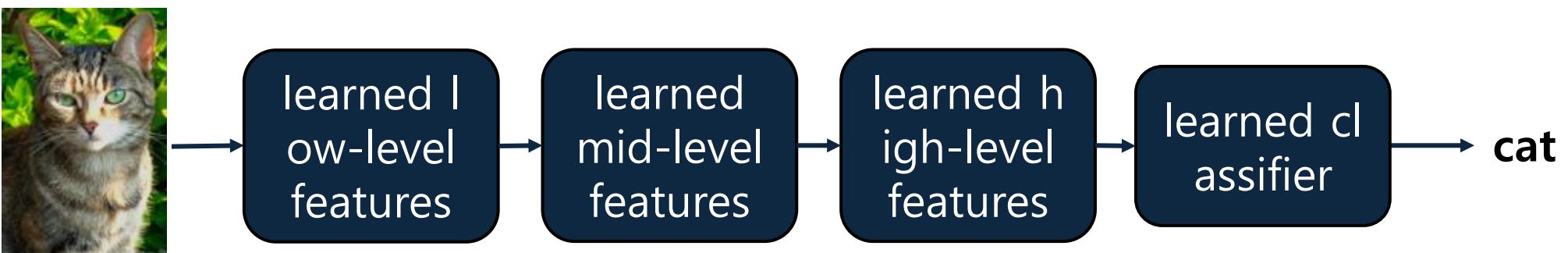


“Traditional” machine learning:



Deep, “end-to-end” learning:



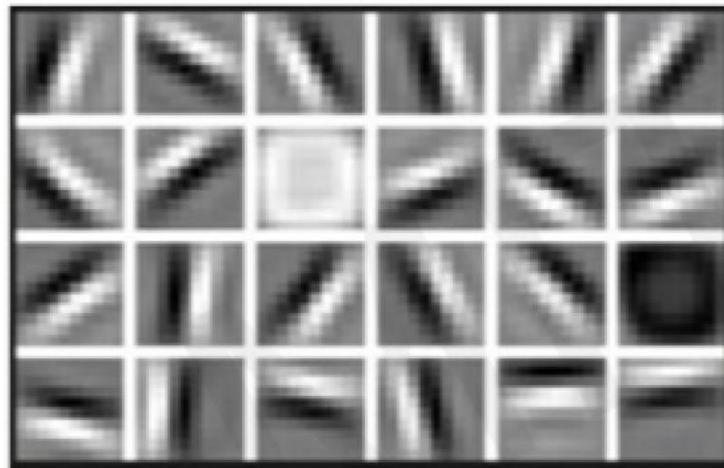
# Why Deep Learning and Why Now?

# 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



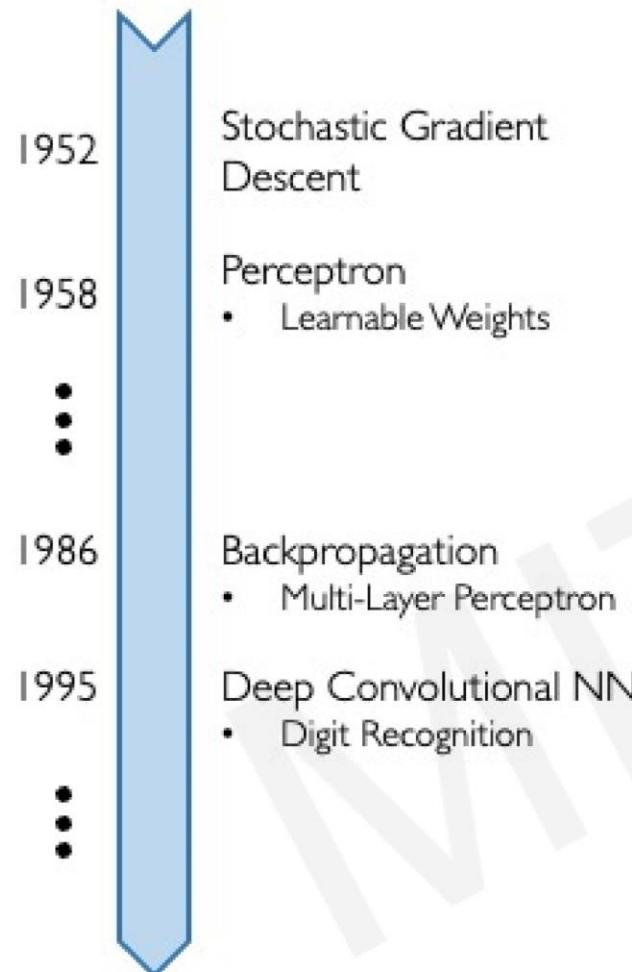
Eyes & Nose & Ears

High Level Features



Facial Structure

# Why Now?

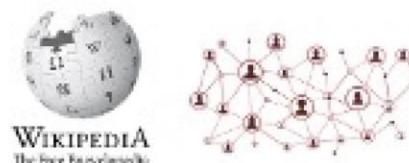


Neural Networks date back decades, so why the dominance?

