Q&A

Q1：训练时有哪些可以调节的超参数？

①BATCH_SIZE;
②Learning_Rate;
③EPOCH_SIZE;
④优化器；

Q2：Pytorch 中有哪些优化器，可用于调节的参数有哪些？

每次反向传播都会给模型各个可学习参数 params 计算出一个偏导数 g_t，用于更新对应的参数 p。通常偏导数 g_t 不会直接作用到对应的可学习参数 p 上，而是通过优化器做一下处理，得到一个新的值 $\hat{g}_t$，处理过程用函数 F 表示（不同的优化器对应的 F 的内容不同），即 $\hat{g}_t = F(g_t)$，然后和学习率 lr 一起用于更新可学习参数 p，即 $p = p - \hat{g}_t \times lr$。

(1)SGD 算法

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
μ (momentum), τ (dampening), *nesterov*, *maximize*

for $t = 1$ **to** ... **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  if $\mu \neq 0$
    if $t > 1$
      $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + (1-\tau)g_t$
    else
      $\mathbf{b}_t \leftarrow g_t$
    if *nesterov*
      $g_t \leftarrow g_t + \mu\mathbf{b}_t$
    else
      $g_t \leftarrow \mathbf{b}_t$
  if *maximize*
    $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$

return θ_t

torch.optim.SGD(params,
lr=<required parameter>,
momentum=0,
dampening=0,

```
        weight_decay=0,
        nesterov=False,
        maximize=False,
        foreach=None,
        differentiable=False
        )
```

参数解释

params：模型里需要被更新的可学习的参数。

lr：学习率。

momentum：动量，通过引入一个新的变量 b 去积累之前的梯度，加速学习过程。

上一次得到的b_{t-1}和当前得到的偏导数g_t做(1)式的运算修改偏导数g_t。

$$b_t = momentum * b_{t-1} + (1-\tau)g_t \tag{1}$$

然后通过(2)式更新参数：

$$p_t = p_{t-1} - b_t \times lr \tag{2}$$

dempening：乘到偏导数g_t上的一个数，即(1)式中的τ。该参数在优化器第一次更新时不起作用。

weight decay：用当前可学习参数 p 的值修改偏导数g_t，即：

$$g_t = g_t + p * \text{weight\_decay} \tag{3}$$

再通过(1)式计算得到b_t

(2)RMSprop 算法

input : α (alpha), γ (lr), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), μ (momentum), *centered*
initialize : $v_0 \leftarrow 0$ (square average), $\mathbf{b}_0 \leftarrow 0$ (buffer), $g_0^{ave} \leftarrow 0$

for $t = 1$ **to** ... **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $v_t \leftarrow \alpha v_{t-1} + (1-\alpha)g_t^2$
  $\tilde{v}_t \leftarrow v_t$
  if centered
    $g_t^{ave} \leftarrow g_{t-1}^{ave}\alpha + (1-\alpha)g_t$
    $\tilde{v}_t \leftarrow \tilde{v}_t - \left(g_t^{ave}\right)^2$
  if $\mu > 0$
    $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + g_t/\left(\sqrt{\tilde{v}_t} + \epsilon\right)$
    $\theta_t \leftarrow \theta_{t-1} - \gamma\mathbf{b}_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t/\left(\sqrt{\tilde{v}_t} + \epsilon\right)$

return θ_t

假设当前的损失函数为：

$$Loss = x^2 + 10y^2 \tag{4}$$

通过求导可得 x 和 y 方向的导数：

$$g_x = 2x \tag{5}$$

$$g_y = 20y \tag{6}$$

显然，当 x=0，y=0 时(4)式最小。假设当前初始值 x=40，y=20，则 $g_x = 80$，$g_y = 400$，可见在 x 方向的移动距离比 y 方向的移动距离小，但是实际 x 离最小点更远。

