Machine Learning, Spring 2020

Pytorch Part I

Python tutorial: http://learnpython.org/

TensorFlow tutorial: https://www.tensorflow.org/tutorials/

PyTorch tutorial: https://pytorch.org/tutorials/

Acknowledge: The slides are partially referred to coursera online machine learning course by Prof. Andrew Ng, and NYU machine learning course. All copyrights owned by original authors. (i.e. http://u.cs.biu.ac.il/~89-570/)

Overview

- Deep learning frameworks
- Computational graph
- Pipeline of projects in Pytorch (Autograd)
- Pytorch – Jupyter Notebook
  - Tensor
  - Variable
  - Module

Deep learning frameworks

Previous. → . Current

- Caffe. → . Caffe2
  UCB Facebook
- T
  NY
- Theano. → . TensorFlow
  U Montreal Google

PyTorch

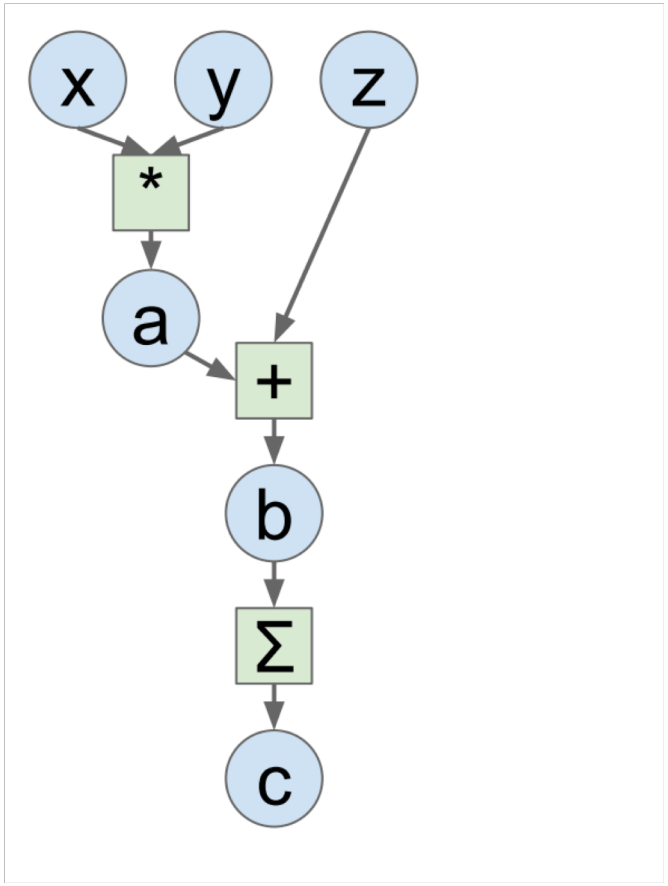
- A Python ic scientific computing package targeted at two sets of audiences:
  - A replacement for NumPy to use the power of GPUs
  - A deep learning research platform that provides maximum flexibility and speed

Overview

- Deep learning frameworks
- Computational graph
- Pipeline of networks in Pytorch
- Pytorch – Jupyter Notebook
  - Tensor
  - Variable
  - Module

Computational Graphs

A **computational graph** is a directed **graph** where the nodes correspond to operations or variables.



$$\Sigma\ x*y+z$$

$$a = x*y$$

$$b = a+z$$

$$c = \Sigma b$$

Computational Graphs

Ideal traits of a DNN framework

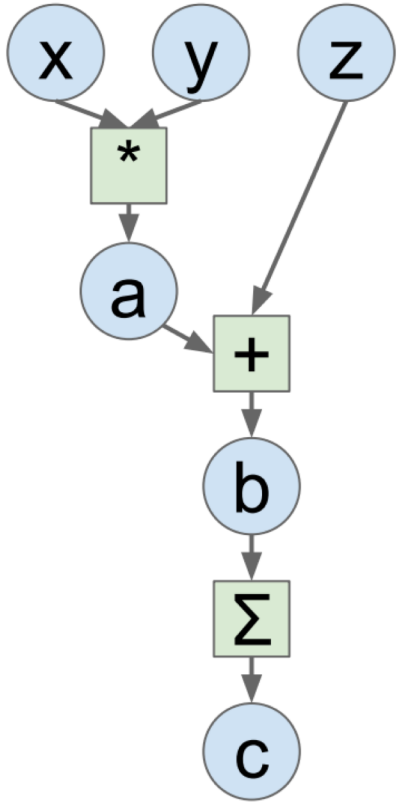
- Easily build big computational graphs
- Easily compute gradients in computational graphs
- Run it efficiently on GPU

