# Introduction to Deep Learning

Alexander Amini

MIT Introduction to Deep Learning

January 9, 2023

Hi everybody, and welcome to MIT 6.S191
CANNY AI

[redacted] • 3 months ago

That is easily the cleanest visual deepfake I've ever seen. It must have taken ages to render, because it just looks flawless.

[redacted] • 3 months ago

I'm aware of the capabilities of DeepFakes but to create it for a class intro is just amazing! This is how you practice what you teach. Love it

a [redacted] • 5 months ago

THAT INTRO TO THE LECTURE IS SAVAGE!!!

S [redacted] • 3 months ago

This is the best example of a Course that sells itself.😅

[redacted] • 4 months ago

Omg i wish my classes were this cool...

1.1M views 🤯

[redacted] • 2 months ago

WOW WOW WOW i am amazed.

J [redacted] • 3 months ago

Plot twist: that actually was the real Obama.

J [redacted] • 4 months ago

damn.... i was about to ask "how can we be sure that the welcoming video is not synthesized?" , then i kept watching till the end xD

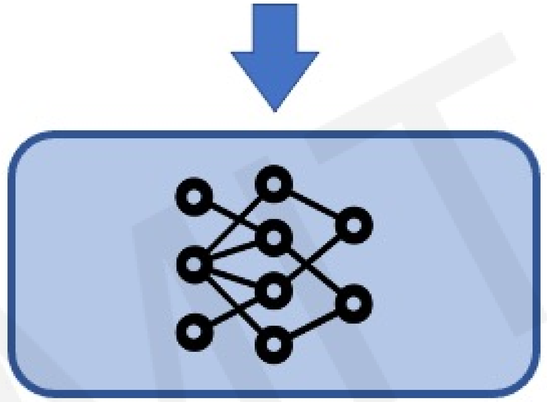
M [redacted] • 10 months ago

Did not see that coming! Simply Amazing! 🔥

VISTA
SIMULATOR
VISTA: Synthesizing photorealistic
environments for training autonomous vehicles
Amini/Wang+ ICRA 2022, Wang/Amini+ ICRA 2022

# Generating Images from Natural Language

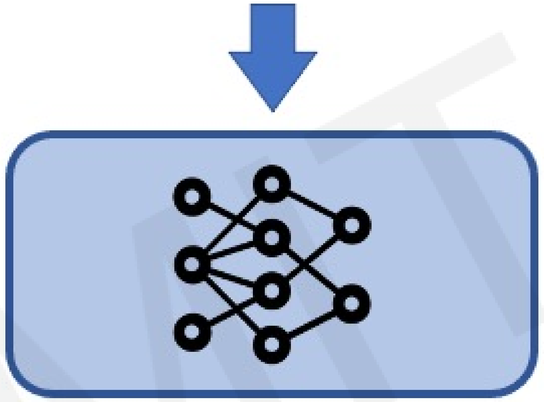
"A photo of an astronaut riding a horse."



Ramesh+ *arXiv* 2022

# Generating Language from Natural Language

"Write code in TensorFlow to train a neural network."



OpenAI.



Certainly! Here is an example of how you can use TensorFlow to train a neural network for binary classification:

Copy code

```
import tensorflow as tf

# Load the data
(x_train, y_train), (x_test, y_test) = load_data()

# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=x_train.shape[1:]),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=
['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=
(x_test, y_test))
```

This code assumes that you have a function `load_data()` that returns the training and test datasets as tuples of NumPy arrays. The input shape of the model is determined by the shape of the training data, and the model has two dense layers with 16 and 1 units, respectively, with ReLU and sigmoid activations. The model is then compiled with an Adam optimizer and a binary cross-entropy loss function, and is trained using the `fit()` method.

I hope this helps! Let me know if you have any questions.

# 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

MIT 6.S191

# Lecture Schedule

**Intro to Deep Learning**

**Lecture 1**
*Jan. 9, 2023*
[Slides] [Video] *coming soon!*

**Deep Sequence Modeling**

**Lecture 2**
*Jan. 9, 2023*
[Slides] [Video] *coming soon!*

**Intro to TensorFlow; Music Generation**

**Software Lab 1**
[Code]

**Deep Computer Vision**

**Lecture 3**
*Jan. 10, 2023*
[Slides] [Video] *coming soon!*

**Deep Generative Modeling**

**Lecture 4**
*Jan. 10, 2023*
[Slides] [Video] *coming soon!*

**Facial Detection Systems**

**Software Lab 2**
[Paper] [Code]

**Deep Reinforcement Learning**

**Lecture 5**
*Jan. 11, 2023*
[Slides] [Video] *coming soon!*

**Uncertainty and Bias**

**Lecture 6**
*Jan. 12, 2023*
[Info] [Slides] [Video] *coming soon!*

**Debiasing, Unceratinty, and Robustness**

**Software Lab 3**
[Code]

**Limitations and New Frontiers**

**Lecture 7**
*Jan. 11, 2023*
[Slides] [Video] *coming soon!*

**The Modern Era of Statistics**

**Lecture 8**
*Jan. 12, 2023*
[Info][Slides] [Video] *coming soon!*

**Final Project**

**Work on final projects**

**Robot Learning**

**Lecture 9**
*Jan. 13, 2023*
[Info] [Slides] [Video] *coming soon!*

**Text-to-Image Generation**

**Lecture 10**
*Jan. 13, 2023*
[Info] [Slides] [Video] *coming soon!*

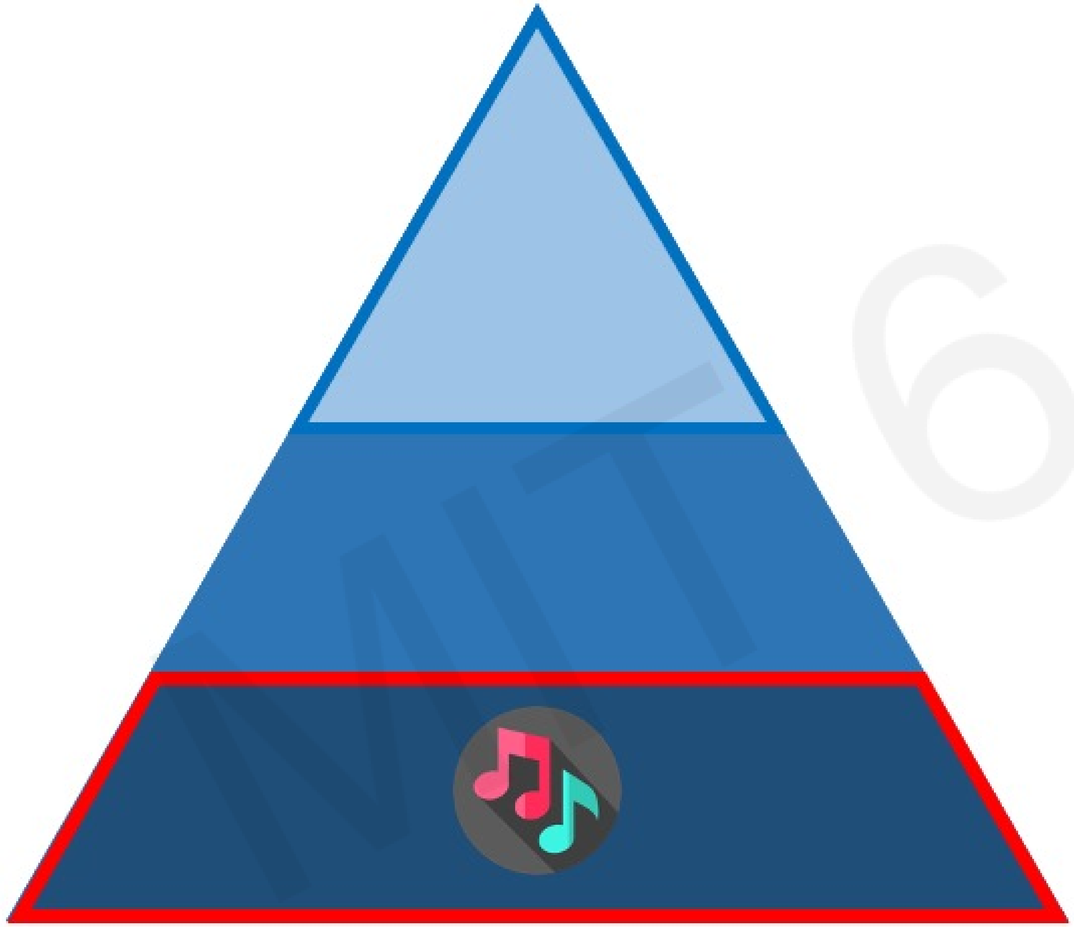
**Project Competition**

**Project pitches and final awards!**

- 1/9/23 – 1/13/23
- Lecture + Lab Breakdown
- Competitions
- Final Projects + Prizes!

# Labs and Prizes

**All due Thursday 1/12 at 11:00pm ET. Instructions: bit.ly/deeplearning-syllabus**



## Music Generation

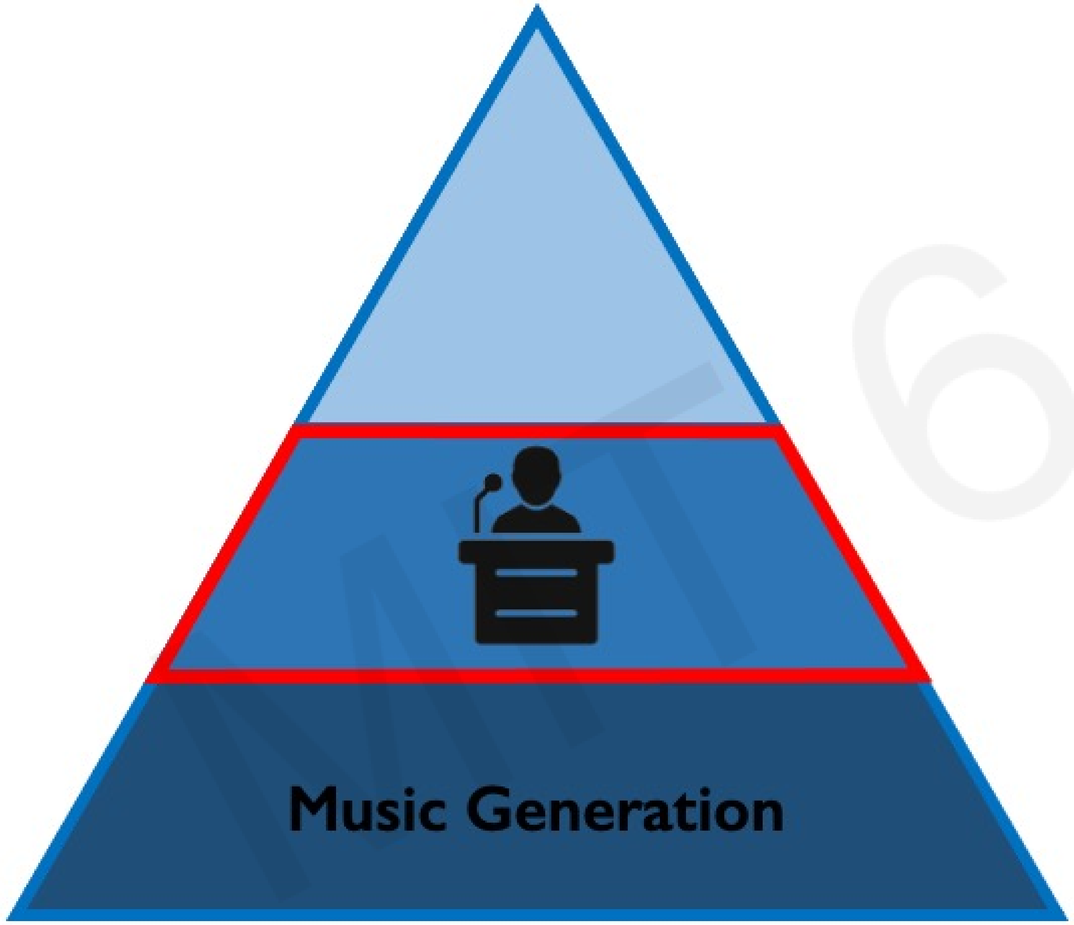
Build a neural network that can learn the genre of Irish folk songs and use it to generate brand new songs!

**Prize:**

MIT 6.S191

# Labs and Prizes

**All due Thursday 1/12 at 11:00pm ET. Instructions: bit.ly/deeplearning-syllabus**




## Project Pitch Competition

Present a novel deep learning research idea or application (3 minutes, strict)

Presentations on **Friday, Jan 13**

Submit groups by **Wed 1/11 11pm ET**

Submit slides by **Thu 1/12 11pm ET**

Instructions: bit.ly/deeplearning-syllabus

## Prizes:

**Gold:**
NVIDIA 3070 GPU

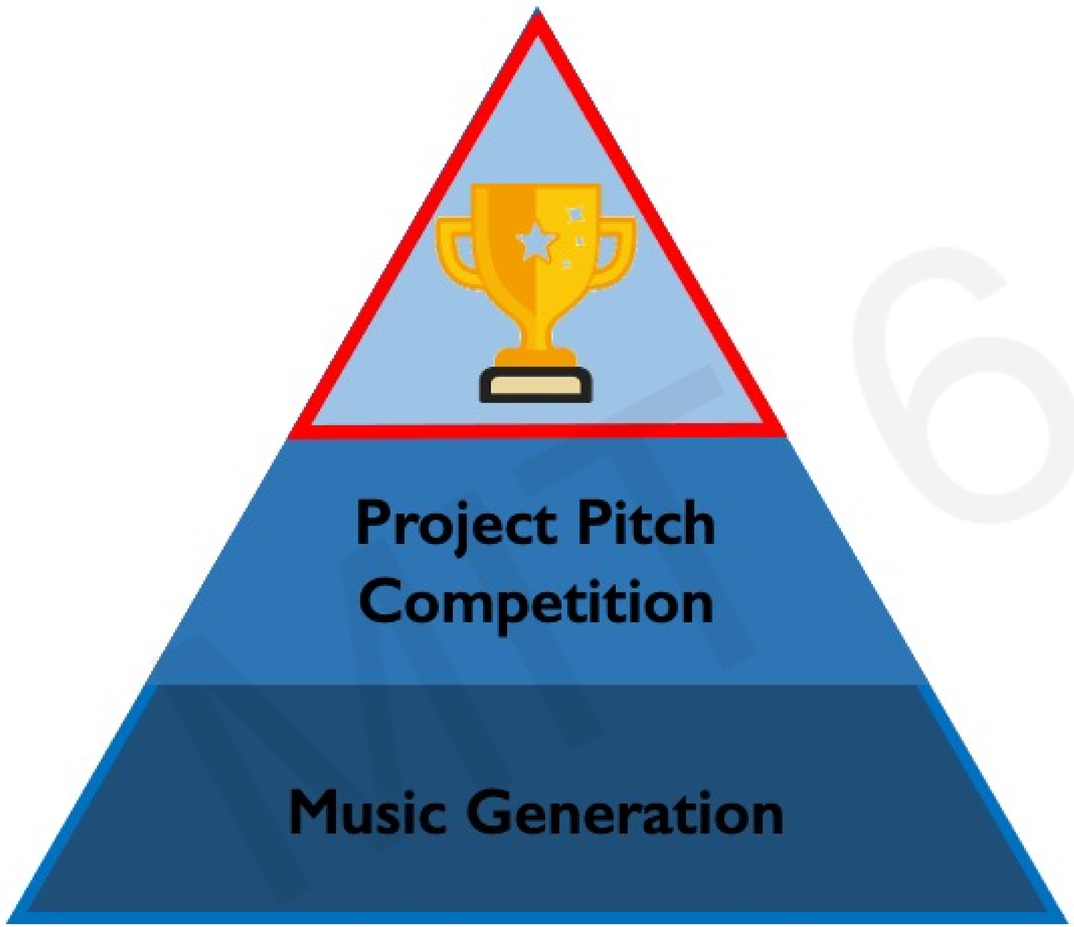
**Silver:**
Smartwatch

**Bronze:**
HD Monitor

MIT 6.S191

# Labs and Prizes

**All due Thursday 1/12 at 11:00pm ET. Instructions: bit.ly/deeplearning-syllabus**




## Trustworthy Deep Learning

Build solutions to improve robustness, mitigate bias, and increase accuracy of state-of-the-art vision systems!

(Software labs 2 and 3)

## Prizes:

**Gold**
**$1000**

**Silver**
**$750**

**Bronze**
**$500**

MIT 6.S191

# Program Support

- All lectures will be held in person in 32-123
- Software labs + office hours in 32-123
- Piazza: http://piazza.com/mit/spring2023/6s191
  - Useful for discussing labs
- Program Website: http://introtodeeplearning.com
  - Lecture schedule
  - Slides and lecture recordings
  - Software labs
- Syllabus: bit.ly/deeplearning-syllabus
- Labs: https://github.com/aamini/introtodeeplearning
- Email us: introtodeeplearning-staff@mit.edu