RMSprop 算法有效解决了这个问题。通过累计各个变量的梯度的平方 *v*，然后用每个变量的梯度除以 *v*，即可有效缓解变量间的梯度差异。

```
torch.optim.RMSprop(params,
                    lr=0.01,
                    alpha=0.99,
                    eps=1e-08,
                    weight_decay=0,
                    momentum=0,
                    centered=False,
                    foreach=None,
                    maximize=False,
                    differentiable=False
                    )
```

参数解释

params：模型里需要被更新的可学习的参数。

lr：学习率。

alpha：平滑常数。将梯度的平方进行平滑处理：

$$v_t = \alpha * v_{t-1} + (1-\alpha) g_t^2$$

eps：加在分母上防止除零

weight_decay：用当前可学习参数 p 的值修改偏导数 g_t。参看式(3)。

momentum：动量。

centered：

当为 False 时，分母为 $\sqrt{v} + \varepsilon$；

当为 True 时，分母的计算过程变为：

$$\bar{g}_t = \alpha \bar{g}_{t-1} + (1-\alpha) g_t \tag{7}$$

$$v_t = v_t - (\bar{g}_t)^2 \tag{8}$$

(3)Adagrad 算法

在一般的优化算法中，目标函数自变量的每一个元素在相同迭代次数中都使用同一个学习率来自我迭代。当每个元素的梯度值有较大差别时，需要选择足够小的学习率使得自变量在梯度值较大的维度上不发散，但这样会导致自变量在梯度值较小的维度上迭代过慢。

Adagrad 算法可以自适应的给所有的参数分配学习率，根据自变量在每个维度的梯度值的大小来调整各个维度上的学习率，从而避免统一的学习率难以适应所有维度的问题。

```
torch.optim.Adagrad(params,
                    lr=0.01,
                    lr_decay=0,
                    weight_decay=0,
                    initial_accumulator_value=0,
                    eps=1e-10,
                    foreach=None,
                    maximize=False
                    )
```

---

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
τ (initial accumulator value), η (lr decay)

initialize : $state\_sum_0 \leftarrow 0$

---

for $t = 1$ **to** ... **do**

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$\tilde{\gamma} \leftarrow \gamma/(1 + (t-1)\eta)$

if $\lambda \neq 0$

$g_t \leftarrow g_t + \lambda\theta_{t-1}$

$state\_sum_t \leftarrow state\_sum_{t-1} + g_t^2$

$\theta_t \leftarrow \theta_{t-1} - \tilde{\gamma}\dfrac{g_t}{\sqrt{state\_sum_t} + \epsilon}$

---

return θ_t

---

(4)Adam 算法

在 RMSProp 的基础上，做两个改进：梯度滑动平均和偏差纠正。

```
torch.optim.Adam(params,
                 lr=0.001,
                 betas=(0.9, 0.999),
                 eps=1e-08,
                 weight_decay=0,
                 amsgrad=False,
                 foreach=None,
                 maximize=False,
                 capturable=False,
                 differentiable=False,
                 fused=False
                 )
```

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

for $t = 1$ **to** ... **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \left(1-\beta_1^t\right)$
  $\widehat{v_t} \leftarrow v_t / \left(1-\beta_2^t\right)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t} / \left(\sqrt{\widehat{v_t}^{max}} + \epsilon\right)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \epsilon\right)$

return θ_t

先计算所有可学习参数的梯度 g_t，将梯度进行累计得到 m_t：

$$m_t = \beta_1 * m_{t-1} + (1-\beta_1) * g_t \tag{9}$$

然后累计梯度的平方：

$$v_t = \beta_2 * v_{t-1} + (1-\beta_2) * g_t^2 \tag{10}$$

计算偏差纠正 m 和 v：

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{11}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{12}$$

最后通过 $\theta_t = \theta_{t-1} - \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} * lr$ 更新参数

