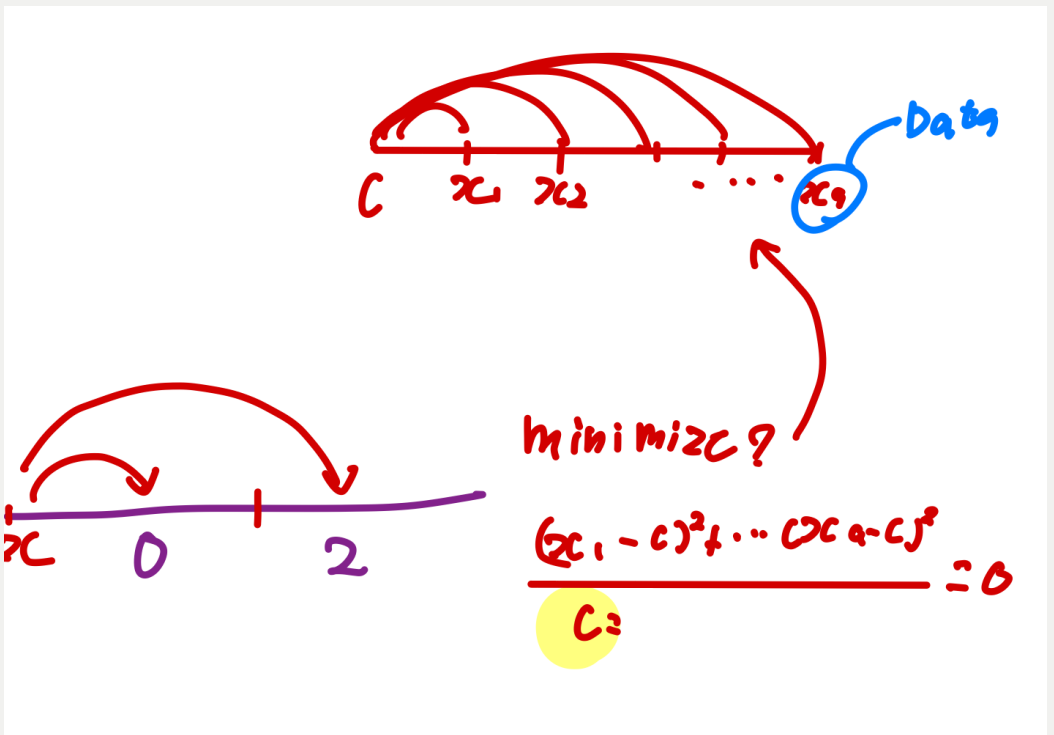
ch.8 Gradient Descent

Gradient Descent (경사 하강법)

경사하강법의 아이디어 (The Idea Behind Gradient Descent)

- 우리는 종종 함수 f를 최대화(또는 최소화)해야 할 필요가 있을 것입니다. 즉, 우리는 가능한 가장 작은(또는 가장 큰) 값을 생성하는 입력 v를 찾아야 합니다.

> 💡 함수 f를 최대화(또는 최소화)해야 합니다. 즉, 가능한 가장 작은(또는 가장 큰) 값을 만드는 입력 v를 찾아야 합니다.
>
> 이것은 많은 문제에서 발생하는 일입니다. 예를 들어, 비용 함수(cost function)를 최소화하여 모델의 성능을 최적화하거나,
>
> 이익 함수(profit function)를 최대화하여 비즈니스의 수익을 극대화하는 등의 경우에 해당

> 💡 수식
>
> $$v = \underset{x}{\operatorname{argmin}} f(x)$$
>
> 
>

- 그래디언트 ∇f(편미분의 벡터)는 함수가 가장 빠르게 감소하거나 증가하는 입력 방향을 제공합니다.

> 💡 그래디언트(기울기) 벡터인 ∇f는 함수가 가장 빠르게 감소하거나 증가하는 방향을 나타냅니다. 각 요소는 해당 변수에 대한 편미분 값.
>
> 함수가 현재 위치에서 가장 급격하게 증가하는 방향을 따르면, 그래디언트 벡터는 양수 값
>
> 반대로 함수가 가장 급격하게 감소하는 방향을 따르면, 그래디언트 벡터는 음수 값
>
> → 경사 하강법에서는 현재 위치에서 그래디언트의 반대 방향으로 이동하여 함수를 최소화

수식

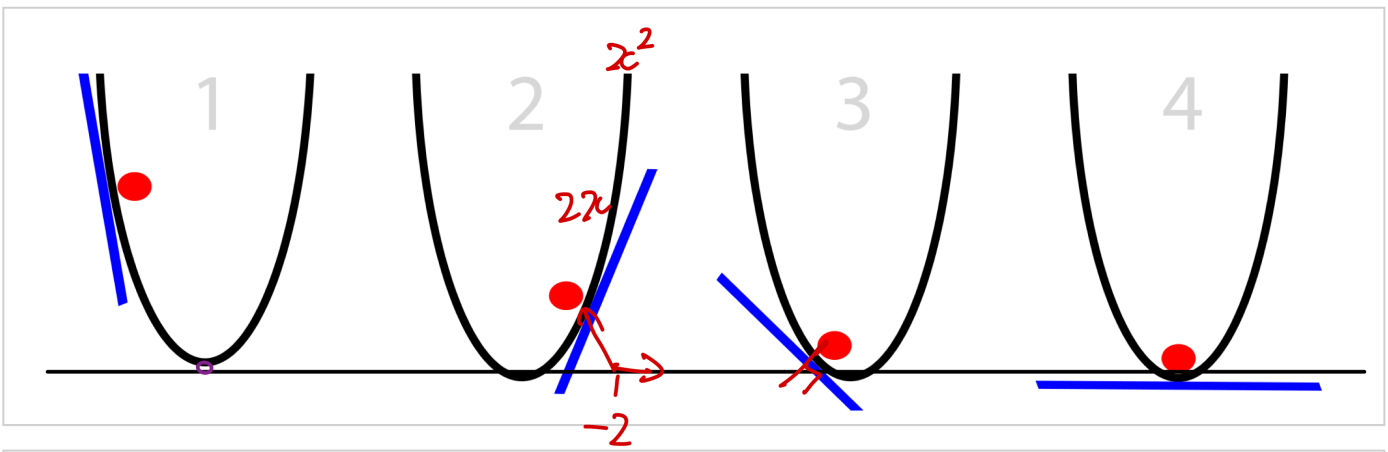
$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)$$

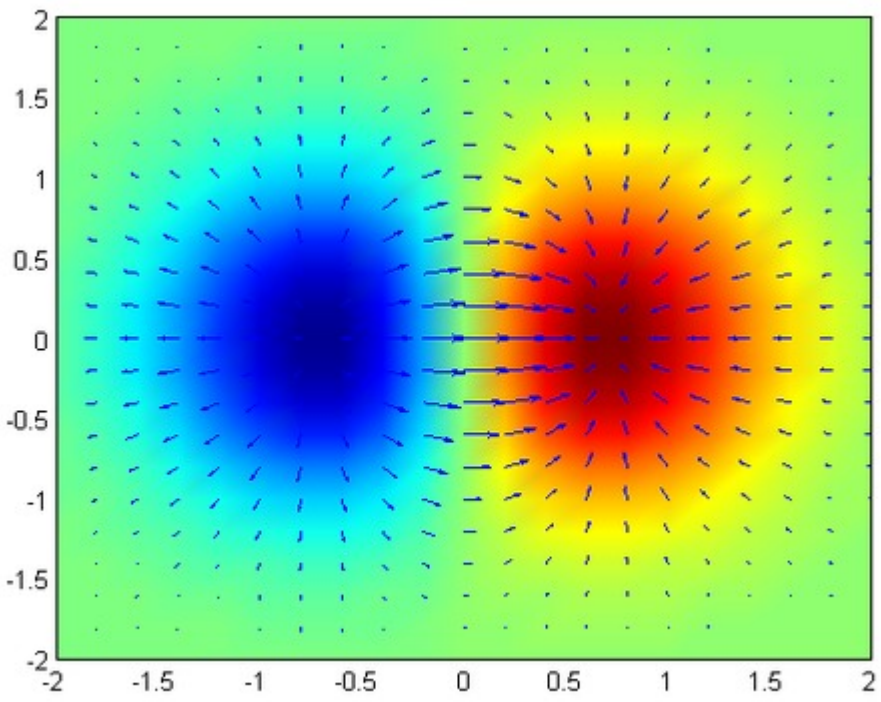
- f의 구배는 최소 또는 최대 0 : $\nabla f(\mathbf{v})=0$

설명.