MIT 6.S191

# Program Staff

Alexander Amini
Lead Instructor

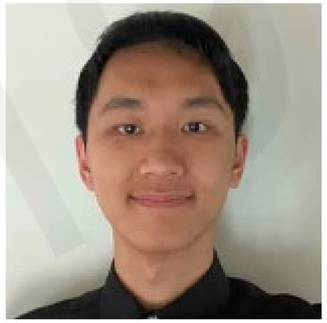
Ava Amini
Lead Instructor

Sadhana Lolla
Lead TA

Daniela Rus
Director of CSAIL

## Program TAs

Eva



James

Johnson

Pranav

Rohil

Sam

**introtodeeplearning-staff@mit.edu**

MIT 6.S191

# Thanks to Sponsors!

Google

THEMIS AI

MIT CSAIL

Vanguard

MIT-IBM Watson AI Lab

# Why Deep Learning and Why Now?

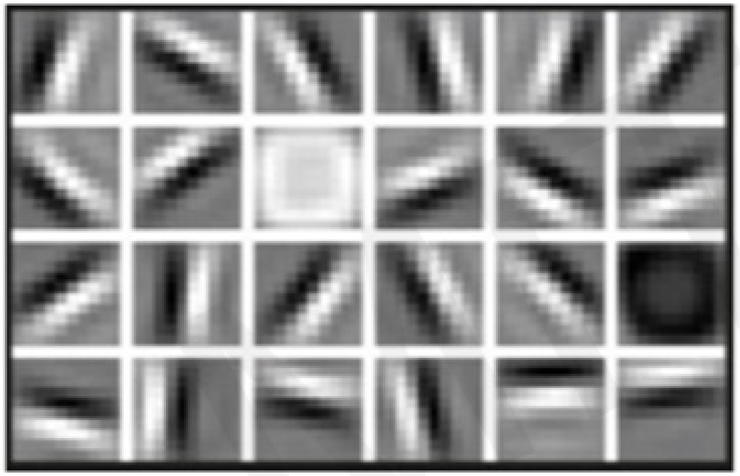
MIT 6.S191

# 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?

1952 Stochastic Gradient Descent

1958 Perceptron
- Learnable Weights

⋮

1986 Backpropagation
- Multi-Layer Perceptron

1995 Deep Convolutional NN
- Digit Recognition

⋮

## 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

TensorFlow

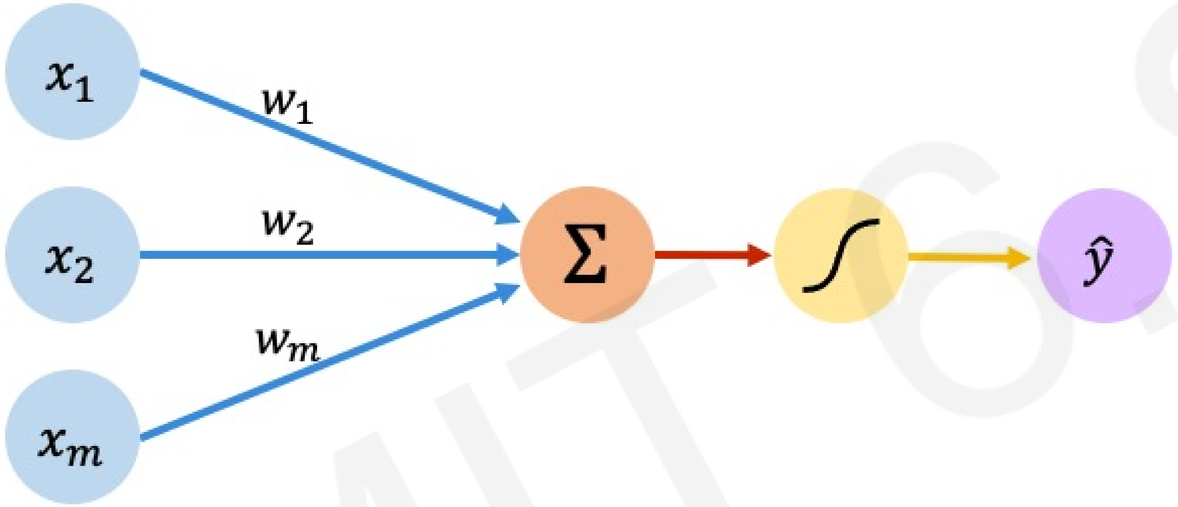
MIT 6.S191

# The Perceptron

## The structural building block of deep learning

MIT 6.S191

# The Perceptron: Forward Propagation




Inputs Weights Sum Non-Linearity Output

Output

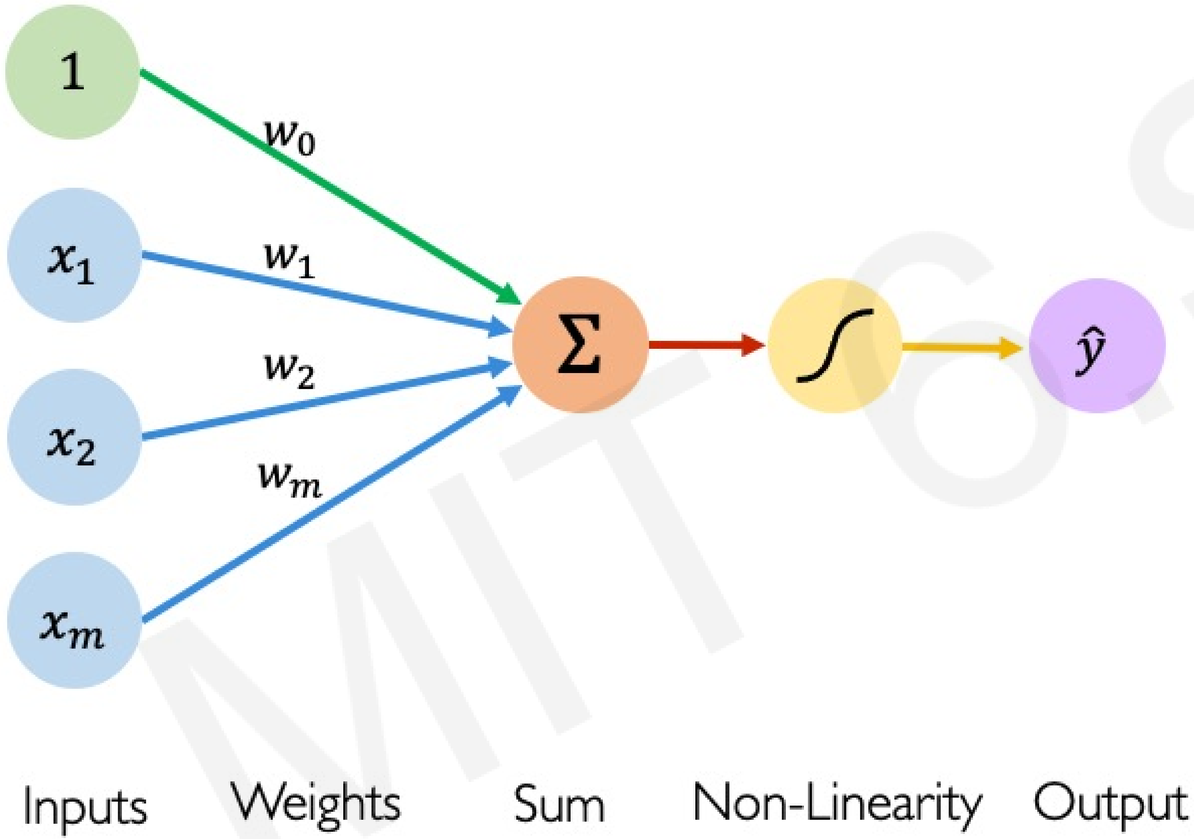
Linear combination of inputs

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

Non-linear activation function

MIT 6.S191

# The Perceptron: Forward Propagation




Output

Linear combination of inputs

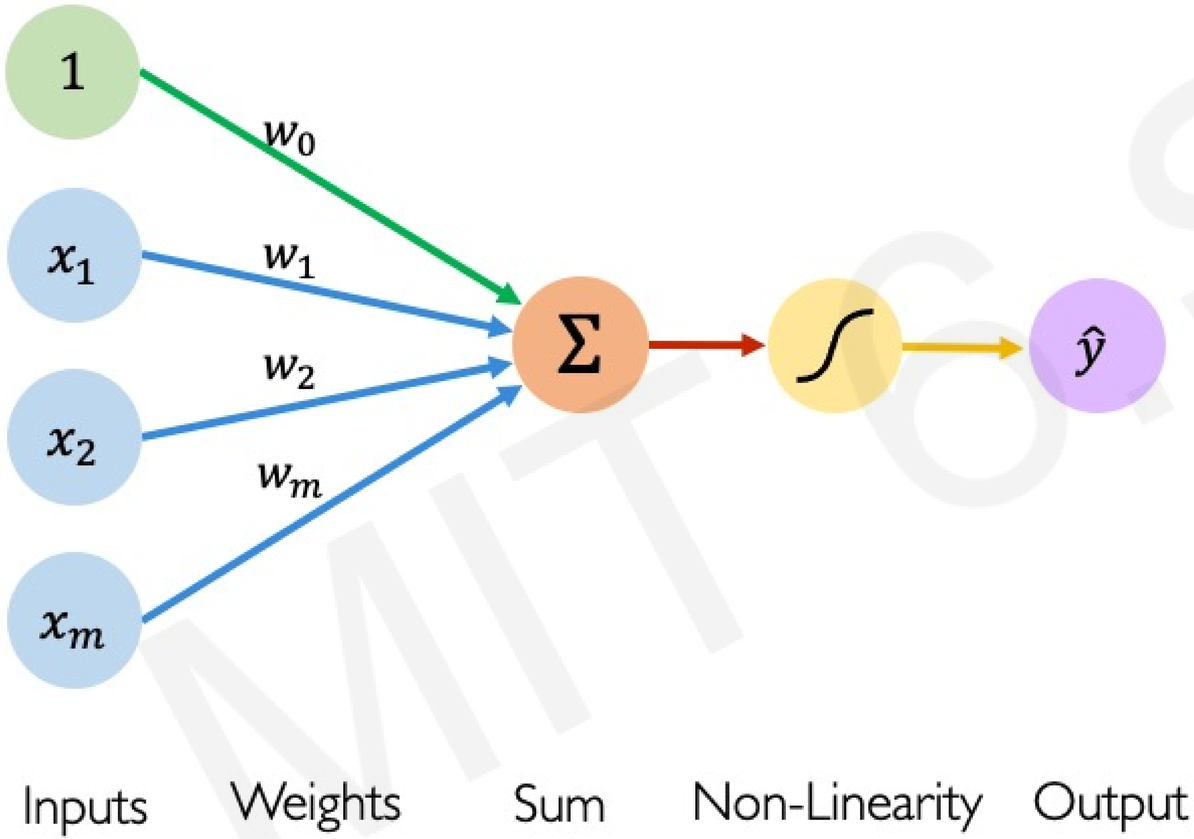
$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i \, w_i\right)$$

Non-linear activation function

Bias

MIT 6.S191

# The Perceptron: Forward Propagation



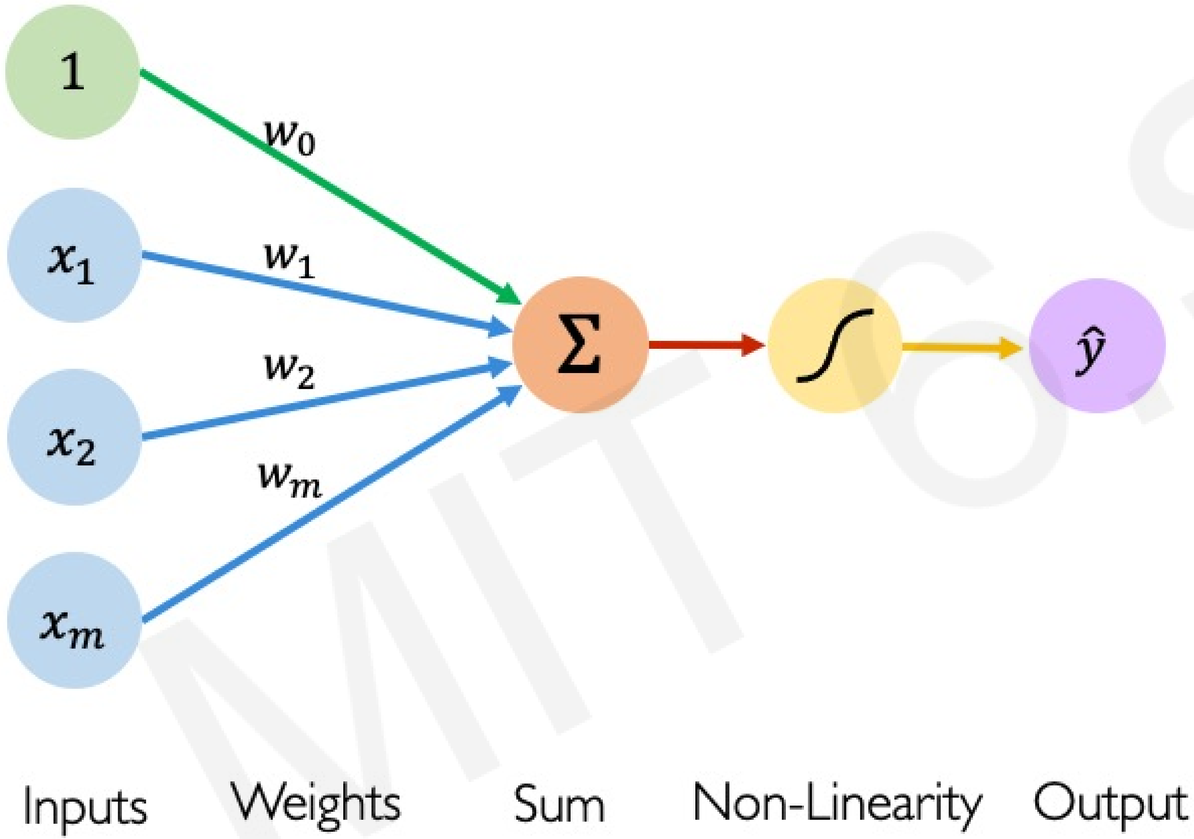

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i \, w_i\right)$$

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

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

MIT 6.S191

# The Perceptron: Forward Propagation

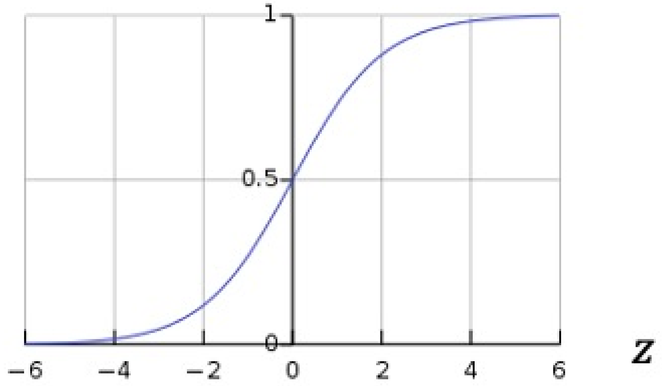



## Activation Functions

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

- Example: sigmoid function

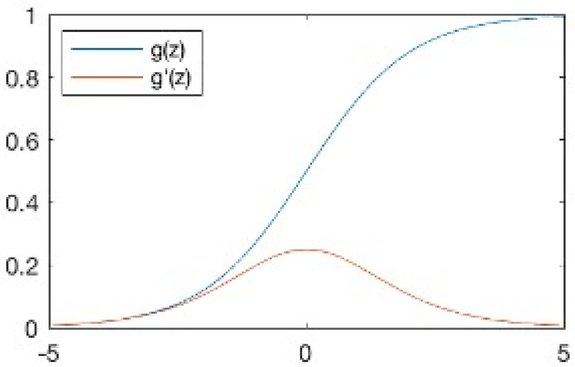
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$




MIT 6.S191

# Common Activation Functions

## Sigmoid Function

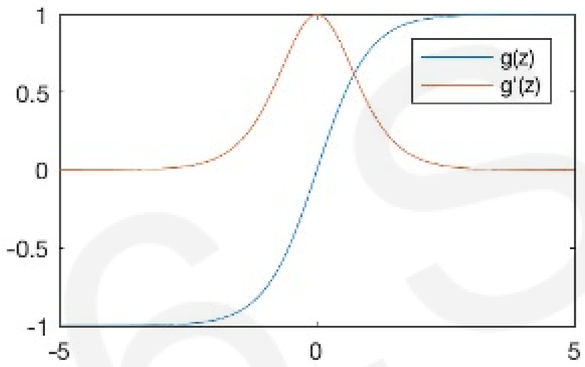



$$g(z) = \frac{1}{1 + e^{-z}}$$

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

```
tf.math.sigmoid(z)
```

## Hyperbolic Tangent

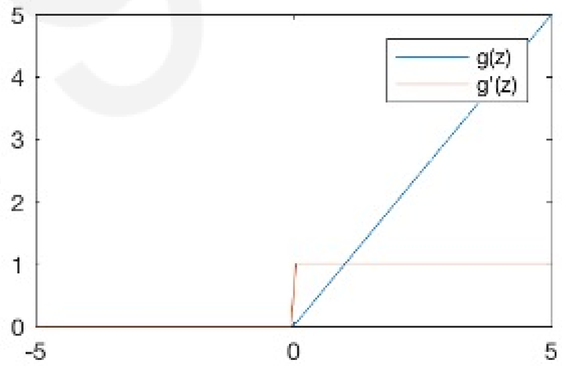



$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

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

```
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)
```

TensorFlow code blocks

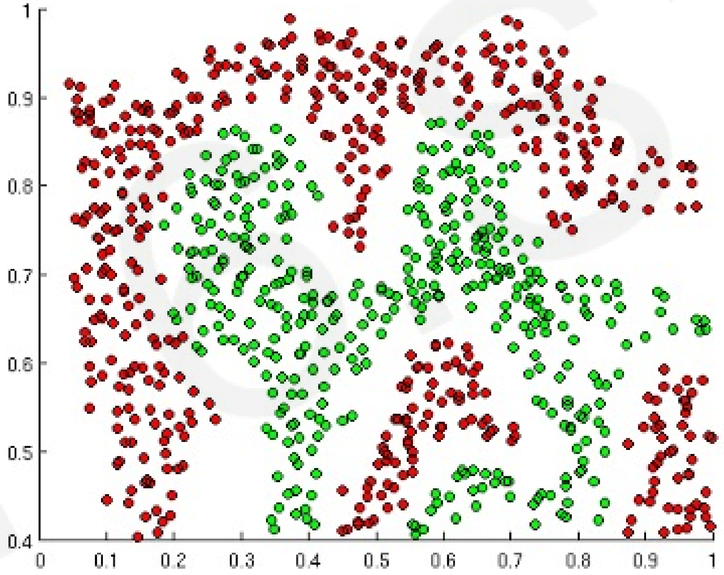
NOTE: All activation functions are non-linear



MIT 6.S191

# Importance of Activation Functions

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

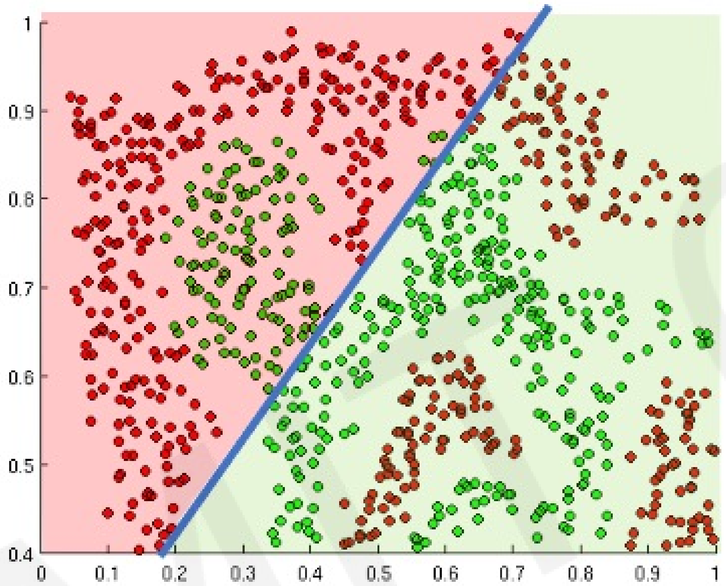



What if we wanted to build a neural network to distinguish green vs red points?

MIT 6S191

# Importance of Activation Functions

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

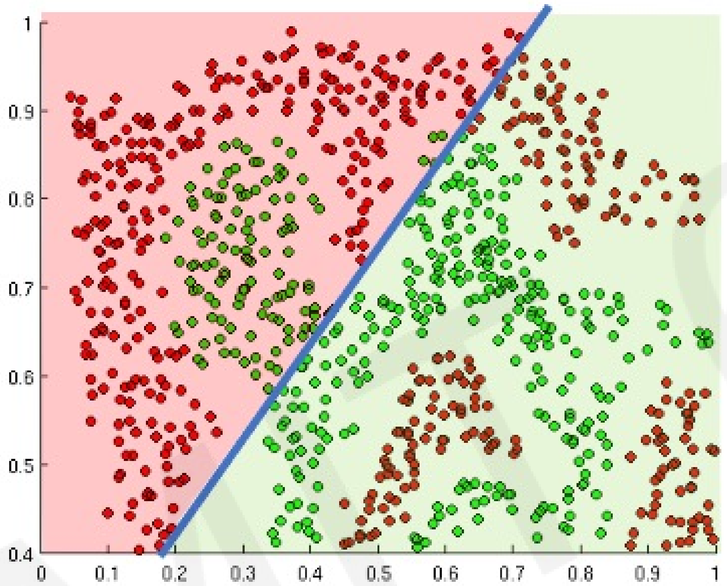



Linear activation functions produce linear decisions no matter the network size

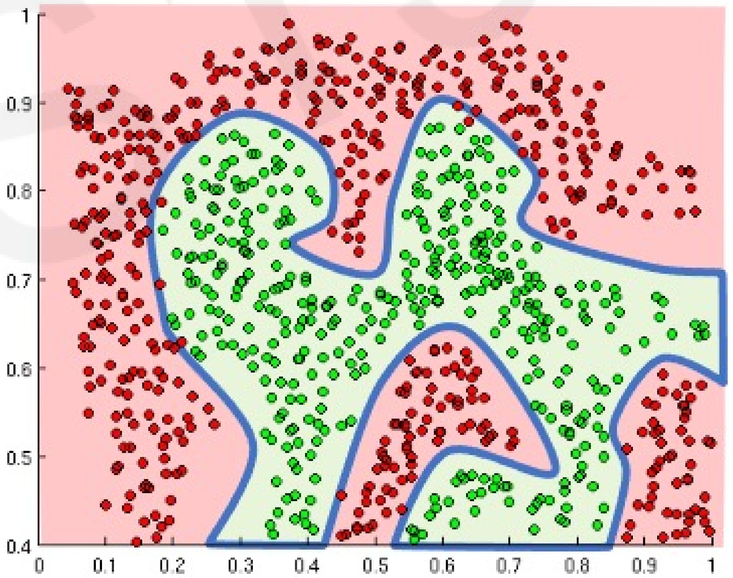
MIT 6.S191

# Importance of Activation Functions

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



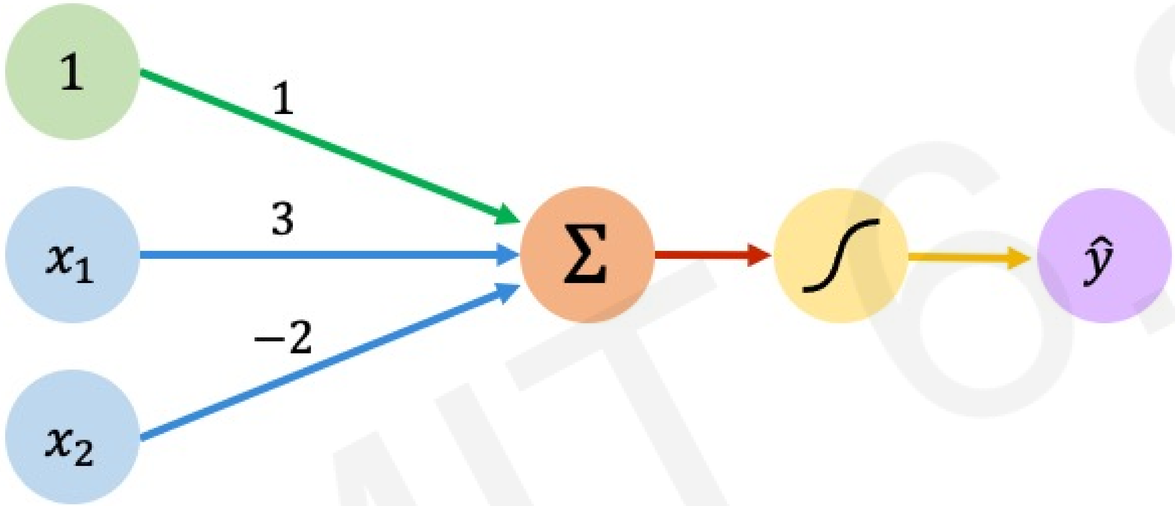

Linear activation functions produce linear decisions no matter the network size




Non-linearities allow us to approximate arbitrarily complex functions

MIT 6.S191

# The Perceptron: Example




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

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

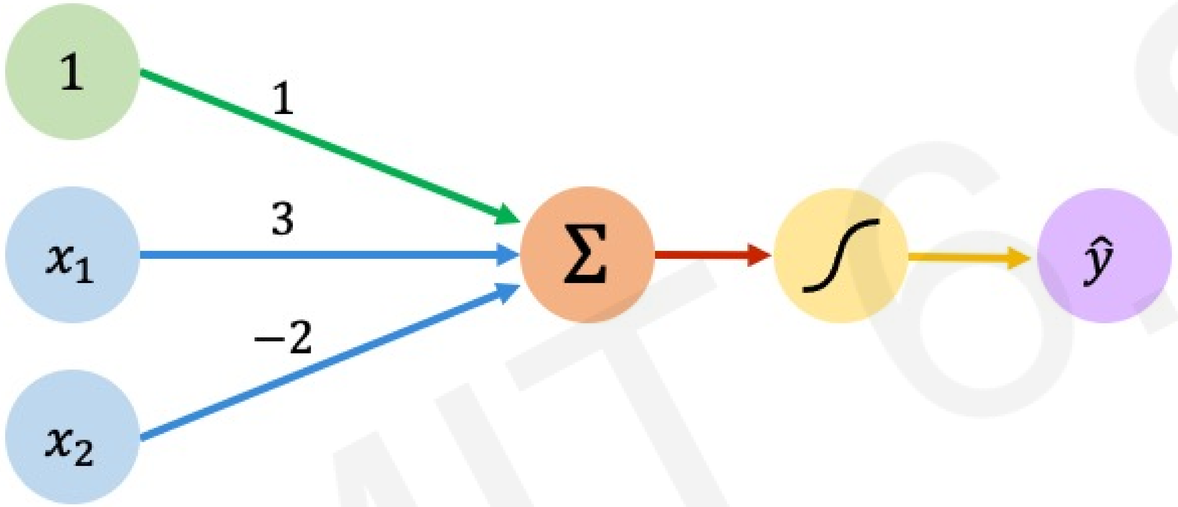
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$

