Vector & Tensor Derivatives

Shashank Srikanth

IIIT Hyderabad

May 19, 2020

Overview

Scalar Derivatives

Given a function $f : \mathbb{R} \to \mathbb{R}$, the derivative of f at a point $x \in \mathbb{R}$ is given as:

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

In the scalar case, the derivative of the function f at the point x tells us how much the function f changes as the input x changes by a small amount ε:

$$f(x+\varepsilon) \approx f(x) + \varepsilon f'(x)$$

Chain rule for scalar derivatives

- The chain rule tells us how to compute the derivative of the composition of functions.
- Suppose that $f, g : \mathbb{R} \to \mathbb{R}$ and $y = f(x)$, $z = g(y)$ then we can also write $z = (g \circ f)(x)$, or draw the following computation graph:

$$x \xrightarrow{f} y \xrightarrow{g} z$$

- The (scalar) chain rule tells us that:

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

Gradient (Vector in - scalar out)

This same intuition carries over into the vector case. Now suppose that f $\mathbb{R}^N \to \mathbb{R}$ takes a vector as input and produces a scalar. The derivative of f at the point $x \in \mathbb{R}^N$ is now called the gradient, and it is defined as:

$$\nabla_x f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{\|h\|}$$

- We can also view the gradient $\frac{\partial y}{\partial x}$ as a vector of partial derivatives:

$$\frac{\partial y}{\partial x} = \left(\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \dots, \frac{\partial y}{\partial x_N} \right)$$

where x_i is the i th coordinate of the vector x, which is a scalar, so each partial derivative $\frac{\partial y}{\partial x_i}$ is also a scalar.

Simplify

- Much of the confusion in taking derivatives involving arrays stems from trying to do too many things at once.
- In order to simplify a given calculation, it is often useful to write out the explicit formula for a single scalar element of the output in terms of nothing but scalar variables.
- Let's try a simple example: Let $\vec{x} \in \mathbb{R}^N$, find $\frac{\partial}{\partial \vec{x}} \left[\vec{x}^T \vec{x} \right]$
- Here $\vec{x} = [x_1, x_2, \dots x_n]^T$ & $\vec{x}^T = [x_1, x_2, \dots x_n]$.
- Then we have:

$$
\begin{aligned}
\vec{x}^T \vec{x} &= [x_1, x_2, \dots x_n]^T [x_1, x_2, \dots x_n] \\
&= x_1^2 + x_2^2 + \cdots + x_n^2 \\
&= \sum_{i=1}^{i=N} x_i^2
\end{aligned}
$$

Simplify

- Let us try to find derivative of $\vec{x}^T\vec{x}$ wrt x_2:

$$\frac{\partial}{\partial x_2}\left[\vec{x}^T\vec{x}\right] = \frac{\partial(x_1^2 + \cdots + x_n^2)}{\partial x_2}$$
$$= 2x_2$$

- Similarly, we have the derivative of $\vec{x}^T\vec{x}$ wrt x_i as $2x_i$.
- Converting these scalar derivatives back into vector form as a vector of partial derivatives, we have:

$$\frac{\partial}{\partial \vec{x}}\left[\vec{x}^T\vec{x}\right] = 2[x_1, x_2, \ldots, x_n]$$
$$= 2\vec{x}$$

- Now, find the derivative of $\frac{\partial \vec{y}}{\partial \vec{x}}$, where $\vec{y} = \sum_{i=1}^{i=N} x_i$

Jacobian (Vector in - Vector out)

Now suppose that $f : \mathbb{R}^N \to \mathbb{R}^M$ takes a vector as input and produces a vector as output. Then the derivative of f at a point x, also called the Jacobian, is the $M \times N$ matrix of partial derivatives. If we again set $y = f(x)$ then we can write:

$$\frac{\partial y}{\partial x} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_M}{\partial x_1} & \cdots & \frac{\partial y_M}{\partial x_N} \end{pmatrix}$$

The Jacobian tells us the relationship between each element of x and each element of y : the (i, j) -th element of $\frac{\partial y}{\partial x}$ is equal to $\frac{\partial y_i}{\partial x_j}$, so it tells us the amount by which y_i will change if x_j is changed by a small amount.

