Pytorch Basic

2021.10.14

LG

KAIST AI
Graduate School of AI

Contents

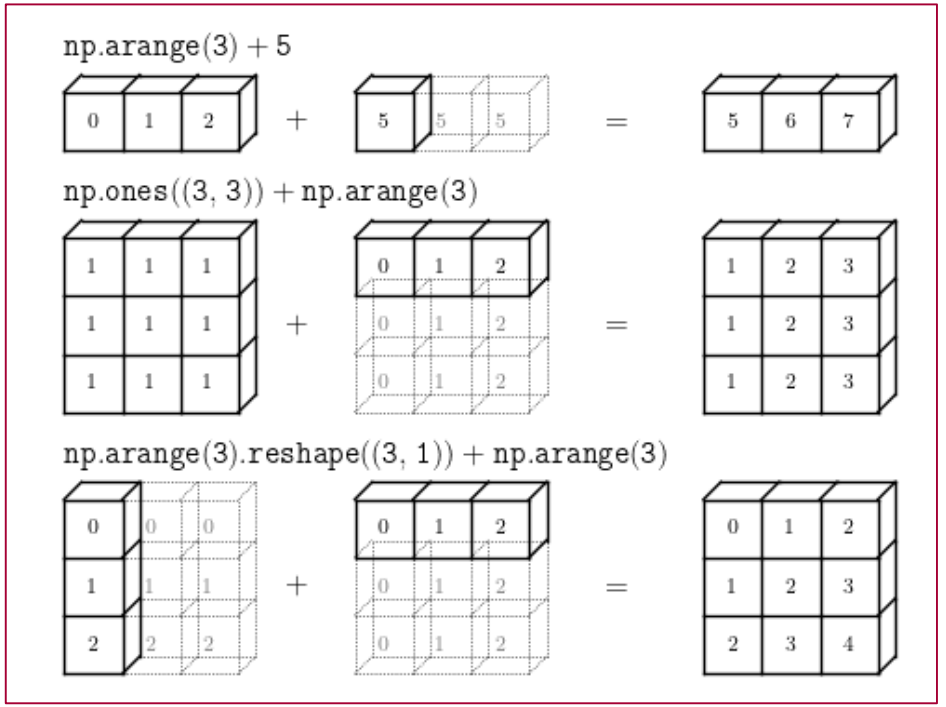
- Numpy
- Pytorch
  - vs. Numpy
  - GPU
  - Autograd
  - Simple Network

Numpy

- Numpy
  - Numpy는 계산과학분야에 핵심적으로 이용한 라이브러리.
  - 다차원 배열을 위한 기능과 선형 대수 연산, 수학 함수, 유사 난수 생성기를 포함.
  - 핵심 기능은 다차원 배열인 ndarray 클래스로, 이 배열의 모든 원소는 동일한 데이터 타입.

- Broadcasting
  - 브로드캐스팅은 Numpy가 shape이
  - 다른 배열 간에도 산술 연산이 가능하게 하는 알고리즘.
  - 작은 배열과 큰 배열이 있을 때, 큰 배열을 대상으로
  - 작은 배열을 여러 번 연산하고자 할 때가 있음.

  - 브로드캐스팅이 일어날 수 있는 조건.
  - 차원의 크기가 1일 때 가능하다.
  - 차원의 짝이 맞을 때 가능하다.



Numpy

- Numpy: Numerical Python Package (다차원 배열 처리에 대하여 빠르고 효율적임)

```python
import numpy as np

data = np.random.randn(2,3)

print (data)
print (data*5)
print (data+data)
print (data.shape)
print (data.dtype)
```

```
[[-0.03773005 -0.88780588 -1.00776151]
 [ 2.2996338  -0.84833909 -0.14052547]]
[[ -0.18865026  -4.43902938  -5.03880754]
 [ 11.49816901  -4.24169543  -0.70262737]]
[[-0.0754601  -1.77561175 -2.01552302]
 [ 4.5992676  -1.69667817 -0.28105095]]
(2, 3)
float64
```