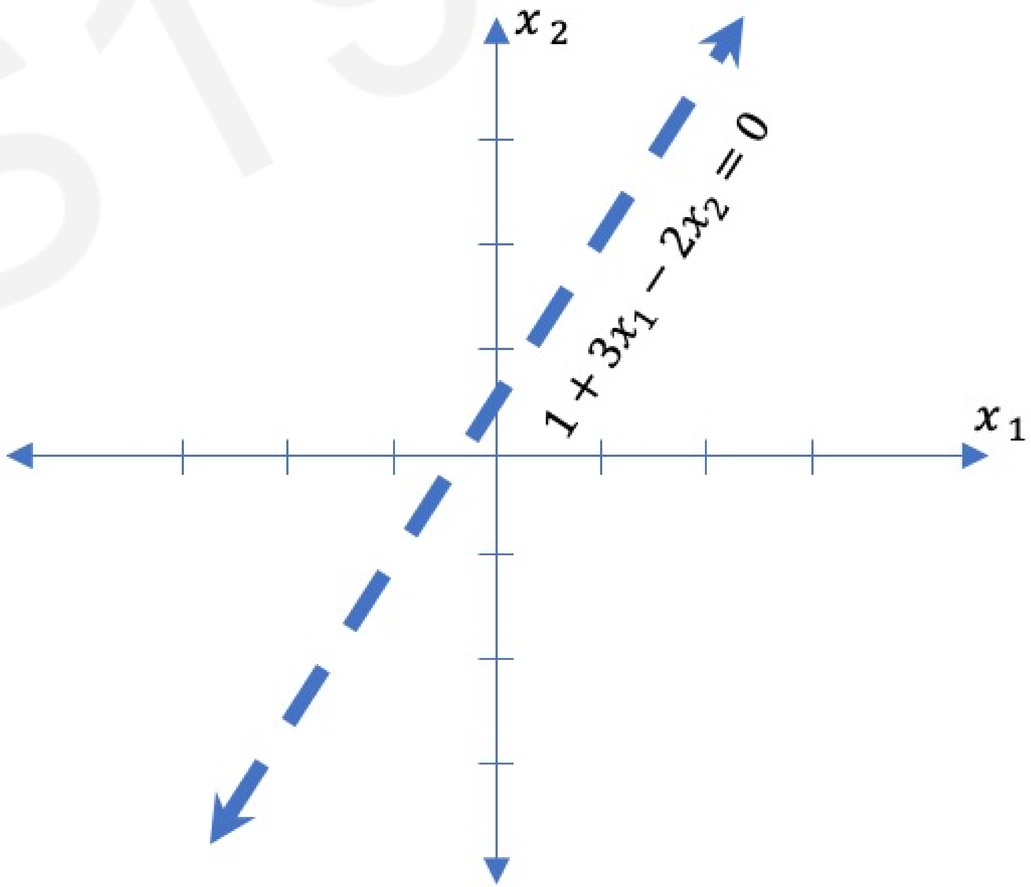
$$\hat{y} = g(\underbrace{1 + 3x_1 - 2x_2}_{\text{This is just a line in 2D!}})$$

MIT 6.S191

# The Perceptron: Example

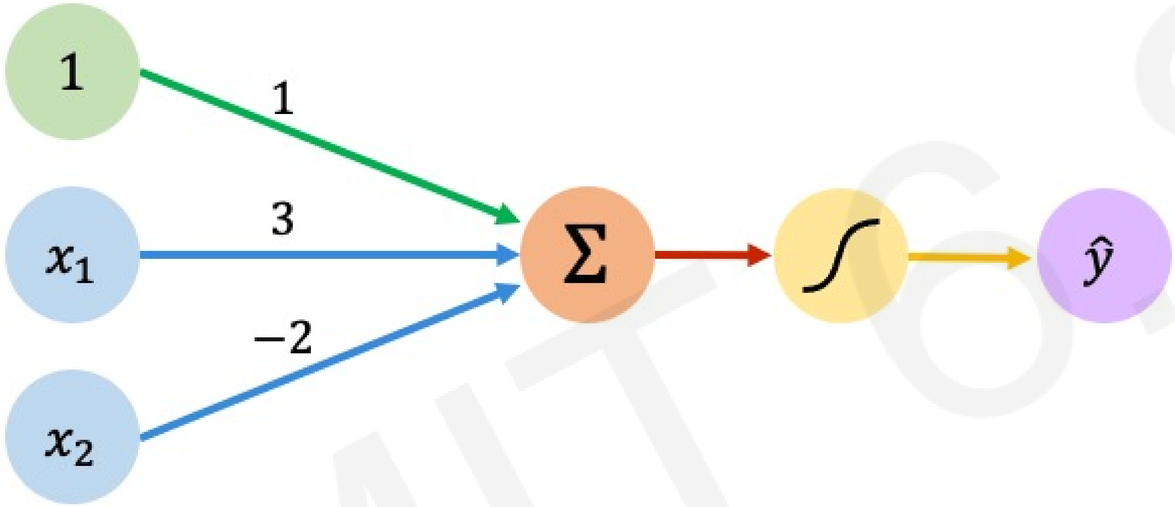
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



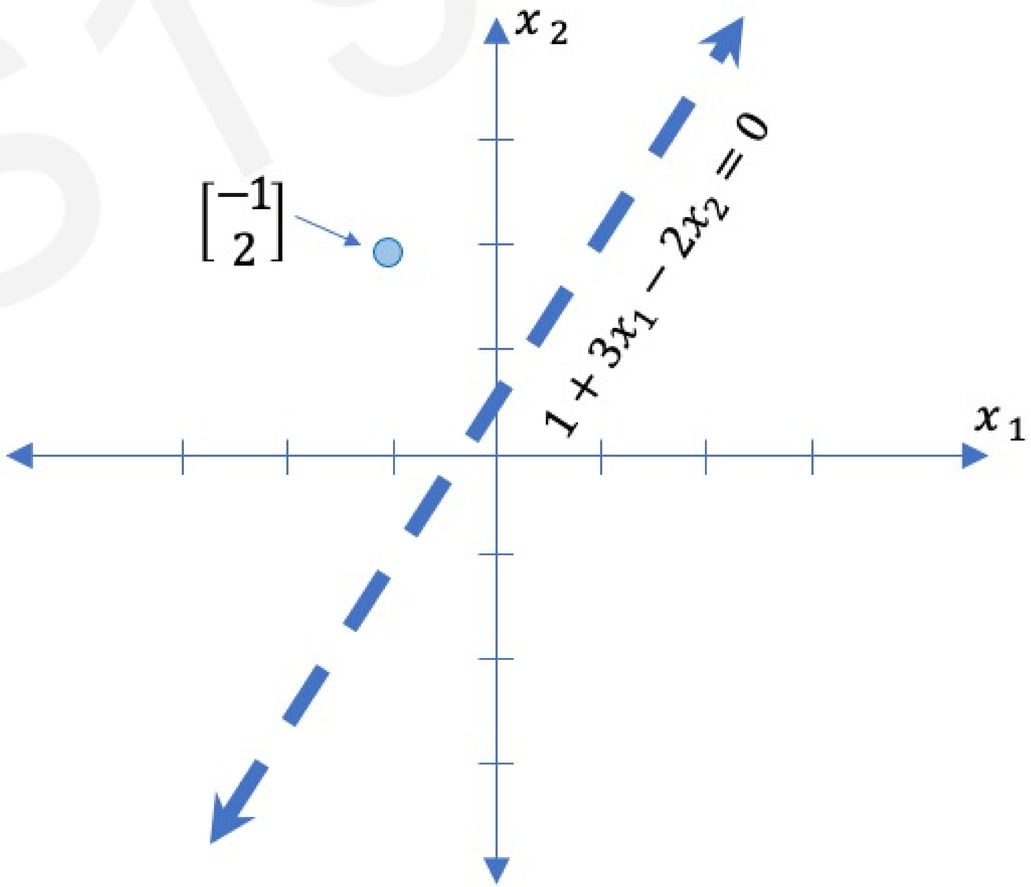




MIT 6.S191

# The Perceptron: Example




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

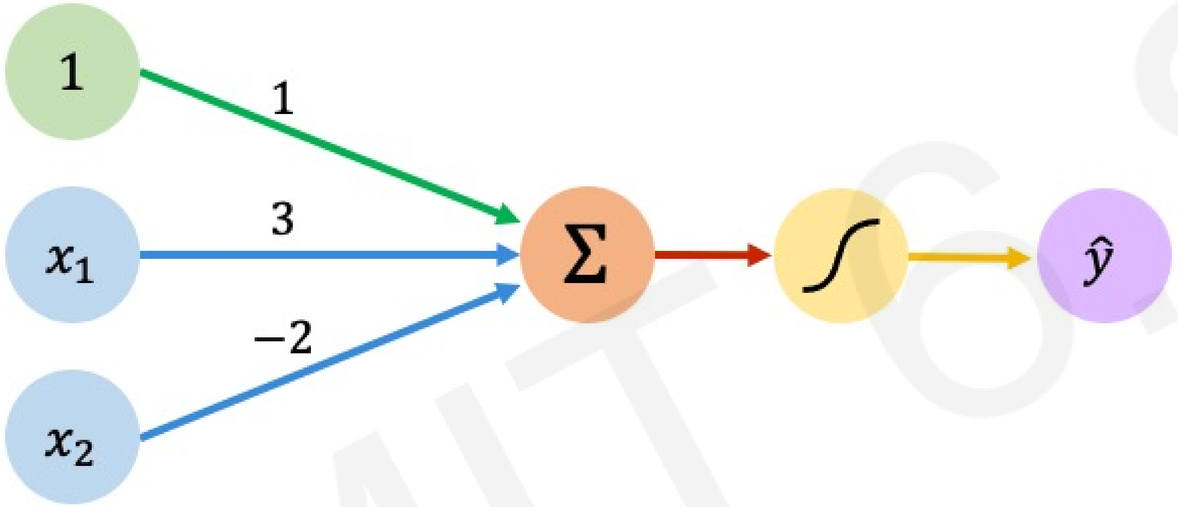



Assume we have input: $\boldsymbol{X} = \begin{bmatrix}-1\\2\end{bmatrix}$

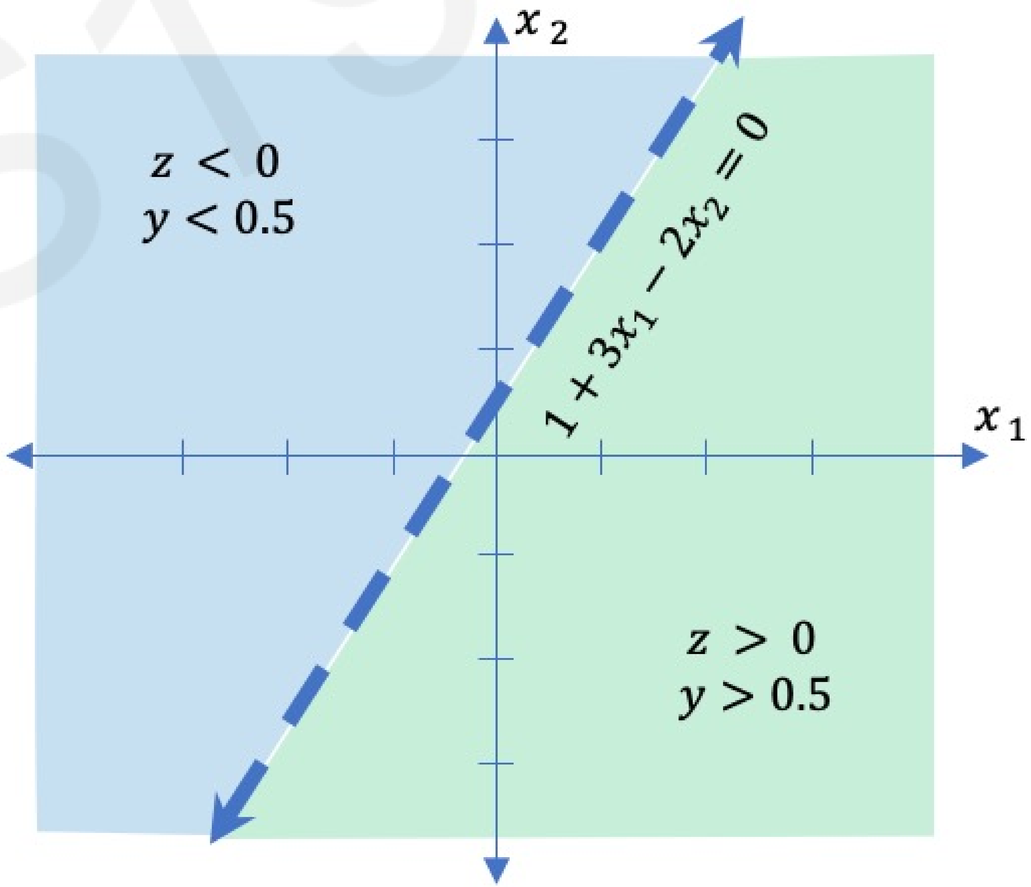
$$\hat{y} = g\big(1 + (3 * -1) - (2 * 2)\big)$$
$$= g(-6) \approx 0.002$$