Simplify

- Let $\vec{y} = \vec{W}\vec{x}$, $\vec{W} \in \mathbb{R}^{MxN}$, $\vec{x} \in \mathbb{R}^N$ & $\vec{y} \in \mathbb{R}^M$.
- Now, the Jacobian $\frac{\partial \vec{y}}{\partial \vec{x}}$ is a matrix of dimensions MxN. To find the value of this Jacobian, we enumerate the partial derivatives of all y_i's with x_j's.
- Let's start by computing one of these, say, the 3 rd component of $\vec{y}$ with respect to the 7th component of $\vec{x}$. That is, we want to compute $\frac{\partial \vec{y}_3}{\partial \vec{x}_7}$.

Simplify

$$
\begin{aligned}
\frac{\partial \vec{y}_3}{\partial \vec{x}_7} &= \frac{\partial \left[\sum_{j=1}^{N} W_{3,j}\vec{x}_j\right]}{\partial \vec{x}_7} \\
&= \frac{\partial}{\partial \vec{x}_7}\left[W_{3,1}\vec{x}_1 + W_{3,2}\vec{x}_2 + \ldots + W_{3,7}\vec{x}_7 + \ldots + W_{3,D}\vec{x}_D\right] \\
&= 0 + 0 + \ldots + \frac{\partial}{\partial \vec{x}_7}\left[W_{3,7}\vec{x}_7\right] + \ldots + 0 \\
&= \frac{\partial}{\partial \vec{x}_7}\left[W_{3,7}\vec{x}_7\right] \\
&= W_{3,7}
\end{aligned}
$$

- Similarly derivative of $\vec{y}_i$ wrt $\vec{x}_j$ is given as $\frac{\partial \vec{y}_i}{\partial \vec{x}_j} = W_{i,j}$

Simplify

This means that the matrix of partial derivatives is

$$\begin{bmatrix} \frac{\partial \vec{y}_1}{\partial \vec{x}_1} & \frac{\partial \vec{y}_1}{\partial \vec{x}_2} & \frac{\partial \vec{y}_1}{\partial \vec{x}_3} & \cdots & \frac{\partial \vec{y}_1}{\partial \vec{x}_D} \\ \frac{\partial \vec{y}_2}{\partial \vec{x}_1} & \frac{\partial \vec{y}_2}{\partial \vec{x}_2} & \frac{\partial \vec{y}_2}{\partial \vec{x}_3} & \cdots & \frac{\partial \vec{y}_2}{\partial \vec{x}_D} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \vec{y}_M}{\partial \vec{x}_1} & \frac{\partial \vec{y}_M}{\partial \vec{x}_2} & \frac{\partial \vec{y}_M}{\partial \vec{x}_3} & \cdots & \frac{\partial \vec{y}_M}{\partial \vec{x}_D} \end{bmatrix} = \begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} & \cdots & W_{1,D} \\ W_{2,1} & W_{2,2} & W_{2,3} & \cdots & W_{2,D} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ W_{M,1} & W_{M,2} & W_{M,3} & \cdots & W_{M,D} \end{bmatrix}$$

This, of course, is just W itself.

- Thus, we can state that $\frac{d\vec{y}}{d\vec{x}} = W$
- Now, find the derivative of $\frac{d\vec{y}}{d\vec{x}}$ in a similar manner, where $\vec{y} = \vec{x}\vec{W}$ (With appropriate dimensions).

Generalized Jacobian: Tensor in, Tensor out

- Just as a vector is a one-dimensional list of numbers and a matrix is a two dimensional grid of numbers, a tensor is a D-dimensional grid of numbers
- Many operations in deep learning accept tensors as inputs and produce tensors as outputs. For e.g. an image is usually represented as a three dimensional grid of numbers, where the three dimensions correspond to the height, width, and color channels of the image.

Suppose now that $f : \mathbb{R}^{N_1 \times \cdots \times N_{D_x}} \to \mathbb{R}^{M_1 \times \cdots \times M_{D_y}}$. Then the input to f is a D_x -dimensional tensor of shape $N_1 \times \cdots \times N_{D_x}$, and the output of f is a D_y -dimensional tensor of shape $M_1 \times \cdots \times M_{D_y}$. If $y = f(x)$ then the derivative $\frac{\partial y}{\partial x}$ is a generalized Jacobian, which is an object with shape

