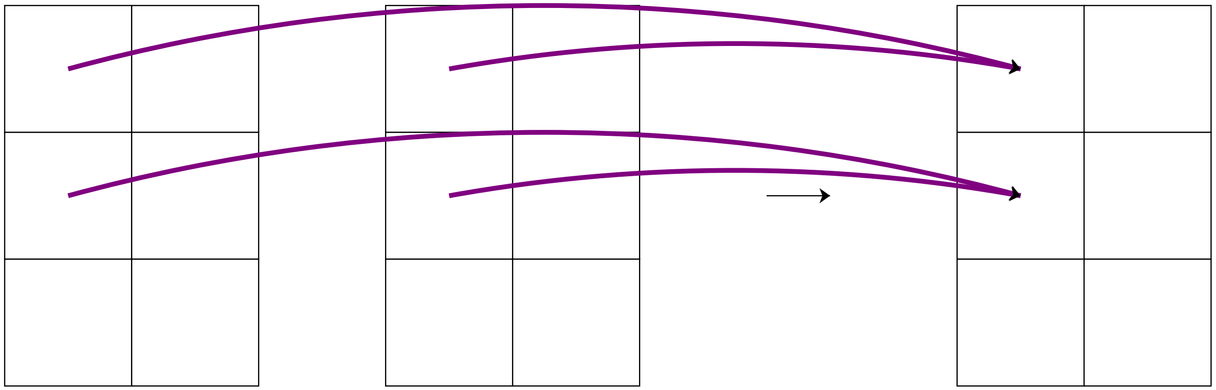
Module 2.4 - Gradients

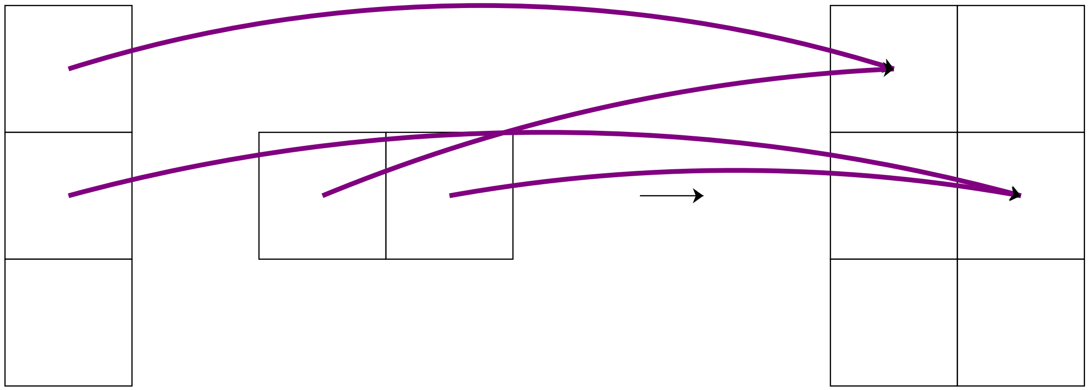
Rules

- **Rule 1**: Dimension of size 1 broadcasts with anything
- **Rule 2**: Extra dimensions of 1 can be added with `view`
- **Rule 3**: Zip automatically adds starting dims of size 1

