LibMTL

Baijiong Lin and Yu Zhang

Sep 16, 2023

GETTING STARTED:

CHAPTER

ONE

INTRODUCTION

LibMTL is an open-source library built on PyTorch for Multi-Task Learning (MTL). This library has the following three characteristics.

- **Unified**: LibMTL provides a unified code base to implement and a consistent evaluation procedure including data processing, metric objectives, and hyper-parameters on several representative MTL benchmark datasets, which allows quantitative, fair, and consistent comparisons between different MTL algorithms.
- **Comprehensive**: LibMTL supports 84 MTL models combined by 7 architectures and 12 loss weighting strategies. Meanwhile, LibMTL provides a fair comparison on 3 computer vision datasets.
- **Extensible**: LibMTL follows the modular design principles, which allows users to flexibly and conveniently add customized components or make personalized modifications. Therefore, users can easily and fast develop novel loss weighting strategies and architectures or apply the existing MTL algorithms to new application scenarios with the support of LibMTL.

1.1 Supported Algorithms

LibMTL currently supports the following algorithms:

- 12 loss weighting strategies.
- 7 architectures.
- 84 combinations of different architectures and loss weighting strategies.

1.2 Citation

If you find LibMTL useful for your research or development, please cite the following:

```
@article{LibMTL,
  title={LibMTL: A Python Library for Multi-Task Learning},
  author={Baijiong Lin and Yu Zhang},
  journal={arXiv preprint arXiv:2203.14338},
  year={2022}
}
```

1.3 Contributors

`LibMTL` is developed and maintained by Baijiong Lin and Yu Zhang.

1.4 Contact Us

If you have any question or suggestion, please feel free to contact us by raising an issue or sending an email to `bj.lin.email@gmail.com`.

1.5 Acknowledgements

We would like to thank the authors that release the public repositories (listed alphabetically): CAGrad, dselect_k_moe, MultiObjectiveOptimization, and mtan.

CHAPTER

TWO

INSTALLATION

2.1 Dependencies

To install `LibMTL`, you need to setup the following libraries:

- Python >= 3.7
- torch >= 1.8.0
- torchvision >= 0.9.0
- numpy >= 1.20

2.2 User Installation

- Create a virtual environment

```
conda create -n libmtl python=3.8
conda activate libmtl
pip install torch==1.8.0 torchvision==0.9.0 numpy==1.20
```

- Clone the repository

```
git clone https://github.com/median-research-group/LibMTL.git
```

- Install `LibMTL`

```
pip install -e .
```

CHAPTER

THREE

QUICK START

We use the NYUv2 dataset [1] as an example to show how to use `LibMTL`. More details and results are provided here.

3.1 Download Dataset

The NYUv2 dataset we used is pre-processed by mtan. You can download this dataset here. The directory structure is as follows:

```
*/nyuv2/
├── train
│   ├── depth
│   ├── image
│   ├── label
│   └── normal
└── val
    ├── depth
    ├── image
    ├── label
    └── normal
```

The NYUv2 dataset is a MTL benchmark dataset, which includes three tasks: 13-class semantic segmentation, depth estimation, and surface normal prediction. `image` contains the input images and `label`, `depth`, `normal` contains the labels for three tasks, respectively. We train the MTL model with the data in `train` and evaluate on `val`.

3.2 Run a Model

The complete training code for the NYUv2 dataset is provided in examples/nyu. The file `train_nyu.py` is the main file for training on the NYUv2 dataset.

You can find the command-line arguments by running the following command.

```
python train_nyu.py -h
```

For instance, running the following command will train a MTL model with *LibMTL.weighting.EW* and *LibMTL.architecture.HPS* on NYUv2 dataset.

```
python train_nyu.py --weighting EW --arch HPS --dataset_path /path/to/nyuv2 --gpu_id 0 --
↪scheduler step
```

If everything works fine, you will see the following outputs which includes the training configurations and the number of model parameters.

```
========================================
General Configuration:
        Wighting: EW
        Architecture: HPS
        Rep_Grad: False
        Multi_Input: False
        Seed: 0
        Device: cuda:0
Optimizer Configuration:
        optim: adam
        lr: 0.0001
        weight_decay: 1e-05
Scheduler Configuration:
        scheduler: step
        step_size: 100
        gamma: 0.5
========================================
Total Params: 71888721
Trainable Params: 71888721
Non-trainable Params: 0
========================================
```

Next, the results will be printed in following format.

```
LOG FORMAT | segmentation_LOSS mIoU pixAcc | depth_LOSS abs_err rel_err | normal_LOSS␣
↪mean median <11.25 <22.5 <30 | TIME
Epoch: 0000 | TRAIN: 1.4417 0.2494 0.5717 | 1.4941 1.4941 0.5002 | 0.3383 43.1593 38.
↪2601 0.0913 0.2639 0.3793 | Time: 81.6612 | TEST: 1.0898 0.3589 0.6676 | 0.7027 0.7027␣
↪0.2615 | 0.2143 32.8732 29.4323 0.1734 0.3878 0.5090 | Time: 11.9699
Epoch: 0001 | TRAIN: 0.8958 0.4194 0.7201 | 0.7011 0.7011 0.2448 | 0.1993 31.5235 27.
↪8404 0.1826 0.4060 0.5361 | Time: 82.2399 | TEST: 0.9980 0.4189 0.6868 | 0.6274 0.6274␣
↪0.2347 | 0.1991 31.0144 26.5077 0.2065 0.4332 0.5551 | Time: 12.0278
```

If the training process ends, the best result on `val` will be printed as follows.

```
Best Result: Epoch 65, result {'segmentation': [0.5377492904663086, 0.7544658184051514],
↪'depth': [0.38453552363844823, 0.1605487049810748], 'normal': [23.573742, 17.04381, 0.
↪35038458555943763, 0.609274380451927, 0.720717279583373]}
```

3.3 References

CHAPTER

FOUR

WHAT IS MULTI-TASK LEARNING?

Multi-Task Learning (MTL) is an active research field in machine learning. It is a learning paradigm which aims to jointly learn several related tasks to improve their generalization performance by leveraging common knowledge among them. In recent years, many researchers have successfully applied MTL to different fields such as computer vision, natural language processing, reinforcement learning, recommendation system and so on.

The recent studies of MTL mainly focus on two perspectives, network architecture design and loss weighting. We implement some general and representative methods in `LibMTL`.

For more relevant introduction, please refer to [1, 2, 3, 4].

4.1 Network Architecture

In the design of network architectures, the simplest and most popular method is the hard parameter sharing (HPS, *LibMTL.architecture.HPS*), as shown in Fig. 4.1, where an encoder is shared among all the tasks and each task has its own specific decoder. Since most of the parameters are shared among tasks, such architecture easily causes negative sharing when tasks are not related enough. To better deal with task relationships, different MTL architectures have been proposed. `LibMTL` supports several state-of-the-art architectures, please refer to *LibMTL.architecture* for details.

There are usually two types of MTL problems: the single-input problem and the multi-input problem. The single-input problem, as shown in the left of Fig. 4.1, means an input data has an output for each task or equivalently all tasks share the input data. The NYUv2 dataset is an example of this problem. The multi-input problem, as shown in the right of Fig. 4.1, indicates each task has its own input data. The Office-31 and Office-Home datasets belong to such problem. `LibMTL` has unified these two cases in a training framework and you just need to set the command-line argument `multi_input` correctly.

4.2 Weighting Strategy

Balancing multiple losses corresponding to multiple tasks is another way to deal with task relationships since the shared parameters are updated by all the task losses. Thus, different methods have been proposed to balance losses or gradients. `LibMTL` supports several state-of-the-art weighting strategies, please see *LibMTL.weighting* for details.

Some gradient balancing methods such as MGDA (*LibMTL.weighting.MGDA*) need to compute the gradient for each task first and then calculate the aggregated gradient in various ways. To reduce the computational cost, it can use the gradients of the representations after the encoder (abbreviated as rep-grad) to approximate the gradients of shared parameters (abbreviated as param-grad).