함수 f의 극소점 또는 극대점에서 그래디언트(기울기)가 0임을 의미합니다. 즉, 함수가 극소점 또는 극대점에 도달하면 그래디언트 벡터의 모든 요소가 0이 됩니다.

이는 함수가 극소점(최소값)이나 극대점(최대값)에 도달했을 때, 함수의 기울기가 더 이상 증가하지 않거나 감소하지 않음을 나타냅니다. 따라서 경사 하강법 알고리즘에서 이러한 점은 최적화된 점이라고 간주.





Minimize or Maximize (최소화 또는 최대화)

- 함수를 최소화하는 한 가지 방법은 다음과 같습니다.

1. 임의의 시작점을 선택합니다.

2. 그래디언트를 계산합니다.
3. 그래디언트의 반대 방향으로 작은 단계를 따릅니다.
4. 새로운 시작점으로 반복합니다.

- 그래디언트는 함수를 가장 많이 감소시키는 방향을 나타냅니다.
- 비슷하게, 함수를 최대화하기 위해 그래디언트 방향으로 작은 단계를 취할 수 있습니다.

Using mathematical notation (수학적 표기법 사용하기)

- with monotonic sequence(수열이 단조감소)하는 성질 의미

💡 **$\mathbf{X_{n+1} = X_n - \gamma_n \nabla f(x_n)}$**

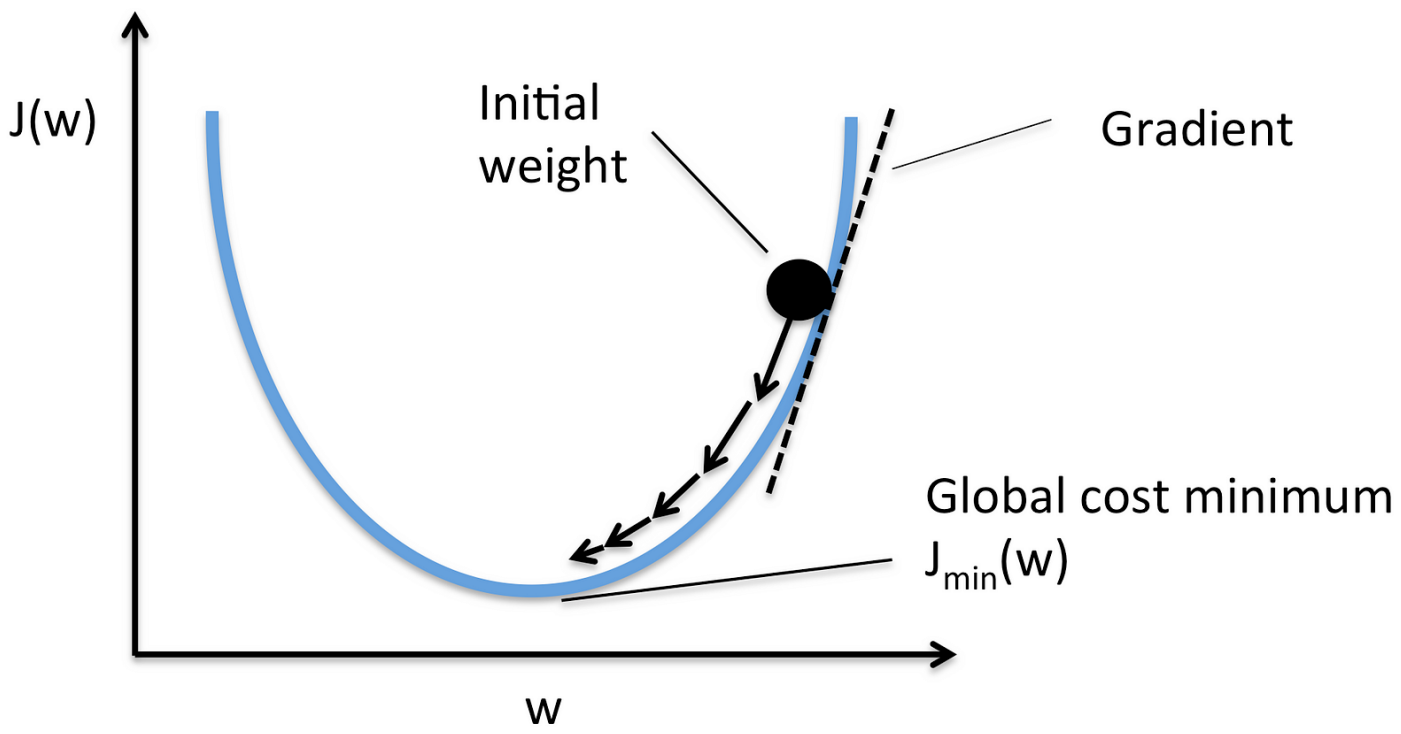
- **x**(*n*)은 n번째 반복에서의 현재 위치, *γn* 은 학습률(learning rate),
- ∇ **f(x(n))**은 현재 위치에서의 함수 f 의 그래디언트(기울기)입니다. 이를 이용하여 다음 위치인 $x(n+1)$을 계산합니다.

💡 점점 작아지는 Sequence → Minimun에 도달.. (언젠간..?)

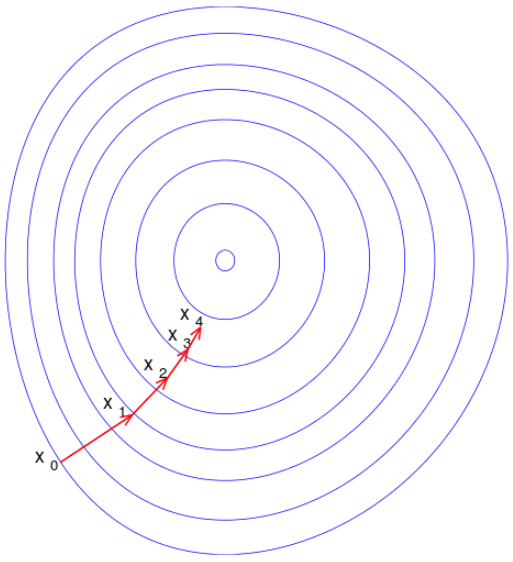
$f(x_0) \geq f(x_1) \geq \ldots$ 와 같이 현재 위치에서의 함수 값이 각 반복에서 점차 감소

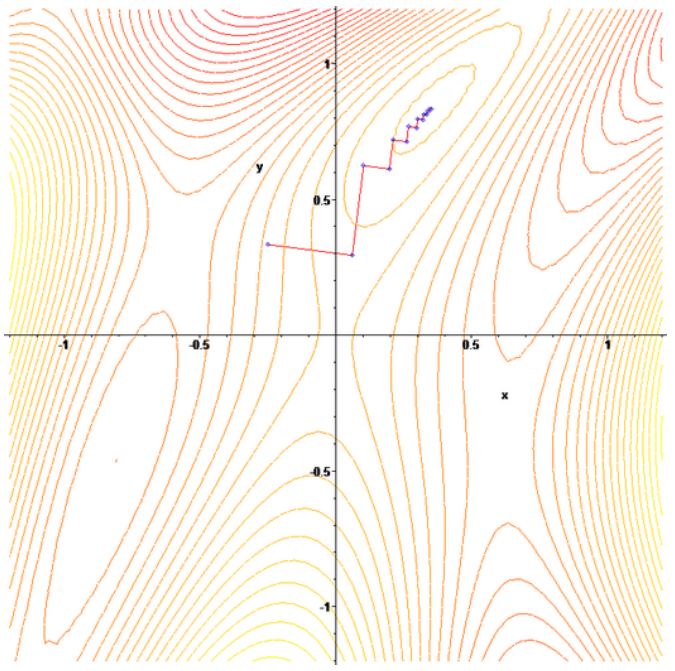
- 즉 이는 경사하강법이 함수의 최소값에 점진적으로 수렴함을 보장

- 우리는 일반적으로 경사 하강 알고리즘을 사용하여 다음을 찾습니다
  - 손실, 비용 또는 오류를 최소화하는 가중치(회귀, 신경망) 또는
  - 확률을 최대화하는 Parameter(MLE: 최대 우도 추정)



- 입력이 2차원인 경우 (When input is two dimemsional)
- 경사 하강법의 "Zig-Zagging" 특성






```
from collections import Counter
from linear_algebra import distance, vector_subtract, scalar_multiply
from functools import reduce
import math, random
```

