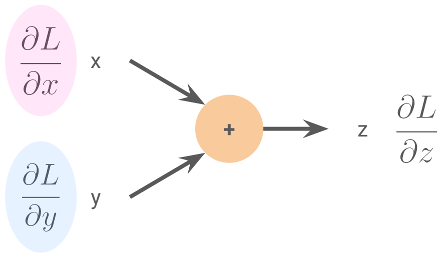
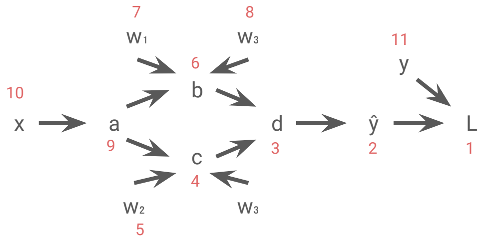
Building a mini autodiff / “autograd” engine

Mateen Ulhaq



SIMON FRASER UNIVERSITY

MASc Eng.
BASc Eng. Phys. Hon. and Math minor



What is autodiff?

- Autodiff (or automatic differentiation) is implemented by the PyTorch engine *autograd* to automatically compute derivatives for you.

- PyTorch builds the graph for you on-the-fly, then finds the derivative during **backpropagation**.

```
loss = (y_target - model(x))**2
loss.backward()         # Compute gradients.
optimizer.step()        # Tell the optimizer the gradients, then step.
optimizer.zero_grad()   # Zero the gradients to start fresh next time.
```

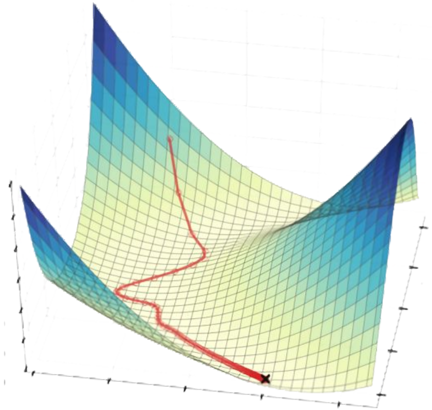
What do we need?

To perform gradient descent, we repeatedly update each weight w_i by the negative gradient scaled by a learning rate η:

$$w_i \leftarrow w_i - \eta \, \partial L / \partial w_i$$

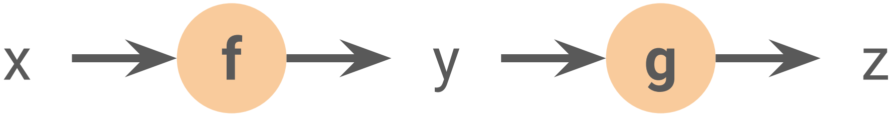
The weights should slowly change so as to minimize L.

Clearly, we need to compute $\partial L / \partial w_i$!



Source:
https://www.hackerearth.com/blog/developers/3-types-gradient-descent-algorithms-small-large-data-sets/