$$(M_1 \times \cdots \times M_{D_y}) \times (N_1 \times \cdots \times N_{D_x})$$

Generalized Jacobian: Tensor in, Tensor out

- The dimensions of $\frac{\partial y}{\partial x}$ is separated into two groups:
  - The first group matches the dimensions of y
  - The second group matches the dimensions of x.
- The generalized Jacobian is the generalization of a matrix, where each "row" has the same shape as y and each "column" has the same shape as x
- Now if we let $i \in \mathbb{Z}^{D_y}$ and $j \in \mathbb{Z}^{D_x}$ be vectors of integer indices, then we can write

$$\left(\frac{\partial y}{\partial x}\right)_{i,j} = \frac{\partial y_i}{\partial x_j}$$

  here y_i and x_j are scalars, so the derivative is scalar as well.

Generalized Jacobian: Tensor in, Tensor out (1)

To deal with a 14-dimensional space, visualize a 3-D space and say 'fourteen' to yourself very loudly. Everyone does it.

Geoffrey Hinton

- The generalized matrix-vector multiply follows the same algebraic rules as a traditional matrix-vector multiply:

$$\left(\frac{\partial y}{\partial x}\Delta x\right)_j = \sum_i \left(\frac{\partial y}{\partial x}\right)_{i,j} (\Delta x)_i = \left(\frac{\partial y}{\partial x}\right)_{j,:} \cdot \Delta x$$

  The only difference is that the indicies i and j are not scalars; instead they are vectors of indicies.
- In the equation above, the term $\left(\frac{\partial y}{\partial x}\right)_{j,:}$ is the j th "row" of the generalized matrix $\frac{\partial y}{\partial x}$, which is a tensor with the same shape as x.

Chain Rule for tensors

- The chain rule also looks the same in the case of tensor-valued functions. Suppose that $y = f(x)$ and $z = g(y)$, where x and y have the same shapes as above and z has shape $K_1 \times \cdots \times K_{D_z}$. Now the chain rule looks the same as before:

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

- $\frac{\partial z}{\partial y}$ has shape of $(K_1 \times \cdots \times K_{D_z}) \times \left(M_1 \times \cdots \times M_{D_y}\right)$
- $\frac{\partial y}{\partial z}$ has shape of $\left(M_1 \times \cdots \times M_{D_y}\right) \times (N_1 \times \cdots \times N_{D_x})$
- The product $\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$ is a generalized matrix-matrix multiply with shape $(K_1 \times \cdots \times K_{D_z}) \times (N_1 \times \cdots \times N_{D_x})$.

Derivative of $x^T Ax$

- Let's find the derivative of $y = x^T Ax$ wrt x and A.
- If y is a scalar, what are the possible dimensions of x & A?
- What are the dimensions of the two derivatives?

Derivative of $x^T Ax$

Enumerating the above matrix product, we have:

$$\begin{aligned}
\mathbf{x}^T A\mathbf{x} &= \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \\
&= \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix} \begin{pmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \end{pmatrix}
\end{aligned}$$

From, above we have

$$\begin{aligned}
\mathbf{x}^T A\mathbf{x} = x_1 \left(a_{11}x_1 + a_{12}x_2 + a_{13}x_3\right) + x_2 \left(a_{21}x_1 + a_{22}x_2 + a_{23}x_3\right) \\
+ x_3 \left(a_{31}x_1 + a_{32}x_2 + a_{33}x_3\right)
\end{aligned}$$

Least Squares Solution

- We know that for linear regression, $h_\theta(x) = \theta^T x$. We can write the cost function for linear regression in the form of normal equations

$$
\begin{aligned}
J(\theta) &= \frac{1}{2m}(X\theta - y)^T(X\theta - y) \\
&= \left((X\theta)^T - y^T\right)(X\theta - y) \\
&= (X\theta)^T X\theta - (X\theta)^T y - y^T(X\theta) + y^T y \\
&= \theta^T X^T X\theta - 2(X\theta)^T y + y^T y
\end{aligned}
$$

- Now, taking vector derivate of the cost function, we have:

$$
\frac{\partial J}{\partial \theta} = 2X^T X\theta - 2X^T y = 0
$$

- This implies $\theta = (X^T X)^{-1} X^T y$.
- Find the solution of weighted least squares in a similar manner and the least norm solution (using Lagrangian method).

