MoCo

논문명 : Momentum Contrast for Unsupervised Visual Representation Learning

SimCLR 특징

- Negative sample을 mini batch로 다룸
- 따라서 높은 성능을 위해서는 큰 batch size를 사용해야함.

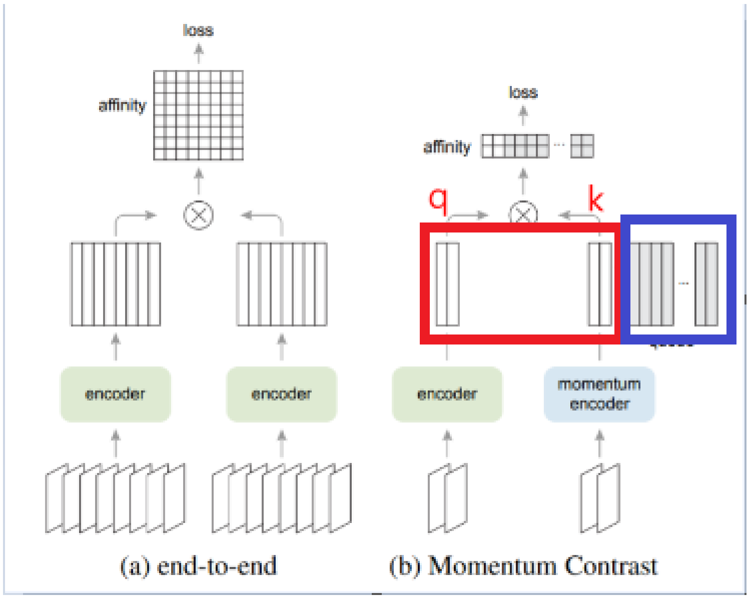
  → 이미지를 batch의 상태로 다루면, 큰 메모리가 필요함.

MoCO 특징

- Queue를 이용하여 embedding 형태로 negative sample을 처리.

  → 기존 : 224 * 224 * 60,000 → Queue : 128 * 60,000

  Queue는 이미지형태가 아니라 백터 형태
- Momentum encoder를 이용하여 모델의 학습에 regularization을 가함.



(a) end-to-end (b) Momentum Contrast

1. Contrastive learning (InfoNCE loss)

$$\mathcal{L}_q = -\log \frac{\exp(q \cdot k_+ / \tau)}{\sum_{i=0}^{K} \exp(q \cdot k_i / \tau)}$$

q : random transform 된 이미지를 임베딩 레이어를 통과한 데이터

k : ‘’ Momentum encoder레이어를 통과한 데이터

positive vs **nagative**

→ batch형태로 또 nagative를 구하는 형태가 아니라

미리 구한 queue가 사용됨

2. Dictionary (Momentum Encoder & Update)

   Contrastive Learning은 positive 와 negative를 비교하는 방법인데 다수의 negative dict와 비교한다

   → dict의 특성은 어때야 한가

   1. 매우 커야 nagative의 noize를 줄일 수 있다

      image가 아닌 임베딩의 형태로 저장

         image : 224 * 224

         embedding : 1 * 128

   2. 일관되어야 한다

      queue가 너무 빠르게 변하면 학습이 어려움

      → momentum encoder를 통해 유지

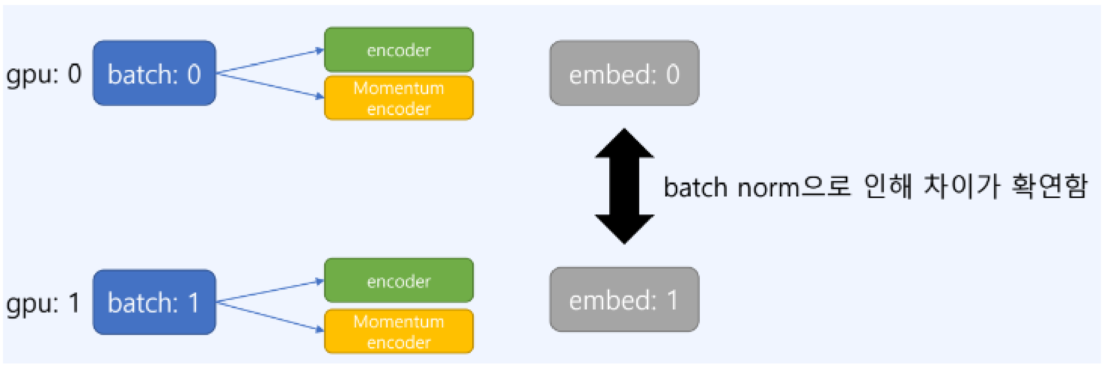
$$\theta_{\mathrm{k}} \leftarrow m\theta_{\mathrm{k}} + (1-m)\theta_{\mathrm{q}}$$

