Backpropagation

TA: Zane Durante

CS 231n April 14, 2023

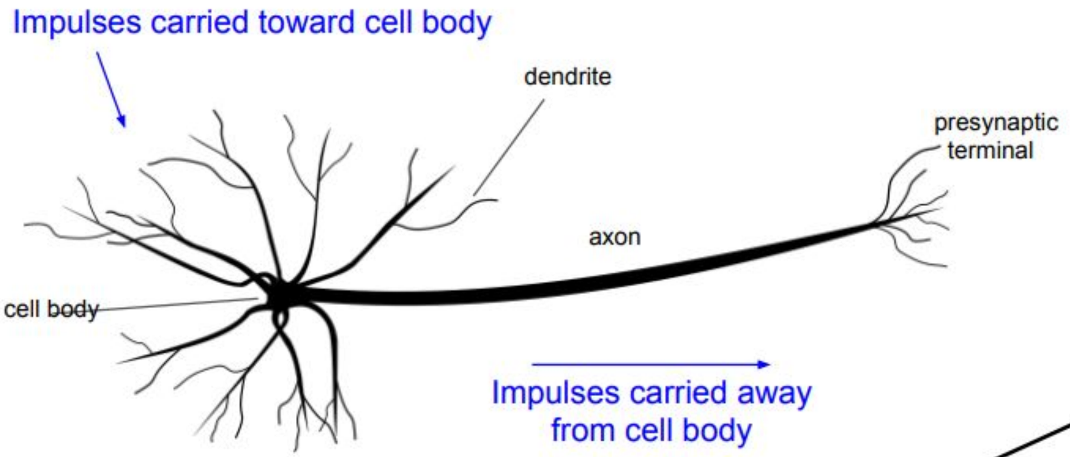
Some slides taken from lecture, credit to: Fei-Fei Li, Yunzhu Li, Ruohan Gao

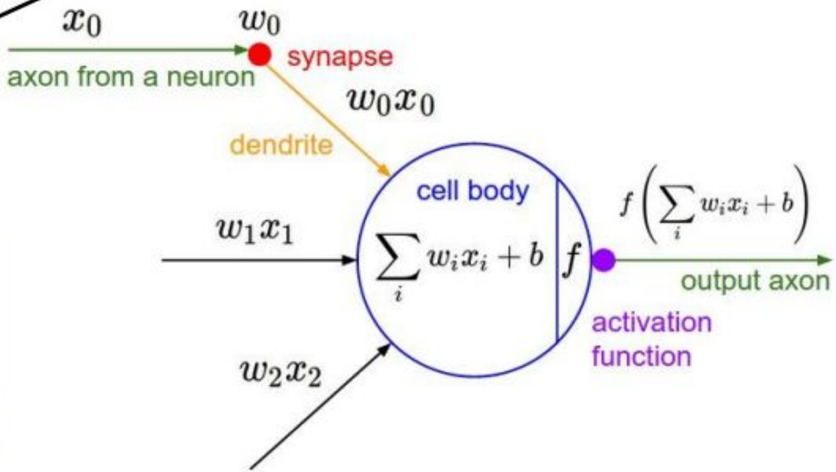
Agenda

- Quick review from lecture
  - Neural Networks
  - Motivation for backprop
- Goal: Deepen your understanding of backprop
  - Math
  - Computation graph
  - Code

Review

Biological Motivation



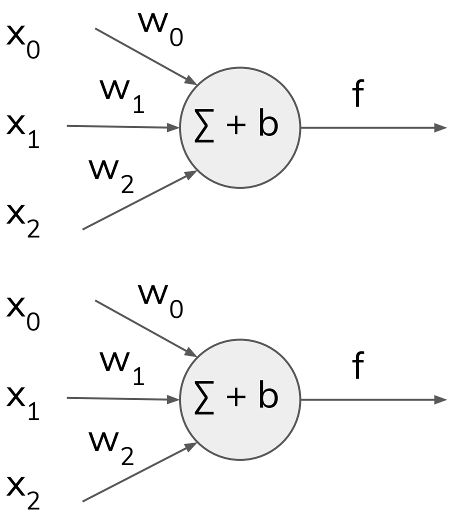



```python
class Neuron:
  # ...
  def neuron_tick(inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation func
    return firing_rate
```

In practice

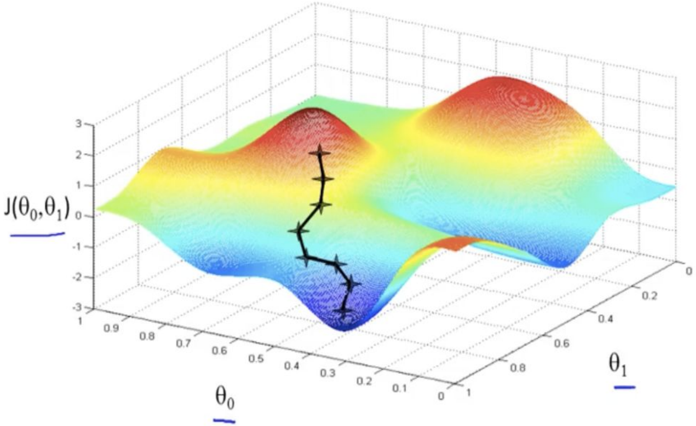
- We use matrix operations instead of computing each neuron separately

$$x \in R^3, W \in R^{3x2}, b \in R^2$$
$$\rightarrow f(W^T x + b) \in R^2$$



Motivation

- Gradient descent is a general method for optimizing parameters of a function
    - Goal: Minimize some loss (cost) function
- Update parameters with the gradient
    1. Calculate gradient of loss $\nabla_\theta J$ wrt parameter
    2. Update parameters with learning rate α
        - $\theta \mathrel{-}= \alpha \nabla_\theta J$
    3. Repeat 1-2 until done training



Credit: zitaoshen.rbind.io/project/optimization/1-min-of-machine-learning-gradient-decent/

Math Review

- Chain rule from calculus
- Neural networks contain a LONG string of operations
  - Backprop ← → Applying chain rule over and over again

$$\frac{d}{dx}\left[\left(f(x)\right)^n\right] = n\left(f(x)\right)^{n-1} \cdot f'(x)$$

$$\frac{d}{dx}\left[f\left(g(x)\right)\right] = f'\left(g(x)\right)g'(x)$$

Calcworkshop.com

Math Review

- Chain rule from calculus
- Neural networks contain a LONG string of operations
  - Backprop ← → Applying chain rule over and over again

$$\frac{d}{dx}\left[\left(f(x)\right)^{n}\right] = n\left(f(x)\right)^{n-1} \cdot f'(x)$$

$$\frac{d}{dx}\left[f\left(g(x)\right)\right] = f'\left(g(x)\right)g'(x)$$

Calcworkshop.com

Fraction notation (can "cancel" terms to simplify)

