第19次作业讲评

2023/5/25

朱成轩

0.1

```python
import torch
import numpy as np

# 创建一个张量 (tensor)
x = torch.tensor([1, 2, 3, 4])
print("x:", x)

# 创建一个矩阵 (matrix)
y = torch.tensor([[1, 2], [3, 4]])
print("y:", y)

# 创建一个全零的张量
z = torch.zeros((2, 3))
print("z:", z)

# 创建一个随机初始化的张量
w = torch.randn((3, 3))
print("w:", w)

# pytorch tensor与numpy array的相互转换
numpy_array = np.array([1, 2, 3, 4, 5])

# 将NumPy数组转换为PyTorch张量
torch_tensor = torch.from_numpy(numpy_array)
print(torch_tensor)

# 将PyTorch张量转换为NumPy数组
numpy_array = torch_tensor.numpy()
print(numpy_array)
```

- 区分torch.Tensor(...)和torch.tensor(...)

```python
import torch
print(torch.Tensor([1,2,3,4]).dtype, torch.tensor([1,2,3,4]).dtype)
print(torch.Tensor([True,False]).dtype, torch.tensor([True,False]).dtype)
print(torch.Tensor([1e5]).dtype, torch.tensor([1e5]).dtype)
```

✓ 0.0s

```
torch.float32 torch.int64
torch.float32 torch.bool
torch.float32 torch.float32
```

- torch.from_numpy(...)和torch.tensor(...)

```python
import numpy as np
import torch
numpy_array = np.array([1., 2., 3., 4., 5.])
torch_tensor = torch.tensor(numpy_array)
torch_from_numpy = torch.from_numpy(numpy_array)
print(torch_tensor.dtype, numpy_array.dtype, torch_from_numpy.dtype)
torch_tensor[0] = 0
torch_from_numpy[-1] = 0
print(numpy_array)
```

✓ 0.0s

```
torch.float64 float64 torch.float64
[1. 2. 3. 4. 0.]
```

0.1

```python
import torch
import numpy as np

# 创建一个张量 (tensor)
x = torch.tensor([1, 2, 3, 4])
print("x:", x)

# 创建一个矩阵 (matrix)
y = torch.tensor([[1, 2], [3, 4]])
print("y:", y)

# 创建一个全零的张量
z = torch.zeros((2, 3))
print("z:", z)

# 创建一个随机初始化的张量
w = torch.randn((3, 3))
print("w:", w)

# pytorch tensor与numpy array的相互转换
numpy_array = np.array([1, 2, 3, 4, 5])

# 将NumPy数组转换为PyTorch张量
torch_tensor = torch.from_numpy(numpy_array)
print(torch_tensor)

# 将PyTorch张量转换为NumPy数组
numpy_array = torch_tensor.numpy()
print(numpy_array)
```

- 同理，.numpy()和np.array(...)

```python
import numpy as np
import torch
t = torch.tensor([1,2,3])
t_numpy = t.numpy()
np_t = np.array(t)
np_t[0] = 0
t_numpy[-1] = 0
print(t)
```

✓ 0.0s

```
tensor([1, 2, 0])
```

TORCH.FROM_NUMPY

torch.from_numpy(*ndarray*) → Tensor

Creates a `Tensor` from a `numpy.ndarray`.

The returned tensor and `ndarray` share the same memory. Modifications to the tensor will be reflected in the `ndarray` and vice versa. The returned tensor is not resizable.

TORCH.TENSOR.NUMPY

Tensor.numpy(*, *force=False*) → numpy.ndarray

Returns the tensor as a NumPy `ndarray`.

If `force` is `False` (the default), the conversion is performed only if the tensor is on the CPU, does not require grad, does not have its conjugate bit set, and is a dtype and layout that NumPy supports. The returned ndarray and the tensor will share their storage, so changes to the tensor will be reflected in the ndarray and vice versa.