보통 m = 0.999라 변화가 거의 없음

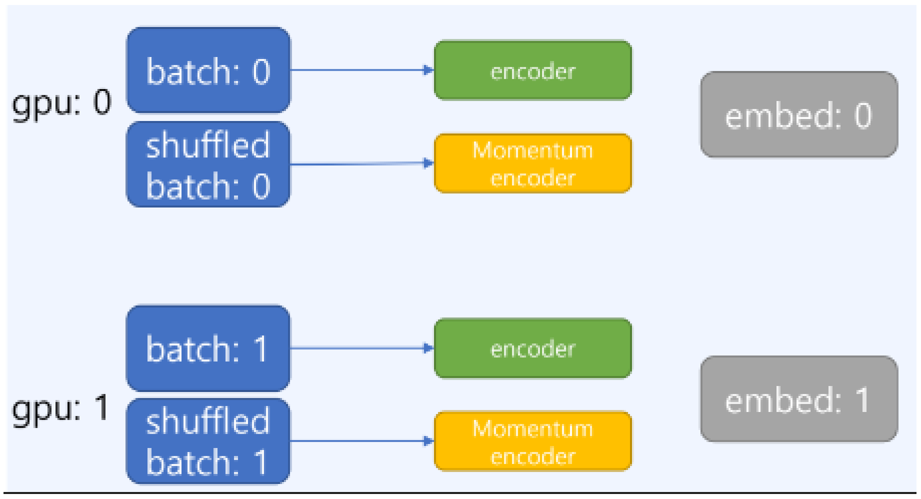
3. Shuffling BatchNorm

   Batch normalization을 통한 정보 누출로 모델의 성능 하락.

   보통 분산처리를 하는데 gpu 0 과 gpu 1의 batch 데이터가 다르므로 각 gpu의 encoder가 다르므로 embedding 결과값이 달라진다

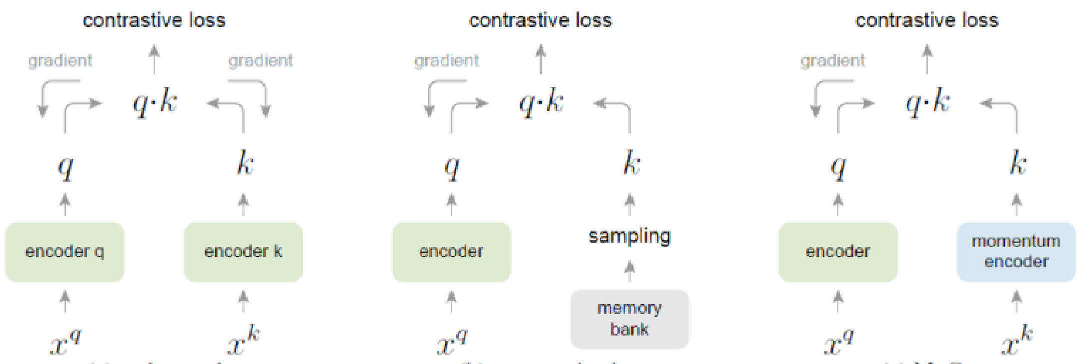


→ key encoder에 데이터를 shuffle하여 입력하고 이를 통해 batch norm에 노이즈를 줌.



