

Autograd in Pytorch

Eunhui Kim/KISTI

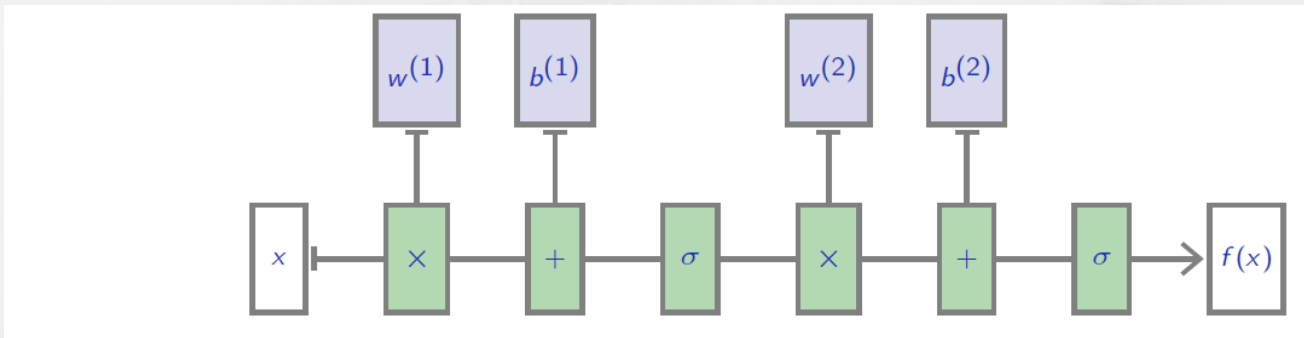
Autograd in Pytorch

목차

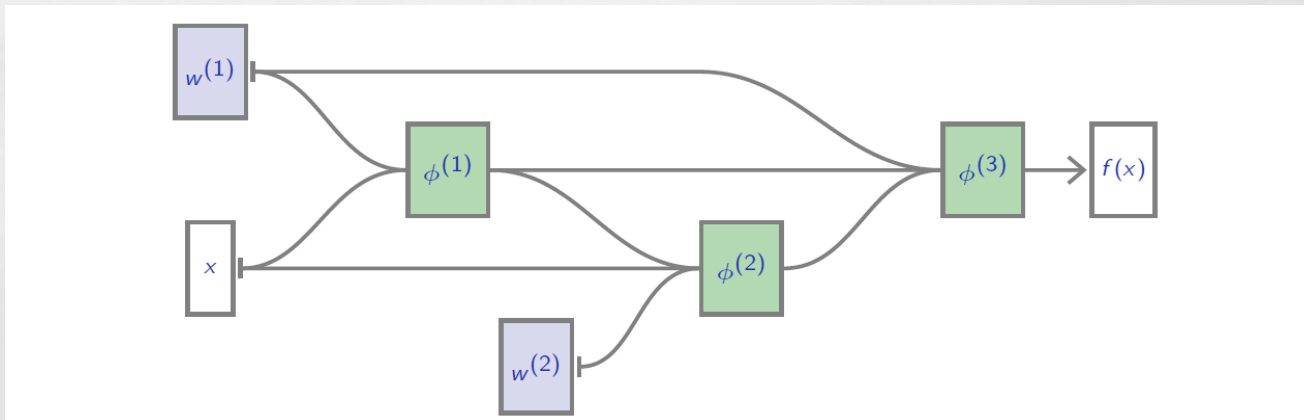
- 1) DAG Network
- 2) DAG Network & Weight Sharing
- 3) AutoGrad
- 4) AutoGrad – `requires_grad`
- 5) AutoGrad – `Tensor.backward()`
- 6) AutoGrad Machinery
- 7) AutoGrad – `no_grad`, `detach()`,
- 8) Autograd – `create_graph = True`
- 9) Pytorch 실습 on Google CoLab

1) DAG Network

- MLP에 대해 살펴본 모든 연산은



- 연산자들에 대한 임의의 “비순환형 그래프”(“Directed Acyclic Graph”)로 일반화 될 수 있다.



1) DAG Network

- 우리가 사용해온 tensorial notation을 기억해 보면,

If $(a_1, \dots, a_Q) = \phi(b_1, \dots, b_R)$, we have

$$\left[\frac{\partial a}{\partial b} \right] = J_\phi = \begin{pmatrix} \frac{\partial a_1}{\partial b_1} & \cdots & \frac{\partial a_1}{\partial b_R} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_Q}{\partial b_1} & \cdots & \frac{\partial a_Q}{\partial b_R} \end{pmatrix}.$$

- 이 notation은 계산 지점을 지정하지 않음.
항상 forward-pass activations을 위한 것임.

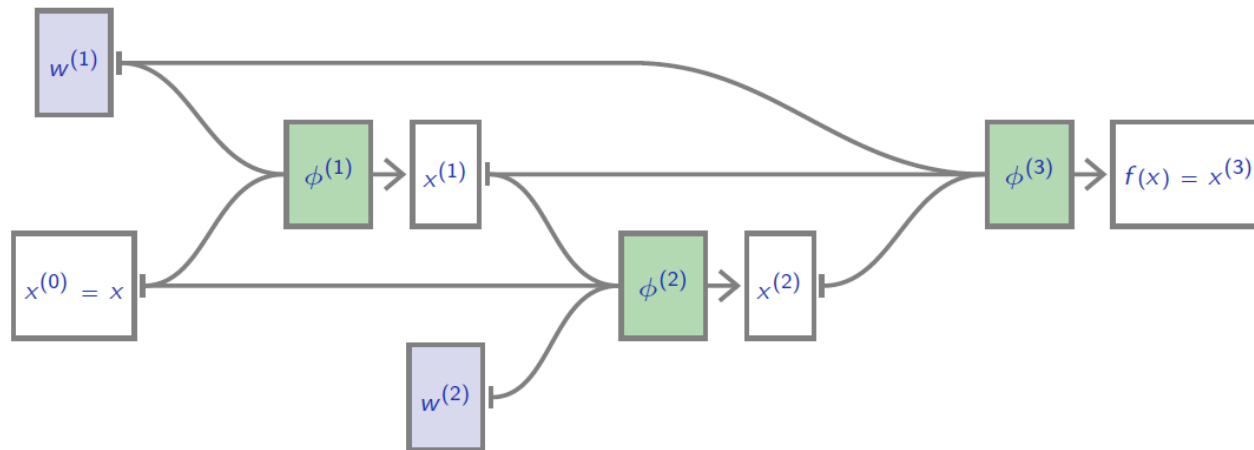
Also, if $(a_1, \dots, a_Q) = \phi(b_1, \dots, b_R, c_1, \dots, c_S)$, we use