MIT 6.S191

# The Perceptron: Example




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



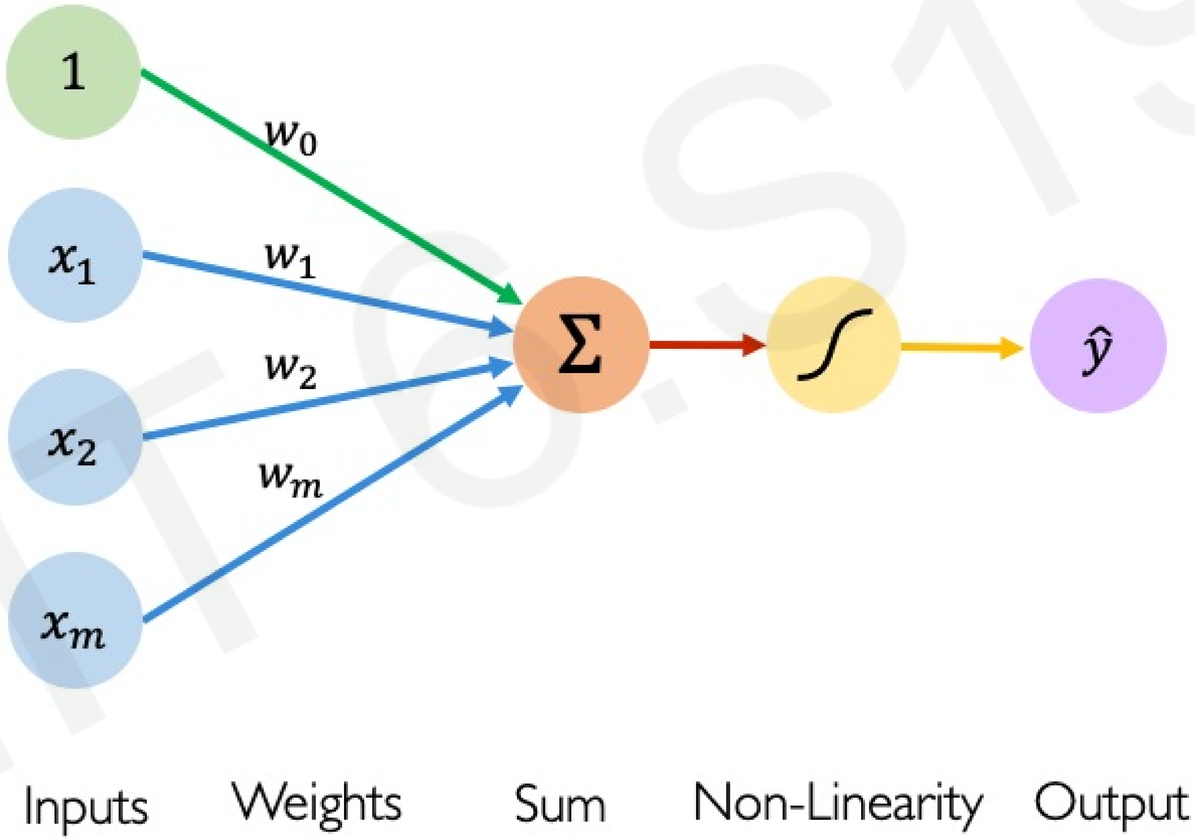

MIT 6.S191

# Building Neural Networks with Perceptrons

MIT 6.S191

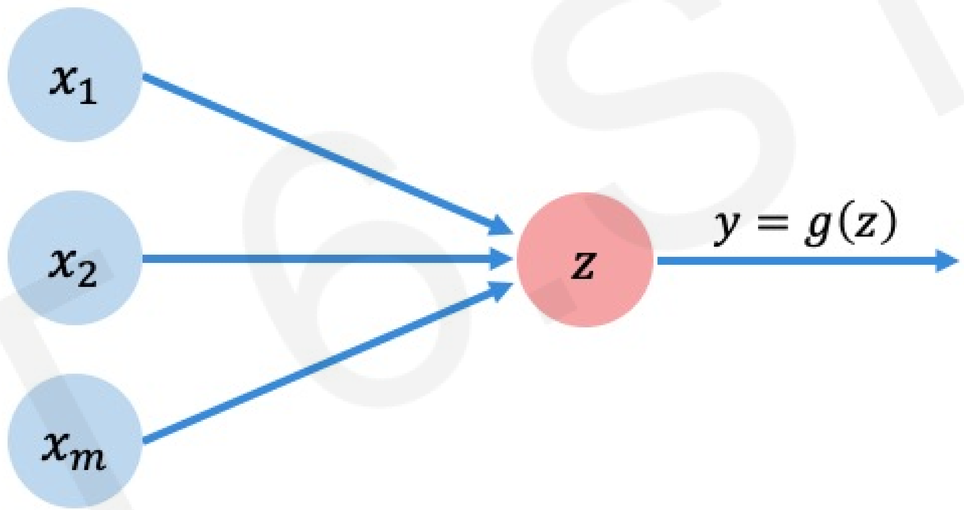
# The Perceptron: Simplified

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




MIT 6.S191

# The Perceptron: Simplified

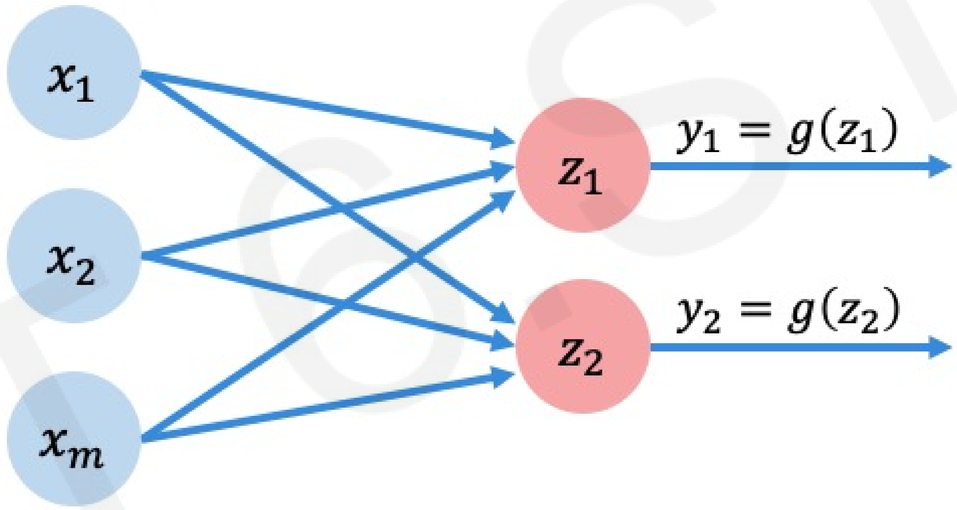



$$z = w_0 + \sum_{j=1}^{m} x_j w_j$$

MIT 6.S191

# Multi Output Perceptron

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




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

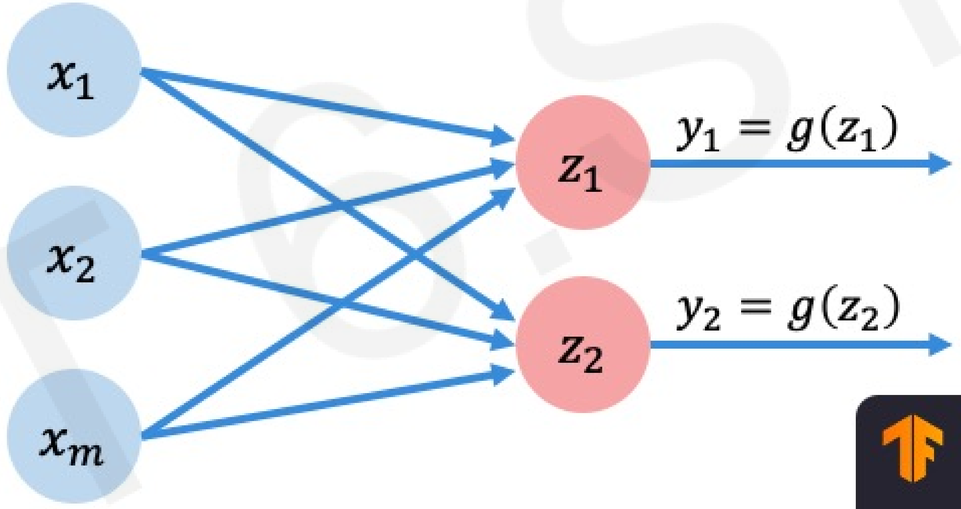
MIT 6.S191

# 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

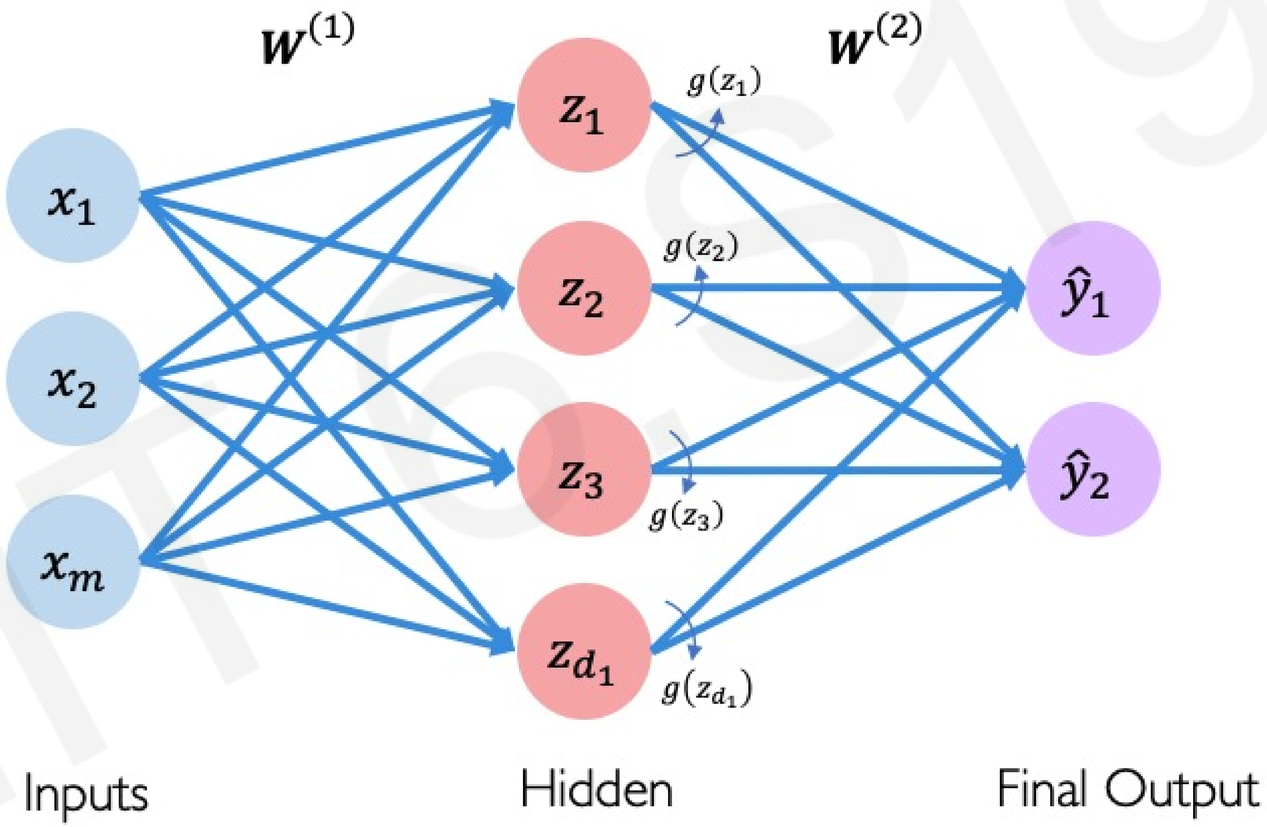



```
import tensorflow as tf

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

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

MIT 6.S191

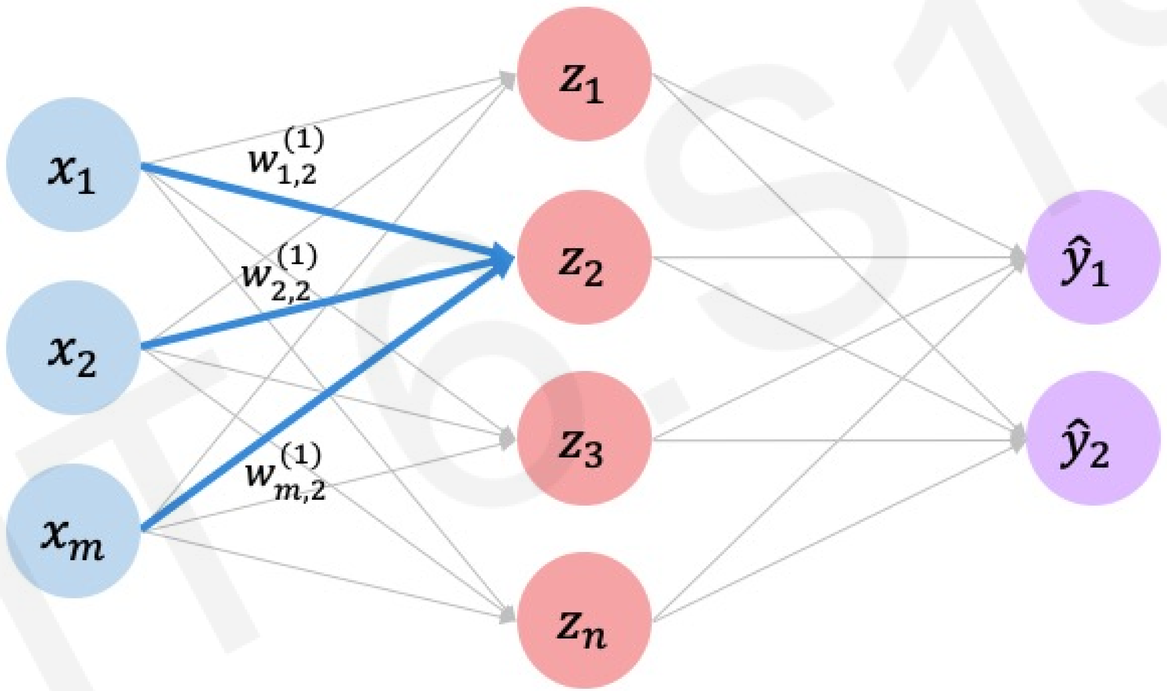
# Single Layer Neural Network




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

MIT 6.S191

# Single Layer Neural Network



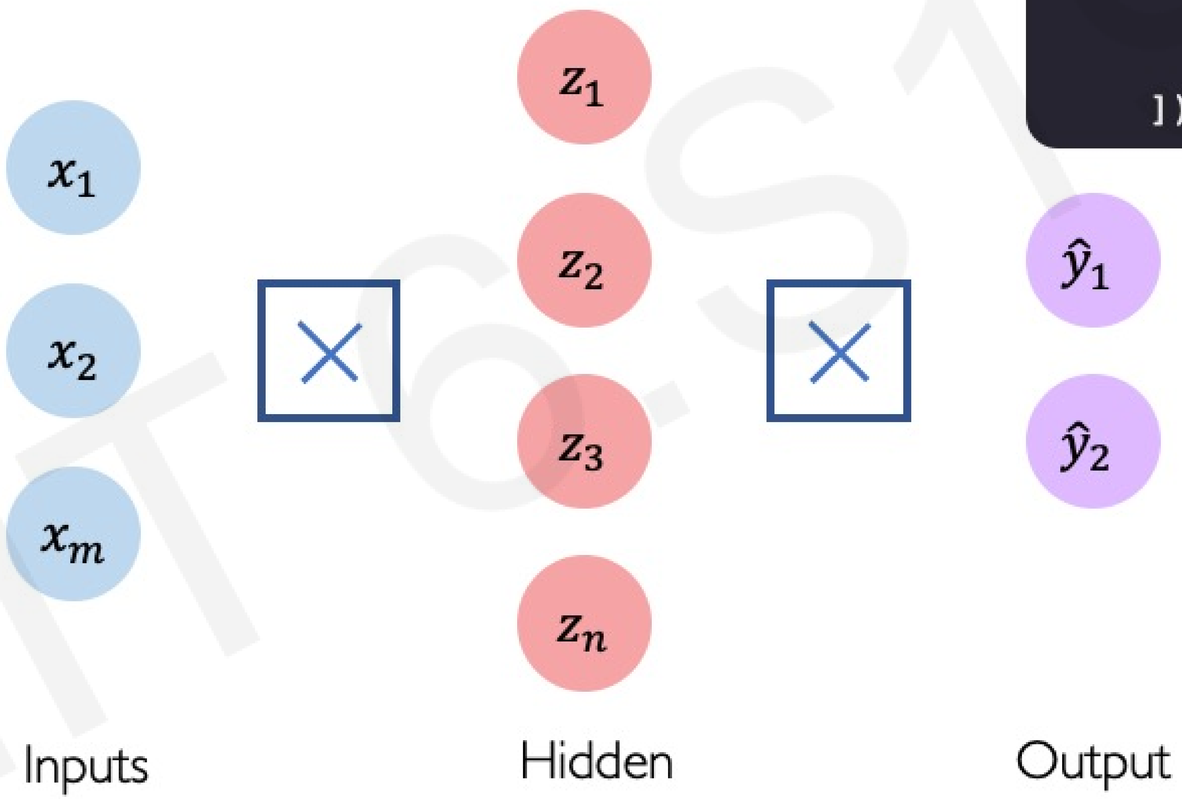

$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,2}^{(1)}$$

$$= w_{0,2}^{(1)} + x_1 \, w_{1,2}^{(1)} + x_2 \, w_{2,2}^{(1)} + x_m \, w_{m,2}^{(1)}$$

MIT 6.S191

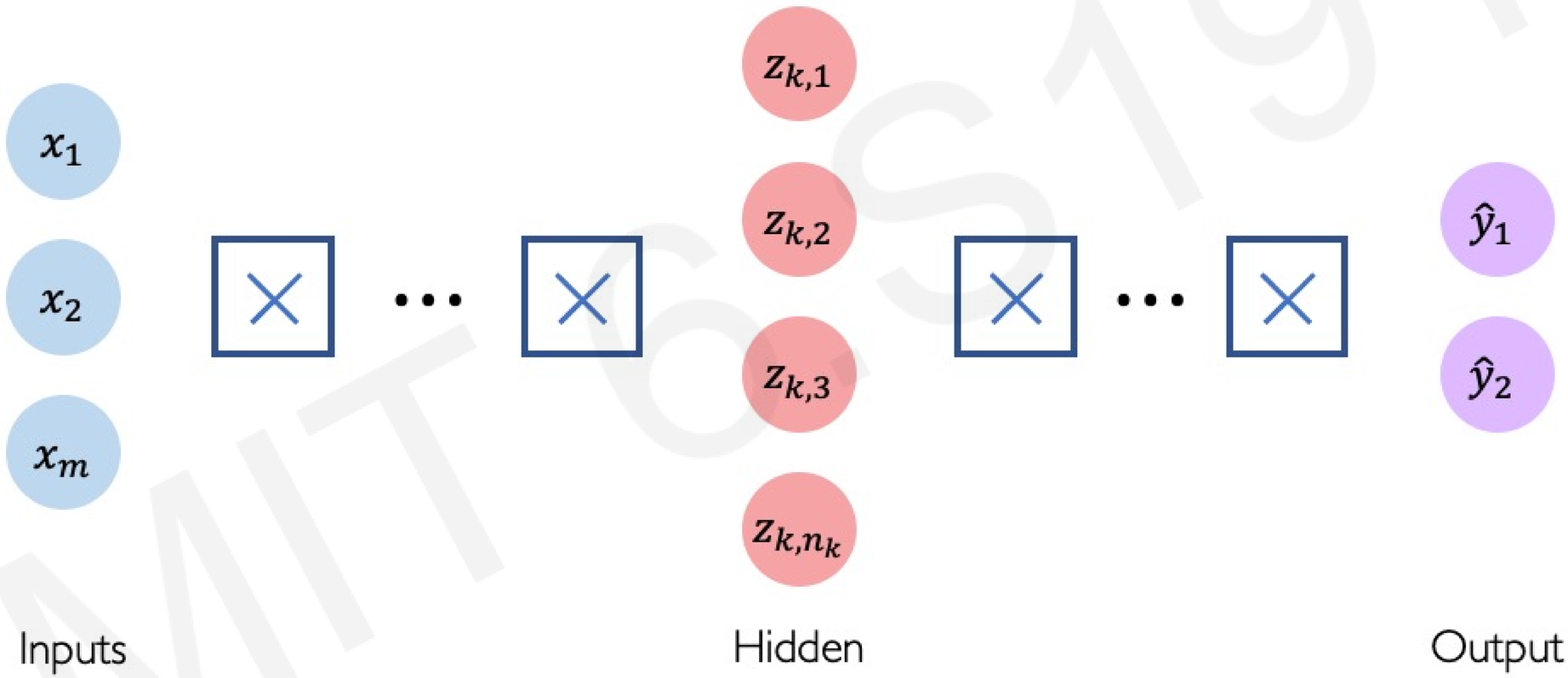
# Multi Output Perceptron

```python
import tensorflow as tf

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




MIT 6.S191

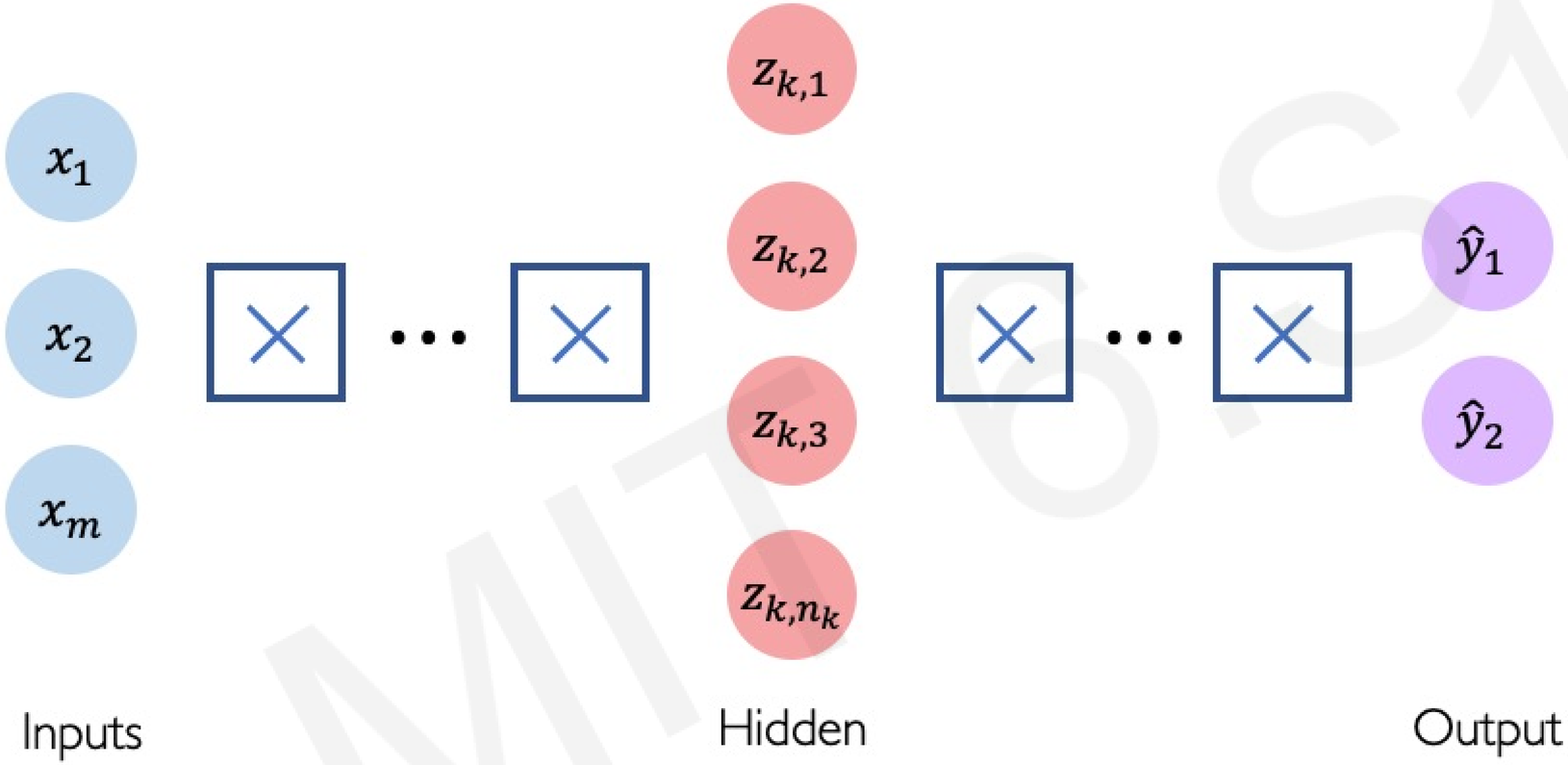
# Deep Neural Network




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

MIT 6S191

# Deep Neural Network




```
import tensorflow as tf

model = tf.keras.Sequential([
  tf.keras.layers.Dense(n1),
  tf.keras.layers.Dense(n2),
  ⋮
  tf.keras.layers.Dense(2)
])
```

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

MIT 6.S191

# Applying Neural Networks

MIT 6.S191

# Example Problem

Will I pass this class?

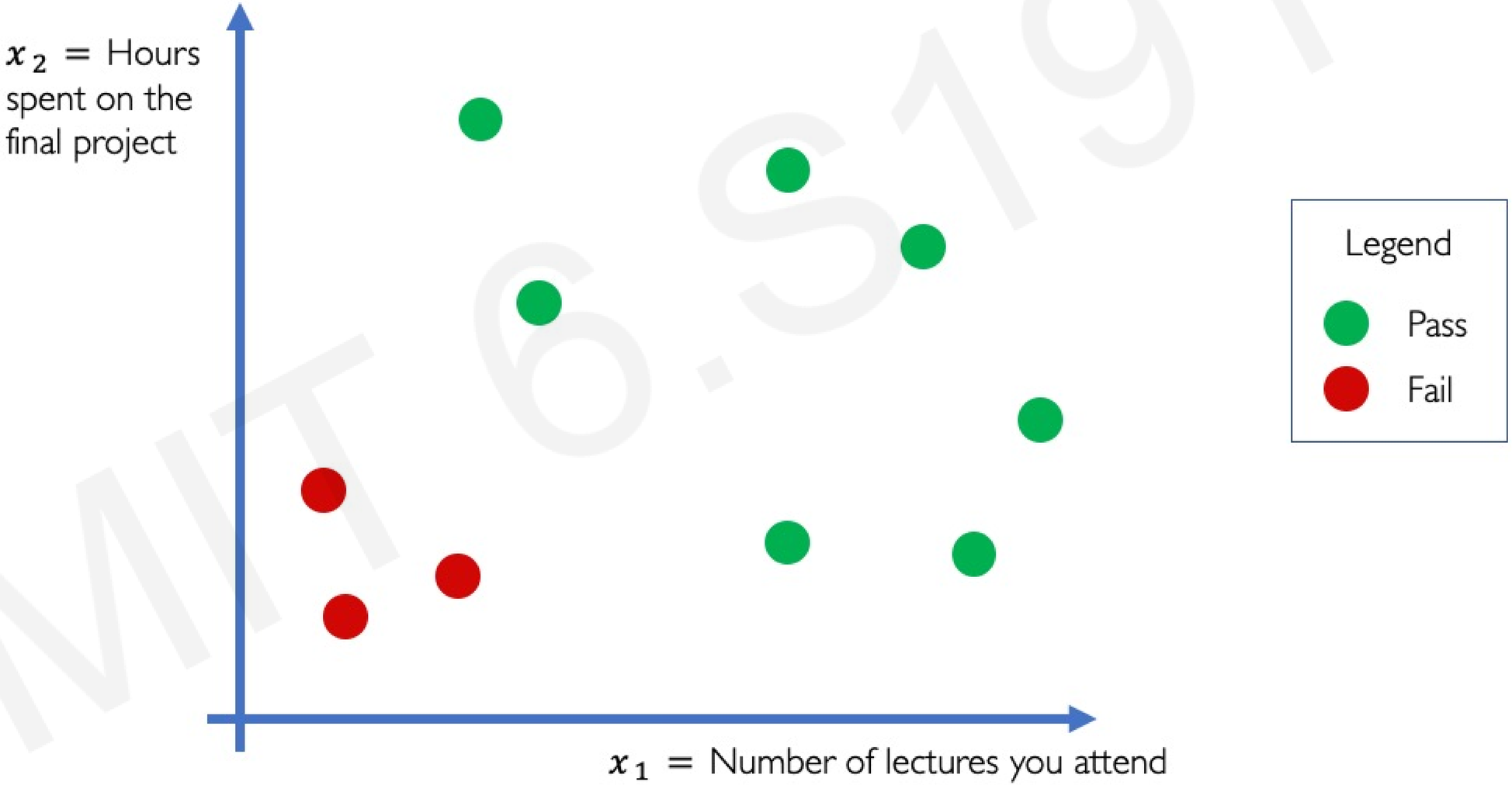
Let's start with a simple two feature model

$x_1$ = Number of lectures you attend

$x_2$ = Hours spent on the final project

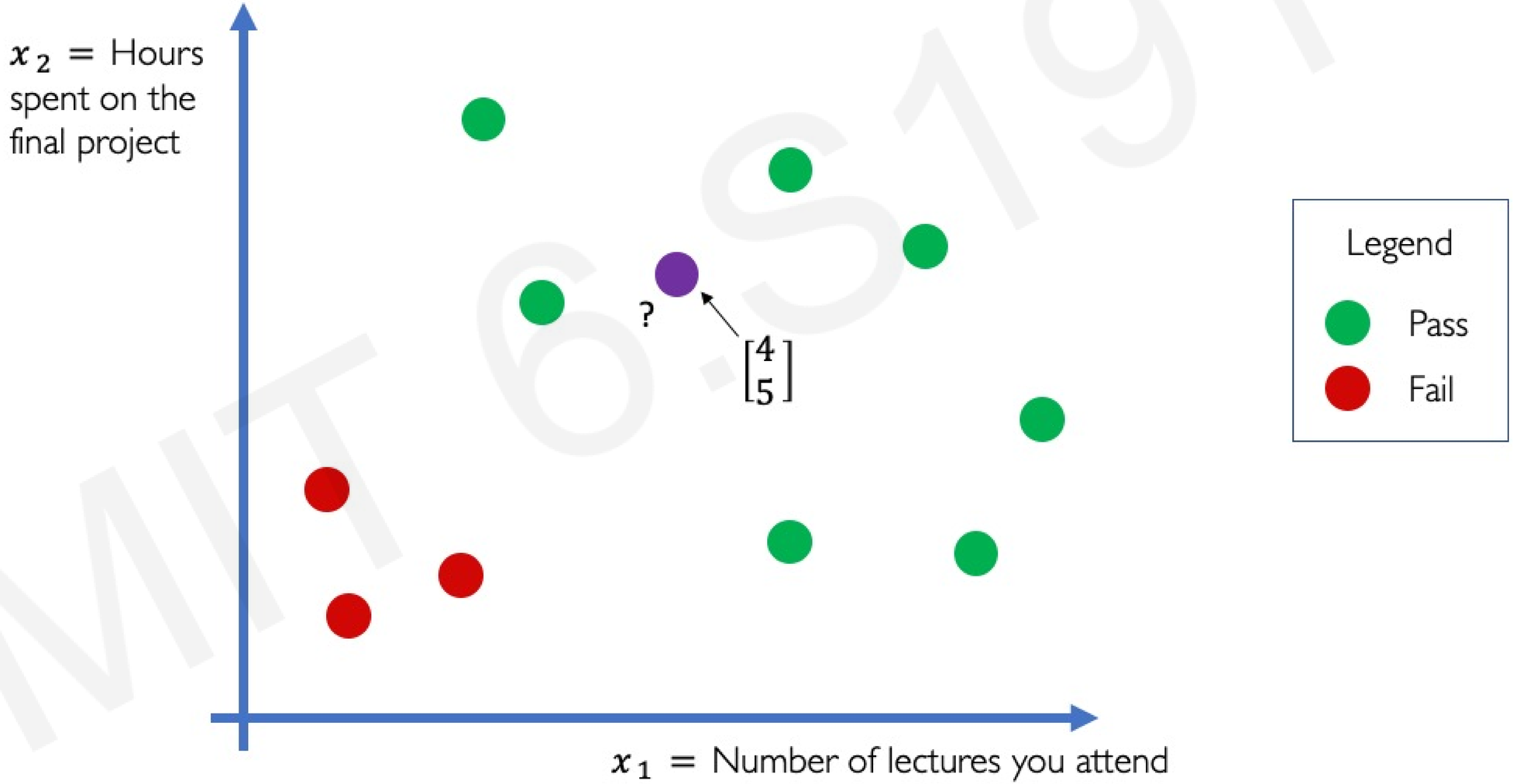
MIT 6.S191

# Example Problem: Will I pass this class?




MIT 6.S191

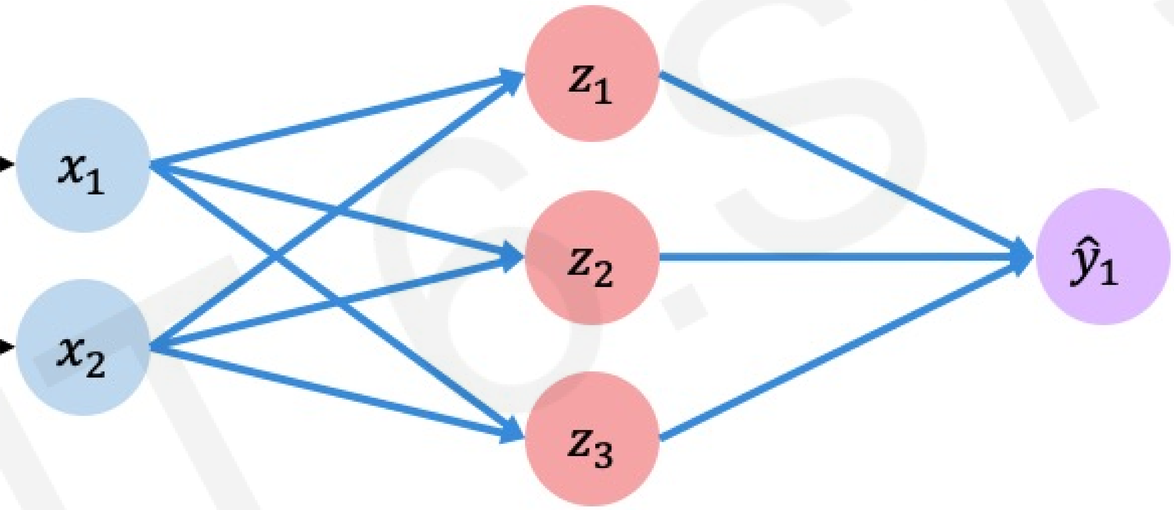
# Example Problem: Will I pass this class?




