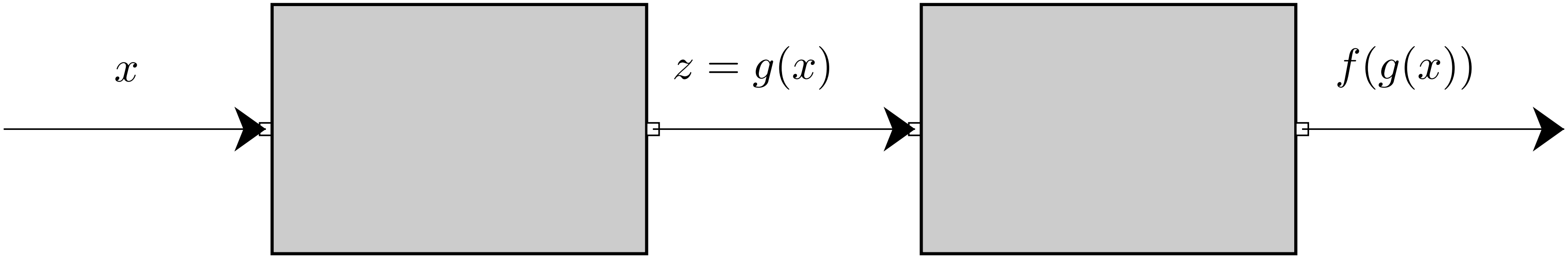
Module 2.0 - Neural Networks

Complex Graphs

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

Chain Rule

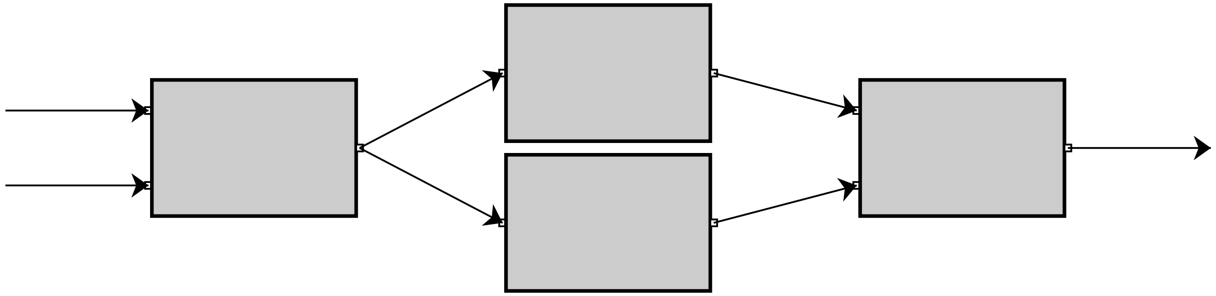
$$z = g(x)$$
$$d = f'(z)$$
$$f'_x(g(x)) = g'(x) \times d$$