Chain rule

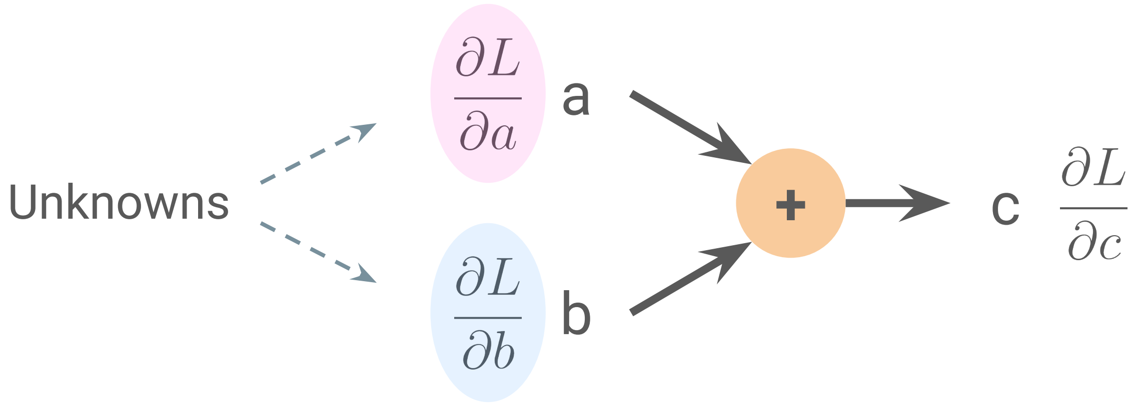


$$y = f(x)$$

$$z = g(y)$$

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

c = a + b



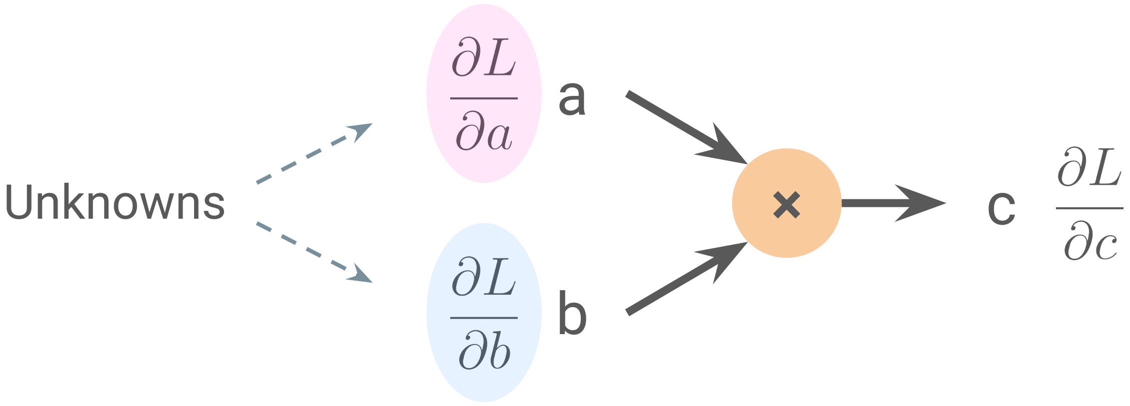
Compute gradients by applying chain rule.

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial c}\frac{\partial c}{\partial a} = \frac{\partial L}{\partial c} \cdot 1$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial c}\frac{\partial c}{\partial b} = \frac{\partial L}{\partial c} \cdot 1$$

⇒ Gradient is “copied” backwards.

c = a × b



$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial c}\frac{\partial c}{\partial a} = \frac{\partial L}{\partial c} \cdot b$$

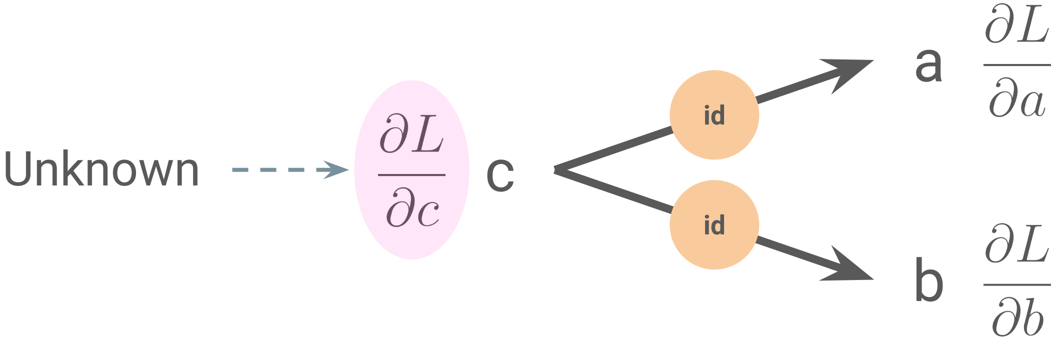
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial c}\frac{\partial c}{\partial b} = \frac{\partial L}{\partial c} \cdot a$$

Compute gradients by applying chain rule.

⇒ Gradient is scaled by the other variable.

a = c

b = c



Compute gradient by applying chain rule.

$$\frac{\partial L}{\partial c} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial c} + \frac{\partial L}{\partial b}\frac{\partial b}{\partial c} = \frac{\partial L}{\partial a} + \frac{\partial L}{\partial b}$$

⇒ Gradient is sum of all gradients of outputs.