shuffled batch를 만들 때 전체 데이터를 섞은 후 넣어 sort cut이 나지 않음

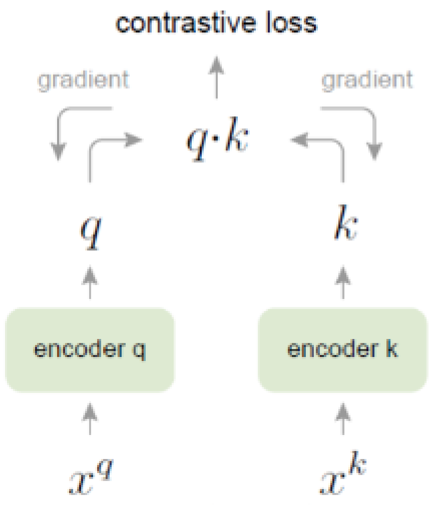
비교군



(a) end-to-end (b) memory bank (c) MoCo

▼ end-to-end

1. 가장 기본적인 구조
   - 두개의 encoder
   - 양방향 back-prop
   - SimCLR의 구조와 동일.
2. Dictionary size가 batch size와 동일
   - 그만큼 GPU의 memory 사용량이 많이 필요.
3. 결과: 성능은 좋지만, dictionary size에 비용이 큼.



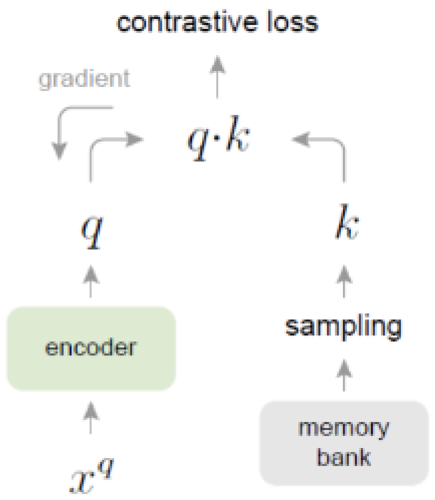
(a) end-to-end

▼ memory bank

1. Memory bank를 사용
   - Memory bank(모든 sample) 로 key 를 구성
   - Dictionary size에 비용이 작게 들어감.
   - Encoder를 통해서 key를 추출하기 때문에,
     계속된 update로 크게 달라진 key를 이용하게 됨.
2. 단방향 back-prop

    - Memory bank가 있기 때문

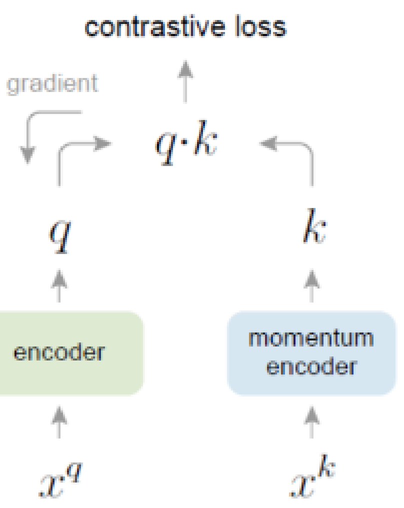
3. 결과: Computational burden이 작게 들어가지만, 성능이 떨어짐.



(b) memory bank

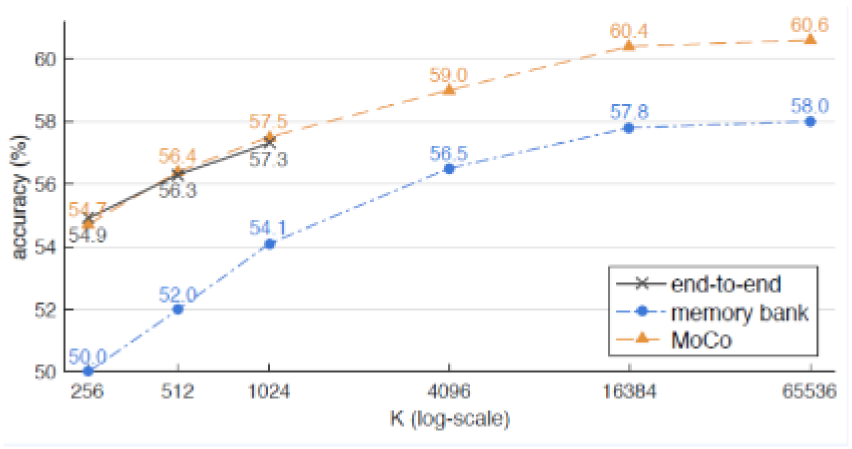
▼ MoCo

1. Momentum encoder를 이용한 dictionary
    - memory bank와 동일한 구조
    - 다만, encoder에서 key를 추출하지 않고, momentum encoder에서 추출
    - 계속된 update로 크게 달라진 key가 아닌, consistent한 dictionary를 얻게 됨.
2. 단방향 back-prop
3. 결과: 가장 좋은 성능