0.2

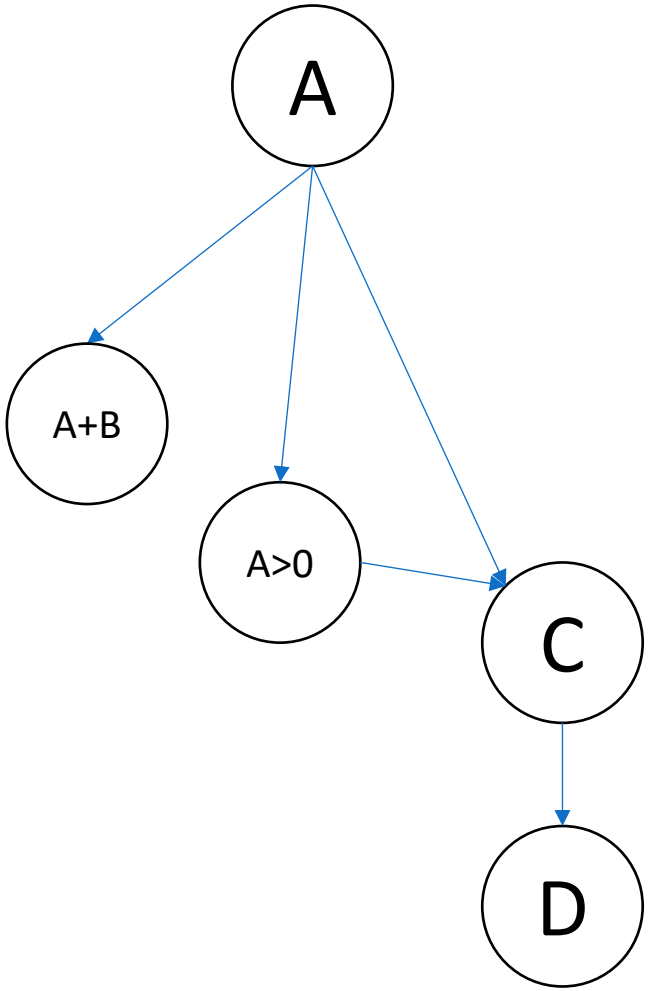
```python
# 创建一个服从标准正态分布的3*4的张量A
A = torch.randn((3, 4)).requires_grad_(True)
print("A:", A)

# 创建一个服从标准正态分布的2*6的张量B
B = torch.randn((2, 6))
print("B:", B)

# 使用view操作将B的形状也转换成3*4，计算A+B和A*B^T（矩阵乘法）
B = B.view(3, 4)
print("A+B:", A + B)
print("A*B^T:", A.matmul(B.t()))

# 取出矩阵A的所有大于0的值，并按照行的顺序排列成一个一维的张量C。
# 例如：[[0, 0.1, 0.2], [-0.3, 0.4, -0.5]]需要转换成[0.1, 0.2, 0.4]
C = A[A > 0]
print("C:", C)

# 将C的每个元素平方，得到D。计算D的平均值mean(D)，并求出mean(D)对A的梯度
D = C ** 2
D = D.clone().detach().requires_grad_(True)  # 需指定requires_grad=True，否则无法求梯度
print("D:", D)
print("mean(D):", D.mean())
D.mean().backward()
print("A.grad:", A.grad)
```



0.2

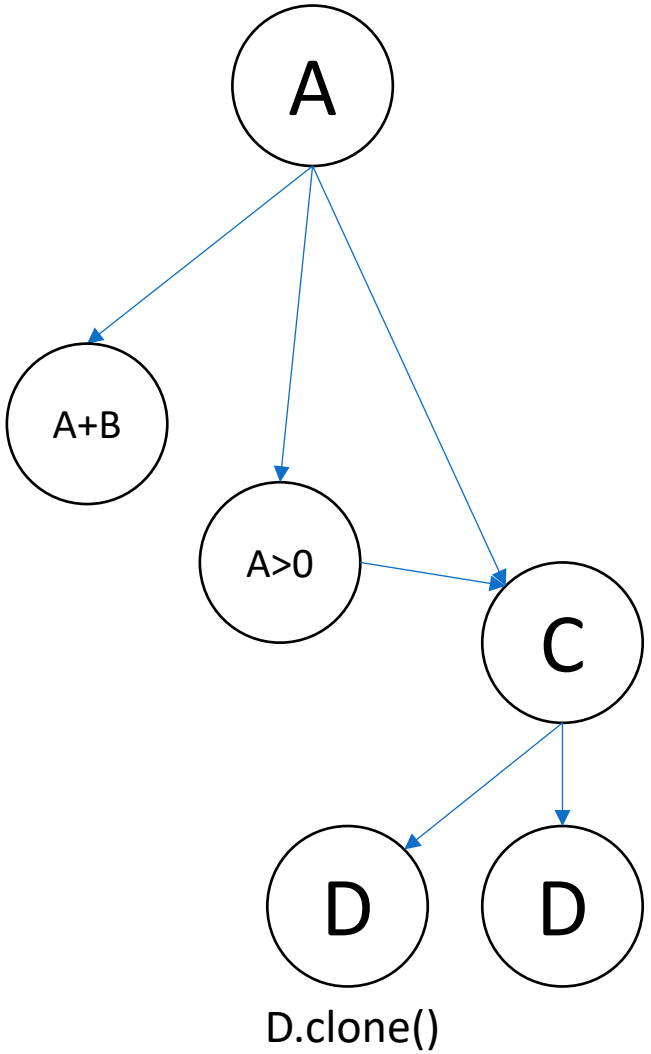
```python
# 创建一个服从标准正态分布的3*4的张量A
A = torch.randn((3, 4)).requires_grad_(True)
print("A:", A)

# 创建一个服从标准正态分布的2*6的张量B
B = torch.randn((2, 6))
print("B:", B)

# 使用view操作将B的形状也转换成3*4，计算A+B和A*B^T（矩阵乘法）
B = B.view(3, 4)
print("A+B:", A + B)
print("A*B^T:", A.matmul(B.t()))

# 取出矩阵A的所有大于0的值，并按照行的顺序排列成一个一维的张量C。
# 例如：[[0, 0.1, 0.2], [-0.3, 0.4, -0.5]]需要转换成[0.1, 0.2, 0.4]
C = A[A > 0]
print("C:", C)

# 将C的每个元素平方，得到D。计算D的平均值mean(D)，并求出mean(D)对A的梯度
D = C ** 2
D = D.clone().detach().requires_grad_(True)  # 需指定requires_grad=True，否则无法求梯度
print("D:", D)
print("mean(D):", D.mean())
D.mean().backward()
print("A.grad:", A.grad)
```



