

# CAP 781

# Machine Learning

Tanzeela Javid Kaloo (32638)

Assistant Professor

System And Architecture

Lovely Professional University

# UNIT – V

# Neural Network and Deep Learning

Tanzeela Javid Kaloo (32638)

Assistant Professor

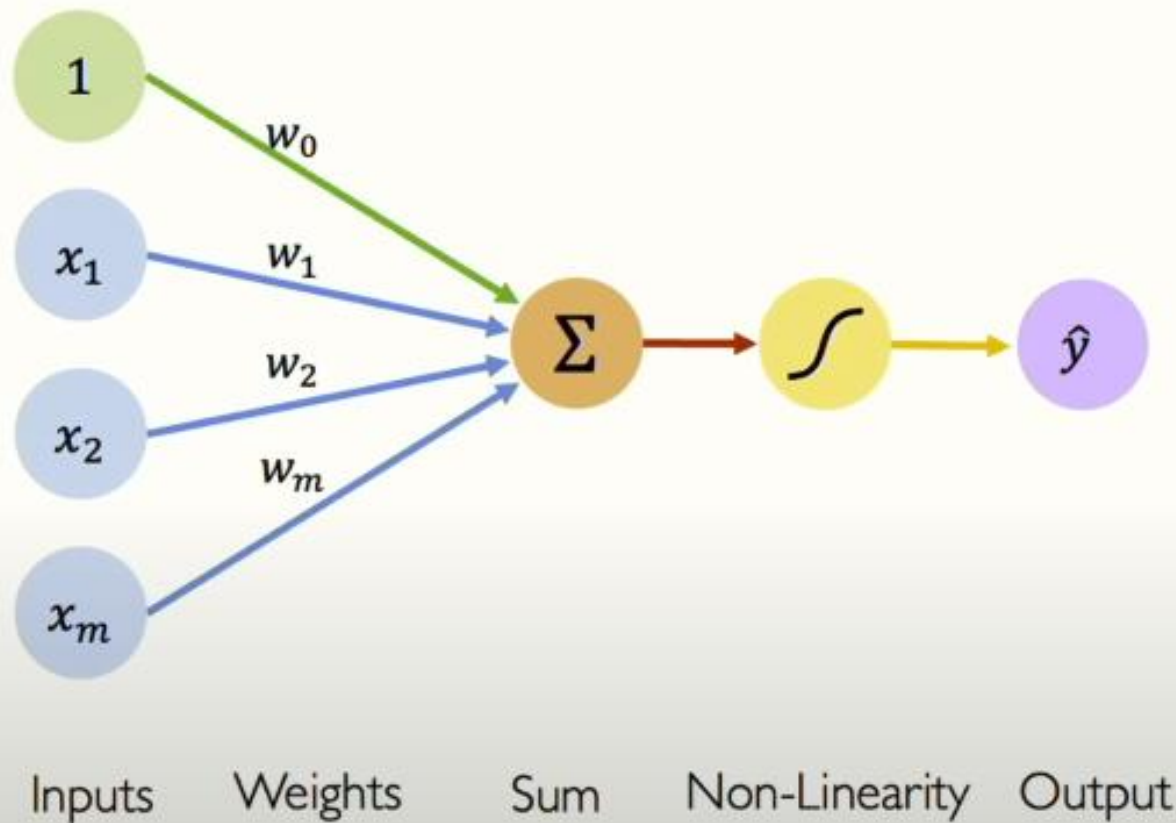
System And Architecture

Lovely Professional University

# Content

- Introduction to Neural Networks,
- Perceptron and Multilayer Perceptron (MLPs),
- Activation Functions: Sigmoid, ReLU,
- Gradient Descent,
- Stochastic Gradient Descent and Backpropagation

# The Perceptron: Forward Propagation



Output

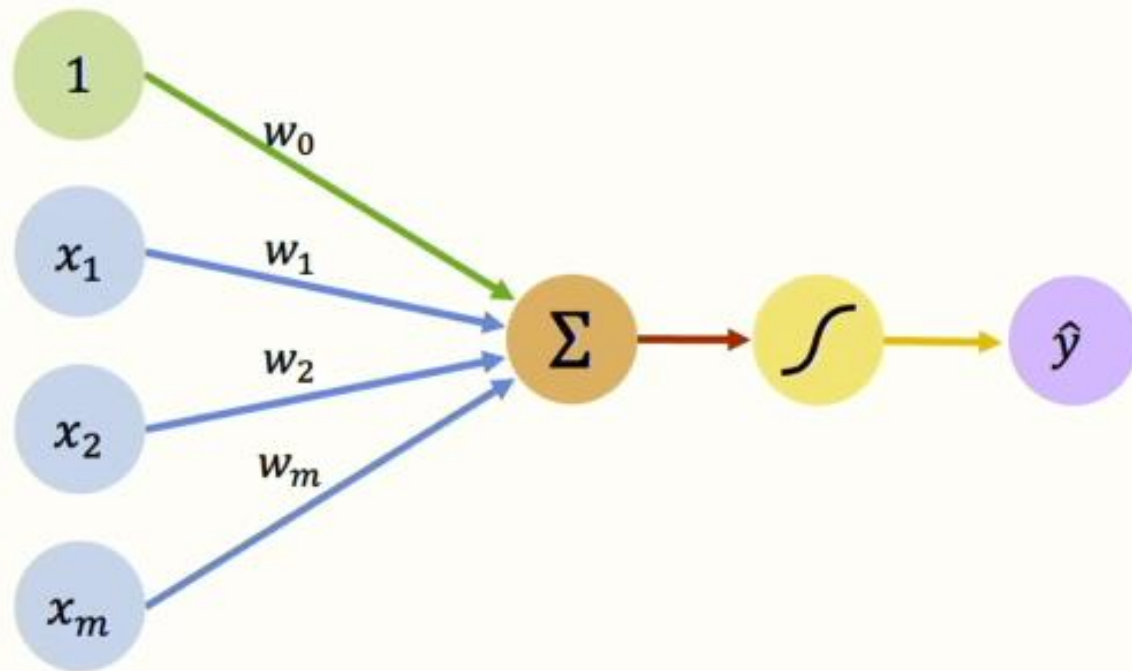
Linear combination of inputs

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

Non-linear activation function

Bias

# The Perceptron: Forward Propagation

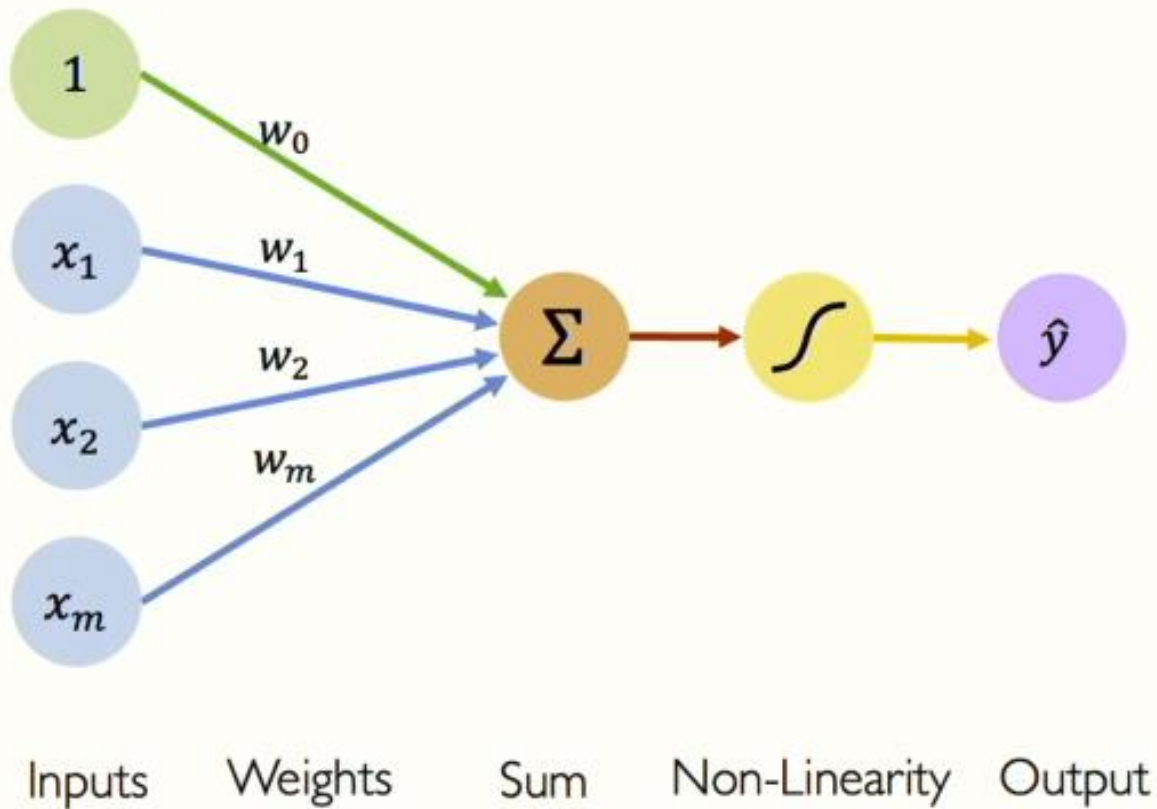


$$\hat{y} = g \left( w_0 + \sum_{i=1}^m x_i w_i \right)$$

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

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

# The Perceptron: Forward Propagation

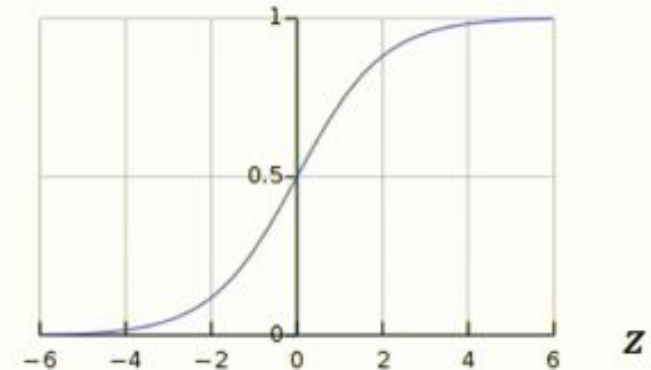


## Activation Functions

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

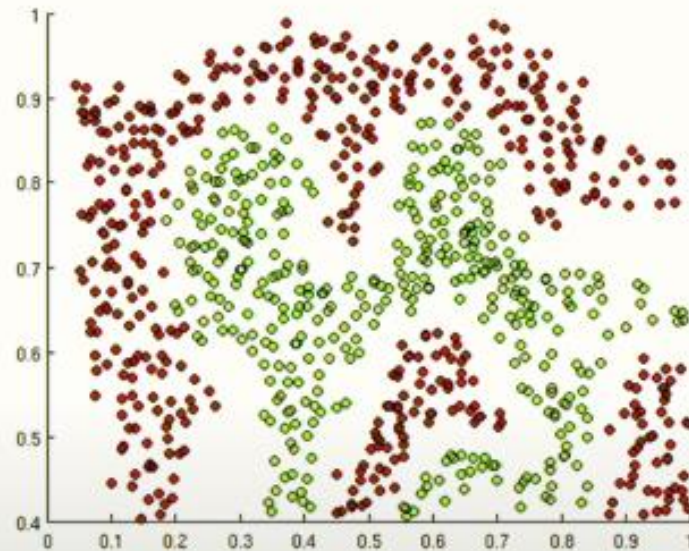
- Example: sigmoid function

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



# Importance of Activation Functions

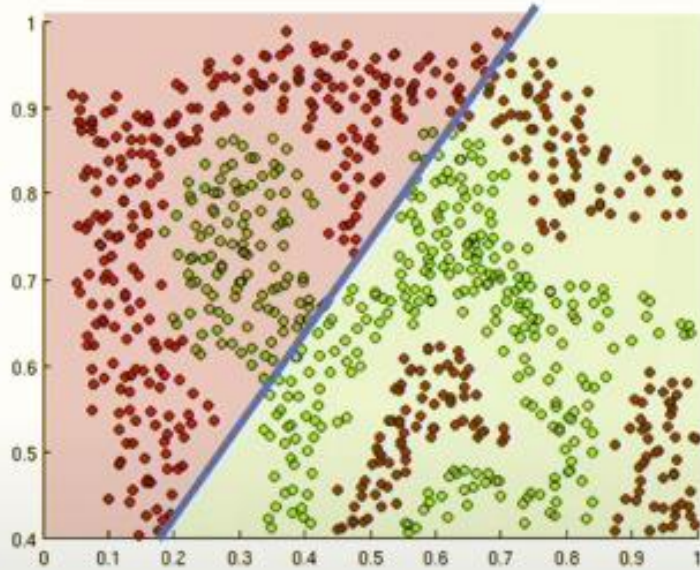
*The purpose of activation functions is to **introduce non-linearities** into the network*



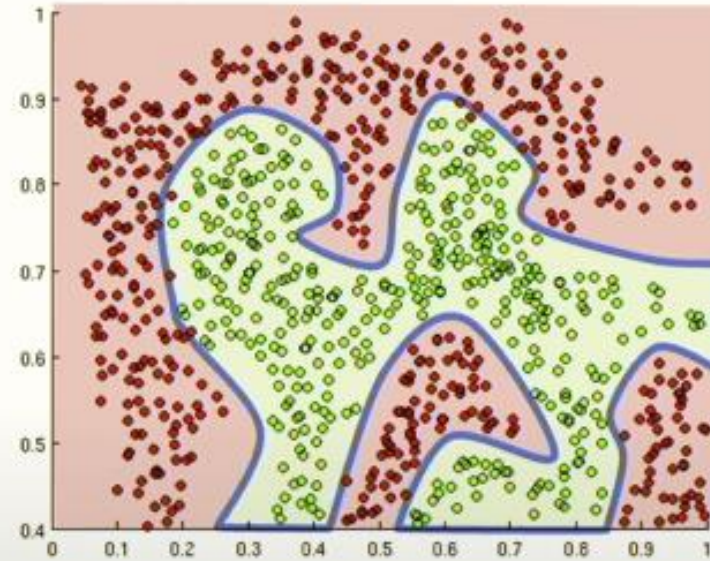
What if we wanted to build a neural network to distinguish green vs red points?