Loading [MathJax]/extensions/Safe.js

Two Arguments: Chain

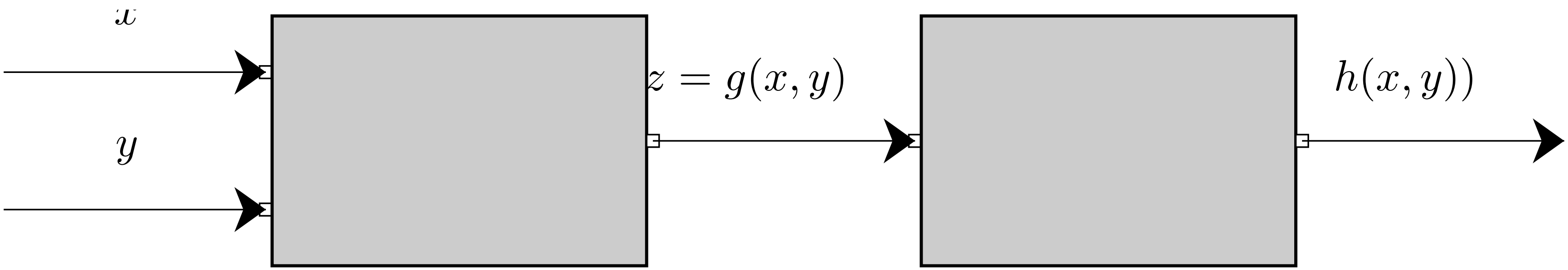
$$f(g(x, y))$$



Loading [MathJax]/extensions/Safe.js

Two Arguments: Chain

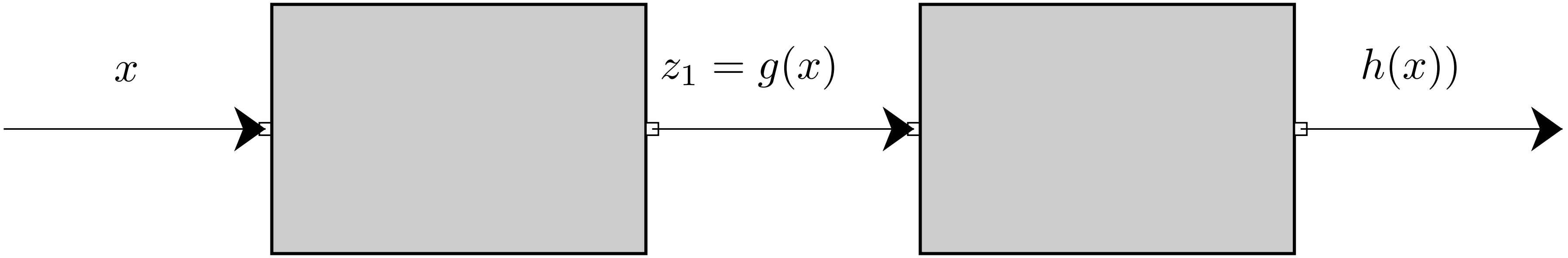
$$z = g(x, y)$$
$$d = f'(z)$$
$$f'_x(g(x, y)) = g'_x(x, y) \times d_{out}$$
$$f'_y(g(x, y)) = g'_y(x, y) \times d_{out}$$



Loading [MathJax]/extensions/Safe.js

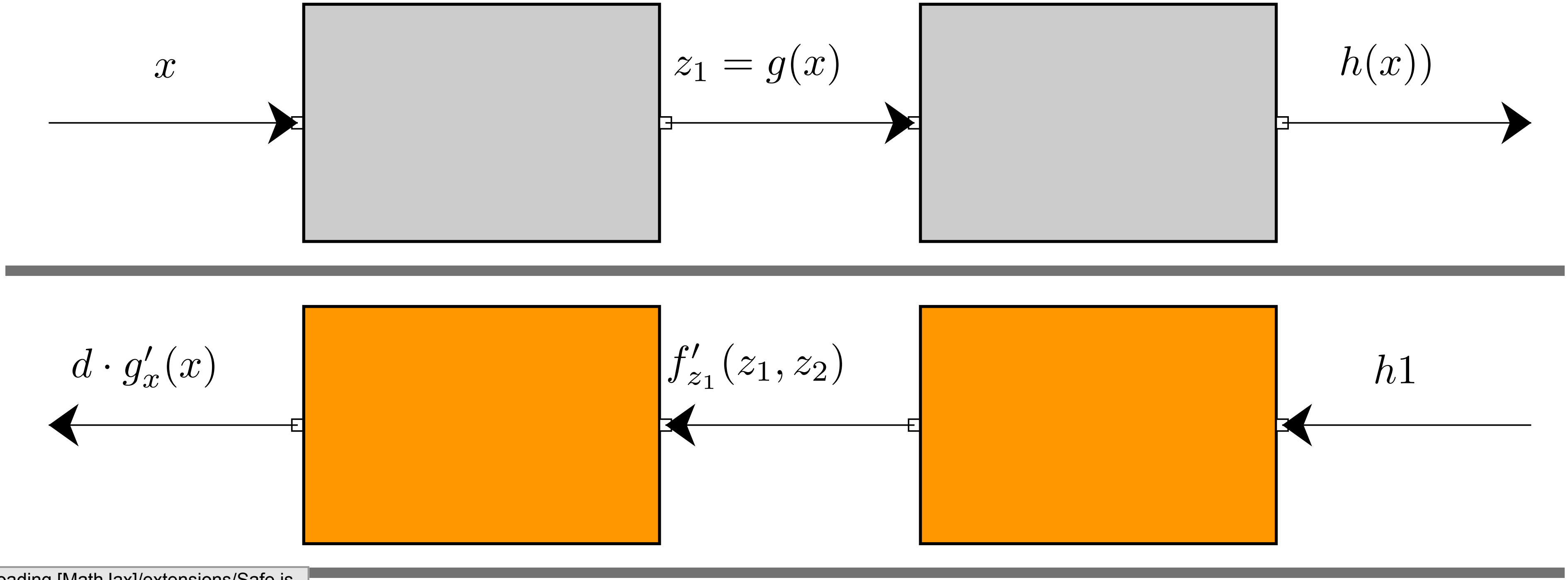
Multivariable Chain

$$f(g(x), h(x))$$



Multivariable Chain

$$d = 1 \cdot f'_{z_1}(z_1, z_2) + 1 \cdot f'_{z_2}(z_1, z_2)$$
$$h'_x(x) = d \cdot g'_x(x)$$



Topological Sorting

- Topological Sorting
- High-level -> Run depth first search and mark nodes.

