머신 러닝·딥러닝에 필요한 수학 기초 with 파이썬

Automatic Differentiation

자동 미분

조준우
metamath@gmail.com

컴퓨터를 이용한 미분

- **심볼릭 방식**
  - Sympy, Mathematica, Wolfram alpha 같은 소프트웨어로 손 계산한 것처럼 도함수를 구함
- **코드 구현**
  - 직접 미분
  - 수치 미분
  - 자동 미분
- 각 방식에 대한 이론과 파이썬 구현을 알아보고 실습
- 실습코드는 jupyter notebook으로 제공

직접 미분 Analytical diff.

- **개념** : 직접 미분하여 결과를 코딩
- **장점** : 정확한 결과, 빠른 속도
- **단점** : 미분을 해야 한다!!!

```
import numpy as np

f = lambda x : (x**2 + 2*x)*np.log(x)
df = lambda x : 2*(x + 1)*np.log(x) + (x + 2)

print(f(1))
#>>> 0.0

print(df(1))
#>>> 3.0
```

$f(x) = (x^2 + 2x)\ln x$ 곱셈의 미분

$$\frac{df(x)}{dx} = (2x + 2)\ln x + (x + 2)$$

수치 미분Numerical diff.

- **개념** : 수치적 계산으로 미분 계수를 근사

전방 차분법

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, \cdots, x_i + \Delta x_i, \cdots, x_n) - f(x_1, \cdots, x_i, \cdots, x_n)}{\Delta x_i}$$

후방 차분법

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, \cdots, x_i, \cdots, x_n) - f(x_1, \cdots, x_i - \Delta x_i, \cdots, x_n)}{\Delta x_i}$$

중앙 차분법

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, \cdots, x_i + \frac{1}{2}\Delta x_i, \cdots, x_n) - f(x_1, \cdots, x_i - \frac{1}{2}\Delta x_i, \cdots, x_n)}{\Delta x_i}$$

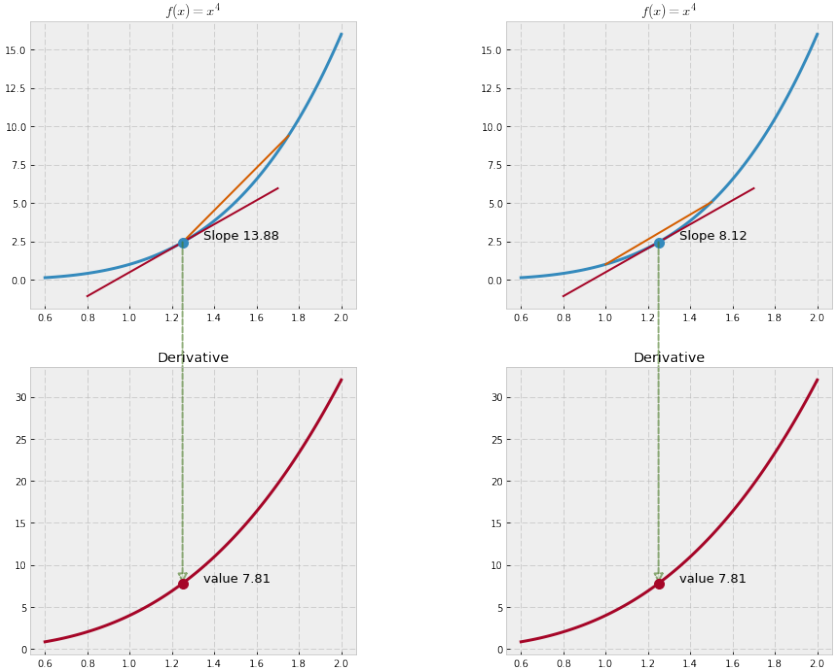
- **장점** : 구현 간단
- **단점** : 변수가 많으면 매우 느리다. 수치적으로 불안하다.
- **활용** : 머신러닝에서는 주로 그래프 방식의 미분 계산이 정확한지 확인할 목적으로 사용

수치 미분 실험

- **오차 실험** : 중앙차분법과 전방차분법의 오차 정도를 그림으로 확인

$\Delta x = 0.5$

0.5가 충분히
작아지면 훌륭한
근사치를 제공



- **정확도**
  - 중앙>전방

jupyter
autodiff.ipynb

수치 미분 - 코드

- 함수와 eps 정의 : **def** f(x) : ..., h=0.001

autodiff.ipynb

전방 차분법

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, \cdots, x_i + \Delta x_i, \cdots, x_n) - f(x_1, \cdots, x_i, \cdots, x_n)}{\Delta x_i}$$

```
diff = (f(x+h) - f(x)) / h
```

중앙 차분법

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, \cdots, x_i + \frac{1}{2}\Delta x_i, \cdots, x_n) - f(x_1, \cdots, x_i - \frac{1}{2}\Delta x_i, \cdots, x_n)}{\Delta x_i}$$

```
diff = (f(x+(h/2)) - f(x-(h/2))) / h
```

다변수 수치 미분 실험

$$f(x, y) = (x^2 + 2x) \ln y$$


```
f_xy = lambda x : (x[0]**2 + 2*x[0])*np.log(x[1])
numer_deriv(f_xy, np.array([1, 2]))

#>>> array([2.7726   , 1.4989])
```

자동 미분Automatic Diff.

