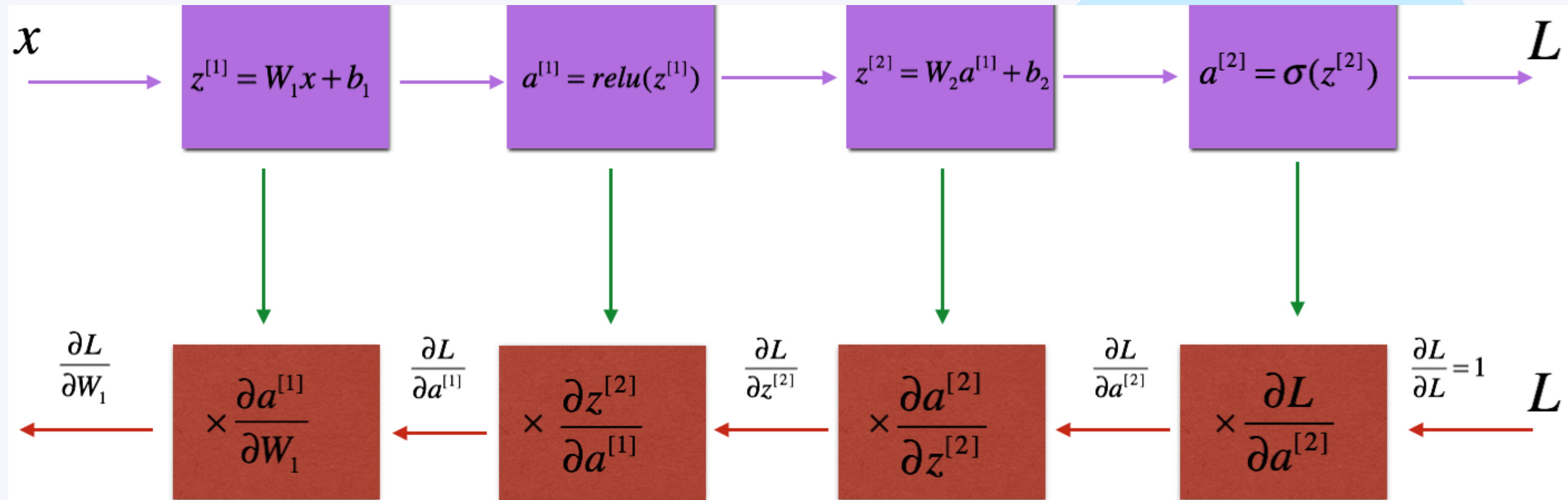



Backpropagation

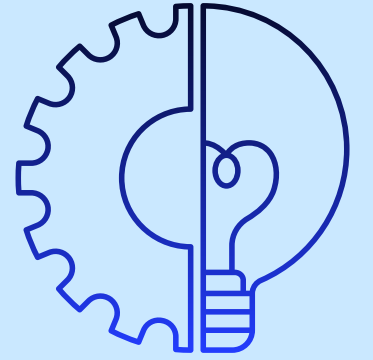
backprop



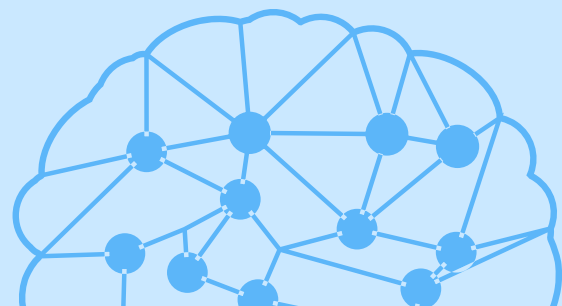

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial W_1}$$



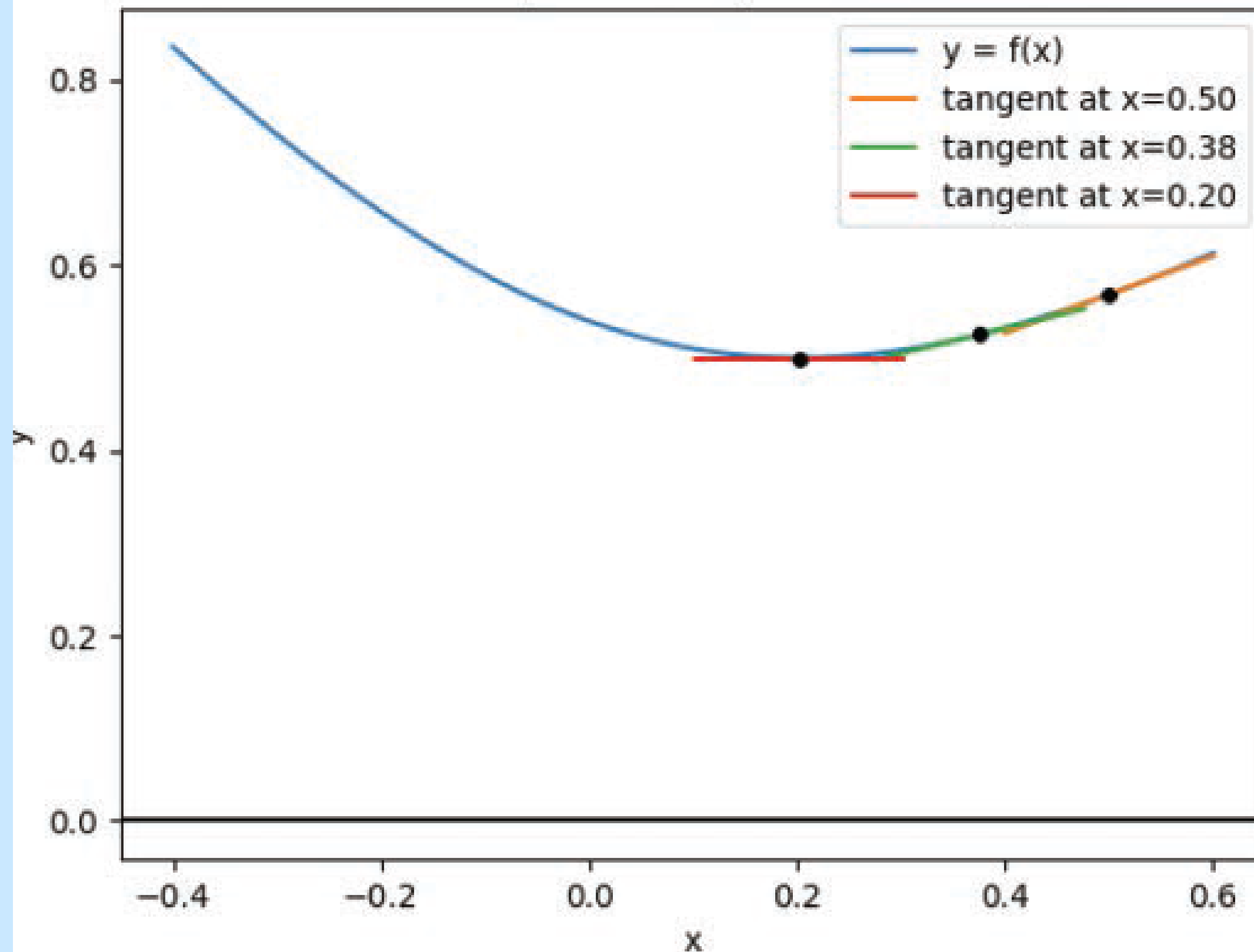
Backpropagation



- **NNs learn by minimizing mistakes**
- **Loss, cost, error or penalty is defined for every mistake**
- **Goal of training is to adjust weights to minimise error so that NN model provides close to actual outputs**