# Importance of Activation Functions

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

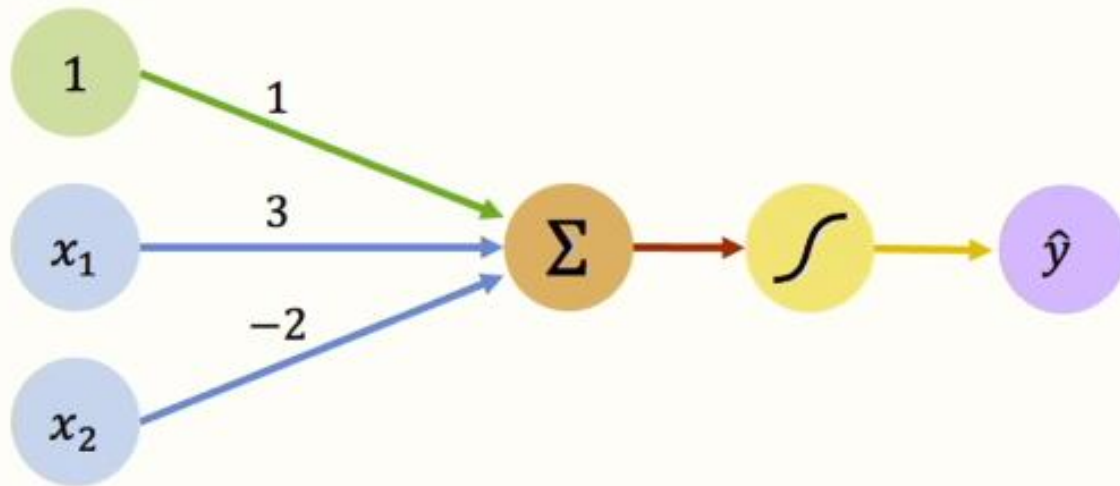


Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

# The Perceptron: Example

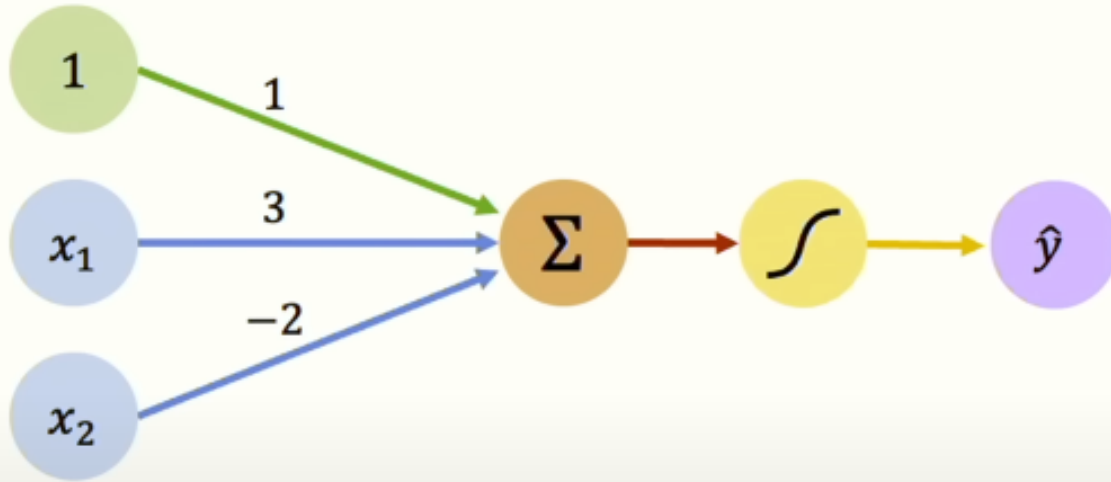


We have:  $w_0 = 1$  and  $\mathbf{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

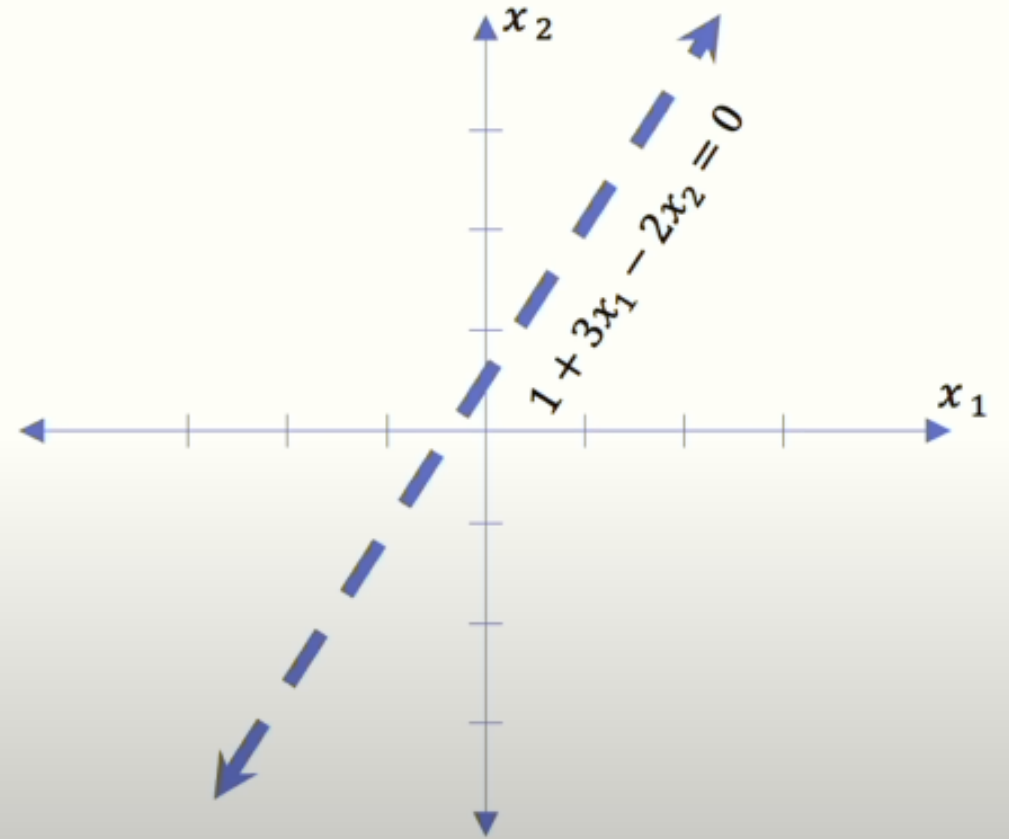
$$\begin{aligned}\hat{y} &= g(w_0 + \mathbf{X}^T \mathbf{W}) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g(1 + 3x_1 - 2x_2)\end{aligned}$$

This is just a line in 2D!

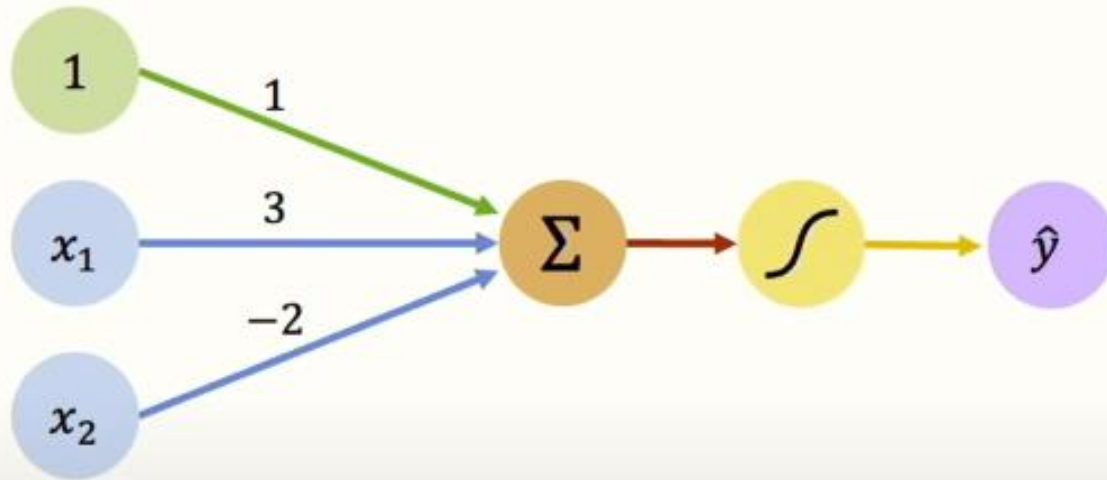
# The Perceptron: Example



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



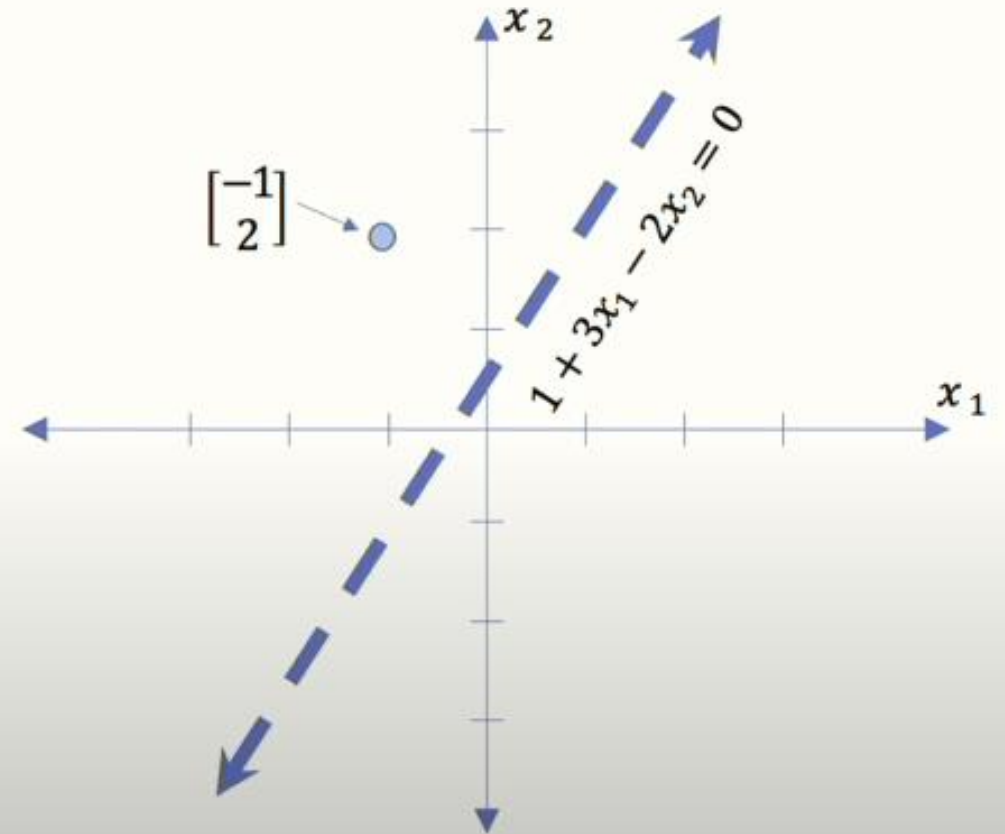
# The Perceptron: Example



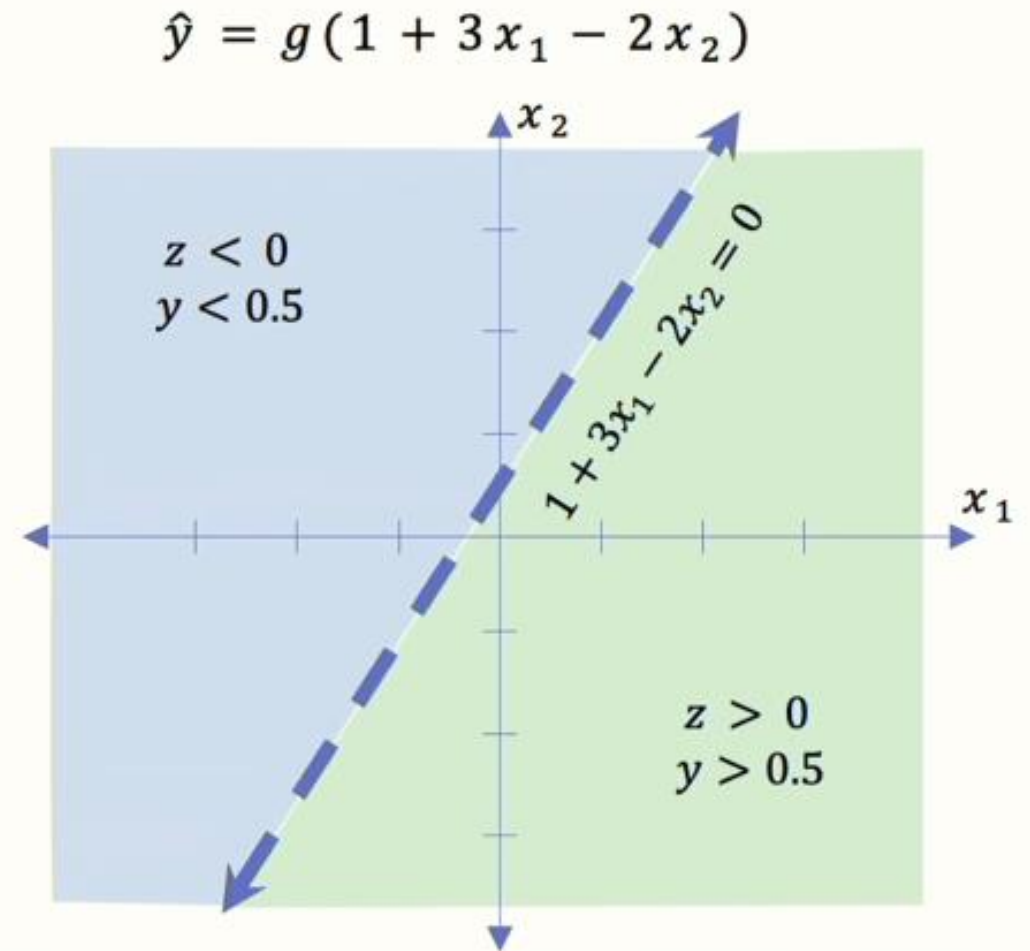
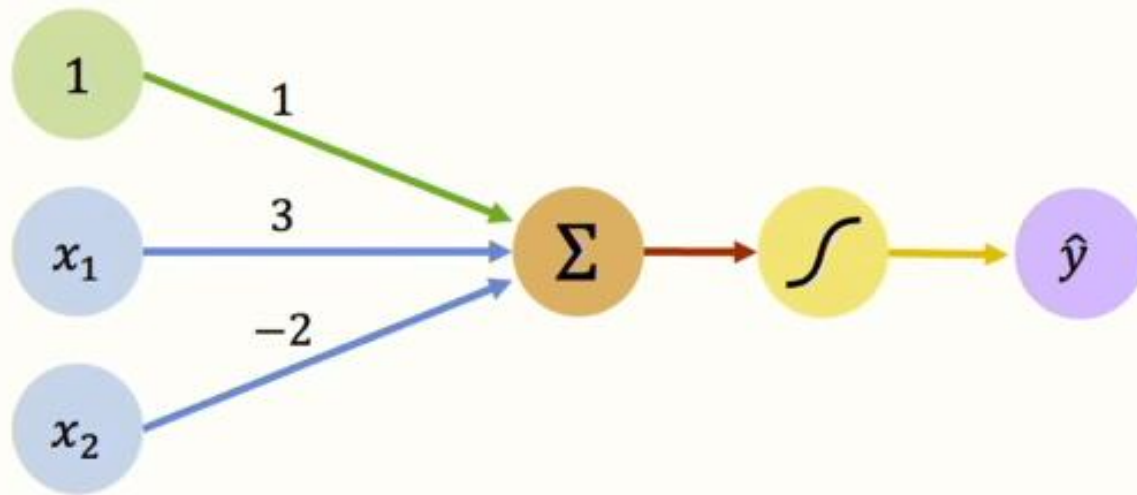
Assume we have input:  $\mathbf{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