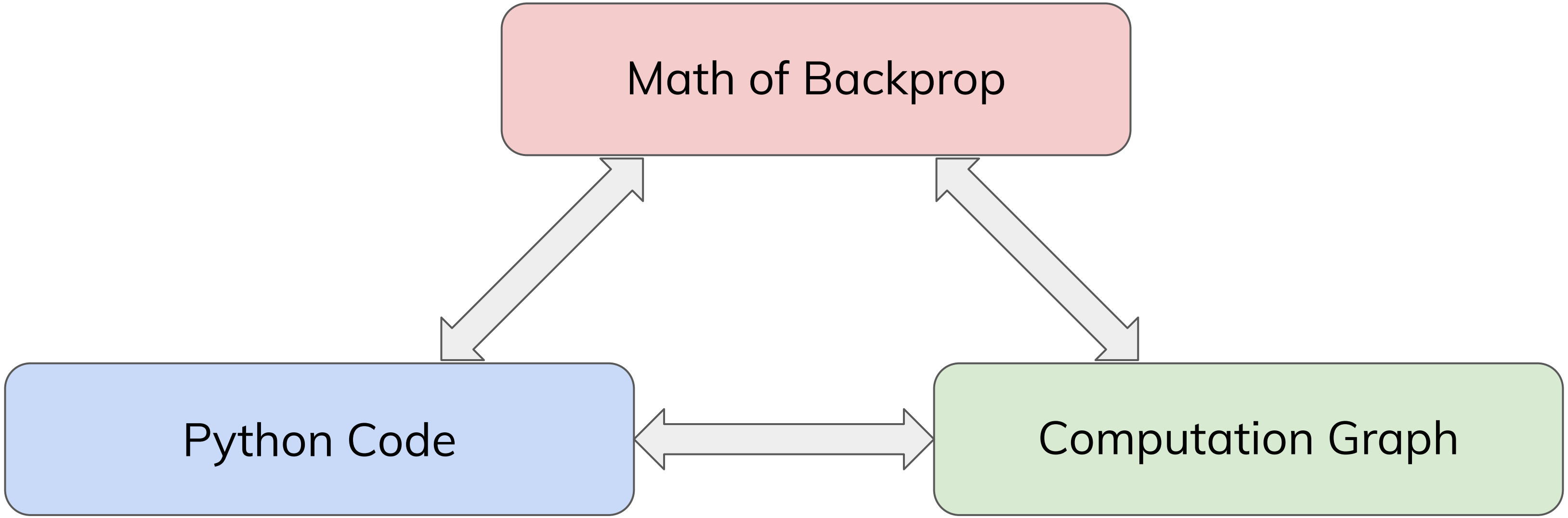
df/dx = df/dg * dg/dx

For vector-valued functions, chain rule goes right to left (only way dimensions match). We use this order for backprop

df/dx = dg/dx * df/dg

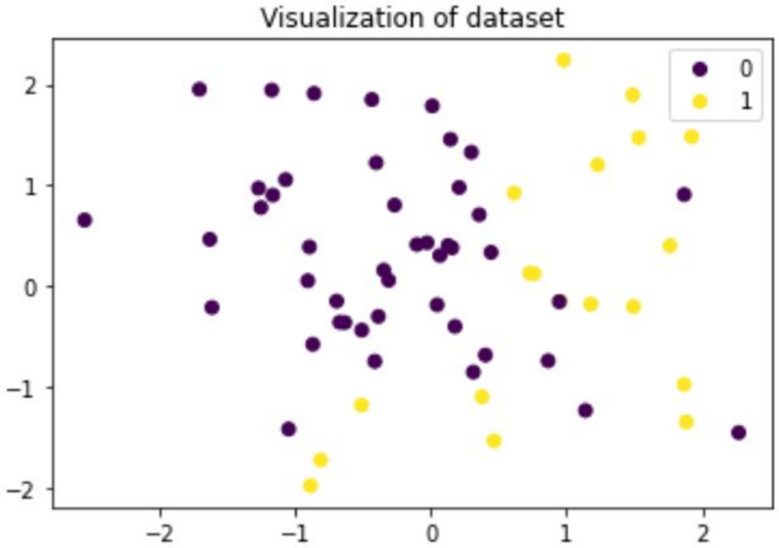
Understanding Backprop

Goal of this section

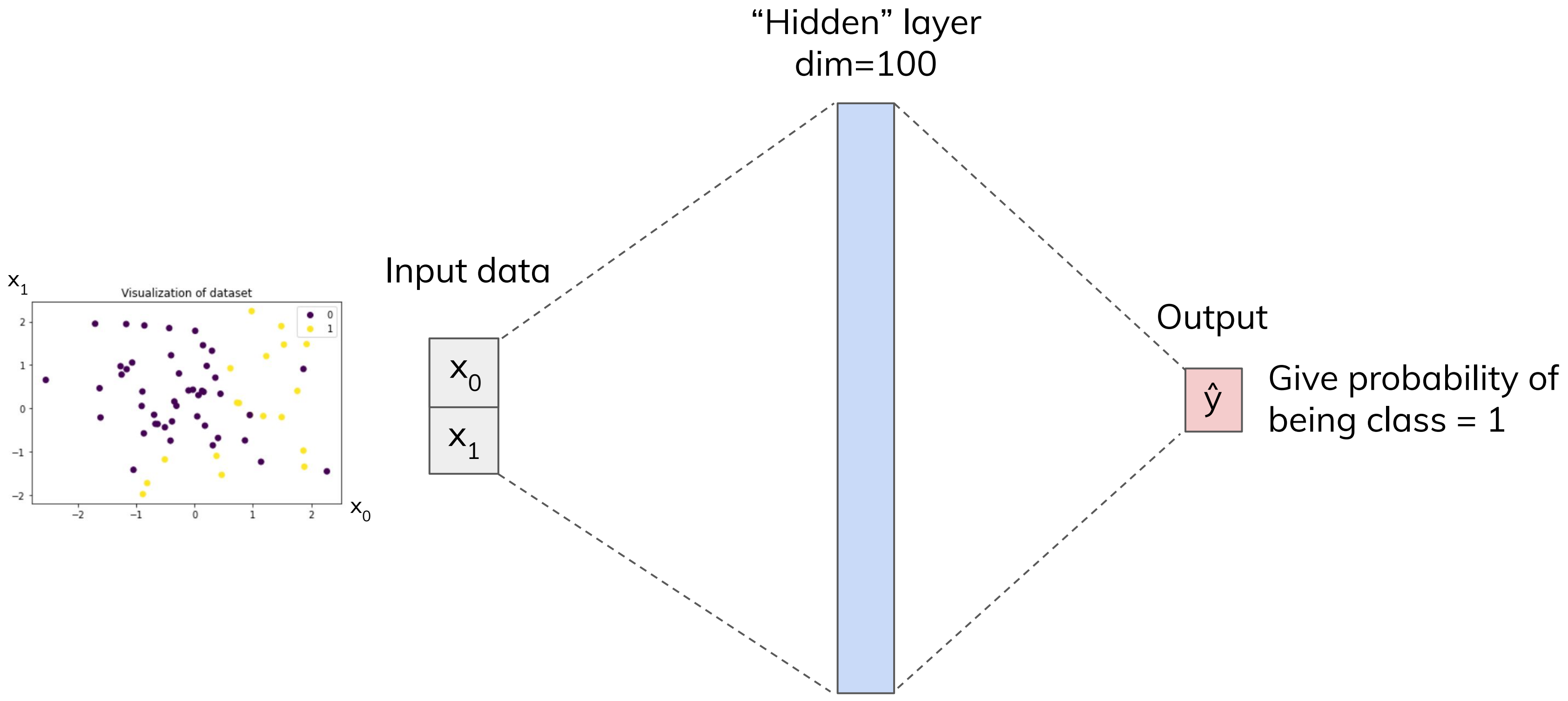


A Simple Use-Case:

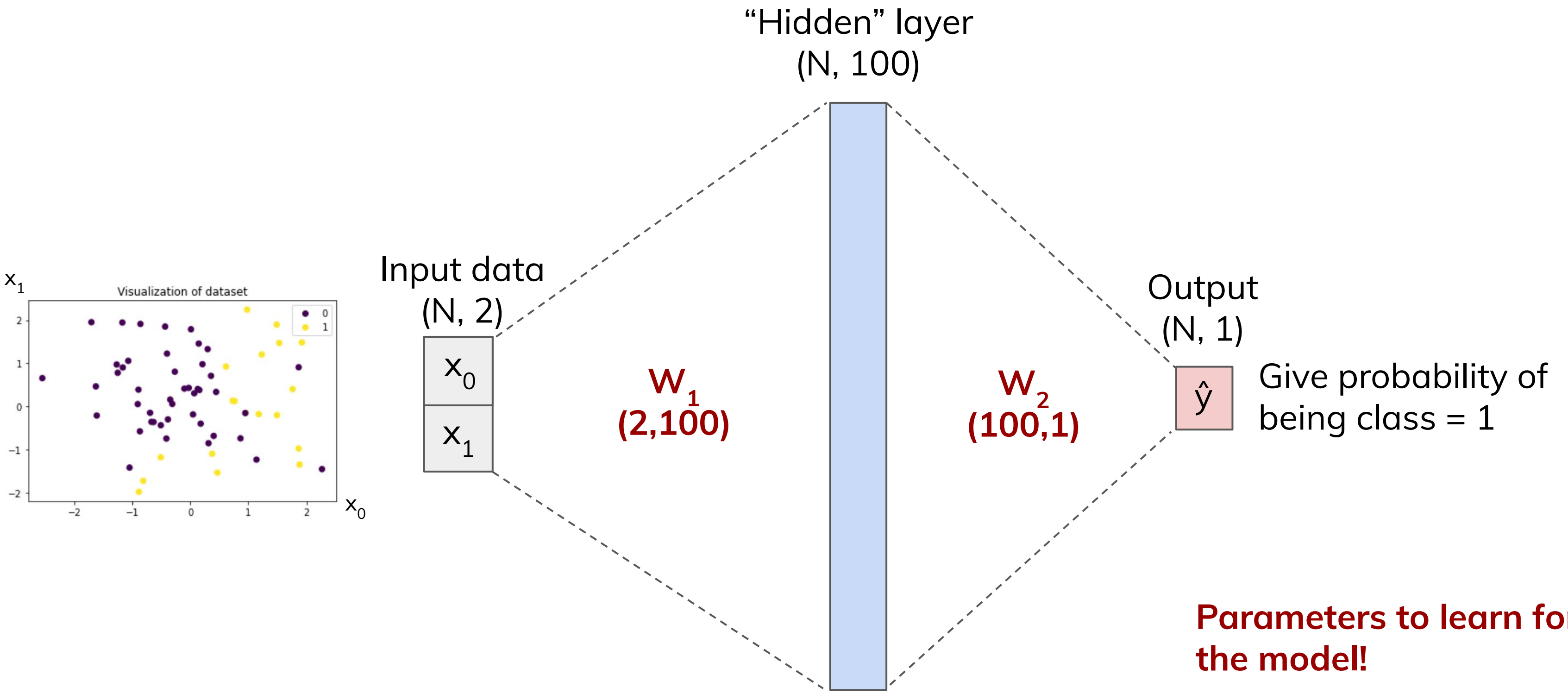
- Train a Neural Network classifier on 2D input data
  1. Describe model
  2. Math
  3. Computation graph
  4. Code



Simple 2-layer Neural Network



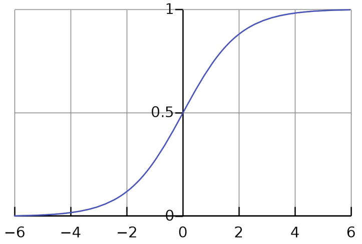
Parameters (weights) of the model

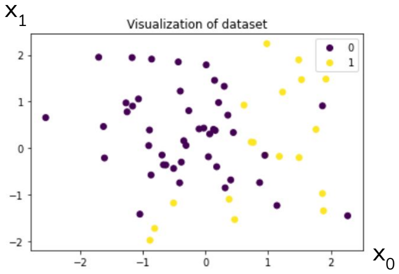


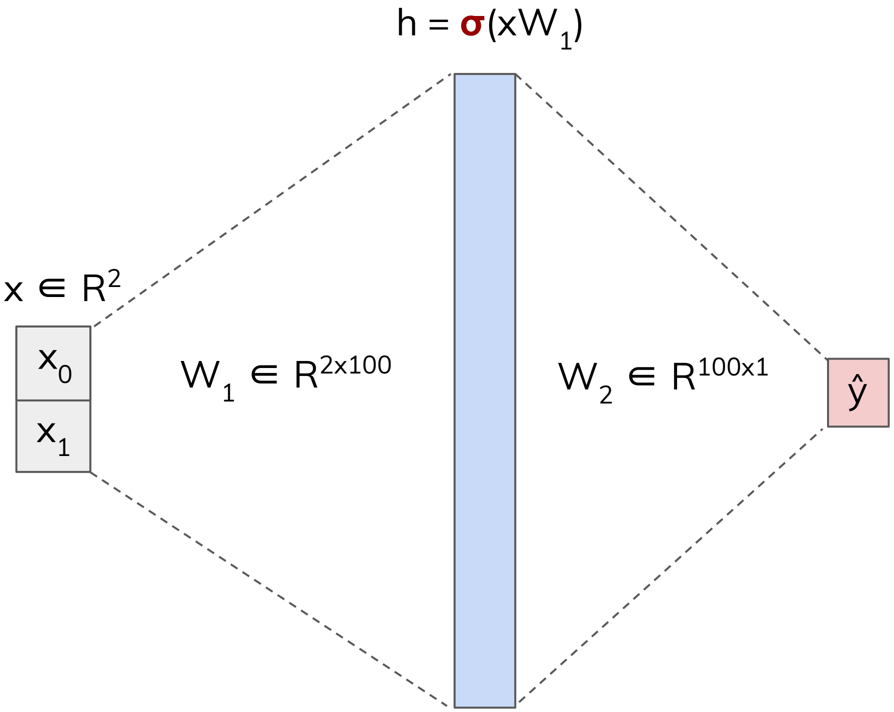
Parameters to learn for the model!

More rigorously:

Sigmoid function:
$\sigma(z) = 1/(1+e^{-z})$

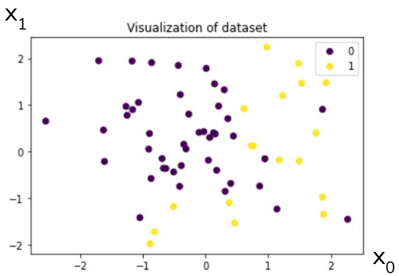


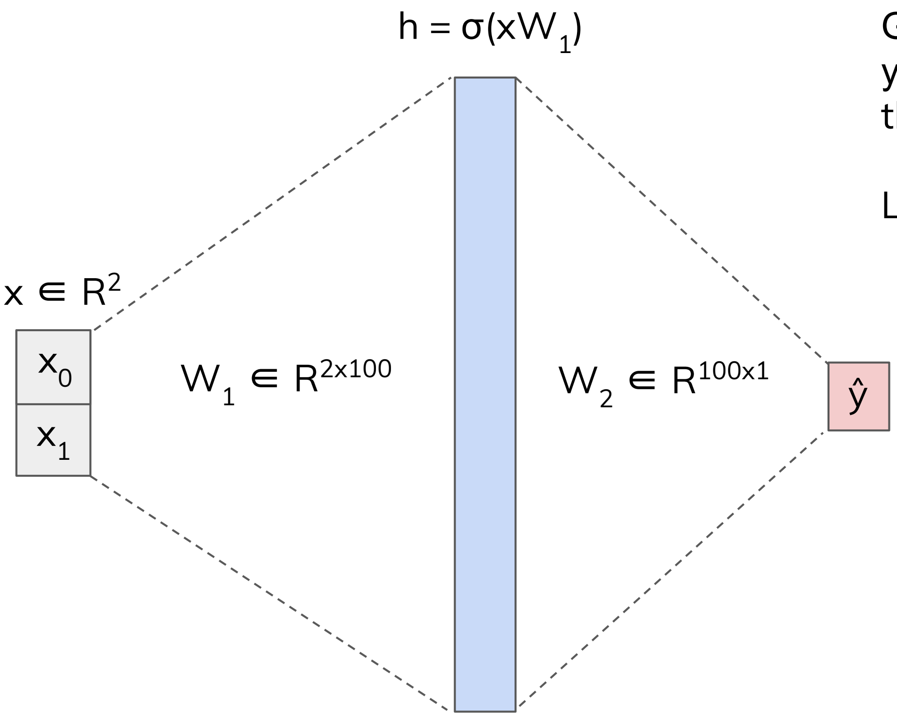




$\hat{y} = \sigma(hW_2)$

Goal: Minimize cross-entropy loss





$\hat{y} = \sigma(hW_2)$

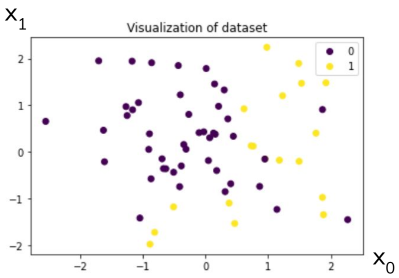
Given a ground truth label y and model prediction $\hat{y}$, the ***cross entropy loss*** is:

$$L = -[y \ln(\hat{y}) + (1-y)\ln(1-\hat{y})]$$

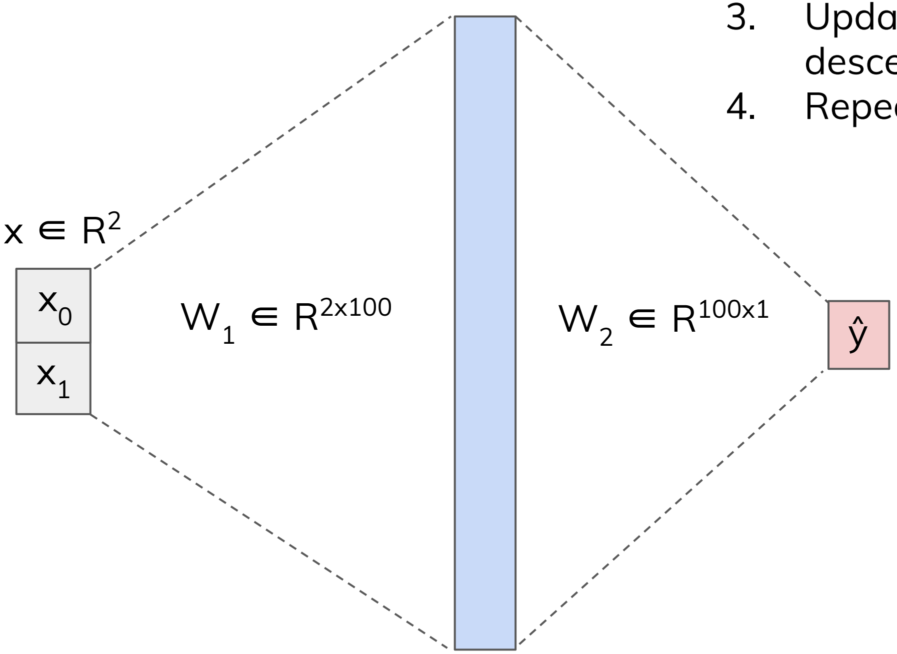
High level method

<u>Neural Network Training Procedure</u>

1. Randomly initialize weights
2. Calculate the **gradient** of the loss wrt parameters W_1 and W_2
3. Update parameters via gradient descent
4. Repeat 2-3 until done training

