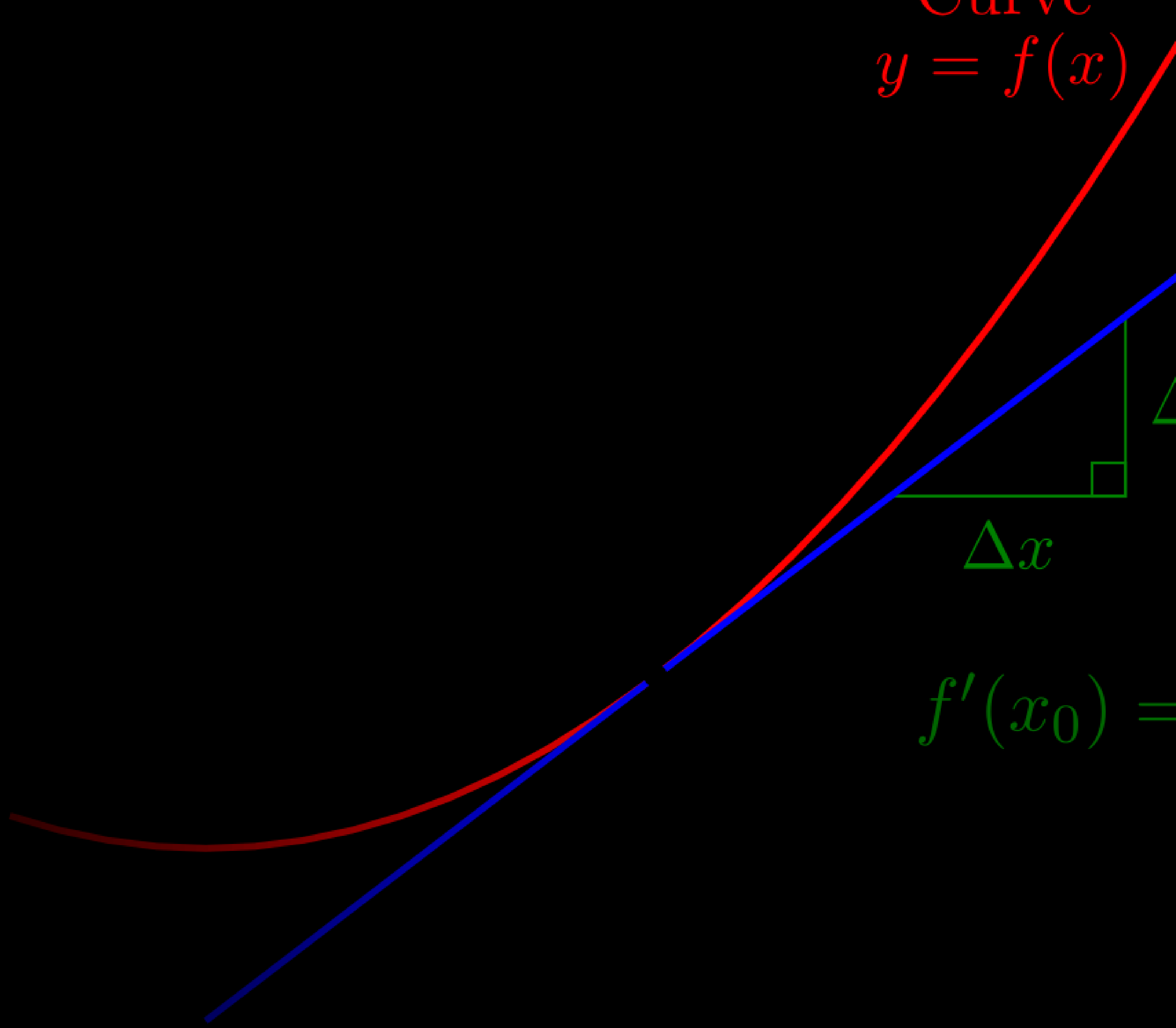
Calculus in Machine Learning

Prof. Kuan-Ting Lai

2023/10/21



Calculus

- Calculus is the mathematical study of continuous change.

- Two major branches: Differential Calculus and Integral Calculus

- We mainly use differential calculus in machine learning

https://en.wikipedia.org/wiki/Calculus

Definition of Derivative

- A function of a real variable *f(x)* is differentiable at a point x of its domain, if its domain contains an open interval containing *x* and the limit exists.
- Derivative measures the "rate of change"

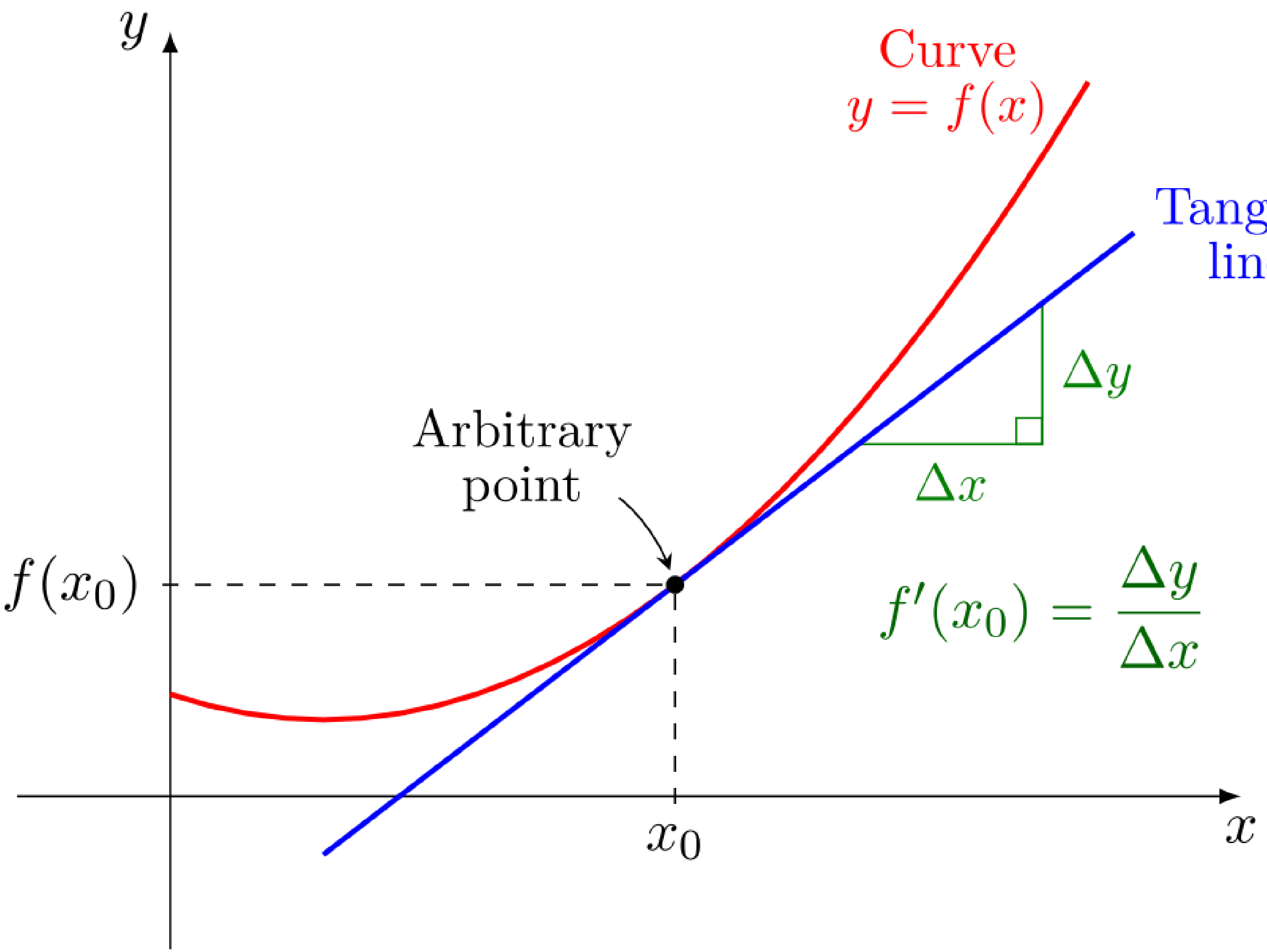
$$f'(x) = \lim_{\triangle x \to 0} \frac{f(x + \triangle x) - f(x)}{\triangle x}$$

OR

$$f'(x) = \lim_{\triangle x \to 0} \frac{f(x + \triangle x) - f(x - \triangle x)}{2 \triangle x}$$

Geometric Definition

- Average rate of change of y with respect to x over the interval.



Basic Rules

- Common derivative rules

$$\frac{d}{dx}x^a = ax^{a-1}$$

$$\frac{d}{dx}e^x = e^x.$$

$$\frac{d}{dx}a^x = a^x \ln(a), \qquad a > 0$$

$$\frac{d}{dx}\ln(x) = \frac{1}{x}, \qquad x > 0.$$

$$\frac{d}{dx}\log_a(x) = \frac{1}{x\ln(a)}, \qquad x, a > 0$$

$$\frac{d}{dx}\sin(x) = \cos(x).$$

$$\frac{d}{dx}\cos(x) = -\sin(x).$$

$$\frac{d}{dx}\tan(x) = \sec^2(x) = \frac{1}{\cos^2(x)} = 1 + \tan^2(x).$$

$$\frac{d}{dx}\arcsin(x) = \frac{1}{\sqrt{1-x^2}}, \qquad -1 < x < 1.$$

$$\frac{d}{dx}\arccos(x) = -\frac{1}{\sqrt{1-x^2}}, \qquad -1 < x < 1.$$

$$\frac{d}{dx}\arctan(x) = \frac{1}{1+x^2}$$

Implement Differentiation

- Use a small value (0.001) to replace Δ

$$\frac{df}{du}(a) = \lim_{\Delta \to 0} \frac{f(a+\Delta) - f(a-\Delta)}{2 \times \Delta}$$

$$\Downarrow$$

$$\frac{df}{du}(a) = \frac{f(a+0.001) - f(a-0.001)}{0.002}$$

Seth Weidman, “Deep Learning from Scratch,” O’Reilly Media, 2019

Derivative Function

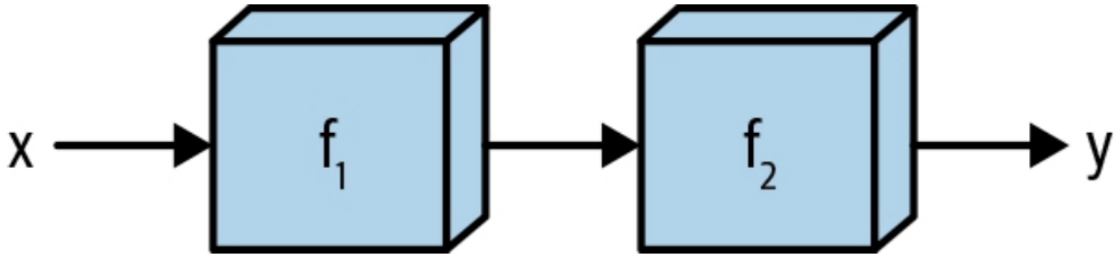
- For any input function, calculate derivative using the definition

```
from typing import Callable

def deriv(func: Callable[[ndarray], ndarray],
          input_: ndarray,
          delta: float = 0.001) -> ndarray:
    '''
    Evaluates the derivative of a function "func" at every element in the
    "input_" array.
    '''
    return (func(input_ + delta) - func(input_ - delta)) / (2 * delta)
```

https://github.com/SethHWeidman/DLFS_code

Nested Functions

- $y = f_2(f_1(x))$




```python
from typing import List

# A Function takes in an ndarray as an argument
Array_Function = Callable[[ndarray], ndarray]

# A Chain is a list of functions
Chain = List[Array_Function]


def chain_length_2(chain: Chain,
                   a: ndarray) -> ndarray:
    '''
    Evaluates two functions in a row, in a "Chain".
    '''
    assert len(chain) == 2, \
    "Length of input 'chain' should be 2"

    f1 = chain[0]
    f2 = chain[1]

    return f2(f1(x))
```

The Chain Rule

- **Chain rule** is a formula that expresses the derivative of the composition of two differentiable functions *f* and *g* in terms of the derivatives of *f* and *g*

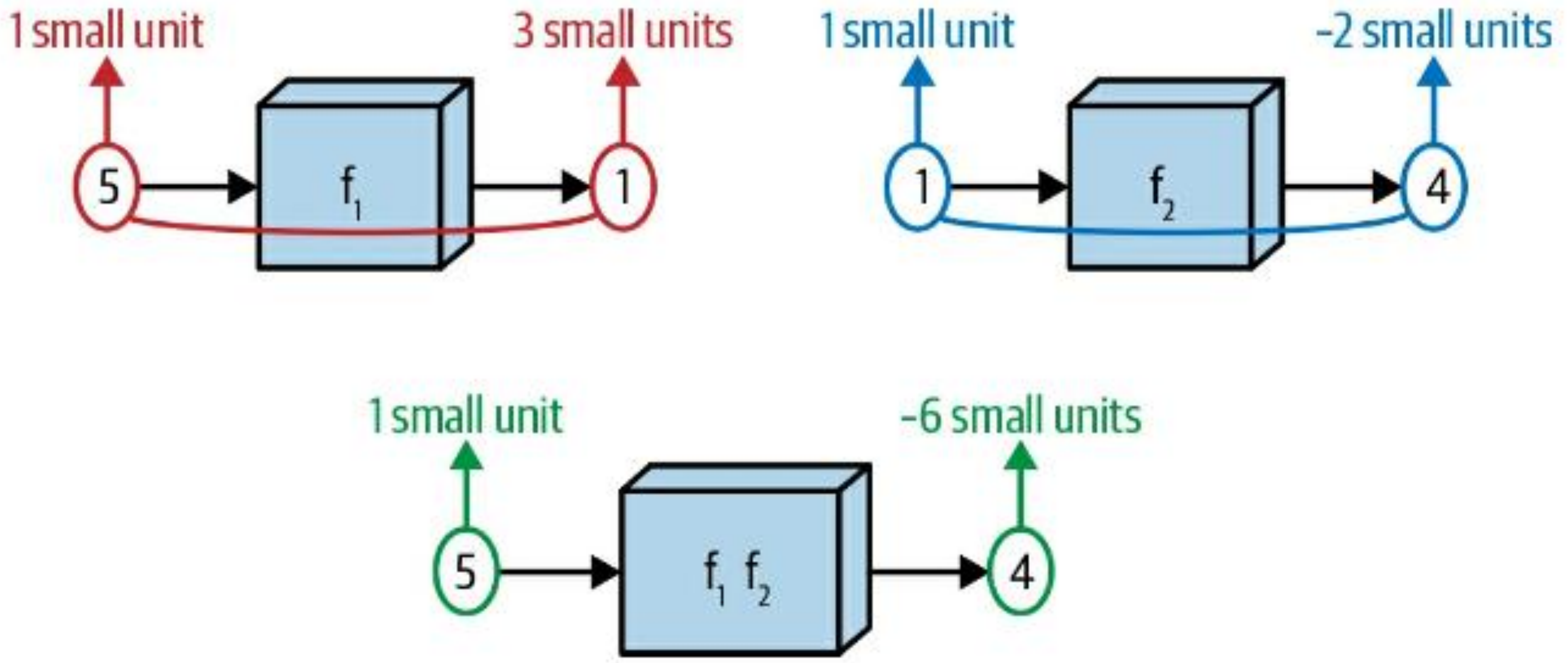
$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx},$$

- Intuitively, the chain rule says that knowing change rate of *z* vs. *y* and *y* vs. *x*, allows one to calculate change rate of *z* vs. *x* as the product of the two rates of change.
  - George F. Simmons: "If a car travels twice as fast as a bicycle and the bicycle is 4 times as fast as a walking man, then the car travels 2 × 4 = 8 times as fast as the man."

https://en.wikipedia.org/wiki/Chain_rule

Illustration of the Chain Rule

- The derivative of the composite function should be a sort of product of the derivatives of its constituent functions.