- sum_of_squares 함수를 최소화하려고 합니다

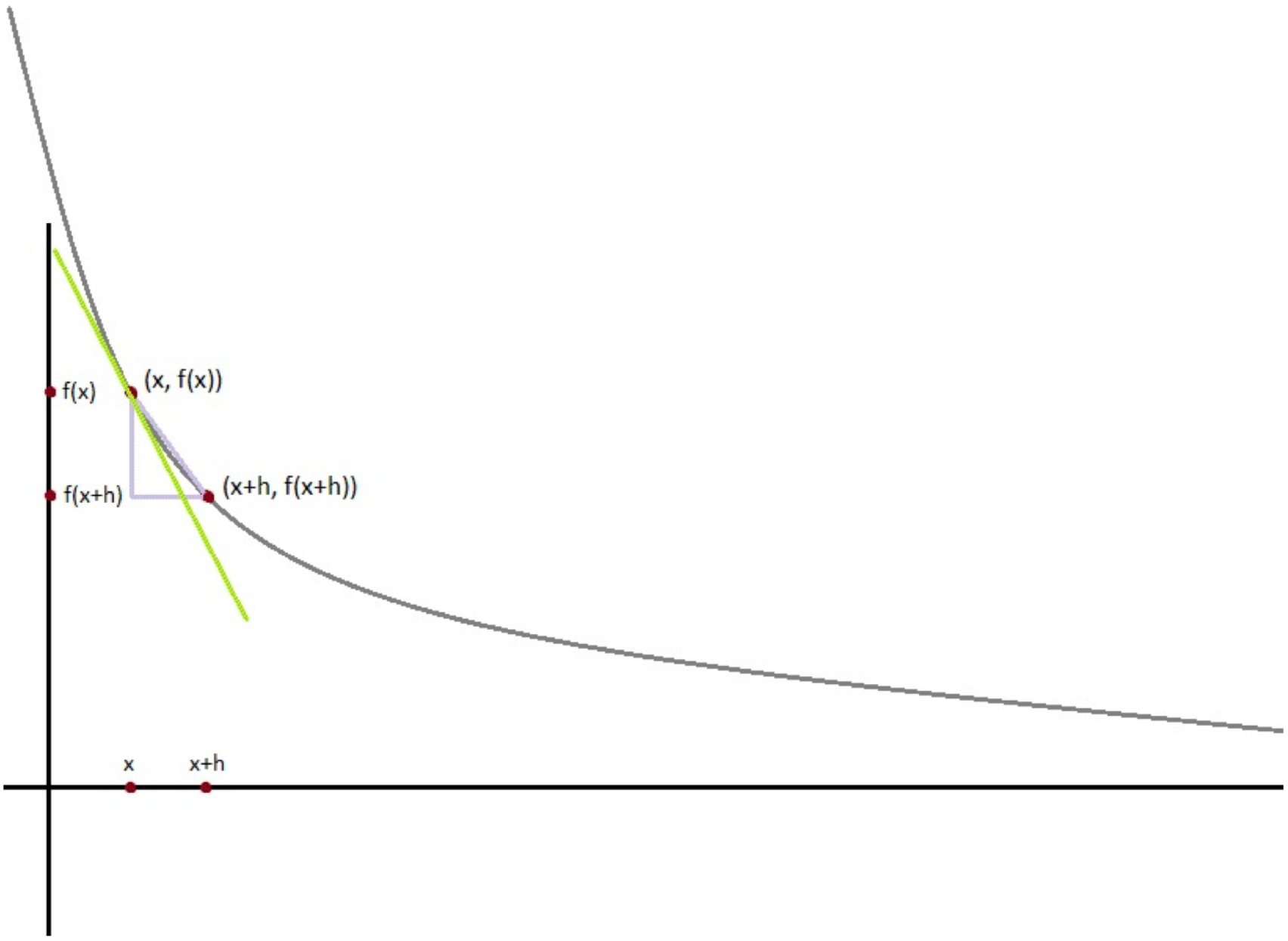
```
def sum_of_squares(v):
    """v 내 요소들의 제곱합을 계산합니다."""
    return sum(v_i ** 2 for v_i in v)
```

💡 **`sum_of_squares` 는 파이썬 리스트나 배열 `v` 안의 각 요소들의 제곱을 모두 더하는 함수입니다. 예를 들어, `v` 가 `[1, 2, 3]` 이라면, 이 함수는 (1^2 + 2^2 + 3^2 = 14)를 반환할 것**

```
def sum_of_squares_np(v):
    """v 내 요소들의 제곱합을 계산합니다. - Numpy Version."""
    return np.sum(v * v)
```

그래디언트 추정 (Estimating the Gradient)

- f가 한 변수의 함수라면, 점 x에서의 도함수는 우리가 x로 아주 작은 변화를 만들 때 f(x)가 어떻게 변하는지 측정합니다.




```
def difference_quotient(f, x, h):
    # 함수 f의 x에서의 차분 근사값을 계산합니다.
    return (f(x + h) - f(x)) / h
```

💡 **주어진 함수 `f`에 대해, 특정 점 `x`에서의 도함수(미분값)를 근사하는 방법을 구현합니다.**

함수 `f`, 점 `x`, 그리고 아주 작은 값 `h`를 입력으로 받아, `f`의 `x`에서의 기울기를 근사화

함수의 정의에 따라, `(f(x + h) - f(x)) / h` 계산은 `x`에서 `h`만큼 변화했을 때, 함수 `f`의 출력값이 얼마나 변하는지를 측정합니다.

이는 `x`에서 함수 `f`의 즉시 변화율이나 기울기를 근사하는 것과 동일합니다. `h`가 0에 가까워질수록, 이 근사값은 `f`의 `x`에서의 실제 도함수 값에 가까워집니다.

도함수는 함수의 변화율을 나타내며, 함수 그래프 상의 특정 점에서의 접선의 기울기에 해당

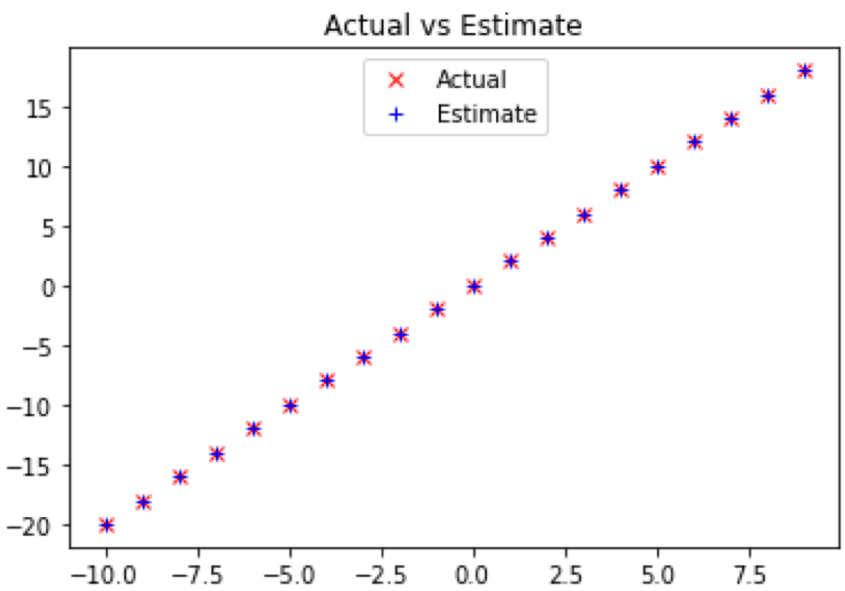
```
def plot_estimated_derivative():
    # 실제 함수 정의
    def square(x):
        """x의 제곱을 반환"""
        return x * x

    # 실제 도함수(미분값) 정의
    def derivative(x):
        """square 함수의 도함수인 2x를 반환"""
        return 2 * x

    # 근사 도함수 계산을 위한 람다 함수
    derivative_estimate = lambda x: difference_quotient(square, x, h=10)
```

```
# 실제 도함수와 근사 도함수를 그래프로 비교
import matplotlib.pyplot as plt
x = range(-10, 10)

# 실제 도함수 그래프 (빨간색 x로 표시)
plt.plot(x, list(map(derivative, x)), 'rx', label='Actual')
# 근사 도함수 그래프 (파란색 +로 표시)
plt.plot(x, list(map(derivative_estimate, x)), 'b+', label='Estimate')
# 범례 위치 설정 (9는 상단 중앙을 의미)
plt.legend(loc=9)

plt.title('Actual vs Estimate')
plt.show()
```

