

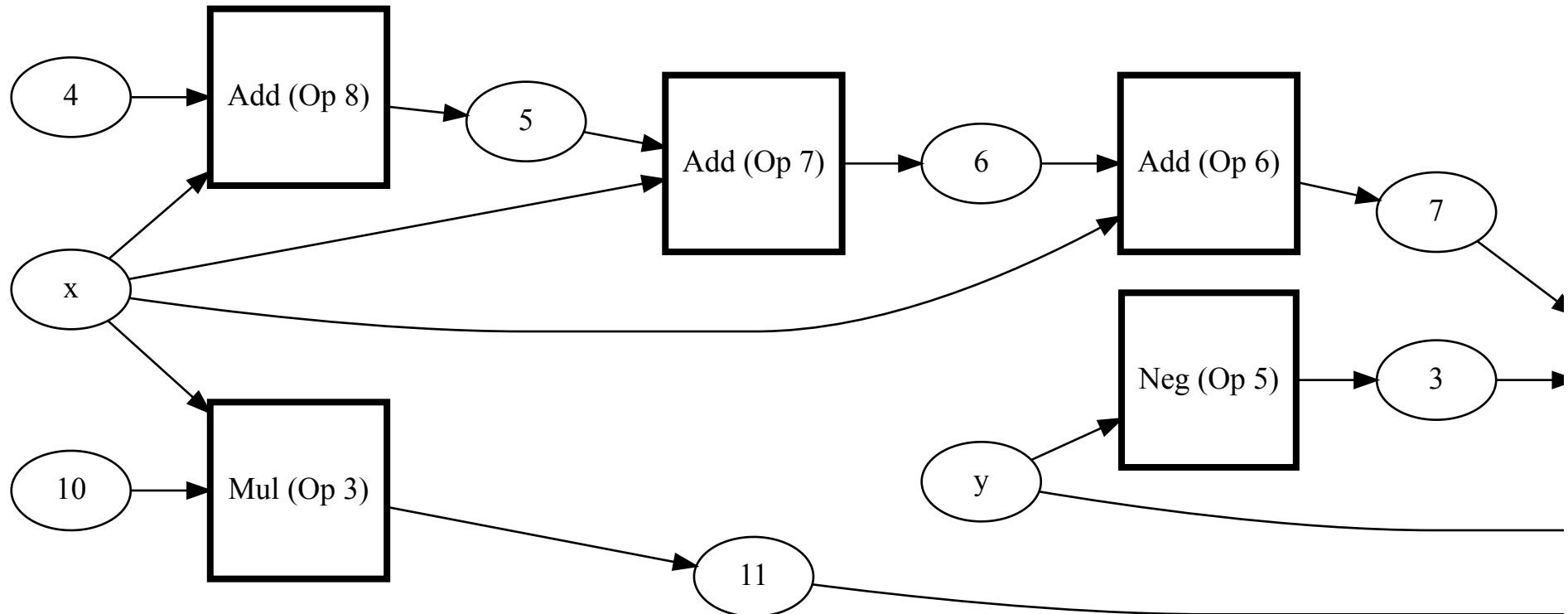
Module 2.0 - Neural Networks

Our Goal

Compute derivative of Python function with respect to inputs.

Example: Function

```
def expression():  
    x = Scalar(1.0, name="x")  
    y = Scalar(1.0, name="y")  
    z = -y * sum([x, x, x]) * y + 10.0 * x  
    h_x_y = z + z  
    return h_x_y
```