Least Norm Solution

- Least-norm solution solves the following optimization problem:

$$\begin{aligned} \min \quad & x^T x \\ \text{s.t.} \quad & Ax = y \end{aligned}$$

- We introduce Lagrange multipliers: $L(x, \lambda) = x^T x + \lambda^T (Ax - y)$
- The optimality conditions give us (vector derivatives needed):

$$\nabla_x L = 2x + A^T \lambda = 0, \quad \nabla_\lambda L = Ax - y = 0$$

- From first condition, $x = -A^T \lambda / 2$
- Substitute into second to get $\lambda = -2 \left(AA^T\right)^{-1} y$
- Hence, $x = A^T \left(AA^T\right)^{-1} y$

Backpropagation for Linear Layers

- In the context of neural networks, a layer f is typically a function of (tensor) inputs x and weights w; the (tensor) output of the layer is then $y = f(x, w)$.
- The layer f is typically embedded in some large neural network with scalar loss L.
- During backpropagation, we assume that we are given $\frac{\partial L}{\partial y}$ and our goal is to compute $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial w}$. By the chain rule we know that

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} \qquad \frac{\partial L}{\partial w} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial w}$$

- Therefore one way to proceed would be to form the (generalized) Jacobians $\frac{\partial y}{\partial x}$ and $\frac{\partial y}{\partial w}$ and use (generalized) matrix multiplication to compute $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial w}$

Backpropagation for Linear Layers

Problem with this approach: The Jacobian matrices $\frac{\partial y}{\partial x}$ and $\frac{\partial y}{\partial w}$ are typically far too large to fit in memory.

- Let f be a linear layer; input is minibatch of N vectors, each of dimension D, and produces a minibatch of N vectors, each of dimension M.
- x: $N \times D$, w: $D \times M$, and $y = xw$: $N \times M$.
- The Jacobian $\frac{\partial y}{\partial x}$ then has shape $(N \times M) \times (N \times D)$.
- In a typical neural network: $N = 64$ and $M = D = 4096$; Then $\frac{\partial y}{\partial x}$ consists of $64 \cdot 4096 \cdot 64 \cdot 4096$ scalar values;
- 68 billion numbers; This Jacobian matrix will take $256\mathrm{GB}$ of memory to store.

Backpropagation for Linear Layers

- However, we can derive expressions that compute the product $\frac{\partial y}{\partial x}\frac{\partial L}{\partial y}$ without explicitly forming the Jacobian $\frac{\partial y}{\partial x}$.
- Let's see how this works out for the case of the linear layer $f(x, w) = xw$ Set $N = 1, D = 2, M = 3$. Then we can explicitly write

$$
\begin{aligned}
y = \begin{pmatrix} y_{1,1} & y_{1,2} & y_{1,3} \end{pmatrix} &= xw \\
&= \begin{pmatrix} x_{1,1} & x_{1,2} \end{pmatrix} \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{pmatrix} \\
&= \begin{pmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} \\ x_{1,1}w_{1,2} + x_{1,2}w_{2,2} \\ x_{1,1}w_{1,3} + x_{1,2}w_{2,3} \end{pmatrix}
\end{aligned}
$$

Backpropagation for Linear Layers

- The dimension of $\frac{\partial L}{\partial y}$ is MxN and is given as:

$$\frac{\partial L}{\partial y} = \begin{pmatrix} dy_{1,1} & dy_{1,2} & dy_{1,3} \end{pmatrix}$$

- Our goal now is to derive an expression for $\frac{\partial L}{\partial x}$ in terms of x, w, and $\frac{\partial L}{\partial y}$ without explicitly forming the entire Jacobian $\frac{\partial y}{\partial x}$.
- $\frac{\partial L}{\partial x}$ will have shape $N \times D$. We also have

$$\frac{\partial L}{\partial x} = \left(\frac{\partial L}{\partial x_{1,1}} \quad \frac{\partial L}{\partial x_{1,2}} \right)$$

- From the chain rule we have:

$$\frac{\partial L}{\partial x_{1,1}} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial x_{1,1}}$$
$$\frac{\partial L}{\partial x_{1,2}} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial x_{1,2}}$$

