MoCo: CIFAR-10

References

This demo is adapted from:

- https://github.com/facebookresearch/moco
- http://github.com/zhirongw/lemniscate.pytorch
- https://github.com/leftthomas/SimCLR

```
gpu_info = !nvidia-smi -i 0
gpu_info = '\n'.join(gpu_info)
print(gpu_info)

from datetime import datetime
from functools import partial
from PIL import Image
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import CIFAR10
from torchvision.models import resnet
from tqdm import tqdm
import argparse
import json
import math
import os
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
Mon Dec 13 03:55:39 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 495.44       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   32C    P8    28W / 149W |      0MiB / 11441MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
```

```
|  No running processes found                                                |
+-----------------------------------------------------------------------------+
```

Set arguments

```
parser = argparse.ArgumentParser(description='Train MoCo on CIFAR-10')

parser.add_argument('-a', '--arch', default='resnet18')

# lr: 0.06 for batch 512 (or 0.03 for batch 256)
parser.add_argument('--lr', '--learning-rate', default=0.06, type=float, metavar='LR'
parser.add_argument('--epochs', default=10, type=int, metavar='N', help='number of to
parser.add_argument('--schedule', default=[120, 160], nargs='*', type=int, help='lear
parser.add_argument('--cos', action='store_true', help='use cosine lr schedule')

parser.add_argument('--batch-size', default=512, type=int, metavar='N', help='mini-ba
parser.add_argument('--wd', default=5e-4, type=float, metavar='W', help='weight decay

# moco specific configs:
parser.add_argument('--moco-dim', default=128, type=int, help='feature dimension')
parser.add_argument('--moco-k', default=4096, type=int, help='queue size; number of n
parser.add_argument('--moco-m', default=0.99, type=float, help='moco momentum of upda
parser.add_argument('--moco-t', default=0.1, type=float, help='softmax temperature')

parser.add_argument('--bn-splits', default=8, type=int, help='simulate multi-gpu beha

parser.add_argument('--symmetric', action='store_true', help='use a symmetric loss fu

# knn monitor
parser.add_argument('--knn-k', default=200, type=int, help='k in kNN monitor')
parser.add_argument('--knn-t', default=0.1, type=float, help='softmax temperature in

# utils
parser.add_argument('--resume', default='', type=str, metavar='PATH', help='path to l
parser.add_argument('--results-dir', default='', type=str, metavar='PATH', help='path

'''
args = parser.parse_args()  # running in command line
'''
args = parser.parse_args('')  # running in ipynb

# set command line arguments here when running in ipynb
args.schedule = []  # cos in use
args.symmetric = False
if args.results_dir == '':
    args.results_dir = './cache-' + datetime.now().strftime("%Y-%m-%d-%H-%M-%S-moco")

print(args)
```

```
Namespace(arch='resnet18', batch_size=512, bn_splits=8, cos=False, epochs=10, kn
```

Define data loaders

```
class CIFAR10Pair(CIFAR10):
    """CIFAR10 Dataset.
    """
    def __getitem__(self, index):
        img = self.data[index]
        img = Image.fromarray(img)

        if self.transform is not None:
            im_1 = self.transform(img)
            im_2 = self.transform(img)

        return im_1, im_2

train_transform = transforms.Compose([
    transforms.RandomResizedCrop(32),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomApply([transforms.ColorJitter(0.4, 0.4, 0.4, 0.1)], p=0.8),
    transforms.RandomGrayscale(p=0.2),
    transforms.ToTensor(),
    transforms.Normalize([0.4914, 0.4822, 0.4465], [0.2023, 0.1994, 0.2010])])

test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.4914, 0.4822, 0.4465], [0.2023, 0.1994, 0.2010])])

# data prepare
train_data = CIFAR10Pair(root='data', train=True, transform=train_transform, download
train_loader = DataLoader(train_data, batch_size=args.batch_size, shuffle=True, num_w

memory_data = CIFAR10(root='data', train=True, transform=test_transform, download=Tru
memory_loader = DataLoader(memory_data, batch_size=args.batch_size, shuffle=False, nu

test_data = CIFAR10(root='data', train=False, transform=test_transform, download=True
test_loader = DataLoader(test_data, batch_size=args.batch_size, shuffle=False, num_wo
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to data/cifa
```

170499072/? [00:03<00:00, 55443107.44it/s]

```
Extracting data/cifar-10-python.tar.gz to data
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserW
  cpuset_checked))
Files already downloaded and verified
Files already downloaded and verified
```