Graph generated by

$$a = x^2$$

$$b = (w_1 \cdot a + w_3)^2$$

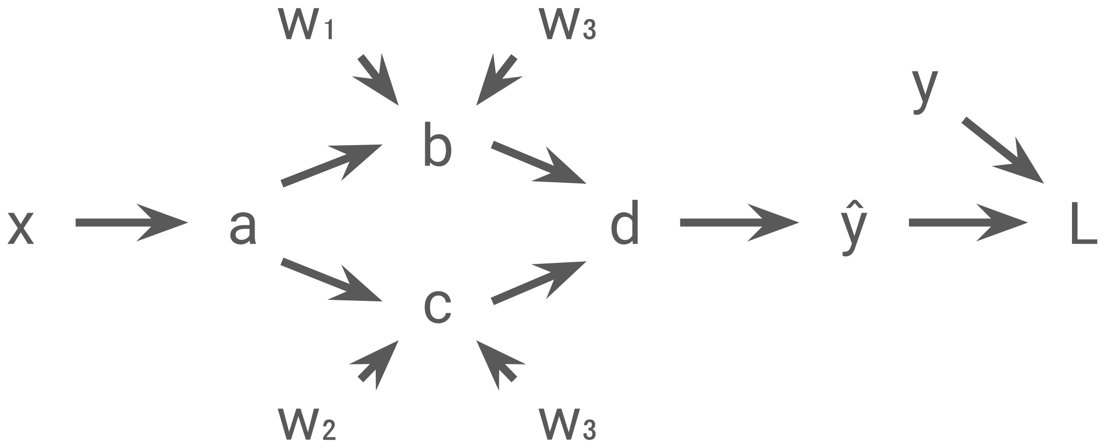
$$c = (w_2 \cdot b + w_3)^2$$

$$d = b + c$$

$$\hat{y} = sin(d)$$

$$L = \sum_i (y_i - \hat{y}_i)^2$$

Operation nodes are omitted for brevity.



To optimize the weights, we need to know

what are $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial w_3}$?

Graph generated by

$$a = x^2$$

$$b = (w_1 \cdot a + w_3)^2$$

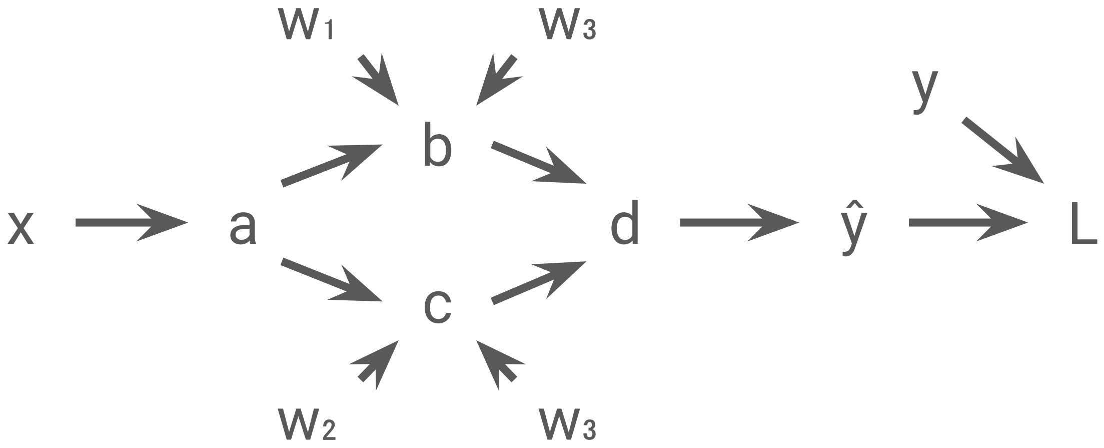
$$c = (w_2 \cdot b + w_3)^2$$

$$d = b + c$$

$$\hat{y} = sin(d)$$

$$L = \sum_i (y_i - \hat{y}_i)^2$$

Operation nodes are omitted for brevity.



Simply use chain rule among all paths, i.e.,
$L \rightarrow \hat{y} \rightarrow d \rightarrow b \rightarrow w_1$ or w_3
$L \rightarrow \hat{y} \rightarrow d \rightarrow c \rightarrow w_2$ or w_3

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial d} \frac{\partial d}{\partial b} \frac{\partial b}{\partial w_1}$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial d} \frac{\partial d}{\partial c} \frac{\partial c}{\partial w_2}$$

For shared weights, incoming gradients are summed.