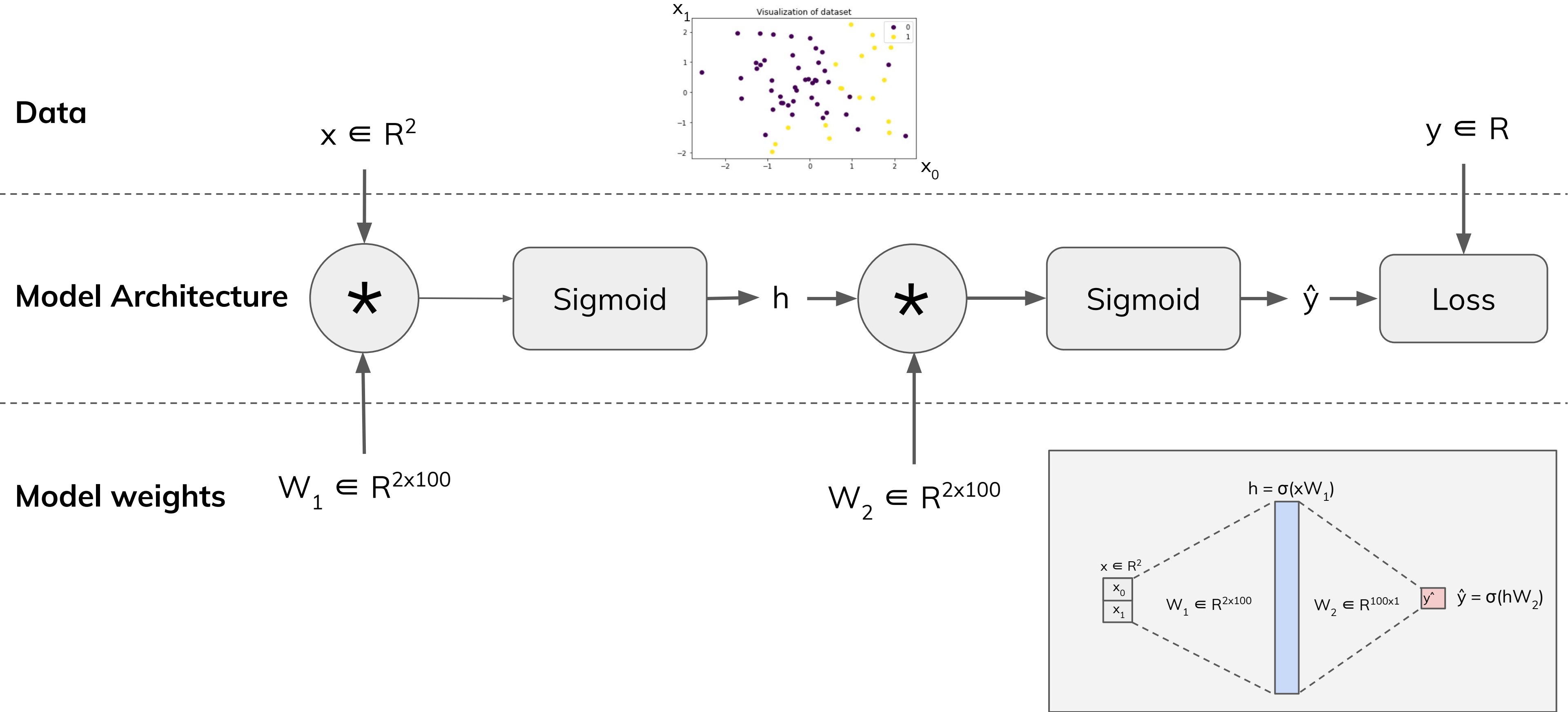


$h = \sigma(xW_1)$

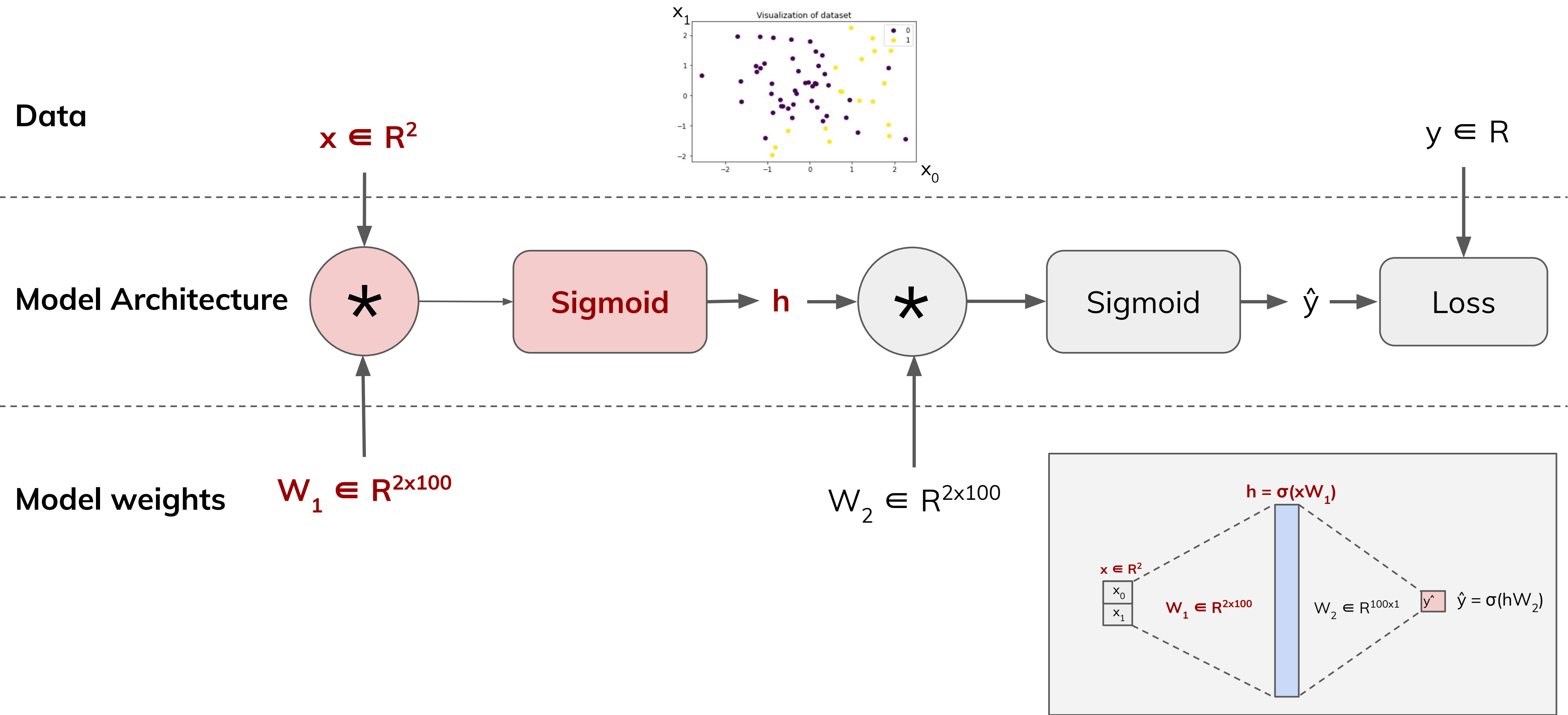


$\hat{y} = \sigma(hW_2)$

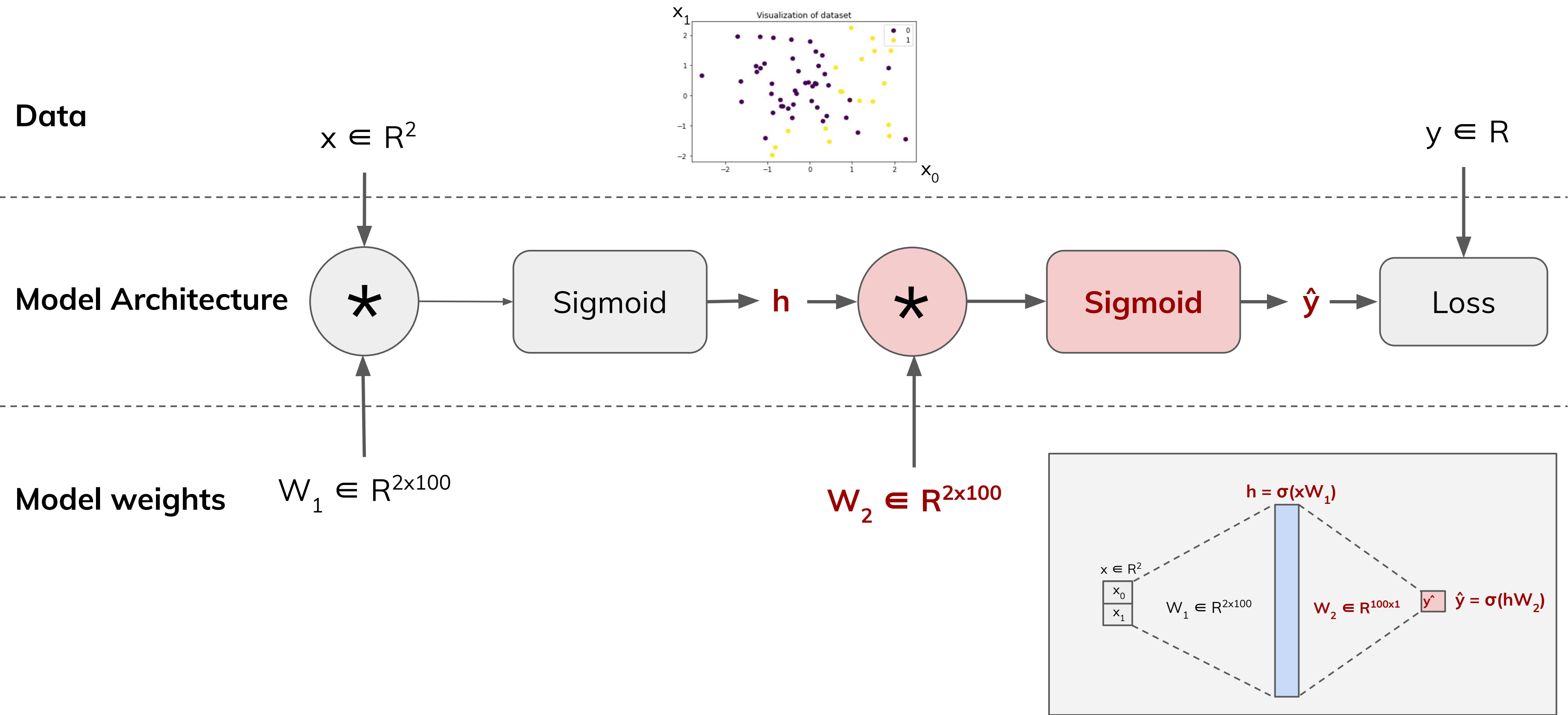
Introducing: the computation graph



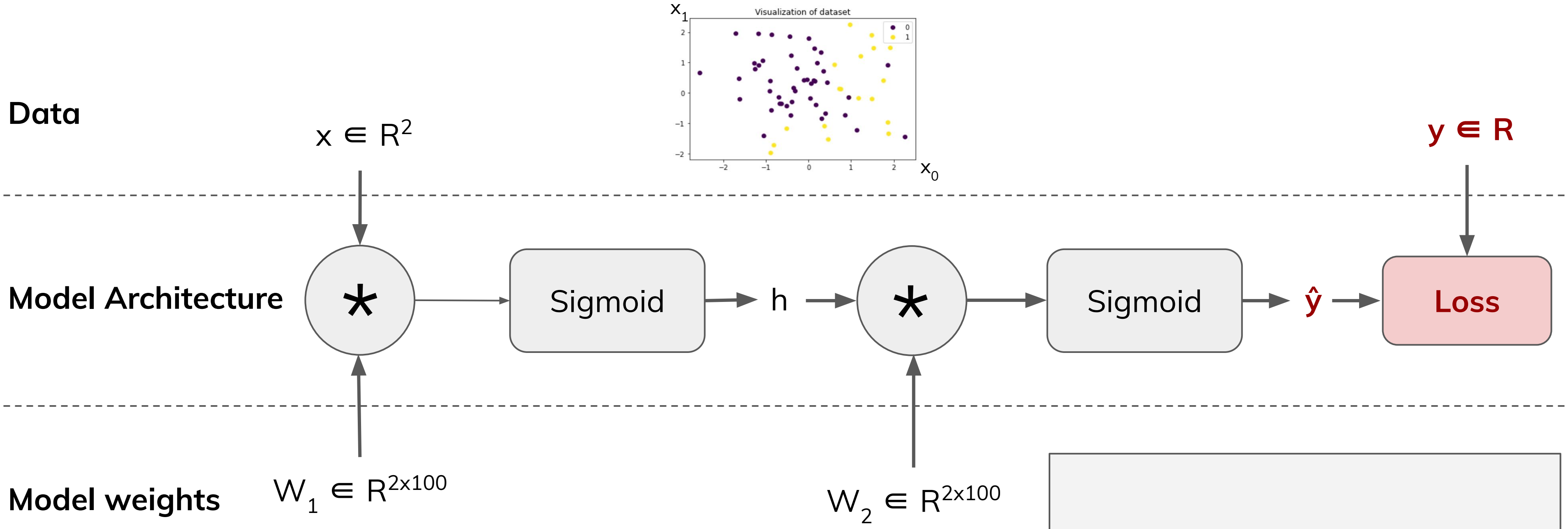
Introducing: the computation graph



Introducing: the computation graph



Introducing: the computation graph



$$L = -[y \ln(\hat{y}) + (1-y)\ln(1-\hat{y})]$$

Math of backpropagation

- Gradient descent optimization strategy:
  - Choose learning rate α
  - Randomly initialize W_1 and W_2
  - Calculate $\nabla W_1 = \partial L/\partial W_1$ and $\nabla W_2 = \partial L/\partial W_2$
  - Update weights:
    - $W_1 \mathrel{-}= \alpha \nabla W_1$
    - $W_2 \mathrel{-}= \alpha \nabla W_2$
- How to calculate $\partial L/\partial W_1$ and $\partial L/\partial W_2$?
  - Answer: Backprop (chain rule)

