dalib Documentation

Release 0.0.1

jiangjunguang, fubo

Aug 04, 2020

Contents

CHAPTER 1

Tutorial

For Domain Adaptation on Computer Vision Tasks

Authors: Junguang Jiang

In this tutorial, you will learn how to use domain adaptation in image classification. If you want to know more about domain adaptation or transfer learning, please refer to A Survey on Transfer Learning or DANN

1.1 Load Data

DALIB provides visual datasets commonly used in domain adatation research, including *Office-31*, *Office-Home*, *VisDA-2017* and so on.

```
# Data augmentation and normalization for training
# Just normalization for validation
import torchvision.transforms as transforms
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.
↪225])
train_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    normalize
])
val_tranform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    normalize
])

from torch.utils.data import DataLoader
from dalib.vision.datasets import Office31
```

(continues on next page)

(continued from previous page)

```
data_dir = "data/office31"
source = "A"  # source domain: amazon
target = "W"  # target domain: webcam
batch_size = 32
# download data automatically from the internet into data_dir
train_source_dataset = Office31(root=data_dir, task=source, download=True,␣
↪transform=train_transform)
train_source_loader = DataLoader(train_source_dataset, batch_size=batch_size,␣
↪shuffle=True, drop_last=True)
train_target_dataset = Office31(root=data_dir, task=target, download=True,␣
↪transform=train_transform)
train_target_loader = DataLoader(train_target_dataset, batch_size=batch_size,␣
↪shuffle=True, drop_last=True)
val_dataset = Office31(root=data_dir, task=target, download=True, transform=val_
↪tranform)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

If you want to implement your own datasets, you can inherit class from *torchvision.datasets.VisionDataset* or *dalib.vision.datasets.ImageList*. For instance, if your task is partial domain adaptation on *Office-31*, you can construct datasets as follows.

```
from dalib.vision.datasets import ImageList

class Office31PDA(ImageList):
"""Datasets for PDA (partial domain adaptation) on Office-31
Parameters:
- **root** (str): Root directory of dataset
- **task** (str): The task (domain) to create dataset. Choices include ``'A'``:␣
↪amazon, \
    ``'D'``: dslr, ``'W'``: webcam, ``'AP'``: partial amazon, ``'DP'``: partial dslr \
    and ``'WP'``: partial webcam.
"""
image_list = {
    "A": "amazon.txt",
    "D": "dslr.txt",
    "W": "webcam.txt",
    "AP": "amazon_partial.txt",
    "DP": "dslr_partial.txt",
    "WP": "webcam_partial.txt",
}

def __init__(self, root, task, **kwargs):
    assert task in self.image_list
    data_list_file = os.path.join(root, self.image_list[task])
    super(Office31PDA, self).__init__(root, num_classes=31, data_list_file=data_list_
↪file, **kwargs)
```

Note:

- Your need to put image list file under *root* before using *Office31PDA*.
- "amazon.txt" list the images of all categories in amazon domain, and "amazon_partial.txt" list the images of some categories in amazon domain.

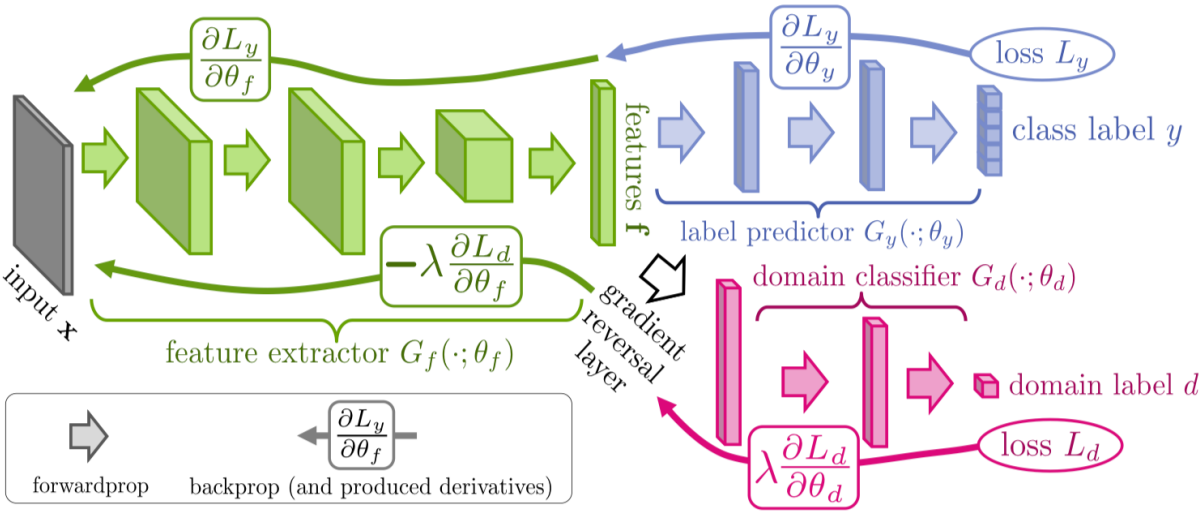
You can refer to *dalib.vision.datasets.ImageList* for the detailed format of image list file.

After constructing *Office31PDA* you can use the same data loading code as described above.

1.2 Parepare models and adaptation algorithms

We will use DANN as an instance. You can find the usage of other adaptation algorithms in DALIB APIs or examples on github.

DANN introduces a minimax game into domain adaptation, where a domain discriminator attempts to distinguish the source from the target, while a feature extractor tries to fool the domain discriminator.



To prepare models for training, you need to

1. load a pretrained model without final fully connected layer.
2. construct a classifier and a domain discriminator.
3. pass the domain discriminator to the DomainAdversarialLoss.