Define base encoder

```
# SplitBatchNorm: simulate multi-gpu behavior of BatchNorm in one gpu by splitting al
# implementation adapted from https://github.com/davidcpage/cifar10-fast/blob/master/
class SplitBatchNorm(nn.BatchNorm2d):
    def __init__(self, num_features, num_splits, **kw):
        super().__init__(num_features, **kw)
        self.num_splits = num_splits

    def forward(self, input):
        N, C, H, W = input.shape
        if self.training or not self.track_running_stats:
            running_mean_split = self.running_mean.repeat(self.num_splits)
            running_var_split = self.running_var.repeat(self.num_splits)
            outcome = nn.functional.batch_norm(
                input.view(-1, C * self.num_splits, H, W), running_mean_split, runnin
                self.weight.repeat(self.num_splits), self.bias.repeat(self.num_splits
                True, self.momentum, self.eps).view(N, C, H, W)
            self.running_mean.data.copy_(running_mean_split.view(self.num_splits, C).
            self.running_var.data.copy_(running_var_split.view(self.num_splits, C).me
            return outcome
        else:
            return nn.functional.batch_norm(
                input, self.running_mean, self.running_var,
                self.weight, self.bias, False, self.momentum, self.eps)

class ModelBase(nn.Module):
    def __init__(self, feature_dim=128, arch=None, bn_splits=16):
        super(ModelBase, self).__init__()

        # use split batchnorm
        norm_layer = partial(SplitBatchNorm, num_splits=bn_splits) if bn_splits > 1 e
        resnet_arch = getattr(resnet, arch)
        net = resnet_arch(num_classes=feature_dim, norm_layer=norm_layer)

        self.net = []
        for name, module in net.named_children():
            if name == 'conv1':
                module = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=Fa
            if isinstance(module, nn.MaxPool2d):
                continue
            if isinstance(module, nn.Linear):
                self.net.append(nn.Flatten(1))
            self.net.append(module)

        self.net = nn.Sequential(*self.net)

    def forward(self, x):
        x = self.net(x)
```

```
        # note: not normalized here
        return x
```