- **The total error for the entire training set, which is usually just the sum of the individual errors should be as low as possible after training NN.**

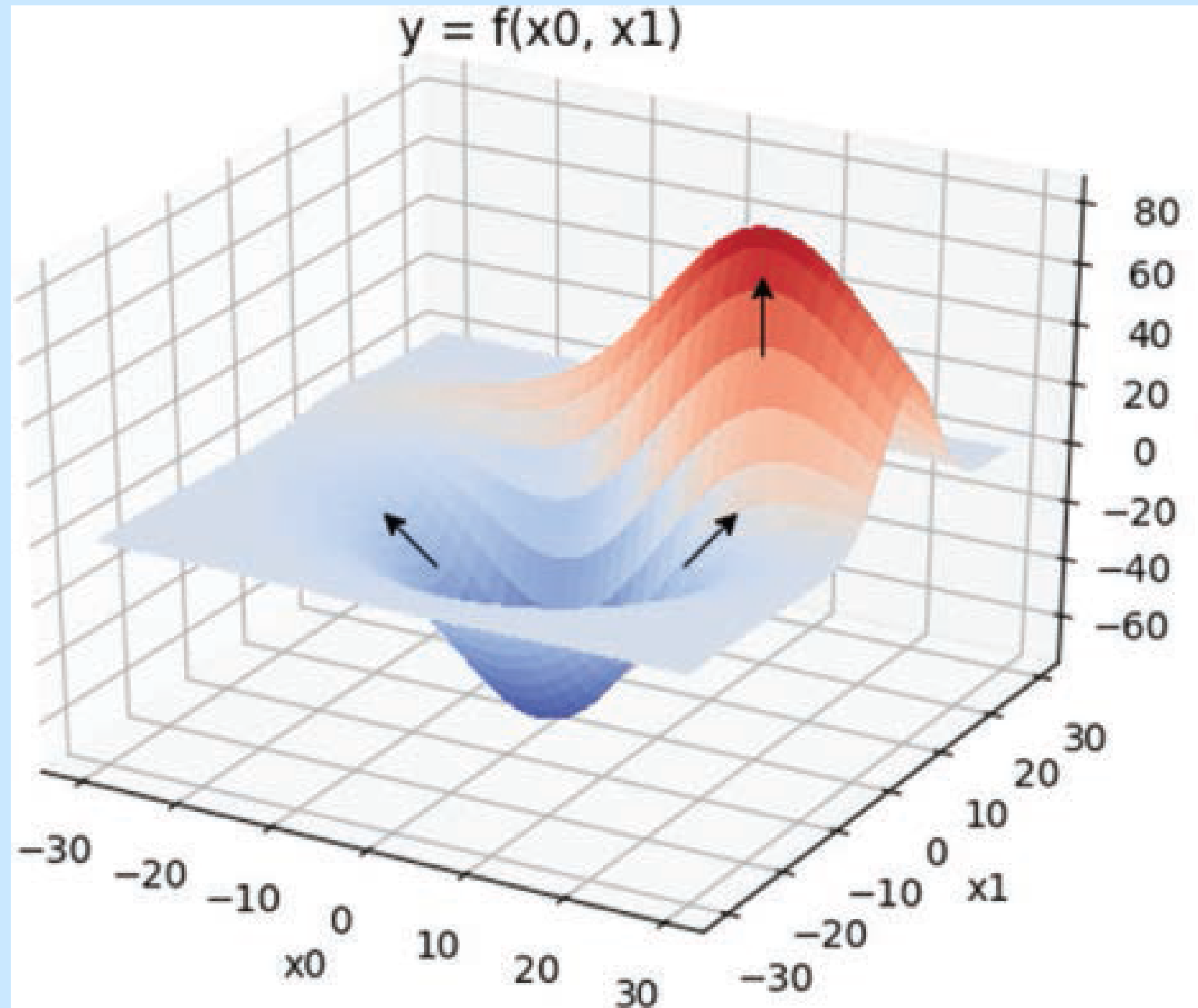


Optimization problem



$$y', f'(x), \frac{dy}{dx}$$

Plot showing a curve $y = f(x)$ and its derivative at the minimum value



**Compute two
partial derivatives**

$$\frac{\partial y}{\partial x_0} \text{ and } \frac{\partial y}{\partial x_1}$$

$$\nabla y = \begin{pmatrix} \frac{\partial y}{\partial x_0} \\ \frac{\partial y}{\partial x_1} \end{pmatrix}$$

