

# Computational Intelligence

Samaneh Hosseini

Isfahan University of Technology

# Outline

- Training Neural Networks
  - Loss Optimization
  - Gradient Descent
  - Backpropagation

# Training Neural Networks

$$\mathcal{L}(\hat{y}, y)$$

# Loss Optimization

We want to find the network weights that **achieve the lowest loss**

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

$\hat{y}^{(i)}$   
↳ Empirical loss

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$

# Loss Optimization

We want to find the network weights that **achieve the lowest loss**

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$



Remember:

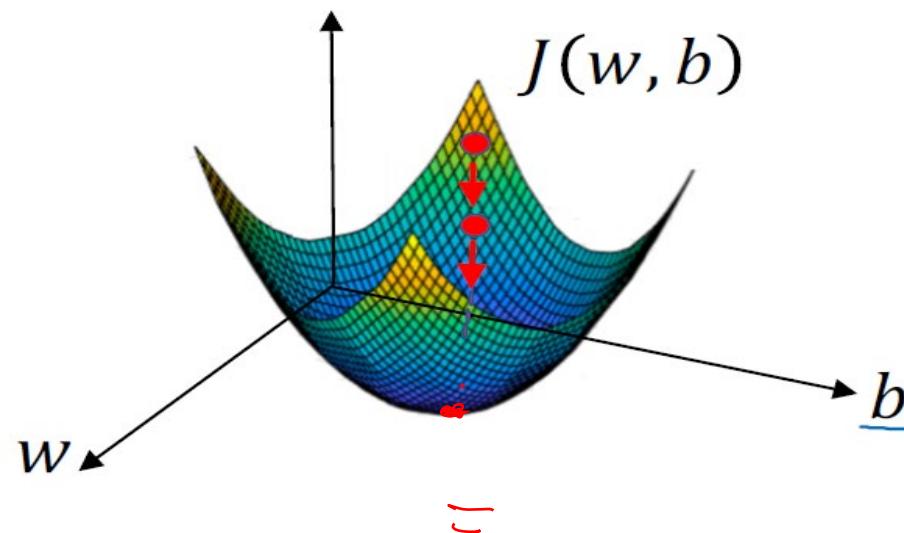
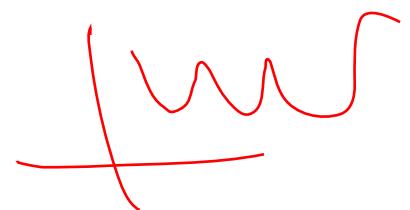
$$\mathbf{W} = \{\mathbf{W}^{(0)}, \mathbf{W}^{(1)}, \dots\}$$

# Cross Entropy Loss Optimization

Recap:  $\hat{y} = \sigma(w^T x + b)$ ,  $\sigma(z) = \frac{1}{1+e^{-z}}$

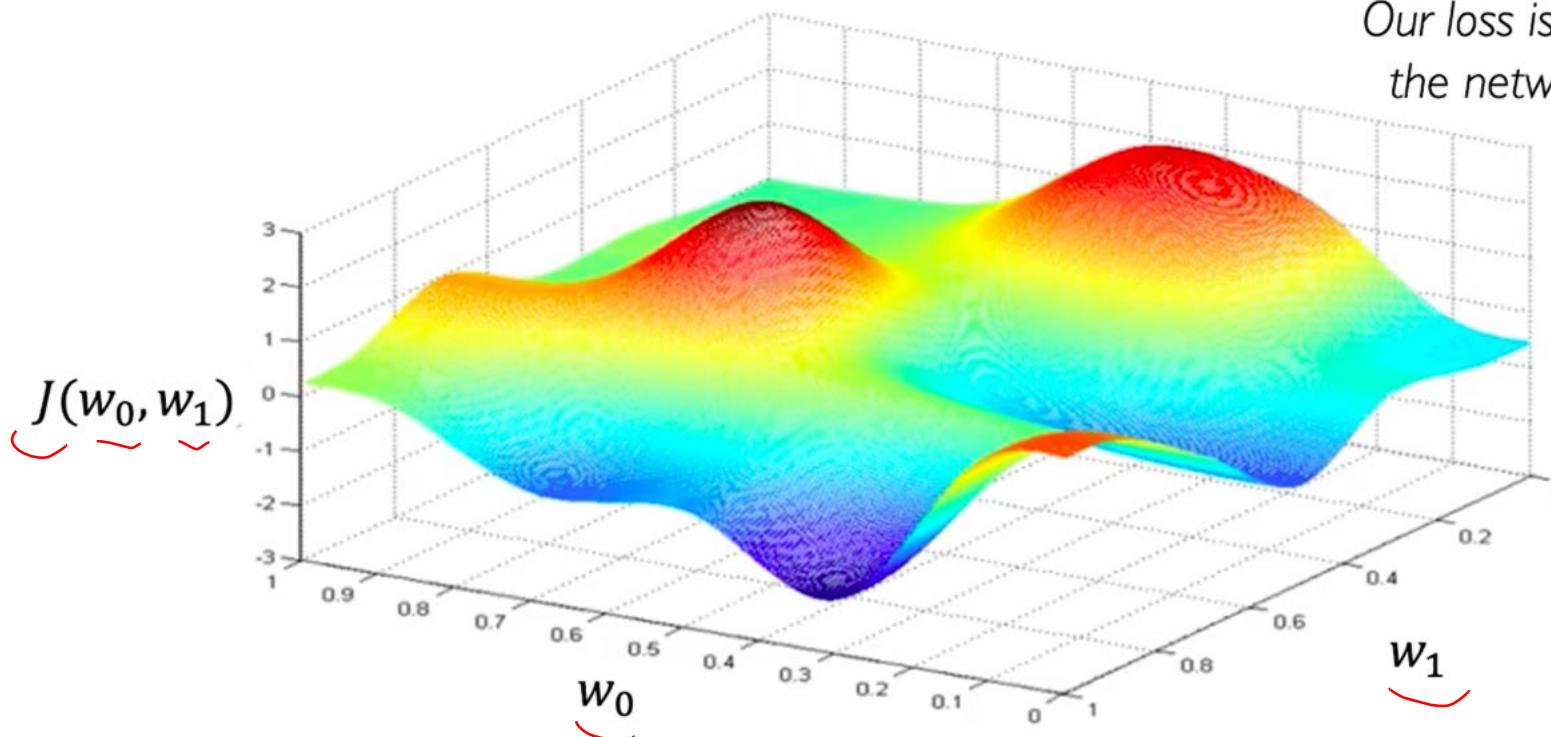
$$J(w, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^m y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})$$

