Module 3.0 - Real Neural Networks

Review: Chain Rule

$f(G(x))$

- $z_1 = g^1(x), z_2 = g^2(x)$
- $d_1 = f'_{z_1}(z_1, z_2), d_2 = f'_{z_2}(z_1, z_2)$
- $f'_x(G(x)) = d_1 g_x^{'1}(x) + d_2 g_x^{'2}(x)$

Review: Chain Rule

$f(G(x))$

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

Tensor Functions

Think of it as many functions with many arguments

$$G(x) = [G^1(x_1, \ldots), G^2(x_1, \ldots), \ldots, G^N(x_1, \ldots)]$$

Derivative

Derivative of i'th output wrt $j'th$ input

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

Full Chain Rule For Gradients

$f(G(x))$

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

Backward Function

Backward function needs to compute:

- d_i - tensor
- $G'^i_{x_j}$ change in i

$\sum_i d_i G'^i_{x_j}(x)$

Special Function: Map

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

Implies:

- $f'_{x_i}(G(x)) = d_i G'^{i}_{x_i}(x)$

Map Gradient

Loading [MathJax]/extensions/Safe.js

Example: Tensor Negation

- $G^i(x) = -x_i$
- $G'^i_{x_i}(x) = -1$
- $f'_{x_i}(G(x)) = -d_i$

Example: Tensor Negation

```
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: Tensor Inversion

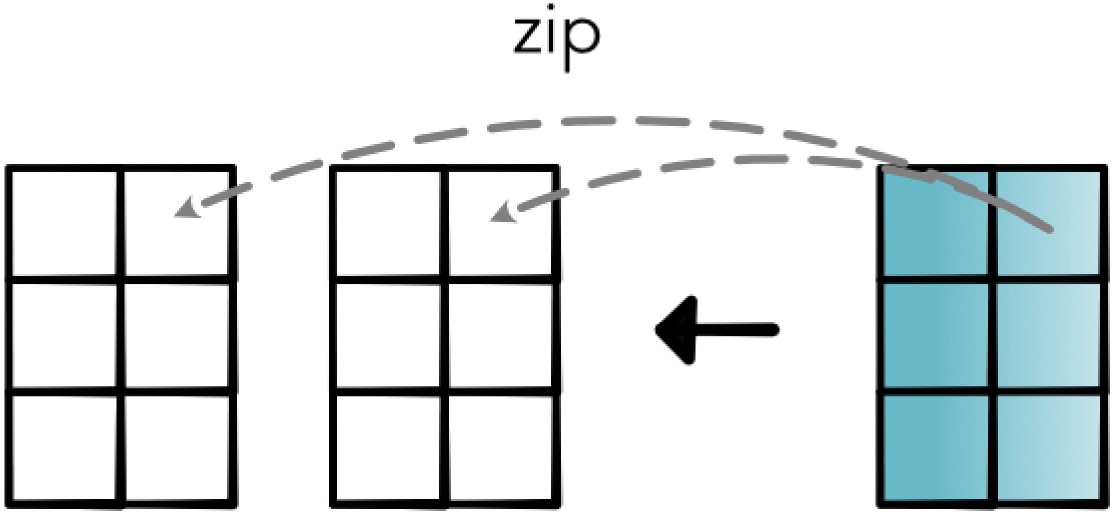
- $G^i(x) = 1/x_i$
- $G'^i_{x_i}(x) = -(x_i)^{-2}$
- $f'_{x_i}(G(x)) = -(x_i)^{-2} * d_i$

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)
```