## I. Big Data

- Larger Datasets
- Easier Collection & Storage

IM<sup>2</sup>GENET



## 2. Hardware

- Graphics Processing Units (GPUs)
- Massively Parallelizable

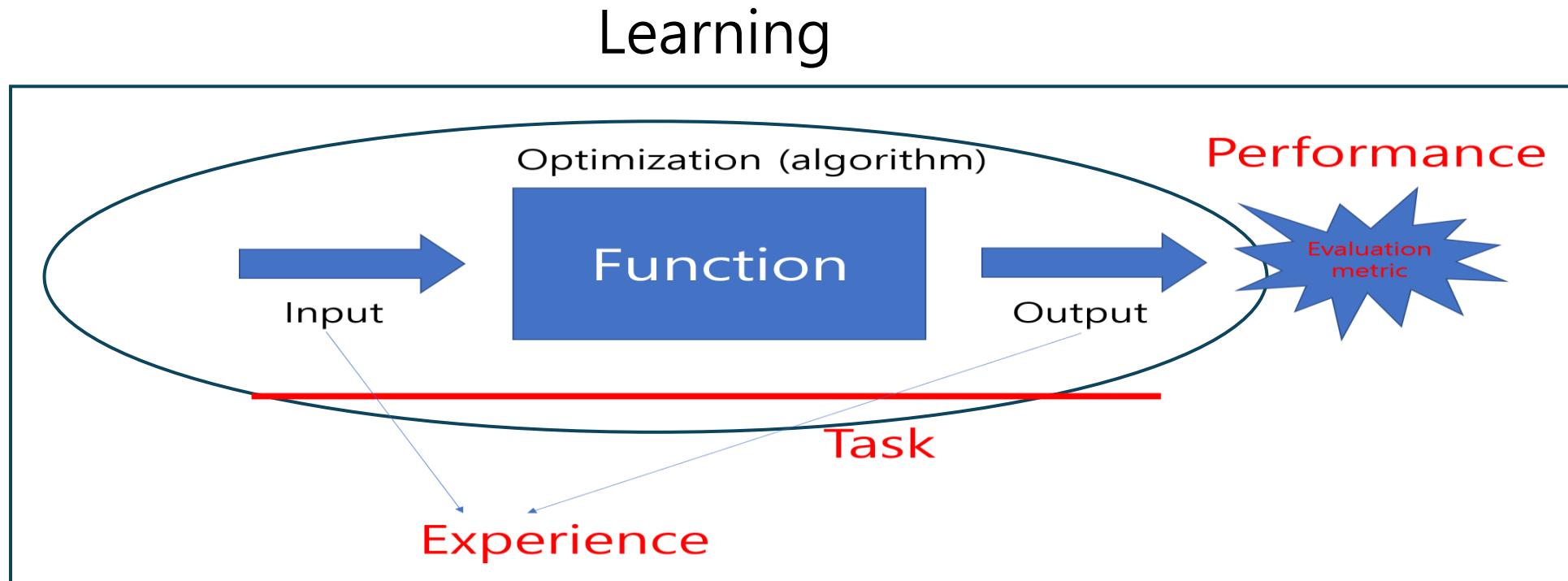


## 3. Software

- Improved Techniques
- New Models
- Toolboxes



# AI Goals



Goal

Function

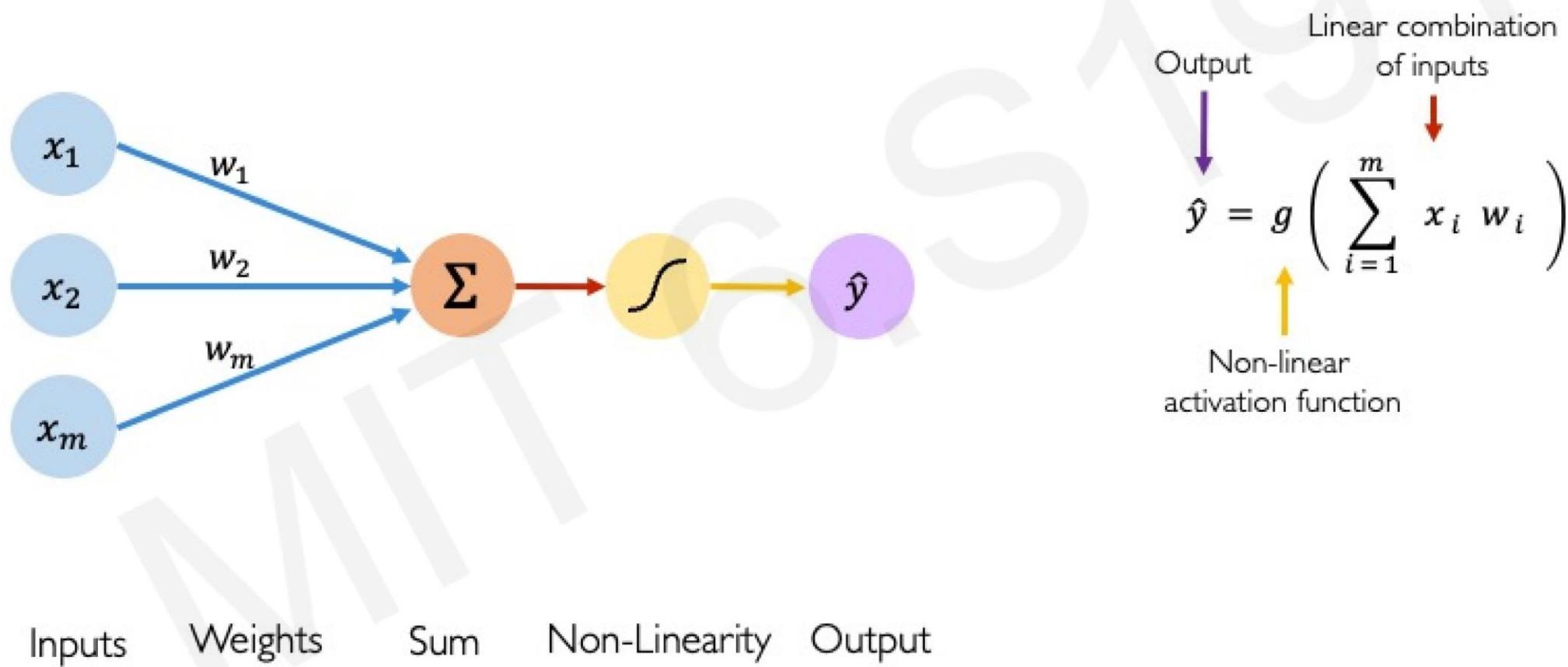
머신러닝으로 근사

Target Function

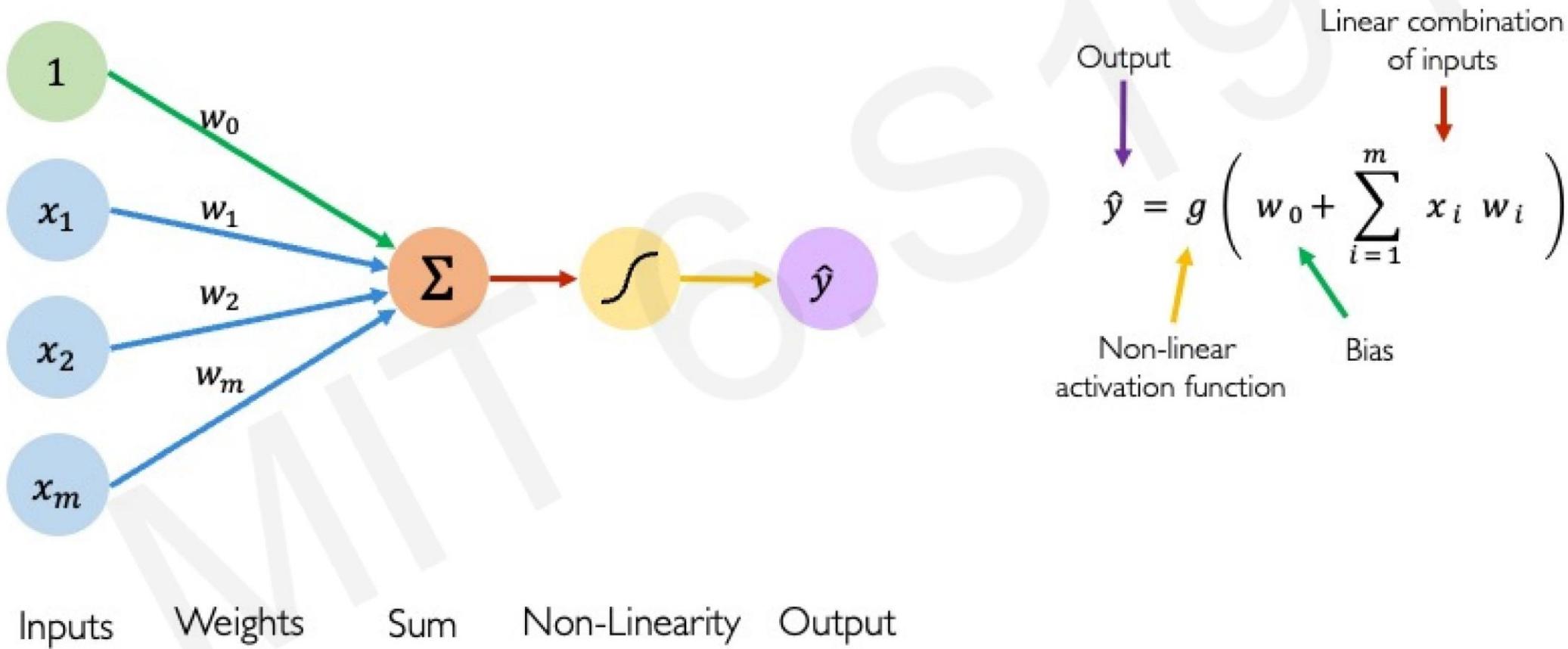