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

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



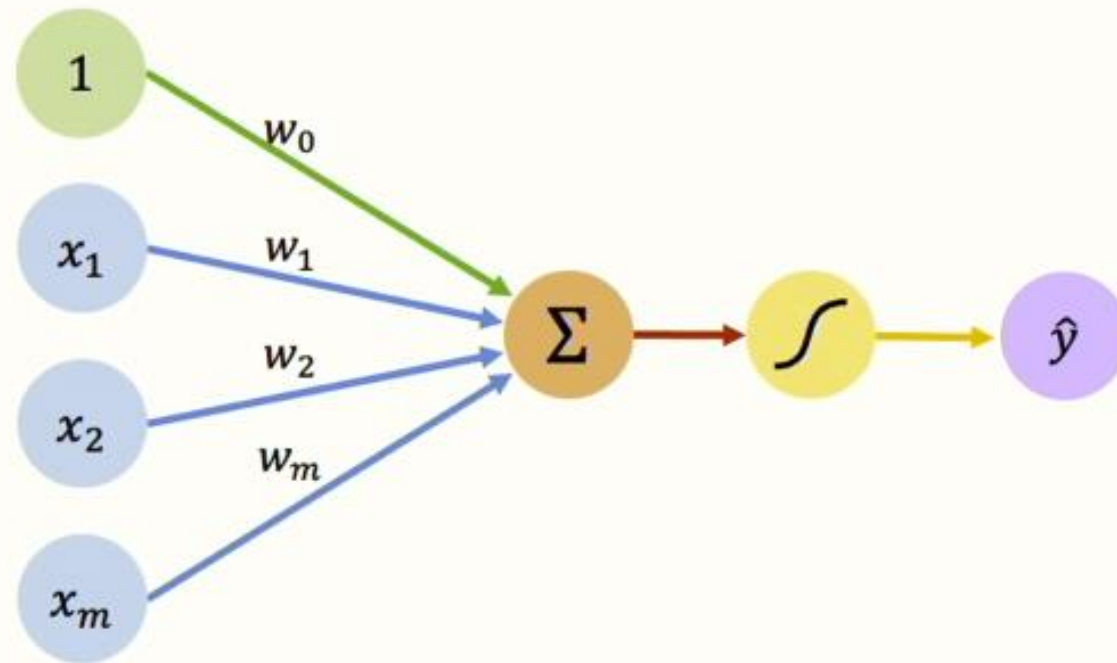
# The Perceptron: Example



# Building Neural Networks with Perceptrons

# The Perceptron: Simplified

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



Inputs

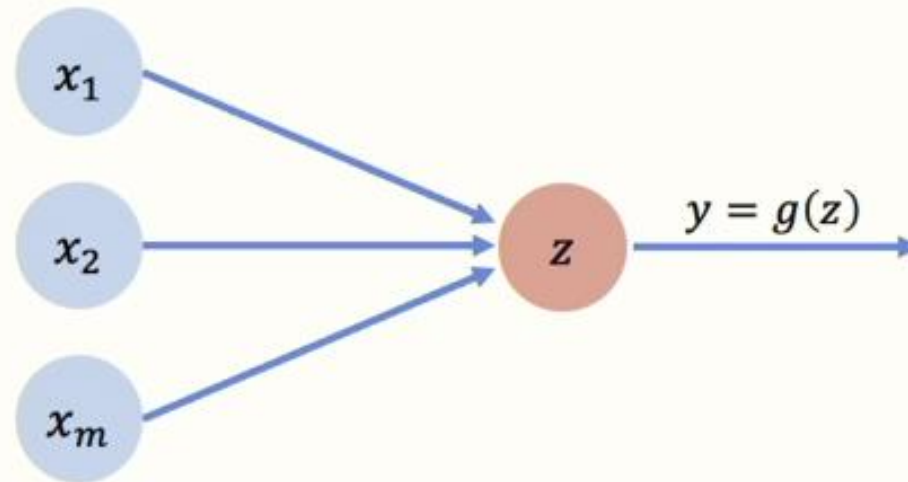
Weights

Sum

Non-Linearity

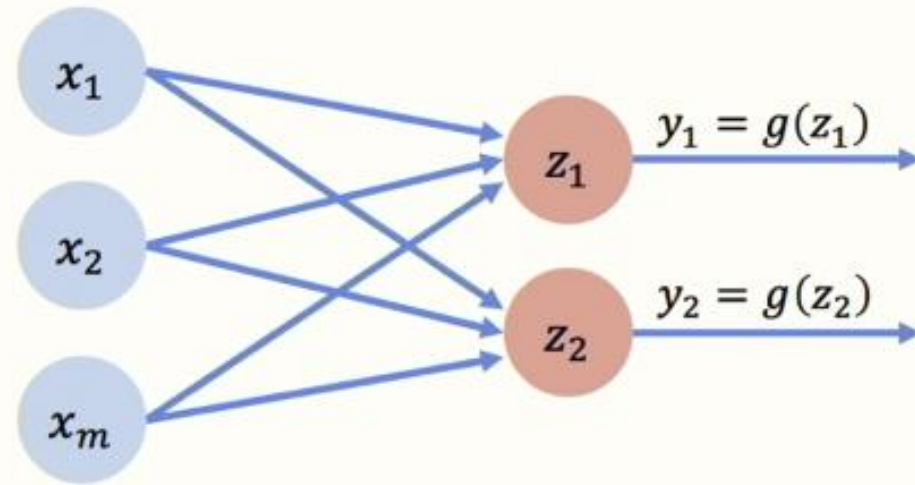
Output

# The Perceptron: Simplified



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

# Multi Output Perceptron



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

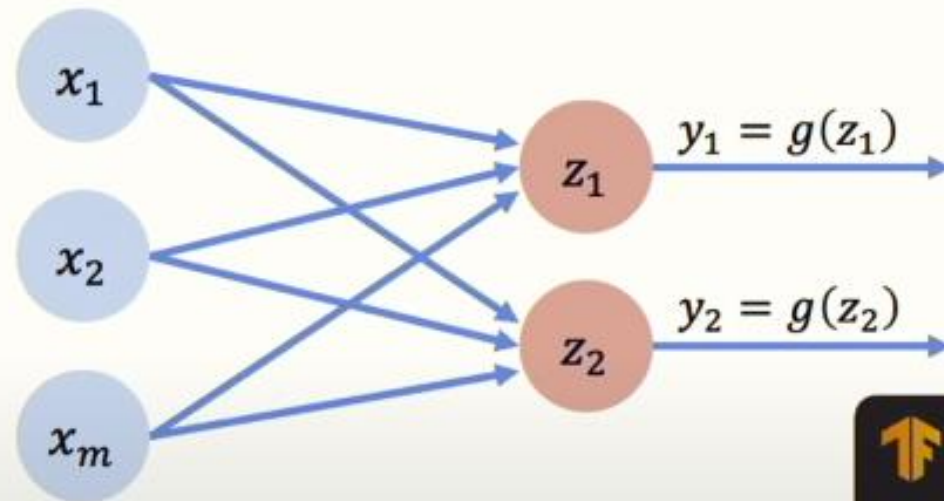
# Dense layer from scratch




```
class MyDenseLayer(tf.keras.layers.Layer):  
    def __init__(self, input_dim, output_dim):  
        super(MyDenseLayer, self).__init__()  
  
        # Initialize weights and bias  
        self.W = self.add_weight([input_dim, output_dim])  
        self.b = self.add_weight([1, output_dim])  
  
    def call(self, inputs):  
        # Forward propagate the inputs  
        z = tf.matmul(inputs, self.W) + self.b  
  
        # Feed through a non-linear activation  
        output = tf.math.sigmoid(z)  
  
        return output
```

# Multi Output Perceptron

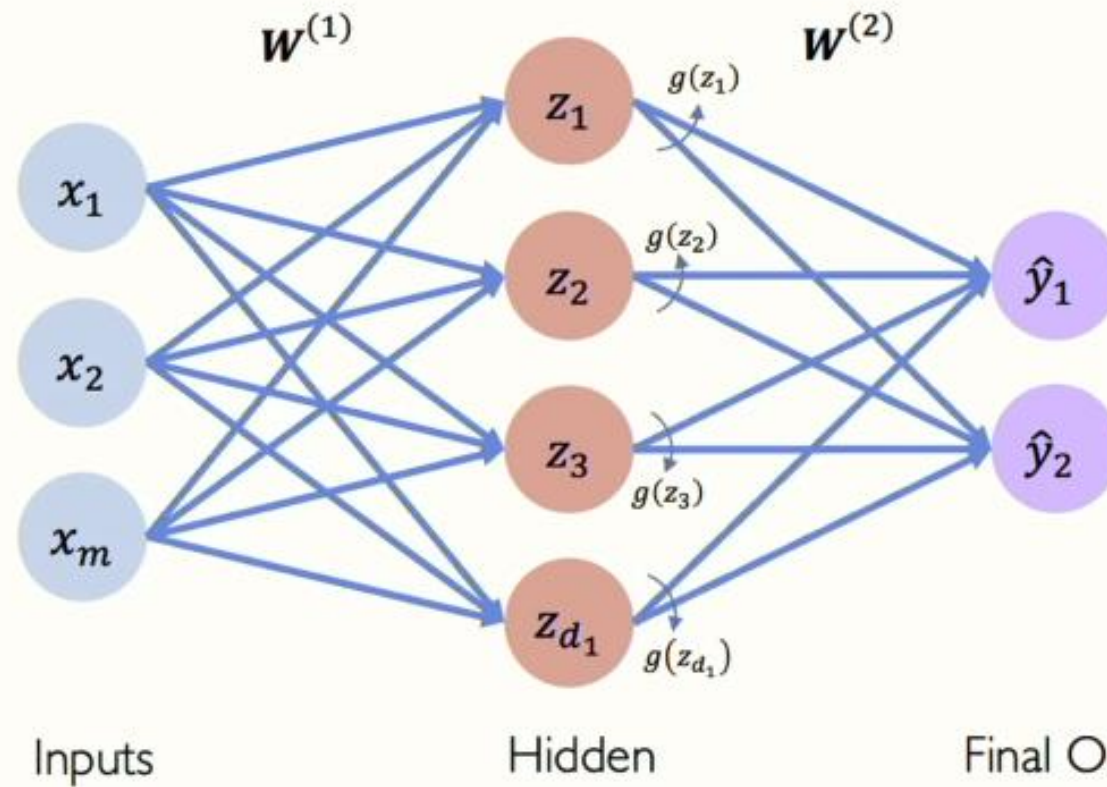
Because all inputs are densely connected to all outputs, these layers are called **Dense** layers



```
 import tensorflow as tf  
  
layer = tf.keras.layers.Dense(  
    units=2)
```

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

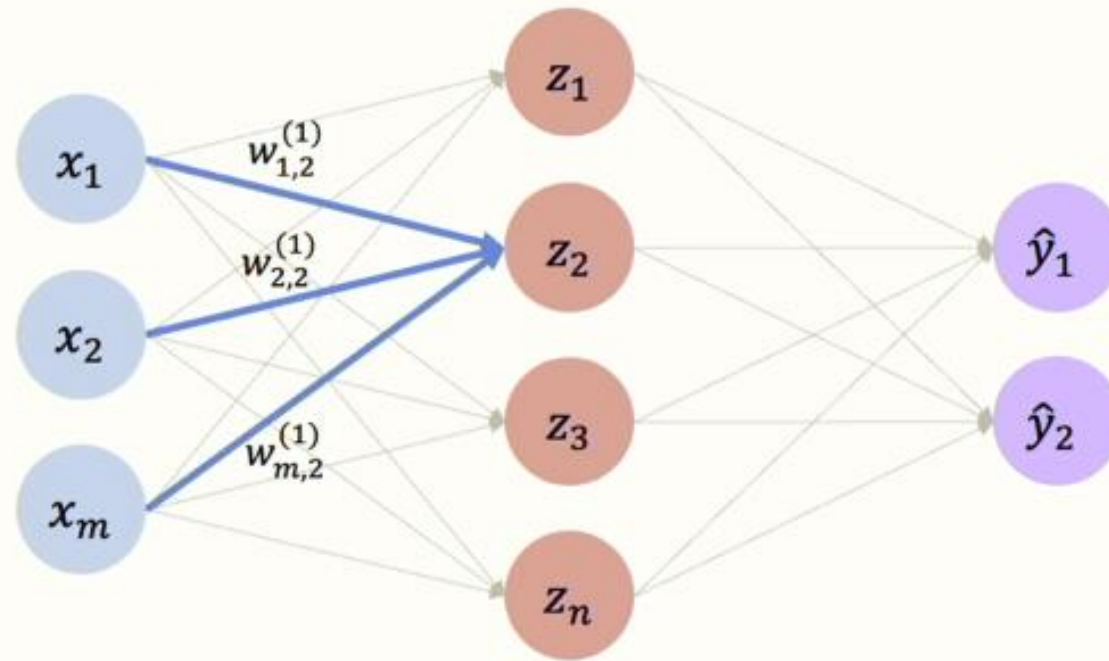
# Single Layer Neural Network



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

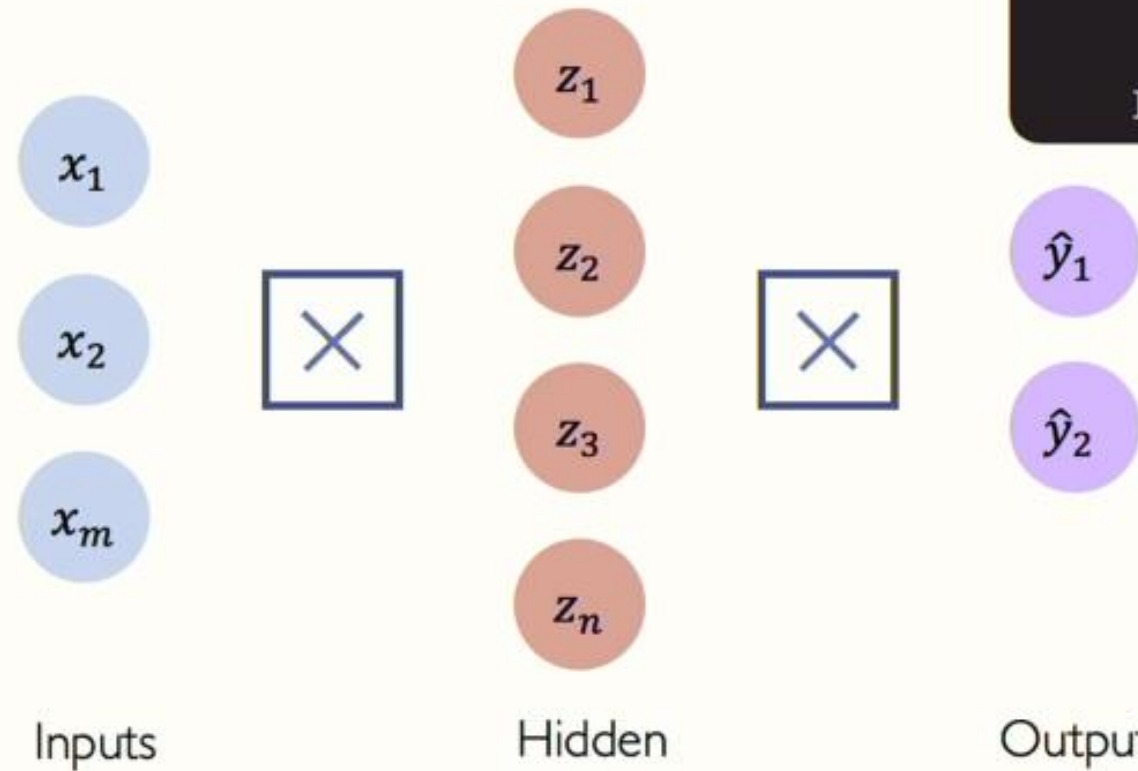
$$\hat{y}_i = g \left( w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) w_{j,i}^{(2)} \right)$$


# Single Layer Neural Network



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

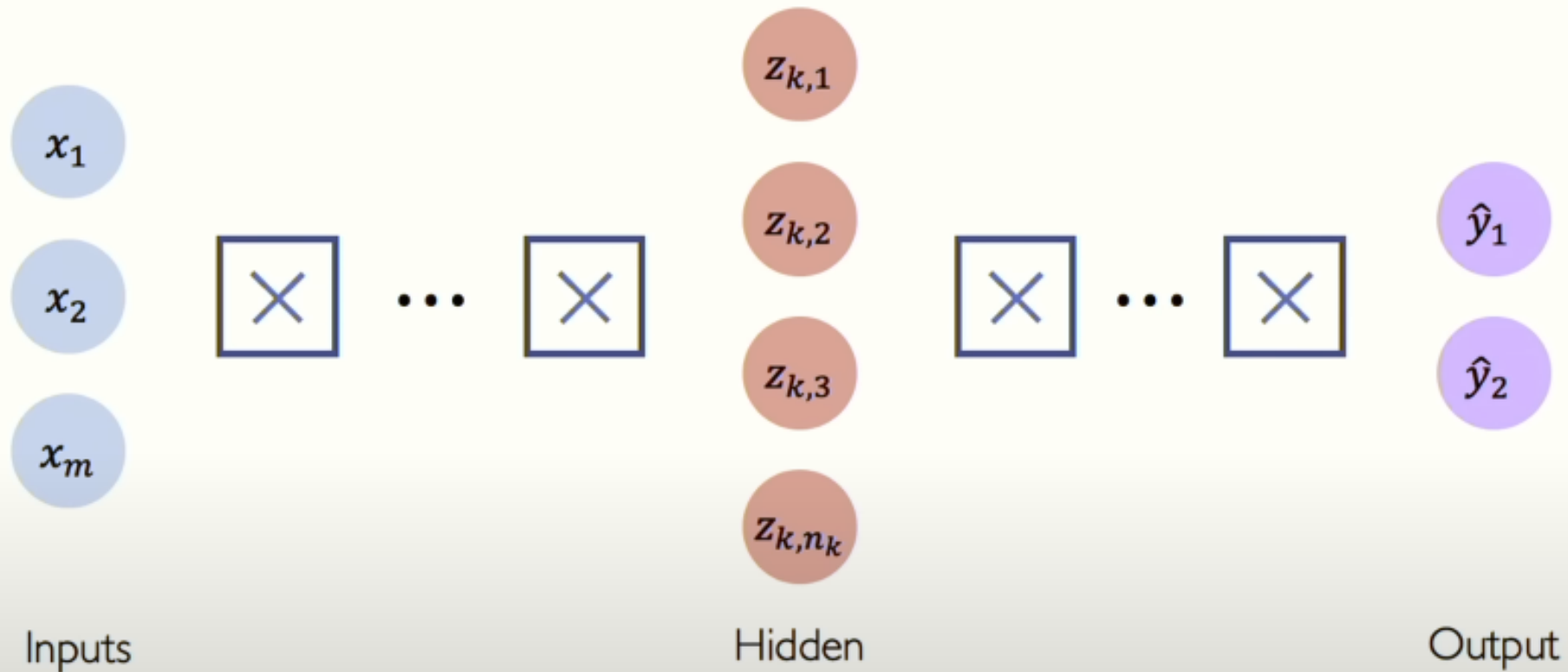
# Multi Output Perceptron



```
 import tensorflow as tf

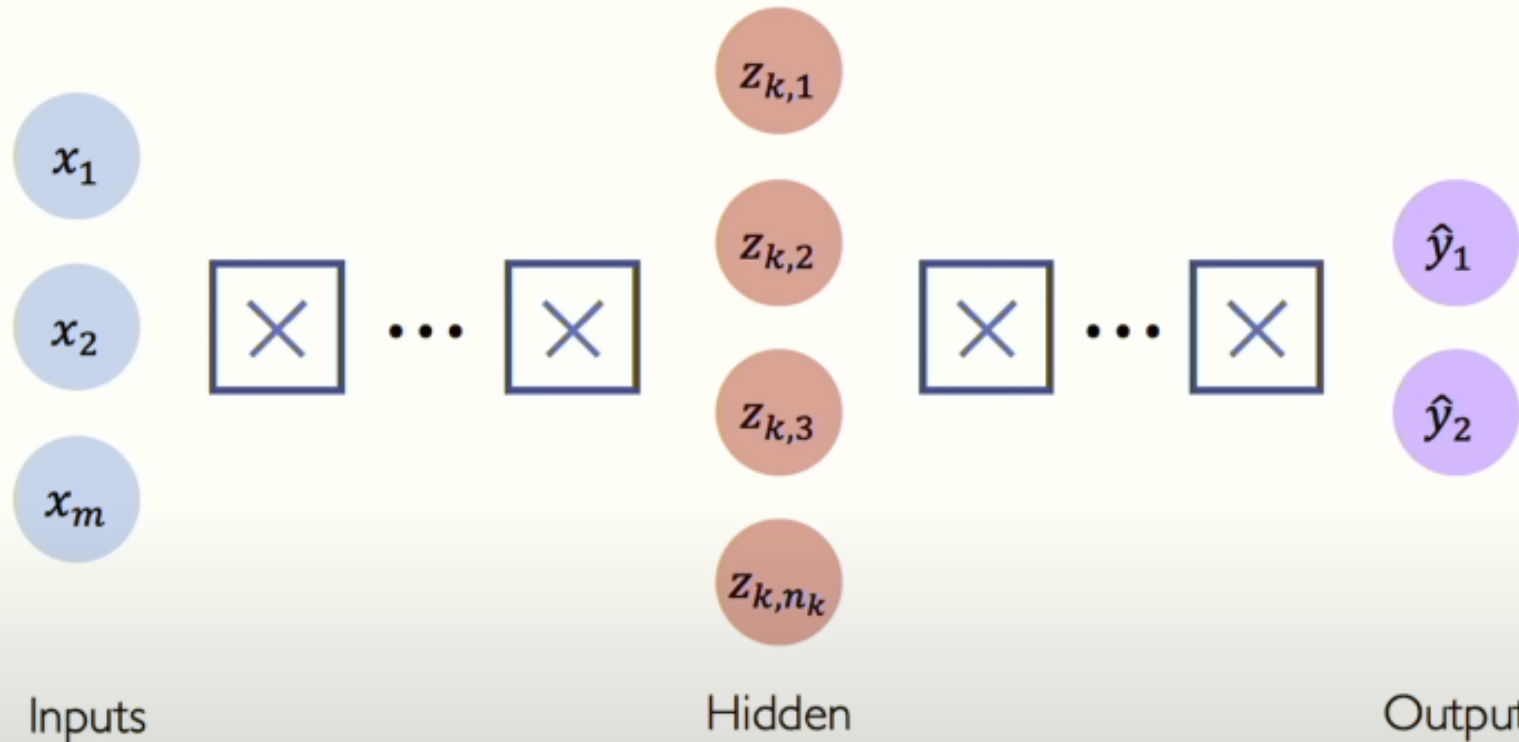
model = tf.keras.Sequential([
    tf.keras.layers.Dense(n),
    tf.keras.layers.Dense(2)
])
```