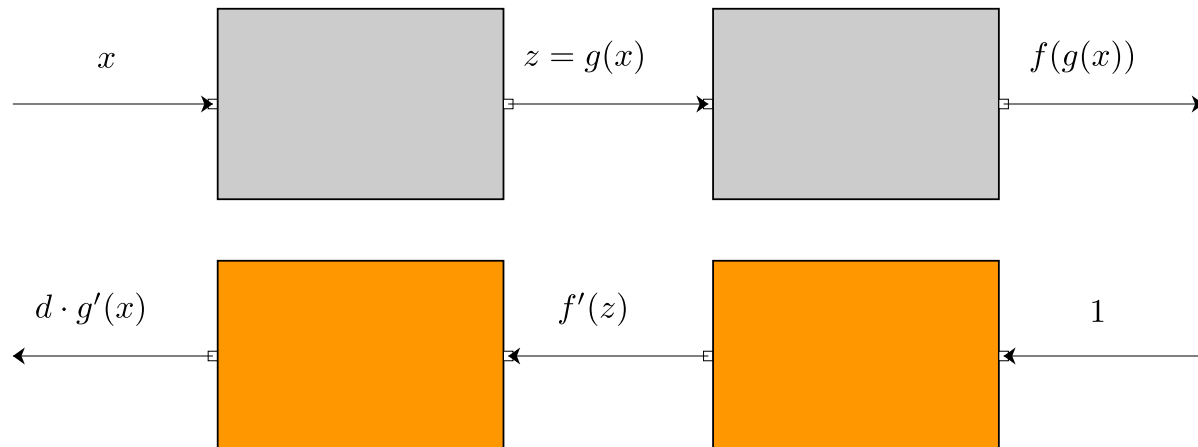


Chain Rule: Simple Case

$$z = g(x)$$

$$d = f'(z)$$

$$f'_x(g(x)) = g'(x) \times d$$



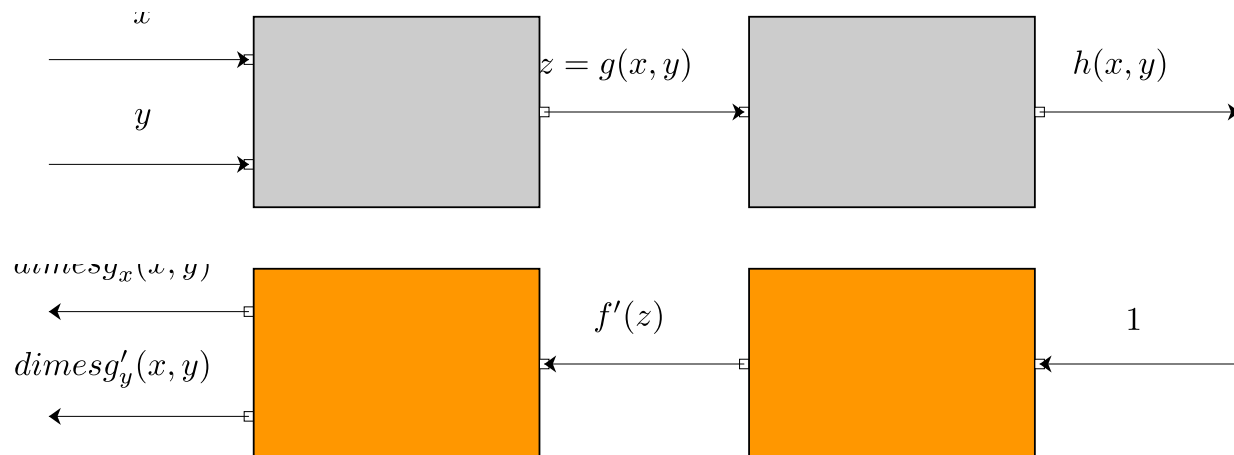
Chain Rule: Two Arguments

$$z = g(x, y)$$

$$d = f'(z)$$

$$f'_x(g(x, y)) = g'_x(x, y) \times d$$

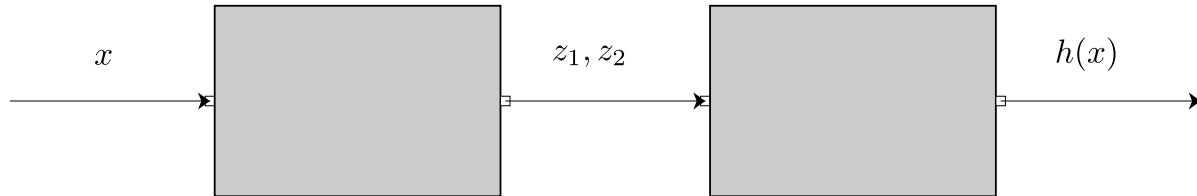
$$f'_y(g(x, y)) = g'_y(x, y) \times d$$



Chain Rule: Repeated Use

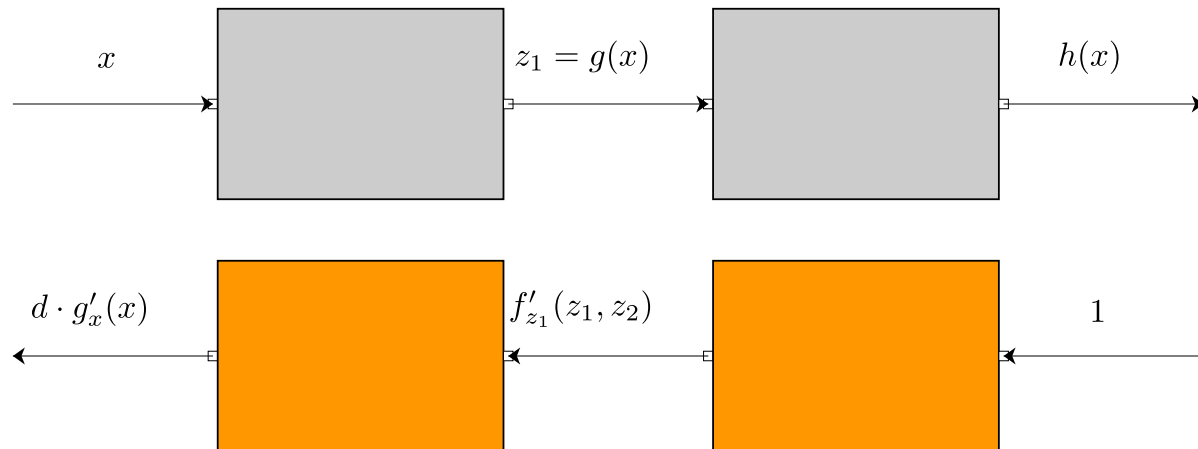
$$z = g(x)$$

$$f(z, z)$$