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial d} \left(\frac{\partial d}{\partial b} \frac{\partial b}{\partial w_3} + \frac{\partial d}{\partial c} \frac{\partial c}{\partial w_3} \right)$$

Code implementation

```python
from __future__ import annotations

from typing import Tuple, Type

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

Nothing important.

Just some imports.

Saves tensors computed in the forward pass that will be needed in the backward pass.

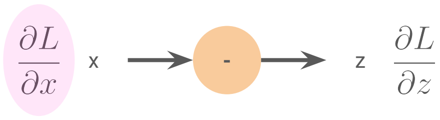
For example, to compute the derivative of $x \cdot y$, we need y.
[Recall: $\partial/\partial x\ (x \cdot y) = y$.]

```
class Context:
    def __init__(self, saved_tensors=()):
        self.saved_tensors = saved_tensors

    def save_for_backward(self, *args):
        self.saved_tensors = args
```

Forward defines the function f(x).

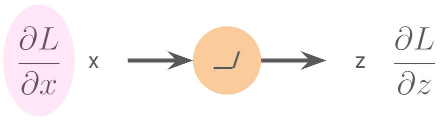
Backward defines the derivative f'(x).

```
class Function:
    @staticmethod
    def forward(ctx: Context, *args: Tensor) -> Tensor:
        raise NotImplementedError

    @staticmethod
    def backward(ctx: Context, *args: Tensor) -> Tuple[Tensor, ...]:
        raise NotImplementedError
```



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot -1$$

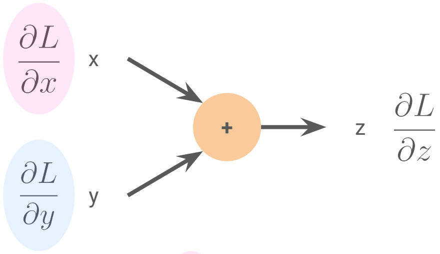
```
class Neg(Function):
    @staticmethod
    def forward(ctx: Context, x: Tensor) -> Tensor:
        return Tensor(-x.data)

    @staticmethod
    def backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, ...]:
        return (-grad_output,)
```



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot H(x)$$

Heaviside
step function

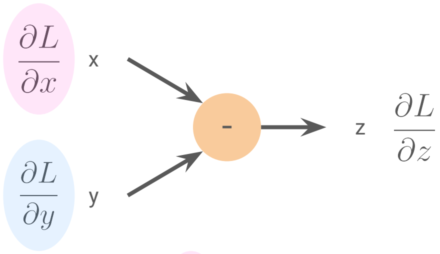
```
class ReLU(Function):
    @staticmethod
    def forward(ctx: Context, x: Tensor) -> Tensor:
        ctx.save_for_backward(x)
        return Tensor(np.maximum(0, x.data))

    @staticmethod
    def backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, ...]:
        (x,) = ctx.saved_tensors
        return (grad_output * (x.data > 0),)
```



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

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

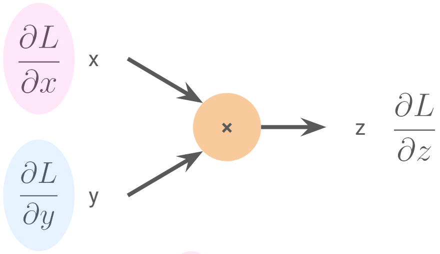
```python
class Add(Function):
    @staticmethod
    def forward(ctx: Context, x: Tensor, y: Tensor) -> Tensor:
        return Tensor(x.data + y.data)

    @staticmethod
    def backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, ...]:
        return grad_output, grad_output
```



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

$$\frac{\partial L}{\partial y} = -\frac{\partial L}{\partial z}$$

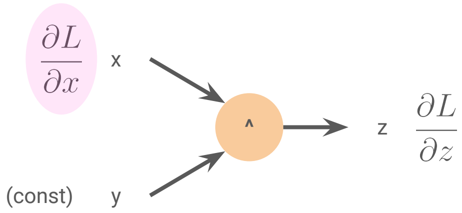
```python
class Sub(Function):
    @staticmethod
    def forward(ctx: Context, x: Tensor, y: Tensor) -> Tensor:
        return Tensor(x.data - y.data)