The PyTorch implemention of rep-grad is shown in Fig. 4.2. We need to separate the computational graph into two parts by the `detach` operation. `LibMTL` has unified the two cases in a training framework and you just need to set the command-line argument `rep_grad` correctly. Besides, the argument `rep_grad` does not conflict with `multi_input`.

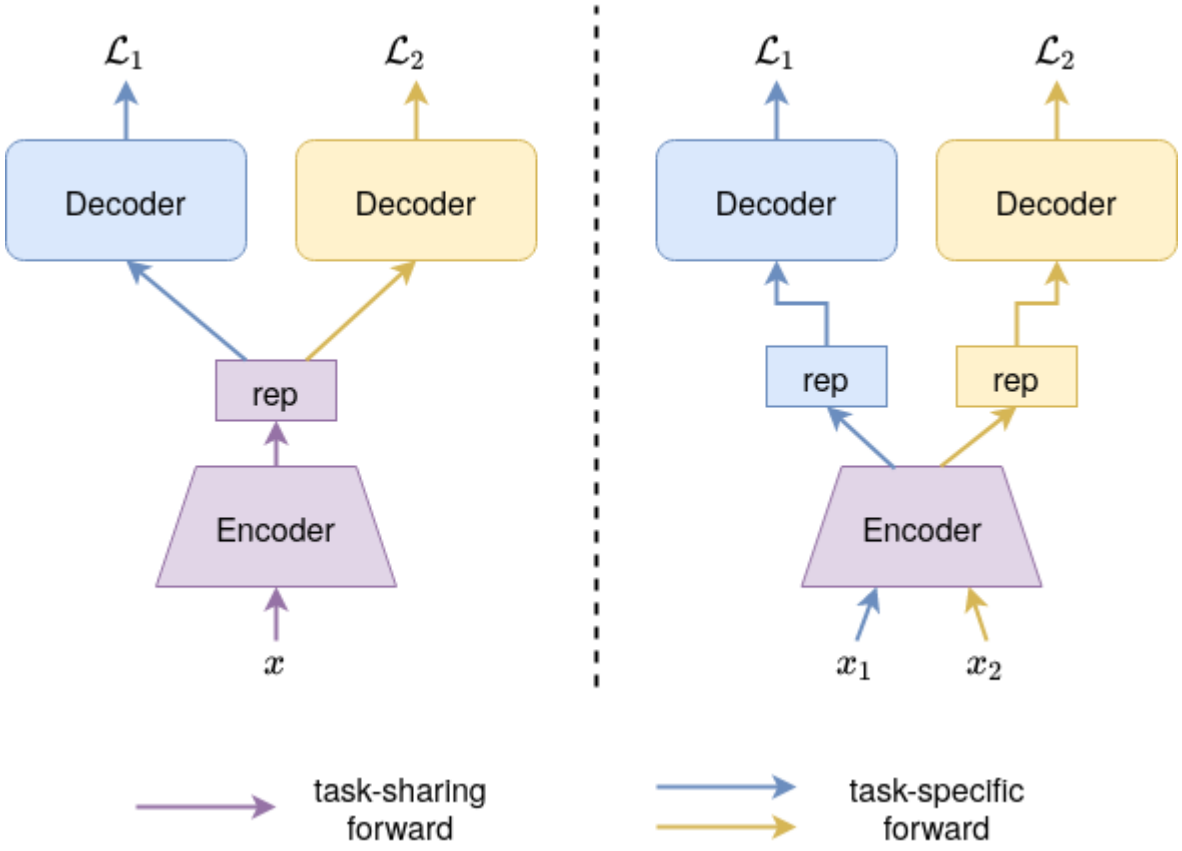


Fig. 4.1: An illustration of the single-input problem (left) and the multi-input problem (right), using hard parameter sharing pattern as an example.

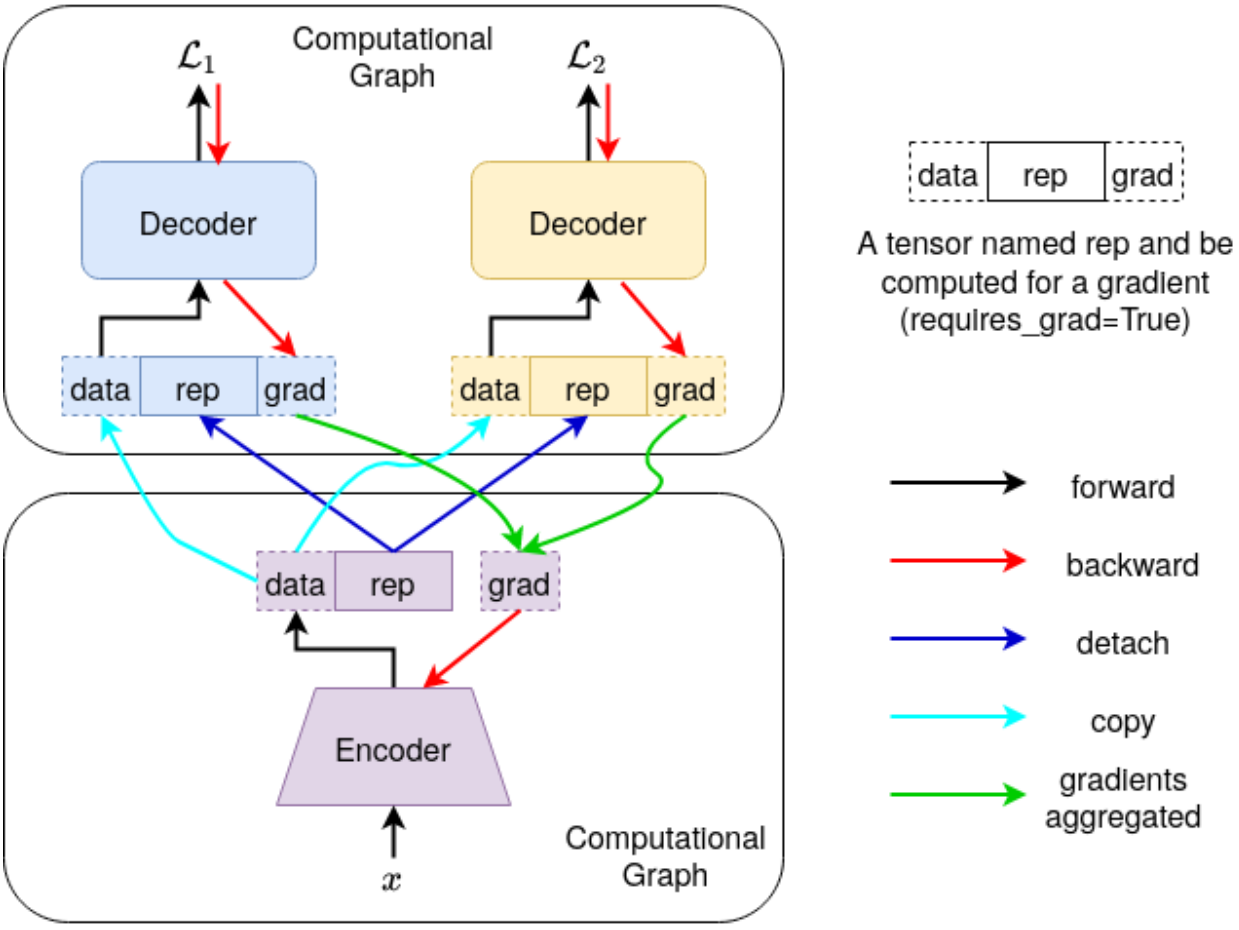


Fig. 4.2: An illustration of how to compute the gradient for representation.

