example:

avg_dic: dict = {} self.net.eval() for iteration, batch in enumerate(self.datasets.loader_valid, 1):

self.input, self.ground_truth = self.get_data_from_batch(batch, self.device) with torch.no_grad():

self.output = self.net(self.input) dic: dict = self.compute_valid()

if avg_dic == {}: avg_dic: dict = dic

else:

for key in dic.keys(): avg_dic[key] += dic[key]

for key in avg_dic.keys(): avg_dic[key] = avg_dic[key] / self.datasets.nsteps_valid

self.watcher.scalars(avg_dic, self.step, tag="Valid") self.loger.write(self.step, self.current_epoch, avg_dic, "Valid", header=self.step <= 1) self._watch_images(tag="Valid") self.net.train()

6.2.2 ClassificationTrainer

class jdit.trainer.**ClassificationTrainer**(*logdir*, *nepochs*, *gpu_ids*, *net*, *opt*, *datasets*, *num_class*)

this is a classification trainer.

compute_loss()

Compute the main loss and observed values.

Compute the loss and other values shown in tensorboard scalars visualization. You should return a main loss for doing backward propagation.

So, if you want some values visualized. Make a `dict()` with key name is the variable's name. The training logic is :

self.input, self.ground_truth = self.get_data_from_batch(batch, self.device) self.output = self.net(self.input) self._train_iteration(self.opt, self.compute_loss, csv_filename="Train")

So, you have *self.net*, *self.input*, *self.output*, *self.ground_truth* to compute your own loss here.

Note: Only the main loss will do backward propagation, which is the first returned variable. If you have the joint loss, please add them up and return one main loss.

Note: All of your variables in returned `dict()` will never do backward propagation with `model.train()`. However, It still compute grads, without using `with torch.autograd.no_grad()`. So, you can compute any grads variables for visualization.

Example:

```
var_dic = {}
labels = self.ground_truth.squeeze().long()
var_dic["MSE"] = loss = nn.MSELoss()(self.output, labels)
return loss, var_dic
```

compute_valid()

Compute the valid_epoch variables for visualization.

Compute the validations. For the validations will only be used in tensorboard scalars visualization. So, if you want some variables visualized. Make a `dict()` with key name is the variable's name. You have *self.net*, *self.input*, *self.output*, *self.ground_truth* to compute your own validations here.

Note: All of your variables in returned `dict()` will never do backward propagation with `model.eval()`. However, It still compute grads, without using `with torch.autograd.no_grad()`. So, you can compute some grads variables for visualization.

Example:: var_dic = {} labels = self.ground_truth.squeeze().long() var_dic["CEP"] = nn.CrossEntropyLoss()(self.output, labels) return var_dic

get_data_from_batch(*batch_data*, *device*)

If you have different behavior. You need to rewrite thisd method and the method *sllf.train_epoch()*

Parameters

- **batch_data** – A Tensor loads from dataset

- **device** – compute device

Returns Tensors,

valid_epoch()
Validate model each epoch.

It will be called each epoch, when training finish. So, do same verification here.

Example:

avg_dic: dict = {} self.net.eval() for iteration, batch in enumerate(self.datasets.loader_valid, 1):

self.input, self.ground_truth = self.get_data_from_batch(batch, self.device) with torch.no_grad():

self.output = self.net(self.input) dic: dict = self.compute_valid()

if avg_dic == {}: avg_dic: dict = dic

else:

for key in dic.keys(): avg_dic[key] += dic[key]

for key in avg_dic.keys(): avg_dic[key] = avg_dic[key] / self.datasets.nsteps_valid

self.watcher.scalars(avg_dic, self.step, tag="Valid") self.loger.write(self.step, self.current_epoch, avg_dic, "Valid", header=self.step <= 1) self._watch_images(tag="Valid") self.net.train()

6.2.3 AutoEncoderTrainer

class jdit.trainer.**AutoEncoderTrainer**(*logdir*, *nepochs*, *gpu_ids*, *net*, *opt*, *datasets*)
this is a autoencoder-decoder trainer. Image to Image

compute_loss()
Compute the main loss and observed values.

