



# Introduction to Deep Learning



# The Rise of Deep Learning

## 'Deep Voice' Software Can Clone Anyone's Voice With Just 3.7 Seconds of Audio

Using snippets of voices, Baidu's 'Deep Voice' can generate new speech, accents, and tones.



## Let There Be Sight: How Deep Learning Is Helping the Blind 'See'



## It's with DEEPMIND STARCRRAFT TRIUMPH



## Technology outpacing security measures

Facial Recognition | Features and Interviews

## AI beats docs in cancer spotting

A new study provides a fresh example of machine learning as an important diagnostic tool. Paul Biegler reports.



## AI Can Help In Predicting Cryptocurrency Value



## 'Creative' AlphaZero leads way for chess computers and, maybe, science

Former chess world champion Garry Kasparov likes what he sees of computer that could be used to find cures for diseases



## How an A.I. 'Cat-and-Mouse Game' Generates Believable Fake Photos

By CADE METZ and KEITH COLLINS - JAN 2, 2018



## Stock Predictions Based On AI: Is the Market Truly Predictable?



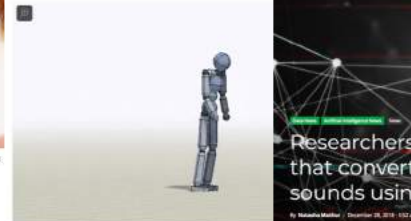
Complex of bacteria-infecting viral proteins modeled in CASP-13. The complex that were modeled individually. PROTEIN DATA BANK

## Google's DeepMind acs protein folding

By Robert F. Service | Dec. 6, 2018, 12:05 PM

## After Millions of Trials, These Simulated Humans Learned to Do Perfect Backflips and Cartwheels

George Douras 4/10/18 11:55am - Pinned to #1



## Neural networks everywhere

New chip reduces neural networks' power consumption by up to 95 percent, making them practical for battery-powered devices.

Wed, 01/16/2019 - 8:00am | Comment by Kenny Walker - Digital Reporter - @RandDMagazine



## Researchers introduce a deep learning method that converts mono audio recordings into 3D sounds using video scenes

By Hannah Marder - December 26, 2018 10:21am

## AI faces show how far AI image generation has advanced in just four years

People on the right aren't real: they're the product of machine learning



## Automation And Algorithms: De-Risking Manufacturing With Artificial Intelligence

Sarah Goehrke Contributor Manufacturing 1 focus on the industrialization of additive manufacturing

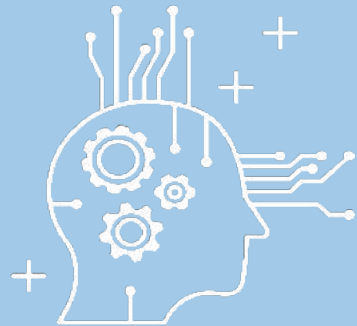
### TWEET THIS

The two key applications of AI in manufacturing are pricing and manufacturability feedback

# What is Deep Learning?

## ARTIFICIAL INTELLIGENCE

Any technique that enables computers to mimic human behavior



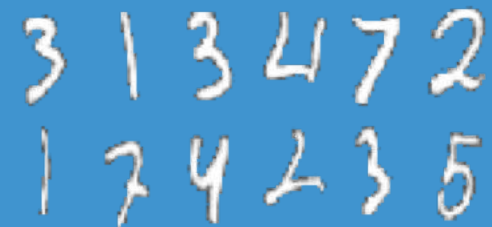
## MACHINE LEARNING

Ability to learn without explicitly being programmed



## DEEP LEARNING

Extract patterns from data using neural networks



# Why Deep Learning and Why Now?

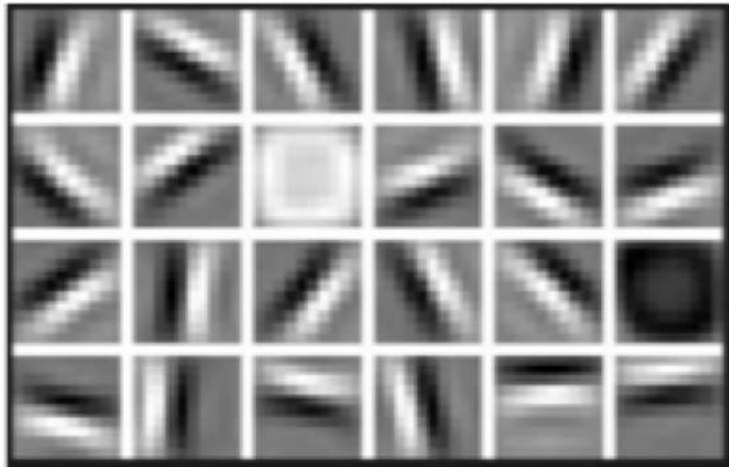


# Why Deep Learning?

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

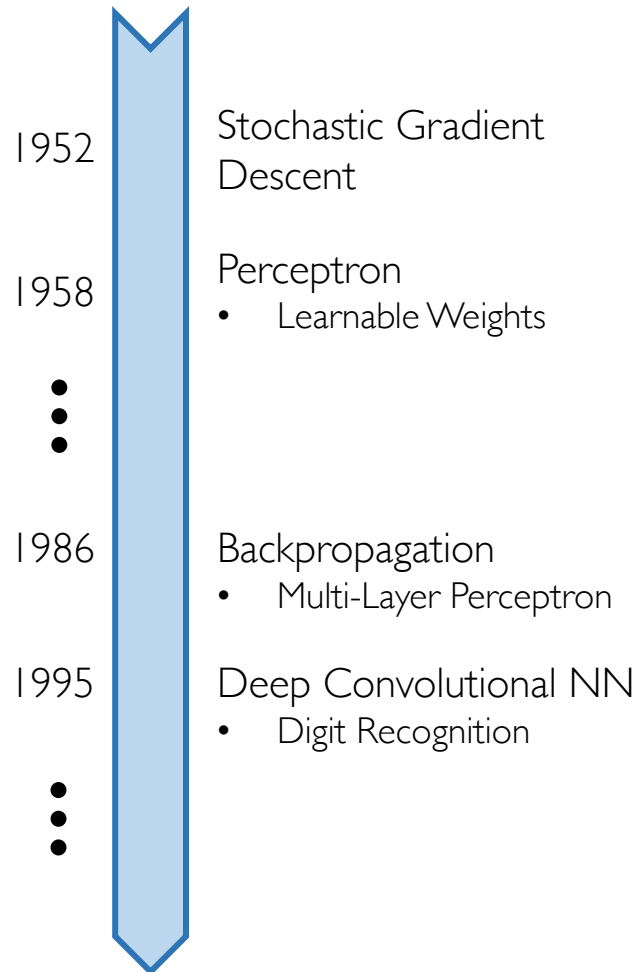
High Level Features



Facial Structure

# Why Now?

Neural Networks date back decades, so why the resurgence?



## 1. Big Data

- Larger Datasets
- Easier Collection & Storage

IMAGENET



WIKIPEDIA  
The Free Encyclopedia



## 2. Hardware

- Graphics Processing Units (GPUs)
- Massively Parallelizable



## 3. Software

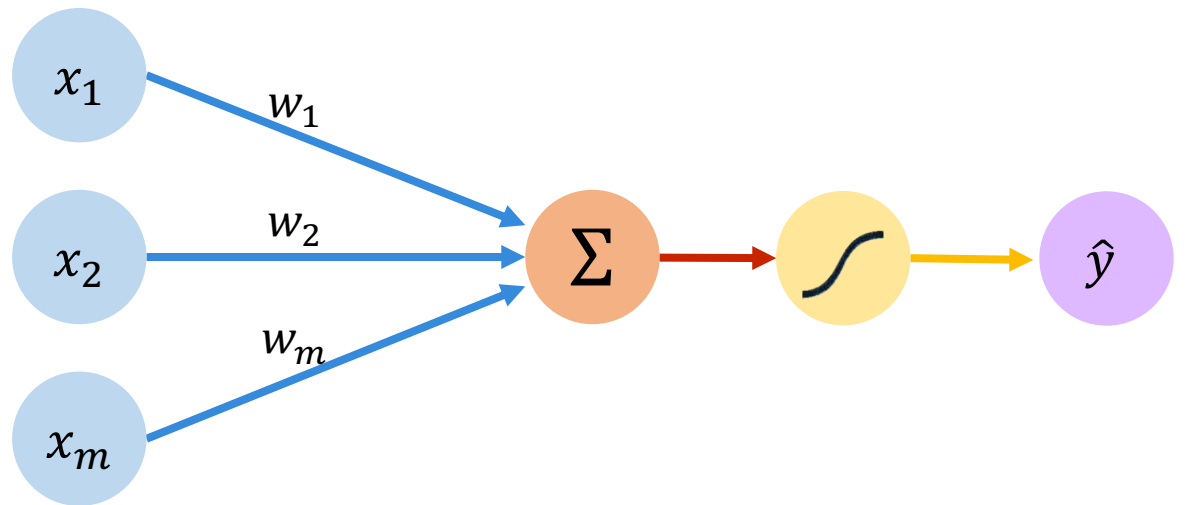
- Improved Techniques
- New Models
- Toolboxes



# The Perceptron

The structural building block of deep learning

# The Perceptron: Forward Propagation



Inputs      Weights      Sum      Non-Linearity      Output

Output

Linear combination of inputs

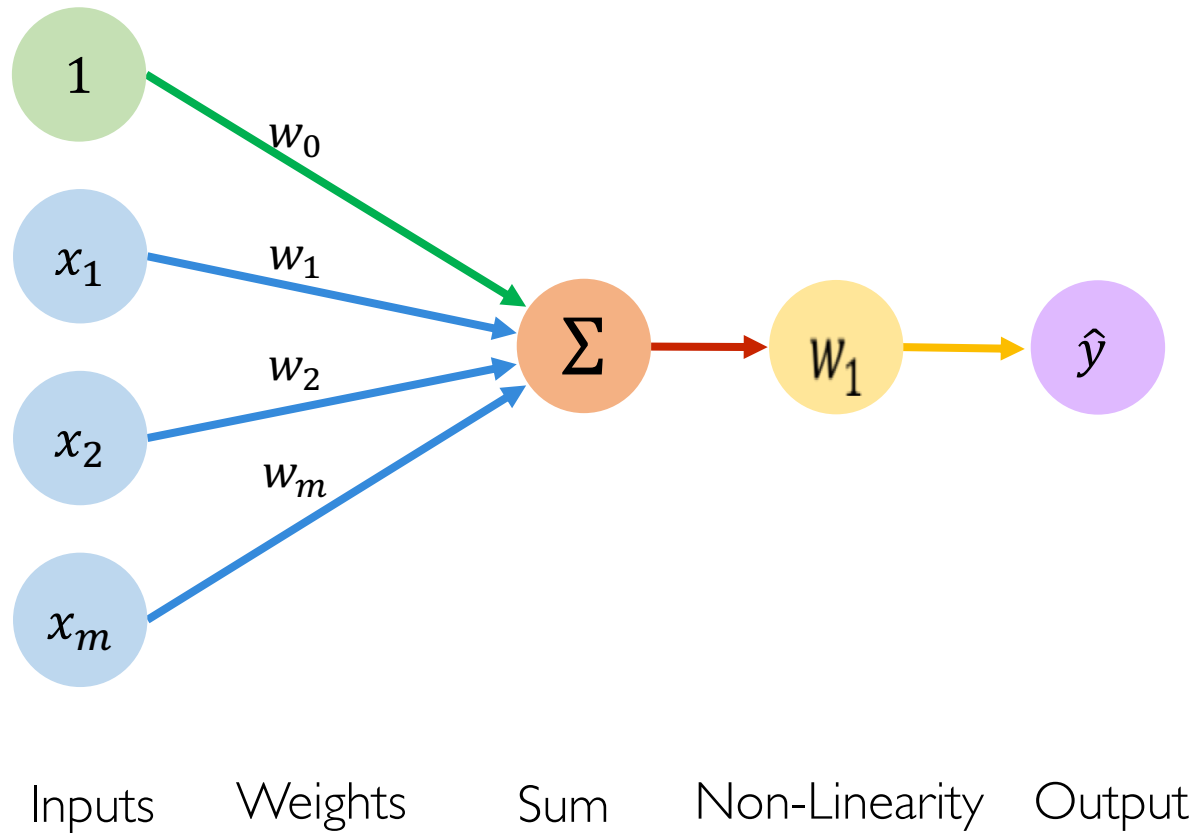
$$\hat{y} = g \left( \sum_{i=1}^m x_i w_i \right)$$

Non-linear activation function

The diagram shows the mathematical representation of the neuron's operation. A purple arrow points from the word "Output" to the  $\hat{y}$  in the equation. A red arrow points from the text "Linear combination of inputs" to the summation term  $\sum_{i=1}^m x_i w_i$ . A yellow arrow points from the text "Non-linear activation function" to the  $g$  function in the equation.



# The Perceptron: Forward Propagation



Output

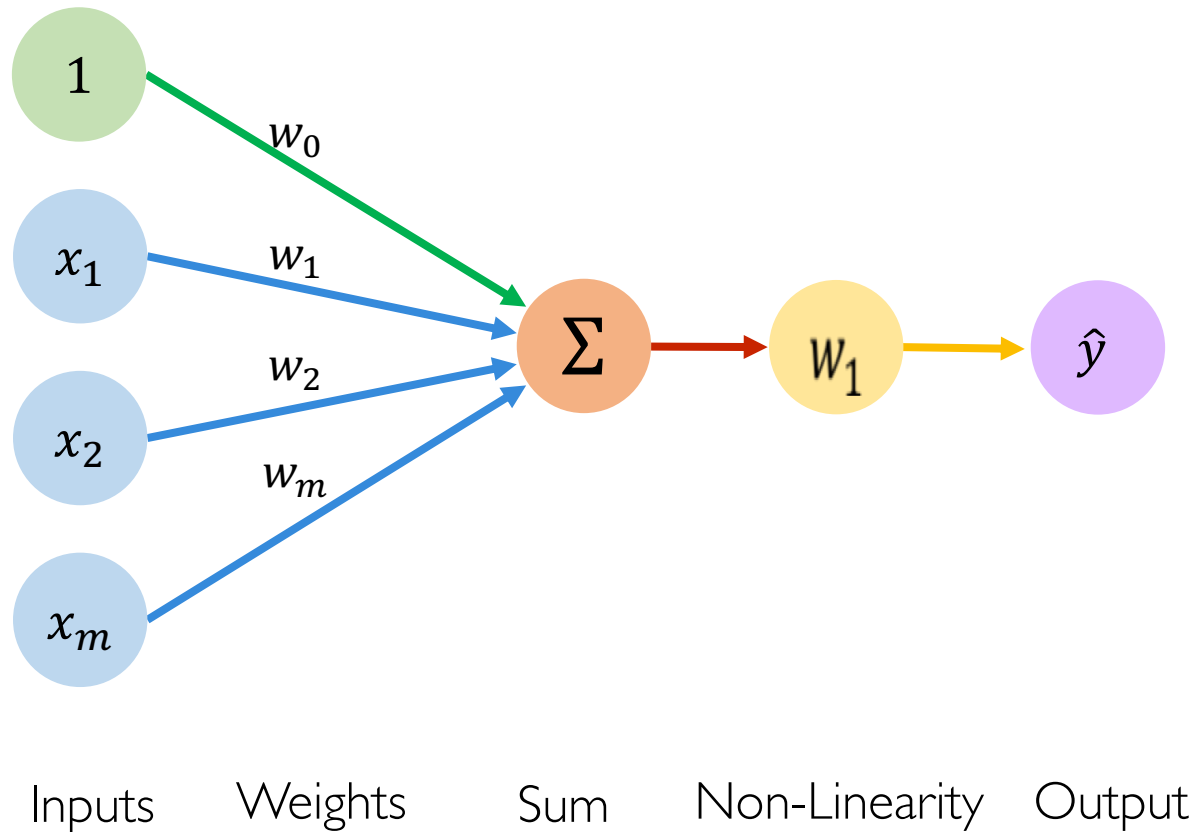
Linear combination of inputs

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

Non-linear activation function

Bias

# The Perceptron: Forward Propagation

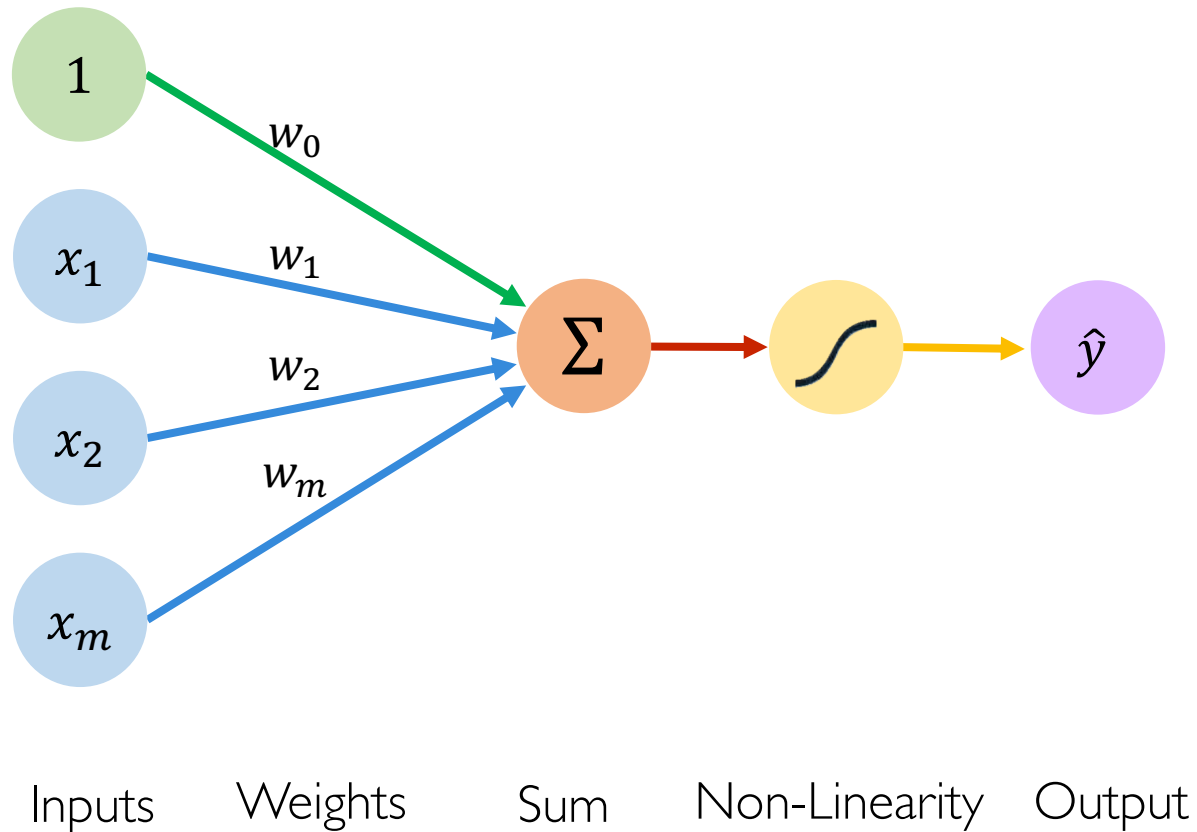


$$\hat{y} = g \left( w_0 + \sum_{i=1}^m x_i w_i \right)$$

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

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

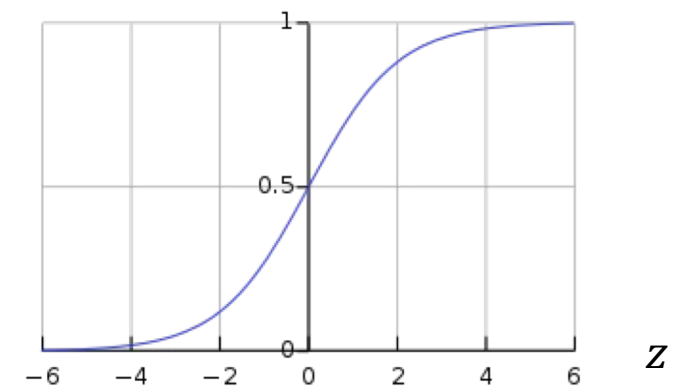


## Activation Functions

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

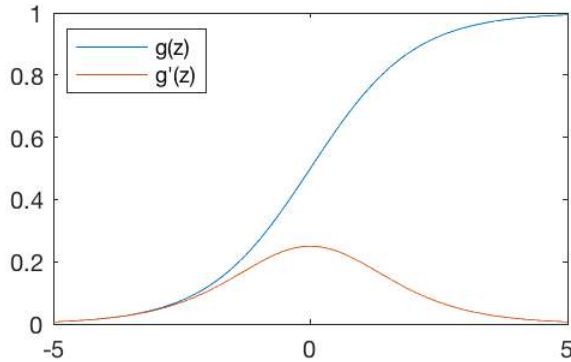
- Example: sigmoid function

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



# Common Activation Functions

Sigmoid Function

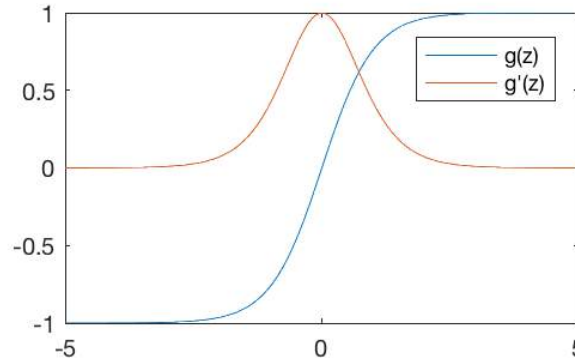


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

 `tf.nn.sigmoid(z)`

Hyperbolic Tangent

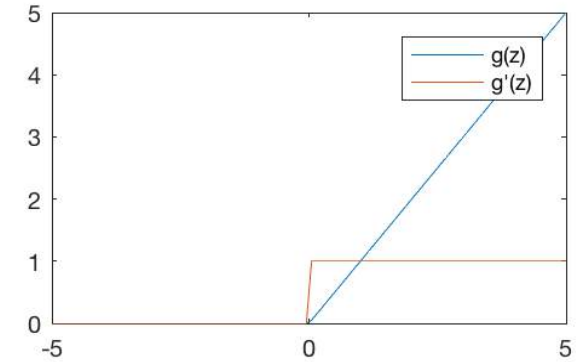


$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

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

 `tf.nn.tanh(z)`

Rectified Linear Unit (ReLU)