Define MoCo wrapper

```
class ModelMoCo(nn.Module):
    def __init__(self, dim=128, K=4096, m=0.99, T=0.1, arch='resnet18', bn_splits=8,
        super(ModelMoCo, self).__init__()

        self.K = K
        self.m = m
        self.T = T
        self.symmetric = symmetric

        # create the encoders
        self.encoder_q = ModelBase(feature_dim=dim, arch=arch, bn_splits=bn_splits)
        self.encoder_k = ModelBase(feature_dim=dim, arch=arch, bn_splits=bn_splits)

        for param_q, param_k in zip(self.encoder_q.parameters(), self.encoder_k.param
            param_k.data.copy_(param_q.data)  # initialize
            param_k.requires_grad = False  # not update by gradient

        # create the queue
        self.register_buffer("queue", torch.randn(dim, K))
        self.queue = nn.functional.normalize(self.queue, dim=0)

        self.register_buffer("queue_ptr", torch.zeros(1, dtype=torch.long))

    @torch.no_grad()
    def _momentum_update_key_encoder(self):
        """
        Momentum update of the key encoder
        """
        for param_q, param_k in zip(self.encoder_q.parameters(), self.encoder_k.param
            param_k.data = param_k.data * self.m + param_q.data * (1. - self.m)

    @torch.no_grad()
    def _dequeue_and_enqueue(self, keys):
        batch_size = keys.shape[0]

        ptr = int(self.queue_ptr)
        assert self.K % batch_size == 0  # for simplicity

        # replace the keys at ptr (dequeue and enqueue)
        self.queue[:, ptr:ptr + batch_size] = keys.t()  # transpose
        ptr = (ptr + batch_size) % self.K  # move pointer

        self.queue_ptr[0] = ptr
```

```
@torch.no_grad()
def _batch_shuffle_single_gpu(self, x):
    """
    Batch shuffle, for making use of BatchNorm.
    """
    # random shuffle index
    idx_shuffle = torch.randperm(x.shape[0]).cuda()

    # index for restoring
    idx_unshuffle = torch.argsort(idx_shuffle)

    return x[idx_shuffle], idx_unshuffle

@torch.no_grad()
def _batch_unshuffle_single_gpu(self, x, idx_unshuffle):
    """
    Undo batch shuffle.
    """
    return x[idx_unshuffle]

def contrastive_loss(self, im_q, im_k):
    # compute query features
    q = self.encoder_q(im_q)  # queries: NxC
    q = nn.functional.normalize(q, dim=1)  # already normalized

    # compute key features
    with torch.no_grad():  # no gradient to keys
        # shuffle for making use of BN
        im_k_, idx_unshuffle = self._batch_shuffle_single_gpu(im_k)

        k = self.encoder_k(im_k_)  # keys: NxC
        k = nn.functional.normalize(k, dim=1)  # already normalized

        # undo shuffle
        k = self._batch_unshuffle_single_gpu(k, idx_unshuffle)

    # compute logits
    # Einstein sum is more intuitive
    # positive logits: Nx1
    l_pos = torch.einsum('nc,nc->n', [q, k]).unsqueeze(-1)
    # negative logits: NxK
    l_neg = torch.einsum('nc,ck->nk', [q, self.queue.clone().detach()])

    # logits: Nx(1+K)
    logits = torch.cat([l_pos, l_neg], dim=1)

    # apply temperature
    logits /= self.T

    # labels: positive key indicators
```

```
        labels = torch.zeros(logits.shape[0], dtype=torch.long).cuda()

        loss = nn.CrossEntropyLoss().cuda()(logits, labels)

        return loss, q, k

    def forward(self, im1, im2):
        """
        Input:
            im_q: a batch of query images
            im_k: a batch of key images
        Output:
            loss
        """

        # update the key encoder
        with torch.no_grad():  # no gradient to keys
            self._momentum_update_key_encoder()

        # compute loss
        if self.symmetric:  # asymmetric loss
            loss_12, q1, k2 = self.contrastive_loss(im1, im2)
            loss_21, q2, k1 = self.contrastive_loss(im2, im1)
            loss = loss_12 + loss_21
            k = torch.cat([k1, k2], dim=0)
        else:  # asymmetric loss
            loss, q, k = self.contrastive_loss(im1, im2)

        self._dequeue_and_enqueue(k)

        return loss

# create model
model = ModelMoCo(
        dim=args.moco_dim,
        K=args.moco_k,
        m=args.moco_m,
        T=args.moco_t,
        arch=args.arch,
        bn_splits=args.bn_splits,
        symmetric=args.symmetric,
    ).cuda()
print(model.encoder_q)
```

```
            (bn2): SplitBatchNorm(128, eps=1e-05, momentum=0.1, affine=True, track_r
            (downsample): Sequential(
              (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
              (1): SplitBatchNorm(128, eps=1e-05, momentum=0.1, affine=True, track_r
            )
          )
          (1): BasicBlock(
            (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
            (bn1): SplitBatchNorm(128, eps=1e-05, momentum=0.1, affine=True, track_r
```

```
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
        (bn2): SplitBatchNorm(128, eps=1e-05, momentum=0.1, affine=True, track_r
      )
    )
    (5): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
        (bn1): SplitBatchNorm(256, eps=1e-05, momentum=0.1, affine=True, track_r
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
        (bn2): SplitBatchNorm(256, eps=1e-05, momentum=0.1, affine=True, track_r
        (downsample): Sequential(
          (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): SplitBatchNorm(256, eps=1e-05, momentum=0.1, affine=True, track_r
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
        (bn1): SplitBatchNorm(256, eps=1e-05, momentum=0.1, affine=True, track_r
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
        (bn2): SplitBatchNorm(256, eps=1e-05, momentum=0.1, affine=True, track_r
      )
    )
    (6): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
        (bn1): SplitBatchNorm(512, eps=1e-05, momentum=0.1, affine=True, track_r
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
        (bn2): SplitBatchNorm(512, eps=1e-05, momentum=0.1, affine=True, track_r
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): SplitBatchNorm(512, eps=1e-05, momentum=0.1, affine=True, track_r
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
        (bn1): SplitBatchNorm(512, eps=1e-05, momentum=0.1, affine=True, track_r
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
        (bn2): SplitBatchNorm(512, eps=1e-05, momentum=0.1, affine=True, track_r
      )
    )
    (7): AdaptiveAvgPool2d(output_size=(1, 1))
    (8): Flatten(start_dim=1, end_dim=-1)
    (9): Linear(in_features=512, out_features=128, bias=True)
  )
```