Compute the loss and other values shown in tensorboard scalars visualization. You should return a main loss for doing backward propagation.

So, if you want some values visualized. Make a `dict()` with key name is the variable's name. The training logic is :

self.input, self.ground_truth = self.get_data_from_batch(batch, self.device) self.output = self.net(self.input) self._train_iteration(self.opt, self.compute_loss, csv_filename="Train")

So, you have *self.net*, *self.input*, *self.output*, *self.ground_truth* to compute your own loss here.

Note: Only the main loss will do backward propagation, which is the first returned variable. If you have the joint loss, please add them up and return one main loss.

Note: All of your variables in returned `dict()` will never do backward propagation with `model.train()`. However, It still compute grads, without using `with torch.autograd.no_grad()`. So, you can compute any grads variables for visualization.

Example:

```
var_dic = {}
var_dic["CEP"] = loss = nn.MSELoss(reduction="mean")(self.output, self.ground_
↪truth)
return loss, var_dic
```

compute_valid()
Compute the valid_epoch variables for visualization.

Compute the caring variables. For the caring variables will only be used in tensorboard scalars visualization. So, if you want some variables visualized. Make a `dict()` with key name is the variable's name.

Note: All of your variables in returned `dict()` will never do backward propagation with `model.eval()`. However, It still compute grads, without using `with torch.autograd.no_grad()`. So, you can compute some grads variables for visualization.

Example:: var_dic = {} var_dic["CEP"] = loss = nn.MSELoss(reduction="mean")(self.output, self.ground_truth) return var_dic

get_data_from_batch(*batch_data*, *device*)
If you have different behavior. You need to rewrite thisd method and the method *sllf.train_epoch()*

Parameters

- **batch_data** – A Tensor loads from dataset
- **device** – compute device

Returns Tensors,

valid_epoch()
Validate model each epoch.

It will be called each epoch, when training finish. So, do same verification here.

Example:

avg_dic: dict = {} self.net.eval() for iteration, batch in enumerate(self.datasets.loader_valid, 1):

self.input, self.ground_truth = self.get_data_from_batch(batch, self.device) with torch.no_grad():

self.output = self.net(self.input) dic: dict = self.compute_valid()

if avg_dic == {}: avg_dic: dict = dic

else:

for key in dic.keys(): avg_dic[key] += dic[key]

for key in avg_dic.keys(): avg_dic[key] = avg_dic[key] / self.datasets.nsteps_valid

self.watcher.scalars(avg_dic, self.step, tag="Valid") self.loger.write(self.step, self.current_epoch, avg_dic, "Valid", header=self.step <= 1) self._watch_images(tag="Valid") self.net.train()

6.3 Generative Adversarial Networks Trainer

6.3.1 SupGanTrainer

class jdit.trainer.**SupGanTrainer**(*logdir*, *nepochs*, *gpu_ids_abs*, *netG: jdit.model.Model*, *netD: jdit.model.Model*, *optG: jdit.optimizer.Optimizer*, *optD: jdit.optimizer.Optimizer*, *datasets: jdit.dataset.DataLoadersFactory*)

compute_d_loss() -> (*<Mock name='mock.Tensor' id='140594777874840'>*, *<class 'dict'>*)
Rewrite this method to compute your own loss Discriminator.

You should return a **loss** for the first position. You can return a `dict` of loss that you want to visualize on the second position.like

Example:

```
d_fake = self.netD(self.fake.detach())
d_real = self.netD(self.ground_truth)
var_dic = {}
var_dic["GP"] = gp = gradPenalty(self.netD, self.ground_truth, self.fake,␣
↪input=self.input,
                                 use_gpu=self.use_gpu)
var_dic["WD"] = w_distance = (d_real.mean() - d_fake.mean()).detach()
var_dic["LOSS_D"] = loss_d = d_fake.mean() - d_real.mean() + gp + sgp
return: loss_d, var_dic
```

compute_g_loss() -> (*<Mock name='mock.Tensor' id='140594777874840'>*, *<class 'dict'>*)
Rewrite this method to compute your own loss of Generator.