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

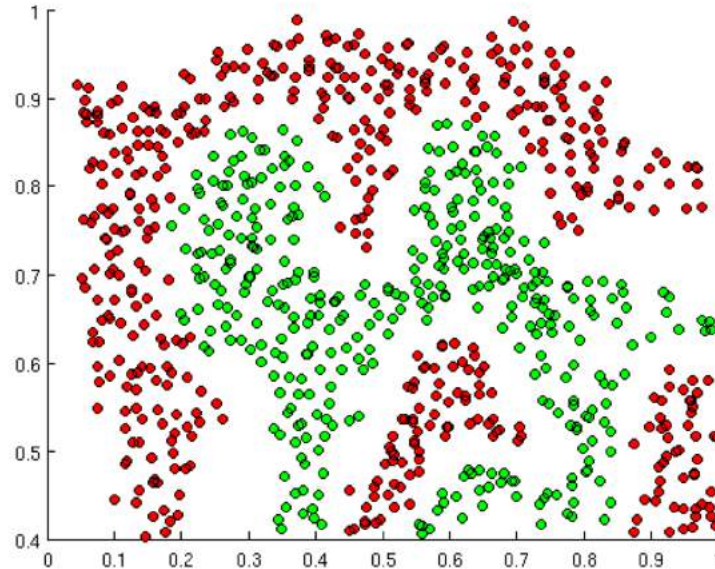
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

 `tf.nn.relu(z)`

NOTE: All activation functions are non-linear

# Importance of Activation Functions

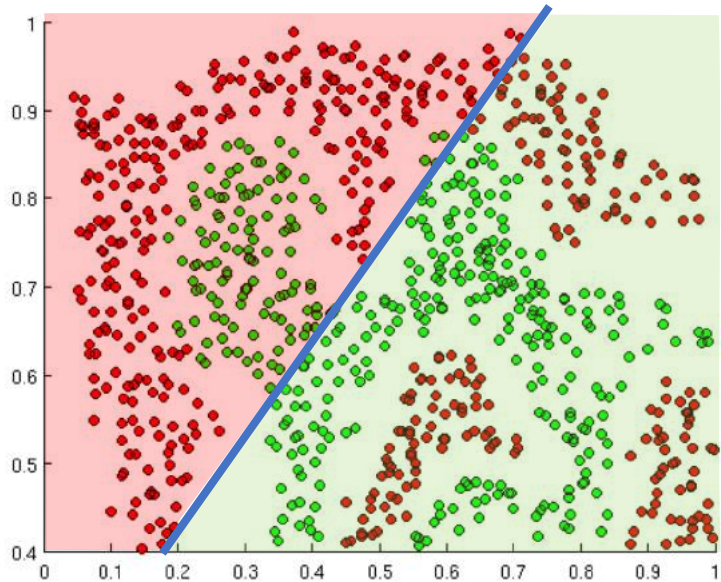
The purpose of activation functions is to **introduce non-linearities** into the network



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

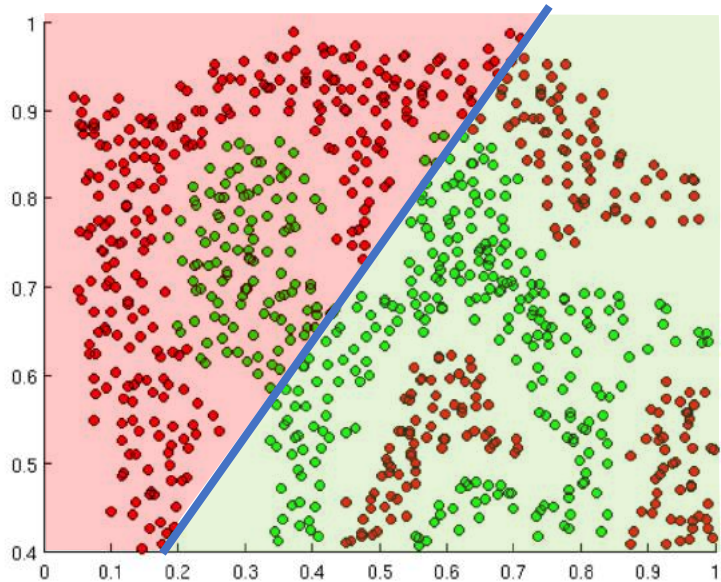


Linear Activation functions produce linear decisions no matter the network size

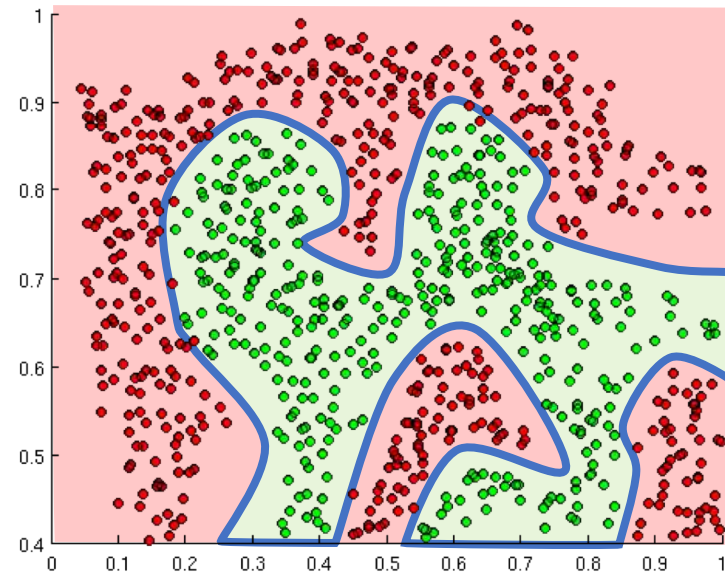


# Importance of Activation Functions

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

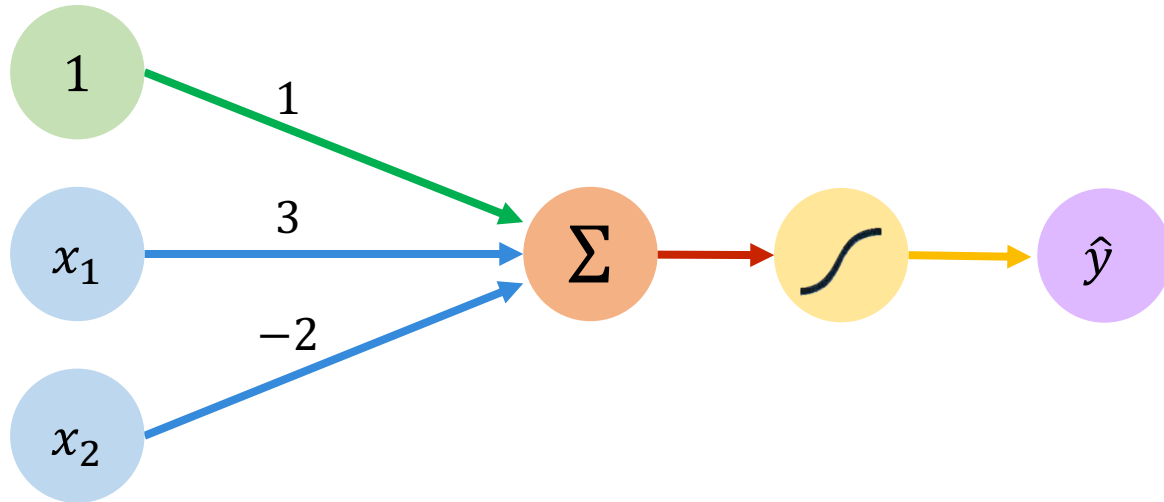


Linear Activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

# The Perceptron: Example

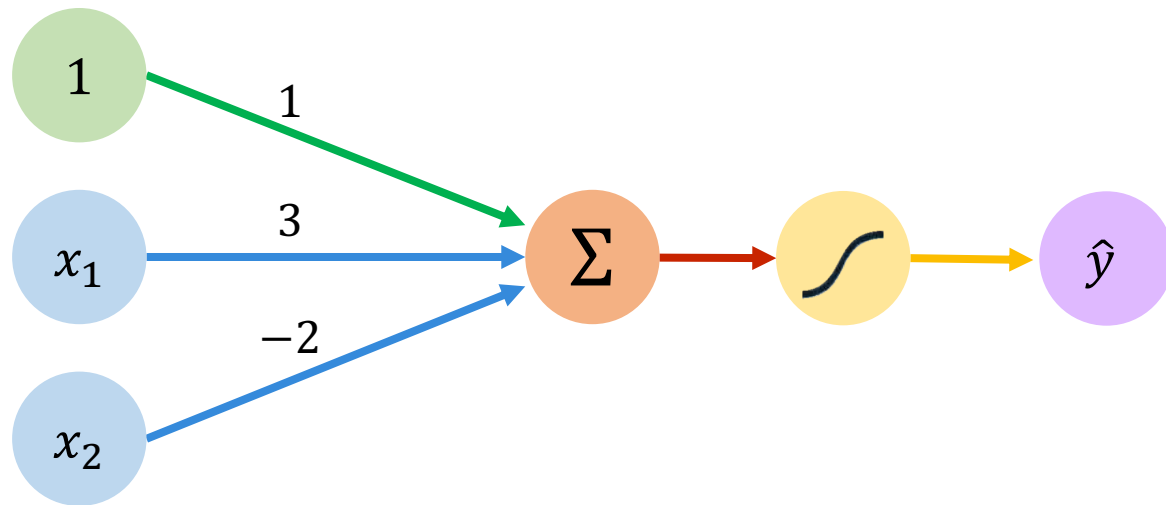


We have:  $w_0 = 1$  and  $\mathbf{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

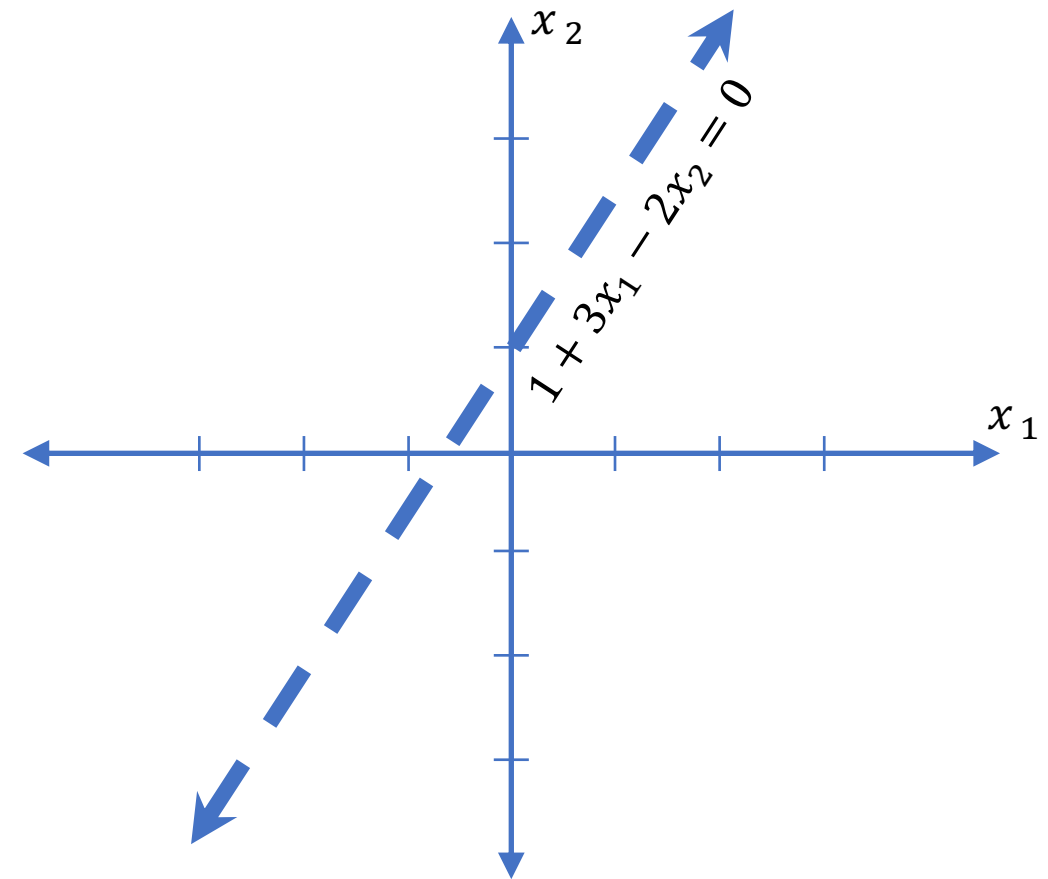
$$\begin{aligned}\hat{y} &= g(w_0 + \mathbf{X}^T \mathbf{W}) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g(1 + \underbrace{3x_1 - 2x_2})\end{aligned}$$

This is just a line in 2D!

