

# Deep Neural Network

# What is Deep Learning?

## ARTIFICIAL INTELLIGENCE

Any technique that enables computers to mimic human behavior



## MACHINE LEARNING

Ability to learn without explicitly being programmed



## DEEP LEARNING

Extract patterns from data using neural networks



# Why Deep Learning?

Hand engineered features are time consuming, brittle, and not scalable in practice

Can we learn the **underlying features** directly from data?

**Low Level Features**



Lines & Edges

**Mid Level Features**



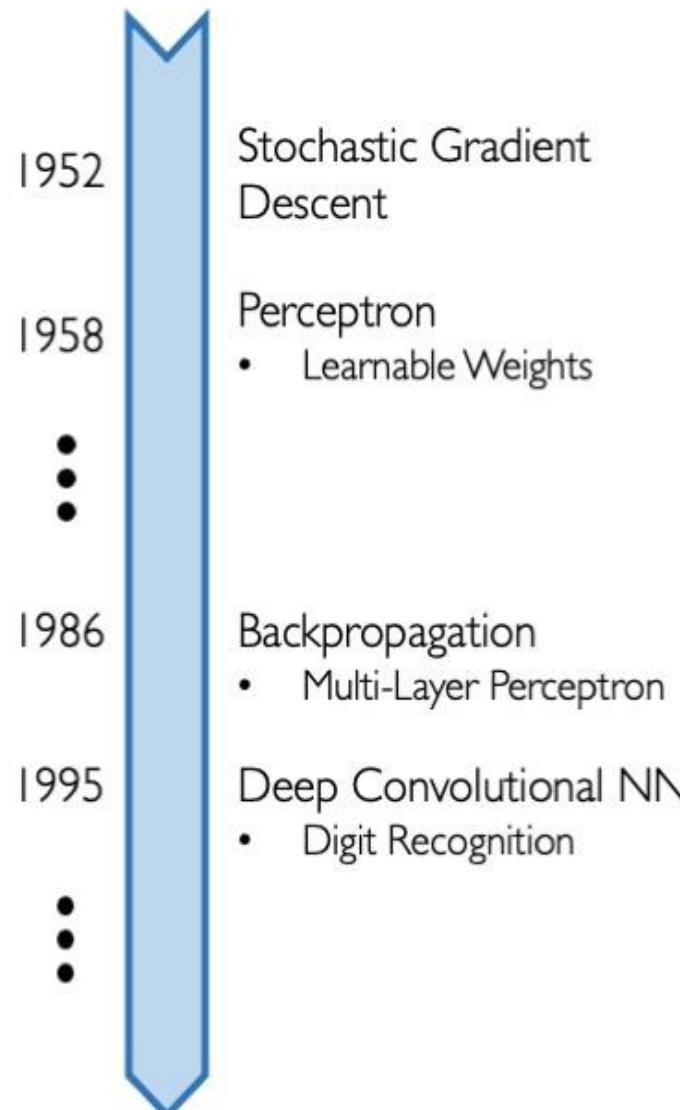
Eyes & Nose & Ears

**High Level Features**



Facial Structure

# Why Now?

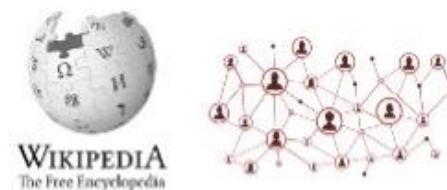


Neural Networks date back decades, so why the resurgence?

## 1. Big Data

- Larger Datasets
- Easier Collection & Storage

IMAGENET



## 2. Hardware

- Graphics Processing Units (GPUs)
- Massively Parallelizable



## 3. Software

- Improved Techniques
- New Models
- Toolboxes

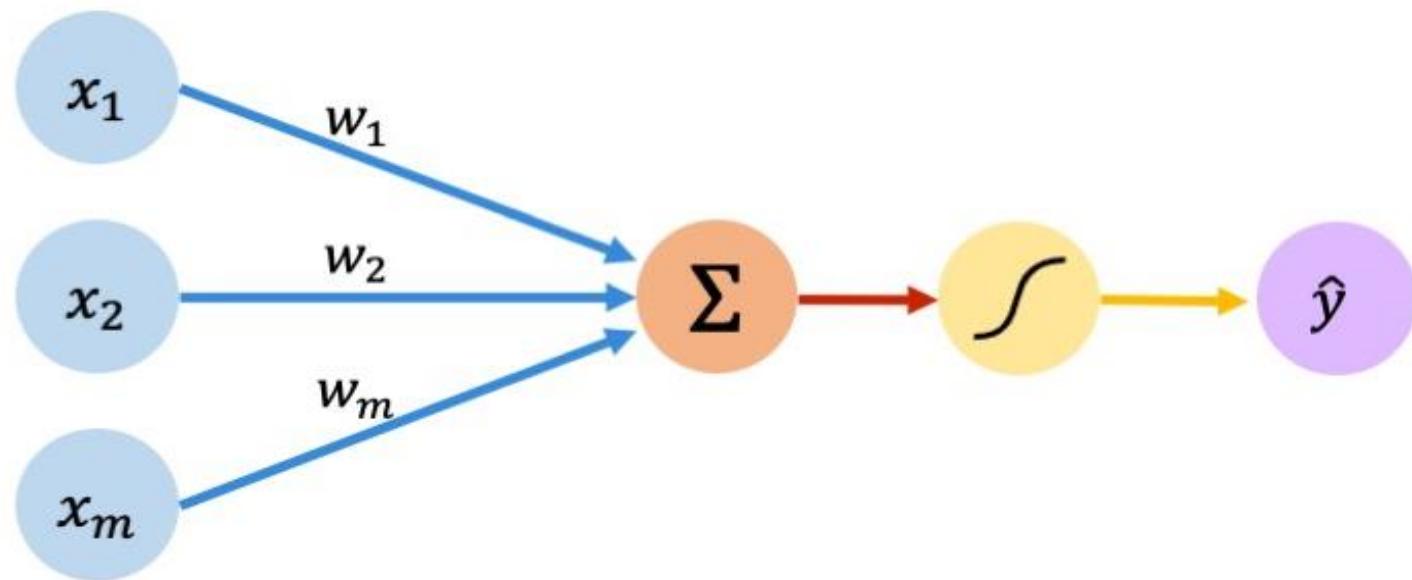


TensorFlow

# The Perceptron

## The structural building block of deep learning

# The Perceptron: Forward Propagation



Inputs      Weights      Sum      Non-Linearity      Output

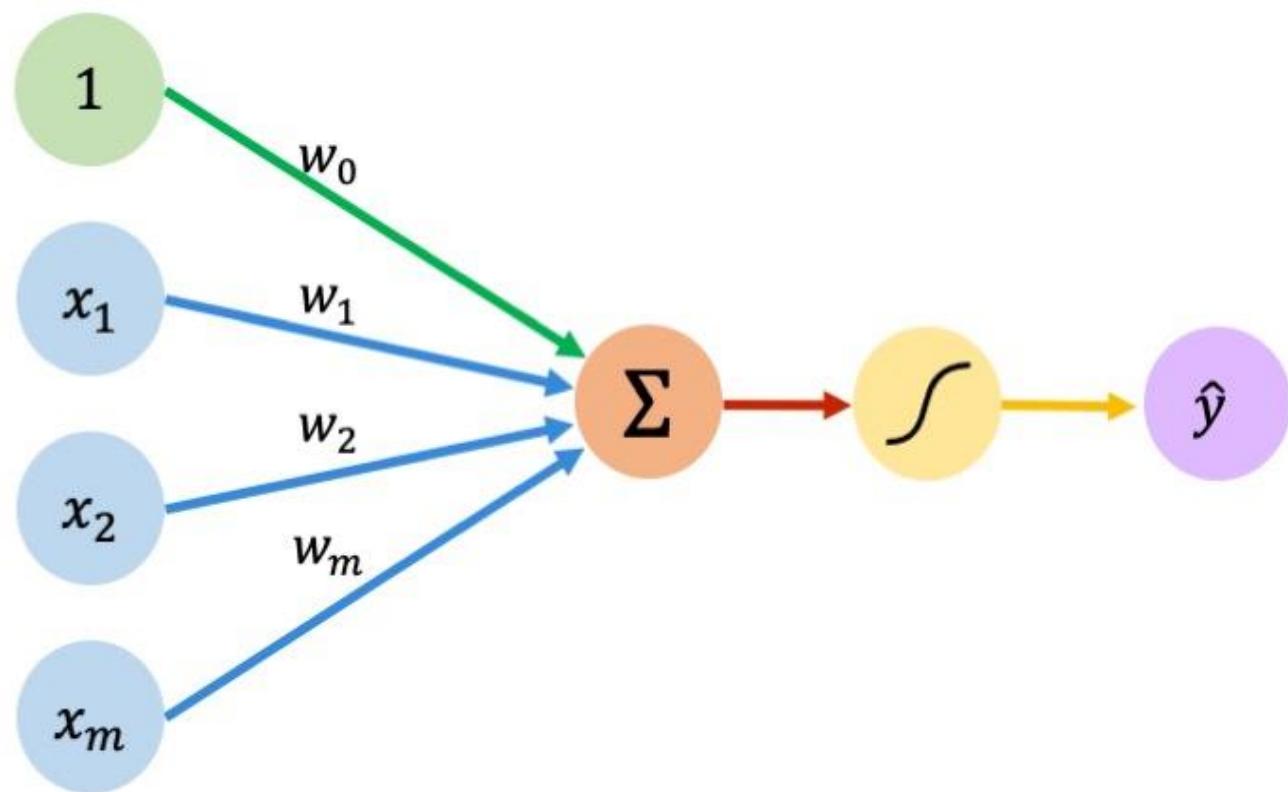
Output

Linear combination of inputs

$$\hat{y} = g \left( \sum_{i=1}^m x_i w_i \right)$$

Non-linear activation function

# The Perceptron: Forward Propagation



Output

$$\hat{y} = g \left( w_0 + \sum_{i=1}^m x_i w_i \right)$$

Linear combination of inputs

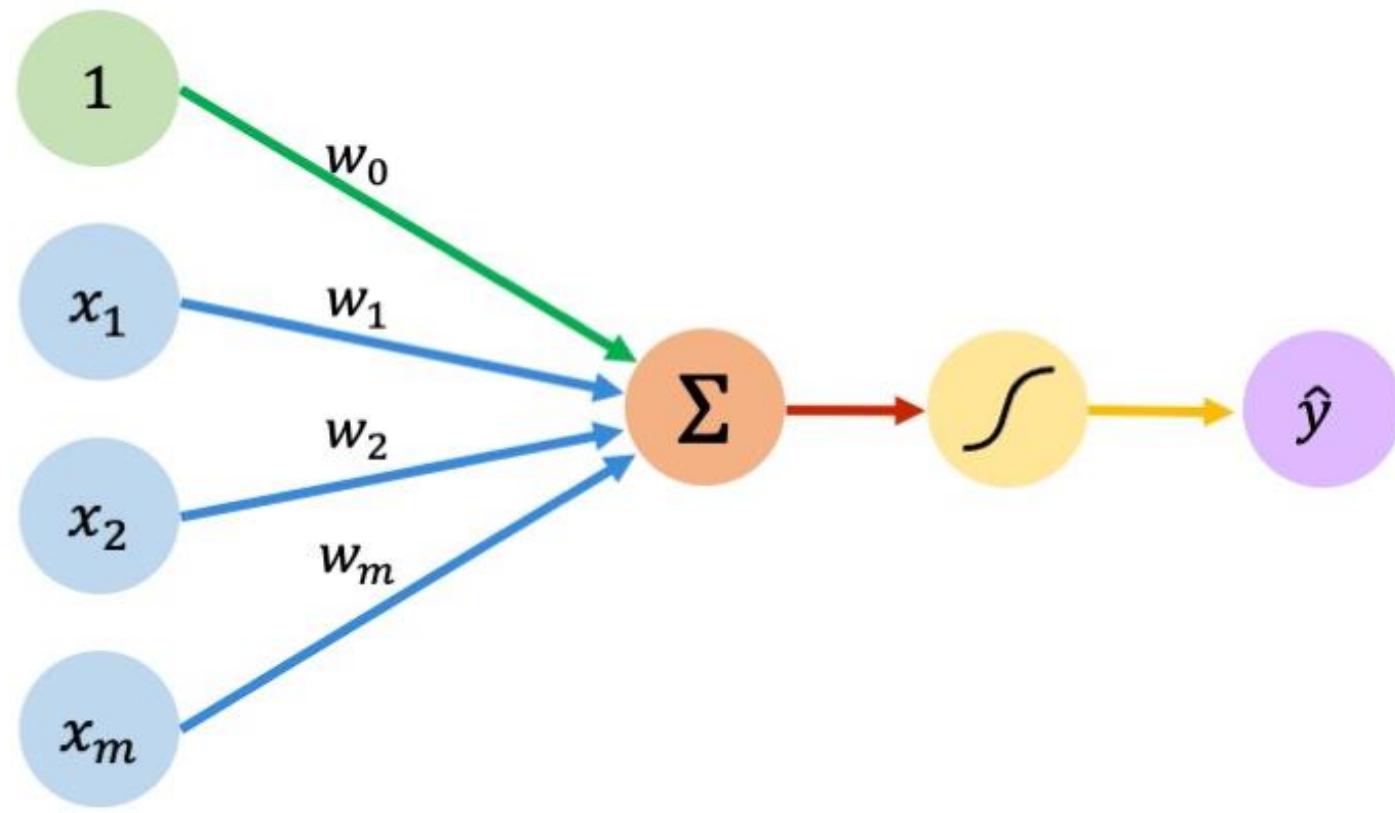
Non-linear activation function

Bias

Diagram annotations: A purple arrow points to the output  $\hat{y}$ . A red arrow points to the term  $w_0 + \sum_{i=1}^m x_i w_i$  in the equation, labeled "Linear combination of inputs". A green arrow points to the term  $w_0$ , labeled "Bias". A yellow arrow points to the activation function  $g$ , labeled "Non-linear activation function".

Inputs    Weights    Sum    Non-Linearity    Output

# The Perceptron: Forward Propagation



Inputs

Weights

Sum

Non-Linearity

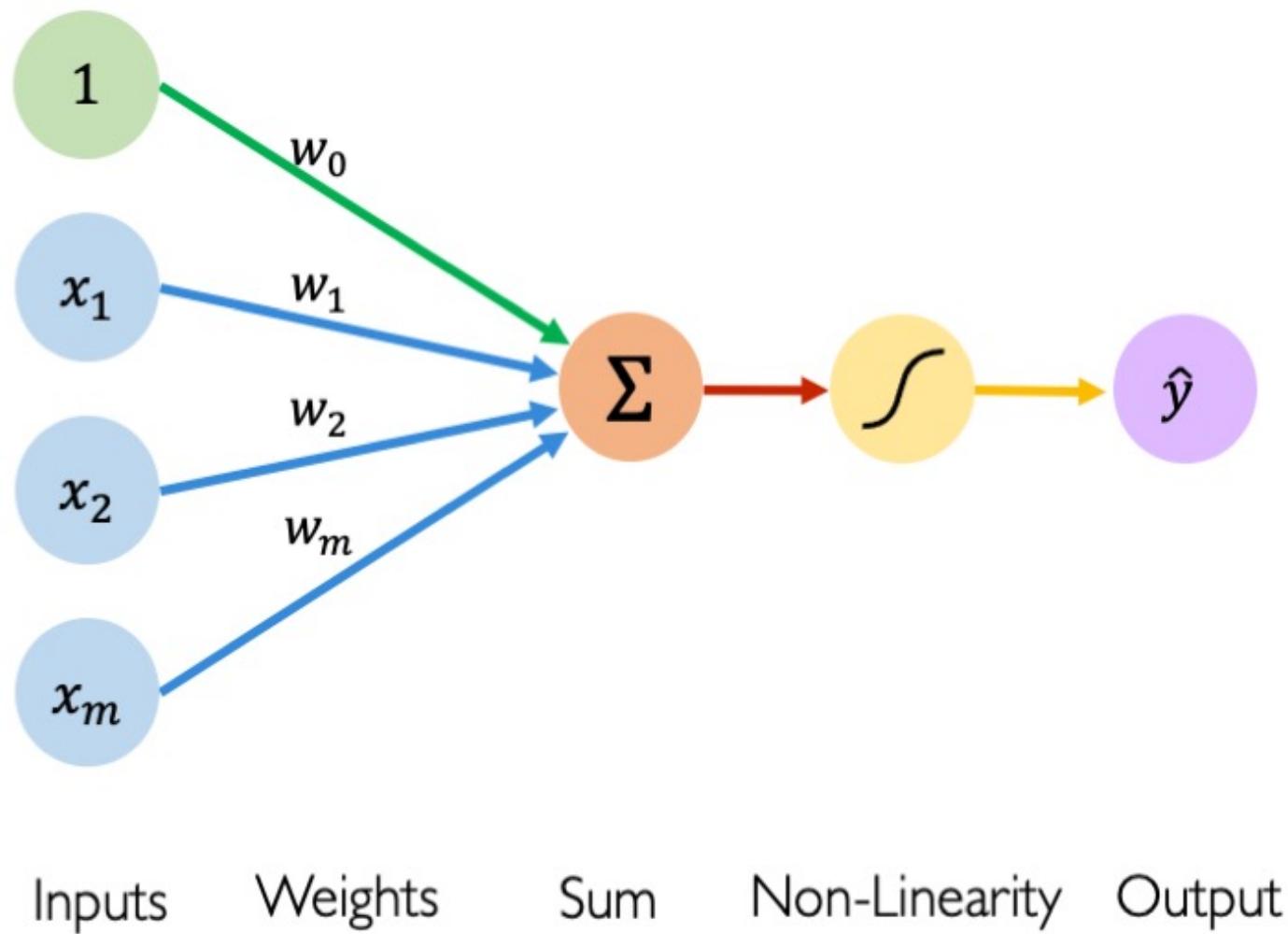
Output

$$\hat{y} = g \left( w_0 + \sum_{i=1}^m x_i w_i \right)$$

$$\hat{y} = g ( w_0 + \mathbf{X}^T \mathbf{W} )$$

where:  $\mathbf{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$  and  $\mathbf{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

# The Perceptron: Forward Propagation

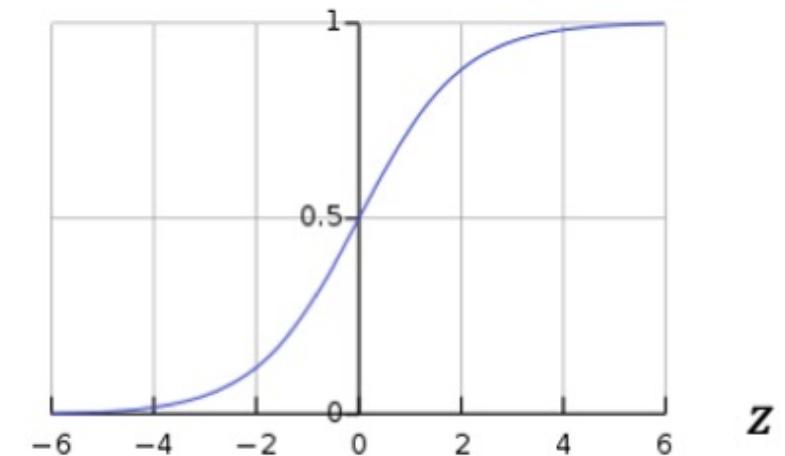


## Activation Functions

$$\hat{y} = g(w_0 + X^T W)$$

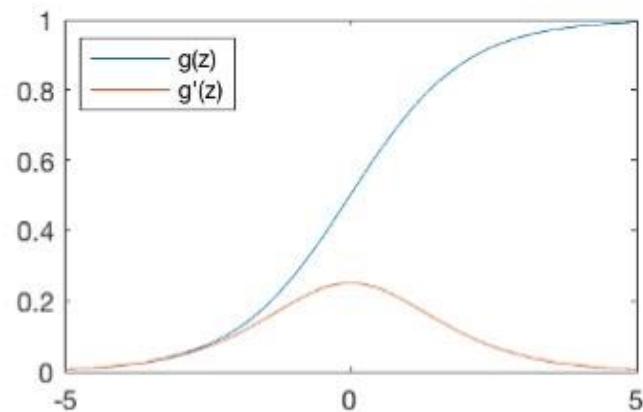
- Example: sigmoid function

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



# Common Activation Functions

Sigmoid Function

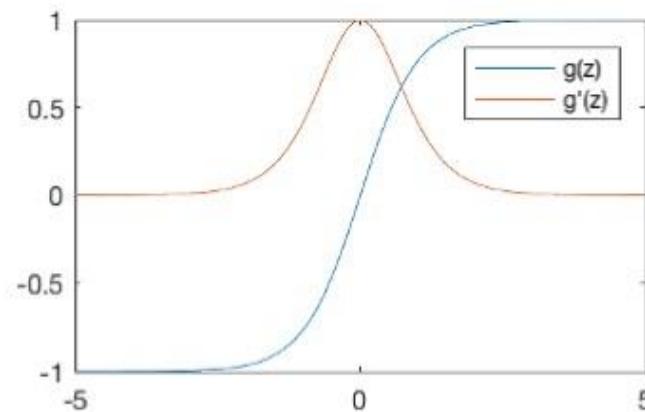


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

 `tf.math.sigmoid(z)`

Hyperbolic Tangent

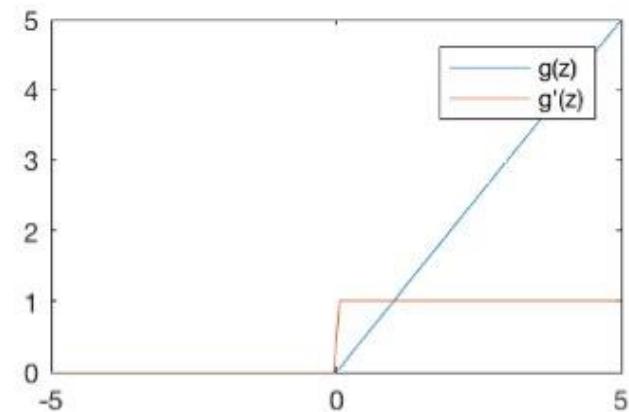


$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

 `tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



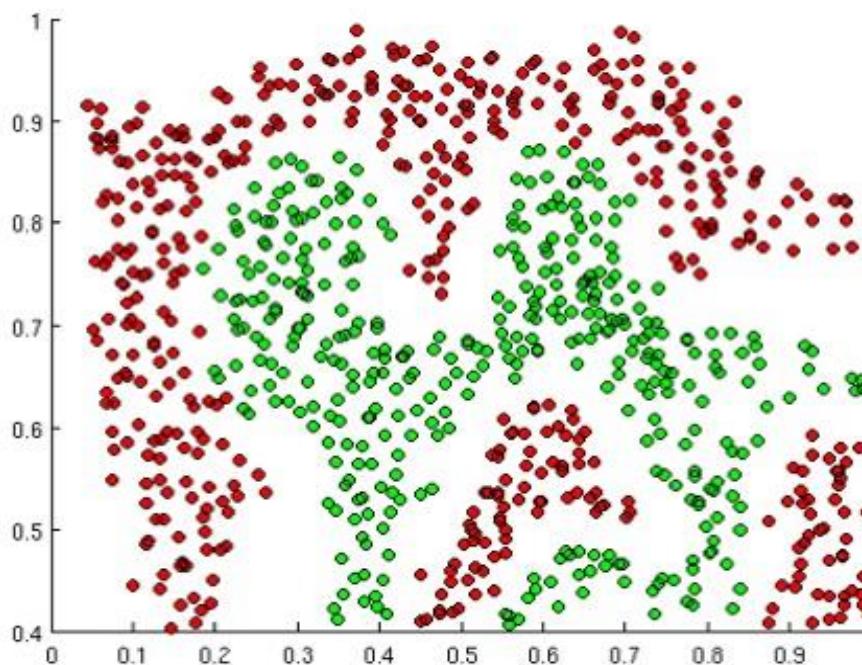
$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