0.2

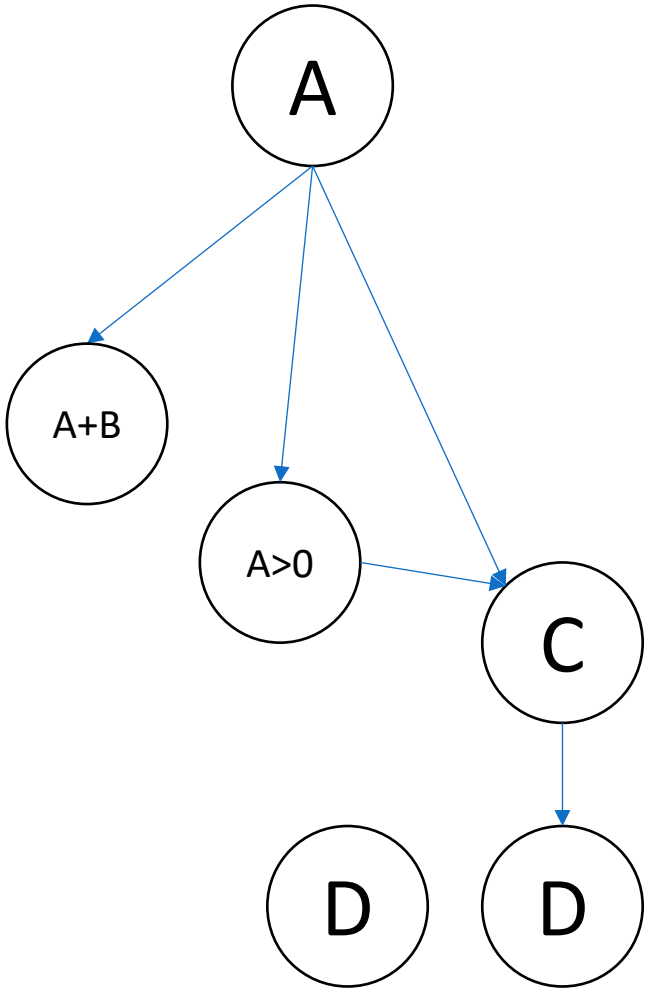
```python
# 创建一个服从标准正态分布的3*4的张量A
A = torch.randn((3, 4)).requires_grad_(True)
print("A:", A)

# 创建一个服从标准正态分布的2*6的张量B
B = torch.randn((2, 6))
print("B:", B)

# 使用view操作将B的形状也转换成3*4，计算A+B和A*B^T（矩阵乘法）
B = B.view(3, 4)
print("A+B:", A + B)
print("A*B^T:", A.matmul(B.t()))

# 取出矩阵A的所有大于0的值，并按照行的顺序排列成一个一维的张量C。
# 例如：[[0, 0.1, 0.2], [-0.3, 0.4, -0.5]]需要转换成[0.1, 0.2, 0.4]
C = A[A > 0]
print("C:", C)

# 将C的每个元素平方，得到D。计算D的平均值mean(D)，并求出mean(D)对A的梯度
D = C ** 2
D = D.clone().detach().requires_grad_(True)  # 需指定requires_grad=True，否则无法求梯度
print("D:", D)
print("mean(D):", D.mean())
D.mean().backward()
print("A.grad:", A.grad)
```



D.clone().detach()

