neural_networks_tutorial

February 4, 2020

[18]:
```
import torch

a = torch.randn(100, 2)
b = torch.randn(200, 2)
```

[19]:
```
# a * b
```

[20]:
```
# a + b
```

[21]:
```
a.dtype  # quantization (in development)
```

[21]: torch.float32

[22]:
```
a.shape
```

[22]: torch.Size([100, 2])

[23]:
```
b.shape
```

[23]: torch.Size([200, 2])

[24]:
```
a = a.unsqueeze(1)
b = b.unsqueeze(0)
a.shape, b.shape
```

[24]: (torch.Size([100, 1, 2]), torch.Size([1, 200, 2]))

[25]:
```
# a - b  # broadcast
```

[26]:
```
((a - b)**2).sum(-1).shape
```

[26]: torch.Size([100, 200])

[27]:
```
def pairwise_distance(a, b):
    diff = a[:, None, :] - b[None, :, :] # Broadcast
    diff_squared = diff ** 2
    return torch.sum(diff_squared, dim=2)
```

```
%timeit pairwise_distance(a, b)
```

```
346 µs ± 20.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

1 Neural Networks

[28]:
```
import matplotlib

%matplotlib inline
```

[29]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convolution␣
 ↪kernel
        self.conv1 = nn.Conv2d(1, 6, 5)  # FIXME Change 1 to 3 channels later
        # Max pooling over a (2, 2) window
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)  # 5*5 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
print(net)
```

```
Net(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
```

```
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

You just have to define the `forward` function, and the `backward` function (where gradients are computed) is automatically defined for you using `autograd`. You can use any of the Tensor operations in the `forward` function.

The learnable parameters of a model are returned by `net.parameters()`

[30]:
```
params = list(net.parameters())
print(len(params))
print(params[0].size())  # conv1's .weight
```

```
10
torch.Size([6, 1, 5, 5])
```

Let's try a random 32x32 input. Note: expected input size of this net (LeNet) is 32x32. To use this net on the MNIST dataset, please resize the images from the dataset to 32x32.

[31]:
```
input = torch.randn(1, 1, 32, 32)
output = net(input)
print(output)
```

```
tensor([[-0.0639,  0.1027, -0.0272, -0.0796,  0.1247, -0.0108,  0.0845,  0.0992,
         -0.0223,  0.0430]], grad_fn=<AddmmBackward>)
```

[32]:
```
target = torch.randn(10)  # a dummy target, for example
target = target.view(1, -1)  # make it the same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)
```

```
tensor(0.7431, grad_fn=<MseLossBackward>)
```

2 Gradient

Now, if you follow `loss` in the backward direction, using its `.grad_fn` attribute, you will see a graph of computations that looks like this:

::

```
input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d
      -> view -> linear -> relu -> linear -> relu -> linear
      -> MSELoss
      -> loss
```

So, when we call `loss.backward()`, the whole graph is differentiated w.r.t. the loss, and all Tensors in the graph that has `requires_grad=True` will have their `.grad` Tensor accumulated with the gradient.

For illustration, let us follow a few steps backward:

[33]:
```
print(loss.grad_fn)  # MSELoss
print(loss.grad_fn.next_functions[0][0])  # Linear
print(loss.grad_fn.next_functions[0][0].next_functions[0][0])  # ReLU
```

```
<MseLossBackward object at 0x128a4be50>
<AddmmBackward object at 0x128a4b050>
<AccumulateGrad object at 0x128a4be50>
```

2.1 Backprop

To backpropagate the error all we have to do is to `loss.backward()`. You need to clear the existing gradients though, else gradients will be accumulated to existing gradients.

Now we shall call `loss.backward()`, and have a look at conv1's bias gradients before and after the backward.

[34]:
```
net.zero_grad()     # zeroes the gradient buffers of all parameters

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

```
conv1.bias.grad before backward
None
conv1.bias.grad after backward
tensor([-0.0031, -0.0039,  0.0046, -0.0095,  0.0068,  0.0153])
```

[35]:
```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```

.. Note::

Observe how gradient buffers had to be manually set to zero using
``optimizer.zero_grad()``. This is because gradients are accumulated
as explained in the `Backprop`_ section.

3 DataLoader

[36]:
```python
import torch
import torchvision
import torchvision.transforms as transforms
```

[37]:
```python
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)
```

```
Files already downloaded and verified
```

[38]:
```python
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

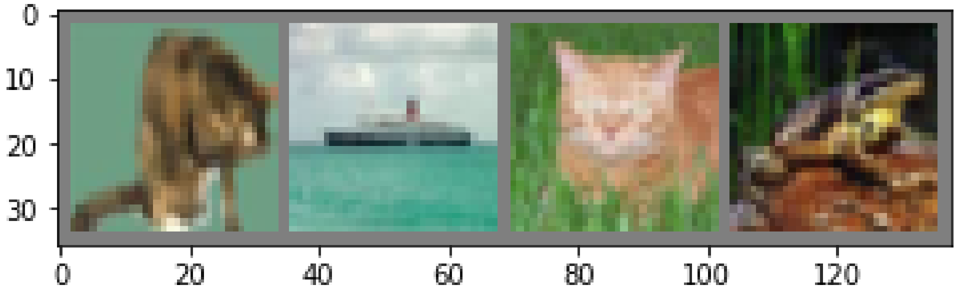
[39]:
```python
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```



```
cat  ship  cat  frog
```

[40]:
```python
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

[41]:
```python
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

[42]:
```python
for epoch in range(2):  # loop over the dataset multiple times
```

```
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)

        # zero the parameter gradients
        optimizer.zero_grad()

        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

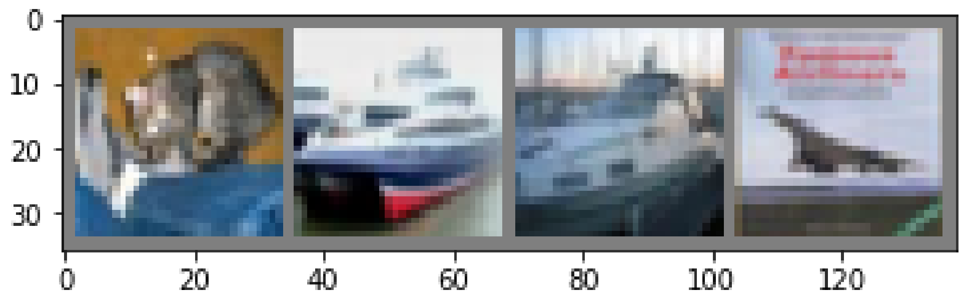
```
[1,  2000] loss: 2.162
[1,  4000] loss: 1.882
[1,  6000] loss: 1.684
[1,  8000] loss: 1.585
[1, 10000] loss: 1.510
[1, 12000] loss: 1.463
[2,  2000] loss: 1.375
[2,  4000] loss: 1.354
[2,  6000] loss: 1.342
[2,  8000] loss: 1.295
[2, 10000] loss: 1.291
[2, 12000] loss: 1.271
Finished Training
```

[43]:
```
testset = torchvision.datasets.CIFAR10(
    root='./data', train=False,
    download=True, transform=transform
)
testloader = torch.utils.data.DataLoader(
    testset, batch_size=4,
    shuffle=False, num_workers=2
)
```

```
dataiter = iter(testloader)
images, labels = dataiter.next()

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

```
Files already downloaded and verified
```



```
GroundTruth:    cat  ship  ship plane
```

[44]:

```
outputs = net(images)

_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```

```
Predicted:   bird  ship  ship plane
```