reference (https://zhuanlan.zhihu.com/p/59205847)

本文代码基于PyTorch 1.0版本，需要用到以下包

```
import collections
import os
import shutil
import tqdm

import numpy as np
import PIL.Image
import torch
import torchvision
```

1. 基础配置

检查PyTorch版本

```
torch.__version__               # PyTorch version
torch.version.cuda              # Corresponding CUDA version
torch.backends.cudnn.version()  # Corresponding cuDNN version
torch.cuda.get_device_name(0)   # GPU type
```

更新PyTorch

PyTorch将被安装在anaconda3/lib/python3.7/site-packages/torch/目录下。

```
conda update pytorch torchvision -c pytorch
```

固定随机种子

```
torch.manual_seed(0)
torch.cuda.manual_seed_all(0)
```

指定程序运行在特定GPU卡上

在命令行指定环境变量

```
CUDA_VISIBLE_DEVICES=0,1 python train.py
```

或在代码中指定

```
os.environ['CUDA_VISIBLE_DEVICES'] = '0,1'
```

判断是否有CUDA支持

```
torch.cuda.is_available()
```

设置为cuDNN benchmark模式

Benchmark模式会提升计算速度，但是由于计算中有随机性，每次网络前馈结果略有差异。

```
torch.backends.cudnn.benchmark = True
```

如果想要避免这种结果波动，设置

```
torch.backends.cudnn.deterministic = True
```

清除GPU存储

有时Control-C中止运行后GPU存储没有及时释放，需要手动清空。在PyTorch内部可以

```
torch.cuda.empty_cache()
```

或在命令行可以先使用ps找到程序的PID，再使用kill结束该进程

```
ps aux | grep python
kill -9 [pid]
```

或者直接重置没有被清空的GPU

```
nvidia-smi --gpu-reset -i [gpu_id]
```

2. 张量处理

张量基本信息

```
tensor.type()   # Data type
tensor.size()   # Shape of the tensor. It is a subclass of Python tuple
tensor.dim()    # Number of dimensions.
```

数据类型转换

```
# Set default tensor type. Float in PyTorch is much faster than double.
torch.set_default_tensor_type(torch.FloatTensor)

# Type convertions.
tensor = tensor.cuda()
tensor = tensor.cpu()
tensor = tensor.float()
tensor = tensor.long()
```

torch.Tensor与np.ndarray转换

```
# torch.Tensor -> np.ndarray.
ndarray = tensor.cpu().numpy()

# np.ndarray -> torch.Tensor.
tensor = torch.from_numpy(ndarray).float()
tensor = torch.from_numpy(ndarray.copy()).float()  # If ndarray has negative stride
```

torch.Tensor与PIL.Image转换

PyTorch中的张量默认采用N×D×H×W的顺序，并且数据范围在[0, 1]，需要进行转置和规范化。

```
# torch.Tensor -> PIL.Image.
image = PIL.Image.fromarray(torch.clamp(tensor * 255, min=0, max=255
    ).byte().permute(1, 2, 0).cpu().numpy())
image = torchvision.transforms.functional.to_pil_image(tensor)  # Equivalently way

# PIL.Image -> torch.Tensor.
tensor = torch.from_numpy(np.asarray(PIL.Image.open(path))
    ).permute(2, 0, 1).float() / 255
tensor = torchvision.transforms.functional.to_tensor(PIL.Image.open(path))  # Equivalently way
```

np.ndarray与PIL.Image转换

```
# np.ndarray -> PIL.Image.
image = PIL.Image.fromarray(ndarray.astypde(np.uint8))

# PIL.Image -> np.ndarray.
ndarray = np.asarray(PIL.Image.open(path))
```

从只包含一个元素的张量中提取值

这在训练时统计loss的变化过程中特别有用。否则这将累积计算图，使GPU存储占用量越来越大。

```
value = tensor.item()
```

张量形变

张量形变常常需要用于将卷积层特征输入全连接层的情形。相比torch.view，torch.reshape可以自动处理输入张量不连续的情况。

```
tensor = torch.reshape(tensor, shape)
```

打乱顺序

```
tensor = tensor[torch.randperm(tensor.size(0))]  # Shuffle the first dimension
```