- **개념** : 복잡한 함수를 작은 연산 단위로 분리하여 각각에 대해 처리
- **종류** : 포워드 모드forward mode, 리버스 모드reverse mode
  - 순전파 : 그래프를 따라가면서 함수 값을 계산
  - 역전파 : 연쇄법칙을 사용하여 미분계수를 계산
- **장점** : 복잡한 함수의 미분도 루틴의 반복으로 계산, 빠른 속도
- **단점** : 무슨 소리하는지 잘 모를 수 있다?! 구현 어려움
- **활용** : 신경망의 역전파 알고리즘

파이토치pytorch

- PyTorch is a Python-based scientific computing package serving two broad purposes:
  - A replacement for NumPy to use the power of GPUs and other accelerators.
  - An automatic differentiation library that is useful to implement neural networks

파이토치 텐서

- 텐서
  - 물리학에서 물리량을 표현하기 위한 수학 도구
  - 인공지능 분야에서는 다차원 배열
  - 넘파이 어레이 ↔ 파이토치 텐서
  - 넘파이 어레이에서 텐서로 변환하는 다양한 방법 존재

autodiff.ipynb

```
import numpy as np
import torch

np.random.seed(0)                    #❶

x = np.random.rand(6).reshape(2,3) #❷

x_tensor = torch.tensor(x)           #❸
x_from_numpy = torch.from_numpy(x)
x_Tensor = torch.Tensor(x)
x_as_tensor = torch.as_tensor(x)

print(x, x.dtype)                    #❹
#>>> [[0.5488 0.7152 0.6028]
      [0.5449 0.4237 0.6459]] float64

print(x_tensor, x_tensor.dtype, x_tensor.requires_grad)
#>>> tensor([[0.5488, 0.7152, 0.6028],
            [0.5449, 0.4237, 0.6459]], dtype=torch.float64) torch.float64 False

print(x_from_numpy, x_from_numpy.dtype, x_from_numpy.requires_grad)
#>>> tensor([[0.5488, 0.7152, 0.6028],
            [0.5449, 0.4237, 0.6459]], dtype=torch.float64) torch.float64 False

print(x_Tensor, x_Tensor.dtype, x_Tensor.requires_grad)
#>>> tensor([[0.5488, 0.7152, 0.6028],
            [0.5449, 0.4237, 0.6459]]) torch.float32 False

print(x_as_tensor, x_as_tensor.dtype, x_as_tensor.requires_grad)
#>>> tensor([[0.5488, 0.7152, 0.6028],
            [0.5449, 0.4237, 0.6459]], dtype=torch.float64) torch.float64 False
```

자동 미분 가능한 텐서

```
import numpy as np
import torch

np.random.seed(0)                  #❶

x = np.random.rand(6).reshape(2,3) #❷

x_tensor = torch.tensor(x)         #❸
x_from_numpy = torch.from_numpy(x)
x_Tensor = torch.Tensor(x)
x_as_tensor = torch.as_tensor(x)

print(x, x.dtype)                  #❹
#>>> [[0.5488 0.7152 0.6028]
      [0.5449 0.4237 0.6459]] float64

print(x_tensor, x_tensor.dtype, x_tensor.requires_grad)
#>>> tensor([[0.5488, 0.7152, 0.6028],
            [0.5449, 0.4237, 0.6459]], dtype=torch.float64) torch.float64 False

print(x_from_numpy, x_from_numpy.dtype, x_from_numpy.requires_grad)
#>>> tensor([[0.5488, 0.7152, 0.6028],
            [0.5449, 0.4237, 0.6459]], dtype=torch.float64) torch.float64 False

print(x_Tensor, x_Tensor.dtype, x_Tensor.requires_grad)
#>>> tensor([[0.5488, 0.7152, 0.6028],
            [0.5449, 0.4237, 0.6459]]) torch.float32 False

print(x_as_tensor, x_as_tensor.dtype, x_as_tensor.requires_grad)
#>>> tensor([[0.5488, 0.7152, 0.6028],
            [0.5449, 0.4237, 0.6459]], dtype=torch.float64) torch.float64 False
```