Algorithm: Outer Loop

1. Call topological sort

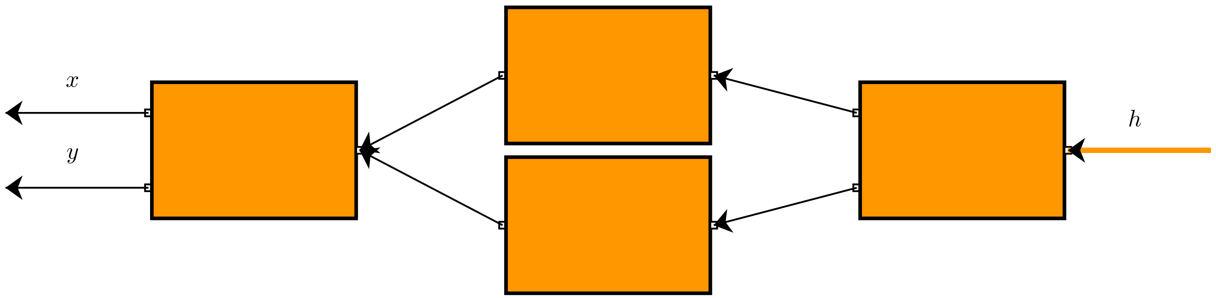
2. Create dict of Variables and derivatives

3. For each node in backward order:

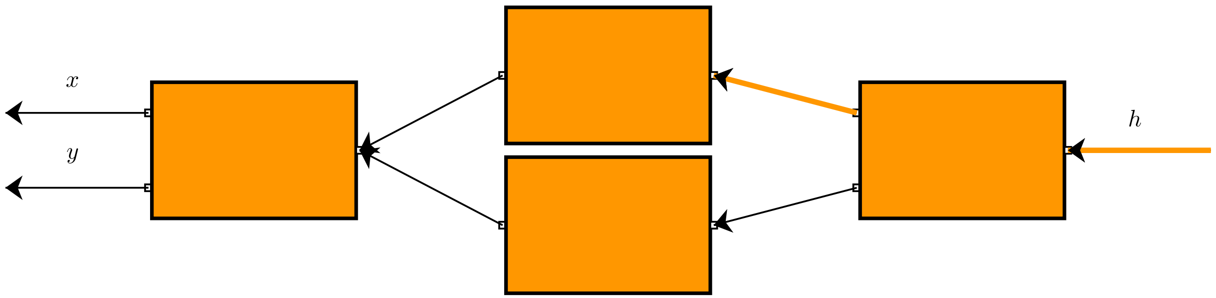
Algorithm: Inner Loop

1. if Variable is leaf, add its final derivative
2. if the Variable is not a leaf, 1) call backward with d 2) loop through all the Variables+derivative 3) accumulate derivatives for the Variable