水平翻转

PyTorch不支持tensor[::-1]这样的负步长操作，水平翻转可以用张量索引实现。

```
# Assume tensor has shape N*D*H*W.
tensor = tensor[:, :, :, torch.arange(tensor.size(3) - 1, -1, -1).long()]
```

复制张量

有三种复制的方式，对应不同的需求。

```
# Operation                 |  New/Shared memory | Still in computation graph |
tensor.clone()            # |        New         |          Yes               |
tensor.detach()           # |      Shared        |          No                |
tensor.detach.clone()()   # |        New         |          No                |
```

拼接张量

注意torch.cat和torch.stack的区别在于torch.cat沿着给定的维度拼接，而torch.stack会新增一维。例如当参数是3个10×5的张量，torch.cat的结果是30×5的张量，而torch.stack的结果是3×10×5的张量。

```
tensor = torch.cat(list_of_tensors, dim=0)
tensor = torch.stack(list_of_tensors, dim=0)
```

将整数标记转换成独热（one-hot）编码

PyTorch中的标记默认从0开始。

```
N = tensor.size(0)
one_hot = torch.zeros(N, num_classes).long()
one_hot.scatter_(dim=1, index=torch.unsqueeze(tensor, dim=1), src=torch.ones(N, num_classes).lor
```

得到非零/零元素

```
torch.nonzero(tensor)               # Index of non-zero elements
torch.nonzero(tensor == 0)          # Index of zero elements
torch.nonzero(tensor).size(0)       # Number of non-zero elements
torch.nonzero(tensor == 0).size(0)  # Number of zero elements
```

判断两个张量相等

```
torch.allclose(tensor1, tensor2)  # float tensor
torch.equal(tensor1, tensor2)     # int tensor
```

张量扩展

```
# Expand tensor of shape 64*512 to shape 64*512*7*7.
torch.reshape(tensor, (64, 512, 1, 1)).expand(64, 512, 7, 7)
```

矩阵乘法

```
# Matrix multiplication: (m*n) * (n*p) -> (m*p).
result = torch.mm(tensor1, tensor2)

# Batch matrix multiplication: (b*m*n) * (b*n*p) -> (b*m*p).
result = torch.bmm(tensor1, tensor2)

# Element-wise multiplication.
result = tensor1 * tensor2
```

计算两组数据之间的两两欧式距离

```
# X1 is of shape m*d.
X1 = torch.unsqueeze(X1, dim=1).expand(m, n, d)
# X2 is of shape n*d.
X2 = torch.unsqueeze(X2, dim=0).expand(m, n, d)
# dist is of shape m*n, where dist[i][j] = sqrt(|X1[i, :] - X[j, :]|^2)
dist = torch.sqrt(torch.sum((X1 - X2) ** 2, dim=2))
```

3. 模型定义

卷积层

最常用的卷积层配置是

```
conv = torch.nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=True)
conv = torch.nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, padding=0, bias=True)
```

如果卷积层配置比较复杂，不方便计算输出大小时，可以利用如下可视化工具辅助
(https://ezyang.github.io/convolution-visualizer/index.html)

GAP（Global average pooling）层

```
gap = torch.nn.AdaptiveAvgPool2d(output_size=1)
```

双线性汇合（bilinear pooling）[1]

```
X = torch.reshape(N, D, H * W)                        # Assume X has shape N*D*H*W
X = torch.bmm(X, torch.transpose(X, 1, 2)) / (H * W)  # Bilinear pooling
assert X.size() == (N, D, D)
X = torch.reshape(X, (N, D * D))
X = torch.sign(X) * torch.sqrt(torch.abs(X) + 1e-5)   # Signed-sqrt normalization
X = torch.nn.functional.normalize(X)                  # L2 normalization
```

多卡同步BN（Batch normalization）

当使用torch.nn.DataParallel将代码运行在多张GPU卡上时，PyTorch的BN层默认操作是各卡上数据独立地计算均值和标准差，同步BN使用所有卡上的数据一起计算BN层的均值和标准差，缓解了当批量大小（batch size）比较小时对均值和标准差估计不准的情况，是在目标检测等任务中一个有效的提升性能的技巧。
(https://github.com/vacancy/Synchronized-BatchNorm-PyTorch)

