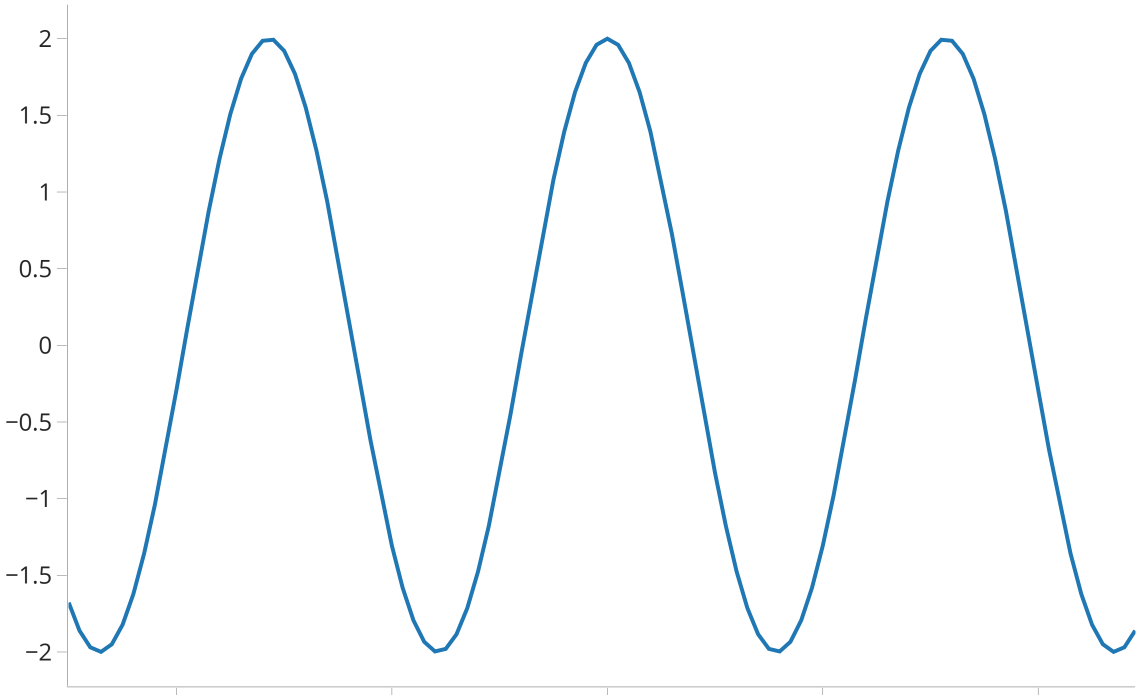
Module 1.2 - Autodifferentiation

Symbolic Derivative

$$f(x) = \sin(2x) \Rightarrow f^{'}(x) = 2\cos(2x)$$

f'(x) = 2*cos(2x)