Backpropagation for Linear Layers

- Now, we can also compute the following:

$$\frac{\partial y}{\partial x_{1,1}} = \begin{pmatrix} \frac{\partial y_{1,1}}{\partial x_{1,1}} & \frac{\partial y_{1,2}}{\partial x_{1,1}} & \frac{\partial y_{1,3}}{\partial x_{1,1}} \end{pmatrix} = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \end{pmatrix}$$
$$\frac{\partial y}{\partial x_{1,2}} = \begin{pmatrix} \frac{\partial y_{1,1}}{\partial x_{1,2}} & \frac{\partial y_{1,2}}{\partial x_{1,2}} & \frac{\partial y_{1,3}}{\partial x_{1,2}} \end{pmatrix} = \begin{pmatrix} w_{2,1} & w_{2,2} & w_{2,3} \end{pmatrix}$$

- Using the above and previous equations, we have:

$$\frac{\partial L}{\partial x_{1,1}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial x_{1,1}} = dy_{1,1} w_{1,1} + dy_{1,2} w_{1,2} + dy_{1,3} w_{1,3}$$
$$\frac{\partial L}{\partial x_{1,2}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial x_{1,2}} = dy_{1,1} w_{2,1} + dy_{1,2} w_{2,2} + dy_{1,3} w_{2,3}$$

Backpropagation for Linear Layers

- Using the previous expression, we can derive the final expression for $\frac{\partial L}{\partial x}$:

$$\begin{aligned}\frac{\partial L}{\partial x} &= \left(\frac{\partial L}{\partial x_{1,1}} \frac{\partial L}{\partial x_{1,2}}\right) \\ &= \begin{pmatrix} dy_{1,1}w_{1,1} + dy_{1,2}w_{1,2} + dy_{1,3}w_{1,3} \\ dy_{1,1}w_{2,1} + dy_{1,2}w_{2,2} + dy_{1,3}w_{2,3} \end{pmatrix}^T \\ &= \frac{\partial L}{\partial y} w^T\end{aligned}$$

- Use dimension matching trick to verify the answer.
- Using similar logic, we can derive $\frac{\partial L}{\partial w}$.

Backpropagation for Linear Layers

Backpropagation for Linear Layers

Let $Y = XW$, X: $N \times D$, W: $D \times M$, and $Y = XW$: $N \times M$, then:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W^T$$
$$\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Y}$$

Let $Y = WX$, W: $M \times D$, X: $D \times N$, , and $Y = WX$: $M \times N$, then:

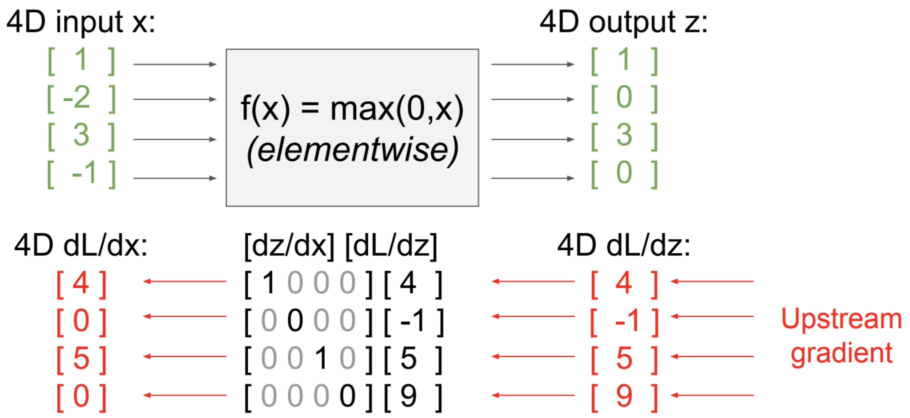
$$\frac{\partial L}{\partial X} = W^T \frac{\partial L}{\partial Y}$$
$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y} X^T$$

Backprop for ReLU

- The gradient of ReLU using backpropagation is given below. Prove it

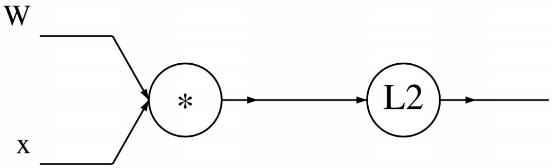
$$\left(\frac{\partial L}{\partial x}\right)_i = \begin{cases} \left(\frac{\partial L}{\partial z}\right)_i & \text{if } x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

