

# Mathematics for Artificial Intelligence

3강: 경사하강법 (순한맛)

---

임성빈



인공지능대학원 & 산업공학과  
Learning Intelligent Machine Lab

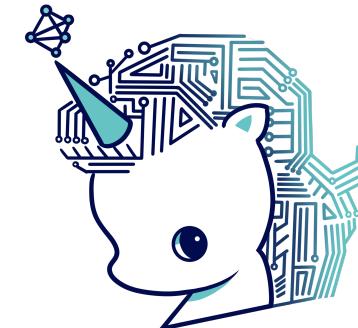


# 미분이 뭔가요?

---

- 미분(differentiation)은 변수의 움직임에 따른 함수값의 변화를 측정하기 위한 도구로 최적화에서 제일 많이 사용하는 기법입니다

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$



미분은 변화율의 극한(limit)  
으로 정의합니다

# 미분이 뭔가요?

---

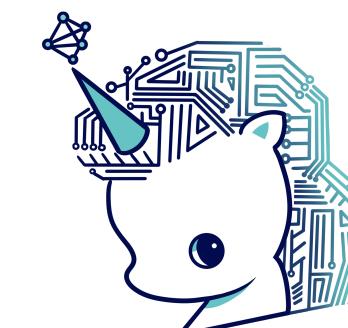
- 미분(differentiation)은 변수의 움직임에 따른 함수값의 변화를 측정하기 위한 도구로 최적화에서 제일 많이 사용하는 기법입니다

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

$$f(x) = x^2 + 2x + 3$$

$$f'(x) = 2x + 2$$

$$\frac{f(x + h) - f(x)}{h} = 2x + 2 + h$$



미분을 손으로 계산하려면 일일이  $h \rightarrow 0$  극한을 계산해야 합니다

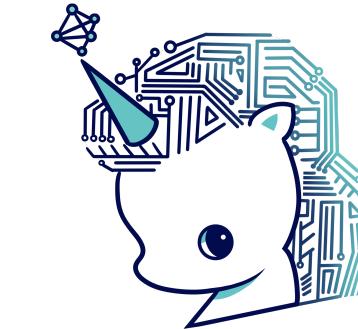
X

수식

# 미분이 뭔가요?

- 미분(differentiation)은 **변수의 움직임에 따른 함수값의 변화를 측정하기 위한 도구**로 최적화에서 제일 많이 사용하는 기법입니다
- 최근엔 미분을 손으로 직접 계산하는 대신 컴퓨터가 계산해줄 수 있습니다

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$



요즘은 **sympy.diff** 를 가지고  
미분을 컴퓨터로 계산할 수 있습니다

$$f(x) = x^2 + 2x + 3$$

$$f'(x) = 2x + 2$$

×

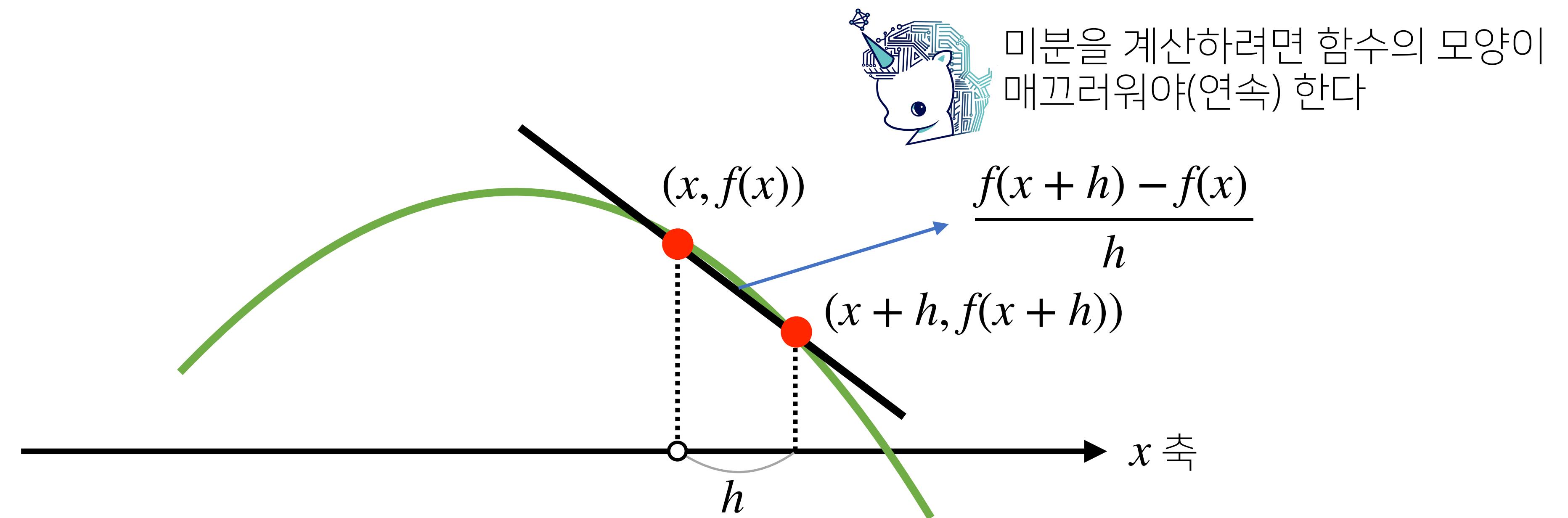
수식

코드

```
1 import sympy as sym
2 from sympy.abc import x
3
4 sym.diff(sym.poly(x**2 + 2*x + 3), x)
Poly(2*x + 2, x, domain='ZZ')
```