Say, $x = (x_1, x_2)$ & $f(x) = f(x_1, x_2)$

Solving Learning Problem with Gradient Descent

Ideally $y(\text{actual}) - y(\text{predicted}) = 0$

$$y - \hat{y} = 0$$

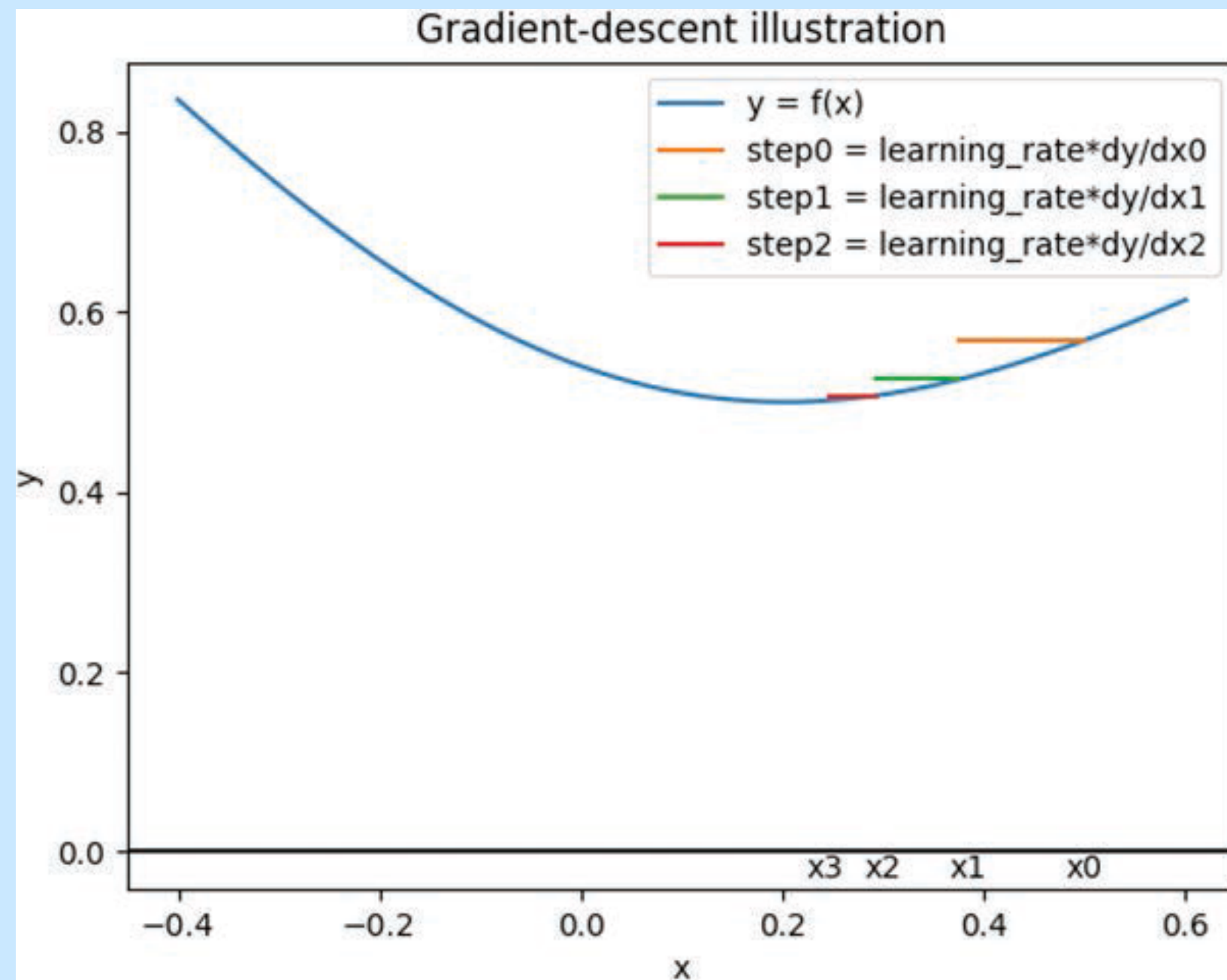
- For multiple samples, use a measure such as mean absolute error (MAE) or mean squared error (MSE).

$$\frac{1}{m} \sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})^2 \quad (\text{mean squared error})$$

- Goal to find weights that minimize the value of the error function

Solving Learning Problem with Gradient Descent

- A closed form solution to find weights that minimize mean squared error (MSE) is challenging.
- A numerical solution is possible through a method: **gradient descent**
- An iterative method starts with an initial guess of the solution and then gradually refine it.



Parameter Update & Learning Rate

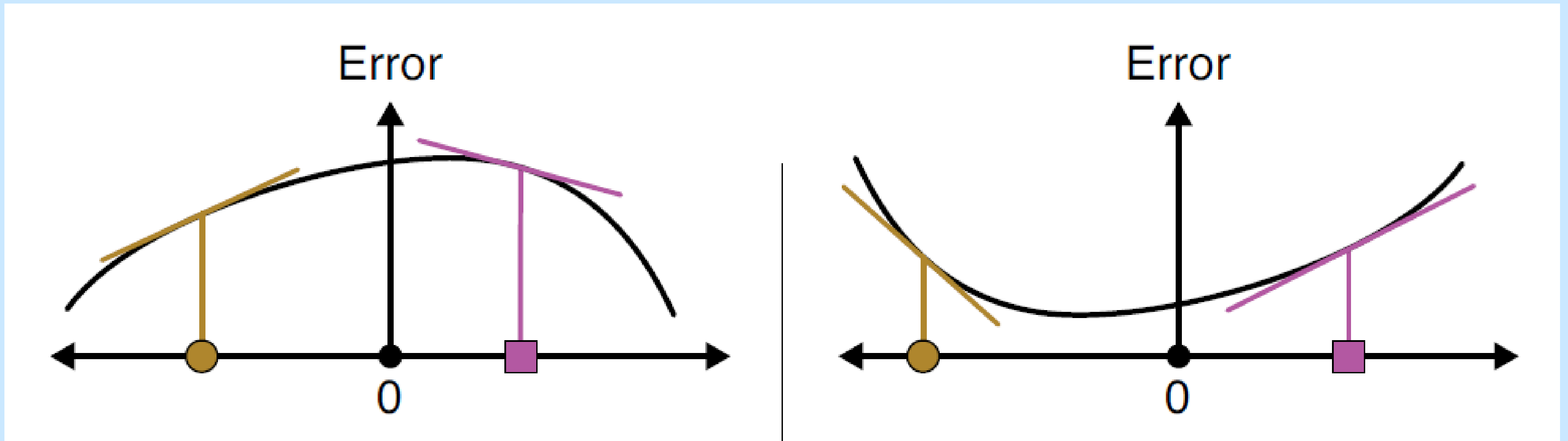
$$x_{n+1} = x_n - \eta f'(x_n)$$

η (Greek letter eta) is a parameter known as the *learning rate*.