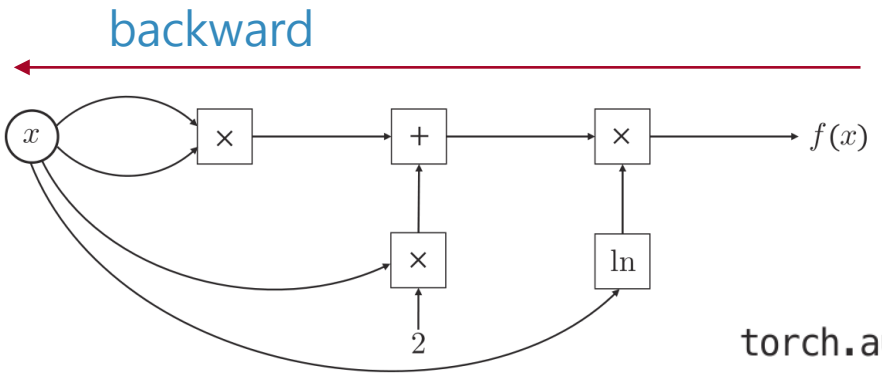
```
x_tensor_grad = torch.tensor(x, requires_grad=True)

print(x_tensor_grad, x_tensor_grad.dtype, x_tensor_grad.requires_grad)

#>>> tensor([[100.0000,   0.7152,   0.6028],
            [  0.5449,   0.4237,   0.6459]], dtype=torch.float64,
           requires_grad=True) torch.float64 True
```

미분계수를 가질 수 있는 변수

torch.autograd.backward

- 출력에 관계된 모든 입력에 대한 미분 계수 계산

$$f(x) = (x^2 + 2x)\ln x$$




```
x = torch.tensor([1.0], requires_grad=True)
f = (x**2 + 2*x) * torch.log(x)

print(x)
#>>> tensor([1.], requires_grad=True)

print(f)
#>>> tensor([0.], grad_fn=<MulBackward0>)

print(x.grad)
#>>> None
```

```
torch.autograd.backward (f, retain_graph=True)
print(x.grad)

#>>> tensor([3.])
```

torch.autograd.grad

- 특정 입력을 지정하여 미분 계수를 계산

```
df = torch.autograd.grad(f, x, retain_graph=True)
print(df)

#>>> (tensor([3.]),)
```

- 변수 여러 개 지정가능

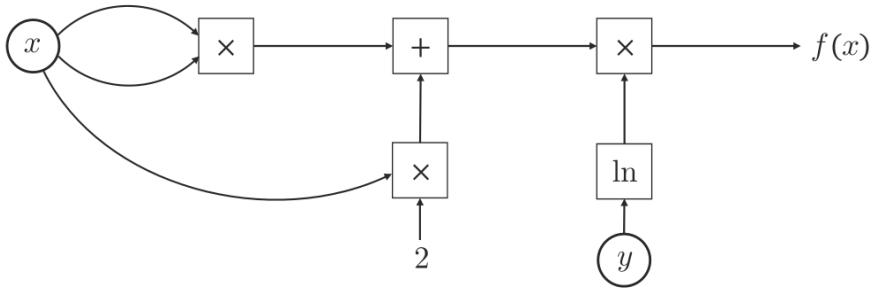
```
df = torch.autograd.grad(f, (x, x), retain_graph=True)
print(df)

#>>> (tensor([3.]), tensor([3.]))
```

파이토치 자동미분

$$f(x, y) = (x^2 + 2x) \ln y$$

```
x = torch.tensor([1.0], requires_grad=True)
y = torch.tensor([2.0], requires_grad=True)
f_xy = (x**2 + 2*x) * torch.log(y)

torch.autograd.backward(f_xy, retain_graph=True)

print(x.grad)
#>>> tensor([2.7726])

print(y.grad)
#>>> tensor([1.5000])
```

```
df = torch.autograd.grad(f_xy, (x, y), retain_graph=True)
print(df)
#>>> (tensor([2.7726]), tensor([1.5000]))
```