 `tf.nn.relu(z)`

# Importance of Activation Functions

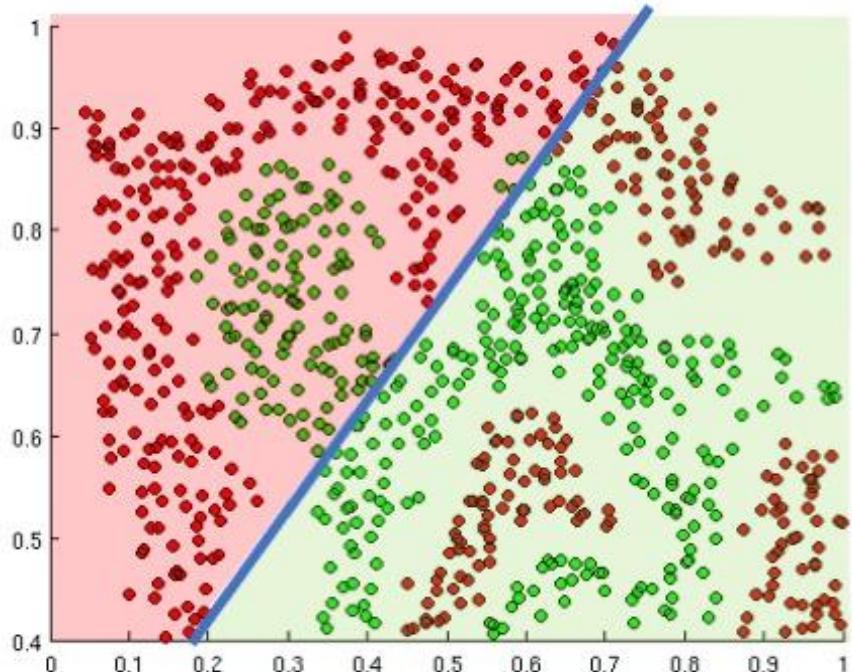
*The purpose of activation functions is to **introduce non-linearities** into the network*



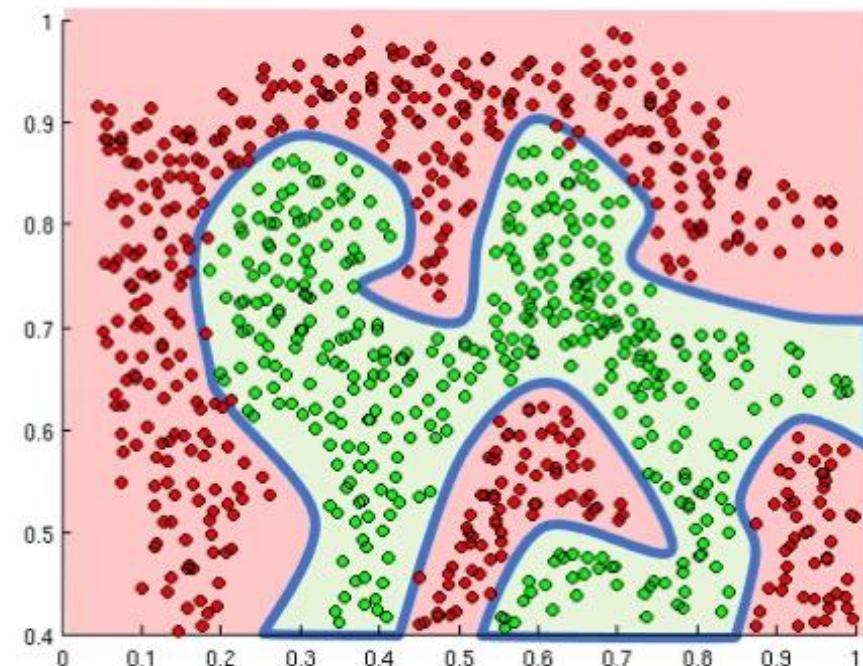
What if we wanted to build a neural network to  
distinguish green vs red points?

# Importance of Activation Functions

*The purpose of activation functions is to **introduce non-linearities** into the network*

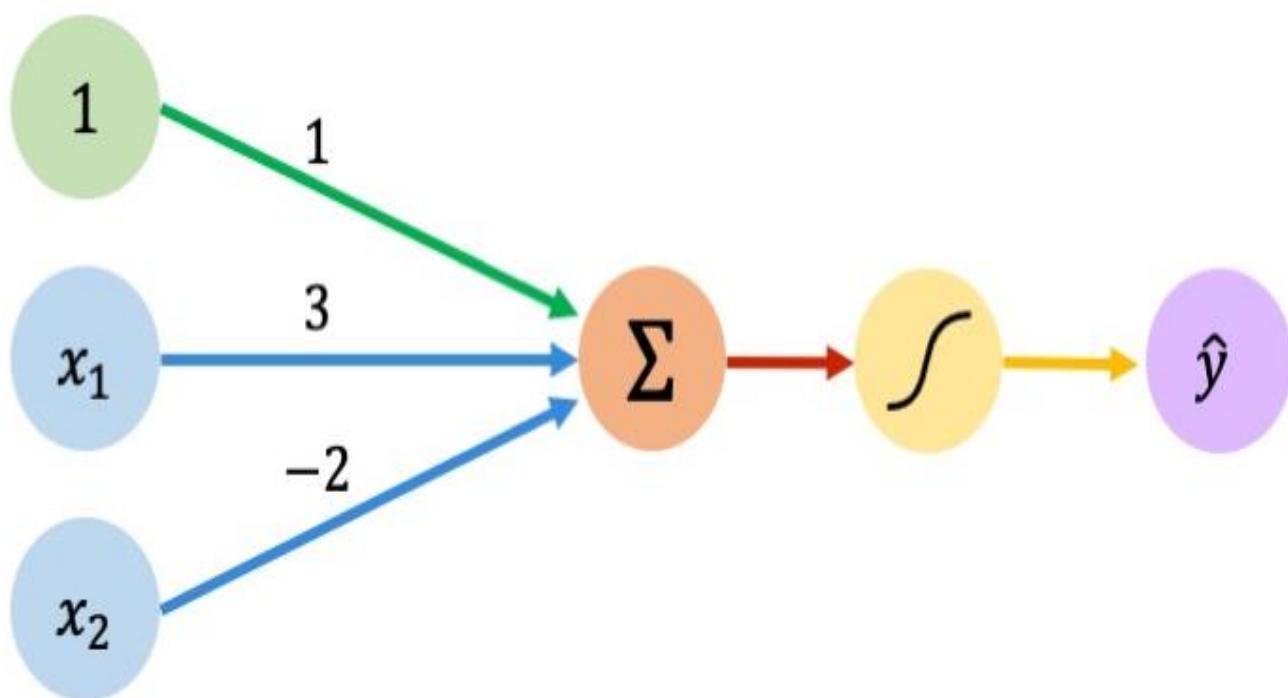


Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

# The Perceptron: Example

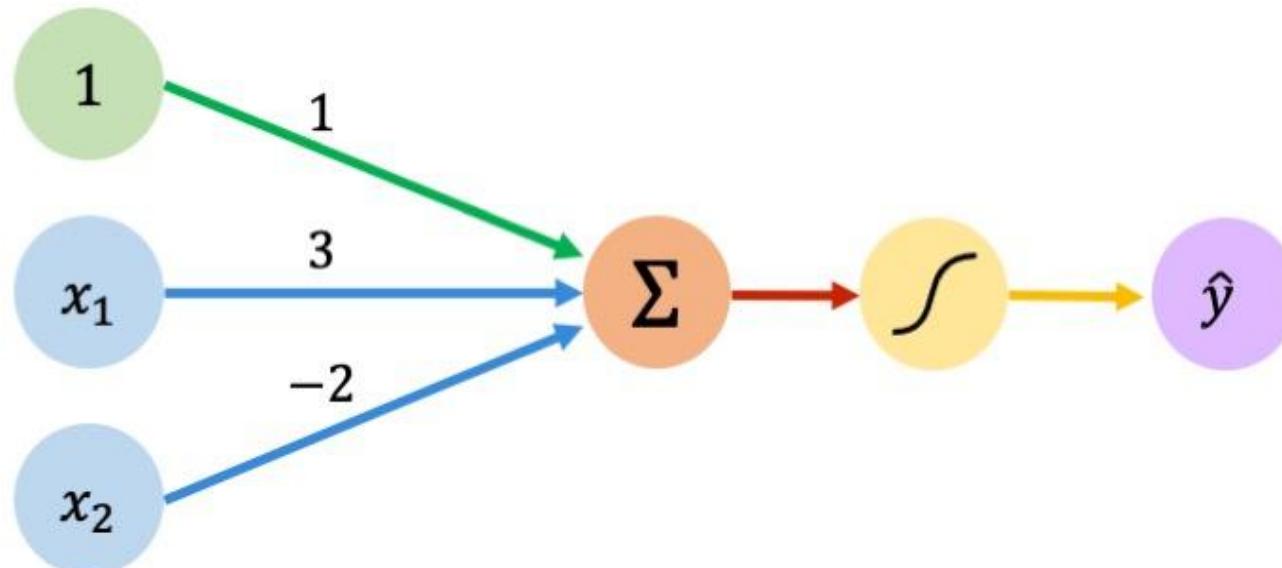


We have:  $w_0 = 1$  and  $\mathbf{w} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

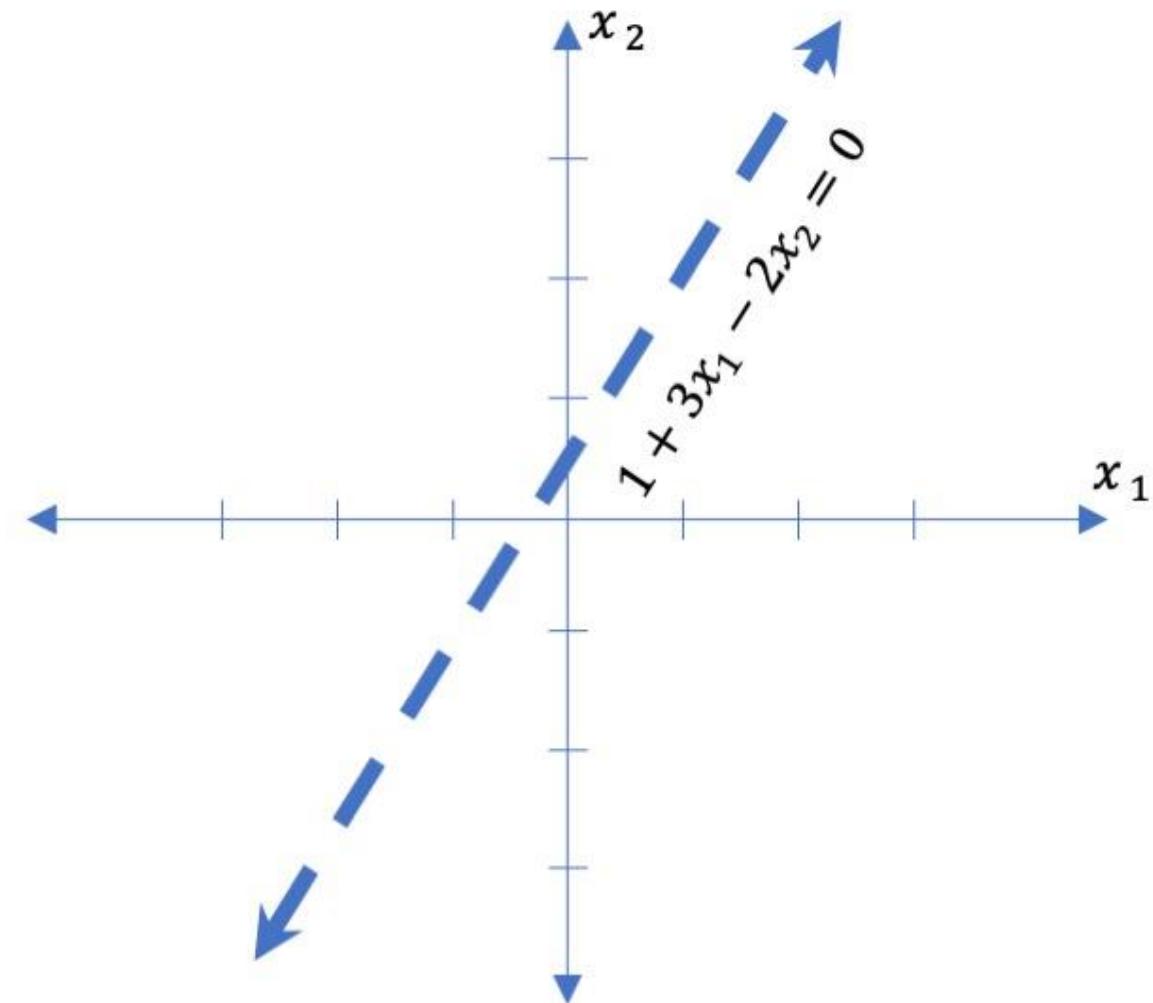
$$\begin{aligned}\hat{y} &= g(w_0 + \mathbf{X}^T \mathbf{w}) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g(1 + 3x_1 - 2x_2)\end{aligned}$$

This is just a line in 2D!

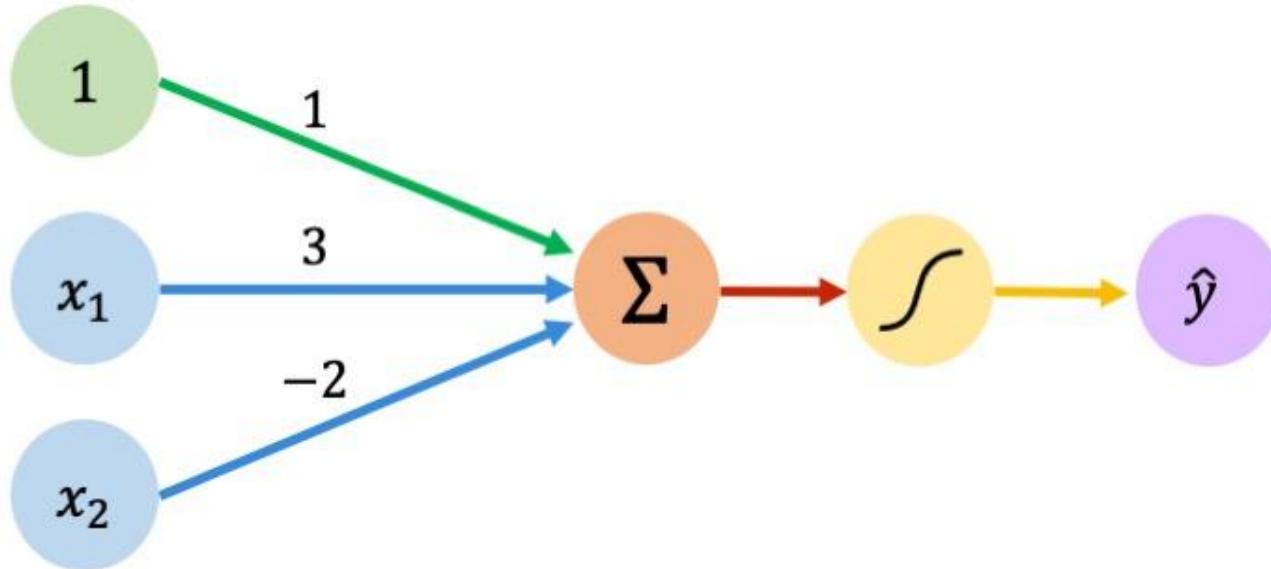
# The Perceptron: Example



$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



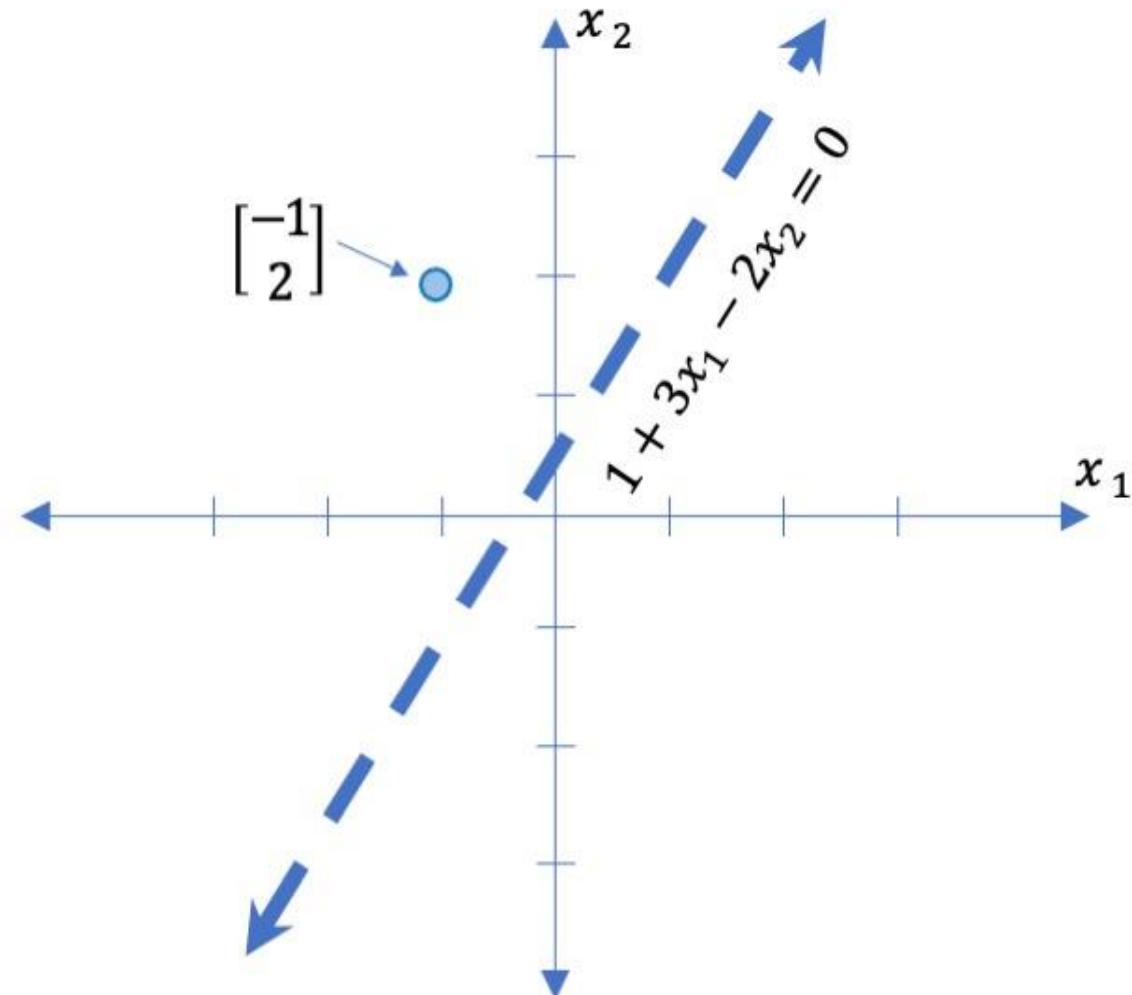
# The Perceptron: Example



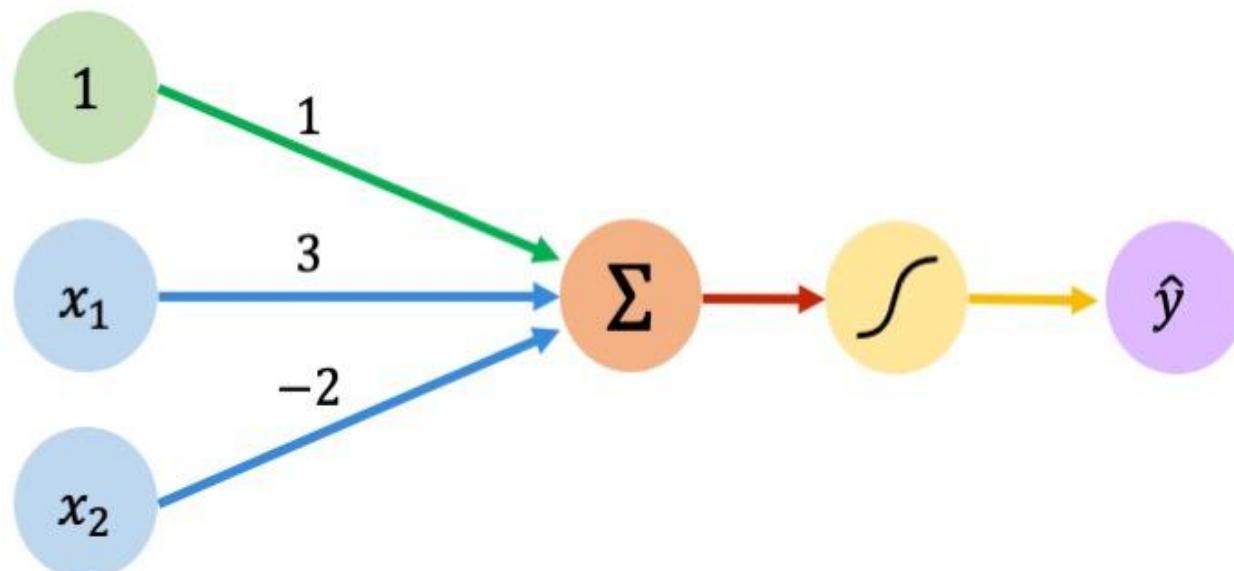
Assume we have input:  $\mathbf{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$$\begin{aligned}\hat{y} &= g(1 + (3 * -1) - (2 * 2)) \\ &= g(-6) \approx 0.002\end{aligned}$$