Chain Rule: Repeated Use

$$d = f'_{z_1}(z_1, z_2) + f'_{z_2}(z_1, z_2)$$
$$h'_x(x) = d \times g'_x(x)$$



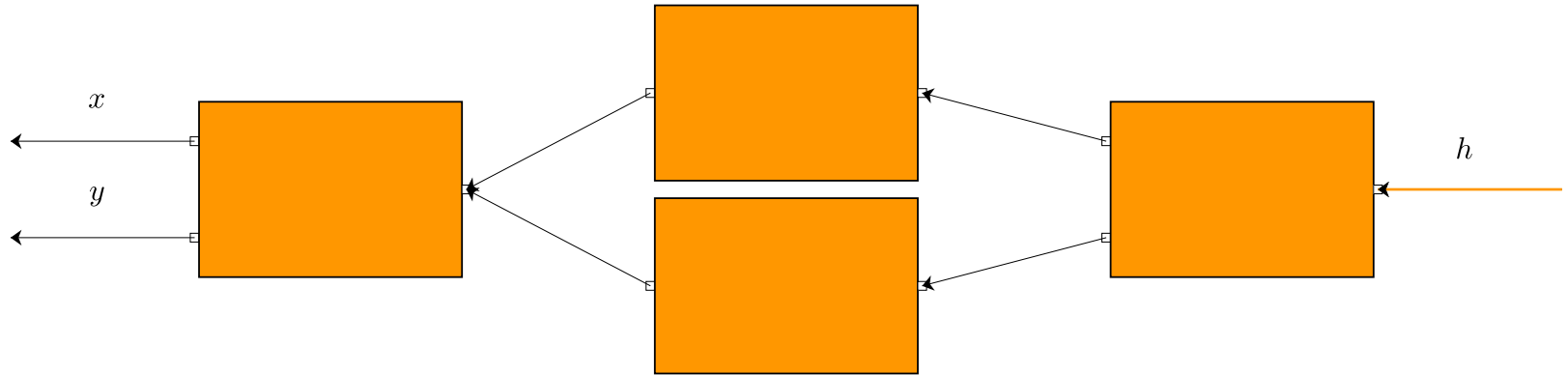
Algorithm: Outer Loop

1. Call topological sort
2. Create dict of edges and empty d values.
3. For each edge and d in topological order:

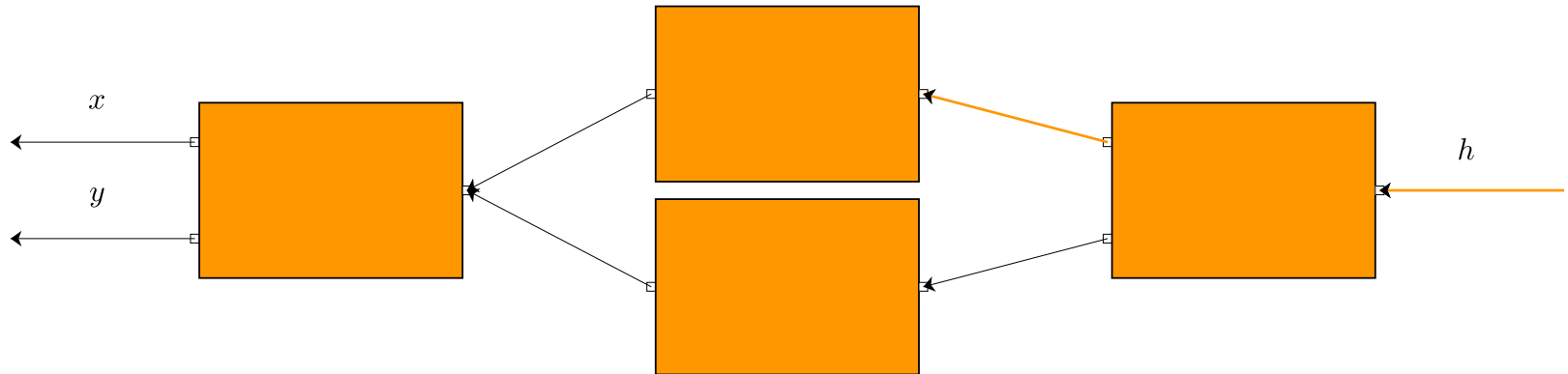
Algorithm: Inner Loop

1. If edge goes to Leaf, done
2. Call `backward` with d on previous box
3. Loop through all its input edges and add derivative