(c) MoCo

실험결과



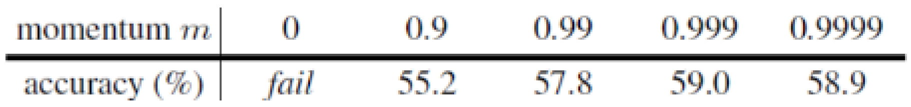
k = dict size

> end-to-end는 Performance: good, dictionary size: Bad.
>
> → 1024이상으로 학습 못함 → 용량초과

> memory bank는 Performance: bad, dictionary size: good.

> MoCo는 Performance: good, dictionary size: good.

평가



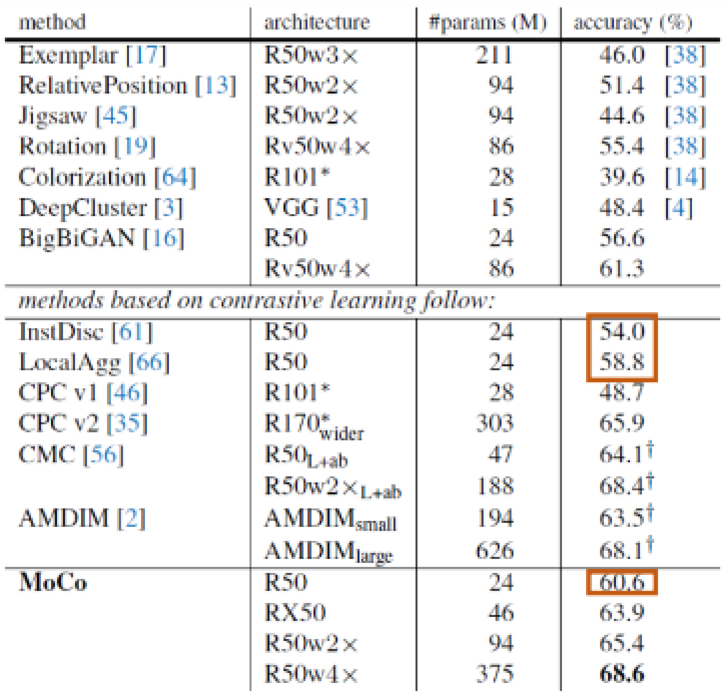
momentum m	0	0.9	0.99	0.999	0.9999
accuracy (%)	*fail*	55.2	57.8	59.0	58.9

💡 Momentum encoder 가 Encoder과 같을 때(m=0)

- 학습이 안됨.
- Dictionary가 너무 빠르게 변하기 때문에, consistency가 지켜지지 않음.

💡 Momentum encoder의 weight가 천천히 변할 수록 성능이 향상 되는 경향을 보임.

💡 0.999일 때, 가장 좋은 성능을 보여주었음.



method	architecture	#params (M)	accuracy (%)
Exemplar [17]	R50w3×	211	46.0 [38]
RelativePosition [13]	R50w2×	94	51.4 [38]
Jigsaw [45]	R50w2×	94	44.6 [38]
Rotation [19]	Rv50w4×	86	55.4 [38]
Colorization [64]	R101*	28	39.6 [14]
DeepCluster [3]	VGG [53]	15	48.4 [4]
BigBiGAN [16]	R50	24	56.6
	Rv50w4×	86	61.3
methods based on contrastive learning follow:			
InstDisc [61]	R50	24	54.0
LocalAgg [66]	R50	24	58.8
CPC v1 [46]	R101*	28	48.7
CPC v2 [35]	$R170^*_{wider}$	303	65.9
CMC [56]	$R50_{L+ab}$	47	$64.1^\dagger$
	$R50w2\times_{L+ab}$	188	$68.4^\dagger$
AMDIM [2]	$AMDIM_{small}$	194	$63.5^\dagger$
	$AMDIM_{large}$	626	$68.1^\dagger$
MoCo	R50	24	60.6
	RX50	46	63.9
	R50w2×	94	65.4
	R50w4×	375	**68.6**