MIT 6.S191

# Example Problem: Will I pass this class?

$$x^{(1)} = [4\ ,5\,]$$

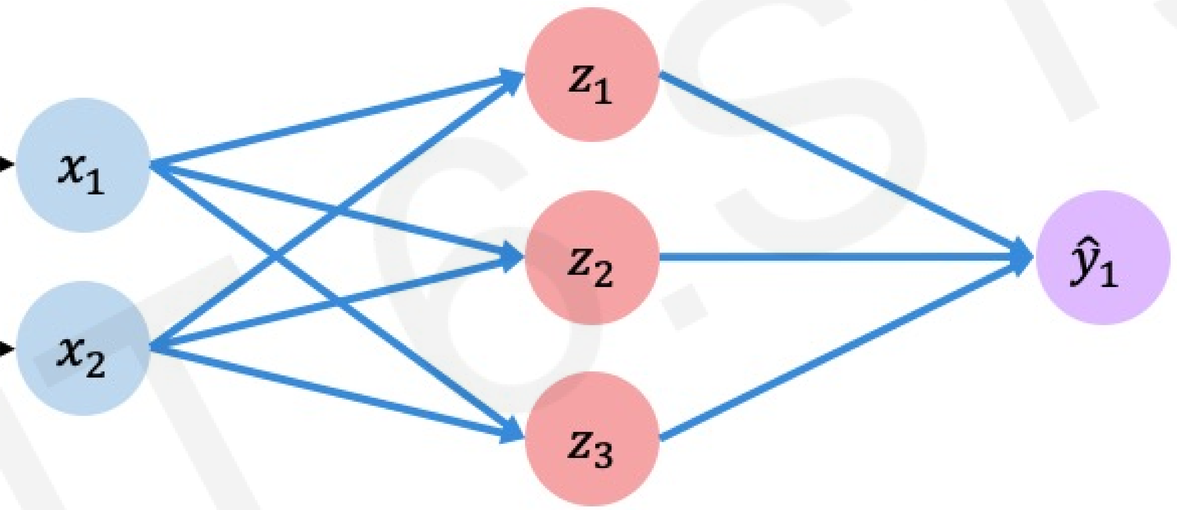



Predicted: **0.1**

MIT6.S191

# Example Problem: Will I pass this class?

$$x^{(1)} = [4\ ,5\,]$$



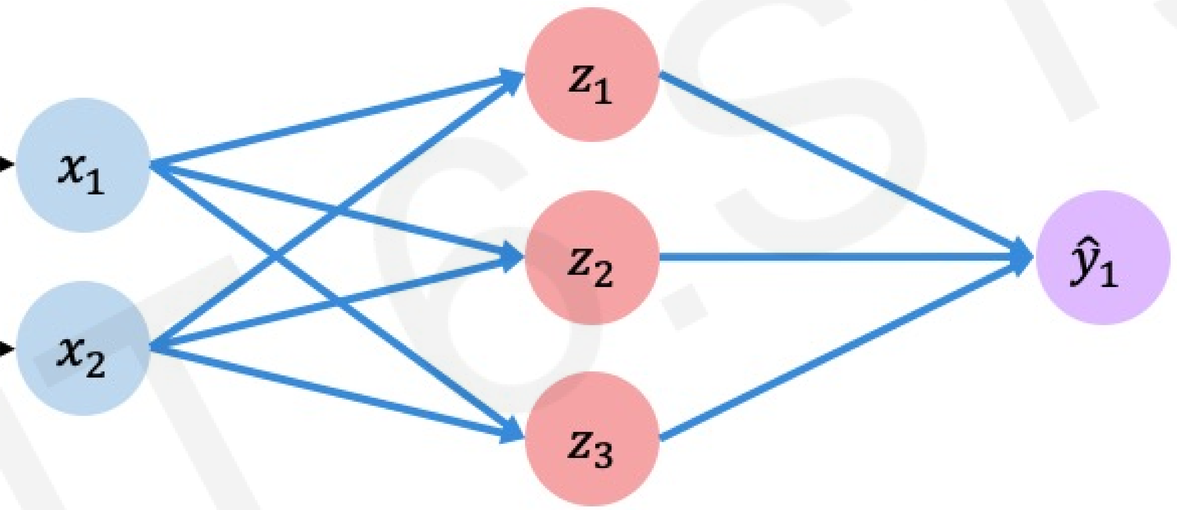

Predicted: 0.1
Actual: 1

MIT6.S191

# Quantifying Loss

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

$$x^{(1)} = [4\ ,5\,]$$




Predicted: **0.1**
Actual: **1**

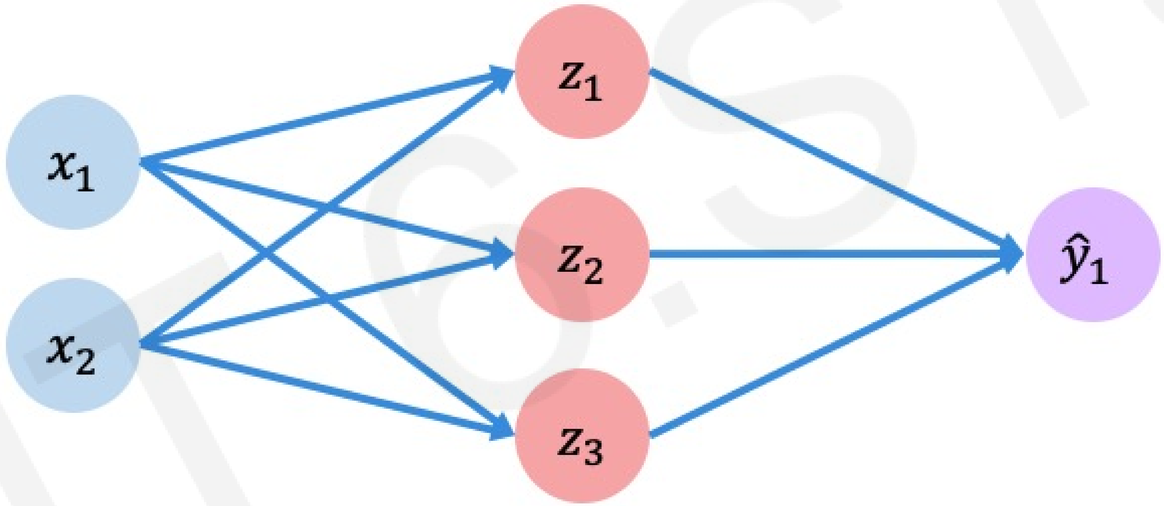
$$\mathcal{L}\left(\underbrace{f\left(x^{(i)}; \boldsymbol{W}\right)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\right)$$

MIT 6.S191

# Empirical Loss

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

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




$$\begin{array}{ccc} f(x) & & y \\ \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} & \begin{matrix} \times \\ \times \\ \checkmark \\ \\ \end{matrix} & \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix} \end{array}$$

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

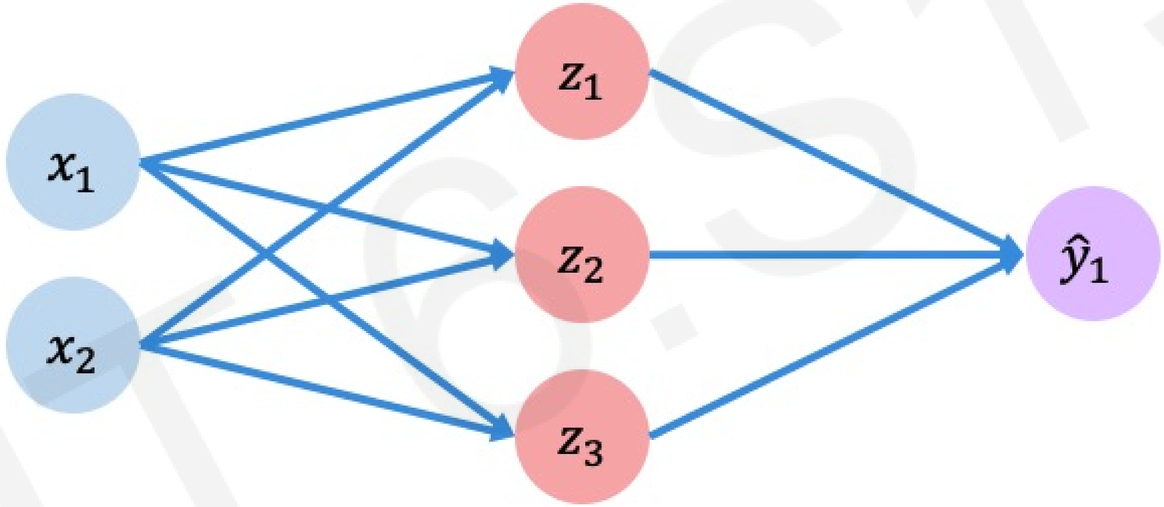
Also known as:
- Objective function
- Cost function
- Empirical Risk

MIT6.S191

# Binary Cross Entropy Loss

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

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




$$\begin{array}{ccc} f(x) & & y \\ \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} & \begin{matrix} \times \\ \times \\ \checkmark \\ \phantom{\vdots} \end{matrix} & \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix} \end{array}$$

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

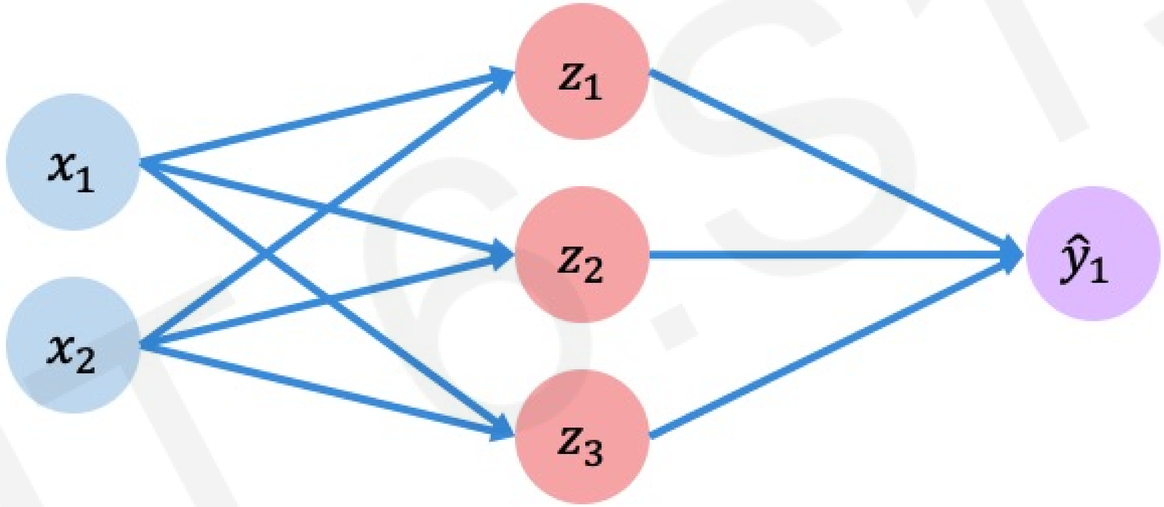
```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```

MIT 6.S191

# Mean Squared Error Loss

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

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




$$\begin{array}{ccc} f(x) & & y \\ \begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix} & \begin{matrix} \times \\ \times \\ \checkmark \\ \\ \end{matrix} & \begin{bmatrix} 90 \\ 20 \\ 95 \\ \vdots \end{bmatrix} \end{array}$$

Final Grades (percentage)

$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\left(\underbrace{y^{(i)}}_{\text{Actual}} - \underbrace{f\left(x^{(i)};\boldsymbol{W}\right)}_{\text{Predicted}}\right)^2$$