现在PyTorch官方已经支持同步BN操作

```
sync_bn = torch.nn.SyncBatchNorm(num_features, eps=1e-05, momentum=0.1, affine=True,
                                 track_running_stats=True)
```

将已有网络的所有BN层改为同步BN层

```
def convertBNtoSyncBN(module, process_group=None):
    '''Recursively replace all BN layers to SyncBN layer.

    Args:
        module[torch.nn.Module]. Network
    '''
    if isinstance(module, torch.nn.modules.batchnorm._BatchNorm):
        sync_bn = torch.nn.SyncBatchNorm(module.num_features, module.eps, module.momentum,
                                         module.affine, module.track_running_stats, process_grouŗ
        sync_bn.running_mean = module.running_mean
        sync_bn.running_var = module.running_var
        if module.affine:
            sync_bn.weight = module.weight.clone().detach()
            sync_bn.bias = module.bias.clone().detach()
        return sync_bn
    else:
        for name, child_module in module.named_children():
            setattr(module, name) = convert_syncbn_model(child_module, process_group=process_grc
        return module
```

类似BN滑动平均

如果要实现类似BN滑动平均的操作，在forward函数中要使用原地（inplace）操作给滑动平均赋值。

```
class BN(torch.nn.Module)
    def __init__(self):
        ...
        self.register_buffer('running_mean', torch.zeros(num_features))

    def forward(self, X):
        ...
        self.running_mean += momentum * (current - self.running_mean)
```

计算模型整体参数量

```
num_parameters = sum(torch.numel(parameter) for parameter in model.parameters())
```

类似Keras的model.summary()输出模型信息

(https://github.com/sksq96/pytorch-summary)

模型权值初始化

注意model.modules()和model.children()的区别：model.modules()会迭代地遍历模型的所有子层，而model.children()只会遍历模型下的一层。

```
# Common practise for initialization.
for layer in model.modules():
    if isinstance(layer, torch.nn.Conv2d):
        torch.nn.init.kaiming_normal_(layer.weight, mode='fan_out',
                                      nonlinearity='relu')
        if layer.bias is not None:
            torch.nn.init.constant_(layer.bias, val=0.0)
    elif isinstance(layer, torch.nn.BatchNorm2d):
        torch.nn.init.constant_(layer.weight, val=1.0)
        torch.nn.init.constant_(layer.bias, val=0.0)
    elif isinstance(layer, torch.nn.Linear):
        torch.nn.init.xavier_normal_(layer.weight)
        if layer.bias is not None:
            torch.nn.init.constant_(layer.bias, val=0.0)

# Initialization with given tensor.
layer.weight = torch.nn.Parameter(tensor)
```

部分层使用预训练模型

注意如果保存的模型是torch.nn.DataParallel，则当前的模型也需要是torch.nn.DataParallel。torch.nn.DataParallel(model).module == model。

```
model.load_state_dict(torch.load('model,pth'), strict=False)
```

将在GPU保存的模型加载到CPU

```
model.load_state_dict(torch.load('model,pth', map_location='cpu'))
```

4. 数据准备、特征提取与微调

图像分块打散（image shuffle）/区域混淆机制（region confusion mechanism，RCM）[2]

```
# X is torch.Tensor of size N*D*H*W.
# Shuffle rows
Q = (torch.unsqueeze(torch.arange(num_blocks), dim=1) * torch.ones(1, num_blocks).long()
    + torch.randint(low=-neighbour, high=neighbour, size=(num_blocks, num_blocks)))
Q = torch.argsort(Q, dim=0)
assert Q.size() == (num_blocks, num_blocks)

X = [torch.chunk(row, chunks=num_blocks, dim=2)
    for row in torch.chunk(X, chunks=num_blocks, dim=1)]
X = [[X[Q[i, j].item()][j] for j in range(num_blocks)]
    for i in range(num_blocks)]

# Shulle columns.
Q = (torch.ones(num_blocks, 1).long() * torch.unsqueeze(torch.arange(num_blocks), dim=0)
    + torch.randint(low=-neighbour, high=neighbour, size=(num_blocks, num_blocks)))
Q = torch.argsort(Q, dim=1)
assert Q.size() == (num_blocks, num_blocks)
X = [[X[i][Q[i, j].item()] for j in range(num_blocks)]
    for i in range(num_blocks)]

Y = torch.cat([torch.cat(row, dim=2) for row in X], dim=1)
```