```
# load pretrained backbone
from dalib.vision.models.resnet import resnet50
backbone = resnet50(pretrained=True)

from dalib.modules.domain_discriminator import DomainDiscriminator
from dalib.adaptation.dann import DomainAdversarialLoss, ImageClassifier
# classifier has a backbone (pretrained resnet50), a bottleneck layer
# and a head layer (nn.Linear)
classifier = ImageClassifier(backbone, train_source_dataset.num_classes).cuda()

# domain discriminator is a 3-layer fully connected networks, which distinguish
# whether the input features come from the source domain or the target domain
domain_discriminator = DomainDiscriminator(in_feature=classifier.features_dim, hidden_
↪size=1024).cuda()

# define loss function
dann_loss = DomainAdversarialLoss(domain_discriminator).cuda()

# define optimizer and lr scheduler
from tools.lr_scheduler import StepwiseLR
optimizer = SGD(classifier.get_parameters() + domain_discriminator.get_parameters(),
```

(continues on next page)

(continued from previous page)

```
                lr=0.01, momentum=0.9, weight_decay=1e-3, nesterov=True)
# learning rate will drop from 0.01 each step
lr_scheduler = StepwiseLR(optimizer, init_lr=0.01, gamma=0.001, decay_rate=0.75)
```

Note: We will use some functions from tools, such as *StepwiseLR* and *ForeverDataIterator* for clearer code. We will only explain their functionality. Please refer to Tutorial for runnable code.

1.3 Training the model

Now, let's write a general process to train a model.

```
# start training
best_acc1 = 0.
for epoch in range(args.epochs):
    # train for one epoch
    train(train_source_iter, train_target_iter, classifier, dann_loss, optimizer, lr_
↪scheduler)

    # evaluate on validation set
    acc1 = validate(val_loader, classifier)

    # remember best acc@1
    best_acc1 = max(acc1, best_acc1)
```

During training, we explicitly set 1 epochs equal to 500 steps.

```
import torch.nn.functional as F

def train(train_source_iter, train_target_iter, model, dann_loss, optimizer,␣
↪scheduler):
    # switch to train mode
    model.train()
    dann_loss.train()

    # train_source_iter and train_target_iter is data iterator that will never stop␣
↪producing data
    T = 500
    for i in range(T):
        scheduler.step()
        # data from source domain
        x_s, labels_s = next(train_source_iter)
        # data from target domain
        x_t, _ = next(train_target_iter)

        x_s = x_s.cuda()
        x_t = x_t.cuda()
        labels_s = labels_s.cuda()

        # compute output
        y_s, f_s = model(x_s)

        # cross entropy loss on source domain
```

(continues on next page)

(continued from previous page)

```
        cls_loss = F.cross_entropy(y_s, labels_s)
        _, f_t = model(x_t)

        # domain adversarial loss
        transfer_loss = dann_loss(f_s, f_t)
        loss = cls_loss + transfer_loss

        # compute gradient and do SGD step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

The evaluation code is similar as in supervised learning.

```
from tools.util import AverageMeter, accuracy

def validate(val_loader, model):
    top1 = AverageMeter('Acc@1', ':6.2f')

    # switch to evaluate mode
    model.eval()

    with torch.no_grad():
        end = time.time()
        for i, (images, target) in enumerate(val_loader):
            images = images.cuda()
            target = target.cuda()

            # compute output
            output, _ = model(images)

            # measure accuracy and record loss
            acc1,  = accuracy(output, target, topk=(1, ))
            top1.update(acc1[0], images.size(0))

        print(' * Acc@1 {top1.avg:.3f} Acc@5 {top5.avg:.3f}'
              .format(top1=top1, top5=top5))

    return top1.avg
```

1.4 Visualizing the results

After the training is finished, we can visualize the representations of task A $\rightarrow$ W by t-SNE.

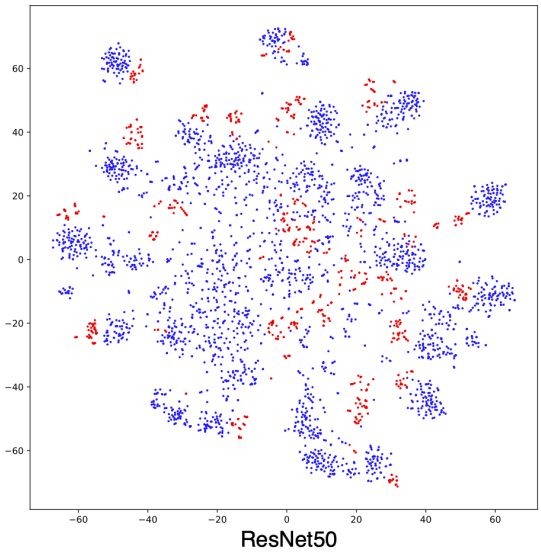
```
# get features from source and target domain
classifier.load_state_dict(best_model)
classifier.eval()

features, domains = [], []
source_val_dataset = dataset(root=data_dir, task=source, download=True, transform=val_
↪tranform)
source_val_loader = DataLoader(source_val_dataset, batch_size=batch_size,␣
↪shuffle=False)
```

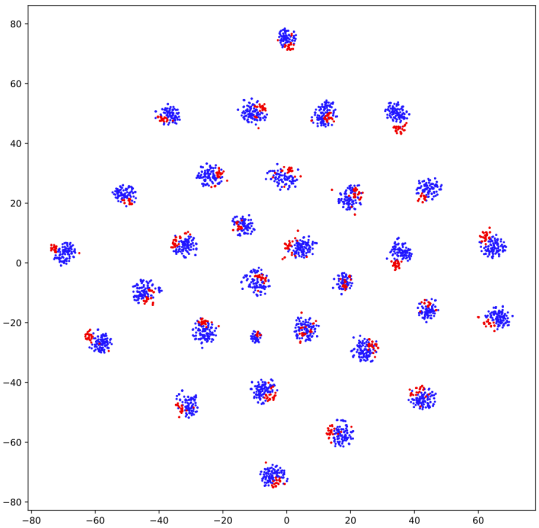
(continues on next page)

(continued from previous page)

```
with torch.no_grad():
    for loader in [source_val_loader, val_loader]:
        for i, (images, target) in enumerate(loader):
            images = images.cuda()
            target = target.cuda()

            # compute output
            _, f = classifier(images)
            features.extend(f.cpu().numpy().tolist())

features = np.array(features)

# map features to 2-d using TSNE
X_tsne = TSNE(n_components=2, random_state=33).fit_transform(features)

# domain labels, 1 represents source while 0 represents target
domains = np.concatenate((np.ones(len(source_val_dataset)), np.zeros(len(val_
↪dataset))))

# visualize using matplotlib
import matplotlib.pyplot as plt
import matplotlib.colors as col
plt.figure(figsize=(10, 10))
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=domains, cmap=col.ListedColormap(["r", "b
↪"]), s=2)
plt.show()
```

Figures below shows the T-SNE visualization results of A → W on ResNet50 (source only) and DANN.