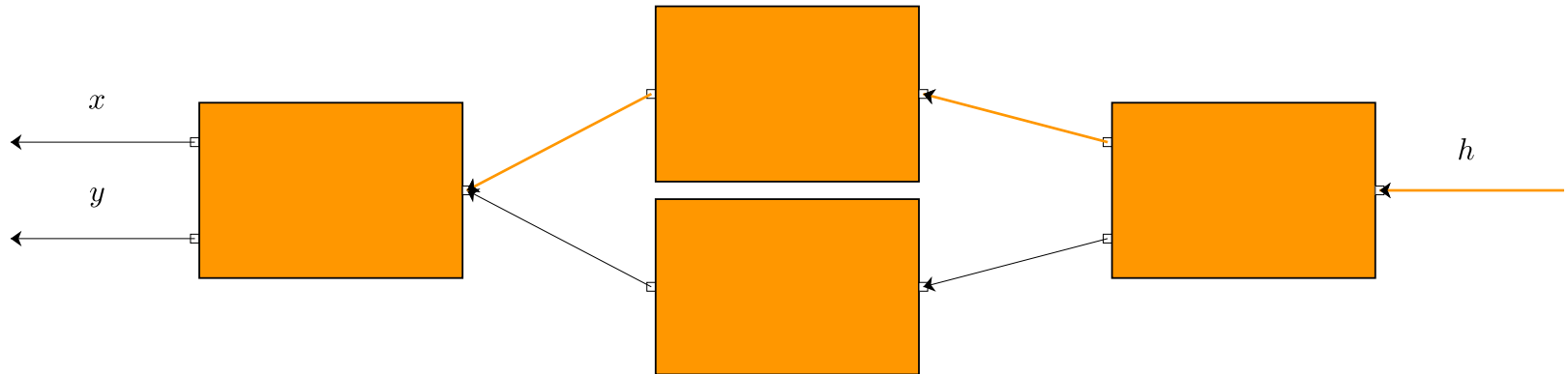
Example



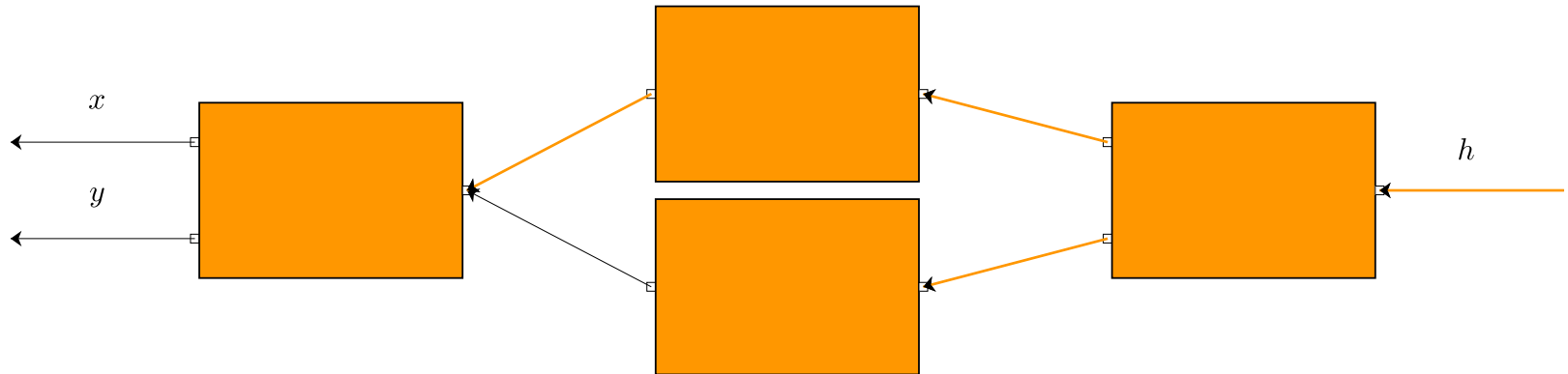
Example



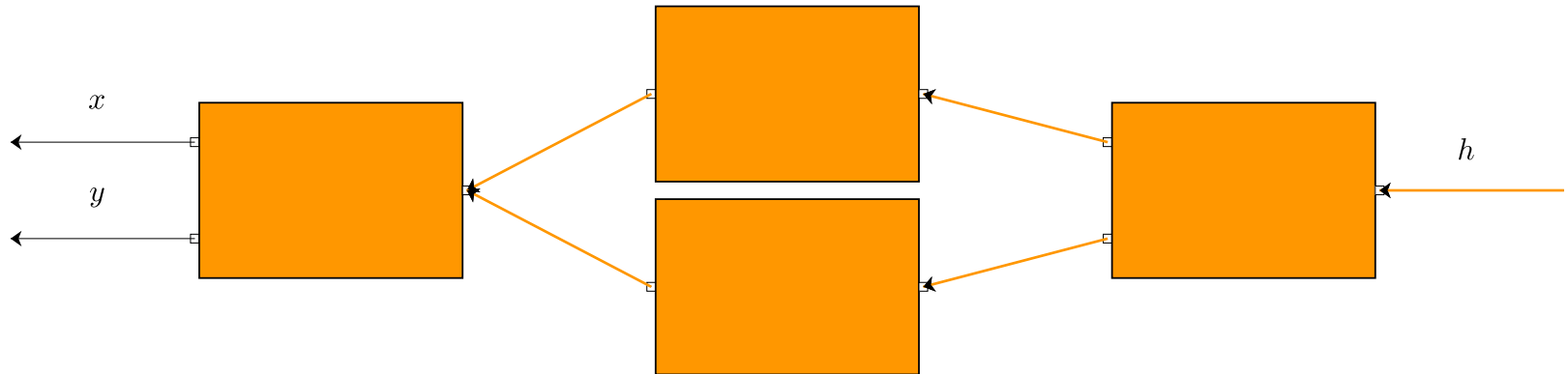
Example



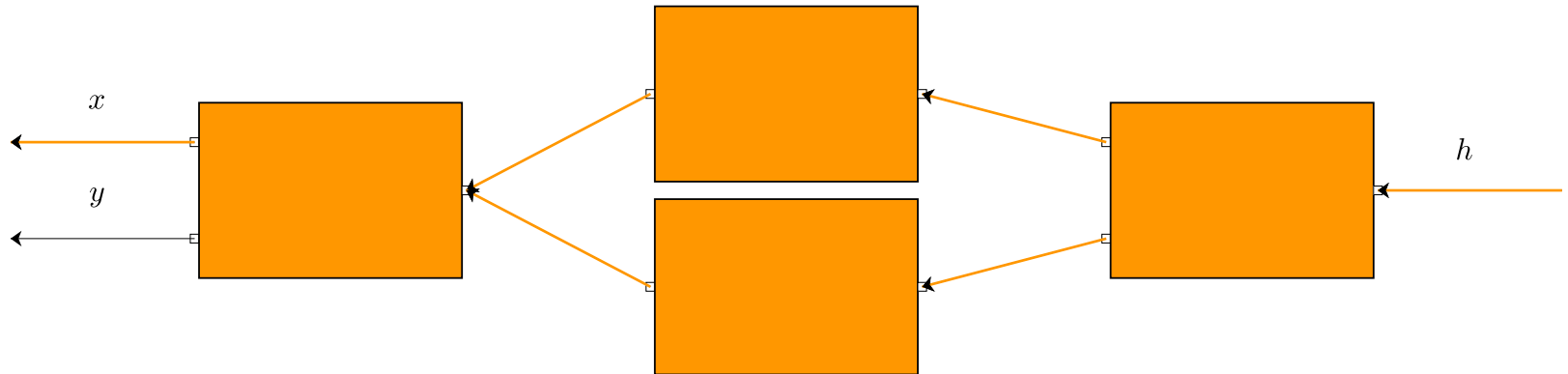
Example



Example



Example



Example

Quiz

Outline

- Model Training
- Neural Networks
- Modern Models

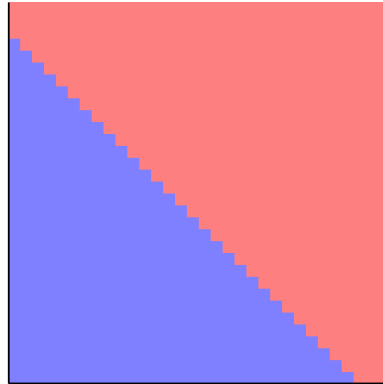
Model Training

Reminder: MiniML

- Dataset - Data to fit
- Model - Shape of fit
- Loss - Goodness of fit

Model 1

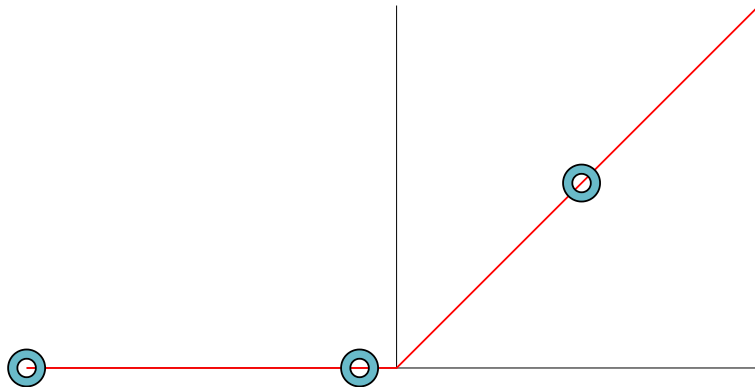
- Linear Model



Point Loss

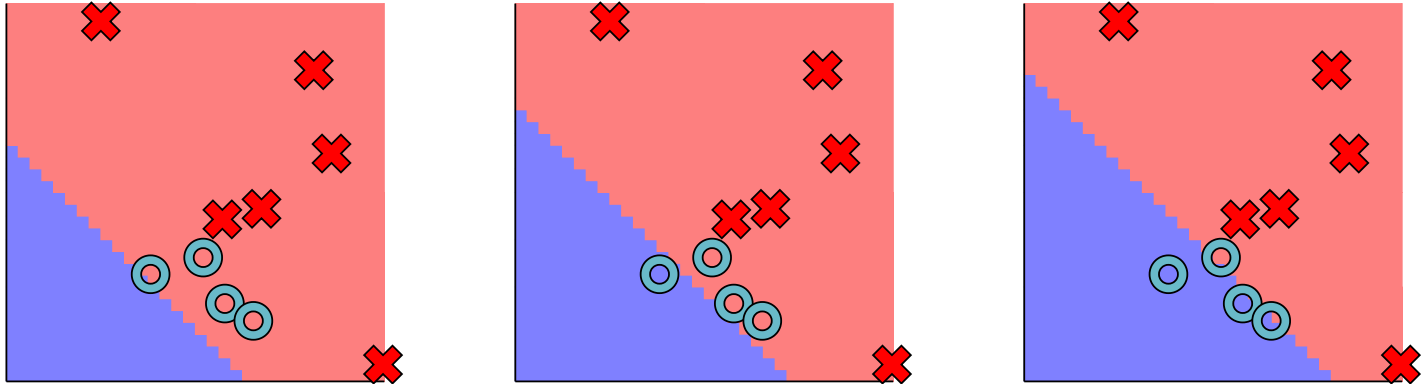
```
def point_loss(x):  
    return minitorch.operators.relu(x)  
  
def full_loss(m):  
    l = 0  
    for x, y in zip(s.X, s.y):  
        l += point_loss(-y * m.forward(*x))  
    return -l
```

```
graph(point_loss, [], [-2, -0.2, 1])
```



Class Goal

- Find parameters that minimize loss



Parameter Fitting

1. (Forward!) Compute the loss function, $L(w_1, w_2, b)$
2. (Backward!) See how small changes would change the loss
3. Update to parameters to locally reduce the loss

Update Procedure

Module for Linear

```
class LinearModule(minitorch.Module):
    def __init__(self):
        super().__init__()