0.2

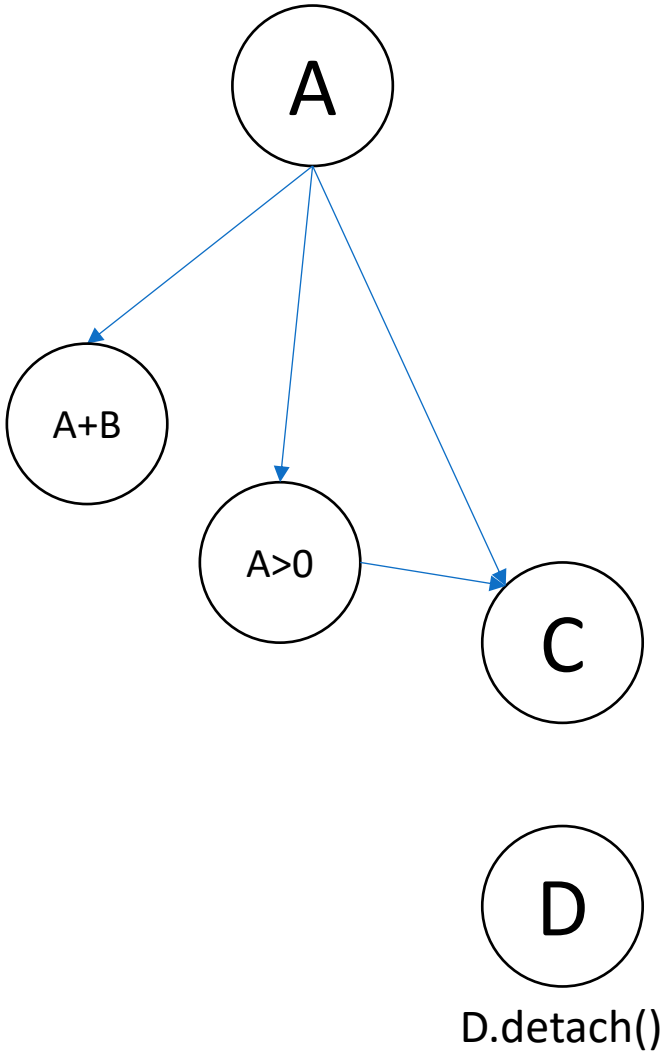
```python
# 创建一个服从标准正态分布的3*4的张量A
A = torch.randn((3, 4)).requires_grad_(True)
print("A:", A)

# 创建一个服从标准正态分布的2*6的张量B
B = torch.randn((2, 6))
print("B:", B)

# 使用view操作将B的形状也转换成3*4，计算A+B和A*B^T（矩阵乘法）
B = B.view(3, 4)
print("A+B:", A + B)
print("A*B^T:", A.matmul(B.t()))

# 取出矩阵A的所有大于0的值，并按照行的顺序排列成一个一维的张量C。
# 例如：[[0, 0.1, 0.2], [-0.3, 0.4, -0.5]]需要转换成[0.1, 0.2, 0.4]
C = A[A > 0]
print("C:", C)

# 将C的每个元素平方，得到D。计算D的平均值mean(D)，并求出mean(D)对A的梯度
D = C ** 2
D = D.clone().detach().requires_grad_(True)  # 需指定requires_grad=True，否则无法求梯度
print("D:", D)
print("mean(D):", D.mean())
D.mean().backward()
print("A.grad:", A.grad)
```



0.2

```python
# 创建一个服从标准正态分布的3*4的张量A
A = torch.randn((3, 4)).requires_grad_(True)
print("A:", A)

# 创建一个服从标准正态分布的2*6的张量B
B = torch.randn((2, 6))
print("B:", B)

# 使用view操作将B的形状也转换成3*4，计算A+B和A*B^T（矩阵乘法）
B = B.view(3, 4)
print("A+B:", A + B)
print("A*B^T:", A.matmul(B.t()))

# 取出矩阵A的所有大于0的值，并按照行的顺序排列成一个一维的张量C。
# 例如：[[0, 0.1, 0.2], [-0.3, 0.4, -0.5]]需要转换成[0.1, 0.2, 0.4]
C = A[A > 0]
print("C:", C)

# 将C的每个元素平方，得到D。计算D的平均值mean(D)，并求出mean(D)对A的梯度
D = C ** 2
D = D.clone().detach().requires_grad_(True)  # 需指定requires_grad=True，否则无法求梯度
print("D:", D)
print("mean(D):", D.mean())
D.mean().backward()
print("A.grad:", A.grad)
```