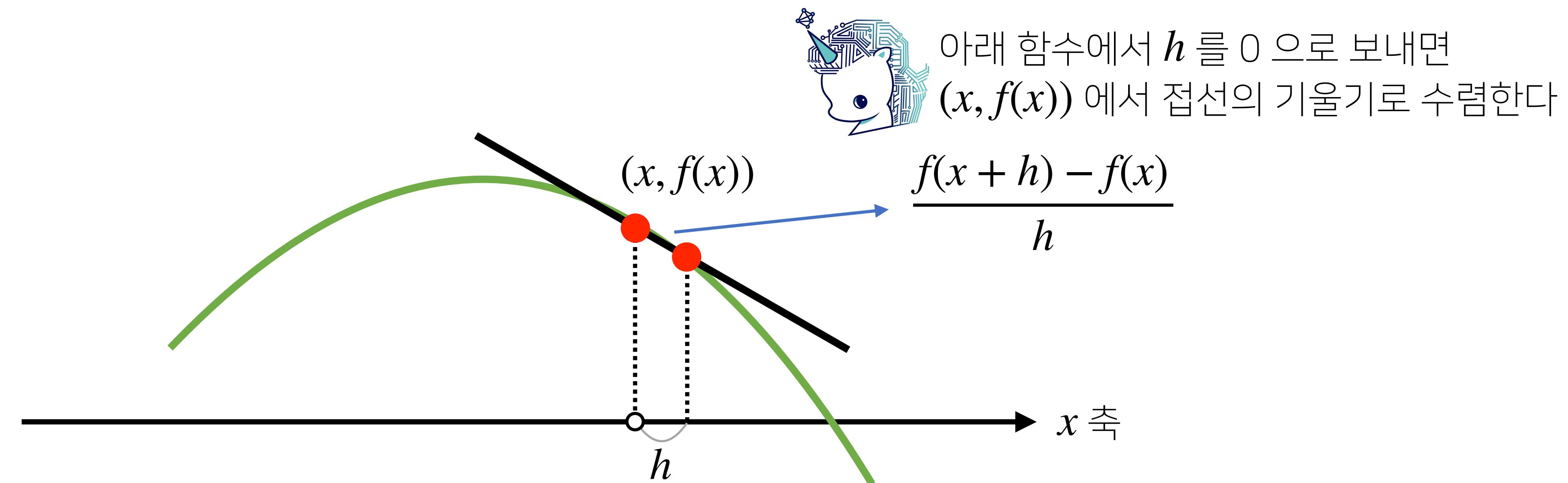
# 미분을 그림으로 이해해보자

- 미분은 함수  $f$ 의 주어진 점  $(x, f(x))$ 에서의 **접선의 기울기**를 구한다



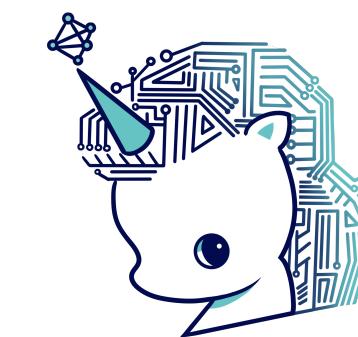
# 미분을 그림으로 이해해보자

- 미분은 함수  $f$ 의 주어진 점  $(x, f(x))$ 에서의 **접선의 기울기**를 구한다

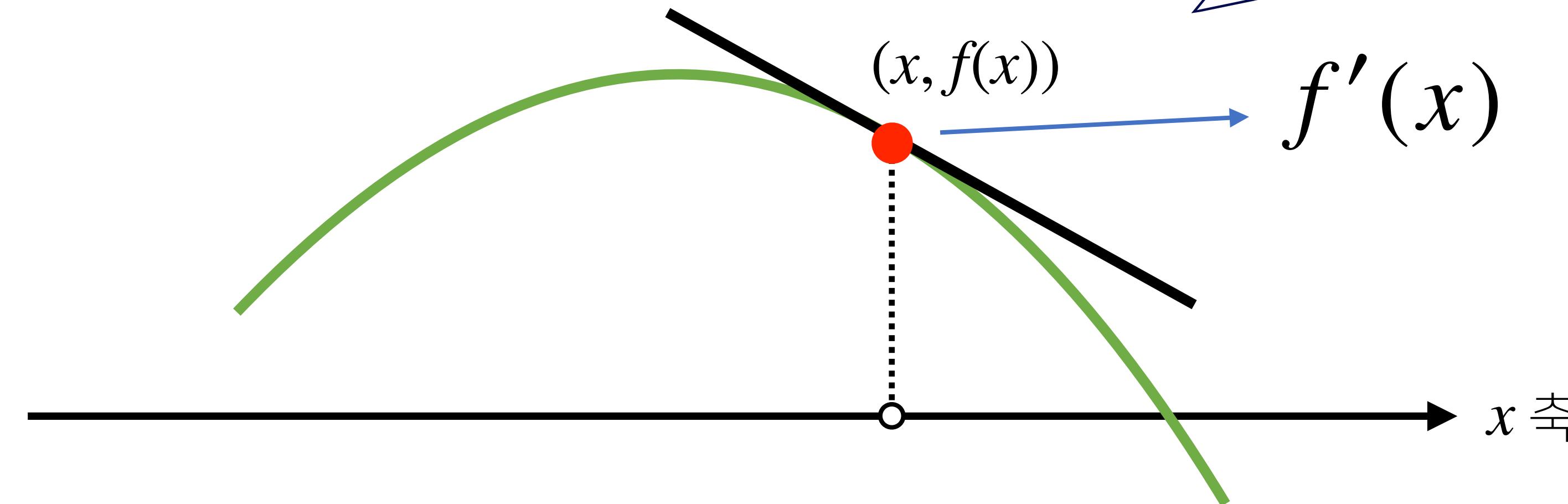


# 미분을 어디에 쓸까?

- 미분은 함수  $f$ 의 주어진 점  $(x, f(x))$ 에서의 **접선의 기울기**를 구한다
- 한 점에서 접선의 기울기를 알면 어느 방향으로 점을 움직여야 함수값이 **증가**하는지/**감소**하는지 알 수 있다

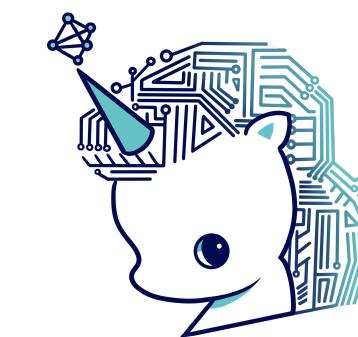


**증가**시키고 싶다면 미분값을 **더하고**  
**감소**시키고 싶으면 미분값을 **빼합니다**

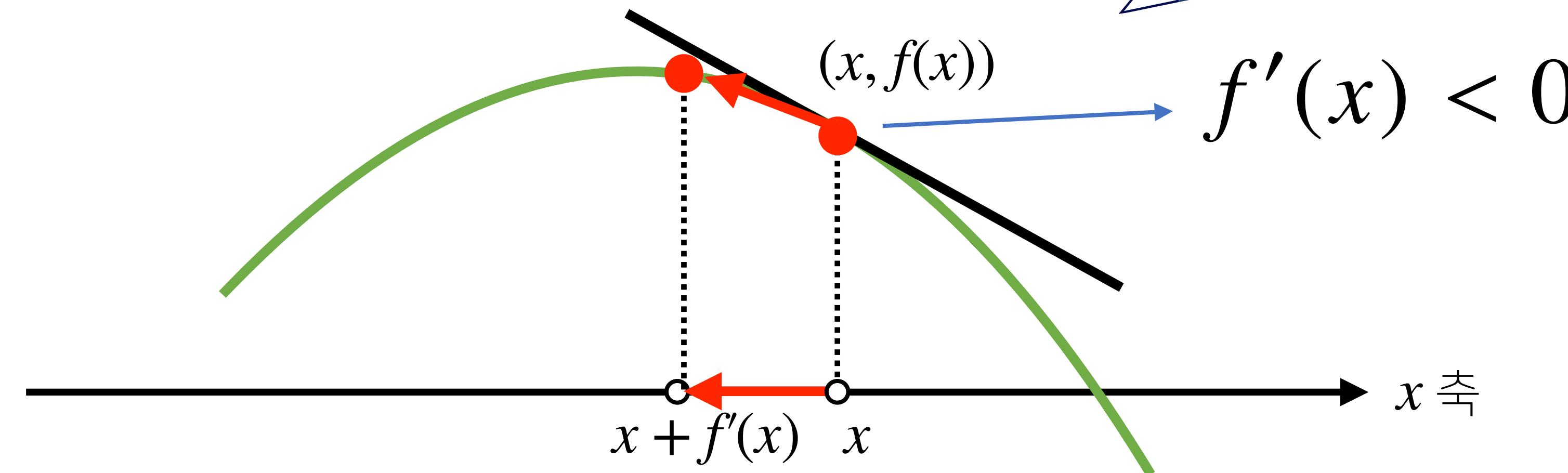


# 미분을 어디에 쓸까?

- 미분은 함수  $f$ 의 주어진 점  $(x, f(x))$ 에서의 **접선의 기울기**를 구한다
- 한 점에서 접선의 기울기를 알면 어느 방향으로 점을 움직여야 함수값이 **증가하는지/감소하는지** 알 수 있다

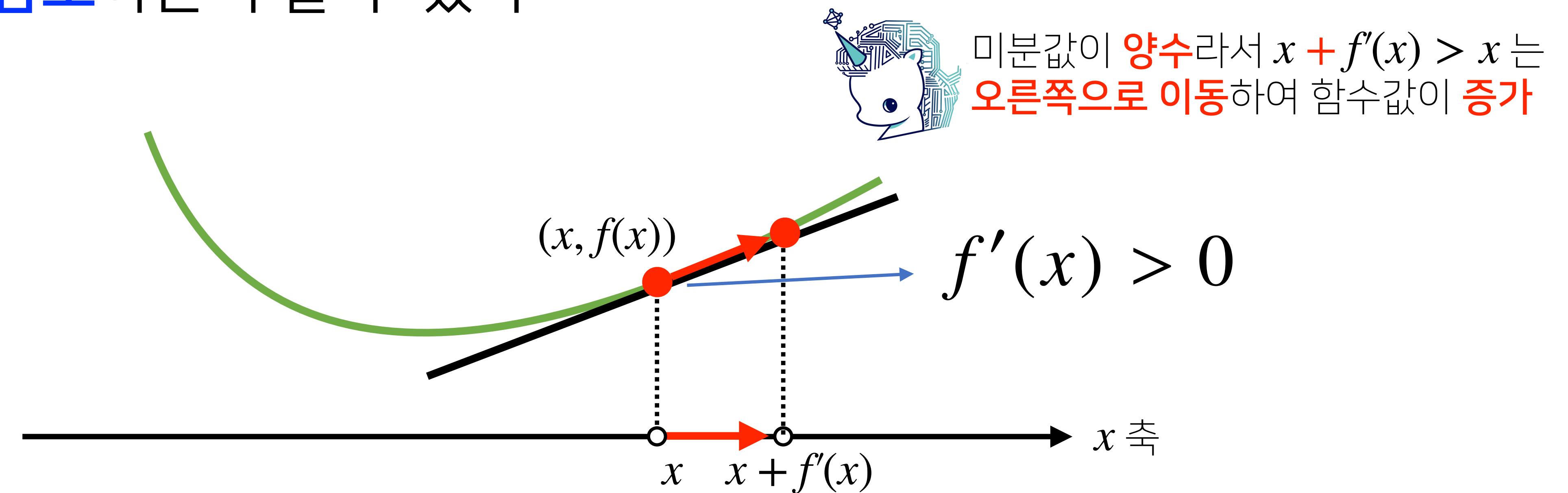


미분값이 **음수**라서  $x + f'(x) < x$ 는  
**왼쪽으로 이동**하여 함수값이 **증가**



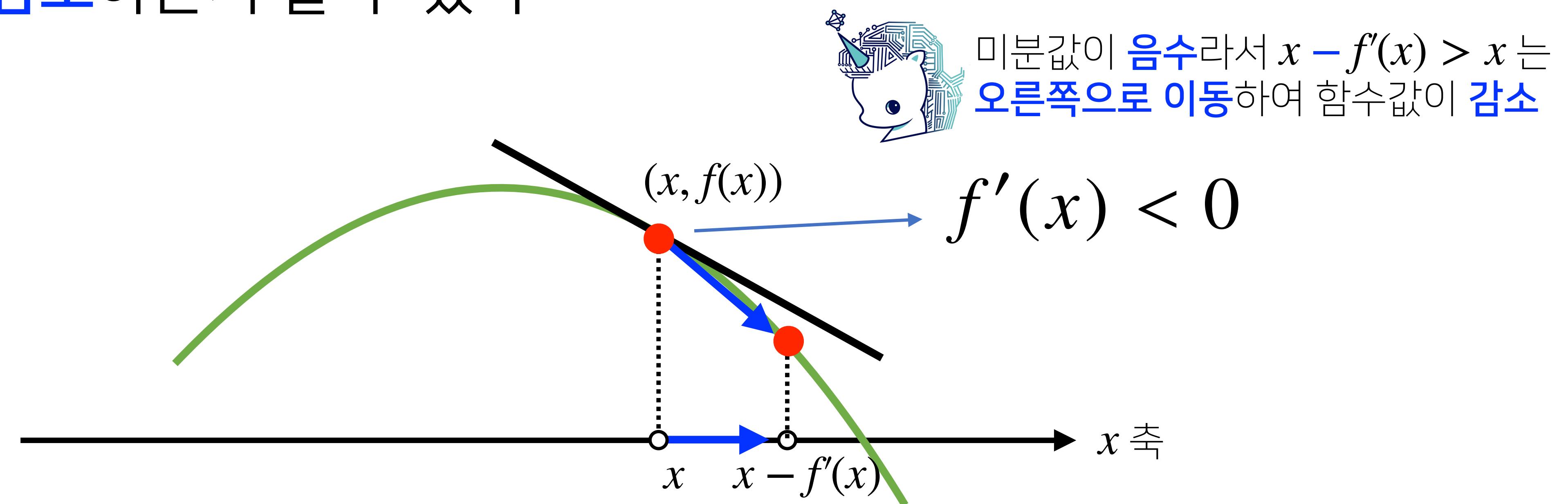
# 미분을 어디에 쓸까?