You should return a **loss** for the first position. You can return a `dict` of loss that you want to visualize on the second position.like

Example:

```
d_fake = self.netD(self.fake)
var_dic = {}
var_dic["JC"] = jc = jcbClamp(self.netG, self.input, use_gpu=self.use_gpu)
var_dic["LOSS_D"] = loss_g = -d_fake.mean() + jc
return: loss_g, var_dic
```

compute_valid() → dict
Rewrite this method to compute your validation values.

You can return a `dict` of validation values that you want to visualize.

Example:

```
# It will do the same thing as ``compute_g_loss()`` and ``self.compute_d_
↪loss()``
g_loss, _ = self.compute_g_loss()
d_loss, _ = self.compute_d_loss()
var_dic = {"LOSS_D": d_loss, "LOSS_G": g_loss}
return var_dic
```

d_turn = 1
The training times of Discriminator every ones Generator training.

get_data_from_batch(*batch_data: list, device: <Mock name='mock.device' id='140594778247688'>*)

Load and wrap data from the data lodaer.

Split your one batch data to specify variable.

Example:

```
# batch_data like this [input_Data, ground_truth_Data]
input_cpu, ground_truth_cpu = batch_data[0], batch_data[1]
# then move them to device and return them
return input_cpu.to(self.device), ground_truth_cpu.to(self.device)
```

Parameters

- **batch_data** – one batch data load from `DataLoader`
- **device** – A device variable. `torch.device`

Returns input Tensor, ground_truth Tensor

train_epoch(*subbar_disable=False*)

You get train loader and do a loop to deal with data.

> **Caution:** You must record your training step on `self.step` in your loop by doing things like this `self.step += 1`.

Example:

```
for iteration, batch in tqdm(enumerate(self.datasets.loader_train, 1)):
    self.step += 1
    self.input_cpu, self.ground_truth_cpu = self.get_data_from_batch(batch,␣
↪self.device)
    self._train_iteration(self.opt, self.compute_loss, tag="Train")
```

Returns

valid_epoch()

Validate model each epoch.

It will be called each epoch, when training finish. So, do same verification here.

Example:

```
avg_dic: dict = {}
self.netG.eval()
self.netD.eval()
# Load data from loader_valid.
for iteration, batch in enumerate(self.datasets.loader_valid, 1):
    self.input, self.ground_truth = self.get_data_from_batch(batch)
    with torch.no_grad():
        self.fake = self.netG(self.input)
        # You can write this function to apply your computation.
        dic: dict = self.compute_valid()
    if avg_dic == {}:
        avg_dic: dict = dic
    else:
```

(continues on next page)

(continued from previous page)

```
        for key in dic.keys():
            avg_dic[key] += dic[key]

for key in avg_dic.keys():
    avg_dic[key] = avg_dic[key] / self.datasets.nsteps_valid

self.watcher.scalars(avg_dic, self.step, tag="Valid")
self._watch_images(tag="Valid")
self.netG.train()
self.netD.train()
```

6.3.2 Pix2pixGanTrainer

class jdit.trainer.**Pix2pixGanTrainer**(*logdir*, *nepochs*, *gpu_ids_abs*, *netG*, *netD*, *optG*, *optD*, *datasets*)

compute_d_loss()

Rewrite this method to compute your own loss Discriminator.