Zip Gradient



Example: Tensor Inversion

- $G^i(x, y) = x_i + y_i$
- $G'^i_{x_i}(x, y) = 1$
- $f'_{x_i}(G(x)) = d_i$

Example: Add

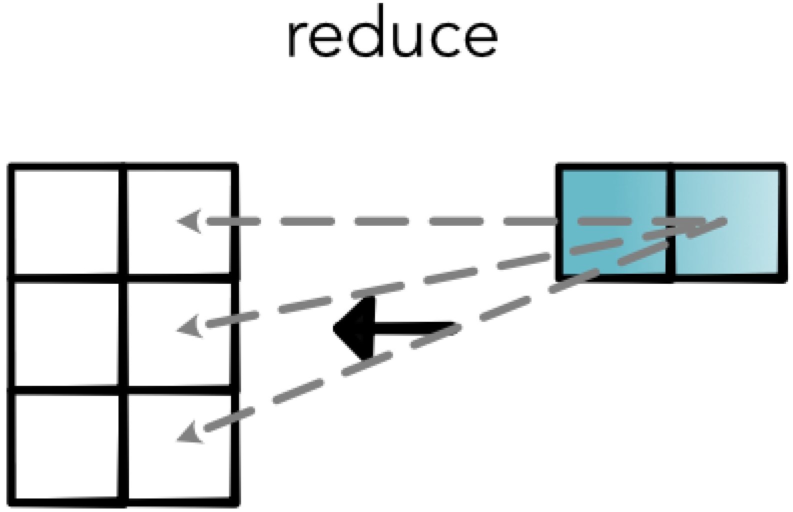
```python
class Add(minitorch.Function):
    @staticmethod
    def forward(ctx, t1: Tensor, t2: Tensor) -> Tensor:
        return t1.f.add_zip(t1, t2)

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

Example: Tensor Inversion

- $G(x) = \sum_i x_i$
- $G'_{x_i}(x) = 1$
- $f'_{x_i}(G(x)) = d$

Reduce Gradient



Quiz

Outline

- Training
- Simple NLP

Training

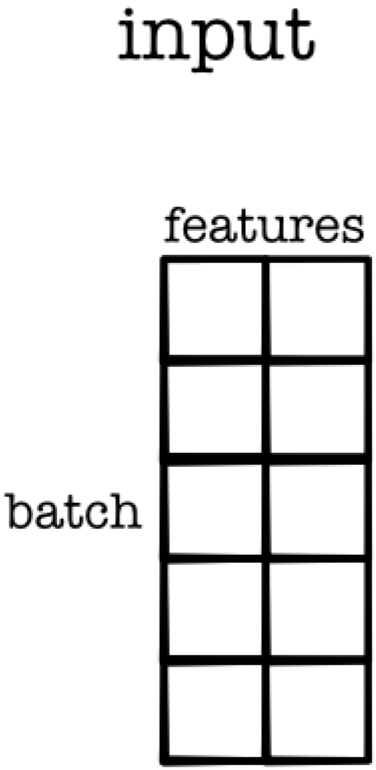
Parameter Fitting

1. Compute the loss function, :math:`L(w_1, w_2, b)`

2. See how small changes would change the loss

3. Update to parameters to locally reduce the loss

Batching



Loss

1) Compute Loss ::

```
out = model.forward(X).view(data.N)
    loss = -((out * y) + (out - 1.0) * (y - 1.0)).log()
```

Model: Math

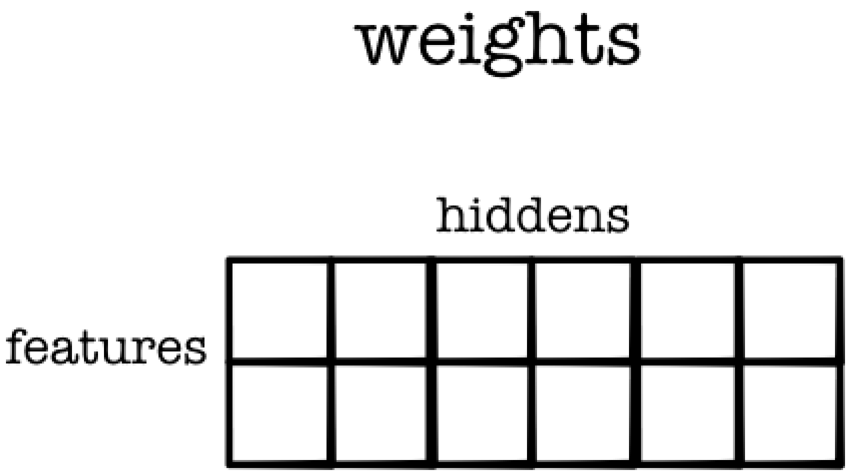
$$
\begin{aligned}
\mathrm{lin}(x; w, b) &= x_1 \times w_1 + x_2 \times w_2 + b \\
h_1 &= \mathrm{ReLU}(\mathrm{lin}(x; w^0, b^0)) \\
h_2 &= \mathrm{ReLU}(\mathrm{lin}(x; w^1, b^1)) \\
m(x_1, x_2) &= \mathrm{lin}(h; w, b)
\end{aligned}
$$

Model: Code

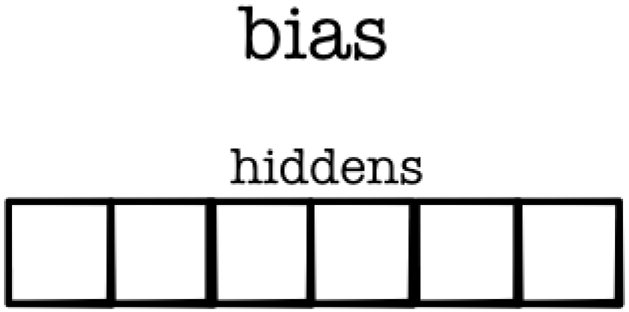
1) Model