💡 **`square` 함수는 입력값의 제곱을 반환하며, 이 함수의 실제 도함수는 `2x` 입니다.**

근사 도함수는 `difference_quotient` 함수를 사용하여 계산되며, 여기서 `h` 는 근사 계산에 사용되는 매우 작은 값입니다.

```
%matplotlib inline

plot_estimated_derivative()
```



- 자동 미분은 그라데이션을 올바르게 계산합니다 (**automatic differentiation computes the gradient correctly)**

```
!pip install autograd
```

```
# autograd 라이브러리를 사용하여 자동 미분을 수행하는 예제입니다.
import autograd.numpy as np
from autograd import grad

# f라는 함수를 정의합니다. 이 함수는 입력 x에 대해 x^2의 값을 반환합니다.
def f(x):
    return x * x

# f 함수의 도함수(미분 함수)를 자동으로 생성합니다.
df_dx = grad(f)
```

```
# 생성된 도함수를 이용하여 x=5.0일 때의 도함수 값(미분 값)을 계산하고 출력합니다.
# f(x) = x^2의 도함수는 2x이므로, x=5일 때 도함수의 값은 10입니다.
print(df_dx(5.0))

# 시그모이드 함수를 정의합니다. 시그모이드 함수는 일반적으로
# 머신러닝과 딥러닝에서 활성화 함수로 많이 사용됩니다.
def sigmoid(x):
    return 1./(1. + np.exp(-x))

# 시그모이드 함수의 도함수를 자동으로 생성합니다.
dsigmoid_dx = grad(sigmoid)

# 생성된 도함수를 이용하여 x=0일 때의 도함수 값(미분 값)을 계산하고 출력합니다.
# 시그모이드 함수의 미분 값은 sigmoid(x) * (1 - sigmoid(x))이고,
# x=0일 때, 이 값은 0.25입니다.
print(dsigmoid_dx(0.))

# 시그모이드 함수의 x=0일 때의 함수 값 자체를 계산하고 출력합니다.
# 시그모이드 함수는 x=0에서 0.5의 값을 가집니다.
print(sigmoid(0))
```

```
10.0
0.25
0.5
```

```
import autograd.numpy as np
from autograd import grad

# 함수 정의: h(x, y) = x^2 + y
def h(x, y):
    return x**2 + y

# x에 대한 h의 편미분 함수 생성
dh_dx = grad(h, argnum=0)
# y에 대한 h의 편미분 함수 생성
dh_dy = grad(h, argnum=1)

# x=2, y=2에서의 편미분 값 계산 및 배열로 반환
np.array([dh_dx(2., 2.), dh_dy(2., 2.)])
```

```
array([4., 1.])
```

- 자동 미분을 사용하여 x=0.5에서 f(x) = exp(2x) / (1 + sin(x^2))

```
함수를 다음과 같이 일련의 기본 연산으로 분해할 수 있습니다:

Step 1: Let z1 = 2x
Step 2: Let z2 = exp(z1)
Step 3: Let z3 = x^2
Step 4: Let z4 = sin(z3)
Step 5: Let z5 = 1 + z4
Step 6: Let z6 = z2 / z5

다시 각 단계에서 x에 대한 함수와 그 도함수의 값을 추적합니다.
```

```
먼저 그 자체에 대한 출력의 도함수를 1로 설정한 다음 연쇄법칙을 사용하여
x에 대한 각 중간변수의 도함수를 계산합니다.

dz1/dx = 2
dz2/dx = dz1/dx * exp(z1)
dz3/dx = 2x
dz4/dx = cos(z3) * dz3/dx
dz5/dx = dz4/dx
dz6/dx = (dz2/dx * z5 - z2 * dz5/dx) / z5^2
```

Numerical gradient

- f가 많은 변수의 함수일 때, f는 여러 개의 편미분을 가지며, 각각은 입력 변수 중 하나에서 작은 변화를 만들 때 f가 어떻게 변하는지 나타냅니다.
- 우리는 다른 변수들을 고정한 채로 i번째 변수만의 함수로 취급하여 i번째 편미분을 계산합니다:

```
def partial_difference_quotient(f, v, i, h):
    # 주어진 함수 f의 i번째 변수에 대한 중앙 차분법을 사용하여 편미분을 근사하는 함수

    # v에서 i번째 변수에만 h를 더합니다.
    w = [v_j + (h if j == i else 0)
         for j, v_j in enumerate(v)]

    # w와 v에서의 함수 값의 차이를 h로 나누어 편미분을 근사합니다.
    return (f(w) - f(v)) / h
```

💡 설명

- 함수 *f*의 *i*번째 변수에 대한 편미분을 근사하는 중앙 차분법을 사용합니다.
- v에서의 *i*번째 변수에만 작은 변화 *h*를 추가한 *w*를 만듭니다.
- *w*와 *v*에서의 함수 값의 차이를 *h*로 나누어 편미분을 근사합니다.

```
def estimate_gradient(f, v, h=0.00001):
    # 주어진 함수 f의 그래디언트(기울기)를 중앙 차분법을 사용하여 근사하는 함수

    # 각 변수에 대해 partial_difference_quotient 함수(중앙 차분법)를 사용하여 그래디언트를 계산.
    return [partial_difference_quotient(f, v, i, h)
            for i, _ in enumerate(v)]
```

```
estimate_gradient(sum_of_squares, [1,1,1])
```

```
2.00001000001393, 2.00001000001393, 2.00001000001393]
```

```
import numpy as np

def estimate_gradient_np(f, v, h=0.00001):
    # f: 미분하고자 하는 다변수 함수
```

```
    # v: 함수 f에 대한 입력 벡터
    # h: 미분 계산을 위한 아주 작은 값, 기본값은 0.00001

    # np.eye(v.shape[0])는 v의 차원에 맞는 단위행렬을 생성합니다.
    # v + h * np.eye(v.shape[0])는 각 변수에 대해 h만큼 증가시킨 새로운 벡터들을 생성합니다.
    # np.apply_along_axis는 생성된 각 벡터에 대해 함수 f를 적용합니다.
    gradients = np.apply_along_axis(f, 1, v + h * np.eye(v.shape[0])) - f(v)

    # 계산된 차이를 h로 나누어 각 변수에 대한 함수 f의 그래디언트를 추정합니다.
    return gradients / h
```

💡 v + h * np.eye(v.shape[0]) → Numpy Broadcasting (v1+h .... vn)

```
estimate_gradient_np(lambda v: np.sum(v * v), np.array([1.,1.,1]))

# Result: array([2.00001, 2.00001, 2.00001])
```

💡 **단위행렬의 사용: `np.eye(v.shape[0])`을 통해 생성된 단위행렬은, 각 변수에 독립적으로 `h`만큼의 변화를 주기 위해 사용됩니다. 이렇게 함으로써, 각 변수에 대한 함수의 변화율(미분계수)을 독립적으로 계산할 수 있습니다.**

함수 적용과 그래디언트 계산: `np.apply_along_axis` 함수는 변형된 벡터 각각에 대해 원래 함수 `f`를 적용하고, 이를 통해 얻어진 함수 값의 변화를 `h`로 나눔으로써, 각 변수별로 함수의 그래디언트를 추정합니다.

V * V = V의 Dot Product → (V1**2 + V2**2 + V3**2 / 2V1 * 2V2 * 2V3 ~ th)

Using the Gradient

- sum_of_squares 함수는 입력 v가 0의 벡터일 때 가장 작습니다.
- 경사 하강법을 사용하여 사실을 확인하고자 합니다.
- (기울기 하강법) 그래디언트를 사용하여 모든 3차원 벡터 중 최소값을 찾아봅시다. 우리는 임의의 시작점을 선택한 다음 그래디언트가 매우 작은 지점에 도달할 때까지 그래디언트의 반대 방향으로 아주 작은 단계를 밟을 것입니다

```
def step(v, direction, step_size):
    """move step_size in the direction from v"""
    return [v_i + step_size * direction_i
            for v_i, direction_i in zip(v, direction)]
```

💡 **`step` 함수는 주어진 방향 `direction`으로 `step_size`만큼 이동하는 기능**

`v`: 현재 위치를 나타내는 벡터.

`direction`: 이동하고자 하는 방향을 나타내는 벡터. 이 방향은 보통 목표 함수의 기울기에 의해 결정됩니다.

`step_size`: 한 번에 이동할 거리. 즉, 이동의 크기를 결정합니다.