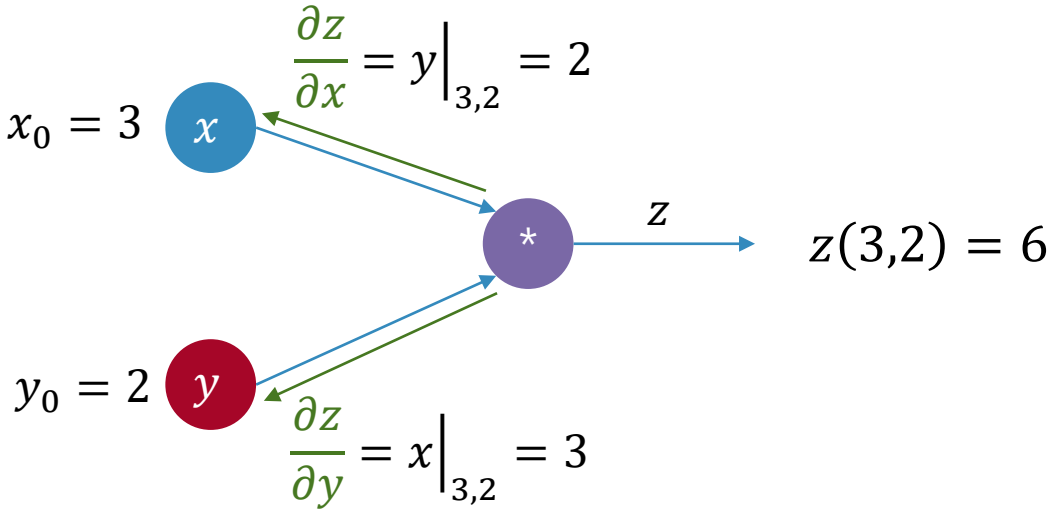
미분 방식 결과 비교

	심파이 직접미분	수치미분	파이토치 자동미분
x에 대한 미분계수	2.7726	2.7726	2.7726
y에 대한 미분계수	1.5	1.4989	1.5

- 수치미분은 오차 존재
- 파이토치는 어떻게 정확한 미분 결과를 만들어 내나?
  - 연쇄법칙 이용!

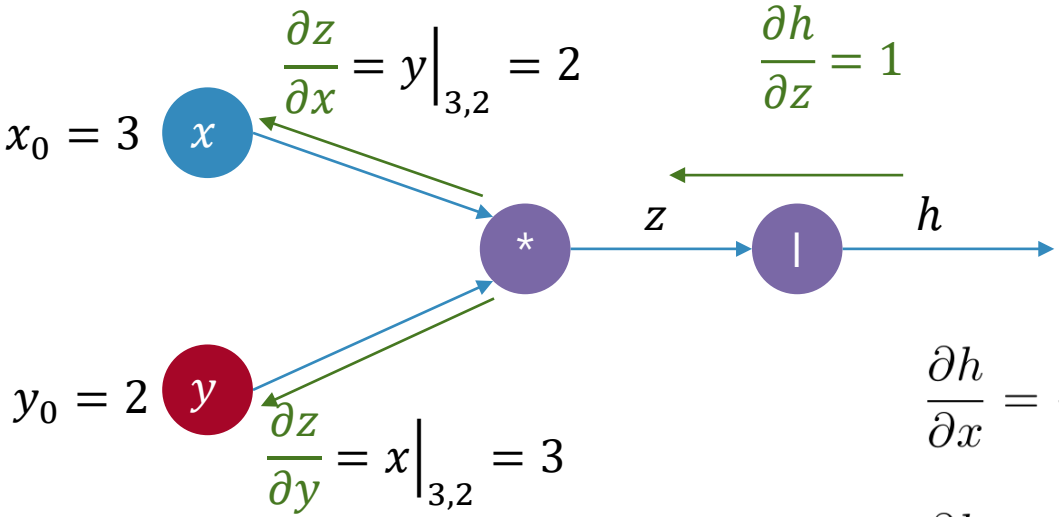
초(超)간단 예제

- **초간단 예제** $z(x, y) = x \times y$



초(超)간단 예제

- **초간단 예제** $z(x, y) = x \times y$



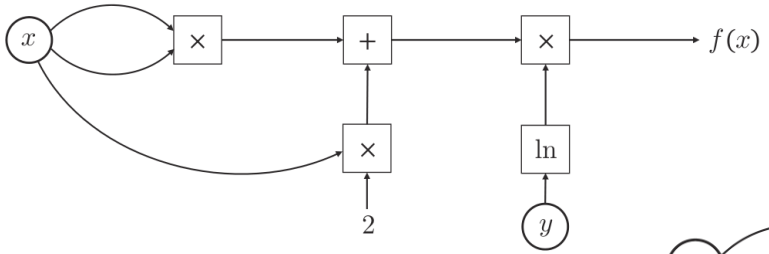
상류층 미분계수
upstream gradient

$$\frac{\partial h}{\partial x} = \frac{\partial z}{\partial x}\boxed{\frac{dh}{dz}} = y_0 \times 1 = 2 \times 1 = 2$$