```
class Network(minitorch.Module):
      def __init__(self):
        ...
        self.layer1 = Linear(2, HIDDEN)
        self.layer2 = Linear(HIDDEN, HIDDEN)
        self.layer3 = Linear(HIDDEN, 1)
```

Layer 1: Weight



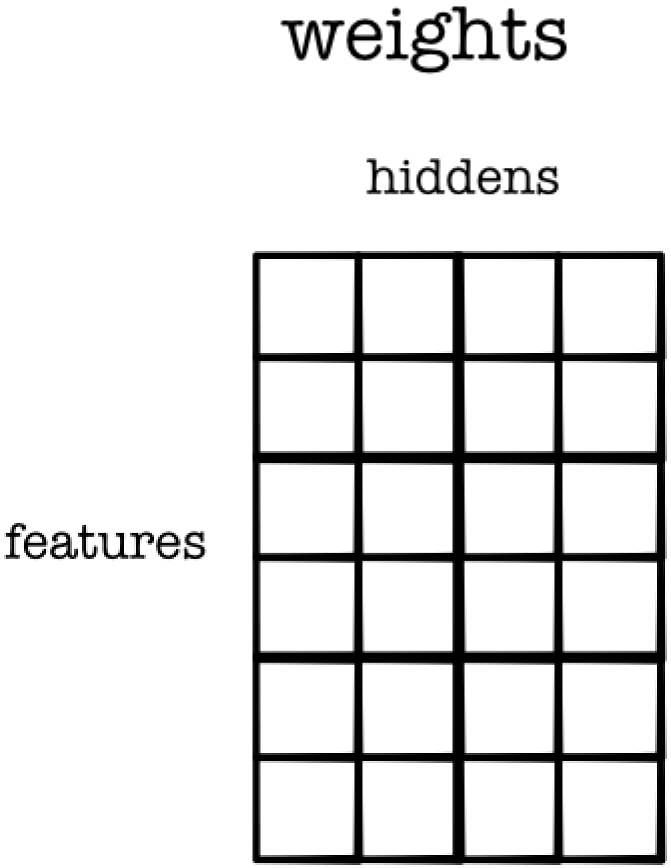
Layer 1: Bias



Key Task

- Use broadcasting to implement the linear function
- Hint: Align `batch` x `features` x `hidden` to make it work

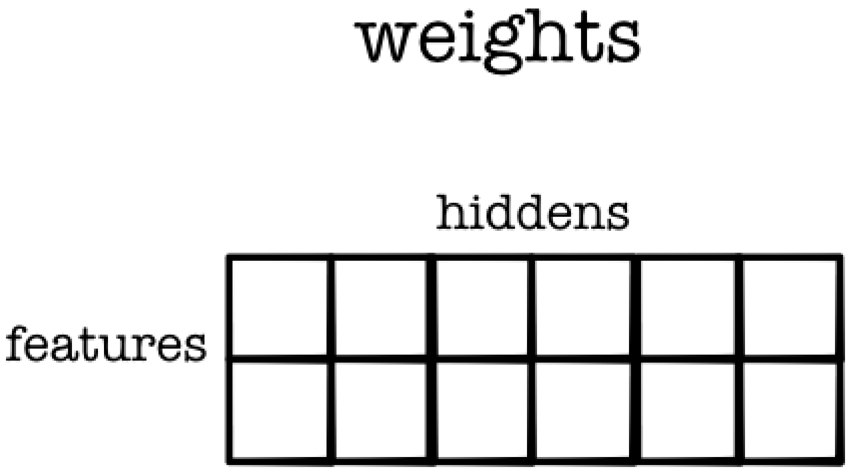
Layer 2: Weights



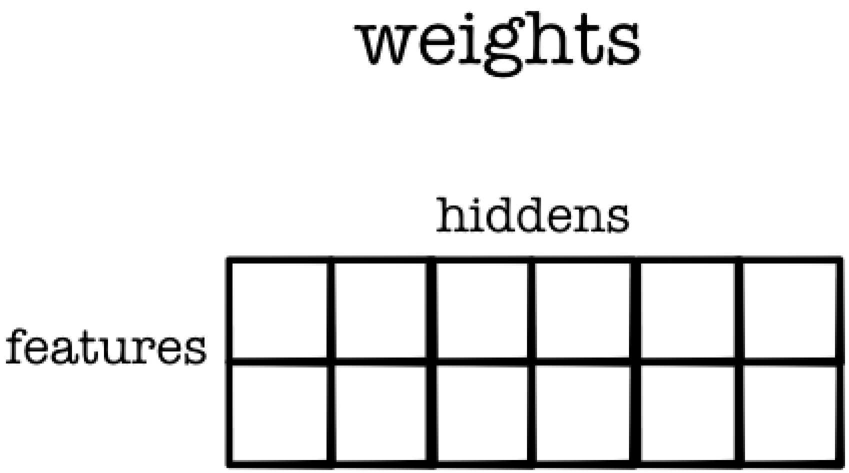
Compute Derivatives