반환값: 새로운 위치를 나타내는 벡터.

```
def sum_of_squares_gradient(v):
    return [2 * v_i for v_i in v]
```

💡 **주어진 벡터 `v`에 대해 `sum_of_squares` 함수의 기울기를 계산**

매개변수: `v` : 기울기를 계산하고자 하는 위치를 나타내는 벡터.반환값:

`sum_of_squares` 함수의 `v`에서의 기울기 벡터. 각 요소는 `2 * v_i`로, `sum_of_squares` 함수는 각 변수의 제곱의 합으로 이루어져 있기 때문에, 그 미분값은 `2 * v_i`가 됩니다.

sum_of_squares_gradient(v) → 2V1, 2V2, 2VN.... - Gradient Function.

- 비고: step_size는 learning_rate라고도 합니다

```
def step(v, direction, step_size):
    """v에서 direction 방향으로 step_size만큼 이동"""
    return [v_i + step_size * direction_i
            for v_i, direction_i in zip(v, direction)]

def sum_of_squares_gradient(v):
"""v의 제곱합 함수의 기울기(gradient) 계산"""
return [2 * v_i for v_i in v]
```

💡 설명

`step` 함수

목적: 현재 위치 `v`에서 주어진 `direction` 방향으로 `step_size`만큼 이동한 새로운 위치를 계산합니다.

`v`: 현재 위치를 나타내는 벡터입니다.

`direction`: 이동할 방향을 나타내는 벡터입니다. 보통 최적화하려는 함수의 기울기(gradient) 반대 방향이 됩니다.

`step_size`: 한 번에 이동할 거리를 나타냅니다. 이 값이 너무 크면 최적점을 넘어서게 되고, 너무 작으면 최적화 과정이 매우 느려질 수 있습니다.반환값: 새로운 위치를 나타내는 벡터입니다. 이는 `v`의 각 요소에 `direction`의 해당 요소와 `step_size`를 곱한 값을 더해서 계산됩니다.

`sum_of_squares_gradient` 함수

목적: 주어진 벡터 `v`에 대해 제곱합 함수의 기울기를 계산합니다. 제곱합 함수는 모든 요소의 제곱을 더한 것이며, 이 함수는 최적화에서 자주 사용되는 간단한 예시입니다.

파라미터: `v` : 기울기를 계산할 벡터입니다.반환값: 제곱합 함수의 기울기를 나타내는 벡터입니다. 제곱합 함수의 각 변수에 대한 편미분은 `2 * v_i`이므로, 결과 벡터는 입력 벡터 `v`의 각 요소에 2를 곱한 값으로 구성됩니다.

이 두 함수는 경사하강법(gradient descent) 알고리즘의 기본 구성 요소입니다. 경사하강법은 함수의 최소값을 찾기 위해 기울기(또는 그래디언트) 정보를 사용하여 반복적으로 현재 위치를 업데이트하는 방법입니다.

`sum_of_squares_gradient` 함수는 최적화하려는 함수의 기울기를 계산하고, `step` 함수는 이 기울기를 사용하여 다음 위치로 이동합니다.

```
print("using the gradient")
# try range(n) n = 1,2,3,4,5,...
v = [random.randint(-10,10) for i in range(2)]  # 임의의 초기 벡터 v 생성

tolerance = 0.0000001  # 수렴 기준 값 (10**-5, 10의 -5승 까지)

while True:
    #print(v, sum_of_squares(v))
    gradient = sum_of_squares_gradient(v)   # v에서의 기울기 계산
    next_v = step(v, gradient, -0.01)       # 음의 기울기 방향으로 한 걸음 이동
```

```
    if distance(next_v, v) < tolerance:     # 이전 v와 현재 v의 거리가 tolerance보다 작으면 중단
        break
    v = next_v                              # 아니면 계속

print("minimum v", v)                       # 최소화된 v 출력
print("minimum value", sum_of_squares(v))   # 최소화된 값 출력
```

💡 설명

`v`: 최적화를 시작할 임의의 벡터입니다. 여기서는 2차원 벡터를 사용합니다.

`tolerance`: 알고리즘이 수렴했다고 판단하는 기준입니다. `v`의 연속된 두 값의 차이가 이 값보다 작으면 반복을 멈춥니다.

`while True` 루프: 알고리즘이 수렴 조건을 만족할 때까지 반복합니다.

`gradient = sum_of_squares_gradient(v)`: 현재 위치 `v`에서 목적 함수의 기울기를 계산합니다. → Gradient Step Size 만큼 걸어다면 다음 Step Size

`next_v = step(v, gradient, -0.01)`: 계산된 기울기의 반대 방향(최소화 방향)으로 작은 걸음(-0.01)을 이동하여 새로운 위치를 계산합니다.

`if distance(next_v, v) < tolerance`: 새로운 위치와 이전 위치 사이의 거리가 `tolerance`보다 작으면, 즉 변화가 충분히 작아 수렴했다고 판단하면 반복을 멈춥니다.

최종적으로, 수렴한 위치 `v`와 그 때의 목적 함수 값 `sum_of_squares(v)`를 출력합니다.

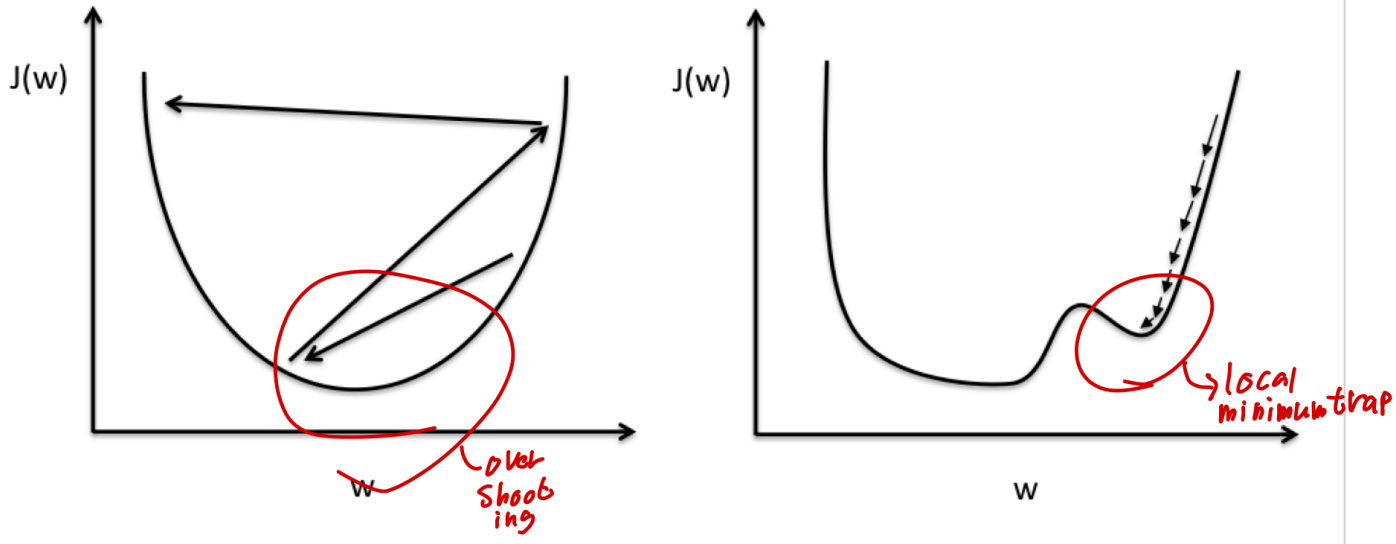
```
using the gradient
minimum v [-4.157730989669425e-06, -2.7718206597796163e-06]
minimum value 2.4969716752438604e-11
```

Choosing the Right Step Size (or Learning rate) - 올바른 스텝 크기 (또는 학습 속도) 선택

- gradient에 역행하여 움직일 수 있는 근거는 분명하지만, 얼마나 멀리 움직일 것인지는 명확하지 않습니다. 실제로 올바른 스텝 크기를 선택하는 것은 과학이라기보다는 예술에 가깝습니다. 인기 있는 옵션은 다음과 같습니다:
  - 고정 스텝 크기 사용 (Using a fixed step size)
  - 시간이 지남에 따라 단계 크기가 점차 축소됨 (Gradually shrinking the step size over time)
  - 각 단계에서 목적 함수의 값을 최소화하는 단계 크기 선택 (At each step, choosing the step size that minimizes the value of the objective function)
- 마지막은 최적인 것처럼 들리지만 실제로는 비용이 많이 드는 계산입니다. 우리는 다양한 단계 크기를 시도하고 목적 함수의 값이 가장 작은 단계를 선택함으로써 그 근사치를 구할 수 있습니다:

```
step_sizes = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001]
```

- 학습률이 너무 크면 편차가 발생합니다.
- 학습 속도가 너무 작으면 수렴 속도가 너무 느리거나 로컬 최소값으로 떨어질 수 있습니다.
  - 지역적인 Global Local Minimum



Large learning rate: Overshooting.

