张量处理

张量基本信息

```
tensor = torch.randn(3,4,5)
print(tensor.type())  # 数据类型
print(tensor.size())  # 张量大小
print(tensor.dim())   # 维度的数量
```

张量命名

```
NCHW = [‘N’, ‘C’, ‘H’, ‘W’]
images = torch.randn(32, 3, 56, 56, names=NCHW)
images.sum('C')
images.select('C', index=0)
```

torch.Tensor与np.ndarray转换

```
ndarray = tensor.cpu().numpy()
tensor = torch.from_numpy(ndarray).float()
```

Torch.tensor与PIL.Image转换

```
# torch.Tensor -> PIL.Image
image = torchvision.transforms.functional.to_pil_image(tensor)
# PIL.Image -> torch.Tensor
path = r'./figure.jpg'
tensor =torchvision.transforms.functional.to_tensor(PIL.Image.open(path))
```

np.ndarray与PIL.Image的转换

```
image = PIL.Image.fromarray(ndarray.astype(np.uint8))
ndarray = np.asarray(PIL.Image.open(path))
```

张量拼接

torch.cat()：沿着给定的维度拼接

torch.stack()：新增一个维度

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

将整数标签转为one-hot编码

```
# pytorch 的标记默认从 0 开始
tensor = torch.tensor([0, 2, 1, 3])
N = tensor.size(0) num_classes = 4
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).long())
```

矩阵乘法

```
# Matrix multiplcation: (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
```

模型定义

两层卷积网络的示例

```
class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.fc = nn.Linear(7*7*32, num_classes)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out) return out
model = ConvNet(num_classes).to(device)
```

计算模型整体参数量

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

模型权重初始化

model.modules()： 迭代地遍历模型的所有子层

model.children()： 只遍历模型下的一层

```python
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)
layer.weight = torch.nn.Parameter(tensor)
```

将在 GPU 保存的模型加载到 CPU

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

数据处理

计算数据集的均值和标准差

```python
import os
import cv2
import numpy as np
from torch.utils.data import Dataset
from PIL import Image
def compute_mean_and_std(dataset):
    # 输入 PyTorch 的 dataset，输出均值和标准差
    mean_r = 0
    mean_g = 0
    mean_b = 0
    for img, _ in dataset:
        img = np.asarray(img) # PIL Image转为numpy array
        mean_b += np.mean(img[:, :, 0])
        mean_g += np.mean(img[:, :, 1])
        mean_r += np.mean(img[:, :, 2])

    mean_b /= len(dataset)
    mean_g /= len(dataset)
    mean_r /= len(dataset)

    diff_r = 0
    diff_g = 0
    diff_b = 0
    N = 0
    for img, _ in dataset:
        img = np.asarray(img)

        diff_b += np.sum(np.power(img[:, :, 0] - mean_b, 2))
        diff_g += np.sum(np.power(img[:, :, 1] - mean_g, 2))
        diff_r += np.sum(np.power(img[:, :, 2] - mean_r, 2))

```

```
        N += np.prod(img[:, :, 0].shape)

    std_b = np.sqrt(diff_b / N)
    std_g = np.sqrt(diff_g / N)
    std_r = np.sqrt(diff_r / N)

    mean = (mean_b.item() / 255.0, mean_g.item() / 255.0, mean_r.item() / 255.0)
    std = (std_b.item() / 255.0, std_g.item() / 255.0, std_r.item() / 255.0)
    return mean, std
```

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)), ])
```

模型训练和测试

分类模型训练代码

```
# 损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
# 训练模型
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i ,(images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        # 计算损失
        outputs = model(images)
        loss = criterion(outputs, labels)

    # 梯度反向传播
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    if (i+1) % 100 == 0:
        print('Epoch: [{}/{}], Step: [{}/{}], Loss: {}'.format(epoch+1, num_epochs, i+1, total_step, loss.item()))
```

分类模型测试代码

```
# 测试模型
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    print('Test accuracy of the model on the 10000 test images: {} %'
.format(100 * correct / total))
```

自定义损失函数

```
class MyLoss(torch.nn.Moudle):
    def __init__(self):
        super(MyLoss, self).__init__()
    def forward(self, x, y):
        loss = torch.mean((x - y) ** 2)
    return loss
```

预训练模型修改

```
class Net(nn.Module):
    def __init__(self , model):
        super(Net, self).__init__()
        # 忽略模型的最后两层
        self.resnet_layer = nn.Sequential(*list(model.children())[:-2])
        # 自定义层
        self.transion_layer = nn.ConvTranspose2d(2048, 2048, kernel_size=14,
stride=3)
        self.pool_layer = nn.MaxPool2d(32)
        self.Linear_layer = nn.Linear(2048, 8)

    def forward(self, x):
        x = self.resnet_layer(x)
        x = self.transion_layer(x)
        x = self.pool_layer(x)
        x = x.view(x.size(0), -1)
        x = self.Linear_layer(x)
        return x

resnet = models.resnet50(pretrained= True)
model = Net(resnet)
```

学习率衰减策略

```
# 定义优化器
```

```
optimizer_ExpLR = torch.optim.SGD(net.parameters(),lr=0.1)
# 指数衰减
ExpLR = torch.optim.lr_scheduler.ExponentialLR(optimizer_ExpLR,
                                               gamma=0.98)
# 固定步长衰减
optimizer_StepLR = torch.optim.SGD(net.parameters(), lr=0.1)
StepLR = torch.optim.lr_scheduler.StepLR(optimizer_StepLR,
                                         step_size=step_size,
                                         gamma=0.65)
# 多步长衰减
optimizer_MultiStepLR = torch.optim.SGD(net.parameters(), lr=0.1)
torch.optim.lr_scheduler.MultiStepLR(optimizer_MultiStepLR,
                                     milestones=[200, 300, 320, 340, 200],
                                     gamma=0.8)
# 余弦退火衰减
optimizer_CosineLR = torch.optim.SGD(net.parameters(), lr=0.1)
CosineLR = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer_CosineLR,
                                                      T_max=150,
                                                      eta_min=0)
```

保存与加载断点

```
# 加载模型
if resume:
    model_path = os.path.join('model', 'best_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 {}.'.format(start_epoch))
    print('Best accuracy so far {}.'.format(best_acc))
# 训练模型
for epoch in range(start_epoch, num_epochs):
    ...
    # 测试模型
    ...
    # 保存checkpoint
    is_best = current_acc > best_acc
    best_acc = max(current_acc, best_acc)
    checkpoint = { 'best_acc': best_acc,
                   'epoch': epoch + 1,
                   'model': model.state_dict(),
                   'optimizer': optimizer.state_dict(), }
    model_path = os.path.join('model', 'checkpoint.pth.tar')
    best_model_path = os.path.join('model', 'best_checkpoint.pth.tar')
    torch.save(checkpoint, model_path)
    if is_best: shutil.copy(model_path, best_model_path)
```

注意事项

- model(x) 定义好后，用 model.train() 和 model.eval() 切换模型状态。
- 使用with torch.no_grad() 包含无需计算梯度的代码块

- model.eval()与torch.no_grad的区别：前者是将模型切换为测试态，例如BN和Dropout在训练和测试阶段使用不同的计算方法；后者是关闭张量的自动求导机制，减少存储和加速计算。
- torch.nn.CrossEntropyLoss 等价于 torch.nn.functional.log_softmax + torch.nn.NLLLoss。
- ReLU可使用inplace操作减少显存消耗。
- 使用半精度浮点数 half() 可以节省计算资源同时提升模型计算速度，但需要小心数值精度过低带来的稳定性问题。