4.3 References

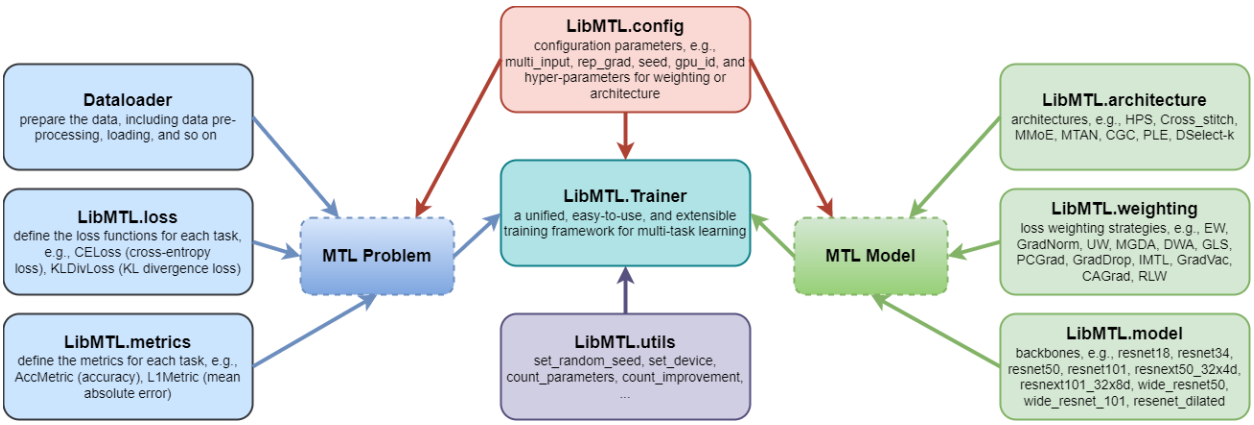
CHAPTER

FIVE

OVERALL FRAMEWORK

`LibMTL` provides a unified framework to train a MTL model with several architectures and weighting strategies on benchmark datasets. The overall framework consists of nine modules as introduced below.

- The Dataloader module is responsible for data pre-processing and loading.
- The LibMTL.loss module defines loss functions for each task.
- The LibMTL.metrics module defines evaluation metrics for all the tasks.
- The LibMTL.config module is responsible for all the configuration parameters involved in the training process, such as the corresponding MTL setting (i.e. the multi-input case or not), the potential hyper-parameters of loss weighting strategies and architectures, the training configuration (e.g., the batch size, the running epoch, the random seed, and the learning rate), and so on. This module adopts command-line arguments to enable users to conveniently set those configuration parameters.
- The LibMTL.Trainer module provides a unified framework for the training process under different MTL settings and for different MTL approaches
- The LibMTL.utils module implements some useful functionalities for the training process such as calculating the total number of parameters in an MTL model.
- The LibMTL.architecture module contains the implementations of various architectures in MTL.
- The LibMTL.weighting module contains the implementations of various loss weighting strategies in MTL.
- The LibMTL.model module includes some popular backbone networks (e.g., ResNet).



CHAPTER

SIX

RUN A BENCHMARK

Here we introduce some MTL benchmark datasets and show how to run models on them for a fair comparison.

6.1 NYUv2

The NYUv2 dataset [1] is an indoor scene understanding dataset, which consists of video sequences recorded by the RGB and Depth cameras in the Microsoft Kinect. It contains 795 and 654 images with ground-truths for training and validation, respectively.

We use the pre-processed NYUv2 dataset in [2], which can be downloaded here. Each input image has been resized to 3x288x384 and has labels for three tasks, including 13-class semantic segmentation, depth estimation, and surface normal prediction. Thus, it is a single-input problem, which means `multi_input` must be `False`.

The training codes are mainly modified from mtan and available in `examples/nyu`. We use DeepLabV3+ architecture [3], where a ResNet-50 network pretrained on the ImageNet dataset with dilated convolutions [4] is used as a shared encoder among tasks and the Atrous Spatial Pyramid Pooling (ASPP) module [3] is used as task-specific head for each task.

Following [2], the evaluation metrics of three tasks are adopted as follows. Mean Intersection over Union (mIoU) and Pixel Accuracy (Pix Acc) are used for the semantic segmentation task. Absolute and relative errors (denoted by Abs Err and Rel Err) are used for the depth estimation task. Five metrics are used for the surface normal estimation task: mean absolute of the error (Mean), median absolute of the error (Median), and percentage of pixels with the angular error below a threshold ϵ with ϵ as $11.25°, 22.5°, 30°$ (abbreviated as <11.25, <22.5, <30), respectively. Among them, higher scores of mIoU, Pix Acc, <11.25, <22.5, and <30 mean better performance and lower scores of Abs Err, Rel Err, Mean, and Median indicate better performance.

6.1.1 Run a Model

The script `train_nyu.py` is the main file for training and evaluating an MTL model on the NYUv2 dataset. A set of command-line arguments is provided to allow users to adjust the training configuration.

Some important arguments are described as follows.

- `weighting`: The weighting strategy. Refer to here.
- `arch`: The MTL architecture. Refer to here.
- `gpu_id`: The id of gpu. The default value is '0'.
- `seed`: The random seed for reproducibility. The default value is 0.
- `scheduler`: The type of the learning rate scheduler. We recommend to use 'step' here.
- `optim`: The type of the optimizer. We recommend to use 'adam' here.

- `dataset_path`: The path of the NYUv2 dataset.
- `aug`: If `True`, the model is trained with a data augmentation.
- `train_bs`: The batch size of training data. The default value is 8.
- `test_bs`: The batch size of test data. The default value is 8.

The complete command-line arguments and their descriptions can be found by running the following command.

```
python train_nyu.py -h
```

If you understand those command-line arguments, you can train an MTL model by executing the following command.

```
python train_nyu.py --weighting WEIGHTING --arch ARCH --dataset_path PATH/nyuv2 --gpu_id␣
↪GPU_ID --scheduler step
```

6.1.2 References

6.2 Office-31 and Office-Home

The Office-31 dataset [1] consists of three classification tasks on three domains: Amazon, DSLR, and Webcam, where each task has 31 object categories. It can be download here. This dataset contains 4,110 labeled images and we randomly split these samples, with 60% for training, 20% for validation, and the rest 20% for testing.

The Office-Home dataset [2] has four classification tasks on four domains: Artistic images (abbreviated as Art), Clip art, Product images, and Real-world images. It can be download here. This dataset has 15,500 labeled images in total and each domain contains 65 classes. We divide the entire data into the same proportion as the Office-31 dataset.

Both datasets belong to the multi-input setting in MTL. Thus, the `multi_input` must be `True` for both of the two office datasets.

The training codes are available in `examples/office`. We use the ResNet-18 network pretrained on the ImageNet dataset followed by a fully connected layer as a shared encoder among tasks and a fully connected layer is applied as a task-specific output layer for each task. All the input images are resized to 3x224x224.

6.2.1 Run a Model

The script `train_office.py` is the main file for training and evaluating a MTL model on the Office-31 or Office-Home dataset. A set of command-line arguments is provided to allow users to adjust the training parameter configuration.

Some important arguments are described as follows.

- `weighting`: The weighting strategy. Refer to here.
- `arch`: The MTL architecture. Refer to here.
- `gpu_id`: The id of gpu. The default value is '0'.
- `seed`: The random seed for reproducibility. The default value is 0.
- `optim`: The type of the optimizer. We recommend to use 'adam' here.
- `dataset`: Training on Office-31 or Office-Home. Options: 'office-31', 'office-home'.
- `dataset_path`: The path of the Office-31 or Office-Home dataset.

- `bs`: The batch size of training, validation, and test data. The default value is 64.

The complete command-line arguments and their descriptions can be found by running the following command.

```
python train_office.py -h
```

If you understand those command-line arguments, you can train a MTL model by running a command like this.

```
python train_office.py --weighting WEIGHTING --arch ARCH --dataset_path PATH --gpu_id␣
↪GPU_ID --multi_input
```

6.2.2 References

CHAPTER

SEVEN

APPLY TO A NEW DATASET

Here we would like to introduce how to apply `LibMTL` to a new dataset.

7.1 Define a MTL problem

Firstly, you need to know the type of this MTL problem (i.e. a single-input problem or a multi-input problem, refer to here) and the information of each task, including the task's name, evaluation metrics, loss functions, and indicators determined whether the higher the metric score is, the better the performance is.