DANN

The source and target are not aligned well with ResNet (source only), better aligned with DANN. For better alignment, you are encouraged to replace DANN with CDAN.

Runnable code can be found in Tutorial. The following script is expected to achieve ~86% accuracy.

```
python examples/tutorials.py data/office31 -d Office31 -s A -t W -a resnet50 --epochs␣
↪10 --seed 0
```

CHAPTER 2

DALIB Basic Modules

2.1 Classifier

class dalib.modules.classifier.**Classifier**(*backbone: torch.nn.modules.module.Module*, *num_classes: int*, *bottleneck: Optional[torch.nn.modules.module.Module] = None*, *bottleneck_dim: Optional[int] = -1*, *head: Optional[torch.nn.modules.module.Module] = None*)

Bases: `torch.nn.modules.module.Module`

A generic Classifier class for domain adaptation.

Parameters:

- **backbone** (class:*nn.Module* object): Any backbone to extract 1-d features from data
- **num_classes** (int): Number of classes
- **bottleneck** (class:*nn.Module* object, optional): Any bottleneck layer. Use no bottleneck by default
- **bottleneck_dim** (int, optional): Feature dimension of the bottleneck layer. Default: -1
- **head** (class:*nn.Module* object, optional): Any classifier head. Use *nn.Linear* by default

Note: Different classifiers are used in different domain adaptation algorithms to achieve better accuracy respectively, and we provide a suggested *Classifier* for different algorithms. Remember they are not the core of algorithms. You can implement your own *Classifier* and combine it with the domain adaptation algorithm in this algorithm library.

Note: The learning rate of this classifier is set 10 times to that of the feature extractor for better accuracy by default. If you have other optimization strategies, please over-ride *get_parameters*.

Inputs:

- **x** (tensor): input data fed to *backbone*

Outputs: predictions, features

- **predictions**: classifier's predictions
- **features**: features after *bottleneck* layer and before *head* layer

Shape:

- Inputs: (minibatch, *) where * means, any number of additional dimensions
- predictions: (minibatch, *num_classes*)
- features: (minibatch, *features_dim*)

features_dim
The dimension of features before the final *head* layer

get_parameters() → List[Dict[KT, VT]]
A parameter list which decides optimization hyper-parameters, such as the relative learning rate of each layer

2.2 Domain Discriminator

class dalib.modules.domain_discriminator.**DomainDiscriminator**(*in_feature: int*, *hidden_size: int*)

Bases: torch.nn.modules.module.Module

Domain discriminator model from "Domain-Adversarial Training of Neural Networks"

Distinguish whether the input features come from the source domain or the target domain. The source domain label is 1 and the target domain label is 0.

Parameters:

- **in_feature** (int): dimension of the input feature
- **hidden_size** (int): dimension of the hidden features

Shape:

- Inputs: (minibatch, *in_feature*)
- Outputs: $(minibatch, 1)$

2.3 GRL

class dalib.modules.grl.**WarmStartGradientReverseLayer**(*alpha: Optional[float] = 1.0*, *lo: Optional[float] = 0.0*, *hi: Optional[float] = 1.0*, *max_iters: Optional[int] = 1000.0*, *auto_step: Optional[bool] = False*)

Bases: torch.nn.modules.module.Module

Gradient Reverse Layer $\mathcal{R}(x)$ with warm start

The forward and backward behaviours are:

$$\mathcal{R}(x) = x,$$
$$\frac{d\mathcal{R}}{dx} = -\lambda I.$$

λ is initiated at lo and is gradually changed to hi using the following schedule:

$$\lambda = \frac{2(hi - lo)}{1 + \exp(-\frac{i}{N})} - (hi - lo) + lo$$

where i is the iteration step.

Parameters:

- **alpha** (float, optional): . Default: 1.0
- **lo** (float, optional): Initial value of λ. Default: 0.0
- **hi** (float, optional): Final value of λ. Default: 1.0
- **max_iters** (int, optional): N. Default: 1000
- **auto_step** (bool, optional): If True, increase i each time *forward* is called. Otherwise use function *step* to increase i. Default: False

step()
Increase iteration number i by 1

2.4 Kernels

class dalib.modules.kernels.**GaussianKernel**(*sigma: Optional[float] = None*, *track_running_stats: Optional[bool] = True*, *alpha: Optional[float] = 1.0*)

Bases: torch.nn.modules.module.Module

Gaussian Kernel Matrix

Gaussian Kernel k is defined by

$$k(x_1, x_2) = \exp\left(-\frac{\|x_1 - x_2\|^2}{2\sigma^2}\right)$$

where $x_1, x_2 \in R^d$ are 1-d tensors.

Gaussian Kernel Matrix K is defined on input group $X = (x_1, x_2, ..., x_m)$,

$$K(X)_{i,j} = k(x_i, x_j)$$

Also by default, during training this layer keeps running estimates of the mean of L2 distances, which are then used to set hyperparameter σ. Mathematically, the estimation is $\sigma^2 = \frac{\alpha}{n^2}\sum_{i,j}\|x_i - x_j\|^2$. If `track_running_stats` is set to `False`, this layer then does not keep running estimates, and use a fixed σ instead.

Parameters:

- sigma (float, optional): bandwidth σ. Default: None
- track_running_stats (bool, optional): If `True`, this module tracks the running mean of σ^2. Otherwise, it won't track such statistics and always uses fix σ^2. Default: `True`

- alpha (float, optional): α which decides the magnitude of σ^2 when track_running_stats is set to `True`

Inputs:

- X (tensor): input group X

Shape:

- Inputs: $(minibatch, F)$ where F means the dimension of input features.
- Outputs: $(minibatch, minibatch)$

CHAPTER 3

DALIB Algorithms

The adaptation subpackage contains definitions for the following domain adaptation algorithms:

- DANN
- DAN
- JAN
- CDAN
- MCD
- MDD

Besides specific algorithms, this package also provides a recommended image classifier for each algorithms.

We provide benchmarks of different domain adaptation algorithms on *Office-31*, *Office-Home* and *VisDA-2017* as follows. Note that *Origin* means the accuracy reported by the original paper, while *Avg* is the accuracy reported by DALIB.