- 미분은 함수  $f$ 의 주어진 점  $(x, f(x))$ 에서의 **접선의 기울기**를 구한다
- 한 점에서 접선의 기울기를 알면 어느 방향으로 점을 움직여야 함수값이 **증가하는지/감소하는지** 알 수 있다



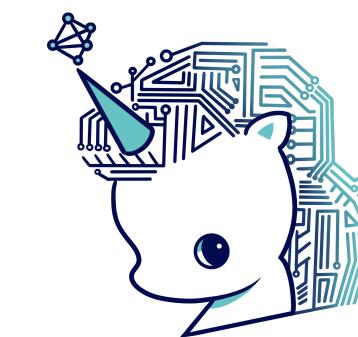
# 미분을 어디에 쓸까?

- 미분은 함수  $f$ 의 주어진 점  $(x, f(x))$ 에서의 **접선의 기울기**를 구한다
- 한 점에서 접선의 기울기를 알면 어느 방향으로 점을 움직여야 함수값이 **증가하는지/감소하는지** 알 수 있다

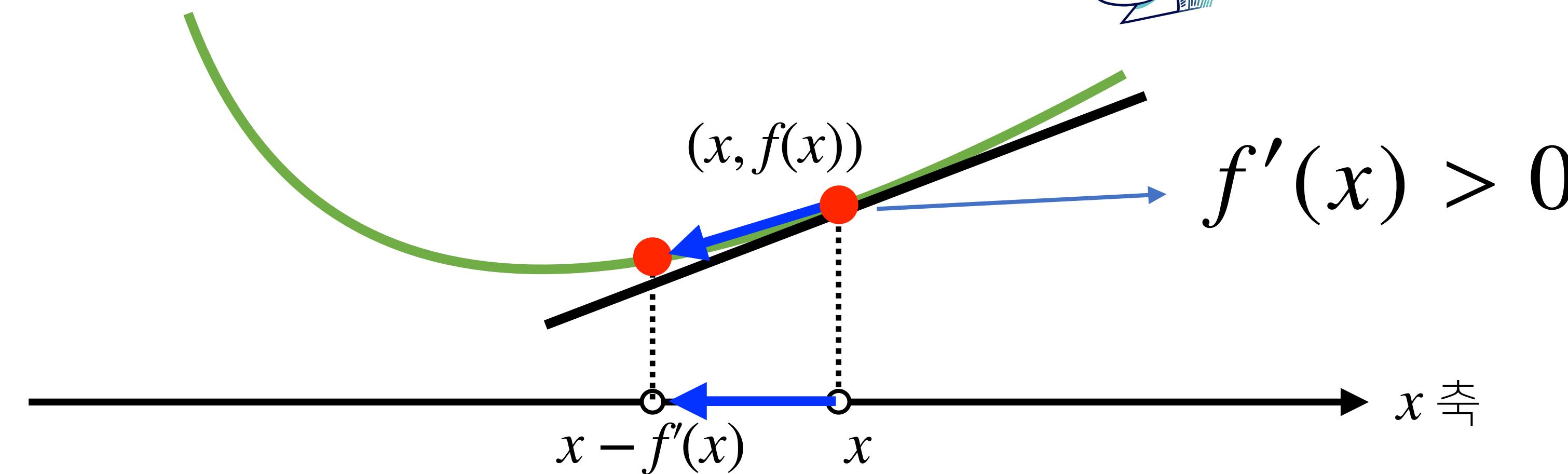


# 미분을 어디에 쓸까?

- 미분은 함수  $f$ 의 주어진 점  $(x, f(x))$ 에서의 **접선의 기울기**를 구한다
- 한 점에서 접선의 기울기를 알면 어느 방향으로 점을 움직여야 함수값이 **증가하는지/감소하는지** 알 수 있다

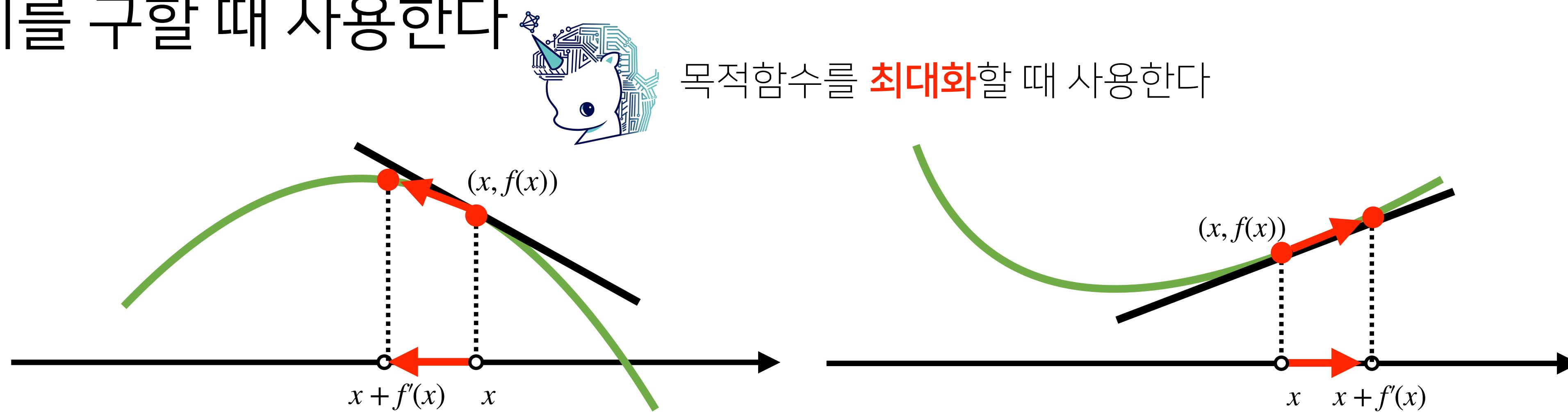


미분값이 **양수**라서  $x - f'(x) > x$ 는 **왼쪽으로 이동**하여 함수값이 **감소**



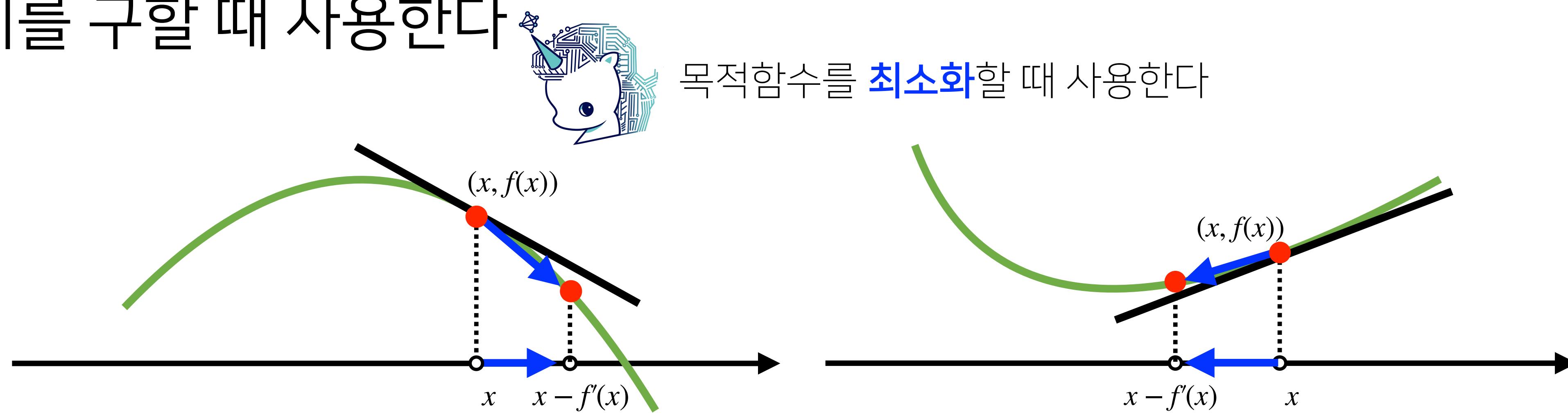
# 미분을 어디에 쓸까?