得到视频数据基本信息

```
import cv2
video = cv2.VideoCapture(mp4_path)
height = int(video.get(cv2.CAP_PROP_FRAME_HEIGHT))
width = int(video.get(cv2.CAP_PROP_FRAME_WIDTH))
num_frames = int(video.get(cv2.CAP_PROP_FRAME_COUNT))
fps = int(video.get(cv2.CAP_PROP_FPS))
video.release()
```

TSN每段（segment）采样一帧视频[3]

```
K = self._num_segments
if is_train:
    if num_frames > K:
        # Random index for each segment.
        frame_indices = torch.randint(
            high=num_frames // K, size=(K,), dtype=torch.long)
        frame_indices += num_frames // K * torch.arange(K)
    else:
        frame_indices = torch.randint(
            high=num_frames, size=(K - num_frames,), dtype=torch.long)
        frame_indices = torch.sort(torch.cat((
            torch.arange(num_frames), frame_indices)))[0]
else:
    if num_frames > K:
        # Middle index for each segment.
        frame_indices = num_frames / K // 2
        frame_indices += num_frames // K * torch.arange(K)
    else:
        frame_indices = torch.sort(torch.cat((
            torch.arange(num_frames), torch.arange(K - num_frames))))[0]
assert frame_indices.size() == (K,)
return [frame_indices[i] for i in range(K)]
```

提取ImageNet预训练模型某层的卷积特征

```
# VGG-16 relu5-3 feature.
model = torchvision.models.vgg16(pretrained=True).features[:-1]
# VGG-16 pool5 feature.
model = torchvision.models.vgg16(pretrained=True).features
# VGG-16 fc7 feature.
model = torchvision.models.vgg16(pretrained=True)
model.classifier = torch.nn.Sequential(*list(model.classifier.children())[:-3])
# ResNet GAP feature.
model = torchvision.models.resnet18(pretrained=True)
model = torch.nn.Sequential(collections.OrderedDict(
    list(model.named_children())[:-1]))

with torch.no_grad():
    model.eval()
    conv_representation = model(image)
```

提取ImageNet预训练模型多层的卷积特征

```
class FeatureExtractor(torch.nn.Module):
    """Helper class to extract several convolution features from the given
    pre-trained model.

    Attributes:
        _model, torch.nn.Module.
        _layers_to_extract, list<str> or set<str>

    Example:
        >>> model = torchvision.models.resnet152(pretrained=True)
        >>> model = torch.nn.Sequential(collections.OrderedDict(
                list(model.named_children())[:-1]))
        >>> conv_representation = FeatureExtractor(
                pretrained_model=model,
                layers_to_extract={'layer1', 'layer2', 'layer3', 'layer4'})(image)
    """
    def __init__(self, pretrained_model, layers_to_extract):
        torch.nn.Module.__init__(self)
        self._model = pretrained_model
        self._model.eval()
        self._layers_to_extract = set(layers_to_extract)

    def forward(self, x):
        with torch.no_grad():
            conv_representation = []
            for name, layer in self._model.named_children():
                x = layer(x)
                if name in self._layers_to_extract:
                    conv_representation.append(x)
            return conv_representation
```

其他预训练模型

(https://github.com/Cadene/pretrained-models.pytorch)

微调全连接层

```
model = torchvision.models.resnet18(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
model.fc = nn.Linear(512, 100)  # Replace the last fc layer
optimizer = torch.optim.SGD(model.fc.parameters(), lr=1e-2, momentum=0.9, weight_decay=1e-4)
```

以较大学习率微调全连接层，较小学习率微调卷积层

```
model = torchvision.models.resnet18(pretrained=True)
finetuned_parameters = list(map(id, model.fc.parameters()))
conv_parameters = (p for p in model.parameters() if id(p) not in finetuned_parameters)
parameters = [{'params': conv_parameters, 'lr': 1e-3},
              {'params': model.fc.parameters()}]
optimizer = torch.optim.SGD(parameters, lr=1e-2, momentum=0.9, weight_decay=1e-4)
```