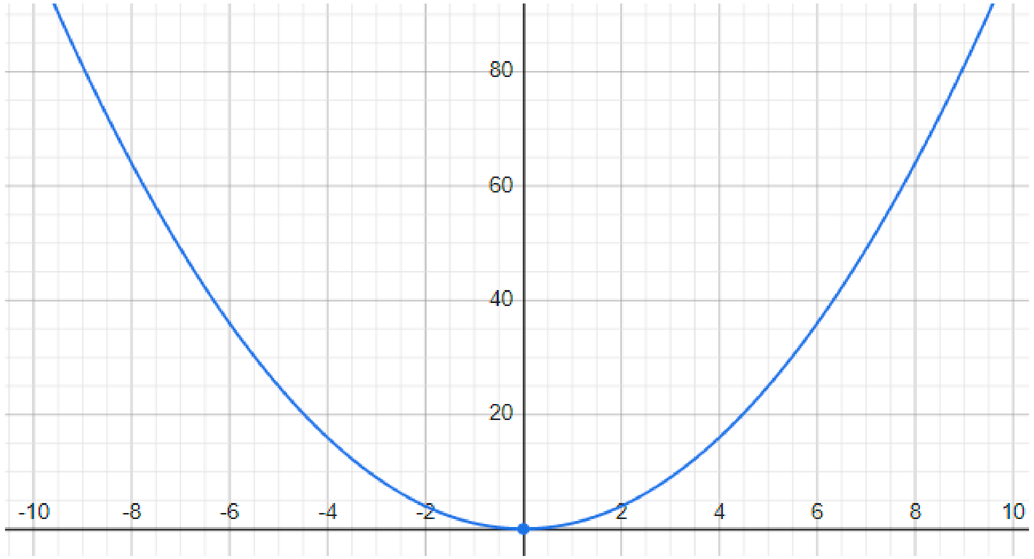
Implement the Chain Rule

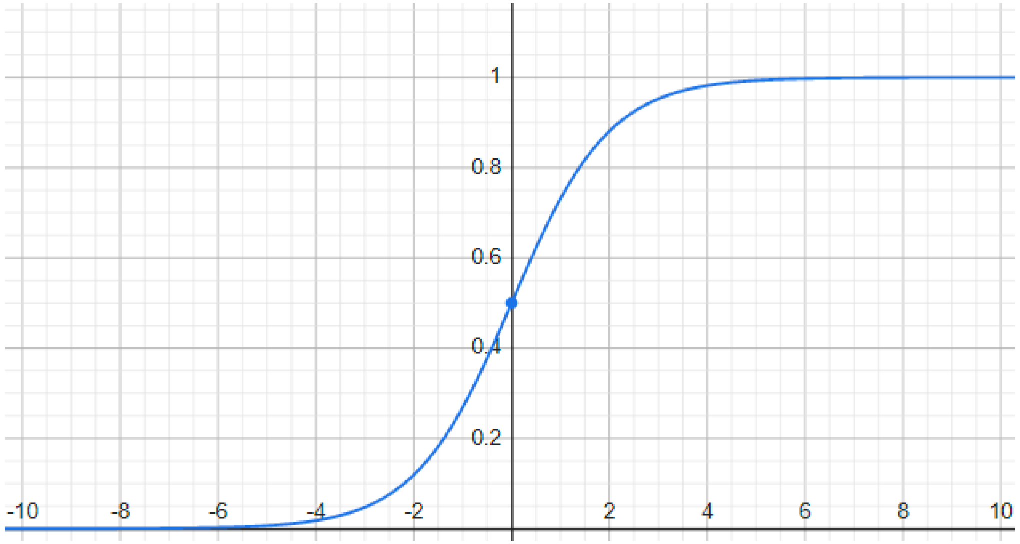
```
def chain_deriv_2(chain: Chain, input_range: ndarray) -> ndarray:

    assert len(chain) == 2
    assert input_range.ndim == 1

    f1 = chain[0]
    f2 = chain[1]

    # df1/dx
    f1_of_x = f1(input_range)

    # df1/du
    df1dx = deriv(f1, input_range)

    # df2/du(f1(x))
    df2du = deriv(f2, f1(input_range))

    return df1dx * df2du
```

$$\frac{df_2}{du}(x) = \frac{df_2}{du}(f_1(x)) \times \frac{df_1}{du}(x)$$

Chain Rule of the Square and Sigmoid

- Implement the Square and Sigmoid functions





Visualizing Functions and Derivatives

- Plot sigmoid(square(x)) and square(sigmoid(x))

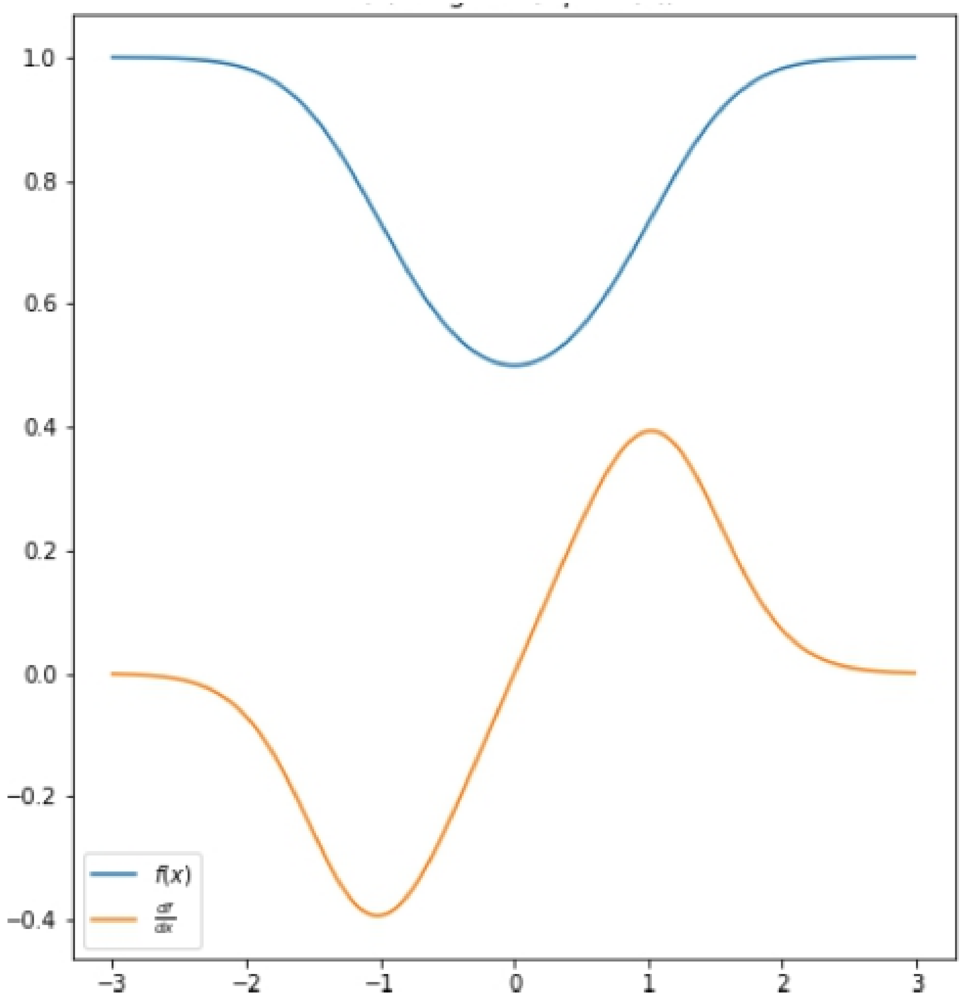
```
def plot_chain(ax, chain: Chain, input_range: ndarray) ->
None:
    assert input_range.ndim == 1, "Function requires a 1
dimensional ndarray as input_range"

    output_range = chain_length_2(chain, input_range)
    ax.plot(input_range, output_range)
```

```
def plot_chain_deriv(ax, chain: Chain, input_range: ndarray)
-> ndarray:
    output_range = chain_deriv_2(chain, input_range)
    ax.plot(input_range, output_range)
```

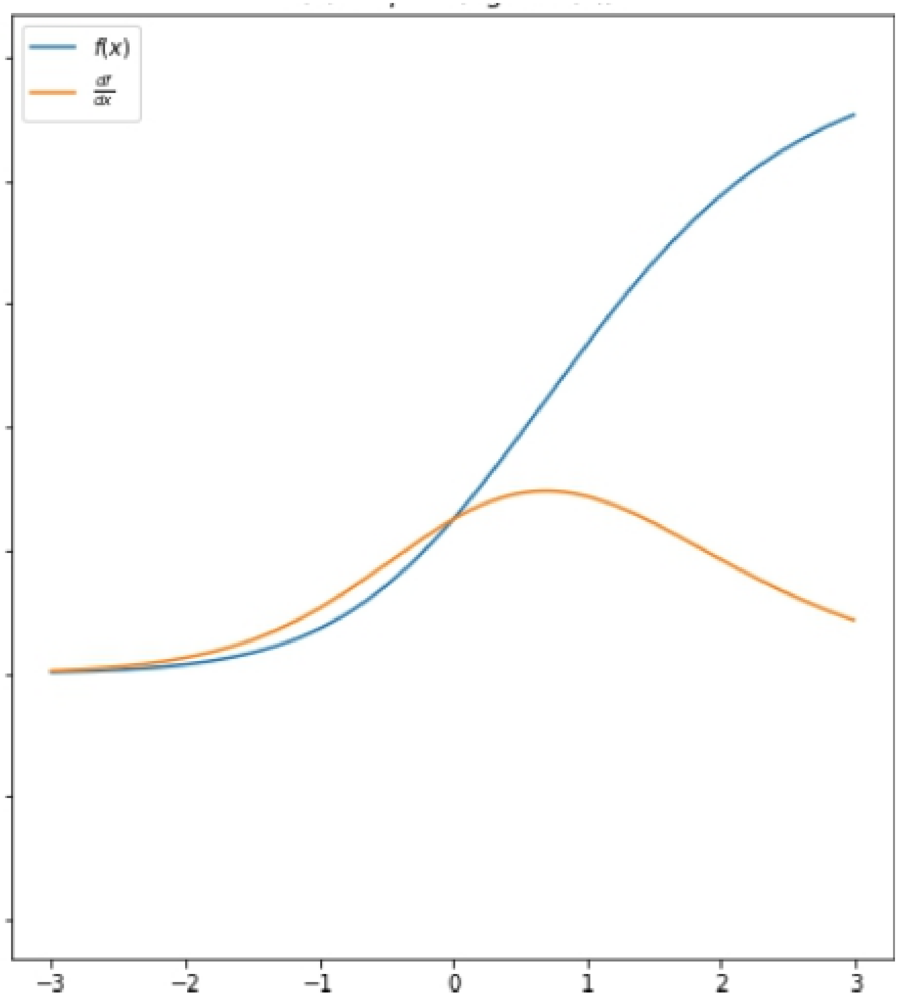
```
PLOT_RANGE = np.arange(-3, 3, 0.01)

chain_1 = [square, sigmoid]
chain_2 = [sigmoid, square]

plot_chain(chain_1, PLOT_RANGE)
plot_chain_deriv(chain_1,
PLOT_RANGE)

plot_chain(chain_2, PLOT_RANGE)
plot_chain_deriv(chain_2,
PLOT_RANGE)
```

Original Functions and their Derivatives

f(x) = sigmoid(square(x))



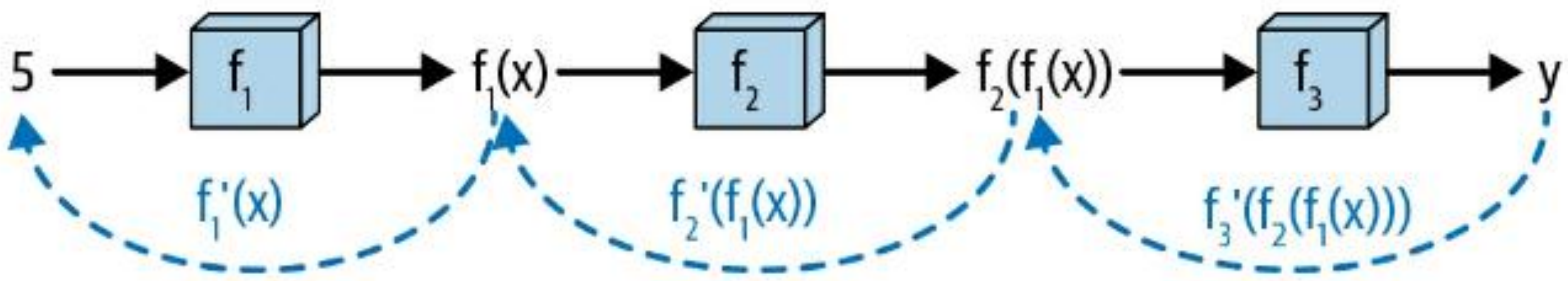
f(x) = square(sigmoid(x))