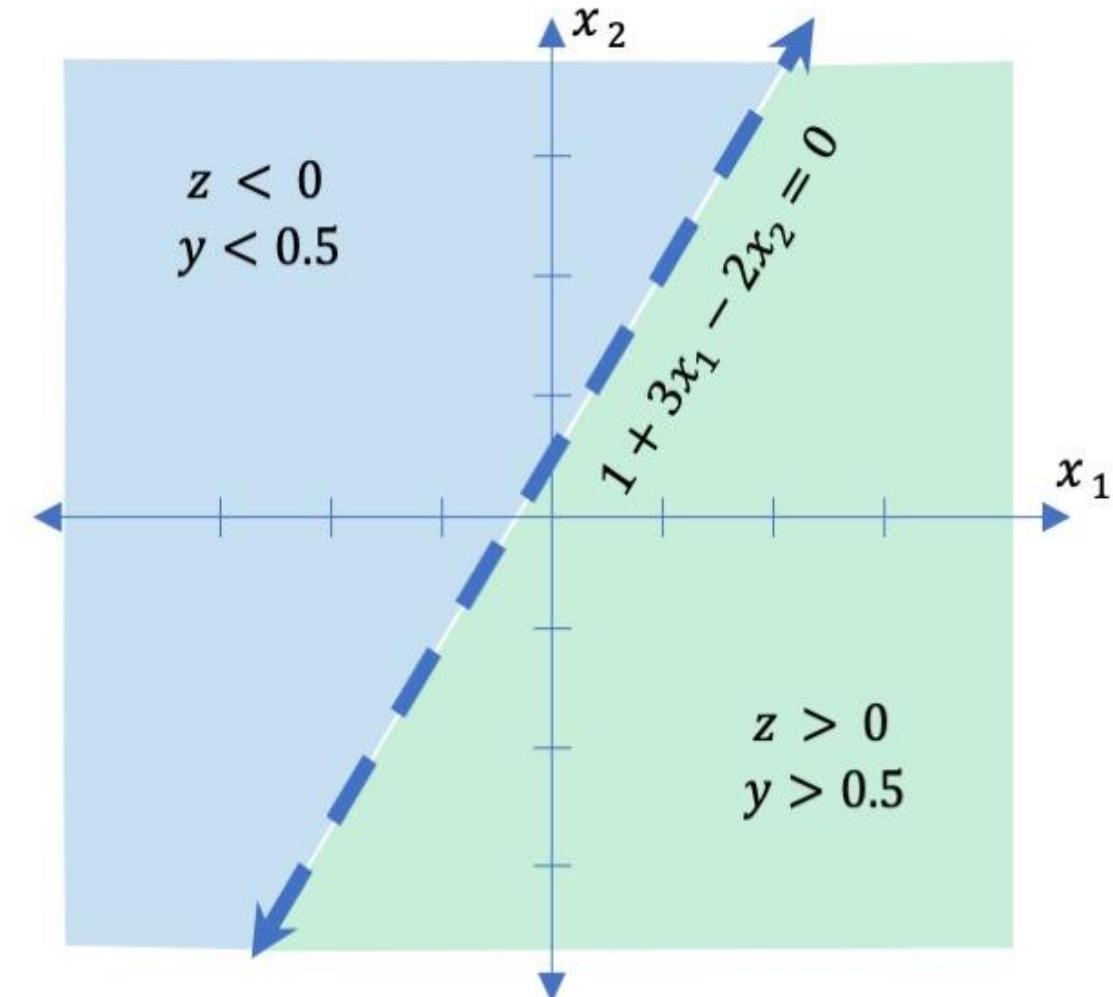
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



# The Perceptron: Example



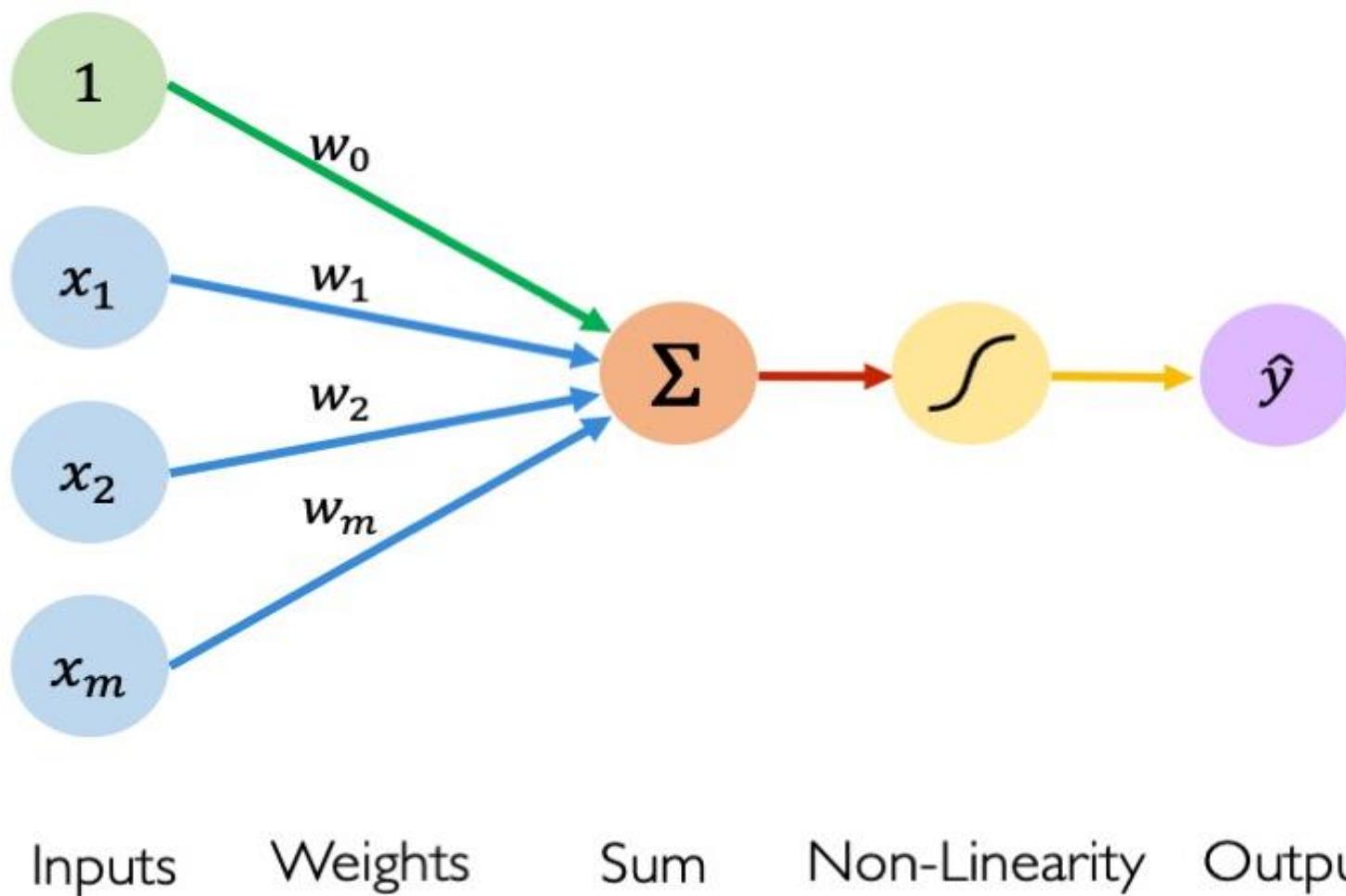
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



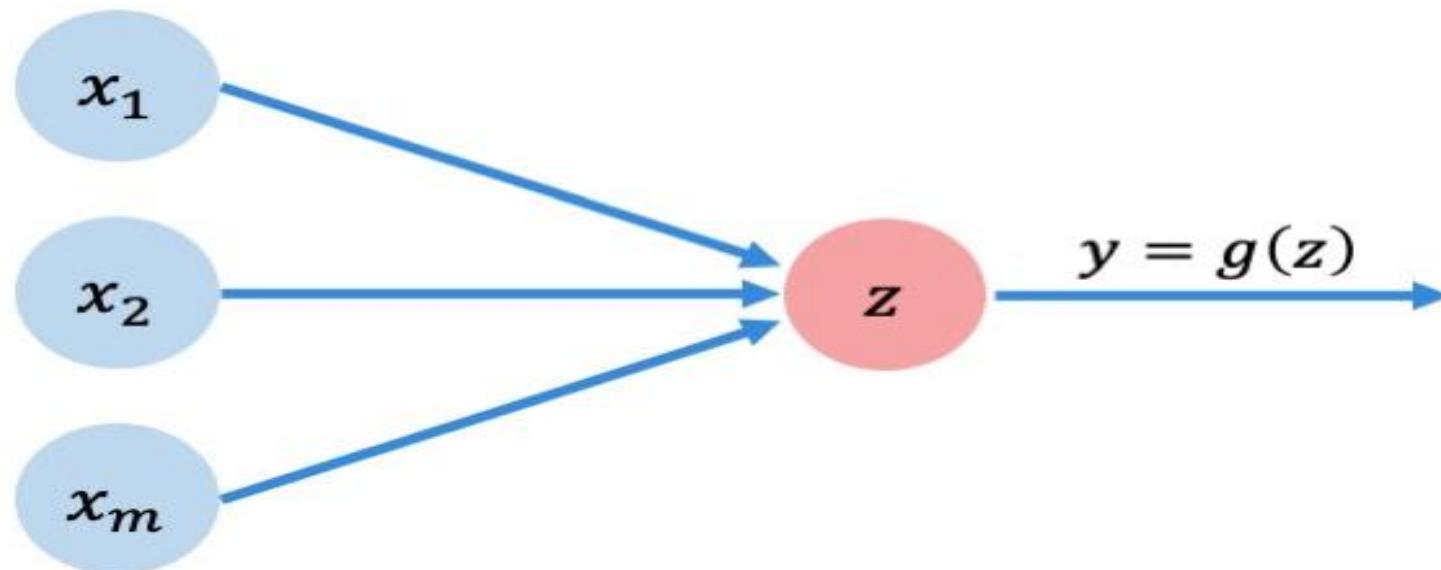
# Building Neural Networks with Perceptrons

# The Perceptron: Simplified

$$\hat{y} = g(w_0 + \mathbf{X}^T \mathbf{w})$$



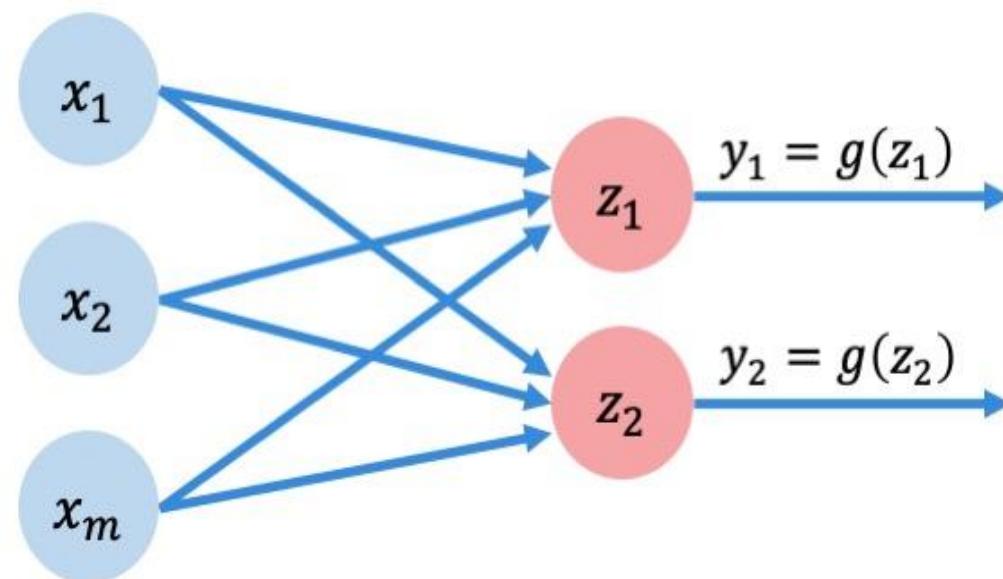
# The Perceptron: Simplified



$$z = w_0 + \sum_{j=1}^m x_j w_j$$

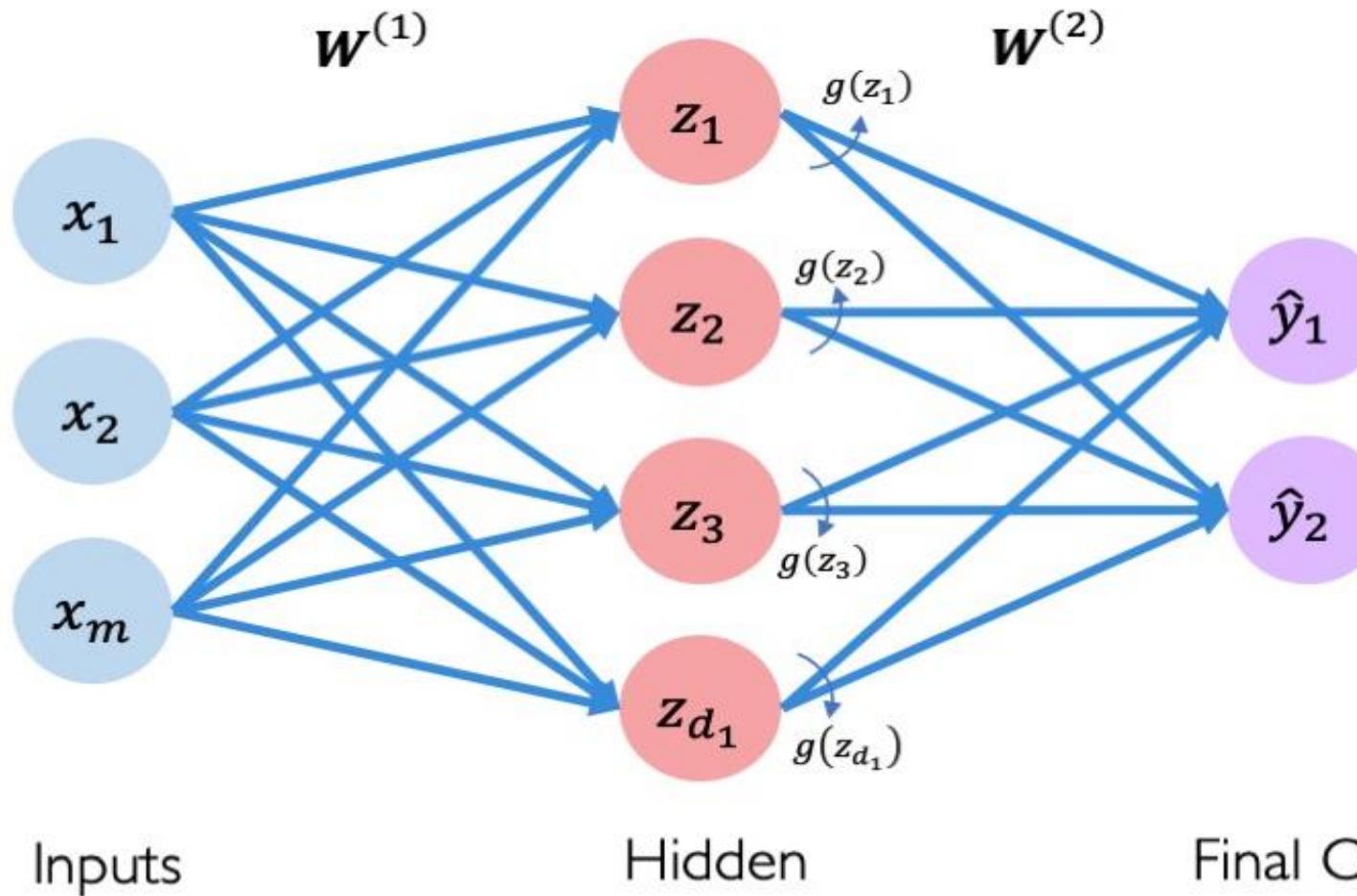
# Multi Output Perceptron

Because all inputs are densely connected to all outputs, these layers are called **Dense** layers