    @staticmethod
    def backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, ...]:
        return grad_output, -grad_output
```



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

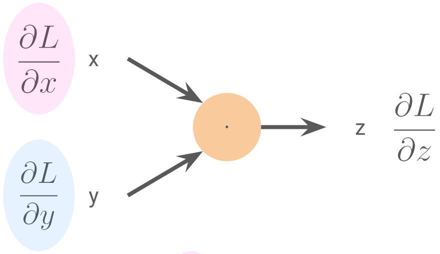
$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \cdot x$$

```python
class Mul(Function):
    @staticmethod
    def forward(ctx: Context, x: Tensor, y: Tensor) -> Tensor:
        ctx.save_for_backward(x, y)
        return Tensor(x.data * y.data)

    @staticmethod
    def backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, ...]:
        x, y = ctx.saved_tensors
        return grad_output * y, grad_output * x
```



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot y \cdot x^{y-1}$$

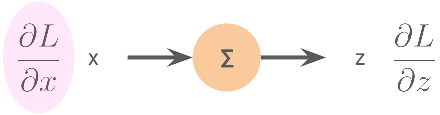
```python
class PowConst(Function):
    @staticmethod
    def forward(ctx: Context, x: Tensor, const: Tensor) -> Tensor:
        ctx.save_for_backward(x, const)
        return Tensor(x.data**const.data)

    @staticmethod
    def backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, ...]:
        x, const = ctx.saved_tensors
        return grad_output * const * x ** (const - 1), None
```



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

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

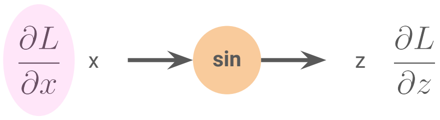
```
class Dot(Function):
    @staticmethod
    def forward(ctx: Context, x: Tensor, y: Tensor) -> Tensor:
        ctx.save_for_backward(x, y)
        return Tensor(x.data.dot(y.data))

    @staticmethod
    def backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, ...]:
        x, y = ctx.saved_tensors
        return grad_output * y, grad_output * x
```



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot (1 + x - x)$$

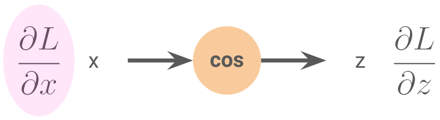
1 with the same shape as x.

```
class Sum(Function):
    @staticmethod
    def forward(ctx: Context, x: Tensor) -> Tensor:
        ctx.save_for_backward(x)
        return Tensor(np.sum(x.data))

    @staticmethod
    def backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, ...]:
        (x,) = ctx.saved_tensors
        return (grad_output * np.ones_like(x.data),)
```



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot \cos x$$

```python
class Sin(Function):
    @staticmethod
    def forward(ctx: Context, x: Tensor) -> Tensor:
        ctx.save_for_backward(x)
        return Tensor(np.sin(x.data))

    @staticmethod
    def backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, ...]:
        (x,) = ctx.saved_tensors
        return (grad_output * x.cos(),)
```



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot -\sin x$$

```python
class Cos(Function):
    @staticmethod
    def forward(ctx: Context, x: Tensor) -> Tensor:
        ctx.save_for_backward(x)
        return Tensor(np.cos(x.data))

    @staticmethod
    def backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, ...]:
        (x,) = ctx.saved_tensors
        return (grad_output * -x.sin(),)
```

Value of current tensor, e.g. z. → data: np.ndarray
Gradient of current tensor, e.g. ∂L/∂z. → grad: Tensor
Creator, e.g. Add (+). → creator: Type[Function]
Parent tensors, e.g. x and y. → parents: Tuple[Tensor, ...]
Saved tensors needed by backward(). → ctx: Context

