Autograd & Modules

Pierre Stock

Deep Learning DIY

ENS ULM
• **Autograd**
  o Autograd in a Nutshell
  o Visualizing the Computation Graph

• **Modules**
  o Build your own module
  o Forward and Backward Hooks
  o Buffers
  o Extending Autograd & Grad-checking
  o CUDA API, float API, train/eval API

• **Autograd Tips & Tricks**
  o Execution Time
  o Pointers & Nodes
Refresher – Training Loop

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

# define network structure
net = nn.Sequential(nn.Linear(3 * 32 * 32, 1000), nn.ReLU(), nn.Linear(1000, 10))
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr = 0.01, momentum=0.9, weight_decay=1e-4)

# load data
train_set = torchvision.datasets.CIFAR10(root='.', train=True, transform=transform_list)
train_loader = torch.utils.data.DataLoader(train_set, batch_size=64)

# training loop
for (batch, target) in train_loader:
    output = net(batch)
    loss = criterion(output, targets)

    optimizer.zero_grad()
    loss.backward()  # Magic! Autograd
    optimizer.step()
```

facebook Artificial Intelligence Research
Autograd in a Nutshell

• Autograd = Automatic Differentiation
• Idea: use the chain rule on a graph of operations with leaves and nodes

• Dynamic vs. Static graph
• Computation graph is created during each forward call

• Back in the time (until v4.0), use of Variables encapsulating Tensors
• Now only Tensors with data and grad attributes
Hands on!

- Autograd handles well almost every basic tensor operation you could think of!

```python
# don't hit enter before to guessed the answer!
x = torch.Tensor(4, 10)
x.requires_grad=True
loss = x[:, :4].sum()
loss.backward()
x.grad
```
Hands on!

- Autograd handles well almost every basic tensor operation you could think of!

```python
# don't hit enter before to guessed the answer!
x = torch.Tensor(4, 10)
x.requires_grad=True
loss = x[:, :4].sum()
loss.backward()
x.grad
```

```
tensor([[1., 1., 1., 1., 0., 0., 0., 0., 0., 0.],
        [1., 1., 1., 1., 0., 0., 0., 0., 0., 0.],
        [1., 1., 1., 1., 0., 0., 0., 0., 0., 0.],
        [1., 1., 1., 1., 0., 0., 0., 0., 0., 0.]])
```
Hands on!

- Autograd handles well almost every basic tensor operation you could think of!

```python
# don't hit enter before to guessed the answer!
x = torch.Tensor([2, 3])
x.requires_grad=True
y = torch.Tensor([[1, 2], [3, 4]])
loss = y.mm(x).sum()
loss.backward()
x.grad
```
Hands on!

- Autograd handles well almost every basic tensor operation you could think of!

```python
# don't hit enter before to guessed the answer!
x = torch.Tensor([2, 3])
x.requires_grad=True
y = torch.Tensor([[1, 2], [3, 4]])
loss = y.mm(x).sum()
loss.backward()
x.grad

tensor([[4., 4., 4.],
        [6., 6., 6.]])
```
Hands on!

- Autograd handles well almost every basic tensor operation you could think of!
- **Bonus exercise**: what about second order gradients?

```python
# don't hit enter before to guessed the answer!
x = torch.Tensor([[1, 2, 3, 4]])
x.requires_grad=True
y = 2 * x
y.backward(torch.Tensor([[1, 0, 0, 0]]))
x.grad
```
Hands on!

- Autograd handles well almost every basic tensor operation you could think of!
- **Bonus exercise**: what about second order gradients?

```python
# don't hit enter before to guessed the answer!
x = torch.Tensor([[1, 2, 3, 4]])
x.requires_grad=True
y = 2 * x
y.backward(torch.Tensor([[1, 0, 0, 0]]))
x.grad
tensor([[2., 0., 0., 0.]])
```
Visualizing the Computation Graph

- Autograd handles well almost every basic tensor operation you could think of!
- **Bonus exercise**: clone and test [https://github.com/szagoruyko/pytorchviz](https://github.com/szagoruyko/pytorchviz)
Volatile mode

- Volatile mode, why does it exist?

