

Deep Learning with PyTorch



Outline



Popular Deep Learning Frameworks

- Imperative-style programs execute the computation **immediately**.

```
import numpy as np
a = np.ones(10)
b = np.ones(10) * 2
c = b * a
d = c + 1
```

- Symbolic-style programs trace the computation **graph** first, and then execute it later.

```
A = Variable('A')
B = Variable('B')
C = B * A
D = C + Constant(1)
# compiles the function
f = compile(D)
d = f(A=np.ones(10), B=np.ones(10)*2)
```

PyTorch: Levels of Abstraction

- **Tensor:**
 - A multidimensional high-performance array.
 - Can run on GPU.
 - Same behavior as numpy ndarrays for the most part.
- **Autograd:**
 - A tensor also acts as a node in a computation graph.
 - Tensor $C = A + B$ retains information about how it was created.
 - *Autograd* uses this information to compute gradients automatically.
- **Module:**
 - Generic neural network building blocks.
 - Linear layer, convolutional layer, etc.

PyTorch: Tensors

```
x = torch.rand(5, 3)  
print(x)
```

PyTorch: Tensors

```
tensor([[0.5683, 0.1334, 0.7669],  
        [0.8196, 0.1895, 0.6653],  
        [0.3621, 0.8314, 0.6239],  
        [0.8716, 0.2269, 0.1005],  
        [0.3598, 0.1292, 0.5585]])
```

PyTorch: Tensors

- Acts as a multi-dimensional array.
- Ex:
 - `[[0, 1, 2], [3, 4, 5]]`
 - `torch.tensor([[0, 1, 2], [3, 4, 5]])`
 - Represent the same data.

PyTorch: Autograd

- All tensors track their computation paths and compute their own gradients!

```
x = torch.ones(2, 2, requires_grad=True)
print(x)
```


PyTorch: Autograd

- All tensors track their computation paths and compute their own gradients!

```
y = x + 2  
print(y)
```

PyTorch: Autograd

- All tensors track their computation paths and compute their own gradients!

```
print(y.grad_fn)
```

```
<AddBackward0 object at 0x7f429cdbd588>
```

PyTorch: Autograd

Numpy

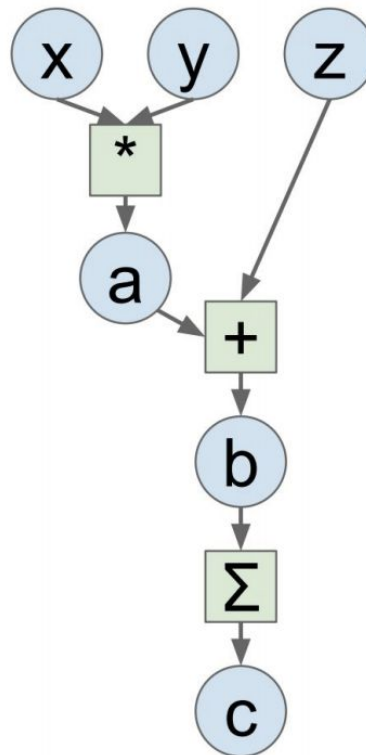
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch: Autograd

- `c.backward()`
- `x.grad`, `y.grad`, `z.grad` all automatically computed!

PyTorch: Modules

- Neural network layer class.
- Neural networks are composed functions:
 - $a = f(x)$
 - $b = g(a)$
 - $c = h(b)$
 - $c = h(g(f(x)))$
- Modules are classes that define one of these composable functions.

PyTorch: Modules

– Convolution layers

- Conv1d
- Conv2d
- Conv3d
- ConvTranspose1d
- ConvTranspose2d
- ConvTranspose3d
- Unfold
- Fold

– Pooling layers

- MaxPool1d
- MaxPool2d
- MaxPool3d
- MaxUnpool1d
- MaxUnpool2d
- MaxUnpool3d
- AvgPool1d
- AvgPool2d
- AvgPool3d
- FractionalMaxPool2d
- LPPool1d
- LPPool2d
- AdaptiveMaxPool1d
- AdaptiveMaxPool2d
- AdaptiveMaxPool3d
- AdaptiveAvgPool1d
- AdaptiveAvgPool2d
- AdaptiveAvgPool3d

– Non-linear activations (weighted sum, nonlinearity)

- ELU
- Hardshrink
- Hardtanh
- LeakyReLU
- LogSigmoid
- MultiheadAttention
- PReLU
- ReLU
- ReLU6
- RReLU
- SELU
- CELU
- Sigmoid
- Softplus
- Softshrink
- Softsign
- Tanh
- Tanhshrink
- Threshold

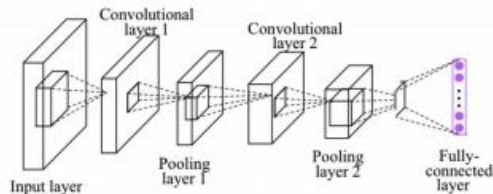
– Recurrent layers

- RNN
- LSTM
- GRU
- RNNCell
- LSTMCell
- GRUCell

– Loss functions

- L1Loss
- MSELoss
- CrossEntropyLoss
- CTCLoss
- NLLLoss
- PoissonNLLLoss
- KLDivLoss
- BCELoss
- BCEWithLogitsLoss
- MarginRankingLoss
- HingeEmbeddingLoss
- MultiLabelMarginLoss
- SmoothL1Loss
- SoftMarginLoss
- MultiLabelSoftMarginLoss
- CosineEmbeddingLoss
- MultiMarginLoss
- TripletMarginLoss

PyTorch: Modules



```
class Net(nn.Module):  
  
    def __init__(self):  
        super(Net, self).__init__()  
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)  
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)  
        self.mp = nn.MaxPool2d(2)  
        self.fc = nn.Linear(320, 10) # 320 -> 10  
  
    def forward(self, x):  
        in_size = x.size(0)  
        x = F.relu(self.mp(self.conv1(x)))  
        x = F.relu(self.mp(self.conv2(x)))  
        x = x.view(in_size, -1) # flatten the tensor  
        x = self.fc(x)  
        return F.log_softmax(x)
```

PyTorch: Modules

