Module 1.0 - Mini-ML

Model

- Models: parameterized functions.
  - $m(x;\theta)$
  - x - input
  - m - model
- Initial Focus:
  - θ - parameters

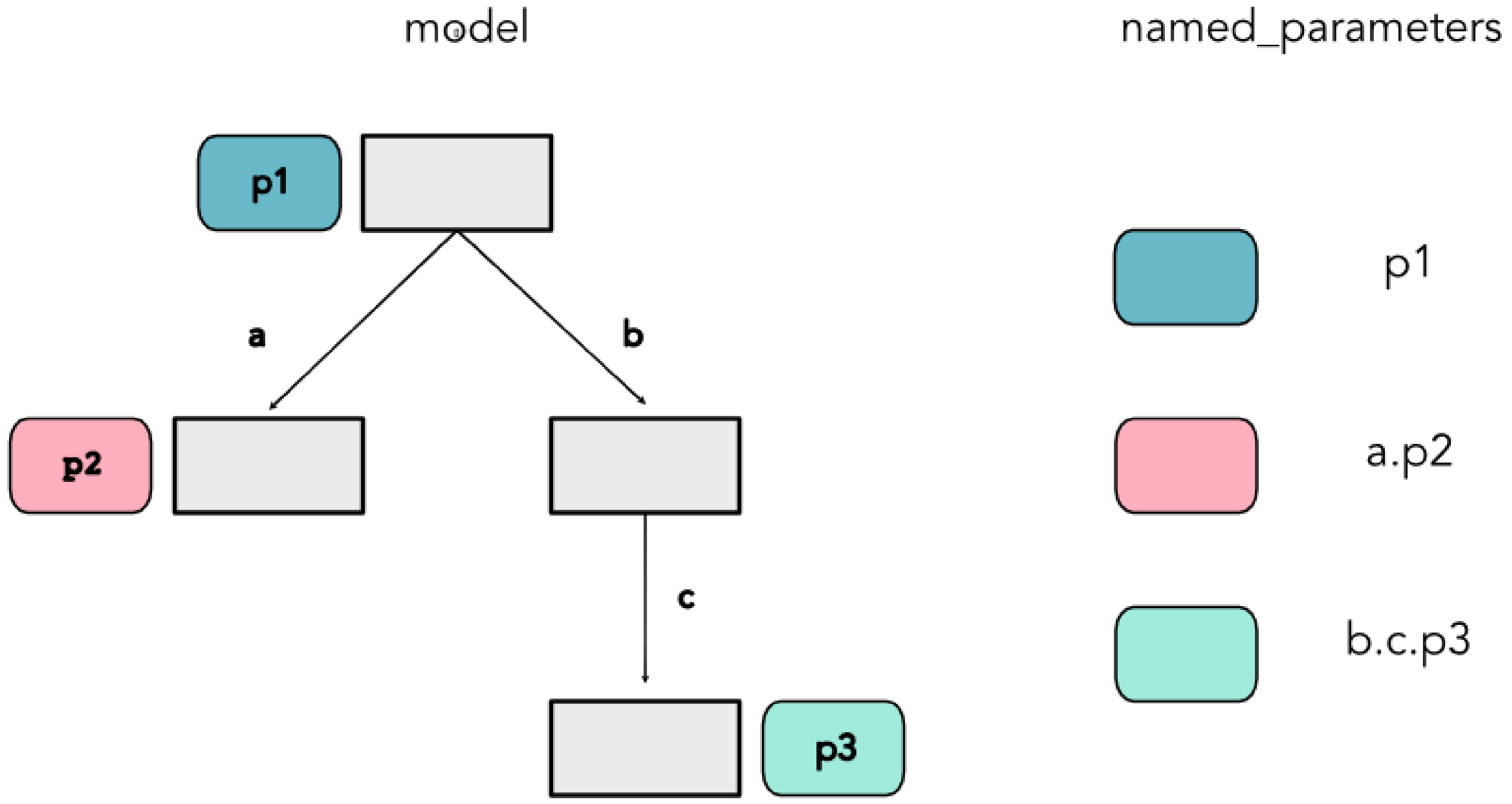
Specifying Parameters

- Datastructures to specify parameters
- Requirements
  - Independent of implementation
  - Compositional