Q3：预训练模型的作用？

加速模型的收敛速度；使模型的效果更好。

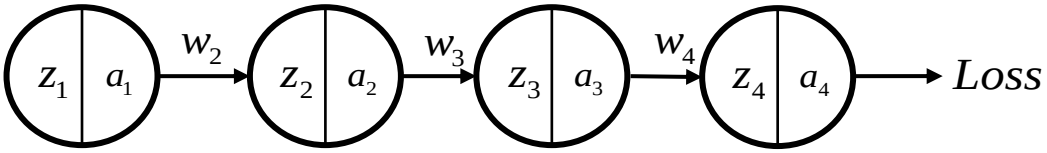
Q4：训练时，影响显存占用的因素有哪些？可以从哪些方面解决？

GPU 的内存占用率主要是模型的大小，包括网络的宽度，深度，参数量，中

间每一层的缓存，都会在内存中开辟空间来进行保存，所以模型本身会占用很大一部分内存。

其次是 batch_size 的大小。模型的加载很难控制，一般调节 batch_size 来控制显存的占用。

Q5：有哪几种权重初始化的方式？



参数传递过程示意图

其中 $a_j=\sigma(z_j)$，$\sigma(\cdot)$ 为激活函数，$z_j=w_j\cdot a_{j-1}+b_j$ 为神经元的输入。

损失函数关于第一个神经元的梯度为：

$$\frac{\partial L}{\partial b_1}=\frac{\partial L}{\partial a_4}\times\sigma^{'}(z_4)\times w_4\times\sigma^{'}(z_3)\times w_3\times\sigma^{'}(z_2)\times w_2\times\sigma^{'}(z_1) \tag{13}$$

通过式(13)可以看出梯度和权重矩阵和激活函数相关，为了防止出现梯度消失和梯度爆炸现象，要对权重的方差加以控制。

有 Xavier 初始化（适合适用于 tanh 和 sigmoid 激活函数）、He 初始化（适用于 ReLU 激活函数）、RandomNormal 和 RandomUniform 初始化

Q6：数据读取 Dataset 的操作流程是怎样的？

P.S.以 num_woker=0 时为例

DataLoader 进入迭代器，启用_SigleProcessDataLoaderIter 一次迭代一个 batch_size 大小的数据。其过程是：

通过 BathSampler()的 sampler 随机产生索引值然后再迭代，当迭代够一个 batch_size 大小后返回一次索引值，然后 DataSet 的_getitem_获取返回的索引值，得到图片数据和类别标签(data, label)，再通过_MapDataset Fetcher 得到 batch_size 哥图像信息（其中每个信息为 tuple 类型，batch_size 个 tuple 组合为一个 list 类型的数据），一个 batch 的数据通过 collate_fn 进行数据打包为 Tensor 类型(batch, channel, height, width)

Q7：使用 Pytorch 搭建时卷积层、池化层、全连接层等算子有什么参数可以调节？

①卷积层

kernel_size（卷积核大小）、stride（卷积步长）、padding（填充）、dilation（膨胀系数）、group（将原始输入通道划分成的组数）

②池化层

kernel_size（卷积核大小）、stride（卷积步长）、padding（填充）、dilation（膨

胀系数）

③全连接层

in_feature（输入样本的大小）、out_feature（输出样本的大小）、bias（如果设置为 False，则图层不会学习附加偏差）

Q8：模型在训练和测试时有哪些差异？

BatchNormal 在训练和测试时有不同的计算流程：

training	track_running_stats	状态
True	Ture	running_mean 和 running_var 会跟踪整个训练过程中 batch 的统计特性，而每组输入数据用当前 batch 的 mean 和 var 统计特性做归一化，然后再更新 running_mean 和 running_var。
Ture	False	每组输入数据会根据当前 batch 的统计特性做归一化，但不会有 running_mean 和 running_var 参数了
False	Ture	使用 running_mean 和 running_var 做归一化，并且不会对其进行更新
False	False	效果同第二点，只不过处于推理状态，不会学习 weight 和 bias 两个参数。一般不采用该状态