Numpy

- Data를 생성하는 여러가지 방법들

```
data = np.zeros([2,4])
print (data)
data = np.ones([2,4])
print (data)
data = np.eye(4)
print (data)
```

```
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

Numpy

- List와 차이점 및 벡터 내적

```
lst1 = [1,2]
lst2 = [3,4]

np1 = np.array([1,2])
np2 = np.array([3,4])

print (lst1+lst2)
print (np1+np2)
print (np1*np2)

print (sum(np1*np2))
print (np1.dot(np2))
print (np.matmul(np1.reshape(1,-1),np2)[0])
```

```
[1, 2, 3, 4]
[4 6]
[3 8]
11
11
11
```

Numpy

- Matrix Multiplication

```
mat1 = np.array([[2, 5, 8], [5, 3, 1]])
mat2 = np.array([[3, 2], [5, 4], [7, 8]])
print (mat1)
print (mat2)
print (np.matmul(mat1, mat2))
```

```
[[2 5 8]
 [5 3 1]]
[[3 2]
 [5 4]
 [7 8]]
[[87 88]
 [37 30]]
```

Numpy

- 인덱스 / 슬라이싱

```
data = np.random.randn(4,3)
print (data)
print (data[2,:])
print (data[:,1])
print (data[2][1])
```

```
[[ 0.47000948 -1.50267905  0.11622734]
 [ 0.62878282 -0.59722714 -0.66938042]
 [-0.23668732 -1.14314775  0.62515476]
 [-0.19260966 -1.02565417  0.46766608]]
[-0.23668732 -1.14314775  0.62515476]
[-1.50267905 -0.59722714 -1.14314775 -1.02565417]
-1.14314775291
```

Numpy

- 기본적인 통계 관련 함수

```
print (np.sum(data))
print (np.mean(data))
print (np.var(data))
print (np.std(data))
```

```
-3.05954503188
-0.25496208599
0.497826245518
0.705568030397
```

Numpy

- Numpy 조건문

```
print (np.where(data>0))
print (np.where(data>0, True, False))
print (data[np.where(data>0, True, False)])
```

```
(array([0, 0, 1, 2, 3]), array([0, 2, 0, 2, 2]))
[[ True False  True]
 [ True False False]
 [False False  True]
 [False False  True]]
[ 0.47000948  0.11622734  0.62878282  0.62515476  0.46766608]
```

Pytorch (Tensors)

- 텐서 (tensor)는 배열 (array)이나 행렬 (matrix)과 매우 유사한 특수한 자료구조
- Pytorch에서는 텐서를 사용하여 모델의 입력 (input)과 출력 (output), 그리고 모델의 매개변수들을 부호화 (encode)함.
- 텐서는 GPU나 다른 하드웨어 가속기에서 실행할 수 있다는 점만 제외하면 Numpy의 ndarray와 유사

Pytorch (vs. Numpy)

- Pytorch: A replacement for NumPy to use the power of GPUs

```
import torch

data = torch.randn(2,3)

print (data)
print (data*5)
print (data+data)
print (data.shape)
print (data.dtype)
```

```
tensor([[-0.4156, -1.0064,  0.7929],
        [-0.5827,  1.5382, -1.3191]])
tensor([[-2.0780, -5.0321,  3.9644],
        [-2.9137,  7.6909, -6.5953]])
tensor([[-0.8312, -2.0129,  1.5857],
        [-1.1655,  3.0764, -2.6381]])
torch.Size([2, 3])
torch.float32
```

Pytorch (vs. Numpy)

- Data를 생성하는 여러가지 방법들

```
data = torch.zeros([2,4])
print (data)
data = torch.ones([2,4])
print (data)
data = torch.eye(4)
print (data)
```

```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.]])
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.]])
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]])
```

Pytorch (vs. Numpy)

- List와 차이점 및 벡터 내적