D

0.2

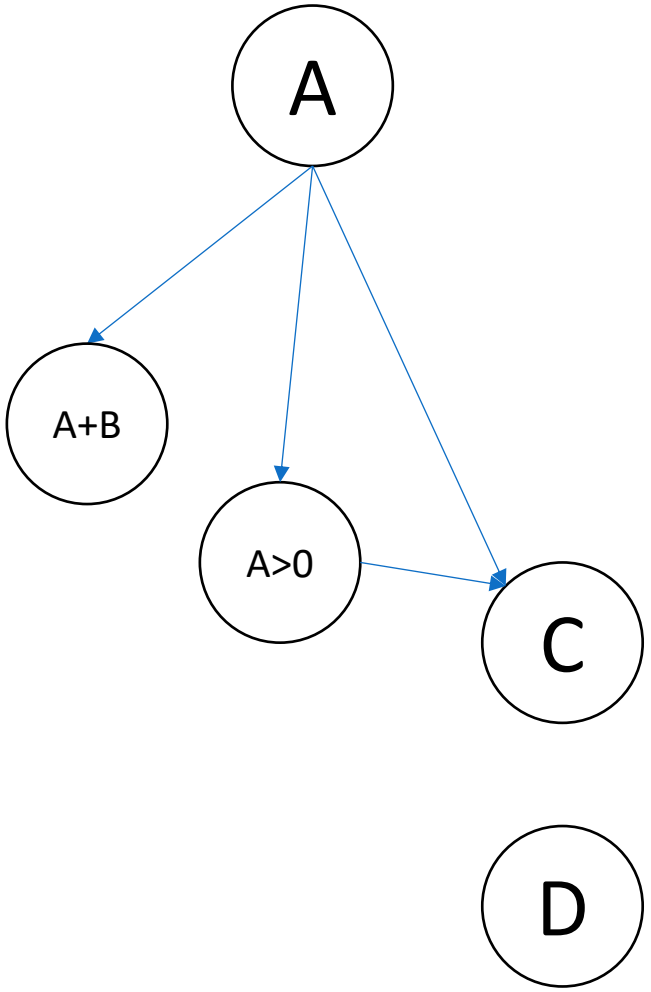
```python
# 创建一个服从标准正态分布的3*4的张量A
A = torch.randn((3, 4)).requires_grad_(True)
print("A:", A)

# 创建一个服从标准正态分布的2*6的张量B
B = torch.randn((2, 6))
print("B:", B)

# 使用view操作将B的形状也转换成3*4，计算A+B和A*B^T（矩阵乘法）
B = B.view(3, 4)
print("A+B:", A + B)
print("A*B^T:", A.matmul(B.t()))

# 取出矩阵A的所有大于0的值，并按照行的顺序排列成一个一维的张量C。
# 例如：[[0, 0.1, 0.2], [-0.3, 0.4, -0.5]]需要转换成[0.1, 0.2, 0.4]
C = A[A > 0]
print("C:", C)

# 将C的每个元素平方，得到D。计算D的平均值mean(D)，并求出mean(D)对A的梯度
with torch.no_grad():
    D = C ** 2
D = D.requires_grad_(True)  # 需指定requires_grad=True，否则无法求梯度
print("D:", D)
print("mean(D):", D.mean())
D.mean().backward()
print(D.grad)
print("A.grad:", A.grad)
```



0.3

```python
import torch
import torch.nn as nn

# 定义一个简单的神经网络
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, sel
        self.fc1 = nn.Linear
        self.fc2 = nn.Linear
        self.relu = nn.ReLU(

    def forward(self, x):
        x = self.relu(self.f
        x = self.fc2(x)
        return x

# 创建网络实例
net = SimpleNet()
print(net)
```

```python
import torch
import torch.nn as nn

# 定义一个简单的神经网络
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init_
        self.fc1 = nn.Sequential(
            nn.Linear(10, 5),
            nn.ReLU(),
            nn.Linear(5, 2)
        )

    def forward(self, x):
        return self.fc1(x)

# 创建网络实例
net = SimpleNet()
print(net)
```

✓ 0.0s

```
SimpleNet(
  (fc1): Sequential(
    (0): Linear(in_features=10, out_featur
    (1): ReLU()
    (2): Linear(in_features=5, out_feature
  )
)
```

```python
import torch
import torch.nn as nn

# 定义一个简单的神经网络
class SimpleNet(nn.Module):
    def __init__(self, layers=5):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.ModuleList()
        for i in range(layers):
            if i>0:
                self.fc1.append(nn.ReLU())
            self.fc1.append(nn.Linear(10-i,10-i-1))
        self.fc1 = nn.Sequential(*self.fc1)

    def forward(self, x):
        return self.fc1(x)

# 创建网络实例
net = SimpleNet()
print(net)
```

✓ 0.0s

```
SimpleNet(
  (fc1): Sequential(
    (0): Linear(in_features=10, out_features=9, bias=True)
    (1): ReLU()
    (2): Linear(in_features=9, out_features=8, bias=True)
    (3): ReLU()
    (4): Linear(in_features=8, out_features=7, bias=True)
    (5): ReLU()
    (6): Linear(in_features=7, out_features=6, bias=True)
```