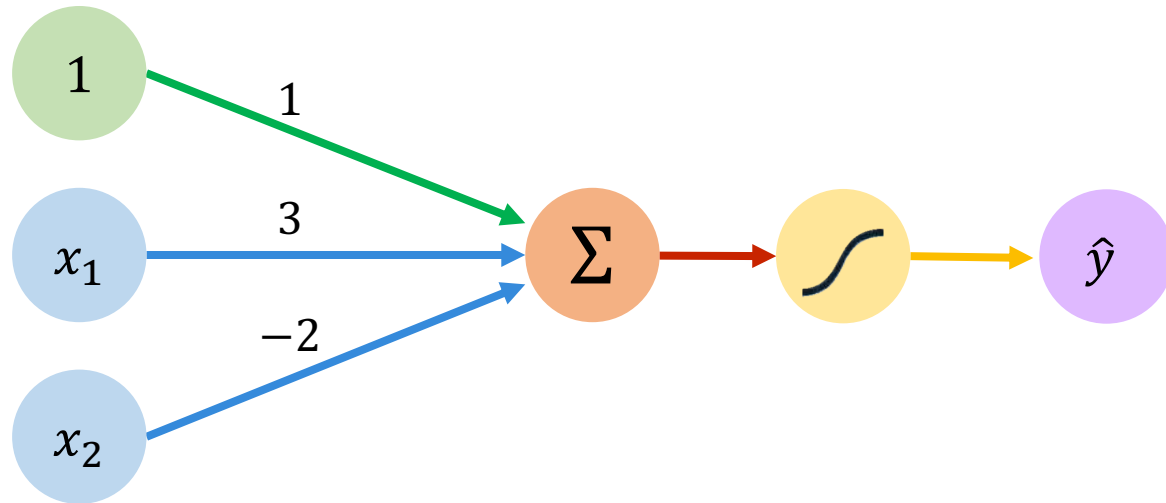
# The Perceptron: Example



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



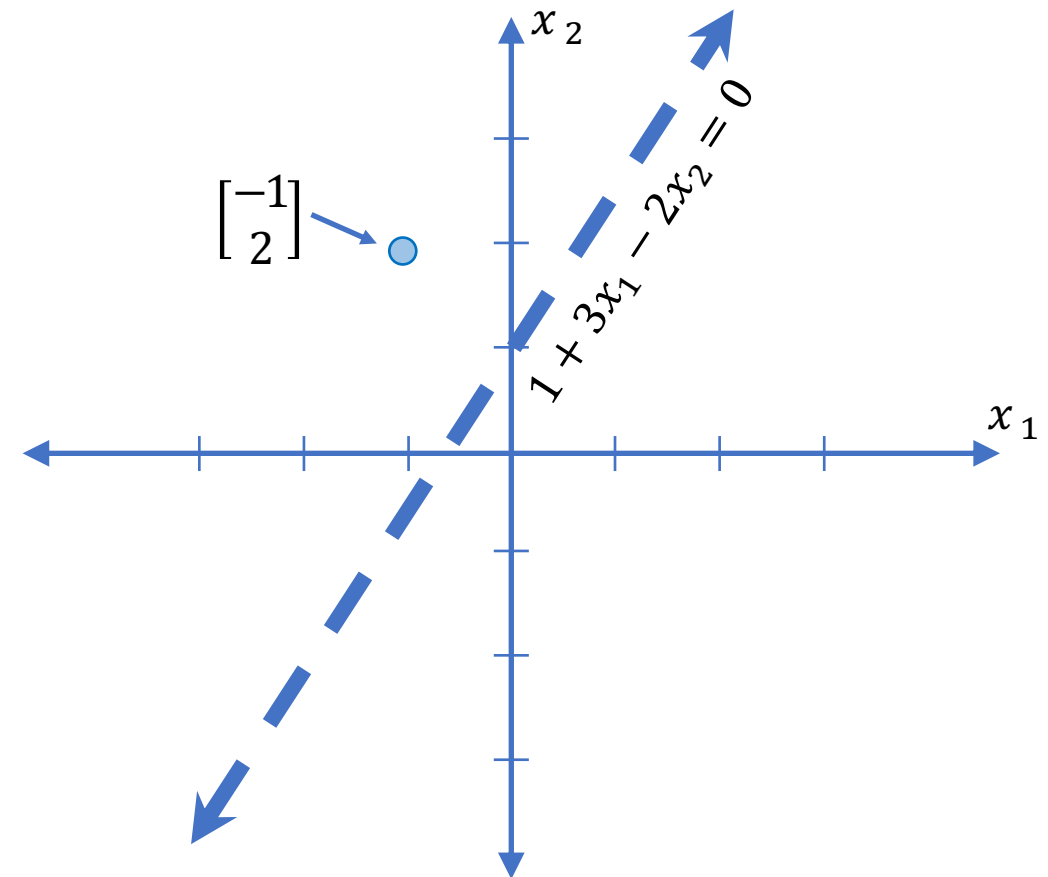
# The Perceptron: Example



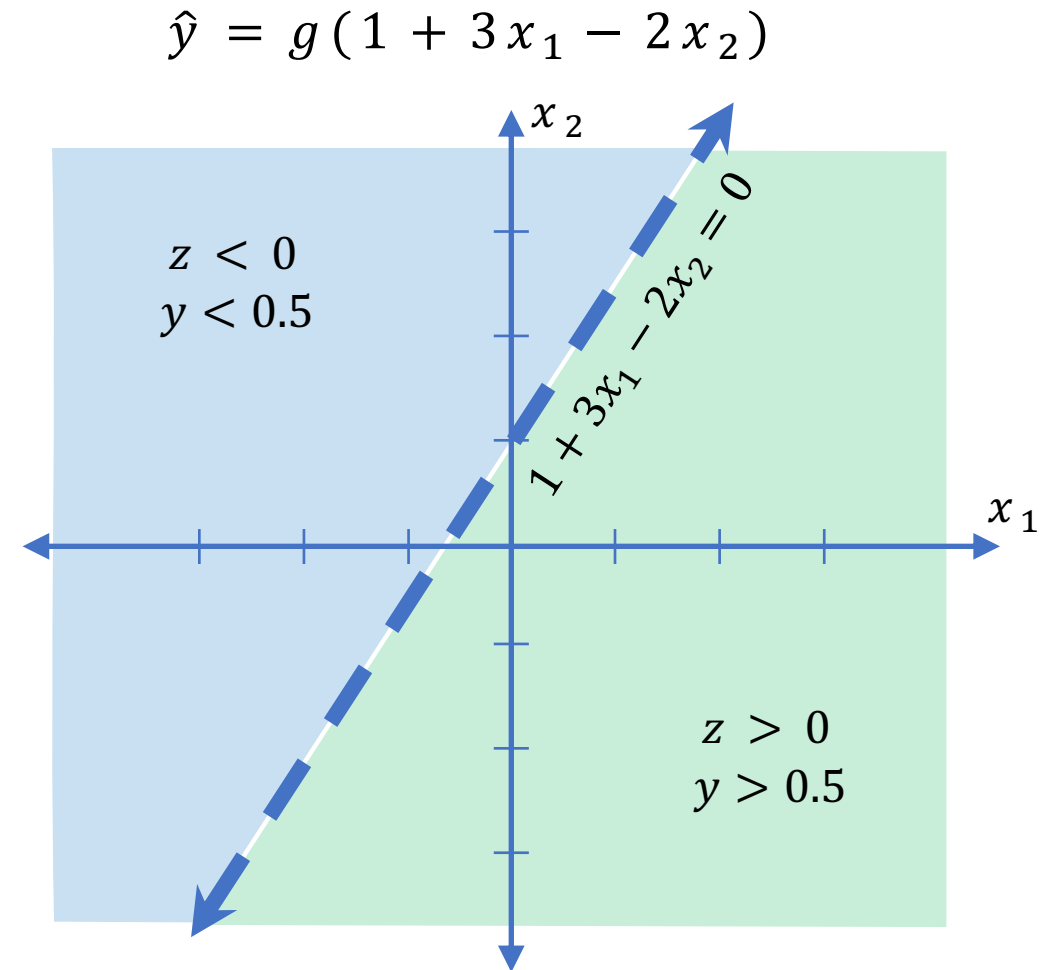
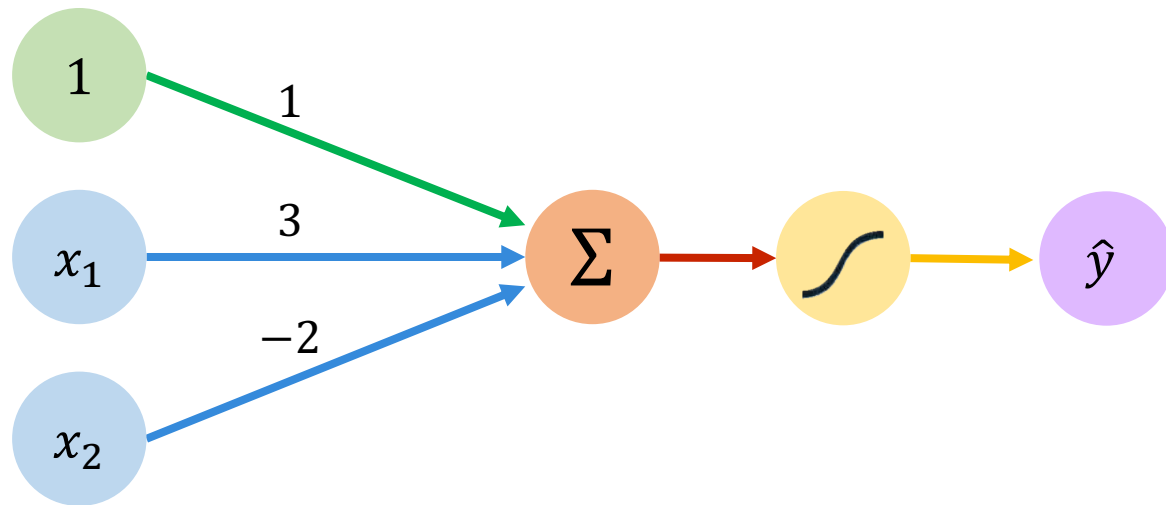
Assume we have input:  $\mathbf{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

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

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



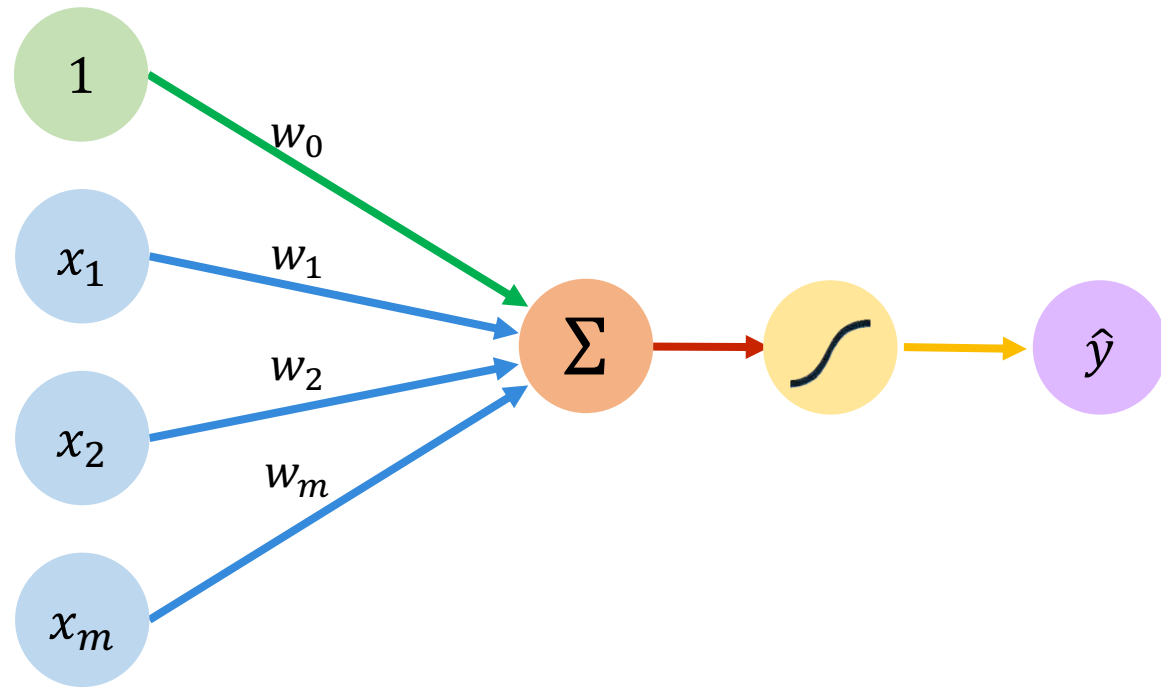
# The Perceptron: Example



# Building Neural Networks with Perceptrons



# The Perceptron: Simplified



Inputs

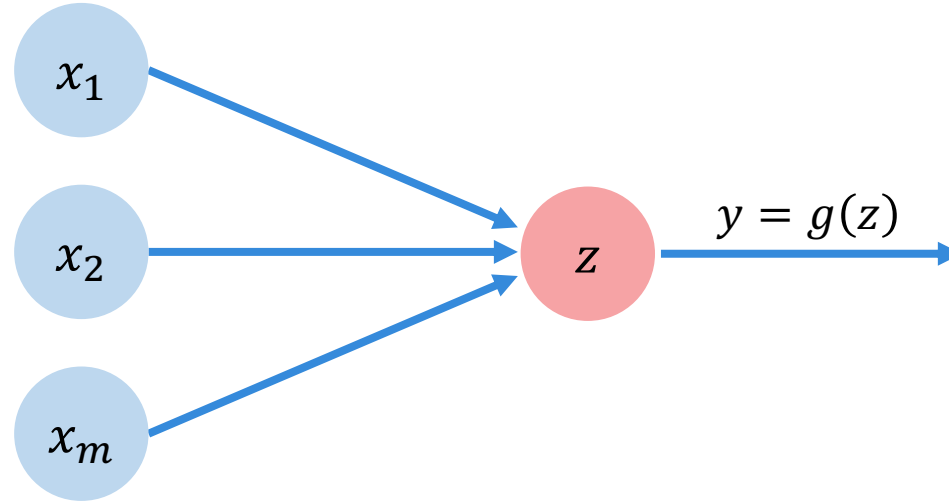
Weights

Sum

Non-Linearity

Output

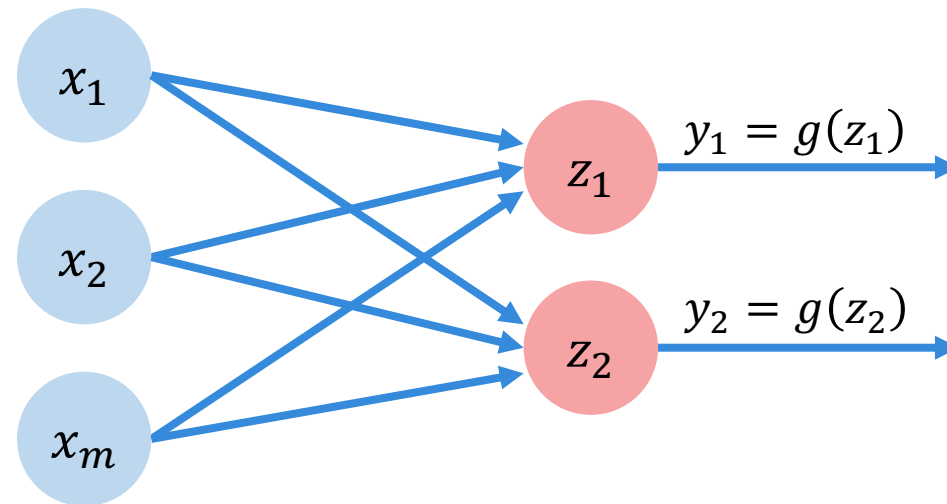
# The Perceptron: Simplified



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

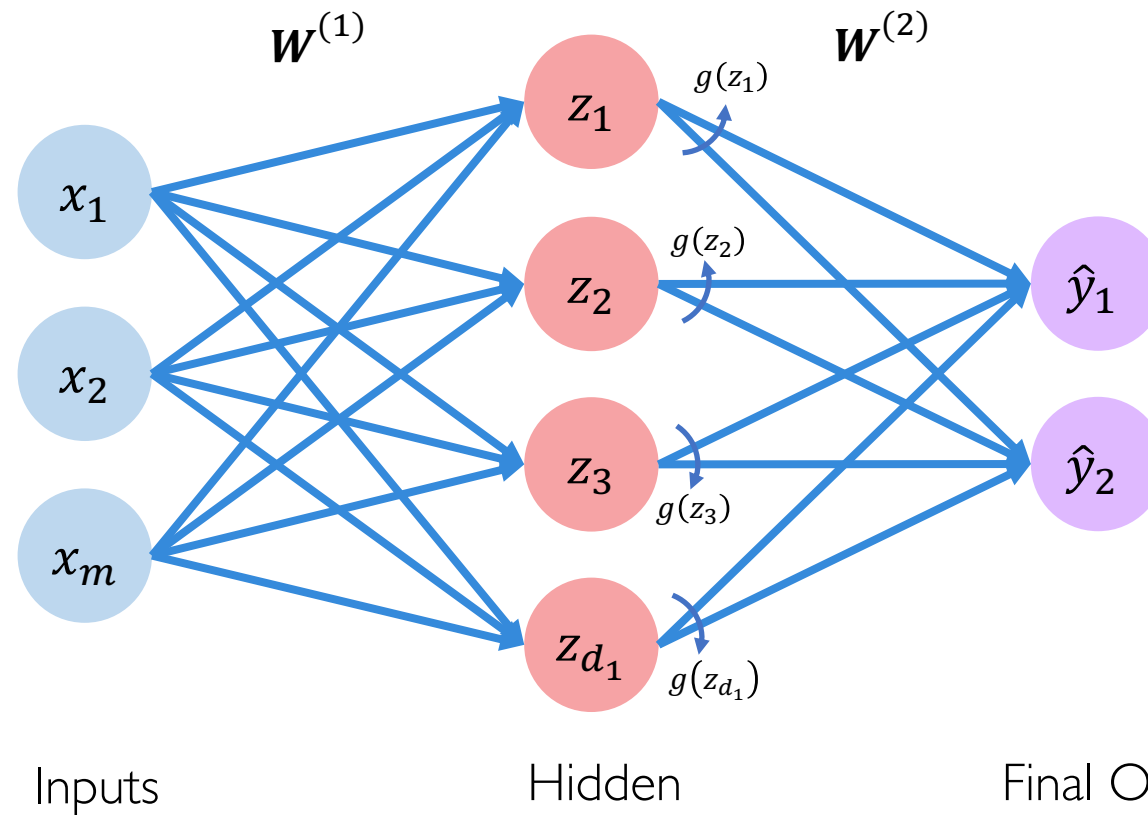
---

# Multi Output Perceptron



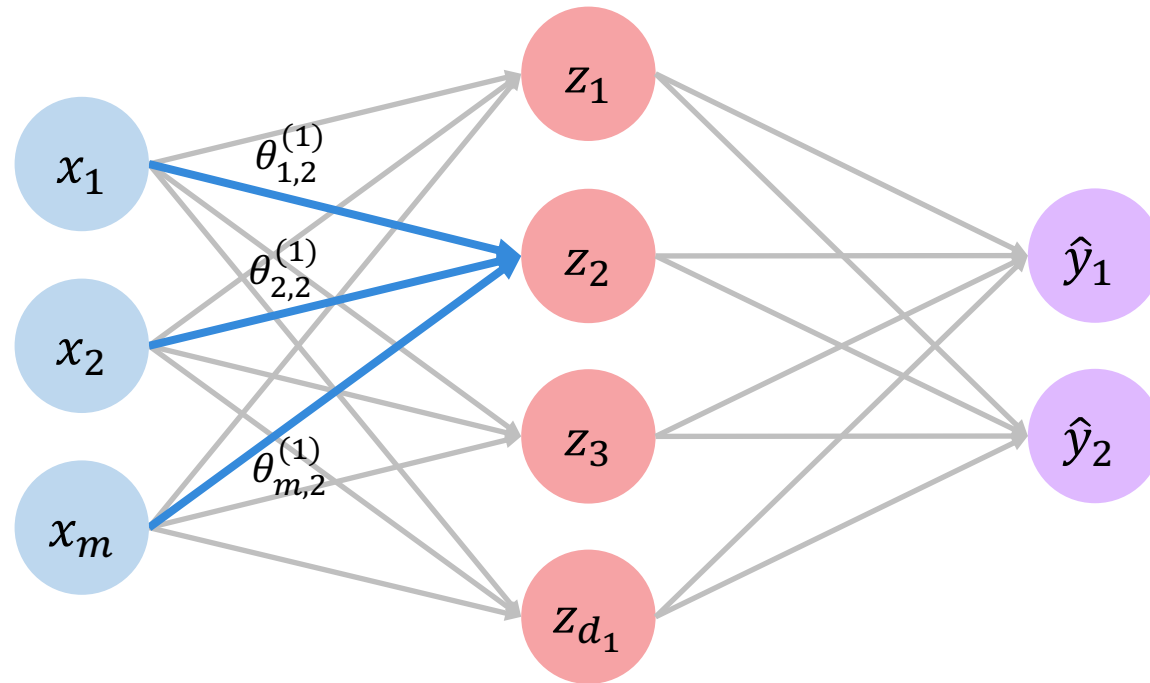
$$z_i = w_{0,i} + \sum_{j=1}^m x_j w_{j,i}$$

# Single Layer Neural Network



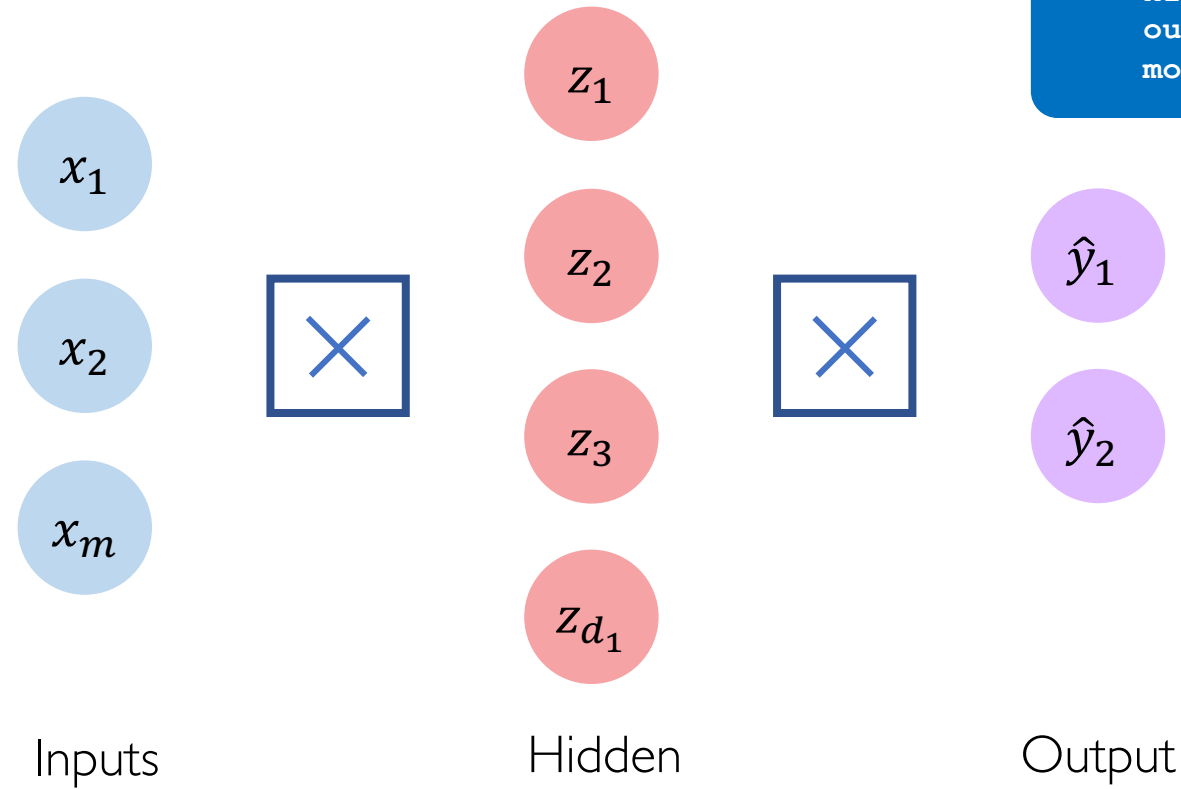
$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^m x_j w_{j,i}^{(1)} \quad \hat{y}_i = g \left( w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)} \right)$$

# Single Layer Neural Network



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

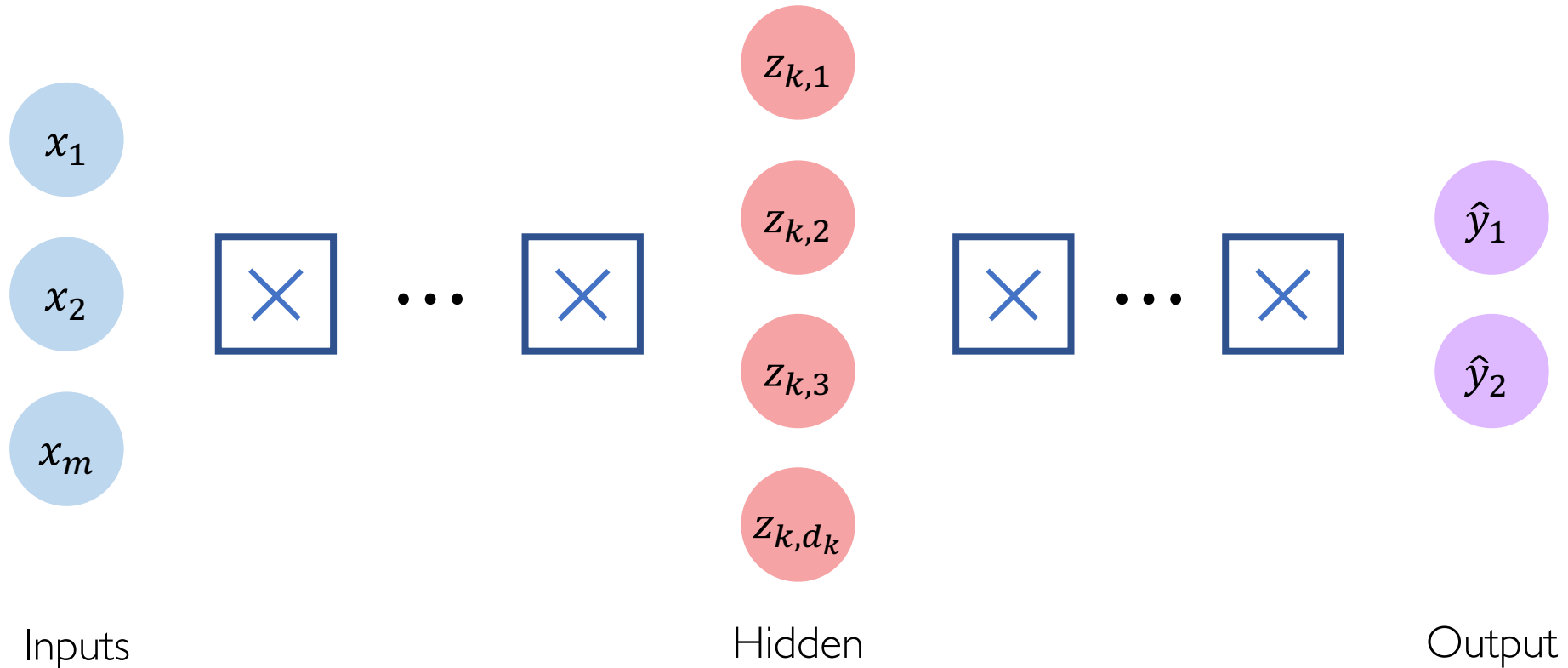
# Multi Output Perceptron



```
from tf.keras.layers import *  
  
inputs = Inputs(m)  
hidden = Dense(d1)(inputs)  
outputs = Dense(2)(hidden)  
model = Model(inputs, outputs)
```



# Deep Neural Network



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

# Applying Neural Networks

# Example Problem

Will I pass this class?