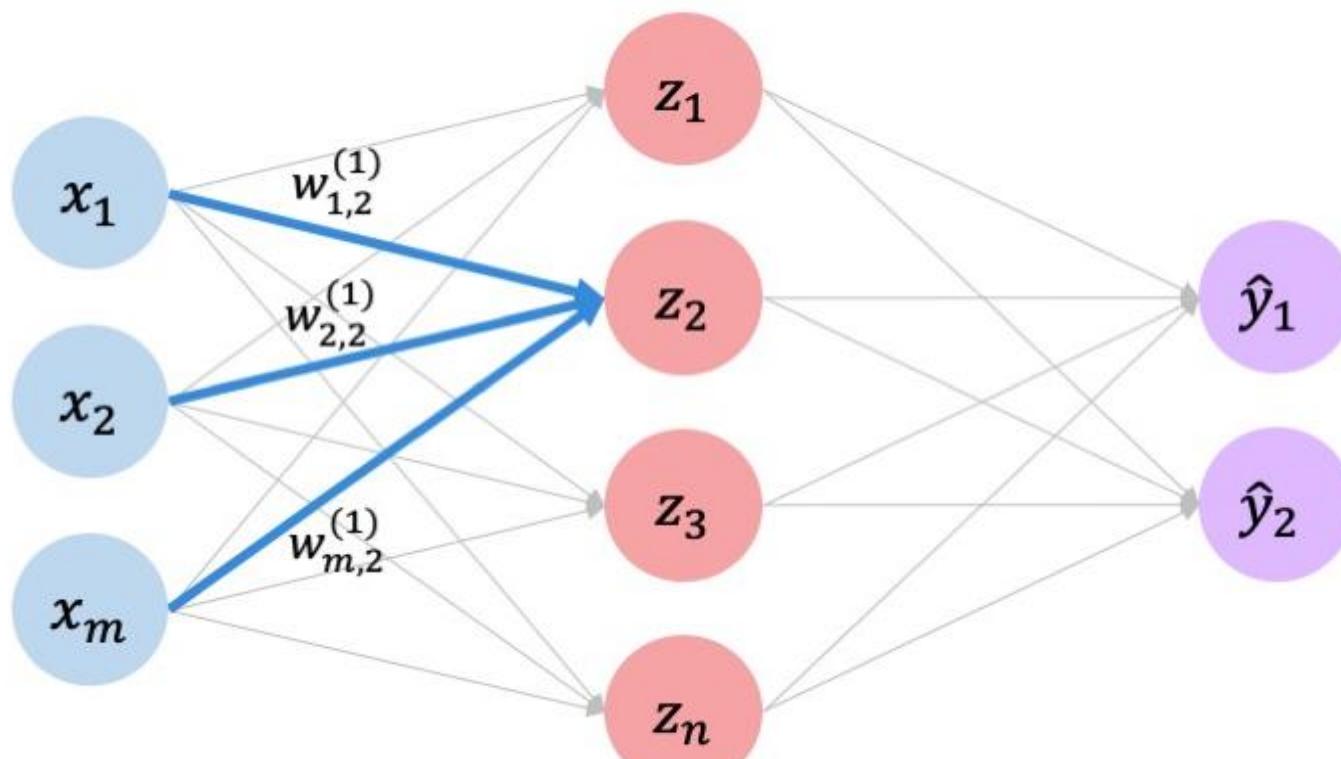


$$z_i = w_{0,i} + \sum_{j=1}^m x_j w_{j,i}$$

# Single Layer Neural Network

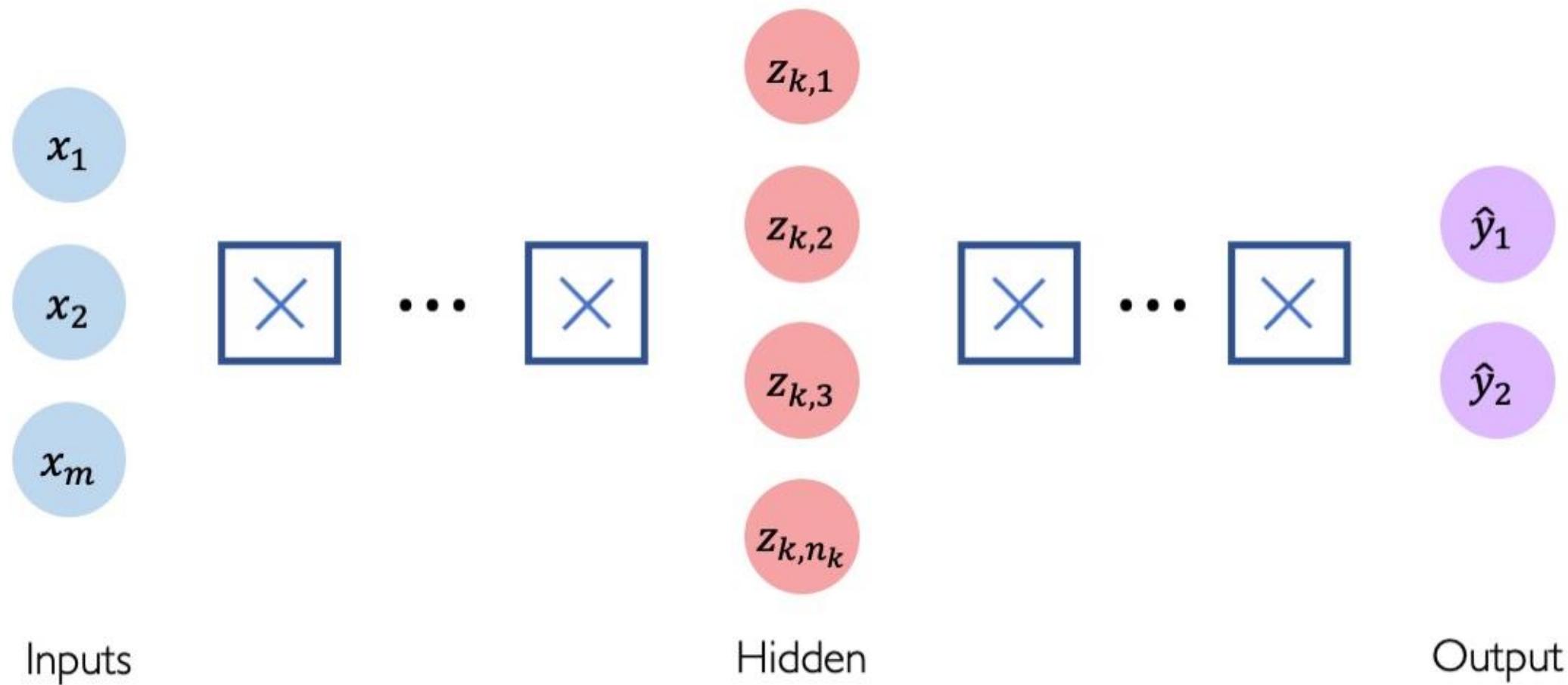


# Single Layer Neural Network



$$\begin{aligned} z_2 &= w_{0,2}^{(1)} + \sum_{j=1}^m x_j w_{j,2}^{(1)} \\ &= w_{0,2}^{(1)} + x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + x_m w_{m,2}^{(1)} \end{aligned}$$

# Deep Neural Network



$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j}) w_{j,i}^{(k)}$$

# Applying Neural Networks

# Example Problem

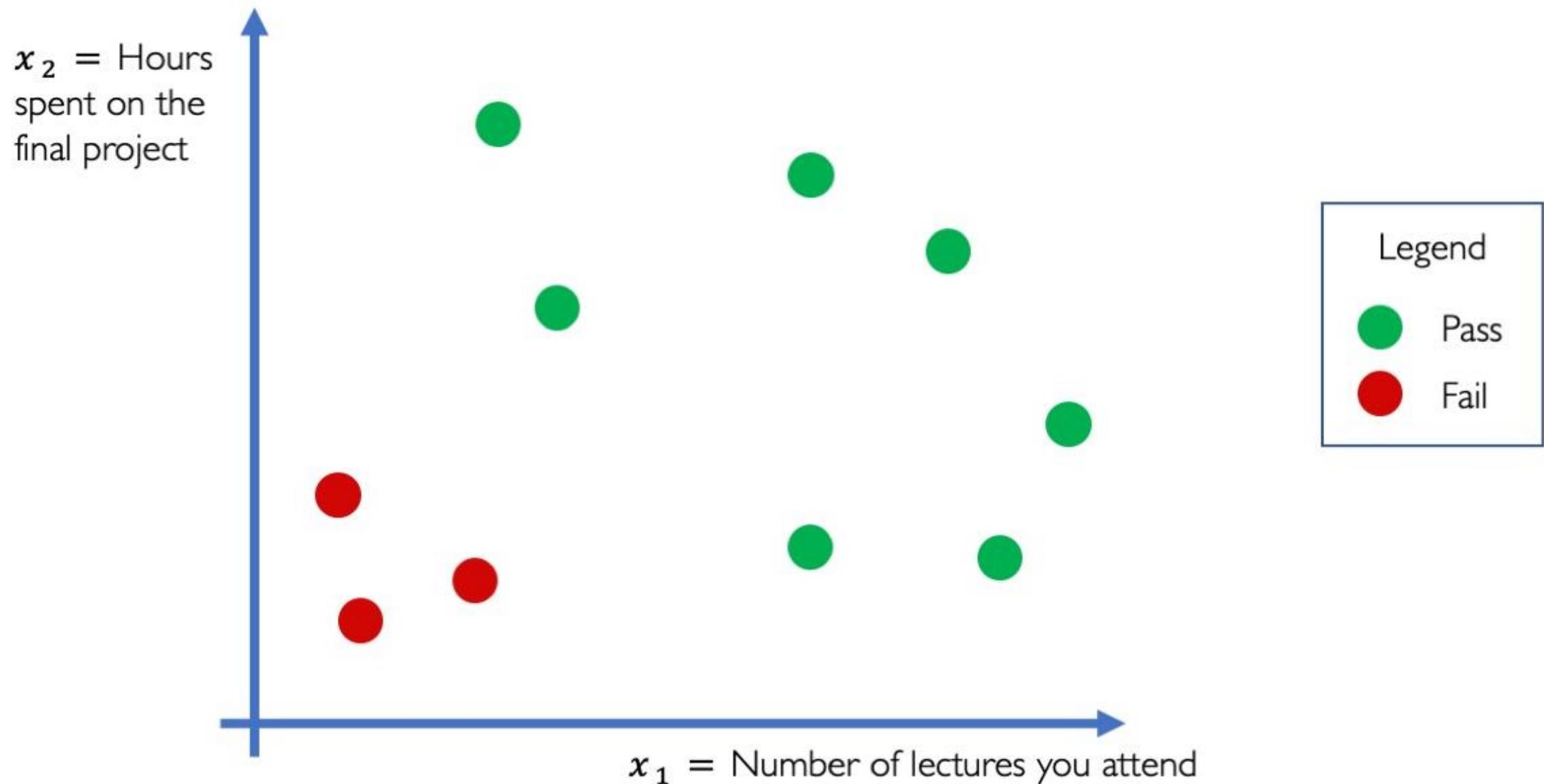
Will I pass this class?

Let's start with a simple two feature model

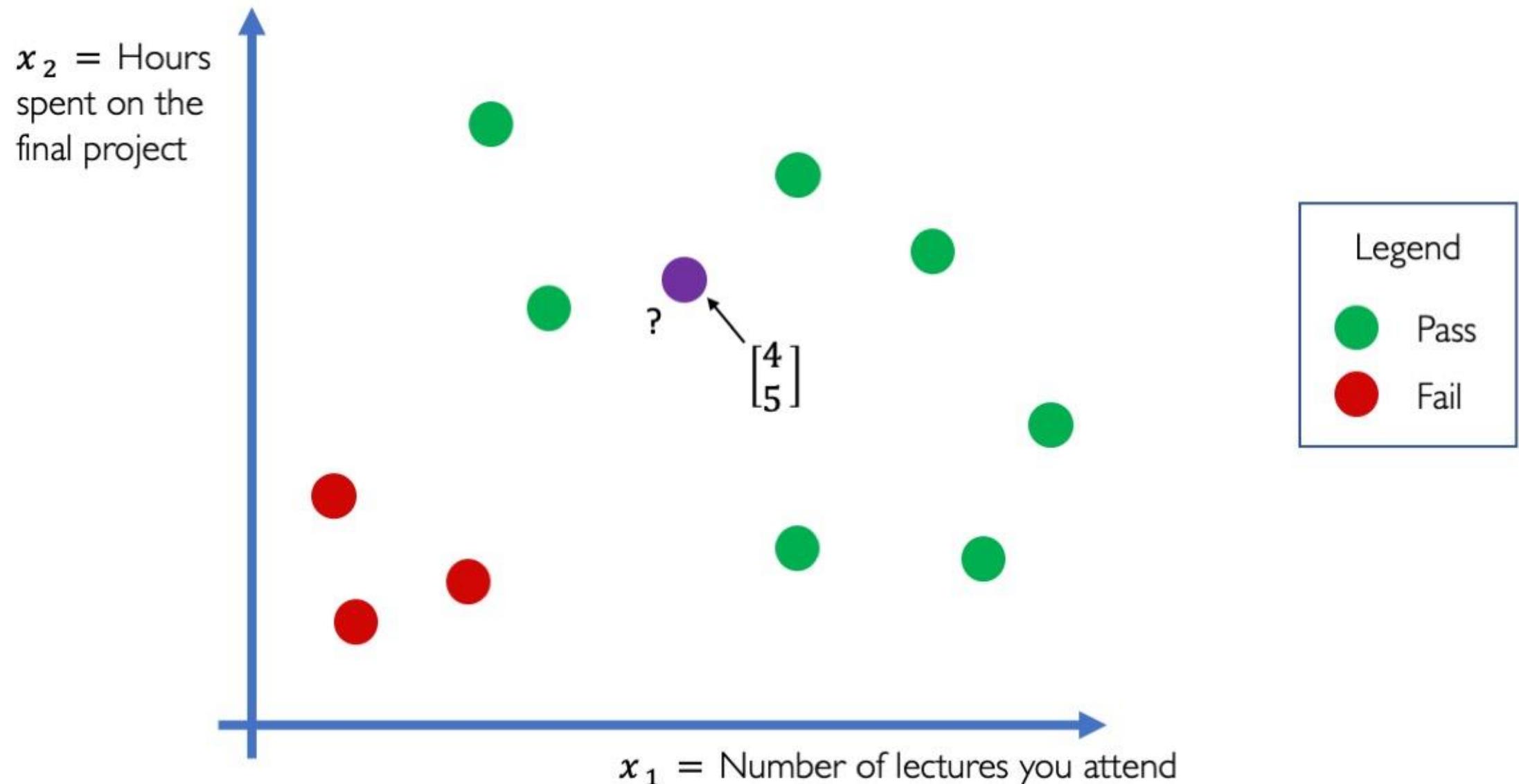
$x_1$  = Number of lectures you attend

$x_2$  = Hours spent on the final project

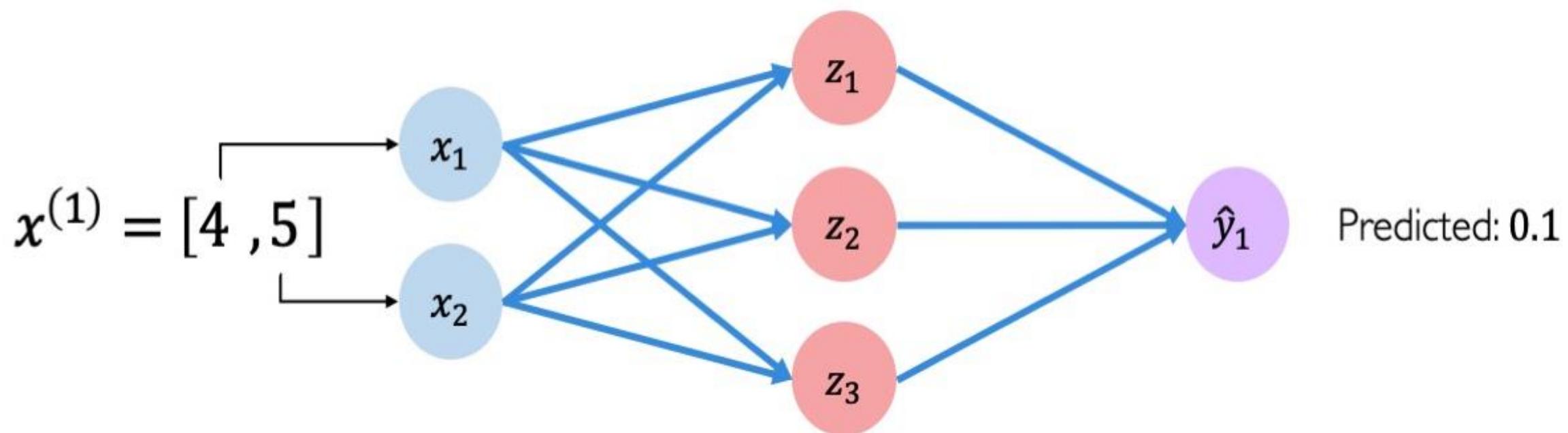
# Example Problem: Will I pass this class?



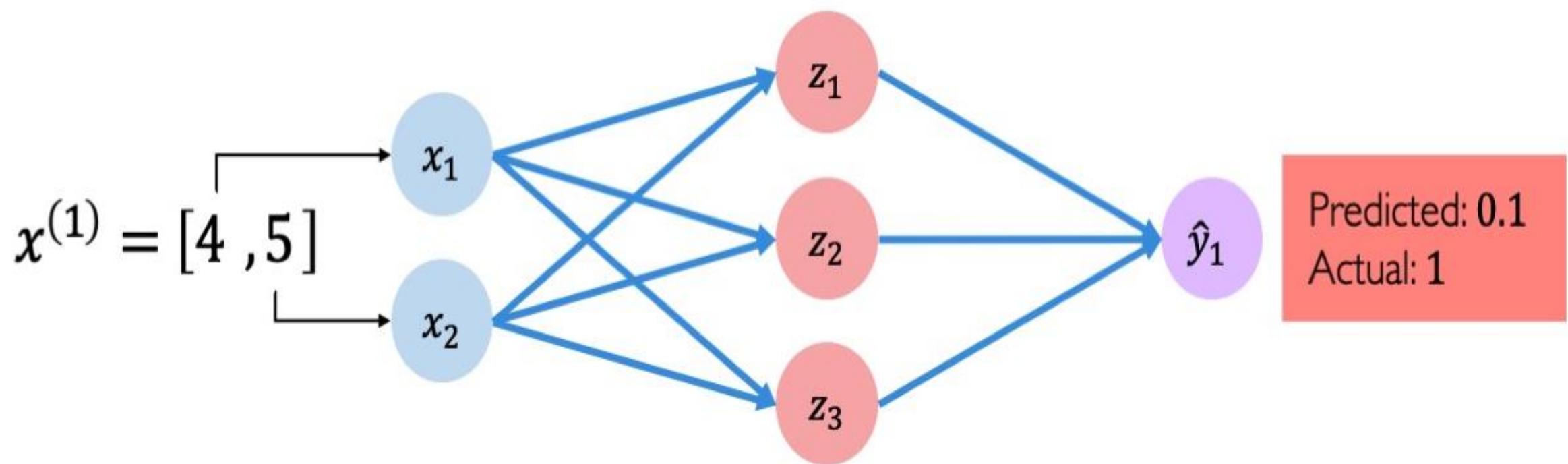
# Example Problem: Will I pass this class?



# Example Problem: Will I pass this class?

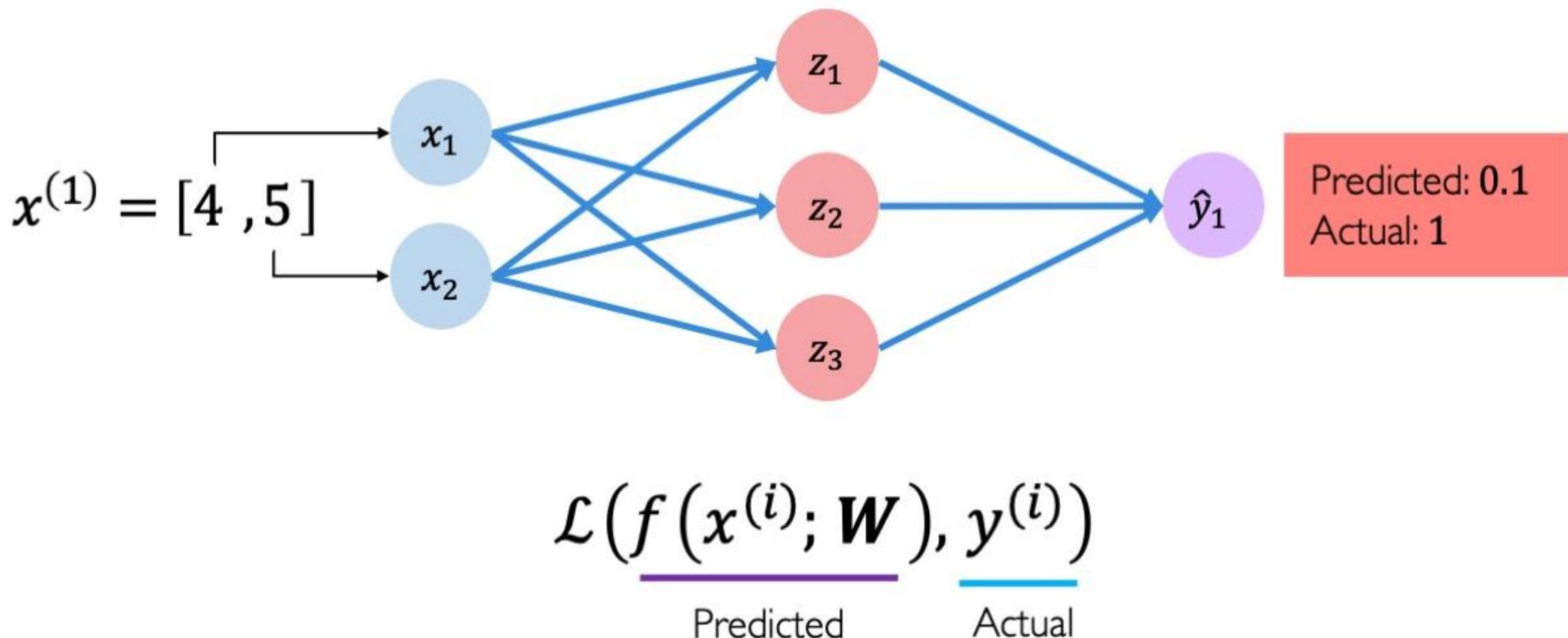


# Example Problem: Will I pass this class?



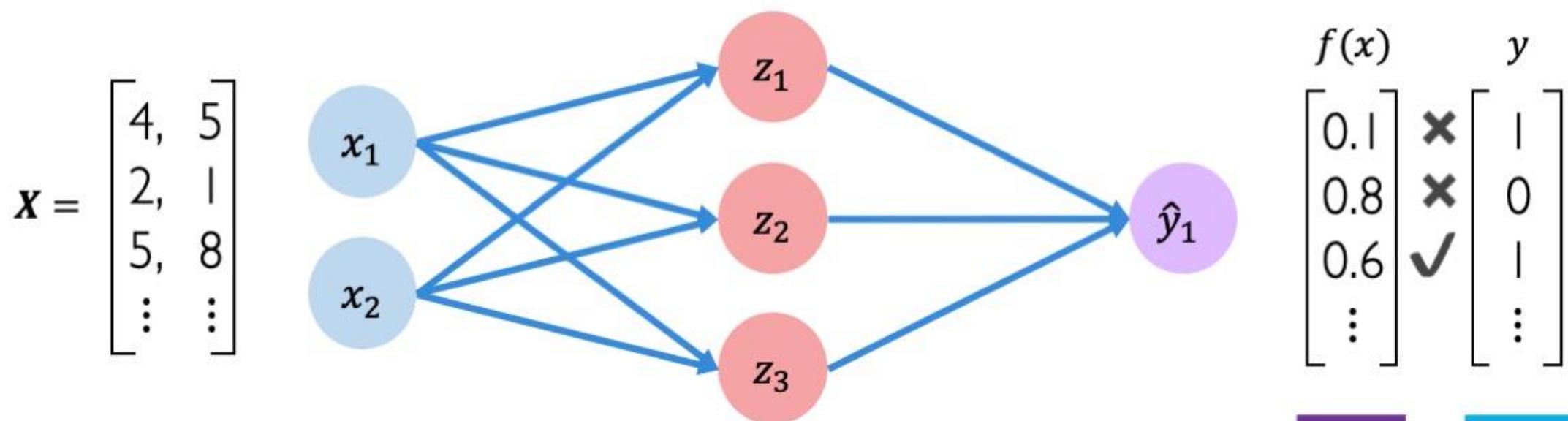
# Quantifying Loss

The *loss* of our network measures the cost incurred from incorrect predictions



# Empirical Loss

The **empirical loss** measures the total loss over our entire dataset



Also known as:

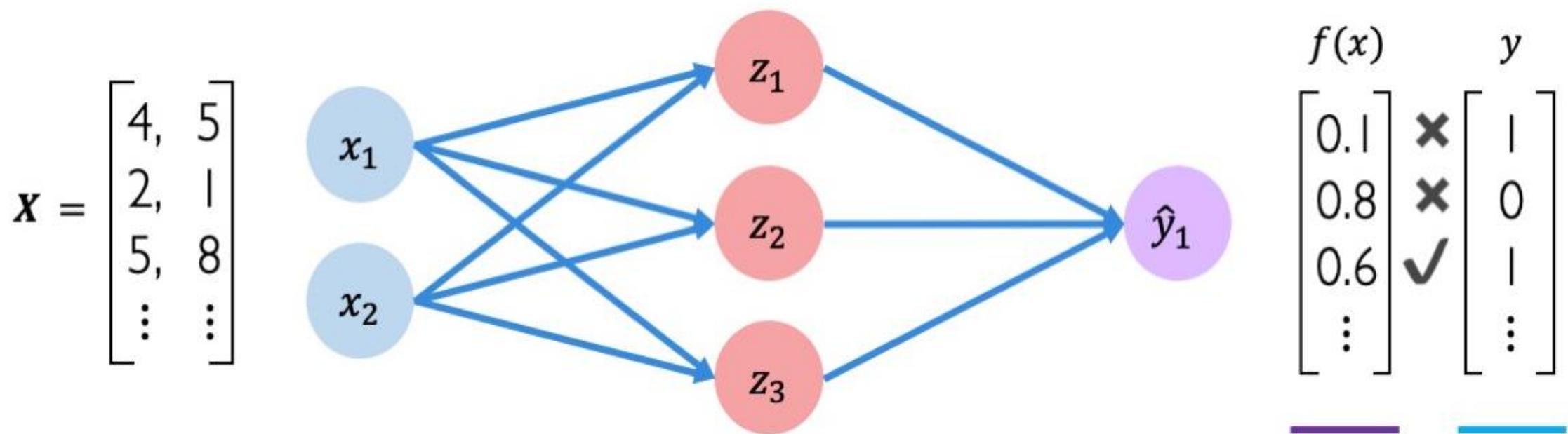
- Objective function
- Cost function
- Empirical Risk

$\curvearrowleft$  
$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(\mathbf{x}^{(i)}; \mathbf{W}), \mathbf{y}^{(i)})$$

Predicted      Actual

# Binary Cross Entropy Loss

**Cross entropy loss** can be used with models that output a probability between 0 and 1



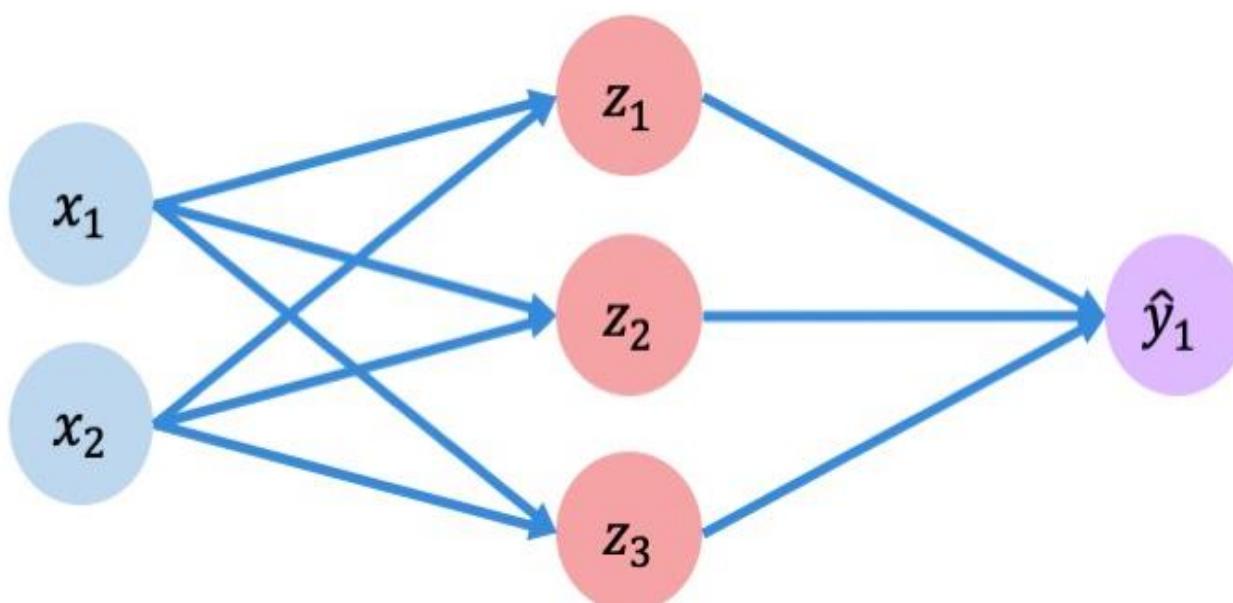
$$J(\mathbf{W}) = -\frac{1}{n} \sum_{i=1}^n \underbrace{y^{(i)} \log(f(x^{(i)}; \mathbf{W}))}_{\text{Actual}} + \underbrace{(1 - y^{(i)}) \log(1 - f(x^{(i)}; \mathbf{W}))}_{\text{Actual}}$$

Predicted      Actual      Predicted      Actual

# Mean Squared Error Loss

**Mean squared error loss** can be used with regression models that output continuous real numbers

$$X = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots, & \vdots \end{bmatrix}$$



$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \left( \underline{y^{(i)}} - \underline{f(x^{(i)}; \mathbf{W})} \right)^2$$

Actual      Predicted

$f(x)$	$y$
30	✗ 90
80	✗ 20
85	✓ 95
$\vdots$	$\vdots$

—      —

Final Grades  
(percentage)

# Training Neural Networks

# Loss Optimization

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

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

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

# Loss Optimization

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

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

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

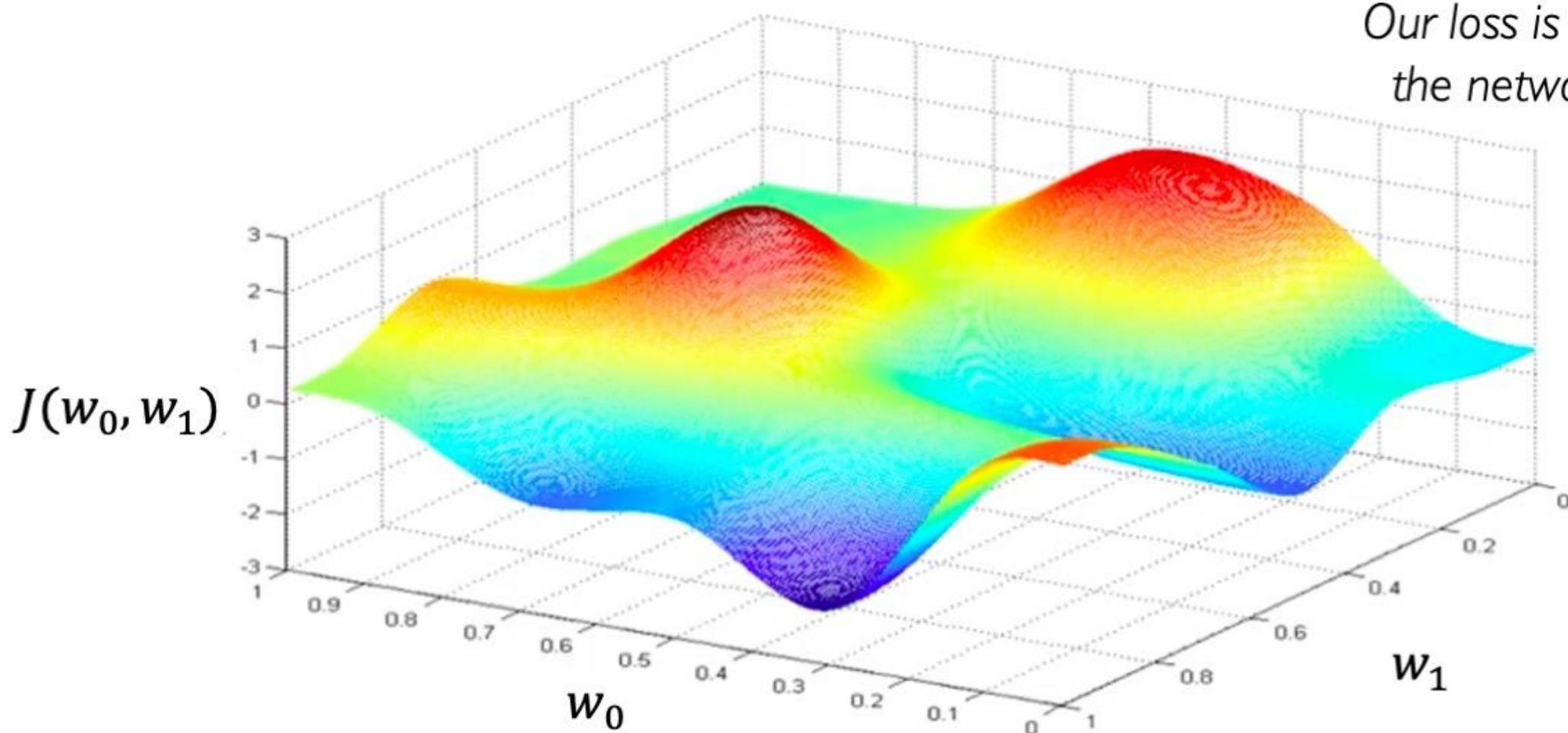


Remember:

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

# Loss Optimization

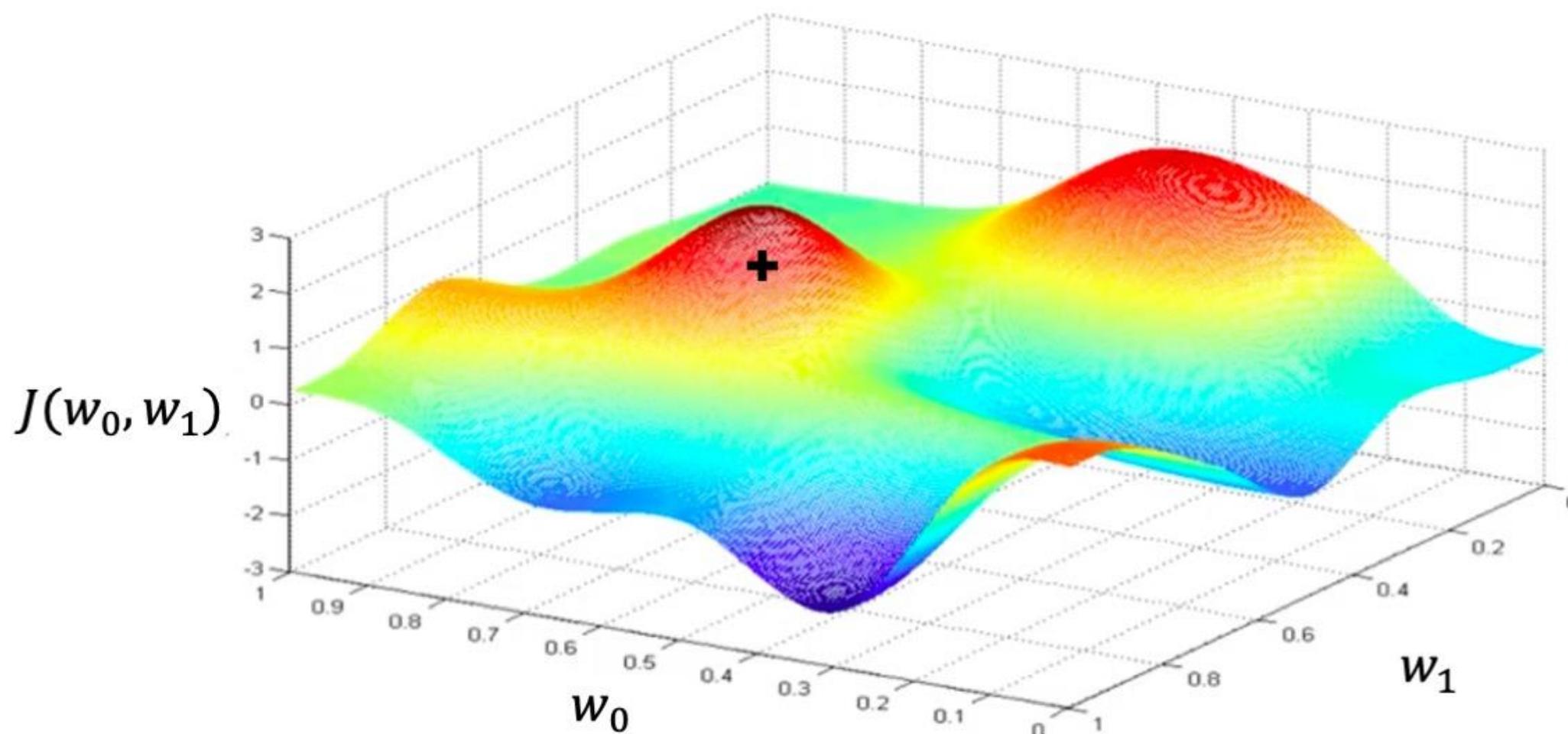
$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$



