CS 391R PyTorch Tutorial

Zhenyu Jiang

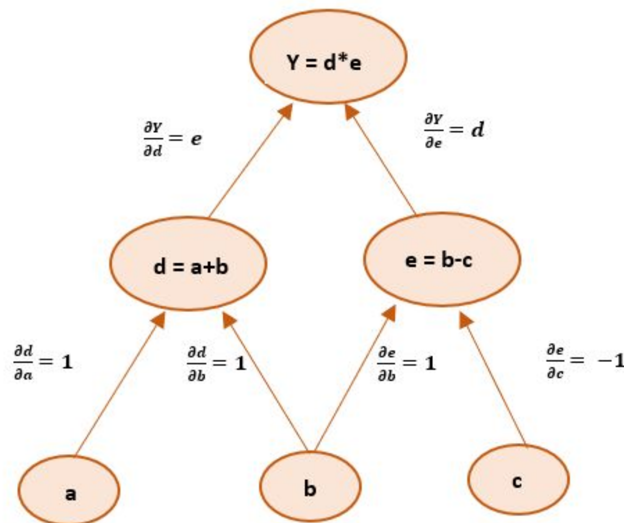
September 29

Disclaimer: adopted from gatech tutorial and pytorch tutorial

Why do we need deep learning libraries?

Autograd

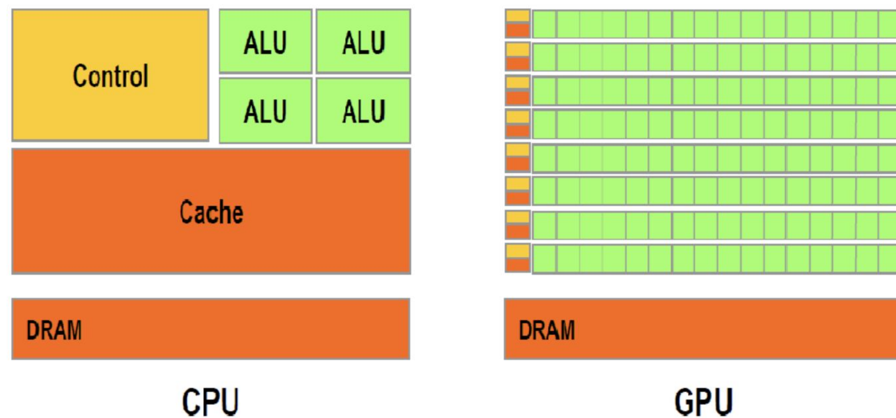
- gradient-based methods are used to optimize deep neural networks.
- back propagation is the core of gradient computation
- we use a computation graph to record all the operation during forward, and use chain rule to compute gradient backward.



Why do we need deep learning libraries?

GPU acceleration

- GPU is well suited for deep learning because of
  - high bandwidth main memory
  - hiding memory access latency under thread parallelism
  - large and fast register and L1 memory



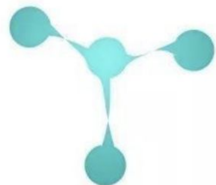
https://www.quora.com/Why-are-GPUs-well-suited-to-deep-learning

Deep Learning Libraries

Microsoft CNTK

Caffe

Caffe2



PYTORCH

Chainer

K Keras

TensorFlow

theano

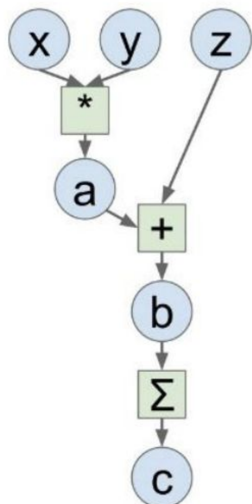
dy/net

mxnet

GLUON

Why PyTorch

Computation Graph



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
```

PyTorch

```python
import torch

N, D = 3, 4

x = torch.rand((N, D),requires_grad=True)
y = torch.rand((N, D),requires_grad=True)
z = torch.rand((N, D),requires_grad=True)

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

c.backward()
```

Tensors

Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

Common operations for creation and manipulation of these Tensors are similar to those for ndarrays in NumPy. (rand, ones, zeros, indexing, slicing, reshape, transpose, cross product, matrix product, element wise multiplication)

Initialize a tensor

Directly from data

```python
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

From numpy array

```python
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

From other tensors

```python
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")
```

Attributes of a tensor

Tensor attributes describe their shape, datatype, and the device on which they are stored.

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Operation on tensors

Moving tensor between devices

```
# We move our tensor to the GPU if available
if torch.cuda.is_available():
    tensor = tensor.to('cuda')
```

Standard numpy-like indexing and slicing ...

Bridge between numpy

```
n = np.ones(5)
t = torch.from_numpy(n)
```

```
t = torch.ones(5)
n = t.numpy()
```

Example

1. Data: Datasets & DataLoaders
2. Preprocess: Transforms
3. Neural Network: nn.Module
4. Loss function
5. Optimization: loss.backward(), optimizer.step()
6. Save & Load Model: torch.save(), torch.load()