평가

💡 Dictionary 를 이용한 Contrastive learning
Large Dictionary: 임베딩을 memory bank에 저장하는 방식
Consistent Dictionary: Momentum encoder를 이용.
좋은 성능과 효율적인 비용관리.

▼ 코드

```
import argparse
import builtins
import math
import os
import random
import shutil
import time
import warnings
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.distributed as dist
import torch.optim
import torch.multiprocessing as mp
import torch.utils.data
import torch.utils.data.distributed
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torchvision.models as models
import torchvision
import torch.nn.functional as F
from PIL import ImageFilter
from pathlib import Path
from tqdm import tqdm
from PIL import Image, ImageOps, ImageFilter

model_names = sorted(name for name in models.__dict__
                     if name.islower() and not name.startswith("__")
                     and callable(models.__dict__[name]))

parser = argparse.ArgumentParser(description='PyTorch ImageNet Training')
parser.add_argument('--data', metavar='DIR', default='/mnt/MONG/benchmark/ImageNe
t/ILSVRC/Data/ImageNet',
                    help='path to dataset')
parser.add_argument('-a', '--arch', metavar='ARCH', default='resnet50', choices=mo
del_names,
                    help='model architecture: ' + ' | '.join(model_names) +
                         ' (default: resnet50)')
parser.add_argument('-j', '--workers', default=64, type=int, metavar='N',
                    help='number of data loading workers (default: 32)')
parser.add_argument('--epochs', default=100, type=int, metavar='N', help='number o
f total epochs to run')
parser.add_argument('--start-epoch', default=0, type=int, metavar='N', help='manua
l epoch number (useful on restarts)')
parser.add_argument('-b', '--batch-size', default=256, type=int, metavar='N',
                    help='mini-batch size (default: 512), this is the total '
                         'batch size of all GPUs on the current node when '
                         'using Data Parallel or Distributed Data Parallel')
parser.add_argument('--lr', '--learning-rate', default=0.2, type=float, metavar='L
R',
                    help='initial (base) learning rate', dest='lr')
parser.add_argument('--momentum', default=0.9, type=float, metavar='M', help='mome
ntum of SGD solver')
parser.add_argument('--wd', '--weight-decay', default=1e-4, type=float, metavar
='W',
                    help='weight decay (default: 1e-4)', dest='weight_decay')
parser.add_argument('-p', '--print-freq', default=500, type=int, metavar='N', help
='print frequency (default: 10)')
parser.add_argument('--resume', default=True, type=bool)
parser.add_argument('--world-size', default=-1, type=int, help='number of nodes fo
r distributed training')
parser.add_argument('--rank', default=-1, type=int, help='node rank for distribute
d training')
parser.add_argument('--dist-url', default='tcp://224.66.41.62:23456', type=str,
                    help='url used to set up distributed training')
parser.add_argument('--gpu', default='0,1,2,3', type=str, help='GPU id to use.')

parser.add_argument('--projector', default='8192-8192-8192', type=str, metavar='ML
P', help='projector MLP')
parser.add_argument('--tau', default=0.1, type=float, help='temperature (default:
 0.1)')
parser.add_argument('--dim', default=2048, type=int, help='feature dimension (defa
ult: 2048)')
parser.add_argument('--pred-dim', default=512, type=int, help='hidden dimension of
the predictor (default: 512)')
parser.add_argument('--fix-pred-lr', action='store_true', help='Fix learning rate
 for the predictor')
parser.add_argument('--checkpoint', default='./checkpoint', type=Path, metavar='DI
R',
                    help='path to checkpoint directory')


def main():
    args = parser.parse_args()
    os.environ["CUDA_VISIBLE_DEVICES"] = args.gpu
    args.ngpus_per_node = len(args.gpu.split(','))
    args.rank = 0
    args.dist_url = f'tcp://localhost:{random.randrange(49152, 65535)}'
    args.world_size = args.ngpus_per_node
    args.distributed = True

    if args.rank == 0:
        args.checkpoint.mkdir(parents=True, exist_ok=True)
    print('start')