        # 0.0 is start value for param
        self.w1 = Parameter(Scalar(0.0))
        self.w2 = Parameter(Scalar(0.0))
        self.bias = Parameter(Scalar(0.0))

    def forward(self, x1: Scalar, x2: Scalar) -> Scalar:
        return x1 * self.w1.value + x2 * self.w2.value + self.bias.value
```


Training Loop

```
def train_step(optim, model, data):  
    # Step 1 - Forward (Loss function)  
    x_1, x_2 = Scalar(data[0]), Scalar(data[1])  
    loss = model.forward(x_1, x_2).relu()  
    # Step 2 - Backward (Compute derivative)  
    loss.backward()  
    # Step 3 - Update Params  
    optim.step()
```


More Features: Linear Model

$$\text{lin}(x; w, b) = x_1 \times w_1 + \dots + x_n \times w_n + b$$

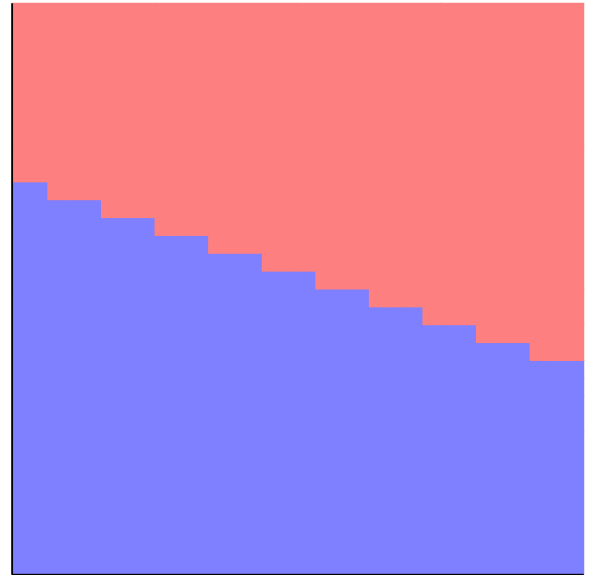
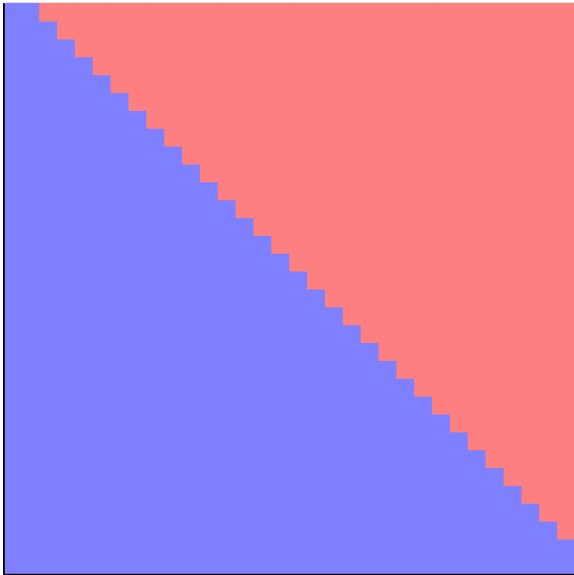
More Features: Linear (Code)

```
class LinearModule(minitorch.Module):
    def __init__(self, in_size):
        super().__init__()
        self.weights = []
        self.bias = []
        # Need add parameter
        for i in range(in_size):
            self.weights.append(
                self.add_parameter(
                    f"weight_{i}", 0.0
                )
            )
```