1.1

```
class kernel(nn.Module):
    def __init__(self):
        super(kernel,self).__init__()
        # TODO, 设计卷积核的格式并将卷积核的权重初始化为指定的内容
        self.conv = nn.Conv2d(1,1,3,1,1)
        self.conv.weight = nn.Parameter(torch.Tensor([[[[0,-1,0], [-1,4,-1],[0,-1,0]]]]), requires_grad=False)
    def forward(self,x):
        # TODO
        return self.conv(x)
```

- 学习如何初始化参数
  （一般设置requires_grad=True）

```
def convolution(image, kernel):
    # TODO
    image = ToTensor()(image).unsqueeze(0)
    res = kernel(image)
    return res.squeeze(0).permute(1,2,0).detach().numpy()

conv_kernel = kernel()
res = convolution(lena,conv_kernel)
plt.imshow(res,'gray')
```

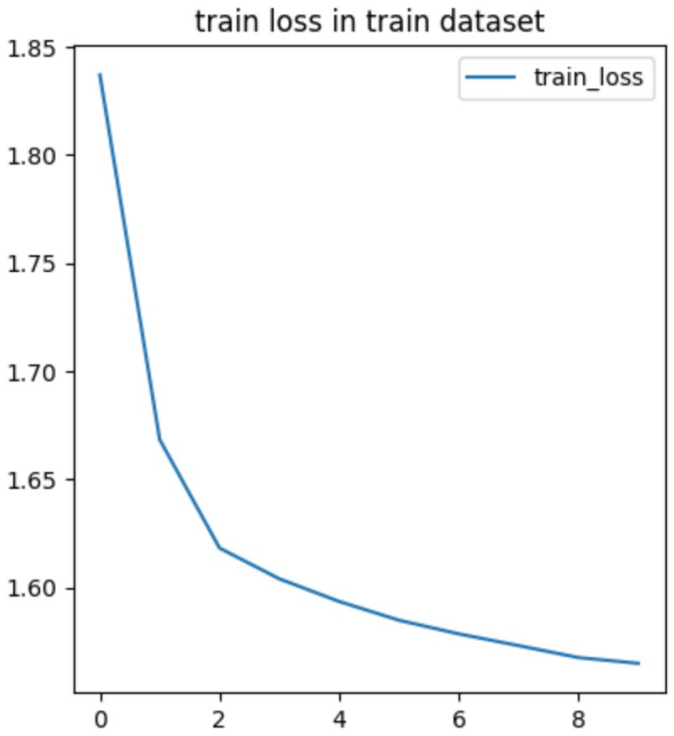
1.2

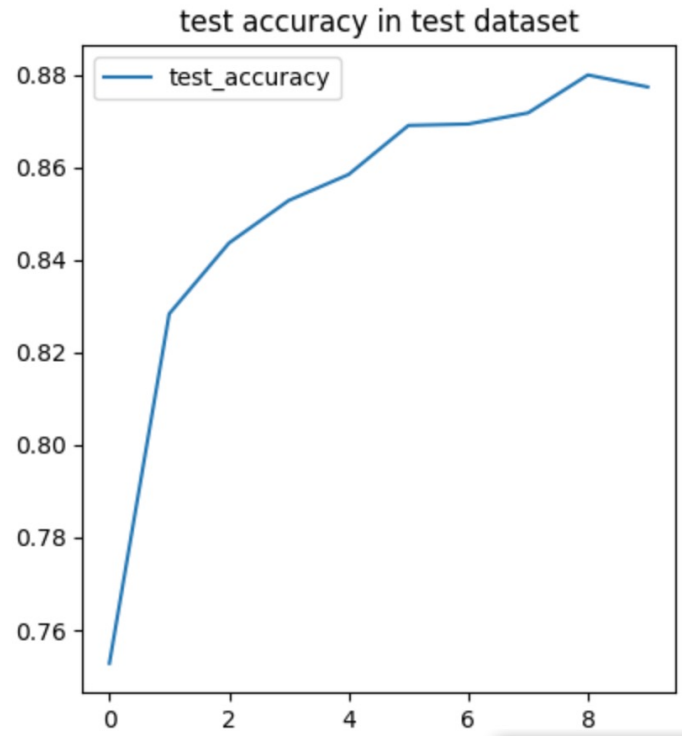
```python
def forward(self, x):
    # TODO
    x = self.conv_1(x)
    x = nn.functional.relu(x)
    x = nn.functional.max_pool2d(x,2,2)
    x = self.conv_2(x)
    x = nn.functional.relu(x)
    x = nn.functional.max_pool2d(x,2,2)
    x = self.conv_3(x)
    x = nn.functional.relu(x)
    x = flatten(x,1)
    # print(x.shape)
    x = self.fc_1(x)
    x = nn.functional.relu(x)
    x = self.fc_2(x)
    output = nn.functional.softmax(x,dim=1)
    return output
```

```python
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    #TODO
    train_loss.append(train_loop(train_dataloader,model,nn.CrossEntropyLoss(),optimizer))
    test_accuracy.append(test_loop(test_dataloader,model,nn.CrossEntropyLoss()))
```

```python
y_pred = model(image.to(device))
loss = loss_fn(y_pred,label.to(device))
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Loss minimum ~1.46115, but acc seems OK, why?





张润博 2100017798

Loss func.

- `torch.nn.LogSoftmax()(x)`
- `torch.nn.NLLLoss()(x,y)`
  - Negative Log-Likelihood Loss
- `torch.nn.CrossEntropyLoss()(x,y)`
  - 即LogSoftmax + NLLLoss

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

$$l_n = -w_{y_n} x_{n,y_n}$$

x: 已经过softmax $\in [0,1]^N$

x: 未经过softmax $\in \mathbb{R}^N$

NLLLoss不如改叫negative likelihood loss，反正本身没log

1.2