Deep learning frameworks

Computational Graphs

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
```



Numpy:

- Can not run on GPU
- Have to compute manual-defined gradients

Computational Graphs

Numpy

```python
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
```

TensorFlow

```python
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Static graph

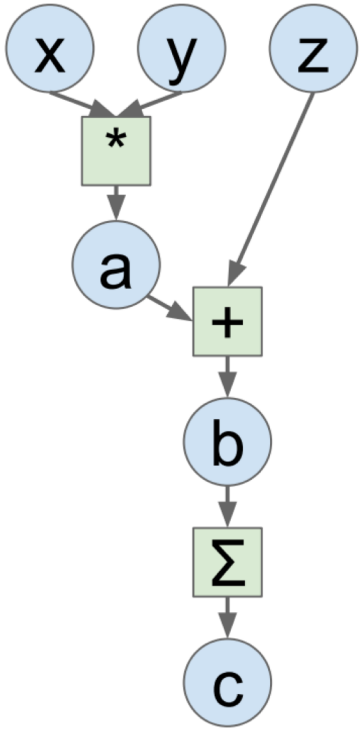
PyTorch

```python
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
y = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
z = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)

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

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

Dynamic graph

Pytorch

Computational Graphs



Run on GPU by casting to .cuda()

Call backward() to compute gradients

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
y = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
z = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)

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

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

Overview

- Deep learning frameworks
- Computational graph
- Pipeline of project in Pytorch (Autograd)
- Pytorch – Jupyter Notebook
  - Tensor
  - Variable
  - Module

Pipeline of networks in Pytorch

- General pipeline of a project:
  - Create variables
    - inputs
  - Establish networks (functions with variables)
    - outputs
  - Define loss
    - loss = targets-outputs
  - Calculate gradient to backpropagate loss
    - gradient
  - Update weights

Done by Pytorch

Simple example

Input x (matrix)
Target y (matrix)
Aims: find a function f mapping x to y
y = f(x) ←y^*=net(x)

```
from torch.autograd import Variable
```

```
x = Variable(torch.Tensor([10]),requires_grad=True)
y = Variable(torch.Tensor([5]),requires_grad=True)
z=x*y*5
z.backward()
print(x.grad)
print(y.grad)
```

```
tensor([25.])
tensor([50.])
```

Create variables

```
N, D_in, H, D_out = 5, 100, 15, 10
x = Variable(torch.rand(N, D_in), requires_grad = False)
y = torch.randn((N, D_out), requires_grad = False)
w1 = Variable(torch.randn(D_in, H), requires_grad = True)
w2 = Variable(torch.randn(H, D_out), requires_grad = True)
```

```
z = 2*x
w3 = 2*w1
print(z.grad_fn)
print(w3.grad_fn)
```

```
None
<MulBackward0 object at 0x11e214278>
```

Establish functions

Define loss

Propagate loss

Update weights

```
learning_rate = 1e-4
for t in range(100):
    y_pred = x.mm(w1).clamp(min=0).mm(w2) # torch.mm() per
                                          # torch.clamp cl
    loss = (y_pred - y).pow(2).sum()
    if w1.grad.sum(): w1.grad.data.zero_()
    if w1.grad.sum(): w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data #.data pick th
    w2.data -= learning_rate * w2.grad.data
```

Another example with CNN

- Aim: regress a unknown function *f: X→Y*
- Input: X (a 4D matrix with [1,1,32,32])

```
tensor([[[[-1.8048,  0.0721,  0.6041,  ...,  1.1225, -0.5207,  1.1020],
          [ 0.2645,  0.1578,  2.5684,  ..., -0.0643,  0.4185,  0.7173],
          [ 0.6431,  0.8453, -0.9664,  ..., -1.2289,  1.9743,  0.1737],
          ...,
          [ 0.8599, -0.5783,  1.4800,  ..., -0.0817, -0.6023, -0.0265],
          [-2.0356, -0.5759, -1.8809,  ...,  0.0890,  0.5400, -0.2750],
          [-0.5475,  0.7634,  0.0718,  ..., -0.3680, -1.6368,  0.8780]]]])
```

- Target: Y (a vector [1,10])