# Deep Neural Network



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

# Deep Neural Network



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



```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(n_1),
    tf.keras.layers.Dense(n_2),
    ...,
    tf.keras.layers.Dense(2)
])
```

# Applying Neural Networks

# Example Problem

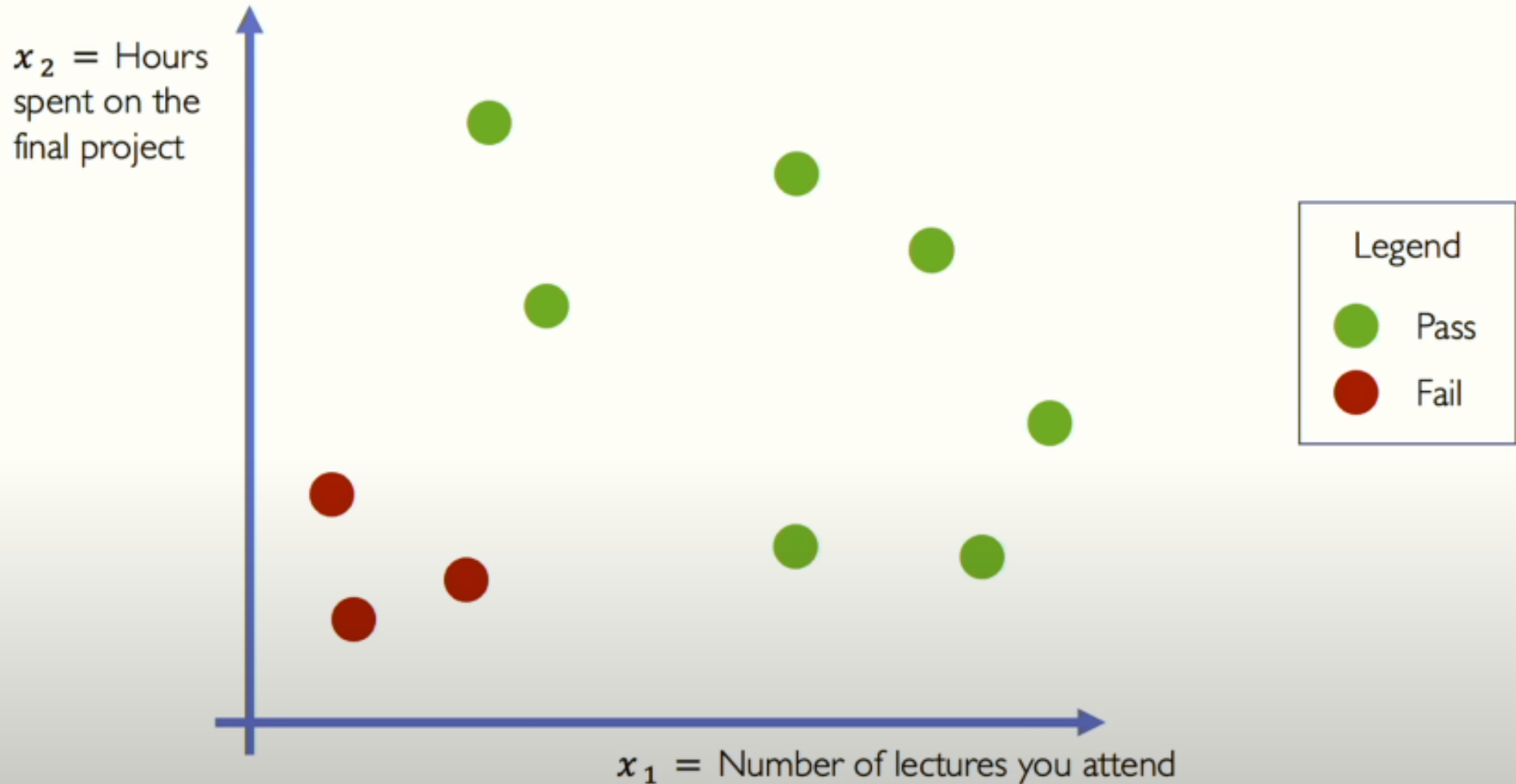
Will I pass this class?

Let's start with a simple two feature model

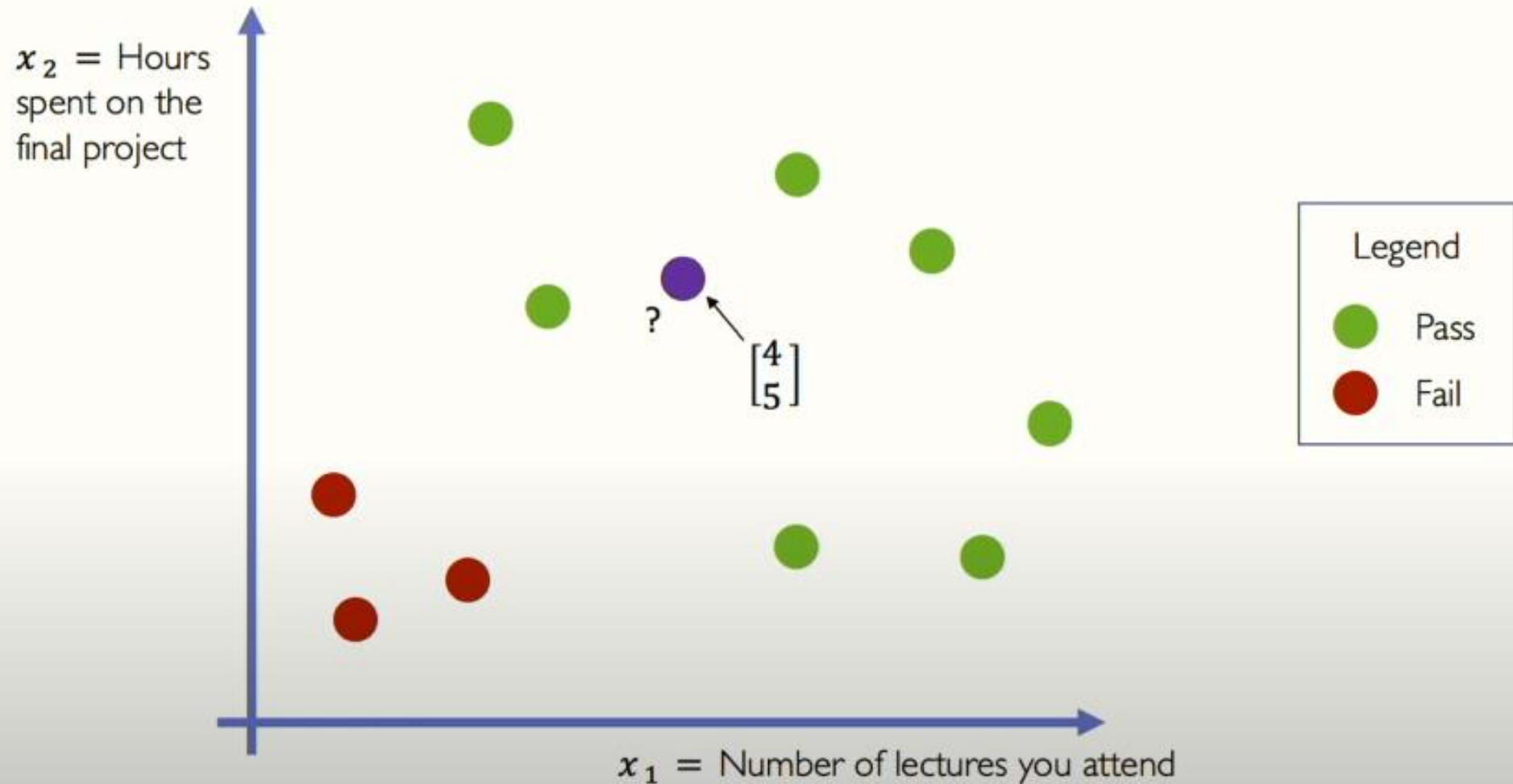
$x_1$  = Number of lectures you attend

$x_2$  = Hours spent on the final project

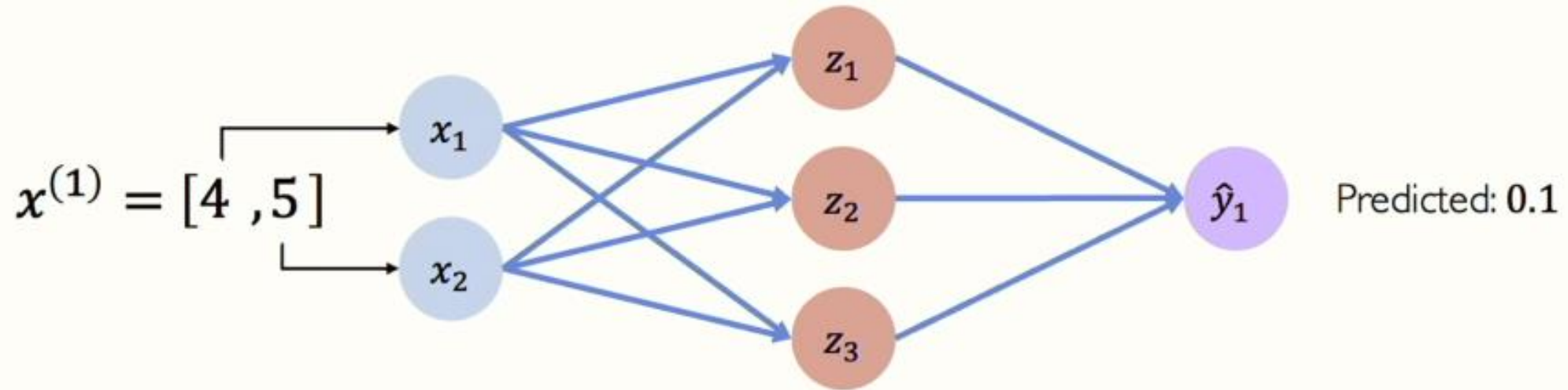
# Example Problem: Will I pass this class?



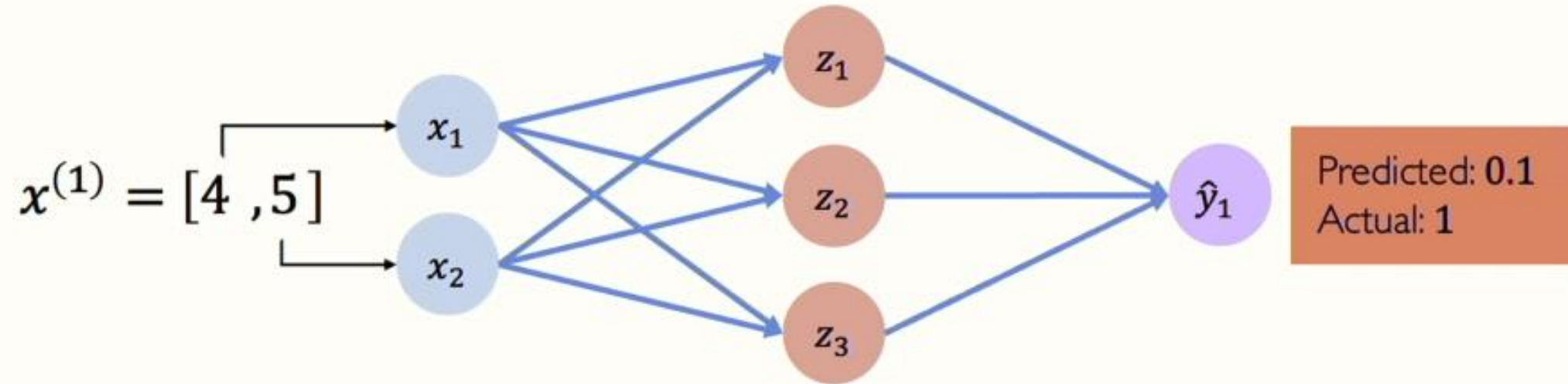
# Example Problem: Will I pass this class?