```
import torchvision.transforms as T
```

PyTorch关于图像的数据增强工具

https://pytorch.org/vision/stable/transforms.html

Color

ColorJitter([brightness, contrast, ...])

Grayscale([num_output_channels])

GaussianBlur(kernel_size[, sigma])

RandomEqualize([p])

Geometry

Resize(size[, interpolation, max_size, ...])

RandomCrop(size[, padding, pad_if_needed

RandomResizedCrop(size[, scale, ratio,

CenterCrop(size)

Pad(padding[, fill, padding_mode])

RandomAffine(degrees[, translate, scale, ...])

RandomHorizontalFlip([p])

RandomVerticalFlip([p])

Conversion

ToPILImage([mode])

ToTensor()

PILToTensor()

ConvertImageDtype(dtype)

Composition

Compose(transforms)

v2.Compose(transforms)

RandomApply(transforms[, p])

v2.RandomApply(transforms[, p])

RandomChoice(transforms[, p])

v2.RandomChoice(transforms[, p])

RandomOrder(transforms)

v2.RandomOrder(transforms)

1.2

```
import torchvision.transforms as T
```

PyTo

https://

Auto-Augmentation

AutoAugment is a common Data Augmentation technique that can improve the accuracy of Image Classification models. Though the data augmentation policies are directly linked to their trained dataset, empirical studies show that ImageNet policies provide significant improvements when applied to other datasets. In TorchVision we implemented 3 policies learned on the following datasets: ImageNet, CIFAR10 and SVHN. The new transform can be used standalone or mixed-and-matched with existing transforms:

`AutoAugmentPolicy`(value)	AutoAugment policies learned on different datasets.
`AutoAugment`([policy, interpolation, fill])	AutoAugment data augmentation method based on "AutoAugment: Learning Augmentation Strategies from Data".
`v2.AutoAugment`([policy, interpolation, fill])	[BETA] AutoAugment data augmentation method based on "AutoAugment: Learning Augmentation Strategies from Data".
`RandAugment`([num_ops, magnitude, ...])	RandAugment data augmentation method based on "RandAugment: Practical automated data augmentation with a reduced search space".

1.2

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size=len(dataloader.dataset)
    for batch, (image, label) in enumerate(dataloader):
        # TODO, 训练过程 (4分)
        pred = model(image)
        loss = loss_fn(pred, label)
        # 反向传播并更新梯度
        optimizer.zero_grad() # 在每次迭代之前，将所有参数的梯度清零
        loss.backward()
        optimizer.step()
        if batch % 100 == 0:
            print("{} / {} , loss: {:.6f}".format(batch*len(label), size, loss.item()))
    return
```

一般数据按照(B, C, H, W)给出

- 每个iteration从dataloader得到数据的batch size <= 预设的batchsize

```
DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
           batch_sampler=None, num_workers=0, collate_fn=None,
           pin_memory=False, drop_last=False, timeout=0,
           worker_init_fn=None, *, prefetch_factor=2,
           persistent_workers=False)
```

1.2

CLASS torch.utils.data.DataLoader(*dataset*, *batch_size=1*, *shuffle=None*, *sampler=None*, *batch_sampler=None*, *num_workers=0*, *collate_fn=None*, *pin_memory=False*, *drop_last=False*, *timeout=0*, *worker_init_fn=None*, *multiprocessing_context=None*, *generator=None*, ***, *prefetch_factor=None*, *persistent_workers=False*, *pin_memory_device=''*) [SOURCE]

- **num_workers** (*int*, *optional*) – how many subprocesses to use for data loading. `0` means that the data will be loaded in the main process. (default: `0`)
  - num_workers越多越好吗？

- **pin_memory** (*bool*, *optional*) – If `True`, the data loader will copy Tensors into device/CUDA pinned memory before returning them. If your data elements are a custom type, or your `collate_fn` returns a batch that is a custom type, see the example below.