$$\left[\frac{\partial a}{\partial c} \right] = J_{\phi|c} = \begin{pmatrix} \frac{\partial a_1}{\partial c_1} & \cdots & \frac{\partial a_1}{\partial c_S} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_Q}{\partial c_1} & \cdots & \frac{\partial a_Q}{\partial c_S} \end{pmatrix}.$$



1) DAG Network – Forward Pass

순방향 연산(Forward Pass)



$$x^{(0)} = x$$

$$x^{(1)} = \phi^{(1)}(x^{(0)}; w^{(1)})$$

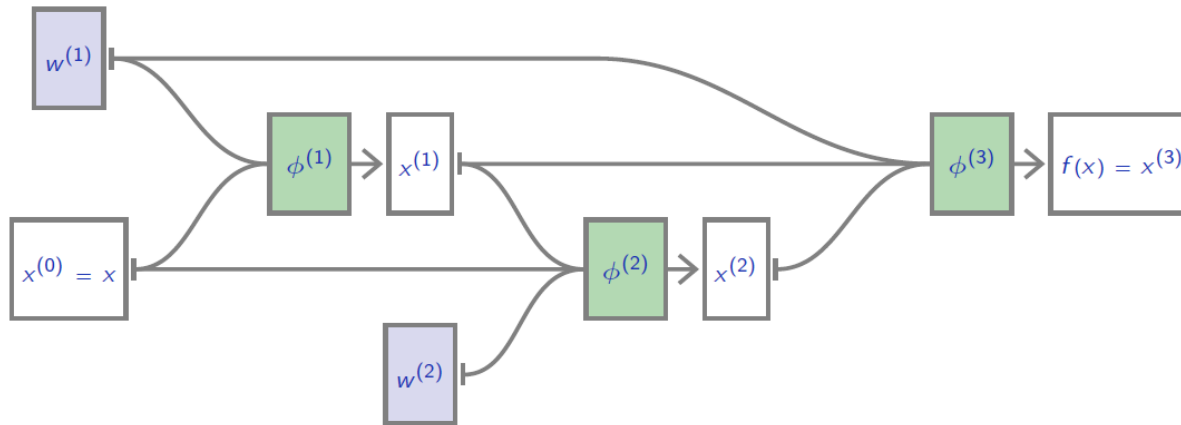
$$x^{(2)} = \phi^{(2)}(x^{(0)}, x^{(1)}; w^{(2)})$$

$$f(x) = x^{(3)} = \phi^{(3)}(x^{(1)}, x^{(2)}; w^{(1)})$$



1) DAG Network – Backward Pass

역방향 연산(Backward Pass), 활성함수에 따라 미분



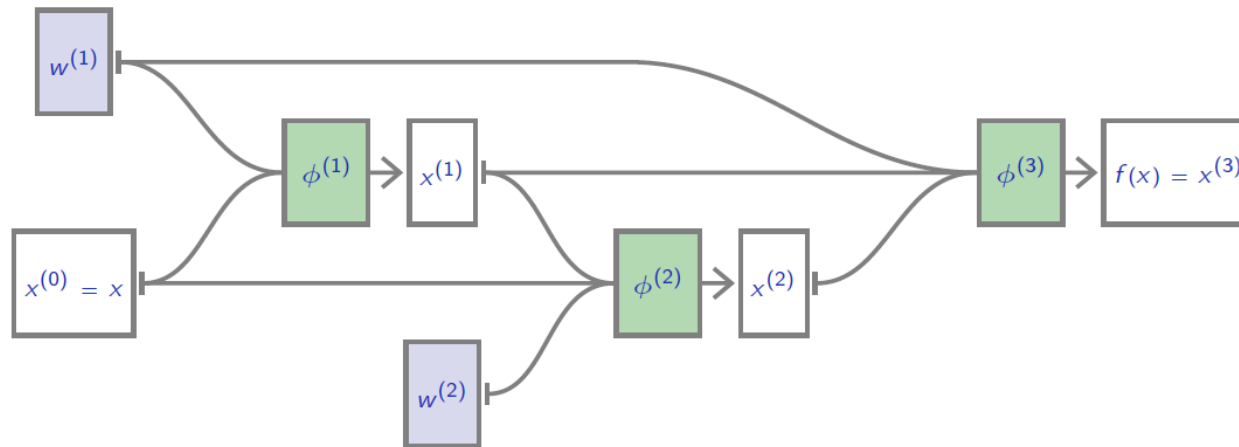
$$\left[\frac{\partial \ell}{\partial x^{(2)}} \right] = \left[\frac{\partial x^{(3)}}{\partial x^{(2)}} \right] \left[\frac{\partial \ell}{\partial x^{(3)}} \right] = J_{\phi^{(3)}|x^{(2)}} \left[\frac{\partial \ell}{\partial x^{(3)}} \right]$$