You should return a **loss** for the first position. You can return a `dict` of loss that you want to visualize on the second position.like The training logic is :

self.input, self.ground_truth = self.get_data_from_batch(batch, self.device) self.fake = self.netG(self.input) self._train_iteration(self.optD, self.compute_d_loss, csv_filename="Train_D") if (self.step % self.d_turn) == 0:

self._train_iteration(self.optG, self.compute_g_loss, csv_filename="Train_G")

So, you use *self.input* , *self.ground_truth*, *self.fake*, *self.netG*, *self.optD* to compute loss. Example:

```
d_fake = self.netD(self.fake.detach())
d_real = self.netD(self.ground_truth)
var_dic = {}
var_dic["LS_LOSSD"] = loss_d = 0.5 * (torch.mean((d_real - 1) ** 2) + torch.
↪mean(d_fake ** 2))
return loss_d, var_dic
```

compute_g_loss()

Rewrite this method to compute your own loss of Generator.

You should return a **loss** for the first position. You can return a `dict` of loss that you want to visualize on the second position.like The training logic is :

self.input, self.ground_truth = self.get_data_from_batch(batch, self.device) self.fake = self.netG(self.input) self._train_iteration(self.optD, self.compute_d_loss, csv_filename="Train_D") if (self.step % self.d_turn) == 0:

self._train_iteration(self.optG, self.compute_g_loss, csv_filename="Train_G")

So, you use *self.input* , *self.ground_truth*, *self.fake*, *self.netG*, *self.optD* to compute loss. Example:

```
d_fake = self.netD(self.fake, self.input)
var_dic = {}
var_dic["LS_LOSSG"] = loss_g = 0.5 * torch.mean((d_fake - 1) ** 2)
return loss_g, var_dic
```

compute_valid()

Rewrite this method to compute valid_epoch values.

You can return a `dict` of values that you want to visualize.

Note: This method is under `torch.no_grad():`. So, it will never compute grad. If you want to compute grad, please use `torch.enable_grad():` to wrap your operations.

Example:

```
d_fake = self.netD(self.fake.detach())
d_real = self.netD(self.ground_truth)
var_dic = {}
var_dic["WD"] = w_distance = (d_real.mean() - d_fake.mean()).detach()
return var_dic
```

get_data_from_batch(*batch_data: list, device: <Mock name='mock.device' id='140594778247688'>*)

Load and wrap data from the data lodaer.

Split your one batch data to specify variable.

Example:

```
# batch_data like this [input_Data, ground_truth_Data]
input_cpu, ground_truth_cpu = batch_data[0], batch_data[1]
# then move them to device and return them
return input_cpu.to(self.device), ground_truth_cpu.to(self.device)
```

Parameters

- **batch_data** – one batch data load from `DataLoader`
- **device** – A device variable. `torch.device`

Returns input Tensor, ground_truth Tensor

test()

Test your model when you finish all epochs.

This method will call when all epochs finish.

Example:

```
for index, batch in enumerate(self.datasets.loader_test, 1):
    # For test only have input without groundtruth
    input = batch.to(self.device)
    self.netG.eval()
    with torch.no_grad():
        fake = self.netG(input)
    self.watcher.image(fake, self.current_epoch, tag="Test/fake", grid_
↪size=(4, 4), shuffle=False)
self.netG.train()
```

valid_epoch()

Validate model each epoch.

It will be called each epoch, when training finish. So, do same verification here.

Example:

```
avg_dic: dict = {}
self.netG.eval()
self.netD.eval()
# Load data from loader_valid.
for iteration, batch in enumerate(self.datasets.loader_valid, 1):
    self.input, self.ground_truth = self.get_data_from_batch(batch)
    with torch.no_grad():
        self.fake = self.netG(self.input)
        # You can write this function to apply your computation.
        dic: dict = self.compute_valid()
    if avg_dic == {}:
        avg_dic: dict = dic
    else:
        for key in dic.keys():
            avg_dic[key] += dic[key]

for key in avg_dic.keys():
    avg_dic[key] = avg_dic[key] / self.datasets.nsteps_valid

self.watcher.scalars(avg_dic, self.step, tag="Valid")
self._watch_images(tag="Valid")
self.netG.train()
self.netD.train()
```

6.3.3 GenerateGanTrainer

class jdit.trainer.**GenerateGanTrainer**(*logdir*, *nepochs*, *gpu_ids_abs*, *netG*, *netD*, *optG*, *optD*, *datasets*, *latent_shape*)

compute_d_loss()

Rewrite this method to compute your own loss Discriminator.