Small learning rate: Many iterations until convergence and trapping in local minima.

Experiment with various learning rates (다양한 Learning Rate로 실험)

```
# in class, changes lr = 10, 1.1, 1, 0.1, 0.01
# 10 : diverge
# 1.1: diverge
# 1: oscilliating
# 0.1: good pace
# 0.01 : two slow
```

```
import numpy as np
import matplotlib.pyplot as plt

def sum_of_squares_gradient_np(v):
    """제곱합 함수의 그래디언트를 계산합니다."""
    return 2 * v

def gradient_descent(gradient_f, init_x, lr=0.01, step_num=10000, tolerance=0.0000001):
    """그래디언트 하강법을 사용해 최소값을 찾습니다."""
    x = init_x
    x_history = []
    for i in range(step_num):
        x_history.append(x.copy())
        x_prev = x.copy()
        x -= lr * gradient_f(x)  # 그래디언트 스텝
        if np.linalg.norm(x - x_prev) < tolerance:  # 수렴 조건
            break
    return x, x_history

init_x = np.array([-1.0, 1.0])
lr = 1.1  # 학습률, # try with 10, 1.1, 1, 0.1, 0.01
step_num = 100
x, x_history = gradient_descent(sum_of_squares_gradient_np, init_x, lr=lr, step_num=step_num)

# 시각화
plt.plot( [-5, 5], [0,0], '--b')
plt.plot( [0,0], [-5, 5], '--b')
x_history = np.array(x_history)
plt.plot(x_history[:,0], x_history[:,1], 'o')
```

```
plt.xlim(-3.5, 3.5)
plt.ylim(-4.5, 4.5)
plt.xlabel("X0")
plt.ylabel("X1")
plt.axis('equal')
plt.show()
```

💡 설명

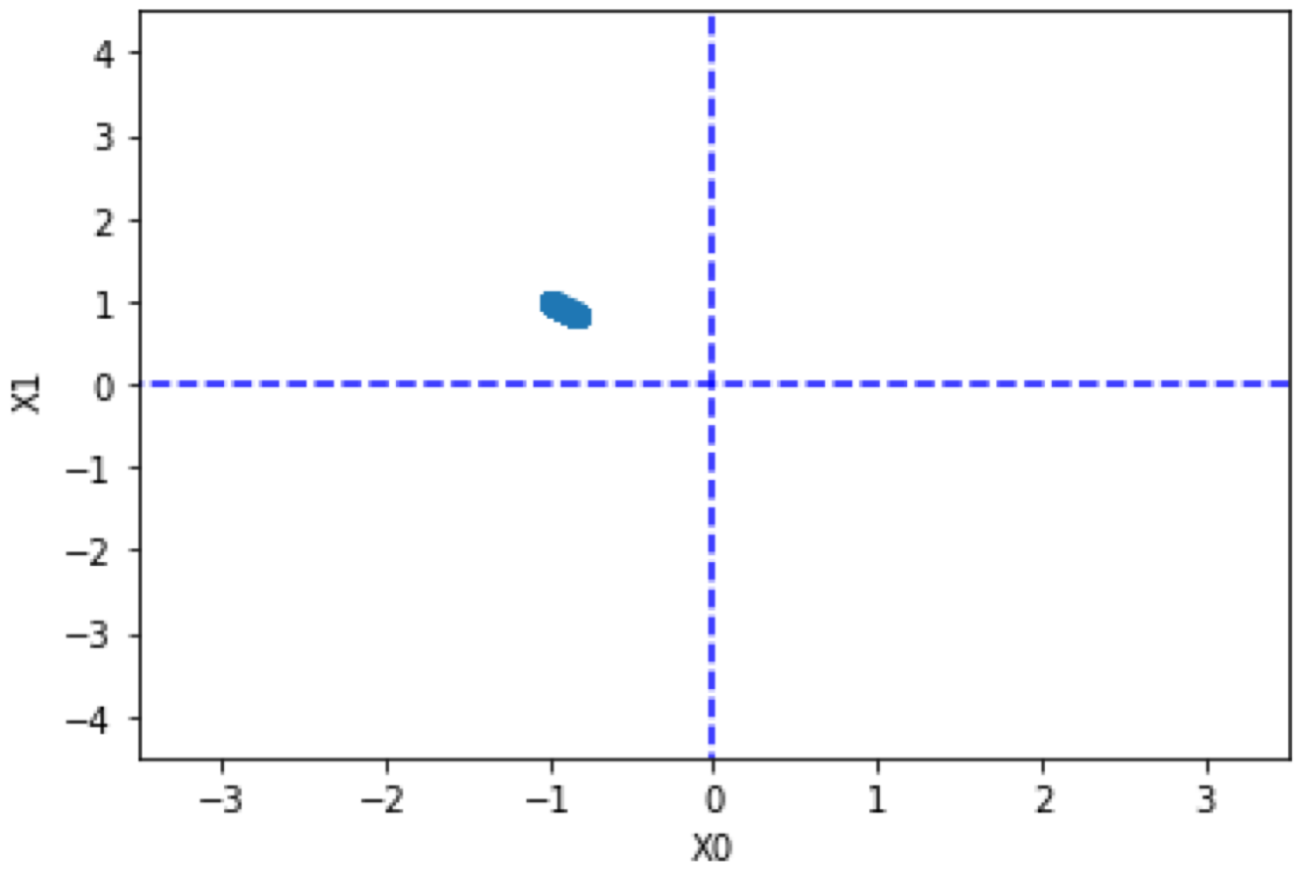
이 코드는 초기 점(init_x)에서 시작하여 제곱합 함수의 최소값을 찾기 위한 경로를 시각화합니다.

경사 하강법 함수(`gradient_descent`)는 주어진 그래디언트 함수(`gradient_f`), 초기 점(`init_x`), 학습률(`lr`), 최대 스텝 수(`step_num`), 그리고 수렴 기준(`tolerance`)을 인자로 받아 최소값을 찾는 과정을 수행합니다.

학습률(`lr`)은 경사 하강법에서 매우 중요한 하이퍼파라미터입니다. 너무 크면 발산할 수 있고, 너무 작으면 수렴이 매우 느려질 수 있습니다.

이 예제에서 `lr` 값을 다양하게 변경해보며 그 영향을 시각적으로 관찰할 수 있습니다.

`x -= lr * gradient_f(x)` → x의 Gradient Function (Gradient의 반대방향으로 걷는다.)



- (Out of function domain) 특정 스텝 크기로 인해 함수에 대한 입력이 잘못됨
- 따라서 잘못된 입력에 대해 무한대를 반환하는 "안전 적용" 함수를 만들어야 합니다:

```
def safe(f):
    """f 함수를 안전하게 실행하는 새로운 함수를 반환한다."""
    def safe_f(*args, **kwargs):
        try:
            return f(*args, **kwargs)  # f를 실행해봄 -> 모든 인자, keyword & argument
        except:
            return float('inf')        # 예외가 발생하면 무한대를 반환
    return safe_f
```

💡 **함수 `f`를 받아, 해당 함수를 실행할 때 예외가 발생하면 프로그램이 중단되지 않고 대신 무한대(`float('inf')`)를 반환하는 새로운 함수 `safe_f`를 반환**

Putting It All Together

- 일반적인 경우에는 최소화하려는 target_fn이 있고, gradient_fn도 있습니다.
- 예를 들어 target_fn은 모델의 오류를 매개변수의 함수로 나타낼 수 있으며, 오류를 가능한 작게 만드는 매개변수를 찾고자 할 수 있습니다.

```
def minimize_batch(target_fn, gradient_fn, theta_0, tolerance=0.000001):
    """경사 하강법을 사용하여 목표 함수를 최소화하는 theta를 찾습니다."""
    # tolerance -> f값이 이것보다 낮아지면 끝.

    # 각 단계 크기를 설정합니다.
    step_sizes = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001]

    # theta를 초기 값으로 설정합니다.
    theta = theta_0

    # target_fn의 안전 버전을 만듭니다.
    target_fn = safe(target_fn)

    # 최소화할 값을 설정합니다.
    value = target_fn(theta)

    while True:
        # 현재 theta에서의 기울기를 계산합니다.
        gradient = gradient_fn(theta)

        # 다음 단계 후보를 생성합니다.
        next_thetas = [step(theta, gradient, -step_size)
                       for step_size in step_sizes]

        # 오차 함수를 최소화하는 것을 선택합니다.
        next_theta = min(next_thetas, key=target_fn) # target function 최소값 구함
        next_value = target_fn(next_theta) # value & next value의 차이

        # 수렴하는 경우 멈춥니다.
        if abs(value - next_value) < tolerance:
            return theta
        else:
            theta, value = next_theta, next_value
```