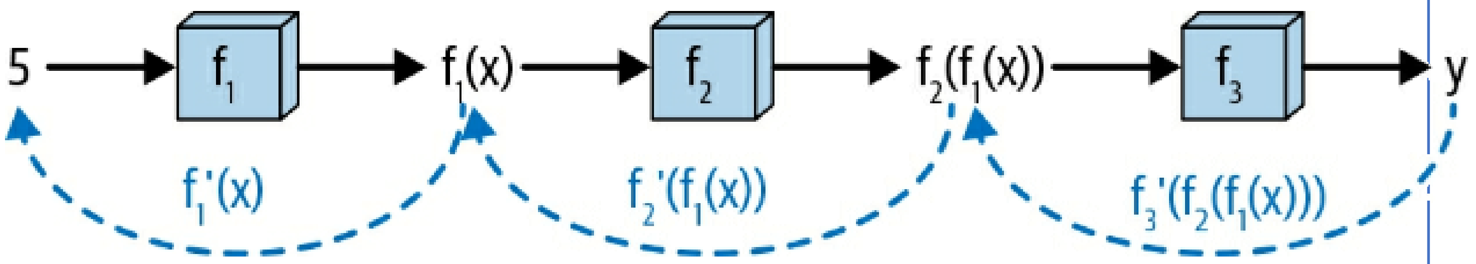
Longer Chain Rule

- Let us try 3 functions

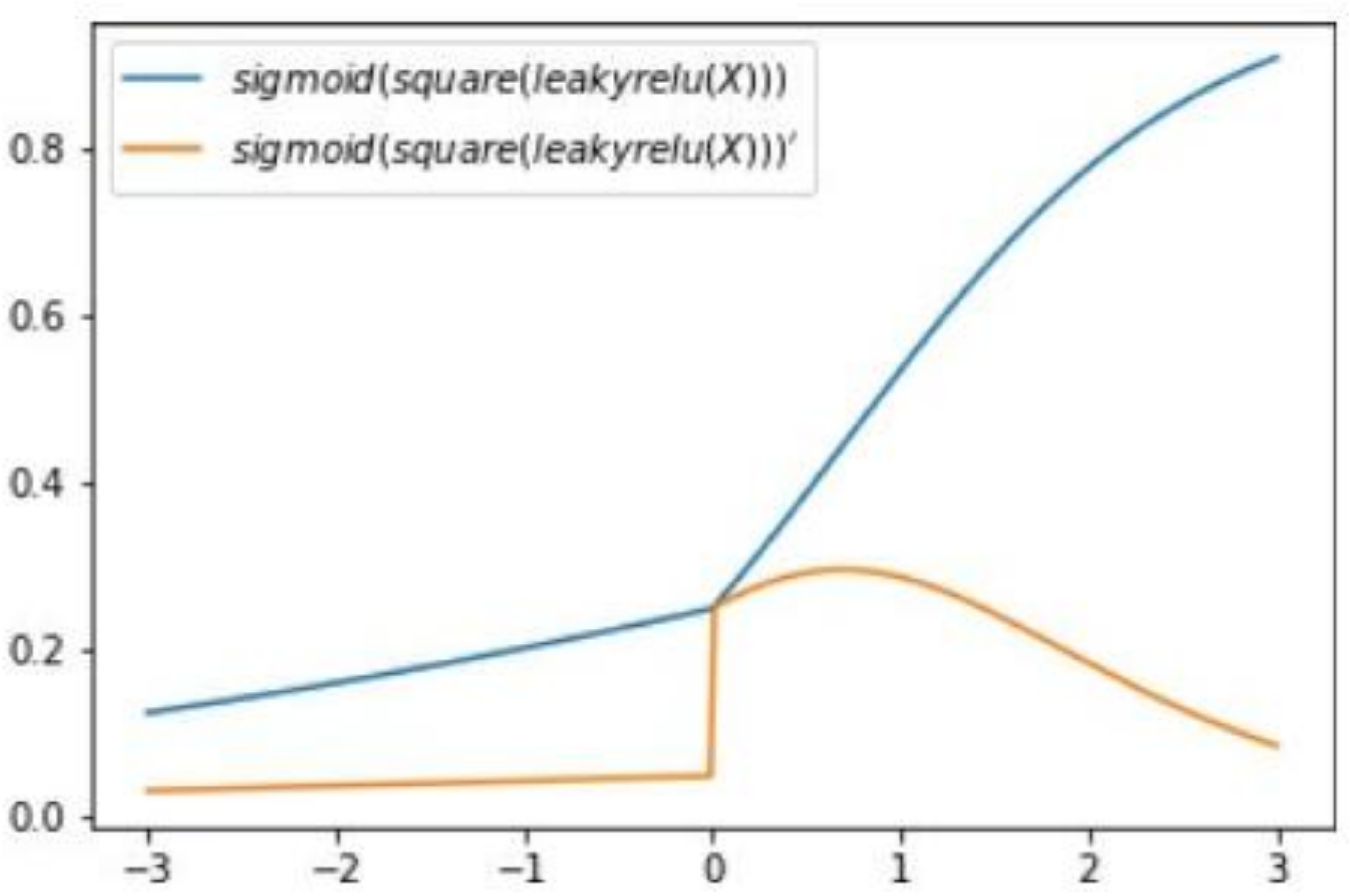
$$\frac{df_3}{du}(x) = \frac{df_3}{du}(f_2(f_1(x))) \times \frac{df_2}{du}(f_1(x)) \times \frac{df_1}{du}(x))$$



```python
def chain_deriv_3(chain: Chain, input_range: ndarray) -> ndarray:
    # Uses the chain rule to compute the derivative of three nested functions:
    # (f3(f2(f1)))' = f3'(f2(f1(x))) * f2'(f1(x)) * f1'(x)
    assert len(chain) == 3, "This function requires 'Chain' objects to have length 3"

    f1 = chain[0]
    f2 = chain[1]
    f3 = chain[2]

    # f1(x)
    f1_of_x = f1(input_range)
    # f2(f1(x))
    f2_of_x = f2(f1_of_x)
    # df3du
    df3du = deriv(f3, f2_of_x)
    # df2du
    df2du = deriv(f2, f1_of_x)
    # df1dx
    df1dx = deriv(f1, input_range)

    # Multiplying these quantities together at each point
    return df1dx * df2du * df3du
```

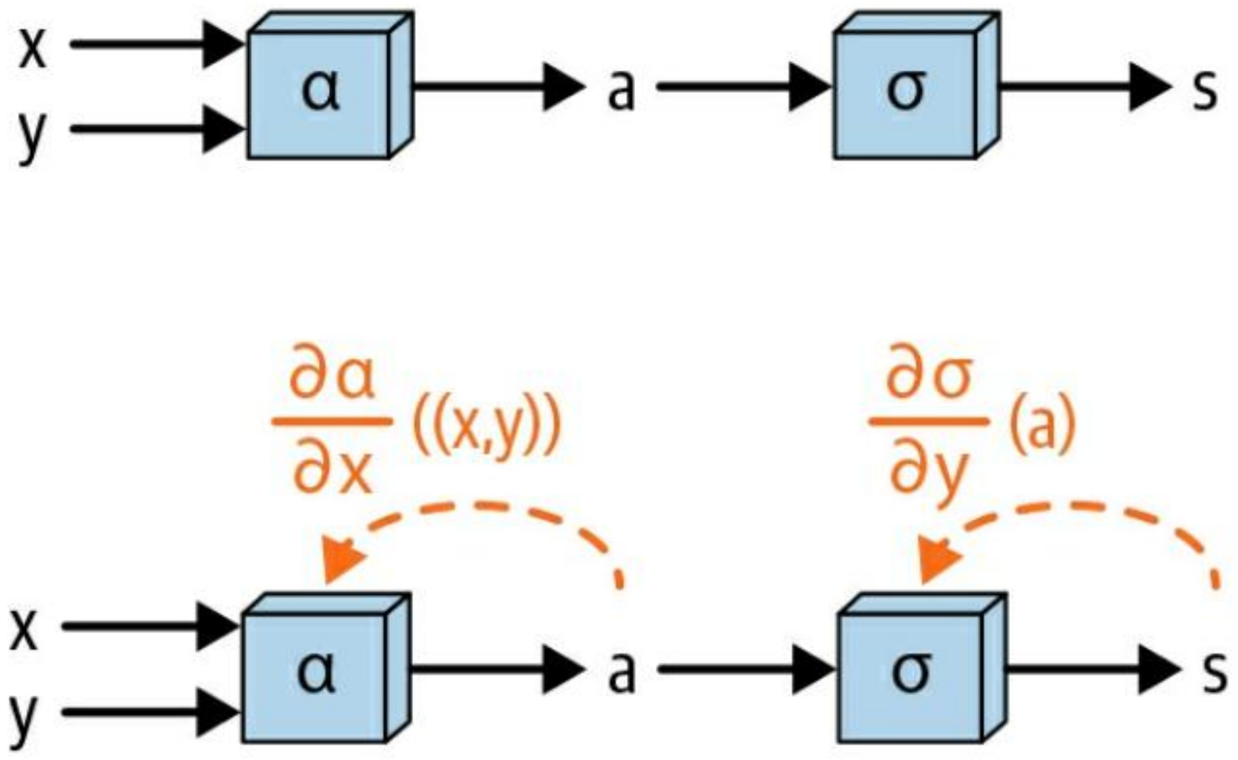


Visualize Our Nested Functions



Functions with Two Inputs

- $\alpha(x, y) = x + y$



Partial Derivative

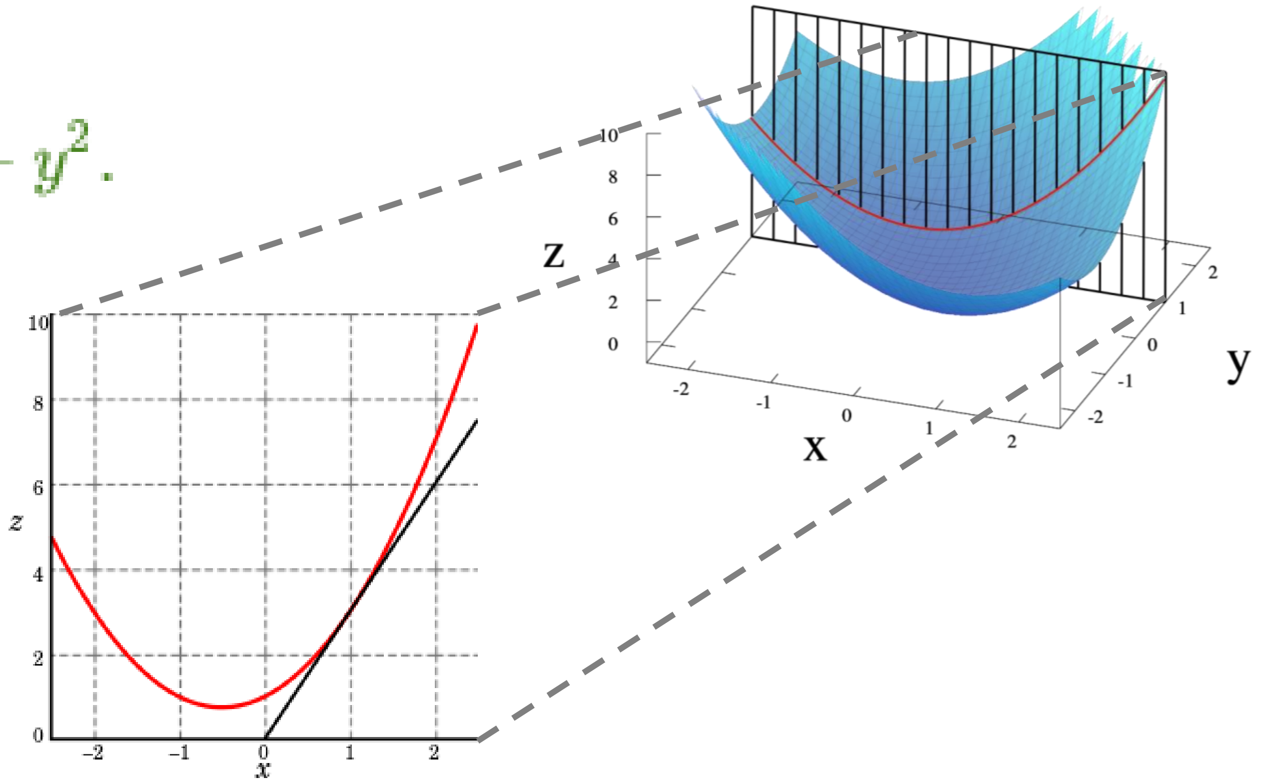
- Partial derivative of a function of several variables is its derivative with respect to one of those variables, with the others held constant

Example:

$$z = f(x, y) = x^2 + xy + y^2.$$

$$\frac{\partial z}{\partial x} = 2x + y.$$

So at (1, 1), by substitution, the slope is 3



https://en.wikipedia.org/wiki/Partial_derivative

Gradient

- An important example of a function of several variables is the case of a scalar-valued function $f(x_1, \ldots, x_n)$ on a domain in Euclidean space $\mathbb{R}^n$. In this case f has a partial derivative with respect to each variable x_j. At the point a, these partial derivatives define the vector

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

Total Derivative

- The chain rule has a particularly elegant statement in terms of total derivatives. It says that, for two functions f and g, the total derivative of the composite function $g \circ f$ at a satisfies

$$d(g \circ f)_a = dg_{f(a)} \cdot df_a .$$

https://en.wikipedia.org/wiki/Total_derivative

Chain Rule for Two functions

Suppose that $x = g(t)$ and $y = h(t)$ are differentiable functions of t and $z = f(x, y)$ is a differentiable function of x and y. Then $z = f(x(t), y(t))$ is a differentiable function of t and

$$\frac{dz}{dt} = \frac{\partial z}{\partial x} \cdot \frac{dx}{dt} + \frac{\partial z}{\partial y} \cdot \frac{dy}{dt}, \tag{14.5.1}$$

where the ordinary derivatives are evaluated at t and the partial derivatives are evaluated at (x, y).

https://math.libretexts.org/Bookshelves/Calculus/Calculus_(OpenStax)/14%3A_Differentiation_of_Functions_of_Several_Variables/14.05%3A_The_Chain_Rule_for_Multivariable_Functions