训练时

running_mean = (1 - momentum) * mean_old + momentum * mean_new

running_var = (1 - momentum) * var_old + momentum * var_new

测试时

running_mean = mean_old

running_var = var_old

Dropout 在训练和测试时有不同：

在 model.train()时随机取一部分网络连接来训练更新参数。

在 model.eval()时利用到了所有网络连接，不进行随机舍弃神经元。

Q9：如果一个分类模型只能进行二分类，怎么修改让它进行十分类？

将最后一层全连接层 nn.Linear 的 out_features 改为所需的类别数

Q10：在 Pytorch 中，哪些常用的操作、方法、成员？

resize、permute（维度调换）、cat（维度拼接）、unsqueeze（维度扩充）

疑问

（1）if rank in[-1, 0]这个判断起到什么作用呢？

```
if rank in [-1, 0]:
    val_dataset = MyDataset(cfg=cfg, mode='val')
    val_loader = DataLoader(val_dataset,
                            batch_size=cfg.TRAIN.BATCHSIZE_PER_CARD,
                            shuffle=False,
                            num_workers=cfg.TRAIN.NUM_WORKERS,
                            drop_last=cfg.TRAIN.DROP_LAST,
                            collate_fn=val_dataset.collate_fn)

net = get_model(cfg, num_classes)   # 创建模型
logger.info(get_parameter_number(net))  # 计算模型参数量
net = net.to(device)

optimizer = build_optimizer(net, cfg, logger)   # 创建优化器
scheduler = build_scheduler(optimizer, cfg)   # 创建学习率
loss_function = get_loss(cfg)   # 创建损失函数
start_epoch = 0
```

（2）resnet 中

```
def _forward_impl(self, x):
    # See note [TorchScript super()]
    output = []
    x = self.conv1(x)
    output.append(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)

    x = self.layer1(x)
    output.append(x)
    x = self.layer2(x)
    output.append(x)
    x = self.layer3(x)
    output.append(x)
    x = self.layer4(x)
    output.append(x)
    #
    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.fc(x)
```

这个_forward_impl 什么时候会调用这一块？

（3）TripletLoss 这一块看不懂，在后面的学习过程中会涉及到嘛？如果会的话可以先不用帮我解释。

记录：

参数解析

①先将所有节点初始化，通过 merge_from_file()来 update

```
import yaml
import argparser
from fvcore.common.config import CfgNode
cfg = CfgNode()
cfg.TRAIN= CfgNode()     #每一级都要这样新建一个节点
cfg.TRAIN.RESUME_PATH = "Train"
cfg.TRAIN.DATASET = "ucf24"  # `ava`, `ucf24` or `jhmdb21`
cfg.TRAIN.BATCH_SIZE=10
cfg.TRAIN.TOTAL_BATCH_SIZE=128
...
cfg.SOLVER= CfgNode()  #每一级都要这样新建一个节点
cfg.SOLVER.MOMENTUM=0.9
cfg.SOLVER.WEIGHT_DECAY=5e-4
...

yaml_path = "yaml_test.yaml"
cfg.merge_from_file(yaml_path)

#访问方法
print(cfg.TRAIN.RESUME_PATH)
```

如果不将所有的参数都初始化一遍的话会报以下形式的错误：

```
KeyError: 'Non-existent config key: GLOBAL'
```

②

通过 opt = yaml.load(file.read, Loader=yaml.FullLoader)来读取 yaml 文件，访问的方式可以通过字典查询的方式，e.g.:opt[‘一级’][‘二级’]

③

通过 argparse.Namespace(**)来读取

④

更适合参数少，且每次改动不多的情况

Argparse.ArgumentParse()

.add_argument()

.parse_args()