Module Example

```python
from minitorch import Module, Parameter


class OtherModule(Module):
    pass


class MyModule(Module):
    def __init__(self):
        # Must initialize the super class!
        super().__init__()

        # Type 1, a parameter.
        self.parameter1 = Parameter(15)

        # Type 2, user data
        self.data = 25

        # Type 3. another Module
        self.sub_module = OtherModule()
```

Module Naming



Lecture Quiz

Quiz

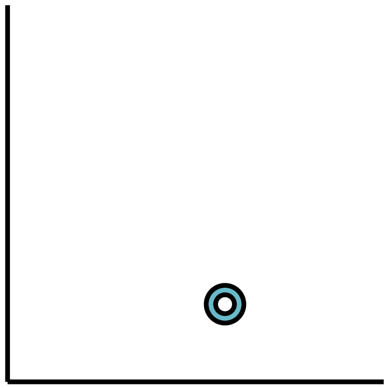
Outline

- Model
- Parameters
- Loss

Datasets

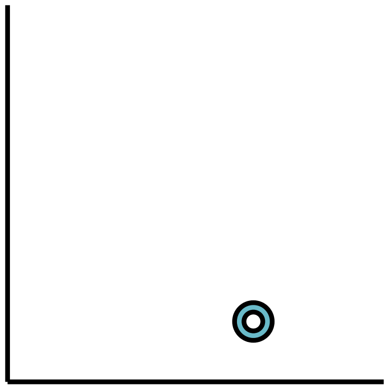
Data Points

- Convention x



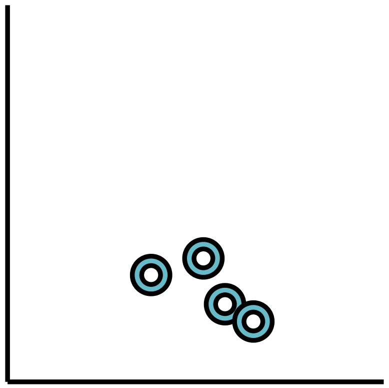
Data Points

- Convention x



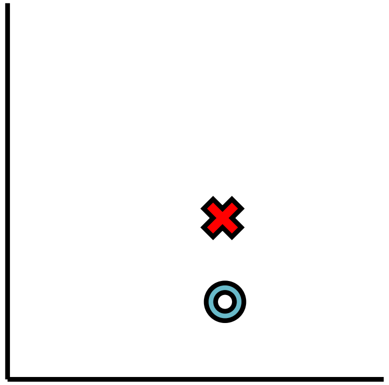
Data Points

- Convention x



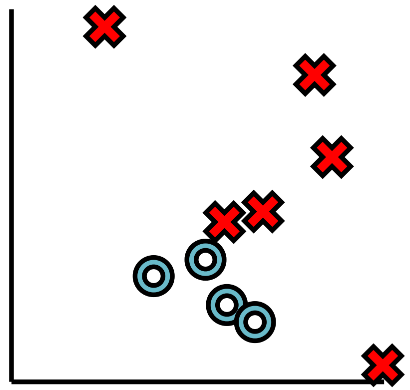
Data Labels

- Convention y



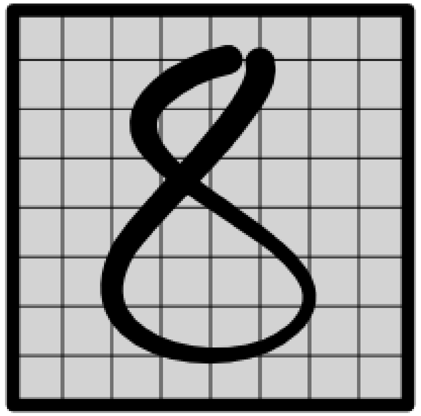
Training Data

- Set of datapoints, each (x, y)