```python
net = nn.Linear(10, 5)
print('Data: ', net.weight.data)  # <--- some data
print('Grad: ', net.weight.grad)  # <--- None

# normal mode
x = torch.rand(2, 10)
y = net(x).sum()
y.backward()
print('Data: ', net.weight.data)  # <--- some data
print('Grad: ', net.weight.grad)  # <--- some data

# volatile mode
with torch.no_grad():
    x = torch.rand(2, 10)
y = net(x).sum()
y.backward()
print('Data: ', net.weight.data)  # <--- some data
print('Grad: ', net.weight.grad)  # <--- this will raise an error
```
Build your own module

class Linear(nn.Module):
    ""
    This is a docstring.
    This must be filled by you.
    This is important.
    ""

def __init__(self, in_features, out_features):
    super(Linear, self).__init__()
    self.in_features = in_features
    self.out_features = out_features
    self.weight = Parameter(torch.Tensor(out_features, in_features))
    self.reset_parameters()

def reset_parameters(self):
    stdv = 1. / math.sqrt(self.weight.size(1))
    torch.nn.init.uniform_(self.weight.data, -stdv, stdv)

def forward(self, input):
    return F.linear(input, self.weight, self.bias)

def extra_repr(self):
    return 'in_features={}, out_features={}, bias={} is not None'.format(
        self.in_features, self.out_features, self.bias)
Build your own module

• Permute the features of an input (input = batch x features).
• How to test it?
• Wait, is this module even useful?
• **Bonus exercise:** how do you write the backprop by hand?

```python
class Permutation(nn.Module):
    def __init__(self, input_features, axis=1):
        super(Permutation, self).__init__()
        self.input_features = input_features
        self.axis = axis
        self.perm = # ... (use torch.randperm)

    def forward(self, input):
        return # ... (use torch.index_select(axis, perm))
```
Forward and Backward Hooks

- Print/plot the activations norms in a “deep” FC network. What do you observe?
- **Bonus exercise**: how about storing the activation norms when you want during training?

```python
def deep_net(n_layers, features=1000):
    # returns a FC network with n_layers of size features
    # use: nn.Sequential(*layers)

def forward_hook(module, input, output):
    # provides, for module, input and output activations
    # use: .norm()

def register_forward_hooks(net, forward_hook):
    # register hooks for every layer in net
    # use: net.children() to get the layers
    # use: layer.register_forward_hook(forward_hook) for every layer
    # ...
```
Forward and Backward Hooks

- Importance of a good normalization!
- Vanishing/exploding gradients (search the web for it...)

![Graph showing the evolution of activations, no BatchNorm, ReLU](image)

**Bad**
Forward and Backward Hooks

- Importance of a good normalization!
- Vanishing/exploding gradients (search the web for it...)
Forward and Backward Hooks

- Print the backpropagated gradient sizes
- Does this match with the gradients wrt the weights?
- **Bonus exercise:** use grad_output and input to recover weight.grad

```python
# print sizes
def backward_hook(module, grad_input, grad_output):
    # use: .size()

# register hook for every layer
def register_backward_hooks(net, backward_hook):
    # guess how to register a backward hook
    # ...
```
Buffers

- You want a stateful part of your model that is not a parameter, appears in state_dict()
- Normalize each batch channel by its current mean
- Accumulate an exponential moving average of the means over the iterations (why?)
- How to test it?
- **Bonus exercise**: what about dividing by the std? Numerical stability?