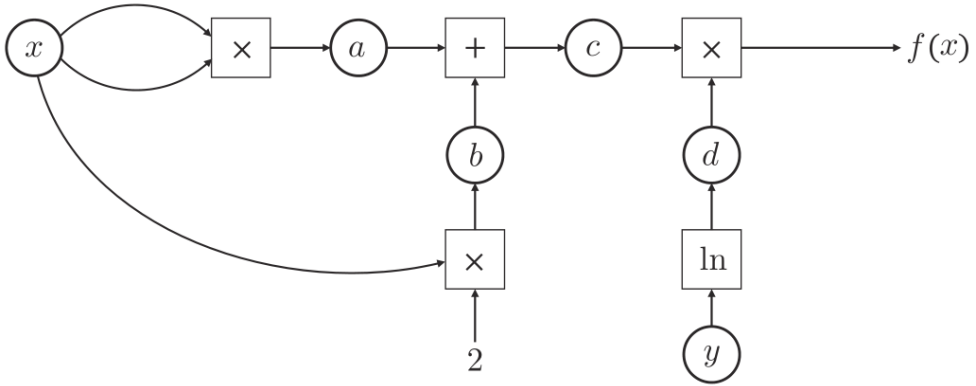
$$\frac{\partial h}{\partial y} = \frac{\partial z}{\partial y}\boxed{\frac{dh}{dz}} = x_0 \times 1 = 3 \times 1 = 3$$