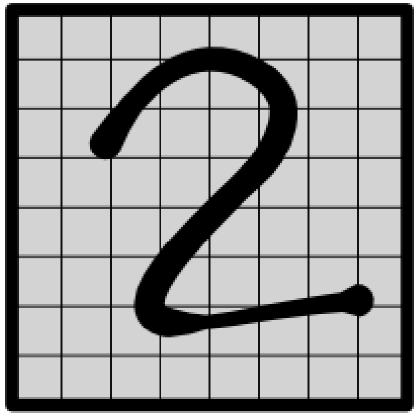
Data Points

- Convention x

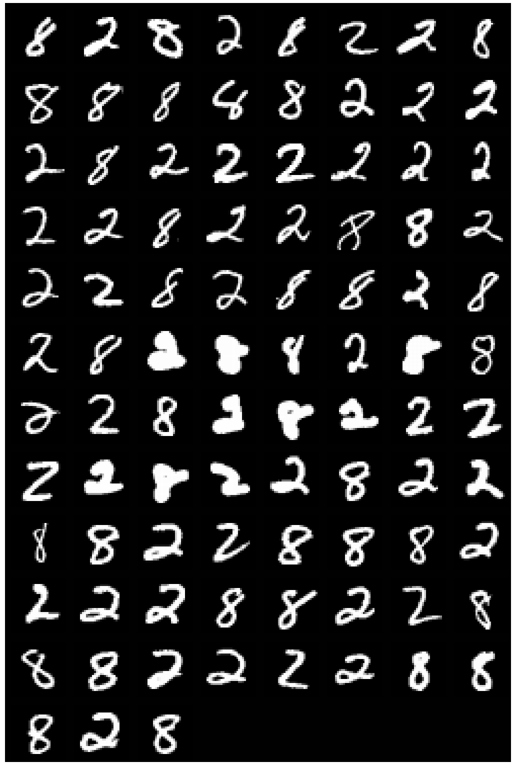


Data Points

- Convention x

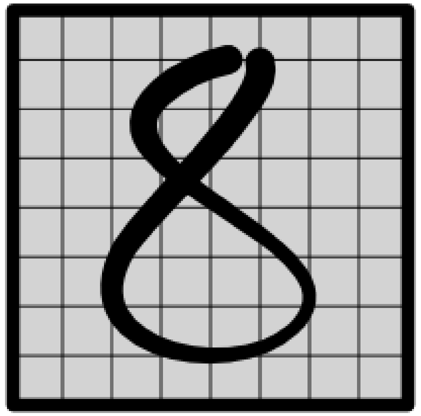


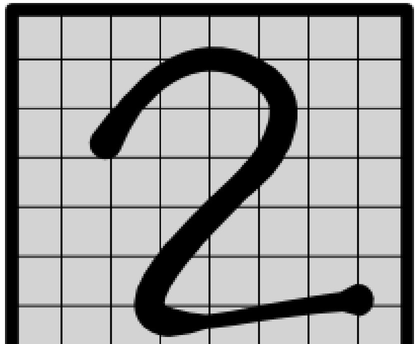
Data Set



Data Labels

- Convention y





Model

Models

- Functions from data points to labels
- Functions $m(x; \theta)$
- Any function is okay (e.g. Modules)

Example Model

- Example of a simple model

  $x = (0.5, 0.2)$

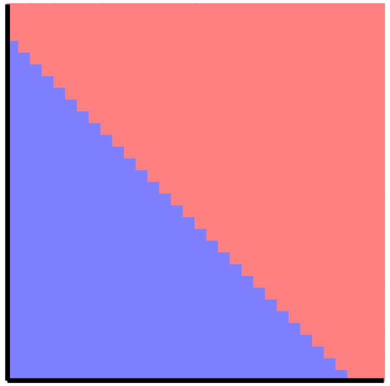
```python
@dataclass
class Model:
    def forward(self, x):
        return 0 if x[0] < 0.5 else 1
```

Model 1

- Linear Model

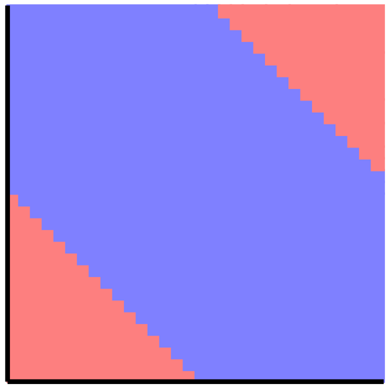
```python
from minitorch import Parameter, Module
class Linear(Module):
    def __init__(self, w1, w2, b):
        super().__init__()
        self.w1 = Parameter(w1)
        self.w2 = Parameter(w2)
        self.b = Parameter(b)

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

Model 1



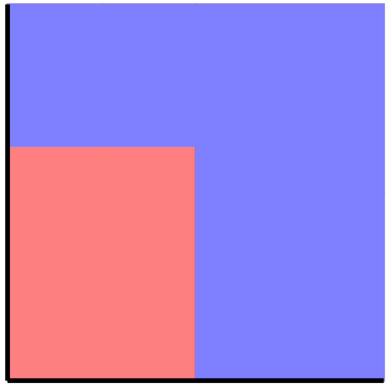
Model 2