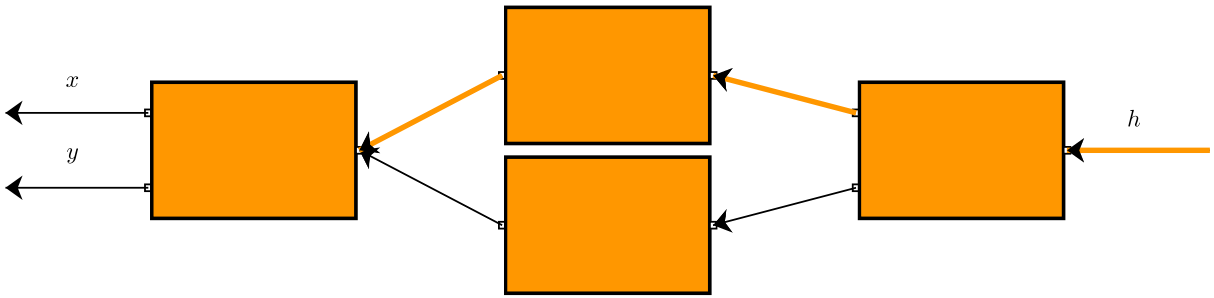
Example



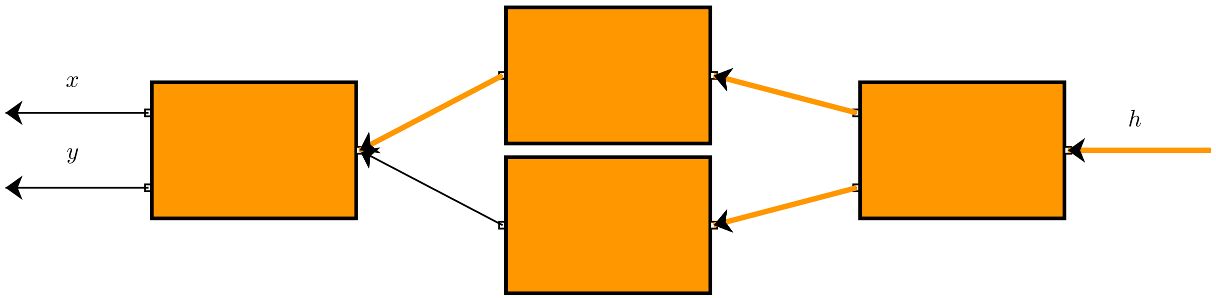
Example



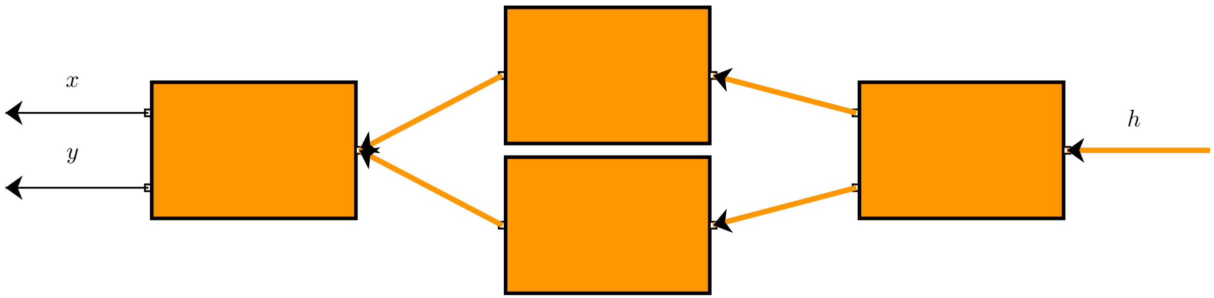
Example



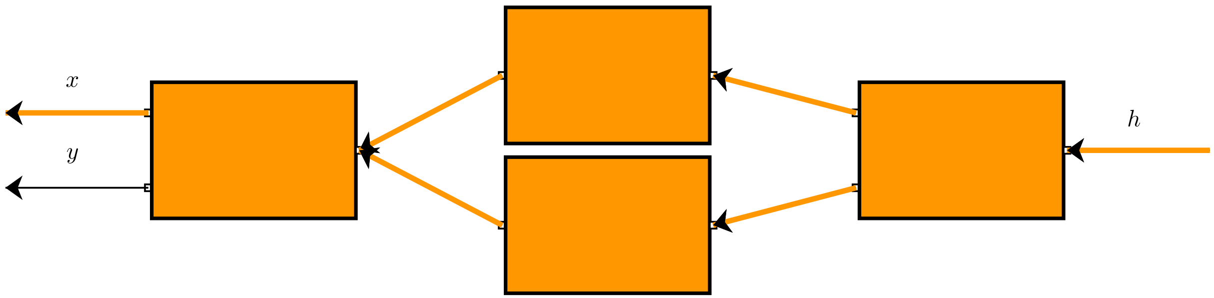
Example



Example



Example



Example

Lecture Quiz

Outline

- Model Training
- Neural Networks
- Modern Models

Model Training

Reminder

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

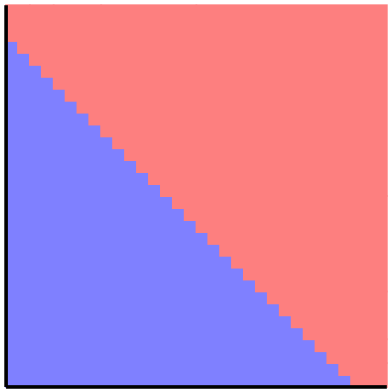
Model 1

- Linear Model

```
@dataclass
class Linear:
    # Parameters (Now scalars)
    w1: Scalar
    w2: Scalar
    b: Scalar

    def forward(self, x1, x2):
        return self.w1 * x1 + self.w2 * x2 + self.b
```

