Module 2.0 - Neural Networks

Loading [MathJax]/extensions/Safe.js

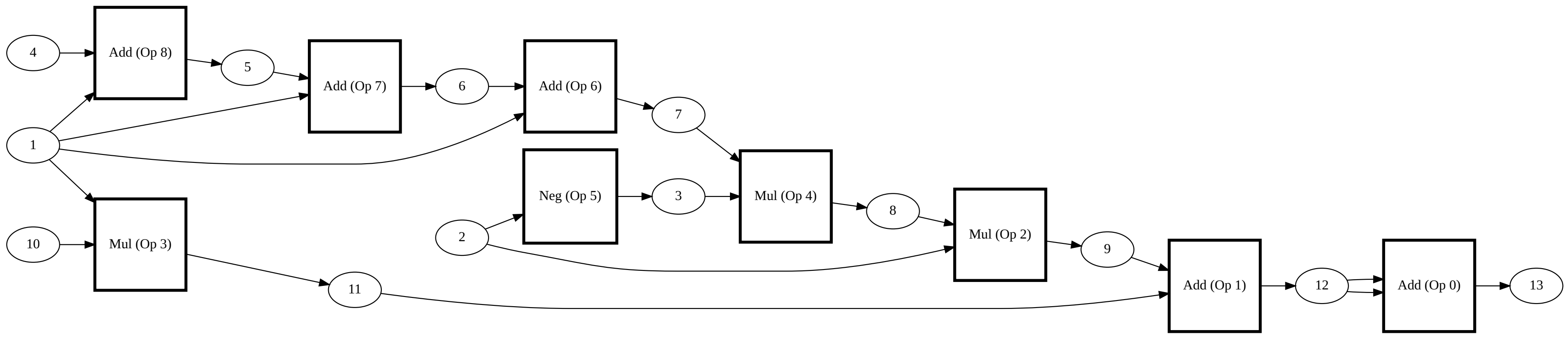
Our Goal

Compute derivative of Python function with respect to inputs.

Loading [MathJax]/extensions/Safe.js

Example: Function

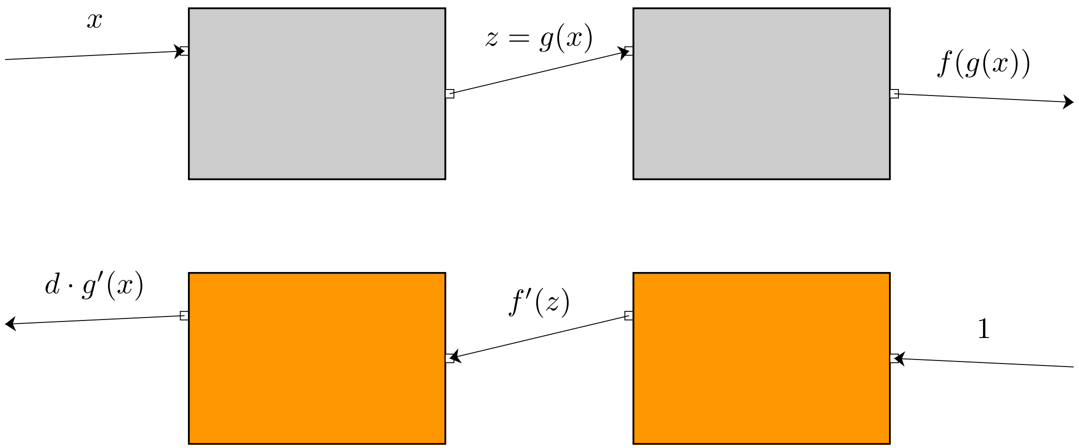
```python
def expression():
    x = Scalar(1.0)
    y = Scalar(1.0)
    z = -y * sum([x, x, x]) * y + 10.0 * x
    h_x_y = z + z
    return h_x_y
```



Loading [MathJax]/extensions/Safe.js

Chain Rule: Simple Case

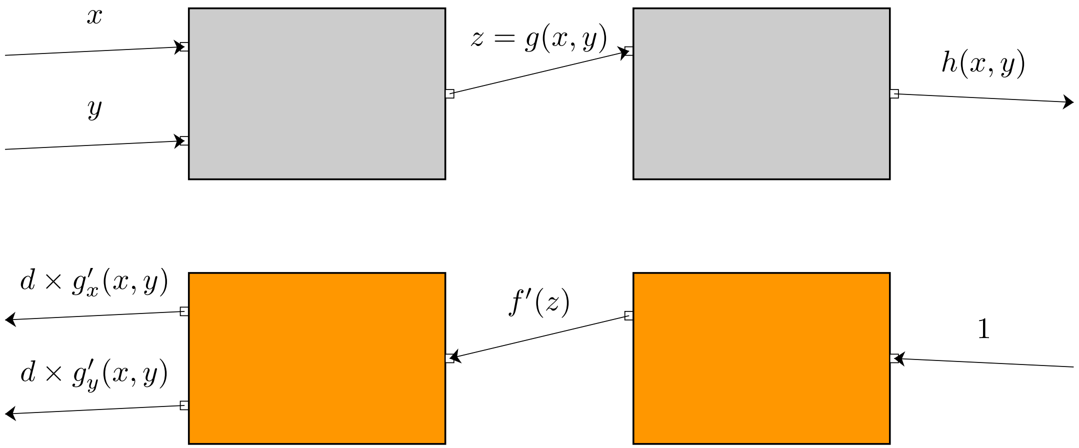
$$z = g(x)$$
$$d = f'(z)$$
$$f'_x(g(x)) = g'(x) \times d$$