Want to find  $w, b$  that minimize  $J(w, b)$



# Loss Optimization

$$\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmin}} J(\mathbf{W})$$

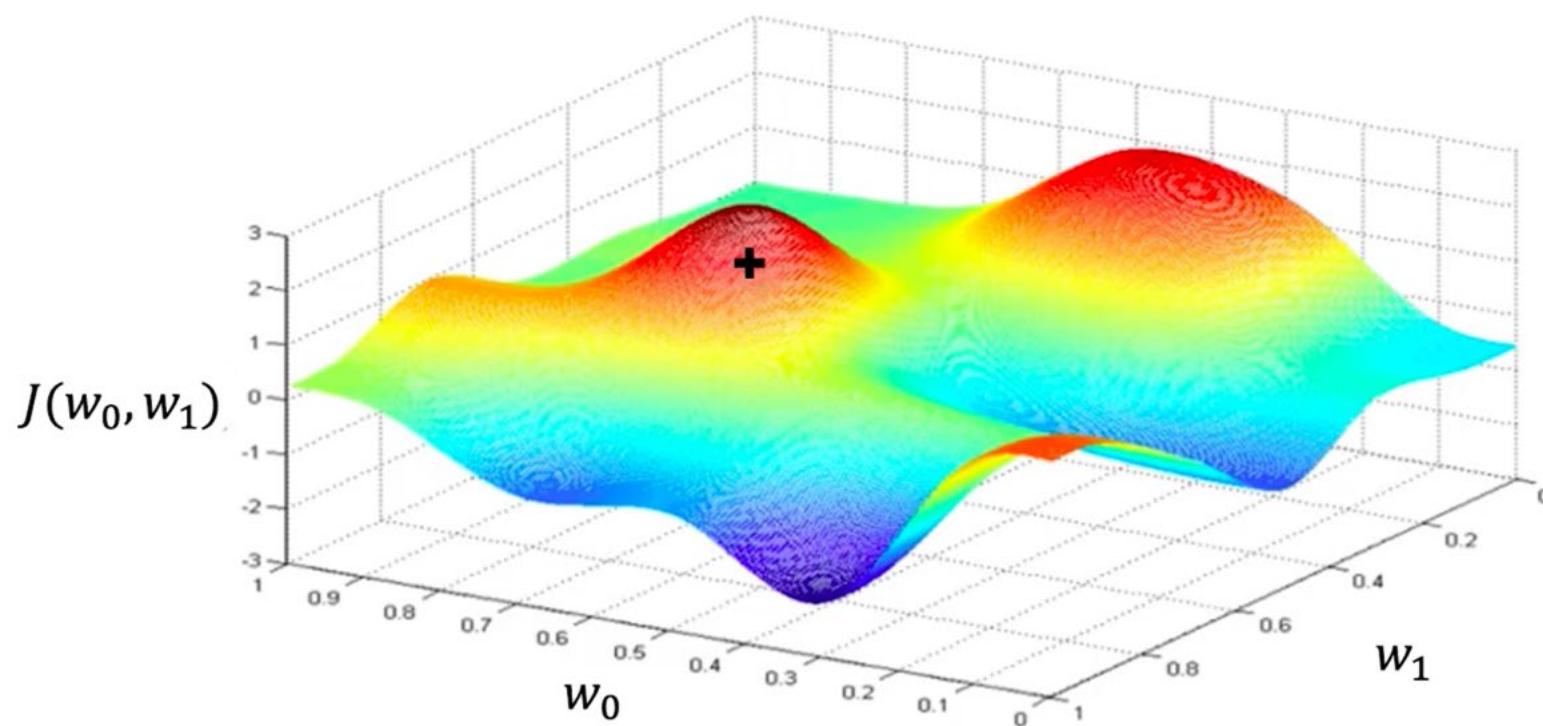


Remember:  
Our loss is a function of  
the network weights!