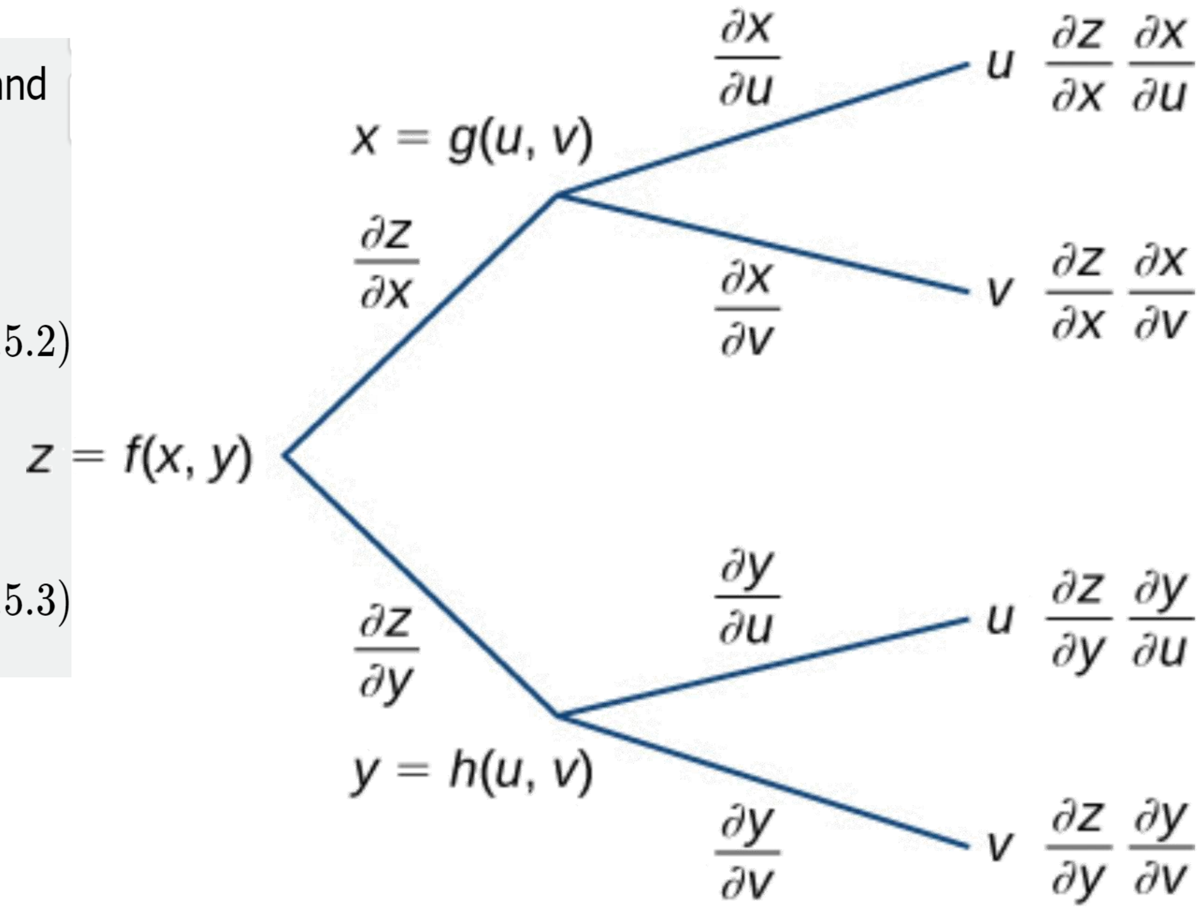
Chain Rule for 2 Functions & 2 Variables

Suppose $x = g(u, v)$ and $y = h(u, v)$ are differentiable functions of u and v, and $z = f(x, y)$ is a differentiable function of x and y. Then, $z = f(g(u, v), h(u, v))$ is a differentiable function of u and v, and

$$\frac{\partial z}{\partial u} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial u} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial u} \qquad (14.5.2)$$

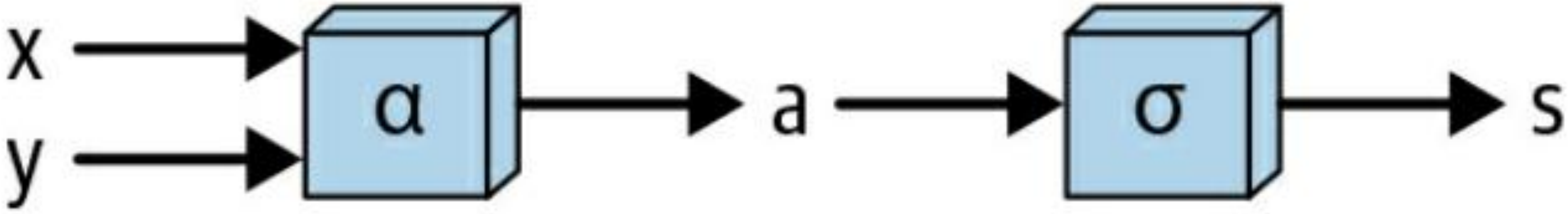
and

$$\frac{\partial z}{\partial v} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial v} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial v}. \qquad (14.5.3)$$



Derivative of Two-Input Function

$$f(x,y) = s(a(x,y)) \quad a = a(x,y) = x + y$$



Derivative of Two-Input Function

$$f(x,y) = s(a(x,y)),\ a = a(x,y) = x + y$$

$$\frac{\partial f}{\partial x} = \frac{\partial \sigma}{\partial u}\big(a(x,y)\big) * \frac{\partial a}{\partial x}\big((x,y)\big) = \frac{\partial \sigma}{\partial u}(x+y) * \frac{\partial a}{\partial x}\big((x,y)\big) = 1$$

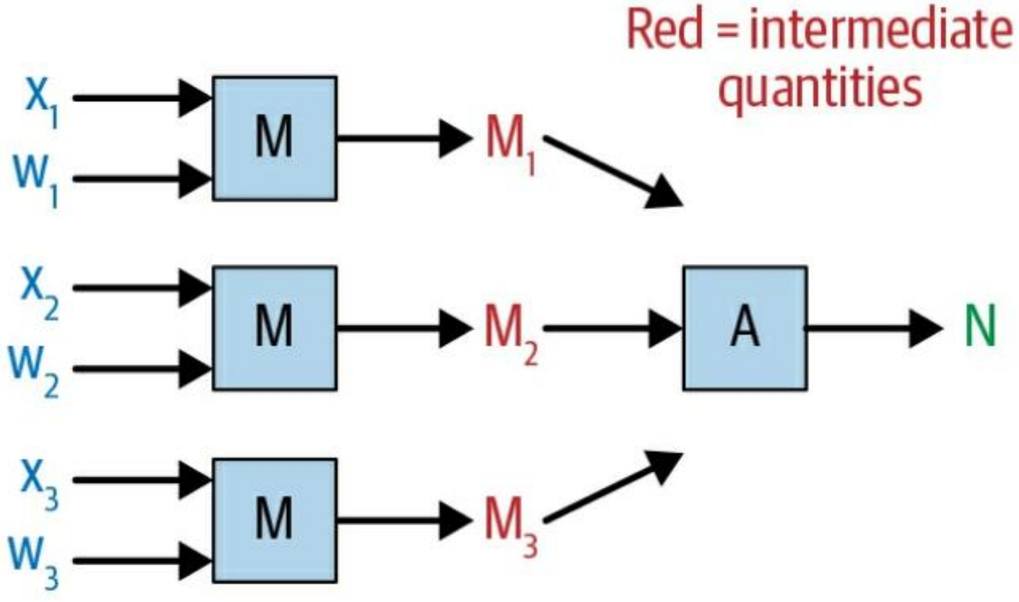
$$\frac{\partial f}{\partial y} = \frac{\partial \sigma}{\partial u}\big(a(x,y)\big) * \frac{\partial a}{\partial y}\big((x,y)\big) = \frac{\partial \sigma}{\partial u}(x+y)$$

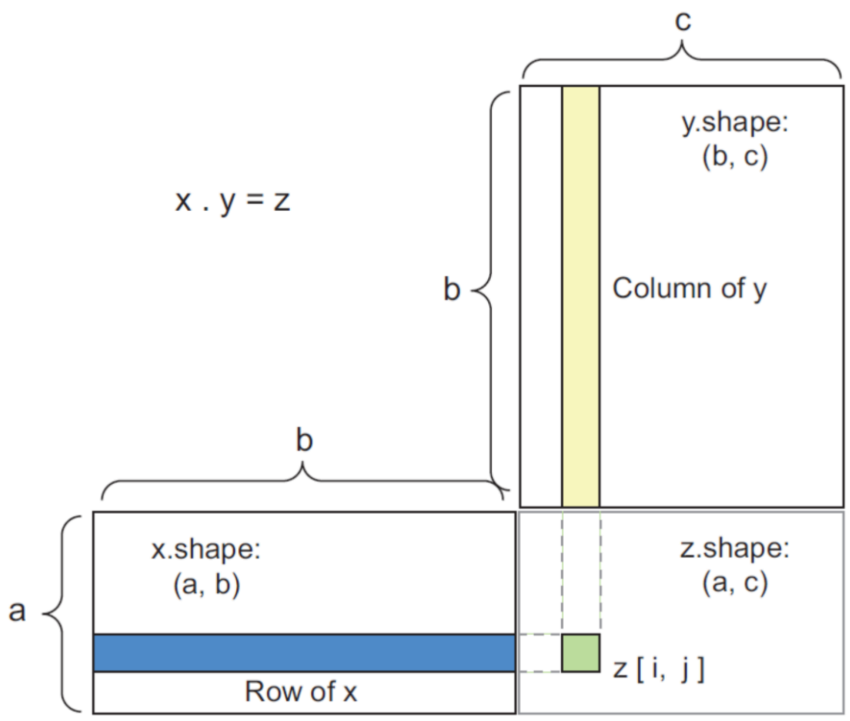
Derivative of Two Inputs Function

```
def multiple_inputs_add_backward(x: ndarray,
                                 y: ndarray,
                                 sigma: Array_Function) -> float:
    '''
    Computes the derivative of this simple function with respect to both inputs.
    '''
    # Compute "forward pass"
    a = x + y

    # Compute derivatives

    dsda = deriv(sigma, a)

    dadx, dady = 1, 1

    return dsda*dadx, dsda*dady
```

Derivative of Multi-Inputs Function

- Dot product (or matrix multiplication) is a concise way to represent many individual operations





Matrix Derivative

- "the derivative regarding a matrix" really means "the derivative regarding each element of the matrix."

$$\frac{\partial v}{\partial X} = \left[\frac{\partial v}{\partial x_1} \quad \frac{\partial v}{\partial x_2} \quad \frac{\partial v}{\partial x_3}\right]$$

$$\frac{\partial v}{\partial x_1} = w_1$$

$$\frac{\partial v}{\partial x_2} = w_2$$

$$\frac{\partial v}{\partial x_3} = w_3$$

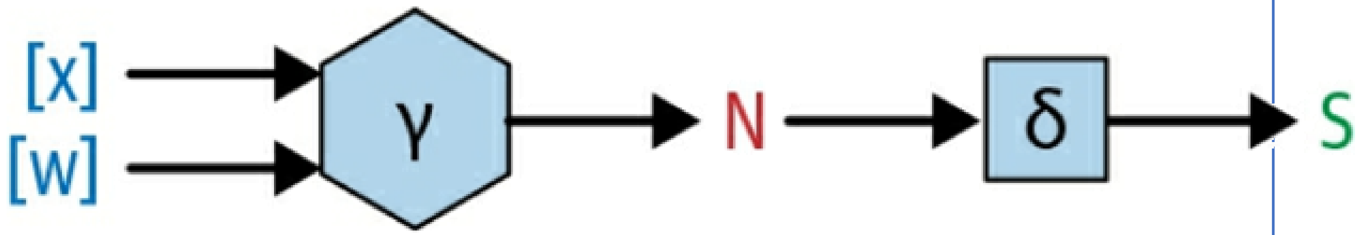
Partial Derivative

$$\frac{\partial v}{\partial X} = [w_1 \quad w_2 \quad w_3] = W^T$$

$$\frac{\partial v}{\partial W} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = X^T$$

Vector Functions and Their Derivatives

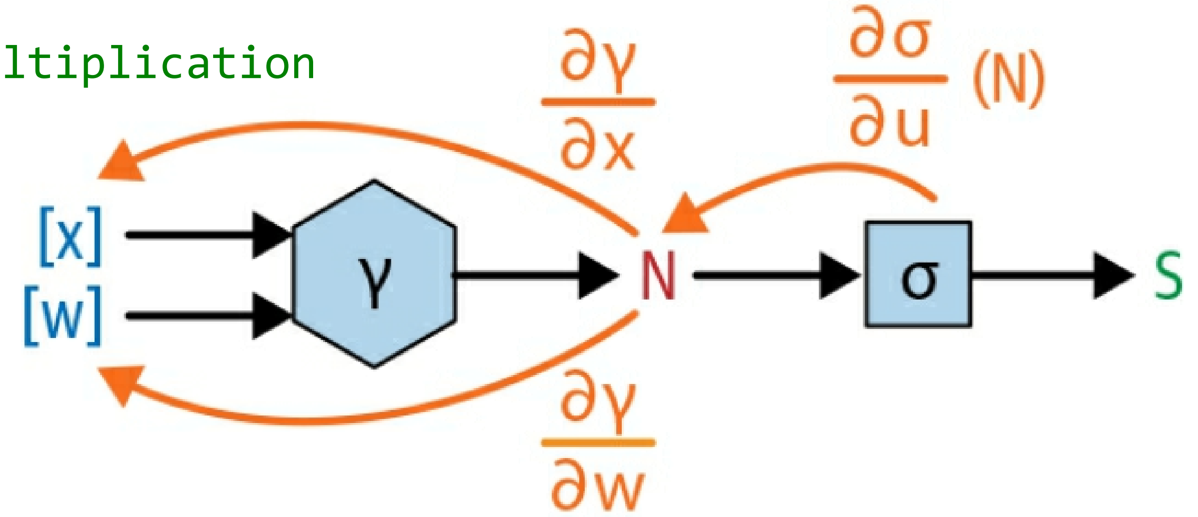
```
def matmul_forward(X: ndarray, W: ndarray) -> ndarray:
    assert X.shape[1] == W.shape[0]
    # matrix multiplication
    N = np.dot(X, W)
    return N

def matmul_backward_first(X: ndarray, W: ndarray) -> ndarray:
    # backward pass
    dNdX = np.transpose(W, (1, 0))
    return dNdX

def matrix_forward_extra(X: ndarray, W: ndarray, sigma: Array_Function) -> ndarray:
    assert X.shape[1] == W.shape[0]
    # matrix multiplication
    N = np.dot(X, W)
    S = sigma(N)
    return S
```



$$\frac{\partial \nu}{\partial X} = [w_1 \quad w_2 \quad w_3] = W^T$$

Vector Functions and Their Derivatives

```
def matrix_function_backward_1(X: ndarray,
                               W: ndarray,
                               sigma: Array_Function) -> ndarray:
        assert X.shape[1] == W.shape[0]

    # matrix multiplication
    N = np.dot(X, W)

    # feeding the output of the matrix multiplication
    S = sigma(N)

    # backward calculation
    dSdN = deriv(sigma, N)

    # dNdX
    dNdX = np.transpose(W, (1, 0))

    # multiply them together; since dNdX is 1x1 here, order doesn't matter
    return np.dot(dSdN, dNdX)
```



Computational Graph with Two 2D Matrix Inputs

- What are the gradients of the output S with respect to X and W?
- Can we simply use the chain rule again?