Loading [MathJax]/extensions/Safe.js

Chain Rule: Two Arguments

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

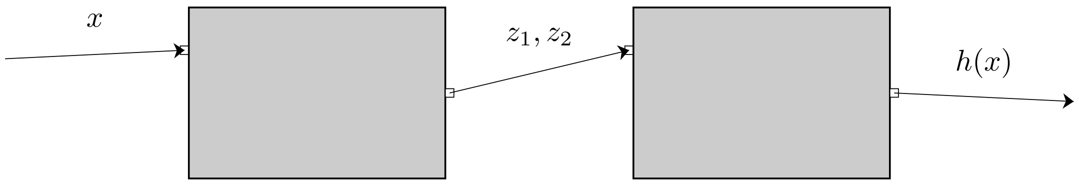


Loading [MathJax]/extensions/Safe.js

Chain Rule: Repeated Use

$$z = g(x)$$

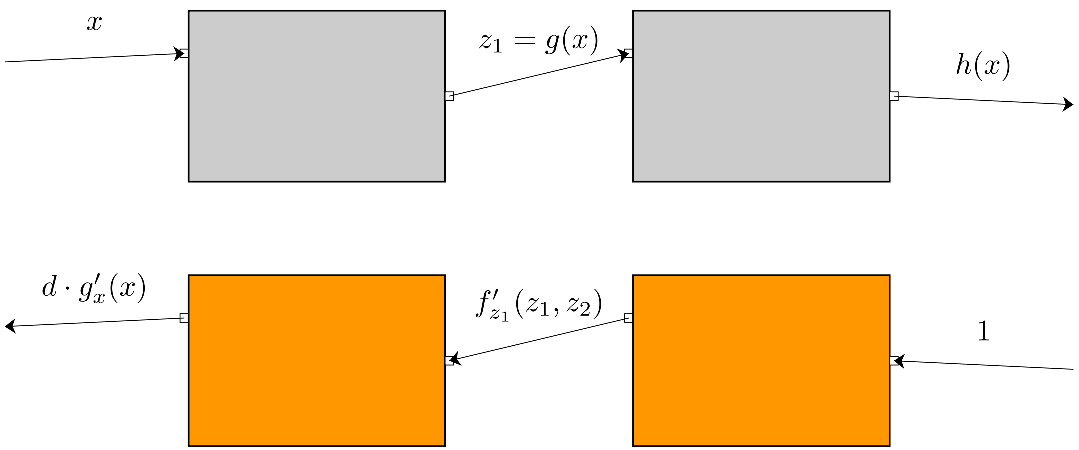
$$f(z, z)$$



Loading [MathJax]/extensions/Safe.js

Chain Rule: Repeated Use

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



Loading [MathJax]/extensions/Safe.js

Algorithm: Outer Loop

0. Call topological sort

1. Create dict of edges and empty d values.

2. For each edge and d in topological order:

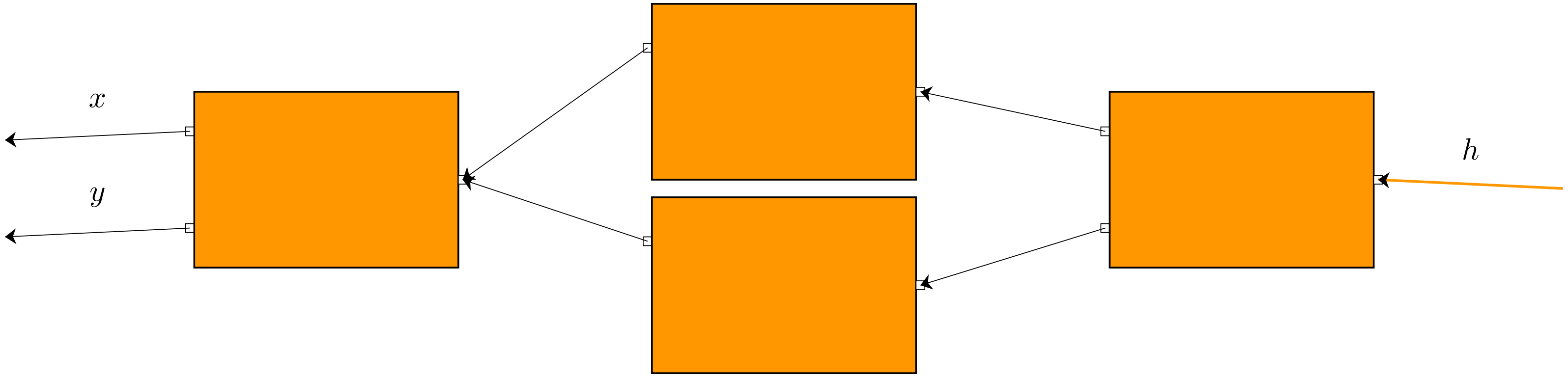
Loading [MathJax]/extensions/Safe.js