The `multi_input` is a command-line argument and all tasks' information needs to be defined as a dictionary. `LibMTL` provides some common loss functions and metrics, and refer to *LibMTL.loss* and *LibMTL.metrics*, respectively. Some examples are listed as follows.

7.1.1 Example 1 (The Office-31 Dataset)

```
from LibMTL.loss import CELoss
from LibMTL.metrics import AccMetric

# define tasks
task_name = ['amazon', 'dslr', 'webcam']
task_dict = {task: {'metrics': ['Acc'],
                    'metrics_fn': AccMetric(),
                    'loss_fn': CELoss(),
                    'weight': [1]} for task in task_name}
```

Besides, `LibMTL` also supports to customize new losses and metrics. For example, if we would like to develop the metric classes for the segmentation task on the NYUv2 dataset, we need to inherit *LibMTL.metrics.AbsMetric* and rewrite the corresponding methods like `update_fun()`, `score_fun()`, and `reinit()`. Please see *LibMTL.metrics.AbsMetric* for details. The loss class for segmentation is customized similarly. Please refer to *LibMTL.loss.AbsLoss* for details.

7.1.2 Example 2 (The NYUv2 Dataset)

```
from LibMTL.metrics import AbsMetric

# seg
class SegMetric(AbsMetric):
    def __init__(self):
        super(SegMetric, self).__init__()

        self.num_classes = 13
        self.record = torch.zeros((self.num_classes, self.num_classes), dtype=torch.
↪int64)

    def update_fun(self, pred, gt):
        self.record = self.record.to(pred.device)
        pred = pred.softmax(1).argmax(1).flatten()
        gt = gt.long().flatten()
        k = (gt >= 0) & (gt < self.num_classes)
        inds = self.num_classes * gt[k].to(torch.int64) + pred[k]
        self.record += torch.bincount(inds, minlength=self.num_classes**2).reshape(self.
↪num_classes, self.num_classes)

    def score_fun(self):
        h = self.record.float()
        iu = torch.diag(h) / (h.sum(1) + h.sum(0) - torch.diag(h))
        acc = torch.diag(h).sum() / h.sum()
        return [torch.mean(iu).item(), acc.item()]

    def reinit(self):
        self.record = torch.zeros((self.num_classes, self.num_classes), dtype=torch.
↪int64)
```

The customized loss and metric classes of three tasks on the NYUv2 dataset are put in `examples/nyu/utils.py`. After that, the three-task MTL problem on the NYUv2 dataset is defined as follows.

```
from utils import *

# define tasks
task_dict = {'segmentation': {'metrics':['mIoU', 'pixAcc'],
                              'metrics_fn': SegMetric(),
                              'loss_fn': SegLoss(),
                              'weight': [1, 1]},
             'depth': {'metrics':['abs_err', 'rel_err'],
                       'metrics_fn': DepthMetric(),
                       'loss_fn': DepthLoss(),
                       'weight': [0, 0]},
             'normal': {'metrics':['mean', 'median', '<11.25', '<22.5', '<30'],
                        'metrics_fn': NormalMetric(),
                        'loss_fn': NormalLoss(),
                        'weight': [0, 0, 1, 1, 1]}}
```

7.2 Prepare Dataloaders

Secondly, you need to prepare the dataloaders with a correct format. For a multi-input problem like the Office-31 dataset, each task has its own dataloader and all dataloaders are put in a dictionary with the task names as the corresponding keys.

7.2.1 Example 1 (The Office-31 Dataset)

```
train_dataloaders = {'amazon': amazon_dataloader,
                     'dslr': dslr_dataloader,
                     'webcam': webcam_dataloader}
```

For single-input problem like the NYUv2 dataset, all tasks share a common dataloader, which outputs a list in every iteration. The first element of this list is the input data tensor and the second is a dictionary of the label tensors with the task names as the corresponding keys. An example is shown as follows.

7.2.2 Example 2 (The NYUv2 Dataset)

```
nyuv2_train_loader = xx
# print(iter(nyuv2_train_loader).next())
# [torch.Tensor, {'segmentation': torch.Tensor,
#                 'depth': torch.Tensor,
#                 'normal': torch.Tensor}]
```

7.3 Define Encoder and Decoders

Thirdly, you need to define the shared encoder and task-specific decoders. `LibMTL` provides some neural networks like ResNet-based network. Please see *`LibMTL.model`* for details. Also, you can customize the encoder and decoders.

Note that the encoder does not be instantiated while the decoders should be instantiated.

7.3.1 Example 1 (The Office-31 Dataset)

```
import torch
import torch.nn as nn
from LibMTL.model import resnet18

# define encoder and decoders
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        hidden_dim = 512
        self.resnet_network = resnet18(pretrained=True)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.hidden_layer_list = [nn.Linear(512, hidden_dim),
                                  nn.BatchNorm1d(hidden_dim), nn.ReLU(), nn.Dropout(0.5)]
        self.hidden_layer = nn.Sequential(*self.hidden_layer_list)
```

(continues on next page)

(continued from previous page)

```
        # initialization
        self.hidden_layer[0].weight.data.normal_(0, 0.005)
        self.hidden_layer[0].bias.data.fill_(0.1)

    def forward(self, inputs):
        out = self.resnet_network(inputs)
        out = torch.flatten(self.avgpool(out), 1)
        out = self.hidden_layer(out)
        return out

decoders = nn.ModuleDict({task: nn.Linear(512, class_num) for task in task_name})
```

If the customized encoder is a ResNet-based network and you would like to use *LibMTL.architecture.MTAN*, please make sure that the encoder has an attribute named `resnet_network` corresponding to the ResNet network.

7.3.2 Example 2 (The NYUv2 Dataset)

```
from aspp import DeepLabHead
from LibMTL.model import resnet_dilated

# define encoder and decoders
def encoder_class():
        return resnet_dilated('resnet50')
num_out_channels = {'segmentation': 13, 'depth': 1, 'normal': 3}
decoders = nn.ModuleDict({task: DeepLabHead(encoder.feature_dim,
                                            num_out_channels[task]) for task in␣
↪list(task_dict.keys())})
```

7.4 Instantiate the Training Framework

Fourthly, you need to instantiate the training framework. Please see *LibMTL.Trainer* for more details.

7.4.1 Example 1 (The Office-31 Dataset)

```
from LibMTL import Trainer

officeModel = Trainer(task_dict=task_dict,
                      weighting=weighting_method.__dict__[params.weighting],
                      architecture=architecture_method.__dict__[params.arch],
                      encoder_class=Encoder,
                      decoders=decoders,
                      rep_grad=params.rep_grad,
                      multi_input=params.multi_input,
                      optim_param=optim_param,
                      scheduler_param=scheduler_param,
                      **kwargs)
```

Also, you can inherit the *LibMTL.Trainer* class and rewrite some functions like `process_preds()`.

7.4.2 Example 2 (The NYUv2 Dataset)

```
from LibMTL import Trainer

class NYUtrainer(Trainer):
    def __init__(self, task_dict, weighting, architecture, encoder_class,
                 decoders, rep_grad, multi_input, optim_param, scheduler_param,␣
↪**kwargs):
        super(NYUtrainer, self).__init__(task_dict=task_dict,
                                         weighting=weighting_method.__dict__[weighting],
                                         architecture=architecture_method.__dict__
↪[architecture],
                                         encoder_class=encoder_class,
                                         decoders=decoders,
                                         rep_grad=rep_grad,
                                         multi_input=multi_input,
                                         optim_param=optim_param,
                                         scheduler_param=scheduler_param,
                                         **kwargs)

    def process_preds(self, preds):
        img_size = (288, 384)
        for task in self.task_name:
            preds[task] = F.interpolate(preds[task], img_size, mode='bilinear', align_
↪corners=True)
        return preds

NYUmodel = NYUtrainer(task_dict=task_dict,
                      weighting=params.weighting,
                      architecture=params.arch,
                      encoder_class=encoder_class,
                      decoders=decoders,
                      rep_grad=params.rep_grad,
                      multi_input=params.multi_input,
                      optim_param=optim_param,
                      scheduler_param=scheduler_param,
                      **kwargs)
```

7.5 Run a Model

Finally, you can train the model by using the `train()` function like this.

```
officeModel.train(train_dataloaders=train_dataloaders,
                  val_dataloaders=val_dataloaders,
                  test_dataloaders=test_dataloaders,
                  epochs=100)
```

When the training process ends, the best results on the test dataset will be printed automatically. Please see *LibMTL.Trainer.train()* and *LibMTL.utils.count_improvement()* for details.

