PyTorch_NN

March 6, 2018

PyTorch Neural Networks

0.1 Define the network

Neural networks can be constructed using the `torch.nn` package.

`nn` depends on `autograd` to define models and differentiate them. An `nn.Module` contains layers, and a method `forward(input)` that returns the `output`.

A typical training procedure for a neural network is as follows:

- Define the neural network that has some learnable parameters (or weights)
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss (how far is the output from being correct)
- Propagate gradients back into the network's parameters
- Update the weights of the network, typically using a simple update rule: `weight = weight - learning_rate * gradient`

Here is an example of a simple feed-forward network. It has following layers: 1.conv, 2. subsampling, 3. conv, 4. subsampling, 5. full connection, 6. full connection, 7. Gaussian connections.

In [1]:
```
import torch
from torch.autograd import Variable
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, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

```
        def forward(self, x):
            # Max pooling over a (2, 2) window
            x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
            # If the size is a square you can only specify a single number
            x = F.max_pool2d(F.relu(self.conv2(x)), 2)
            x = x.view(-1, self.num_flat_features(x))
            x = F.relu(self.fc1(x))
            x = F.relu(self.fc2(x))
            x = self.fc3(x)
            return x

        def num_flat_features(self, x):
            size = x.size()[1:]  # all dimensions except the batch dimension
            num_features = 1
            for s in size:
                num_features *= s
            return num_features


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

```
Net (
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear (400 -> 120)
  (fc2): Linear (120 -> 84)
  (fc3): Linear (84 -> 10)
)
```

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()`.

```
In [2]: params = list(net.parameters()) # Use net.parameters() to see the parameters.
        print(len(params))
        for i in range(len(params)):
            print(params[i].size())
```

```
10
torch.Size([6, 1, 5, 5])
torch.Size([6])
torch.Size([16, 6, 5, 5])
torch.Size([16])
torch.Size([120, 400])
torch.Size([120])
```

```
torch.Size([84, 120])
torch.Size([84])
torch.Size([10, 84])
torch.Size([10])
```

```
In [3]: input = Variable(torch.randn(1, 1, 32, 32))
        out = net(input) # this object is "callable", look at explanations below.
        print(out)
```

```
Variable containing:
 0.0210  0.0804 -0.0869  0.0386  0.0267  0.0725 -0.1889 -0.0007 -0.1421  0.1407
[torch.FloatTensor of size 1x10]
```

`forward()` is a method in `nn.Module` class. Document said that it should be overriden by all subclasses.

Callable

Look at the `net(input)` statement (highlighted by a "*"). This object is *callable*. For the definition of "callable" objects, here is a short introduction from Stack Overflow. "The built-in callable (PyCallable_Check in objects.c) checks if the argument is either:

- an instance of a class with a **call** method or
- is of a type that has a non null tp_call (c struct) member which indicates callability otherwise (such as in functions, methods etc.)

The method named `__call__` is called when the instance is "called" as a function.

Since the `Net()` class is extened from `nn.Module`, each of its subclasses should override the `forward()` method. Actually, the `__call__` of `nn.Module` calls the `forward()`, which is overriden in each of the subclasses.

```
In [4]: net.zero_grad() # Zero the gradient buffers of all parameters and backprops with random
        out.backward(torch.randn(1,10)) # the argument in Variable.backward() is gradient.
```

`torch.nn` only supports **mini-batches** The entire `torch.nn` package only supports inputs that are a mini-batch of samples, and **not** a single sample. For example, `nn.Conv2d` will take in a 4D Tensor of `nSamples x nChannels x Height x Width`. If you have a single sample, just use `input.unsqueeze(0)` to add a fake batch dimension.

More Recap: - `np.Paramter`: A kind of Variable, that is automatically registered as a parameter when assigned as an attribute to a Module. - `autograd.Function`: Implements forward and backward definitions of an autograd operation. Every Variable operation, creates at least a single Function node, that connects to functions that created a Variable and encodes its history.

0.2 Loss function

A loss function takes the (output, target) pair of inputs, and computes a value that estimates how far away the output is from the target. There are several different loss functions under the nn package . A simple loss is: `nn.MSELoss` which computes the mean-squared error between the input and the target. Use *Dash* to search `torch.nn` for more information.

```
In [5]: output = net(input)
        target = Variable(torch.arange(1,11))
        print("Target Data:")
        print(target.data)
        print("Output Data:")
        print()
```

```
Target Data:

  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
[torch.FloatTensor of size 10]

Output Data:
```

```
In [6]: criterion = nn.MSELoss()
        loss = criterion(output, target)
        print(loss)
```

```
Variable containing:
 38.6209
[torch.FloatTensor of size 1]
```

```
In [7]: print(loss.grad_fn)
        print(loss.grad_fn.next_functions[0][0])
        print(loss.grad_fn.next_functions[0][0]
                          .next_functions[0][0])
```

```
<torch.autograd.function.MSELossBackward object at 0x7fce17cb5408>
<torch.autograd.function.AddmmBackward object at 0x7fce17cb5318>
<AccumulateGrad object at 0x7fce64777438>
```

0.3 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.

```
In [8]: net.zero_grad()
        print('conv1.bias.grad before backward')
        print(net.conv1.bias.grad)

conv1.bias.grad before backward
Variable containing:
 0
 0
 0
 0
 0
 0
[torch.FloatTensor of size 6]



In [9]: loss.backward()

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

conv1.bias.grad after backward
Variable containing:
 0.0847
 0.1343
-0.0375
-0.0675
 0.1161
 0.0870
[torch.FloatTensor of size 6]
```

0.4 Update the weights

The simplest update rule used in practice is the Stochastic Gradient Descent (SGD):

`weight = weight - learning_rate * gradient`

We can implement this using simple python code:

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

However, as you use neural networks, you want to use various different update rules such as SGD, Nesterov-SGD, Adam, RMSProp, etc. To enable this, we can use a small package: `torch.optim` that implements all these methods. Here's how to use it:

```
In [10]: 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
```