PyTorch Tutorial for Beginner

CSE446

Department of Computer Science & Engineering
University of Washington

February 2018

PyTorch

- Python package for machine learning, backed by Facebook.
- **Documentation:** `http://pytorch.org/docs/0.3.1/`
- **Repository:** `https://github.com/pytorch/pytorch`
- **Examples (very nice):** `https://github.com/pytorch/examples`
- Used in the next homework

TensorFlow vs. PyTorch

- Biggest difference: Static vs. dynamic computation graphs
- Creating a static graph beforehand is unnecessary
- *Reverse-mode auto-diff* implies a computation graph
  - PyTorch takes advantage of this
  - **We use PyTorch.**

See the Difference: Linear Regression

Tensorflow: Create optimizer before feeding data

```
... # Create placeholders X, Y and variables W, b
# Construct linear model and specify cost function
Yhat = tf.add(tf.multiply(X, W), b)
cost = tf.reduce_sum(tf.pow(Yhat-Y, 2))/(2*n_samples)
optimizer =
    tf.train.GradientDescentOptimizer(learning_rate)
            .minimize(cost)
...
# Start training
with tf.Session() as sess:
    ...
    # Fit all training data
    for epoch in range(training_epochs):
        for (x, y) in zip(train_X, train_Y):
            sess.run(optimizer, feed_dict={X: x, Y: y})
```

From TensorFlow-Examples

See the Difference: Linear Regression

PyTorch: Create optimizer while feeding data

```
# Define linear regression model (a function)
Yhat = torch.nn.Linear(W.size(0), 1)
for epoch in range(training_epochs):
    batch_x, batch_y = get_batch() # Get data
    Yhat.zero_grad() # Reset gradients
    # Forward pass
    output = F.mse_loss(Yhat(batch_x), batch_y)
    loss = output.data[0]
    output.backward()  # Backward pass

    # Apply gradients
    for param in fc.parameters():
        param.data.add_(-learning_rate * param.grad.data)
```

From pytorch/examples

Even Better

PyTorch: Create optimizer while feeding data

```
import torch.optim as optim
# Define linear regression model (a function)
Yhat = torch.nn.Linear(W.size(0), 1)
opt = optim.SGD(my_model.parameters(), lr=learning_rate)
for epoch in range(training_epochs):
    batch_x, batch_y = get_batch() # Get data
    Yhat.zero_grad() # Reset gradients
    # Forward pass
    output = F.mse_loss(Yhat(batch_x), batch_y)
    loss = output.data[0]
    output.backward()  # Backward pass

    # Updates parameters!
    opt.step()
```

From pytorch/examples

Essentials

```python
import torch.nn.functional as F
import torch.nn as nn
import torch.nn.init as init
```

Building Block: Tensor

- Multi-dimensional matrix. (Float/Byte/Long)
- Can initialize from and convert to numpy arrays.

```
# torch.Tensor = torch.FloatTensor
t1 = torch.Tensor(4, 6)
t1.size() # Returns torch.Size([4, 6])
t2 = torch.Tensor([3.2, 4.3, 5.5])
t3 = torch.Tensor(np.array([[3.2], [4.3], [5.5]]))
t4 = torch.rand(4, 6)
t5 = t1 + t2 # addition
t6 = t2 * t3 # entry-wise product
t7 = t2 @ t3 # matrix multiplication
t8 = t1.view(2,12) # reshapes t1 to be 2 by 12
t8 = t1.view(2,-1) # same as above
t9 = t1[:, -1] # last column from the left
```

- Broadcasting is present, but use with caution.

Building Block: Variables and Autograd I

```
# Variable is in the autograd module
from torch.autograd import Variable

# Variables wrap tensors
x = Variable(torch.Tensor([1, 2, 3]), requires_grad=True)
# You can access the underlying tensor with the .data
    attribute
print(x.data)

# Any operation you could use on Tensors, you can use
# on Variables. Operations between Variables produce
# Variables.
y = Variable(torch.Tensor([4, 5, 6]), requires_grad=True)
z = x + y
print(z.data)

# z knows its function to compute gradient.
print(z.grad_fn) # Magic of auto differentiation.
```

Building Block: Variables and Autograd II

```
z_sum = torch.sum(z)
```

This gives $z_{sum} = x_0 + y_0 + x_1 + y_1 + x_2 + y_2$. Now, if we want to compute

$$\frac{\partial z_{sum}}{\partial x_0}$$

```
z_sum.backward()
print(x.grad) # x.grad is a Variable(Tensor([1 1 1]))
```

Acknowledgement

Thanks Benjamin Evans for code examples and Kaiyu Zheng for drafting the slides.