Calculating $\partial L/\partial W_1$ and $\partial L/\partial W_2$ (with code)

- The neural network can be represented as a series of computations
- $f(x; W_1, W_2) = \sigma((\sigma(xW_1)\ W_2)$
  - $h = \sigma(xW_1)$
  - $\hat{y} = \sigma(hW_2)$
- Broken down even more:
  - $z_1 = xW_1$
  - $h = \sigma(z_1)$
  - $z_2 = hW_2$
  - $\hat{y} = \sigma(z_2)$

```python
class Classifier():
    def __init__(self):
        self.w1, self.w2 = get_weights()

    def forward(self, x):
        self.x = x
        z1 = self.x @ self.w1
        self.h = sigmoid(z1)
        z2 = self.h @ self.w2
        y_pred = sigmoid(z2)
        return y_pred
```

Calculating ∂L/∂W_1 and ∂L/∂W_2 (with code)

- Broken down even more:
  - $z_1 = xW_1$
  - $h = \sigma(z_1)$
  - $z_2 = hW_2$
  - **$\hat{y} = \sigma(z_2)$**
- First step: calculate **∂L/∂ŷ**
  - $L = -[y \ln(\hat{y}) + (1-y)\ln(1-\hat{y})]$
  - → **∂L/∂ŷ** = **- [y/ŷ - (1-y)/(1-ŷ)]**

```
def backward(self, y_pred, y):
    grad_y_pred = -(np.divide(y, y_pred) - np.divide(1-y, 1-y_pred))
    grad_w2 = self.h.T @ (y_pred * (1-y_pred) * grad_y_pred)
    grad_h = self.w2.T * (grad_y_pred * y_pred * (1-y_pred))
    grad_w1 =  self.x.T @ (self.h * (1-self.h) * grad_h)

    # Update parameters
    self.w1 -= 1e-4 * grad_w1
    self.w2 -= 1e-4 * grad_w2
```

Calculating $\partial L/\partial W_1$ and $\partial L/\partial W_2$ (with code)

- Broken down even more:
    - $z_1 = xW_1$
    - $h = \sigma(z_1)$
    - $z_2 = hW_2$
    - $\hat{y} = \sigma(z_2)$
- Next step: calculate $\partial L/\partial z_2$
    - $\hat{y} = \sigma(z_2)$
    - $\rightarrow \partial L/\partial z_2 = \partial \hat{y}/\partial z_2 * \partial L/\partial \hat{y}$
    - Fact:
        - $\sigma'(x) = \sigma(x)\ (1 - \sigma(x))$
    - $\rightarrow \partial \hat{y}/\partial z_2 = \hat{y}\ (1 - \hat{y})$

```
def backward(self, y_pred, y):
    grad_y_pred = -(np.divide(y, y_pred) - np.divide(1-y, 1-y_pred))
    grad_w2 = self.h.T @ (y_pred * (1-y_pred)) * grad_y_pred)
    grad_h = self.w2.T * (grad_y_pred * y_pred * (1-y_pred))
    grad_w1 =  self.x.T @ (self.h * (1-self.h) * grad_h)

    # Update parameters
    self.w1 -= 1e-4 * grad_w1
    self.w2 -= 1e-4 * grad_w2
```

Calculating $\partial L/\partial W_1$ and $\partial L/\partial W_2$ (with code)

- Broken down even more:
  - $z_1 = xW_1$
  - $h = \sigma(z_1)$
  - **$z_2 = hW_2$**
  - $\hat{y} = \sigma(z_2)$
- Next step: calculate **$\partial L/\partial W_2$**
  - Just calculated: **$\partial L/\partial z_2$**
  - **$\partial L/\partial W_2$** = **$\partial z_2/\partial W_2$** * **$\partial L/\partial z_2$**
  - Since $z_2 = hW_2$
  - Order for vector chain rule is left to right
    - Only way the dims match!

```
def backward(self, y_pred, y):
    grad_y_pred = -(np.divide(y, y_pred) - np.divide(1-y, 1-y_pred))
    grad_w2 = self.h.T @ (y_pred * (1-y_pred) * grad_y_pred)
    grad_h = self.w2.T * (grad_y_pred * y_pred * (1-y_pred))
    grad_w1 =  self.x.T @ (self.h * (1-self.h) * grad_h)

    # Update parameters
    self.w1 -= 1e-4 * grad_w1
    self.w2 -= 1e-4 * grad_w2
```

Calculating $\partial L/\partial W_1$ and $\partial L/\partial W_2$ (with code)

- Broken down even more:
  - $z_1 = xW_1$
  - $h = \sigma(z_1)$
  - $\mathbf{z_2 = hW_2}$
  - $\hat{y} = \sigma(z_2)$
- Next step: calculate **$\partial L/\partial h$**
  - Previously calculated: **$\partial L/\partial z_2$**
  - **$\partial L/\partial h$** = **$\partial z_2/\partial h$** * **$\partial L/\partial z_2$**
  - **$\partial z_2/\partial h$** = W_2
  - **$\partial L/\partial z_2$** is a scalar

```
def backward(self, y_pred, y):
    grad_y_pred = -(np.divide(y, y_pred) - np.divide(1-y, 1-y_pred))
    grad_w2 = self.h.T @ (y_pred * (1-y_pred) * grad_y_pred)
    grad_h = self.w2.T * (grad_y_pred * y_pred * (1-y_pred))
    grad_w1 =  self.x.T @ (self.h * (1-self.h) * grad_h)

    # Update parameters
    self.w1 -= 1e-4 * grad_w1
    self.w2 -= 1e-4 * grad_w2
```

Calculating $\partial L/\partial W_1$ and $\partial L/\partial W_2$ (with code)

- Broken down even more:
  - $z_1 = xW_1$
  - $h = \sigma(z_1)$
  - $z_2 = hW_2$
  - $\hat{y} = \sigma(z_2)$
- Next step: calculate $\boldsymbol{\partial L/\partial z_1}$
  - Previously calculated: $\boldsymbol{\partial L/\partial h}$
  - $\boldsymbol{\partial L/\partial z_1} = \boldsymbol{\partial h/\partial z_1} * \boldsymbol{\partial L/\partial h}$
  - Fact:
    - $\sigma'(x) = \sigma(x)\ (1 - \sigma(x))$
  - Since $h=\sigma(z_1)$, $\boldsymbol{\partial h/\partial z_1 = h\ (1 - h)}$