Office-31 accuracy on ResNet-50

Methods	Origin	Avg	A → W	D → W	W → D	A → D	D → A	W → A
Source Only	76.1	79.5	75.8	95.5	99.0	79.3	63.6	63.8
DANN	82.2	86.4	91.7	97.9	100.0	82.9	72.8	73.3
DAN	80.4	83.7	84.2	98.4	100.0	87.3	66.9	65.2
JAN	84.3	87.3	93.7	98.4	100.0	89.4	71.2	71.0
CDAN	87.7	88.7	93.1	98.6	100.0	93.4	75.6	71.5
MCD	/	85.9	91.8	98.6	100.0	89.0	69.0	66.9
MDD	88.9	89.2	93.6	98.6	100.0	93.6	76.7	72.9

Office-Home accuracy on ResNet-50

Methods	Origin	Avg	Ar → Cl	Ar → Pr	Ar → Rw	Cl → Ar	Cl → Pr	Cl → Rw	Pr → Ar	Pr → Cl	Pr → Rw	Rw → Ar	Rw → Cl	Rw → Pr
Source Only	46.1	58.2	41.5	65.8	73.6	52.2	59.5	63.6	51.5	36.4	71.3	65.2	42.8	75.4
DANN	57.6	65.5	52.7	61.8	73.4	57.4	67.2	69.6	57.2	55.4	79.0	71.4	60.0	81.1
DAN	56.3	61.6	45.5	67.9	73.9	57.6	63.7	66.2	55.2	39.7	74.3	66.8	49.1	78.7
JAN	58.3	65.9	50.4	71.8	76.7	60.0	67.7	68.9	60.4	49.8	77.0	71.2	55.6	81.0
CDAN	65.8	68.8	54.4	70.9	77.9	61.6	71.1	71.9	62.3	54.9	80.7	75.1	60.8	83.7
MCD	/	67.8	51.7	72.2	78.2	63.7	69.5	70.8	61.5	52.8	78.0	74.5	58.4	81.8
MDD	68.1	69.5	56.2	74.9	78.8	63.4	72.5	72.6	63.8	54.6	80.0	73.5	60.1	83.7

VisDA-2017 accuracy on ResNet-50 and ResNet-101

Methods	Origin	DALIB	Origin	DALIB
Backbone	ResNet-50	ResNet-50	ResNet-101	ResNet-101
Source Only	/	55.1	52.4	58.3
DANN	/	72.6	57.4	72.9
DAN	/	60.6	61.1	64.8
JAN	61.6	64.9	/	68.0
CDAN	66.8	74.6	/	74.5
MCD	69.2	69.1	71.9	77.3
MDD	74.6	74.9	/	78.5

DomainNet accuracy on ResNet-101

Source Only	clp	inf	pnt	real	skt	Avg
clp	N/A	18.0	32.7	50.6	39.4	35.2
inf	35.7	N/A	31.1	50.0	26.5	35.8
pnt	41.1	17.8	N/A	56.8	35.0	37.7
real	48.6	22.9	48.8	N/A	36.1	39.1
skt	49.0	15.3	34.8	46.1	N/A	36.3
Avg	43.6	18.5	36.9	50.9	34.3	36.8

DANN	clp	inf	pnt	real	skt	Avg
clp	N/A	19.7	35.4	53.9	44.2	38.3
inf	26.7	N/A	23.8	28.8	23.7	25.8
pnt	37.2	18.7	N/A	51.1	36.0	35.8
real	50.6	22.1	47.9	N/A	39.0	39.9
skt	54.0	19.7	42.7	52.8	N/A	42.3
Avg	42.1	20.1	37.5	46.7	35.7	36.4

DAN	clp	inf	pnt	real	skt	Avg
clp	N/A	17.3	37.9	54.0	42.6	38.0
inf	34.9	N/A	33.4	46.5	29.9	36.2
pnt	43.9	17.7	N/A	55.9	39.3	39.2
real	50.1	20.0	48.6	N/A	38.4	39.3
skt	54.2	17.5	44.2	53.4	N/A	42.3
Avg	45.8	18.1	41.0	52.5	37.6	39.0

CDAN	clp	inf	pnt	real	skt	Avg
clp	N/A	20.8	40.0	56.1	45.5	40.6
inf	31.2	N/A	30.0	41.4	24.7	31.8
pnt	44.6	20.5	N/A	57.0	39.9	40.5
real	55.3	24.1	52.6	N/A	42.4	43.6
skt	56.7	21.3	46.2	55.0	N/A	44.8
Avg	47.0	21.7	42.2	52.4	38.1	40.3

MDD	clp	inf	pnt	real	skt	Avg
clp	N/A	21.2	42.9	59.5	47.5	42.8
inf	35.3	N/A	34.0	49.6	29.4	37.1
pnt	48.6	19.7	N/A	59.4	42.6	42.6
real	58.3	24.9	53.7	N/A	46.2	45.8
skt	58.7	20.7	46.5	57.7	N/A	45.9
Avg	50.2	21.6	44.3	56.6	41.4	42.8

3.1 DANN

class dalib.adaptation.dann.**DomainAdversarialLoss**(*domain_discriminator: torch.nn.modules.module.Module*, *reduction: Optional[str] = 'mean'*)

Bases: `torch.nn.modules.module.Module`

The Domain Adversarial Loss

Domain adversarial loss measures the domain discrepancy through training a domain discriminator. Given domain discriminator D, feature representation f, the definition of DANN loss is

$$loss(\mathcal{D}_s, \mathcal{D}_t) = \mathbb{E}_{x_i^s \sim \mathcal{D}_s} log[D(f_i^s)] + \mathbb{E}_{x_j^t \sim \mathcal{D}_t} log[1 - D(f_j^t)].$$

Parameters:

- **domain_discriminator** (class:*nn.Module* object): A domain discriminator object, which predicts the domains of features. Its input shape is (N, F) and output shape is (N, 1)
- **reduction** (string, optional): Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Default: `'mean'`

Inputs: f_s, f_t

- **f_s** (tensor): feature representations on source domain, f^s
- **f_t** (tensor): feature representations on target domain, f^t

Shape:

- f_s, f_t: (N, F) where F means the dimension of input features.
- Outputs: scalar by default. If :attr:`reduction` is `'none'`, then $(N,)$.