Example : Minimizing sum_of_squares

```
minimize_batch(sum_of_squares, sum_of_squares_gradient, [10,20,4,5])
```

```
[0.0006805647338418772,
 0.0013611294676837543,
```

```
 0.00027222589353675085,
 0.0003402823669209386]
```

```
minimize_batch(sum_of_squares, sum_of_squares_gradient, [10,20,4,5,0,1])
```

```
[0.0006805647338418772,
 0.0013611294676837543,
 0.00027222589353675085,
 0.0003402823669209386,
 0.0,
 6.805647338418771e-05]
```

Example : Centering a certain point

```
def myf(v):
    # 주어진 벡터 v의 각 요소와 3, 2 각각을 뺀 후 제곱하여 반환합니다.
    return (v[0]-3)**2 + (v[1]-2)**2

def myf_gradient(v):
    # 주어진 벡터 v의 각 요소에 2를 곱하고, 각각에 6 또는 4를 뺀 결과를 리스트로 반환합니다.
    return [2.0*v[0]-6, 2.0*v[1]-4]

# minimize_batch 함수를 호출하여 myf 함수를 최소화합니다.
minimize_batch(myf, myf_gradient, [5000., 50.])
```

💡 myf_gradient 함수를 그래디언트로 사용하여 경사 하강법을 수행합니다.

초기 theta 값은 [5000., 50.]으로 설정되어 있습니다.

이 값은 사용자가 문제에 따라 적절하게 설정해야 합니다.

```
[3.0016059738814325, 2.000015426605225]
```

```
from functools import partial

def f1(x, c):
    # 주어진 벡터 x와 상수 벡터 c의 차이를 제곱하여 합산한 결과를 반환합니다.
    x = np.array(x)
    c = np.array(c)
    return np.sum((x - c)**2) # x, c 사이 거리

def f1_gradient(x, c):
    # 주어진 벡터 x와 상수 벡터 c에 각각 2를 곱하고, 각각에 상수 벡터 c를 뺀 결과를 반환합니다.
    x = np.array(x)
    c = np.array(c)
    return 2*x - 2*c

def numerical_gradient(v, f, h=0.00001):
    # 중앙 차분법을 사용하여 주어진 벡터 v에서 함수 f의 그래디언트를 계산합니다.
    return (np.apply_along_axis(f, 1, v + h * np.eye(len(v))) - f(v)) / h
```

```
c = np.array([7,70,7,4])

# f1 함수를 상수 c를 고정시켜 부분 함수로 생성합니다 -> f function에 인자 먼저
f = partial(f1, c=c)

# f1_gradient 함수를 상수 c를 고정시켜 부분 함수로 생성합니다.
gradient_f = partial(f1_gradient, c=c)

# minimize_batch 함수를 호출하여 f를 최소화합니다.
# gradient_f를 그래디언트로 사용하여 경사 하강법을 수행합니다.
# 초기 theta 값은 [0,0,0,0]으로 설정되어 있습니다.
minimize_batch(f, gradient_f, [0,0,0,0])
```

💡 f1 함수와 f1_gradient 함수에서는 주어진 벡터 x와 상수 벡터 c에 대해 각각 함수 값을 계산하고 그래디언트를 반환합니다.

functools 모듈의 partial 함수를 사용하여 c를 고정시킨 부분 함수를 생성합니다.

이렇게 생성된 부분 함수들을 minimize_batch 함수의 인수로 전달하여 최적화를 수행

```
[6.999843894783611, 69.99843894783609, 6.999843894783611, 3.9999107970192056]
```

- 때로는 함수의 음의 기울기를 최소화함 으로써 함수를 최대화할 수도 있습니다(해당 음의 기울기를 가짐):

```
def negate(f):
    """주어진 함수 f에 대해 -f(x)를 반환하는 함수를 반환합니다."""
    return lambda *args, **kwargs: -f(*args, **kwargs)

def negate_all(f):
    """리스트를 반환하는 함수 f의 각 결과에 대해 음수를 취한 리스트를 반환하는 함수를 반환합니다."""
    return lambda *args, **kwargs: [-y for y in f(*args, **kwargs)] # -y: 반대 - function화

def maximize_batch(target_fn, gradient_fn, theta_0, tolerance=0.000001):
    # minimize_batch 함수를 호출하여 목표 함수를 최대화하는 theta 값을 찾습니다.
    # negate 함수를 사용하여 목표 함수를 음수로 변환하고, gradient_fn의 결과를 음수로 변환하는 negate_all 함수
    return minimize_batch(negate(target_fn),
                          negate_all(gradient_fn),
                          theta_0,
                          tolerance)
```

💡 minimize_batch 함수를 호출하여 주어진 목표 함수를 최대화하는 theta 값을 찾습니다. maximize_batch 함수는 주어진 목표 함수와 그래디언트 함수를 negate 함수와 negate_all 함수를 사용하여 음수로 변환한 후 minimize_batch 함수에 전달

Maximizing batch Example

- 정규 pdf를 최대화하는 변수를 찾습니다.

nal pdf.

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- normal pdf의 도함수는 ... (deriv 가능)

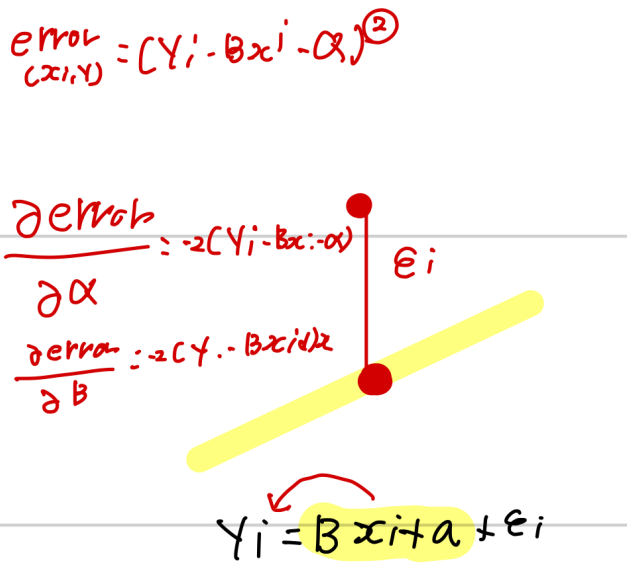
```
from functools import partial

def normal_pdf(npx, mu, sigma):
    # 정규 분포의 확률 밀도 함수를 계산합니다.
    x = npx[0]
    return ((1/(np.sqrt(2*np.pi)*sigma)*np.exp(-(x-mu)**2/(2*sigma**2))))

def numerical_gradient(v, f, h=0.00001):
    # 주어진 함수 f의 그래디언트를 중앙 차분법을 사용하여 근사하는 함수입니다.
    return (np.apply_along_axis(f, 1, v + h * np.eye(len(v))) - f(v)) / h

# normal_pdf 함수에 대한 부분 함수를 생성합니다. mu=1, sigma=1로 고정됩니다.
f = partial(normal_pdf, mu=1, sigma=1)

# numerical_gradient 함수에 대한 부분 함수를 생성합니다.
gradient_f = partial(numerical_gradient, f=f)

# 초기값을 설정합니다.
init_x = np.array([1.])

# maximize_batch 함수를 호출하여 normal_pdf 함수를 최대화합니다.
# gradient_f를 그래디언트로 사용하여 경사 하강법을 수행합니다.
maximize_batch(f, gradient_f, init_x)
```

💡 `maximize_batch` 함수를 사용하여 주어진 함수 `normal_pdf`를 최대화하는 예제입니다.

주어진 함수 `normal_pdf`는 정규 분포의 확률 밀도 함수를 계산합니다.

이 함수에 대한 그래디언트는 `numerical_gradient` 함수를 사용하여 근사합니다.

초기값은 `init_x`로 설정되어 있습니다. 최종적으로 `maximize_batch` 함수를 호출하여 최대화된 값을 찾습니다.