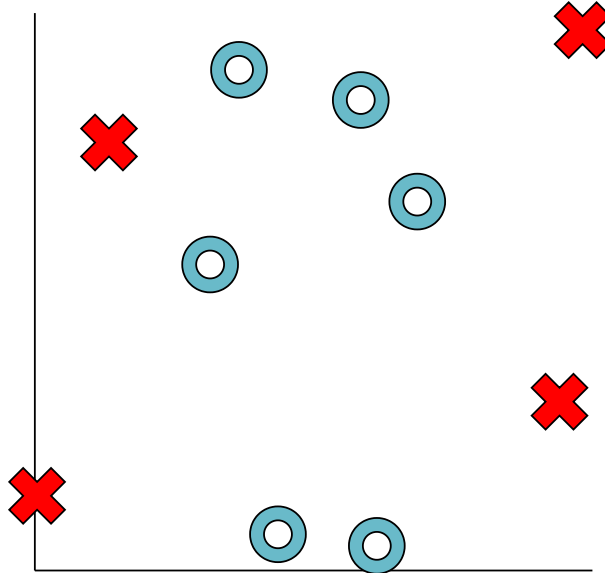

Neural Networks

Linear Model Example

- Parameters

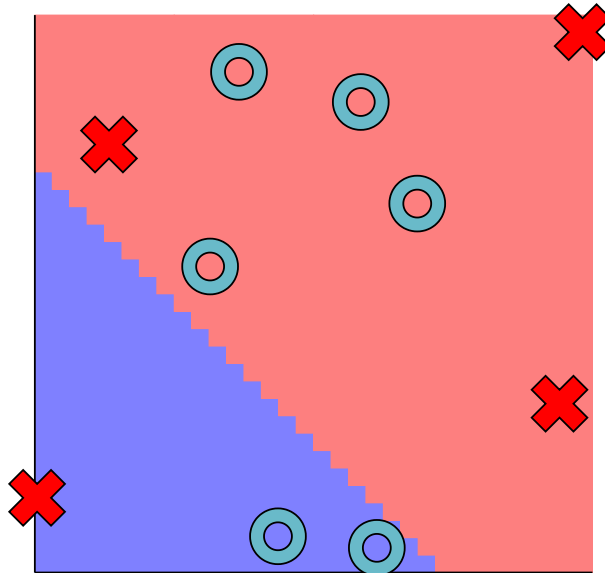


Harder Datasets



Harder Datasets

- Model may not be good with *any* parameters.