# Example Problem: Will I pass this class?

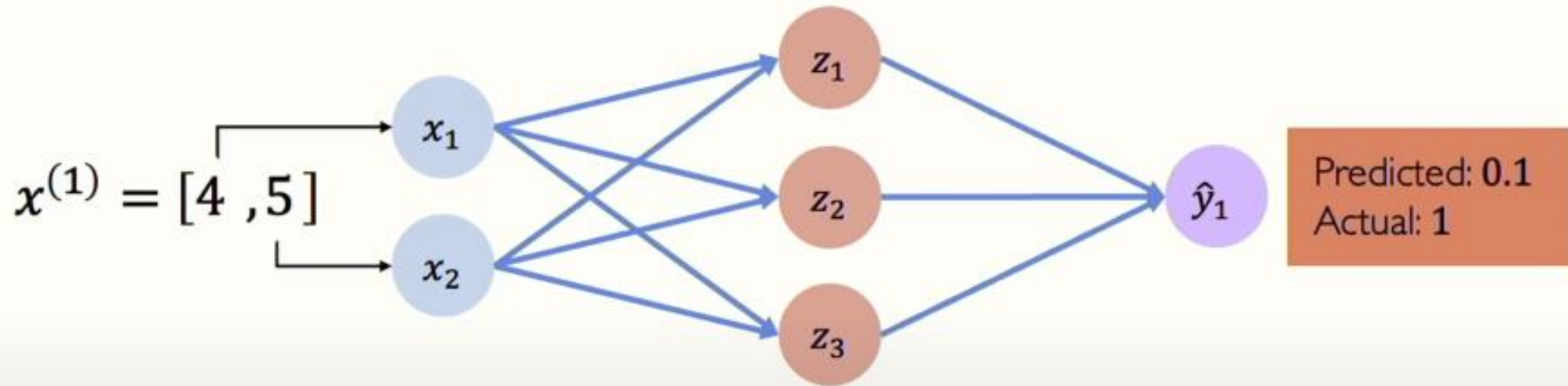


# Example Problem: Will I pass this class?



# Quantifying Loss

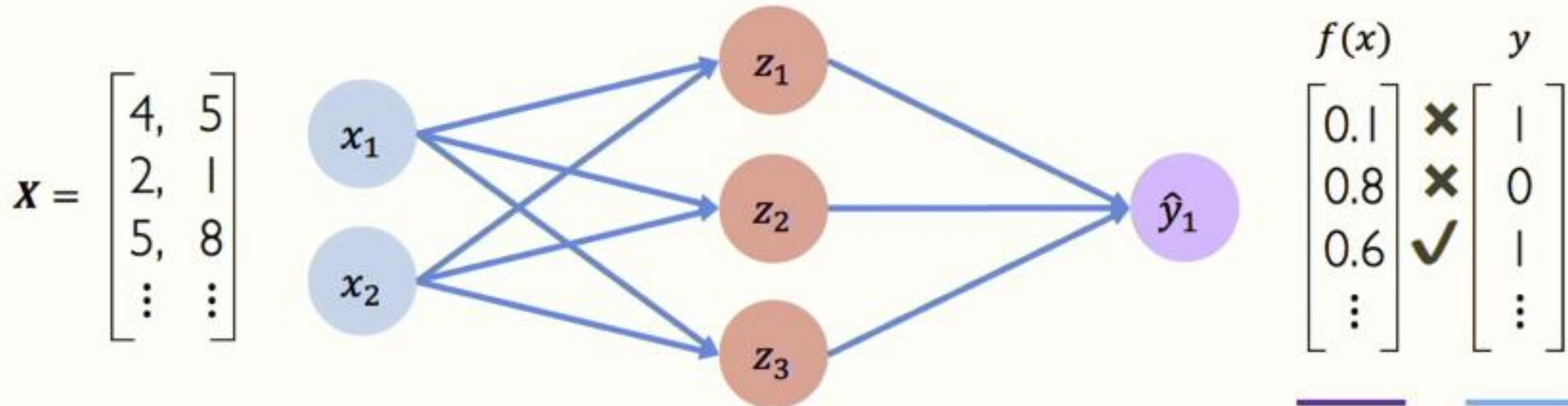
The **loss** of our network measures the cost incurred from incorrect predictions



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

# Empirical Loss

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



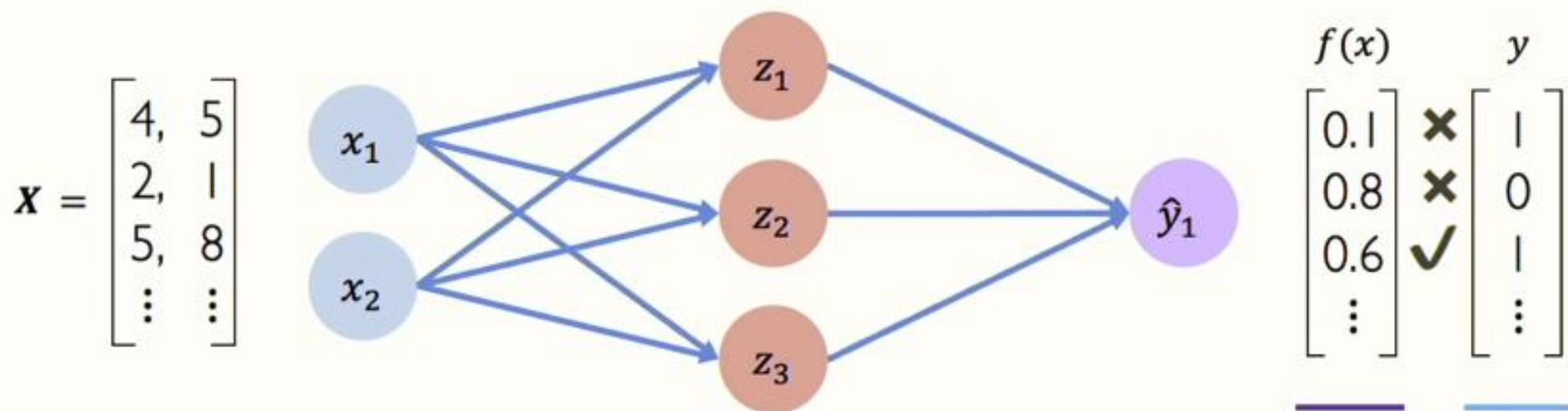
Also known as:

- 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



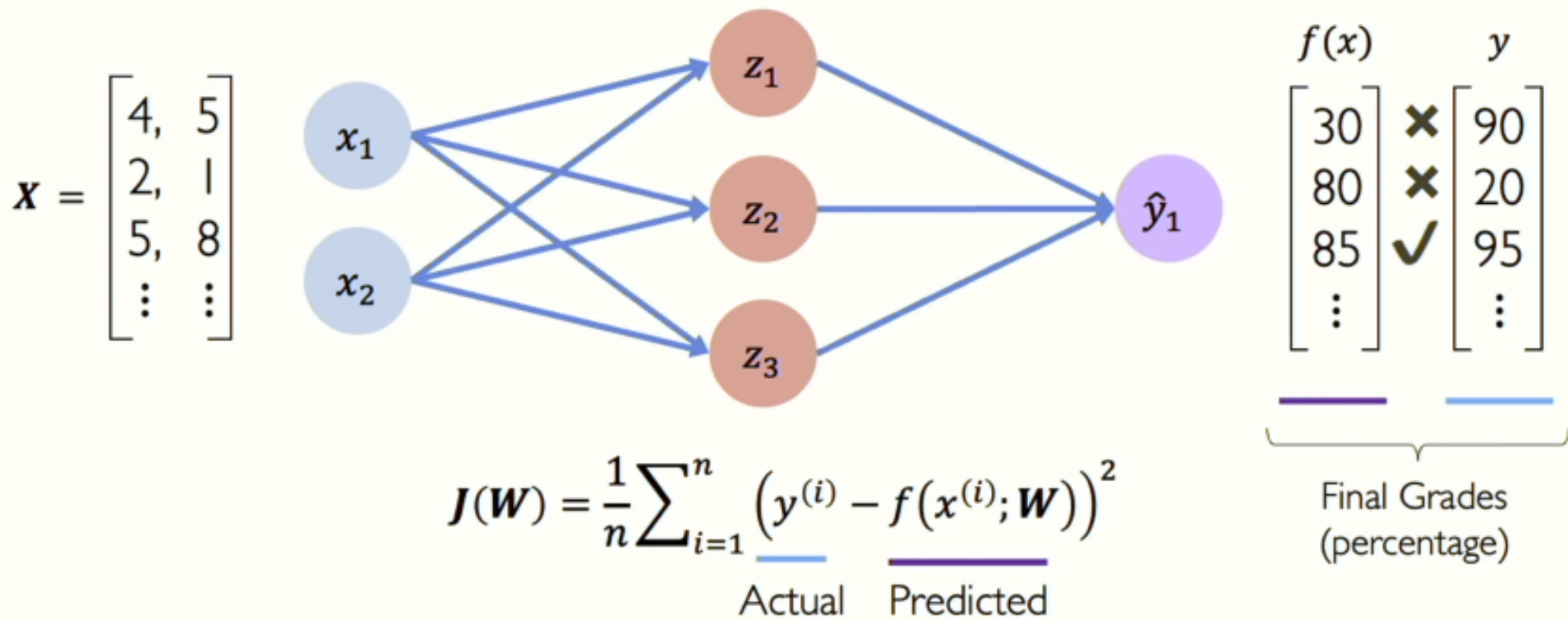
$$J(W) = -\frac{1}{n} \sum_{i=1}^n \underbrace{y^{(i)}}_{\text{Actual}} \log \left( \underbrace{f(x^{(i)}; W)}_{\text{Predicted}} \right) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log \left( 1 - \underbrace{f(x^{(i)}; W)}_{\text{Predicted}} \right)$$



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

# 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(y, predicted)) )  
loss = tf.keras.losses.MSE( y, predicted )
```

# 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*

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

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

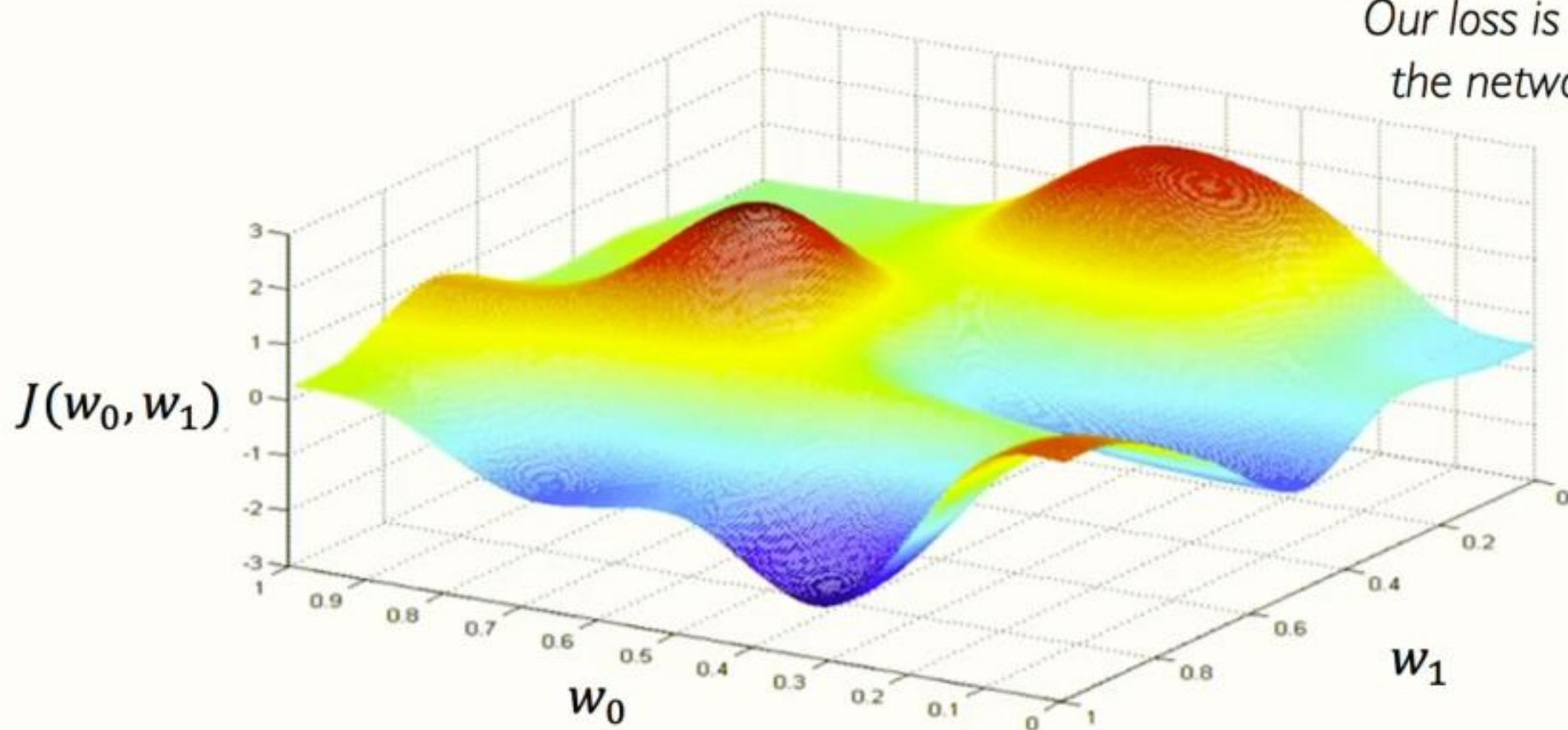
Remember:

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

# Loss Optimization

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

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





# 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

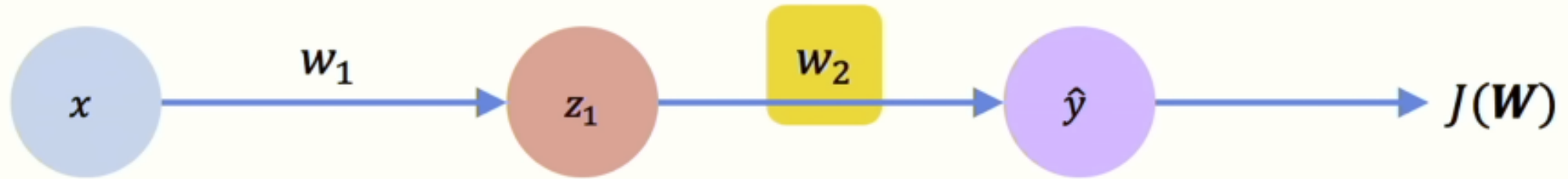
```
import tensorflow as tf