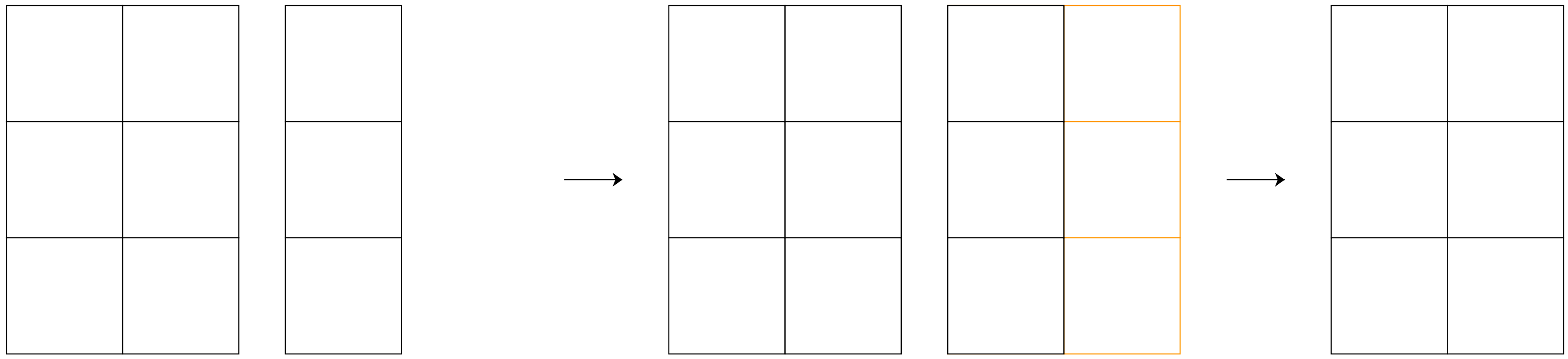
Zip



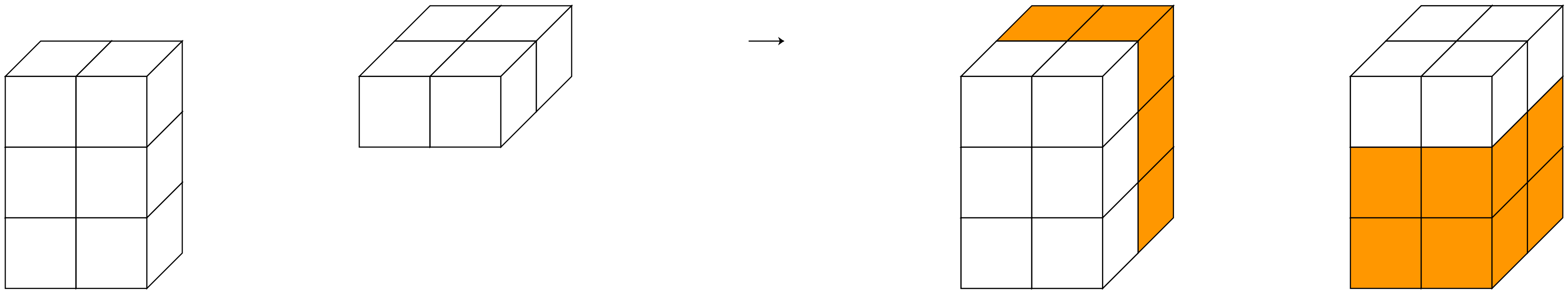
Zip Broadcasting



Matrix-Vector



Example



Quiz

Gradients

Derivatives

- Want to extend derivatives to tensors

- Each tensor function has many different derivatives

Derivatives

- Function with a tensor input is like multiple args
- Function with a tensor output is like multiple functions
- Backward: chain rule from each output to each input.

Terminology

- Scalar -> Tensor
- Derivative -> Gradient
- Recommendation: Reason through gradients as many derivatives

Example

What is backward?

```
x = minitorch.rand((4, 5), requires_grad=True)
y = minitorch.rand((4, 5), requires_grad=True)
z = x * y
z.sum().backward()
```

Notation: Gradient

Function from tensor to a tensor

$$G(x)$$

Example: Product

$$G([x_1, x_2, x_3]) = x_1 x_2 x_3$$

Example: Product

$$G'_{x_1}([x_1, x_2, x_3]) = x_2 x_3$$

$$G'_{x_2}([x_1, x_2, x_3]) = x_1 x_3$$

$$G'_{x_3}([x_1, x_2, x_3]) = x_1 x_2$$

Example: Product Gradient

The gradient is a tensor of derivatives.

$$G'([x_1, x_2, x_3]) = [z/x_1, z/x_2, z/x_3]$$

$$z = x_1 x_2 x_3$$

Original G tensor-to-scalar. Gradient G' tensor-to-tensor.

Example: Product Chain

$$f(G([x_1, x_2, x_3]))$$

$$d = f'(z)$$

$$f'_{x_1}(G([x_1, x_2, x_3])) = x_2 x_3 d$$

$$f'_{x_2}(G([x_1, x_2, x_3])) = x_1 x_3 d$$

$$f'_{x_3}(G([x_1, x_2, x_3])) = x_1 x_2 d$$

Implementation

```
class Prod3(minitorch.Function):
    def forward(ctx, x: Tensor) -> Tensor:
        prod = x[0] * x[1] * x[2]
        ctx.save_for_backward(prod, x)
        return prod

    def backward(ctx, d: Tensor) -> Tensor:
        prod, x = ctx.saved_values
        return d * prod / x
```

Harder Gradients

General Case

What if G returns a tensor?

So far we have only dealt with single values.