```python
class Split:
    def __init__(self, linear1, linear2):
        super().__init__()
        # Submodules
        self.m1 = linear1
        self.m2 = linear2

    def forward(self, x1, x2):
        return self.m1.forward(x1, x2) * self.m2.forward(x1, x2)
```



Model 3

```
class Part:
    def forward(self, x1, x2):
        return 1 if (0.0 <= x1 < 0.5 and 0.0 <= x2 < 0.6) else 0
```



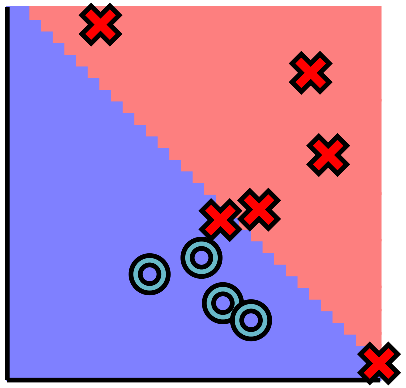
Parameters

Parameters

- Knobs that control the model
- Any information that controls the model shape

Parameters

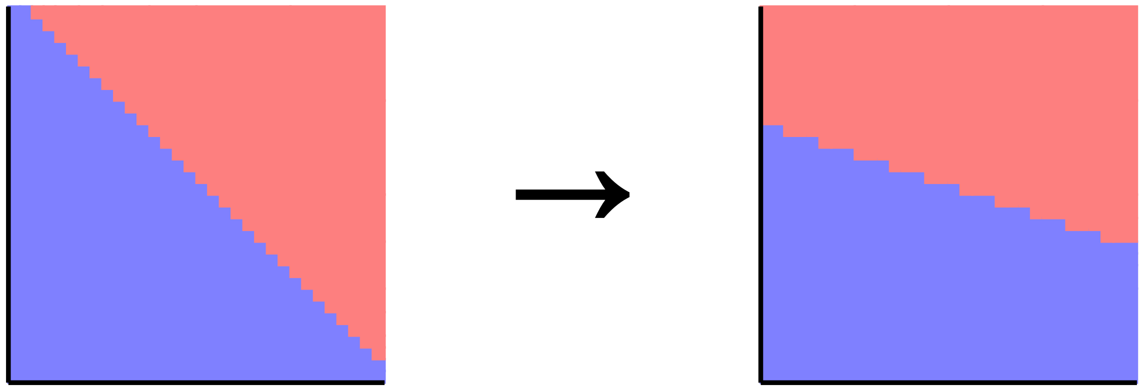
- Change θ



Linear Parameters

a. rotating the linear separator

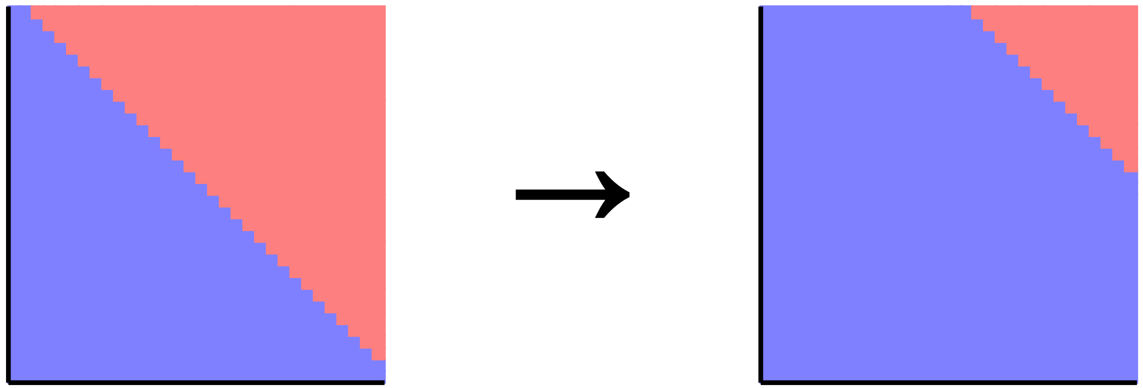
```
model1 = Linear(w1=1, w2=1, b=-1.0)
model2 = Linear(w1=0.5, w2=1.5, b=-1.0)
```



Linear Parameters

b. changing the separator cutoff