CHAPTER

EIGHT

CUSTOMIZE AN ARCHITECTURE

Here we introduce how to customize a new architecture with the support of `LibMTL`.

8.1 Create a New Architecture Class

Firstly, you need to create a new architecture class by inheriting class *LibMTL.architecture.AbsArchitecture*.

```
from LibMTL.architecture import AbsArchitecture

class NewArchitecture(AbsArchitecture):
    def __init__(self, task_name, encoder_class, decoders, rep_grad,
                       multi_input, device, **kwargs):
        super(NewArchitecture, self).__init__(task_name, encoder_class, decoders, rep_
↪grad,
                                                          multi_input, device,␣
↪**kwargs)
```

8.2 Rewrite Relevant Methods

There are four important functions in *LibMTL.architecture.AbsArchitecture*.

- `forward()`: The forward function and its input/output format can be found in *LibMTL.architecture.AbsArchitecture.forward()*. To rewrite this function, you need to consider the case of `single-input` and `multi-input` (refer to here) and the case of `rep-grad` and `param-grad` (refer to here) if you want to combine your architecture with more weighting strategies or apply your architecture to more datasets.
- `get_share_params()`: This function is used to return the shared parameters of the model. It returns all the parameters of the encoder by default. You can rewrite it if necessary.
- `zero_grad_share_params()`: This function is used to set gradients of the shared parameters to zero. It will set the gradients of all the encoder parameters to zero by default. You can rewrite it if necessary.
- `_prepare_rep()`: This function is used to compute the gradients for representations. More details can be found here.

CHAPTER

NINE

CUSTOMIZE A WEIGHTING STRATEGY

Here we introduce how to customize a new weighting strategy with the support of `LibMTL`.

9.1 Create a New Weighting Class

Firstly, you need to create a new weighting class by inheriting class *LibMTL.weighting.AbsWeighting*.

```
from LibMTL.weighting import AbsWeighting

class NewWeighting(AbsWeighting):
    def __init__(self):
        super(NewWeighting, self).__init__()
```

9.2 Rewrite Relevant Methods

There are four important functions in *LibMTL.weighting.AbsWeighting*.

- `backward()`: It is the main function of a weighting strategy whose input and output formats can be found in *LibMTL.weighting.AbsWeighting.backward()*. To rewrite this function, you need to consider the case of `single-input` and `multi-input` (refer to here) and the case of `rep-grad` and `param-grad` (refer to here) if you want to combine your weighting method with more architectures or apply your method to more datasets.
- `init_param()`: This function is used to define and initialize some trainable parameters. It does nothing by default and can be rewritten if necessary.
- `_get_grads()`: This function is used to return the gradients of representations or shared parameters (corresponding to the case of `rep-grad` and `param-grad`, respectively).
- `_backward_new_grads()`: This function is used to reset the gradients and make a backward pass (corresponding to the case of `rep-grad` and `param-grad`, respectively).

The `_get_grads()` and `_backward_new_grads()` functions are very useful to rewrite the `backward()` function and you can find more details here.

CHAPTER

TEN

LIBMTL

class Trainer(*task_dict*, *weighting*, *architecture*, *encoder_class*, *decoders*, *rep_grad*, *multi_input*, *optim_param*, *scheduler_param*, *save_path=None*, *load_path=None*, ***kwargs*)

Bases: torch.nn.Module

A Multi-Task Learning Trainer.

This is a unified and extensible training framework for multi-task learning.

Parameters

- **task_dict** (*dict*) – A dictionary of name-information pairs of type (str, dict). The sub-dictionary for each task has four entries whose keywords are named **metrics**, **metrics_fn**, **loss_fn**, **weight** and each of them corresponds to a list. The list of **metrics** has m strings, repersenting the name of m metrics for this task. The list of **metrics_fn** has two elements, i.e., the updating and score functions, meaning how to update thoes objectives in the training process and obtain the final scores, respectively. The list of **loss_fn** has m loss functions corresponding to each metric. The list of **weight** has m binary integers corresponding to each metric, where 1 means the higher the score is, the better the performance, 0 means the opposite.
- **weighting** (*class*) – A weighting strategy class based on LibMTL.weighting.abstract_weighting.AbsWeighting.
- **architecture** (*class*) – An architecture class based on LibMTL.architecture.abstract_arch.AbsArchitecture.
- **encoder_class** (*class*) – A neural network class.
- **decoders** (*dict*) – A dictionary of name-decoder pairs of type (str, torch.nn.Module).
- **rep_grad** (*bool*) – If True, the gradient of the representation for each task can be computed.
- **multi_input** (*bool*) – Is True if each task has its own input data, otherwise is False.
- **optim_param** (*dict*) – A dictionary of configurations for the optimizier.
- **scheduler_param** (*dict*) – A dictionary of configurations for learning rate scheduler. Set it to None if you do not use a learning rate scheduler.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting and architecture methods.

Note: It is recommended to use *LibMTL.config.prepare_args()* to return the dictionaries of optim_param, scheduler_param, and kwargs.

Examples:

```
import torch.nn as nn
from LibMTL import Trainer
from LibMTL.loss import CE_loss_fn
from LibMTL.metrics import acc_update_fun, acc_score_fun
from LibMTL.weighting import EW
from LibMTL.architecture import HPS
from LibMTL.model import ResNet18
from LibMTL.config import prepare_args

task_dict = {'A': {'metrics': ['Acc'],
                   'metrics_fn': [acc_update_fun, acc_score_fun],
                   'loss_fn': [CE_loss_fn],
                   'weight': [1]}}

decoders = {'A': nn.Linear(512, 31)}

# You can use command-line arguments and return configurations by ``prepare_args``.
# kwargs, optim_param, scheduler_param = prepare_args(params)
optim_param = {'optim': 'adam', 'lr': 1e-3, 'weight_decay': 1e-4}
scheduler_param = {'scheduler': 'step'}
kwargs = {'weight_args': {}, 'arch_args': {}}

trainer = Trainer(task_dict=task_dict,
                  weighting=EW,
                  architecture=HPS,
                  encoder_class=ResNet18,
                  decoders=decoders,
                  rep_grad=False,
                  multi_input=False,
                  optim_param=optim_param,
                  scheduler_param=scheduler_param,
                  **kwargs)
```

process_preds(*self*, *preds*, *task_name=None*)

The processing of prediction for each task.

- The default is no processing. If necessary, you can rewrite this function.
- If `multi_input` is `True`, `task_name` is valid and `preds` with type `torch.Tensor` is the prediction of this task.
- otherwise, `task_name` is invalid and `preds` is a `dict` of name-prediction pairs of all tasks.

Parameters

- **preds** (*dict or torch.Tensor*) – The prediction of `task_name` or all tasks.
- **task_name** (*str*) – The string of task name.

train(*self*, *train_dataloaders*, *test_dataloaders*, *epochs*, *val_dataloaders=None*, *return_weight=False*)

The training process of multi-task learning.

Parameters

- **train_dataloaders** (*dict or torch.utils.data.DataLoader*) – The dataloaders used for training. If `multi_input` is `True`, it is a dictionary of name-dataloader pairs.

Otherwise, it is a single dataloader which returns data and a dictionary of name-label pairs in each iteration.

- **test_dataloaders** (*dict or torch.utils.data.DataLoader*) – The dataloaders used for the validation or testing. The same structure with `train_dataloaders`.
- **epochs** (*int*) – The total training epochs.
- **return_weight** (*bool*) – if `True`, the loss weights will be returned.

test(*self*, *test_dataloaders*, *epoch=None*, *mode='test'*, *return_improvement=False*)

The test process of multi-task learning.