```
lst1 = [1,2]
lst2 = [3,4]

tor1 = torch.tensor([1,2])
tor2 = torch.tensor([3,4])

print (lst1+lst2)
print (tor1+tor2)
print (tor1*tor2)

print (sum(tor1*tor2))
print (tor1.dot(tor2))
print (torch.matmul(tor1.view(1,-1),tor2)[0])
```

```
[1, 2, 3, 4]
tensor([4, 6])
tensor([3, 8])
tensor(11)
tensor(11)
tensor(11)
```

Pytorch (vs. Numpy)

- Matrix Multiplication

```
mat1 = torch.tensor([[2, 5, 8], [5, 3, 1]])
mat2 = torch.tensor([[3, 2], [5, 4], [7, 8]])
print (mat1)
print (mat2)
print (torch.matmul(mat1, mat2))
```

```
tensor([[2, 5, 8],
        [5, 3, 1]])
tensor([[3, 2],
        [5, 4],
        [7, 8]])
tensor([[87, 88],
        [37, 30]])
```

Pytorch (vs. Numpy)

- 인덱스 / 슬라이싱

```
data = torch.randn(4,3)
print (data)
print (data[2,:])
print (data[:,1])
print (data[2][1])
```

```
tensor([[ 0.4979,  1.0363, -0.5115],
        [ 0.3155,  0.1153, -0.1279],
        [ 0.6958, -1.8841,  1.4718],
        [ 0.1904, -1.7743, -0.2923]])
tensor([ 0.6958, -1.8841,  1.4718])
tensor([ 1.0363,  0.1153, -1.8841, -1.7743])
tensor(-1.8841)
```

Pytorch (vs. Numpy)

- 기본적인 통계 관련 함수

```
print (torch.sum(data))
print (torch.mean(data))
print (torch.var(data))
print (torch.std(data))
print ()
print (torch.sum(data).item())
print (torch.mean(data).item())
print (torch.var(data).item())
print (torch.std(data).item())
```

```
tensor(-0.2671)
tensor(-0.0223)
tensor(1.0161)
tensor(1.0080)

-0.26709261536598206
-0.022257717326283455
1.0160642862319946
1.008000135421753
```

Pytorch (vs. Numpy)

- Pytorch 조건문

```
print (torch.where(data>0, torch.ones(data.shape), torch.zeros(data.shape)))
```

```
tensor([[1., 1., 0.],
        [1., 1., 0.],
        [1., 0., 1.],
        [1., 0., 0.]])
```

Pytorch (vs. Numpy)

- Numpy와 호환

numpy --> torch / torch --> numpy

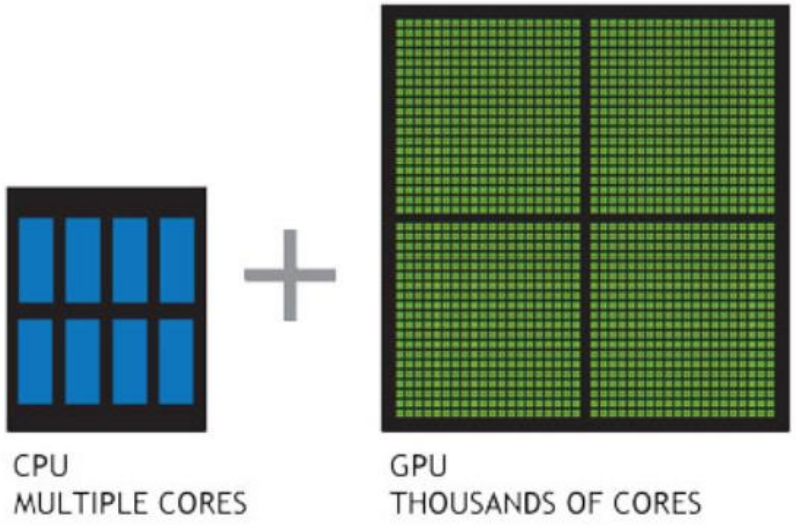
```
a = torch.ones(5)
print (a)
b = a.numpy()
print (b)
a = torch.from_numpy(b)
print (a)
```

