PyTorch_Training_Example

March 6, 2018

1 Training a classifier

Now let's move further.

1.1 What about data?

- For images, packages such as Pillow, OpenCV are useful.
- For audio, packages such as scipy and librosa
- For text, either raw Python or Cython based loading, or NLTK and SpaCy are useful.

But in this problem, we are using `torchvision`. For this tutorial, we will use the CIFAR10 dataset. It has the classes: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'. The images in CIFAR-10 are of size 3x32x32, i.e. 3-channel color images of 32x32 pixels in size.

1.2 Training an image classifier

We will do the following steps in order: 1. Load and normalizing the CIFAR10 training and test datasets using `torchvision` 2. Define a Convolution Neural Network 3. Define a loss function 4. Train the network on the training data 5. Test the network on the test data

1.2.1 Loading and normalizing CIFAR10

```
In [1]: import torch
        import torchvision
        import torchvision.transforms as transforms
```

```
In [2]: 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)

        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)

        classes = ('plane', 'car', 'bird', 'cat',
                   'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Files already downloaded and verified
Files already downloaded and verified
```

1.2.2 Define a Convolution Neural Network

Copy the neural network from the Neural Networks section before and modify it to take 3-channel images (instead of 1-channel images as it was defined).

```
In [3]: from torch.autograd import Variable
        import torch.nn as nn
        import torch.nn.functional as F
```

```
In [4]: # Define the class
        class Net(nn.Module):
            def __init__(self):
                super(Net, self).__init__()
                self.conv1 = nn.Conv2d(3, 6, 5)
                self.pool = nn.MaxPool2d(2, 2) # Don't remember this appeared last time
                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 is the input data
                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()
```

What is `x.view()`?

`view(*args)` => Tensor. Returns a new tensor with the same data but different size. The returned tensor shares the same data and must have the same number of elements, but may have a different size. A tensor must be `contiguous()` to be viewed. `x.view(-1,8)`, the size `-1` is inferred from other dimensions. Here's an example. **In PyTorch, this thing is exactly the `reshape` thing.**

```
In [5]: eg = torch.randn(4, 4)
        eg.size()
        eg
```

```
Out[5]:
 1.7011  0.5518  0.5657 -0.8732
-0.2426  0.6228  0.9798 -0.9898
-0.6091 -0.6677 -0.1475 -0.3337
-0.2990 -0.2693  0.2793  0.7511
[torch.FloatTensor of size 4x4]

In [6]: eg_view = eg.view(16) # reshape to 16
        eg_view

Out[6]:
 1.7011
 0.5518
 0.5657
-0.8732
-0.2426
 0.6228
 0.9798
-0.9898
-0.6091
-0.6677
-0.1475
-0.3337
-0.2990
-0.2693
 0.2793
 0.7511
[torch.FloatTensor of size 16]

In [7]: eg_view.size()

Out[7]: torch.Size([16])

In [8]: eg_view2 = eg.view(-1, 8)
        eg_view2

Out[8]:
 1.7011  0.5518  0.5657 -0.8732 -0.2426  0.6228  0.9798 -0.9898
-0.6091 -0.6677 -0.1475 -0.3337 -0.2990 -0.2693  0.2793  0.7511
[torch.FloatTensor of size 2x8]

In [9]: eg_view2.size()

Out[9]: torch.Size([2, 8])
```

Now, let's get back to the code. ### Define a Loss function and optimizer Let's use a Classification Cross-Entropy loss and SGD with momentum.

```
In [10]: import torch.optim as optim

         criterion = nn.CrossEntropyLoss()
         optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
         print(type(criterion))
```

```
<class 'torch.nn.modules.loss.CrossEntropyLoss'>
```

1.2.3 Train the network

We simply have to loop over our data iterator, and feed the inputs to the network and optimize

```
In [11]: for epoch in range(2):
             running_loss = 0.0
             for i, data in enumerate(trainloader, 0):
                 # get the inputs
                 inputs, labels = data

                 # wrap them in Variable. Why?
                 inputs, labels = Variable(inputs), Variable(labels)

                 #zero the parameter gradients
                 optimizer.zero_grad()

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

                 #print statistics
                 running_loss += loss.data[0] # Summing up all the loss in each data instance
                 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: 2.099
[1,  4000] loss: 1.948
[1,  6000] loss: 1.941
[1,  8000] loss: 1.950
[1, 10000] loss: 1.941
[1, 12000] loss: 1.930
[2,  2000] loss: 1.957
[2,  4000] loss: 1.921
[2,  6000] loss: 1.945
```

```
[2,  8000] loss: 1.930
[2, 10000] loss: 1.933
[2, 12000] loss: 1.939
Finished Training
```