Parameters

- **test_dataloaders** (*dict or torch.utils.data.DataLoader*) – If `multi_input` is `True`, it is a dictionary of name-dataloader pairs. Otherwise, it is a single dataloader which returns data and a dictionary of name-label pairs in each iteration.
- **epoch** (*int, default=None*) – The current epoch.

10.1 LibMTL.architecture

class AbsArchitecture(*task_name*, *encoder_class*, *decoders*, *rep_grad*, *multi_input*, *device*, ***kwargs*)

Bases: `torch.nn.Module`

An abstract class for MTL architectures.

Parameters

- **task_name** (*list*) – A list of strings for all tasks.
- **encoder_class** (*class*) – A neural network class.
- **decoders** (*dict*) – A dictionary of name-decoder pairs of type (`str`, `torch.nn.Module`).
- **rep_grad** (*bool*) – If `True`, the gradient of the representation for each task can be computed.
- **multi_input** (*bool*) – Is `True` if each task has its own input data, otherwise is `False`.
- **device** (*torch.device*) – The device where model and data will be allocated.
- **kwargs** (*dict*) – A dictionary of hyperparameters of architectures.

forward(*self*, *inputs*, *task_name=None*)

Parameters

- **inputs** (*torch.Tensor*) – The input data.
- **task_name** (*str, default=None*) – The task name corresponding to `inputs` if `multi_input` is `True`.

Returns
A dictionary of name-prediction pairs of type (`str`, `torch.Tensor`).

Return type
dict

get_share_params(*self*)

Return the shared parameters of the model.

zero_grad_share_params(*self*)

Set gradients of the shared parameters to zero.

class HPS(*task_name*, *encoder_class*, *decoders*, *rep_grad*, *multi_input*, *device*, ***kwargs*)

Bases: `LibMTL.architecture.abstract_arch.AbsArchitecture`

Hard Parameter Sharing (HPS).

This method is proposed in Multitask Learning: A Knowledge-Based Source of Inductive Bias (ICML 1993) and implemented by us.

class Cross_stitch(*task_name*, *encoder_class*, *decoders*, *rep_grad*, *multi_input*, *device*, ***kwargs*)

Bases: `LibMTL.architecture.abstract_arch.AbsArchitecture`

Cross-stitch Networks (Cross_stitch).

This method is proposed in Cross-stitch Networks for Multi-task Learning (CVPR 2016) and implemented by us.

Warning:

- *Cross_stitch* does not work with multiple inputs MTL problem, i.e., `multi_input` must be `False`.
- *Cross_stitch* is only supported by ResNet-based encoders.

class MMoE(*task_name*, *encoder_class*, *decoders*, *rep_grad*, *multi_input*, *device*, ***kwargs*)

Bases: `LibMTL.architecture.abstract_arch.AbsArchitecture`

Multi-gate Mixture-of-Experts (MMoE).

This method is proposed in Modeling Task Relationships in Multi-task Learning with Multi-gate Mixture-of-Experts (KDD 2018) and implemented by us.

Parameters

- **img_size** (*list*) – The size of input data. For example, [3, 244, 244] denotes input images with size 3x224x224.
- **num_experts** (*int*) – The number of experts shared for all tasks. Each expert is an encoder network.

forward(*self*, *inputs*, *task_name=None*)

Parameters

- **inputs** (*torch.Tensor*) – The input data.
- **task_name** (*str, default=None*) – The task name corresponding to `inputs` if `multi_input` is `True`.

Returns
A dictionary of name-prediction pairs of type (`str`, `torch.Tensor`).

Return type
dict

get_share_params(*self*)

Return the shared parameters of the model.

zero_grad_share_params(*self*)

Set gradients of the shared parameters to zero.

class MTAN(*task_name*, *encoder_class*, *decoders*, *rep_grad*, *multi_input*, *device*, ***kwargs*)

Bases: LibMTL.architecture.abstract_arch.AbsArchitecture

Multi-Task Attention Network (MTAN).

This method is proposed in End-To-End Multi-Task Learning With Attention (CVPR 2019) and implemented by modifying from the official PyTorch implementation.

Warning: *MTAN* is only supported by ResNet-based encoders.

forward(*self*, *inputs*, *task_name=None*)

Parameters

- **inputs** (*torch.Tensor*) – The input data.
- **task_name** (*str, default=None*) – The task name corresponding to inputs if multi_input is True.

Returns
A dictionary of name-prediction pairs of type (str, torch.Tensor).

Return type
dict

get_share_params(*self*)

Return the shared parameters of the model.

zero_grad_share_params(*self*)

Set gradients of the shared parameters to zero.

class CGC(*task_name*, *encoder_class*, *decoders*, *rep_grad*, *multi_input*, *device*, ***kwargs*)

Bases: LibMTL.architecture.MMoE.MMoE

Customized Gate Control (CGC).

This method is proposed in Progressive Layered Extraction (PLE): A Novel Multi-Task Learning (MTL) Model for Personalized Recommendations (ACM RecSys 2020 Best Paper) and implemented by us.

Parameters

- **img_size** (*list*) – The size of input data. For example, [3, 244, 244] denotes input images with size 3x224x224.
- **num_experts** (*list*) – The numbers of experts shared by all the tasks and specific to each task, respectively. Each expert is an encoder network.

forward(*self*, *inputs*, *task_name=None*)

class PLE(*task_name*, *encoder_class*, *decoders*, *rep_grad*, *multi_input*, *device*, ***kwargs*)

Bases: LibMTL.architecture.abstract_arch.AbsArchitecture

Progressive Layered Extraction (PLE).

This method is proposed in Progressive Layered Extraction (PLE): A Novel Multi-Task Learning (MTL) Model for Personalized Recommendations (ACM RecSys 2020 Best Paper) and implemented by us.

Parameters

- **img_size** (*list*) – The size of input data. For example, [3, 244, 244] denotes input images with size 3x224x224.

- **num_experts** (*list*) – The numbers of experts shared by all the tasks and specific to each task, respectively. Each expert is an encoder network.

Warning:

- *PLE* does not work with multi-input problems, i.e., `multi_input` must be `False`.
- *PLE* is only supported by ResNet-based encoders.

forward(*self*, *inputs*, *task_name=None*)

Parameters

- **inputs** (*torch.Tensor*) – The input data.
- **task_name** (*str, default=None*) – The task name corresponding to `inputs` if `multi_input` is `True`.

Returns
A dictionary of name-prediction pairs of type (`str`, `torch.Tensor`).

Return type
dict

get_share_params(*self*)

Return the shared parameters of the model.

zero_grad_share_params(*self*)

Set gradients of the shared parameters to zero.

class DSelect_k(*task_name*, *encoder_class*, *decoders*, *rep_grad*, *multi_input*, *device*, ***kwargs*)

Bases: `LibMTL.architecture.MMoE.MMoE`

DSelect-k.

This method is proposed in DSelect-k: Differentiable Selection in the Mixture of Experts with Applications to Multi-Task Learning (NeurIPS 2021) and implemented by modifying from the official TensorFlow implementation.

Parameters

- **img_size** (*list*) – The size of input data. For example, [3, 244, 244] denotes input images with size 3x224x224.
- **num_experts** (*int*) – The number of experts shared by all the tasks. Each expert is an encoder network.
- **num_nonzeros** (*int*) – The number of selected experts.
- **kgamma** (*float, default=1.0*) – A scaling parameter for the smooth-step function.

forward(*self*, *inputs*, *task_name=None*)

class LTB(*task_name*, *encoder_class*, *decoders*, *rep_grad*, *multi_input*, *device*, ***kwargs*)

Bases: `LibMTL.architecture.abstract_arch.AbsArchitecture`

Learning To Branch (LTB).

This method is proposed in Learning to Branch for Multi-Task Learning (ICML 2020) and implemented by us.

Warning:

- *LTB* does not work with multi-input problems, i.e., `multi_input` must be `False`.
- *LTB* is only supported by ResNet-based encoders.

forward(*self*, *inputs*, *task_name=None*)

Parameters

- **inputs** (*torch.Tensor*) – The input data.
- **task_name** (*str, default=None*) – The task name corresponding to `inputs` if `multi_input` is `True`.