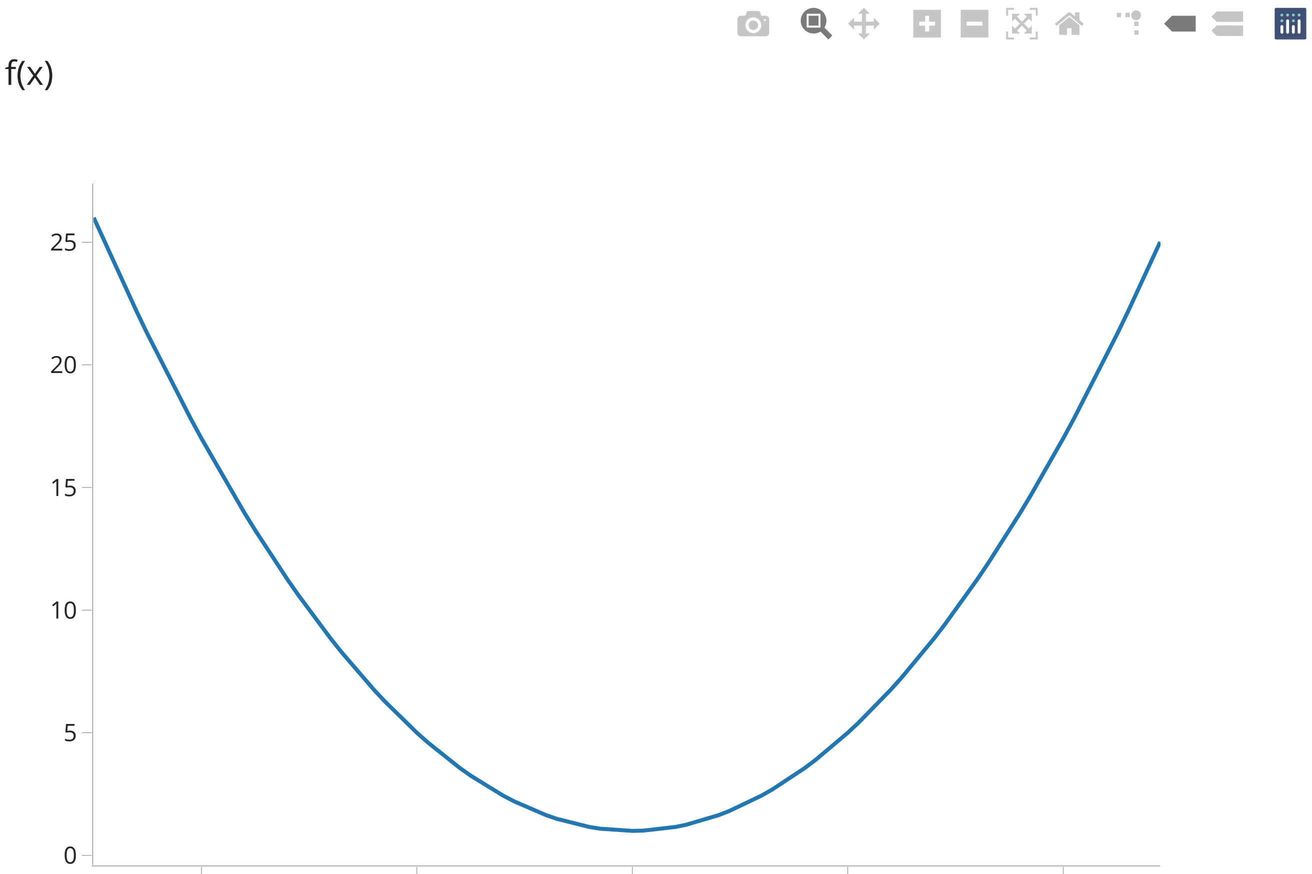
Processing math: 100%

Derivatives with Multiple Arguments

$$f_x'(x, y) = \cos(x) \quad f_y'(x, y) = -2\sin(y)$$

Review: Derivative

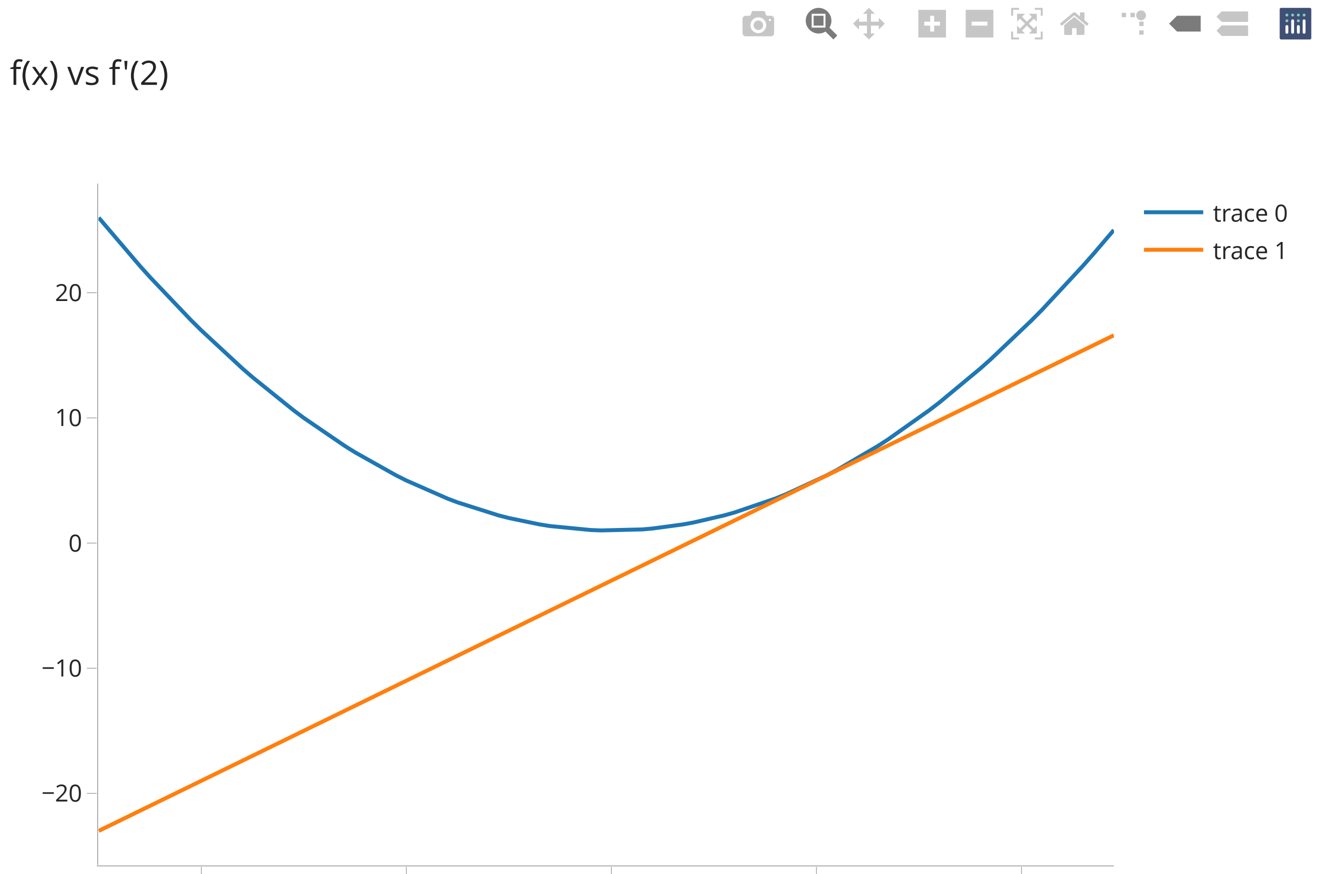
$$f(x) = x^2 + 1$$



Processing math: 100%

Review: Derivative

$$f'(x) = 2x$$

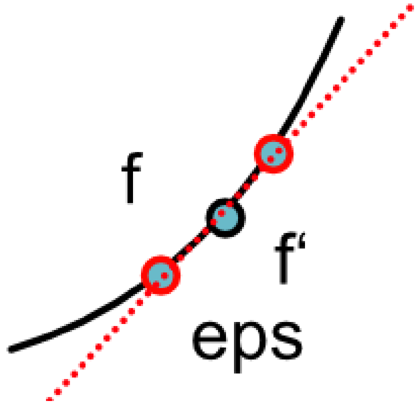


Processing math: 100%

Numerical Derivative: Central Difference

Approximate derivatative

$$f^{'}(x) \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$



Processing math: 100%

Derivative as higher-order function

$$f(x) = \ldots$$

$$f'(x) = \ldots$$

```
def derivative(f: Callable[[float], float]) -> Callable[[float], float]:
    def f_prime(x: float) -> float: ...

    return f_prime
```

Quiz

Processing math: 100%

Outline

- Autodifferentiation
- Computational Graph
- Backward
- Chain Rule

Autodifferentiation

Goal

- Write down arbitrary code
- Transform to compute deriviative
- Use this to fit models

How does this differ?