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \qquad W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

X*W is a Matrix

- For the notion of a “gradient” regarding matrix outputs, we need to sum the final array in the sequence so that the notion of “how much will changing each element of X affect the output” will even make sense.

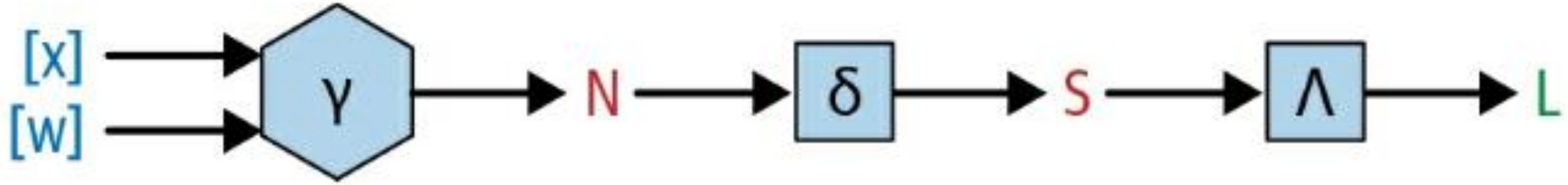
$$\sigma(X * W) = \begin{bmatrix} \sigma(x_{11} * w_{11} + x_{12} * w_{21} + x_{13} * w_{31}) & \sigma(x_{11} * w_{12} + x_{12} * w_{22} + x_{13} * w_{32}) \\ \sigma(x_{21} * w_{11} + x_{22} * w_{21} + x_{23} * w_{31}) & \sigma(x_{21} * w_{12} + x_{22} * w_{22} + x_{23} * w_{32}) \\ \sigma(x_{31} * w_{11} + x_{32} * w_{21} + x_{33} * w_{31}) & \sigma(x_{31} * w_{12} + x_{32} * w_{22} + x_{33} * w_{32}) \end{bmatrix}$$

$$= \begin{bmatrix} \sigma(XW_{11}) & \sigma(XW_{12}) \\ \sigma(XW_{21}) & \sigma(XW_{22}) \\ \sigma(XW_{31}) & \sigma(XW_{32}) \end{bmatrix}$$

Weidman, Seth. Deep Learning from Scratch: Building with Python from First Principles (pp. 30-31). O'Reilly Media.

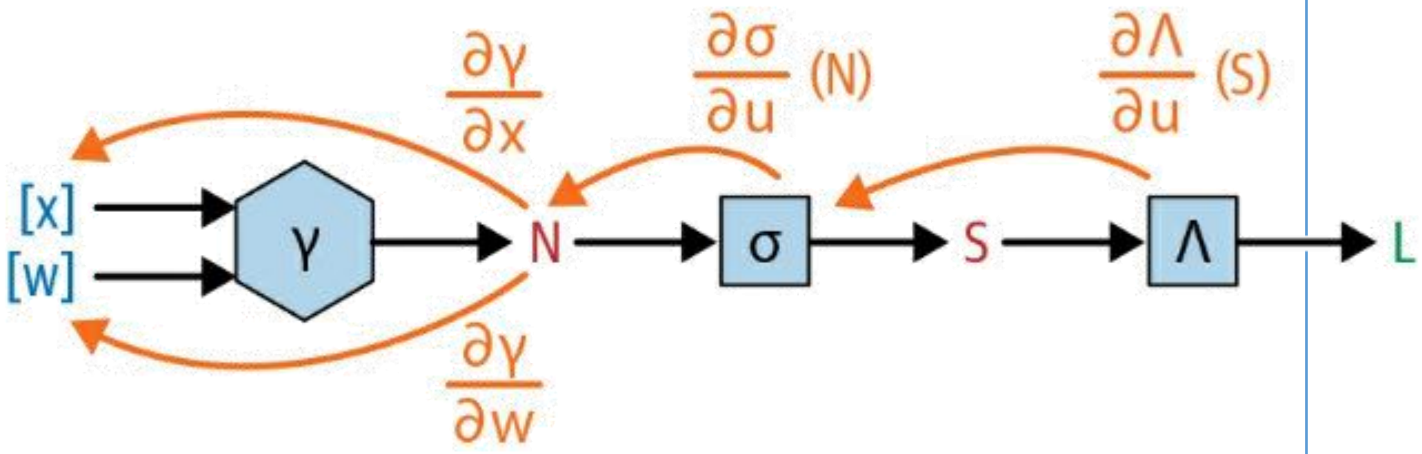
Sum Up the Matrix Output

- Add a sum up function Λ




```
def matrix_function_forward_sum(X: ndarray, W: ndarray,
                                sigma: Array_Function) -> float:
    assert X.shape[1] == W.shape[0]

    # matrix multiplication
    N = np.dot(X, W)
    # feeding the output of the matrix multiplication through sigma
    S = sigma(N)
    # sum all the elements
    L = np.sum(S)
    return L
```

```
def matrix_function_backward_sum_1(X: ndarray, W: ndarray,
                                    sigma: Array_Function) -> ndarray:
    assert X.shape[1] == W.shape[0]
    # matrix multiplication
    N = np.dot(X, W)
    S = sigma(N)
    # sum all the elements
    L = np.sum(S)

    # dLdS - just 1s
    dLdS = np.ones_like(S)
    # dSdN
    dSdN = deriv(sigma, N)
    # dLdN
    dLdN = dLdS * dSdN
    # dNdX
    dNdX = np.transpose(W, (1, 0))
    # dLdX
    dLdX = np.dot(dSdN, dNdX)

    return dLdX
```



Optimization

The *standard form* of a continuous optimization problem is[1]

min.
s.b.t.

$$
\begin{aligned}
&\underset{x}{\text{minimize}} && f(x) \\
&\text{subject to} && g_i(x) \leq 0, \quad i = 1, \dots, m \\
& && h_j(x) = 0, \quad j = 1, \dots, p
\end{aligned}
$$

where

- $f : \mathbb{R}^n \to \mathbb{R}$ is the **objective function** to be minimized over the *n*-variable vector x,
- $g_i(x) \leq 0$ are called **inequality constraints**
- $h_j(x) = 0$ are called **equality constraints**, and
- $m \geq 0$ *and* $p \geq 0$.

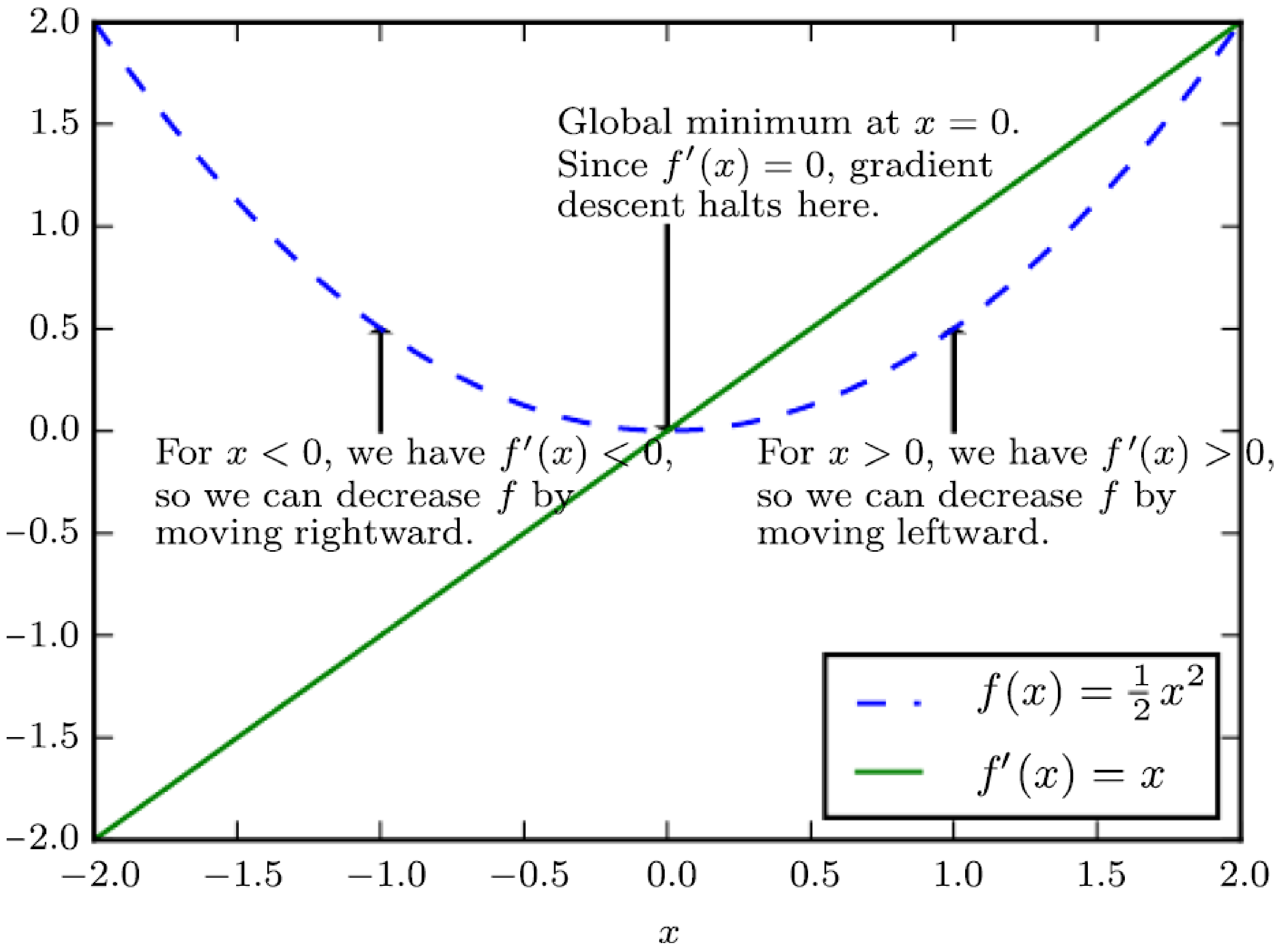
https://en.wikipedia.org/wiki/Optimization_problem

Gradient-based Optimization

- Gradient Descent (Cauchy, 1847):

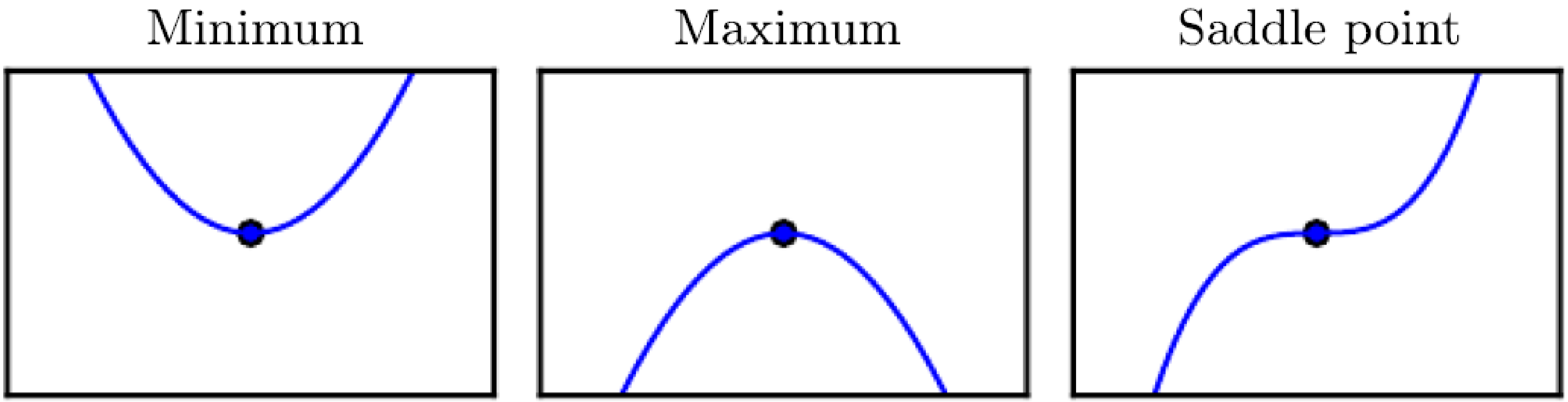
Reduce f(x) by moving x in small steps with opposite sign of the derivative