Backprop for ReLU



CS231n Problem 1

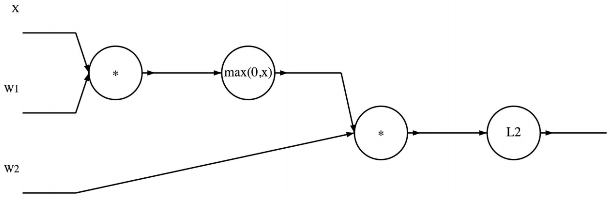
A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^{n} (W \cdot x)_i^2$



CS231n Problem 2

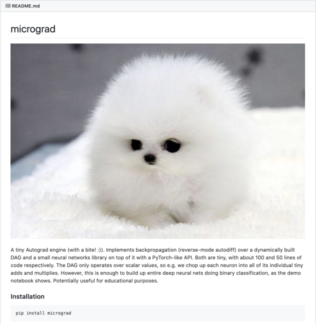
In discussion section: A matrix example...

$$z_1 = XW_1$$
$$h_1 = \mathrm{ReLU}(z_1)$$
$$\hat{y} = h_1 W_2$$
$$L = ||\hat{y}||_2^2$$
$$\frac{\partial L}{\partial W_2} = \textbf{?}$$
$$\frac{\partial L}{\partial W_1} = \textbf{?}$$



Autograd code

- Congratulations on surviving till here. Now, its time to write our own Autograd library and train neural networks using it.
- Let's look at Mircograd code, written by none other than Andrej Karpathy.



Code Usage

```python
from micrograd.engine import Value

a = Value(-4.0)
b = Value(2.0)
c = a + b
d = a * b + b**3
c += c + 1
c += 1 + c + (-a)
d += d * 2 + (b + a).relu()
d += 3 * d + (b - a).relu()
e = c - d
f = e**2
g = f / 2.0
g += 10.0 / f
print(f'{g.data:.4f}') # prints 24.7041, the outcome of this forward pass
g.backward()
print(f'{a.grad:.4f}') # prints 138.8338, i.e. the numerical value of dg/da
print(f'{b.grad:.4f}') # prints 645.5773, i.e. the numerical value of dg/db
```

Tensor object

```
class Value:
    """ stores a single scalar value and its gradient """

    def __init__(self, data, _children=(), _op=''):
        self.data = data
        self.grad = 0
        # internal variables used for autograd graph construction
        self._backward = lambda: None
        self._prev = set(_children)
        self._op = _op # the op that produced this node, for graphviz / debugging / etc
```

Add Function

```
def __add__(self, other):
    other = other if isinstance(other, Value) else Value(other)
    out = Value(self.data + other.data, (self, other), '+')

    def _backward():
        self.grad += out.grad
        other.grad += out.grad
    out._backward = _backward

    return out
```

Multiply function

```
def __mul__(self, other):
    other = other if isinstance(other, Value) else Value(other)
    out = Value(self.data * other.data, (self, other), '*')

    def _backward():
        self.grad += other.data * out.grad
        other.grad += self.data * out.grad
    out._backward = _backward

    return out
```

Power function

```
def __pow__(self, other):
    assert isinstance(other, (int, float)), "only supporting int/float powers for now"
    out = Value(self.data**other, (self,), f'**{other}')

    def _backward():
        self.grad += (other * self.data**(other-1)) * out.grad
    out._backward = _backward

    return out
```

ReLU function

```
def relu(self):
    out = Value(0 if self.data < 0 else self.data, (self,), 'ReLU')

    def _backward():
        self.grad += (out.data > 0) * out.grad
    out._backward = _backward

    return out
```

Backward Function

```
def backward(self):

    # topological order all of the children in the graph
    topo = []
    visited = set()
    def build_topo(v):
        if v not in visited:
            visited.add(v)
            for child in v._prev:
                build_topo(child)
            topo.append(v)
    build_topo(self)

    # go one variable at a time and apply the chain rule to get its gradient
    self.grad = 1
    for v in reversed(topo):
        v._backward()
```

References

[1] Justin Johnson. Derivatives, backpropagation, and vectorization. http://cs231n.stanford.edu/handouts/derivatives.pdf.

[2] Erik Learned-Miller. Vector, matrix, and tensor derivatives. http://cs231n.stanford.edu/vecDerivs.pdf.

[3] Andrej Karpathy et al. Cs231n convolutional neural networks for visual recognition. *Neural networks*, 1:1, 2016.

[4] Andrej Karpathy. micrograd. https://github.com/karpathy/micrograd.