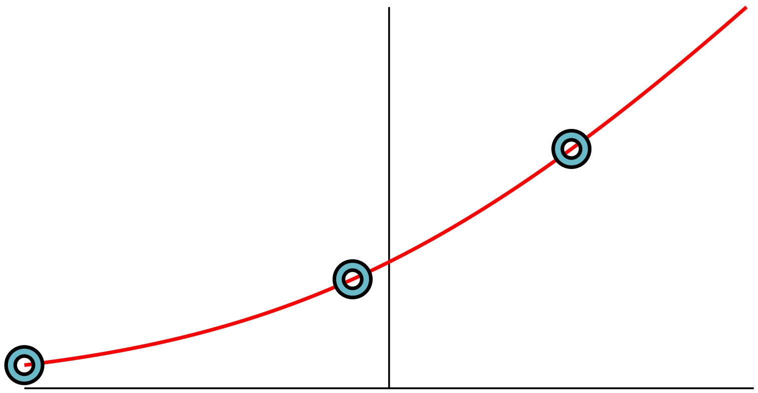
Decision Boundary: Model 1



Point Loss

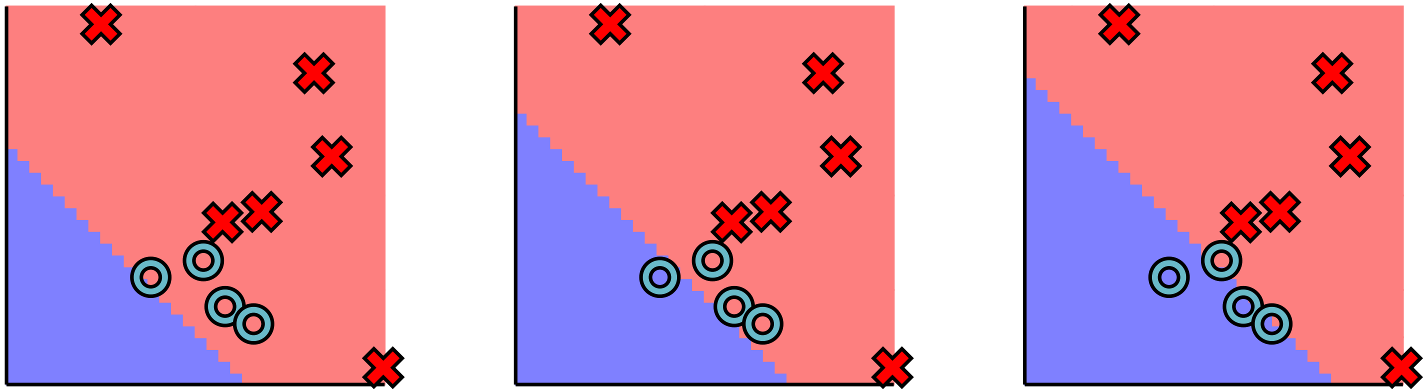
```
def point_loss(x):
    return -math.log(minitorch.operators.sigmoid(-x))


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

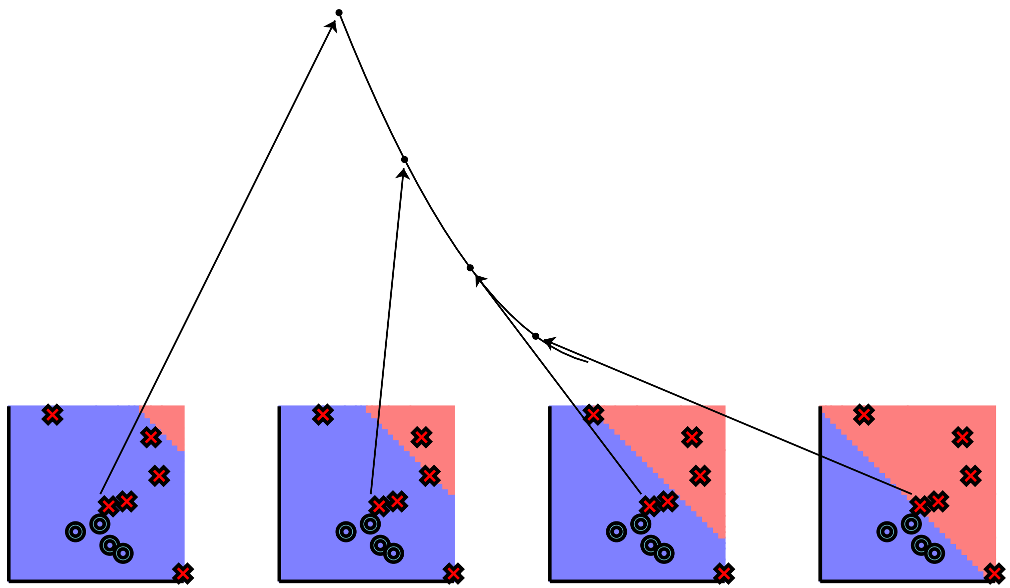


Class Goal

- Find parameters that minimize loss



Update Procedure



Full: Module

```python
class LinearModule(minitorch.Module):
    def __init__(self):
        super().__init__()
        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
```

Full: Loop