간단 예제: 단위함수 세분화

- **간단 예제** $f(x, y) = (x^2 + 2x) \ln y$

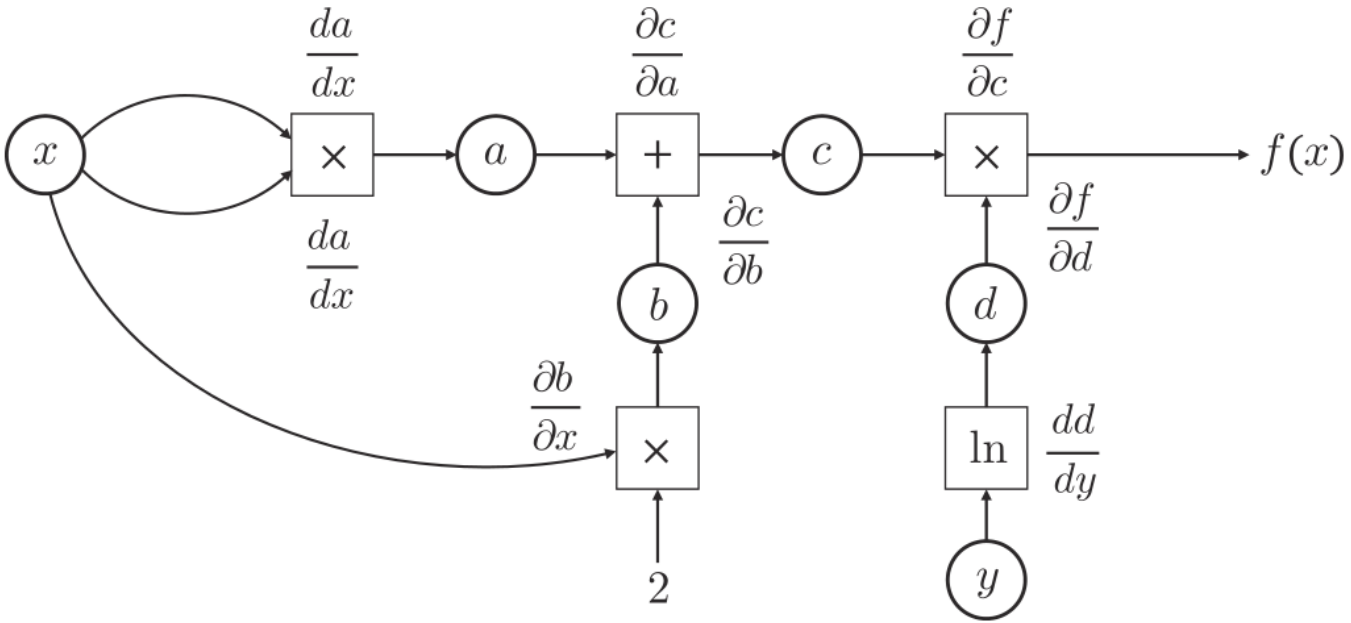


단위 함수로 세분화



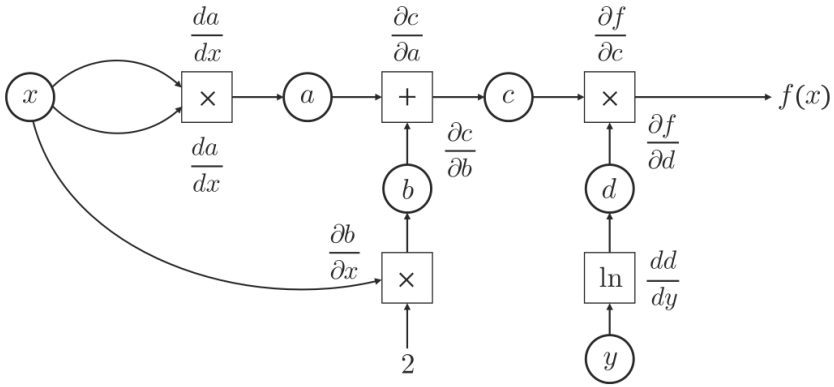
간단 예제: 단위 함수별 미분계수

- **간단 예제** $f(x, y) = (x^2 + 2x) \ln y$



간단 예제: 순전파, 역전파

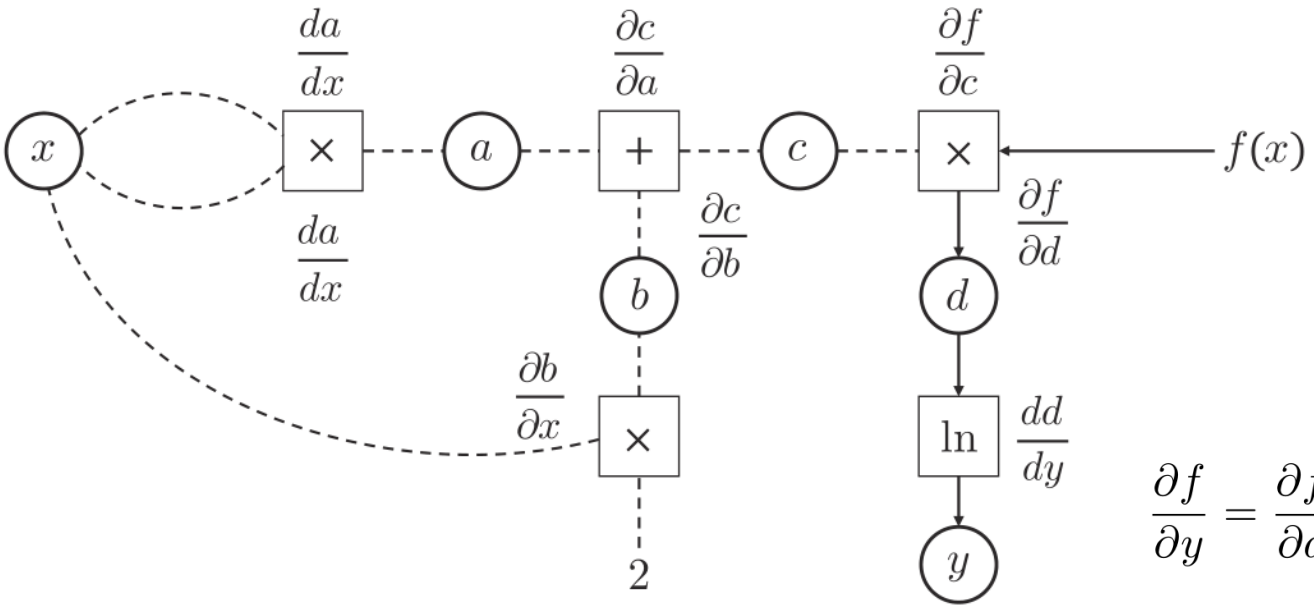
- **간단 예제** $f(x, y) = (x^2 + 2x) \ln y$



순전파	역전파
$x_0 = 1$	
$y_0 = 2$	
$a = 1$	$\frac{\partial a}{\partial x} = 1$
$b = 2$	$\frac{\partial b}{\partial x} = 2$
$c = 3$	$\frac{\partial c}{\partial a} = 1, \quad \frac{\partial c}{\partial b} = 1$
$d = 0.6931472$	$\frac{dd}{dy} = \frac{1}{2}$
$f = 2.07944164$	$\frac{\partial f}{\partial c} = 0.6931472, \quad \frac{\partial f}{\partial d} = 3$

간단 예제: $\partial f / \partial y$

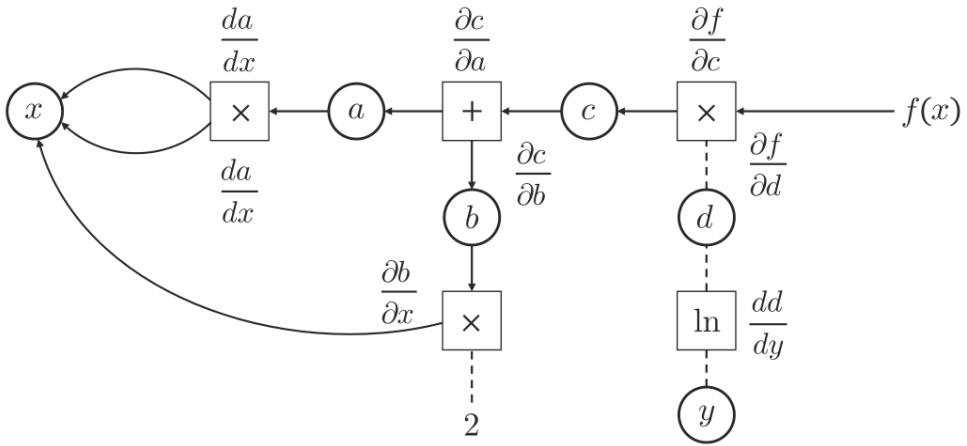
- **간단 예제** $f(x, y) = (x^2 + 2x) \ln y$