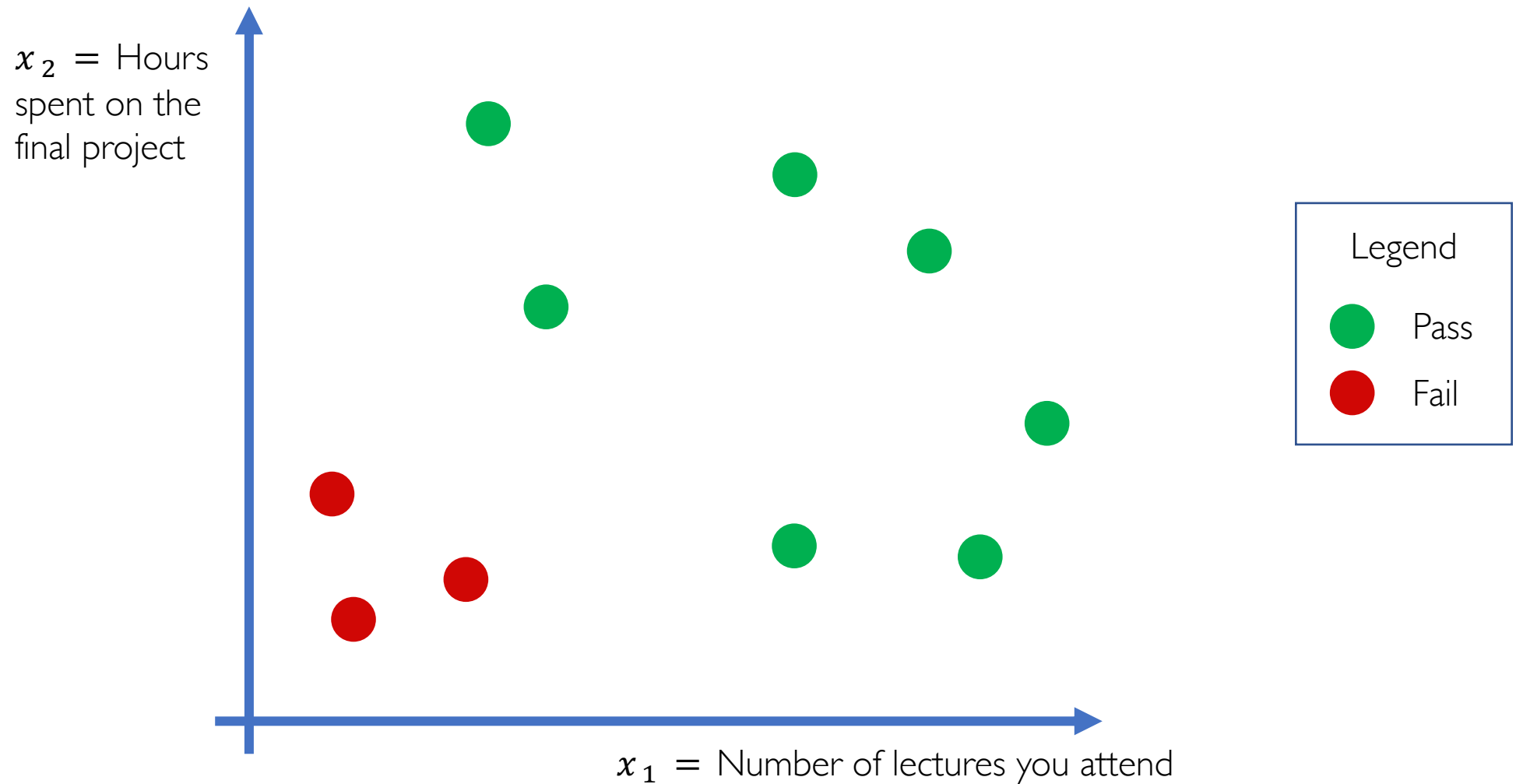
Let's start with a simple two feature model

$x_1$  = Number of lectures you attend

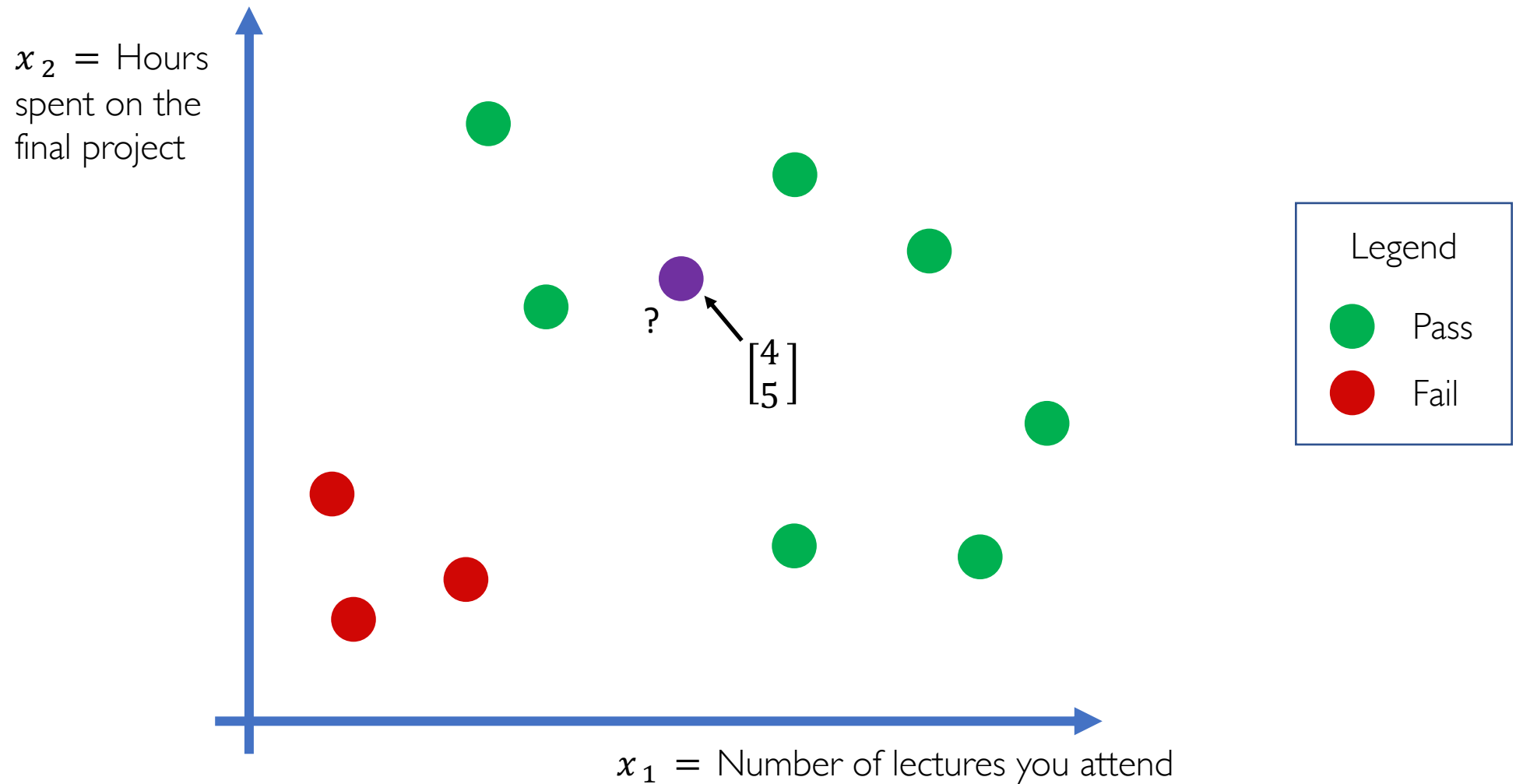
$x_2$  = Hours spent on the final project

---

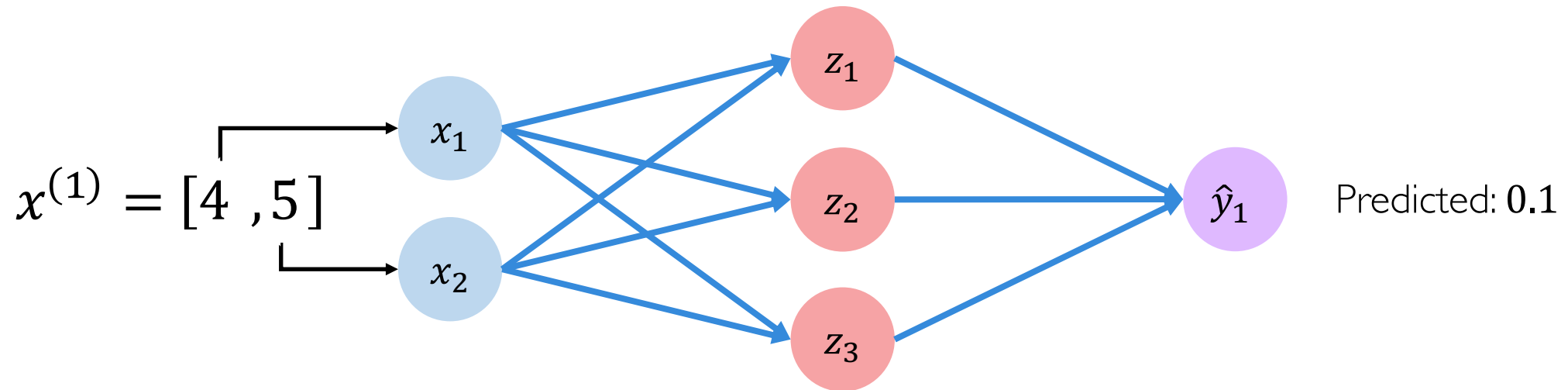
# Example Problem: Will I pass this class?



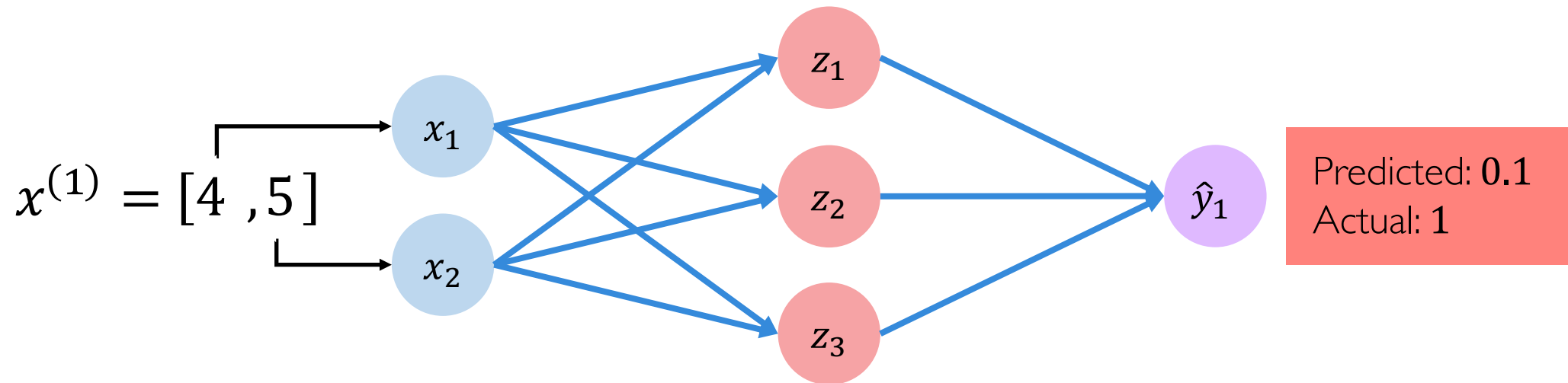
# Example Problem: Will I pass this class?



# Example Problem: Will I pass this class?

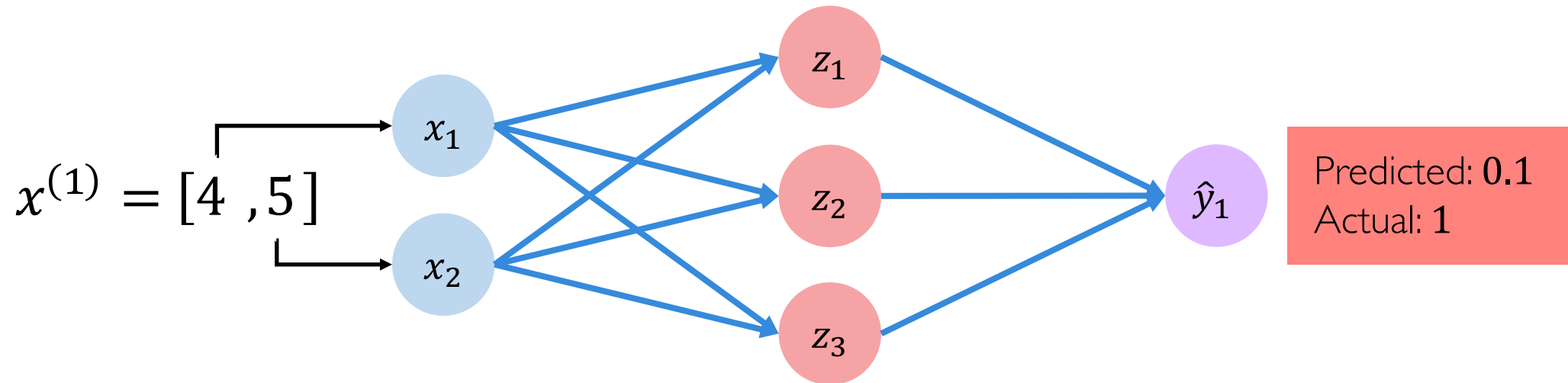


# Example Problem: Will I pass this class?



# Quantifying Loss

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

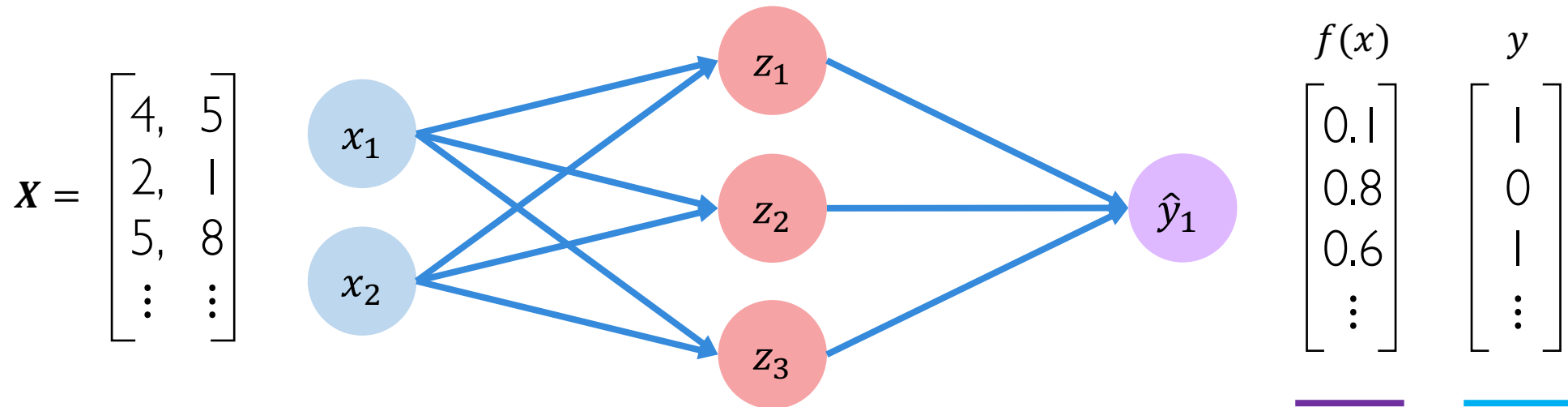


$$\mathcal{L}(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$



# Empirical Loss

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

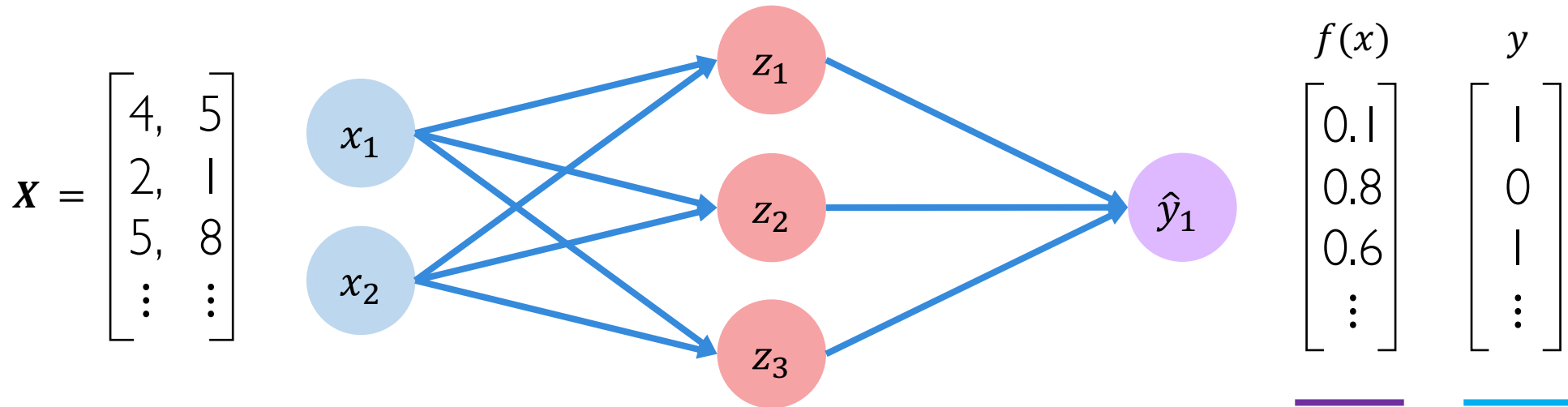


- Also known as:
- Objective function
  - Cost function
  - Empirical Risk

$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

# Binary Cross Entropy Loss

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



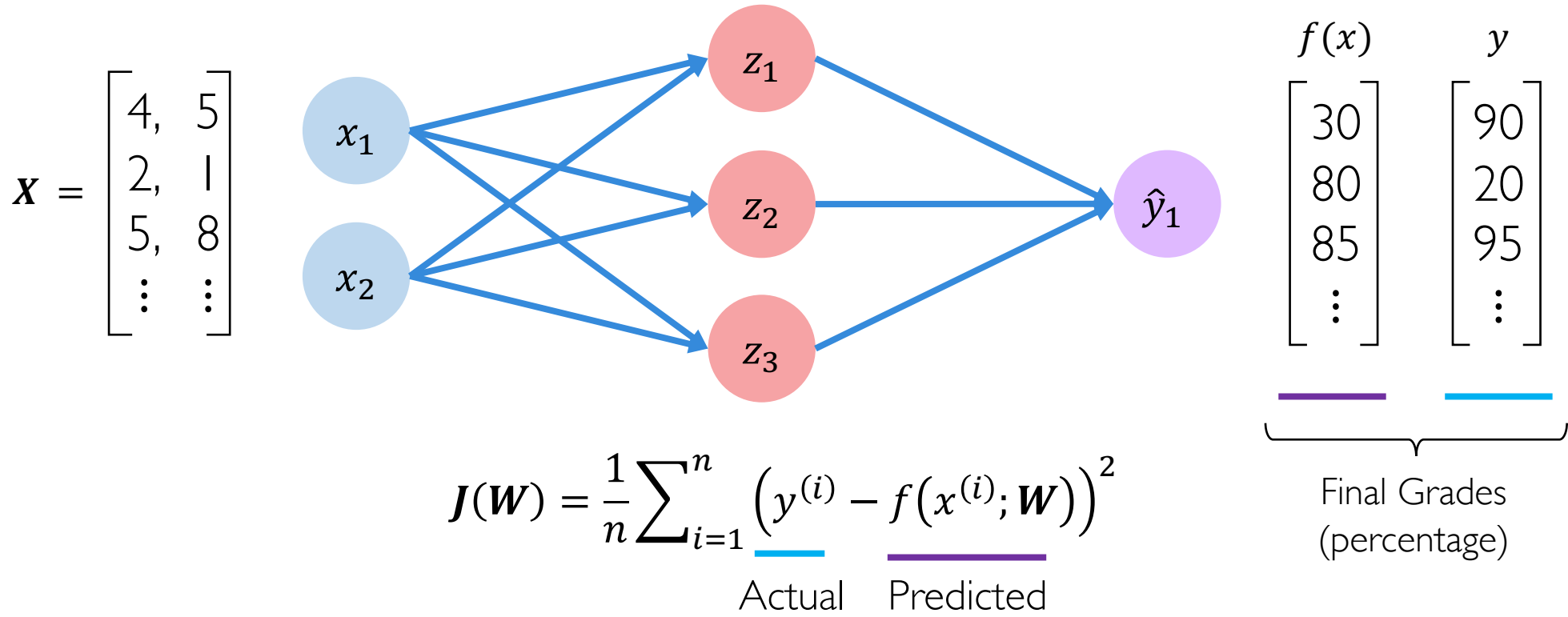
$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \underbrace{y^{(i)}}_{\text{Actual}} \log \left( \underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}} \right) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log \left( 1 - \underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}} \right)$$



```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(model.y, model.pred) )
```