- 미분은 함수  $f$ 의 주어진 점  $(x, f(x))$ 에서의 **접선의 기울기**를 구한다
- 한 점에서 접선의 기울기를 알면 어느 방향으로 점을 움직여야 함수값이 **증가하는지/감소하는지** 알 수 있다
- **미분값을 더하면 경사상승법(gradient ascent)**이라 하며 함수의 **극대값**의 위치를 구할 때 사용한다



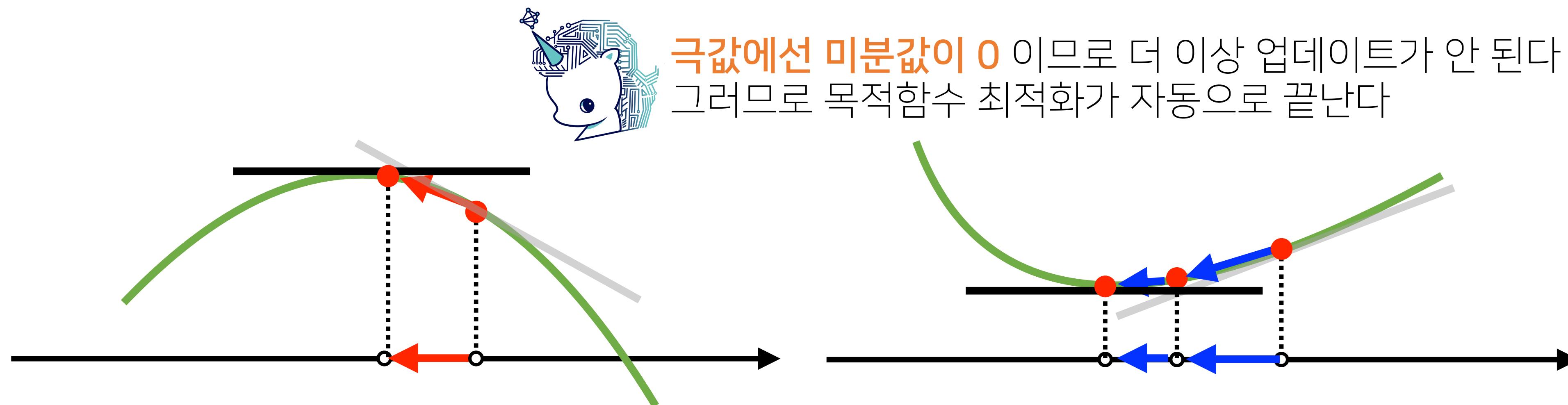
# 미분을 어디에 쓸까?

- 미분은 함수  $f$ 의 주어진 점  $(x, f(x))$ 에서의 **접선의 기울기**를 구한다
- 한 점에서 접선의 기울기를 알면 어느 방향으로 점을 움직여야 함수값이 **증가하는지/감소하는지** 알 수 있다
- **미분값을 빼면 경사하강법(gradient descent)**이라 하며 함수의 **극소값**의 위치를 구할 때 사용한다



# 미분을 어디에 쓸까?

- 미분은 함수  $f$ 의 주어진 점  $(x, f(x))$ 에서의 **접선의 기울기**를 구한다
- 한 점에서 접선의 기울기를 알면 어느 방향으로 점을 움직여야 함수값이 **증가하는지/감소하는지** 알 수 있다
- **경사상승/경사하강** 방법은 극값에 도달하면 움직임을 멈춘다



# 경사하강법: 알고리즘

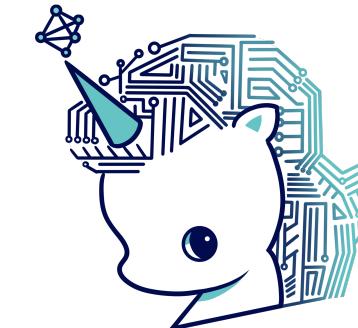
---

Input: gradient, init, lr, eps, Output: var

---

```
# gradient: 미분을 계산하는 함수  
# init: 시작점, lr: 학습률, eps: 알고리즘 종료조건
```

```
var = init  
grad = gradient(var)  
while(abs(grad) > eps):  
    var = var - lr * grad  
    grad = gradient(var)
```



컴퓨터로 계산할 때 미분이 정확히 0이 되는 것은 불가능하므로 **eps** 보다 작을 때 종료하는 조건이 필요하다.

X

# 경사하강법: 알고리즘

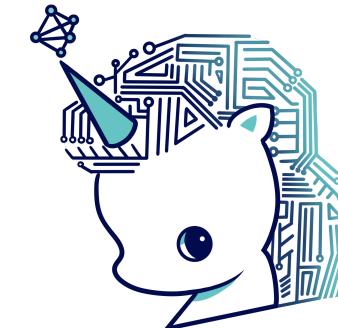
---

Input: gradient, init, lr, eps, Output: var

---

```
# gradient: 미분을 계산하는 함수  
# init: 시작점, lr: 학습률, eps: 알고리즘 종료조건
```

```
var = init  
grad = gradient(var)  
while(abs(grad) > eps):  
    var = var - lr * grad  
    grad = gradient(var)
```



이 부분이  $x - \lambda f'(x)$  을 계산하는 부분. lr은 학습률로서 미분을 통해 업데이트하는 속도를 조절한다

X

# 경사하강법: 알고리즘

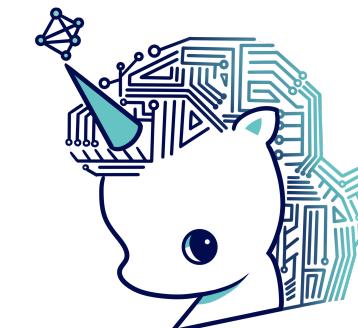
---

Input: gradient, init, lr, eps, Output: var

---

```
# gradient: 미분을 계산하는 함수  
# init: 시작점, lr: 학습률, eps: 알고리즘 종료조건
```

```
var = init  
grad = gradient(var)  
while(abs(grad) > eps):  
    var = var - lr * grad  
    grad = gradient(var)
```



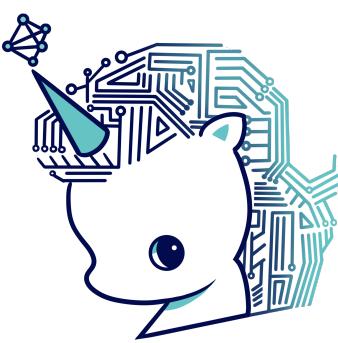
종료조건이 성립하기 전까지 미분값을 계속 업데이트한다

# 경사하강법: 알고리즘

Input: gradient, init, lr, eps, Output: var

```
# gradient: 미분을 계산하는 함수  
# init: 시작점, lr: 학습률, eps: 알고리즘 종료조건
```

```
var = init  
grad = gradient(var)  
while(abs(grad) > eps):  
    var = var - lr * grad  
    grad = gradient(var)
```



함수가  $f(x) = x^2 + 2x + 3$  일 때  
경사하강법으로 최소점을 찾는 코드

```
1 def func(val):  
2     fun = sym.poly(x**2 + 2*x + 3)  
3     return fun.subs(x, val), fun  
4  
5 def func_gradient(fun, val):  
6     _, function = fun(val)  
7     diff = sym.diff(function, x)  
8     return diff.subs(x, val), diff  
9  
10 def gradient_descent(fun, init_point, lr_rate=1e-2, epsilon=1e-5):  
11     cnt=0  
12     val = init_point  
13     diff, _ = func_gradient(fun, init_point)  
14     while np.abs(diff) > epsilon:  
15         val = val - lr_rate*diff  
16         diff, _ = func_gradient(fun, val)  
17         cnt+=1  
18  
19     print("함수: {}, 연산횟수: {}, 최소점: ({}, {})".format(fun(val)[1], cnt, val, fun(val)[0]))  
20  
21 gradient_descent(fun=func, init_point=np.random.uniform(-2,2))
```

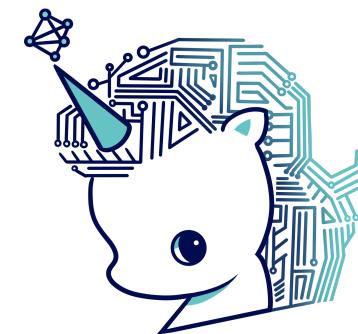
© 함수: Poly(x\*\*2 + 2\*x + 3, x, domain='ZZ'), 연산횟수: 636, 최소점: (-0.999995047967832, 2.00000000002452)



# 변수가 벡터이면요?

- 미분(differentiation)은 **변수의 움직임에 따른 함수값의 변화를 측정하기 위한 도구**로 최적화에서 제일 많이 사용하는 기법입니다
- 벡터가 입력인 다변수 함수의 경우 **편미분(partial differentiation)**을 사용한다

$$\partial_{x_i} f(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}$$



$\mathbf{e}_i$ 는  $i$  번째 값만 1이고  
나머지는 0인 단위벡터

$$f(x, y) = x^2 + 2xy + 3 + \cos(x + 2y)$$

$$\partial_x f(x, y) = 2x + 2y - \sin(x + 2y)$$

×

수식

코드

```
1 import sympy as sym
2 from sympy.abc import x, y
3
4 sym.diff(sym.poly(x**2 + 2*x*y + 3) + sym.cos(x + 2*y), x)
2*x + 2*y - sin(x + 2*y)
```

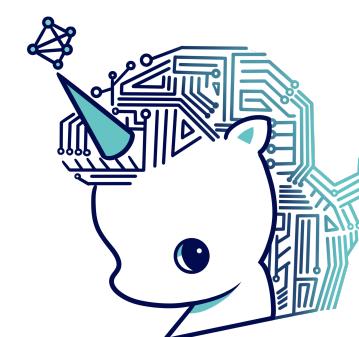
# 변수가 벡터이면요?