```python
def train_sketch():
    x_1, x_2 = Scalar(x[i][0]), Scalar(x[i][1])
    # Forward (loss function)
    loss = -self.model.forward((x_1, x_2)).log()
    # Backward
    loss.backward()
    # Update Params
    ...
```

Linear Model

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

Loading [MathJax]/extensions/Safe.js

More Features

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

Neural Networks

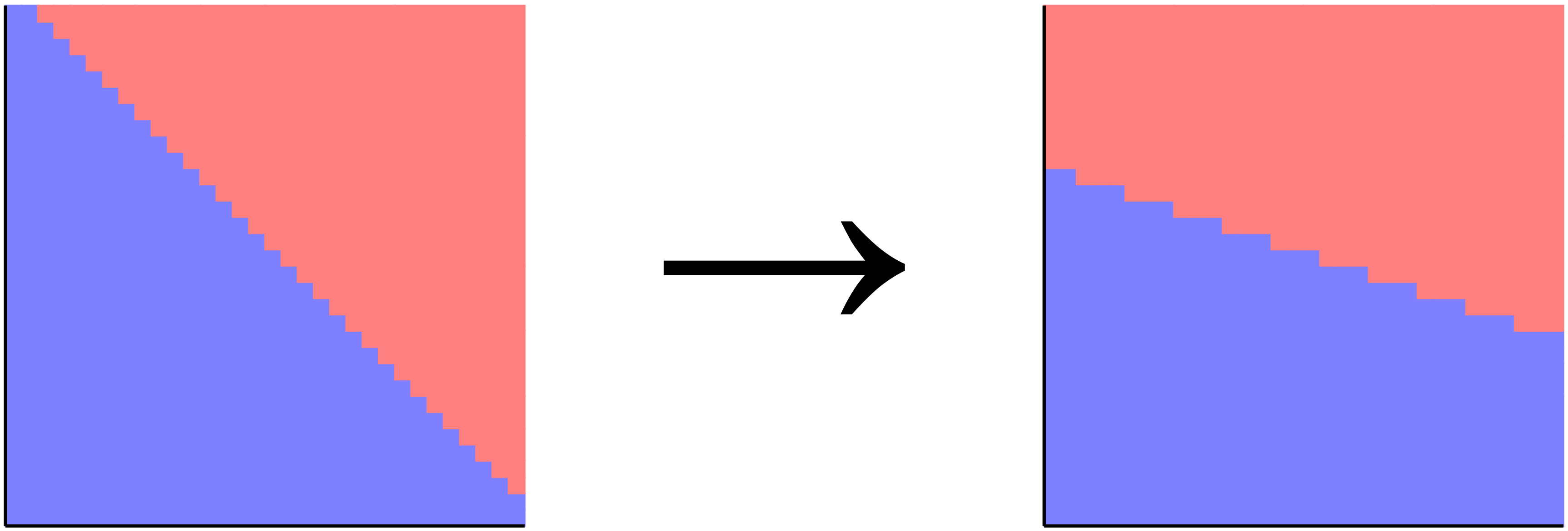
Loading [MathJax]/extensions/Safe.js

Reminder

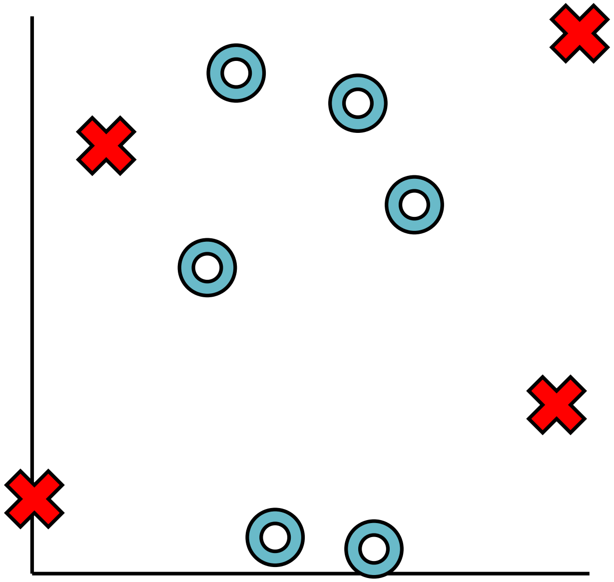
- Dataset - Data to fit
- Model - Shape of fit
- Loss - Goodness of fit

Linear Model Example

- Parameters

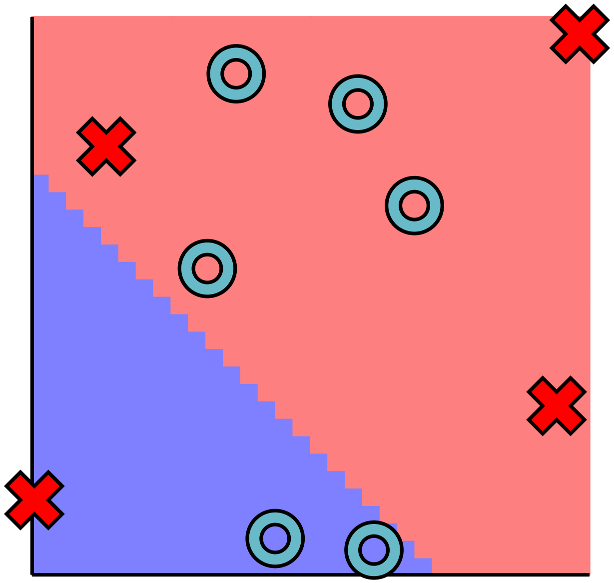


Harder Datasets



Harder Datasets

- Model may be too "weak"



Neural Networks

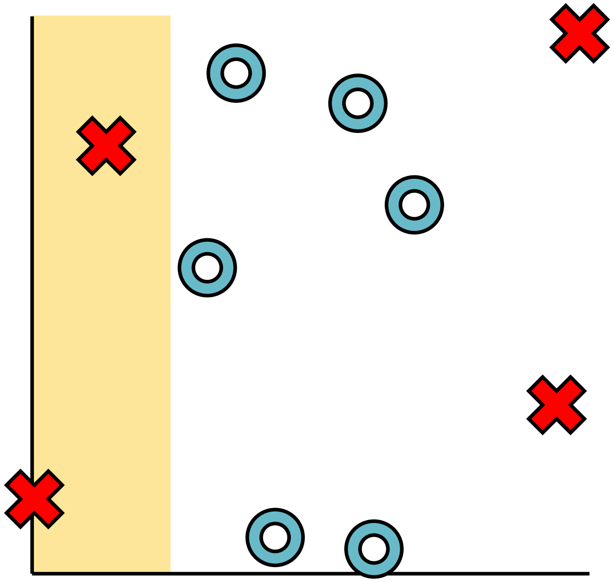
- New *model*
- Uses repeated splits of data
- Loss will not change

Intuition: Neural Networks

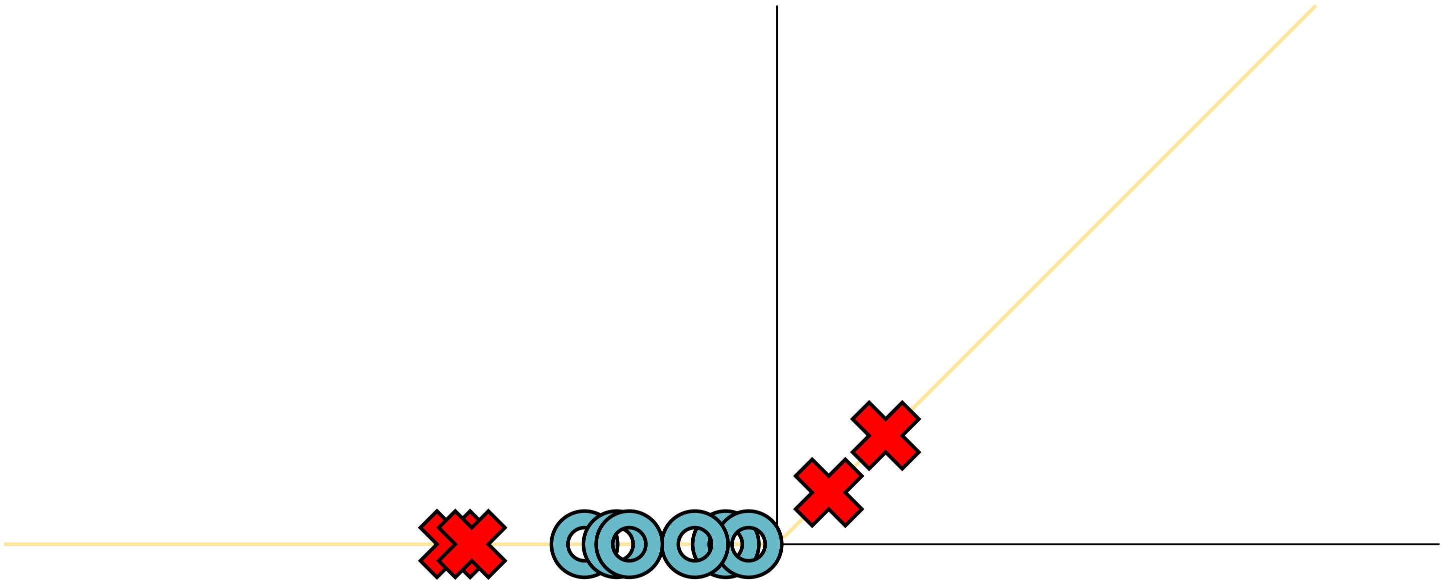
1) Apply many linear seperators 2) Reshape the data space based on results 3) Apply a linear model on new space

Intuition: Split 1