# Mean Squared Error Loss

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



```
loss = tf.reduce_mean( tf.square( tf.subtract( model.y, model.pred ) ) )
```

# Training Neural Networks

# Loss Optimization

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

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

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

---

# Loss Optimization

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

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

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



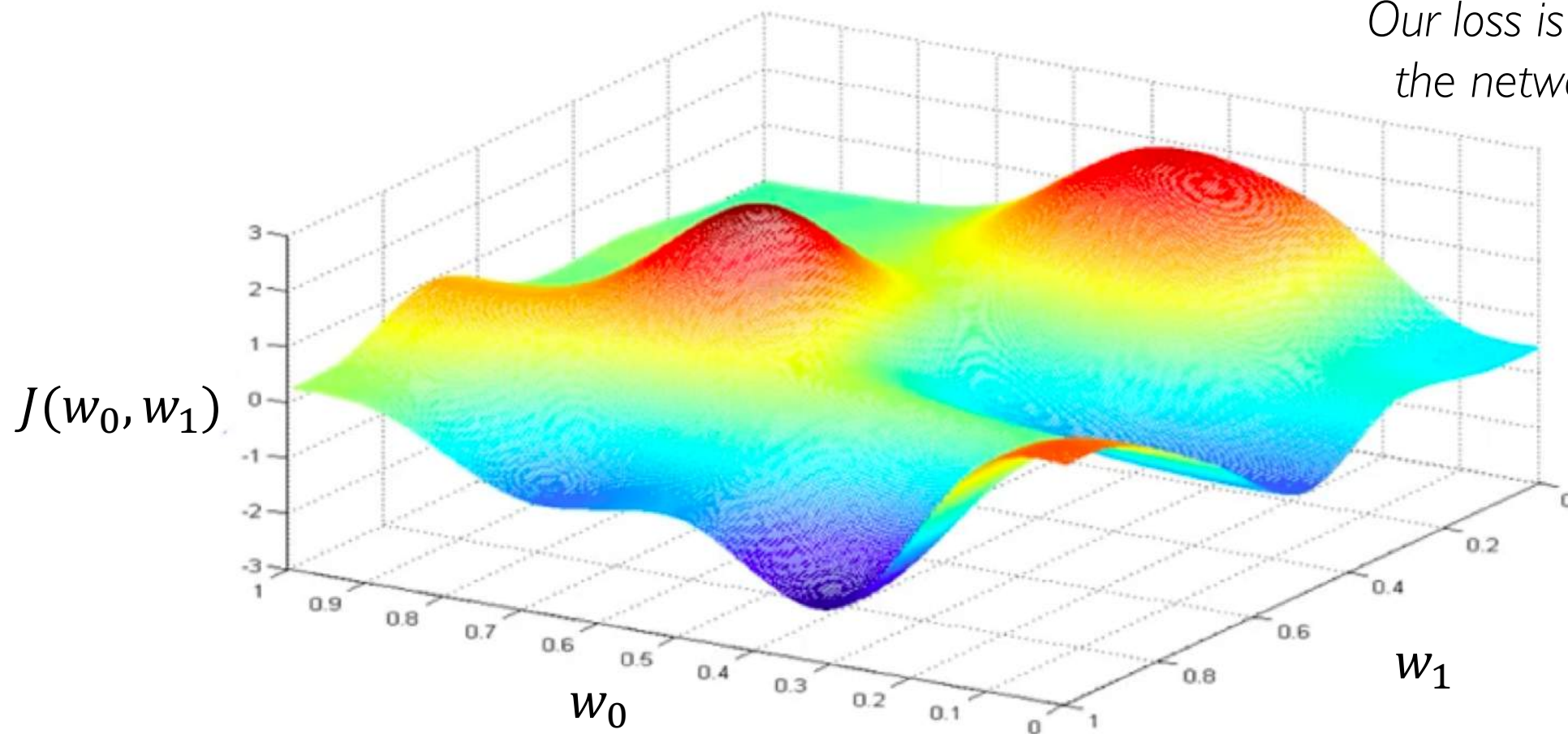
Remember:

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

# Loss Optimization

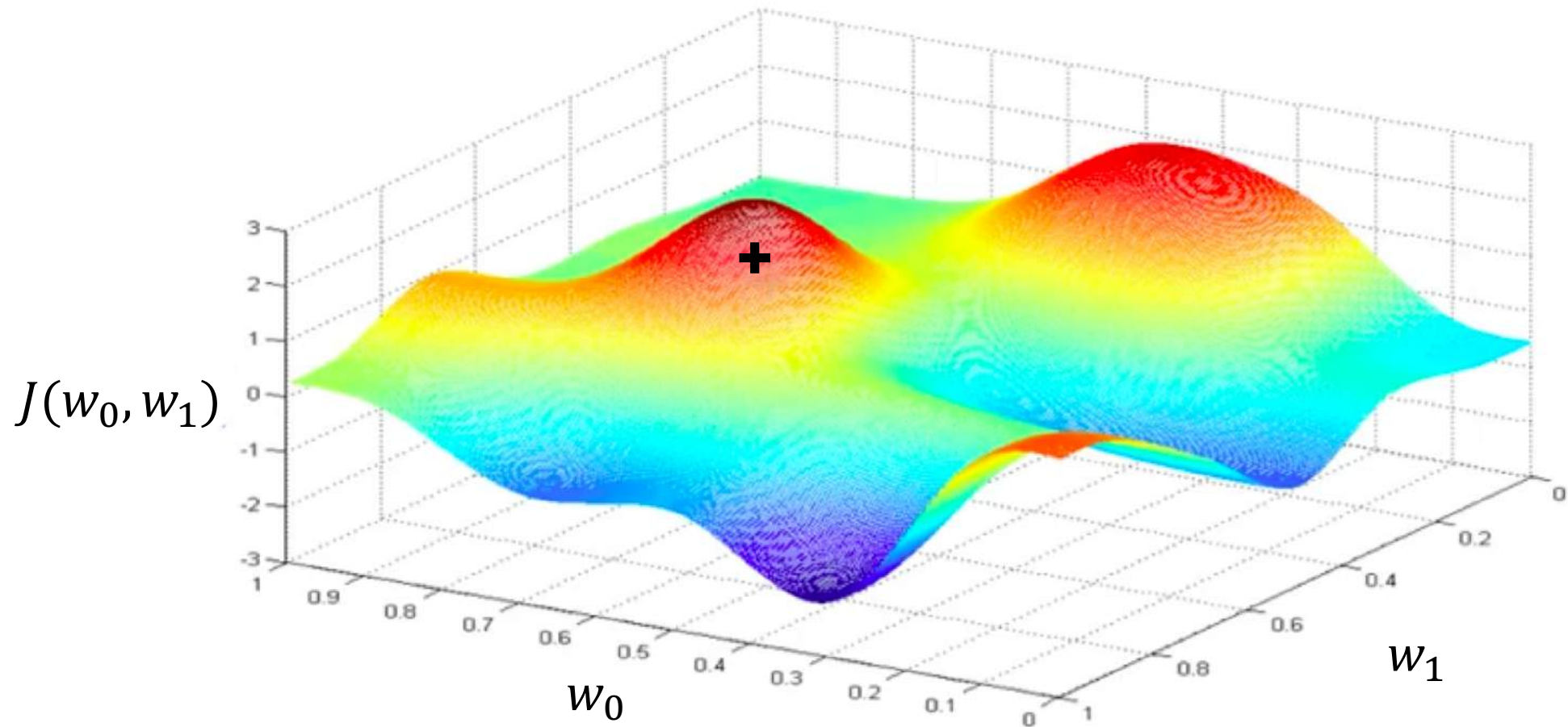
$$W^* = \underset{W}{\operatorname{argmin}} J(W)$$

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



# Loss Optimization

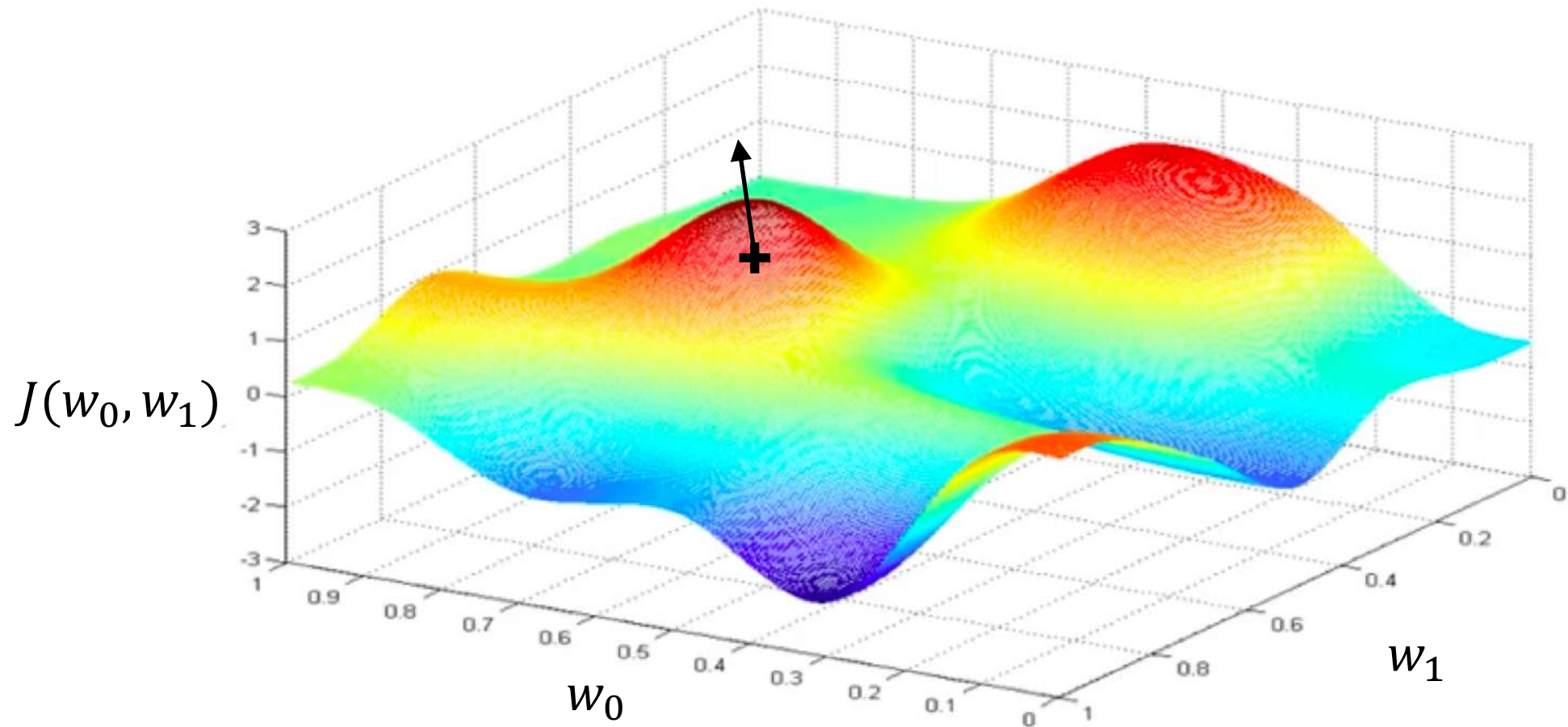
Randomly pick an initial  $(w_0, w_1)$





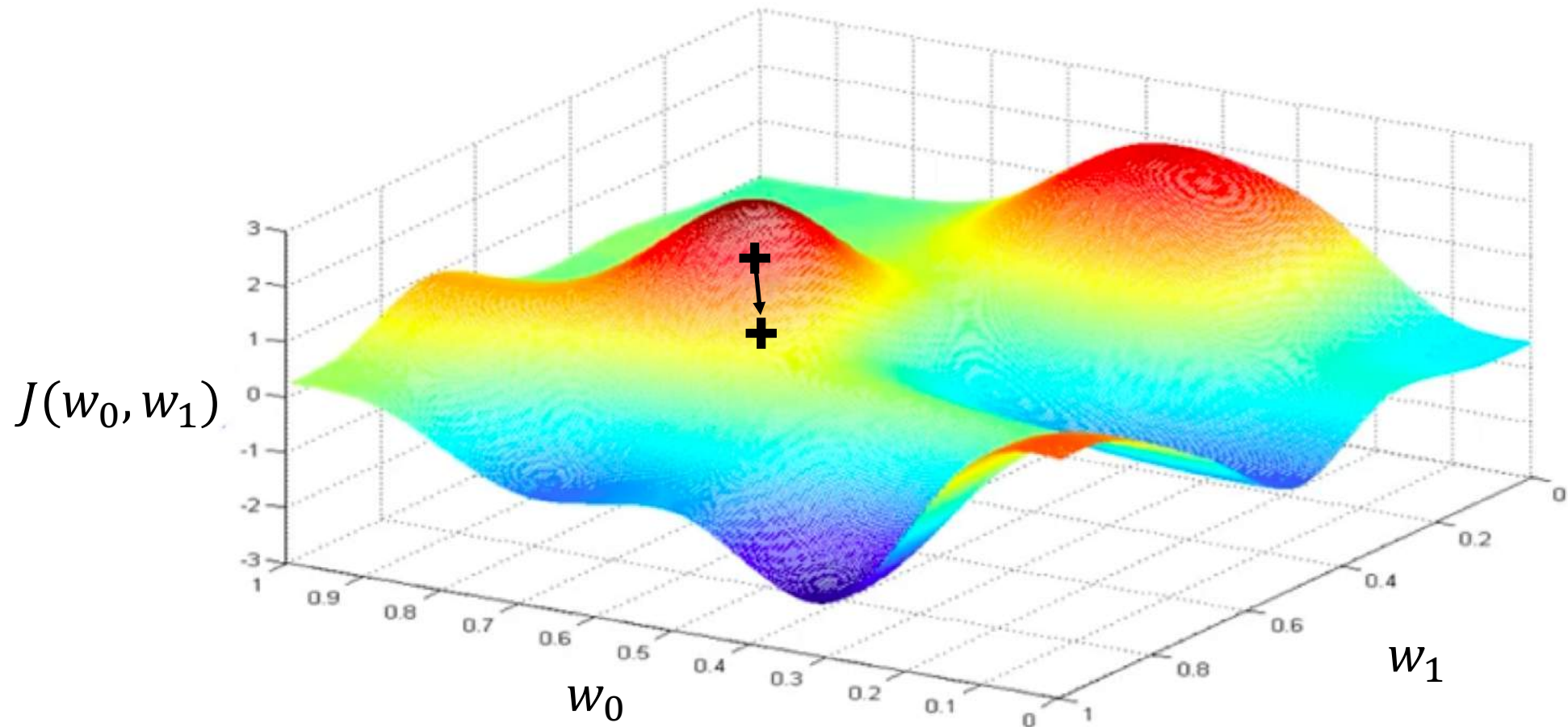
# Loss Optimization

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



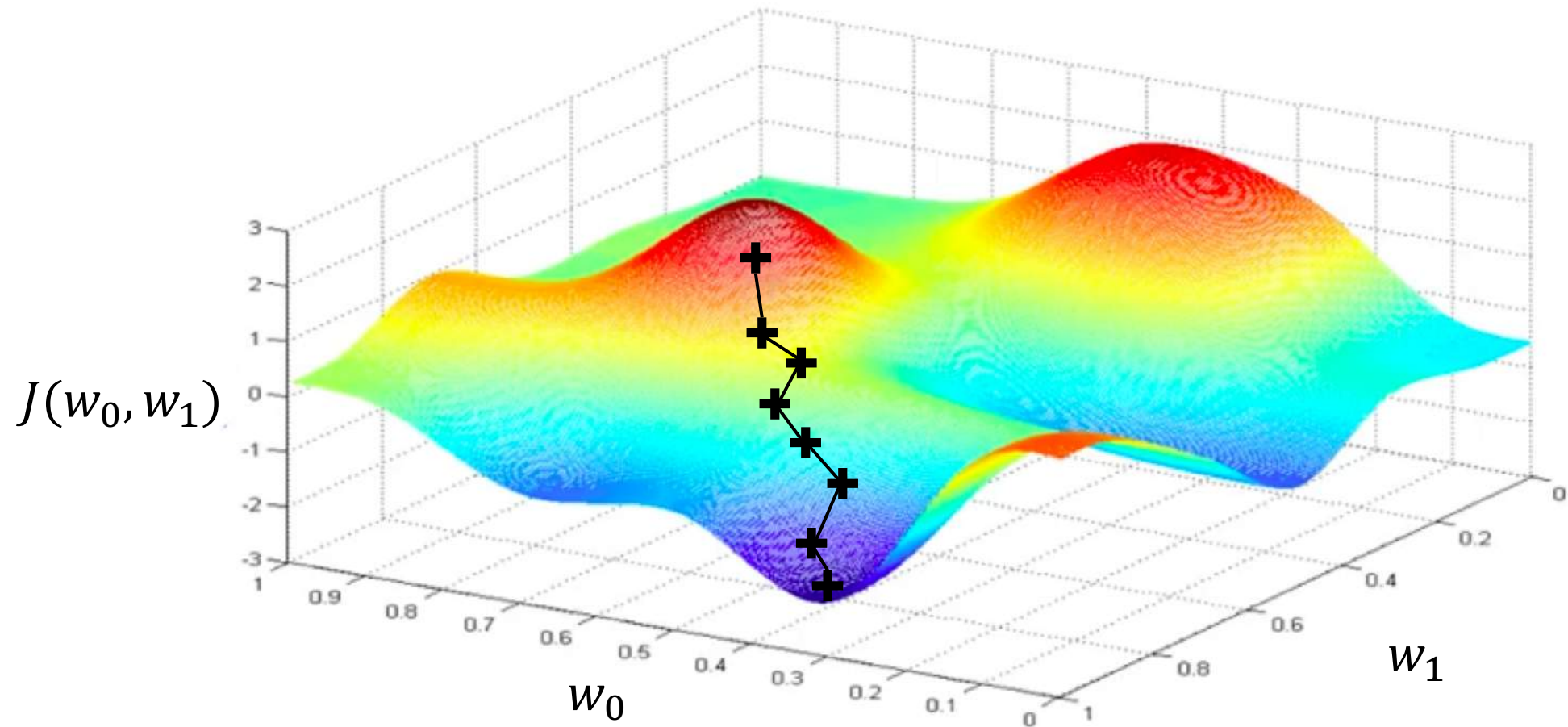
# Loss Optimization

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

```
 weights = tf.random_normal(shape, stddev=sigma)
```

```
 grads = tf.gradients(ys=loss, xs=weights)
```

```
 weights_new = weights.assign(weights - lr * grads)
```

# Gradient Descent

## Algorithm

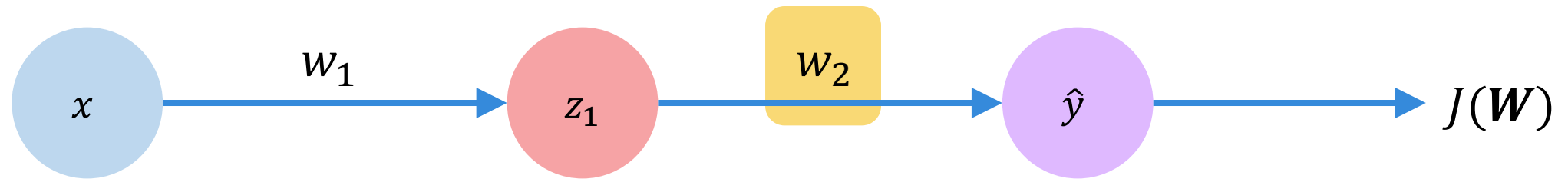
1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient,  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

```
 weights = tf.random_normal(shape, stddev=sigma)
```

```
 grads = tf.gradients(ys=loss, xs=weights)
```

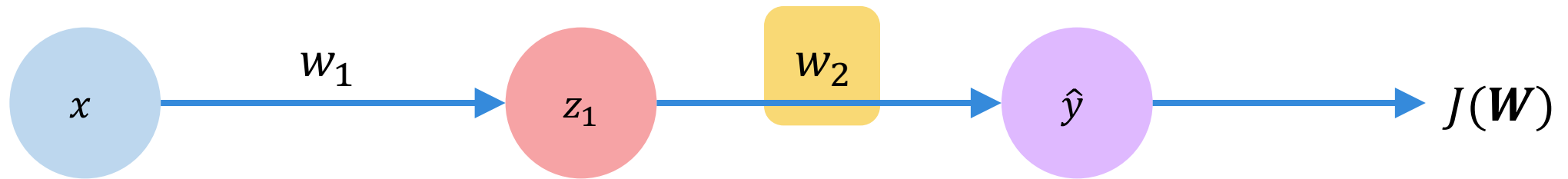
```
 weights_new = weights.assign(weights - lr * grads)
```

# Computing Gradients: Backpropagation



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

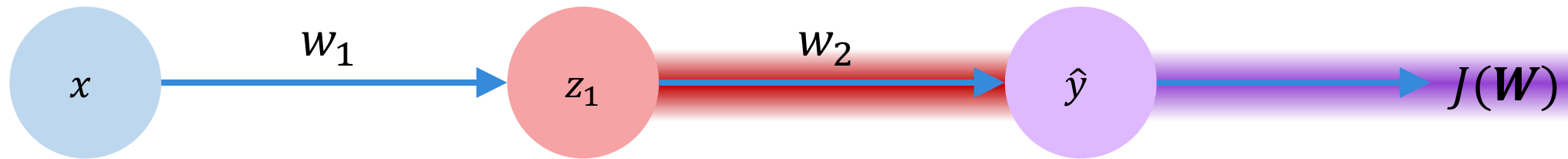
# Computing Gradients: Backpropagation



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

Let's use the chain rule!