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} - \eta \nabla y$$

$$x - \eta \nabla y$$

Parameter Update using Gradient Descent



- When gradient of error is high, move the weight in opposite direction (reduce the weight)
- When gradient of error is low, move the weight in same direction (increase the weight)

Gradient Descent for a Function of Two Variables

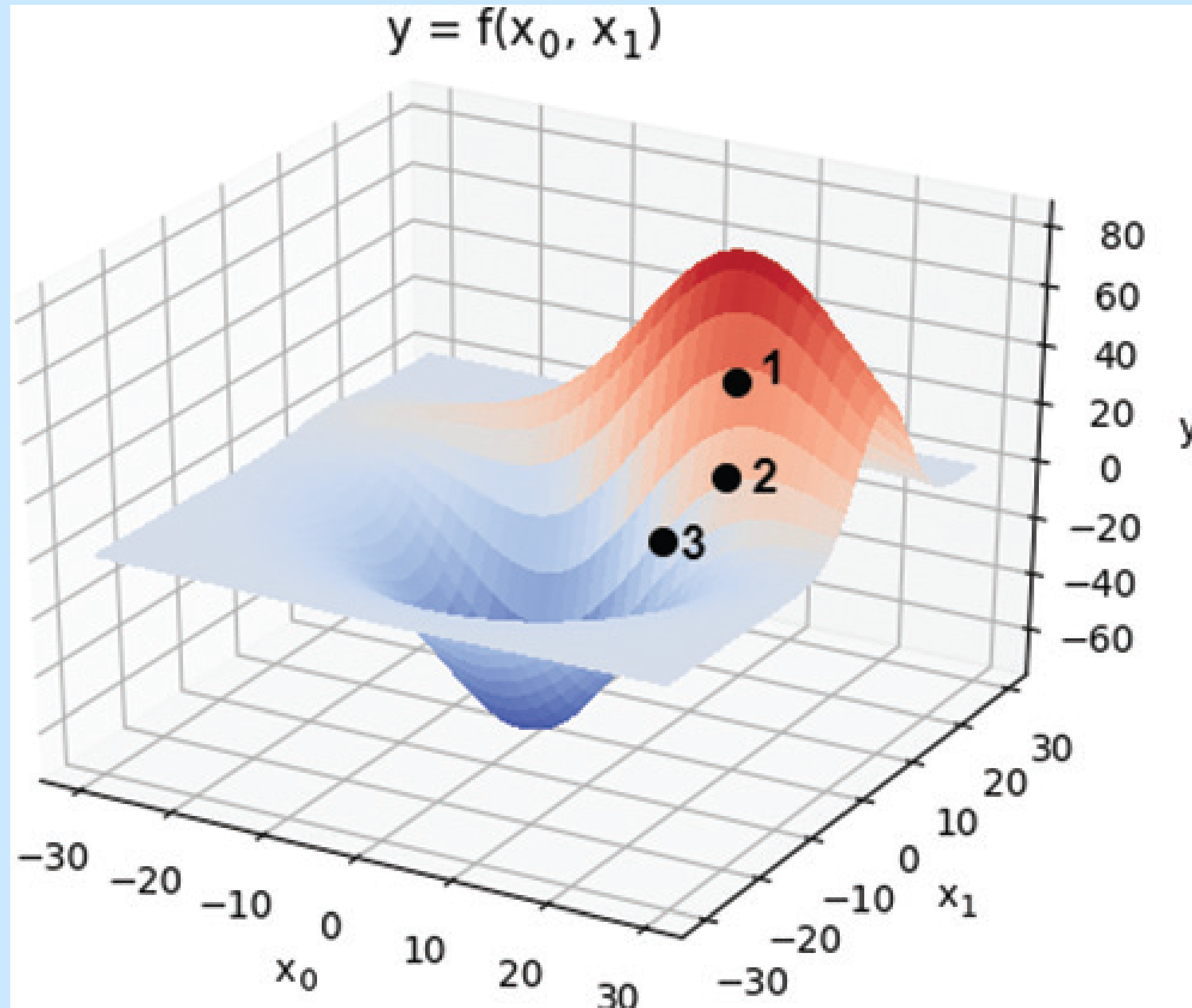


Illustration: Gradient Descent for Perceptron

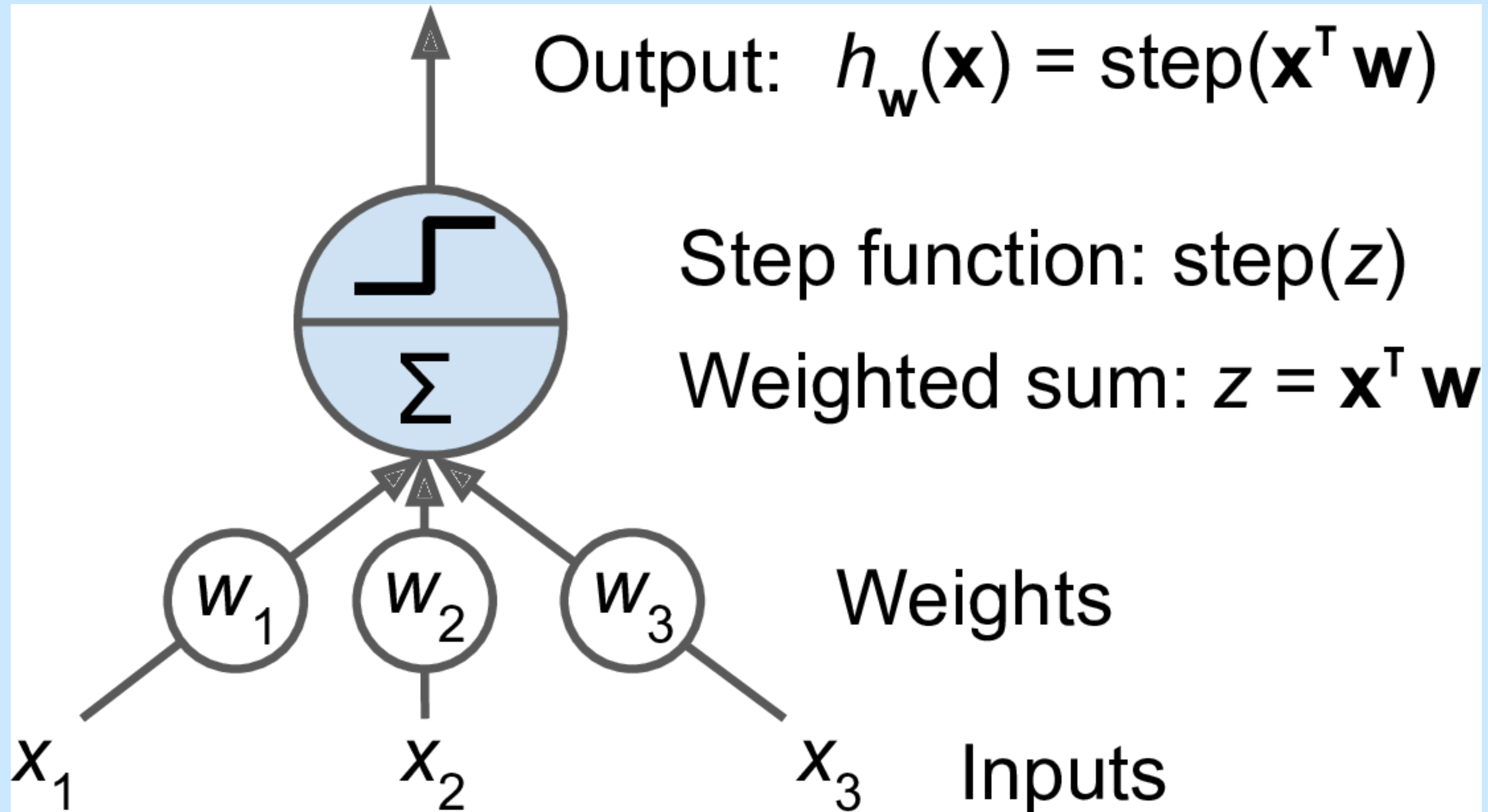


Illustration: Gradient Descent for Perceptron

$$\mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}, \quad y$$

$$Z = x_0 w_0 + x_1 w_1 + x_2 w_2$$

The three partial derivatives with respect to weights

$$\nabla_Z = \begin{pmatrix} \frac{\partial Z}{\partial w_0} \\ \frac{\partial Z}{\partial w_1} \\ \frac{\partial Z}{\partial w_2} \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

Illustration: Gradient Descent for Perceptron

Given the current weight vector \mathbf{w} , and the gradient ∇z , we can now compute a new attempt at \mathbf{w} that will result in a smaller z -value by using gradient decent. Our new \mathbf{w} will be

$$\mathbf{w} - \eta \nabla z$$

which expands to the following for each component of the vector \mathbf{w} :

$$\begin{pmatrix} w_0 - \eta x_0 \\ w_1 - \eta x_1 \\ w_2 - \eta x_2 \end{pmatrix}$$