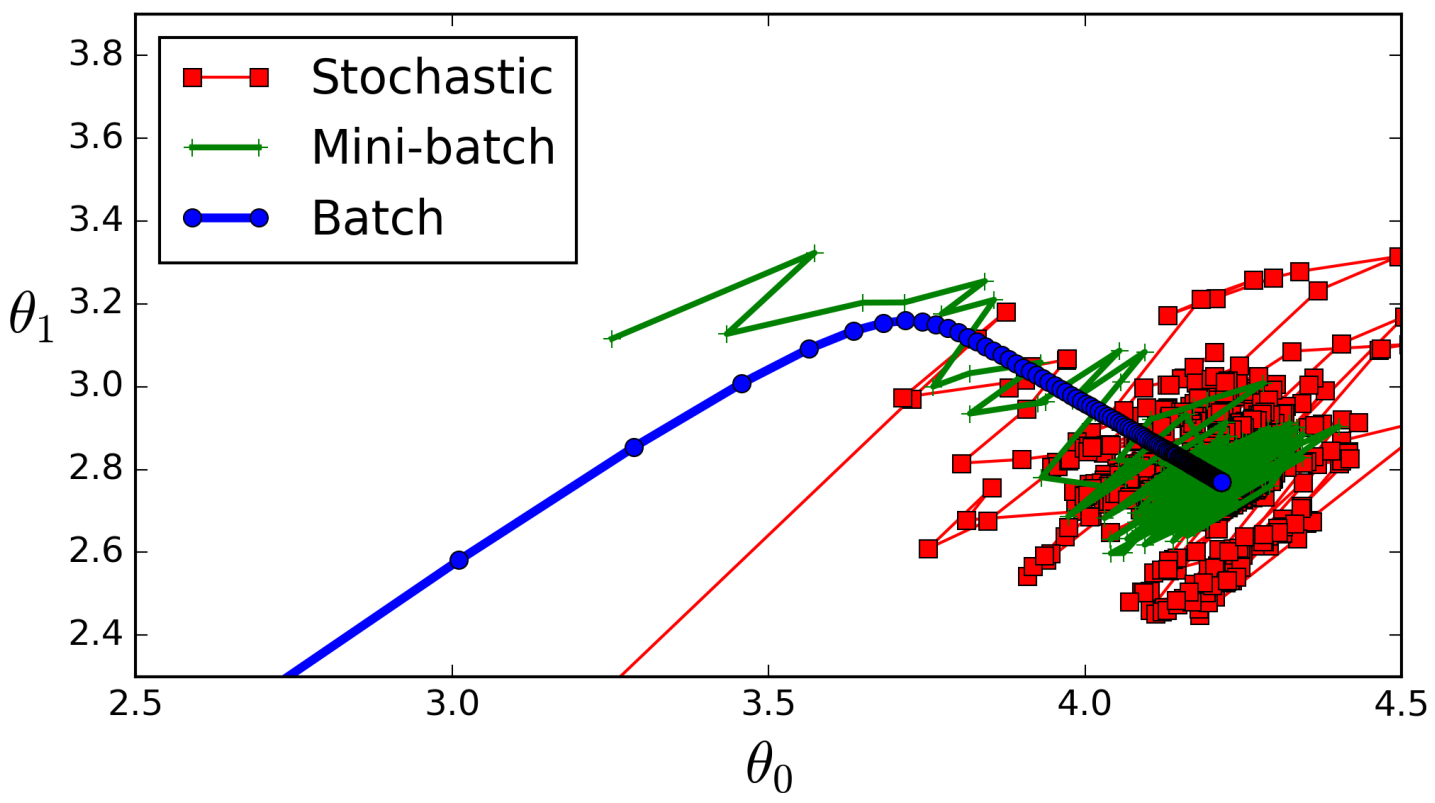

```
array([1.])
```

확률적 경사 하강법 (Stochastic Gradient Descent)

- 배치 경사 하강
  - 배치 접근 방식에서 각 그라데이션 단계는 예측을 수행하고 전체 데이터 세트에 대한 그라데이션을 계산해야 하므로 각 단계에 오랜 시간이 걸립니다.
- 일반적으로 오차 함수는 가산적이며, 이는 전체 데이터 세트에 대한 예측 오차가 단순히 각 데이터 포인트에 대한 예측 오차의 합이라는 것을 의미합니다.
- 확률적 경사 하강
  - 확률적 경사 하강법은 한 번에 한 점에 대해서만 경사도를 계산합니다(그리고 단계를 밟습니다).
  - 정지 지점에 도달할 때까지 데이터를 반복적으로 순환합니다. 매 주기 동안 우리는 데이터를 무작위 순서로 반복하기를 원할 것입니다:

Batch vs SGD vs Mini-batch

방법	설명	장점	단점
Batch	전체 학습 데이터셋에서 기울기를 계산함	정확함	데이터 크기가 매우 크면 1) 느림, 2) 메모리에 맞추기 어려움
SGD	하나의 샘플에서 기울기를 계산하고 업데이트	빠름	과도한 오버슈팅 및 수렴이 어려움 (학습률 감소로 해결 가능)
Mini-batch	n개의 샘플의 미니배치에서 기울기를 계산하고 업데이트	분산 감소, 안정적인 업데이트	빠른 계산, (하드웨어/소프트웨어의 강점을 활용). 이것은 신경망을 훈련하는 알고리즘입니다.



💡 전체 Error = 부분 에러의 합 → 1개 씩만 본다.

NeuralNet Terminology: Epoch

- 한 epoch는 전체 데이터 세트가 훈련을 위해 소비되는 경우입니다.

```
import random

def in_random_order(data):
    # 주어진 데이터를 무작위 순서로 반환하는 제너레이터입니다.
    # Args: data: 무작위로 반환할 데이터, Returns: 무작위로 섞인 데이터

    indexes = [i for i, _ in enumerate(data)]  # 데이터의 인덱스 목록을 생성합니다.
```

```
    random.shuffle(indexes)                    # 인덱스를 섞습니다.
    for i in indexes:                          # 해당 순서대로 데이터를 반환합니다.
        yield data[i]
```

💡 `in_random_order` 제너레이터 함수를 정의합니다. 이 함수는 주어진 데이터를 무작위 순서로 반환합니다.

`enumerate` 함수를 사용하여 데이터의 인덱스 목록을 생성하고, `random.shuffle` 함수를 사용하여 인덱스를 섞습니다. 그런 다음 섞인 인덱스에 따라 데이터를 반환

Understanding SGD Code

- x는 Training 데이터 세트입니다
- y는 Lavel(또는 클래스) 집합입니다

```
def minimize_stochastic(target_fn, gradient_fn, x, y, theta_0, alpha_0=0.01):

    data = list(zip(x, y))
    theta = theta_0                                # 초기 추정값
    alpha = alpha_0                                # 초기 스텝 크기
    min_theta, min_value = None, float("inf")   # 현재까지의 최솟값
    iterations_with_no_improvement = 0

    # 100회 반복하여 개선 없으면 종료
    while iterations_with_no_improvement < 100: # Total Error
        value = sum(target_fn(x_i, y_i, theta) for x_i, y_i in data)

        if value < min_value:
            # 새로운 최솟값을 찾았으면 기억하고 초기 스텝 크기로 돌아감
            min_theta, min_value = theta, value
            iterations_with_no_improvement = 0
            alpha = alpha_0
        else:
            # 개선이 없다면 스텝 크기를 줄여봄
            iterations_with_no_improvement += 1
            alpha *= 0.9

        # 각 데이터 포인트에 대해 그래디언트 스텝을 취함
        for x_i, y_i in in_random_order(data):
            gradient_i = gradient_fn(x_i, y_i, theta)
            theta = vector_subtract(theta, scalar_multiply(alpha, gradient_i))

    return min_theta
```

💡 `target_fn:` 목표 함수

`gradient_fn:` 목표 함수의 그래디언트(기울기)

`x:` 입력 데이터

`y:` 출력 데이터

`theta_0:` 초기 theta 값

`alpha_0:` 초기 학습률 (기본값: 0.01)

Returns: 최대값을 가지는 theta 값

```
def maximize_stochastic(target_fn, gradient_fn, x, y, theta_0, alpha_0=0.01):
    """
    확률적 경사 상승법을 사용하여 목표 함수를 최대화하는 함수입니다.
    """
    return minimize_stochastic(negate(target_fn),
                               negate_all(gradient_fn),
                               x, y, theta_0, alpha_0)
```

💡 `target_fn`은 목표 함수이고, `gradient_fn`은 목표 함수의 그래디언트(기울기) 함수입니다. `x`는 입력 데이터, `y`는 출력 데이터, `theta_0`은 초기 추정값입니다. `alpha_0`은 초기 학습률로, 기본값은 0.01입니다. 최적화된 `theta` 값을 반환