EE-559 – Deep learning

5.3. PyTorch optimizers

François Fleuret

https://fleuret.org/ee559/

Sat Nov 10 11:27:22 UTC 2018

idiap RESEARCH INSTITUTE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

The PyTorch module `torch.optim` provides many optimizers.

An optimizer has an internal state to keep quantities such as moving averages, and operates on an iterator over `Parameter`s.

- Values specific to the optimizer can be specified to its constructor, and
- its `step` method updates the internal state according to the `grad` attributes of the `Parameter`s, and updates the latter according to the internal state.

We implemented the standard SGD as follows

```
for e in range(nb_epochs):
    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input[b:b+batch_size])
        loss = criterion(output, train_target[b:b+batch_size])
        model.zero_grad()
        loss.backward()
        with torch.no_grad():
            for p in model.parameters(): p -= eta * p.grad
```

We implemented the standard SGD as follows

```
for e in range(nb_epochs):
    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input[b:b+batch_size])
        loss = criterion(output, train_target[b:b+batch_size])
        model.zero_grad()
        loss.backward()
        with torch.no_grad():
            for p in model.parameters(): p -= eta * p.grad
```

which can be re-written with the `torch.optim` package as

```
optimizer = torch.optim.SGD(model.parameters(), lr = eta)

for e in range(nb_epochs):
    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input[b:b+batch_size])
        loss = criterion(output, train_target[b:b+batch_size])
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

We have at our disposal many variants of the SGD:

- `torch.optim.SGD` (momentum, and Nesterov's algorithm),
- `torch.optim.Adam`
- `torch.optim.Adadelta`
- `torch.optim.Adagrad`
- `torch.optim.RMSprop`
- `torch.optim.LBFGS`
- ...

An optimizer can also operate on several iterators, each corresponding to a group of `Parameter`s that should be handled similarly. For instance, different layers may have different learning rates or momentums.

So to use Adam, with its default setting, we just have to replace in our example

```
optimizer = optim.SGD(model.parameters(), lr = eta)
```

with

```
optimizer = optim.Adam(model.parameters(), lr = eta)
```

So to use Adam, with its default setting, we just have to replace in our example

```
optimizer = optim.SGD(model.parameters(), lr = eta)
```

with

```
optimizer = optim.Adam(model.parameters(), lr = eta)
```

The learning rate may have to be different if the functional was not properly scaled.

An example putting all this together

We now have the tools to build and train a deep network:

- fully connected layers,
- convolutional layers,
- pooling layers,
- ReLU.

And we have the tools to optimize it:

- Loss,
- back-propagation,
- stochastic gradient descent.

The only piece missing is the policy to initialize the parameters.

We now have the tools to build and train a deep network:

- fully connected layers,
- convolutional layers,
- pooling layers,
- ReLU.

And we have the tools to optimize it:

- Loss,
- back-propagation,
- stochastic gradient descent.

The only piece missing is the policy to initialize the parameters.

PyTorch initializes parameters with default rules when modules are created. They normalize weights according to the layer sizes (Glorot and Bengio, 2010) and behave usually very well. We will come back to this.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size = 5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size = 5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), kernel_size = 3))
        x = F.relu(F.max_pool2d(self.conv2(x), kernel_size = 2))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
train_set = torchvision.datasets.MNIST('./data/mnist/',
                                       train = True, download = True)
train_input = train_set.train_data.view(-1, 1, 28, 28).float()
train_target = train_set.train_labels

lr, nb_epochs, batch_size = 1e-1, 10, 100

model = Net()

optimizer = torch.optim.SGD(model.parameters(), lr = lr)
criterion = nn.CrossEntropyLoss()

model.to(device)
criterion.to(device)
train_input, train_target = train_input.to(device), train_target.to(device)

mu, std = train_input.mean(), train_input.std()
train_input.sub_(mu).div_(std)

for e in range(nb_epochs):
    for input, target in zip(train_input.split(batch_size),
                             train_target.split(batch_size)):
        output = model(input)
        loss = criterion(output, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

The end

References

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.