$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial d}\frac{dd}{dy} = 3 \times \frac{1}{2} = 1.5$$

간단 예제: $\partial f / \partial x$

- **간단 예제** $f(x, y) = (x^2 + 2x) \ln y$



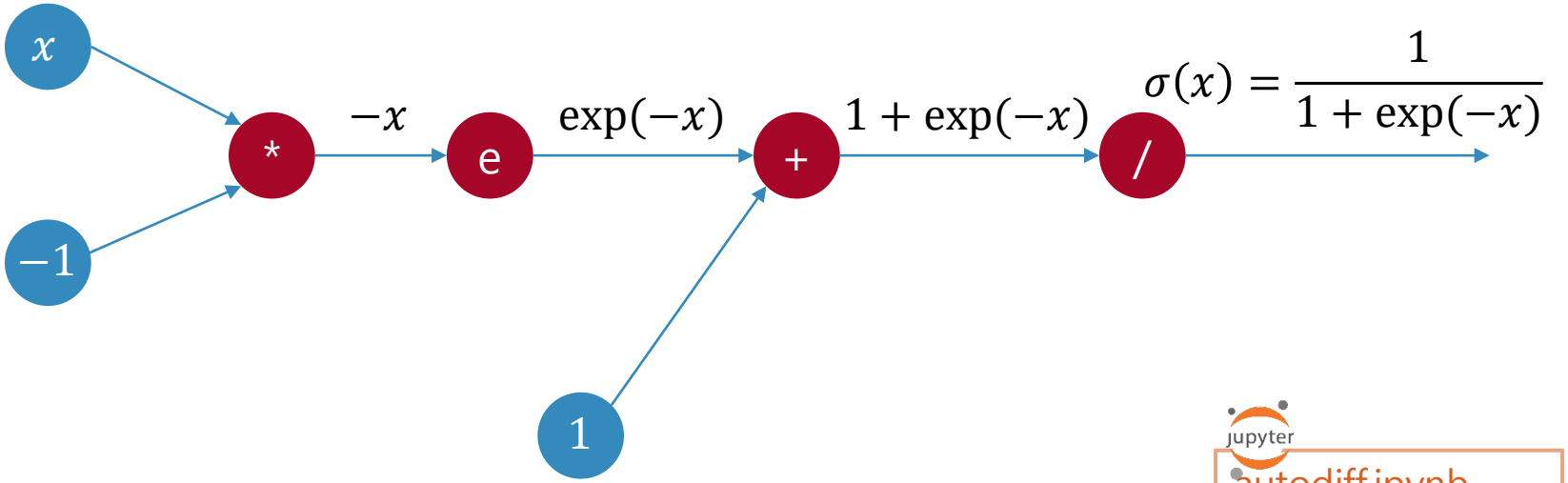
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial x}$$

$$= 0.6931472 \times 1 \times 1 + 0.6931472 \times 1 \times 1 + 0.6931472 \times 1 \times 2$$