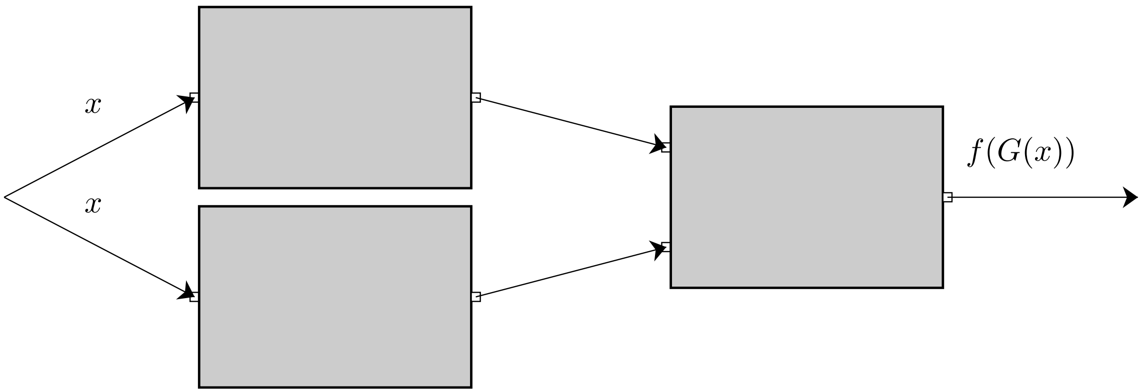
Function to Tensor

Trick: Pretend G is actually many different scalar functions.

$$G(x) = [G^1(x), G^2(x), \ldots, G^N(x)]$$

Example: Chain Rule For Gradients

- $G(x) = [G^1(x), G^2(x)]$ - scalar to tensor
- $f(x)$ - tensor to scalar



Mathematical form: Chain Rule For Gradients

$f(G(x))$

- $z_1 = G^1(x), z_2 = G^2(x), \ldots$
- $d_1 = f'_{z_1}(z), d_2 = f'_{z_2}(z), \ldots$
- $f'_{x_j}(G(x)) = \sum_i d_i G'^i_{x_j}(x)$

Main Change

- There is now one d for each one of G^i
- The d is given as a tensor.

Example: Fun

$$G([x_1, x_2]) = [x_1, x_1 x_2]$$

Example: Fun Derivatives

$$G([x_1, x_2]) = [x_1, x_1 x_2]$$

$$G'^{1}_{x_1}([x_1, x_2]) = 1$$

$$G'^{1}_{x_2}([x_1, x_2]) = 0$$

$$G'^{2}_{x_1}([x_1, x_2]) = x_2$$

$$G'^{2}_{x_2}([x_1, x_2]) = x_1$$

Example: Fun Derivatives

$$f'_x(G(x))$$

$$d_1 = f'(z_1)$$

$$d_2 = f'(z_2)$$

$$f'_{x_1}(G([x_1, x_2])) = d_1 \times 1 + d_2 \times x_2$$

$$f'_{x_2}(G([x_1, x_2])) = d_2 \times x_1$$

Implementation

```
class MyFun(minitorch.Function):
    def forward(ctx, x: Tensor) -> Tensor:
        ctx.save_for_backward(x)
        return minitorch.tensor([x[0], x[0] * x[1]])

    def backward(ctx, d: Tensor) -> Tensor:
        x, = ctx.saved_values
        return minitorch.tensor([d[0] * 1 + d[1] * x[1], d[1] * x[0]])
```

Avoiding Gradients

Avoiding Gradient Math

- All of this is just notation for scalars
- Can often reason about it with scalars directly

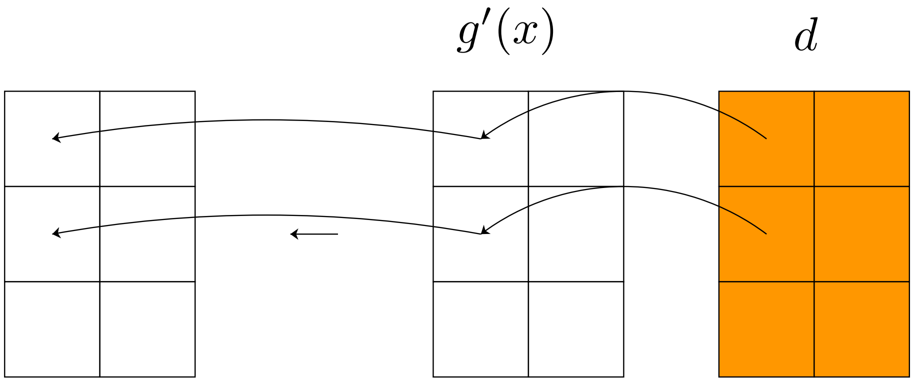
Special Function: Map

$G'^{i}_{x_j}([x_1, \ldots, x_N])$?