Non-Convex

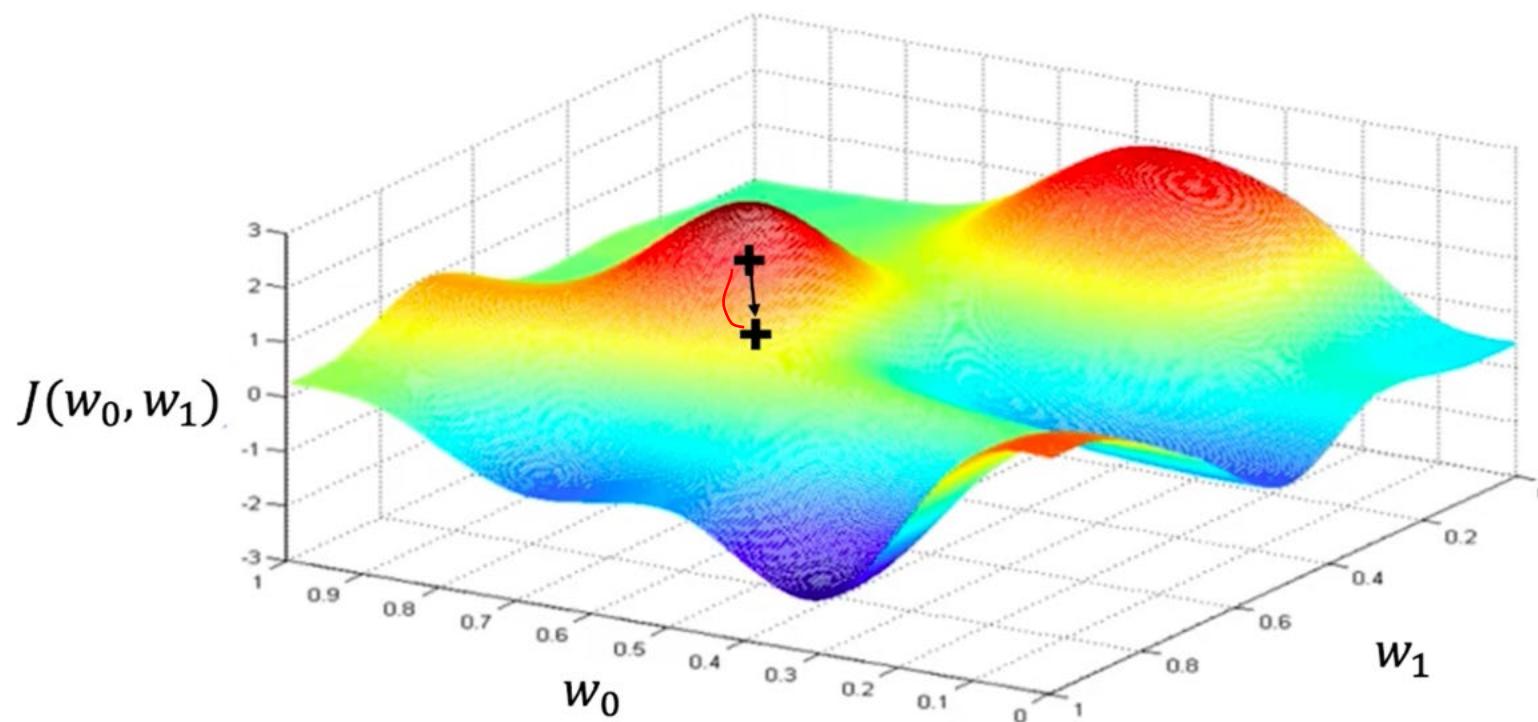
# Loss Optimization

Randomly pick an initial  $(\underline{w_0}, \underline{w_1})$



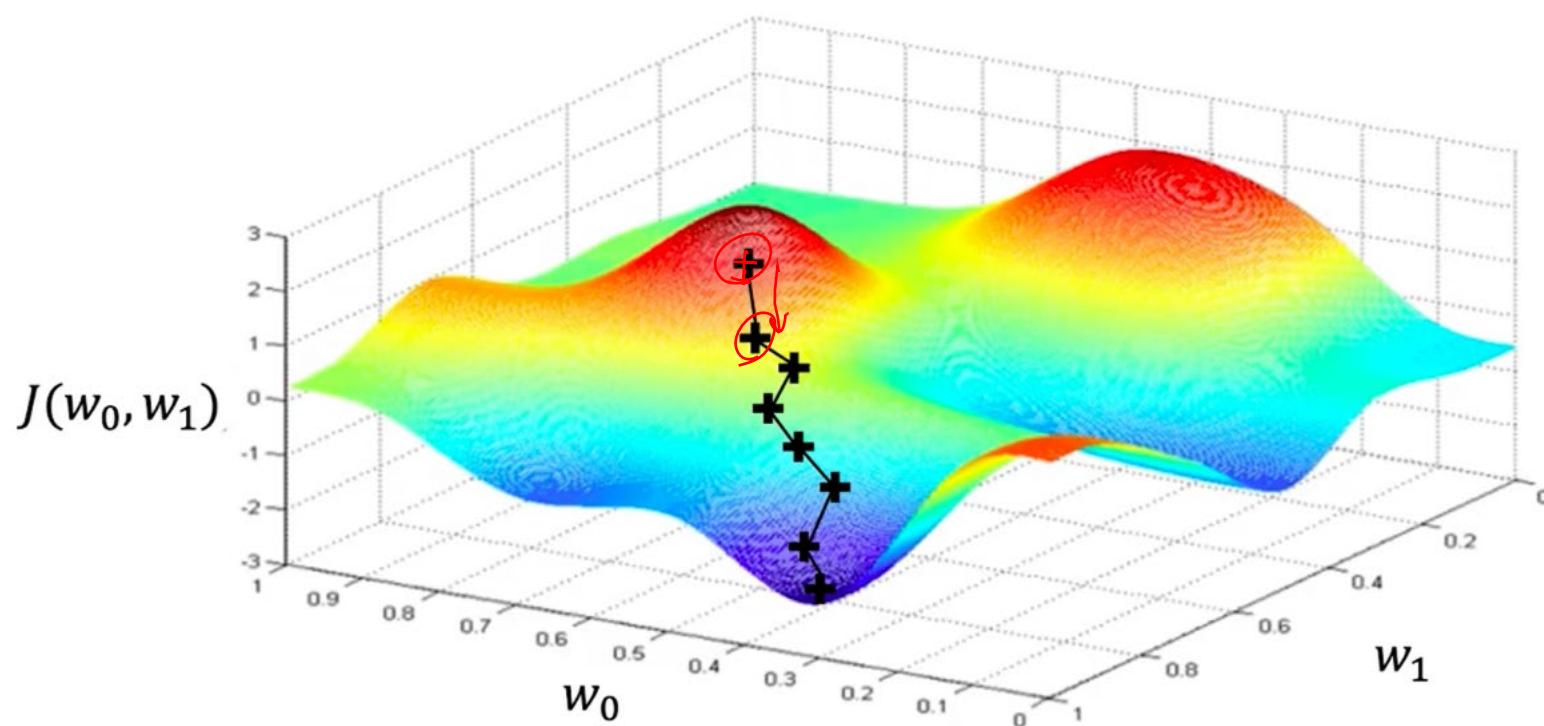
# Loss Optimization

Take small step in opposite direction of gradient



# Loss Optimization

Repeat until convergence



# Gradient Descent

“Walking downhill and always taking a step in the direction that goes down the most.”

## Algorithm

1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient,  $\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$