$$-f(x - \alpha * f'(x))$$

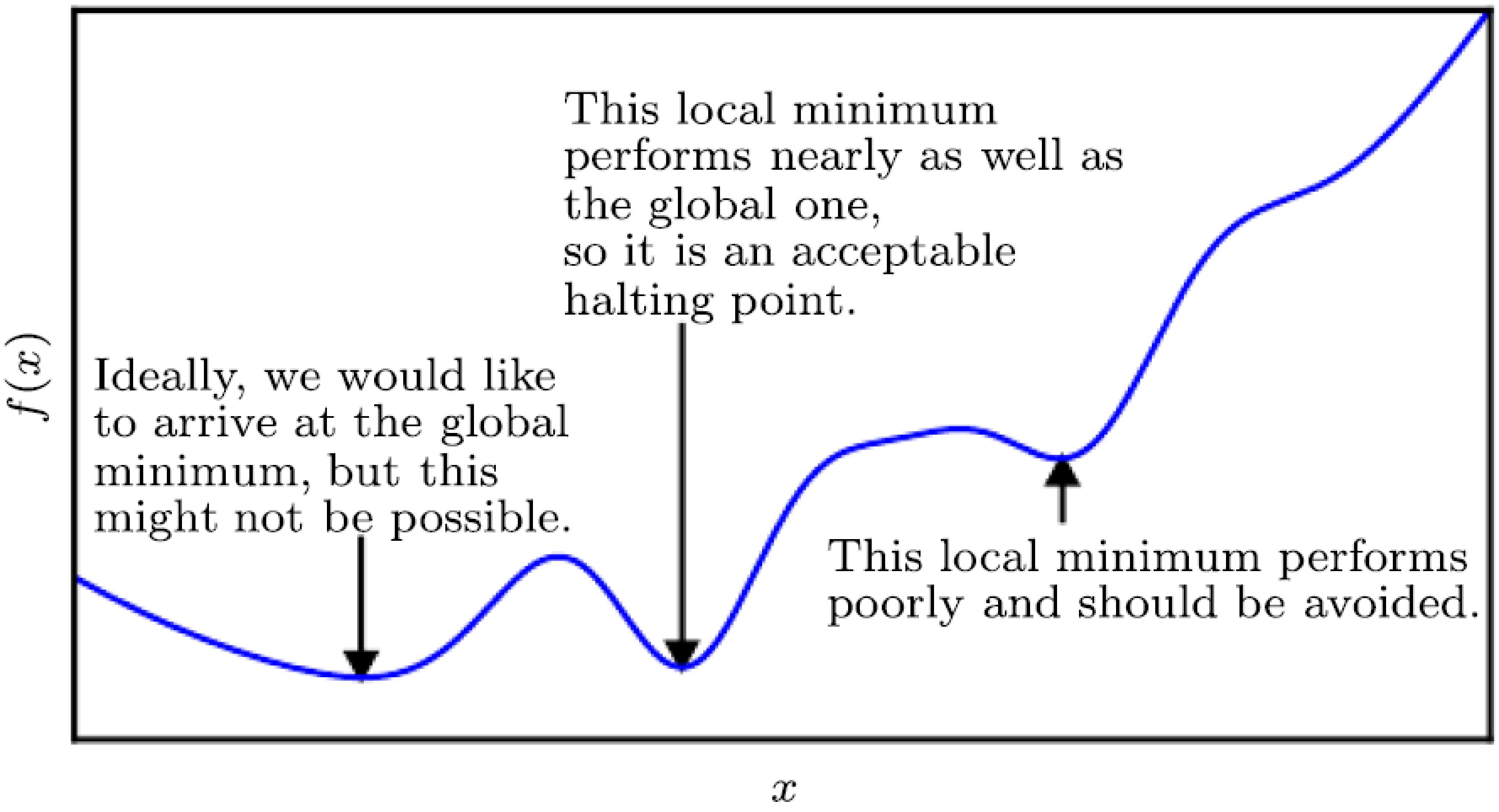


Critical Points (Stationary Points)

- $f'(x)=0$

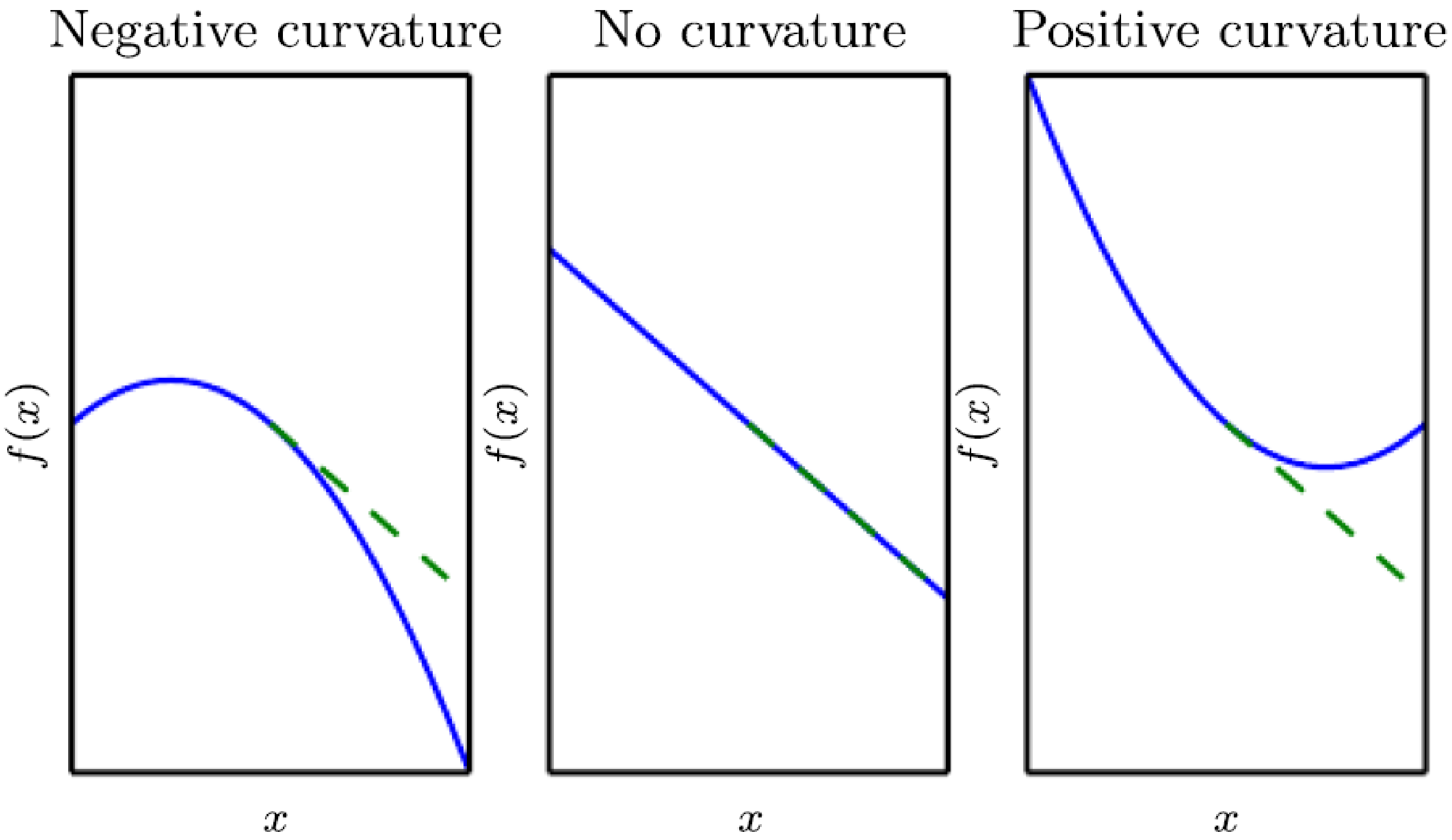


Local Minimum vs. Global Minimum



Second Derivative $f''(x)$

- Second Derivative $f''(x)$ *measures the curvature*



Hessian Matrix

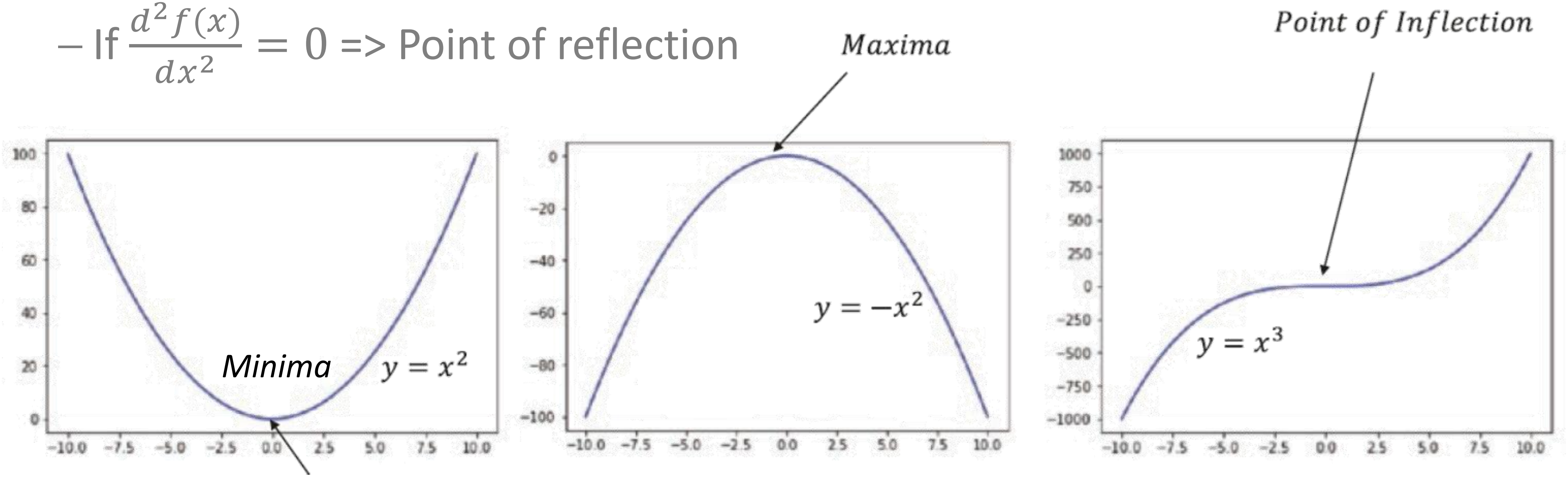
- denoted by H or, ∇^2

$$\mathbf{H}_f = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\,\partial x_n} \\[2.2ex] \dfrac{\partial^2 f}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\,\partial x_n} \\[2.2ex] \vdots & \vdots & \ddots & \vdots \\[2.2ex] \dfrac{\partial^2 f}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

https://en.wikipedia.org/wiki/Hessian_matrix

Maxima and Minima for Univariate Function

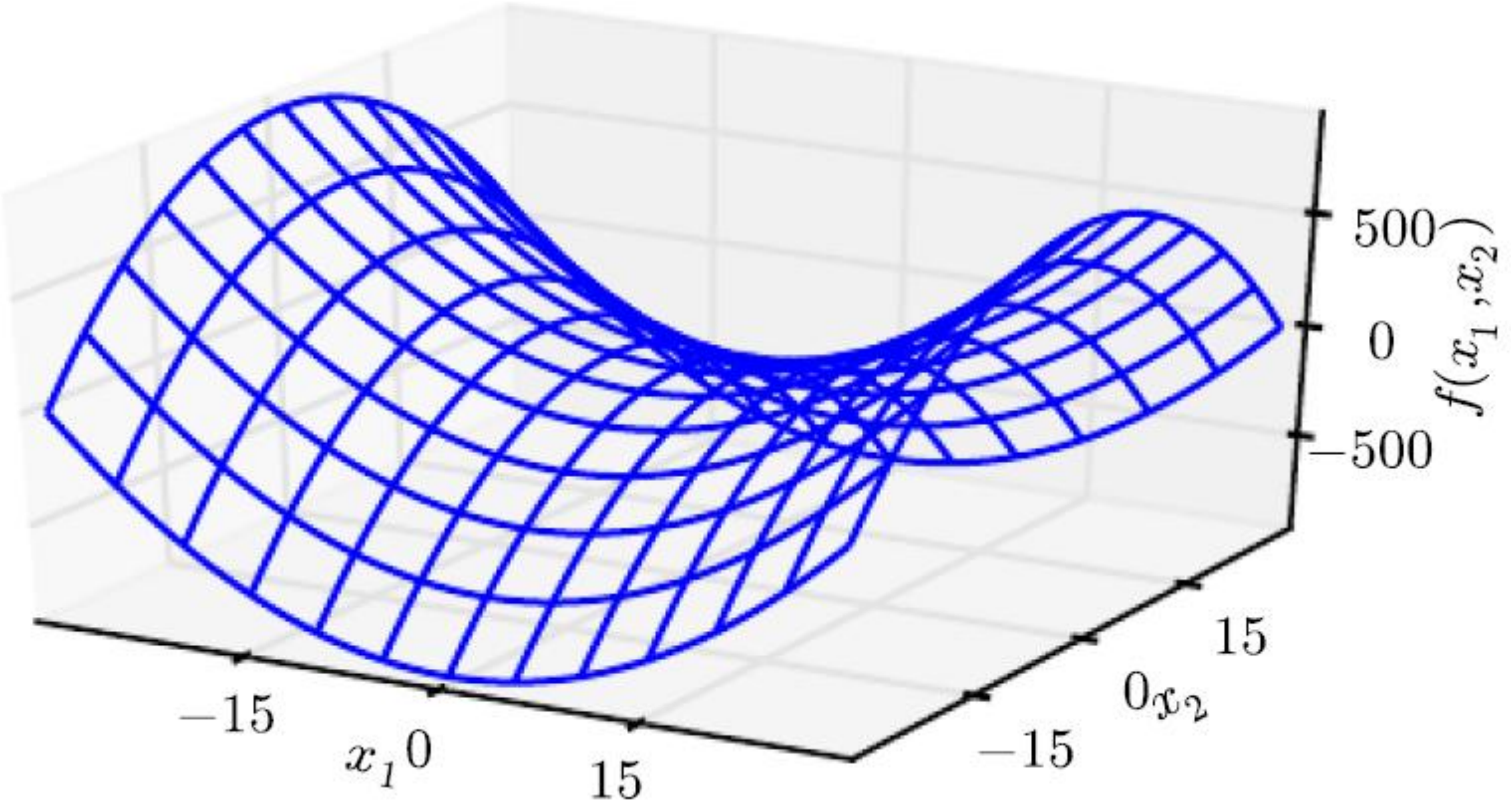
- If $\frac{df(x)}{dx} = 0$, it's a minima or a maxima point, then we study the second derivative:
  - If $\frac{d^2f(x)}{dx^2} < 0$ => Maxima
  - If $\frac{d^2f(x)}{dx^2} > 0$ => Minima
  - If $\frac{d^2f(x)}{dx^2} = 0$ => Point of reflection