$$\left[\frac{\partial \ell}{\partial x^{(1)}} \right] = \left[\frac{\partial x^{(2)}}{\partial x^{(1)}} \right] \left[\frac{\partial \ell}{\partial x^{(2)}} \right] + \left[\frac{\partial x^{(3)}}{\partial x^{(1)}} \right] \left[\frac{\partial \ell}{\partial x^{(3)}} \right] = J_{\phi^{(2)}|x^{(1)}} \left[\frac{\partial \ell}{\partial x^{(2)}} \right] + J_{\phi^{(3)}|x^{(1)}} \left[\frac{\partial \ell}{\partial x^{(3)}} \right]$$

$$\left[\frac{\partial \ell}{\partial x^{(0)}} \right] = \left[\frac{\partial x^{(1)}}{\partial x^{(0)}} \right] \left[\frac{\partial \ell}{\partial x^{(1)}} \right] + \left[\frac{\partial x^{(2)}}{\partial x^{(0)}} \right] \left[\frac{\partial \ell}{\partial x^{(2)}} \right] = J_{\phi^{(1)}|x^{(0)}} \left[\frac{\partial \ell}{\partial x^{(1)}} \right] + J_{\phi^{(2)}|x^{(0)}} \left[\frac{\partial \ell}{\partial x^{(2)}} \right]$$

1) DAG Network – Backward Pass

역방향 연산(Backward Pass), 매개 변수에 따라 미분



$$\left[\frac{\partial \ell}{\partial w^{(1)}} \right] = \left[\frac{\partial x^{(1)}}{\partial w^{(1)}} \right] \left[\frac{\partial \ell}{\partial x^{(1)}} \right] + \left[\frac{\partial x^{(3)}}{\partial w^{(1)}} \right] \left[\frac{\partial \ell}{\partial x^{(3)}} \right] = J_{\phi^{(1)}|w^{(1)}} \left[\frac{\partial \ell}{\partial x^{(1)}} \right] + J_{\phi^{(3)}|w^{(1)}} \left[\frac{\partial \ell}{\partial x^{(3)}} \right]$$

$$\left[\frac{\partial \ell}{\partial w^{(2)}} \right] = \left[\frac{\partial x^{(2)}}{\partial w^{(2)}} \right] \left[\frac{\partial \ell}{\partial x^{(2)}} \right] = J_{\phi^{(2)}|w^{(2)}} \left[\frac{\partial \ell}{\partial x^{(2)}} \right]$$

1) DAG Network

- 따라서 만일 우리가 “tensor 연산자”에 대한 library를 지니고 있다면, 그 구현은 다음과 같음:

$$\begin{aligned}(x_1, \dots, x_d, w) &\mapsto \phi(x_1, \dots, x_d; w) \\ \forall c, (x_1, \dots, x_d, w) &\mapsto J_{\phi|_{x_c}}(x_1, \dots, x_d; w) \\ (x_1, \dots, x_d, w) &\mapsto J_{\phi|_w}(x_1, \dots, x_d; w),\end{aligned}$$

- 우리는 각 노드에서 이러한 연산자들을 이용하여 임의의 비 순환형 graph(DAG)를 그리고, 결과 mapping 연산과 back-prop gradient 계산이 가능함.



1) DAG Network – Framework

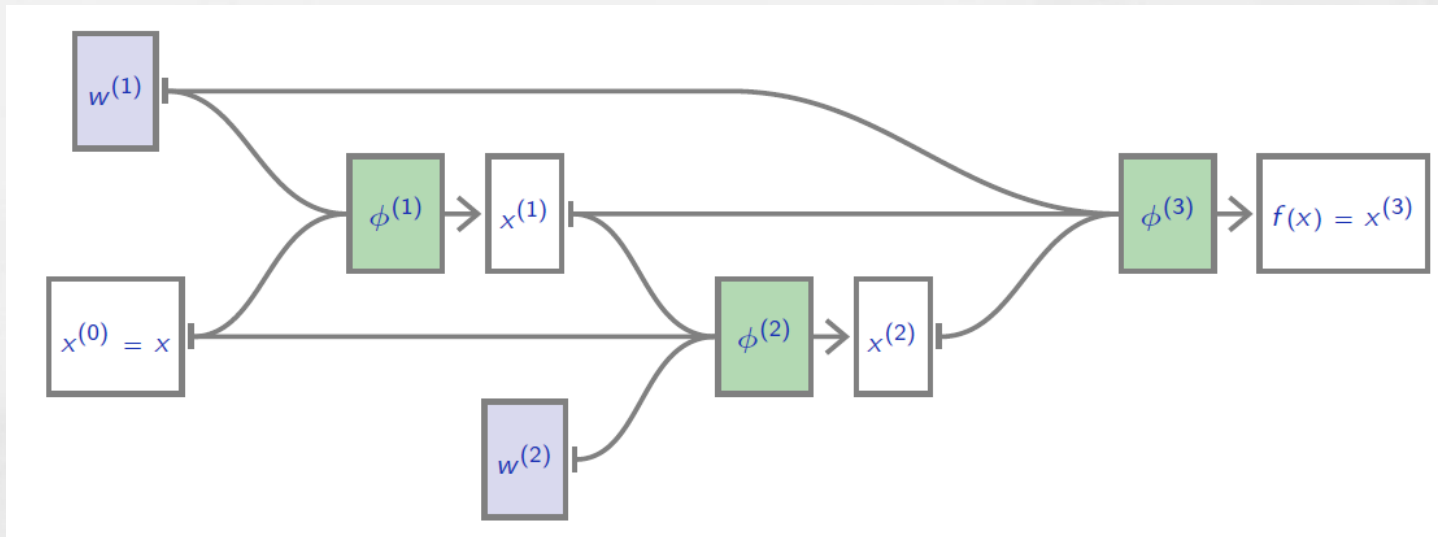
- 여러 framework들이 DAG와 자동으로 이를 미분하는 tensor 연산자 library 와 mechanisms을 제공함.