4. Update weights,  $\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$   
*learning rate*
5. Return weights

## Why update the weight proportional to the negative of the partial derivative?

- ▶ Suppose that we want to find the minimum of  $y = x^2$ .

▶ Start with  $x = x_0$ .

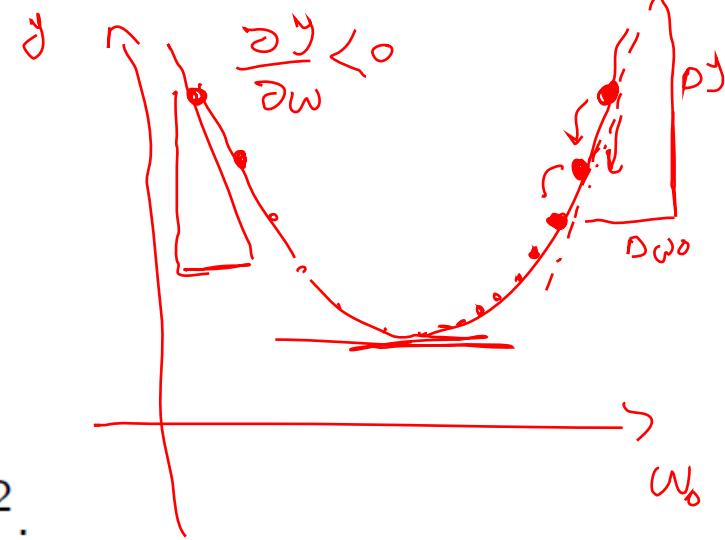
- ▶ In what direction should we change the value of  $x$ ?

$$\frac{\partial y}{\partial w}$$

- ▶ By what amount should we change the value of  $x$ ?