Examples::

```
>>> from dalib.modules.domain_discriminator import DomainDiscriminator
>>> discriminator = DomainDiscriminator(in_feature=1024, hidden_size=1024)
>>> loss = DomainAdversarialLoss(discriminator, reduction='mean')
>>> # features from source domain and target domain
>>> f_s, f_t = torch.randn(20, 1024), torch.randn(20, 1024)
>>> output = loss(f_s, f_t)
```

3.2 DAN

class dalib.adaptation.dan.**MultipleKernelMaximumMeanDiscrepancy**(*kernels: Sequence[torch.nn.modules.module.Modu*
linear: Optional[bool] = False,
quadratic_program: Optional[bool] = False)

Bases: `torch.nn.modules.module.Module`

The Multiple Kernel Maximum Mean Discrepancy (MK-MMD) used in Learning Transferable Features with Deep Adaptation Networks

Given source domain $\mathcal{D}_s$ of n_s labeled points and target domain $\mathcal{D}_t$ of n_t unlabeled points drawn i.i.d. from P and Q respectively, the deep networks will generate activations as $\{z_i^s\}_{i=1}^{n_s}$ and $\{z_i^t\}_{i=1}^{n_t}$. The MK-MMD $D_k(P, Q)$ between probability distributions P and Q is defined as

$$D_k(P, Q) \triangleq \|E_p[\phi(z^s)] - E_q[\phi(z^t)]\|^2_{\mathcal{H}_k},$$

k is a kernel function in the function space

$$\mathcal{K} \triangleq \{k = \sum_{u=1}^{m} \beta_u k_u\}$$

where k_u is a single kernel.

Using kernel trick, MK-MMD can be computed as

$$\begin{aligned}\hat{D}_k(P, Q) = &\frac{1}{n_s^2}\sum_{i=1}^{n_s}\sum_{j=1}^{n_s} k(z_i^s, z_j^s) \\ &+ \frac{1}{n_t^2}\sum_{i=1}^{n_t}\sum_{j=1}^{n_t} k(z_i^t, z_j^t) \\ &- \frac{2}{n_s n_t}\sum_{i=1}^{n_s}\sum_{j=1}^{n_t} k(z_i^s, z_j^t).\end{aligned}$$

Parameters:

- **kernels** (tuple(*nn.Module*)): kernel functions.
- **linear** (bool): whether use the linear version of DAN. Default: False
- **quadratic_program** (bool): whether use quadratic program to solve β. Default: False

Inputs: z_s, z_t

- **z_s** (tensor): activations from the source domain, z^s
- **z_t** (tensor): activations from the target domain, z^t

Shape:

- Inputs: $(minibatch, *)$ where * means any dimension
- Outputs: scalar

Note: Activations z^s and z^t must have the same shape.

Note: The kernel values will add up when there are multiple kernels.

Examples::

```
>>> from dalib.modules.kernels import GaussianKernel
>>> feature_dim = 1024
>>> batch_size = 10
>>> kernels = (GaussianKernel(alpha=0.5), GaussianKernel(alpha=1.),␣
↪GaussianKernel(alpha=2.))
>>> loss = MultipleKernelMaximumMeanDiscrepancy(kernels)
>>> # features from source domain and target domain
>>> z_s, z_t = torch.randn(batch_size, feature_dim), torch.randn(batch_size,␣
↪feature_dim)
>>> output = loss(z_s, z_t)
```

3.3 JAN

class dalib.adaptation.jan.**JointMultipleKernelMaximumMeanDiscrepancy**(*kernels: Sequence[Sequence[torch.nn.modu linear: Optional[bool] = True, thetas: Sequence[torch.nn.modules.modul = None*)

Bases: `torch.nn.modules.module.Module`

The Joint Multiple Kernel Maximum Mean Discrepancy (JMMD) used in Deep Transfer Learning with Joint Adaptation Networks

Given source domain $\mathcal{D}_s$ of n_s labeled points and target domain $\mathcal{D}_t$ of n_t unlabeled points drawn i.i.d. from P and Q respectively, the deep networks will generate activations in layers $\mathcal{L}$ as $\{(z_i^{s1}, ..., z_i^{s|\mathcal{L}|})\}_{i=1}^{n_s}$ and $\{(z_i^{t1}, ..., z_i^{t|\mathcal{L}|})\}_{i=1}^{n_t}$. The empirical estimate of $\hat{D}_{\mathcal{L}}(P, Q)$ is computed as the squared distance between the

empirical kernel mean embeddings as

$$\hat{D}_{\mathcal{L}}(P, Q) = \frac{1}{n_s^2} \sum_{i=1}^{n_s} \sum_{j=1}^{n_s} \prod_{l \in \mathcal{L}} k^l(z_i^{sl}, z_j^{sl}) + \frac{1}{n_t^2} \sum_{i=1}^{n_t} \sum_{j=1}^{n_t} \prod_{l \in \mathcal{L}} k^l(z_i^{tl}, z_j^{tl}) - \frac{2}{n_s n_t} \sum_{i=1}^{n_s} \sum_{j=1}^{n_t} \prod_{l \in \mathcal{L}} k^l(z_i^{sl}, z_j^{tl}).$$

Parameters:

- **kernels** (tuple(tuple(*nn.Module*))): kernel functions, where *kernels[r]* corresponds to kernel $k^{\mathcal{L}[r]}$.
- **linear** (bool): whether use the linear version of JAN. Default: False
- **thetas** (list(*Theta*): use adversarial version JAN if not None. Default: None

Inputs: z_s, z_t

- **z_s** (tuple(tensor)): multiple layers' activations from the source domain, z^s
- **z_t** (tuple(tensor)): multiple layers' activations from the target domain, z^t

Shape:

- z^{sl} and z^{tl}: $(minibatch, *)$ where * means any dimension
- Outputs: scalar

Note: Activations z^{sl} and z^{tl} must have the same shape.

Note: The kernel values will add up when there are multiple kernels for a certain layer.