	Language(s)	License	Main backer
PyTorch	Python	BSD	Facebook
Caffe2	C++, Python	Apache	Facebook
TensorFlow	Python, C++	Apache	Google
MXNet	Python, C++, R, Scala	Apache	Amazon
CNTK	Python, C++	MIT	Microsoft
Torch	Lua	BSD	Facebook
Theano	Python	BSD	U. of Montreal
Caffe	C++	BSD 2 clauses	U. of CA, Berkeley

2) DAG Network & Weight Sharing

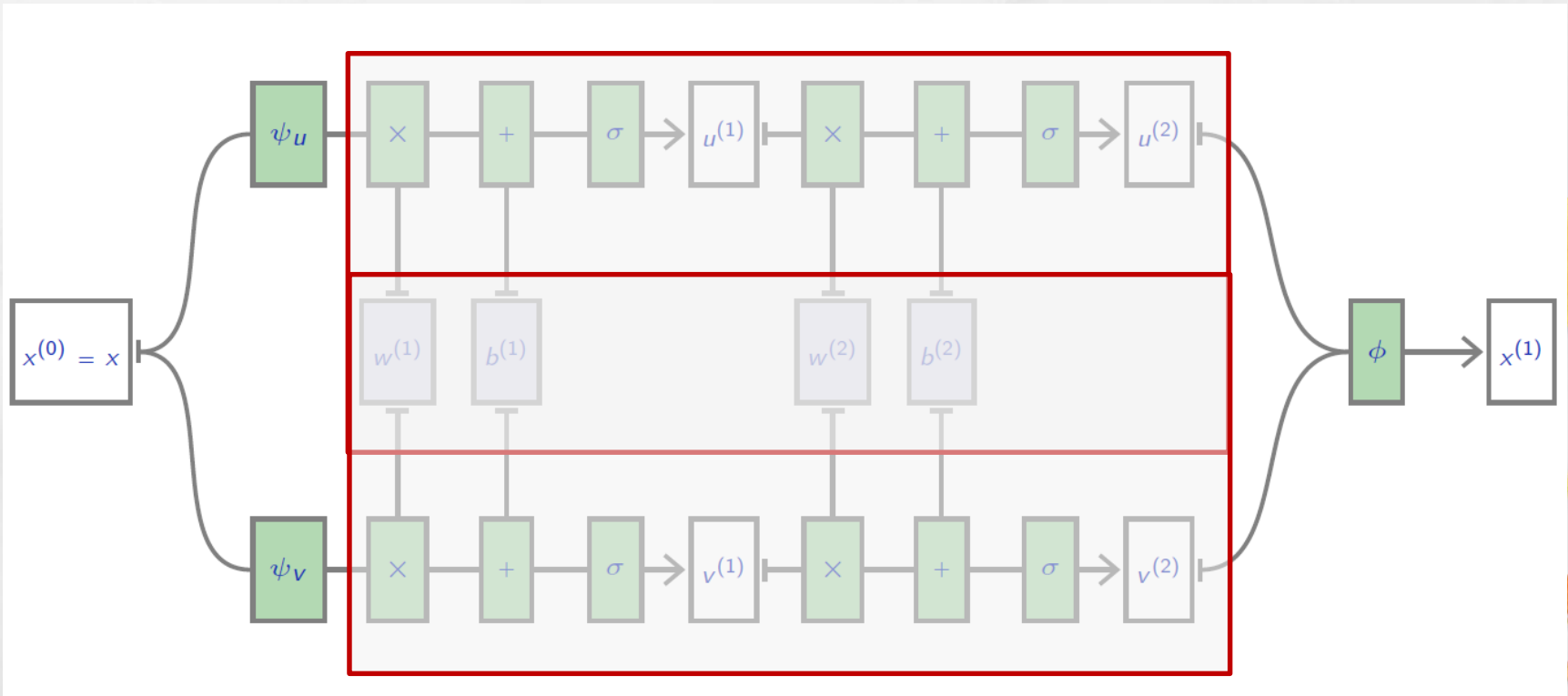
- 우리의 일반화된 DAG 공식에서, 특히 동일한 매개변수가 서로 다른 process에서 암묵적으로 사용되는 것을 허용함.
- 다음 예시에서, 예를 들어 $w^{(1)}$ 은 두 노드 $\phi^{(1)}$ 과 $\phi^{(3)}$ 에서 매개변수 화 됨.



- 이를 가중치 공유(weight sharing)이라 칭함.

2) DAG Network & Weight Sharing

- Weight sharing 는 sub-network 이 여러 번 반복되는 경우, 특히 쌍둥이 네트워크 (siamese networks)을 구축하게 함.



3) AutoGrad

- 개념적으로, the forward pass는 표준 tensor 연산이며, DAG의 tensor 연산은 미분을 계산하는 경우에만 필요함.
- tensor 연산 실행 시, PyTorch는 관련 모든 텐서의 모든 수량의 기울기 계산을 위해 연산자 그래프를 자동 생성함.
- 이러한 “autograd” mechanism의 (Paszke et al., 2017) 두 장점으로 :
 - 문법의 단순성: Python 연산자의 표준 sequence로 forward pass를 작성하기만 하면 됨,
 - 큰 유연성: graph가 고정되어 있지 않으므로, forward pass는 동적으로 모듈화 가능함.