  Host to GPU copies are much faster when they originate from pinned (page-locked) memory. See Use pinned memory buffers for more details on when and how to use pinned memory generally.

  For data loading, passing `pin_memory=True` to a `DataLoader` will automatically put the fetched data Tensors in pinned memory, and thus enables faster data transfer to CUDA-enabled GPUs.

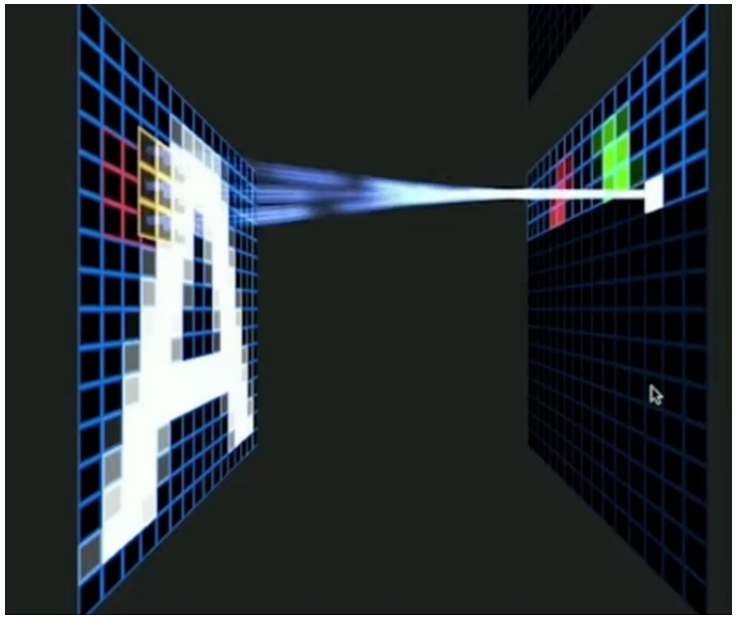
```python
def test_loop(dataloader, model, loss_fn):
    with torch.no_grad():
        correct = 0
        for image, label in dataloader:
            # TODO, 测试过程 (4分)
            y_pred = model(image.to(device)).cpu()
            correct += (y_pred.argmax(1) == label).type(torch.float).sum().item()
        print("Accuracy: {:.2f}%" .format(correct/len(dataloader.dataset) * 100))
    return correct/len(dataloader.dataset)
```

- `torch.no_grad()` 可以在测试阶段节省许多内存/显存！

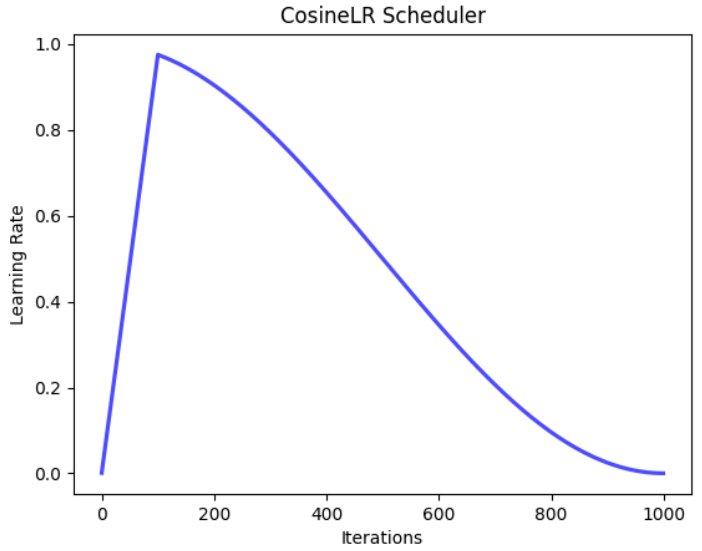
- Why GPU?

  深度学习涉及较多可并行计算（例如卷积）

  # CPU core 10^1, # GPU core 10^3



改进？



- 数据增强

- 卷积核参数：channels, stride, dilation, kernel size…
- 激活函数：LeakyReLU、ELU、GELU…
- 初始化：He, Xavier, …
- 学习率scheduler：warmup, cosine, step, SGDR, …
- 超参数：batchsize, patchsize, sampling strategy, …

总结

- 熟悉PyTorch和numpy类似的一些性质（构造、矩阵乘…）
- 熟悉Tensor和ndarray相互转换的函数（是否共用内存？）
- 熟悉神经网络的设计
- 了解PyTorch如何求导、计算图是什么
- 使用torchvision.transforms进行data augmentation