    torch.multiprocessing.spawn(main_worker, (args,), args.ngpus_per_node)


def main_worker(gpu, args):
    args.gpu = gpu
    args.rank = gpu

    torch.distributed.init_process_group(
        backend='nccl', init_method=args.dist_url,
        world_size=args.world_size, rank=args.rank)

    # create model
    print("=> creating model '{}'".format(args.arch))
    model = MoCo()

    # infer learning rate before changing batch size
    init_lr = args.lr * args.batch_size / 256

    torch.cuda.set_device(args.gpu)
    torch.backends.cudnn.benchmark = True
    model.cuda(args.gpu)

    model = torch.nn.SyncBatchNorm.convert_sync_batchnorm(model)
    # When using a single GPU per process and per
    # DistributedDataParallel, we need to divide the batch size
    # ourselves based on the total number of GPUs we have
    per_batch_size = int(args.batch_size / args.ngpus_per_node)
    args.workers = args.ngpus_per_node
    model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[args.gp
u])
    optim_params = model.parameters()
    optimizer = torch.optim.SGD(optim_params, init_lr, momentum=args.momentum, wei
ght_decay=args.weight_decay)

    # optionally resume from a checkpoint
    if args.resume:
        if (args.checkpoint / 'checkpoint.pth').is_file():
            print("=> loading checkpoint '{}'".format(args.checkpoint))
            checkpoint = torch.load(args.checkpoint / 'checkpoint.pth', map_locati
on='cpu')
            args.start_epoch = checkpoint['epoch']
            model.load_state_dict(checkpoint['model'])
            print("=> loaded checkpoint '{}' (epoch {})"
                  .format(args.checkpoint, '{}/checkpoint.pth'.format(checkpoint
['epoch'])))
        else:
            print("=> no checkpoint found at '{}'".format(args.checkpoint))

    cudnn.benchmark = True

    # Data loading code
    traindir = args.data

    augmentation = [
        transforms.RandomResizedCrop(224, scale=(0.2, 1.)),
        transforms.RandomApply([
            transforms.ColorJitter(0.4, 0.4, 0.4, 0.1)  # not strengthened
        ], p=0.8),
        transforms.RandomGrayscale(p=0.2),
        transforms.RandomApply([GaussianBlur([.1, 2.])], p=0.5),
        transforms.RandomHorizontalFlip(),
        Solarization(p=0.2),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.22
5])
    ]

    train_dataset = datasets.ImageFolder(traindir, TwoCropsTransform(transforms.Co
mpose(augmentation)))
    train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)
    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=per_batch
_size, num_workers=args.workers, drop_last=True,
                                               pin_memory=True, sampler=train_samp
ler)


    for epoch in range(args.start_epoch, args.epochs):
        train_sampler.set_epoch(epoch)
        adjust_learning_rate(optimizer, init_lr, epoch, args)

        # train for one epoch
        train(train_loader, model, optimizer, epoch, args)
        if args.rank == 0:
            save_model(args, args.checkpoint, epoch, model.module)

    if args.rank == 0:
        # save final model
        torch.save(model.module.online_encoder.state_dict(), args.checkpoint / 're
snet50.pth')

def train(train_loader, model, optimizer, epoch, args):
    batch_time = AverageMeter('Time', ':6.3f')
    data_time = AverageMeter('Data', ':6.3f')
    losses = AverageMeter('Loss', ':.4f')
    progress = ProgressMeter(
        len(train_loader),
        [batch_time, data_time, losses],
        prefix="Epoch: [{}]".format(epoch))

    print('=>  train start')
    # switch to train mode
    model.train()

    end = time.time()
    for i, (images, _) in enumerate(tqdm(train_loader)):
        # measure data loading time
        data_time.update(time.time() - end)

        images[0] = images[0].cuda(args.gpu, non_blocking=True)
        images[1] = images[1].cuda(args.gpu, non_blocking=True)

        # compute output and loss
        loss = model(images[0], images[1])

        losses.update(loss.item(), images[0].size(0))

        # compute gradient and do SGD step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        model.module.update_moving_average()
        '''BYOL'''