5. 模型训练

常用训练和验证数据预处理

其中ToTensor操作会将PIL.Image或形状为H×W×D，数值范围为[0, 255]的np.ndarray转换为形状为D×H×W，数值范围为[0.0, 1.0]的torch.Tensor。

```
train_transform = torchvision.transforms.Compose([
    torchvision.transforms.RandomResizedCrop(size=224,
                                             scale=(0.08, 1.0)),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean=(0.485, 0.456, 0.406),
                                     std=(0.229, 0.224, 0.225)),
])
val_transform = torchvision.transforms.Compose([
    torchvision.transforms.Resize(256),
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean=(0.485, 0.456, 0.406),
                                     std=(0.229, 0.224, 0.225)),
])
```

训练基本代码框架

```
for t in epoch(80):
    for images, labels in tqdm.tqdm(train_loader, desc='Epoch %3d' % (t + 1)):
        images, labels = images.cuda(), labels.cuda()
        scores = model(images)
        loss = loss_function(scores, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

标记平滑（label smoothing）[4]

```
for images, labels in train_loader:
    images, labels = images.cuda(), labels.cuda()
    N = labels.size(0)
    # C is the number of classes.
    smoothed_labels = torch.full(size=(N, C), fill_value=0.1 / (C - 1)).cuda()
    smoothed_labels.scatter_(dim=1, index=torch.unsqueeze(labels, dim=1), value=0.9)

    score = model(images)
    log_prob = torch.nn.functional.log_softmax(score, dim=1)
    loss = -torch.sum(log_prob * smoothed_labels) / N
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Mixup[5]

```
beta_distribution = torch.distributions.beta.Beta(alpha, alpha)
for images, labels in train_loader:
    images, labels = images.cuda(), labels.cuda()

    # Mixup images.
    lambda_ = beta_distribution.sample([]).item()
    index = torch.randperm(images.size(0)).cuda()
    mixed_images = lambda_ * images + (1 - lambda_) * images[index, :]

    # Mixup loss.
    scores = model(mixed_images)
    loss = (lambda_ * loss_function(scores, labels)
            + (1 - lambda_) * loss_function(scores, labels[index]))

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

L1正则化

```
l1_regularization = torch.nn.L1Loss(reduction='sum')
loss = ...  # Standard cross-entropy loss
for param in model.parameters():
    loss += lambda_ * torch.sum(torch.abs(param))
loss.backward()
```

不对偏置项进行L2正则化/权值衰减（weight decay）

```
bias_list = (param for name, param in model.named_parameters() if name[-4:] == 'bias')
others_list = (param for name, param in model.named_parameters() if name[-4:] != 'bias')
parameters = [{'parameters': bias_list, 'weight_decay': 0},
            {'parameters': others_list}]
optimizer = torch.optim.SGD(parameters, lr=1e-2, momentum=0.9, weight_decay=1e-4)
```

梯度裁剪（gradient clipping）

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=20)
```

计算Softmax输出的准确率

```
score = model(images)
prediction = torch.argmax(score, dim=1)
num_correct = torch.sum(prediction == labels).item()
accuruacy = num_correct / labels.size(0)
```

可视化模型前馈的计算图

(https://github.com/szagoruyko/pytorchviz)

可视化学习曲线

有Facebook自己开发的Visdom和Tensorboard（仍处于实验阶段）两个选择。
(https://github.com/facebookresearch/visdom)
(https://pytorch.org/docs/stable/tensorboard.html)

```
# Example using Visdom.
vis = visdom.Visdom(env='Learning curve', use_incoming_socket=False)
assert self._visdom.check_connection()
self._visdom.close()
options = collections.namedtuple('Options', ['loss', 'acc', 'lr'])(
    loss={'xlabel': 'Epoch', 'ylabel': 'Loss', 'showlegend': True},
    acc={'xlabel': 'Epoch', 'ylabel': 'Accuracy', 'showlegend': True},
    lr={'xlabel': 'Epoch', 'ylabel': 'Learning rate', 'showlegend': True})

for t in epoch(80):
    tran(...)
    val(...)
    vis.line(X=torch.Tensor([t + 1]), Y=torch.Tensor([train_loss]),
             name='train', win='Loss', update='append', opts=options.loss)
    vis.line(X=torch.Tensor([t + 1]), Y=torch.Tensor([val_loss]),
             name='val', win='Loss', update='append', opts=options.loss)
    vis.line(X=torch.Tensor([t + 1]), Y=torch.Tensor([train_acc]),
             name='train', win='Accuracy', update='append', opts=options.acc)
    vis.line(X=torch.Tensor([t + 1]), Y=torch.Tensor([val_acc]),
             name='val', win='Accuracy', update='append', opts=options.acc)
    vis.line(X=torch.Tensor([t + 1]), Y=torch.Tensor([lr]),
             win='Learning rate', update='append', opts=options.lr)
```