Examples::

```
>>> feature_dim = 1024
>>> batch_size = 10
>>> layer1_kernels = (GaussianKernel(alpha=0.5), GaussianKernel(1.),␣
↪GaussianKernel(2.))
>>> layer2_kernels = (GaussianKernel(1.), )
>>> loss = JointMultipleKernelMaximumMeanDiscrepancy((layer1_kernels, layer2_
↪kernels))
>>> # layer1 features from source domain and target domain
>>> z1_s, z1_t = torch.randn(batch_size, feature_dim), torch.randn(batch_size,
↪ feature_dim)
>>> # layer2 features from source domain and target domain
>>> z2_s, z2_t = torch.randn(batch_size, feature_dim), torch.randn(batch_size,
↪ feature_dim)
>>> output = loss((z1_s, z2_s), (z1_t, z2_t))
```

3.4 CDAN

class dalib.adaptation.cdan.**ConditionalDomainAdversarialLoss**(*domain_discriminator: torch.nn.modules.module.Module*, *entropy_conditioning: Optional[bool] = False*, *randomized: Optional[bool] = False*, *num_classes: Optional[int] = -1*, *features_dim: Optional[int] = -1*, *randomized_dim: Optional[int] = 1024*, *reduction: Optional[str] = 'mean'*)

Bases: `torch.nn.modules.module.Module`

The Conditional Domain Adversarial Loss

Conditional Domain adversarial loss measures the domain discrepancy through training a domain discriminator in a conditional manner. Given domain discriminator D, feature representation f and classifier predictions g, the definition of CDAN loss is

$$\begin{aligned} loss(\mathcal{D}_s, \mathcal{D}_t) = &\mathbb{E}_{x_i^s \sim \mathcal{D}_s} log[D(T(f_i^s, g_i^s))] \\ &+ \mathbb{E}_{x_j^t \sim \mathcal{D}_t} log[1 - D(T(f_j^t, g_j^t))], \end{aligned}$$

where T is a *multi linear map* or *randomized multi linear map* which convert two tensors to a single tensor.

Parameters:

- **domain_discriminator** (class:*nn.Module* object): A domain discriminator object, which predicts the domains of features. Its input shape is (N, F) and output shape is (N, 1)
- **entropy_conditioning** (bool, optional): If True, use entropy-aware weight to reweight each training example. Default: False
- **randomized** (bool, optional): If True, use *randomized multi linear map*. Else, use *multi linear map*. Default: False
- **num_classes** (int, optional): Number of classes. Default: -1
- **features_dim** (int, optional): Dimension of input features. Default: -1
- **randomized_dim** (int, optional): Dimension of features after randomized. Default: 1024
- **reduction** (string, optional): Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Default: `'mean'`

Note: You need to provide *num_classes*, *features_dim* and *randomized_dim* **only when** *randomized* is set True.

Inputs: g_s, f_s, g_t, f_t

- **g_s** (tensor): unnormalized classifier predictions on source domain, g^s

- **f_s** (tensor): feature representations on source domain, f^s
- **g_t** (tensor): unnormalized classifier predictions on target domain, g^t
- **f_t** (tensor): feature representations on target domain, f^t

Shape:

- g_s, g_t: $(minibatch, C)$ where C means the number of classes.
- f_s, f_t: $(minibatch, F)$ where F means the dimension of input features.
- Output: scalar by default. If :attr:`reduction` is `'none'`, then $(minibatch,)$.

Examples::

```
>>> from dalib.modules.domain_discriminator import DomainDiscriminator
>>> num_classes = 2
>>> feature_dim = 1024
>>> batch_size = 10
>>> discriminator = DomainDiscriminator(in_feature=feature_dim, hidden_
↪size=1024)
>>> loss = ConditionalDomainAdversarialLoss(discriminator, reduction='mean')
>>> # features from source domain and target domain
>>> f_s, f_t = torch.randn(batch_size, feature_dim), torch.randn(batch_size,␣
↪feature_dim)
>>> # logits output from source domain adn target domain
>>> g_s, g_t = torch.randn(batch_size, num_classes), torch.randn(batch_size,␣
↪num_classes)
>>> output = loss(g_s, f_s, g_t, f_t)
```

class dalib.adaptation.cdan.**RandomizedMultiLinearMap**(*features_dim: int*, *num_classes: int*, *output_dim: Optional[int] = 1024*)

Bases: `torch.nn.modules.module.Module`

Random multi linear map

Given two inputs f and g, the definition is

$$T_{\odot}(f, g) = \frac{1}{\sqrt{d}}(R_f f) \odot (R_g g),$$

where $\odot$ is element-wise product, R_f and R_g are random matrices sampled only once and xed in training.

Parameters:

- **features_dim** (int): dimension of input f
- **num_classes** (int): dimension of input g
- **output_dim** (int, optional): dimension of output tensor. Default: 1024

Shape:

- f: (minibatch, features_dim)
- g: (minibatch, num_classes)
- Outputs: (minibatch, output_dim)

class dalib.adaptation.cdan.**MultiLinearMap**

Bases: `torch.nn.modules.module.Module`

Multi linear map

Shape:

- f: (minibatch, F)
- g: (minibatch, C)
- Outputs: (minibatch, F * C)

3.5 MCD

dalib.adaptation.mcd.**classifier_discrepancy**(*predictions1: torch.Tensor*, *predictions2: torch.Tensor*) → torch.Tensor

The *Classifier Discrepancy* in Maximum Classier Discrepancy for Unsupervised Domain Adaptation. The classifier discrepancy between predictions p_1 and p_2 can be described as:

$$d(p_1, p_2) = \frac{1}{K} \sum_{k=1}^{K} |p_{1k} - p_{2k}|,$$

where K is number of classes.

Parameters:

- **predictions1** (tensor): Classifier predictions p_1. Expected to contain raw, normalized scores for each class
- **predictions2** (tensor): Classifier predictions p_2

dalib.adaptation.mcd.**entropy**(*predictions: torch.Tensor*) → torch.Tensor

Entropy of N predictions $(p_1, p_2, ..., p_N)$. The definition is:

$$d(p_1, p_2, ..., p_N) = -\frac{1}{K} \sum_{k=1}^{K} \log \left(\frac{1}{N} \sum_{i=1}^{N} p_{ik} \right)$$

where K is number of classes.