Returns
A dictionary of name-prediction pairs of type (`str`, `torch.Tensor`).

Return type
dict

10.2 LibMTL.model

resnet18(*pretrained=False*, *progress=True*, ***kwargs*)

ResNet-18 model from "Deep Residual Learning for Image Recognition"

Parameters

- **pretrained** (*bool*) – If True, returns a model pre-trained on the ImageNet dataset.
- **progress** (*bool*) – If True, displays a progress bar of the download to stderr.

resnet34(*pretrained=False*, *progress=True*, ***kwargs*)

ResNet-34 model from "Deep Residual Learning for Image Recognition"

Parameters

- **pretrained** (*bool*) – If True, returns a model pre-trained on the ImageNet dataset.
- **progress** (*bool*) – If True, displays a progress bar of the download to stderr.

resnet50(*pretrained=False*, *progress=True*, ***kwargs*)

ResNet-50 model from "Deep Residual Learning for Image Recognition"

Parameters

- **pretrained** (*bool*) – If True, returns a model pre-trained on the ImageNet dataset.
- **progress** (*bool*) – If True, displays a progress bar of the download to stderr.

resnet101(*pretrained=False*, *progress=True*, ***kwargs*)

ResNet-101 model from "Deep Residual Learning for Image Recognition"

Parameters

- **pretrained** (*bool*) – If True, returns a model pre-trained on the ImageNet dataset.
- **progress** (*bool*) – If True, displays a progress bar of the download to stderr.

resnet152(*pretrained=False*, *progress=True*, ***kwargs*)

ResNet-152 model from "Deep Residual Learning for Image Recognition"

Parameters

- **pretrained** (*bool*) – If True, returns a model pre-trained on the ImageNet dataset.
- **progress** (*bool*) – If True, displays a progress bar of the download to stderr.

resnext50_32x4d(*pretrained=False*, *progress=True*, ***kwargs*)

ResNeXt-50 32x4d model from "Aggregated Residual Transformation for Deep Neural Networks"

Parameters

- **pretrained** (*bool*) – If True, returns a model pre-trained on the ImageNet dataset.
- **progress** (*bool*) – If True, displays a progress bar of the download to stderr.

resnext101_32x8d(*pretrained=False*, *progress=True*, ***kwargs*)

ResNeXt-101 32x8d model from "Aggregated Residual Transformation for Deep Neural Networks"

Parameters

- **pretrained** (*bool*) – If True, returns a model pre-trained on the ImageNet dataset.
- **progress** (*bool*) – If True, displays a progress bar of the download to stderr.

wide_resnet50_2(*pretrained=False*, *progress=True*, ***kwargs*)

Wide ResNet-50-2 model from "Wide Residual Networks"

The model is the same as ResNet except for the number of bottleneck channels which is twice larger in every block. The number of channels in outer 1x1 convolutions is the same, e.g., the last block in ResNet-50 has 2048-512-2048 channels, while in wide ResNet-50-2 there are 2048-1024-2048.

Parameters

- **pretrained** (*bool*) – If True, returns a model pre-trained on the ImageNet dataset.
- **progress** (*bool*) – If True, displays a progress bar of the download to stderr.

wide_resnet101_2(*pretrained=False*, *progress=True*, ***kwargs*)

Wide ResNet-101-2 model from "Wide Residual Networks"

The model is the same as ResNet except for the number of bottleneck channels which is twice larger in every block. The number of channels in outer 1x1 convolutions is the same, e.g., the last block in ResNet-101 has 2048-512-2048 channels, while in wide ResNet-101-2 there are 2048-1024-2048.

Parameters

- **pretrained** (*bool*) – If True, returns a model pre-trained on the ImageNet dataset.
- **progress** (*bool*) – If True, displays a progress bar of the download to stderr.

resnet_dilated(*basenet*, *pretrained=True*, *dilate_scale=8*)

Dilated Residual Network models from "Dilated Residual Networks"

Parameters

- **basenet** (*str*) – The type of ResNet.
- **pretrained** (*bool*) – If True, returns a model pre-trained on ImageNet.
- **dilate_scale** (*{8, 16}, default=8*) – The type of dilating process.

10.3 LibMTL.weighting

class AbsWeighting

Bases: torch.nn.Module

An abstract class for weighting strategies.

init_param(*self*)

Define and initialize some trainable parameters required by specific weighting methods.

property backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

class EW

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Equal Weighting (EW).

The loss weight for each task is always `1 / T` in every iteration, where `T` denotes the number of tasks.

backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

class GradNorm

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Gradient Normalization (GradNorm).

This method is proposed in GradNorm: Gradient Normalization for Adaptive Loss Balancing in Deep Multitask Networks (ICML 2018) and implemented by us.

Parameters

alpha (*float, default=1.5*) – The strength of the restoring force which pulls tasks back to a common training rate.

init_param(*self*)

Define and initialize some trainable parameters required by specific weighting methods.

backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

class MGDA

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Multiple Gradient Descent Algorithm (MGDA).

This method is proposed in Multi-Task Learning as Multi-Objective Optimization (NeurIPS 2018) and implemented by modifying from the official PyTorch implementation.

Parameters
mgda_gn (*{'none', 'l2', 'loss', 'loss+'}, default='none'*) – The type of gradient normalization.

backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

class UW

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Uncertainty Weights (UW).

This method is proposed in Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics (CVPR 2018) and implemented by us.

init_param(*self*)
Define and initialize some trainable parameters required by specific weighting methods.

backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

class DWA

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Dynamic Weight Average (DWA).

This method is proposed in End-To-End Multi-Task Learning With Attention (CVPR 2019) and implemented by modifying from the official PyTorch implementation.

Parameters
T (*float, default=2.0*) – The softmax temperature.

backward(*self*, *losses*, ***kwargs*)

class GLS

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Geometric Loss Strategy (GLS).

This method is proposed in MultiNet++: Multi-Stream Feature Aggregation and Geometric Loss Strategy for Multi-Task Learning (CVPR 2019 workshop) and implemented by us.

backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

class GradDrop

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Gradient Sign Dropout (GradDrop).

This method is proposed in Just Pick a Sign: Optimizing Deep Multitask Models with Gradient Sign Dropout (NeurIPS 2020) and implemented by us.

Parameters

leak (*float, default=0.0*) – The leak parameter for the weighting matrix.

> **Warning:** GradDrop is not supported by parameter gradients, i.e., rep_grad must be True.

backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

class PCGrad

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Project Conflicting Gradients (PCGrad).

This method is proposed in Gradient Surgery for Multi-Task Learning (NeurIPS 2020) and implemented by us.

> **Warning:** PCGrad is not supported by representation gradients, i.e., rep_grad must be False.

backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

class GradVac

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Gradient Vaccine (GradVac).

This method is proposed in Gradient Vaccine: Investigating and Improving Multi-task Optimization in Massively Multilingual Models (ICLR 2021 Spotlight) and implemented by us.

Parameters

- **GradVac_beta** (*float, default=0.5*) – The exponential moving average (EMA) decay parameter.
- **GradVac_group_type** (*int, default=0*) – The parameter granularity (0: whole_model; 1: all_layer; 2: all_matrix).

> **Warning:** GradVac is not supported by representation gradients, i.e., rep_grad must be False.

init_param(*self*)

backward(*self*, *losses*, ***kwargs*)

class IMTL

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Impartial Multi-task Learning (IMTL).

This method is proposed in Towards Impartial Multi-task Learning (ICLR 2021) and implemented by us.

init_param(*self*)

Define and initialize some trainable parameters required by specific weighting methods.

backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

class CAGrad

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Conflict-Averse Gradient descent (CAGrad).

This method is proposed in Conflict-Averse Gradient Descent for Multi-task learning (NeurIPS 2021) and implemented by modifying from the official PyTorch implementation.

Parameters

- **calpha** (*float, default=0.5*) – A hyperparameter that controls the convergence rate.
- **rescale** (*{0, 1, 2}, default=1*) – The type of the gradient rescaling.