- 미분(differentiation)은 **변수의 움직임에 따른 함수값의 변화를 측정하기 위한 도구**로 최적화에서 제일 많이 사용하는 기법입니다
- 벡터가 입력인 다변수 함수의 경우 **편미분(partial differentiation)**을 사용한다
- 각 변수 별로 편미분을 계산한 **그레디언트(gradients)** 벡터를 이용하여 경사하강/경사상승법에 사용할 수 있다

$$\partial_{x_i} f(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}$$

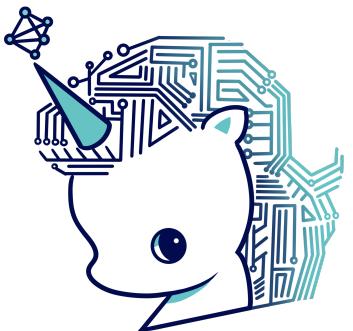
$$\nabla f = (\partial_{x_1} f, \partial_{x_2} f, \dots, \partial_{x_d} f)$$

요 기호를 **nabla** 라 부릅니다

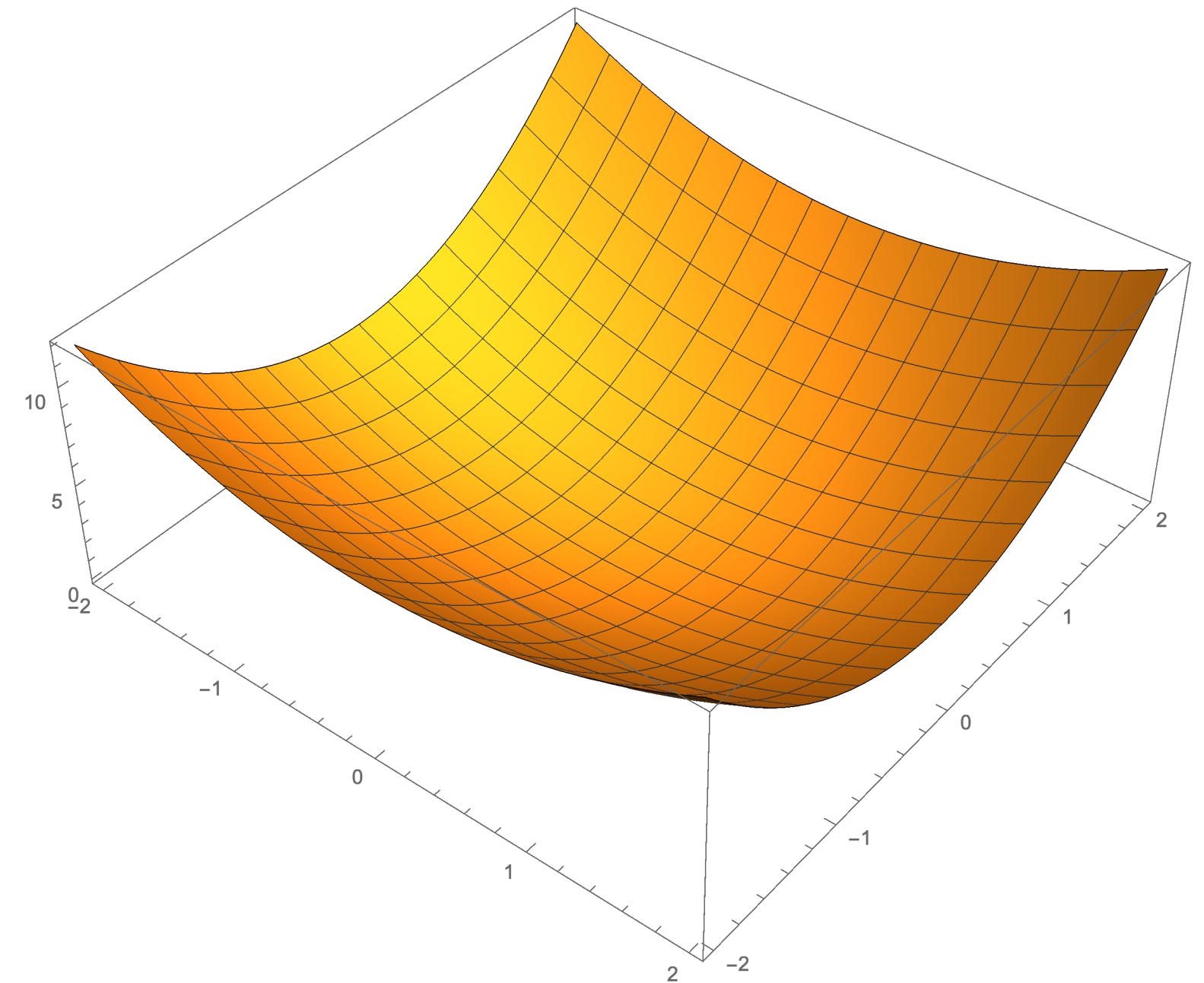


앞서 사용한 미분값인  $f'(x)$  대신 벡터  $\nabla f$ 를 사용하여 변수  $\mathbf{x} = (x_1, \dots, x_d)$ 를 동시에 업데이트 가능합니다

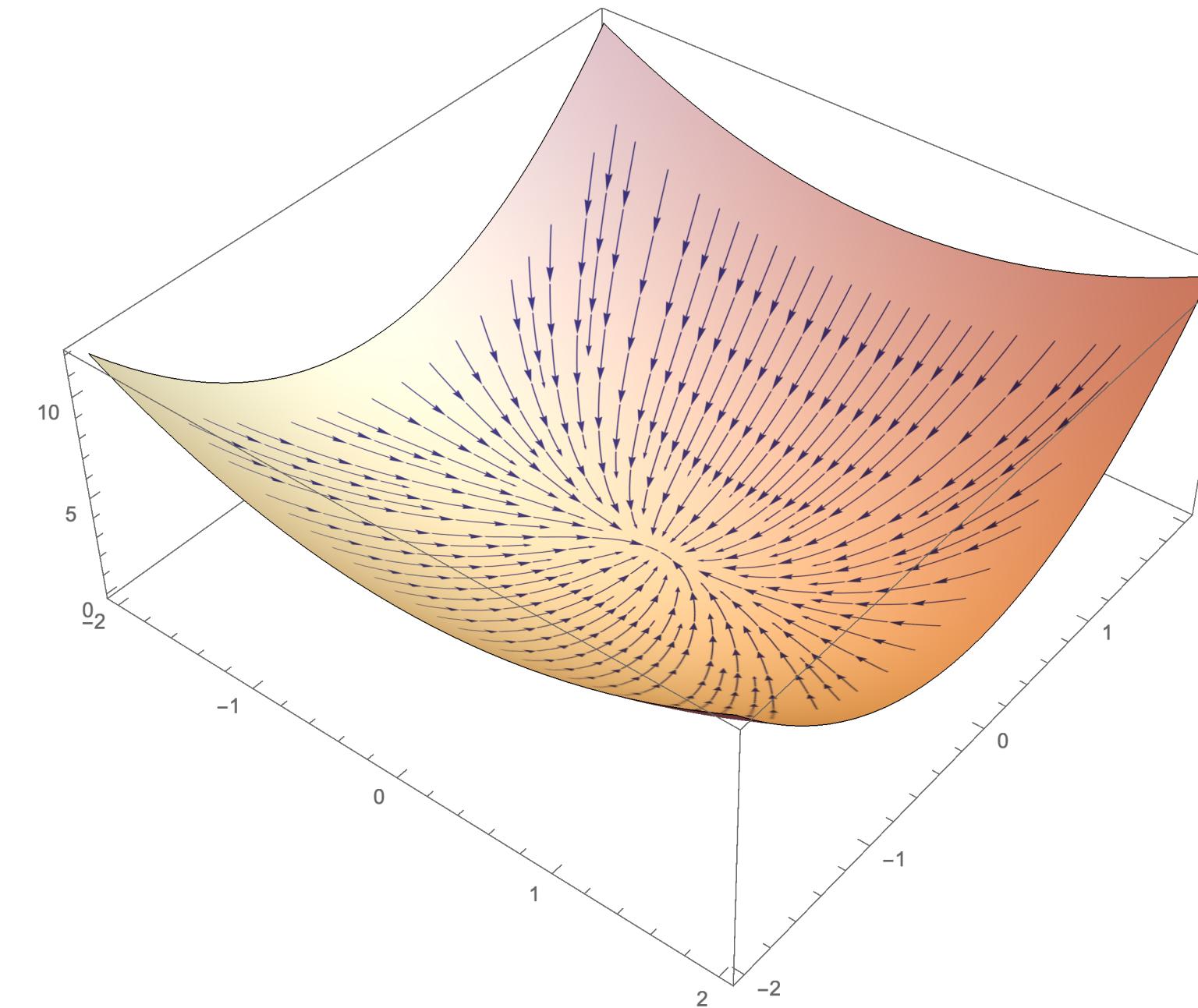
# 그레디언트 벡터가 뭐에요?