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

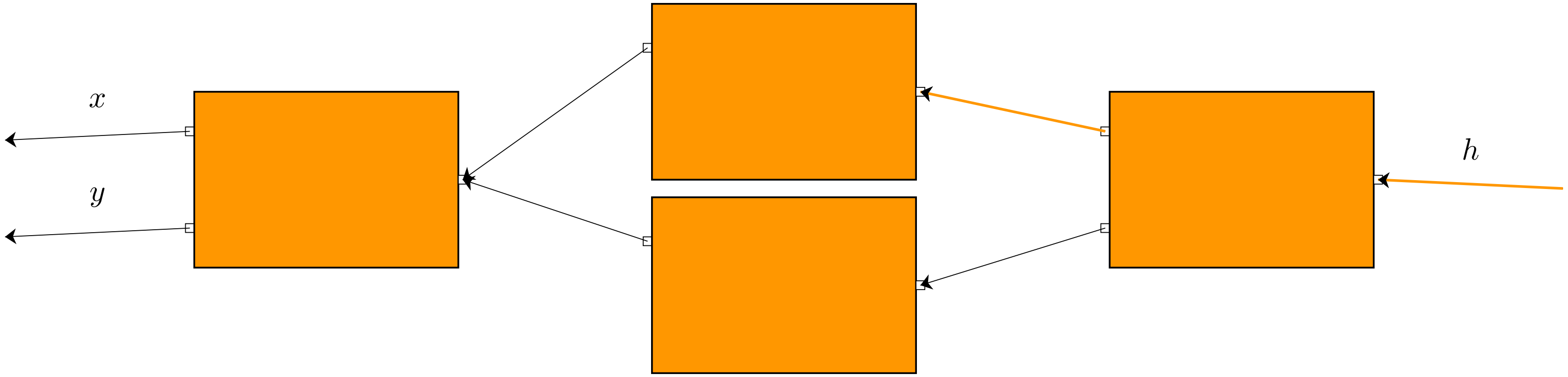
Loading [MathJax]/extensions/Safe.js

Example



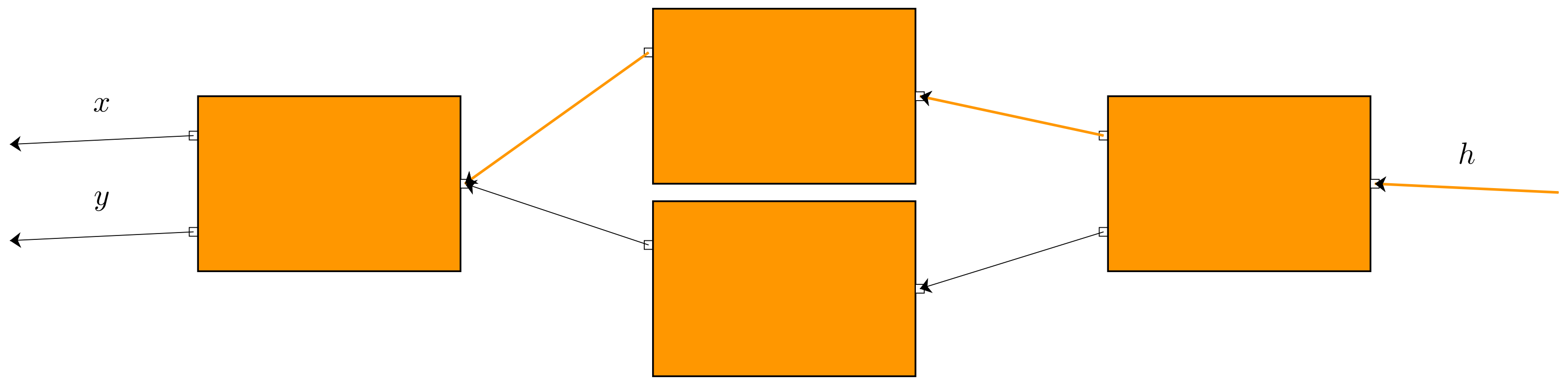
Loading [MathJax]/extensions/Safe.js

Example



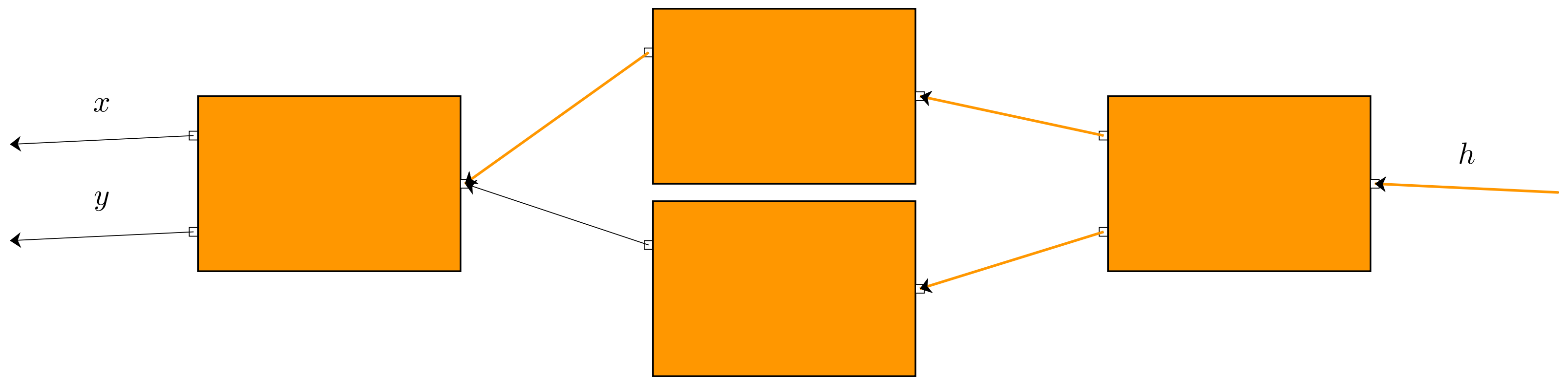
Loading [MathJax]/extensions/Safe.js

Example



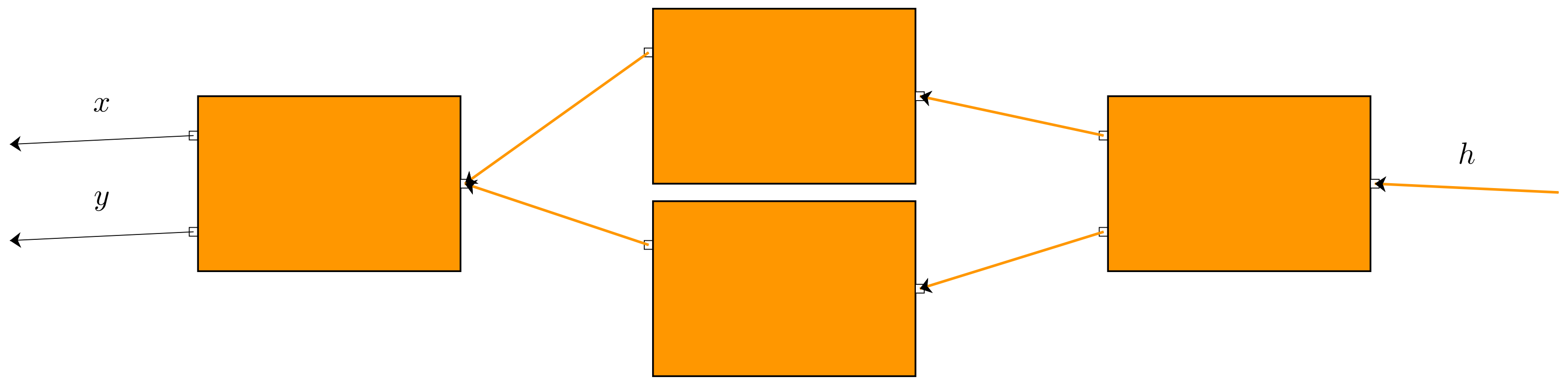
Loading [MathJax]/extensions/Safe.js

Example



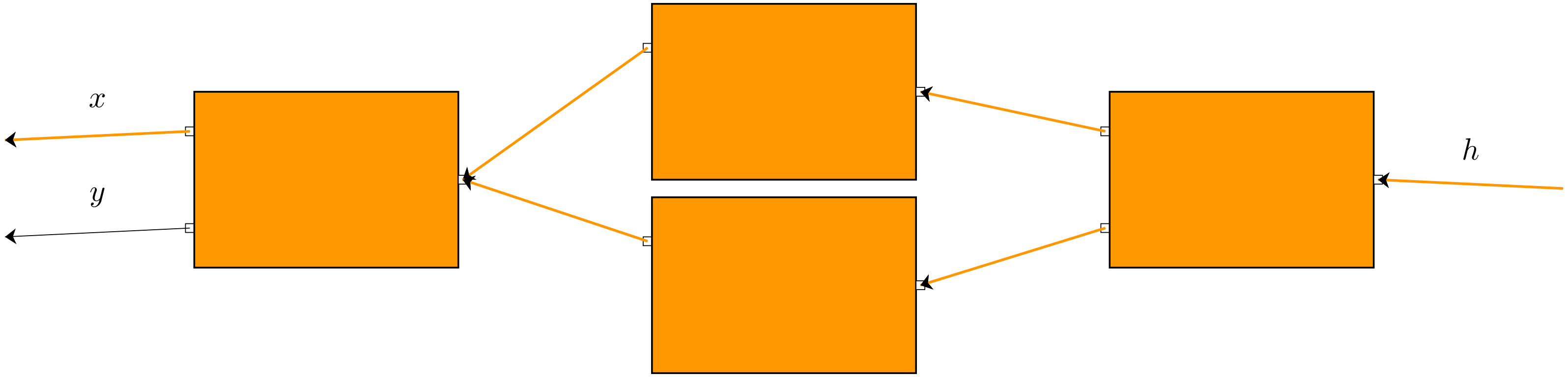
Loading [MathJax]/extensions/Safe.js

Example



Loading [MathJax]/extensions/Safe.js

Example



Loading [MathJax]/extensions/Safe.js

Example

Loading [MathJax]/extensions/Safe.js