```
tensor([1., 1., 1., 1., 1.])
[ 1.  1.  1.  1.  1.]
tensor([1., 1., 1., 1., 1.])
```

Pytorch (GPU)

GPU(Graphical Processing Unit): 병렬 처리 용으로 설계된 수 천 개의 CPU 보다 소형이고 효율적인 코어로 구성됨

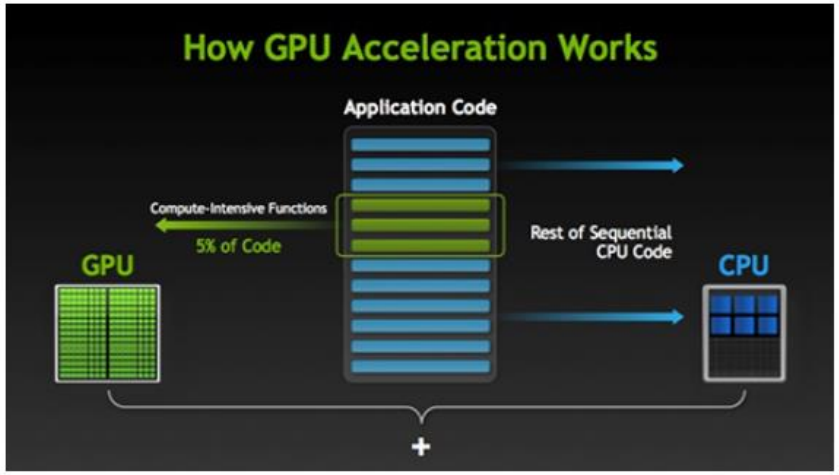
CPU vs. GPU



처리 방식에서의 차이

- CPU: 직렬 처리에 최적화된 몇 개의 코어
- GPU: 병렬 처리 용으로 설계된 수천 개의 소형 코어

GPU가 가속하는 방법



- GPU: 연산집약적인 부분 처리
- CPU: 나머지 부분 처리

Pytorch (GPU)

- code

GPU