```
class Tensor:
    data: np.ndarray
    grad: Tensor
    creator: Type[Function]
    parents: Tuple[Tensor, ...]
    ctx: Context

    def __init__(self, data):
        self.data = np.asanyarray(data)
        self.grad = None
        self.creator = None
        self.parents = ()
        self.ctx = None
```

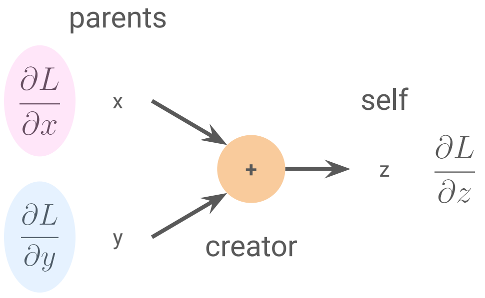
If the tensor is a weight, we use the gradient later to optimize the weights: tensor.data -= tensor.grad * lr

A Tensor tracks which function was used to created it, as well as the inputs to the function.

These are needed when we go backwards.

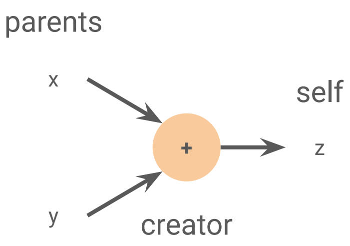
For example,

self = x + y
⇒ creator = Add
parents = [x, y]



Create a new tensor by running the forward method of the creator function (e.g. Add).

Link the output tensor (e.g. z) to the function used, its parent tensors (e.g. x, y), and save any data needed to compute backward (into ctx).




```
class Tensor:

    def _run_forward_op(self, creator: Type[Function], *args) -> Tensor:
        args = [arg if isinstance(arg, Tensor) else Tensor(arg) for arg in args]
        parents = [self, *args]
        ctx = Context()
        tensor = creator.forward(ctx, *parents)
        tensor.creator = creator
        tensor.parents = parents
        tensor.ctx = ctx
        return tensor

    def __neg__(self):
        return self._run_forward_op(Neg)

    def __add__(self, other):
        return self._run_forward_op(Add, other)

    def __sub__(self, other):
        return self._run_forward_op(Sub, other)

    def __mul__(self, other):
        return self._run_forward_op(Mul, other)
```

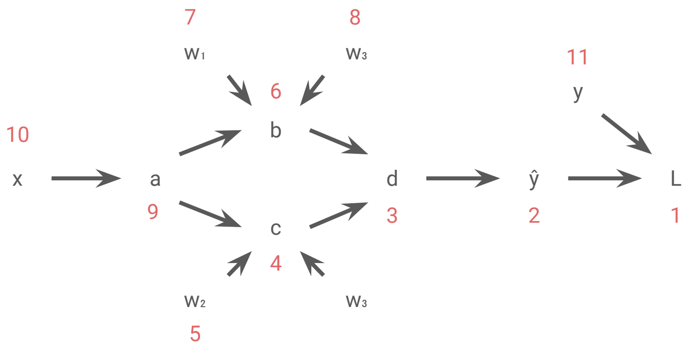
“self + other” gets translated into “self.__add__(other)”

```python
class Tensor:

    def __pow__(self, other):
        return self._run_forward_op(PowConst, other)

    def dot(self, other):
        return self._run_forward_op(Dot, other)

    def sum(self):
        return self._run_forward_op(Sum)

    def sin(self):
        return self._run_forward_op(Sin)

    def cos(self):
        return self._run_forward_op(Cos)

    def relu(self):
        return self._run_forward_op(ReLU)

    ...
```

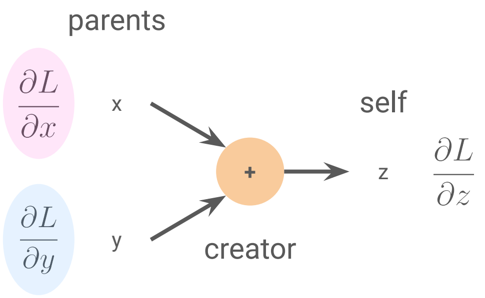
When we try to call backward() on a given tensor, all its input gradients need to be fully computed.

Formally, we seek an ordering $v_1, \dots, v_n$ of the graph such that for all i, the set of all outgoing edges of v_i is a subset of $\{v_1, \dots, v_{i-1}\}$.

The code on the right returns such an ordering.