What is the step size?

$$\eta \quad \frac{\partial y}{\partial w}$$



$$\frac{\partial y}{\partial w_0} > 0 \Rightarrow -\eta \frac{\partial y}{\partial w_0} < 0$$

$$\frac{\partial y}{\partial w} < 0 \Rightarrow -\eta \frac{\partial y}{\partial w} > 0$$

# Gradient Descent



## Algorithm

1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient,  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

```
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

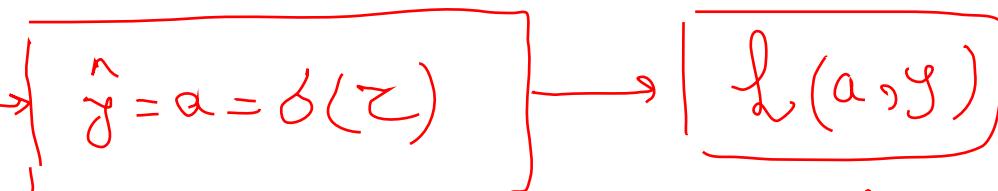
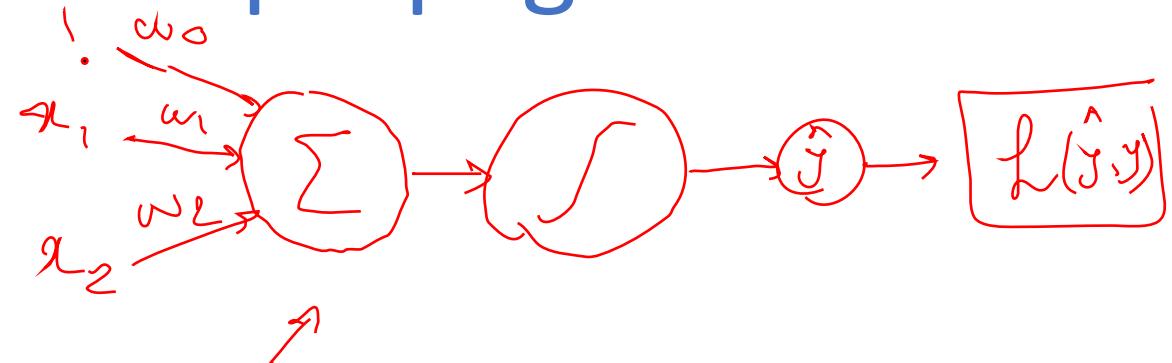
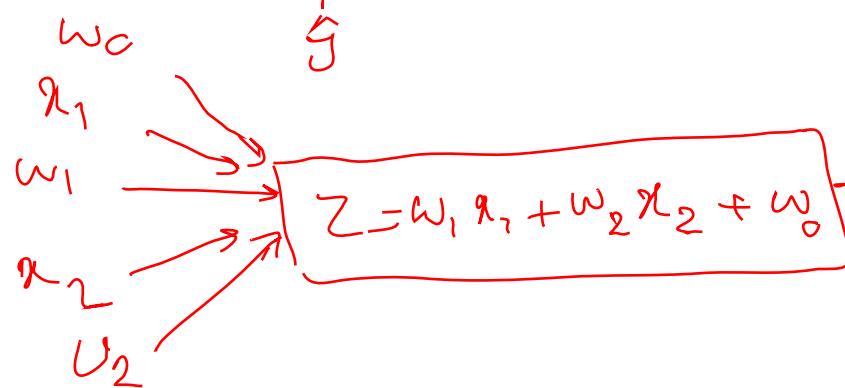
while True:    # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)
    weights = weights - lr * gradient
```