# The Perceptron

## The structural building block of deep learning

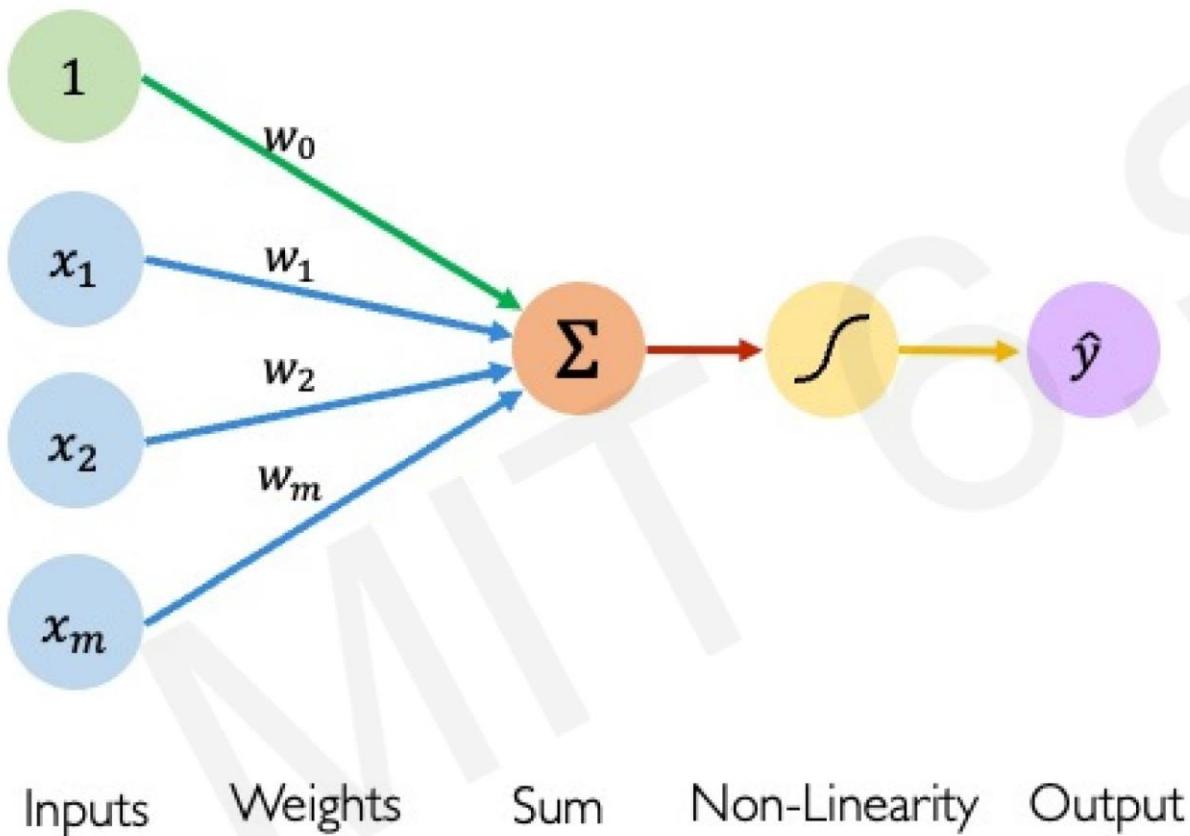
# The Perceptron: Forward Propagation



# The Perceptron: Forward Propagation



# The Perceptron: Forward Propagation

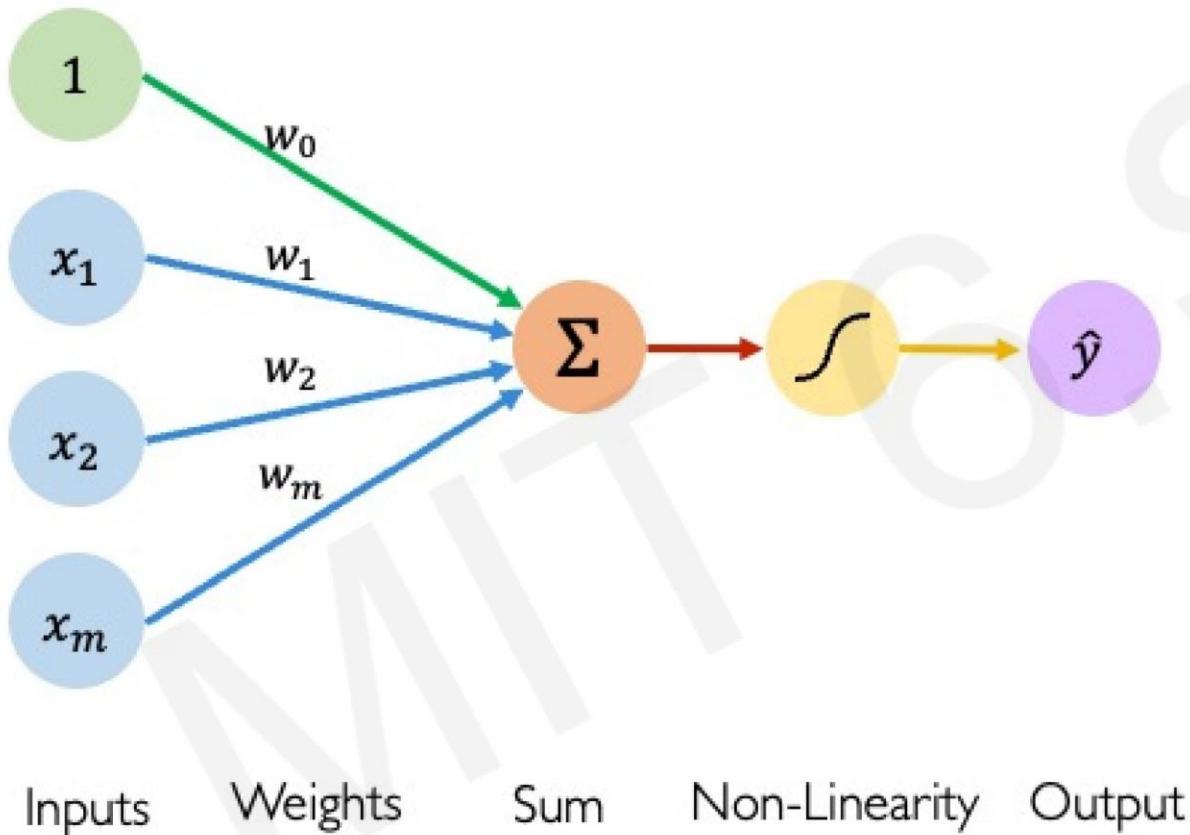


$$\hat{y} = g \left( w_0 + \sum_{i=1}^m x_i w_i \right)$$

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

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