Special Function: Map

- ${G'}^{i}_{x_j}(x) = 0$ if $i \neq j$
- $f'_{x_j}(G(x)) = d_i {G'}^{j}_{x_j}(x)$

Map Gradient



Example: Negation

```python
class Neg(minitorch.ScalarFunction):
    @staticmethod
    def forward(ctx, a: float) -> float:
        return -a

    @staticmethod
    def backward(ctx, d: float) -> float:
        return -d
```

Example: Tensor Negation

```python
class Neg(minitorch.Function):
    @staticmethod
    def forward(ctx, t1: Tensor) -> Tensor:
        return t1.f.neg_map(t1)

    @staticmethod
    def backward(ctx, d: Tensor) -> Tensor:
        return d.f.neg_map(d)
```

Example: Inv

```
class Inv(minitorch.Function):
    @staticmethod
    def forward(ctx, t1: Tensor) -> Tensor:
        ctx.save_for_backward(t1)
        return t1.f.inv_map(t1)

    @staticmethod
    def backward(ctx, d: Tensor) -> Tensor:
        (t1,) = ctx.saved_values
        return d.f.inv_back_zip(t1, d)
```

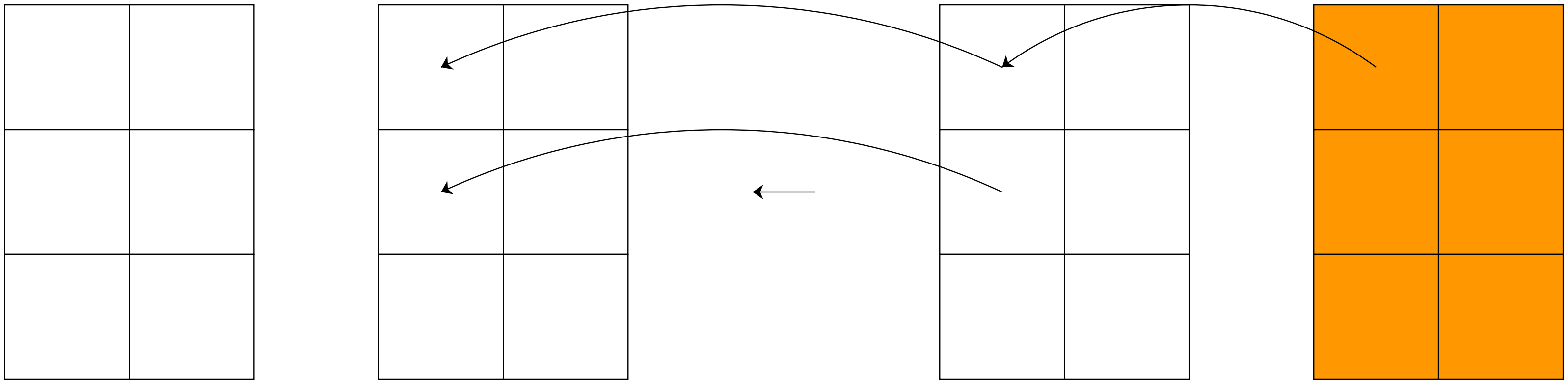
Special Function: Zip

$G'^{i}_{x_j}(x, y)$?

Special Function: Map

- ${G'}^{i}_{x_j}(x) = 0$ if $i \neq j$
- $f'_{x_j}(G(x)) = d_i {g'}^{j}_{x_j}(x, y)$

Zip Gradient

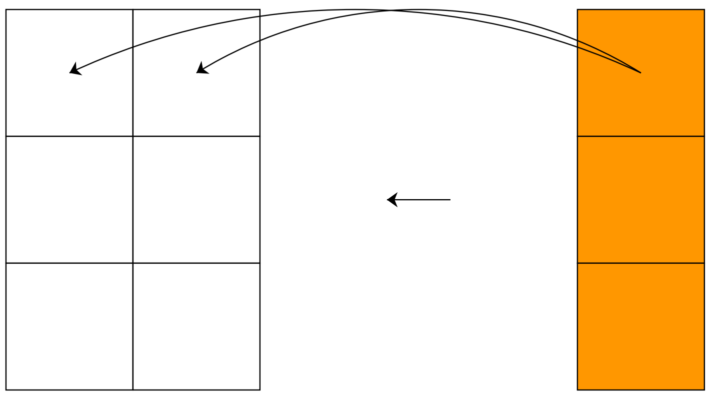


Example: Add

```python
class Add(minitorch.Function):
    @staticmethod
    def forward(ctx, t1: Tensor, t2: Tensor) -> Tensor:
        return t1.f.add_zip(t1, t2)

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

Reduce Gradient



Example: Sum

```python
class Sum(minitorch.Function):
    @staticmethod
    def forward(ctx, a: Tensor, dim: Tensor) -> Tensor:
        ctx.save_for_backward(a.shape, dim)
        return a.f.add_reduce(a, int(dim.item()))

    @staticmethod
    def backward(ctx, grad_output: Tensor) -> Tuple[Tensor, float]:
        a_shape, dim = ctx.saved_values
        return grad_output, 0.0
```

Q&A