4) AutoGrad – requires_grad

- Tensor는 `requires_grad`라는 Boolean field를 갖는데 이는 default로 False 임.
- Gradient 연산이 필요한 경우 `requires_grad`라는 Boolean field 를 True 로 지정해야 함

```
>>> x = torch.tensor([ 1., 2. ])
>>> y = torch.tensor([ 4., 5. ])
>>> z = torch.tensor([ 7., 3. ])
>>> x.requires_grad
False
>>> (x + y).requires_grad
False
>>> z.requires_grad = True
>>> (x + z).requires_grad
True
```



4) AutoGrad – requires_grad

! floating point 유형 tensors만이 gradient 연산을 지원함

```
>>> x = torch.tensor([1., 10.])
>>> x.requires_grad = True
>>> x = torch.tensor([1, 10])
>>> x.requires_grad = True
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: only Tensors of floating point dtype can require gradients
```



4) AutoGrad – torch.autograd.grad

- `torch.autograd.grad(outputs, inputs)`는 입력에 대한 출력의 gradient를 계산하여 return함.

```
>>> t = torch.tensor([1., 2., 4.]).requires_grad_()
>>> u = torch.tensor([10., 20.]).requires_grad_()
>>> a = t.pow(2).sum() + u.log().sum()
>>> torch.autograd.grad(a, (t, u))
(tensor([2., 4., 8.]), tensor([0.1000, 0.0500]))
```

- 입력은 single tensor이지만, 결과 값은 [one element] tuple이 됨
- 만일 출력이 tuple이라면, 그 결과는 gradients의 각 element의 합임.



5) AutoGrad – Tensor.backward()

- 함수 `Tensor.backward()`는 autograd graph의 “leaves”인 `grad` fields에 연산의 결과가 아닌 gradient를 누적함.

```
>>> x = torch.tensor([ -3., 2., 5. ]).requires_grad_()
>>> u = x.pow(3).sum()
>>> x.grad
>>> u.backward()
>>> x.grad
tensor([27., 12., 75.] )
```

- 이 함수는 `torch.autograd.grad(...)` 의 대체 함수이며, 학습 모듈의 표준임



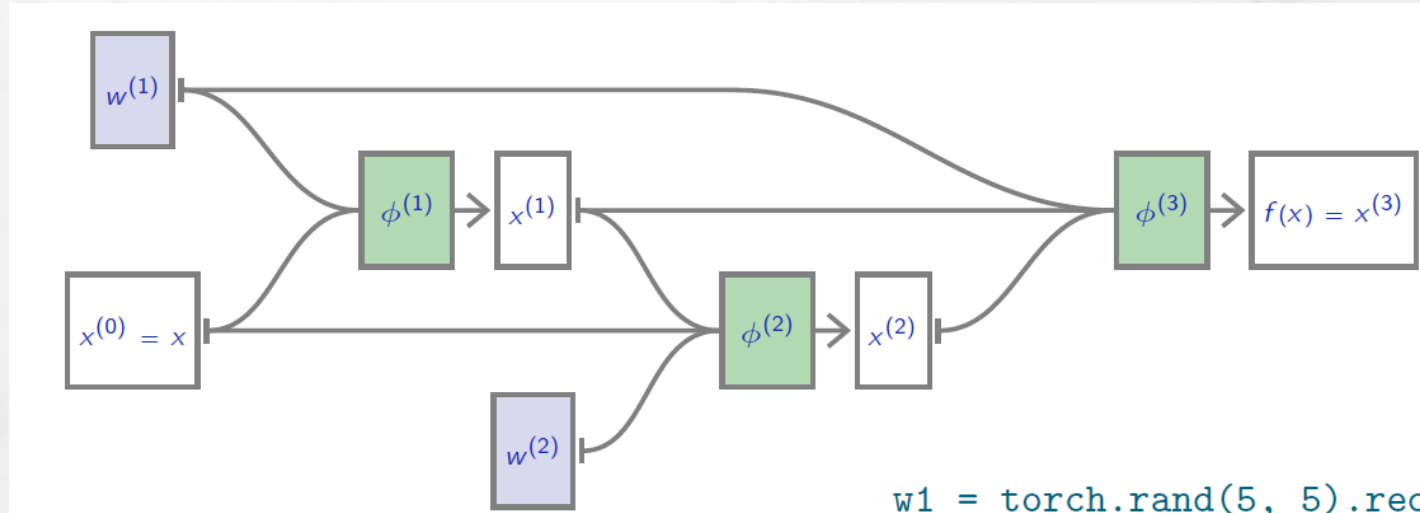
5) Autograd –Tensor.backward()

! Tensor.backward() 는 서로 다른 Tensor들의 gradients를 누적하므로, 호출 이전에 zero로 setting해야 함.

- 이러한 누적 연산은 “mini-batches” 에 걸쳐 loss를 합하는 gradient연산에서 특히 바람직함.
- 즉, loss들의 합의 gradient연산에서 바람직한 연산임.