# Computing Gradients: Backpropagation

$$z = w^T x + w_0$$

$$\hat{y} = a = \sigma(z)$$

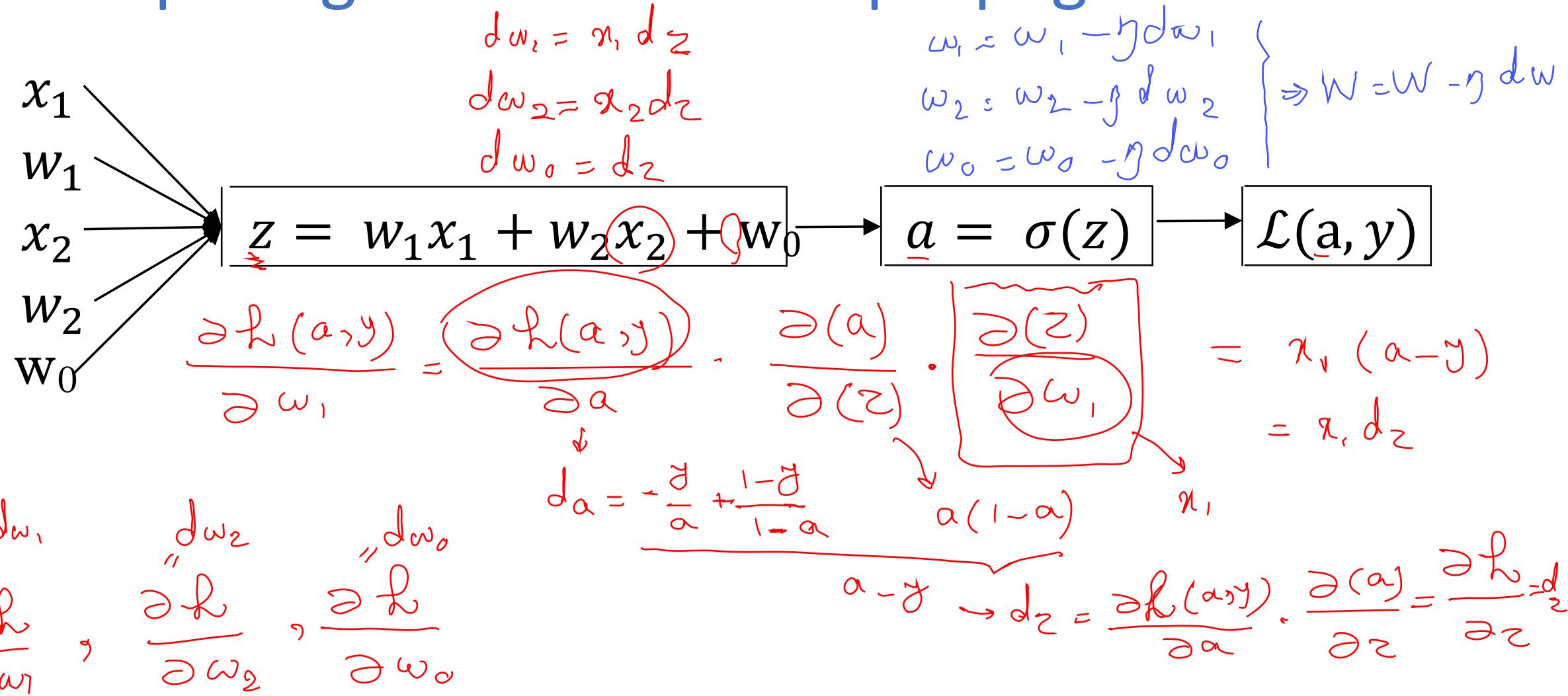
$$\rightarrow \mathcal{L}(a, y) = -(y \log(a) + (1 - y) \log(1 - a))$$



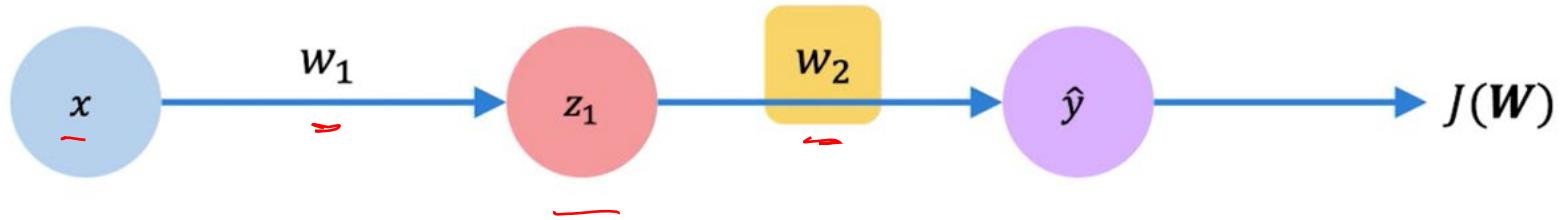
reduce loss

$$\frac{\partial h}{\partial \omega_i} = d\omega_i$$

# Computing Gradients: Backpropagation

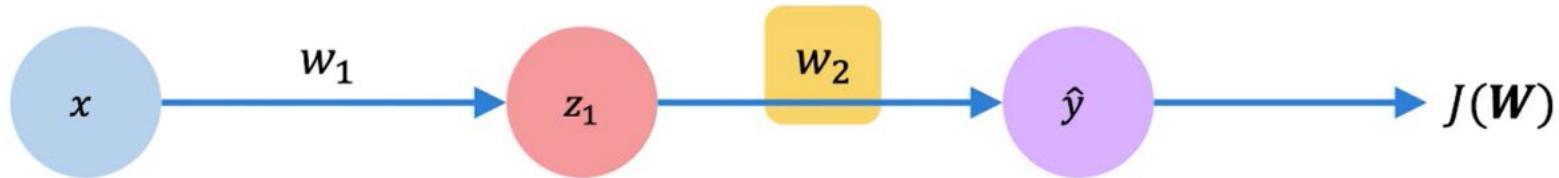


# Computing Gradients: Backpropagation



How does a small change in one weight (ex.  $w_2$ ) affect the final loss  $J(\mathbf{W})$ ?