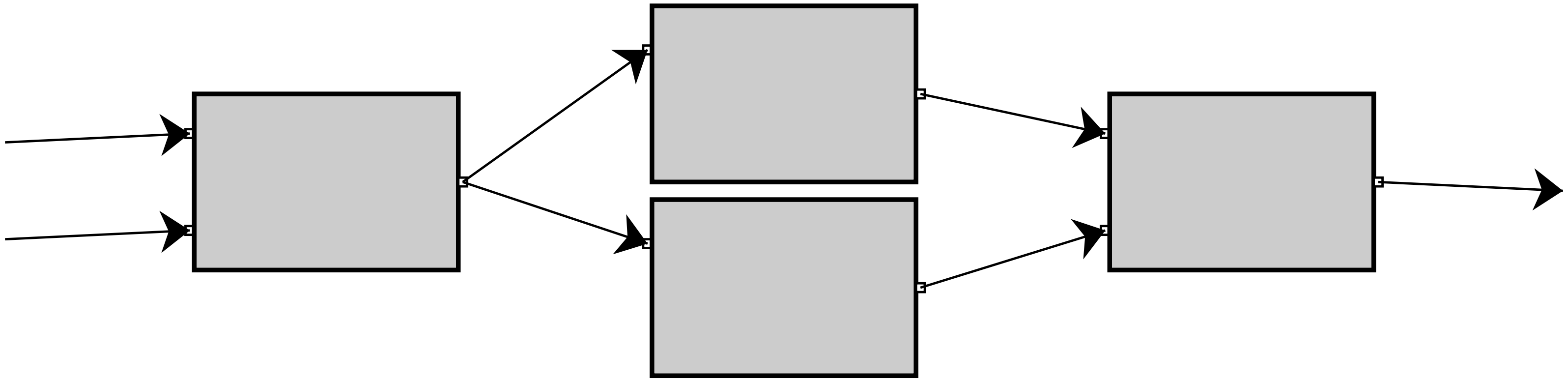
- Are these symbolic derivatives?
    - No, don't get out mathematical form
- Are these numerical derivatives?
    - No, don't use local evaluation.

Overview: Autodifferentiation

- *Forward* Pass - Trace arbitrary function
- *Backward* Pass - Compute derivatives of function