5) Autograd – Tensor.backward()

- 그래서 우리는 forward/backward pass 를 다음과 같이 실행 함.



$$\phi^{(1)} \left(x^{(0)}; w^{(1)} \right) = w^{(1)} x^{(0)}$$

$$\phi^{(2)} \left(x^{(0)}, x^{(1)}; w^{(2)} \right) = x^{(0)} + w^{(2)} x^{(1)}$$

$$\phi^{(3)} \left(x^{(1)}, x^{(2)}; w^{(1)} \right) = w^{(1)} \left(x^{(1)} + x^{(2)} \right)$$

```
w1 = torch.rand(5, 5).requires_grad_()
w2 = torch.rand(5, 5).requires_grad_()
x = torch.empty(5).normal_()
```

```
x0 = x
x1 = w1 @ x0
x2 = x0 + w2 @ x1
x3 = w1 @ (x1 + x2)
```

```
q = x3.norm()
```

```
q.backward()
```

6) Autograd machinery

- The autograd 의 그래프는 Tensors의 `grad_fn` fields와 `Functions` 의 `next_functions` fields로 encoding 됨.

```
>>> x = torch.tensor([ 1.0, -2.0, 3.0, -4.0 ]).requires_grad_()
>>> a = x.abs()
>>> s = a.sum()
>>> s
tensor(10., grad_fn=<SumBackward0>)
>>> s.grad_fn.next_functions
((<AbsBackward object at 0x7ffb2b1462b0>, 0)|,)
```

- `Functions`는 다음 강의에 재 소개 될 예정임



6) Autograd machinery

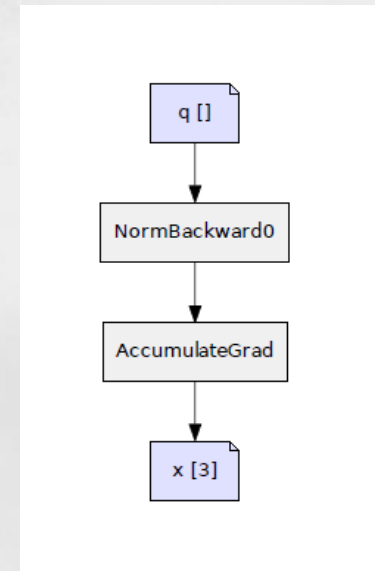
- 우리는 연산 동안 세워진 full graph를 시각화 할 수 있음

```
x = torch.tensor([1., 2., 2.]).requires_grad_()
q = x.norm()
```

이 그래프는 다음 툴

<https://fleuret.org/git/agtree2dot>

과 Graphviz를 통해 생성됨.



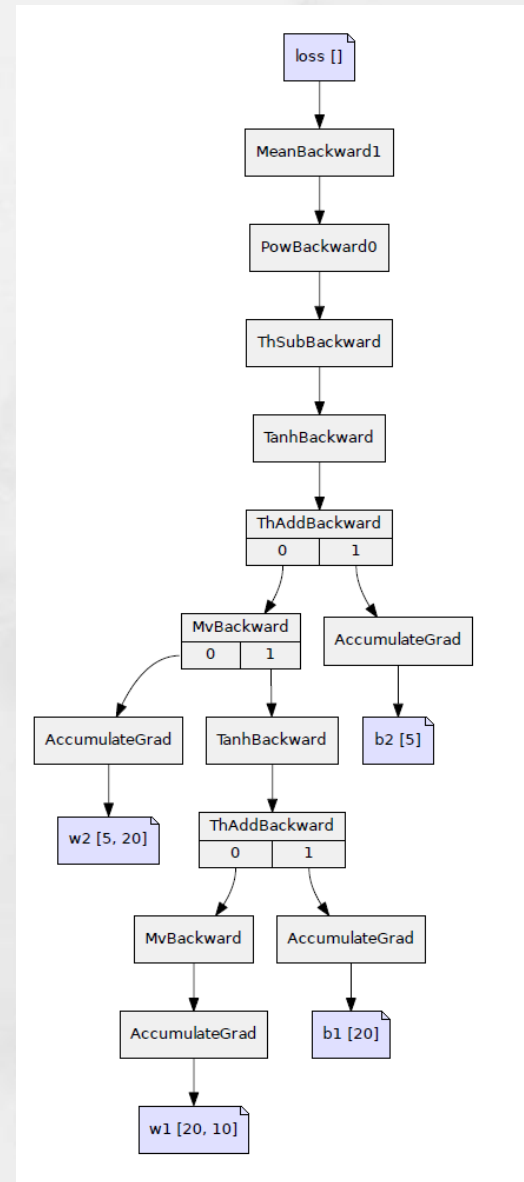
6) Autograd machinery

```
w1 = torch.rand(20, 10).requires_grad_()
b1 = torch.rand(20).requires_grad_()
w2 = torch.rand(5, 20).requires_grad_()
b2 = torch.rand(5).requires_grad_()
```

```
x = torch.rand(10)
h = torch.tanh(w1 @ x + b1)
y = torch.tanh(w2 @ h + b2)
```

```
target = torch.rand(5)
```

```
loss = (y - target).pow(2).mean()
```