```python
class Tensor:

  @staticmethod
  def _backwards_tensors(tensor: Tensor):
    """Reversed topological sort for reverse-mode autodiff."""
    visited = set()
    tensors = [ ]

    def dfs(tensor):
      if tensor in visited:
        return
      visited.add(tensor)
      for parent in tensor.parents:
        dfs(parent)
      tensors.append(tensor)

    dfs(tensor)
    return reversed(tensors)
```




```
class Tensor:

    def backward(self):
        for tensor in self._backwards_tensors(self):
            tensor._backward_visit()

    def _backward_visit(self):
        if self.creator is None:
            return

        if self.grad is None:
            self.grad = Tensor(1)

        grad_tensors = self.creator.backward(self.ctx, self.grad)

        for parent, grad_tensor in zip(self.parents, grad_tensors):
            if grad_tensor is None:
                continue
            if parent.grad is None:
                parent.grad = Tensor(grad_tensor.data.copy())
            else:
                parent.grad.data += grad_tensor.data
```

Example usage:

```
loss = (y - y_hat).sum()
loss.backward()
```

Loop over tensors in the graph.

If the current tensor has no creator, then we cannot backpropagate further.

If no self.grad, then assume we are computing gradients w.r.t. self. (e.g. ∂L/∂L = 1.)

Backpropagate through the function to compute gradients for each parent.

Note: This only supports single outputs at the moment.

For each parent, accumulate the resulting gradients.

Note: Copy usually not needed. Faster to clone-on-write (COW).

Fancy printing. Helpful for debugging.

```
>>> print(tensor)
Tensor([1, 2, 3], grad_fn=AddFunction)
```

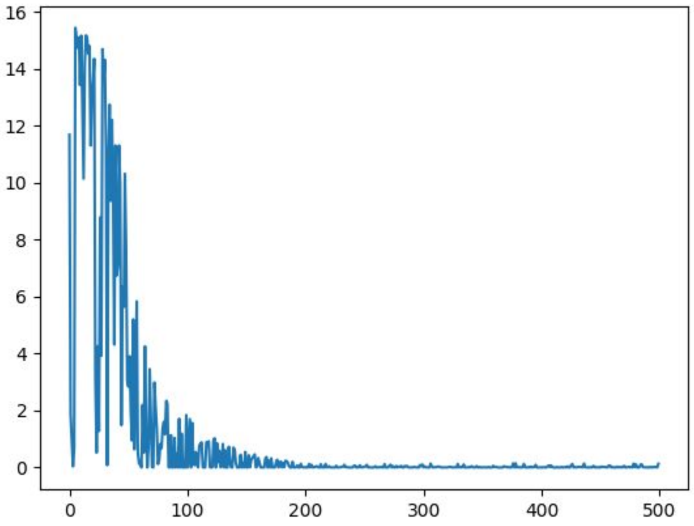
```python
class Tensor:

    def __repr__(self):
        assert isinstance(self.data, np.ndarray)
        data_repr = (
            repr(self.data)
            .removeprefix("array(")
            .removesuffix(")")
            .removesuffix(", dtype=float32")
        )
        grad_fn_repr = self.creator.__name__ if self.creator else None
        return f"Tensor({data_repr}, grad_fn={grad_fn_repr})"
```

Initialize weights.

Just a simple model that outputs in [-4, 4].

```python
class Model:
    def __init__(self):
        self.w1 = Tensor(np.array([[1.7]]))
        self.w2 = Tensor(np.array([[0.2]]))
        self.w3 = Tensor(np.array([[0.6]]))

    def parameters(self):
        return [self.w1, self.w2, self.w3]

    def __call__(self, *args):
        return self.forward(*args)

    def forward(self, x):
        a = x**2
        b = (self.w1 * a + self.w3).relu()
        c = (self.w2 * a + self.w3).relu()
        d = b + c
        y_hat = d.sin() * 4
        return y_hat
```

Loss vs iteration



It works!

```python
def train(lr=1e-3):
    model = Model()
    losses = []

    for i in range(500):
        x = Tensor(np.random.rand(1))
        y = (x**4).sin() * 4
        y_hat = model(x)

        mse_loss = ((y - y_hat) ** 2).sum()
        w_loss = sum(((w**2).sum() for w in model.parameters()), start=Tensor(0))
        loss = mse_loss + w_loss * 0.1
        loss.backward()
        losses.append(mse_loss.data.item())

        for param in model.parameters():
            param.data -= param.grad.data * lr
            param.grad = None

    plt.plot(losses)
    plt.show()


if __name__ == "__main__":
    train()
```

$w_i \leftarrow w_i - \eta\, \partial L/\partial w_i$

“zero_grad”