Saddle Point

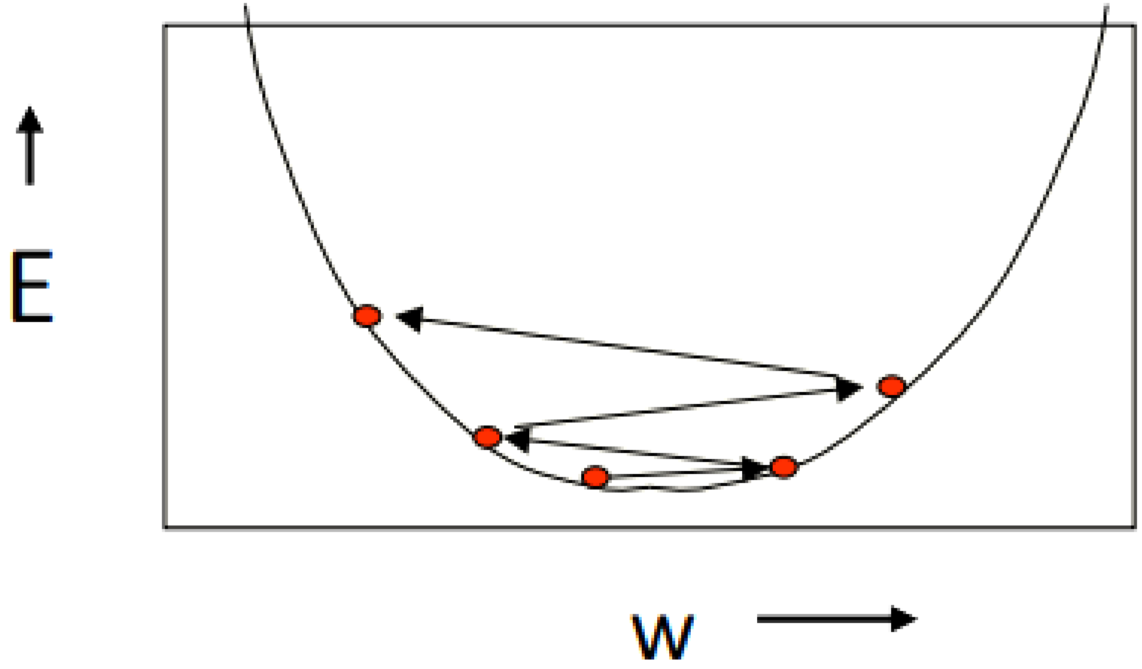
- A saddle point contains both positive and negative curvature.

$$f(x) = x_1^2 - x_2^2$$



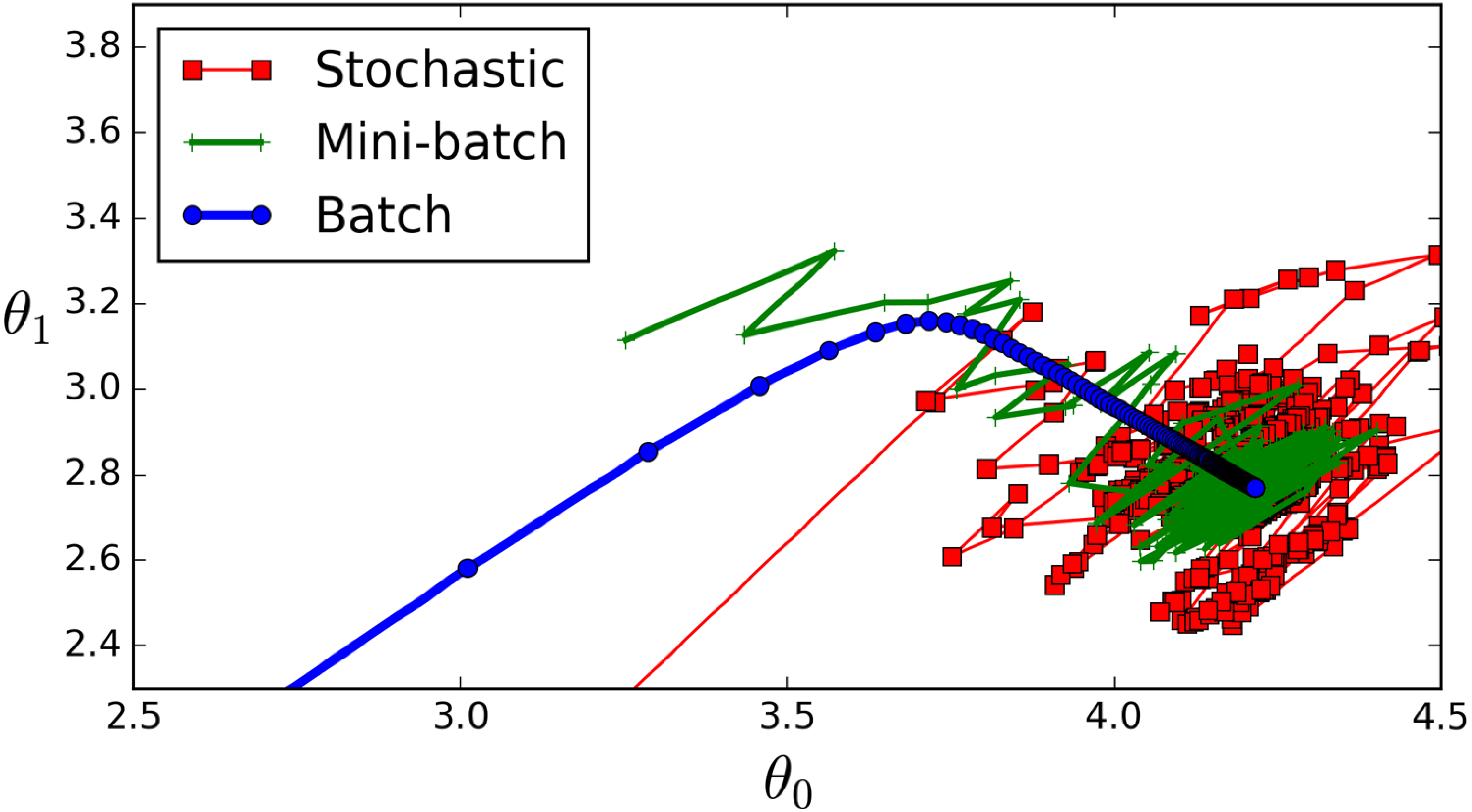
How the Learning Goes Wrong

- If the learning rate is too big, this oscillation diverges
- What we would like to achieve:
  - Move quickly in directions with small but consistent gradients.
  - Move slowly in directions with big but inconsistent gradients.



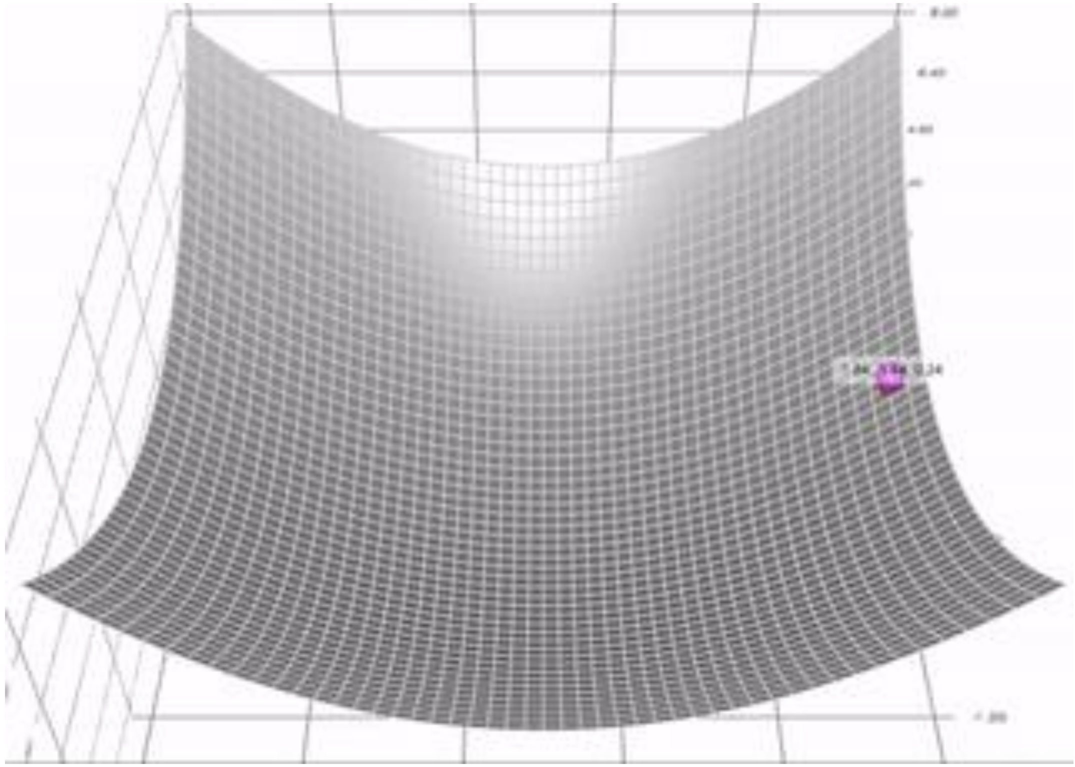
Geoffrey Hinton et al., https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

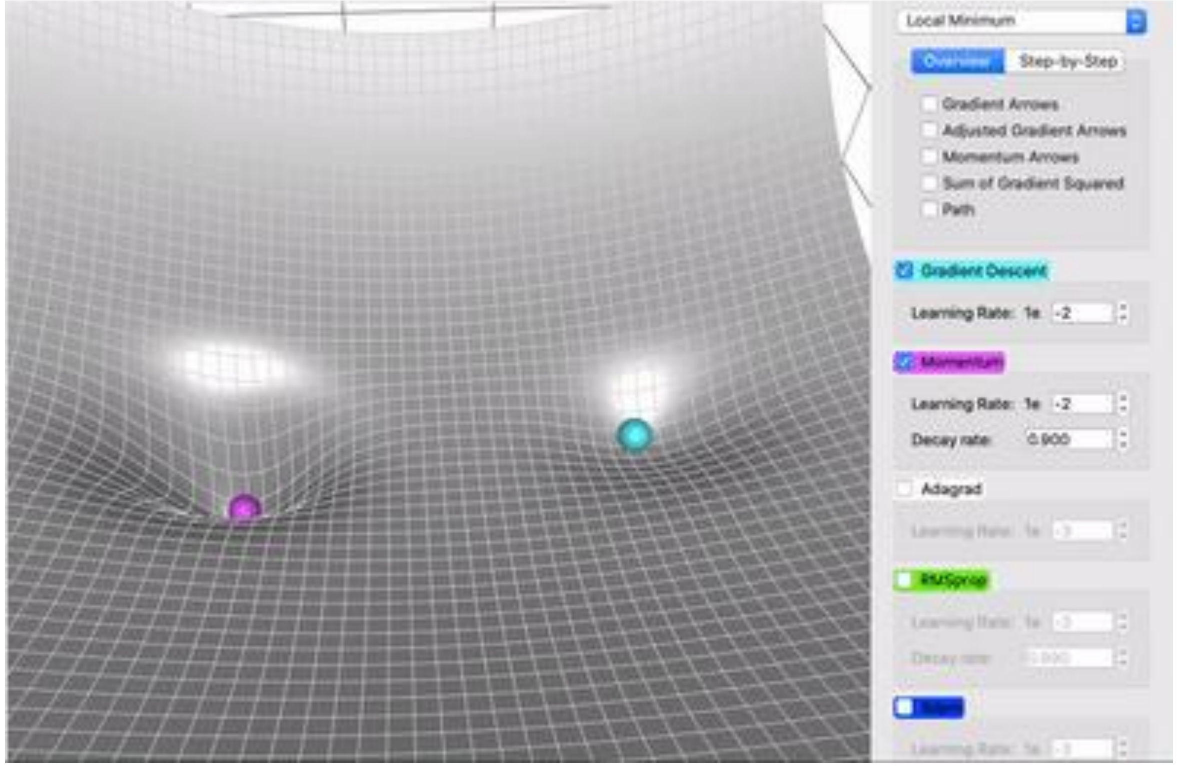
Select Training Samples



Momentum

- $\Delta_t \leftarrow -\alpha * f'(x) + \Delta_{t-1} * \tau$
- $w_t \leftarrow w_{t-1} + \Delta_t$

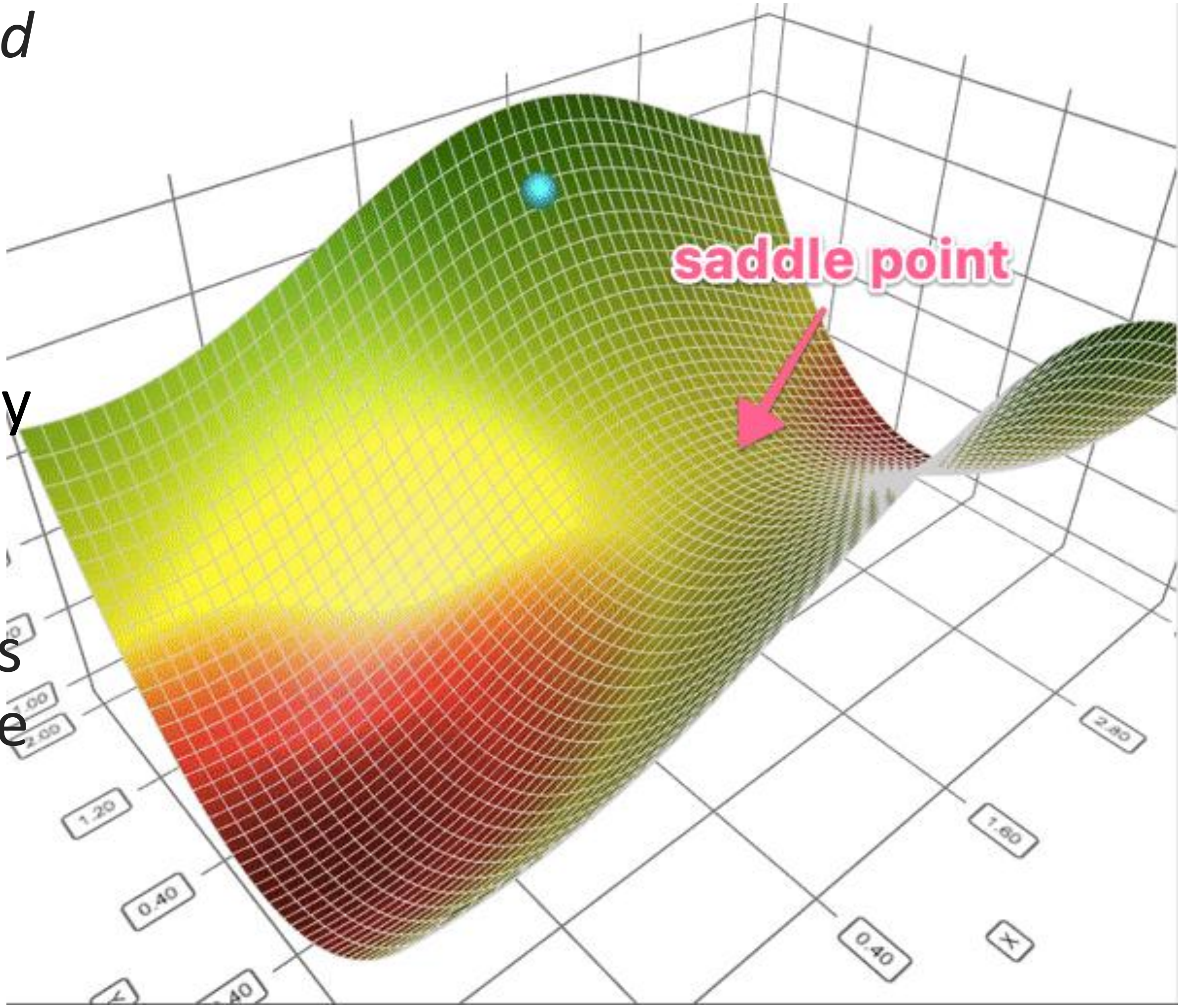




https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325 c

Adaptive Gradient algorithm (AdaGrad)