```
loss = tf.reduce_mean( tf.square(tf.subtract(y, predicted)) )
loss = tf.keras.losses.MSE( y, predicted )
```

MIT 6.S191

# Training Neural Networks

MIT 6.S191

# Loss Optimization

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

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

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

MIT 6.S191

# Loss Optimization

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

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

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

Remember:

$$\boldsymbol{W} = \left\{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}, \dots\right\}$$

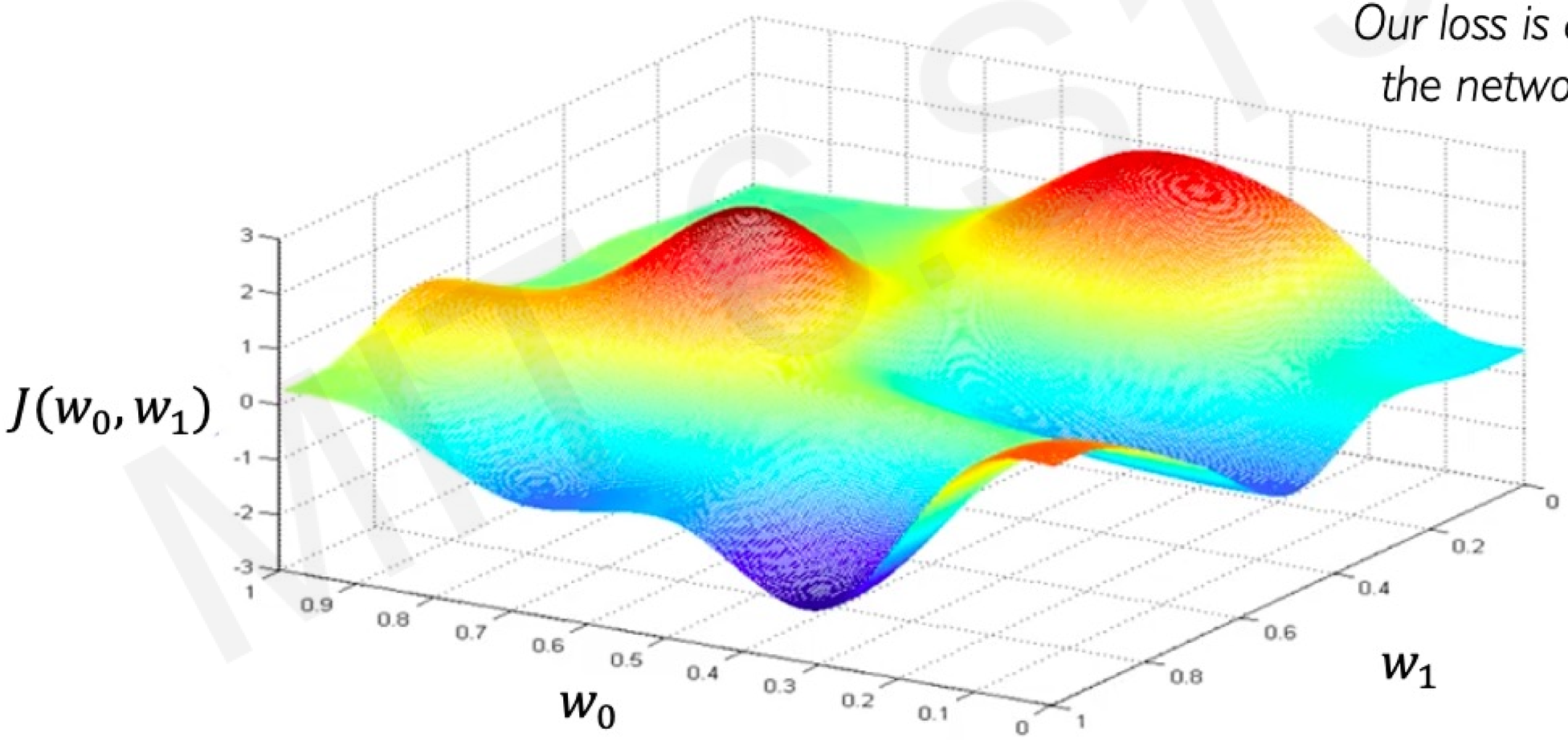
MIT 6.S191

# Loss Optimization

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

Remember:

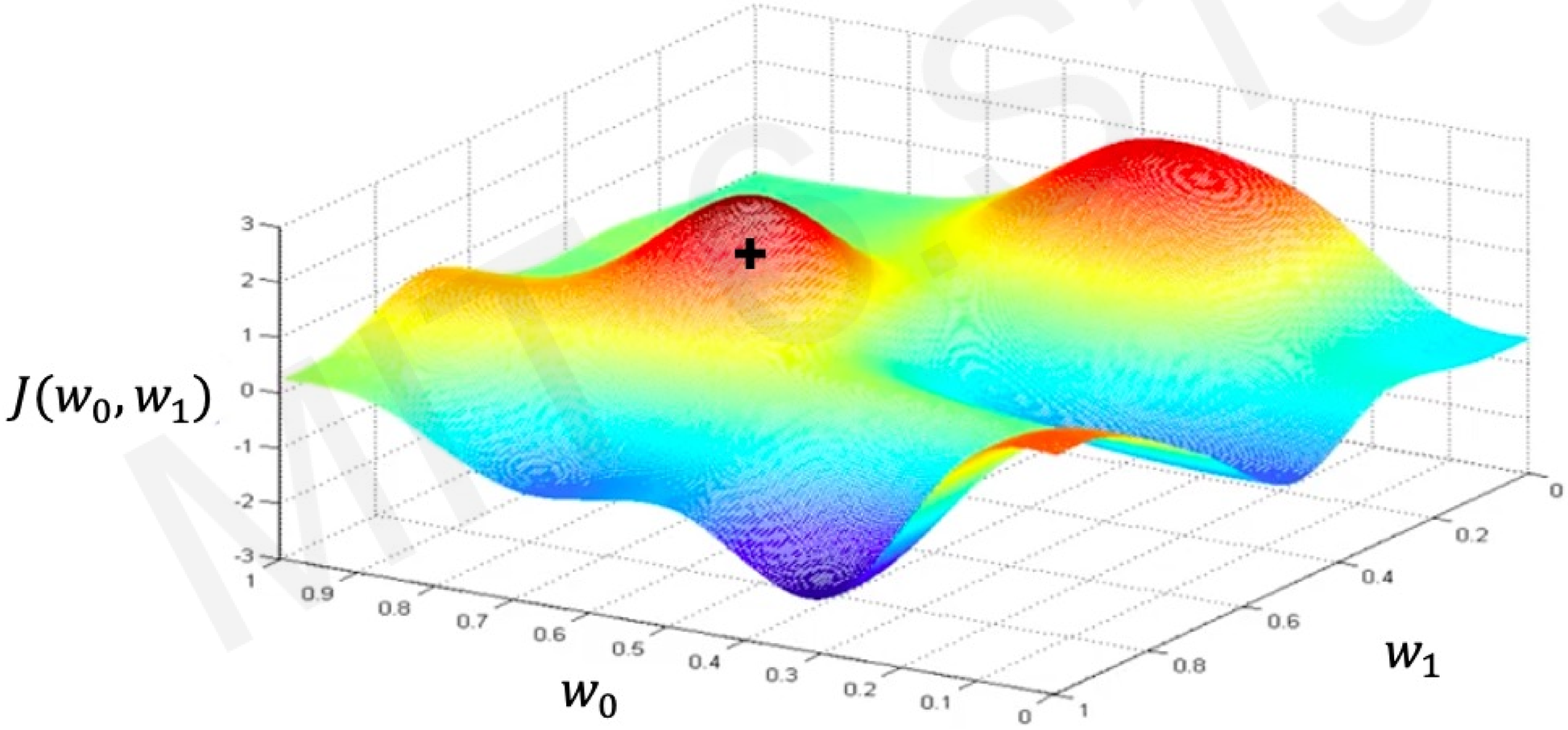
*Our loss is a function of the network weights!*




MIT 6.S191

# Loss Optimization

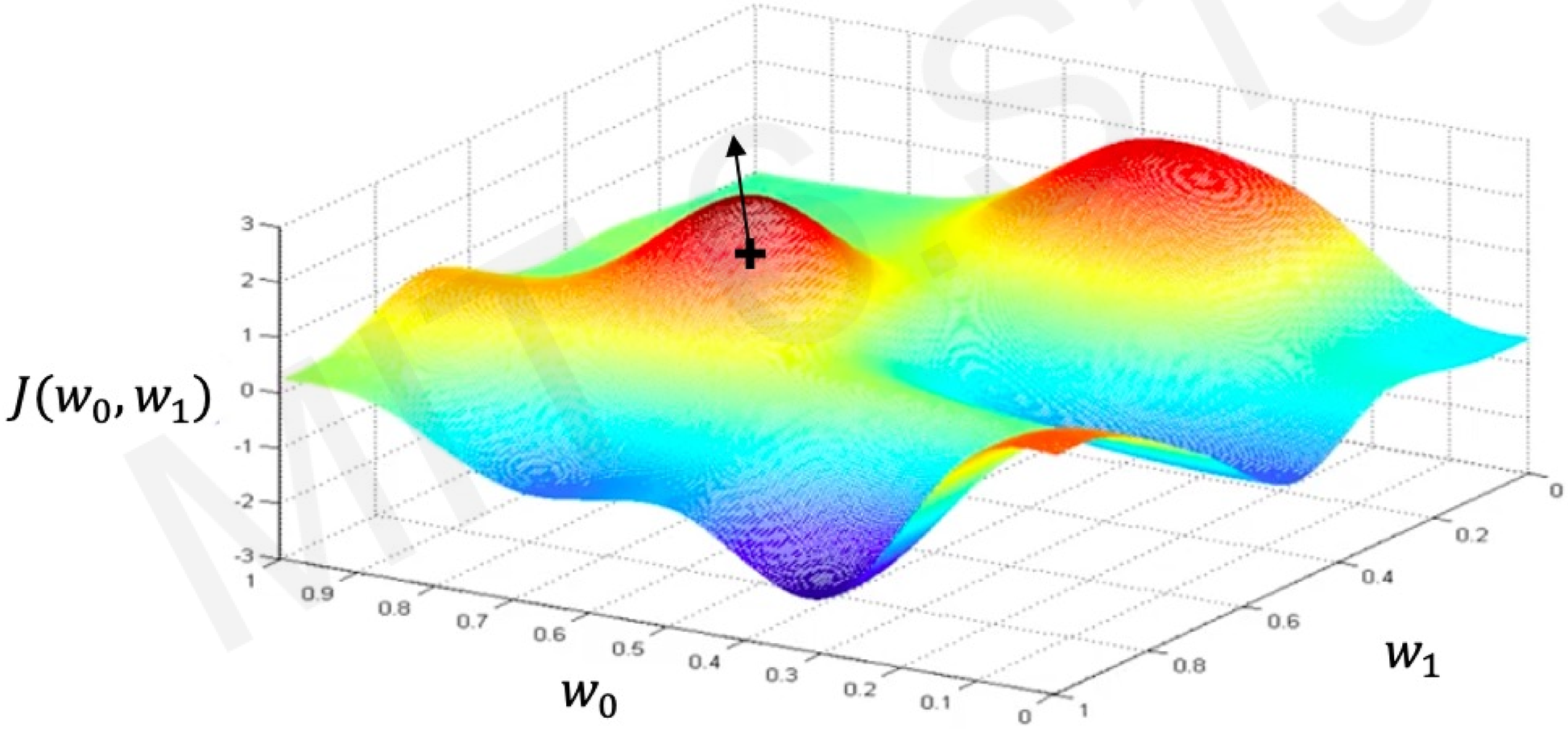
Randomly pick an initial $(w_0, w_1)$




MIT 6.S191

# Loss Optimization

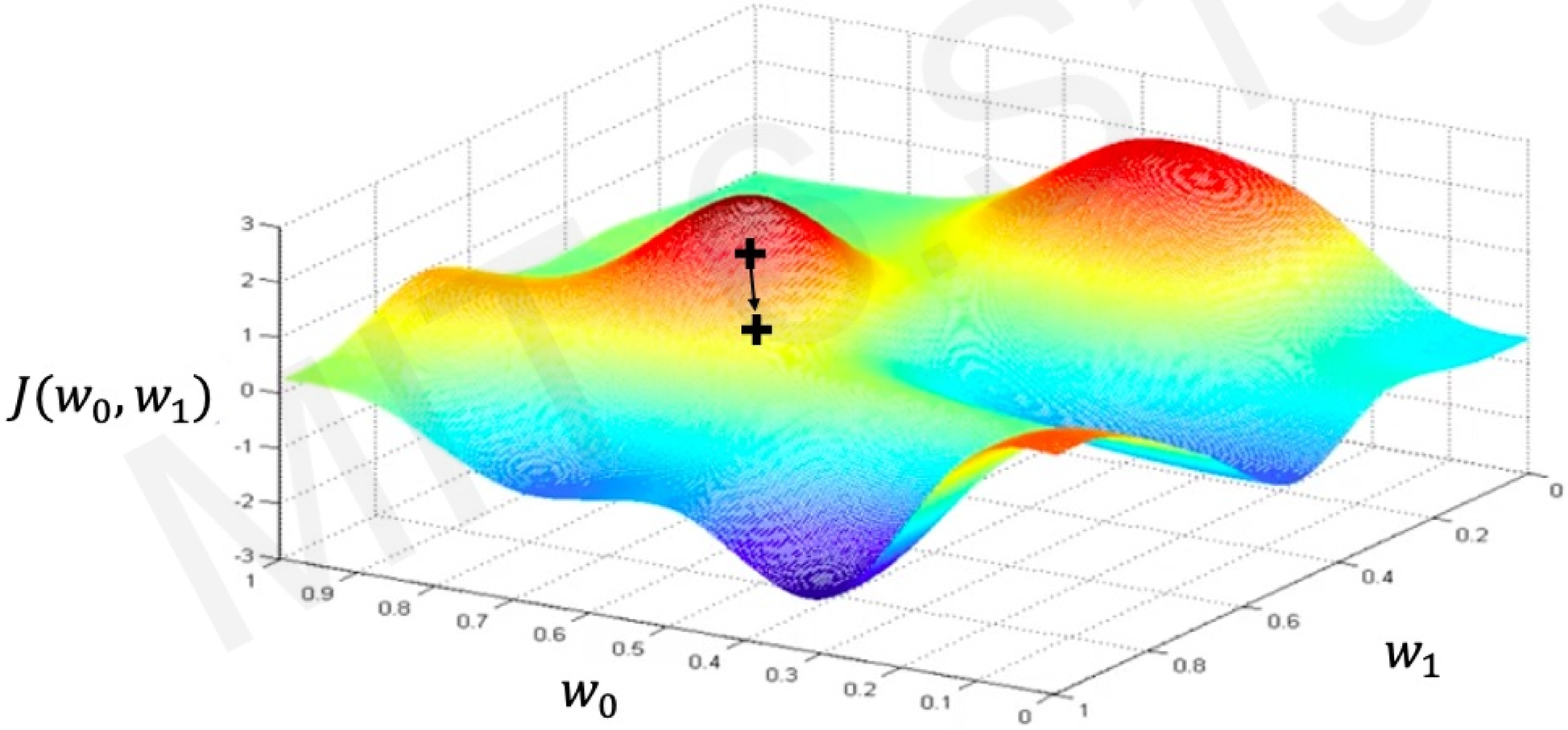
Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$




MIT 6.S191

# Loss Optimization

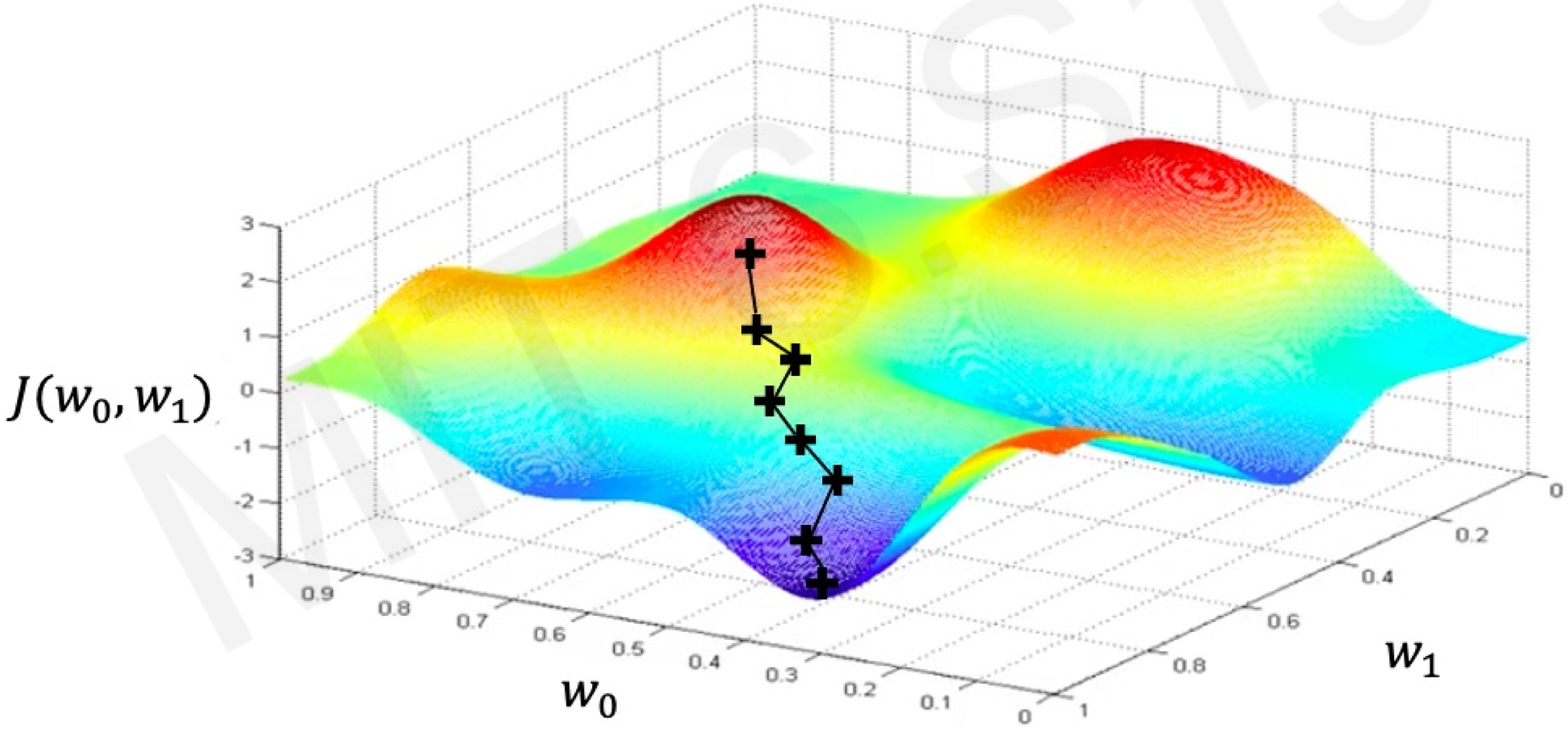
Take small step in opposite direction of gradient




MIT 6.S191

# Gradient Descent

Repeat until convergence




MIT 6.S191

# Gradient Descent

**Algorithm**

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

MIT 6.S191

# Gradient Descent

**Algorithm**

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

# Gradient Descent

**Algorithm**

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
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. $\boldsymbol{w_2}$) affect the final loss $\boldsymbol{J(W)}$?*

MIT 6.S191

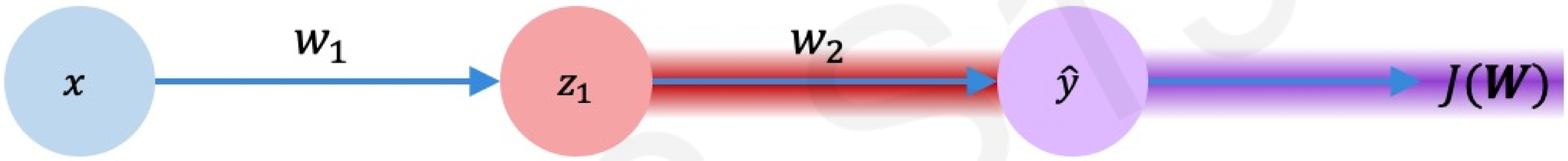
# Computing Gradients: Backpropagation

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

Let's use the chain rule!

MIT 6.S191

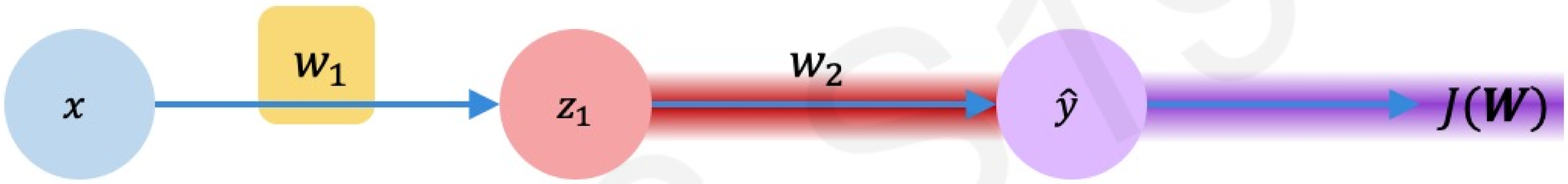
# Computing Gradients: Backpropagation




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

MIT 6.S191

# Computing Gradients: Backpropagation



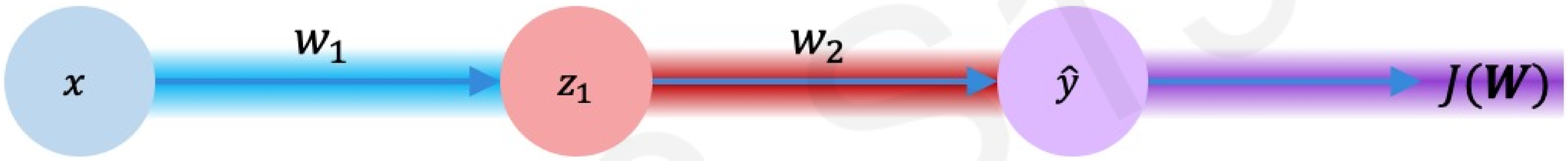

$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

Apply chain rule!

MIT 6.S191

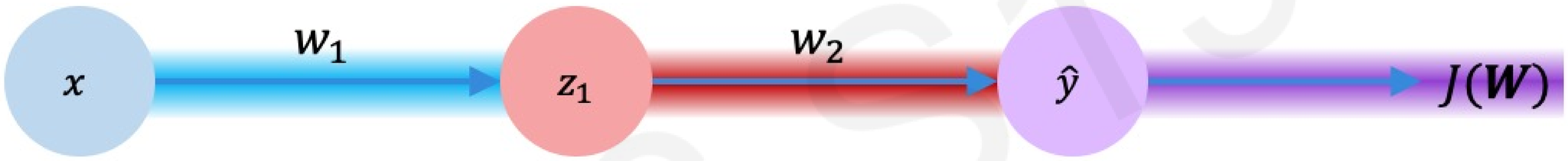
# Computing Gradients: Backpropagation




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

MIT 6.S191

# Computing Gradients: Backpropagation




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

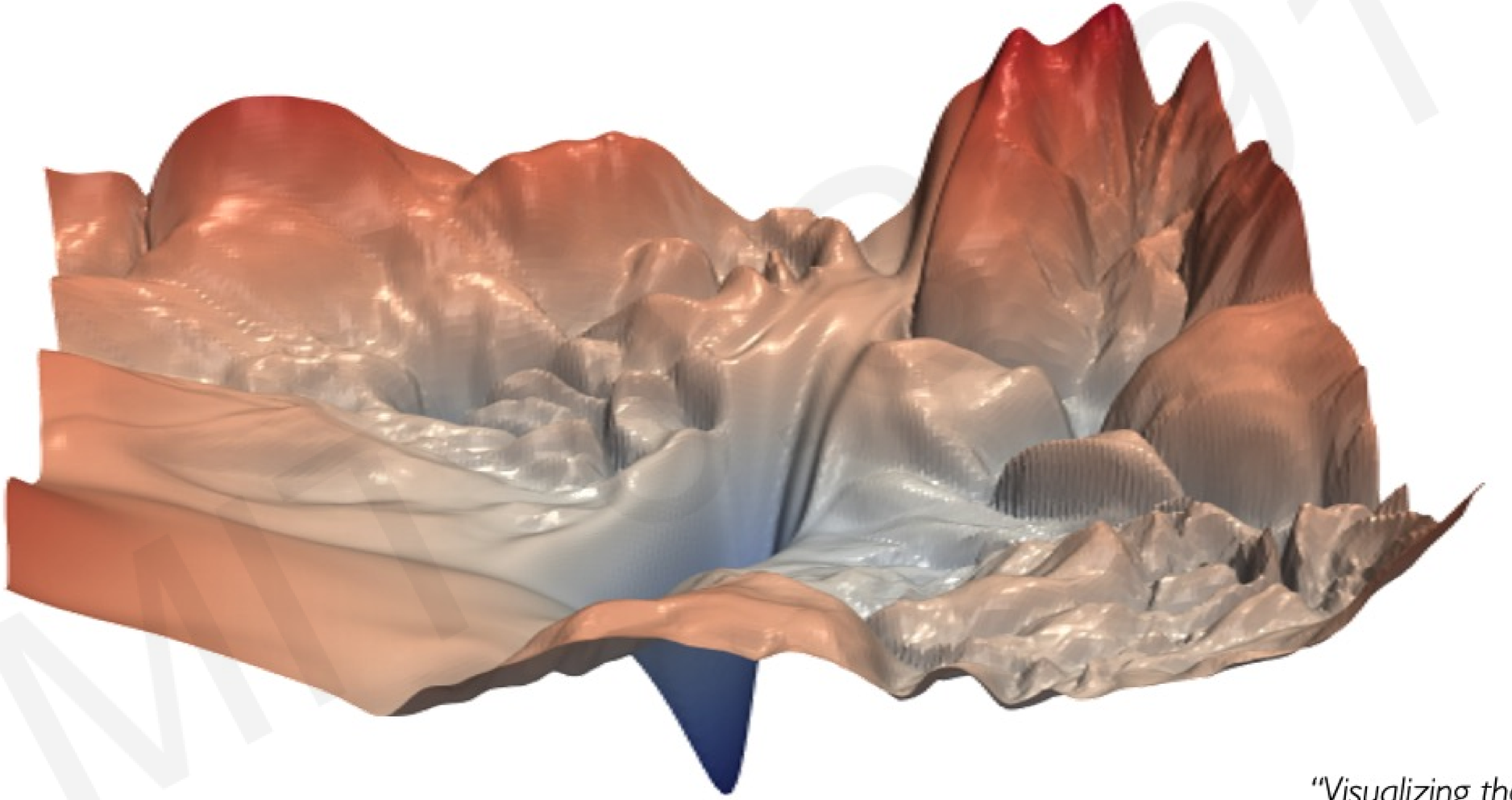
*Repeat this for* ***every weight in the network*** *using gradients from later layers*

MIT 6.S191

# Neural Networks in Practice: Optimization

MIT 6.S191

# 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

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

MIT 6.S191

# Loss Functions Can Be Difficult to Optimize

**Remember:**

Optimization through gradient descent

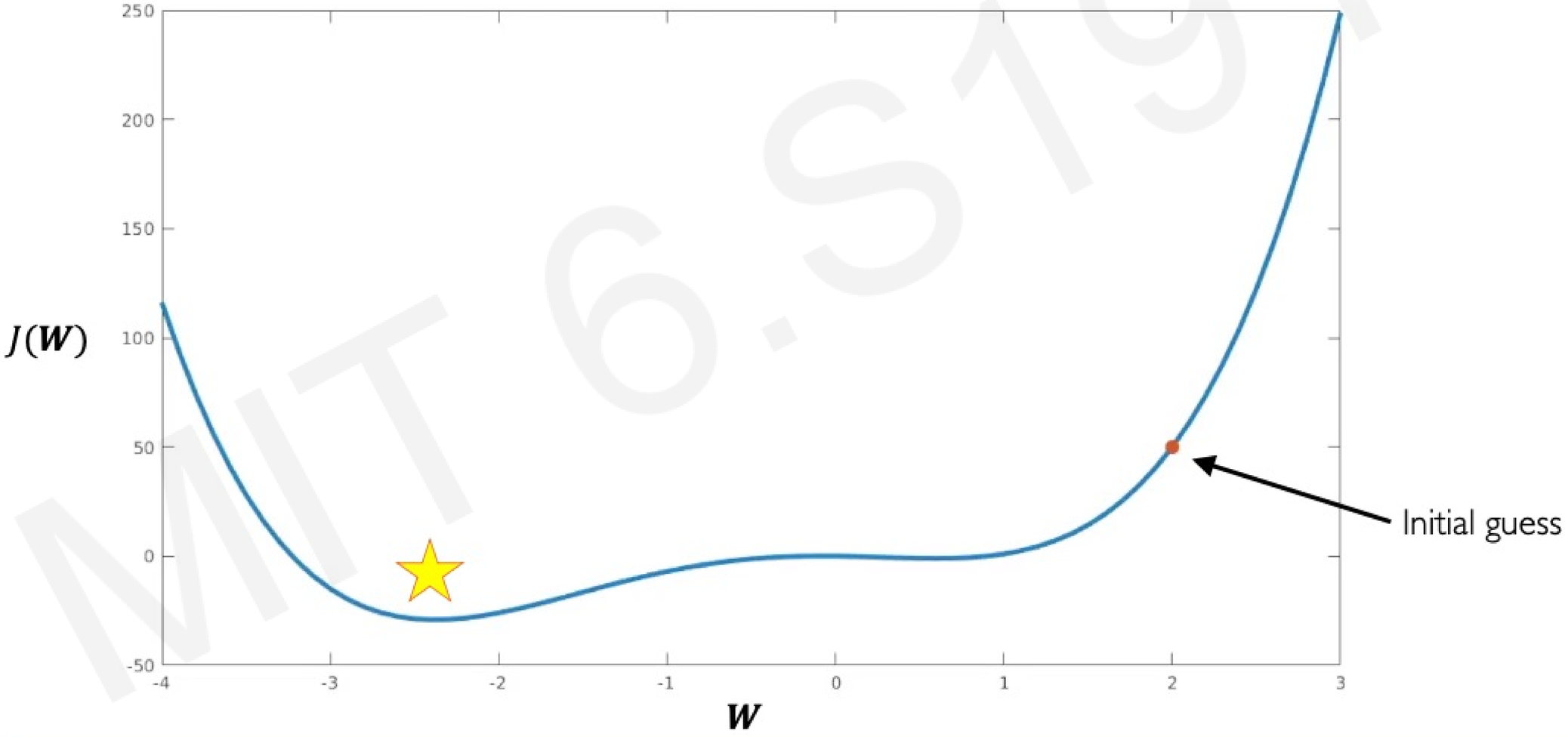
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

How can we set the learning rate?

MIT 6.S191

# Setting the Learning Rate

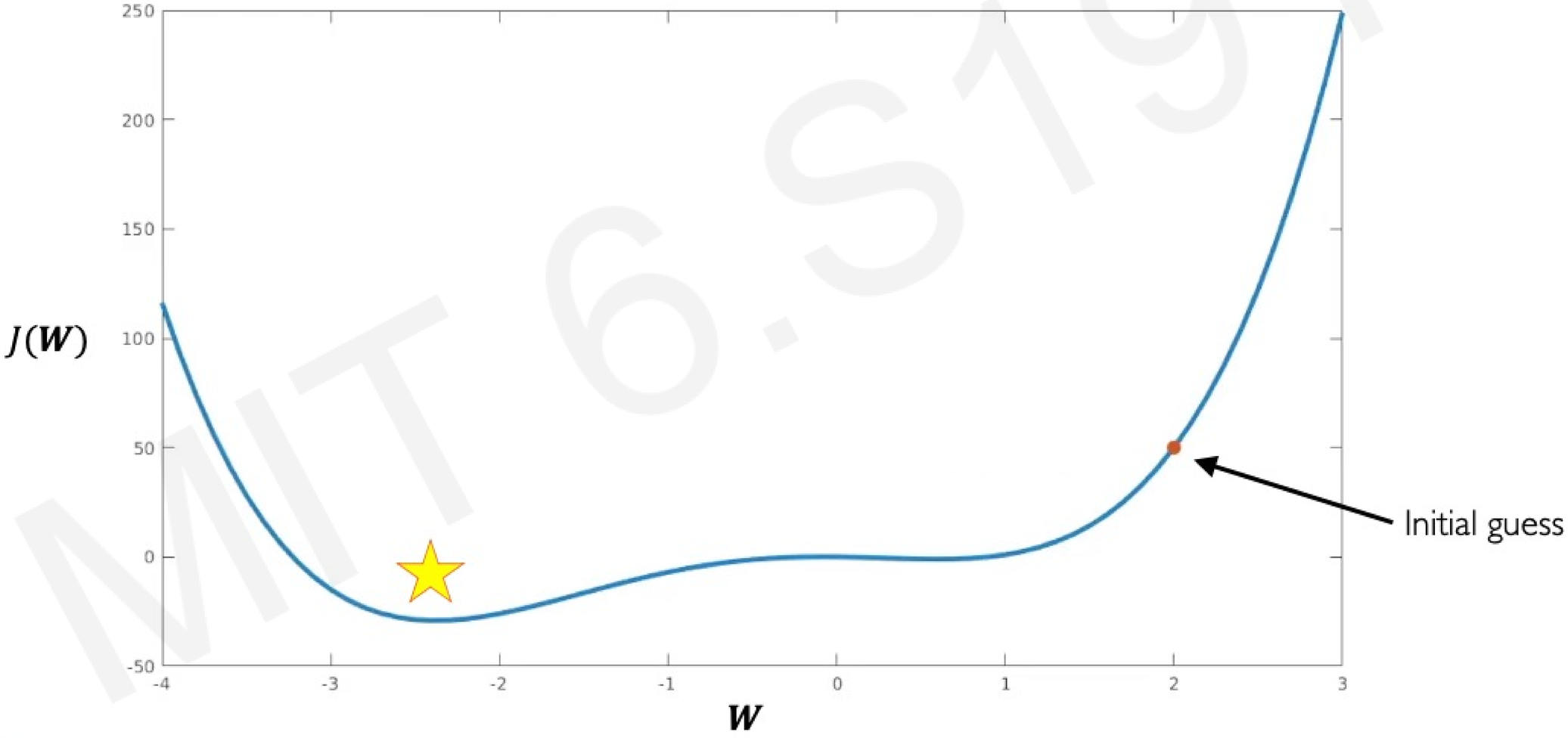
***Small learning rate*** *converges slowly and gets stuck in false local minima*




MIT 6.S191

# Setting the Learning Rate

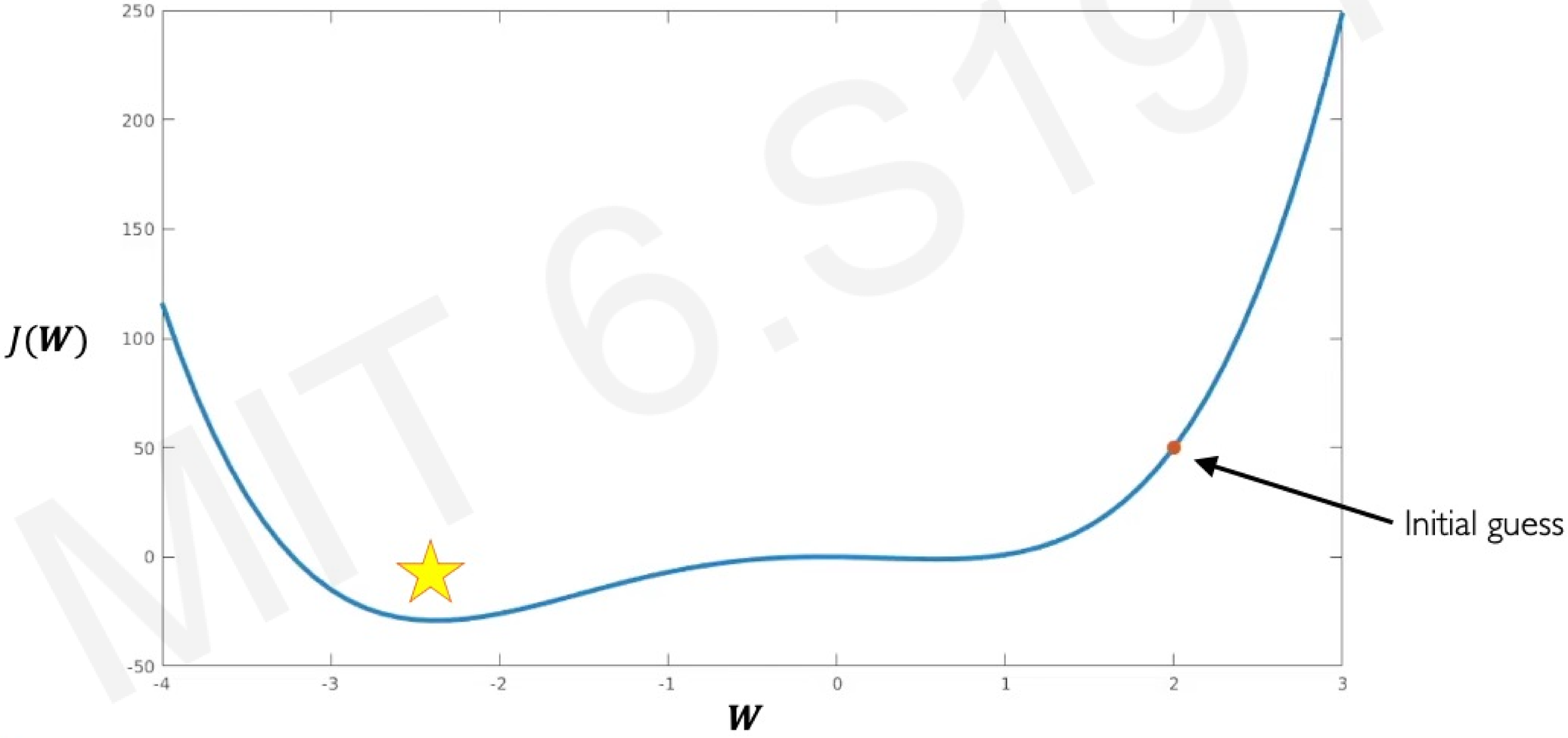
***Large learning rates*** *overshoot, become unstable and diverge*




MIT 6.S191

# Setting the Learning Rate

***Stable learning rates*** *converge smoothly and avoid local minima*




MIT 6.S191

# How to deal with this?

**Idea 1:**

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

MIT 6.S191

# 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

MIT 6.S191

# 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...

MIT 6.S191

# Gradient Descent Algorithms

| Algorithm | TF Implementation | Reference |
|---|---|---|
| • SGD | `tf.keras.optimizers.SGD` | Kiefer & Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function." 1952. |
| • Adam | `tf.keras.optimizers.Adam` | Kingma et al. "Adam: A Method for Stochastic Optimization." 2014. |
| • Adadelta | `tf.keras.optimizers.Adadelta` | Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012. |
| • Adagrad | `tf.keras.optimizers.Adagrad` | Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011. |
| • RMSProp | `tf.keras.optimizers.RMSProp` | |

Additional details: http://ruder.io/optimizing-gradient-descent/

# 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)))
```