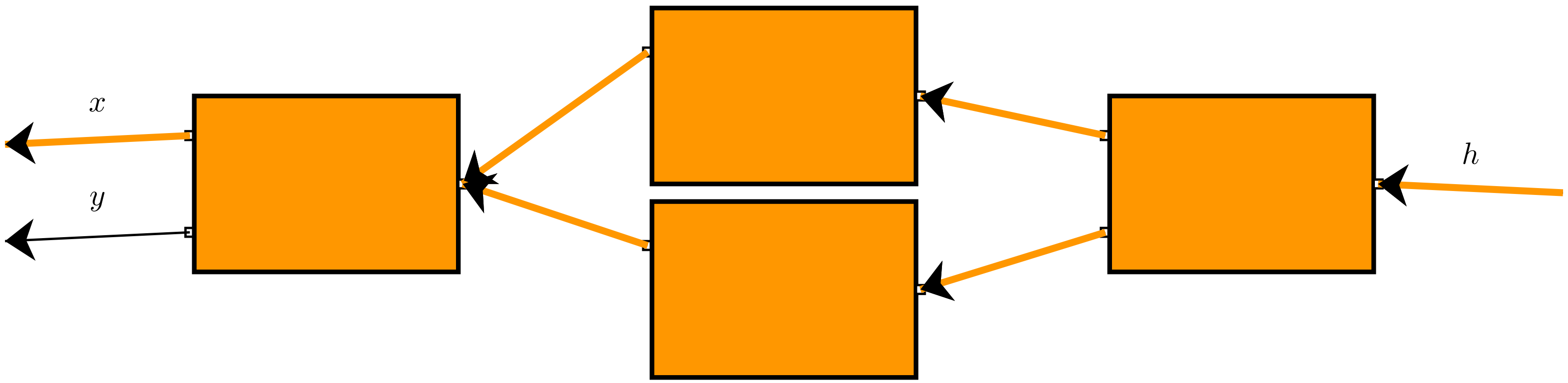
Forward Pass

- User writes mathematical code
- Collect results and computation graph



Backward Pass

- Minitorch uses graph to compute derivative 1, 2,



Example : Linear Model

- Our forward computes

$$\mathrm{L}(w_1, w_2, b) = \mathrm{ReLU}(m(x; w, b))$$

where

$$m(x; w_1, w_2, b) = x_1 \times w_1 + x_2 \times w_2 + b$$

- Our backward computes

$$\mathrm{L}'_{w_1}(w_1, w_2, b) \ \ \mathrm{L}'_{w_2}(w_1, w_2, b) \ \ \mathrm{L}'_{b}(w_1, w_2, b)$$

Processing math: 100%

Derivative Checks

- Property: All three of these should roughly match

Strategy

1. Replace generic numbers.
2. Replace mathematical functions.
3. Track with functions have been applied.

Computation Graph

Strategy

- Act like a numerical value to user
- Trace the operations that are applied
- Hide access to internal storage

Box Diagrams

$$f(x) = \mathrm{ReLU}(x)$$

Box Diagrams

$$f(x, y) = x \times y$$

Code Demo

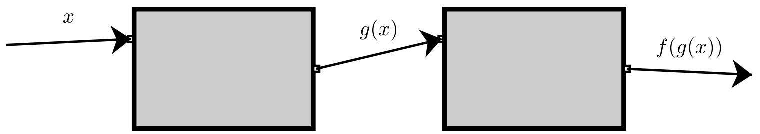
Processing math: 100%

How does this work

- Arrows are intermediate values
- Boxes are function application

$$f(x) = \mathrm{ReLU}(x)$$

$$g(x) = \log(x)$$



Implementation

Functions

- Functions are implemented as static classes
- We implement hidden `forward` and `backward` methods
- User calls `apply` which handles wrapping / unwrapping

Functions

$$f(x) = x \times 5$$


```python
class TimesFive(ScalarFunction):
    @staticmethod
    def forward(ctx: Context, x: float) -> float:
        return x * 5
```

Multi-arg Functions


```
class Mul(ScalarFunction):
    @staticmethod
    def forward(ctx: Context, x: float, y: float) -> float:
        return x * y
```

Processing math: 100%

Variables

- *Wrap* a numerical value

```
x_1 = Scalar(10.0)
x_2 = Scalar(0.0)
```

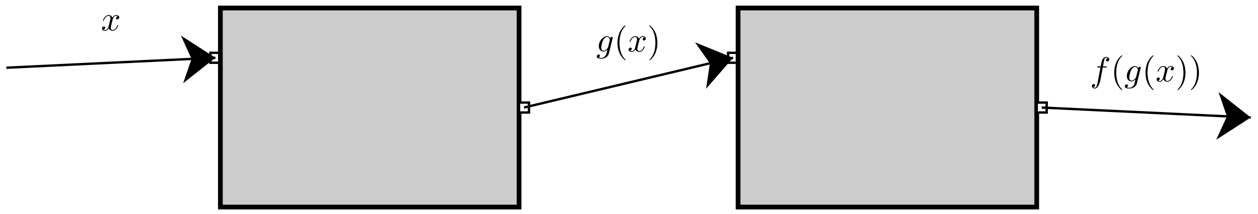
Processing math: 100%

Using scalar variables.

```
x = Scalar(10.0)
z = TimesFive.apply(x)


def apply(cls, val: Scalar) -> Scalar:
    ...
    unwrapped = val.data
    new = cls.forward(unwapped)
    return Scalar(new)
    ...
```