Warning: CAGrad is not supported by representation gradients, i.e., `rep_grad` must be `False`.

backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

class Nash_MTL

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Nash-MTL.

This method is proposed in Multi-Task Learning as a Bargaining Game (ICML 2022) and implemented by modifying from the official PyTorch implementation.

Parameters

- **update_weights_every** (*int, default=1*) – Period of weights update.
- **optim_niter** (*int, default=20*) – The max iteration of optimization solver.
- **max_norm** (*float, default=1.0*) – The max norm of the gradients.

Warning: Nash_MTL is not supported by representation gradients, i.e., `rep_grad` must be `False`.

init_param(*self*)

Define and initialize some trainable parameters required by specific weighting methods.

solve_optimization(*self*, *gtg: numpy.array*)

backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

class RLW

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Random Loss Weighting (RLW).

This method is proposed in Reasonable Effectiveness of Random Weighting: A Litmus Test for Multi-Task Learning (TMLR 2022) and implemented by us.

backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

class MoCo

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

MoCo.

This method is proposed in Mitigating Gradient Bias in Multi-objective Learning: A Provably Convergent Approach (ICLR 2023) and implemented based on the author' sharing code (Heshan Fernando: fernah@rpi.edu).

Parameters

- **MoCo_beta** (*float, default=0.5*) – The learning rate of y.
- **MoCo_beta_sigma** (*float, default=0.5*) – The decay rate of MoCo_beta.
- **MoCo_gamma** (*float, default=0.1*) – The learning rate of lambd.
- **MoCo_gamma_sigma** (*float, default=0.5*) – The decay rate of MoCo_gamma.
- **MoCo_rho** (*float, default=0*) – The ell_2 regularization parameter of lambda's update.

Warning: MoCo is not supported by representation gradients, i.e., `rep_grad` must be `False`.

init_param(*self*)

Define and initialize some trainable parameters required by specific weighting methods.

backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

class Aligned_MTL

Bases: LibMTL.weighting.abstract_weighting.AbsWeighting

Aligned-MTL.

This method is proposed in Independent Component Alignment for Multi-Task Learning (CVPR 2023) and implemented by modifying from the official PyTorch implementation.

backward(*self*, *losses*, ***kwargs*)

Parameters

- **losses** (*list*) – A list of losses of each task.
- **kwargs** (*dict*) – A dictionary of hyperparameters of weighting methods.

CHAPTER

ELEVEN

LIBMTL.LOSS

class AbsLoss

Bases: object

An abstract class for loss functions.

compute_loss(*self*, *pred*, *gt*)

Calculate the loss.

Parameters

- **pred** (*torch.Tensor*) – The prediction tensor.
- **gt** (*torch.Tensor*) – The ground-truth tensor.

Returns
The loss.

Return type
torch.Tensor

class CELoss

Bases: *AbsLoss*

The cross-entropy loss function.

compute_loss(*self*, *pred*, *gt*)

class KLDivLoss

Bases: *AbsLoss*

The Kullback-Leibler divergence loss function.

compute_loss(*self*, *pred*, *gt*)

class L1Loss

Bases: *AbsLoss*

The Mean Absolute Error (MAE) loss function.

compute_loss(*self*, *pred*, *gt*)

class MSELoss

Bases: *AbsLoss*

The Mean Squared Error (MSE) loss function.

compute_loss(*self*, *pred*, *gt*)

CHAPTER

TWELVE

LIBMTL.UTILS

get_root_dir()

Return the root path of project.

set_random_seed(*seed*)

Set the random seed for reproducibility.

Parameters
seed (*int, default=0*) – The random seed.

set_device(*gpu_id*)

Set the device where model and data will be allocated.

Parameters
gpu_id (*str, default='0'*) – The id of gpu.

count_parameters(*model*)

Calculate the number of parameters for a model.

Parameters
model (*torch.nn.Module*) – A neural network module.

count_improvement(*base_result*, *new_result*, *weight*)

Calculate the improvement between two results as

$$\Delta_{\mathrm{p}} = 100\% \times \frac{1}{T}\sum_{t=1}^{T}\frac{1}{M_t}\sum_{m=1}^{M_t}\frac{(-1)^{w_{t,m}}(B_{t,m}-N_{t,m})}{N_{t,m}}.$$

Parameters

- **base_result** (*dict*) – A dictionary of scores of all metrics of all tasks.
- **new_result** (*dict*) – The same structure with `base_result`.
- **weight** (*dict*) – The same structure with `base_result` while each element is binary integer representing whether higher or lower score is better.

Returns
The improvement between `new_result` and `base_result`.

Return type
float

Examples:

```
base_result = {'A': [96, 98], 'B': [0.2]}
new_result = {'A': [93, 99], 'B': [0.5]}
weight = {'A': [1, 0], 'B': [1]}

print(count_improvement(base_result, new_result, weight))
```

CHAPTER

THIRTEEN

LIBMTL.CONFIG

LibMTL_args

prepare_args(*params*)

Return the configuration of hyperparameters, optimizier, and learning rate scheduler.

Parameters

params (*argparse.Namespace*) – The command-line arguments.

CHAPTER

FOURTEEN

LIBMTL.METRICS

class AbsMetric

Bases: object

An abstract class for the performance metrics of a task.

record

A list of the metric scores in every iteration.

Type
list

bs

A list of the number of data in every iteration.

Type
list

property update_fun(*self*, *pred*, *gt*)

Calculate the metric scores in every iteration and update *record*.

Parameters

- **pred** (*torch.Tensor*) – The prediction tensor.
- **gt** (*torch.Tensor*) – The ground-truth tensor.

property score_fun(*self*)

Calculate the final score (when an epoch ends).

Returns
A list of metric scores.

Return type
list

reinit(*self*)

Reset *record* and *bs* (when an epoch ends).

class AccMetric

Bases: *AbsMetric*

Calculate the accuracy.

update_fun(*self*, *pred*, *gt*)

score_fun(*self*)

class L1Metric

Bases: *AbsMetric*

Calculate the Mean Absolute Error (MAE).

update_fun(*self*, *pred*, *gt*)

score_fun(*self*)

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Nathan Silberman, Derek Hoiem, Pushmeet Kohli, and Rob Fergus. Indoor segmentation and support inference from rgbd images. In *Proceedings of the 8th European Conference on Computer Vision*, 746–760. 2012.

[1] Yu Zhang and Qiang Yang. A survey on multi-task learning. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[2] Simon Vandenhende, Stamatios Georgoulis, Wouter Van Gansbeke, Marc Proesmans, Dengxin Dai, and Luc Van Gool. Multi-task learning for dense prediction tasks: a survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[3] Baijiong Lin, Feiyang Ye, and Yu Zhang. A closer look at loss weighting in multi-task learning. *arXiv preprint arXiv:2111.10603*, 2021.

[4] Michael Crawshaw. Multi-task learning with deep neural networks: a survey. *arXiv preprint arXiv:2009.09796*, 2020.

[1] Nathan Silberman, Derek Hoiem, Pushmeet Kohli, and Rob Fergus. Indoor segmentation and support inference from rgbd images. In *Proceedings of the 8th European Conference on Computer Vision*, 746–760. 2012.

[2] Shikun Liu, Edward Johns, and Andrew J. Davison. End-to-end multi-task learning with attention. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 1871–1880. 2019.

[3] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the 14th European Conference on Computer Vision*, volume 11211, 833–851. 2018.

[4] Fisher Yu, Vladlen Koltun, and Thomas A. Funkhouser. Dilated residual networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 636–644. 2017.

[1] Kate Saenko, Brian Kulis, Mario Fritz, and Trevor Darrell. Adapting visual category models to new domains. In *Proceedings of the 6th European Conference on Computer Vision*, 213–226. 2010.

[2] Hemanth Venkateswara, Jose Eusebio, Shayok Chakraborty, and Sethuraman Panchanathan. Deep hashing network for unsupervised domain adaptation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 5018–5027. 2017.

PYTHON MODULE INDEX

L

INDEX

L

M

N

P

R

S

T

U

W

Z