Parameters:

- **predictions** (tensor): Classifier predictions. Expected to contain raw, normalized scores for each class

class dalib.adaptation.mcd.**ImageClassifierHead**(*in_features: int*, *num_classes: int*, *bottleneck_dim: Optional[int] = 1024*)

Bases: torch.nn.modules.module.Module

Classifier Head for MCD. Parameters:

- **in_features** (int): Dimension of input features
- **num_classes** (int): Number of classes
- **bottleneck_dim** (int, optional): Feature dimension of the bottleneck layer. Default: 1024

Shape:

- Inputs: $(minibatch, F)$ where F = *in_features*.
- Output: $(minibatch, C)$ where C = *num_classes*.

3.6 MDD

class dalib.adaptation.mdd.**MarginDisparityDiscrepancy**(*margin: Optional[int] = 4*, *reduction: Optional[str] = 'mean'*)

Bases: torch.nn.modules.module.Module

The margin disparity discrepancy (MDD) is proposed to measure the distribution discrepancy in domain adaptation.

The y^s and y^t are logits output by the main classifier on the source and target domain respectively. The y^s_{adv} and y^t_{adv} are logits output by the adversarial classifier. They are expected to contain raw, unnormalized scores for each class.

The definition can be described as:

$$\mathcal{D}_\gamma(\hat{\mathcal{S}}, \hat{\mathcal{T}}) = \gamma \mathbb{E}_{y^s, y^s_{adv} \sim \hat{\mathcal{S}}} \log \left(\frac{\exp(y^s_{adv}[h_{y^s}])}{\sum_j \exp(y^s_{adv}[j])} \right) + \mathbb{E}_{y^t, y^t_{adv} \sim \hat{\mathcal{T}}} \log \left(1 - \frac{\exp(y^t_{adv}[h_{y^t}])}{\sum_j \exp(y^t_{adv}[j])} \right),$$

where γ is a margin hyper-parameter and h_y refers to the predicted label when the logits output is y. You can see more details in Bridging Theory and Algorithm for Domain Adaptation.

Parameters:

- **margin** (float): margin γ. Default: 4
- **reduction** (string, optional): Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Default: `'mean'`

Inputs: y_s, y_s_adv, y_t, y_t_adv

- **y_s**: logits output y^s by the main classifier on the source domain
- **y_s_adv**: logits output y^s by the adversarial classifier on the source domain
- **y_t**: logits output y^t by the main classifier on the target domain
- **y_t_adv**: logits output y^t_{adv} by the adversarial classifier on the target domain

Shape:

- Inputs: $(minibatch, C)$ where C = number of classes, or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of *K*-dimensional loss.
- Output: scalar. If `reduction` is `'none'`, then the same size as the target: $(minibatch)$, or $(minibatch, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of K-dimensional loss.

Examples::

```
>>> num_classes = 2
>>> batch_size = 10
>>> loss = MarginDisparityDiscrepancy(margin=4.)
>>> # logits output from source domain and target domain
>>> y_s, y_t = torch.randn(batch_size, num_classes), torch.randn(batch_size,␣
↪num_classes)
>>> # adversarial logits output from source domain and target domain
>>> y_s_adv, y_t_adv = torch.randn(batch_size, num_classes), torch.
↪randn(batch_size, num_classes)
>>> output = loss(y_s, y_s_adv, y_t, y_t_adv)
```

class dalib.adaptation.mdd.**ImageClassifier**(*backbone: torch.nn.modules.module.Module*, *num_classes: int*, *bottleneck_dim: Optional[int] = 1024*, *width: Optional[int] = 1024*)

Bases: torch.nn.modules.module.Module

Classifier for MDD. Parameters:

- **backbone** (class:*nn.Module* object): Any backbone to extract 1-d features from data
- **num_classes** (int): Number of classes
- **bottleneck_dim** (int, optional): Feature dimension of the bottleneck layer. Default: 1024
- **width** (int, optional): Feature dimension of the classifier head. Default: 1024

Note: Classifier for MDD has one backbone, one bottleneck, while two classifier heads. The first classifier head is used for final predictions. The adversarial classifier head is only used when calculating MarginDisparityDiscrepancy.

Note: Remember to call function *step()* after function *forward()* **during training phase**! For instance,

```
>>> # x is inputs, classifier is an ImageClassifier
>>> outputs, outputs_adv = classifier(x)
>>> classifier.step()
```

Inputs:

- **x** (Tensor): input data

Outputs: (outputs, outputs_adv)

- **outputs**: logits outputs by the main classifier
- **outputs_adv**: logits outputs by the adversarial classifier

Shapes:

- x: $(minibatch, *)$, same shape as the input of the *backbone*.
- outputs, outputs_adv: $(minibatch, C)$, where C means the number of classes.

dalib.adaptation.mdd.**shift_log**(*x: torch.Tensor*, *offset: Optional[float] = 1e-06*) → torch.Tensor

First shift, then calculate log, which can be described as:

$$y = \max(\log(x + \text{offset}), 0)$$

Used to avoid the gradient explosion problem in log(x) function when x=0.

Parameters:

- **x**: input tensor
- **offset**: offset size. Default: 1e-6

Note: Input tensor falls in [0., 1.] and the output tensor falls in [-log(offset), 0]

CHAPTER 4

Vision Datasets

4.1 ImageList

class dalib.vision.datasets.imagelist.**ImageList**(*root: str, classes: List[str], data_list_file: str, transform: Optional[Callable] = None, target_transform: Optional[Callable] = None*)

Bases: `torchvision.datasets.vision.VisionDataset`

A generic Dataset class for domain adaptation in image classification

Parameters:

- **root** (str): Root directory of dataset
- **classes** (List[str]): The names of all the classes
- **data_list_file** (str): File to read the image list from.
- **transform** (callable, optional): A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`.
- **target_transform** (callable, optional): A function/transform that takes in the target and transforms it.

Note: In *data_list_file*, each line 2 values in the following format.

```
source_dir/dog_xxx.png 0
source_dir/cat_123.png 1
target_dir/dog_xxy.png 0
target_dir/cat_nsdf3.png 1
```

The first value is the relative path of an image, and the second value is the label of the corresponding image. If your data_list_file has different formats, please over-ride *parse_data_file*.

---

num_classes
Number of classes

parse_data_file(*file_name: str*) → List[Tuple[str, int]]
Parse file to data list

Parameters:

- **file_name** (str): The path of data file
- **return** (list): List of (image path, class_index) tuples

4.2 Office-31

class dalib.vision.datasets.office31.**Office31**(*root: str*, *task: str*, *download: Optional[bool] = True*, ***kwargs*)
Bases: *dalib.vision.datasets.imagelist.ImageList*

Office31 Dataset.