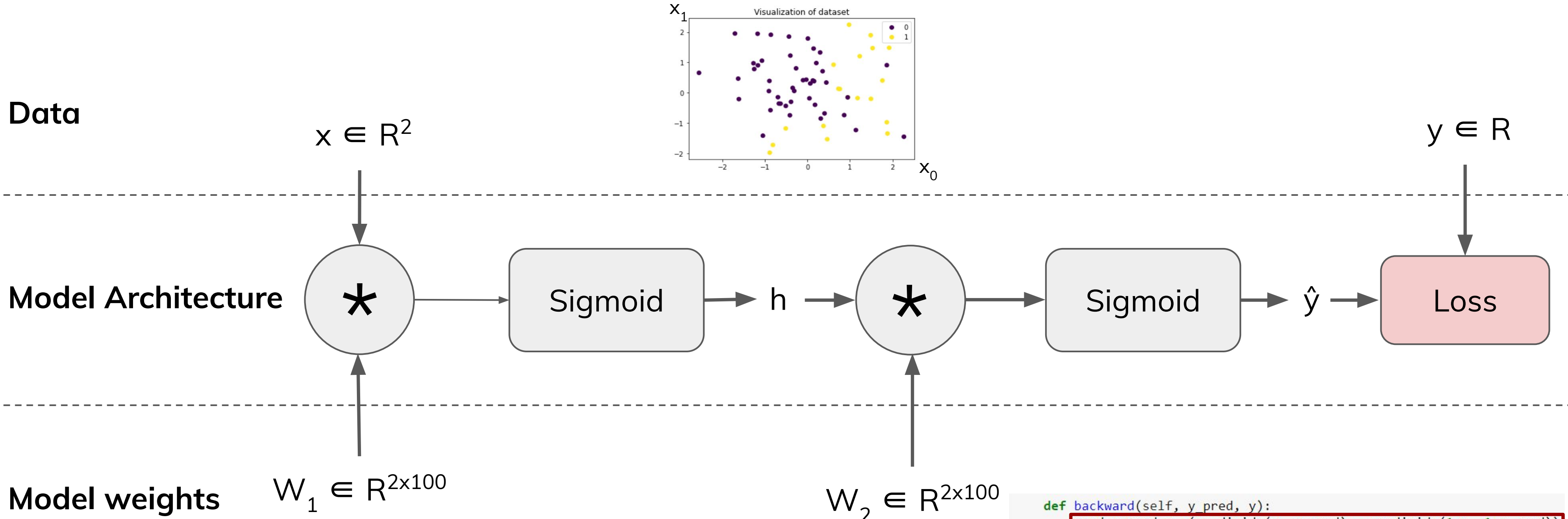
```
def backward(self, y_pred, y):
    grad_y_pred = -(np.divide(y, y_pred) - np.divide(1-y, 1-y_pred))
    grad_w2 = self.h.T @ (y_pred * (1-y_pred) * grad_y_pred)
  * grad_h = self.w2.T * (grad_y_pred * y_pred * (1-y_pred))
    grad_w1 =  self.x.T @ (self.h * (1-self.h) * grad_h)

    # Update parameters
    self.w1 -= 1e-4 * grad_w1
    self.w2 -= 1e-4 * grad_w2
```

Calculating $\partial L/\partial W_1$ and $\partial L/\partial W_2$ (with code)

- Broken down even more:
  - $\mathbf{z_1 = xW_1}$
  - $h = \sigma(z_1)$
  - $z_2 = hW_2$
  - $\hat{y} = \sigma(z_2)$
- Final step: calculate $\boldsymbol{\partial L/\partial W_1}$
  - Previously calculated: $\boldsymbol{\partial L/\partial z_1}$
  - $\boldsymbol{\partial L/\partial W_1} = \boldsymbol{\partial z_1/\partial W_1} * \boldsymbol{\partial L/\partial z_1}$

```
def backward(self, y_pred, y):
    grad_y_pred = -(np.divide(y, y_pred) - np.divide(1-y, 1-y_pred))
    grad_w2 = self.h.T @ (y_pred * (1-y_pred) * grad_y_pred)
    grad_h = self.w2.T * (grad_y_pred * y_pred * (1-y_pred))
    grad_w1 =  self.x.T @ (self.h * (1-self.h) * grad_h)

    # Update parameters
    self.w1 -= 1e-4 * grad_w1
    self.w2 -= 1e-4 * grad_w2
```

Calculating $\partial L/\partial W_1$ and $\partial L/\partial W_2$ + computation graph

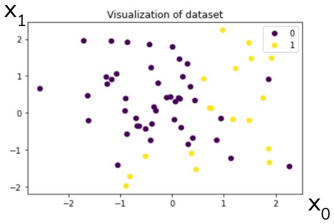


$$L = -[y \ln(\hat{y}) + (1-y)\ln(1-\hat{y})]$$

$$\rightarrow \partial L/\partial \hat{y} = \mathbf{-\ [y/\hat{y} - (1-y)/(1-\hat{y})]}$$

```python
def backward(self, y_pred, y):
    grad_y_pred = -(np.divide(y, y_pred) - np.divide(1-y, 1-y_pred))
    grad_w2 = self.h.T @ (y_pred * (1-y_pred) * grad_y_pred)
    grad_h = self.w2.T * (grad_y_pred * y_pred * (1-y_pred))
    grad_w1 =  self.x.T @ (self.h * (1-self.h) * grad_h)

    # Update parameters
    self.w1 -= 1e-4 * grad_w1
    self.w2 -= 1e-4 * grad_w2
```

Calculating $\partial L/\partial W_1$ and $\partial L/\partial W_2$ + computation graph