Quiz

Loading [MathJax]/extensions/Safe.js

Outline

- Model Training
- Neural Networks
- Modern Models

Loading [MathJax]/extensions/Safe.js

Model Training

Loading [MathJax]/extensions/Safe.js

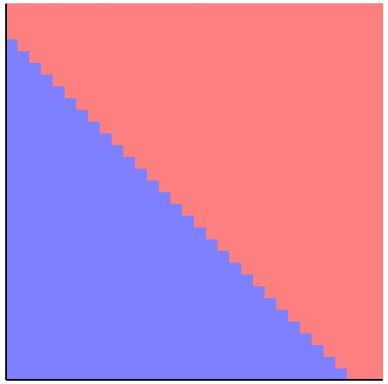
Reminder: MiniML

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

Loading [MathJax]/extensions/Safe.js

Model 1

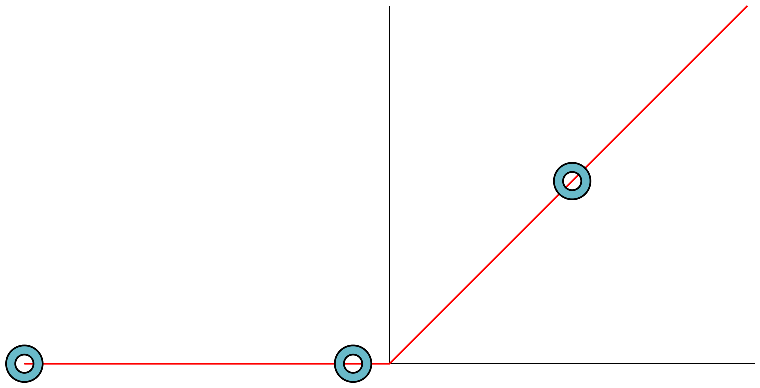
- Linear Model



Loading [MathJax]/extensions/Safe.js

Point Loss

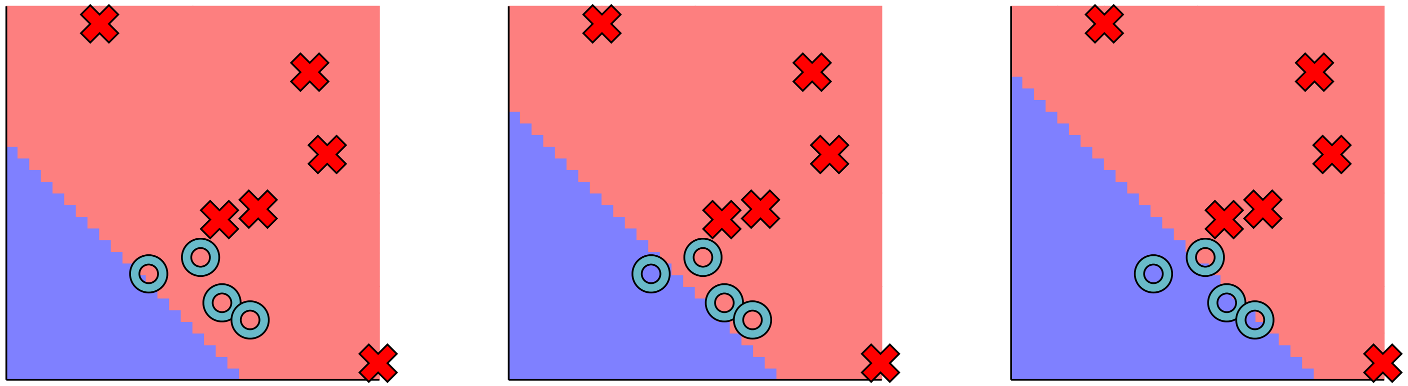
```
def point_loss(x):
    return minitorch.operators.relu(x)


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])
```



Loading [MathJax]/extensions/Safe.js

Class Goal

- Find parameters that minimize loss



Loading [MathJax]/extensions/Safe.js

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

Loading [MathJax]/extensions/Safe.js

Update Procedure

Loading [MathJax]/extensions/Safe.js

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
```

Loading [MathJax]/extensions/Safe.js

Training Loop

```python
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()
```

Loading [MathJax]/extensions/Safe.js

More Features: Linear Model

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

Loading [MathJax]/extensions/Safe.js

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))
```