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

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

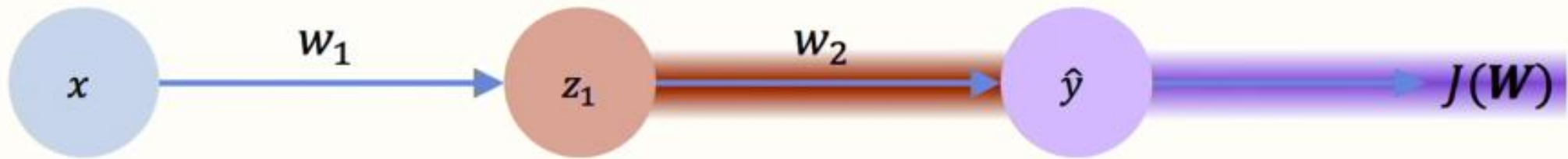
    weights = weights - lr * gradient
```

# Computing Gradients: Backpropagation



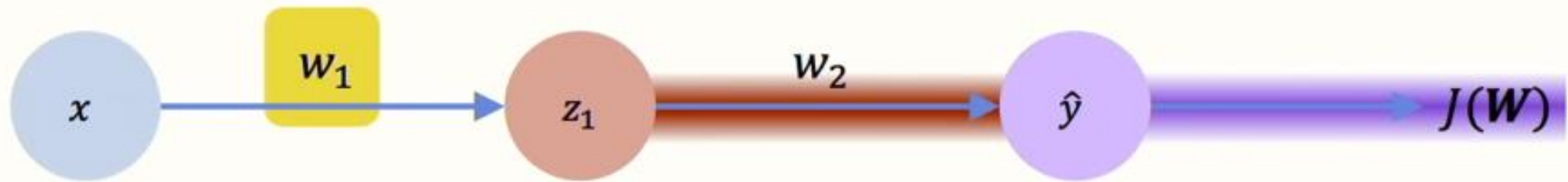
*How does a small change in one weight (ex.  $w_2$ ) affect the final loss  $J(\mathbf{W})$ ?*

# Computing Gradients: Backpropagation



$$\frac{\partial J(\mathbf{W})}{\partial w_2} = \underbrace{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial w_2}}_{\text{red}}$$

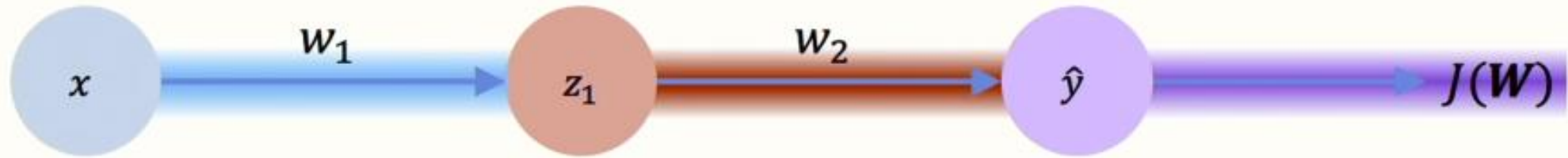
# Computing Gradients: Backpropagation



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

Apply chain rule!      Apply chain rule!

# Computing Gradients: Backpropagation

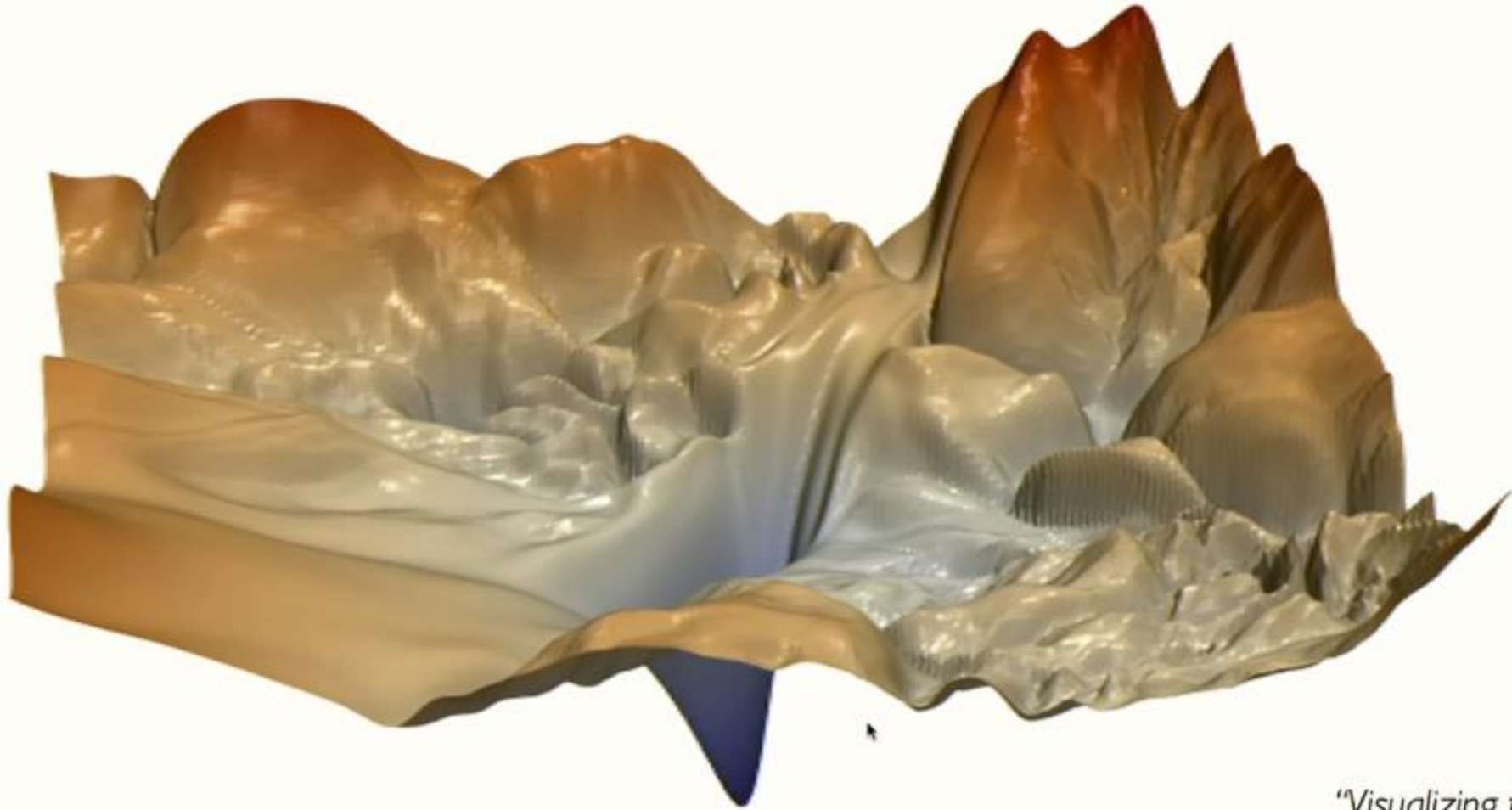


$$\frac{\partial J(W)}{\partial w_1} = \underbrace{\frac{\partial J(W)}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue}}$$

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:**

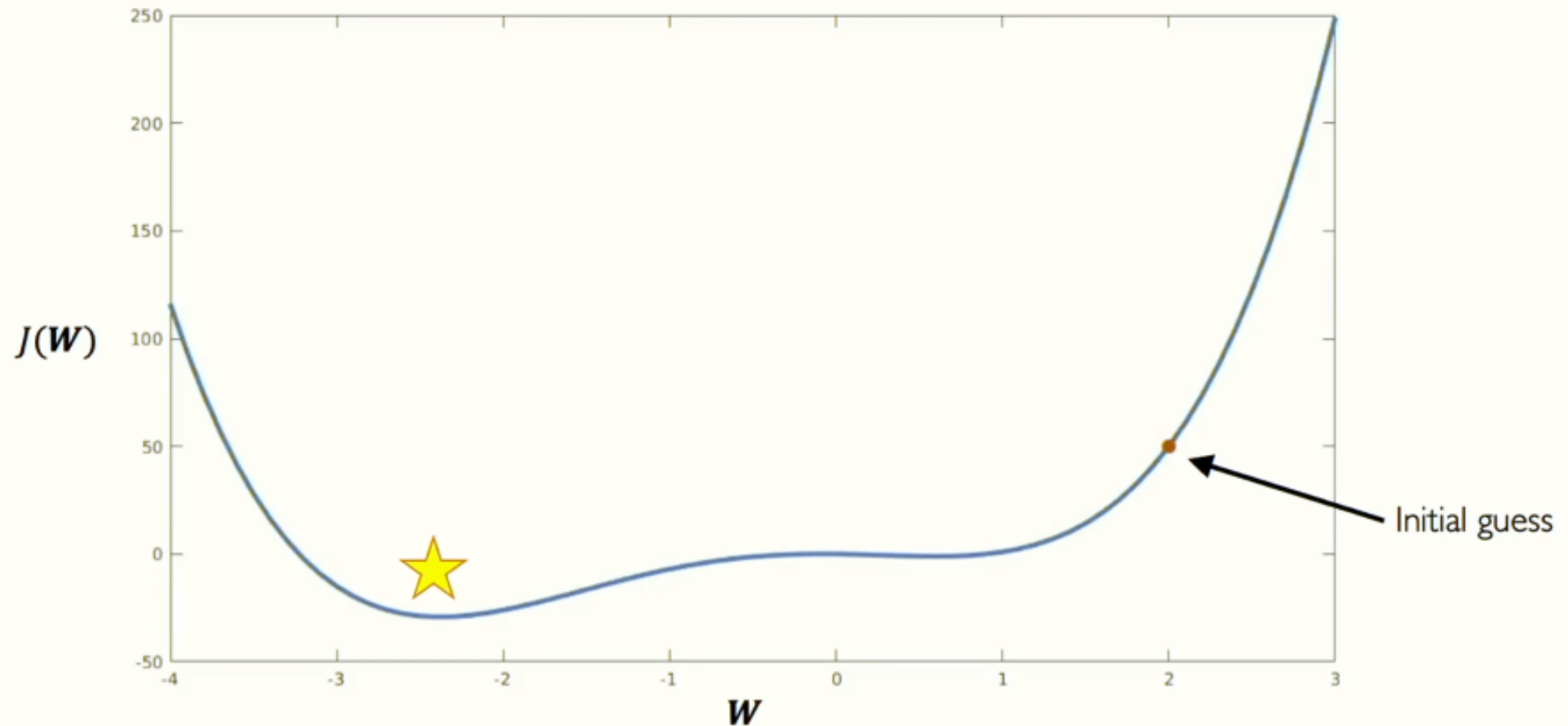
Optimization through gradient descent

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

How can we set the  
learning rate?

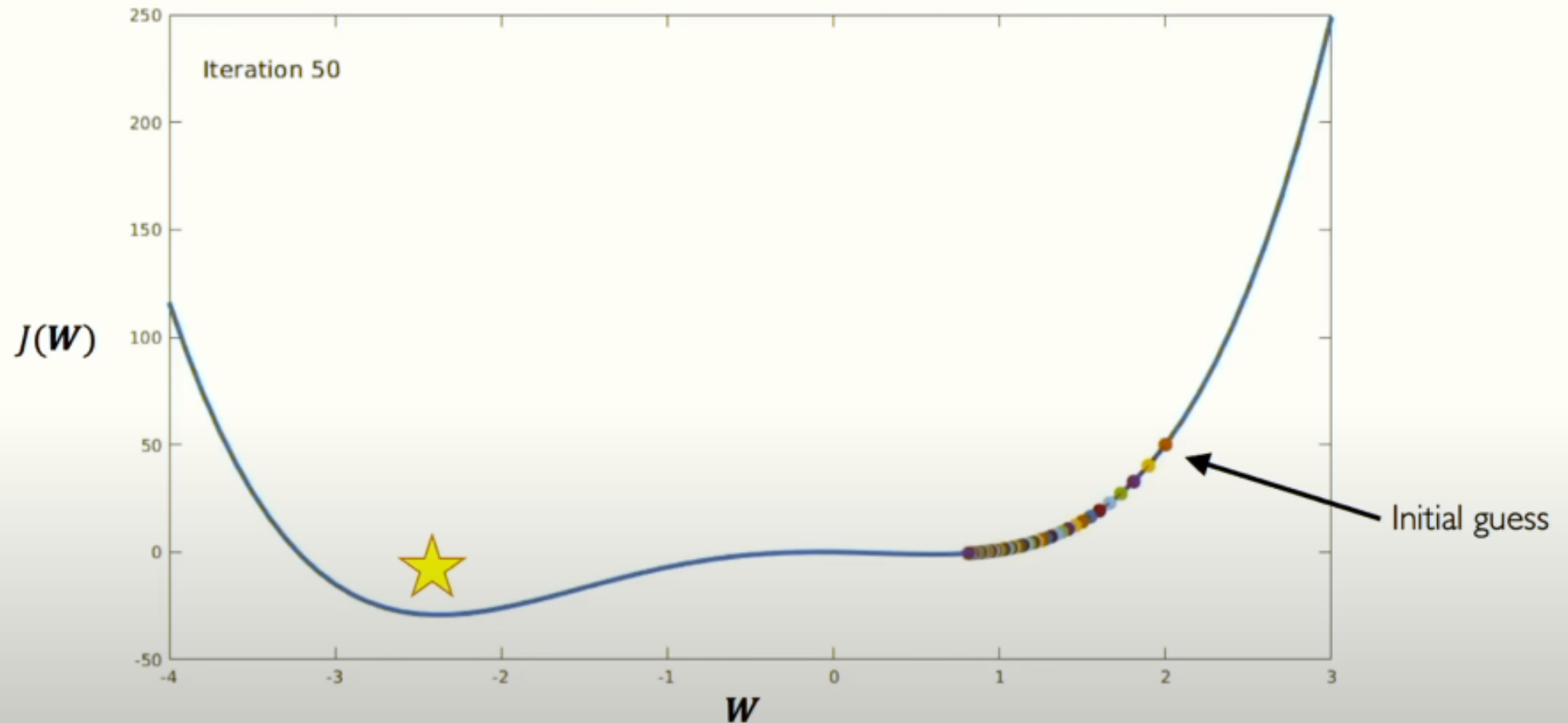
# Setting the Learning Rate

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



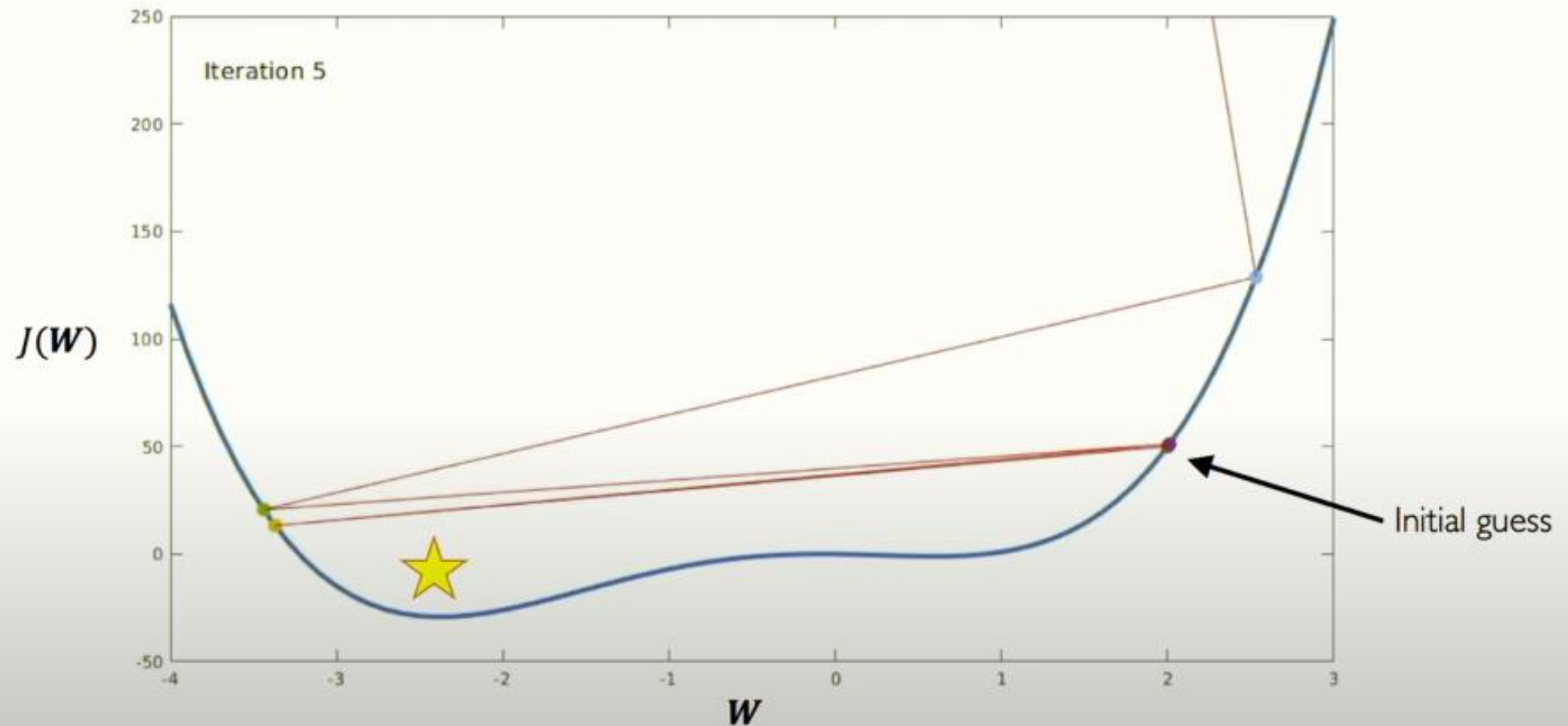
# Setting the Learning Rate

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



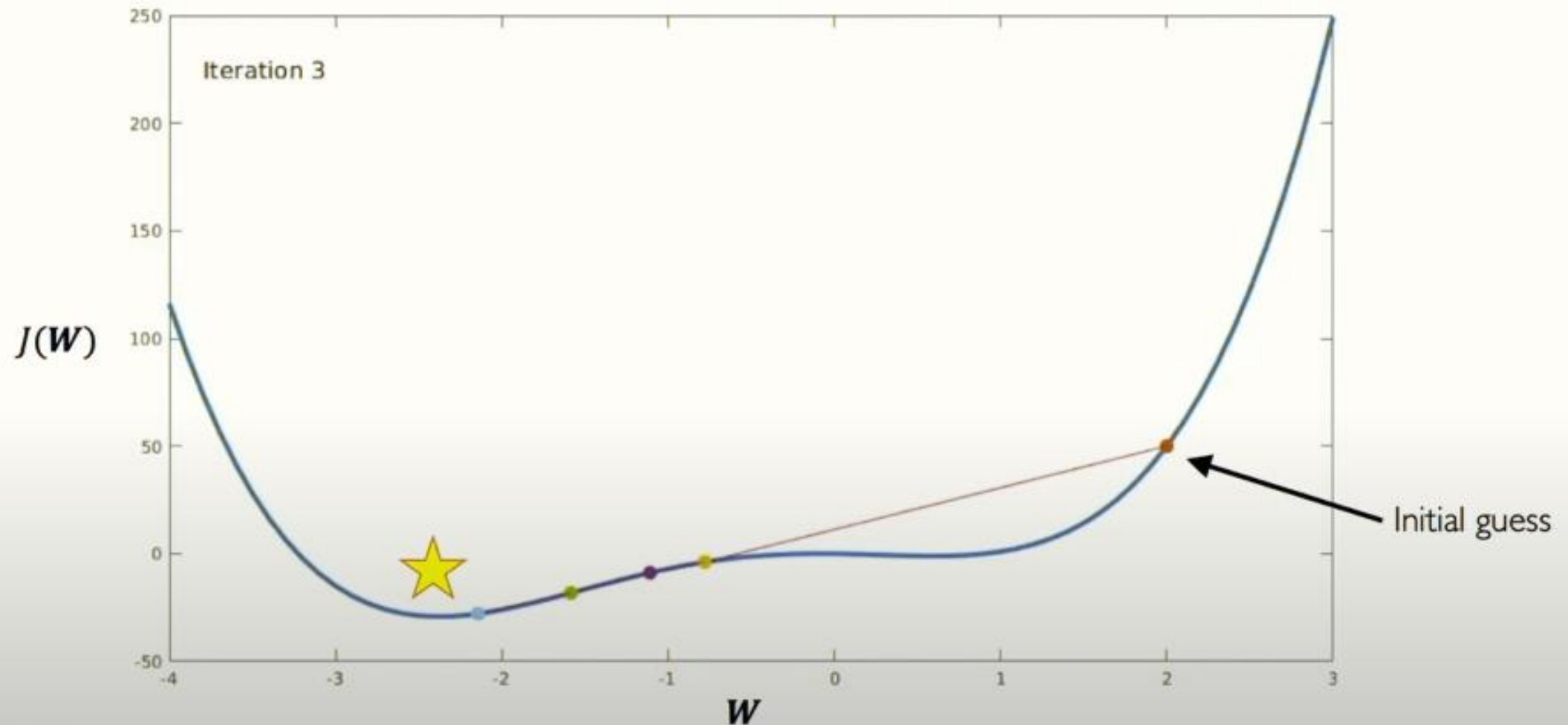
# Setting the Learning Rate

*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...

# 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.

# Putting it all together



```
import tensorflow as tf

model = tf.keras.Sequential([...])