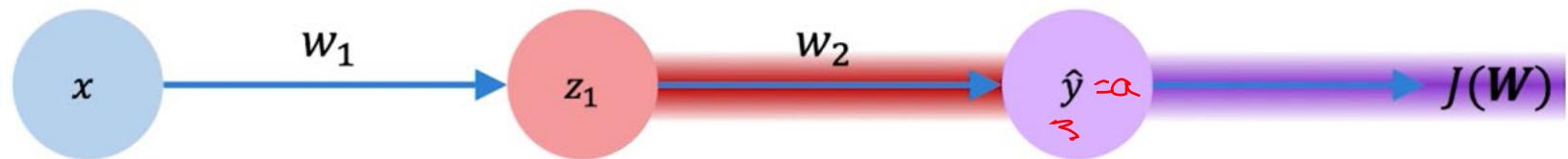
# Computing Gradients: Backpropagation



$$\frac{\partial J(\mathbf{W})}{\partial w_2} =$$

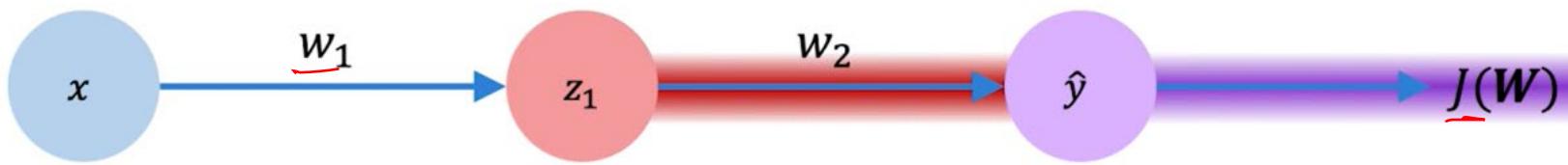
Let's use the chain rule!

# Computing Gradients: Backpropagation



$$\frac{\partial J(\mathbf{W})}{\partial w_2} = \underline{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}} * \underline{\frac{\partial \hat{y}}{\partial w_2}}$$

# Computing Gradients: Backpropagation

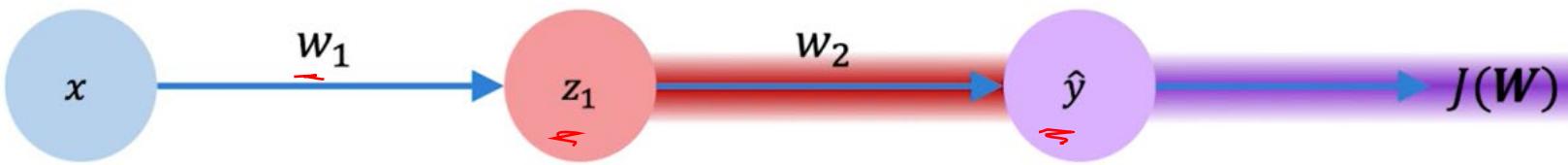


$$\frac{\partial J(\mathbf{W})}{\partial w_2} = \underline{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}} * \underline{\frac{\partial \hat{y}}{\partial w_2}}$$

$$\frac{\partial J(\mathbf{W})}{\partial w_1} =$$

Apply chain rule!

# Computing Gradients: Backpropagation



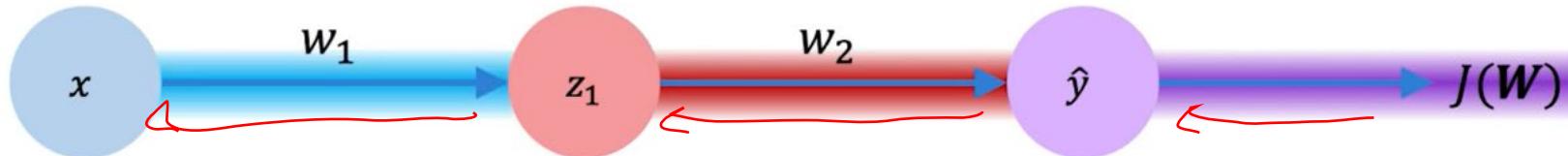
$$\frac{\partial J(\mathbf{W})}{\partial w_2} = \underline{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}} * \underline{\frac{\partial \hat{y}}{\partial w_2}}$$

$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \underline{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}} * \underline{\frac{\partial \hat{y}}{\partial w_1}}$$

Apply chain rule!

Apply chain rule!

# Computing Gradients: Backpropagation



$$\frac{\partial J(\mathbf{W})}{\partial w_2} = \underline{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}} * \underline{\frac{\partial \hat{y}}{\partial w_2}}$$

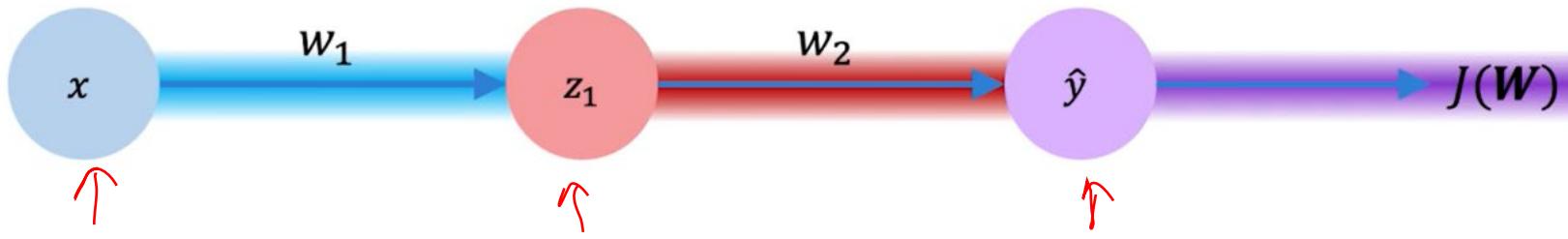
$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \underline{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}} * \underline{\frac{\partial \hat{y}}{\partial w_1}}$$

Apply chain rule!