Define train/test

```
# train for one epoch
```

```
def train(net, data_loader, train_optimizer, epoch, args):
    #tells model it is training (influences behavior of batchnorm layers)
    net.train()

    #adjusts learning rate on a cosine schedule across epochs
    adjust_learning_rate(optimizer, epoch, args)

    #variables for total loss and total amount of training samples created
    #train_bar is the data_loader passed to train() function,
    #in practice it is the train_loader which consists of batches of pairs of
    #two different data augmentations of the same image.
    total_loss, total_num, train_bar = 0.0, 0, tqdm(data_loader)

    #training loop through batches of pairs of augmentations of a given image in CIFA
    for im_1, im_2 in train_bar:
        #data => gpu
        im_1, im_2 = im_1.cuda(non_blocking=True), im_2.cuda(non_blocking=True)

        #net we are training outputs the contrastive loss between a batch of encoded
        #and encoded dictionary keys in a queue
        #consisting of key images
        loss = net(im_1, im_2)

        #backpropagation
        train_optimizer.zero_grad()
        loss.backward()
        train_optimizer.step()

        #increment total image count, and total loss
        total_num += data_loader.batch_size
        total_loss += loss.item() * data_loader.batch_size
        train_bar.set_description('Train Epoch: [{}/{}], lr: {:.6f}, Loss: {:.4f}'.fo

    #returns average loss for the epoch
    return total_loss / total_num

# lr scheduler for training
def adjust_learning_rate(optimizer, epoch, args):
    """Decay the learning rate based on schedule"""
    lr = args.lr
    if args.cos:  # cosine lr schedule
        lr *= 0.5 * (1. + math.cos(math.pi * epoch / args.epochs))
    else:  # stepwise lr schedule
        for milestone in args.schedule:
            lr *= 0.1 if epoch >= milestone else 1.
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr


# test using a knn monitor
def test(net, memory_data_loader, test_data_loader, epoch, args):
```

```
    net.eval()
    classes = len(memory_data_loader.dataset.classes)
    total_top1, total_top5, total_num, feature_bank = 0.0, 0.0, 0, []
    with torch.no_grad():
        # generate feature bank
        for data, target in tqdm(memory_data_loader, desc='Feature extracting'):
            #print(data[-1].shape)
            #print(target[-1].shape)
            feature = net(data.cuda(non_blocking=True))
            feature = F.normalize(feature, dim=1)
            feature_bank.append(feature)
        # [D, N]
        feature_bank = torch.cat(feature_bank, dim=0).t().contiguous()
        # [N]
        feature_labels = torch.tensor(memory_data_loader.dataset.targets, device=feat
        # loop test data to predict the label by weighted knn search
        test_bar = tqdm(test_data_loader)
        for data, target in test_bar:
            data, target = data.cuda(non_blocking=True), target.cuda(non_blocking=Tru
            feature = net(data)
            feature = F.normalize(feature, dim=1)

            pred_labels = knn_predict(feature, feature_bank, feature_labels, classes,

            total_num += data.size(0)
            total_top1 += (pred_labels[:, 0] == target).float().sum().item()
            test_bar.set_description('Test Epoch: [{}/{}] Acc@1:{:.2f}%'.format(epoch

    return total_top1 / total_num * 100

# knn monitor as in InstDisc https://arxiv.org/abs/1805.01978
# implementation follows http://github.com/zhirongw/lemniscate.pytorch and https://gi
def knn_predict(feature, feature_bank, feature_labels, classes, knn_k, knn_t):
    # compute cos similarity between each feature vector and feature bank ---> [B, N]
    sim_matrix = torch.mm(feature, feature_bank)
    # [B, K]
    sim_weight, sim_indices = sim_matrix.topk(k=knn_k, dim=-1)
    # [B, K]
    sim_labels = torch.gather(feature_labels.expand(feature.size(0), -1), dim=-1, ind
    sim_weight = (sim_weight / knn_t).exp()

    # counts for each class
    one_hot_label = torch.zeros(feature.size(0) * knn_k, classes, device=sim_labels.d
    # [B*K, C]
    one_hot_label = one_hot_label.scatter(dim=-1, index=sim_labels.view(-1, 1), value
    # weighted score ---> [B, C]
    pred_scores = torch.sum(one_hot_label.view(feature.size(0), -1, classes) * sim_we
    #weighted sum of the classes of the k most similar feature vectors for each featu
    pred_labels = pred_scores.argsort(dim=-1, descending=True)
    return pred_labels
```