Variable v.s. **Tensor**: `Variable` is a wrapper of `Tensor`. In order to take out the Tensor contained in the Variable, just use `Variable::data`. Also, `Variable` has the gradient element of the Tensor. If we use Variable as the nodes of computational graph, the gradients can be preserved in the Variable.

`torch.optim`

Then introduce a little bit about `torch.optim`. To use an optimizer, you have to construct an optimizer object, that will hold the current state and will update the parameters based on the computed gradients.

A. Constructing it

To construct an `Optimizer` you have to give it an iterable containing the parameters (all should be `Variable` s) to optimize. Then, you can specify optimizer-specific options such as the learning rate, weight decay, etc.

Note: If you need to move a model to GPU via `.cuda()`, please do so before constructing optimizers for it. Parameters of a model after `.cuda()` will be **different** objects with those before the call. In general, you should make sure that optimized parameters live in consistent locations when optimizers are constructed and used.

```
optimizer=optim.SGD(model.parameters(),lr=0.1,momentum=0.9)
optimizer=optim.Adam([var1, var2], lr = 0.0001)
```

B. Per-parameter options

Optimizer s also support specifying per-parameter options. To do this, instead of passing an iterable of `Variable` s, pass in an iterable of `dict` s. Each of them will define a separate parameter group, and should contain a params key, containing a list of parameters belonging to it. Other keys should match the keyword arguments accepted by the optimizers, and will be used as optimization options for this group

```
optim.SGD([
             {'params': model.base.parameters()},
             {'params': model.classifier.parameters(),
              'lr': 1e-3}
          ], lr=1e-2, momentum=0.9)
```

This means that `model.base`'s parameters will use the default learning rate of `1e-2`, `model.classifier`'s parameters will use a learning rate of `1e-3`, and a momentum of `0.9` will be used for all parameters

C. Taking an optimization step

All optimizers implement a `step()` method, that updates the parameters. It can be used in two ways:

(1) `optimizer.step()`

This is a simplified version supported by most optimizers. **The function can be called once the gradients are computed using e.g.** `backward()`**.**

Example:

```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()  # <- First do backward()
    optimzer.step()  # <- Then do optimization step
```

(2) `optimizer.step(closure)`

Some optimization algorithms such as Conjugate Gradient and LBFGS need to reevaluate the function multiple times, so you have to pass in a closure that allows them to recompute your model. The closure should clear the gradients, compute the loss, and return it.

```
for input, target in dataset:
    def closure():
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        return loss
    optimizer.step(closure)
```

There are some ways to adjust the learning rate during the training. Pleaes refer to this link for more information.

1.2.4 Test the network on the test data

The network is trained for 2 passes. We will check this by predicting the class label that the neural network outputs, and checking it against the ground-truth. If the prediction is correct, we add the sample to the list of correct predictions.

```
In [12]: dataiter = iter(testloader)
         images, labels = dataiter.next()
         outputs = net(Variable(images))
         print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]]
                                         for j in range(4)))

GroundTruth:    cat  ship  ship plane


In [13]: _, predicted = torch.max(outputs.data, 1)
         print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                                       for j in range(4)))

Predicted:    cat  ship  ship  ship
```

For `torch.max`, the second parameter is `dim`. If not set, then return 1 max of the whole tensor. If set to 1, then return row-wise max. If set 2, then return column-wise max.