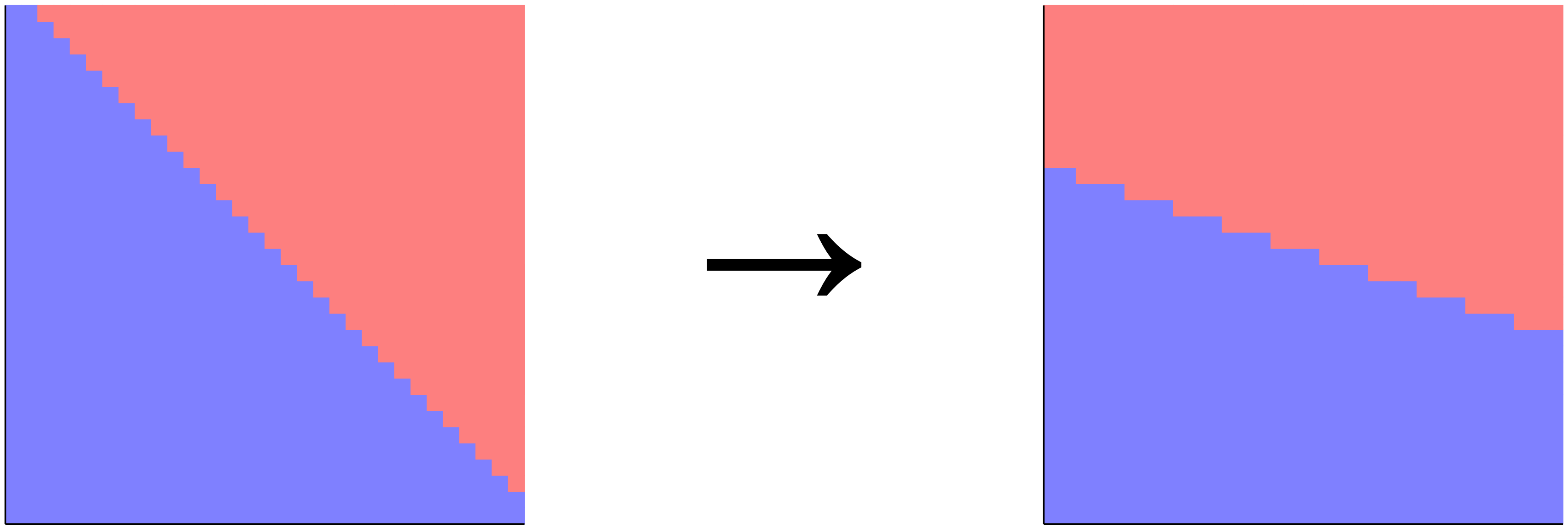
Loading [MathJax]/extensions/Safe.js

Neural Networks

Loading [MathJax]/extensions/Safe.js

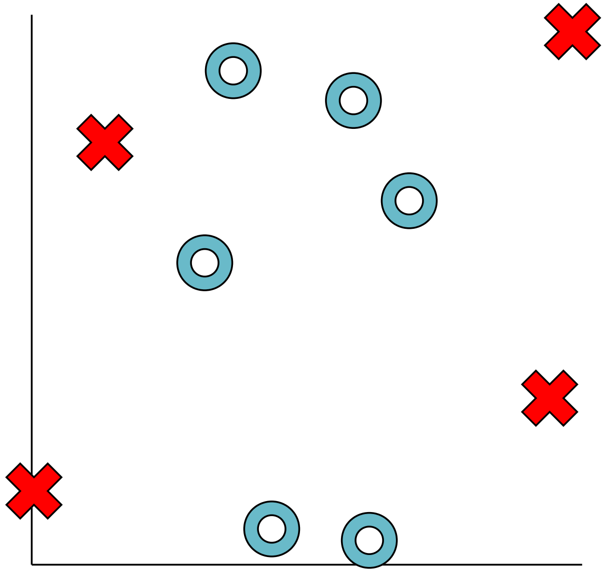
Linear Model Example

- Parameters



Loading [MathJax]/extensions/Safe.js

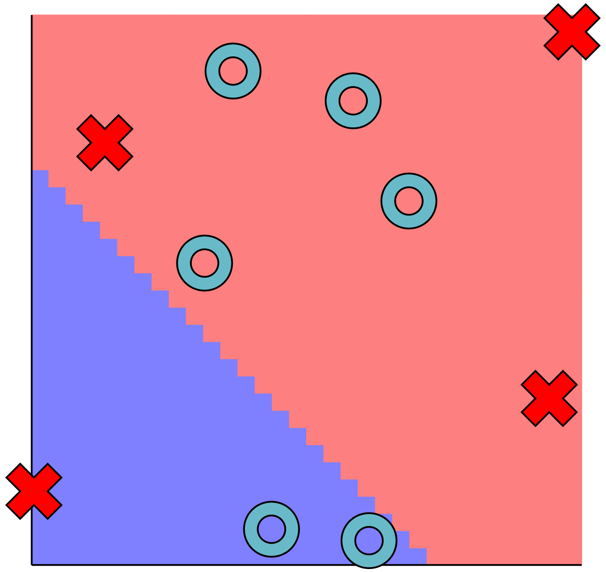
Harder Datasets



Loading [MathJax]/extensions/Safe.js

Harder Datasets

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



Loading [MathJax]/extensions/Safe.js

Neural Networks

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

Loading [MathJax]/extensions/Safe.js

Intuition: Neural Networks

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

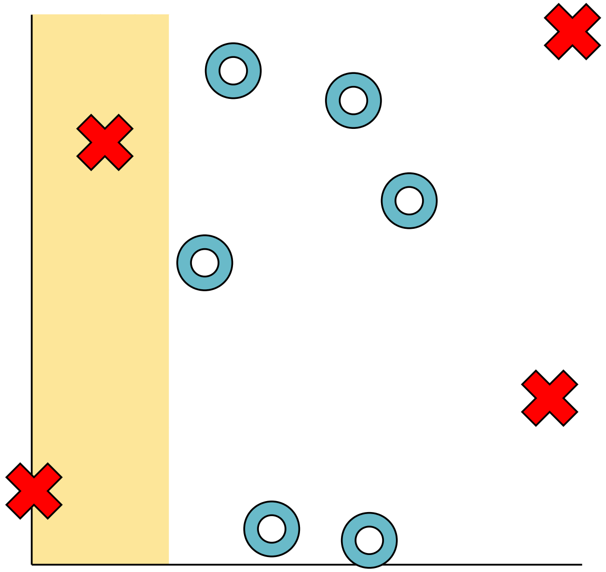
Loading [MathJax]/extensions/Safe.js

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.

