Calculus

Nipun Batra

August 28, 2023

IIT Gandhinagar

Derivative

Derivative

The derivative of a function of a real variable measures the sensitivity to change of the function value (output value) with respect to a change in its argument (input value).

Derivative

The derivative of a function of a real variable measures the sensitivity to change of the function value (output value) with respect to a change in its argument (input value).

Let us consider a function $f(x) = 3x^2$. The derivative of this function is given by: $f'(x) = 6x$.

Derivative

The derivative of a function of a real variable measures the sensitivity to change of the function value (output value) with respect to a change in its argument (input value).

Let us consider a function $f(x) = 3x^2$. The derivative of this function is given by: $f'(x) = 6x$.

For every unit change in x, the value of $f(x)$ changes by $6x$.

Derivative

JAX

```
import jax
import jax.numpy as np

def f(x):
    return 3 * x ** 2

grad_f = jax.grad(f)

x = 2.0
derivative = grad_f(x)

print("f'(x) =", derivative)
```

Torch

```
import torch

def f(x):
    return 3 * x ** 2

x = torch.tensor(2.0,
    requires_grad=True)
y = f(x)

y.backward()
derivative = x.grad

print("f'(x) =", derivative)
```

Partial Derivative

The partial derivative of a function of several variables is its derivative with respect to one of those variables, with the others held constant (as opposed to the total derivative, in which all variables are allowed to vary).

Let us assume a function $f(x, y) = 2x^2 + 3y$. The partial derivative of this function with respect to x is given by: $\frac{\partial f}{\partial x} = 4x$ and with respect to y is given by: $\frac{\partial f}{\partial y} = 3$.

Partial Derivative

JAX

```
f = lambda x, y: 2 * x ** 2 +
     3 * y

grad_f_x = jax.grad(f,
    argnums=0)
grad_f_y = jax.grad(f,
    argnums=1)

x = 2.0
y = 1.5

derivative_x = grad_f_x(x, y)
derivative_y = grad_f_y(x, y)

print("df/dx =", derivative_x
    )
print("df/dy =", derivative_y
    )
```

Torch

```
f = lambda x, y: 2 * x ** 2 +
     3 * y

x = torch.tensor(2.0,
    requires_grad=True)
y = torch.tensor(1.5,
    requires_grad=True)
z = f(x, y)

z.backward()

derivative_x = x.grad
derivative_y = y.grad

print("df/dx =", derivative_x
    )
print("df/dy =", derivative_y
    )
```

Gradient

The gradient is a multi-variable generalization of the derivative. While a derivative can be defined on functions of a single variable, for functions of several variables, the gradient takes its place. The gradient is a vector-valued function, as opposed to a derivative, which is scalar-valued.

Let us assume a function $f(x, y) = 2x^2 + 3y$. The gradient of this function is given by: $\nabla f = \begin{bmatrix} 4x \\ 3 \end{bmatrix}$.

Gradient

JAX

```python
f = lambda x, y: 2 * x ** 2 + 3 * y

grad_f = jax.grad(f, argnums=[0, 1])

x = 2.0
y = 1.5

gradient = grad_f(x, y)

print("Gradient =", gradient)
```

Torch

```python
f = lambda x, y: 2 * x ** 2 + 3 * y

x = torch.tensor(2.0, requires_grad=True)
y = torch.tensor(1.5, requires_grad=True)

z = f(x, y)
z.backward()

gradient = torch.tensor([x.grad, y.grad])

print("Gradient =", gradient)
tensor([8., 3.])
```

Gradient

Torch (alternative)

```
def f2_vectorized(input):
    x, y = input
    return 2*x**2 + 3*y

input = torch.tensor([2.0, 1.5], requires_grad=True)

# Torch version 1 (using .backward)
z = f2_vectorized(input)
z.backward()
print("\nUsing Method 1 Torch")
print("Gradient: ", input.grad)
```

```
Using Method 1 Torch
Gradient:  tensor([8., 3.])
```

Jacobian

The Jacobian is a matrix that contains the partial derivatives of a vector-valued function with respect to its input variables. For example, let us consider the vector valued function $F(x, y, z) = \begin{bmatrix} x^2 + y^2 \\ y - z \end{bmatrix}$. The Jacobian of this function is given by: $J_F = \begin{bmatrix} 2x & 2y & 0 \\ 0 & 1 & -1 \end{bmatrix}$.

In general Jacobian matrix is given as:

$$J_F = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Jacobian

Torch

```
import torch.autograd.functional as F

# We take the Jacobian of the function f(x, y, z) = [x**2 +
     y**2, y - z]
# The Jacobian analytically is [[2x, 2y, 0], [0, 1, -1]]
def f1(x, y, z):
        return x**2 + y**2
def f2(x, y, z):
        return y - z
def f_vectorized(input):
x, y, z = input
return torch.stack([f1(x, y, z), f2(x, y, z)])
print(F.jacobian(f_vectorized, torch.tensor([2.0, 1.0,
    3.0])))
>>>tensor([[ 4.,  2., -0.],
[ 0.,  1., -1.]])
```

Hessian

The Hessian matrix or Hessian is a square matrix of second-order partial derivatives of a scalar-valued function, or scalar field. It describes the local curvature of a function of many variables.

For example, let us consider the function $f(x, y, z) = x^2 + y^2 + xyz$. The Hessian of this function is given by: $H_f = \begin{bmatrix} 2 & z & y \\ z & 2 & x \\ y & x & 0 \end{bmatrix}$.