得到当前学习率

```
# If there is one global learning rate (which is the common case).
lr = next(iter(optimizer.param_groups))['lr']

# If there are multiple learning rates for different layers.
all_lr = []
for param_group in optimizer.param_groups:
    all_lr.append(param_group['lr'])
```

学习率衰减

```
# Reduce learning rate when validation accuarcy plateau.
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max', patience=5, verbos
for t in range(0, 80):
    train(...); val(...)
    scheduler.step(val_acc)

# Cosine annealing learning rate.
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=80)
# Reduce learning rate by 10 at given epochs.
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[50, 70], gamma=0.1)
for t in range(0, 80):
    scheduler.step()
    train(...); val(...)

# Learning rate warmup by 10 epochs.
scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda=lambda t: t / 10)
for t in range(0, 10):
    scheduler.step()
    train(...); val(...)
```

保存与加载断点

注意为了能够恢复训练，我们需要同时保存模型和优化器的状态，以及当前的训练轮数。

```
# Save checkpoint.
is_best = current_acc > best_acc
best_acc = max(best_acc, current_acc)
checkpoint = {
    'best_acc': best_acc,
    'epoch': t + 1,
    'model': model.state_dict(),
    'optimizer': optimizer.state_dict(),
}
model_path = os.path.join('model', 'checkpoint.pth.tar')
torch.save(checkpoint, model_path)
if is_best:
    shutil.copy('checkpoint.pth.tar', model_path)

# Load checkpoint.
if resume:
    model_path = os.path.join('model', 'checkpoint.pth.tar')
    assert os.path.isfile(model_path)
    checkpoint = torch.load(model_path)
    best_acc = checkpoint['best_acc']
    start_epoch = checkpoint['epoch']
    model.load_state_dict(checkpoint['model'])
    optimizer.load_state_dict(checkpoint['optimizer'])
    print('Load checkpoint at epoch %d.' % start_epoch)
```

计算准确率、查准率（**precision**）、查全率（**recall**）

```
# data['label'] and data['prediction'] are groundtruth label and prediction
# for each image, respectively.
accuracy = np.mean(data['label'] == data['prediction']) * 100

# Compute recision and recall for each class.
for c in range(len(num_classes)):
    tp = np.dot((data['label'] == c).astype(int),
                (data['prediction'] == c).astype(int))
    tp_fp = np.sum(data['prediction'] == c)
    tp_fn = np.sum(data['label'] == c)
    precision = tp / tp_fp * 100
    recall = tp / tp_fn * 100
```

6. 模型测试

计算每个类别的查准率（**precision**）、查全率（**recall**）、**F1**和总体指标

```python
import sklearn.metrics

all_label = []
all_prediction = []
for images, labels in tqdm.tqdm(data_loader):
    # Data.
    images, labels = images.cuda(), labels.cuda()

    # Forward pass.
    score = model(images)

    # Save label and predictions.
    prediction = torch.argmax(score, dim=1)
    all_label.append(labels.cpu().numpy())
    all_prediction.append(prediction.cpu().numpy())

# Compute RP and confusion matrix.
all_label = np.concatenate(all_label)
assert len(all_label.shape) == 1
all_prediction = np.concatenate(all_prediction)
assert all_label.shape == all_prediction.shape
micro_p, micro_r, micro_f1, _ = sklearn.metrics.precision_recall_fscore_support(
    all_label, all_prediction, average='micro', labels=range(num_classes))
class_p, class_r, class_f1, class_occurence = sklearn.metrics.precision_recall_fscore_support(
    all_label, all_prediction, average=None, labels=range(num_classes))
# Ci,j = #{y=i and hat_y=j}
confusion_mat = sklearn.metrics.confusion_matrix(
    all_label, all_prediction, labels=range(num_classes))
assert confusion_mat.shape == (num_classes, num_classes)
```