        # measure elapsed time
        batch_time.update(time.time() - end)
        end = time.time()

        if i % args.print_freq == 0 and args.gpu == 0:
            progress.display(i)



def adjust_learning_rate(optimizer, init_lr, epoch, args):
"""Decay the learning rate based on schedule"""
cur_lr = init_lr * 0.5 * (1. + math.cos(math.pi * epoch / args.epochs))
    for param_group in optimizer.param_groups:
        param_group['lr'] = cur_lr


def set_requires_grad(model, val):
    for p in model.parameters():
        p.requires_grad = val






class EMA():
    def __init__(self, beta):
        super().__init__()
        self.beta = beta

    def update_average(self, old, new):
        if old is None:
            return new
        return old * self.beta + (1 - self.beta) * new


def update_moving_average(ema_updater, ma_model, current_model):
    for current_params, ma_params in zip(current_model.parameters(), ma_model.para
meters()):
        old_weight, up_weight = ma_params.data, current_params.data
        ma_params.data = ema_updater.update_average(old_weight, up_weight)


class MoCo(nn.Module):
"""
    Build a MoCo model with: a query encoder, a key encoder, and a queue
    """
def __init__(self, base_encoder, dim=128, K=65536, m=0.999, T=0.07, mlp=True):
"""
        dim: feature dimension (default: 128)
        K: queue size; number of negative keys (default: 65536)
        m: moco momentum of updating key encoder (default: 0.999)
        T: softmax temperature (default: 0.07)
        """
super(MoCo, self).__init__()

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

        # create the encoders
        # num_classes is the output fc dimension
        self.encoder_q = torchvision.models.resnet50(num_classes=dim, zero_init_re
sidual=True)
        self.encoder_k = torchvision.models.resnet50(num_classes=dim, zero_init_re
sidual=True)

        if mlp:  # hack: brute-force replacement
            dim_mlp = self.encoder_q.fc.weight.shape[1]
            self.encoder_q.fc = nn.Sequential(nn.Linear(dim_mlp, dim_mlp), nn.ReLU
(), self.encoder_q.fc)
            self.encoder_k.fc = nn.Sequential(nn.Linear(dim_mlp, dim_mlp), nn.ReLU
(), self.encoder_k.fc)

        for param_q, param_k in zip(self.encoder_q.parameters(), self.encoder_k.pa
rameters()):
            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.parameters
()):
            param_k.data = param_k.data * self.m + param_q.data * (1. - self.m)

    @torch.no_grad()
    def _dequeue_and_enqueue(self, keys):
        # gather keys before updating queue
        keys = concat_all_gather(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   # dim x K
        ptr = (ptr + batch_size) % self.K  # move pointer

        self.queue_ptr[0] = ptr

    @torch.no_grad()
    def _batch_shuffle_ddp(self, x):
"""
        Batch shuffle, for making use of BatchNorm.
        *** Only support DistributedDataParallel (DDP) model. ***
        """
```

```
# gather from all gpus
        batch_size_this = x.shape[0]
        x_gather = concat_all_gather(x)
        batch_size_all = x_gather.shape[0]

        num_gpus = batch_size_all // batch_size_this

        # random shuffle index
        idx_shuffle = torch.randperm(batch_size_all).cuda()

        # broadcast to all gpus
        torch.distributed.broadcast(idx_shuffle, src=0)

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

        # shuffled index for this gpu
        gpu_idx = torch.distributed.get_rank()
        idx_this = idx_shuffle.view(num_gpus, -1)[gpu_idx]

        return x_gather[idx_this], idx_unshuffle

    @torch.no_grad()
    def _batch_unshuffle_ddp(self, x, idx_unshuffle):
"""
        Undo batch shuffle.
        *** Only support DistributedDataParallel (DDP) model. ***
        """
# gather from all gpus
        batch_size_this = x.shape[0]
        x_gather = concat_all_gather(x)
        batch_size_all = x_gather.shape[0]

        num_gpus = batch_size_all // batch_size_this

        # restored index for this gpu
        gpu_idx = torch.distributed.get_rank()
        idx_this = idx_unshuffle.view(num_gpus, -1)[gpu_idx]

        return x_gather[idx_this]