You should return a **loss** for the first position. You can return a `dict` of loss that you want to visualize on the second position.like The train logic is :

self.input, self.ground_truth = self.get_data_from_batch(batch, self.device) self.fake = self.netG(self.input) self._train_iteration(self.optD, self.compute_d_loss, csv_filename="Train_D") if (self.step % self.d_turn) == 0:

self._train_iteration(self.optG, self.compute_g_loss, csv_filename="Train_G")

So, you use *self.input* , *self.ground_truth*, *self.fake*, *self.netG*, *self.optD* to compute loss. Example:

```
d_fake = self.netD(self.fake.detach())
d_real = self.netD(self.ground_truth)
var_dic = {}
var_dic["LS_LOSSD"] = loss_d = 0.5 * (torch.mean((d_real - 1) ** 2) + torch.
↪mean(d_fake ** 2))
return loss_d, var_dic
```

compute_g_loss()

Rewrite this method to compute your own loss of Generator.

You should return a **loss** for the first position. You can return a `dict` of loss that you want to visualize on the second position.like The train logic is :

self.input, self.ground_truth = self.get_data_from_batch(batch, self.device) self.fake = self.netG(self.input) self._train_iteration(self.optD, self.compute_d_loss,

csv_filename="Train_D") if (self.step % self.d_turn) == 0:

self._train_iteration(self.optG, self.compute_g_loss, csv_filename="Train_G")

So, you use *self.input* , *self.ground_truth*, *self.fake*, *self.netG*, *self.optD* to compute loss. Example:

```
d_fake = self.netD(self.fake, self.input)
var_dic = {}
var_dic["LS_LOSSG"] = loss_g = 0.5 * torch.mean((d_fake - 1) ** 2)
return loss_g, var_dic
```

compute_valid()

The train logic is : self.input, self.ground_truth = self.get_data_from_batch(batch, self.device) self.fake = self.netG(self.input) self._train_iteration(self.optD, self.compute_d_loss, csv_filename="Train_D") if (self.step % self.d_turn) == 0:

self._train_iteration(self.optG, self.compute_g_loss, csv_filename="Train_G")

So, you use *self.input* , *self.ground_truth*, *self.fake*, *self.netG*, *self.optD* to compute validations.

Returns

d_turn = 1

The training times of Discriminator every ones Generator training.

get_data_from_batch(*batch_data: list*, *device: <Mock name='mock.device' id='140594778247688'>*)

Load and wrap data from the data lodaer.

Split your one batch data to specify variable.

Example:

```
# batch_data like this [input_Data, ground_truth_Data]
input_cpu, ground_truth_cpu = batch_data[0], batch_data[1]
# then move them to device and return them
return input_cpu.to(self.device), ground_truth_cpu.to(self.device)
```

Parameters

- **batch_data** – one batch data load from `DataLoader`
- **device** – A device variable. `torch.device`

Returns input Tensor, ground_truth Tensor

valid_epoch()

Validate model each epoch.

It will be called each epoch, when training finish. So, do same verification here.