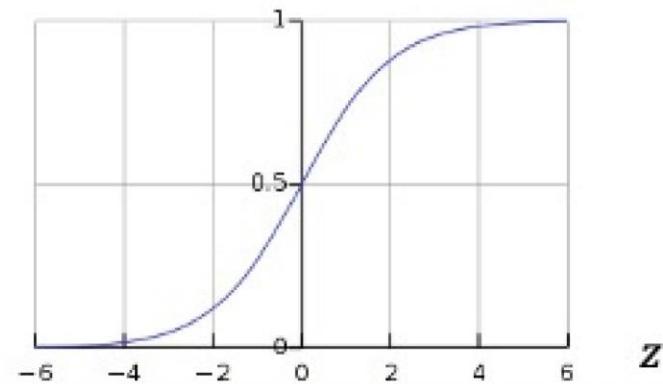


## Activation Functions

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

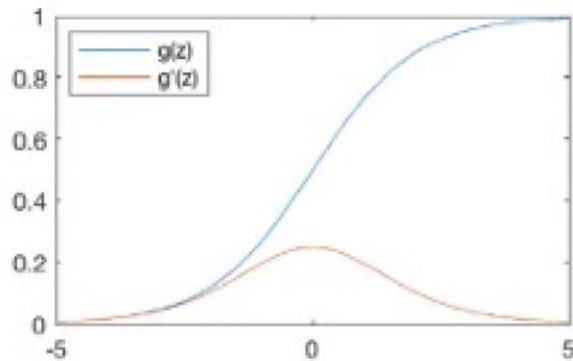
- Example: sigmoid function

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



# Common Activation Functions

Sigmoid Function

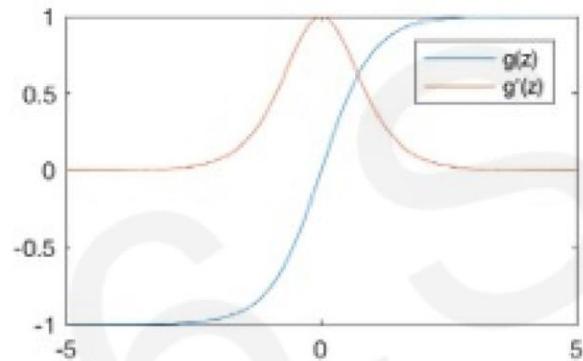


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

`tf.math.sigmoid(z)`

Hyperbolic Tangent