각 점  $(x, y, z)$  공간에서  $f(x, y)$  표면을 따라  
-  $\nabla f$  벡터를 그리면 아래와 같이 그려진다

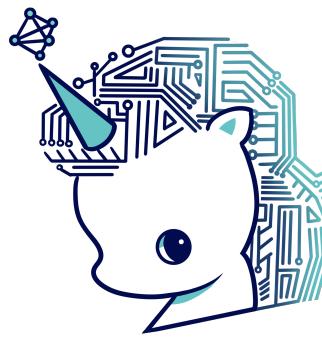


$$f(x, y) = x^2 + 2y^2$$



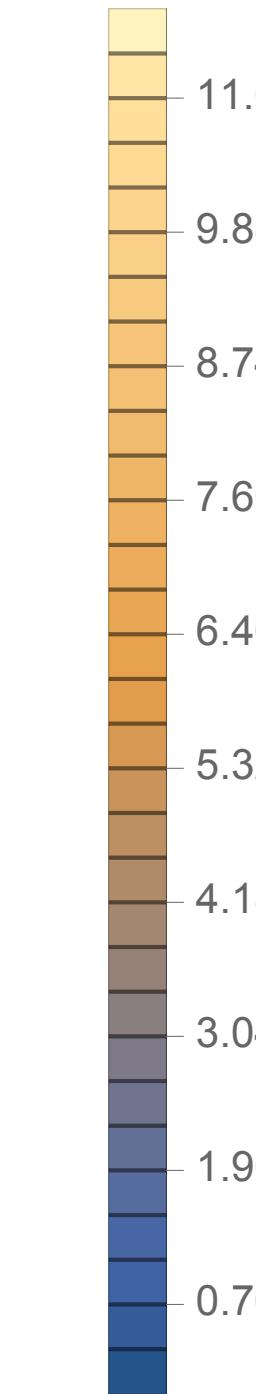
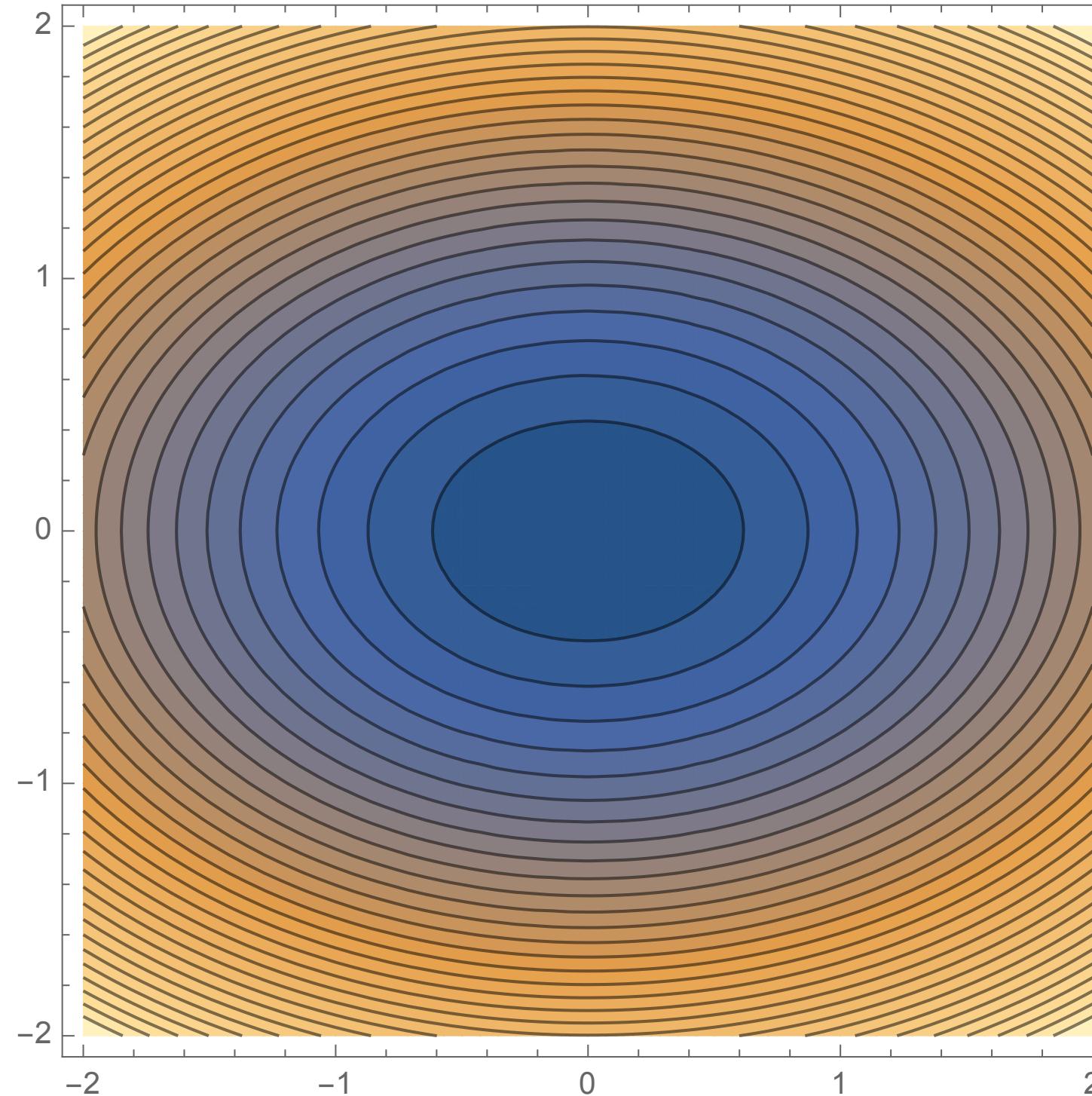
$$-\nabla f = -(2x, 4y)$$