Multiple Steps




```
x = Scalar(10.0)
y = Scalar(5.0)
z = TimesFive.apply(x)
out = TimesFive.apply(z)
```

Tricks

- Use operator overloading to ensure that functions are called

```
out2 = x * y


def __mul__(self, b: Scalar) -> Scalar:
    return Mul.apply(self, b)
```

- Many functions e.g. `sub` can be implemented with other calls.

Notes

- Since each operation creates a new variable, there are no loops.
- Cannot modify a variable. Graph only gets larger.

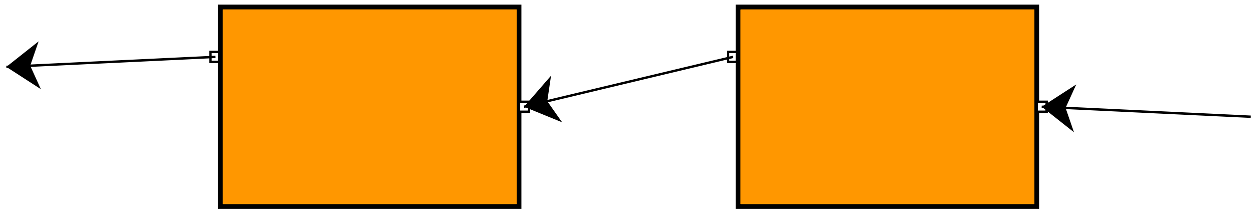
Backwards

How do we get derivatives?

- Base case: compute derivatives for single functions
- Inductive case: define how to propagate a derivative

Base Case: Coding Derivatives

- For each f we need to also provide $f^{'}$
- This part can be done through manual symbolic differentiation



Processing math: 100%

Code

- Backward use $f^{'}$
- Returns $f^{'}(x) \times d$

```
class TimesFive(ScalarFunction):
    @staticmethod
    def forward(ctx, x: float) -> float:
        return x * 5

    @staticmethod
    def backward(ctx, d: float) -> float:
        f_prime = 5
        return f_prime * d
```

Two Arg

- What about $f(x, y)$
- Returns $f_x^{'}(x, y) \times d$ and $f_y^{'}(x, y) \times d$

Code

```python
class AddTimes2(ScalarFunction):
    @staticmethod
    def forward(ctx, x: float, y: float) -> float:
        return x + 2 * y

    @staticmethod
    def backward(ctx, d) -> Tuple[float, float]:
        return d, 2 * d
```

What is Context?

- Context on `forward` is given to `backward`
- May be called at different times.

Context

Consider a function `Square`

- $g(x) = x^2$ that squares x
- Derivative function uses variable $g^{'}(x) = 2 \times x$
- However backward doesn't take args

```
def backward(ctx, d_out): ...
```

Processing math: 100%

Context

Arguments to backward must be saved in context.

```python
class Square(ScalarFunction):
    @staticmethod
    def forward(ctx: Context, x: float) -> float:
        ctx.save_for_backward(x)
        return x * x

    @staticmethod
    def backward(ctx: Context, d_out: float) -> Tuple[float, float]:
        x = ctx.saved_values
        f_prime = 2 * x
        return f_prime * d_out
```

Context Internals

Run `Square`

```
x = minitorch.Scalar(10)
x_2 = Square.apply(x)
x_2.history
```

```
ScalarHistory(last_fn=<class '__main__.Square'>, ctx=Context(no_grad=Fal
se, saved_values=(10.0,)), inputs=[Scalar(10.0)])
```