# Computing Gradients: Backpropagation

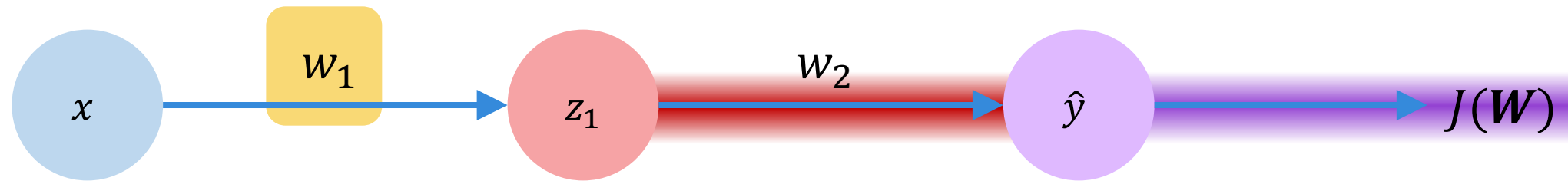


$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

A purple horizontal bar is positioned under the fraction  $\frac{\partial J(W)}{\partial \hat{y}}$ , and a red horizontal bar is positioned under the fraction  $\frac{\partial \hat{y}}{\partial w_2}$ .



# Computing Gradients: Backpropagation

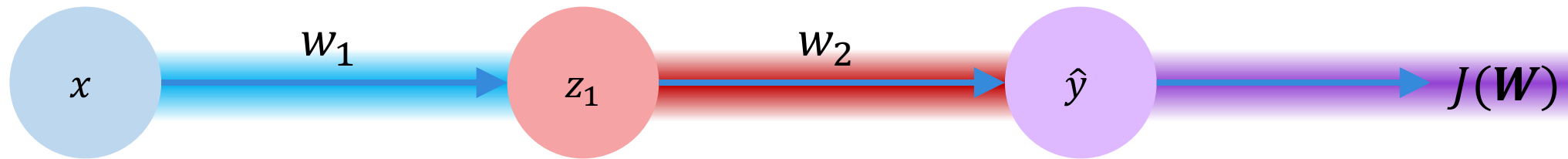


$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \frac{\partial J(\mathbf{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

Apply chain rule!

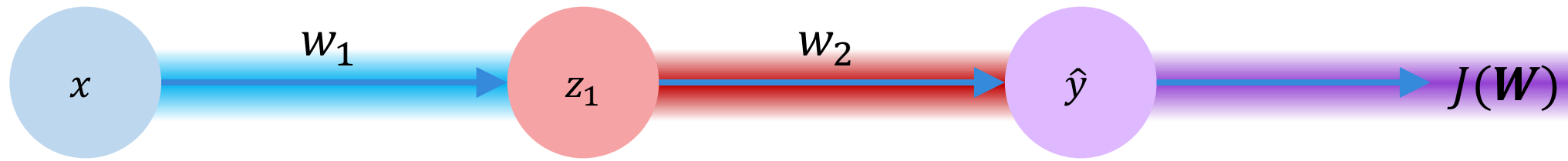
# Computing Gradients: Backpropagation



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



# Computing Gradients: Backpropagation

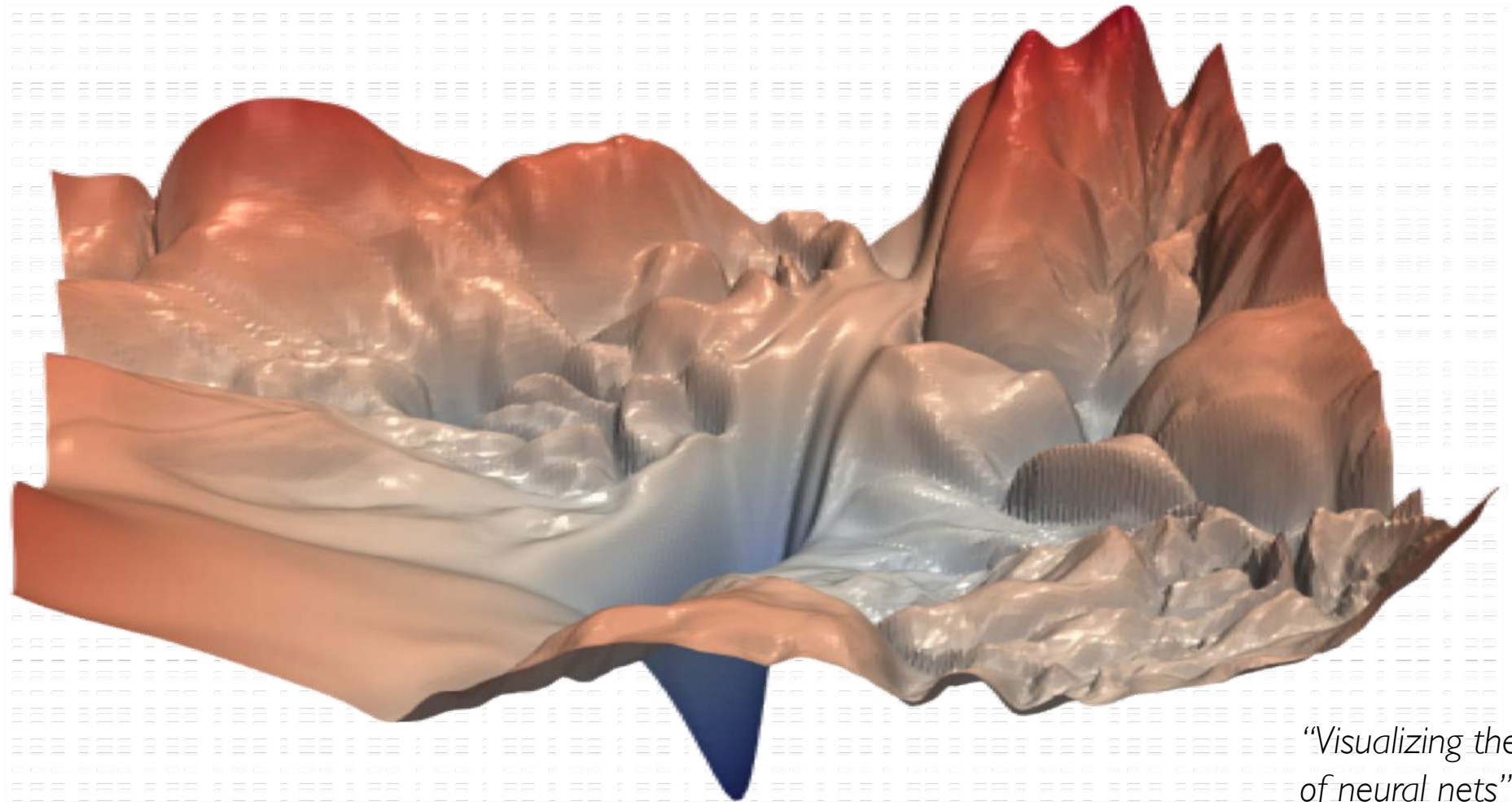


$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \frac{\partial J(\mathbf{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

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:

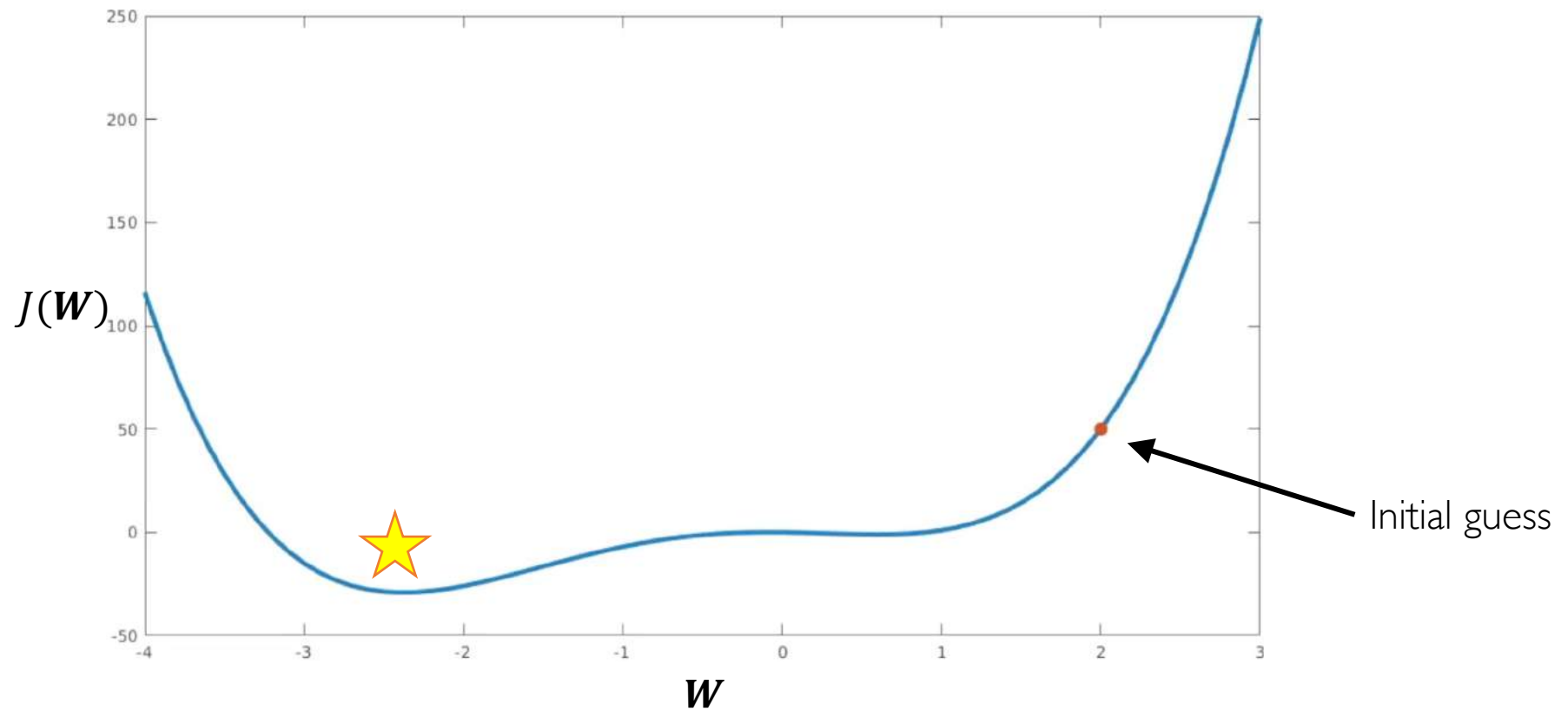
Optimization through gradient descent

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

How can we set the  
learning rate?

# Setting the Learning Rate

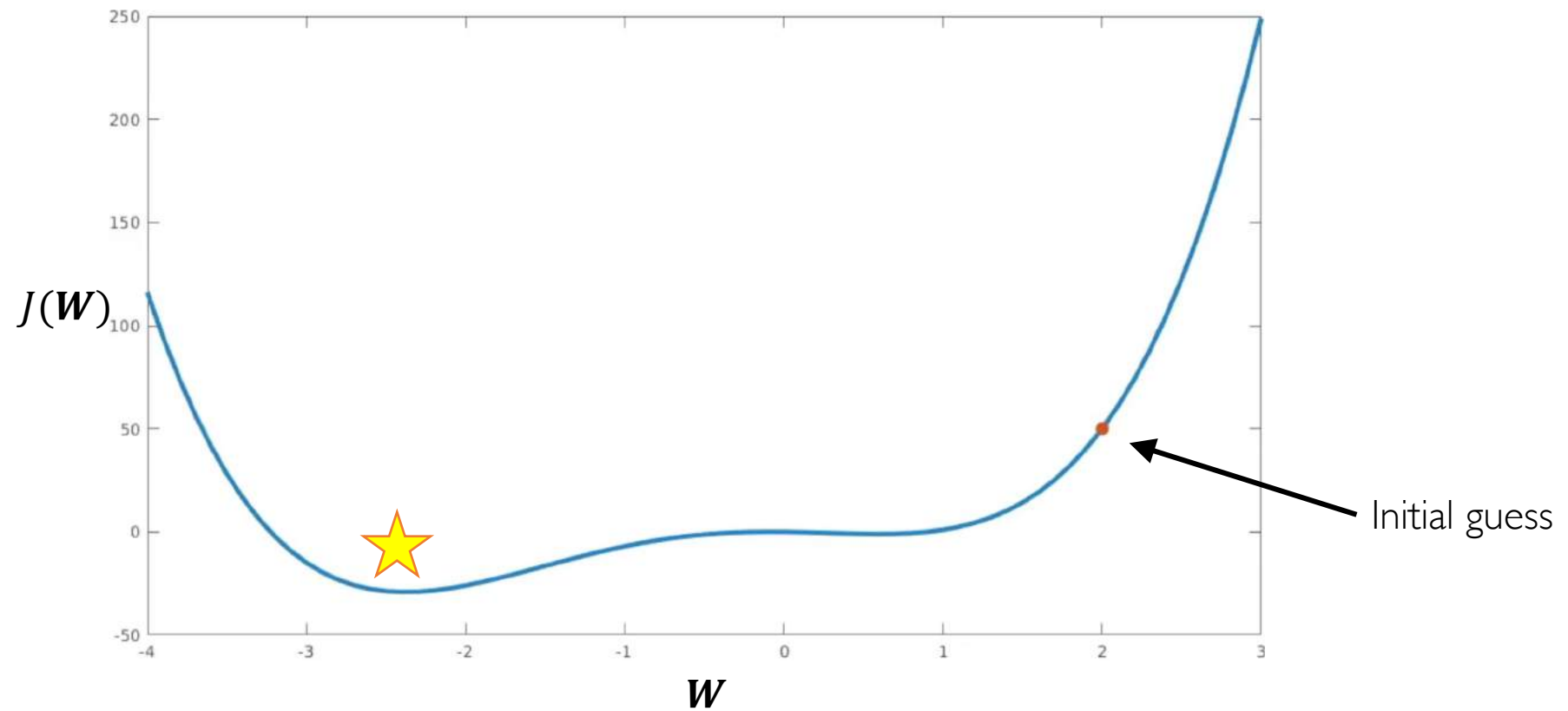
*Small learning rate converges slowly and gets stuck in false local minima*





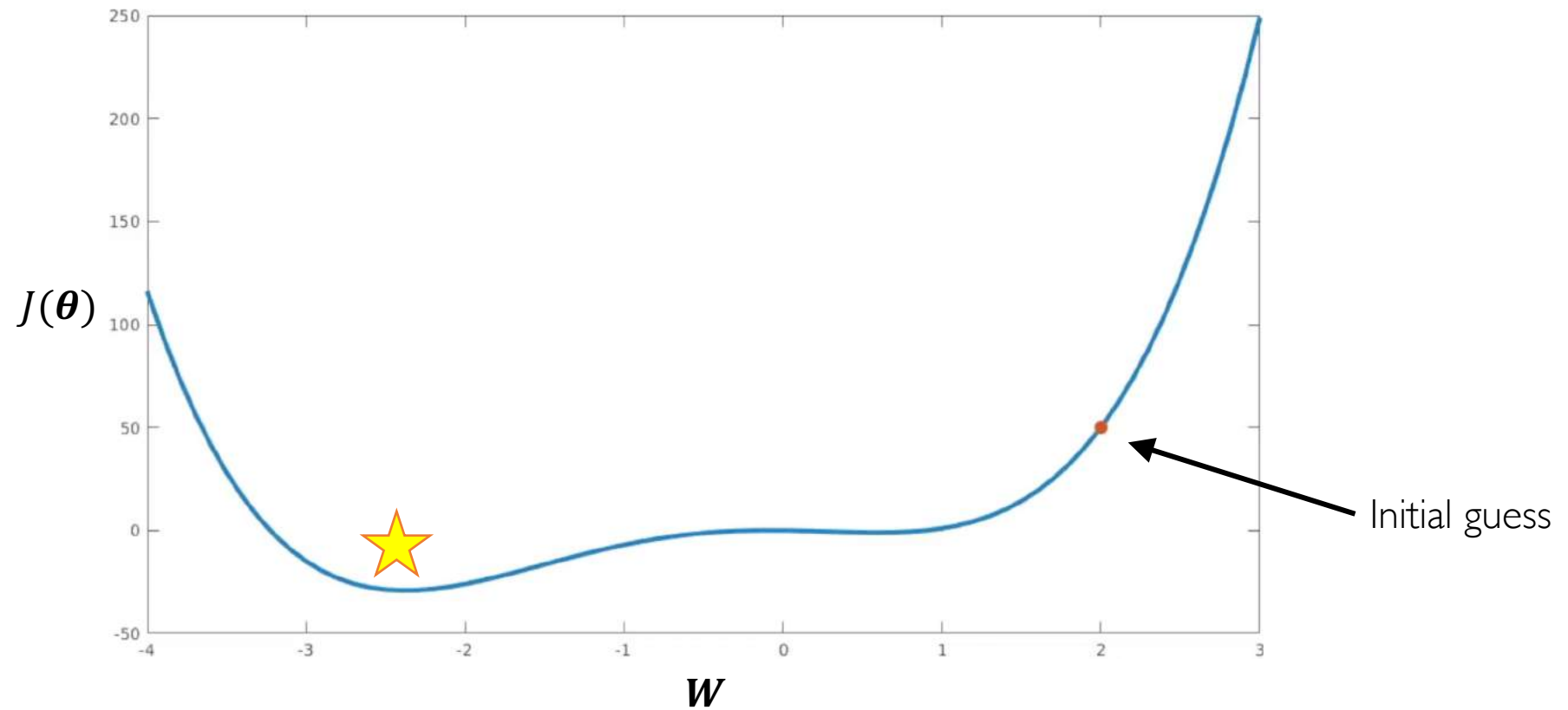
# Setting the Learning Rate

*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 1:**

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...
-

# Adaptive Learning Rate Algorithms

- Momentum



`tf.train.MomentumOptimizer`

Qian et al. "On the momentum term in gradient descent learning algorithms." 1999.