    def forward(self, im_q, im_k):
"""
        Input:
            im_q: a batch of query images
            im_k: a batch of key images
        Output:
            logits, targets
        """
# compute query features
        q = self.encoder_q(im_q)  # queries: NxC
        q = nn.functional.normalize(q, dim=1)

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

            # shuffle for making use of BN
            im_k, idx_unshuffle = self._batch_shuffle_ddp(im_k)

            k = self.encoder_k(im_k)  # keys: NxC
            k = nn.functional.normalize(k, dim=1)
            # undo shuffle
            k = self._batch_unshuffle_ddp(k, idx_unshuffle)

        # compute logits
        # 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()

        # dequeue and enqueue
        self._dequeue_and_enqueue(k)

        loss = F.cross_entropy(logits, labels, reduction='mean')

        return loss, labels


# utils
@torch.no_grad()
def concat_all_gather(tensor):
"""
    Performs all_gather operation on the provided tensors.
    *** Warning ***: torch.distributed.all_gather has no gradient.
    """
tensors_gather = [torch.ones_like(tensor)
        for _ in range(torch.distributed.get_world_size())]
    torch.distributed.all_gather(tensors_gather, tensor, async_op=False)

    output = torch.cat(tensors_gather, dim=0)
    return output



class GaussianBlur(object):
    def __init__(self, p):
        self.p = p

    def __call__(self, img):
        if random.random() < self.p:
            sigma = random.random() * 1.9 + 0.1
            return img.filter(ImageFilter.GaussianBlur(sigma))
        else:
            return img
```

```
class Solarization(object):
    def __init__(self, p):
        self.p = p

    def __call__(self, img):
        if random.random() < self.p:
            return ImageOps.solarize(img)
        else:
            return img


class TwoCropsTransform:
"""Take two random crops of one image as the query and key."""

def __init__(self, base_transform):
        self.base_transform = base_transform

    def __call__(self, x):
        q = self.base_transform(x)
        k = self.base_transform(x)
        return q, k

def save_model(args, checkpoint, epoch, model):
    torch.save(
        {
            'epoch': epoch,
            'args': args,
            'model': model.state_dict()
        },
        f=checkpoint+'/checkpoint.pth'
    )

class AverageMeter(object):
"""Computes and stores the average and current value"""
def __init__(self, name, fmt=':f'):
        self.name = name
        self.fmt = fmt
        self.reset()

    def reset(self):
        self.val = 0
        self.avg = 0
        self.sum = 0
        self.count = 0

    def update(self, val, n=1):
        self.val = val
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count

    def __str__(self):
        fmtstr = '{name} {val' + self.fmt + '} ({avg' + self.fmt + '})'
        return fmtstr.format(**self.__dict__)


class ProgressMeter(object):
    def __init__(self, num_batches, meters, prefix=""):
        self.batch_fmtstr = self._get_batch_fmtstr(num_batches)
        self.meters = meters
        self.prefix = prefix

    def display(self, batch):
        entries = [self.prefix + self.batch_fmtstr.format(batch)]
        entries += [str(meter) for meter in self.meters]
        print('\t'.join(entries))

    def _get_batch_fmtstr(self, num_batches):
        num_digits = len(str(num_batches // 1))
        fmt = '{:' + str(num_digits) + 'd}'
        return '[' + fmt + '/' + fmt.format(num_batches) + ']'


def accuracy(output, target, topk=(1,)):
"""Computes the accuracy over the k top predictions for the specified values of
 k"""
with torch.no_grad():
        maxk = max(topk)
        batch_size = target.size(0)

        _, pred = output.topk(maxk, 1, True, True)
        pred = pred.t()
        correct = pred.eq(target.view(1, -1).expand_as(pred))

        res = []
        for k in topk:
            correct_k = correct[:k].reshape(-1).float().sum(0, keepdim=True)
            res.append(correct_k.mul_(100.0 / batch_size))
        return res


if __name__ == '__main__':
    main()
```