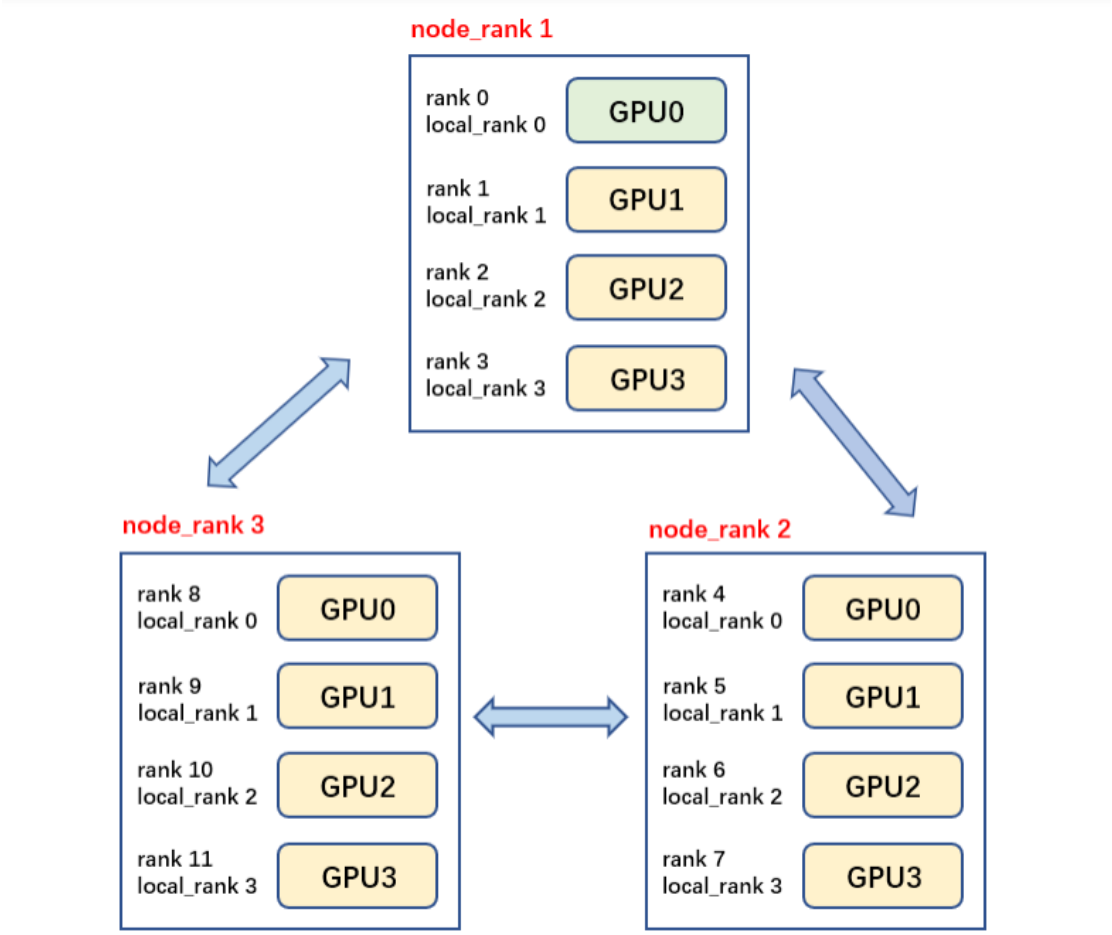
分布式训练

RANK：进程号，在多进程上下文中，我们通常假定 rank 0 是第一个进程或者主进程，其它进程分别具有 1，2，3 不同 rank 号，这样总共具有 4 个进程

NODE：物理节点，可以是一个容器也可以是一台机器，节点内部可以有多个 GPU；nnodes 指物理节点数量，nproc_per_node 指每个物理节点上面进程的数量

LOCAL_RANK：指在一个 node 上进程的相对序号，local_rank 在 node 之间相互独立

WORLD_SIZE：全局进程总个数，即在一个分布式任务中 rank 的数量



学习率设置

lr_scheduler.LambdaLR(optimizer, lr_lambda)

更新学习率的方式是：lr = lr * lr_lambda

不同的层使用不同的学习率调整方法，分别分为权重层 weight，偏执层 bais，和 BN 层，单独调整不同层的学习率可以使得模型训练的更好

torch.optim.lr_scheduler 包用于动态修改 lr，使用的时候需要显式地调用 optimizer.step()和 scheduler.step()来更新 lr，学习率的更新应该放在优化器更新之后

```
model = [Parameter(torch.randn(2, 2, requires_grad=True))]
optimizer = SGD(model, 0.1)
scheduler = ExponentialLR(optimizer, gamma=0.9)

for epoch in range(20):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler.step()
```