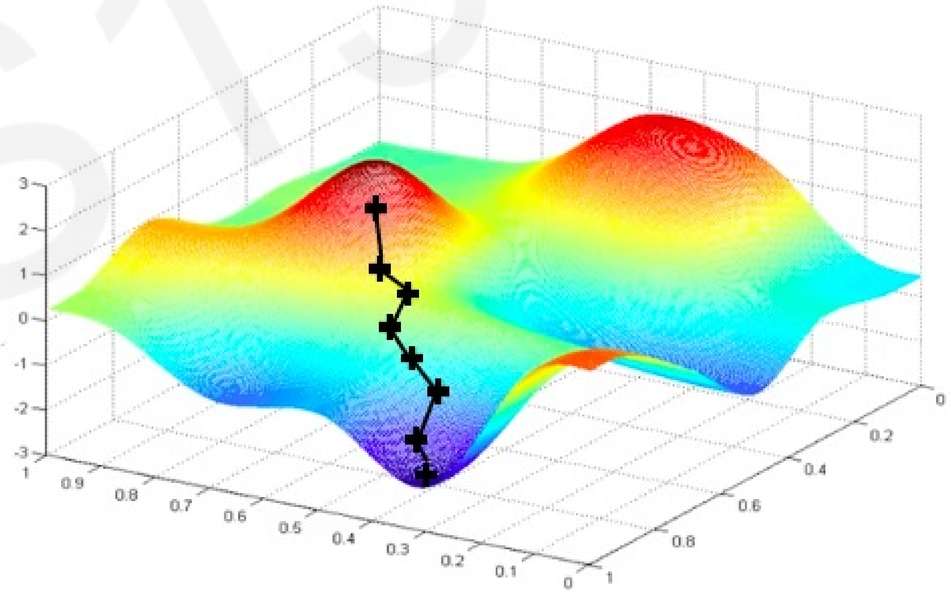
# Neural Networks in Practice: Mini-batches

MIT 6.S191

# Gradient Descent

**Algorithm**

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

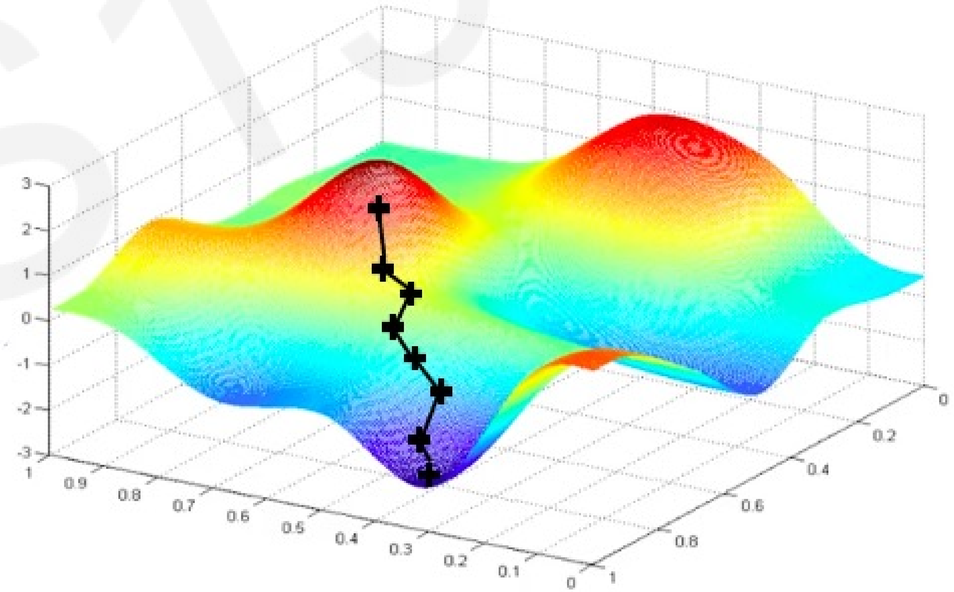



MIT 6.S191

# Gradient Descent

**Algorithm**

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



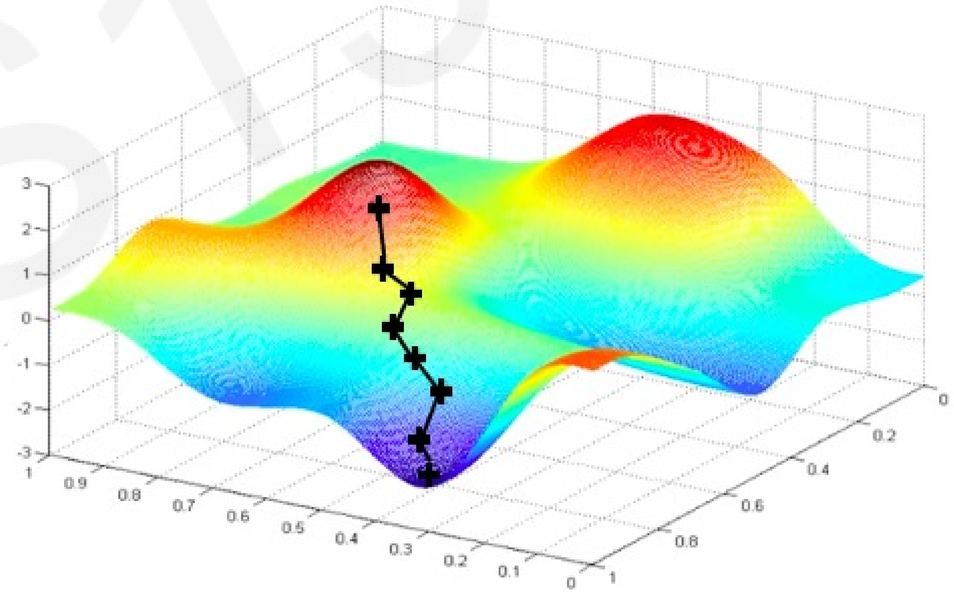
Can be very **computationally intensive** to compute!

MIT 6.S191

# Stochastic Gradient Descent

**Algorithm**

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

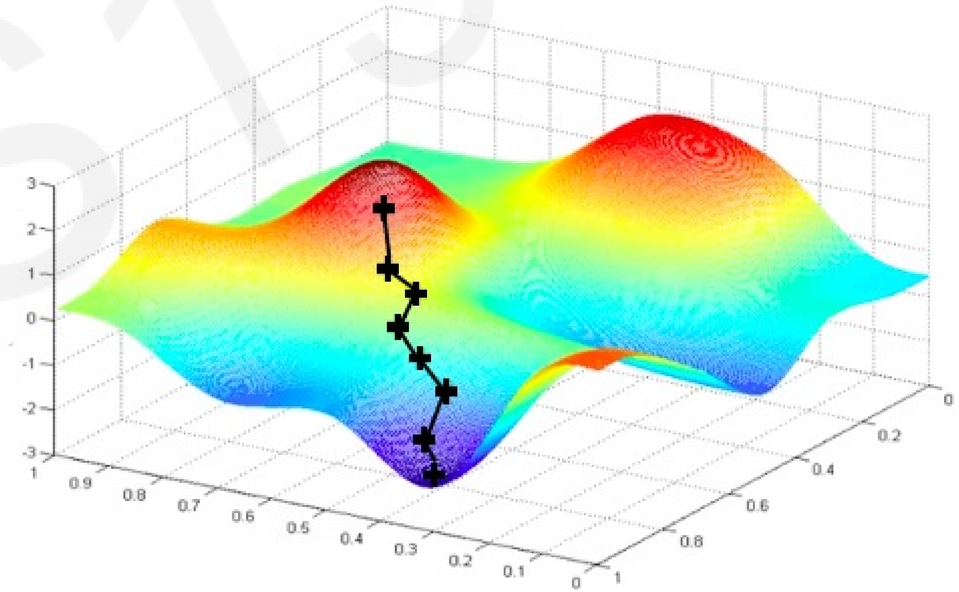



MIT 6.S191

# Stochastic Gradient Descent

**Algorithm**

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



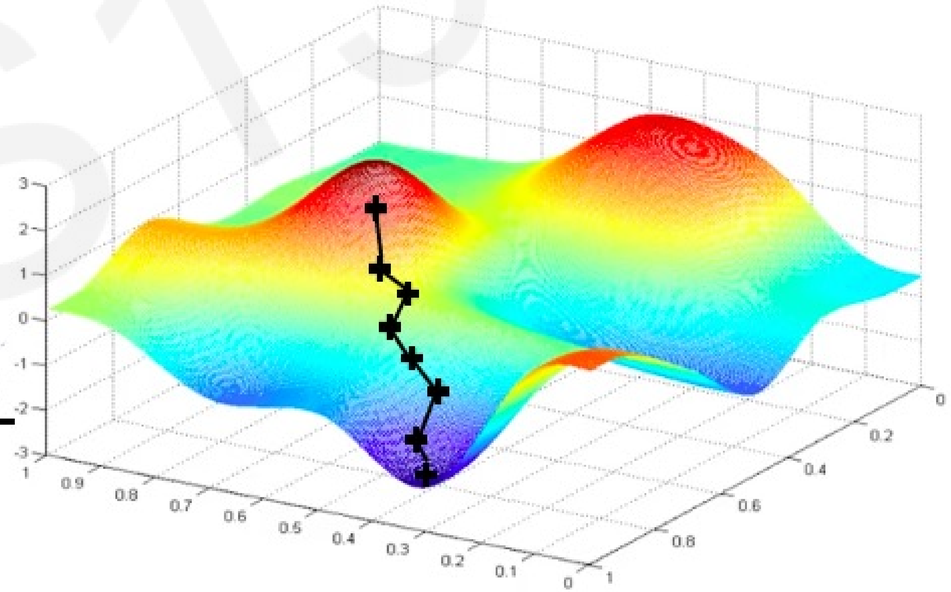
Easy to compute but **very noisy** (stochastic)!

MIT6.S191

# 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, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

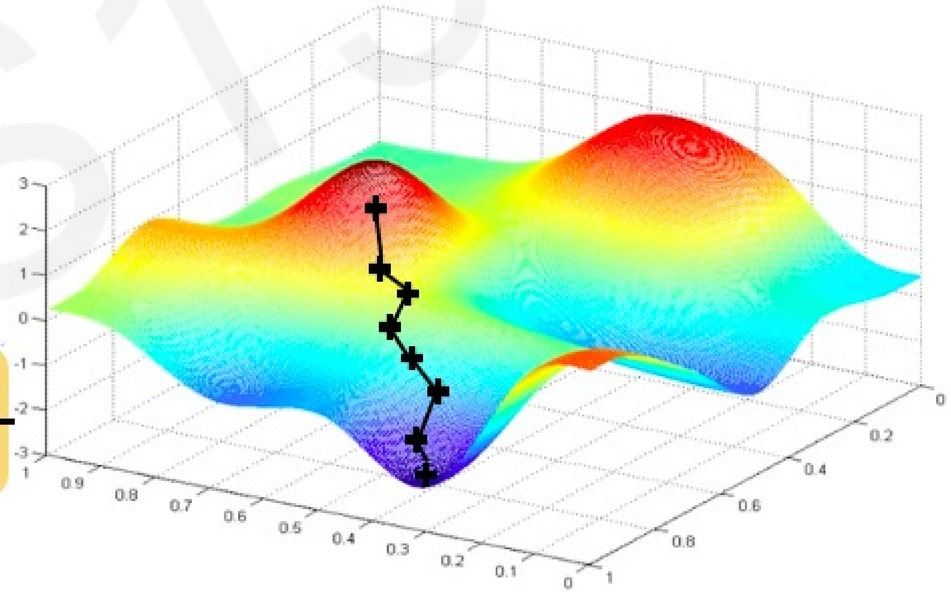


MIT 6.S191

# 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, $\boxed{\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



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

MIT 6.S191

# Mini-batches while training

**More accurate estimation of gradient**

Smoother convergence

Allows for larger learning rates

MIT 6.S191

# Mini-batches while training

**More accurate estimation of gradient**

Smoother convergence

Allows for larger learning rates

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

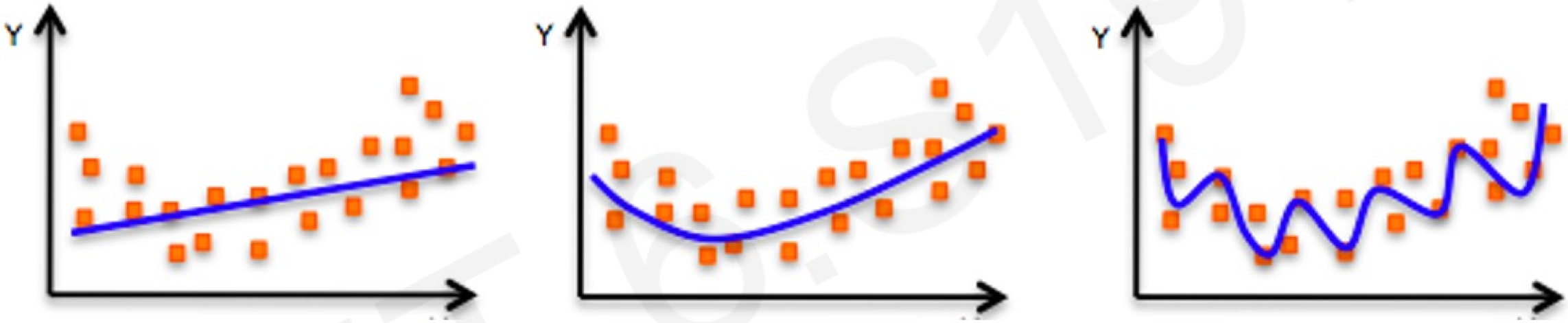
Can parallelize computation + achieve significant speed increases on GPU's