# 그레디언트 벡터가 뭐에요?

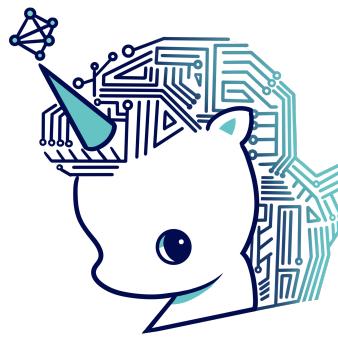


$$f(x, y) = x^2 + 2y^2$$

그레디언트를 이해하기 위해 함수  $f(x, y)$  의 등고선(contour)을 그려서 해석해봅시다



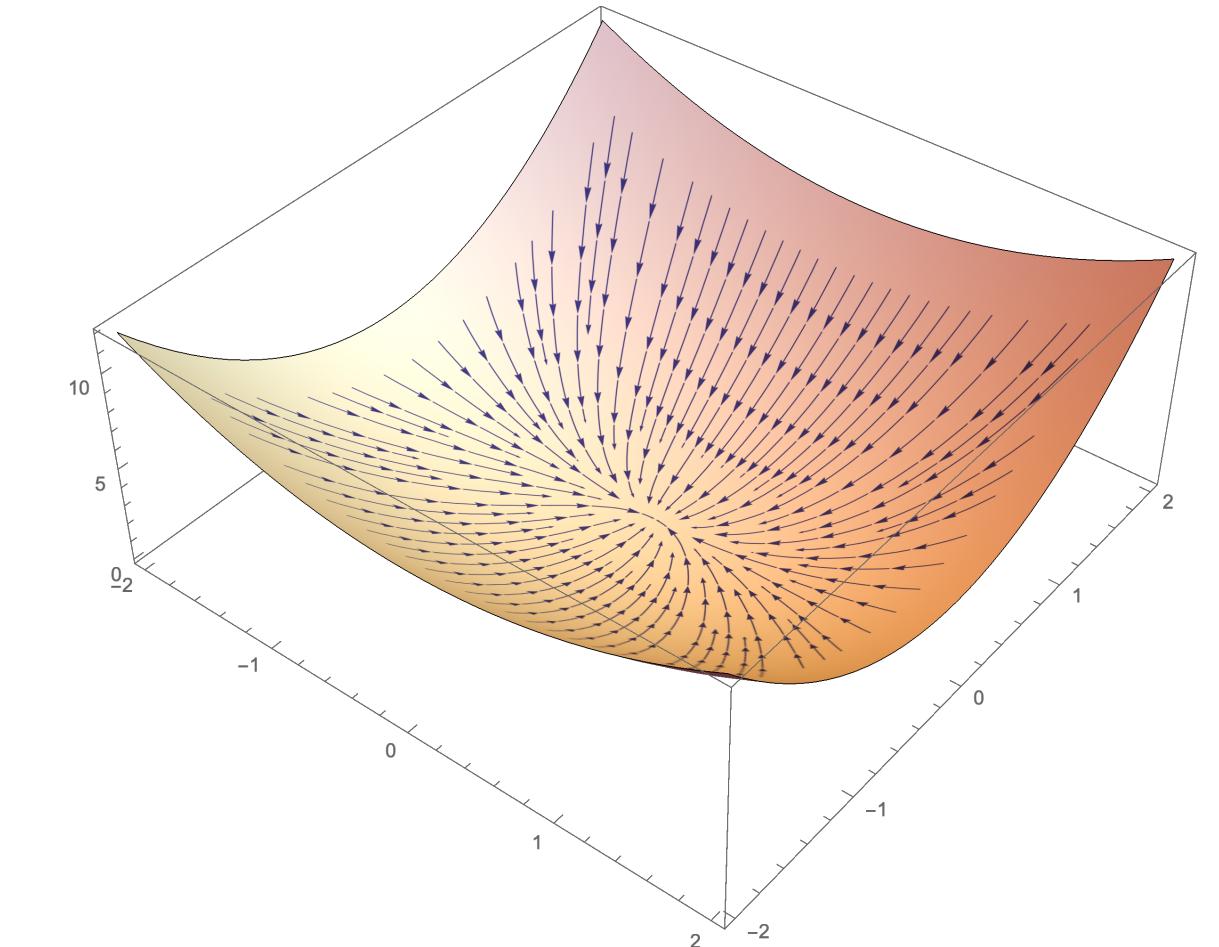
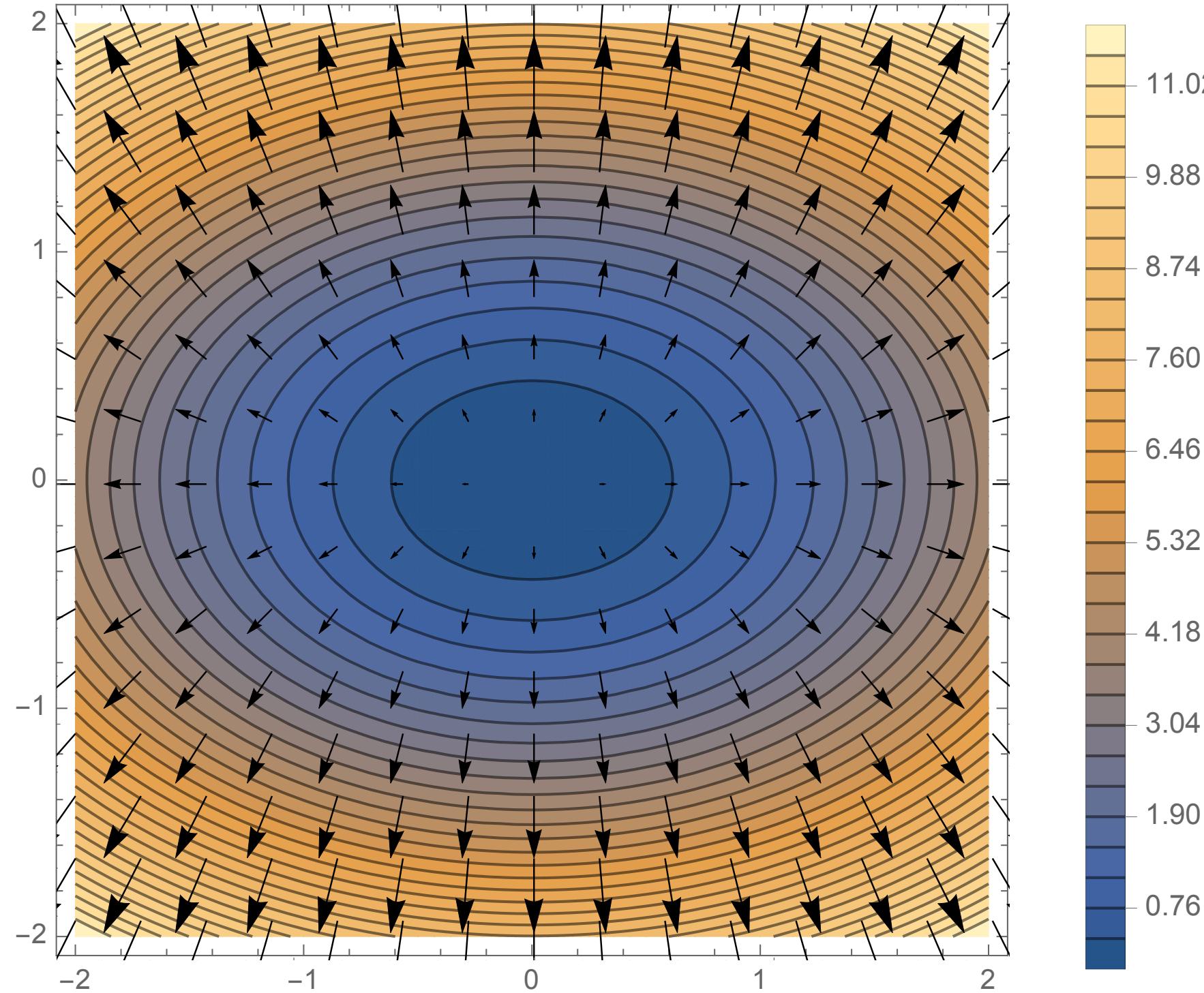
# 그레디언트 벡터가 뭐에요?



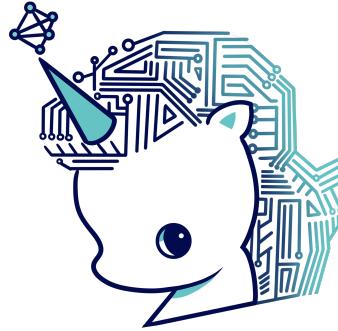
그레디언트 벡터  $\nabla f(x, y)$  는 각 점  $(x, y)$  에서  
**가장 빨리 증가하는 방향**으로 흐르게 된다

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

$$\Rightarrow \nabla f = (2x, 4y)$$



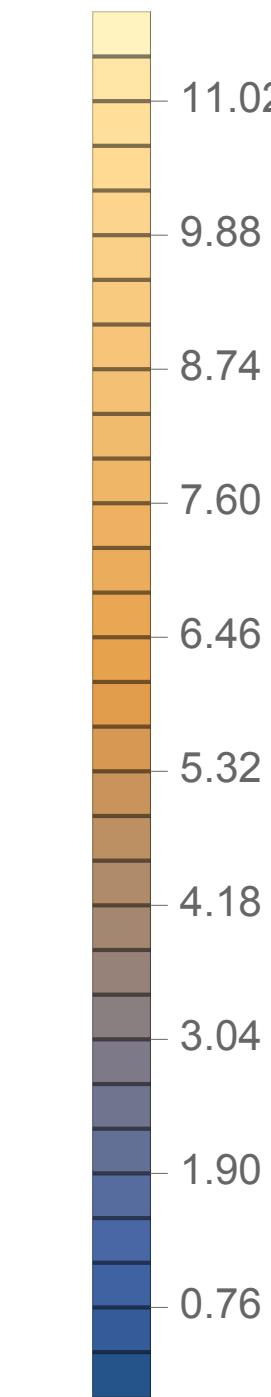
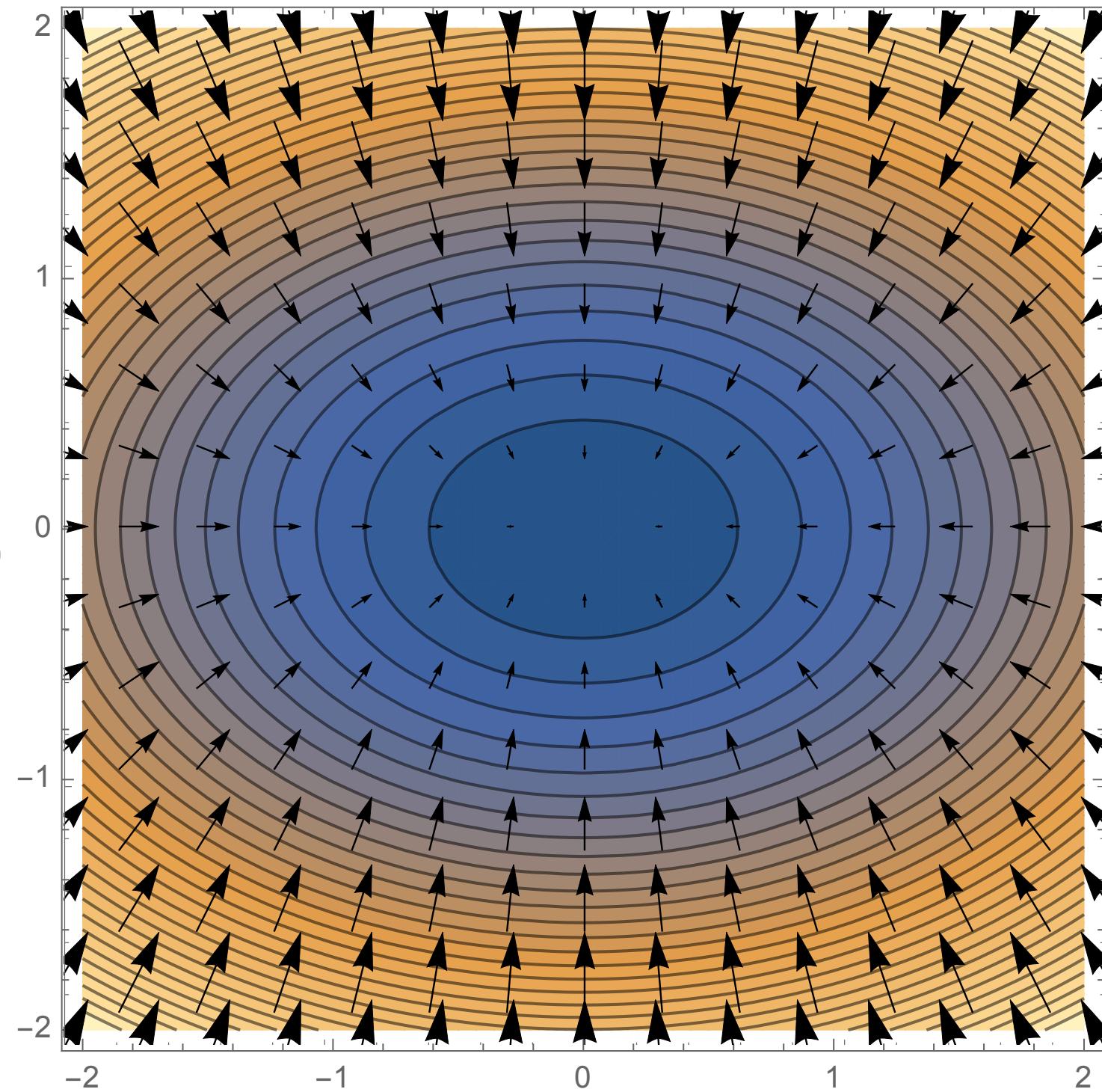
# 그레디언트 벡터가 뭐에요?