$$= 2.7725888$$

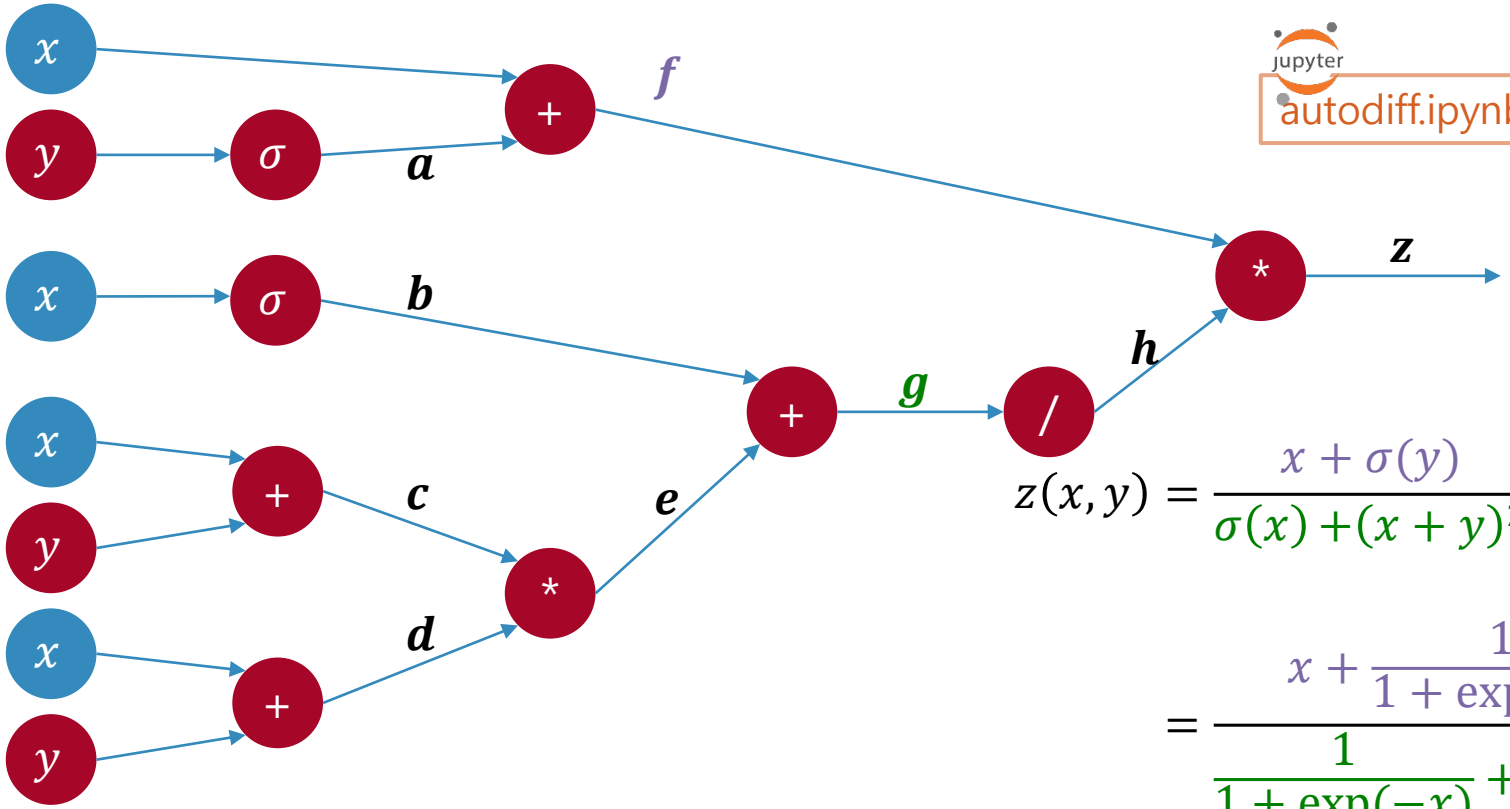
Example 2 Logistic function graph

- **Logistic function**
  - 신경망의 활성화 함수로 자주 등장하는 함수



jupyter
autodiff.ipynb

Example 3 cs231n function graph



jupyter
autodiff.ipynb

$$z(x,y) = \frac{x + \sigma(y)}{\sigma(x) + (x+y)^2}$$

$$= \frac{x + \frac{1}{1+\exp(-y)}}{\frac{1}{1+\exp(-x)} + (x+y)^2}$$

어렵게 배운 자동 미분 어디 쓰지?

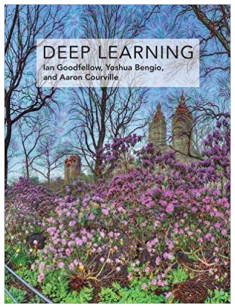
- 다음을 미분하여 간단한 선형 분류기를 만들 수 있다.

$$C(\boldsymbol{w}) = -\sum_{i=1}^{N} y_i \log[\sigma(\mathbf{x}_i \cdot \mathbf{w})] + (1 - y_i)\log[1 - \sigma(\mathbf{x}_i \cdot \mathbf{w})]$$

- 함수 시간에 나왔던 개, 고양이 분류기의 학습된 매개변수를 구할 수 있다.

Relationship between back-propagation and auto differentiation

- **Deep Learning Book,** Ian Goodfellow, Yoshua Bengio, Aaron Courville, page 206



In vector notation, this may be equivalently written as

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z, \tag{6.46}$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.

From this we see that the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$. The back-propagation algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.

Usually we do not apply the back-propagation algorithm merely to vectors, but rather to tensors of arbitrary dimensionality. Conceptually, this is exactly the same as back-propagation with vectors. The only difference is how the numbers are arranged in a grid to form a tensor. We could imagine flattening each tensor into a vector before we run back-propagation, computing a vector-valued gradient, and then reshaping the gradient back into a tensor. In this rearranged view, back-propagation is still just multiplying Jacobians by gradients.