In general Hessian matrix is given as:

$$H_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Hessian

Torch

```
import torch.autograd.functional as F

def f(x, y, z):
    return x**2 + y**2 + x * y * z

x = torch.tensor(2.0, requires_grad=True)
y = torch.tensor(1.0, requires_grad=True)
z = torch.tensor(3.0, requires_grad=True)

torch_v1_hessian = torch.tensor(F.hessian(f, (x, y, z)))
print(torch_v1_hessian)
>>>tensor([[2., 3., 1.],
[3., 2., 2.],
[1., 2., 0.]])
```

Hessian

Torch

```
import torch.autograd.functional as F

def f_vectorized(input):
    x, y, z = input
    return x**2 + y**2 + x * y * z

print("Torch Functional method")
print(F.hessian(f_vectorized, torch.tensor([2.0, 1.0, 3.0])
    ))
>>>tensor([[2., 3., 1.],
[3., 2., 2.],
[1., 2., 0.]])
```

Hessian

We can construct the Hessian by taking the Jacobian of the gradient. For example, let us consider the function $f(x, y, z) = x^2 + y^2 + xyz$. The gradient of this function is given by: $\nabla f = \begin{bmatrix} 2x + yz \\ 2y + xz \\ xy \end{bmatrix}$.

We can consider the first element in this vector as ∇f_1, second element as ∇f_2, and so on...

So, the Hessian of this function is given by:

$$H_f = \begin{bmatrix} \frac{\partial \nabla f_1}{\partial x} & \frac{\partial \nabla f_1}{\partial y} & \frac{\partial \nabla f_1}{\partial z} \\ \frac{\partial \nabla f_2}{\partial x} & \frac{\partial \nabla f_2}{\partial y} & \frac{\partial \nabla f_2}{\partial z} \\ \frac{\partial \nabla f_3}{\partial x} & \frac{\partial \nabla f_3}{\partial y} & \frac{\partial \nabla f_3}{\partial z} \end{bmatrix} = \begin{bmatrix} 2 & z & y \\ z & 2 & x \\ y & x & 0 \end{bmatrix}.$$

Examples of Differentiation Terms

Term	**Input Example**	**Output Example**
Jacobian	$\mathrm{f}(x,y) = \begin{bmatrix} 2x+y \\ 3x-2y \end{bmatrix}$	$\mathrm{J} = \begin{bmatrix} 2 & 1 \\ 3 & -2 \end{bmatrix}$
Hessian	$f(x,y) = x^2 + xy + y^2$	$\mathrm{H} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$
Derivative	$f(x) = 3x^2$	$f'(x) = 6x$
Partial Derivative	$f(x,y) = 2x^2 + 3y$	$\frac{\partial f}{\partial x} = 4x$
Gradient	$f(x,y) = x^2 + y^2$	$\nabla f(x,y) = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$

Gradient in the context of machine learning

Let us assume a simple linear regression model: $y = \theta_0 + \theta_1 x$. We can write this model in the form of a vector as: $y = \begin{bmatrix} \theta_0 & \theta_1 \end{bmatrix} \begin{bmatrix} 1 \\ x \end{bmatrix}$.

The loss is given by: $L = \frac{1}{2} \sum_{i=1}^{n} (y_i - \theta_0 - \theta_1 x_i)^2$.

The loss is a scalar and a function of θ_0 and θ_1.

The gradient of the loss is given by:
$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial \theta_0} \\ \frac{\partial L}{\partial \theta_1} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} (y_i - \theta_0 - \theta_1 x_i) \\ \sum_{i=1}^{n} (y_i - \theta_0 - \theta_1 x_i) x_i \end{bmatrix}$. We can now use a first-order method like gradient descent to find the optimal values of θ_0 and θ_1 as per:

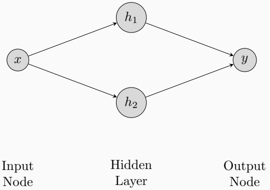
$$\begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \leftarrow \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} - \alpha \nabla L$$

Hessian in the context of machine learning

Instead of using a first-order method like gradient descent, we can use a second-order method like Newton's method to find the optimal values of θ_0 and θ_1.

We can write Hessian H in terms of gradient ∇L as:
$H = \begin{bmatrix} \frac{\partial^2 L}{\partial \theta_0^2} & \frac{\partial^2 L}{\partial \theta_0 \partial \theta_1} \\ \frac{\partial^2 L}{\partial \theta_1 \partial \theta_0} & \frac{\partial^2 L}{\partial \theta_1^2} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} 1 & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 \end{bmatrix}$. Newton's method is given by: $\begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \leftarrow \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial^2 L}{\partial \theta_0^2} & \frac{\partial^2 L}{\partial \theta_0 \partial \theta_1} \\ \frac{\partial^2 L}{\partial \theta_1 \partial \theta_0} & \frac{\partial^2 L}{\partial \theta_1^2} \end{bmatrix}^{-1} \begin{bmatrix} \frac{\partial L}{\partial \theta_0} \\ \frac{\partial L}{\partial \theta_1} \end{bmatrix}$.

Jacobian in the context of machine learning



$h_1 = \mathrm{ReLU}(w_{11}x + b_1)$ $h_2 = \mathrm{ReLU}(w_{12}x+ = b_2)$

Now, let us consider the vector $h = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$.