```
model1 = Linear(w1=1, w2=1, b=-1.0)
model2 = Linear(w1=1, w2=1, b=-1.5)
```



Math

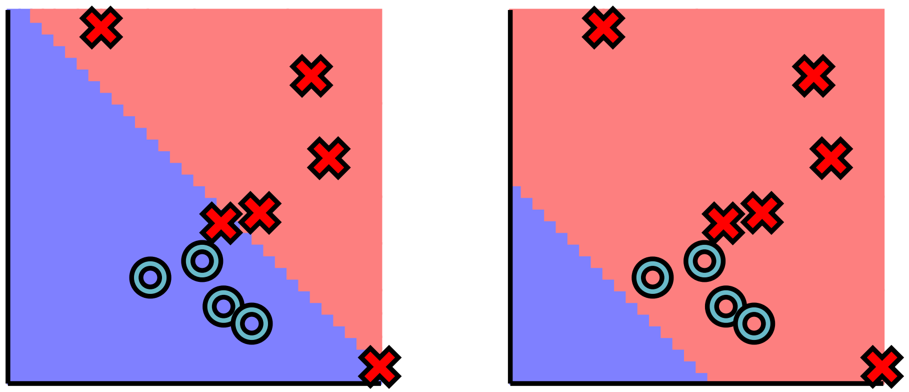
- Linear Model

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

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

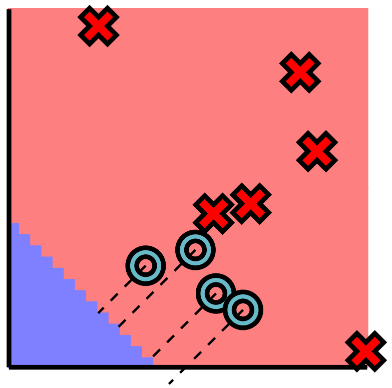
Loss

What is a good model?

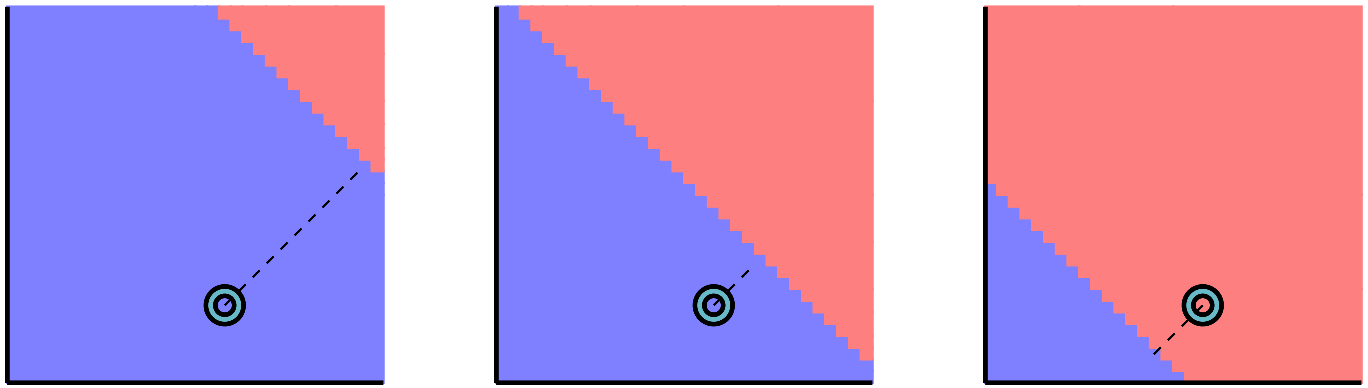