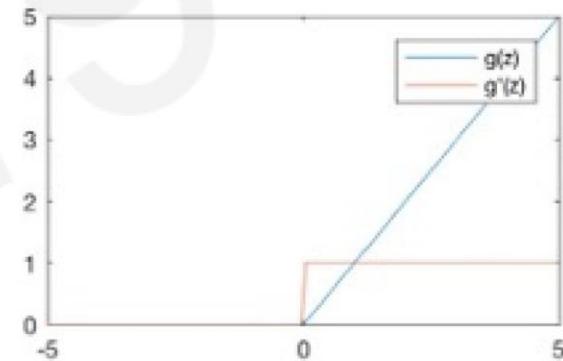


$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

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

`tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



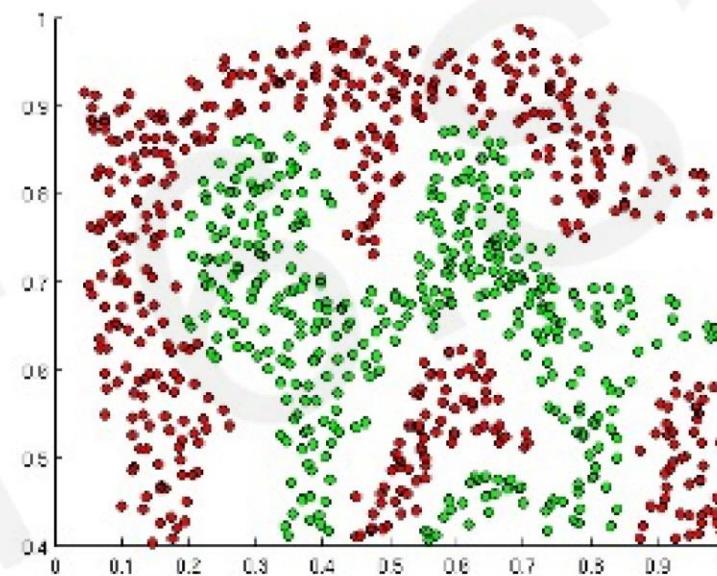
$$g(z) = \max(0, z)$$

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

`tf.nn.relu(z)`

# Importance of Activation Functions

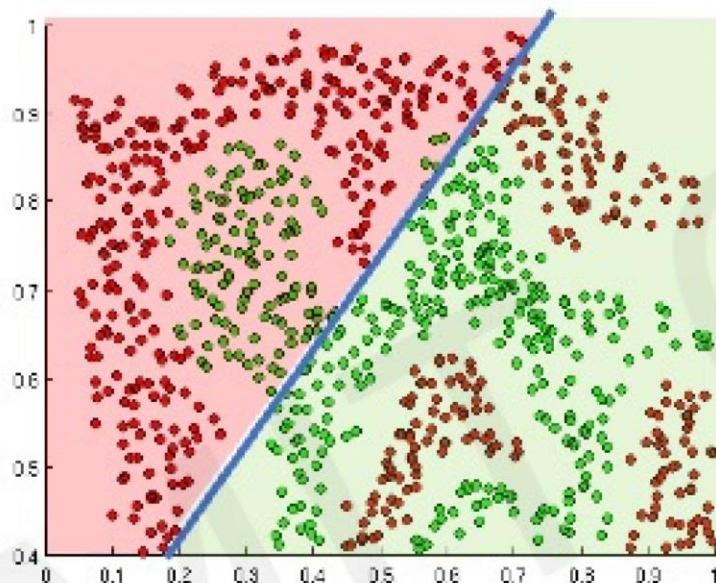