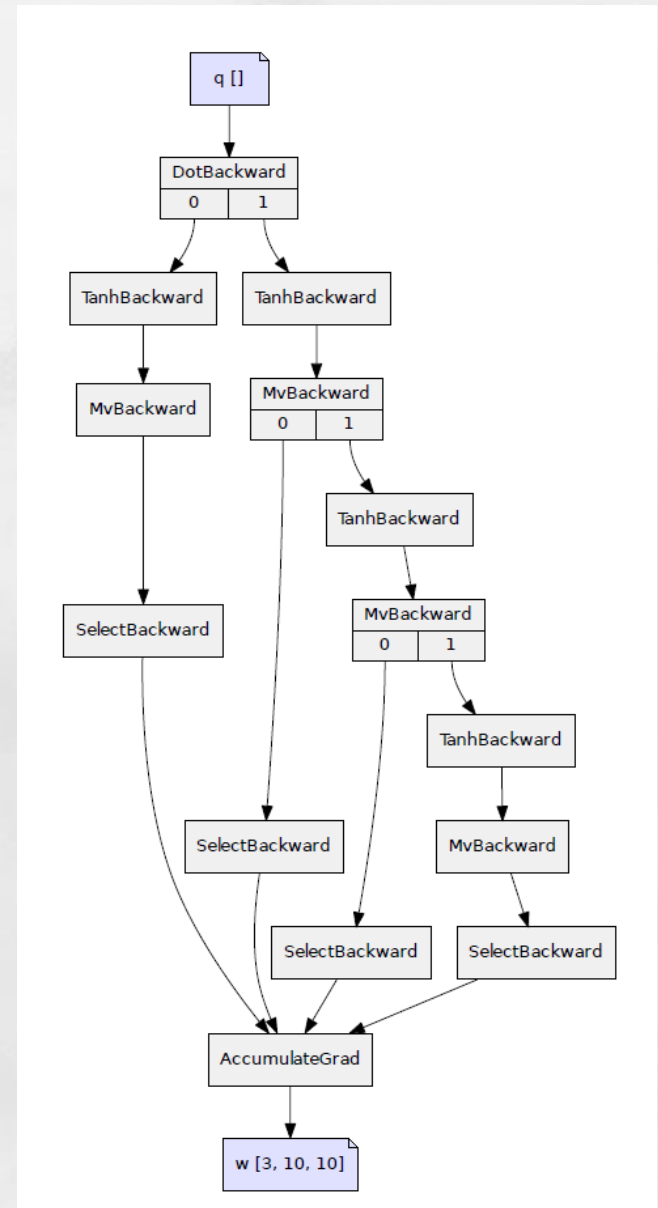


6) Autograd machinery

```
w = torch.rand(3, 10, 10).requires_grad_()
```

```
def blah(k, x):  
    for i in range(k):  
        x = torch.tanh(w[i] @ x)  
    return x
```

```
u = blah(1, torch.rand(10))  
v = blah(3, torch.rand(10))  
q = u.dot(v)
```



6) Autograd machinery

! 비록 연관관계에 있으나, autograd graph가 네트워크의 구조는 아닌 gradient 연산을 위한 graph 연산자 임. 이는 data-dependent하며 network의 부분 구조를 생략하거나 반복할 수 있는 특징을 지님.



7) Autograd – no_grad, detach

- autograd machinery 의 `torch.no_grad()` context 스위치는 매개변수 업데이트와 같은 작업에 사용됨.

```
w = torch.empty(10, 784).normal_(0, 1e-3).requires_grad_()
b = torch.empty(10).normal_(0, 1e-3).requires_grad_()

for k in range(10001):
    y_hat = x @ w.t() + b
    loss = (y_hat - y).pow(2).mean()

    w.grad, b.grad = None, None
    loss.backward()

    with torch.no_grad():
        w -= eta * w.grad
        b -= eta * b.grad
```



7) Autograd – no_grad, detach

- `detach()` 메서드는 데이터를 공유하지만 gradient 연산을 필요로 하지 않는 tensor를 생성함. 그리고 현재 graph에 연결되지 않음.
- 이 메서드는 gradient가 변수를 넘어 전파되지 않아야 하거나, leaf tensor를 업데이트 할 때 사용함.



7) Autograd – no_grad, detach

```
a = torch.tensor( 0.5).requires_grad_()
b = torch.tensor(-0.5).requires_grad_()

for k in range(100):
    l = (a - 1)**2 + (b + 1)**2 + (a - b)**2
    ga, gb = torch.autograd.grad(l, (a, b))
    with torch.no_grad():
        a -= eta * ga
        b -= eta * gb

print('%.06f' % a.item(), '%.06f' % b.item())

prints

0.333333 -0.333333
```



7) Autograd – no_grad, detach

```
a = torch.tensor( 0.5).requires_grad_()
b = torch.tensor(-0.5).requires_grad_()

for k in range(100):
    l = (a - 1)**2 + (b + 1)**2 + (a.detach() - b)**2
    ga, gb = torch.autograd.grad(l, (a, b))
    with torch.no_grad():
        a -= eta * ga
        b -= eta * gb

print('%.06f' % a.item(), '%.06f' % b.item())
```

prints

1.000000 -0.000000



8) Autograd – create_graph = True

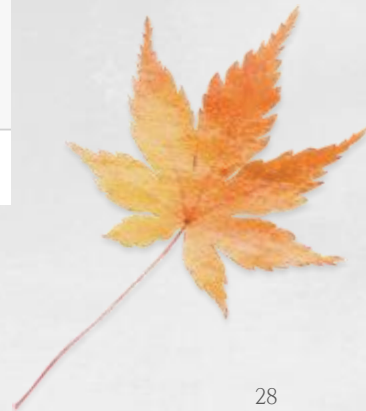
- Autograd는 또한 기울기 자체의 계산을 추적하여 고차 도함수를 허용함.
이는 `create_graph = True`로 지정함.:

```
In [34]: 1 x = torch.tensor([1., 2., 3.]).requires_grad_()
          2 phi = x.pow(2).sum()
          3 g1, = torch.autograd.grad(phi, x, create_graph = True)
          4 g1
```

```
Out [34]: tensor([2., 4., 6.], grad_fn=<MulBackward0>)
```

```
In [35]: 1 psi = g1[0].exp() - g1[2].exp()
          2 g2, = torch.autograd.grad(psi, x)
          3 g2
```

```
Out [35]: tensor([ 14.7781,  0.0000, -806.8576])
```