```
print (torch.cuda.is_available())
print (torch.cuda.device_count())
```

```
True
1
```

```
x = torch.tensor([1, 1]).cuda()
print (x)
x = x.cpu()
print (x)
```

```
tensor([1, 1], device='cuda:0')
tensor([1, 1])
```

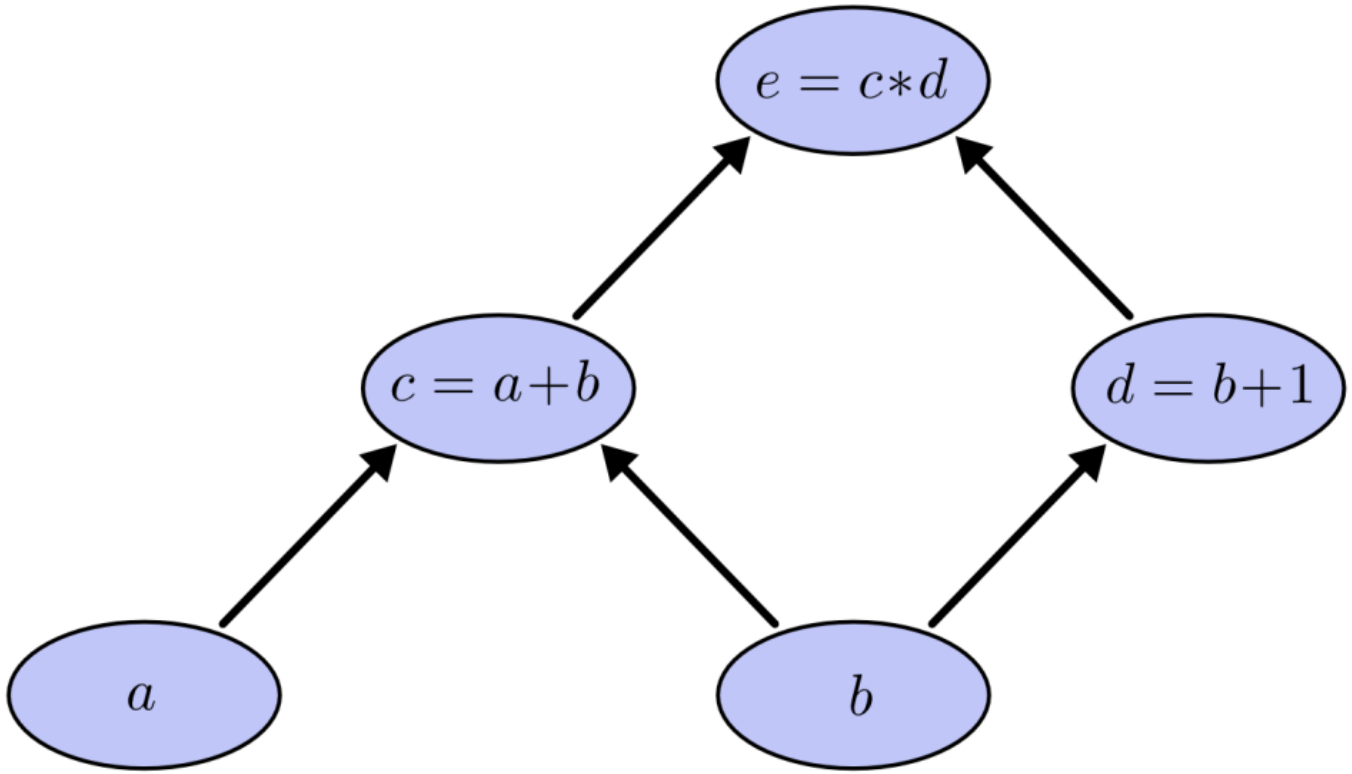
Pytorch (Autograd)

- torch.autograd를 사용한 자동 미분
  - 신경망을 학습할 때 가장 자주 사용되는 알고리즘은 역전파 (back propagation)
  - 역전파 알고리즘에서, 매개변수 (모델의 가중치; weight)는 주어진 매개변수에 대한 손실 함수의 변화도에 따라 조정.
  - 이러한 변화도 계산을 위해 Pytorch에는 torch.autograd라고 불리는 자동 미분 엔진이 내장.

- 변화도 추적 멈추기
  - 기본적으로, requires_grad=True인 모든 텐서들은 연산 기록을 추적하고 변화도 계산을 지원.