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

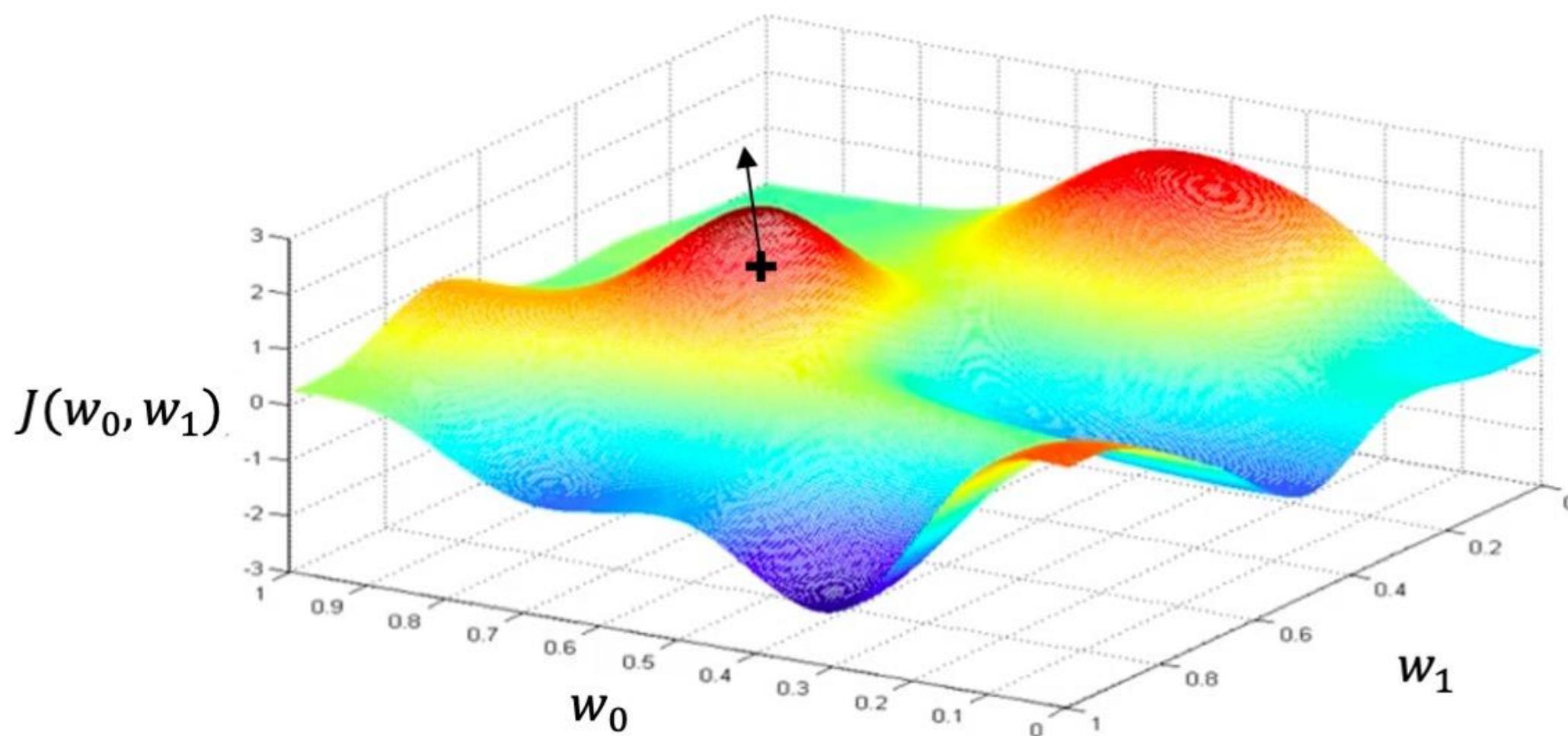
# Loss Optimization

Randomly pick an initial  $(w_0, w_1)$



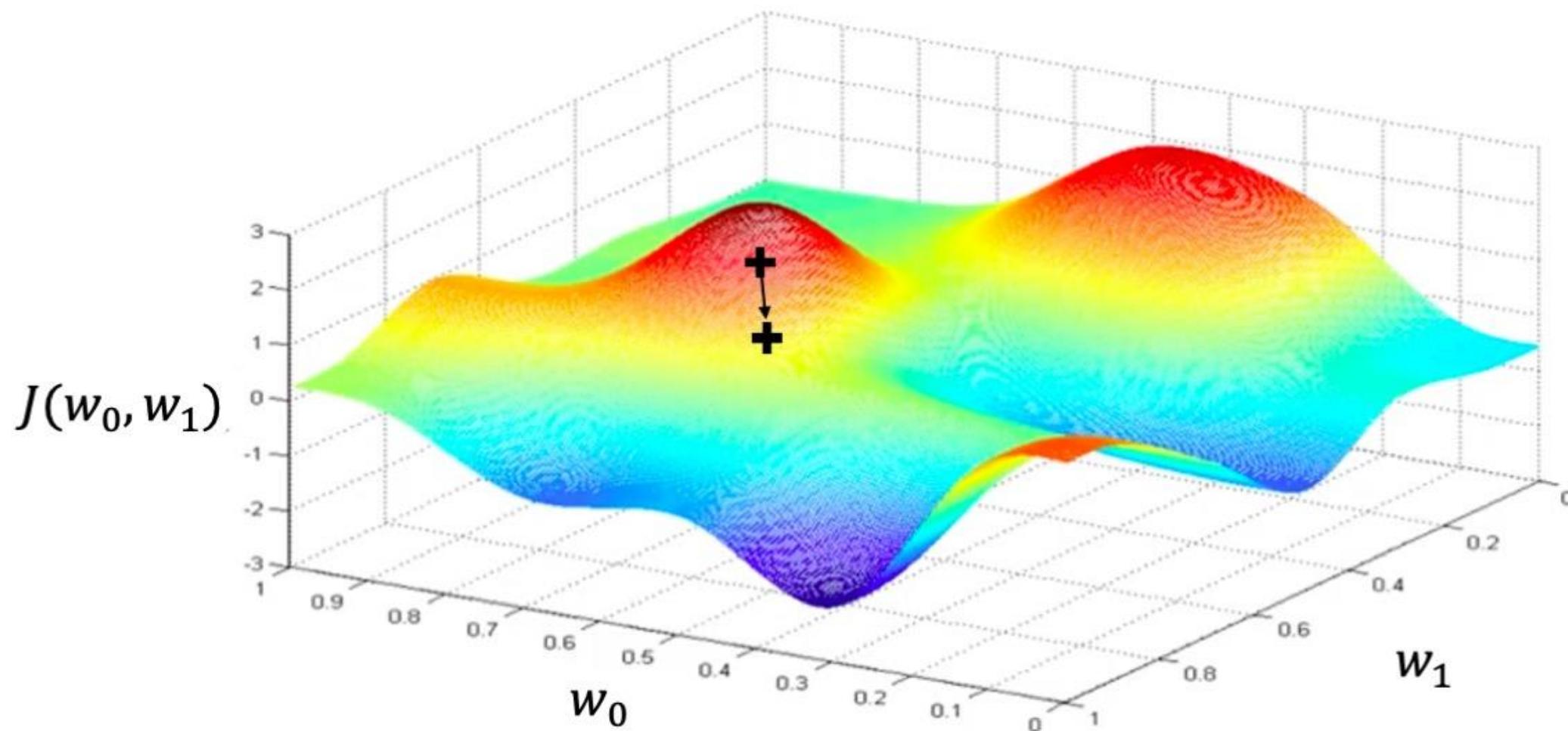
# Loss Optimization

Compute gradient,  $\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$



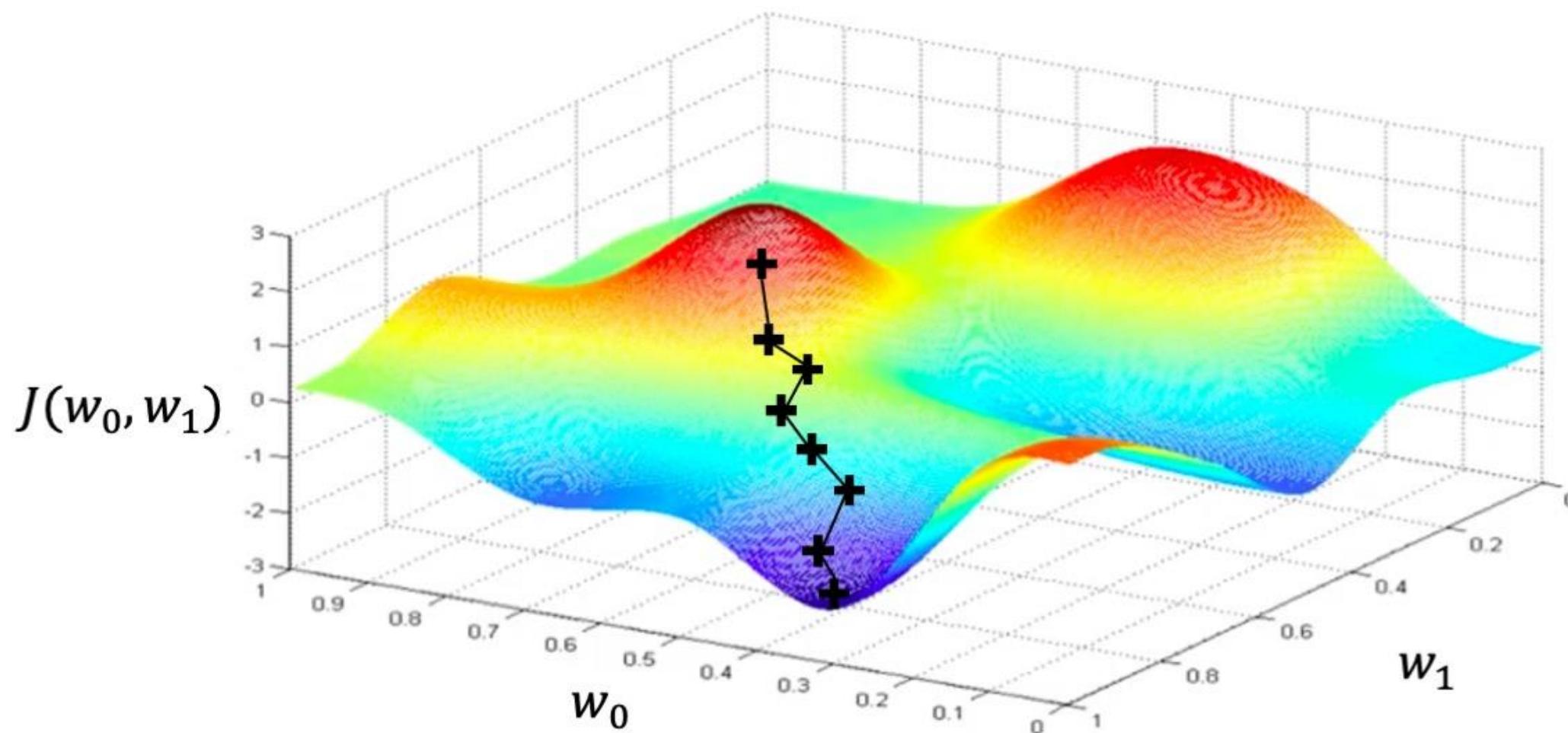
# Loss Optimization

Take small step in opposite direction of gradient



# Gradient Descent

Repeat until convergence

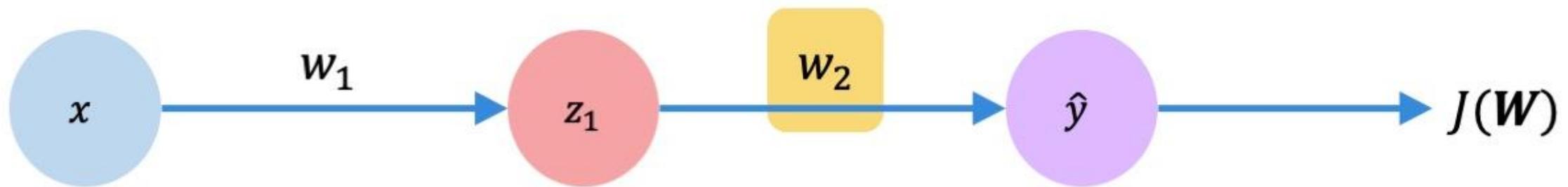


# Gradient Descent

## Algorithm

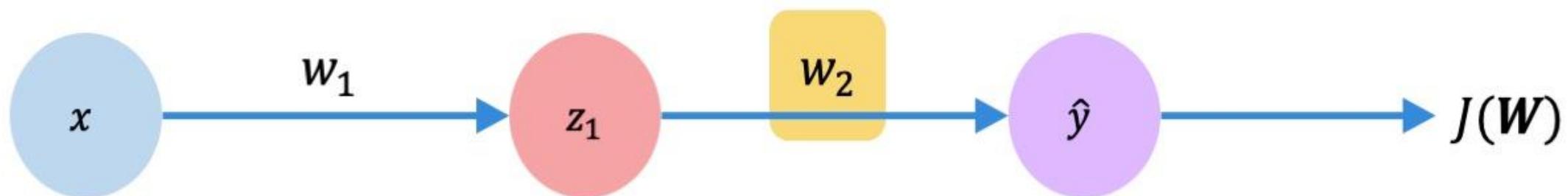
1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient,  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

# Computing Gradients: Backpropagation



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

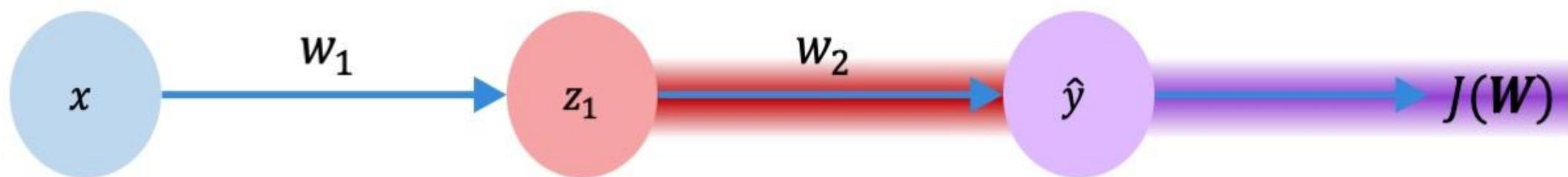
# Computing Gradients: Backpropagation



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

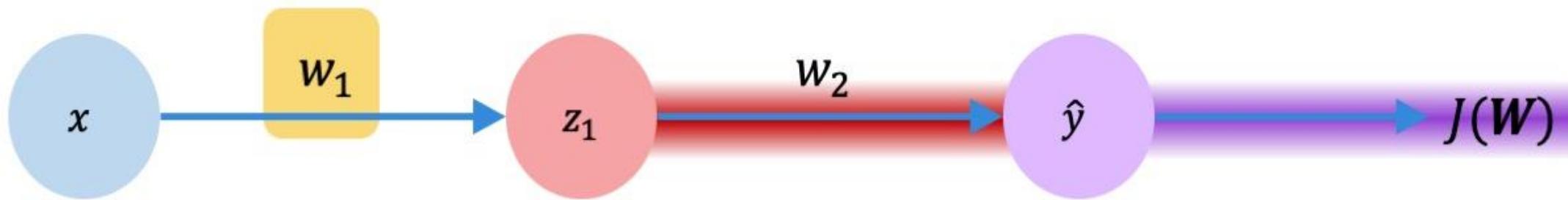
Let's use the chain rule!

# 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

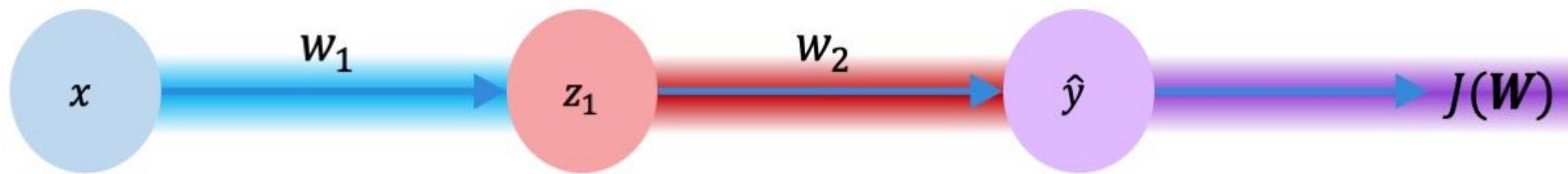


$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

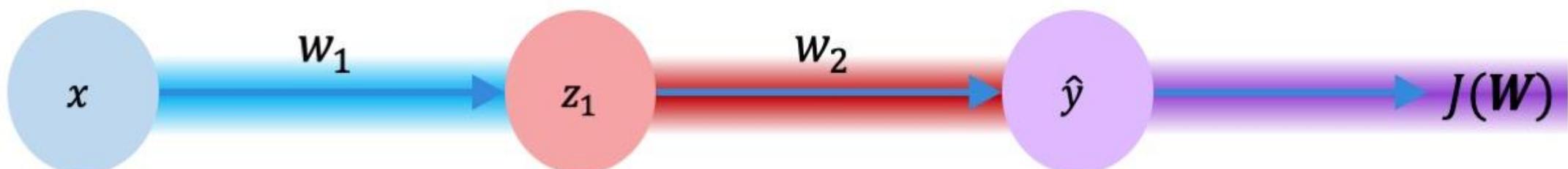
Apply chain rule!

# Computing Gradients: Backpropagation



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

# Computing Gradients: Backpropagation

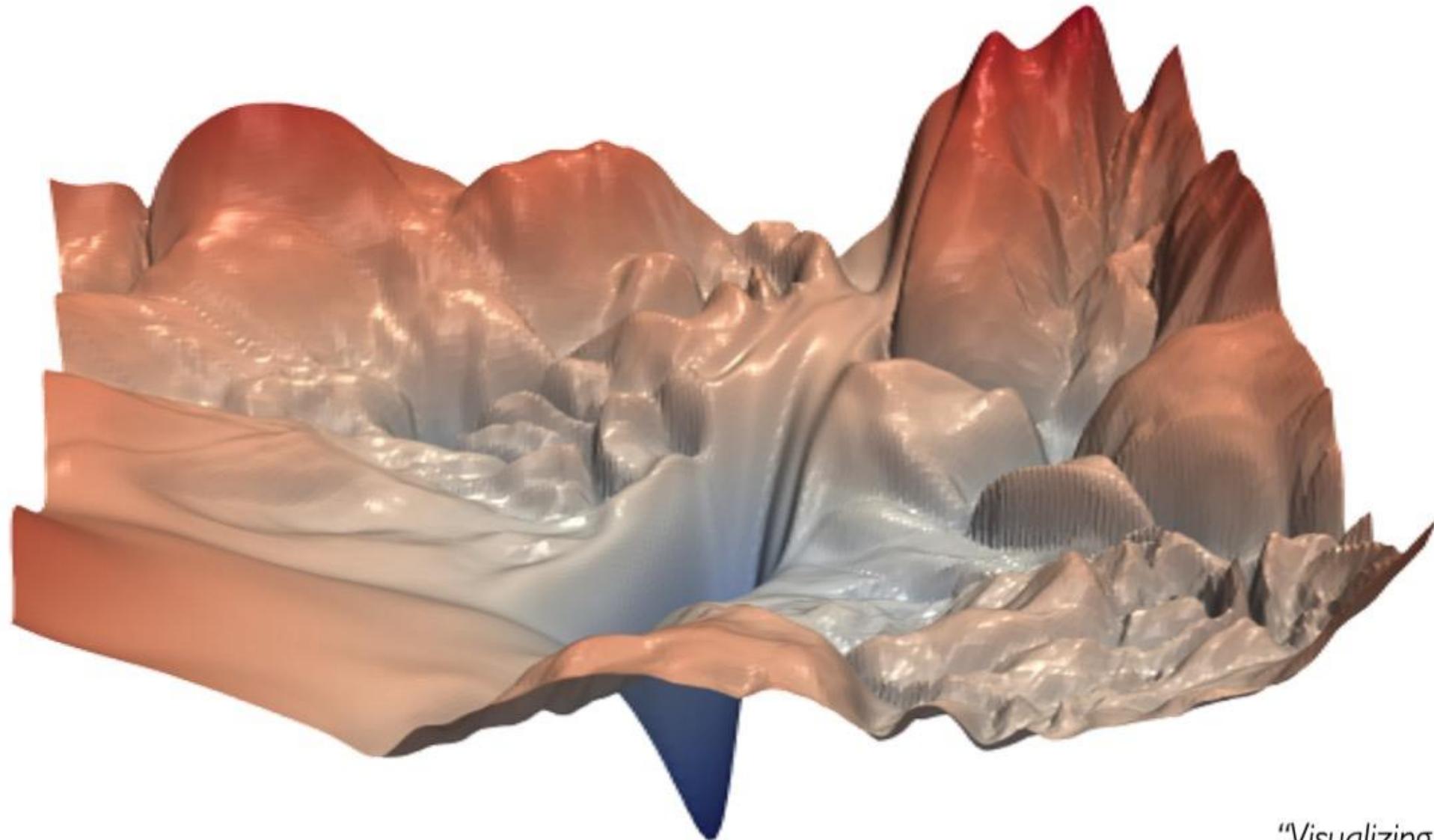


$$\frac{\partial J(W)}{\partial w_1} = \underbrace{\frac{\partial J(W)}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue}}$$

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

# Neural Networks in Practice: Optimization

# Training Neural Networks is Difficult



*"Visualizing the loss landscape of neural nets". Dec 2017.*

# Loss Functions Can Be Difficult to Optimize

**Remember:**

Optimization through gradient descent

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$$

# Loss Functions Can Be Difficult to Optimize

**Remember:**

Optimization through gradient descent

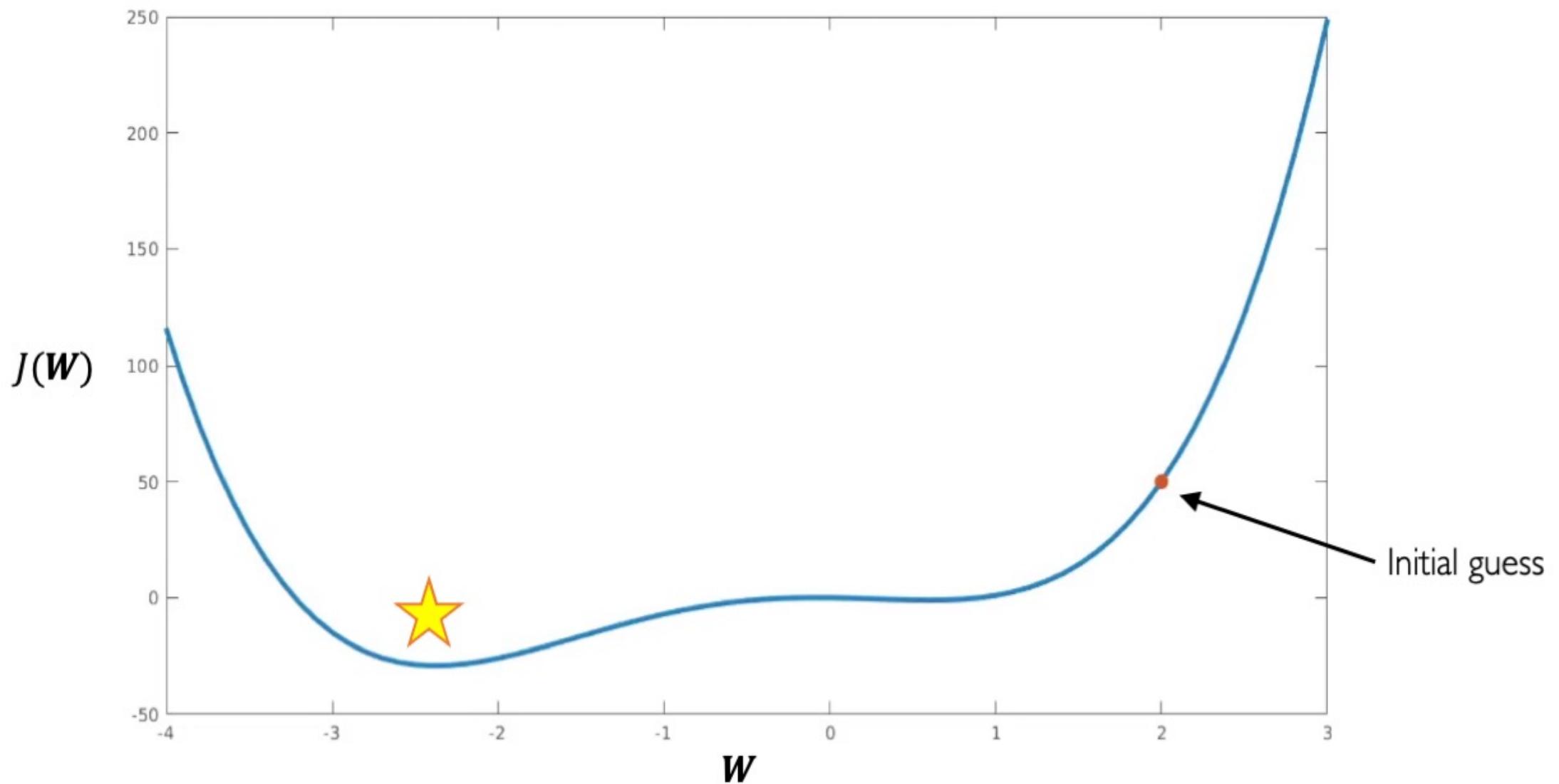
$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$$



How can we set the  
learning rate?