Let's look at how the network performs on the whole dataset.

```
In [14]: correct = 0
         total = 0
         for data in testloader:
             images, labels = data
             outputs = net(Variable(images))
             _, predicted = torch.max(outputs.data, 1)
             total += labels.size(0) # Output dimension 0
             correct += (predicted == labels).sum()

         print('Accuracy of the network on the 10000 test images: %d %%' % (
             100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 26 %
```

```
In [15]: class_correct = list(0. for i in range(10))
         class_total = list(0. for i in range(10))
         for data in testloader:
             images, labels = data
             outputs = net(Variable(images))
             _, predicted = torch.max(outputs.data, 1)
             c = (predicted == labels).squeeze()
             for i in range(4):
                 label = labels[i]
                 class_correct[label] += c[i]
                 class_total[label] += 1


         for i in range(10):
             print('Accuracy of %5s : %2d %%' % (
                 classes[i], 100 * class_correct[i] / class_total[i]))
```

```
Accuracy of plane :  3 %
Accuracy of   car : 15 %
Accuracy of  bird : 10 %
Accuracy of   cat : 17 %
Accuracy of  deer : 40 %
Accuracy of   dog : 11 %
Accuracy of  frog : 41 %
Accuracy of horse : 40 %
Accuracy of  ship : 78 %
Accuracy of truck :  4 %
```

1.2.5 Training on GPU

This will recursively go over all modules and convert their parameters and buffers to CUDA tensors:

```
In [16]: net.cuda() # This will cause an exception on macOS since it doesn't support cuda or GPU
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)

<ipython-input-16-c7e70c4f3ce7> in <module>()
----> 1 net.cuda() # This will cause an exception on macOS since it doesn't support cuda or


/home/ww8/anaconda3/lib/python3.6/site-packages/torch/nn/modules/module.py in cuda(self,
    145                 copied to that device
    146         """
--> 147         return self._apply(lambda t: t.cuda(device_id))
    148
    149     def cpu(self):


/home/ww8/anaconda3/lib/python3.6/site-packages/torch/nn/modules/module.py in _apply(sel
    116     def _apply(self, fn):
    117         for module in self.children():
--> 118             module._apply(fn)
    119
    120         for param in self._parameters.values():


/home/ww8/anaconda3/lib/python3.6/site-packages/torch/nn/modules/module.py in _apply(sel
    122                 # Variables stored in modules are graph leaves, and we don't
    123                 # want to create copy nodes, so we have to unpack the data.
--> 124                 param.data = fn(param.data)
    125                 if param._grad is not None:
    126                     param._grad.data = fn(param._grad.data)


/home/ww8/anaconda3/lib/python3.6/site-packages/torch/nn/modules/module.py in <lambda>(t
    145                 copied to that device
    146         """
--> 147         return self._apply(lambda t: t.cuda(device_id))
    148
    149     def cpu(self):


/home/ww8/anaconda3/lib/python3.6/site-packages/torch/_utils.py in _cuda(self, device, a
```

```
     64         else:
     65             new_type = getattr(torch.cuda, self.__class__.__name__)
---> 66             return new_type(self.size()).copy_(self, async)
     67
     68


/home/ww8/anaconda3/lib/python3.6/site-packages/torch/cuda/__init__.py in _lazy_new(cls,
264 @staticmethod
265 def _lazy_new(cls, *args, **kwargs):
--> 266     _lazy_init()
267     # We need this method only for lazy init, so we can remove it
268     del _CudaBase.__new__


/home/ww8/anaconda3/lib/python3.6/site-packages/torch/cuda/__init__.py in _lazy_init()
 82         raise RuntimeError(
 83             "Cannot re-initialize CUDA in forked subprocess. " + msg)
---> 84     _check_driver()
 85     torch._C._cuda_init()
 86     torch._C._cuda_sparse_init()


/home/ww8/anaconda3/lib/python3.6/site-packages/torch/cuda/__init__.py in _check_driver(
 65 Alternatively, go to: https://pytorch.org/binaries to install
 66 a PyTorch version that has been compiled with your version
---> 67 of the CUDA driver."""".format(str(torch._C._cuda_getDriverVersion())))
 68
 69


AssertionError:
The NVIDIA driver on your system is too old (found version 8000).
Please update your GPU driver by downloading and installing a new
version from the URL: http://www.nvidia.com/Download/index.aspx
Alternatively, go to: https://pytorch.org/binaries to install
a PyTorch version that has been compiled with your version
of the CUDA driver.
```

Remember that you will have to send the inputs and targets at every step to the GPU too:

```
In [ ]: inputs, labels = Variable(inputs.cuda()), Variable(labels.cuda())
```