Parameters:

- **root** (str): Root directory of dataset
- **task** (str): The task (domain) to create dataset. Choices include `'A'`: amazon, `'D'`: dslr and `'W'`: webcam.
- **download** (bool, optional): If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **transform** (callable, optional): A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`.
- **target_transform** (callable, optional): A function/transform that takes in the target and transforms it.

Note: In *root*, there will exist following files after downloading.

```
amazon/
    images/
        backpack/
            *.jpg
            ...
dslr/
webcam/
image_list/
    amazon.txt
    dslr.txt
    webcam.txt
```

4.3 Office-Caltech

class dalib.vision.datasets.officecaltech.**OfficeCaltech**(*root: str*, *task: str*, *download: Optional[bool] = False*, ***kwargs*)
Bases: `torchvision.datasets.folder.DatasetFolder`

Office+Caltech Dataset.

Parameters:

- **root** (str): Root directory of dataset
- **task** (str): The task (domain) to create dataset. Choices include `'A'`: amazon, `'D'`: dslr, `'W'`:webcam and `'C'`: caltech.
- **download** (bool, optional): If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **transform** (callable, optional): A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`.
- **target_transform** (callable, optional): A function/transform that takes in the target and transforms it.

Note: In *root*, there will exist following files after downloading.

```
amazon/
    images/
        backpack/
            *.jpg
            ...
dslr/
webcam/
caltech/
image_list/
    amazon.txt
    dslr.txt
    webcam.txt
    caltech.txt
```

---

num_classes
Number of classes

4.4 Office-Home

class dalib.vision.datasets.officehome.**OfficeHome**(*root: str*, *task: str*, *download: Optional[bool] = False*, ***kwargs*)

Bases: *dalib.vision.datasets.imagelist.ImageList*

OfficeHome Dataset.

Parameters:

- **root** (str): Root directory of dataset
- **task** (str): The task (domain) to create dataset. Choices include `'Ar'`: Art, `'Cl'`: Clipart, `'Pr'`: Product and `'Rw'`: Real_World.
- **download** (bool, optional): If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **transform** (callable, optional): A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`.
- **target_transform** (callable, optional): A function/transform that takes in the target and transforms it.

Note: In *root*, there will exist following files after downloading.

```
Art/
    Alarm_Clock/*.jpg
    ...
Clipart/
Product/
Real_World/
image_list/
    Art.txt
    Clipart.txt
    Product.txt
    Real_World.txt
```

4.5 VisDA-2017

class dalib.vision.datasets.visda2017.**VisDA2017**(*root: str*, *task: str*, *download: Optional[bool] = False*, ***kwargs*)

Bases: *dalib.vision.datasets.imagelist.ImageList*

VisDA-2017 Dataset

Parameters:

- **root** (str): Root directory of dataset
- **task** (str): The task (domain) to create dataset. Choices include `'T'`: training and `'V'`: validation.
- **download** (bool, optional): If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **transform** (callable, optional): A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`.
- **target_transform** (callable, optional): A function/transform that takes in the target and transforms it.

Note: In *root*, there will exist following files after downloading.

```
train/
    aeroplance/
        *.png
        ...
validation/
image_list/
    train.txt
    validation.txt
```

4.6 DomainNet

class dalib.vision.datasets.domainnet.**DomainNet**(*root: str*, *task: str*, *evaluate: Optional[bool] = False*, *download: Optional[float] = False*, ***kwargs*)

Bases: *dalib.vision.datasets.imagelist.ImageList*

DomainNet (cleaned version, recommended)

See Moment Matching for Multi-Source Domain Adaptation for details.

Parameters:

- **root** (str): Root directory of dataset
- **task** (str): The task (domain) to create dataset. Choices include `'c'`:clipart, `'i'`: infograph, `'p'`: painting, `'q'`: quickdraw, `'r'`: real, `'s'`: sketch
- **evaluate** (bool, optional): If true, use the test set. Otherwise, use the training set. Default: False
- **download** (bool, optional): If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **transform** (callable, optional): A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`.
- **target_transform** (callable, optional): A function/transform that takes in the target and transforms it.

Note: In *root*, there will exist following files after downloading.

```
clipart/
infograph/
painting/
quickdraw/
real/
sketch/
image_list/
    clipart.txt
    ...
```

CHAPTER 5

Vision Models

5.1 ResNets

Modified based on torchvision.models.resnet.

class dalib.vision.models.resnet.**ResNet**(**args*, ***kwargs*)

Bases: torchvision.models.resnet.ResNet

ResNets without fully connected layer

forward(*x*)

out_features

The dimension of output features

dalib.vision.models.resnet.**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 ImageNet
- **progress** (bool): If True, displays a progress bar of the download to stderr

dalib.vision.models.resnet.**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 ImageNet
- **progress** (bool): If True, displays a progress bar of the download to stderr

dalib.vision.models.resnet.**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 ImageNet

- **progress** (bool): If True, displays a progress bar of the download to stderr

dalib.vision.models.resnet.**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 ImageNet
- **progress** (bool): If True, displays a progress bar of the download to stderr

dalib.vision.models.resnet.**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 ImageNet
- **progress** (bool): If True, displays a progress bar of the download to stderr

dalib.vision.models.resnet.**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 ImageNet
- **progress** (bool): If True, displays a progress bar of the download to stderr

dalib.vision.models.resnet.**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 ImageNet
- **progress** (bool): If True, displays a progress bar of the download to stderr

dalib.vision.models.resnet.**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 bottleneck number of channels which is twice larger in every block. The number of channels in outer 1x1 convolutions is the same, e.g. last block in ResNet-50 has 2048-512-2048 channels, and in Wide ResNet-50-2 has 2048-1024-2048.

Parameters:

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

dalib.vision.models.resnet.**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 bottleneck number of channels which is twice larger in every block. The number of channels in outer 1x1 convolutions is the same, e.g. last block in ResNet-50 has 2048-512-2048 channels, and in Wide ResNet-50-2 has 2048-1024-2048.

Parameters:

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

Python Module Index

d

Index

R

S

V

W