The purpose of activation functions is to **introduce non-linearities** into the network



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

# Importance of Activation Functions

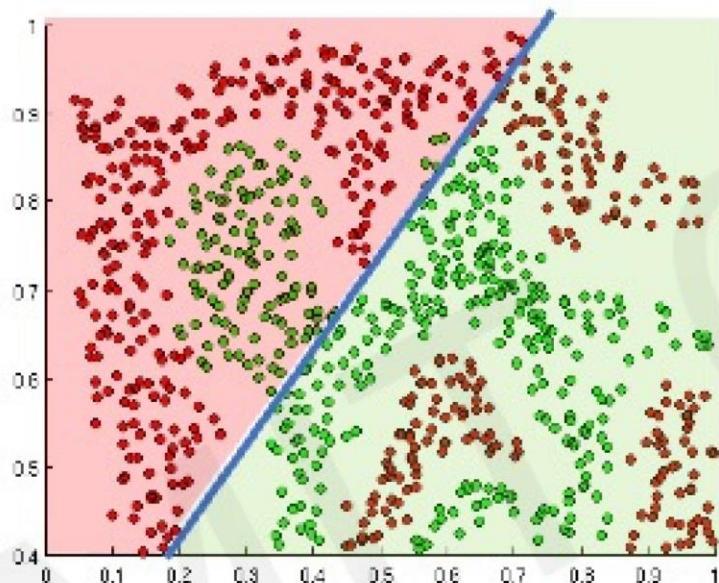
The purpose of activation functions is to **introduce non-linearities** into the network



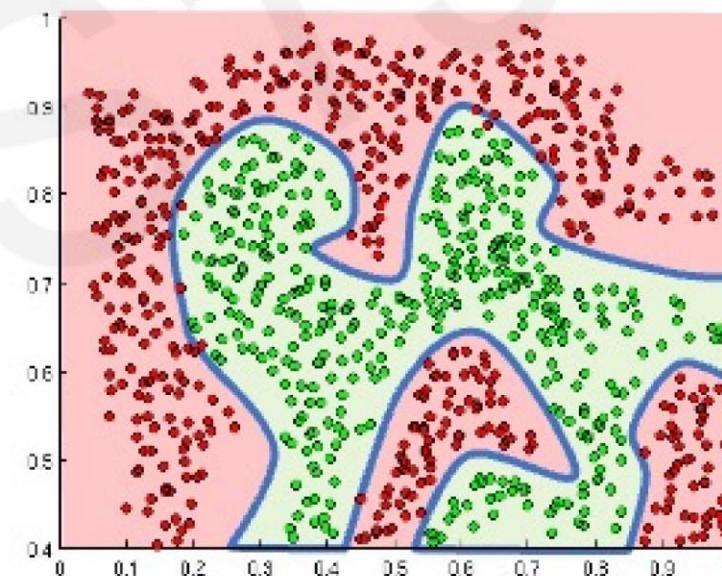
Linear activation functions produce linear decisions no matter the network size