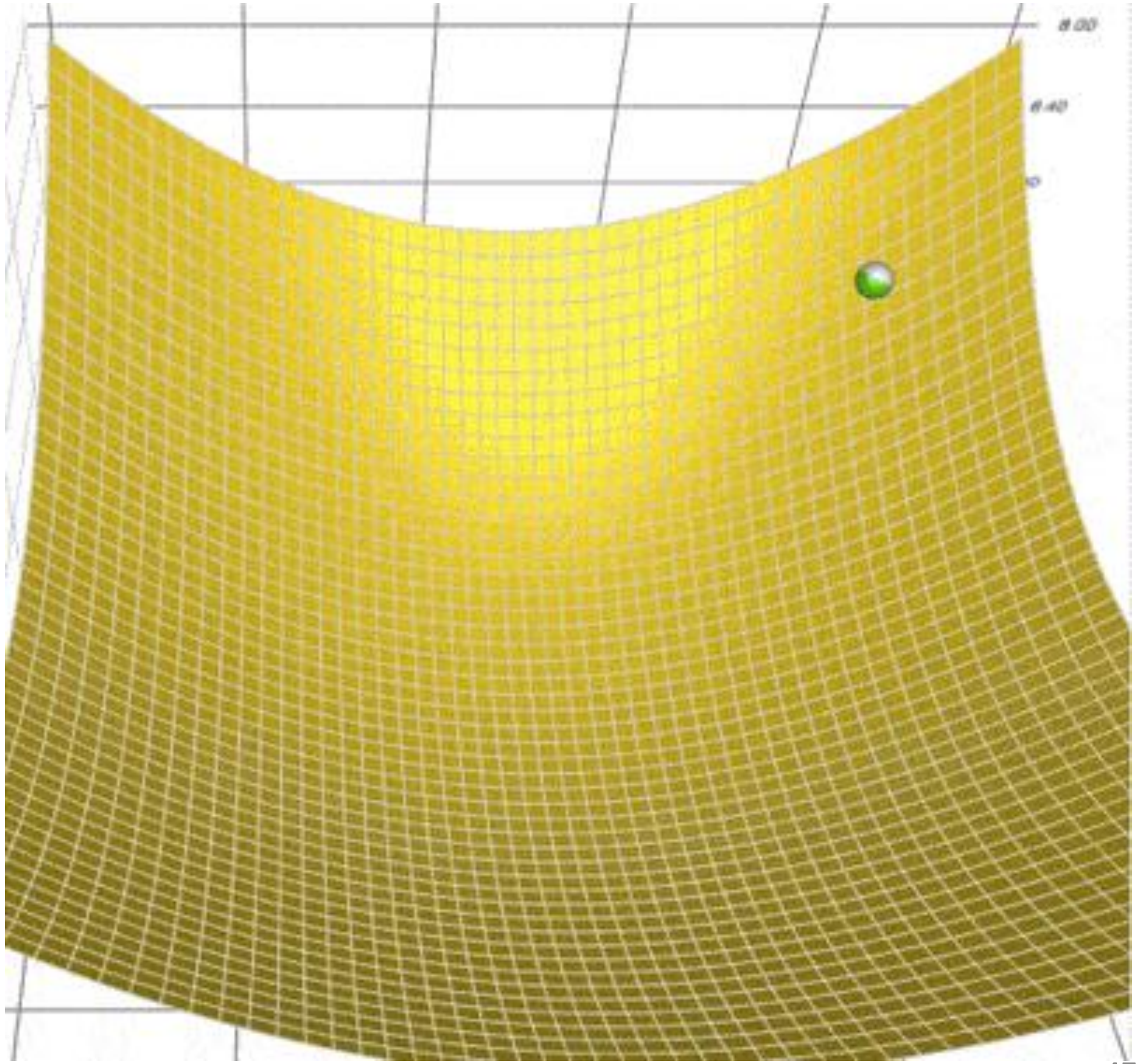
- Keeps track of the sum of gradient *squared*
  - $\Sigma_t \leftarrow \Sigma_{t-1} + \{f'(x)\}^2$
  - $\Delta_t \leftarrow -\alpha * f'(x) * \frac{1}{\sqrt{\Sigma_t}}$
  - $w_t \leftarrow w_{t-1} + \Delta_t$
- In ML optimization, some features are very sparse, so the average gradient is small and training is slow.
- AdaGrad addresses this problem using this idea: the more you have updated a feature already, the less you will update it in the future



https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325 c

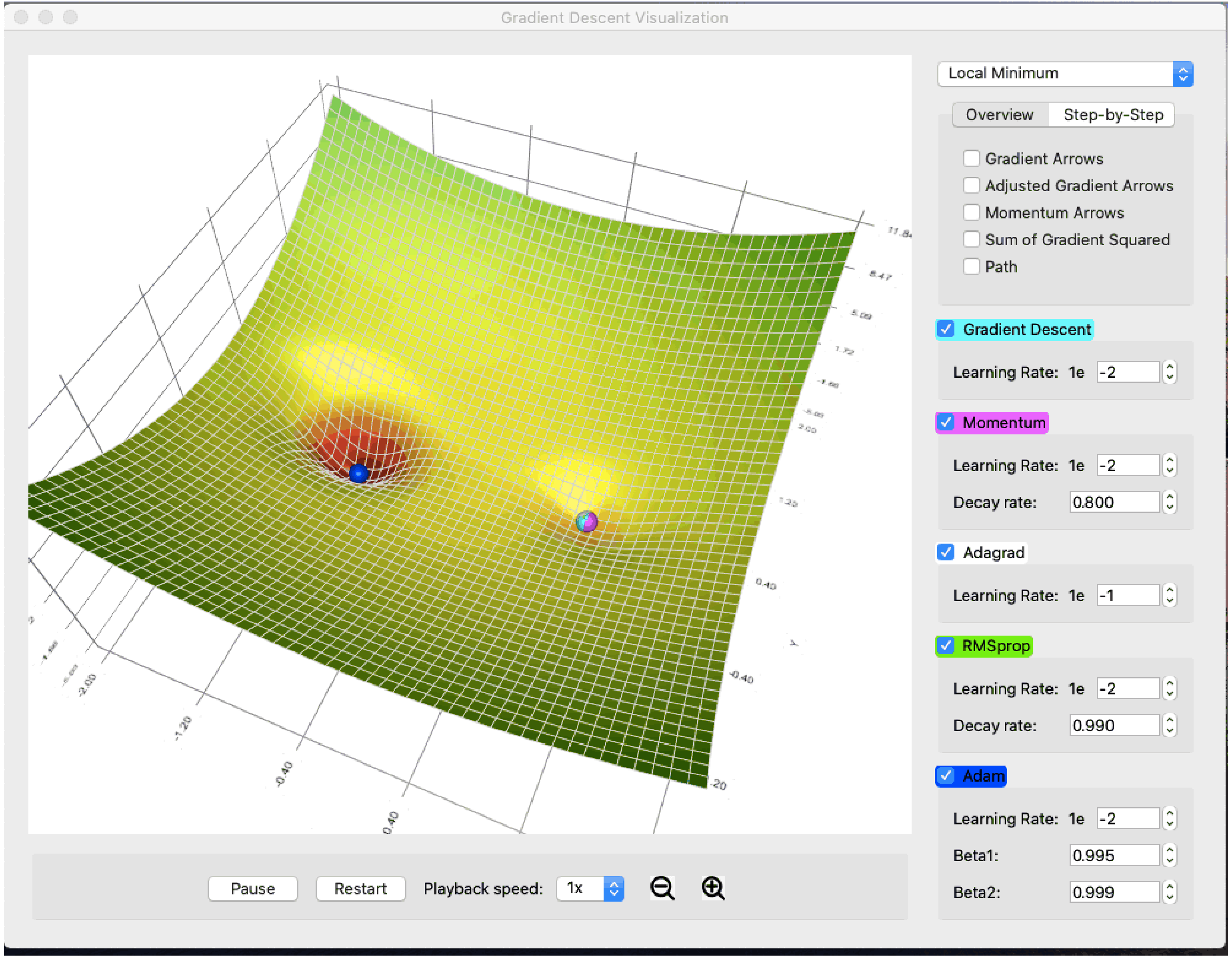
Root Mean Square Propagation (RMSProp)

- AdaGrad is too slow
- RMSProp adds a decay rate ε for updating gradient *squared*
  - $\Sigma_t \leftarrow \Sigma_{t-1} * \varepsilon + \{f'(x)\}^2 * (1-\varepsilon)$
  - $\Delta_t \leftarrow -\alpha * f'(x) * \frac{1}{\sqrt{\Sigma_t}}$
  - $w_t \leftarrow w_{t-1} + \Delta_t$



Adaptive Moment Estimation (ADAM)

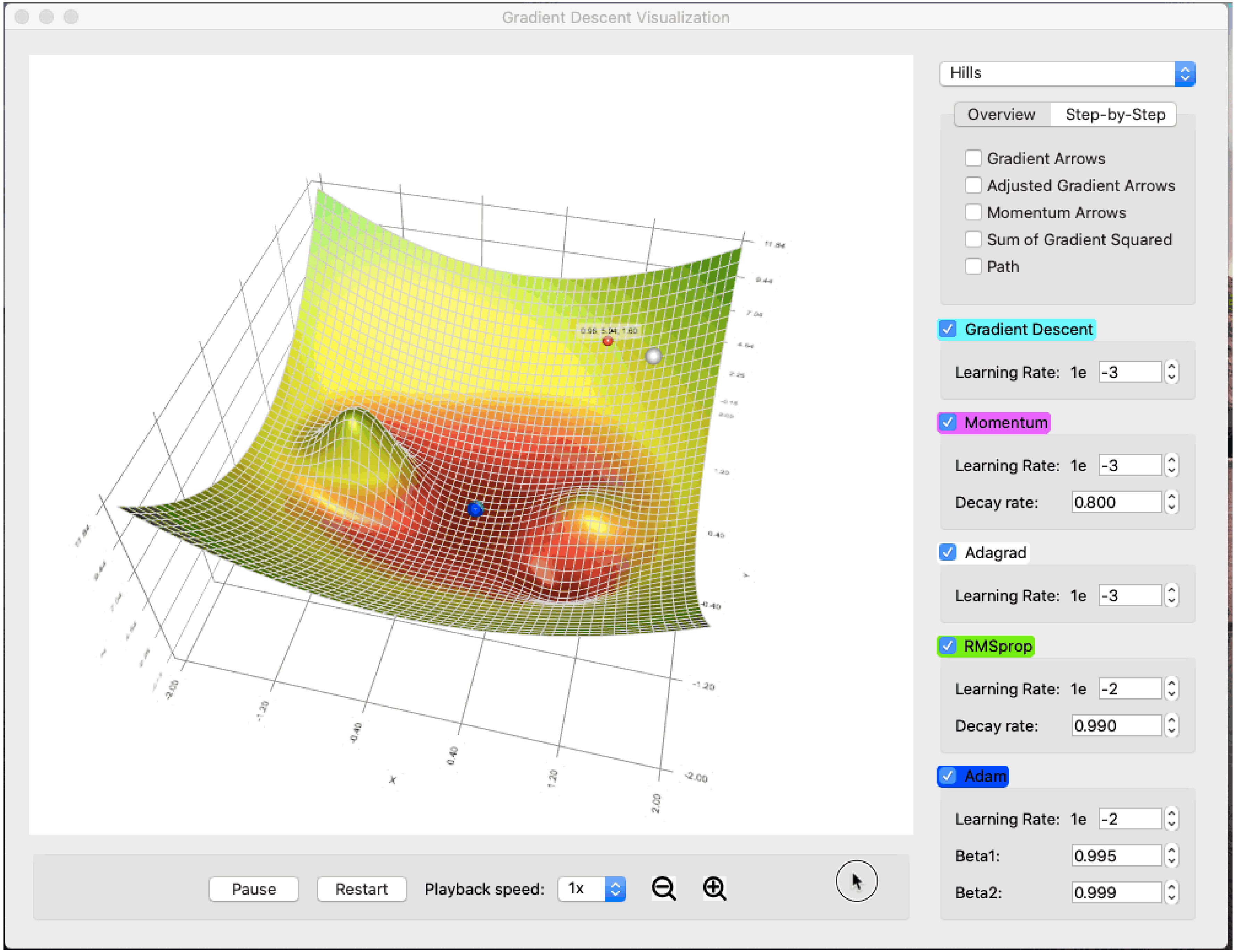
- Momentum + RMSProp
- Lilipads GD Viz tool



https://github.com/lilipads/gradient_descent_viz

Comparing Methods

- RMSProp and ADAM can handle the saddle point better



https://github.com/lilipads/gradient_descent_viz

References

- https://en.wikipedia.org/wiki/Calculus
- Seth Weidman, "Deep Learning from Scratch," Chapter 1, O'Reilly Media, Inc., 2019
- Ian Goodfellow and Yoshua Bengio and Aaron Courville, "Deep Learning," Chapter 4, MIT Press, 2016
- https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325