MIT 6.S191

# Neural Networks in Practice: Overfitting

MIT 6.S191

# 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

MIT 6.S191

# Regularization

***What is it?***

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

MIT 6.S191

# Regularization

*What is it?*

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

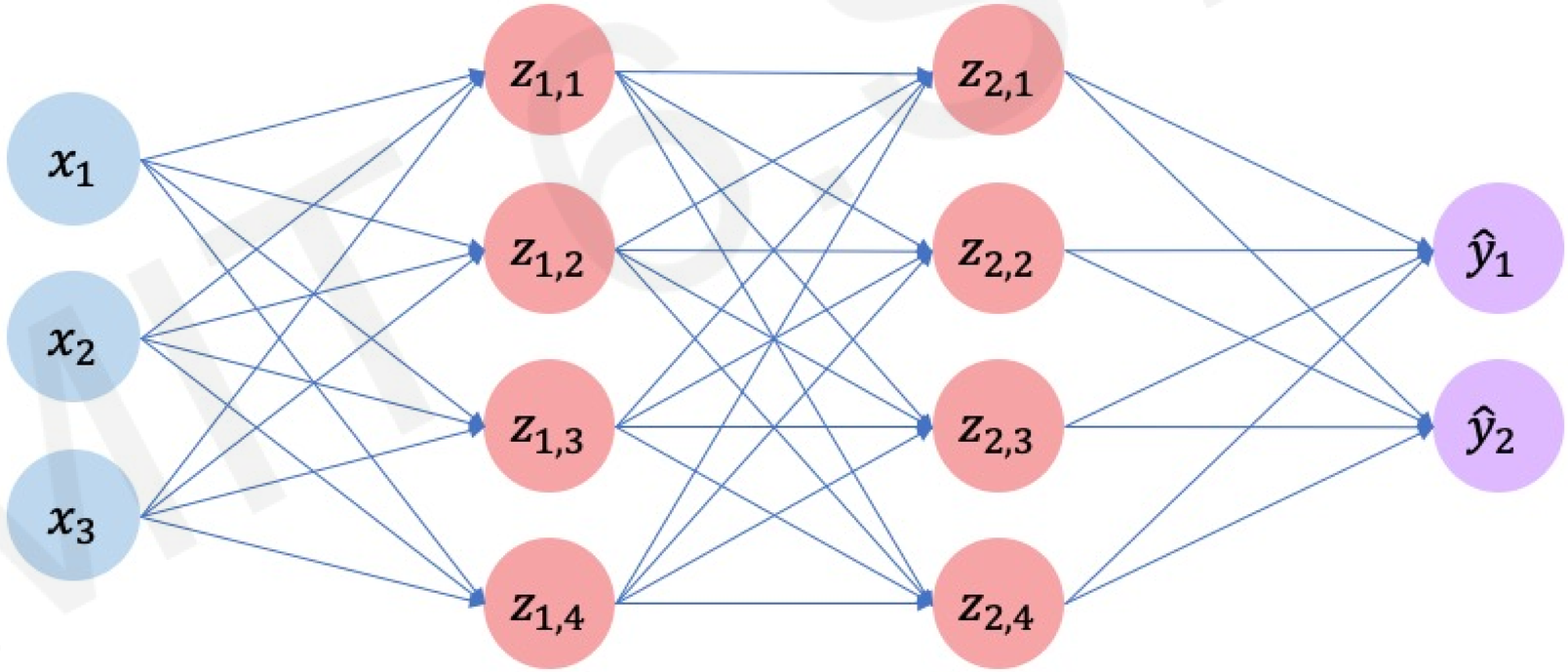
***Why do we need it?***

*Improve generalization of our model on unseen data*

MIT 6.S191

# Regularization 1: Dropout

- During training, randomly set some activations to 0

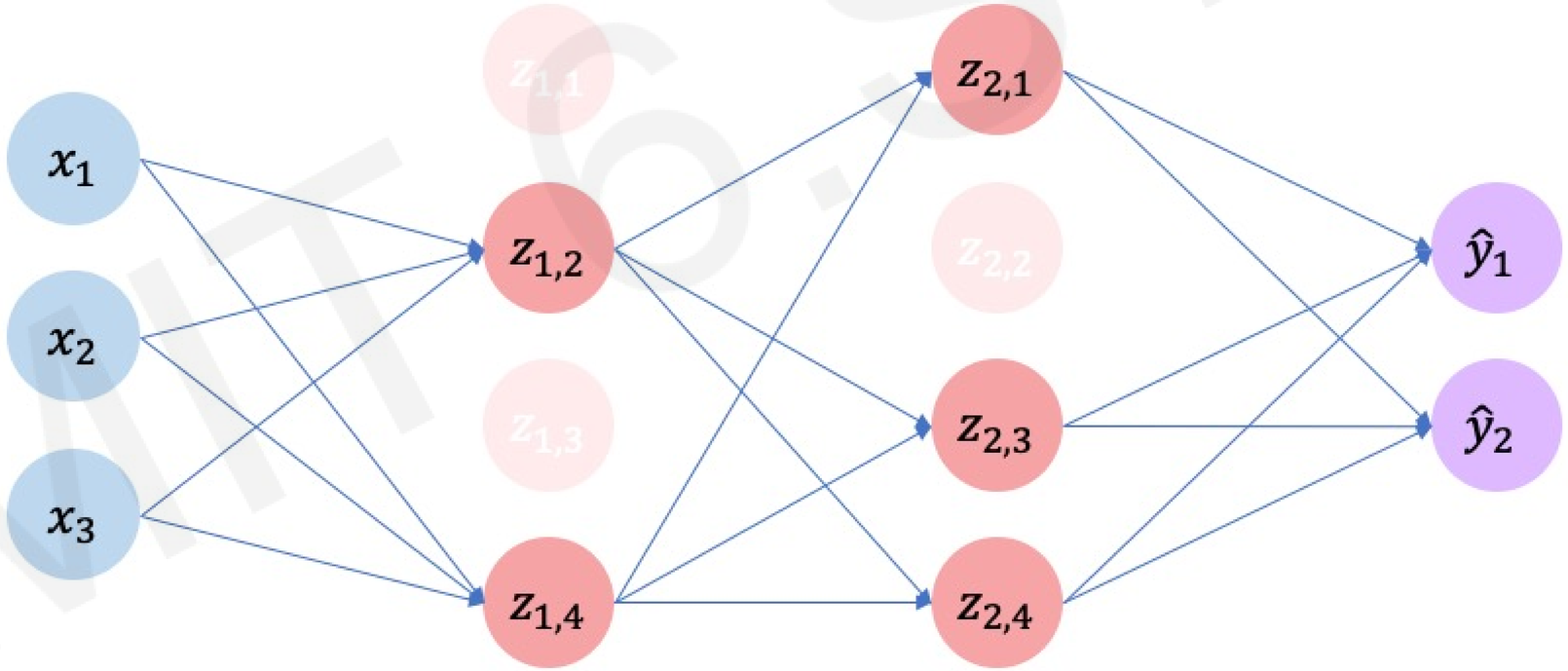



MIT 6.S191

# Regularization 1: 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)
```

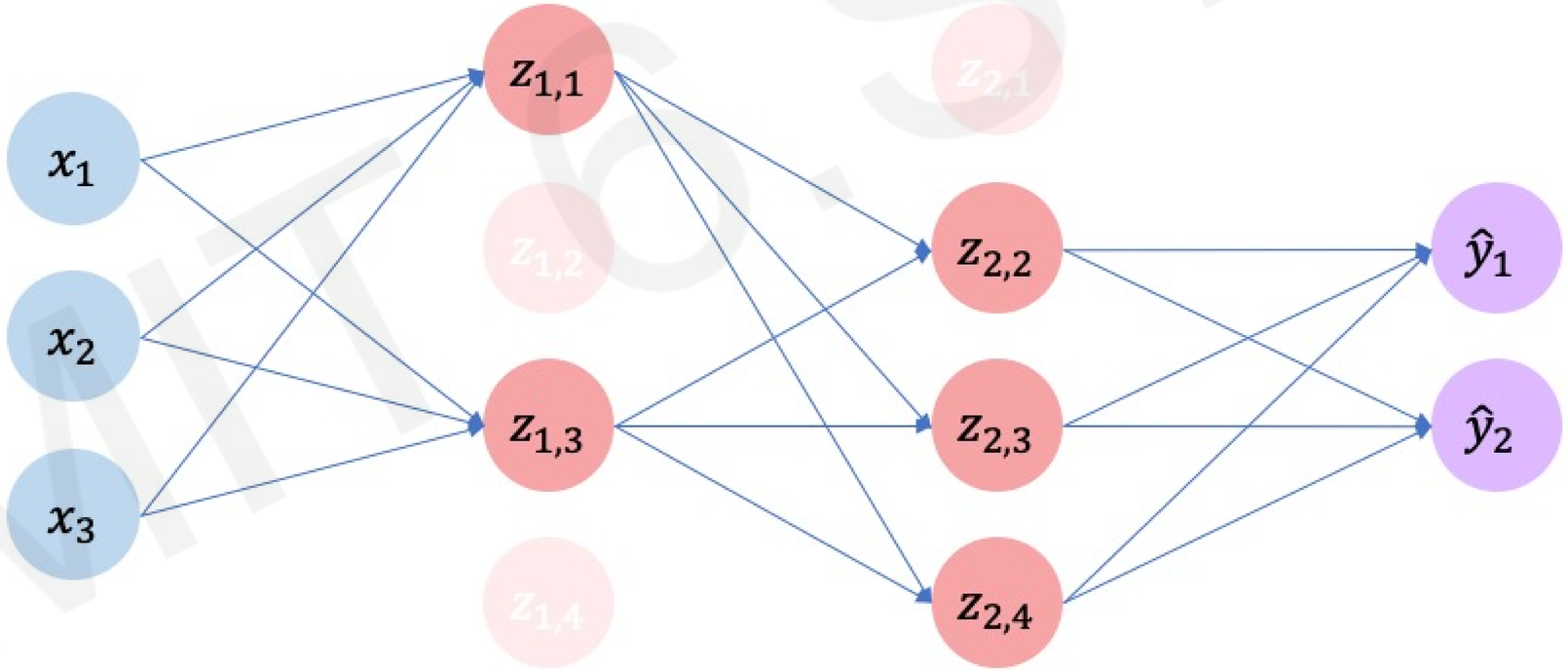



MIT 6.S191

# Regularization 1: 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

MIT 6.S191

# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

MIT 6.S191

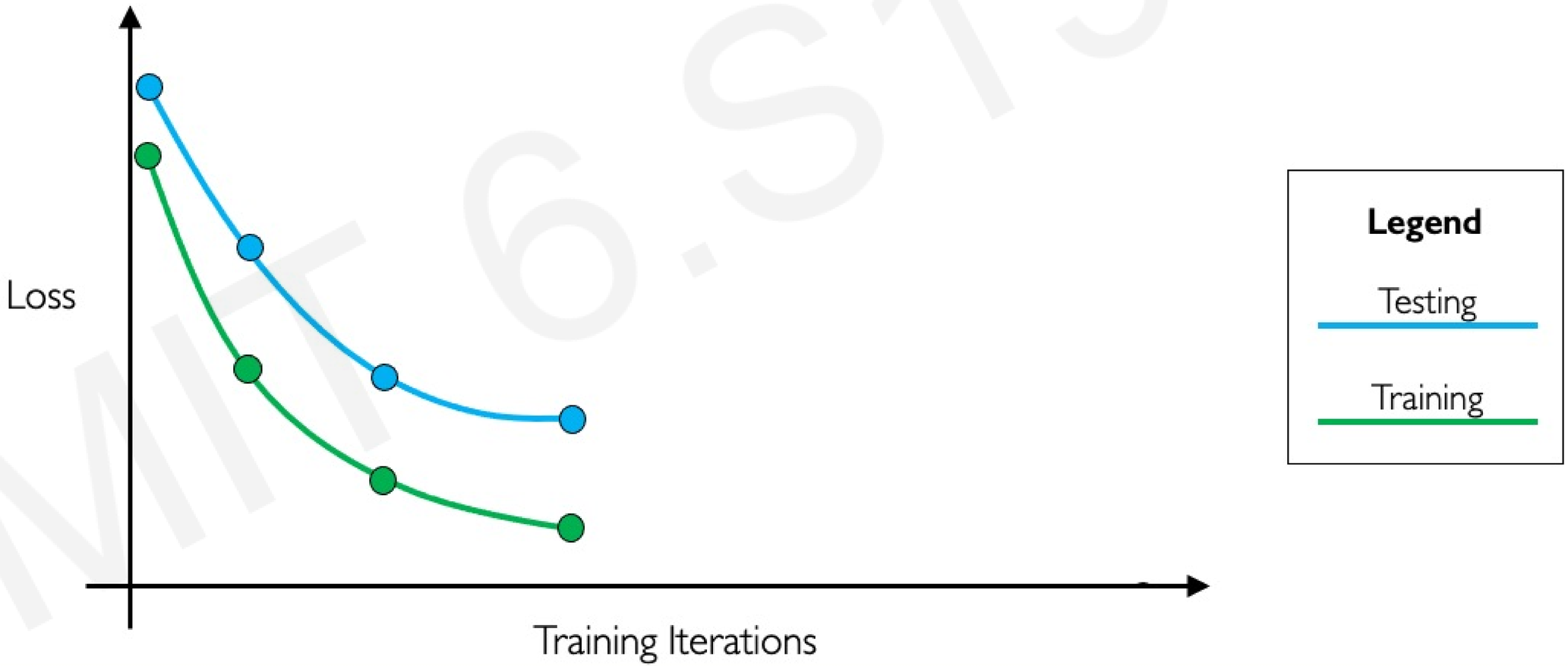
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

MIT 6.S191

# Regularization 2: Early Stopping

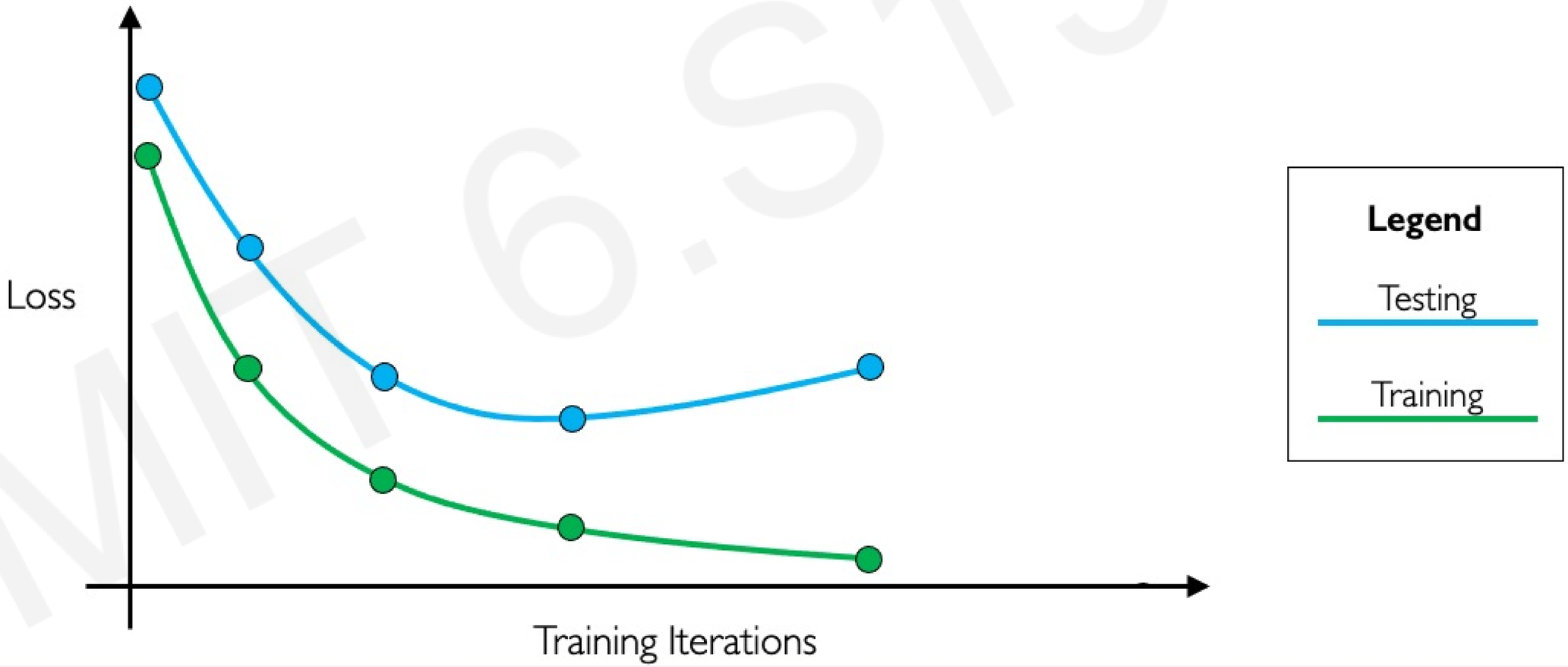
- Stop training before we have a chance to overfit




MIT 6.S191

# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit




MIT 6.S191

# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

MIT 6.S191

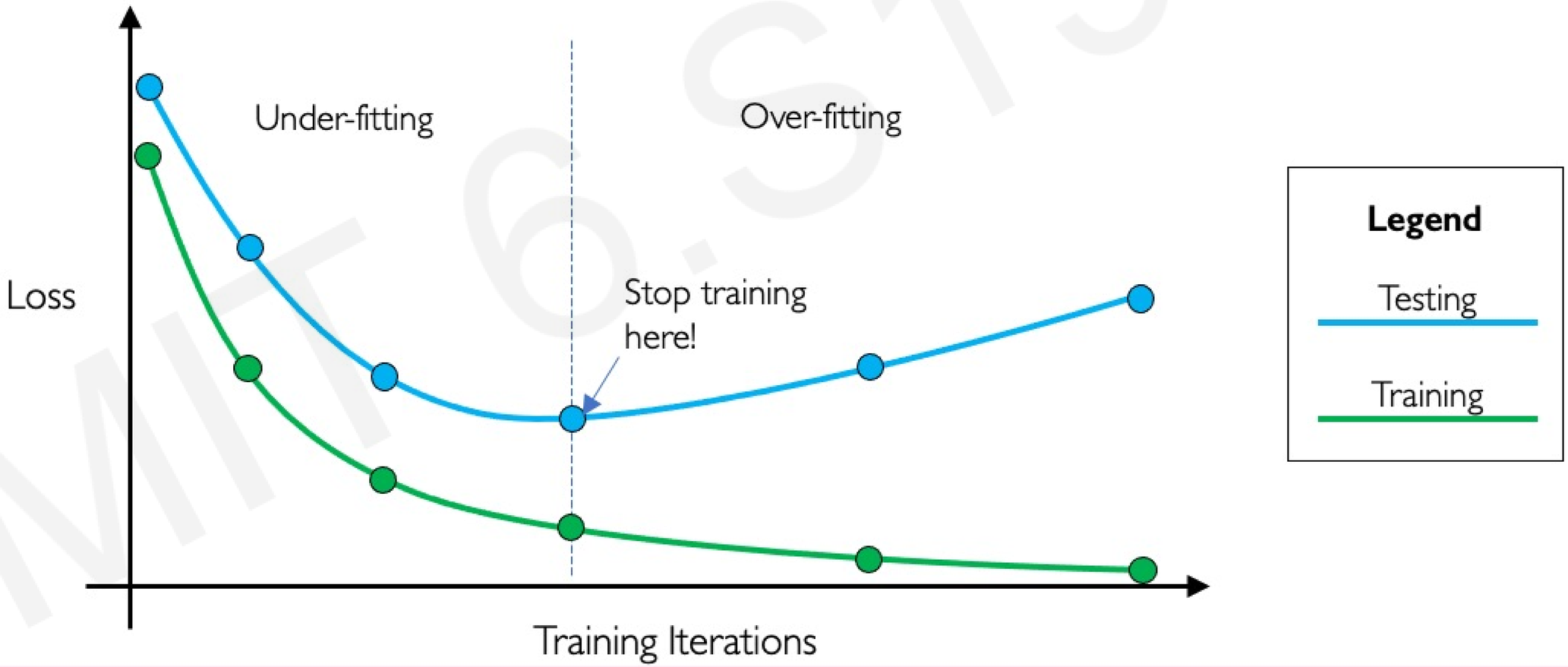
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

MIT 6.S191

# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

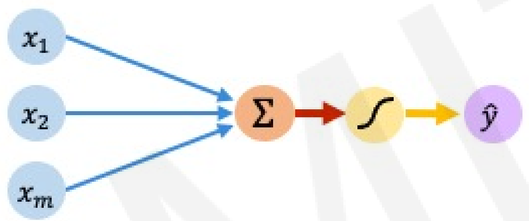



MIT 6.S191

# Core Foundation Review

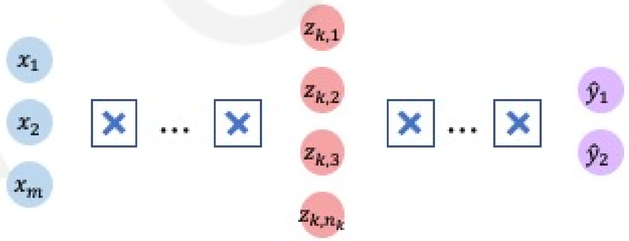
## The Perceptron

- Structural building blocks
- Nonlinear activation functions



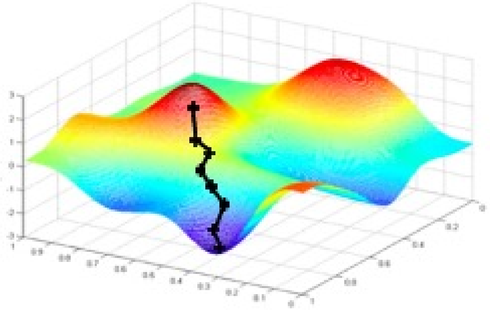

## Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation




## Training in Practice

- Adaptive learning
- Batching
- Regularization



MIT 6.S191

# MIT Introduction to Deep Learning

## Lab 1: Introduction to TensorFlow and Music Generation with RNNs

Link to download labs:
http://introtodeeplearning.com#schedule

1. Open the lab in Google Colab
2. Start executing code blocks and filling in the #TODOs
3. Need help? Come to 32-123!