Neural Networks

- New *model*
- Uses repeated splits of data
- Loss will not change

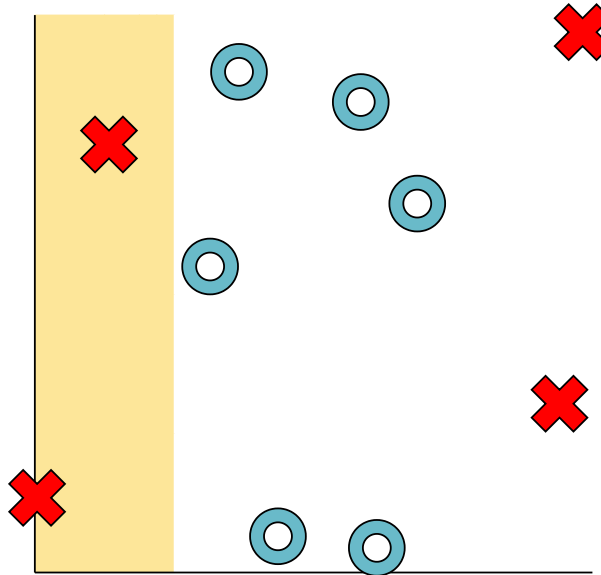
Intuition: Neural Networks

1. Apply many linear separators
2. Reshape the data space based on results
3. Apply a linear model on new space

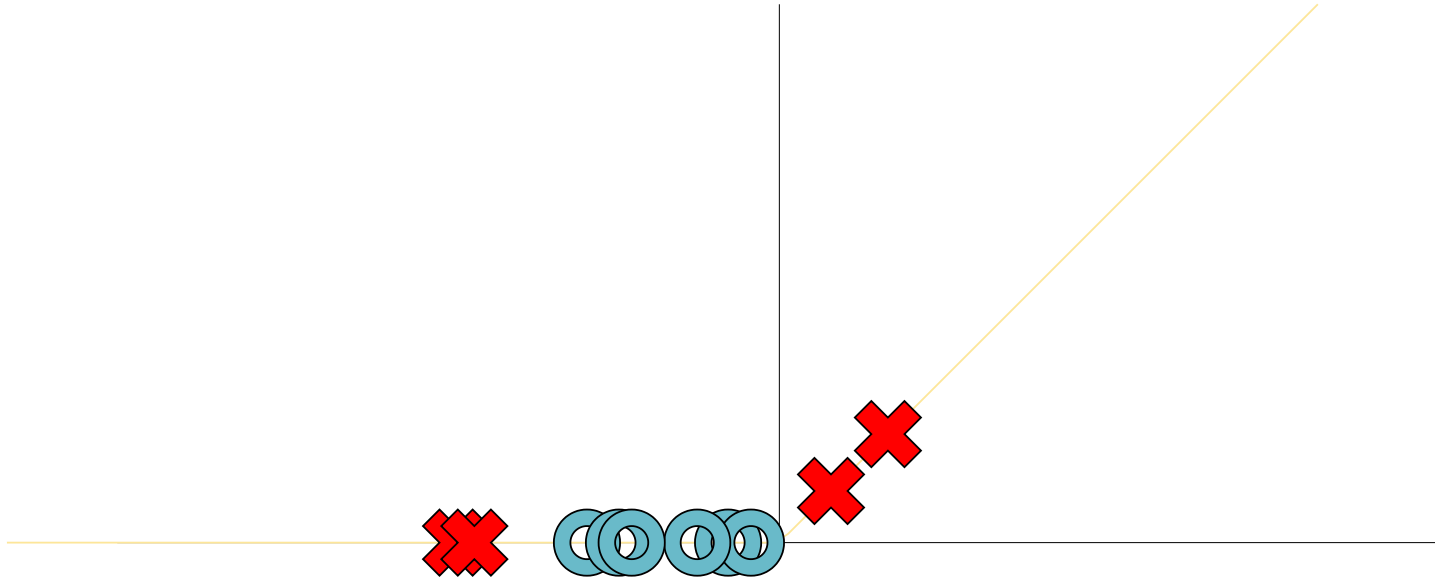
Notation: Multiple Parameters

- Use superscript w^0 and w^1 to indicate different parameters.
- Our final model will have many linears.
- These will become Torch sub-modules.