- Adagrad



`tf.train.AdagradOptimizer`

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

- Adadelta



`tf.train.AdadeltaOptimizer`

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

- Adam



`tf.train.AdamOptimizer`

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

- RMSProp



`tf.train.RMSPropOptimizer`

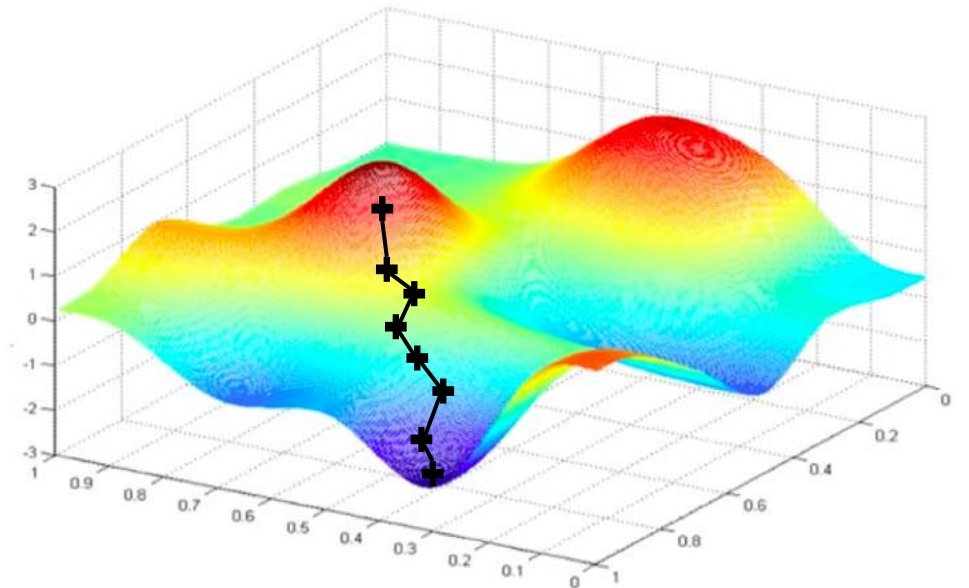


# Neural Networks in Practice: Mini-batches

# 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

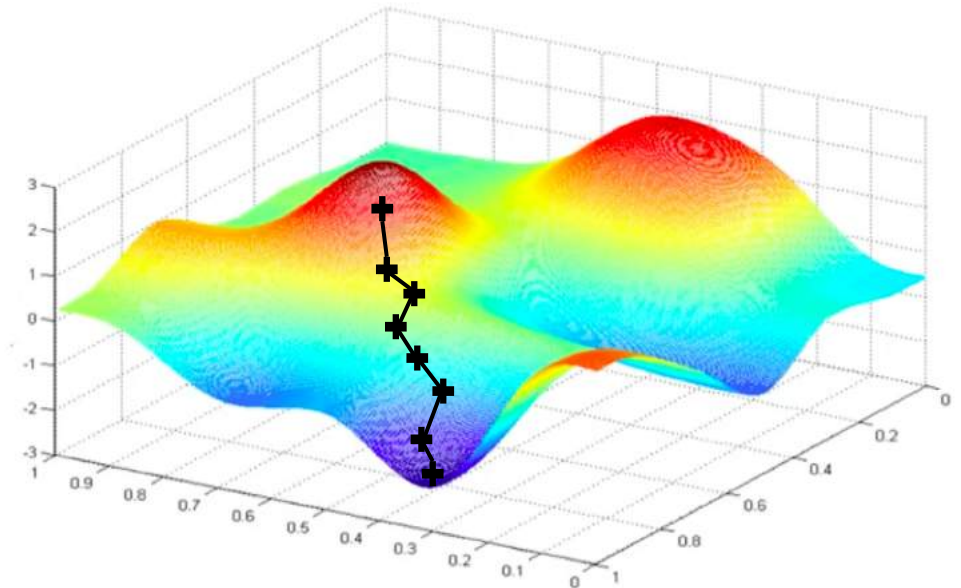




# Gradient Descent

## 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

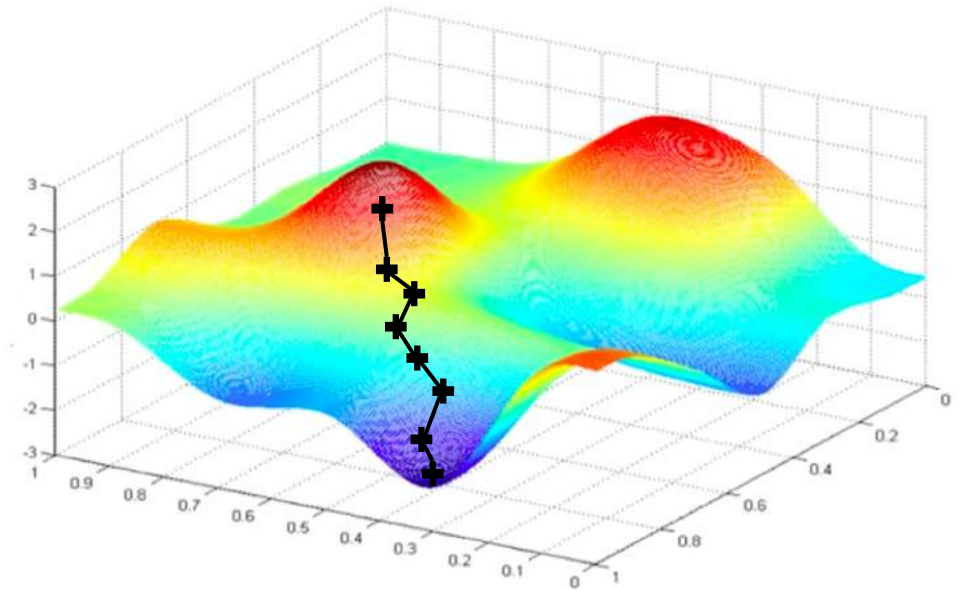


Can be very  
computational to  
compute!

# Stochastic Gradient Descent

## Algorithm

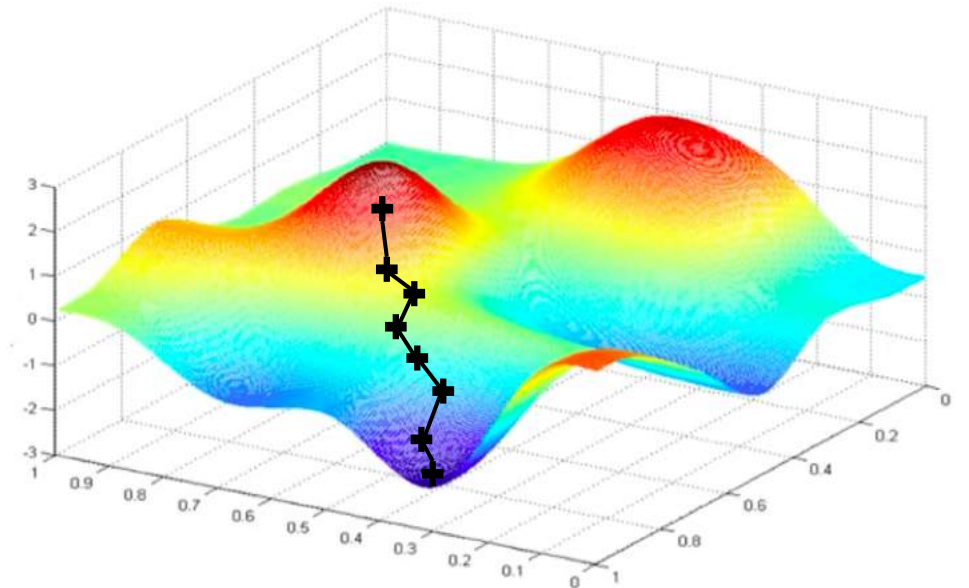
1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
  3. Pick single data point  $i$
  4. Compute gradient,  $\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}}$
  5. Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial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

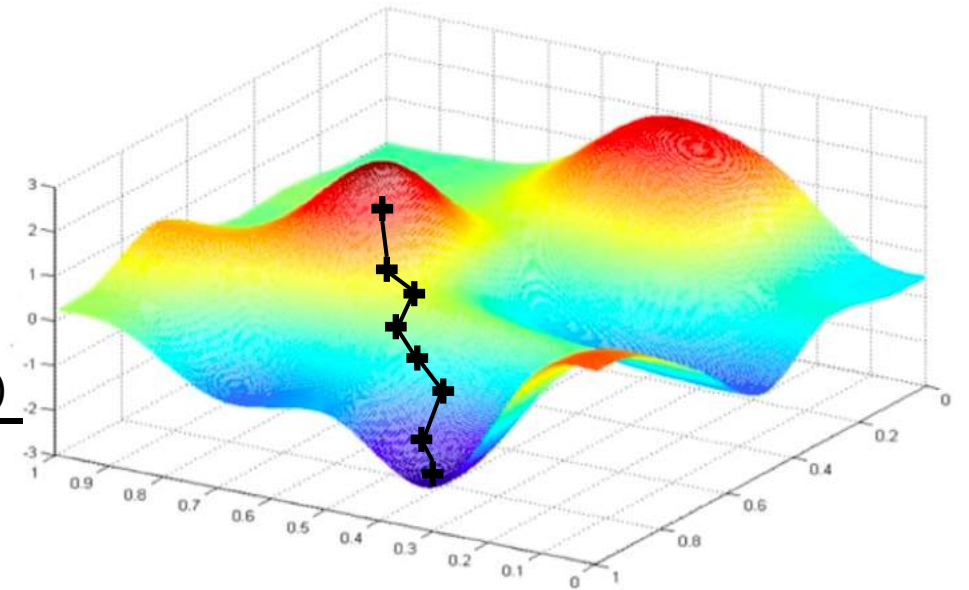


Easy to compute but  
**very noisy**  
(stochastic)!

# Stochastic Gradient Descent

## Algorithm

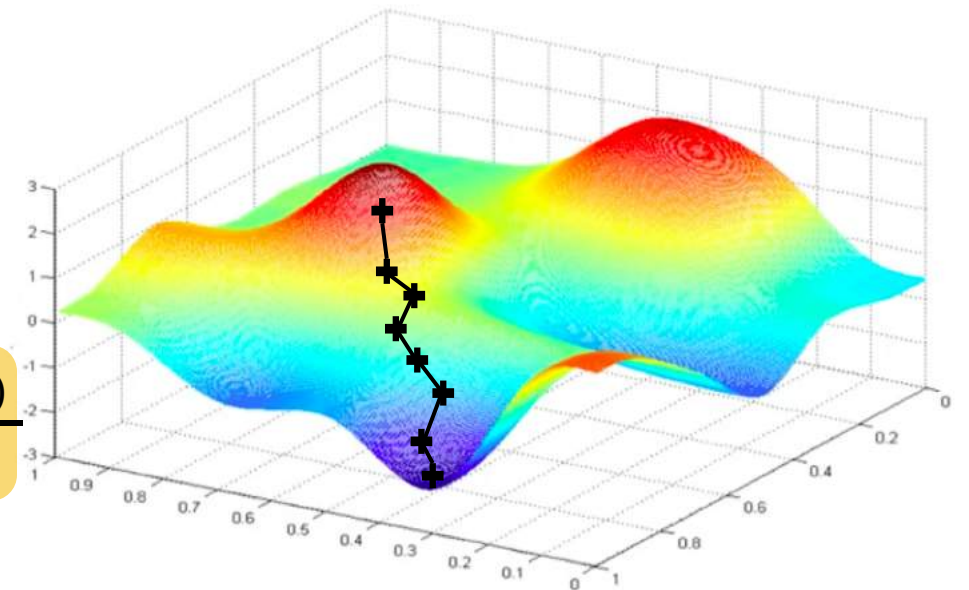
1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of  $B$  data points
4. Compute gradient,  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights, 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

Smother convergence

Allows for larger learning rates

---

# Mini-batches while training

More accurate estimation of gradient

Smother convergence  
Allows for larger learning rates

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

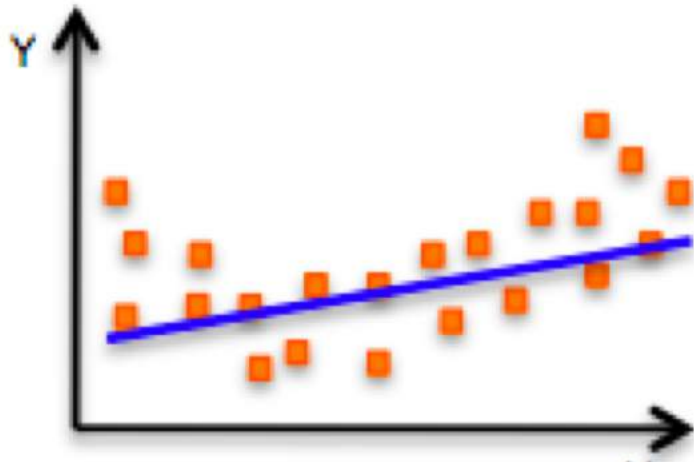
Can parallelize computation + achieve significant speed increases on GPU's