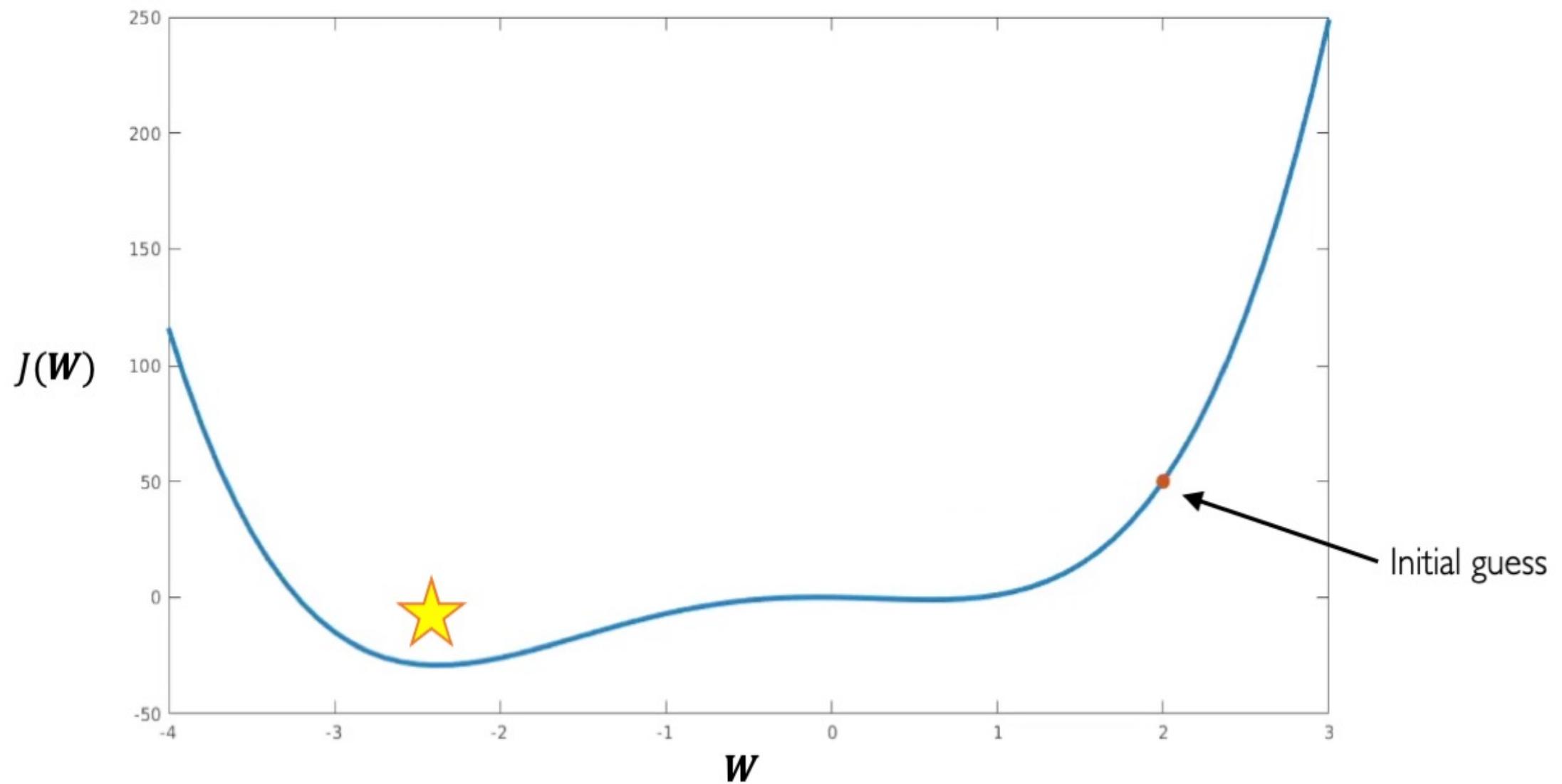
# Setting the Learning Rate

*Small learning rate* converges slowly and gets stuck in false local minima



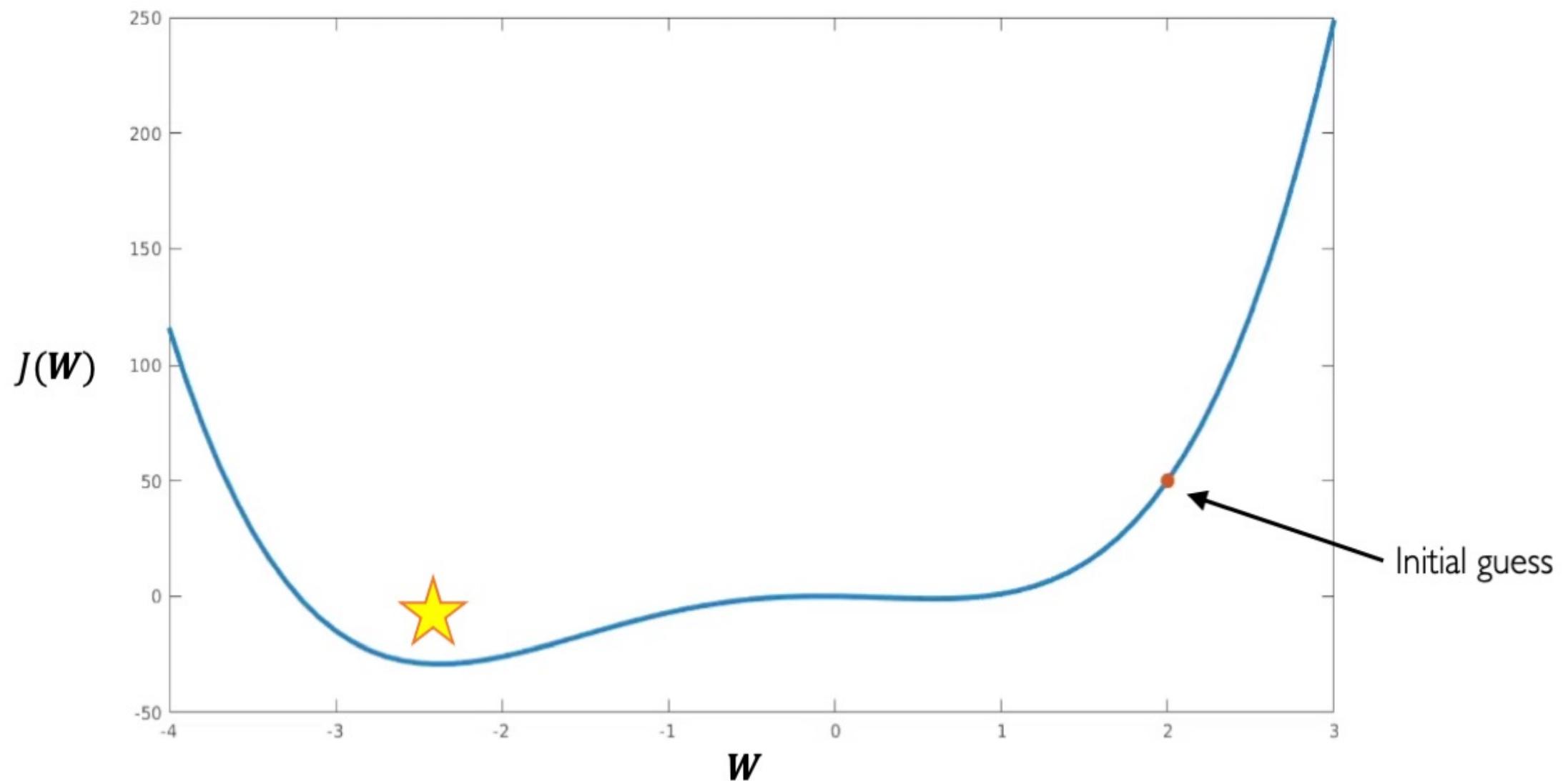
# Setting the Learning Rate

*Large learning rates* overshoot, become unstable and diverge



# Setting the Learning Rate

*Stable learning rates* converge smoothly and avoid local minima



# How to deal with this?

## **Idea I:**

Try lots of different learning rates and see what works “just right”

# How to deal with this?

## Idea 1:

Try lots of different learning rates and see what works “just right”

## Idea 2:

Do something smarter!

Design an adaptive learning rate that “adapts” to the landscape

# Adaptive Learning Rates

- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
  - how large gradient is
  - how fast learning is happening
  - size of particular weights
  - etc...

# Gradient Descent Algorithms

## Algorithm

- SGD
- Adam
- Adadelta
- Adagrad
- RMSProp

## TF Implementation



`tf.keras.optimizers.SGD`



`tf.keras.optimizers.Adam`



`tf.keras.optimizers.Adadelta`



`tf.keras.optimizers.Adagrad`



`tf.keras.optimizers.RMSProp`

## Reference

Kiefer & Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function." 1952.

Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.

Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.

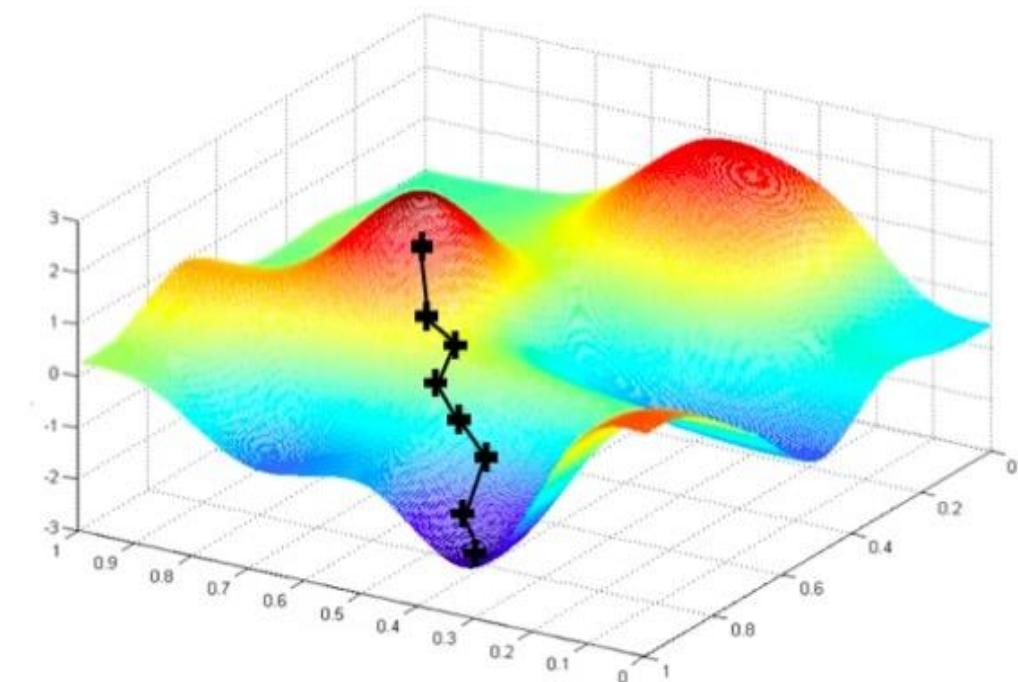
Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.

# Neural Networks in Practice: Mini-batches

# Gradient Descent

## Algorithm

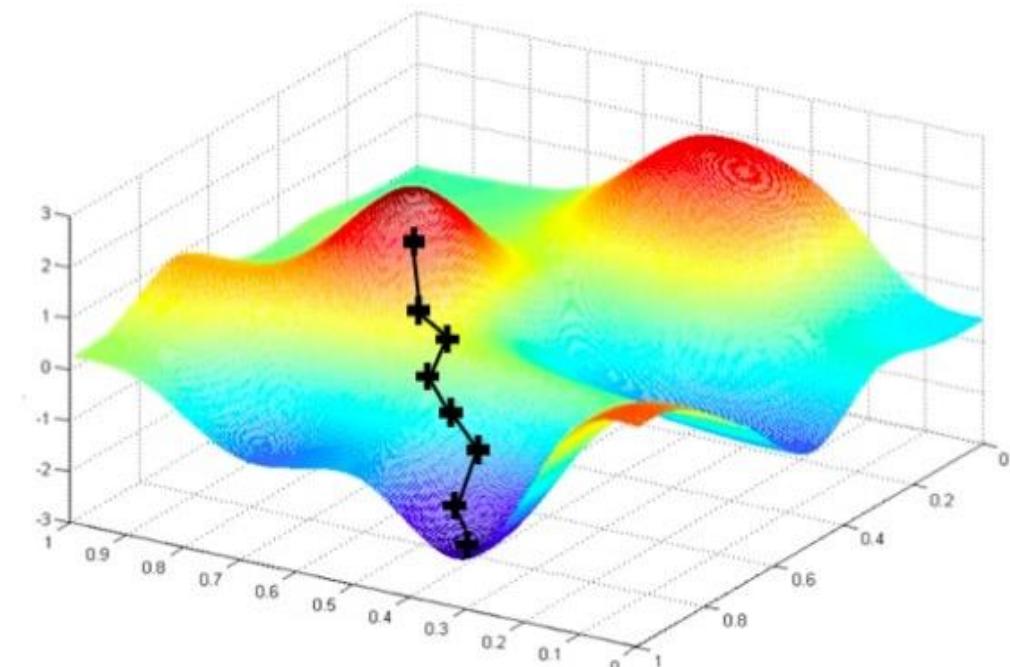
1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient,  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights



# 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

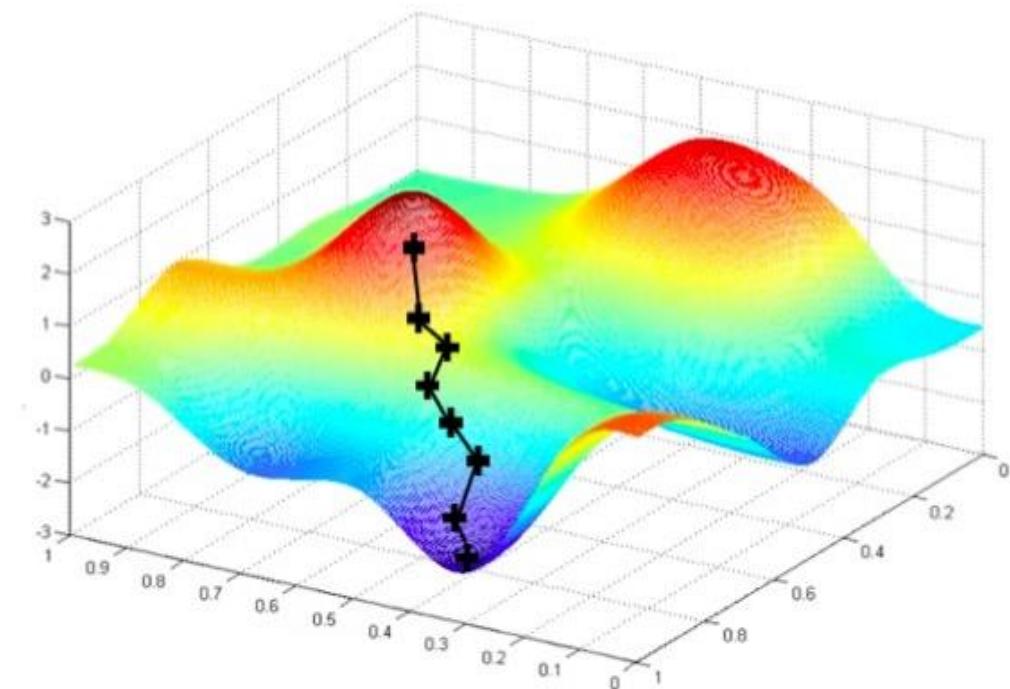


Can be very  
computationally  
intensive to compute!

# Stochastic Gradient Descent

## Algorithm

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

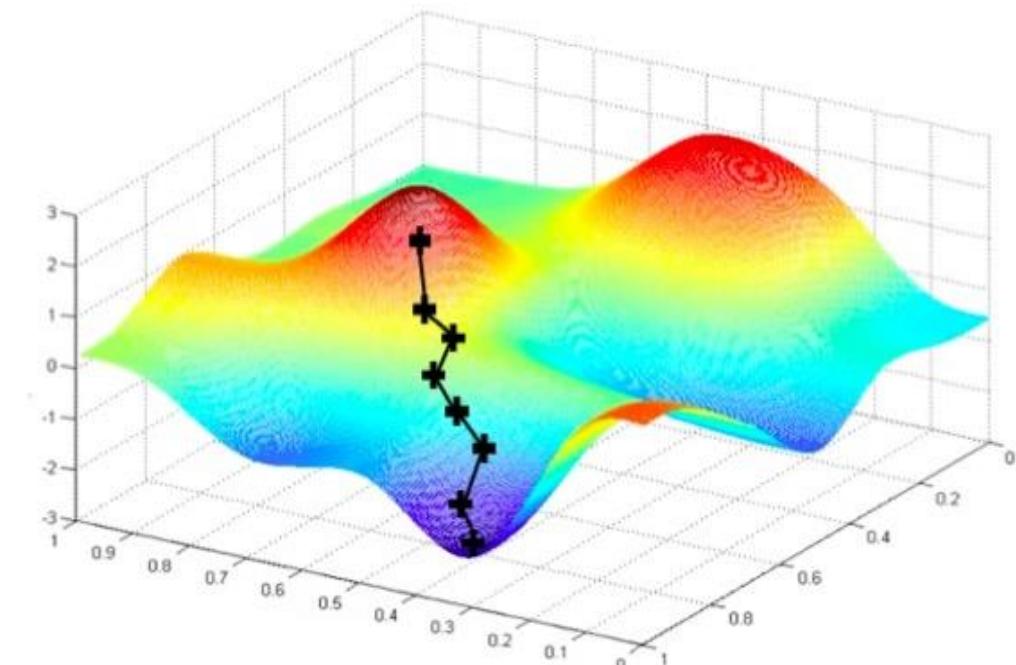


# Stochastic Gradient Descent

## Algorithm

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

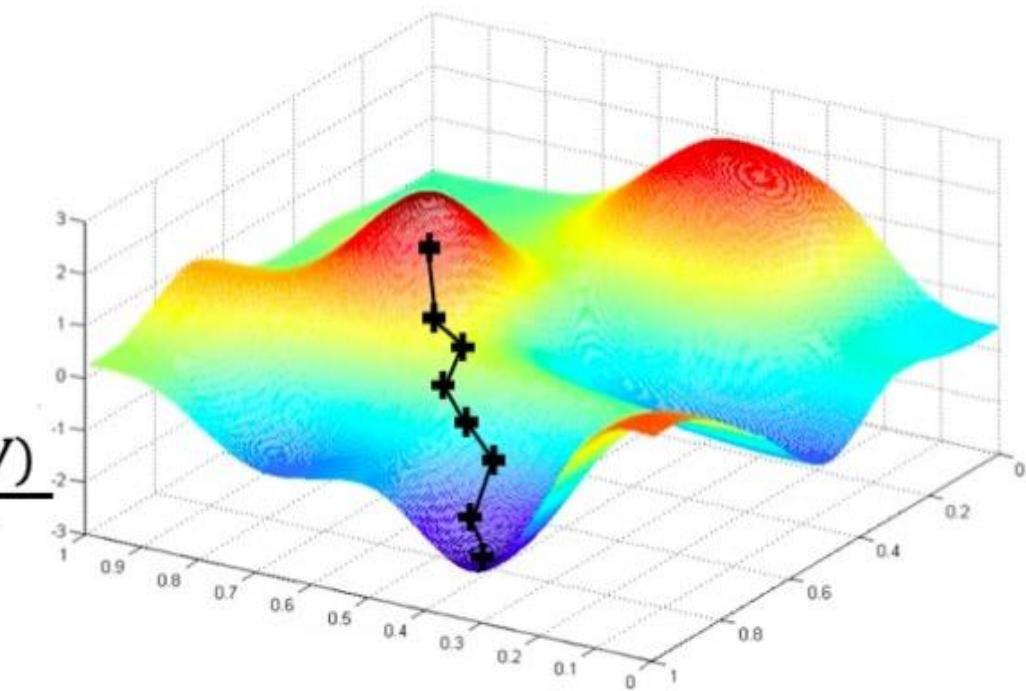
Easy to compute but  
**very noisy** (stochastic)!



# Stochastic Gradient Descent

## Algorithm

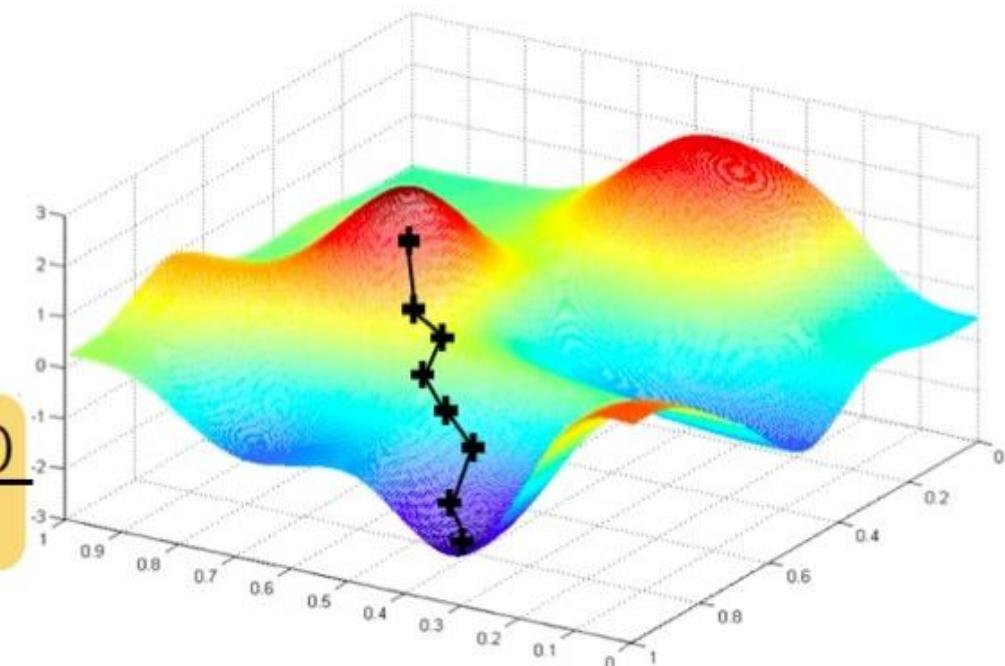
1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of  $B$  data points
4. Compute gradient,  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
6. Return weights



# Stochastic Gradient Descent

## Algorithm

1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of  $B$  data points
4. Compute gradient, 
$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$$
5. Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
6. Return weights



Fast to compute and a much better estimate of the true gradient!