将各类结果写入电子表格

```
import csv

# Write results onto disk.
with open(os.path.join(path, filename), 'wt', encoding='utf-8') as f:
    f = csv.writer(f)
    f.writerow(['Class', 'Label', '# occurence', 'Precision', 'Recall', 'F1',
                'Confused class 1', 'Confused class 2', 'Confused class 3',
                'Confused 4', 'Confused class 5'])
    for c in range(num_classes):
        index = np.argsort(confusion_mat[:, c])[::-1][:5]
        f.writerow([
            label2class[c], c, class_occurence[c], '%4.3f' % class_p[c],
                '%4.3f' % class_r[c], '%4.3f' % class_f1[c],
                '%s:%d' % (label2class[index[0]], confusion_mat[index[0], c]),
                '%s:%d' % (label2class[index[1]], confusion_mat[index[1], c]),
                '%s:%d' % (label2class[index[2]], confusion_mat[index[2], c]),
                '%s:%d' % (label2class[index[3]], confusion_mat[index[3], c]),
                '%s:%d' % (label2class[index[4]], confusion_mat[index[4], c])])
        f.writerow(['All', '', np.sum(class_occurence), micro_p, micro_r, micro_f1,
                    '', '', '', '', ''])
```

7. PyTorch其他注意事项

模型定义

建议有参数的层和汇合（pooling）层使用torch.nn模块定义，激活函数直接使用torch.nn.functional。torch.nn模块和torch.nn.functional的区别在于，torch.nn模块在计算时底层调用了torch.nn.functional，但torch.nn模块包括该层参数，还可以应对训练和测试两种网络状态。使用torch.nn.functional时要注意网络状态，如

```
def forward(self, x):
    ...
    x = torch.nn.functional.dropout(x, p=0.5, training=self.training)
```

model(x)前用model.train()和model.eval()切换网络状态。

不需要计算梯度的代码块用with torch.no_grad()包含起来。model.eval()和torch.no_grad()的区别在于，model.eval()是将网络切换为测试状态，例如BN和随机失活（dropout）在训练和测试阶段使用不同的计算方法。torch.no_grad()是关闭PyTorch张量的自动求导机制，以减少存储使用和加速计算，得到的结果无法进行loss.backward()。

torch.nn.CrossEntropyLoss的输入不需要经过Softmax。torch.nn.CrossEntropyLoss等价于torch.nn.functional.log_softmax + torch.nn.NLLLoss。

loss.backward()前用optimizer.zero_grad()清除累积梯度。optimizer.zero_grad()和model.zero_grad()效果一样。

PyTorch性能与调试

torch.utils.data.DataLoader中尽量设置pin_memory=True，对特别小的数据集如MNIST设置pin_memory=False反而更快一些。num_workers的设置需要在实验中找到最快的取值。
用del及时删除不用的中间变量，节约GPU存储。
使用inplace操作可节约GPU存储，如

```
x = torch.nn.functional.relu(x, inplace=True)
```

此外，还可以通过torch.utils.checkpoint前向传播时只保留一部分中间结果来节约GPU存储使用，在反向传播时需要的内容从最近中间结果中计算得到。

减少CPU和GPU之间的数据传输。例如如果你想知道一个epoch中每个mini-batch的loss和准确率，先将它们累积在GPU中等一个epoch结束之后一起传输回CPU会比每个mini-batch都进行一次GPU到CPU的传输更快。使用半精度浮点数half()会有一定的速度提升，具体效率依赖于GPU型号。需要小心数值精度过低带来的稳定性问题。时常使用assert tensor.size() == (N, D, H, W)作为调试手段，确保张量维度和你设想中一致。除了标记y外，尽量少使用一维张量，使用n*1的二维张量代替，可以避免一些意想不到的一维张量计算结果。统计代码各部分耗时

```
with torch.autograd.profiler.profile(enabled=True, use_cuda=False) as profile:
    ...
print(profile)
```

或者在命令行运行

```
python -m torch.utils.bottleneck main.py
```