```
yellow = Linear(-1, 0, 0.25)
```



Reshape: ReLU



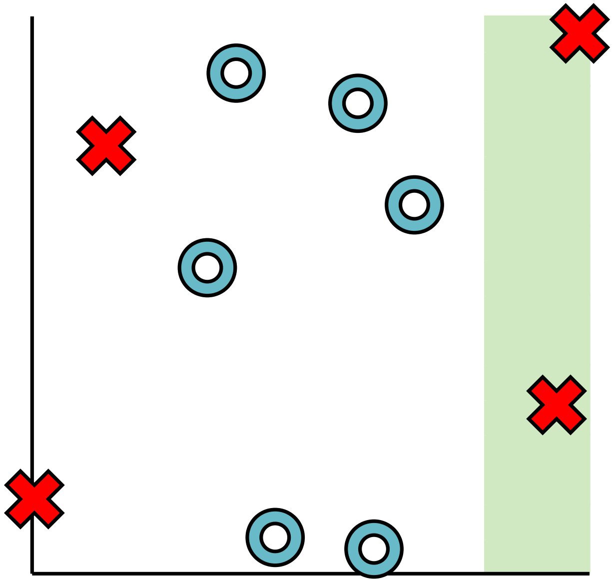
Loading [MathJax]/extensions/Safe.js

Math View

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

Intuition: Split 2

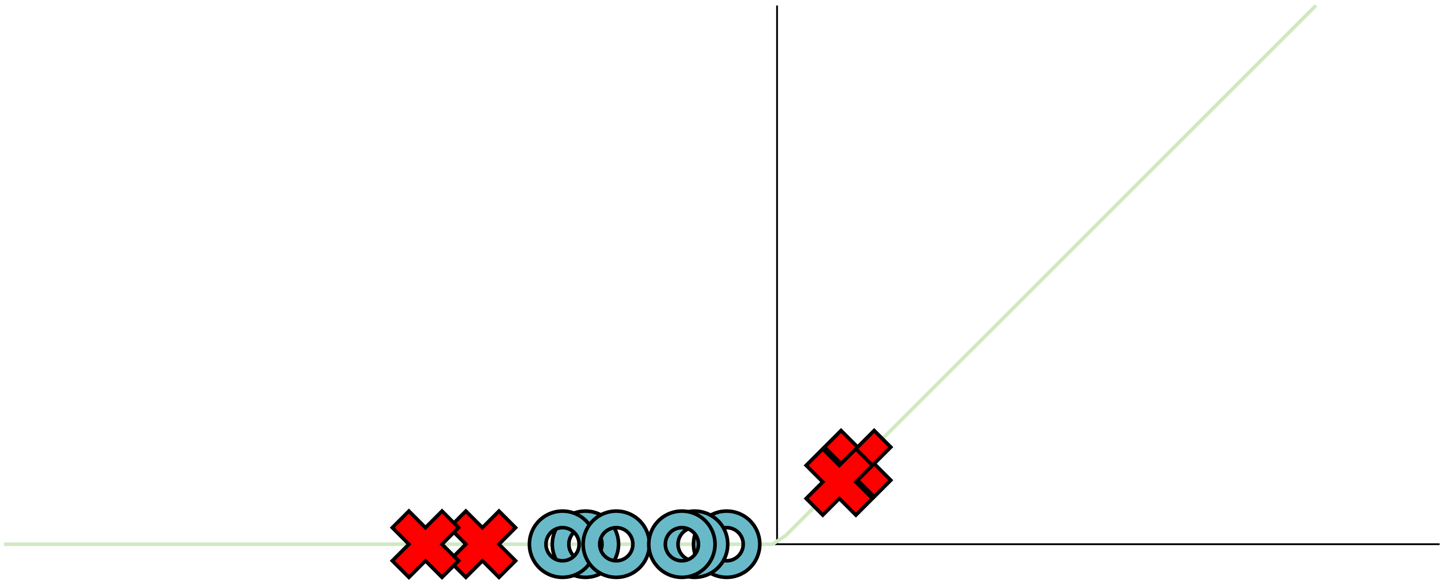
```
green = Linear(1, 0, -0.8)
```



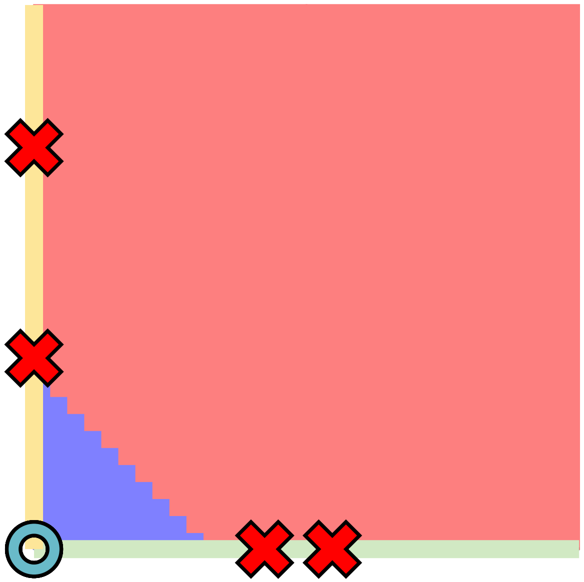
Math View

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

Reshape: ReLU



Reshape: ReLU

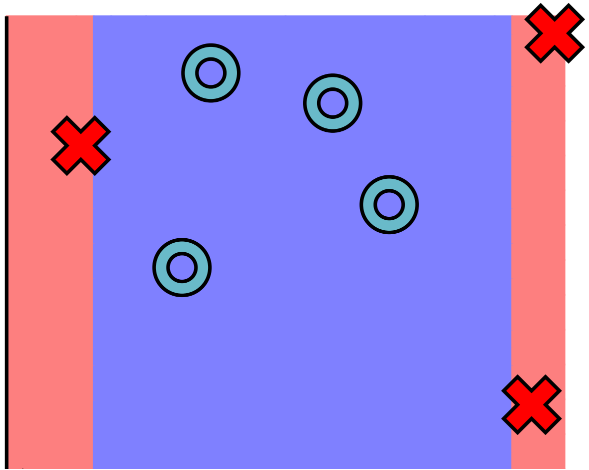


Final Layer