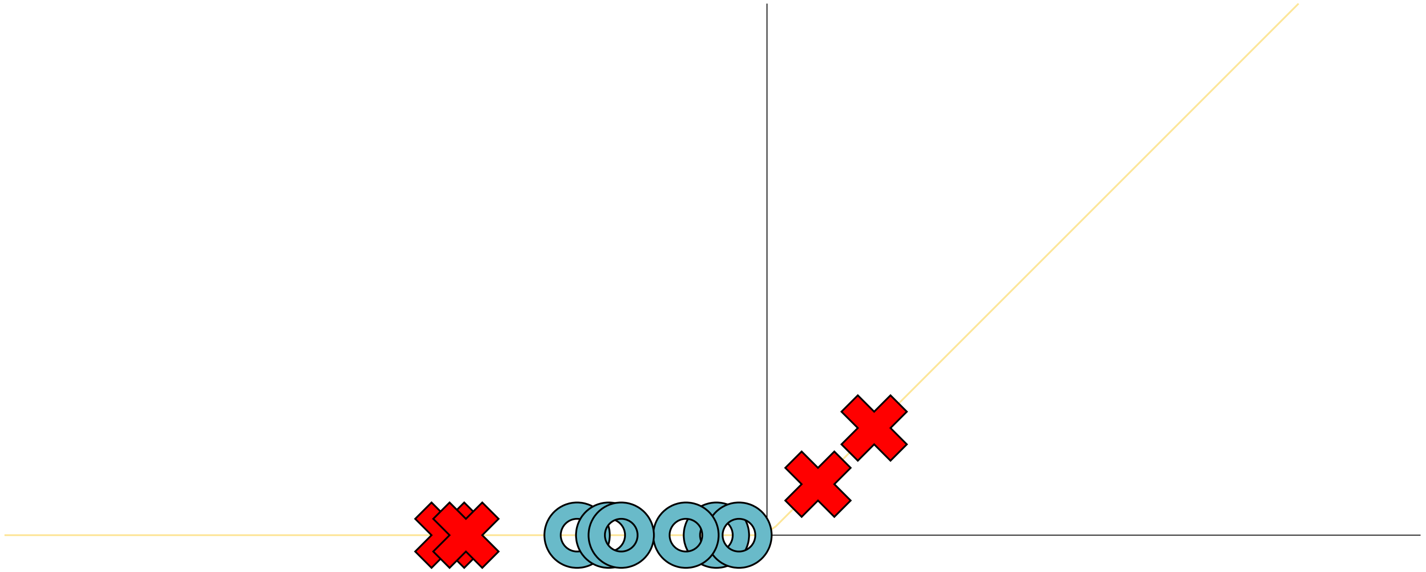
Loading [MathJax]/extensions/Safe.js

Intuition: Split 1



Loading [MathJax]/extensions/Safe.js

Reshape: ReLU



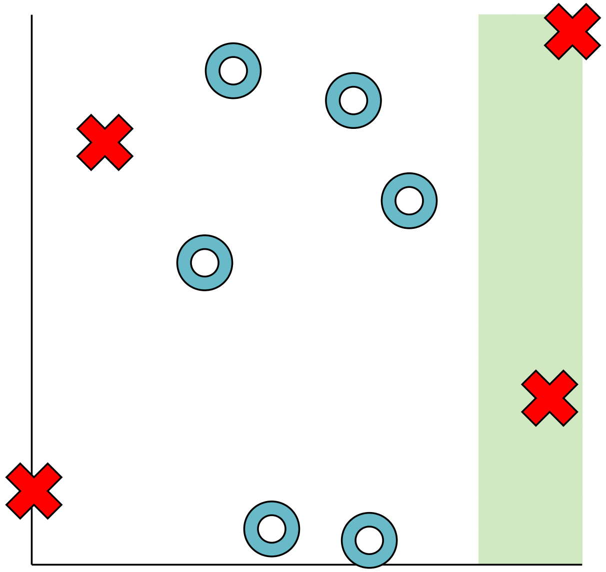
Loading [MathJax]/extensions/Safe.js

Math View

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

Loading [MathJax]/extensions/Safe.js

Intuition: Split 2



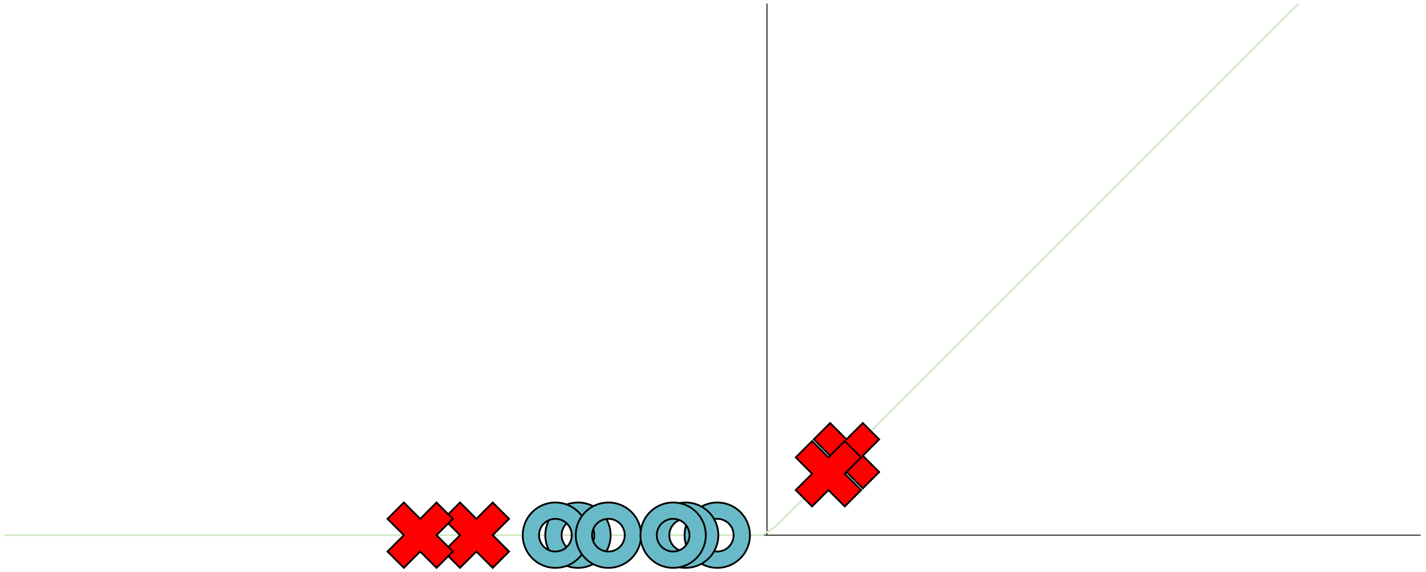
Loading [MathJax]/extensions/Safe.js

Math View

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

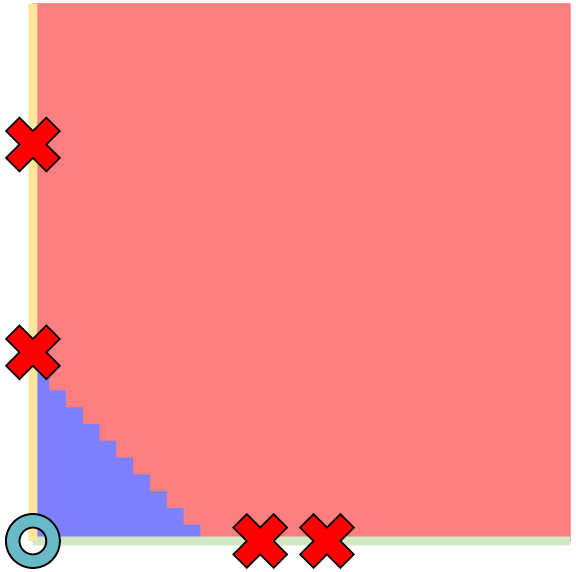
Loading [MathJax]/extensions/Safe.js

Reshape: ReLU



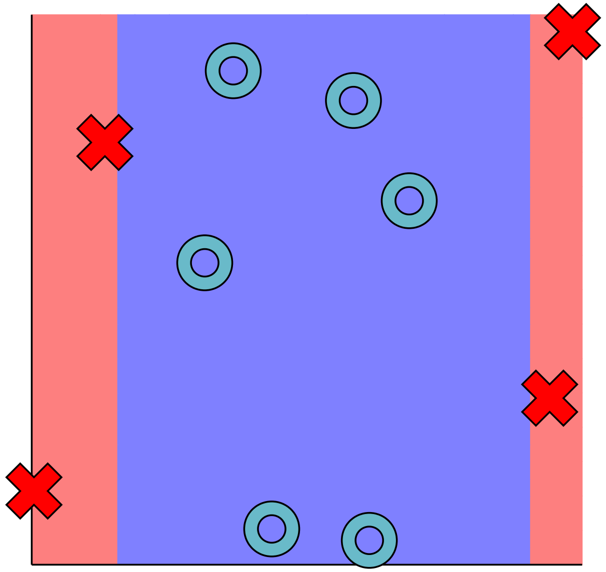
Loading [MathJax]/extensions/Safe.js

Reshape: ReLU



Loading [MathJax]/extensions/Safe.js

Final Layer



Loading [MathJax]/extensions/Safe.js

Math View

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

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

Loading [MathJax]/extensions/Safe.js

Math View (Alt)

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