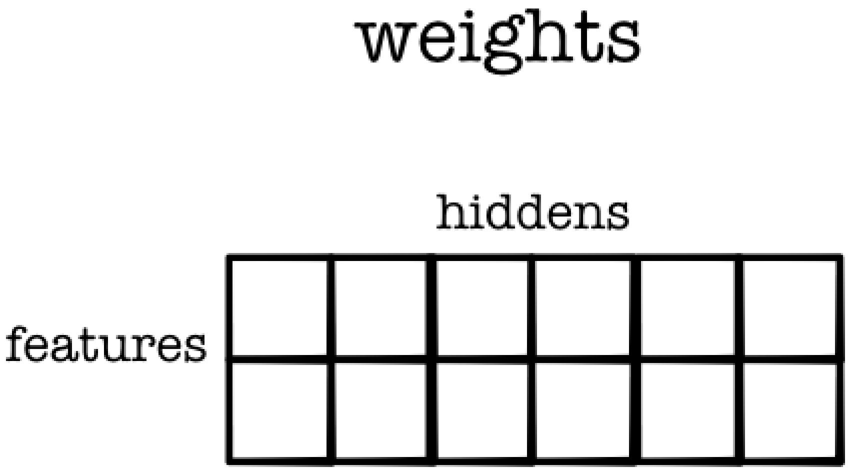
Step 2

```
(loss.sum().view(1)).backward()
print(model.layer1.w_1.value.grad)
```



Layer 1: Weight Grad





Update Parameters

Step 3

```
for p in model.parameters():
        if p.value.grad is not None:
            p.update(p.value - RATE * (p.value.grad / float(data.N)))
```

Broadcasting

- Batches
- Loss Computation
- Linear computation
- Autodifferentiation
- Gradient updates

Observations

- Exactly the same function as Module-1
- No loops within tensors