Start training

```
# define optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=args.lr, weight_decay=args.wd, mom

# load model if resume
epoch_start = 1
if args.resume is not '':
    checkpoint = torch.load(args.resume)
    model.load_state_dict(checkpoint['state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer'])
    epoch_start = checkpoint['epoch'] + 1
    print('Loaded from: {}'.format(args.resume))

# logging
results = {'train_loss': [], 'test_acc@1': []}
if not os.path.exists(args.results_dir):
    os.mkdir(args.results_dir)
# dump args
with open(args.results_dir + '/args.json', 'w') as fid:
    json.dump(args.__dict__, fid, indent=2)

# training loop
for epoch in range(epoch_start, args.epochs + 1):
    train_loss = train(model, train_loader, optimizer, epoch, args)
    results['train_loss'].append(train_loss)
    test_acc_1 = test(model.encoder_q, memory_loader, test_loader, epoch, args)
    results['test_acc@1'].append(test_acc_1)
    # save statistics
    data_frame = pd.DataFrame(data=results, index=range(epoch_start, epoch + 1))
    data_frame.to_csv(args.results_dir + '/log.csv', index_label='epoch')
    # save model
    torch.save({'epoch': epoch, 'state_dict': model.state_dict(), 'optimizer' : optim
```

```
  0%|          | 0/97 [00:00<?, ?it/s]/usr/local/lib/python3.7/dist-packages/tor
  cpuset_checked))
Train Epoch: [1/10], lr: 0.060000, Loss: 3.9896: 100%|██████████| 97/97 [03:46<0
Feature extracting: 100%|██████████| 98/98 [00:47<00:00,  2.05it/s]
Test Epoch: [1/10] Acc@1:64.26%: 100%|██████████| 20/20 [00:13<00:00,  1.51it/s]
Train Epoch: [2/10], lr: 0.060000, Loss: 3.9396: 100%|██████████| 97/97 [03:45<0
Feature extracting: 100%|██████████| 98/98 [00:48<00:00,  2.03it/s]
Test Epoch: [2/10] Acc@1:66.66%: 100%|██████████| 20/20 [00:13<00:00,  1.51it/s]
Train Epoch: [3/10], lr: 0.060000, Loss: 3.8910: 100%|██████████| 97/97 [03:42<0
Feature extracting: 100%|██████████| 98/98 [00:47<00:00,  2.05it/s]
Test Epoch: [3/10] Acc@1:65.92%: 100%|██████████| 20/20 [00:13<00:00,  1.53it/s]
Train Epoch: [4/10], lr: 0.060000, Loss: 3.8470: 100%|██████████| 97/97 [03:44<0
Feature extracting: 100%|██████████| 98/98 [00:48<00:00,  2.04it/s]
Test Epoch: [4/10] Acc@1:67.00%: 100%|██████████| 20/20 [00:13<00:00,  1.52it/s]
Train Epoch: [5/10], lr: 0.060000, Loss: 3.7921: 100%|██████████| 97/97 [03:42<0
Feature extracting: 100%|██████████| 98/98 [00:48<00:00,  2.04it/s]
```

```
Test Epoch: [5/10] Acc@1:66.86%: 100%|██████████| 20/20 [00:13<00:00,  1.49it/s]
Train Epoch: [6/10], lr: 0.060000, Loss: 3.7351: 100%|██████████| 97/97 [03:43<0
Feature extracting: 100%|██████████| 98/98 [00:48<00:00,  2.04it/s]
Test Epoch: [6/10] Acc@1:67.68%: 100%|██████████| 20/20 [00:13<00:00,  1.52it/s]
Train Epoch: [7/10], lr: 0.060000, Loss: 3.7065: 100%|██████████| 97/97 [03:43<0
Feature extracting: 100%|██████████| 98/98 [00:47<00:00,  2.04it/s]
Test Epoch: [7/10] Acc@1:68.57%: 100%|██████████| 20/20 [00:13<00:00,  1.51it/s]
Train Epoch: [8/10], lr: 0.060000, Loss: 3.6652: 100%|██████████| 97/97 [03:43<0
Feature extracting: 100%|██████████| 98/98 [00:47<00:00,  2.05it/s]
Test Epoch: [8/10] Acc@1:68.29%: 100%|██████████| 20/20 [00:13<00:00,  1.51it/s]
Train Epoch: [9/10], lr: 0.060000, Loss: 3.6363: 100%|██████████| 97/97 [03:43<0
Feature extracting: 100%|██████████| 98/98 [00:48<00:00,  2.04it/s]
Test Epoch: [9/10] Acc@1:69.50%: 100%|██████████| 20/20 [00:12<00:00,  1.54it/s]
Train Epoch: [10/10], lr: 0.060000, Loss: 3.5927: 100%|██████████| 97/97 [03:43<
Feature extracting: 100%|██████████| 98/98 [00:48<00:00,  2.04it/s]
Test Epoch: [10/10] Acc@1:69.70%: 100%|██████████| 20/20 [00:12<00:00,  1.54it/s
```

5m 47s completed at 11:01 PM