# pick your favorite optimizer
optimizer = tf.keras.optimizer.SGD()

while True: # loop forever

    # forward pass through the network
    prediction = model(x)

    with tf.GradientTape() as tape:
        # compute the loss
        loss = compute_loss(y, prediction)

    # update the weights using the gradient
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

# Putting it all together



```
import tensorflow as tf

model = tf.keras.Sequential([...])

# pick your favorite optimizer
optimizer = tf.keras.optimizer.SGD()

while True: # loop forever

    # forward pass through the network
    prediction = model(x)

    with tf.GradientTape() as tape:
        # compute the loss
        loss = compute_loss(y, prediction)

    # update the weights using the gradient
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```



Can replace with  
any TensorFlow  
optimizer!

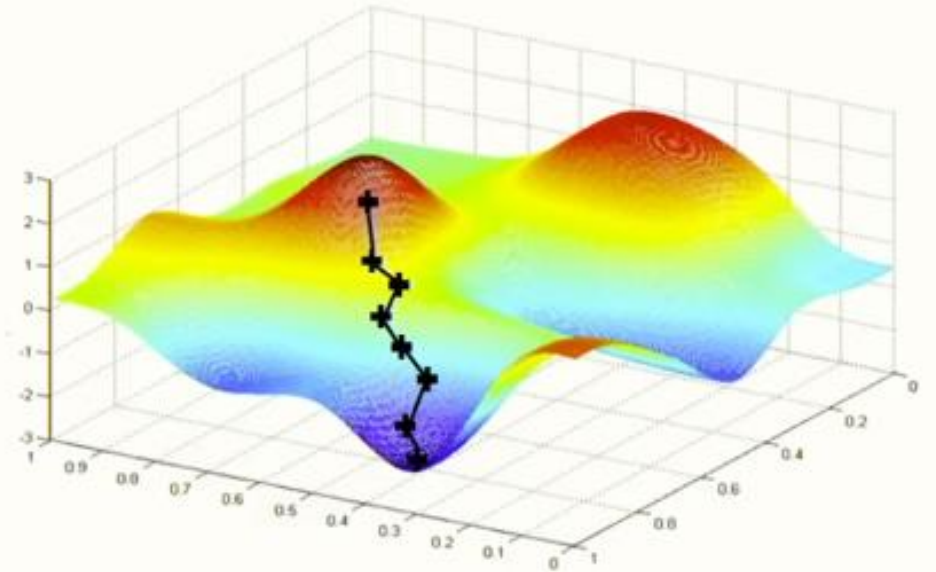
# Neural Networks in Practice: Mini-batches

# 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

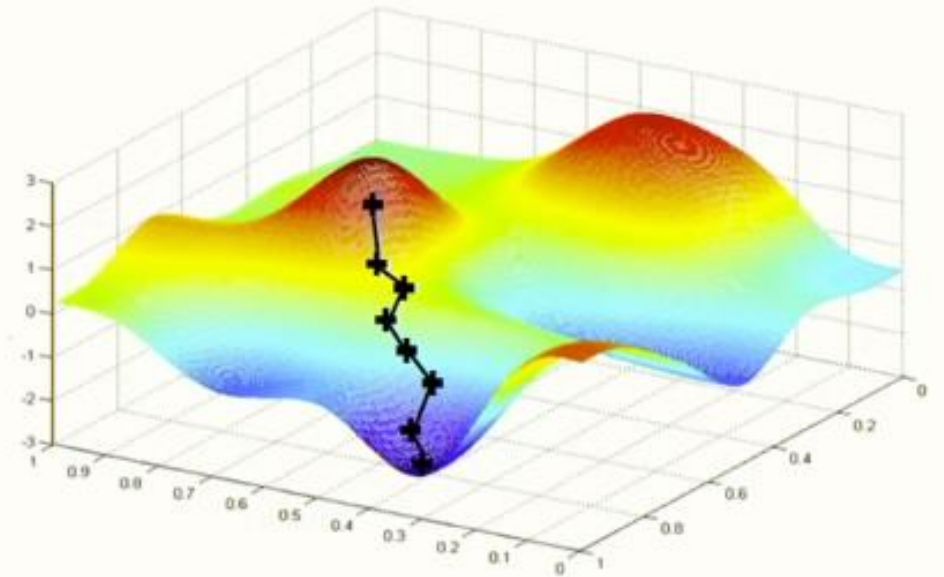
Can be very  
**computationally**  
intensive to compute!



# Stochastic Gradient Descent

## Algorithm

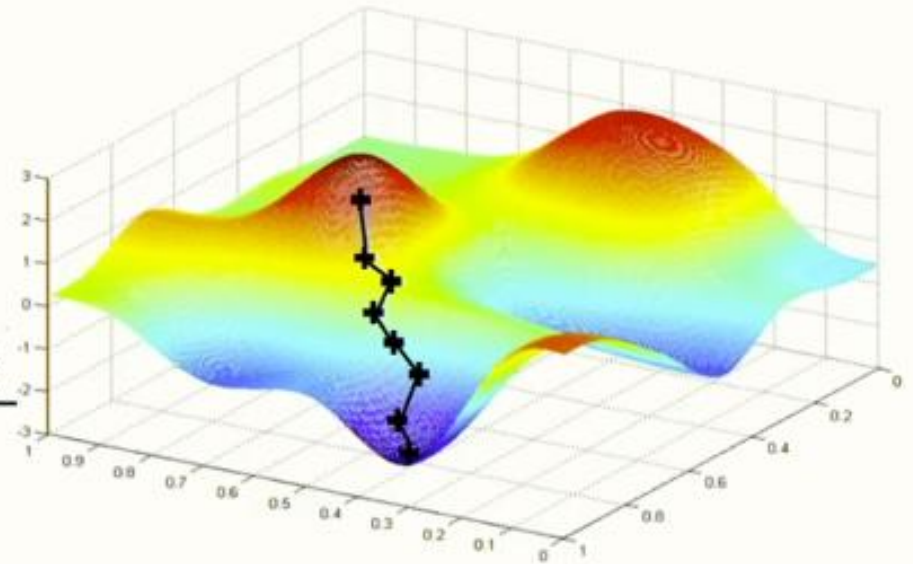
1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3.     Pick single data point  $i$
4.     Compute gradient,  $\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}}$
5.     Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial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



# Stochastic Gradient Descent

## Algorithm

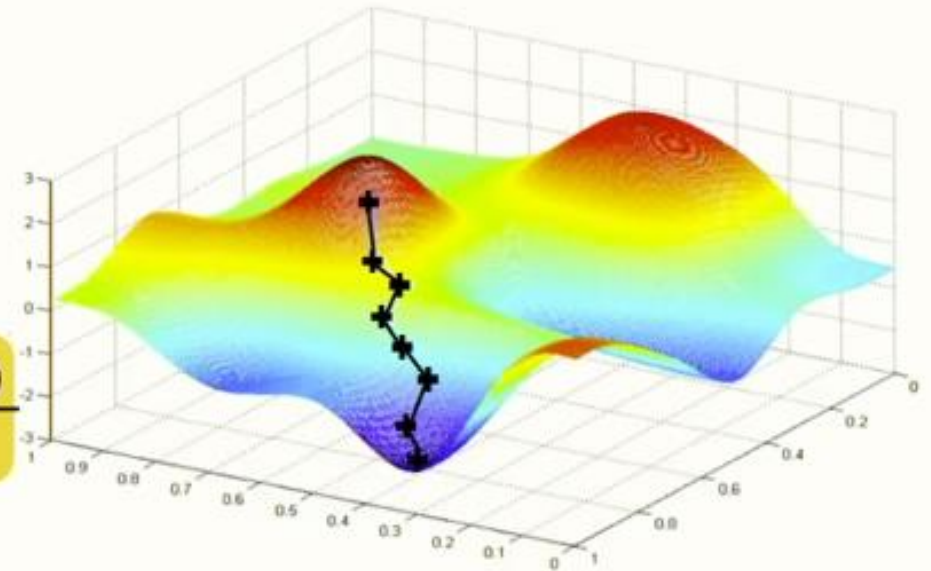
1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3.     Pick batch of  $B$  data points