```python
class Normalize(nn.Module):
    def __init__(self, num_features, momentum=0.1):
        # you know what to write here
        # use: self.register_buffer('running_mean, torch.Tensor(...))
        #

    def reset_parameters(self):
        # ...
```
Buffers

- You want a stateful part of your model that is not a parameter, appears in state_dict()
- Normalize each batch channel by its current mean
- Accumulate an exponential moving average of the means over the iterations (why?)
- How to test it?
- **Bonus exercise**: what about dividing by the std? Numerical stability?

```python
def forward(self, input):
    # training mode: use batch statistics and update running statistics
    if self.training:
        # ...

    # eval mode: use running statistics
    else:
        #...

    # return output
    return #...
```
CUDA API

- `cuda()`, why, where to use it? Don’t forget the `cpu()`

```python
# define network structure
net = nn.Sequential(nn.Linear(3 * 32 * 32, 1000), nn.ReLU(), nn.Linear(1000, 10)).cuda()
criterion = nn.CrossEntropyLoss().cuda()
optimizer = torch.optim.SGD(net.parameters(), lr = 0.01, momentum=0.9, weight_decay=1e-4)

# load data
train_set = torchvision.datasets.CIFAR10(root='.', train=True, transform=transform_list)
train_loader = torch.utils.data.DataLoader(train_set, batch_size=64)

# training loop
for (batch, target) in train_loader:
    batch = batch.cuda()
    target = target.cuda()

    output = net(batch)
    loss = criterion(output, target)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```
Float API

- `double()`, `half()` why, where to use it?

```python
# define network structure
net = nn.Sequential(nn.Linear(3 * 32 * 32, 1000), nn.ReLU(), nn.Linear(1000, 10)).half()
criterion = nn.CrossEntropyLoss().half()
optimizer = torch.optim.SGD(net.parameters(), lr = 0.01, momentum=0.9, weight_decay=1e-4)

# load data
train_set = torchvision.datasets.CIFAR10(root='.', train=True, transform=transform_list)
train_loader = torch.utils.data.DataLoader(train_set, batch_size=64)

# training loop
for (batch, target) in train_loader:
    batch = batch.half()
    target = target.half()

    output = net(batch)
    loss = criterion(output, targets)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```
Other important APIs

- APIsTrain/eval mode. Why is it useful?

```python
net = nn.Linear(2, 2)

print(net.training)  # True, default behavior
net.eval()           # eval mode
print(net.training)  # False
```

- Dataparallel

```python
net = torch.nn.DataParallel(nn.Linear(3, 3))
```
Gradchecking

- Idea: finite differences to check the analytic gradient computed by autograd / defined by you
- Why gradcheck when I rely on autograd?

```python
x = torch.rand(256, 2, requires_grad=True).double()
y = torch.randint(0, 10, (256, ), requires_grad=True).double()
custom_op = nn.Linear(2, 10).double()
res = torch.autograd.gradcheck(custom_op, (x, ))
print(res)
```
Autograd tips & Tricks

- Pointers are everywhere. Don’t forget to clone()

```python
net = nn.Linear(2, 2)
w = net.weight.clone()
print(w)
x = torch.rand(1, 2)
y = net(x).sum()
y.backward()
net.weight.data += 0.01 * net.weight.grad  # <--- What is this?
p
```
Autograd tips & Tricks

- Weight sharing. Why use it?

```python
net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
net[0].weight = net[1].weight  # weight sharing

x = torch.rand(1, 2)
y = net(x).sum()
y.backward()
print(net[0].weight.grad)
print(net[1].weight.grad)
```
Autograd tips & Tricks

• Execution time for complicated layers.
Useful Ressources

- The Notebook on the course website: https://www.di.ens.fr/~lelarge/dldiy/
- PyTorch repo: https://github.com/pytorch/pytorch. Explore it! (typing ‘t’ will prompt a search by filename), in particular torch.nn, torch.optim.
- More about Autograd: https://openreview.net/pdf?id=BJJsrmfCZ
- PyTorch Tutorials: https://pytorch.org/tutorials/