8) Autograd

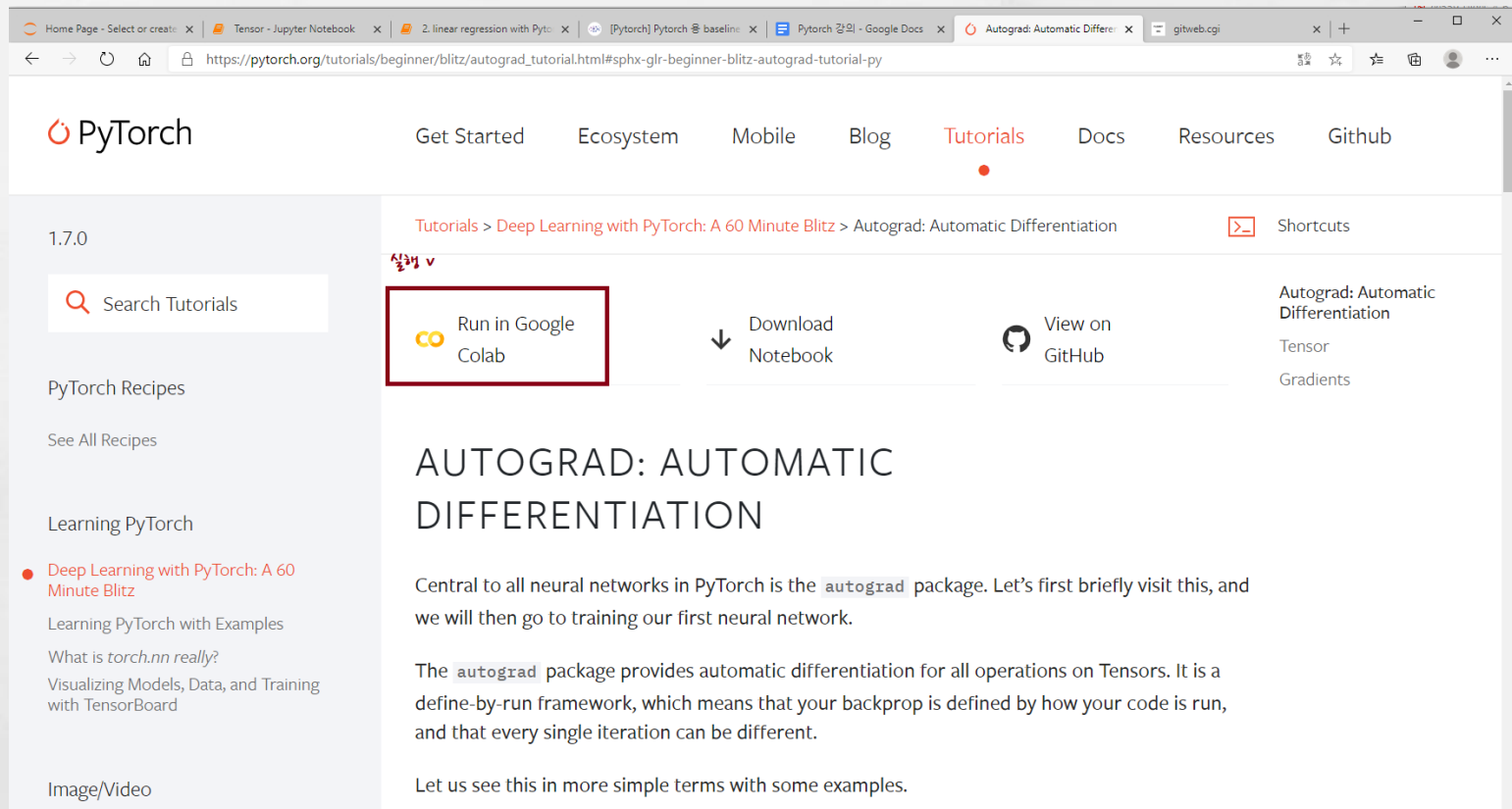
! In-place 연산은 gradient 연산에 필요한 값을 손상시킬 수 있으며, 이는 autograd에 의해 추적됨.

```
>>> x = torch.tensor([1., 2., 3.]).requires_grad_()
>>> y = x.sin()
>>> l = y.sum()
>>> l.backward()
>>> y = x.sin()
>>> y += 1
>>> l = y.sum()
>>> l.backward()
>>> y = x.sin()
>>> y *= y
>>> l = y.sum()
>>> l.backward()
Traceback (most recent call last):
/.../
RuntimeError: one of the variables needed for gradient computation has
been modified by an inplace operation
```

- 연산의 결과가 아닌 전체 계산에 대한 초기입력인 소위 “leaf” tensor에서도 금지됨.

7) Pytorch 실습 on Google CoLab

https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-gl-beginner-blitz-autograd-tutorial-py



The screenshot shows the PyTorch website's tutorial page for Autograd. The browser's address bar displays the URL: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-gl-beginner-blitz-autograd-tutorial-py. The page features a navigation bar with links to 'Get Started', 'Ecosystem', 'Mobile', 'Blog', 'Tutorials' (highlighted), 'Docs', 'Resources', and 'Github'. A sidebar on the left contains a search bar, 'PyTorch Recipes', 'Learning PyTorch', and a list of tutorials, with 'Deep Learning with PyTorch: A 60 Minute Blitz' selected. The main content area shows the breadcrumb 'Tutorials > Deep Learning with PyTorch: A 60 Minute Blitz > Autograd: Automatic Differentiation'. Below this, three options are presented: 'Run in Google Colab' (highlighted with a red box), 'Download Notebook', and 'View on GitHub'. To the right, a 'Shortcuts' section lists 'Autograd: Automatic Differentiation', 'Tensor', and 'Gradients'. The main heading is 'AUTOGRAD: AUTOMATIC DIFFERENTIATION'. The introductory text states: 'Central to all neural networks in PyTorch is the `autograd` package. Let's first briefly visit this, and we will then go to training our first neural network.' The next paragraph explains: 'The `autograd` package provides automatic differentiation for all operations on Tensors. It is a define-by-run framework, which means that your backprop is defined by how your code is run, and that every single iteration can be different.' The final sentence reads: 'Let us see this in more simple terms with some examples.'

감사합니다.