Intuition: Split 1



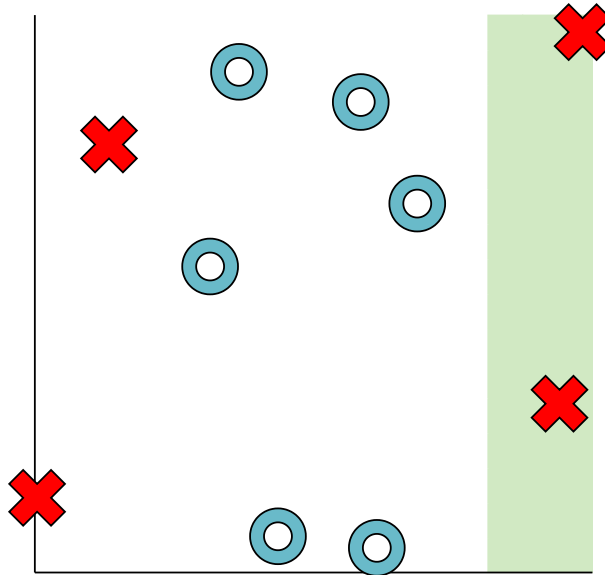
Reshape: ReLU



Math View

$$h_1 = \text{ReLU}(\text{lin}(x; w^0, b^0))$$

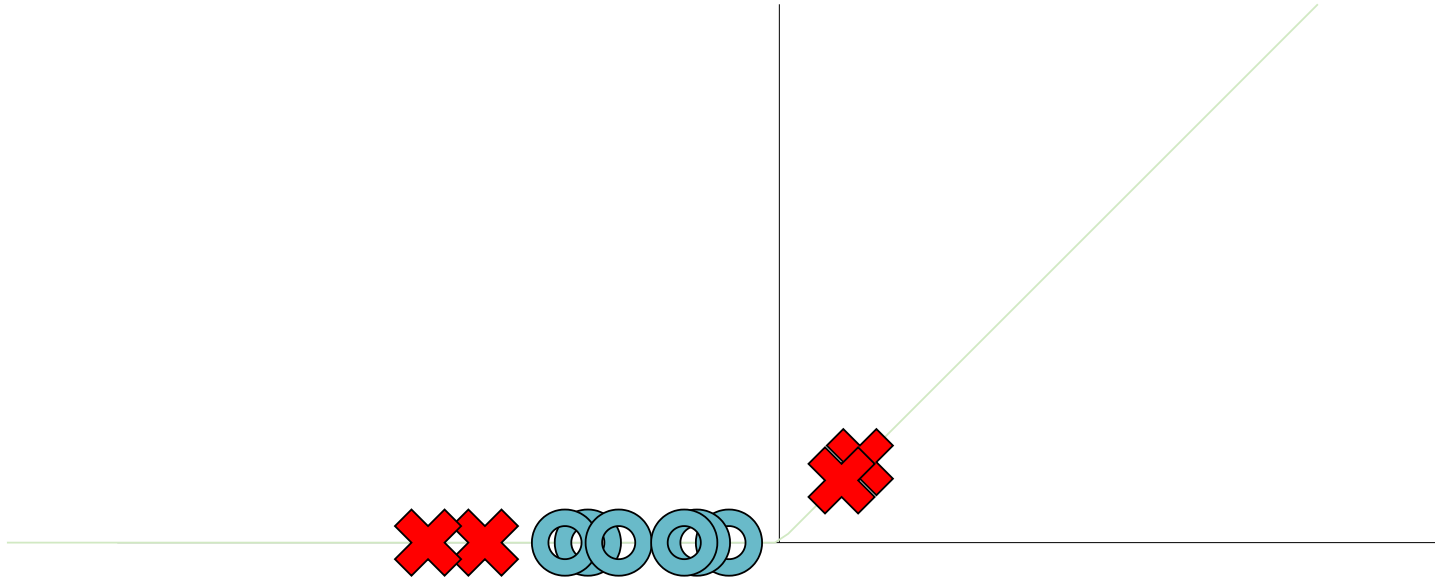
Intuition: Split 2



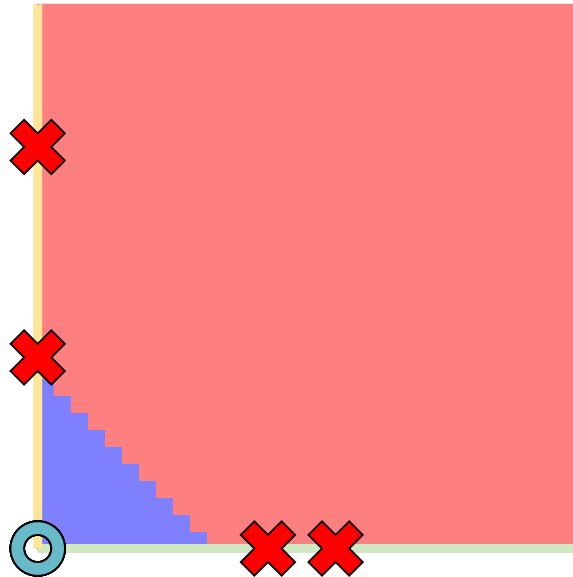
Math View

$$h_2 = \text{ReLU}(\text{lin}(x; w^1, b^1))$$

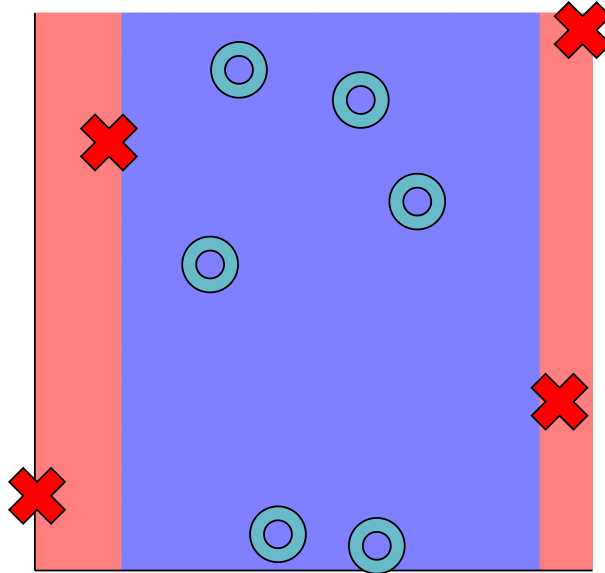
Reshape: ReLU



Reshape: ReLU



Final Layer



Math View

$$h_1 = \text{ReLU}(x_1 \times w_1^0 + x_2 \times w_2^0 + b^0)$$

$$h_2 = \text{ReLU}(x_1 \times w_1^1 + x_2 \times w_2^1 + b^1)$$

$$m(x_1, x_2) = h_1 \times w_1 + h_2 \times w_2 + b$$

Parameters: $w_1, w_2, w_1^0, w_2^0, w_1^1, w_2^1, b, b^0, b^1$

Math View (Alt)

$$h_1 = \text{ReLU}(\text{lin}(x; w^0, b^0))$$

$$h_2 = \text{ReLU}(\text{lin}(x; w^1, b^1))$$

$$m(x_1, x_2) = \text{lin}(h; w, b)$$

Code View

Linear

```
class LinearModule(Module):
    def __init__(self):
        super().__init__()
        self.w_1 = Parameter(Scalar(0.0))
        self.w_2 = Parameter(Scalar(0.0))
        self.b = Parameter(Scalar(0.0))

    def forward(self, inputs):
        return inputs[0] * self.w_1.value + inputs[1] * self.w_2.value + self.b.value
```


Code View

Model

```
class Network(minitorch.Module):  
    def __init__(self):  
        super().__init__()  
        self.unit1 = LinearModule()  
        self.unit2 = LinearModule()  
        self.classify = LinearModule()  
  
    def forward(self, x):  
        h1 = self.unit1.forward(x).relu()  
        h2 = self.unit2.forward(x).relu()  
        return self.classify.forward((h1, h2))
```


Training

- All the parameters in model are leaves
- Computing backward on loss fills their derivative

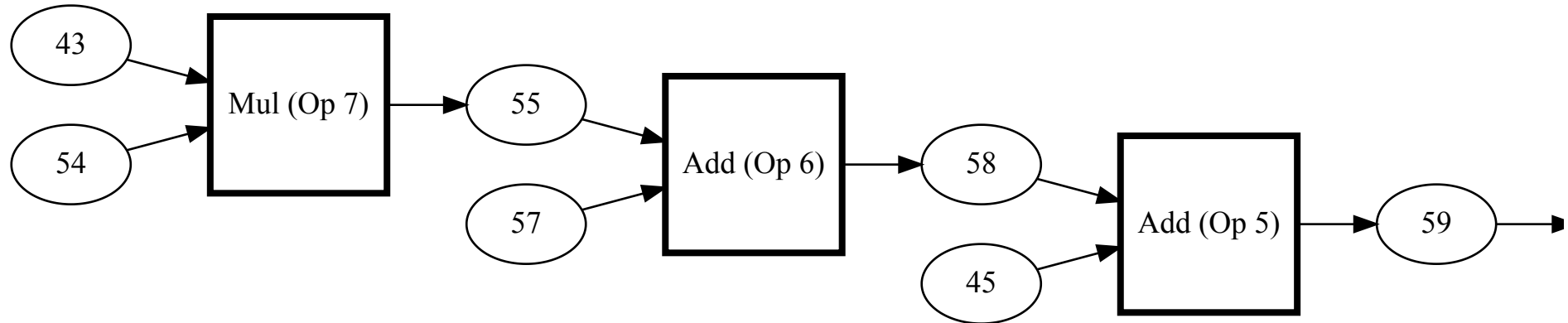
```
model = Network()  
parameters = dict(model.named_parameters())  
parameters
```

```
{'unit1.w_1': Scalar(0.000000),  
 'unit1.w_2': Scalar(0.000000),  
 'unit1.b': Scalar(0.000000),  
 'unit2.w_1': Scalar(0.000000),  
 'unit2.w_2': Scalar(0.000000),  
 'unit2.b': Scalar(0.000000),  
 'classify.w_1': Scalar(0.000000),  
 'classify.w_2': Scalar(0.000000),  
 'classify.b': Scalar(0.000000)}
```


Derivatives

- All the parameters in model are leaf Variables

```
model = Network()  
x1, x2 = Scalar(0.5), Scalar(0.5)  
# Step 1  
out = model.forward((0.5, 0.5))  
loss = out.relu()  
# Step 2  
SVG(make_graph(loss, lr=True))
```



Derivatives

- All the parameters in model are leaf scalars

```
parameters["unit1.w_1"].value.derivative
```


Playground

NN Playground

QA