```
tensor([-1.4679,  0.6938,  1.0870,  0.7209,  0.0471, -0.8229,  0.3634,  0.3670,
        -2.0502,  1.1870])
```

- Goals: using a neural network *Net(·|θ)* to approximate *f(·)*: *Net(·|θ)* → *f(·)*

Neural Network Example

- Input: X (a 4D matrix with [1,1,32,32])
- Target: Y (a vector [1,10])
- Prediction: $Y^* = Net(X|\boldsymbol{\Theta})$
- Loss: $mean\_square\_error = (Y^*-Y)^2$

Create variables

Establish functions

Define loss

To minimize the MSE

- *Calculate the gradient to propagate loss to all parameters* $\theta_i \in \boldsymbol{\Theta}$ *by*

$$grad_{\boldsymbol{\Theta}} = \partial MSE / \partial \boldsymbol{\Theta}$$

Propagate loss

- Update *parameters* $\theta_i \in \boldsymbol{\Theta}$ *by*

$$\boldsymbol{\Theta}^{(n+1)} \leftarrow \boldsymbol{\Theta}^{(n)} - r \times grad_{\boldsymbol{\Theta}}$$

Update weights

Neural Network Example

- Input: *X* (a 4D matrix with [1,1,32,32])
- Target: *Y* (a vector [1,10])
- Loss*: mean_square_error* = $(Y^*-Y)^2$

Create variables

Define loss

```
# input defination
input = torch.randn(1,1,32,32)
output = net(input)
```

```
# label and loss defination
target = torch.randn(10)
print(input)
print(target)
target = target.view(1,-1)
criterion = nn.MSELoss()
loss = criterion(output, target)
```

Create variables

Define loss

Neural Network Example

- Prediction: $Y^* = Net(X|\Theta)$

Establish functions

```
class Net(nn.Module):
    def __init__(self):
        # 1 input image channel, 6 output channels, 5x5 square convolution
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1,6,5) # input of nn.Conv2d is a 4D tensor of nSamples x nCha
        self.conv2 = nn.Conv2d(6,16,5)
        self.fc1 = nn.Linear(16*5*5,120)
        self.fc2 = nn.Linear(120,84)
        self.fc3 = nn.Linear(84,10)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)),(2,2))
        x = F.max_pool2d(F.relu(self.conv2(x)),(2,2))
        x = x.view(-1, self.num_flat_features(x)) # x.view(x.numel()) has the same function
        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:] # do not want to count batch_size
        num_features = 1
        for s in size:
            num_features *= s
        return num_features

net = Net()
print(net)
params = list(net.parameters())
```

Establish functions

Neural Network Example

Module

To minimize the MSE

- *Calculate the gradient to propagate loss to all parameters* $\theta_i \in \Theta$ *by*

$$grad_{\Theta} = \partial MSE / \partial \Theta$$

Propagate loss

- Update *parameters* $\theta_i \in \Theta$ *by*

$$\Theta^{(n+1)} \leftarrow \Theta^{(n)} - r \times grad_{\Theta}$$

Update weights

```
# backprop and weights updates
net.zero_grad()
loss.backward()
print(loss.grad_fn)

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

Propagate loss

Update weights

Neural Network Example

To minimize the MSE

- *Calculate the gradient to propagate loss to all parameters* $\theta_i \in \Theta$ *by*

$$grad_{\Theta} = \partial MSE / \partial \Theta$$

Propagate loss

- Update *parameters* $\theta_i \in \Theta$ *by*

$$\Theta^{(n+1)} \leftarrow \Theta^{(n)} - r \times grad_{\Theta}$$

Update weights

```
import torch.optim as optim

# create SGD optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```

Propagate loss

Update weights

Overview

- Deep learning frameworks
- Computational graph
- Pipeline of projects in Pytorch
- Pytorch – Jupyter Notebook
  - Tensor
  - Variable
  - Module

Pytorch

- Three levels of abstraction (will introduced in codes)
  - Tensor: Imperative nparray but runs on GPU
  - Variable: Node in a computational graph; stores data and gradient
  - Module: A neural network layer; may store state or learnable weights

Reference

- **Pytorch official tutorial: Why Pytorch? (link: https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py/)**
- **Lecture 8: Deep Learning Software of Stanford CS231n (link: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture8.pdf/)**
- **Pytorch documentation (link: https://pytorch.org/docs/0.4.0/)**