Example:

```
avg_dic: dict = {}
self.netG.eval()
self.netD.eval()
# Load data from loader_valid.
for iteration, batch in enumerate(self.datasets.loader_valid, 1):
    self.input, self.ground_truth = self.get_data_from_batch(batch)
    with torch.no_grad():
        self.fake = self.netG(self.input)
```

(continues on next page)

(continued from previous page)

```
        # You can write this function to apply your computation.
        dic: dict = self.compute_valid()
    if avg_dic == {}:
        avg_dic: dict = dic
    else:
        for key in dic.keys():
            avg_dic[key] += dic[key]

for key in avg_dic.keys():
    avg_dic[key] = avg_dic[key] / self.datasets.nsteps_valid

self.watcher.scalars(avg_dic, self.step, tag="Valid")
self._watch_images(tag="Valid")
self.netG.train()
self.netD.train()
```

6.4 instances

6.4.1 instances.FashionClassTrainer

jdit.trainer.instances.**start_fashionClassTrainer**(*gpus=()*, *nepochs=10*, *run_type='train'*)

” An example of fashion-mnist classification

CHAPTER 7

jdit.assessment

7.1 FID

jdit.assessment.**FID_score**(*source*, *target*, *sample_prop=1.0*, *gpu_ids=()*, *dim=2048*, *batch-size=128*, *verbose=True*)

Compute FID score from `Tensor`, `DataLoader` or a directory‘‘path‘‘.

Parameters

- **source** – source data.
- **target** – target data.
- **sample_prop** – If passing a `Tensor` source, set this rate to sample a part of data from source.
- **gpu_ids** – gpu ids.
- **dim** – The number of features. Three options available.
  - 64: The first max pooling features of Inception.
  - 192: The Second max pooling features of Inception.
  - 768: The Pre-aux classifier features of Inception.
  - 2048: The Final average pooling features of Inception.

  Default: 2048.
- **batchsize** – Only using for passing paths of source and target.
- **verbose** – If show processing log.

Returns fid score

Attention: If you are passing `Tensor` as source and target. Make sure you have enough memory to load these data in _InceptionV3. Otherwise, please passing `path` of `DataLoader` to compute them step by step.

Example:

```
>>> from jdit.dataset import Cifar10
>>> loader = Cifar10(root=r"../../datasets/cifar10", batch_shape=(32, 3, 32, 32))
>>> target_tensor = loader.samples_train[0]
>>> source_tensor = loader.samples_valid[0]
>>> # using Tensor to compute FID score
>>> fid_value = FID_score(source_tensor, target_tensor, sample_prop=0.01,␣
↪depth=768)
>>> print('FID: ', fid_value)
>>> # using DataLoader to compute FID score
>>> fid_value = FID_score(loader.loader_test, loader.loader_valid, depth=768)
>>> print('FID: ', fid_value)
```

CHAPTER 8

jdit.parallel

8.1 SupParallelTrainer

class jdit.parallel.**SupParallelTrainer**(*unfixed_params_list: list*, *train_func=None*)

Training parallel

Parameters

- **default_params** – a `dict()` like `{param_1:d1, param_2:d2 ...}`
- **unfixed_params_list** – a `list` like `[{param_1:a1, param_2:a2}, {param_1:b1, param_2:b2}, ...]`.

Note: You must set the value of `task_id` and `gpu_ids_abs`, regardless in `default_params` or `unfixed_params_list`.

`{'task_id': 1`},{'gpu_ids_abs': [0,1]}.

- For the same `task_id` , the tasks will be executed **sequentially** on the certain devices.
- For the different `task_id` , the will be executed **parallelly** on the certain devices.

Example:

```
unfixed_params_list = [
    {'task_id':1, 'lr':1e-3,'gpu_ids_abs': [0] },
    {'task_id':1, 'lr':1e-4,'gpu_ids_abs': [0] },
    {'task_id':2, 'lr':1e-5,'gpu_ids_abs': [2,3] }]
```

This set of `unfixed_params_list` means that:

time	'task_id':1	'task_id':2	
t	'lr':1e-3, 'gpu_ids_abs': [0]	'lr':1e-5, 'gpu_ids_abs': [2,3]	executed parallelly
t+1	'lr':1e-4, 'gpu_ids_abs': [0]		
	executed sequentially		

build_task_trainer(*unfixed_params: dict*)

You need to write this method to build your own `Trainer`.