```python
@dataclass
class MLP:
    lin1: Linear
    lin2: Linear
    final: Linear

    def forward(self, x1, x2):
        x1_1 = minitorch.operators.relu(self.lin1.forward(x1, x2))
        x2_1 = minitorch.operators.relu(self.lin2.forward(x1, x2))
        return self.final.forward(x1_1, x2_1)
```

```python
mlp = MLP(green, yellow, Linear(3, 3, -0.3))
draw_with_hard_points(mlp)
```



Math View

$$h_1 = \mathrm{ReLU}(x_1 \times w_1^0 + x_2 \times w_2^0 + b^0)$$
$$h_2 = \mathrm{ReLU}(x_1 \times w_1^1 + x_2 \times w_2^1 + b^1)$$
$$m(x_1, x_2) = h_1 \times w_1 + h_2 \times w_2 + b$$

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)

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

Parameters: $w_1, w_2, w^0_1, w^0_2, w^1_1, w^1_2, b, b^0, b^1$

Code View

Linear

```
class Linear(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.va
```

Code View

Model

```python
class Network(minitorch.Module):
    def __init__(self):
        super().__init__()
        self.unit1 = Linear()
        self.unit2 = Linear()
        self.classify = Linear()

    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())
```

Derivatives

- All the parameters in model are leaf Variables

```
model = Network()
x1, x2 = Scalar(0.5), Scalar(0.5)
out = model.forward((0.5, 0.5))
loss = -(-out).sigmoid().log()
loss.backward()
```

Derivatives

- All the parameters in model are leaf Variables

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

Playground

NN Playground