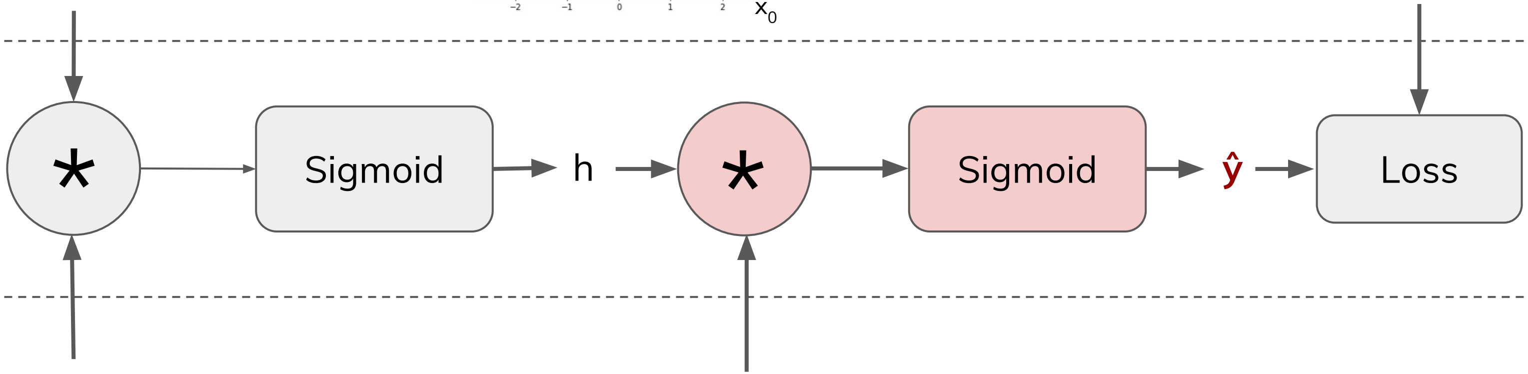


Data

$x \in R^2$

$y \in R$

Model Architecture



Model weights

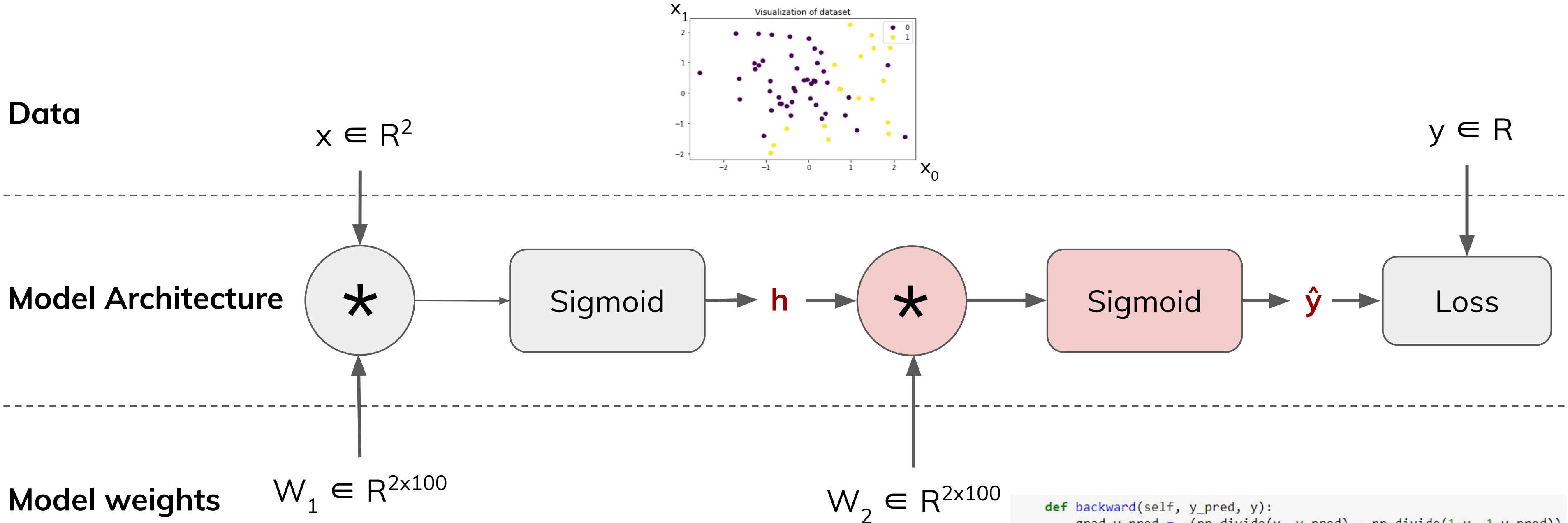
$W_1 \in R^{2x100}$

$W_2 \in R^{2x100}$

```
def backward(self, y_pred, y):
    grad_y_pred = -(np.divide(y, y_pred) - np.divide(1-y, 1-y_pred))
    grad_w2 = self.h.T @ (y_pred * (1-y_pred) * grad_y_pred)
    grad_h = self.w2.T * (grad_y_pred * y_pred * (1-y_pred))
    grad_w1 =  self.x.T @ (self.h * (1-self.h) * grad_h)

    # Update parameters
    self.w1 -= 1e-4 * grad_w1
    self.w2 -= 1e-4 * grad_w2
```

$\partial L/\partial W_2 = \partial z_2/\partial W_2 * \partial L/\partial z_2$

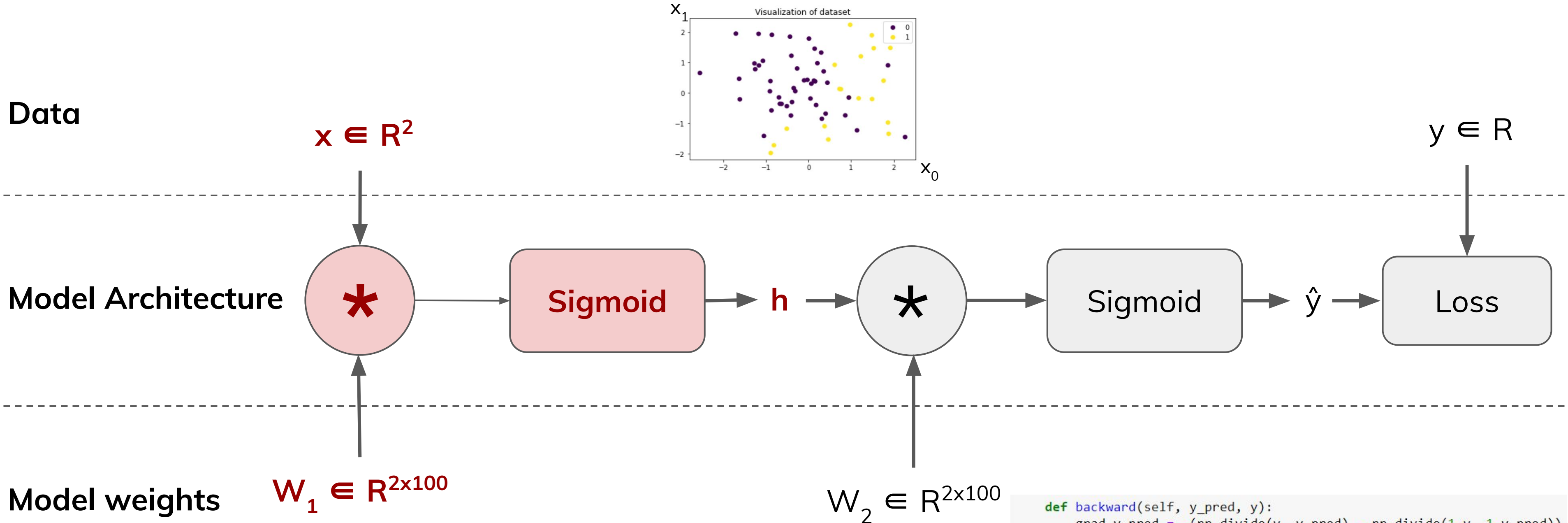
Calculating ∂L/∂W_1 and ∂L/∂W_2 + computation graph




```
def backward(self, y_pred, y):
    grad_y_pred = -(np.divide(y, y_pred) - np.divide(1-y, 1-y_pred))
    grad_w2 = self.h.T @ (y_pred * (1-y_pred) * grad_y_pred)
    grad_h = self.w2.T * (grad_y_pred * y_pred * (1-y_pred))
    grad_w1 =  self.x.T @ (self.h * (1-self.h) * grad_h)

    # Update parameters
    self.w1 -= 1e-4 * grad_w1
    self.w2 -= 1e-4 * grad_w2
```

$\partial L/\partial h = \partial z_2/\partial h * \partial L/\partial z_2$

Calculating $\partial L/\partial W_1$ and $\partial L/\partial W_2$ + computation graph




```python
def backward(self, y_pred, y):
    grad_y_pred = -(np.divide(y, y_pred) - np.divide(1-y, 1-y_pred))
    grad_w2 = self.h.T @ (y_pred * (1-y_pred) * grad_y_pred)
    grad_h = self.w2.T * (grad_y_pred * y_pred * (1-y_pred))
    grad_w1 = self.x.T @ (self.h * (1-self.h) * grad_h)

    # Update parameters
    self.w1 -= 1e-4 * grad_w1
    self.w2 -= 1e-4 * grad_w2
```

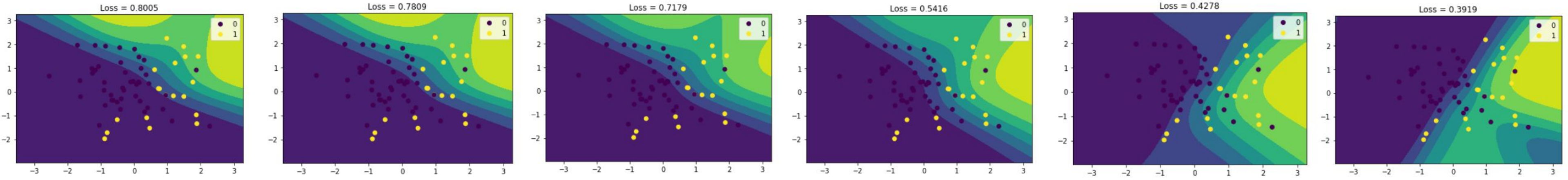
$$\partial L/\partial W_1 = \partial z_1/\partial W_1 * \partial L/\partial z_1$$

Running code + visualizing training

```python
class Classifier():
    def __init__(self):
        self.w1, self.w2 = get_weights()

    def forward(self, x):
        self.x = x
        z1 = self.x @ self.w1
        self.h = sigmoid(z1)
        z2 = self.h @ self.w2
        y_pred = sigmoid(z2)
        return y_pred

    def backward(self, y_pred, y):
        grad_y_pred = -(np.divide(y, y_pred) - np.divide(1-y, 1-y_pred))
        grad_w2 = self.h.T @ (y_pred * (1-y_pred) * grad_y_pred)
        grad_h = self.w2.T * (grad_y_pred * y_pred * (1-y_pred))
        grad_w1 =  self.x.T @ (self.h * (1-self.h) * grad_h)

        # Update parameters
        self.w1 -= 1e-4 * grad_w1
        self.w2 -= 1e-4 * grad_w2
```

```python
for t in range(10000):

    # Visualize classifier
    if t == 5 or t == 20 or t== 100 or t % 1000 == 0:
        plot = plot_decision_boundary(clf, x)
        visualize_dataset(x, y, title="Loss = {}".format(round(loss, 4)))

    # Predict on data (forward)
    y_pred = clf.forward(x)

    # Backpropogate errors
    clf.backward(y_pred, y)

    # Calculate loss for next plot
    loss = cross_entropy(y_pred, y)
```

Model outputs better match data as loss decreases



Recap

- Review of Neural Nets
- Showed math for analytically calculating gradients
  - Related to steps in computation graph
  - Provided code snippets for each part
- Questions?