Loading [MathJax]/extensions/Safe.js

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.va
```

Loading [MathJax]/extensions/Safe.js

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))
```

Loading [MathJax]/extensions/Safe.js

Training

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

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

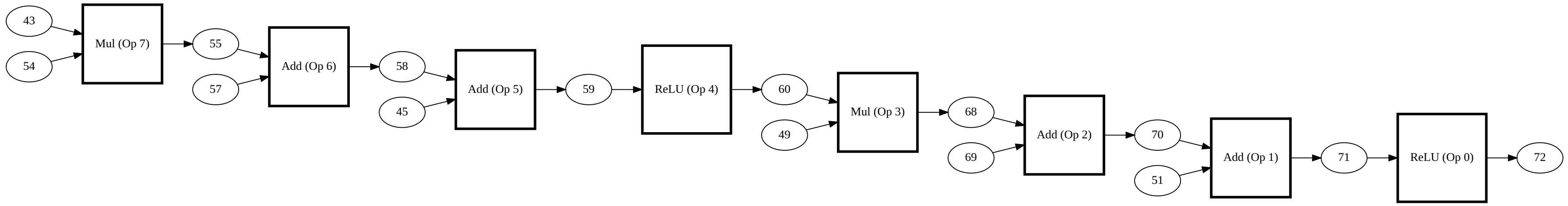
```
{'unit1.w_1': Scalar(0.0),
 'unit1.w_2': Scalar(0.0),
 'unit1.b': Scalar(0.0),
 'unit2.w_1': Scalar(0.0),
 'unit2.w_2': Scalar(0.0),
 'unit2.b': Scalar(0.0),
 'classify.w_1': Scalar(0.0),
 'classify.w_2': Scalar(0.0),
 'classify.b': Scalar(0.0)}
```

Loading [MathJax]/extensions/Safe.js

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))
```



Loading [MathJax]/extensions/Safe.js

Derivatives

- All the parameters in model are leaf scalars

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

Loading [MathJax]/extensions/Safe.js

Playground

NN Playground

Loading [MathJax]/extensions/Safe.js

QA

Loading [MathJax]/extensions/Safe.js