Distance

- $|m(x)|$ correct or incorrect



Points



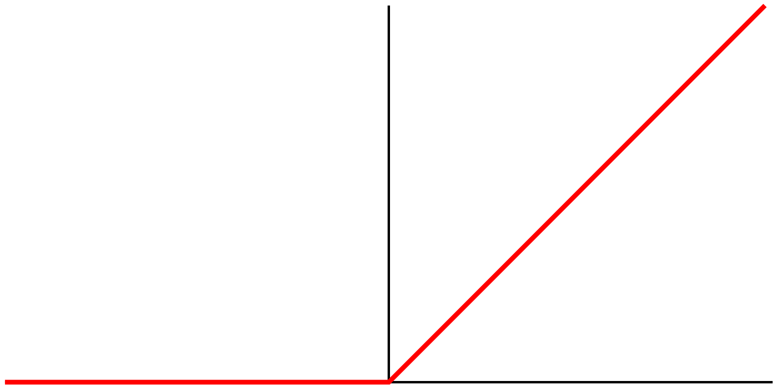
Loss

- Loss weights our incorrect points
- Uses distance from boundary

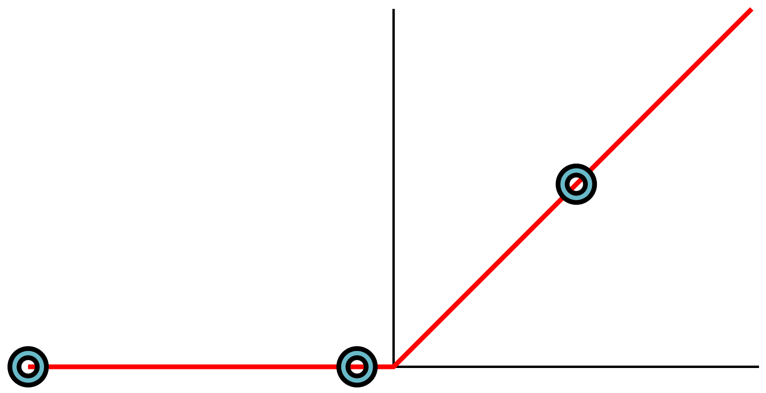
$L(w_1, w_2, b)$ is loss, function of parameters.

Warmup: ReLU

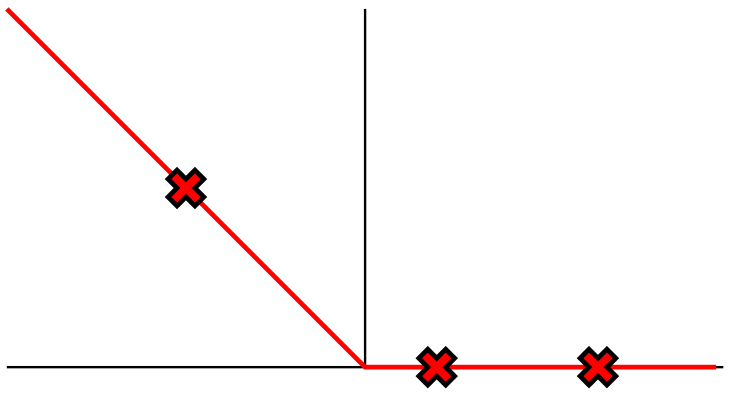
```
def point_loss(m_x):