$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \underline{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}} * \underline{\frac{\partial \hat{y}}{\partial z_1}} * \underline{\frac{\partial z_1}{\partial w_1}}$$

Apply chain rule!

# Computing Gradients: Backpropagation



$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \underbrace{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue}}$$

Repeat this for **every weight in the network** using gradients from later layers

( $\hat{x}^{(i)} \rightarrow \hat{y}^{(i)}$ )

## Gradient descent on n examples

$$J(\omega, \omega_0) = \frac{1}{n} \sum_{i=1}^n h(a_i^{(i)}, \hat{y}^{(i)})$$

$$a^{(i)} = \hat{y}^{(i)} = \sigma(\hat{z}^{(i)}) = \sigma(\omega^\top \hat{x}^{(i)} + \omega_0)$$

$$\partial_{\omega_1}^{(i)}, \partial_{\omega_2}^{(i)}, \partial_{\omega_0}^{(i)}$$

$$\frac{\partial J(\omega)}{\partial \omega_1} = \frac{1}{n} \sum_{i=1}^n \frac{\partial h(a^{(i)}, \hat{y}^{(i)})}{\partial \omega_1}$$

$$\partial_{\omega_1}^{(i)} = (\hat{x}^{(i)}, \hat{y}^{(i)})$$

$$\delta \omega_1 = \frac{\partial J}{\partial \omega_1}$$

## Gradient descent on n examples

$$J=0 ; \quad \delta \omega_1 = \delta \omega_2 = \delta \omega_0 = 0 ;$$

→ For  $i=1$  to n

$$z^{(i)} = \omega^T x^{(i)} + \omega_0$$

$$\alpha^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)})$$

$$J += - [y^{(i)} \log \alpha^{(i)} + (1-y^{(i)}) \log (1-\alpha^{(i)})]$$

$$\delta z^{(i)} = \alpha^{(i)} - \hat{y}^{(i)}$$

$$\delta \omega_0 += \delta z^{(i)}$$

$$\delta \omega_1 += x_1^{(i)} \delta z^{(i)}$$

$$\delta \omega_2 += x_2^{(i)} \delta z^{(i)}$$

$$J = \bar{J} / n ; \quad \delta \omega_0 = \delta \omega_0 / n ; \quad \delta \omega_1 / = n ; \quad \delta \omega_2 / = n ;$$

$$\omega_1 := \omega_1 - \gamma \delta \omega_1$$

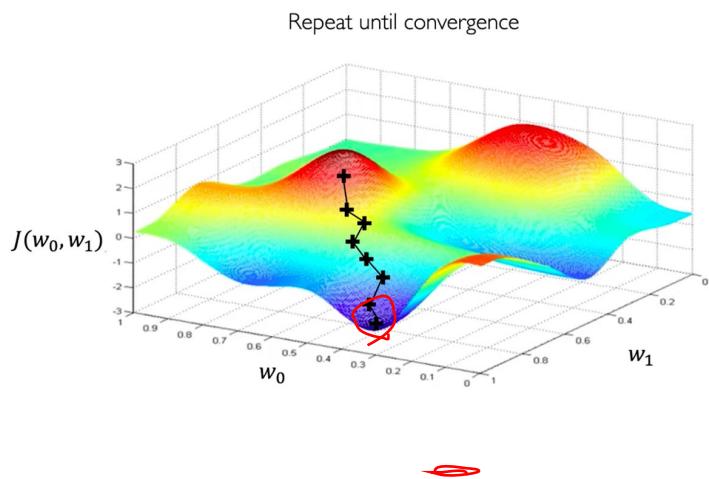
$$\omega_2 := \omega_2 - \gamma \delta \omega_2$$

$$\omega_0 := \omega_0 - \gamma \delta \omega_0$$

Vectorization

# Core Foundation Review

## Loss Optimization

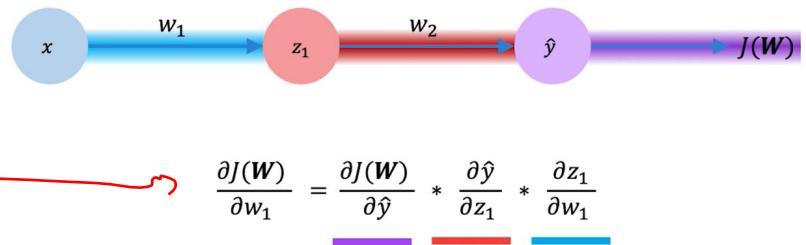


## Gradient Descent

### Algorithm

1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient,  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

## Backpropagation



Repeat this for every weight in the network using gradients from later layers