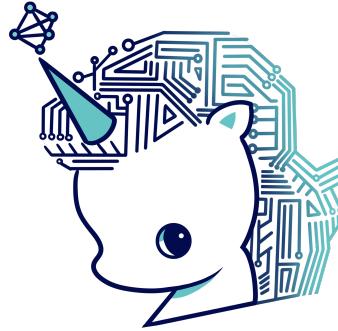
$-\nabla f$ 는  $\nabla(-f)$  량 같고 이는 각 점에서  
**가장 빨리 감소하게 되는 방향**과 같다

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

$$\Rightarrow -\nabla f = -(2x, 4y)$$



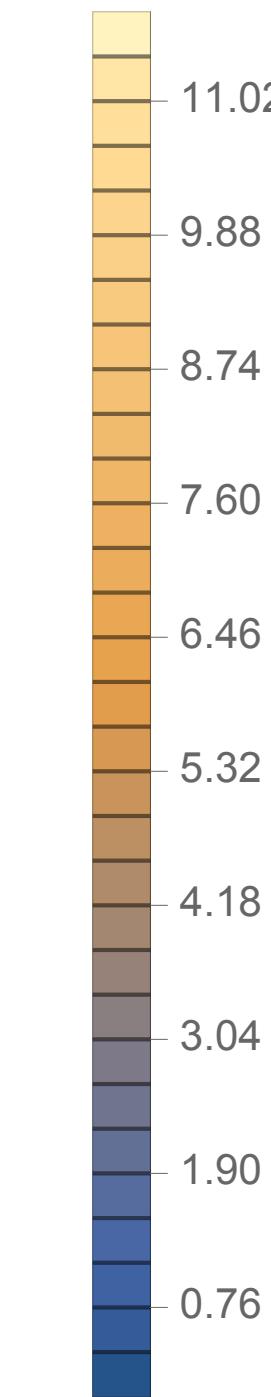
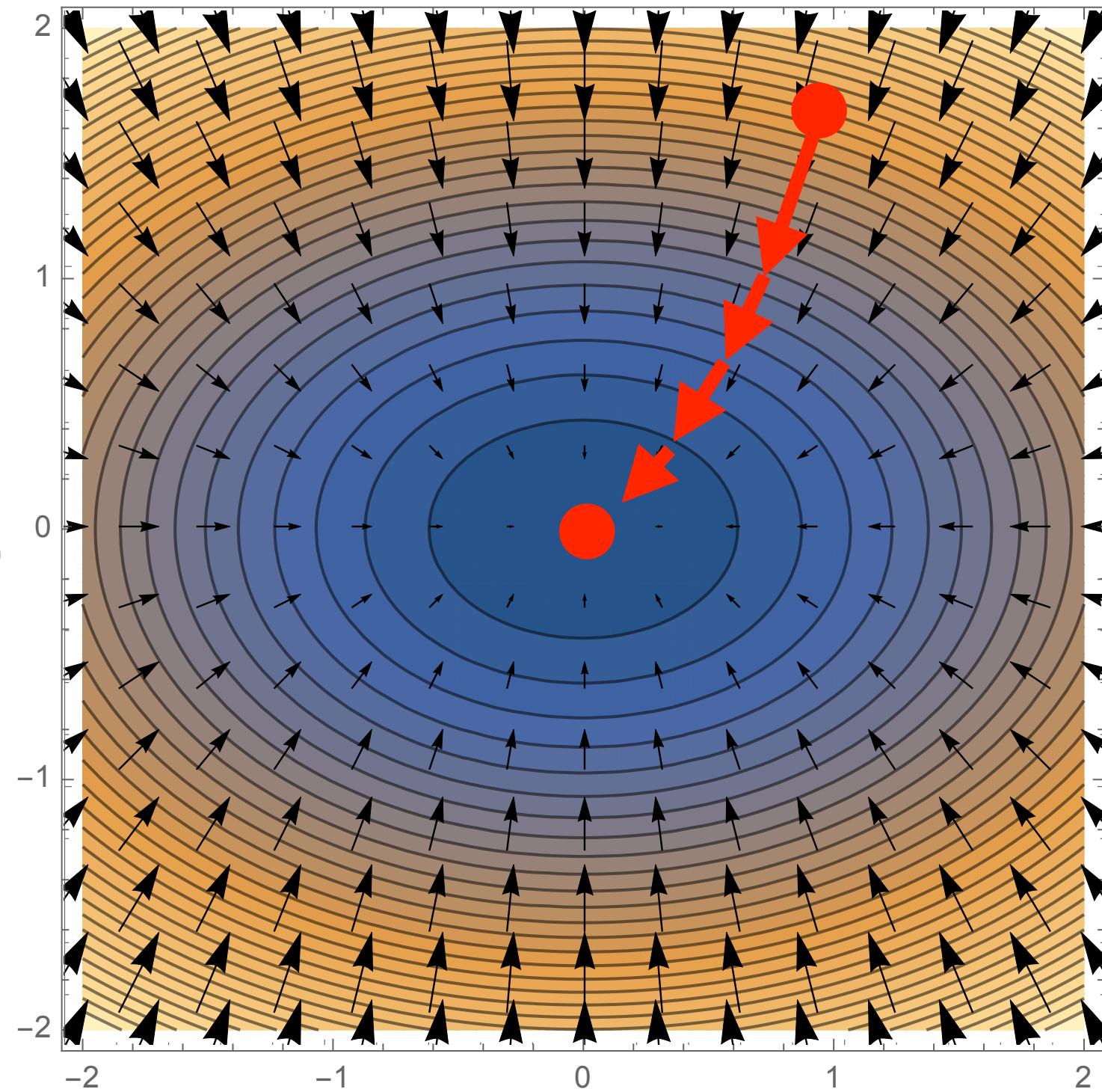
# 그레디언트 벡터가 뭐에요?



$-\nabla f$ 는  $\nabla(-f)$  량 같고 이는 각 점에서  
**가장 빨리 감소하게 되는 방향**과 같다

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

$$\Rightarrow -\nabla f = -(2x, 4y)$$



# 경사하강법: 알고리즘

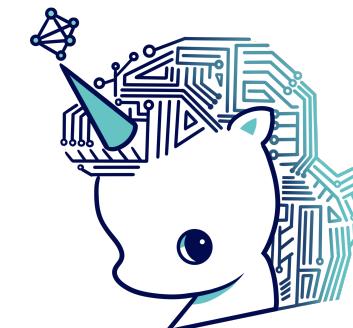
---

Input: gradient, init, lr, eps, Output: var

---

```
# gradient: 그레디언트 벡터를 계산하는 함수  
# init: 시작점, lr: 학습률, eps: 알고리즘 종료조건
```

```
var = init  
grad = gradient(var)  
while(norm(grad) > eps):  
    var = var - lr * grad  
    grad = gradient(var)
```



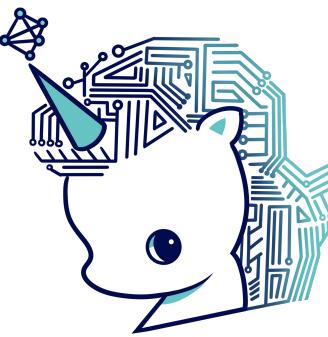
경사하강법 알고리즘은 그대로 적용된다. 그러나 벡터는 절대값 대신 노름 (norm)을 계산해서 종료조건을 설정한다

# 경사하강법: 알고리즘

Input: gradient, init, lr,

# gradient: 그레디언트 벡터를  
# init: 시작점, lr: 학습률, eps:

```
var = init
grad = gradient(var)
while(norm(grad) > eps):
    var = var - lr * grad
    grad = gradient(var)
```



함수가  $f(x) = x^2 + 2y^2$  일 때  
경사하강법으로 최소점을 찾는 코드

```
1 # Multivariate Gradient Descent
2 def eval_(fun, val):
3     val_x, val_y = val
4     fun_eval = fun.subs(x, val_x).subs(y, val_y)
5     return fun_eval
6
7 def func_multi(val):
8     x_, y_ = val
9     func = sym.poly(x**2 + 2*y**2)
10    return eval_(func, [x_, y_]), func
11
12 def func_gradient(fun, val):
13     x_, y_ = val
14     _, function = fun(val)
15     diff_x = sym.diff(function, x)
16     diff_y = sym.diff(function, y)
17     grad_vec = np.array([eval_(diff_x, [x_, y_]), eval_(diff_y, [x_, y_])], dtype=float)
18     return grad_vec, [diff_x, diff_y]
19
20 def gradient_descent(fun, init_point, lr_rate=1e-2, epsilon=1e-5):
21     cnt=0
22     val = init_point
23     diff, _ = func_gradient(fun, val)
24     while np.linalg.norm(diff) > epsilon:
25         val = val - lr_rate*diff
26         diff, _ = func_gradient(fun, val)
27         cnt+=1
28
29     print("함수: {}, 연산횟수: {}, 최소점: ({}, {})".format(fun(val)[1], cnt, val, fun(val)[0]))
30
31 pt=[np.random.uniform(-2, 2), np.random.uniform(-2, 2)]
32 gradient_descent(fun=func_multi, init_point=pt)
```

함수: Poly(x\*\*2 + 2\*y\*\*2, x, y, domain='ZZ'), 연산횟수: 606, 최소점: ([4.95901570e-06 2.88641061e-11], 2.45918366929856E-11)

X

# THE END

---

다음 시간에 보아요!