# Mini-batches while training

## **More accurate estimation of gradient**

Smoother convergence

Allows for larger learning rates

# Mini-batches while training

**More accurate estimation of gradient**

Smoother convergence

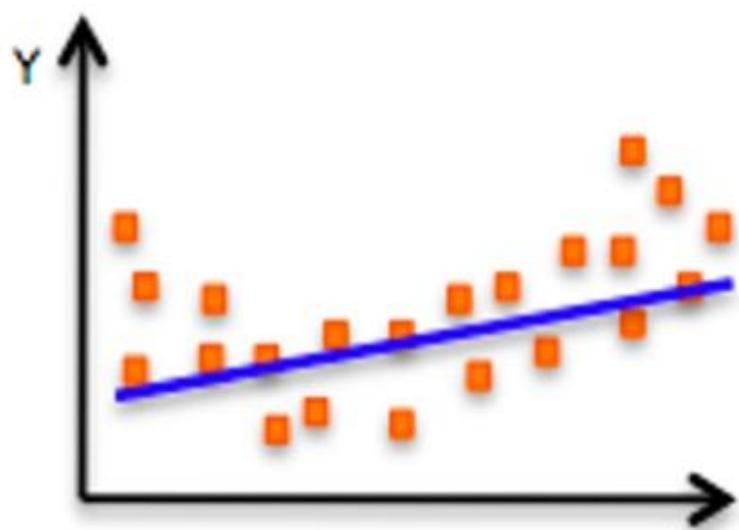
Allows for larger learning rates

**Mini-batches lead to fast training!**

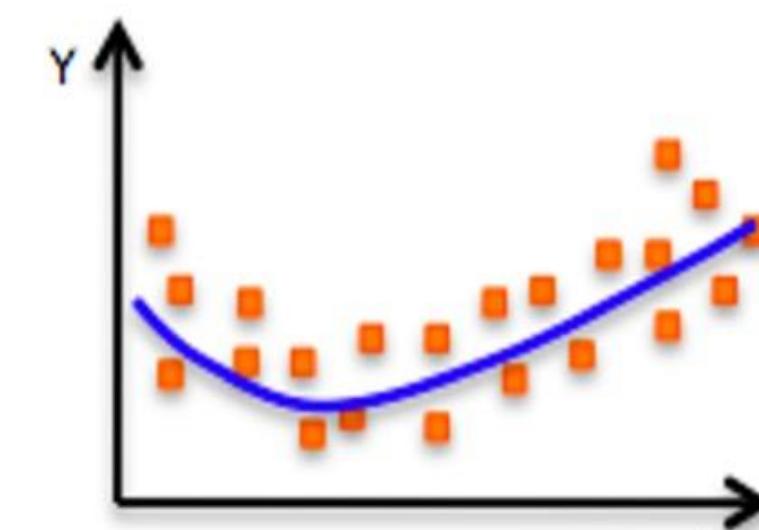
Can parallelize computation + achieve significant speed increases on GPU's

# Neural Networks in Practice: Overfitting

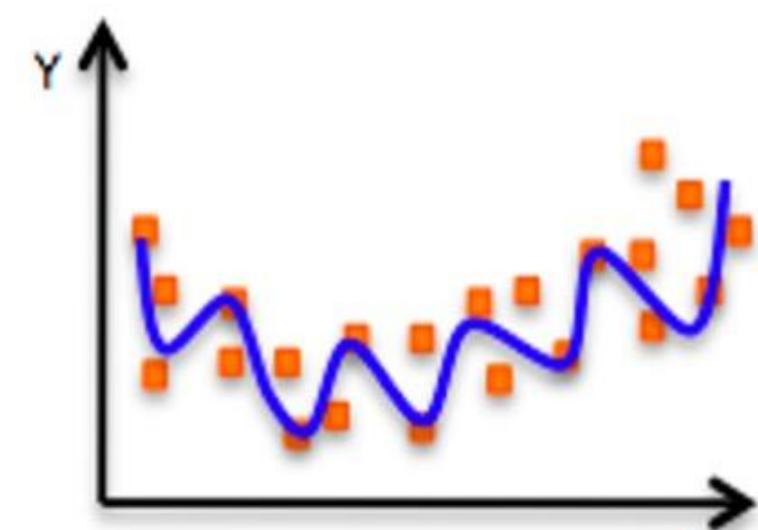
# The Problem of Overfitting



**Underfitting**  
Model does not have capacity  
to fully learn the data



←      **Ideal fit**      →



**Overfitting**  
Too complex, extra parameters,  
does not generalize well

# Regularization

## **What is it?**

*Technique that constrains our optimization problem to discourage complex models*

# Regularization

## *What is it?*

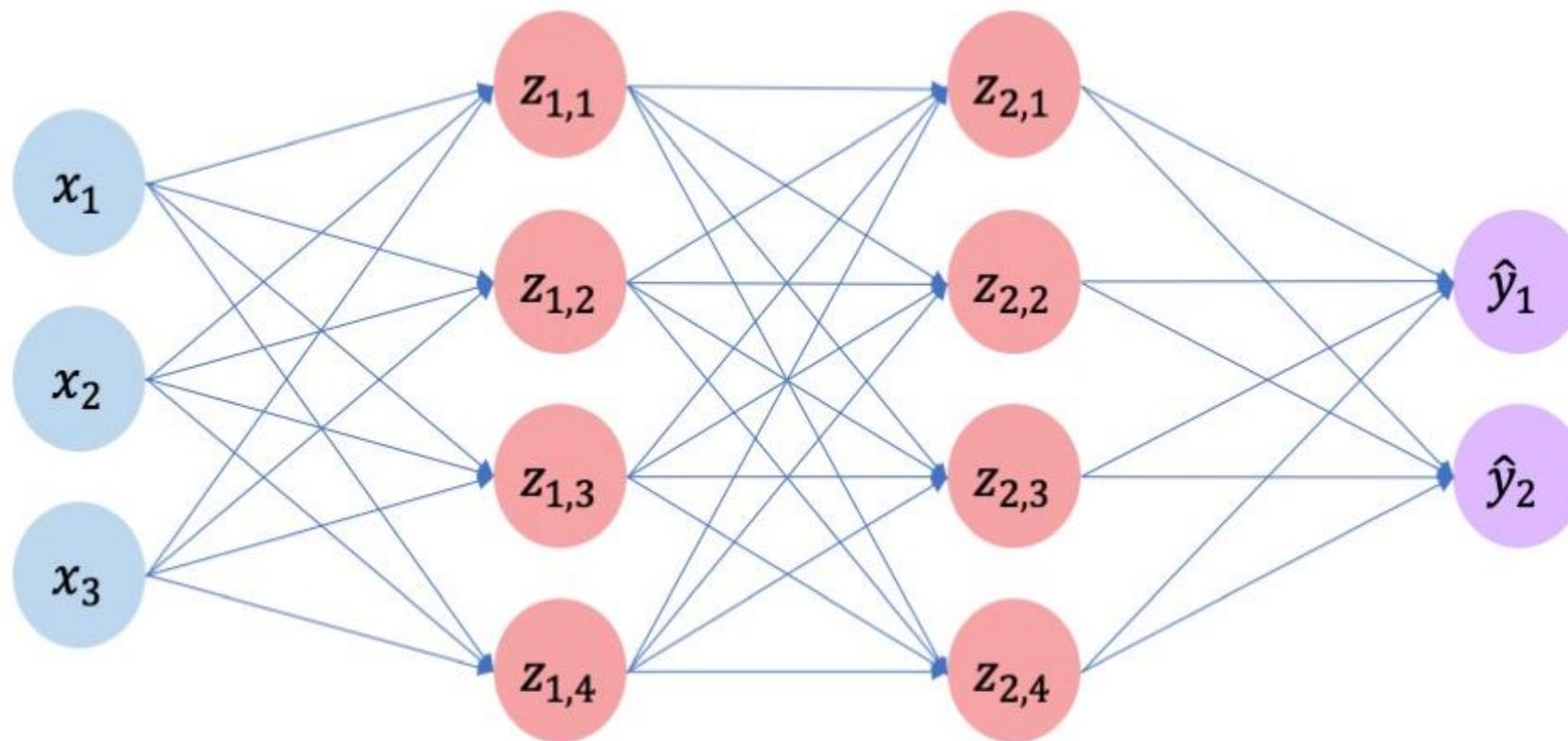
*Technique that constrains our optimization problem to discourage complex models*

## **Why do we need it?**

*Improve generalization of our model on unseen data*

# Regularization I: Dropout

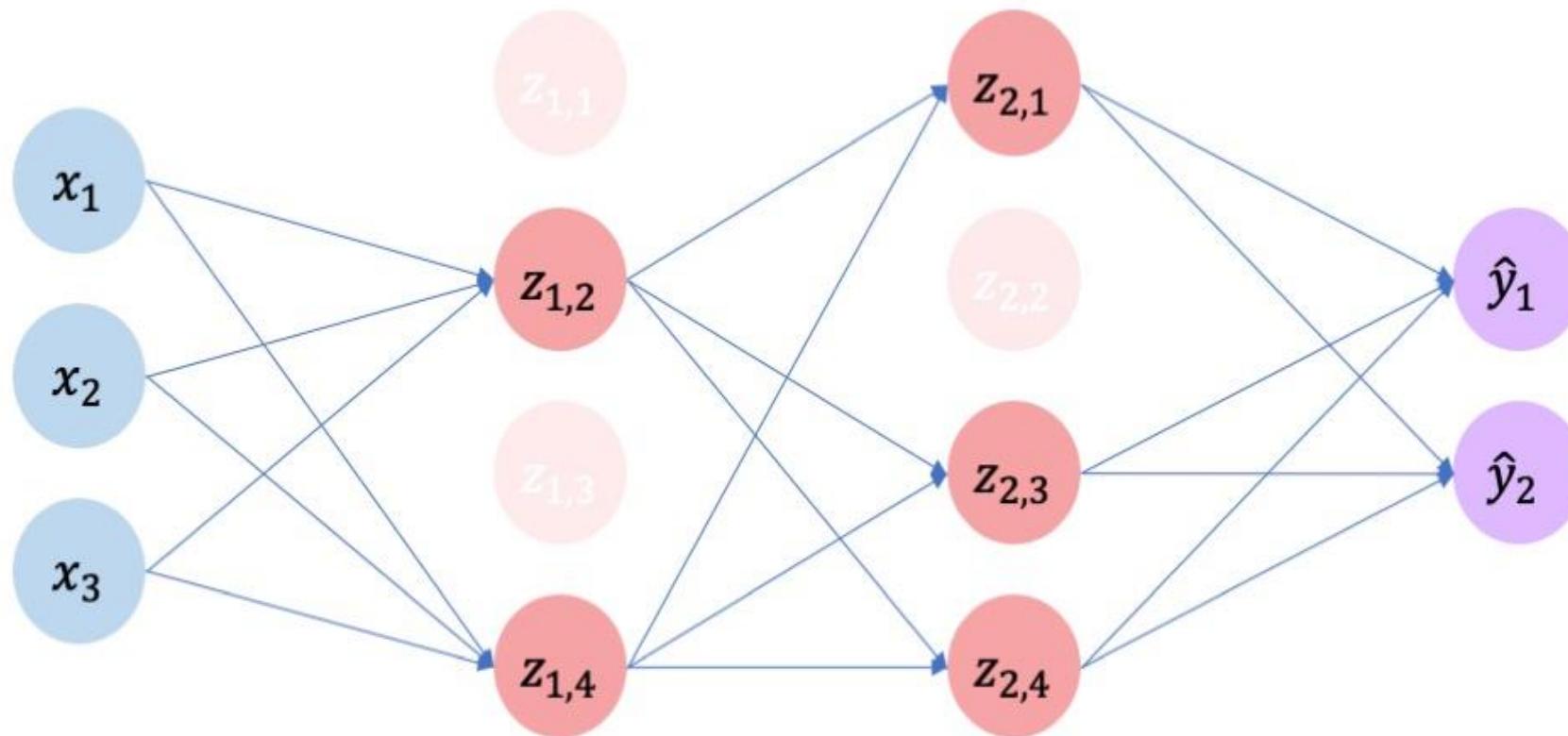
- During training, randomly set some activations to 0



# Regularization I: Dropout

- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

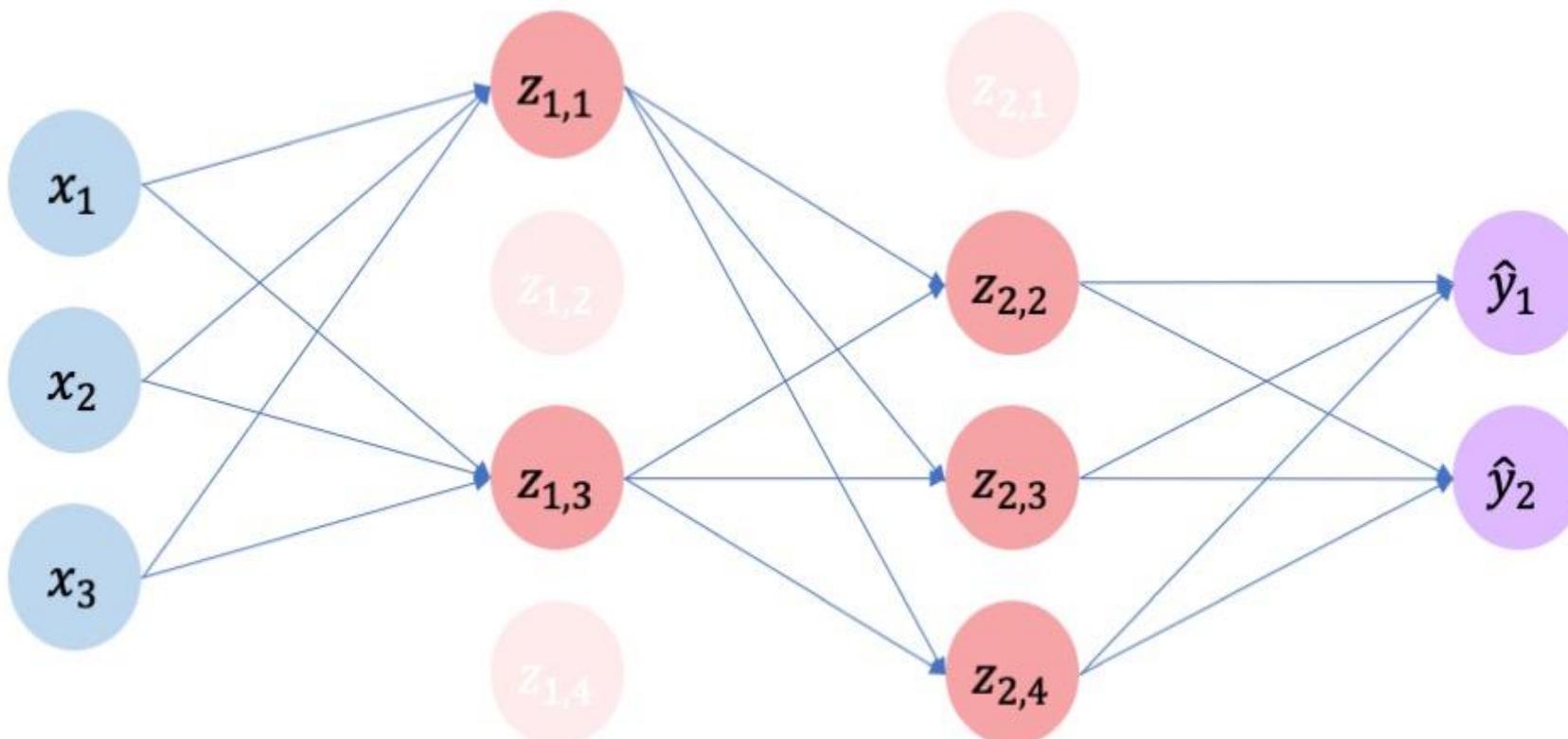
 `tf.keras.layers.Dropout(p=0.5)`



# Regularization I: Dropout

- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

 `tf.keras.layers.Dropout(p=0.5)`



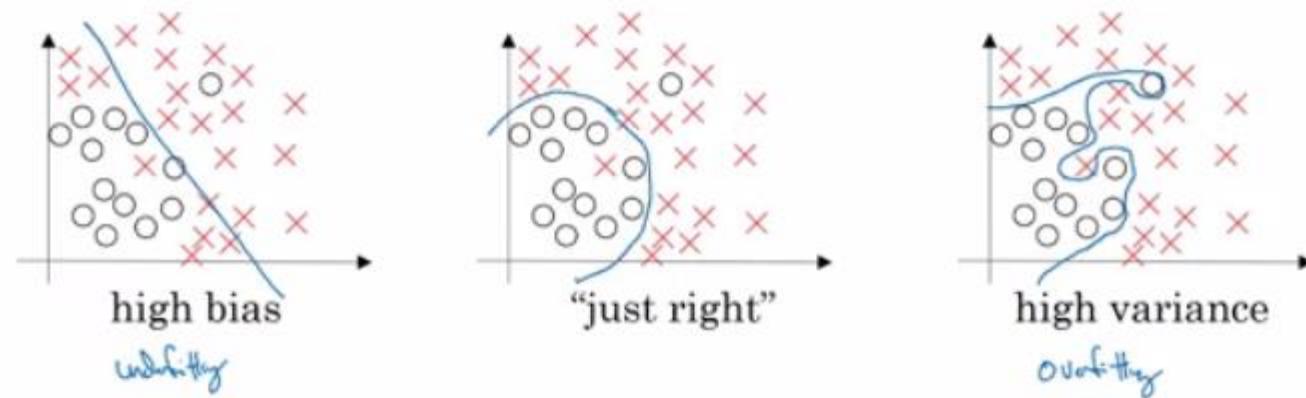
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



- If your model is underfitting (logistic regression of non linear data) it has a "high bias"
- If your model is overfitting then it has a "high variance"
- Your model will be alright if you balance the Bias / Variance

### Bias and Variance



- If your algorithm has a high bias:
  - Try to make your NN bigger (size of hidden units, number of layers)
  - Try a different model that is suitable for your data.
  - Try to run it longer.
  - Different (advanced) optimization algorithms.
- If your algorithm has a high variance:
  - More data.
  - Try regularization.
  - Try a different model that is suitable for your data.
- You should try the previous two points until you have a low bias and low variance.

# Regularization: Data augmentation

- For example in a computer vision data:
  - You can flip all your pictures horizontally this will give you more data instances.
  - You could also apply a random position and rotation to an image to get more data.

# Random Initialization

- In logistic regression it wasn't important to initialize the weights randomly, while in NN we have to initialize them randomly.
- If we initialize all the weights with zeros in NN it won't work (initializing bias with zero is OK):
  - all hidden units will be completely identical (symmetric) - compute exactly the same function
  - on each gradient descent iteration all the hidden units will always update the same
- To solve this we initialize the  $W$ 's with a small random numbers:
- $W_1 = np.random.randn(2,2) * 0.01$  # 0.01 to make it small enough  $b_1 = np.zeros((2,1))$  # its ok to have  $b$  as zero, it won't get us to the symmetry breaking problem

# Random Initialization

- We need small values because in sigmoid (or tanh), for example, if the weight is too large you are more likely to end up even at the very start of training with very large values of  $Z$ . Which causes your tanh or your sigmoid activation function to be saturated, thus slowing down learning. If you don't have any sigmoid or tanh activation functions throughout your neural network, this is less of an issue.
- Constant 0.01 is alright for 1 hidden layer networks, but if the NN is deep this number can be changed but it will always be a small number.

# Parameters vs Hyperparameters

- Main parameters of the NN is  $W$  and  $b$
- Hyper parameters (parameters that control the algorithm) are like:
  - Learning rate.
  - Number of iteration.
  - Number of hidden layers  $L$ .
  - Number of hidden units  $n$ .
  - Choice of activation functions.
- You have to try values yourself of hyper parameters.

# Practical Part:

## Deep Neural Network for Image Classification

- Build and apply a deep neural network to supervised learning.
- Apply it to cat vs non-cat classification.

- Let's Go

# Deep L-layer neural network

- Shallow NN is a NN with one or two layers.
- Deep NN is a NN with three or more layers.
- We will use the notation  $L$  to denote the number of layers in a NN.
- $n[1]$  is the number of neurons in a specific layer  $1$ .
- $n[0]$  denotes the number of neurons in input layer.  $n[L]$  denotes the number of neurons in output layer.
- $g[1]$  is the activation function.
- $a[1] = g[1](z[1])$
- $w[1]$  weights is used for  $z[1]$
- $x = a[0]$ ,  $a[1] = y'$
- These were the notation we will use for deep neural network.
- So we have:
  - A vector  $n$  of shape  $(1, \text{NoOfLayers}+1)$
  - A vector  $g$  of shape  $(1, \text{NoOfLayers})$
  - A list of different shapes  $w$  based on the number of neurons on the previous and the current layer.
  - A list of different shapes  $b$  based on the number of neurons on the current layer.

# Getting your matrix dimensions right

- The best way to debug your matrices dimensions is by a pencil and paper.
- Dimension of  $W$  is  $(n[1], n[1-1])$ . Can be thought by right to left.
- Dimension of  $b$  is  $(n[1], 1)$
- $dw$  has the same shape as  $W$ , while  $db$  is the same shape as  $b$
- Dimension of  $z[1]$ ,  $A[1]$ ,  $dz[1]$ , and  $dA[1]$  is  $(n[1], m)$

## Forward and Backward Propagation

- Pseudo code for forward propagation for layer l:

```
Input A[1-1]
z[1] = w[1]A[1-1] + b[1]
A[1] = g[1](z[1])
Output A[1], cache(z[1])
```

- Pseudo code for back propagation for layer l:

```
Input da[1], Caches
dz[1] = dA[1] * g'[1](z[1])
dw[1] = (dz[1]A[1-1].T) / m
db[1] = sum(dz[1])/m
dA[1-1] = w[1].T * dz[1]
Output dA[1-1], dw[1], db[1]
```

# Dont forget axis=1, keepdims=True  
# The multiplication here are a dot product.

- If we have used our loss function then:

```
dA[L] = (-y/a) + ((1-y)/(1-a)))
```