    return minitorch.operators.relu(m_x)
```



Loss of points



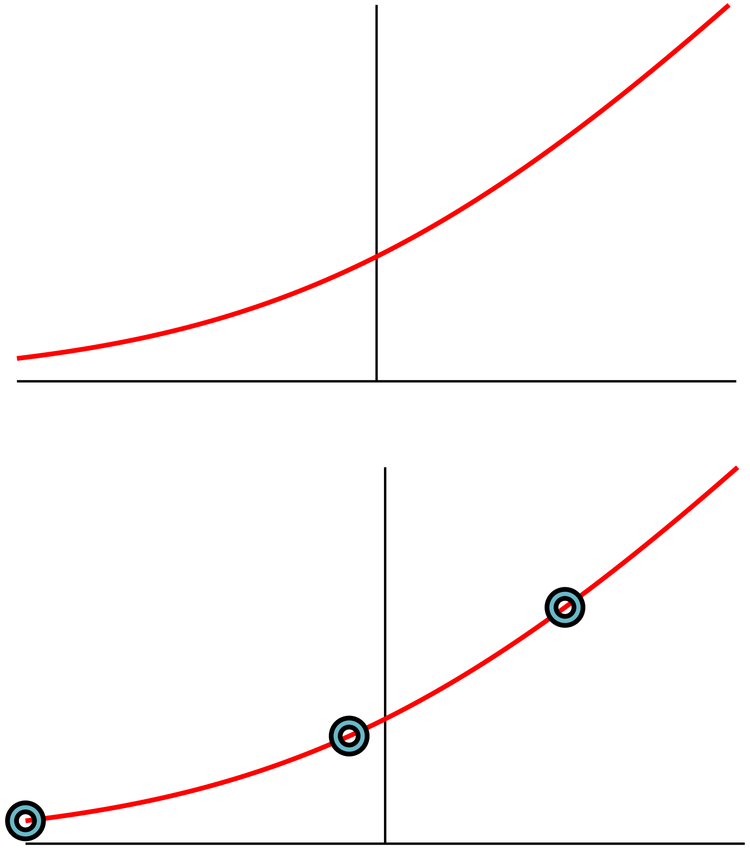
Loss of points



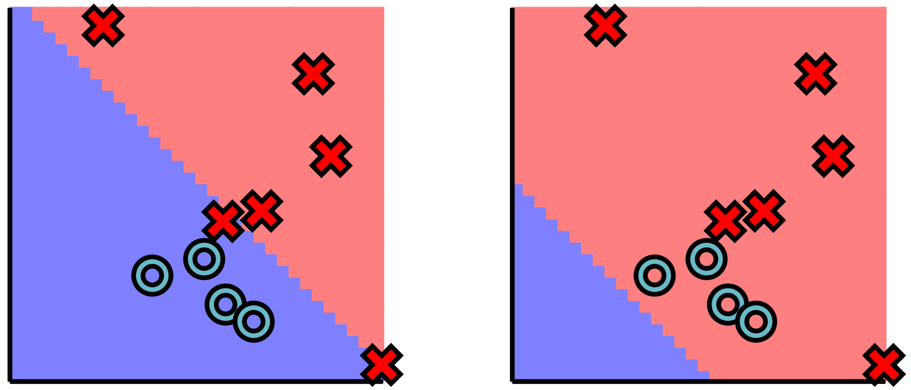
Full Loss

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

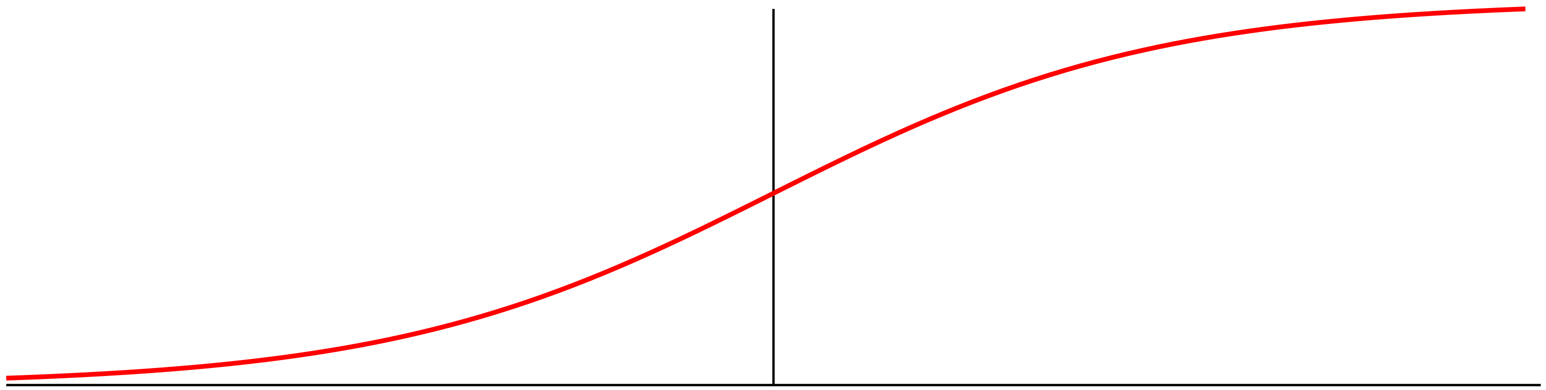
Point Loss



What is a good model?



Sigmoid Function



Playground

Playground

Q&A