4.     Compute gradient,  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
5.     Update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

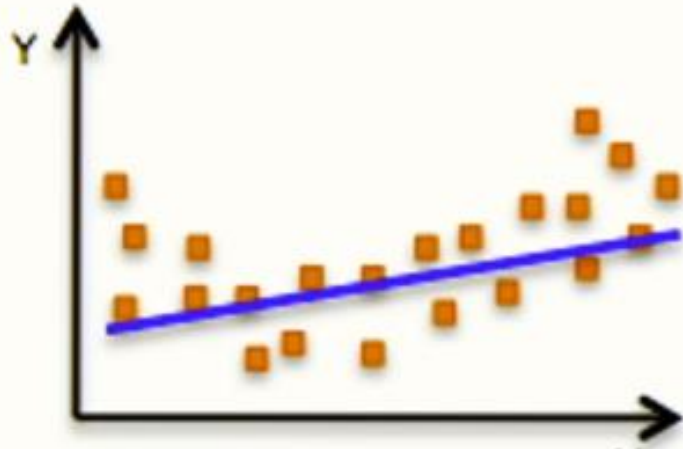
Smother convergence  
Allows for larger learning rates

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

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

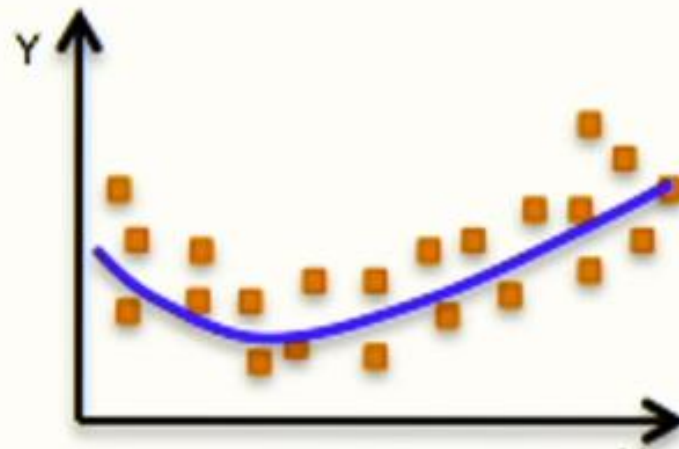
# Neural Networks in Practice: Overfitting

# The Problem of Overfitting

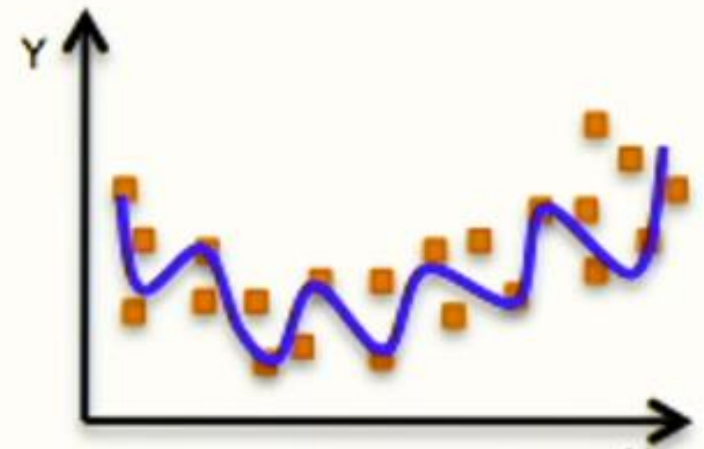


## **Underfitting**

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?*

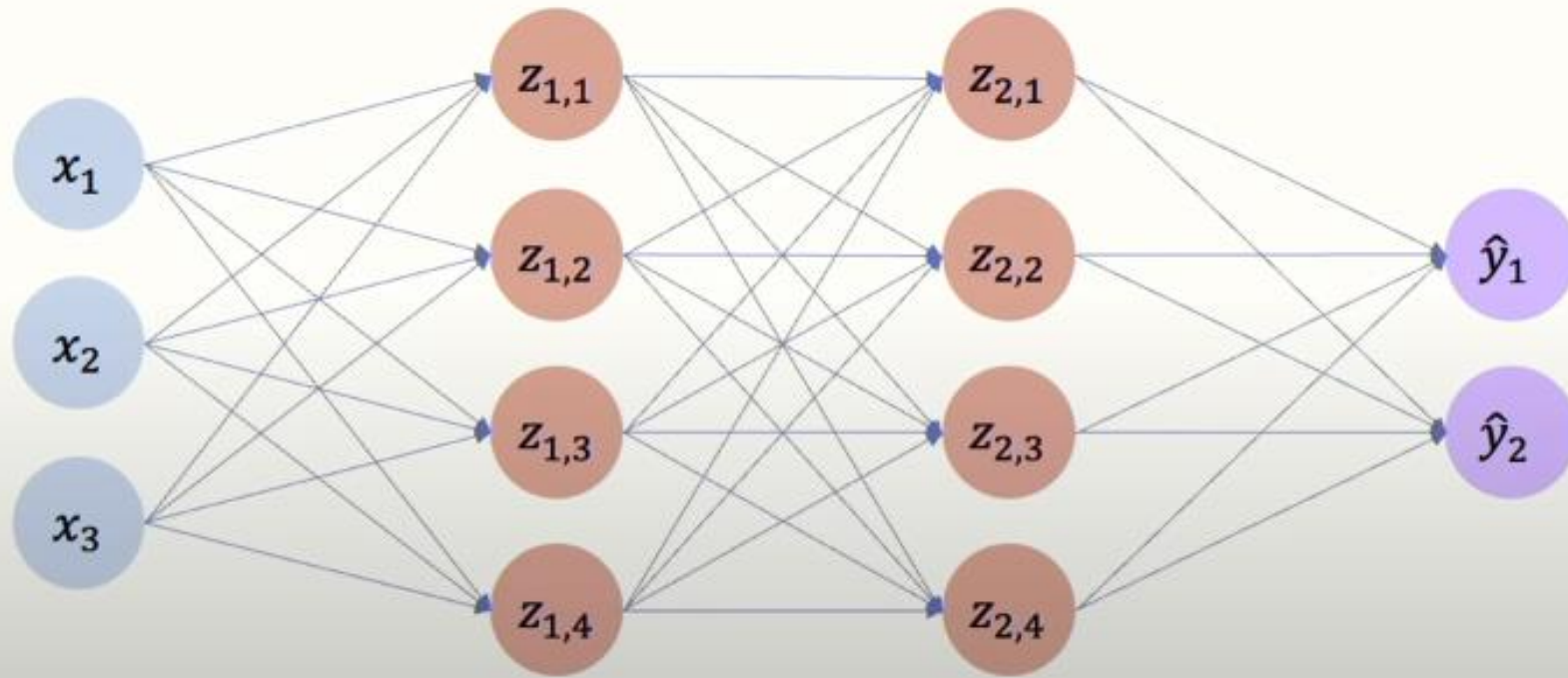
*Technique that constrains our optimization problem to discourage complex models*

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

*Improve generalization of our model on unseen data*

# Regularization I: Dropout

- During training, randomly set some activations to 0

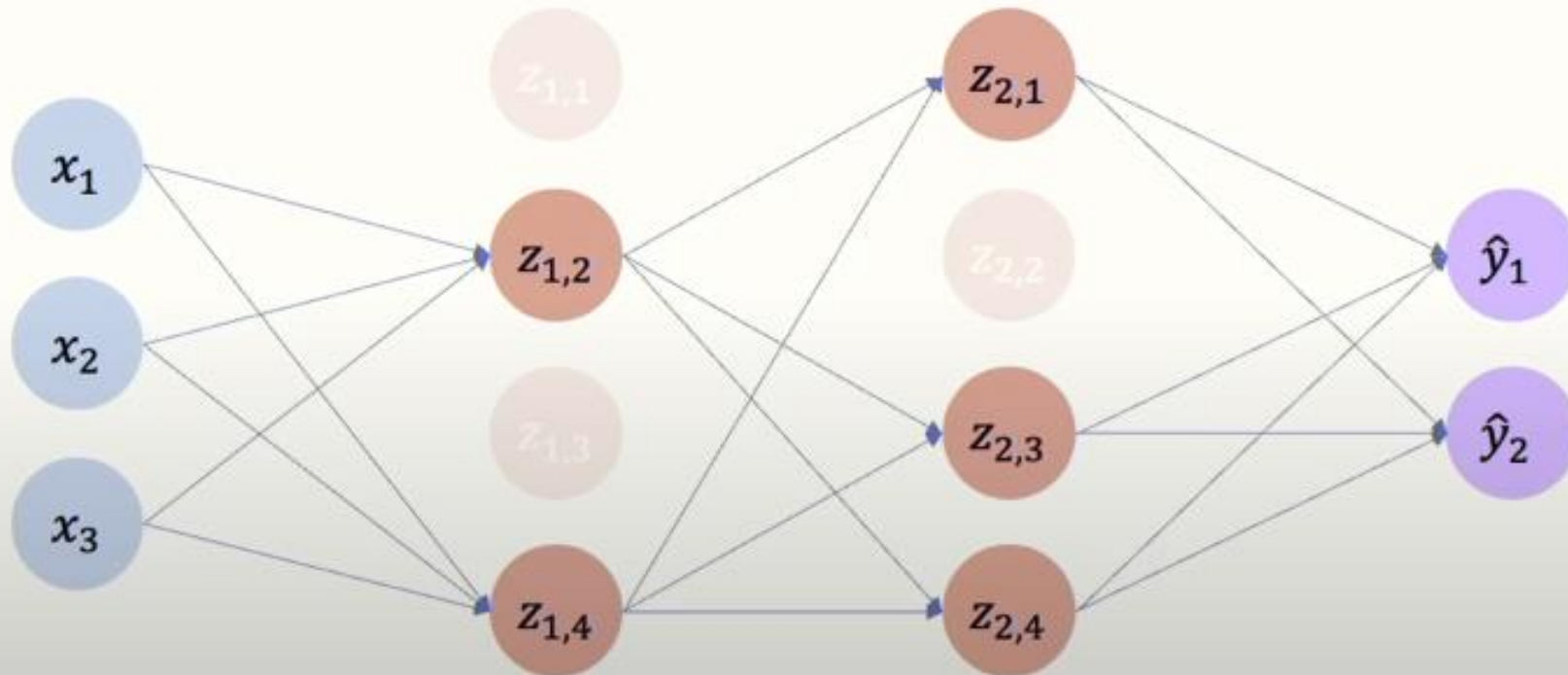


# Regularization I: Dropout

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

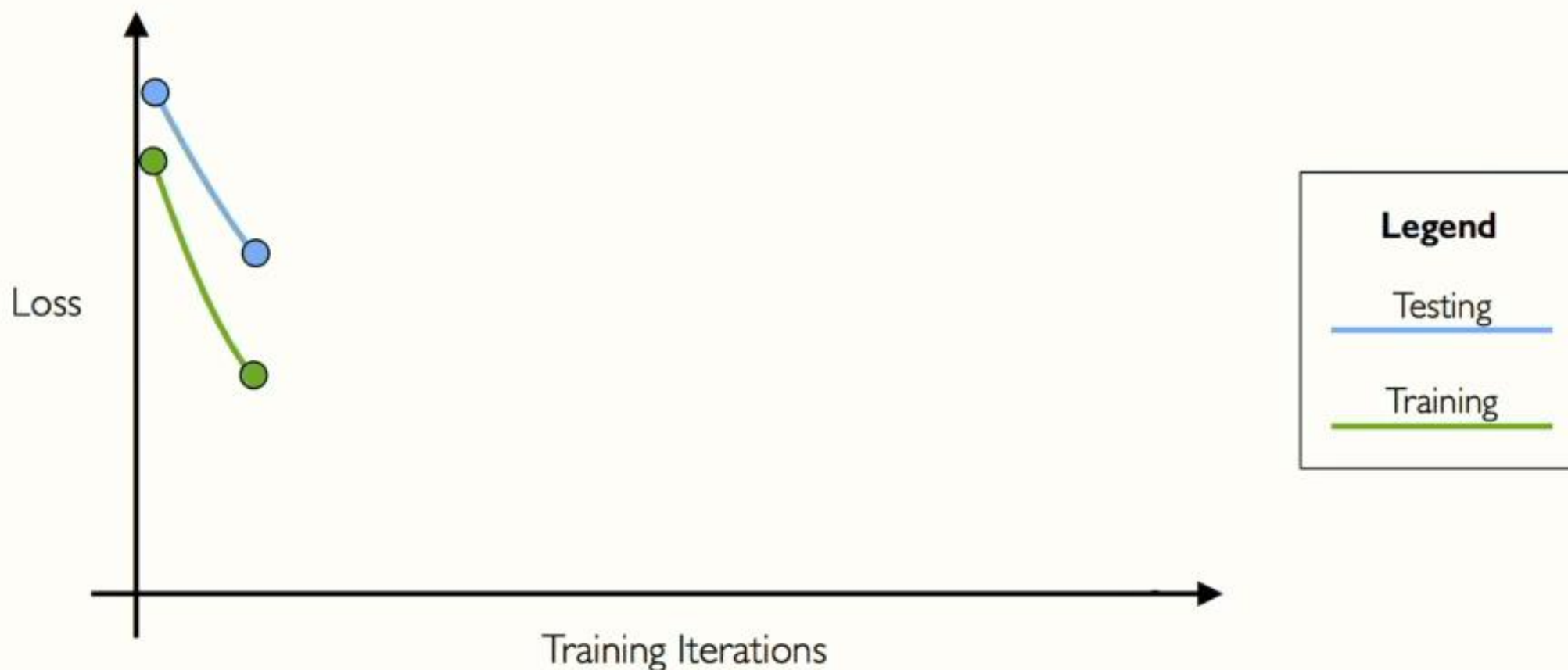


`tf.keras.layers.Dropout(p=0.5)`



# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



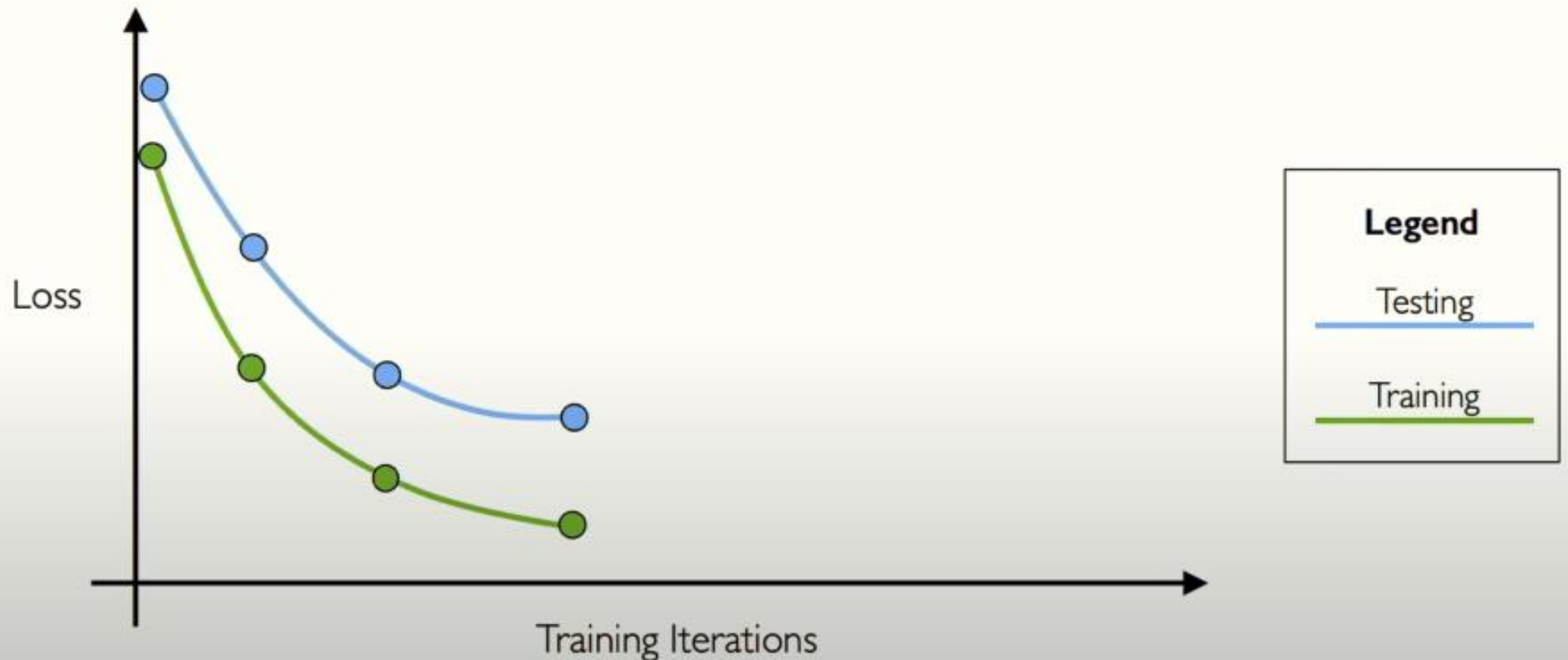
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



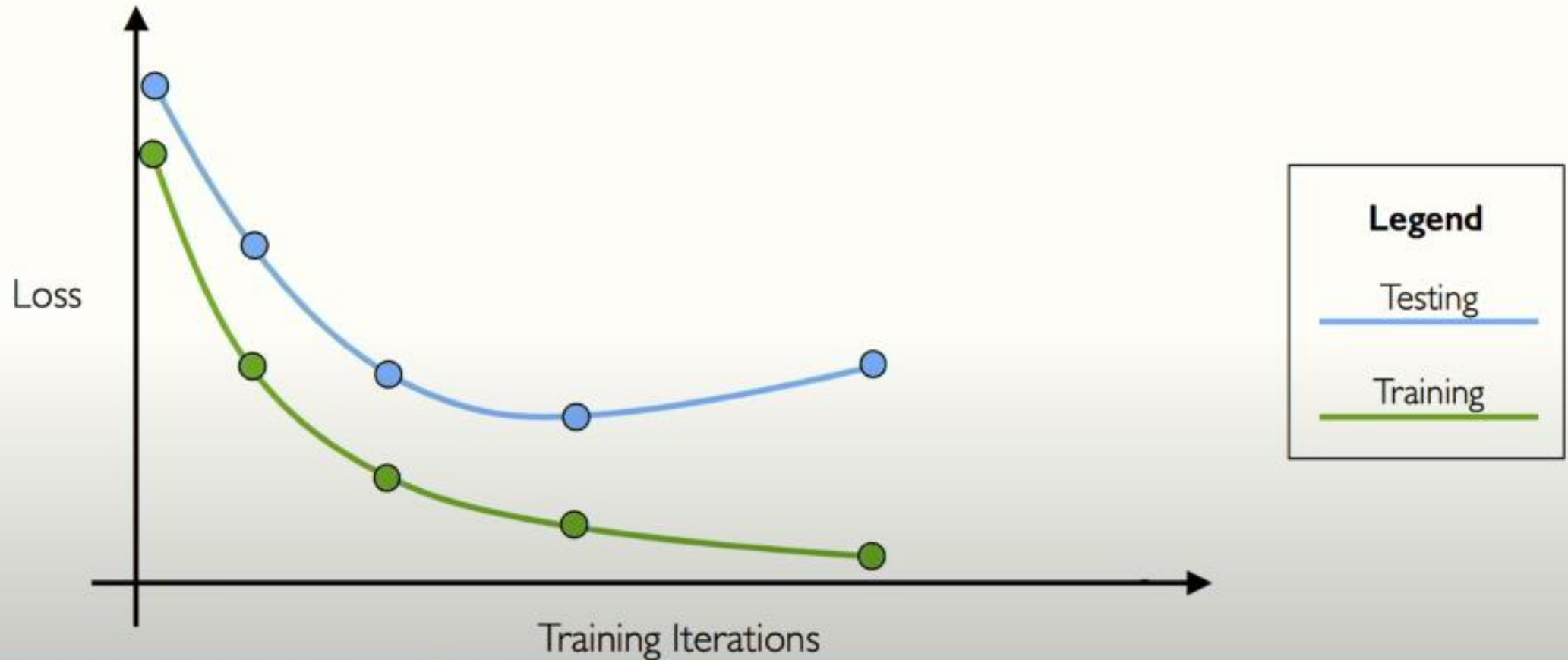
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



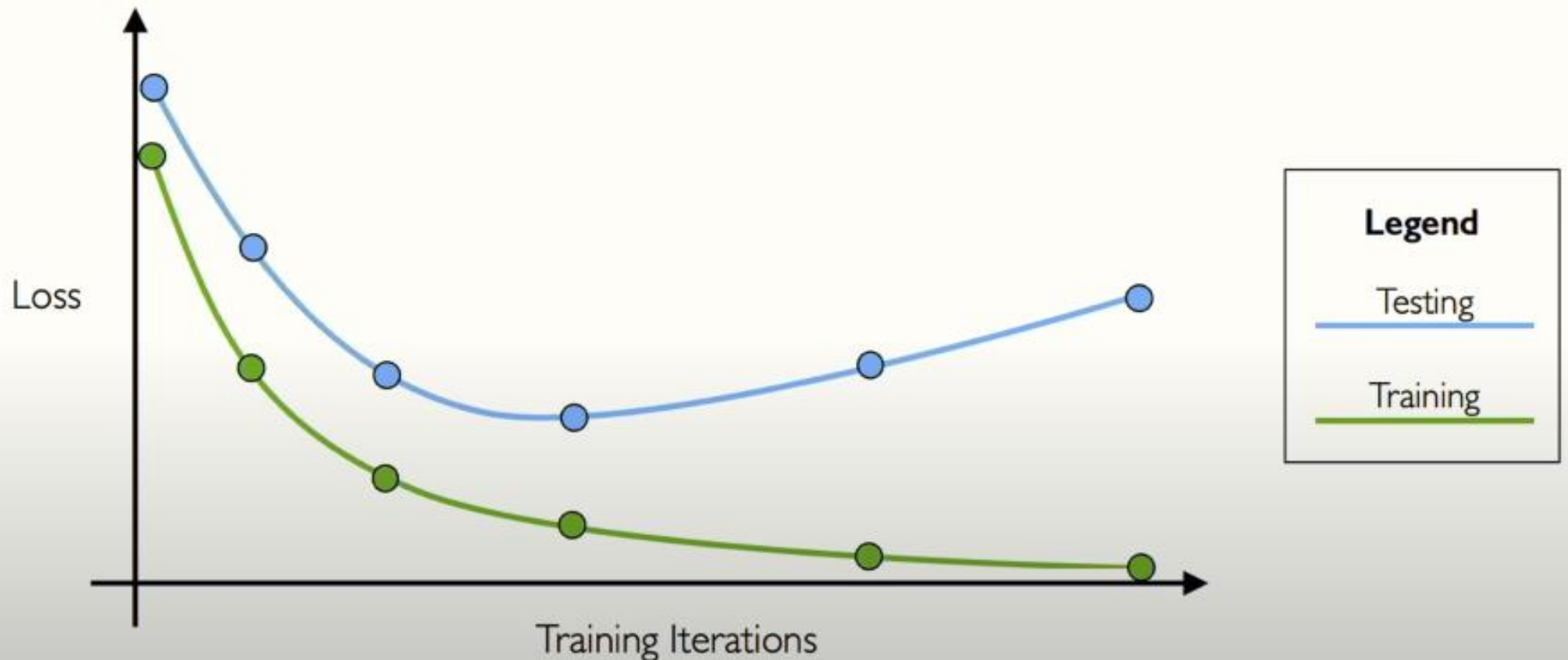
# Regularization 2: Early Stopping

- 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



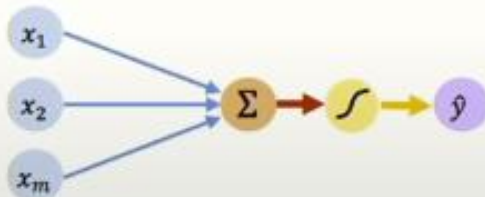
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



## 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