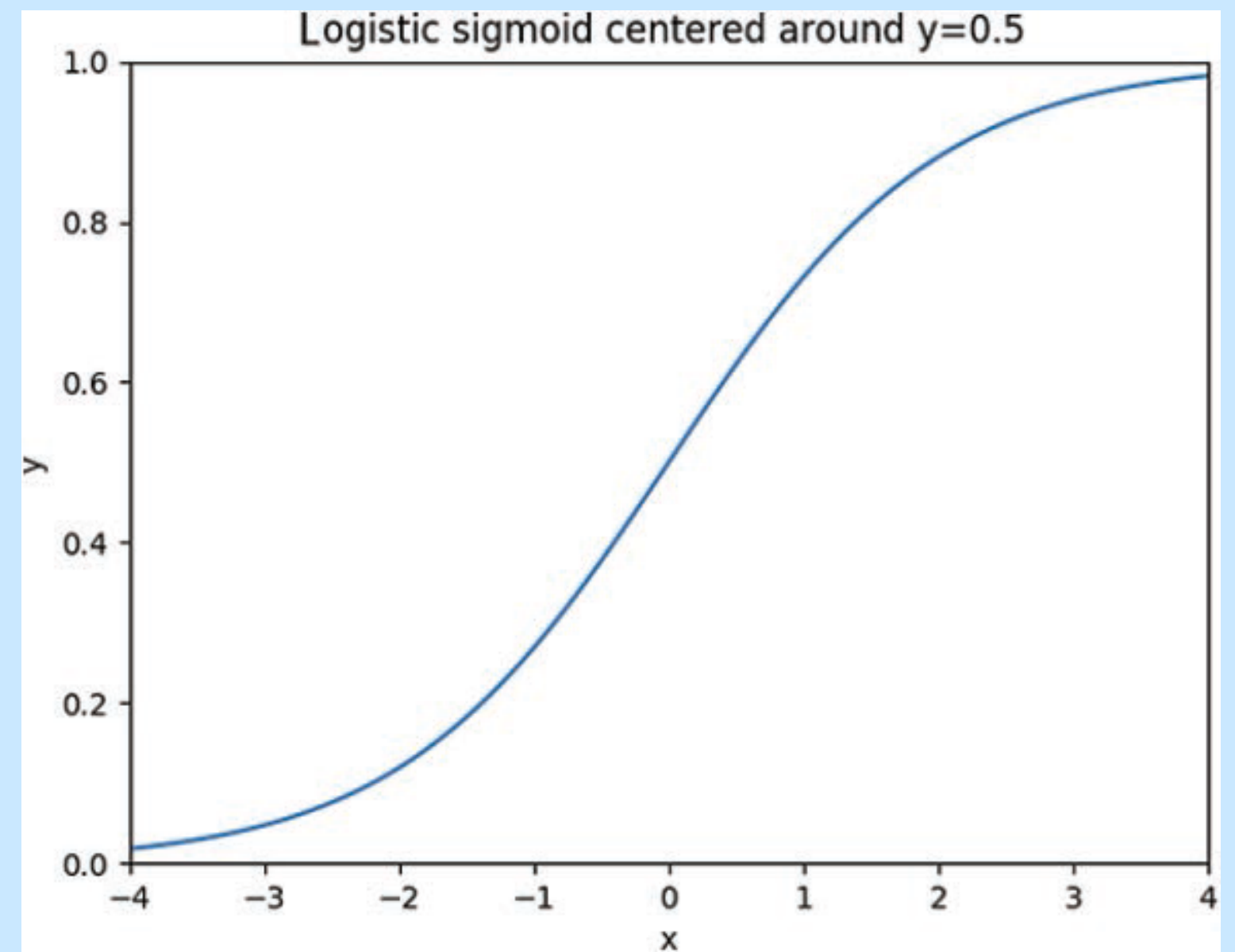
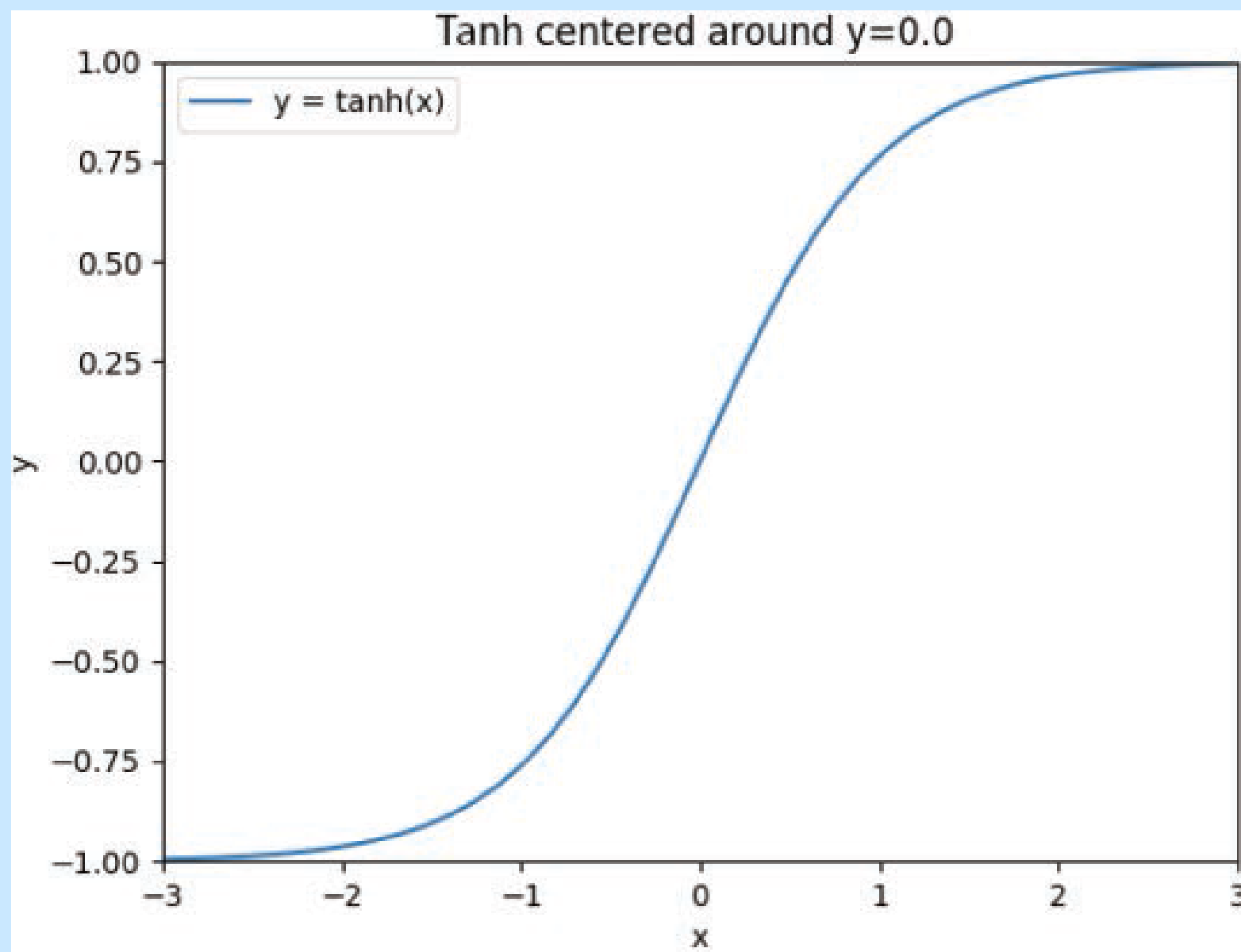
Stochastic gradient descent (SGD)

Gradient descent requires you to compute the gradient for **all input examples** before updating the weights, but **stochastic gradient descent** only requires you to compute the gradient for a **single input example**.