  - 그러나 순전파 (forward propagation) 연산만 필요한 경우 (모델을 학습한 뒤, 입력 데이터에 대한 추론), 이러한 추적이나 지원이 필요 없을 수 있음.
  - 크게 두 가지 방법이 있음: no_grad 메소드 / detach 메소드
  - 신경망의 일부 매개변수를 고정된 매개 변수로 표시하는데, 이는 사전 학습된 신경망을 미세조정할 때 많이 사용됨.
  - 변화도를 추적하지 않는 테서의 연산이 더 효율적이기 때문에, 순전파 단계만 수행할 때 연산 속도가 향상.

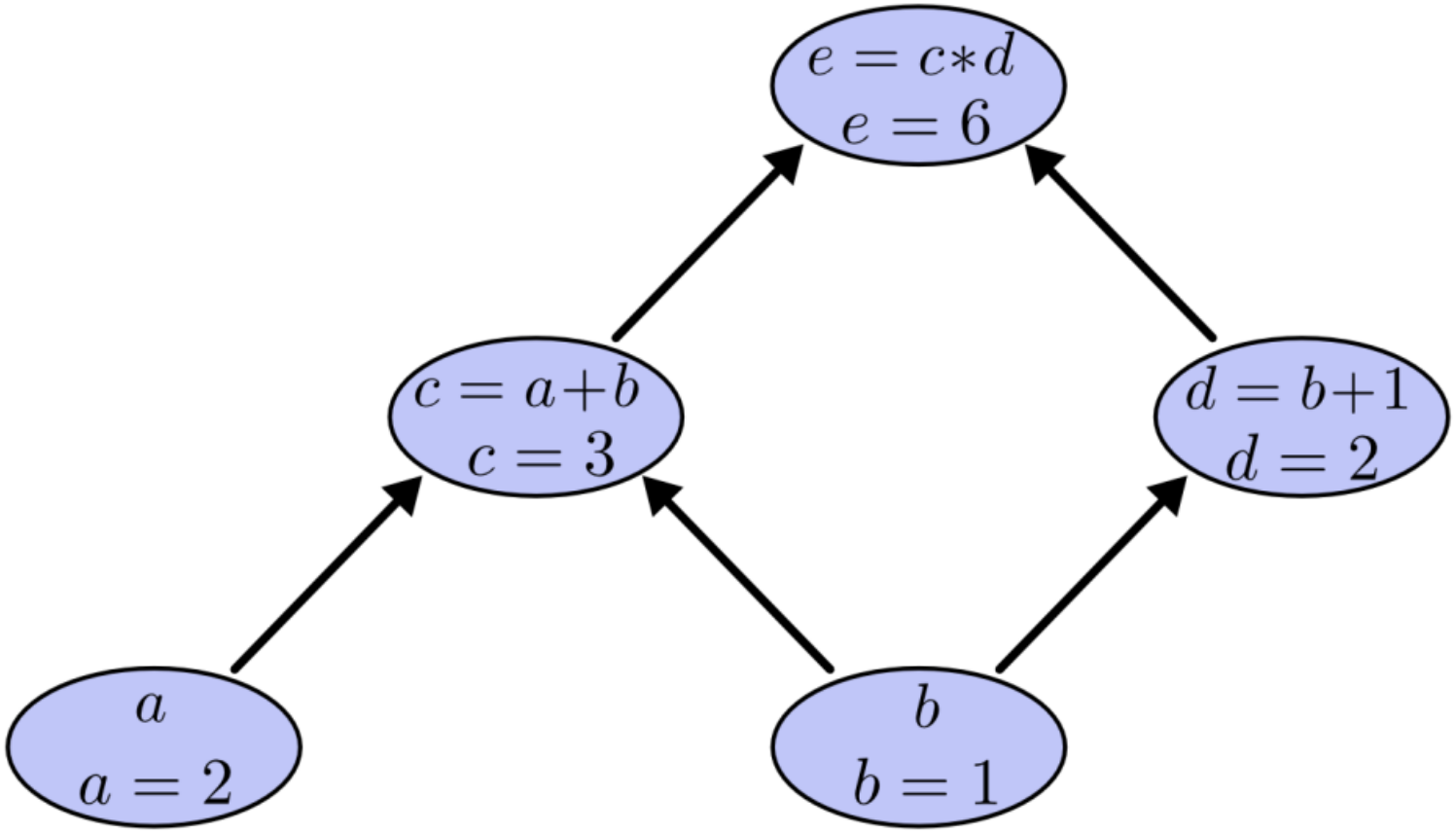
Pytorch (Autograd)

- Computation Graph (the way of Tensorflow: Define and Run)



Pytorch (AutoGrad)

- Computation Graph (the way of Pytorch: Define by Run)



Pytorch (AutoGrad)

- Computation Graph (code) : requires_grad=True