This will run in a certain subprocess. The keys of `params` are compatible with `dataset` , `Model` , `Optimizer` and `Trainer` . You can see parameters in the following example.

These two parameters are special.

- `params["logdir"]` controls the log directory.
- `params["gpu_ids_abs"]` controls the running devices.

You should return a `Trainer` when you finish you building.

Parameters **params** – parameters dictionary.

Returns Trainer

Example:

```
# Using ``params['key']`` to build your Trainer.
logdir = params["logdir"] # necessary!
gpu_ids_abs = params["gpu_ids_abs"] # necessary!
use_benchmark = params["use_benchmark"]
data_root = params["data_root"]
batch_shape = params["batch_shape"]
opt_name = params["opt_name"]
lr = params["lr"]
lr_decay = params["lr_decay"]
lr_minimum = params["lr_minimum"]
weight_decay = params["weight_decay"]
momentum = params["momentum"]
betas = params["betas"]
init_method = params["init_method"]
depth = params["depth"]
mid_channels = params["mid_channels"]
nepochs = params["nepochs"]

torch.backends.cudnn.benchmark = use_benchmark
mnist = FashionMNIST(root=data_root, batch_shape=batch_shape)
T_net = Model(Tresnet18(depth=depth, mid_channels=mid_channels), gpu_ids_
↪abs=gpu_ids_abs,
              init_method=init_method)
opt = Optimizer(T_net.parameters(), lr, lr_decay, weight_decay, momentum,␣
↪betas, opt_name,
                lr_minimum=lr_minimum)
Trainer = FashingClassTrainer(logdir, nepochs, gpu_ids_abs, T_net, opt, mnist)
# You must return a Trainer!
return Trainer
```

error(*msg*)

When a subprocess failed, it will be called.

You can rewrite this method for your purpose. :param msg: error massage

finish(*msg*)

When a subprocess finished, it will be called.

You can rewrite this method for your purpose. :param msg: fin

train(*max_processes=4*)

start parallel task

To start the parallel task that were saved in `self.parallel_plans` dictionary.

Parameters **`max_processes`** – A max amount of processes for setting `Pool(processes = ?)` method.

Jdit is a research processing oriented framework based on pytorch. Only care about your ideas. You don't need to build a long boring code to run a deep learning project to verify your ideas.

You only need to implement you ideas and don't do anything with training framework, multiply-gpus, checkpoint, process visualization, performance evaluation and so on.

CHAPTER 9

Quick start

After building and installing jdit package, you can make a new directory for a quick test. Assuming that you get a new directory *example*. run this code in *ipython* cmd.(Create a *main.py* file is also acceptable.)

```
from jdit.trainer.instances.fashionClassification
import start_fashionClassTrainer
start_fashionClassTrainer()
```

Then you will see something like this as following.

```
===> Build dataset
use 8 thread
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-
↪idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-
↪idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-
↪idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-
↪idx1-ubyte.gz
Processing...
Done
===> Building model
ResNet Total number of parameters: 2776522
ResNet model use CPU
apply kaiming weight init
===> Building optimizer
===> Training
using `tensorboard --logdir=log` to see learning curves and net structure.
training and valid_epoch data, configures info and checkpoint were save in `log`␣
↪directory.
  0%|          | 0/10 [00:00<.., ..epoch/s]
0step [00:00, step/s]
```

- It will search a fashion mnist dataset.

- Then build a resnet18 for classification.
- For training process, you can find learning curves in *tensorboard*.
- It will create a *log* directory in *example/*, which saves training processing data and configures.

Although it is just an example, you still can build your own project easily by using jdit framework. Jdit framework can deal with * Data visualization. (learning curves, images in pilot process) * CPU, GPU or GPUs. (Training your model on specify devices) * Intermediate data storage. (Saving training data into a csv file) * Model checkpoint automatically. * Flexible templates can be used to integrate and custom overrides. So, let's see what is **jdit**.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index