---

# Neural Networks in Practice: Overfitting

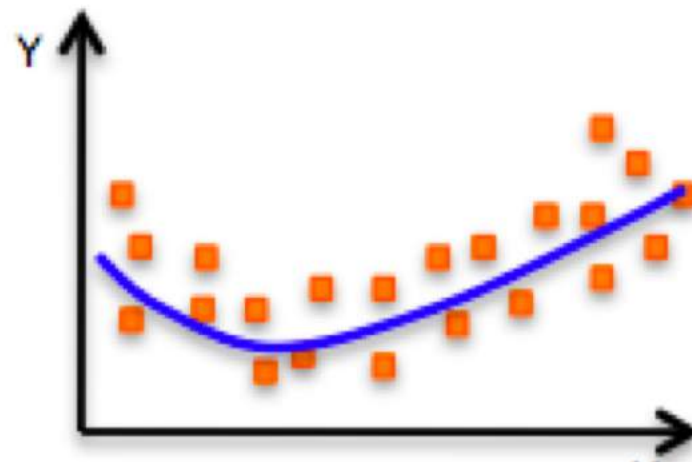


# The Problem of Overfitting

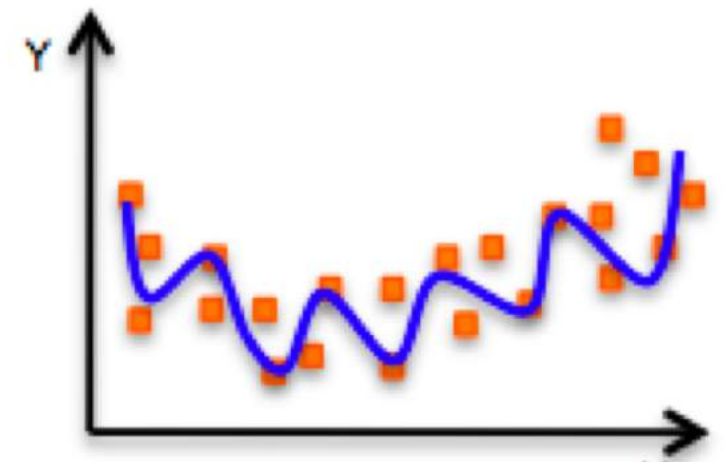


## Underfitting

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*

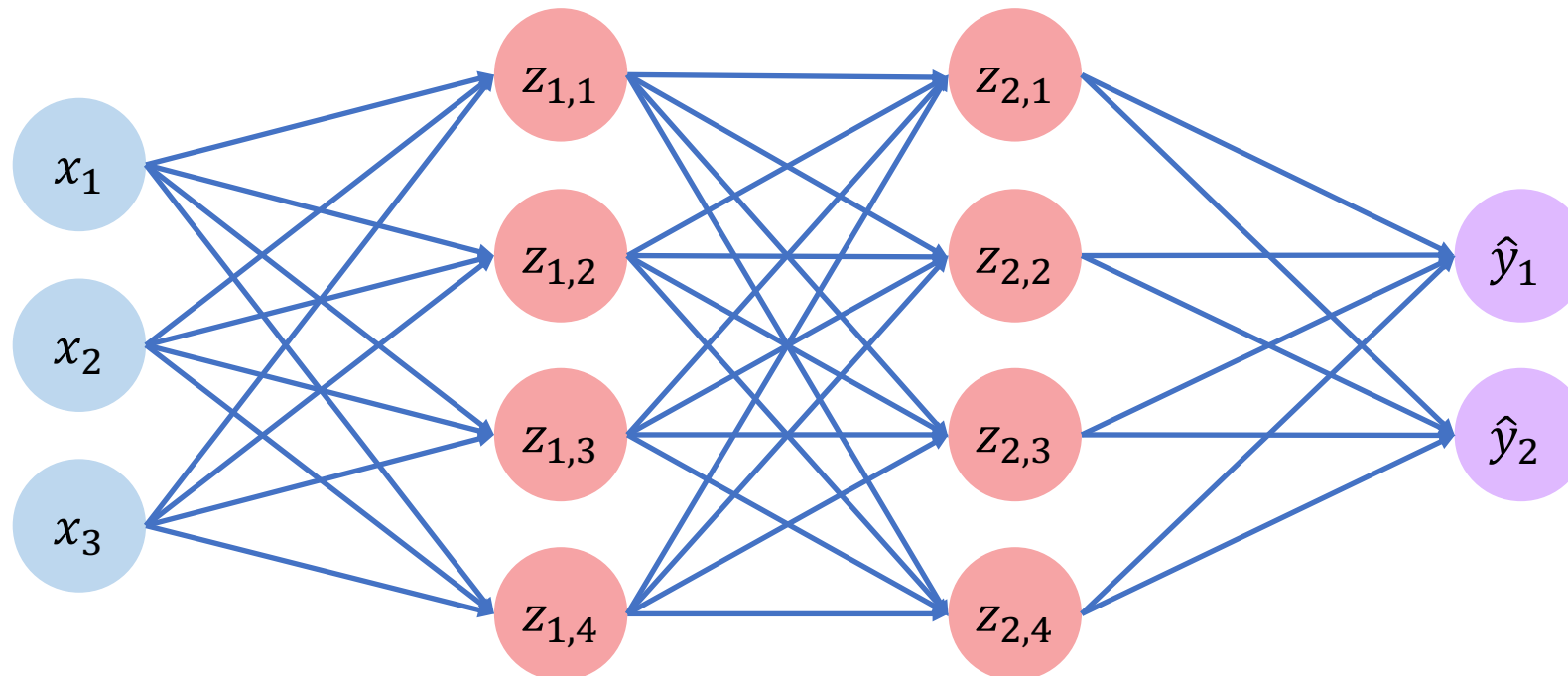
## **Why do we need it?**

*Improve generalization of our model on unseen data*

---

# Regularization I: Dropout

- During training, randomly set some activations to 0

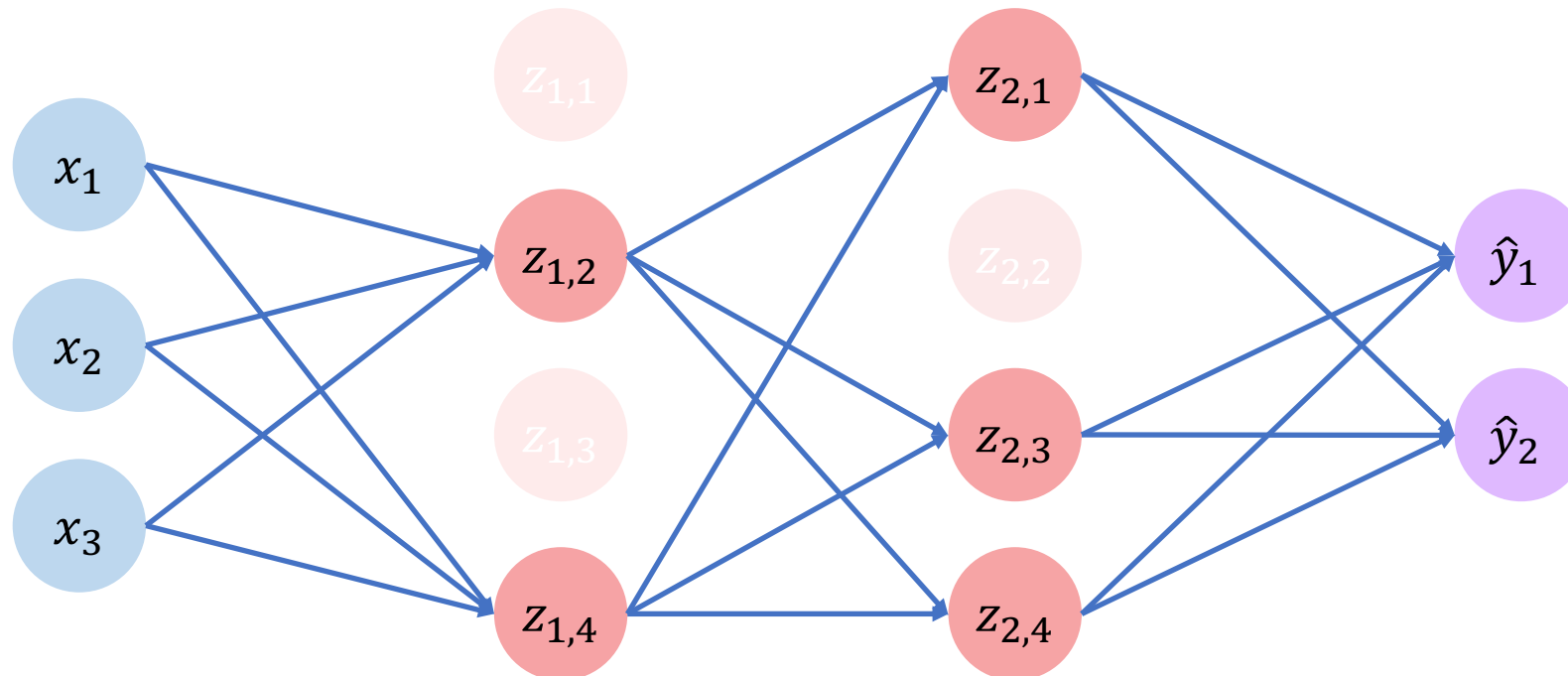


# Regularization I: Dropout

- 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)`

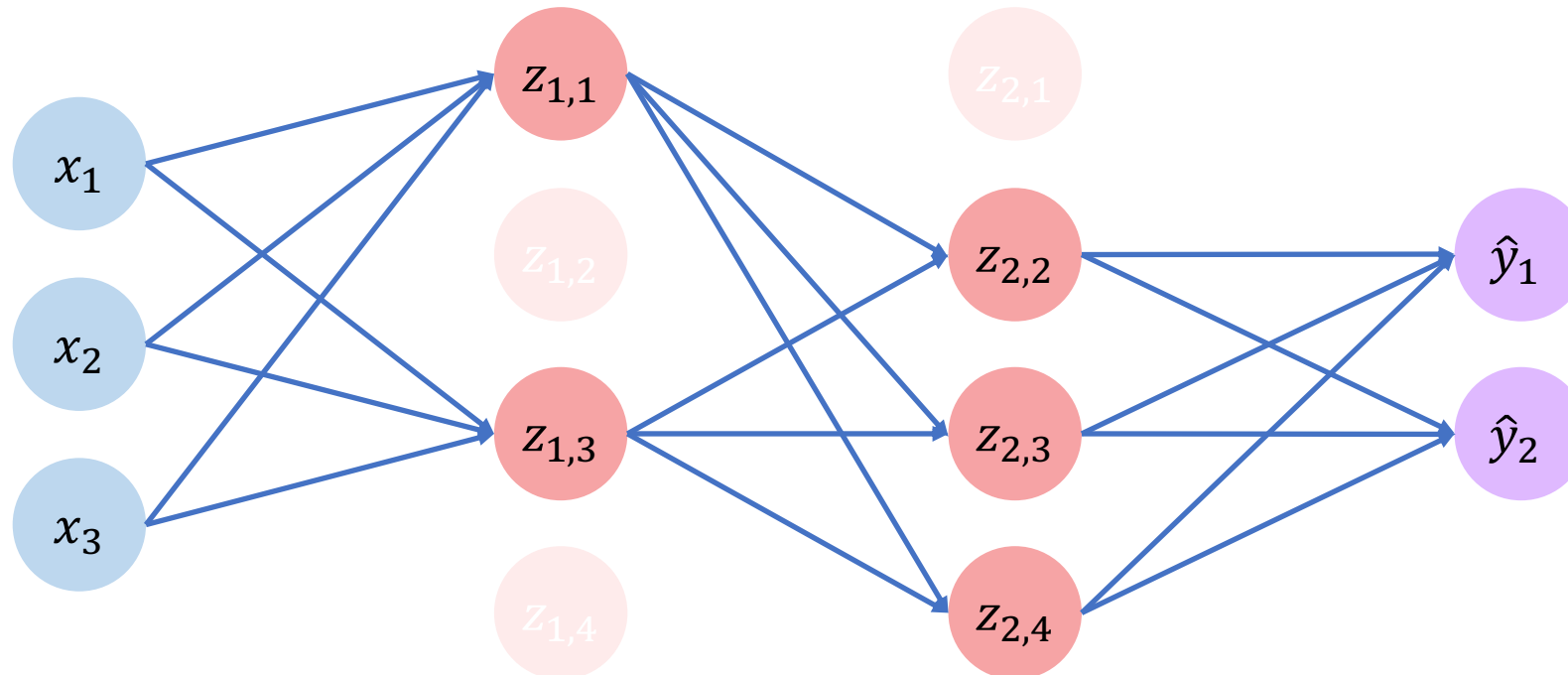


# Regularization I: Dropout

- 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



# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit





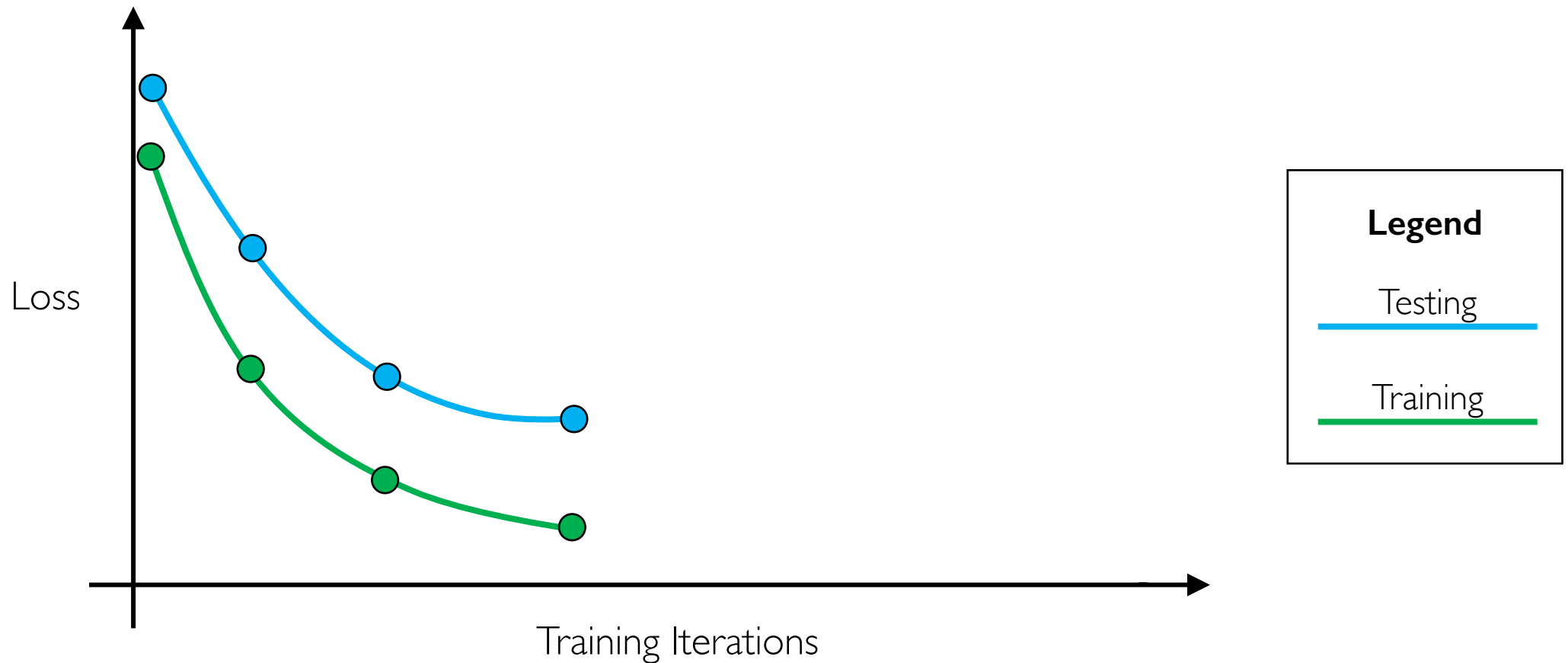
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



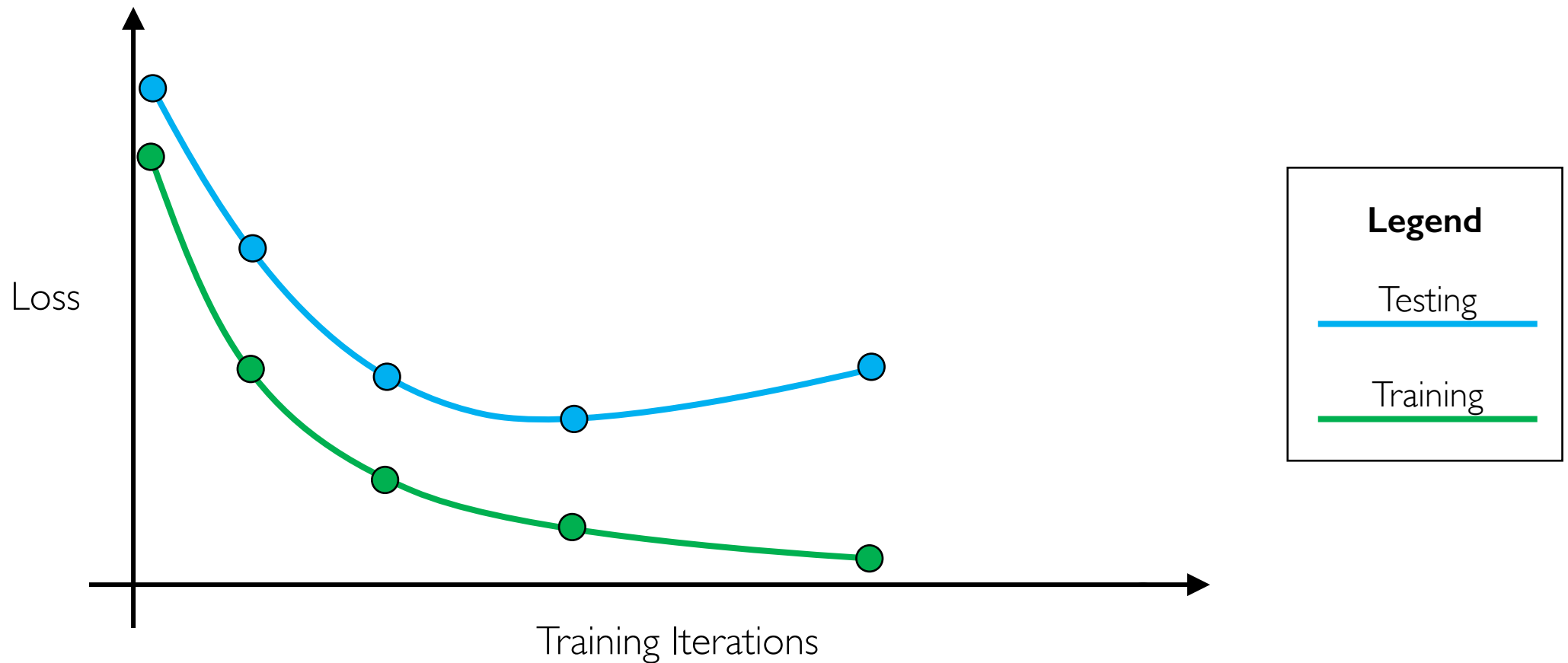
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



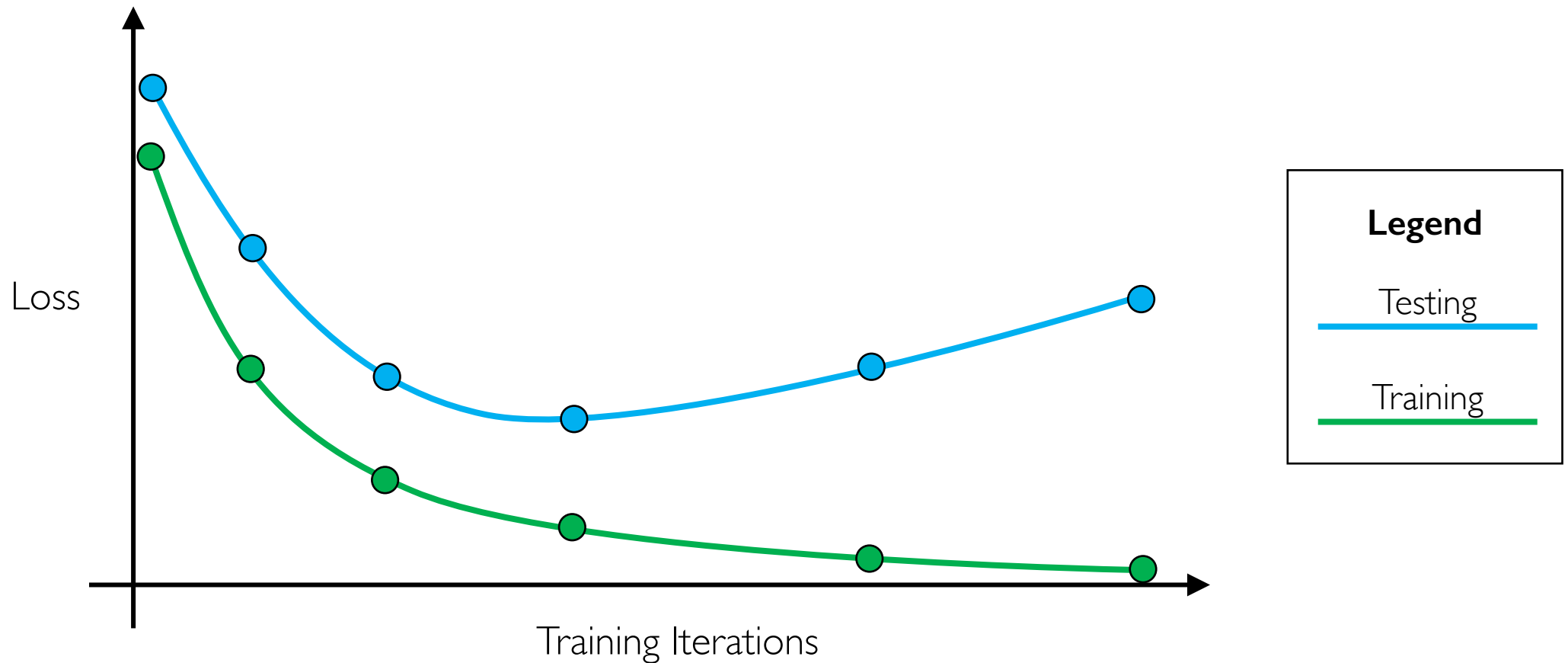
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



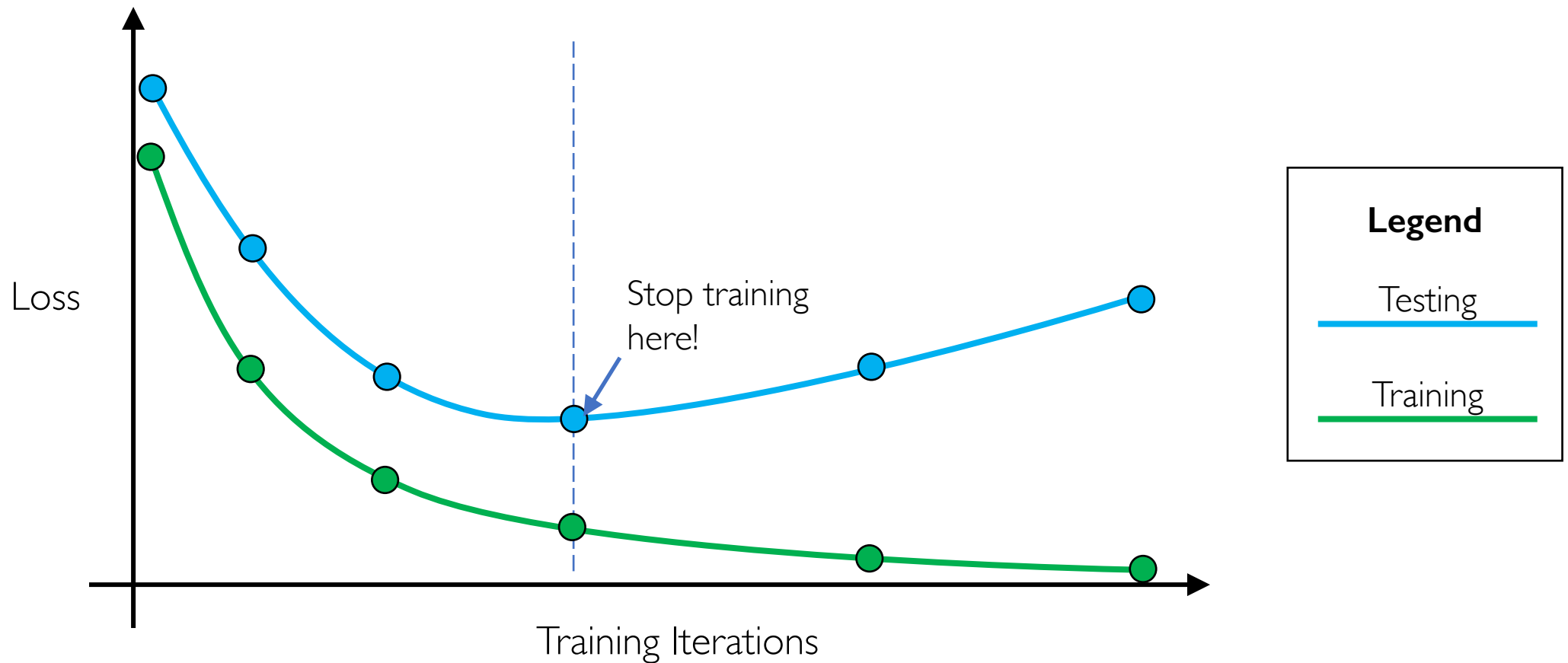
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



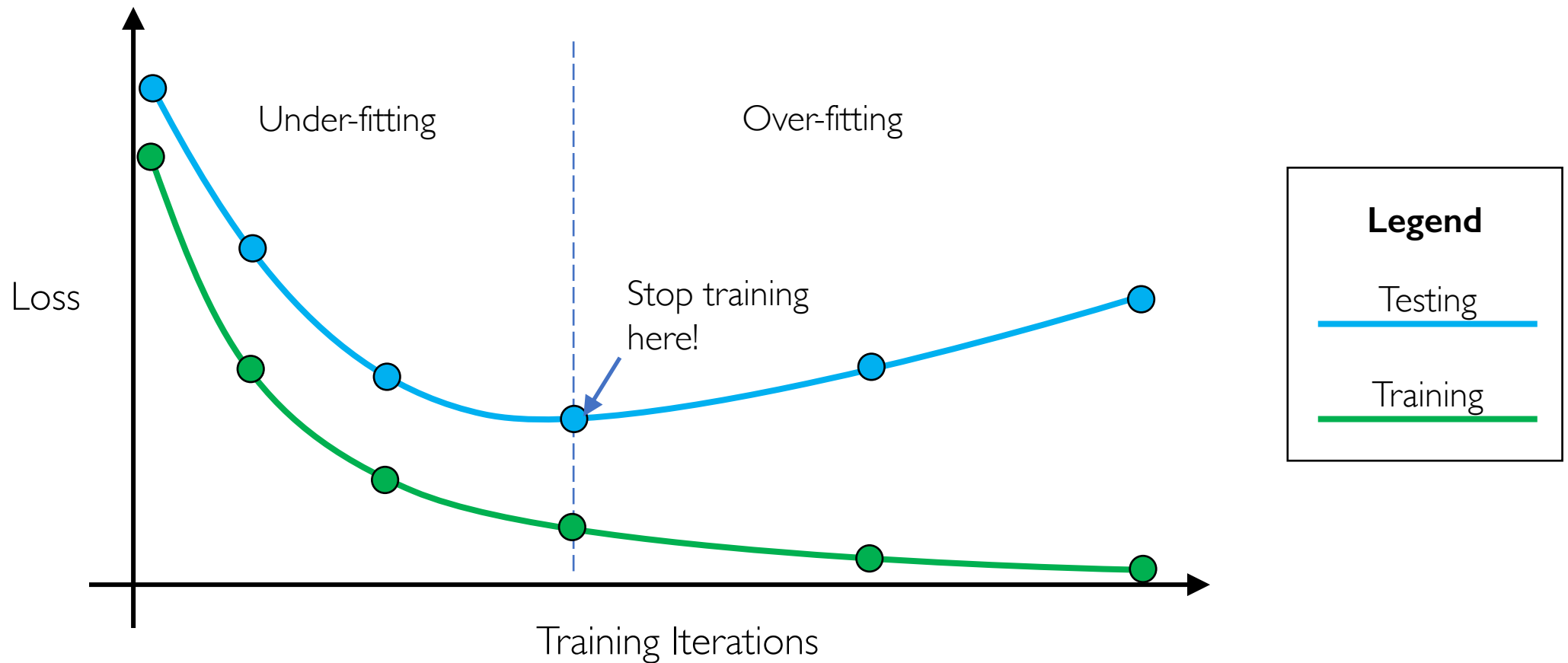
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



# Regularization 2: Early Stopping

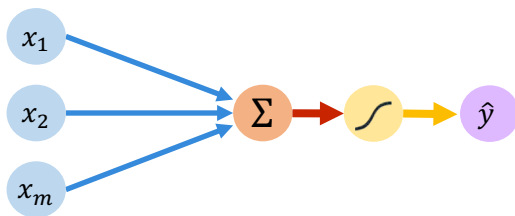
- Stop training before we have a chance to overfit



# Core Foundation Review

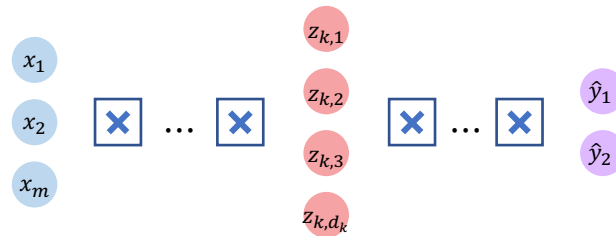
## The Perceptron

- Structural building blocks
- Nonlinear activation functions



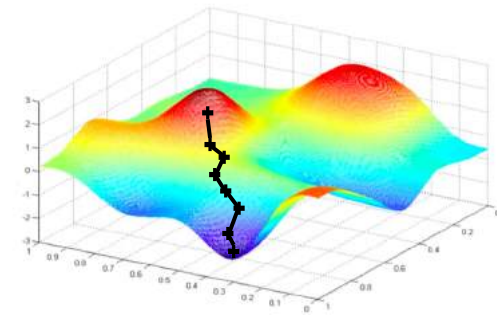
## Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



## Training in Practice

- Adaptive learning
- Batching
- Regularization



Questions?