```
# Computational Graph
a = torch.tensor([2.], requires_grad=True)
b = torch.tensor([1.], requires_grad=True)
c = a+b
d = b+1
e = c*d
print (a)
print (b)
print (c)
print (d)
print (e)
```

```
tensor([2.], requires_grad=True)
tensor([1.], requires_grad=True)
tensor([3.], grad_fn=<ThAddBackward>)
tensor([2.], grad_fn=<AddBackward>)
tensor([6.], grad_fn=<ThMulBackward>)
```

```
print (a.data)
print (a.grad)
print (a.grad_fn)
```

```
tensor([2.])
None
None
```

Pytorch (AutoGrad)

- Computation Graph (code): .backward()

```
e.backward(retain_graph=True) # gradient가 계산이 됨

print (a.grad) # --> (de/da)
print (b.grad) # --> (de/db)
print (c.grad) # --> (de/dc)
print (d.grad) # --> (de/dd)
print (e.grad) # --> (de/de)
```

```
tensor([2.])
tensor([5.])
None
None
None
```

Pytorch (AutoGrad)

- Computation Graph (code): .retain_grad()

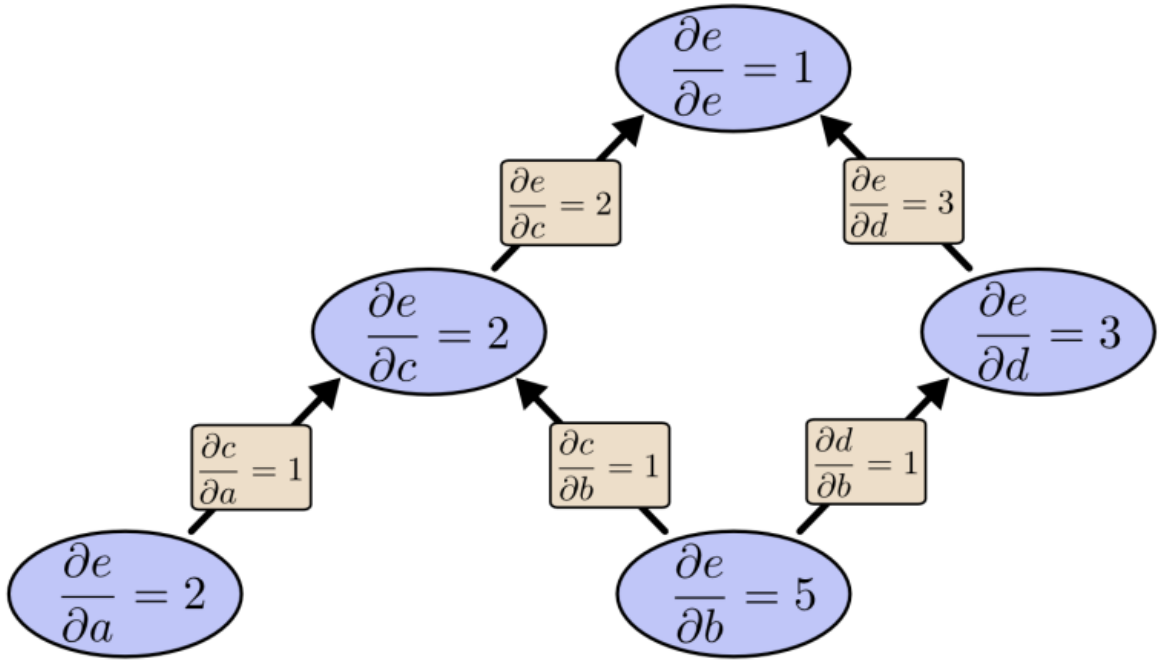
```
# Computational Graph
a = torch.tensor([2.], requires_grad=True)
b = torch.tensor([1.], requires_grad=True)
c = a+b
d = b+1
e = c*d

c.retain_grad()
d.retain_grad()
e.retain_grad()
```

```
e.backward(retain_graph=True)
```

```
print (a.grad) # --> (de/da)
print (b.grad) # --> (de/db)
print (c.grad) # --> (de/dc)
print (d.grad) # --> (de/dd)
print (e.grad) # --> (de/de)
```

```
tensor([2.])
tensor([5.])
tensor([2.])
tensor([3.])
tensor([1.])
```



Pytorch AutoGrad

- Computation Graph (code): .detach() / .requires_grad -> transfer learning

```
a = a.detach()
print (a.requires_grad)
```

```
False
```