The distinction between stochastic and true gradient descent is that, with true gradient descent, we would compute the gradient as the mean value of the gradients for all individual training examples, whereas with SGd, we approximate the gradient by computing it for only a single training example. There are also hybrid approaches in which you approximate the gradient by computing a mean of some, but not all, training examples bit of body text

Sigmoid Neuron & Backpropagation

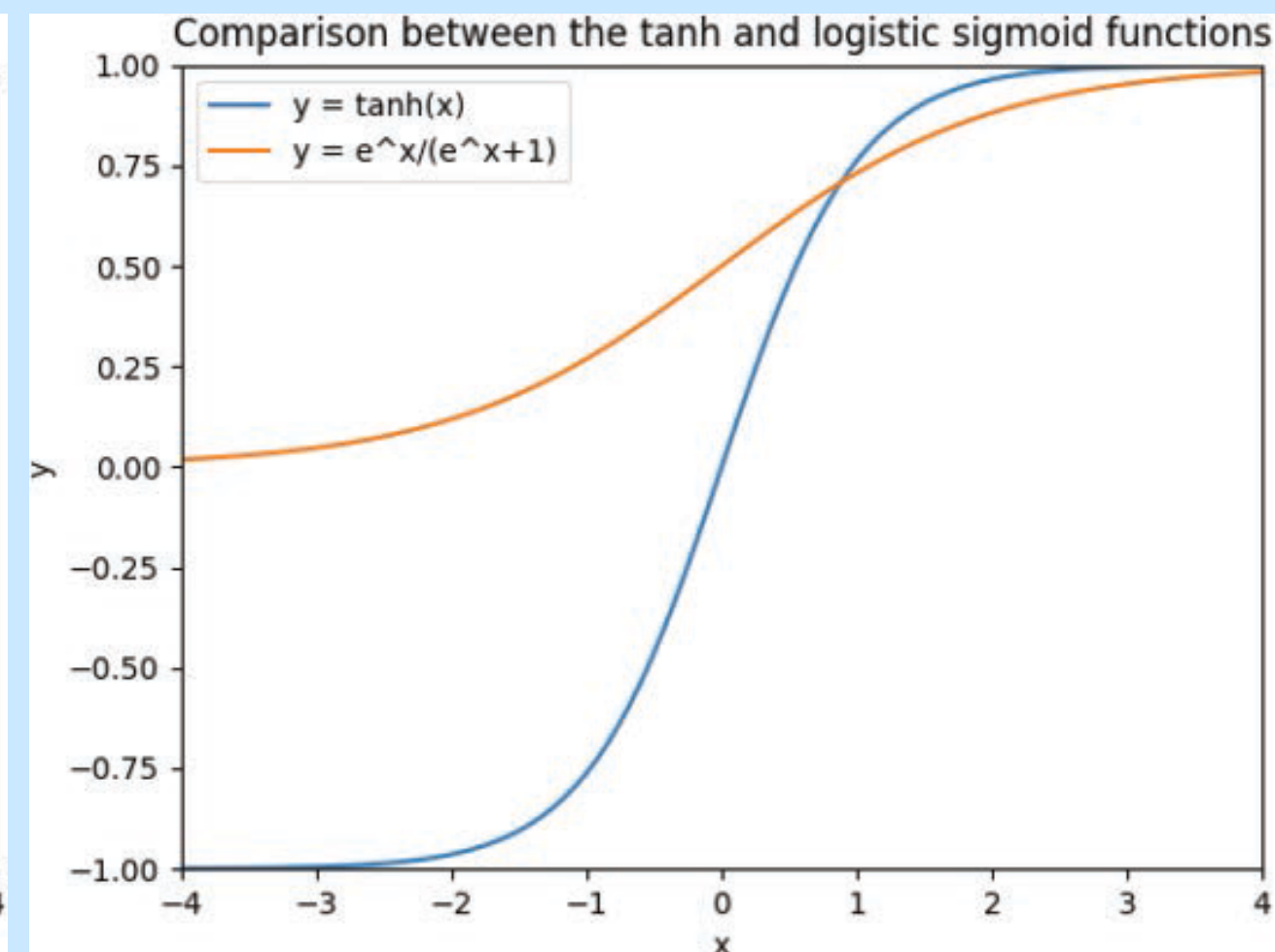
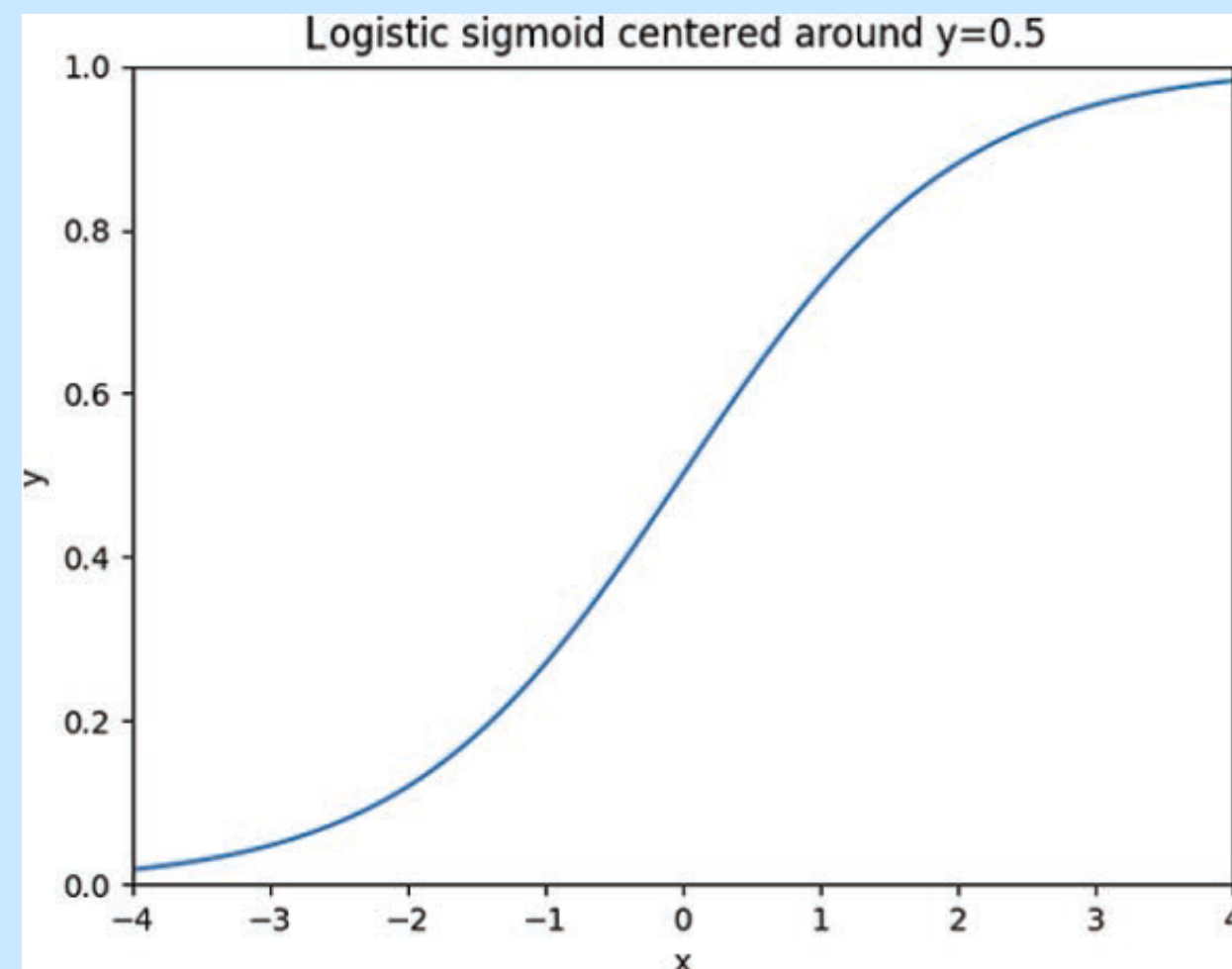
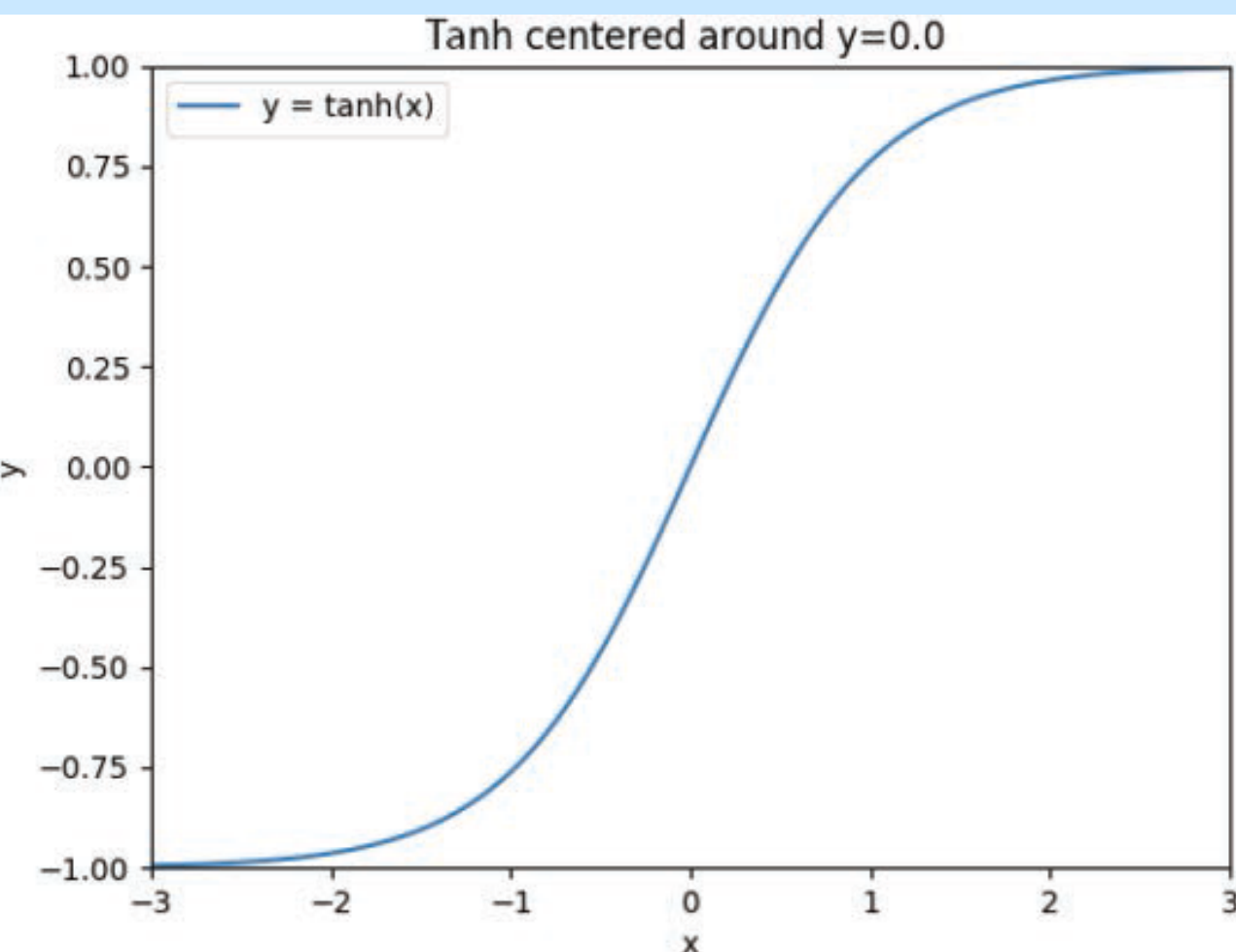
The **backpropagation** algorithm consists of a **forward pass** in which training examples are presented to the network. It is followed by a **backward pass** in which weights are adjusted using **gradient descent**. The gradient is computed using the **backpropagation** algorithm.



Smooth Neurons

Hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$

Logistic sigmoid function: $S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$



Smooth Neurons

Hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$

Logistic sigmoid function: $S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$

Derivative of hyperbolic tangent: $\tanh'(x) = 1 - \tanh^2(x)$

Derivative of logistic sigmoid function: $S'(x) = S(x)(1 - S(x))$

Function Composition & Chain Rule

Assume that we have two functions: $f(x)$, and $g(x)$

Assume that we use the output of function $g(x)$ as an input to function $f(x)$.

Combine them into the composite function $h(x) = f(g(x))$

A common alternative notation is to use the composition operator:

$$h(x) = f \circ g(x), \text{ or just } h = f \circ g$$

Function Composition & Chain Rule

The chain rule states how to compute the derivative of a composition of functions.

$$h = f \circ g$$

then the derivative is

$$h' = (f' \circ g)g'$$

Stated differently, if we have

$$z = f(y) \text{ and } y = g(x), \text{ so } z = f \circ g(x)$$

then

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$$

Using Backpropagation to Compute Gradient