# Importance of Activation Functions

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

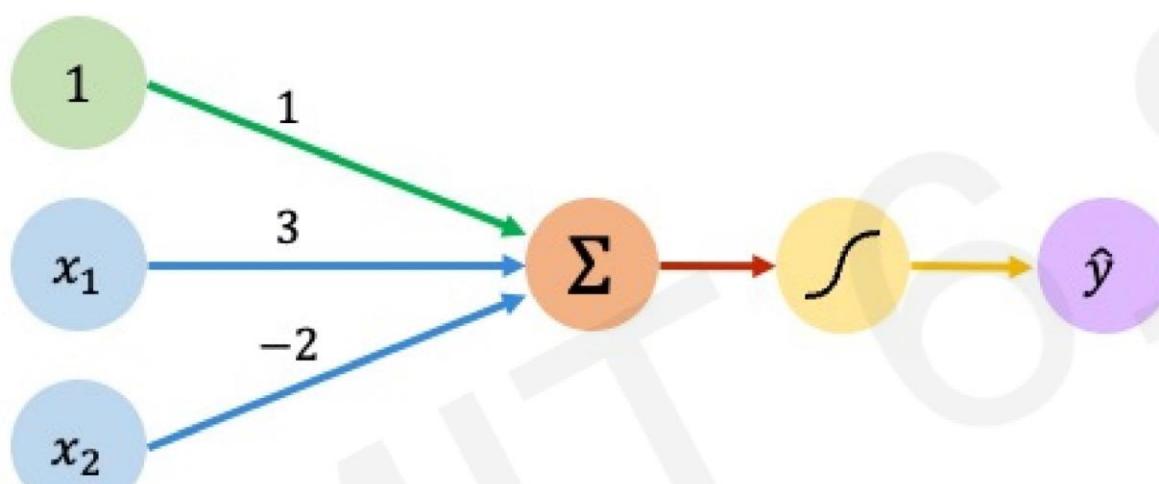


Linear activation functions produce linear decisions no matter the network size



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!

# 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**

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

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



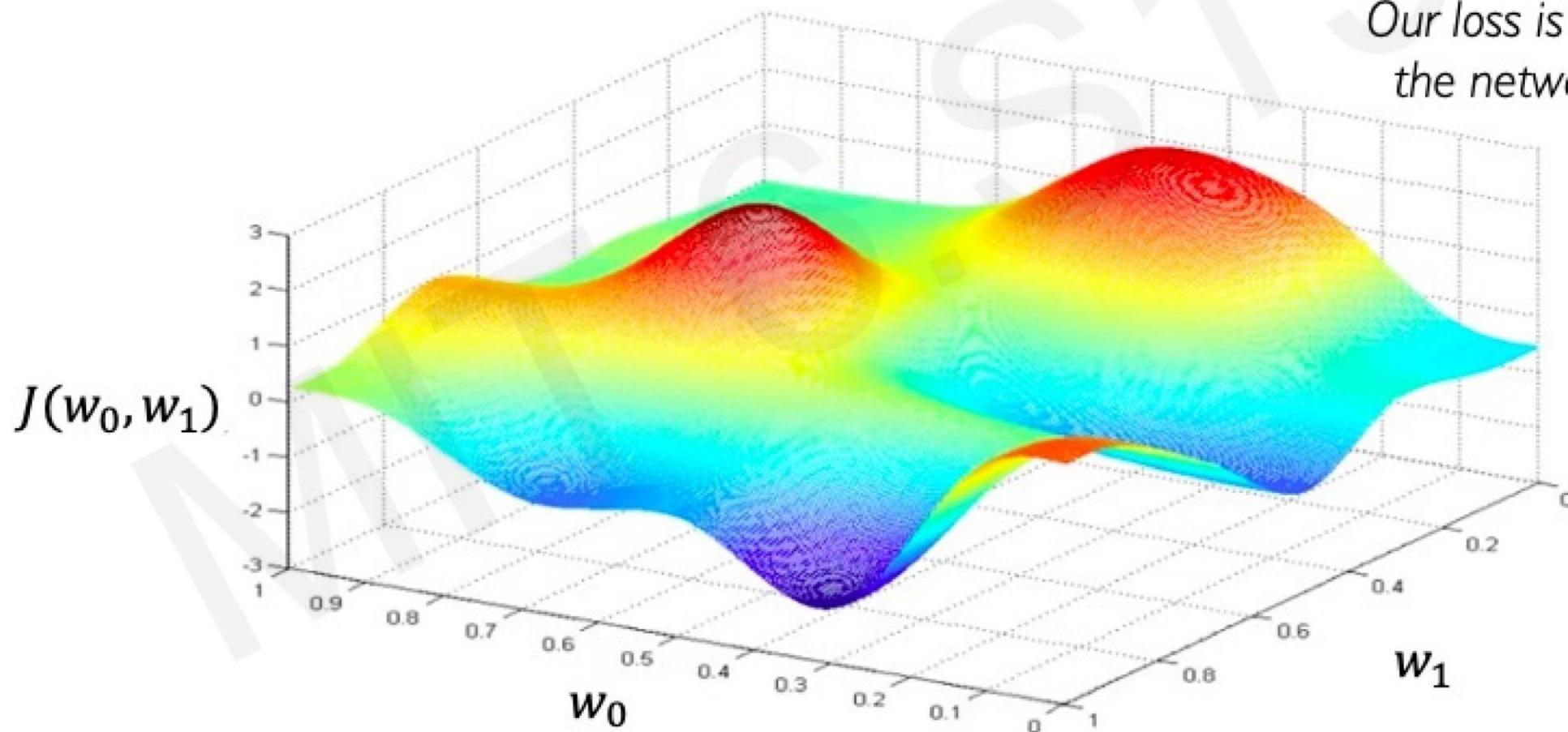
Remember:

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

# Loss Optimization

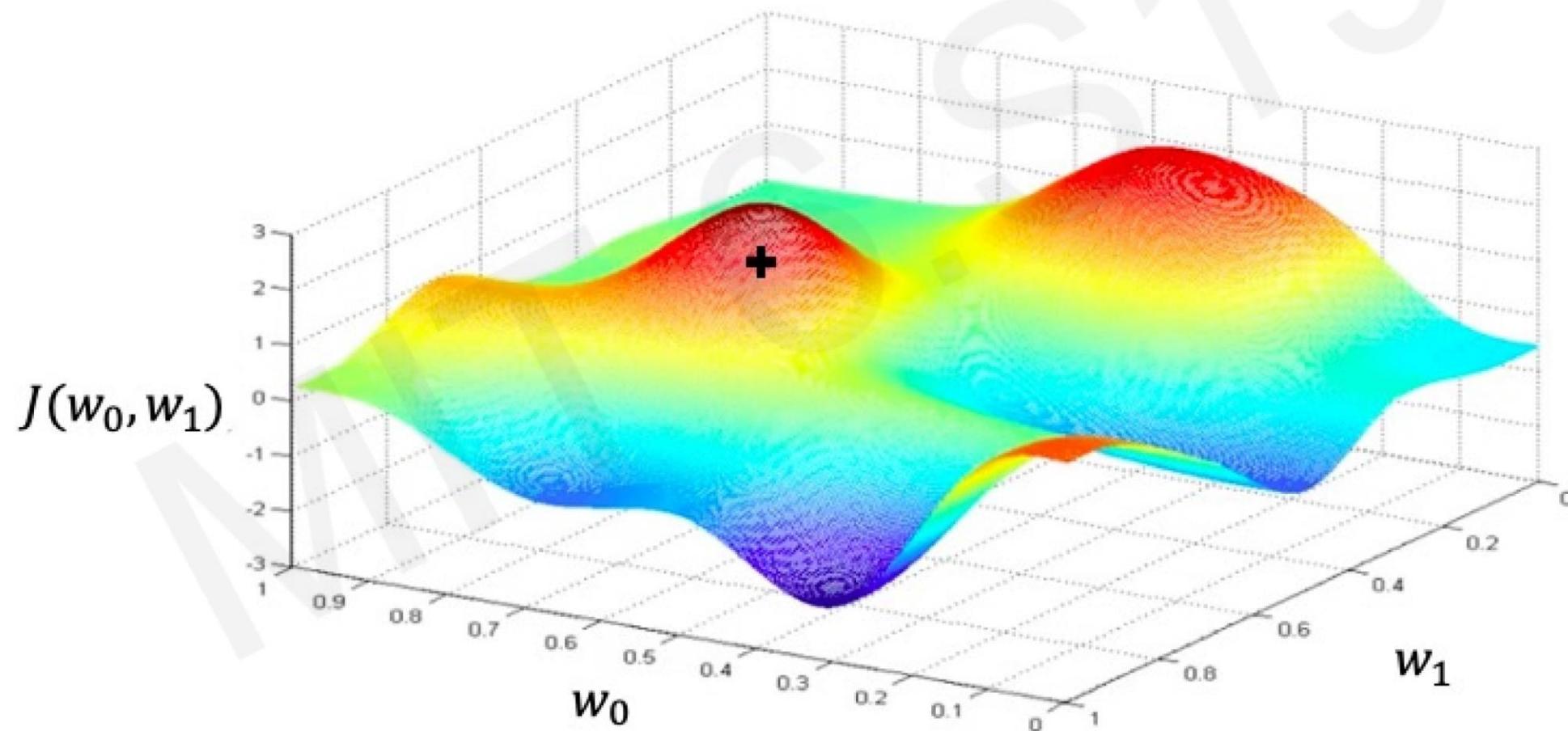
$$\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmin}} J(\mathbf{W})$$

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



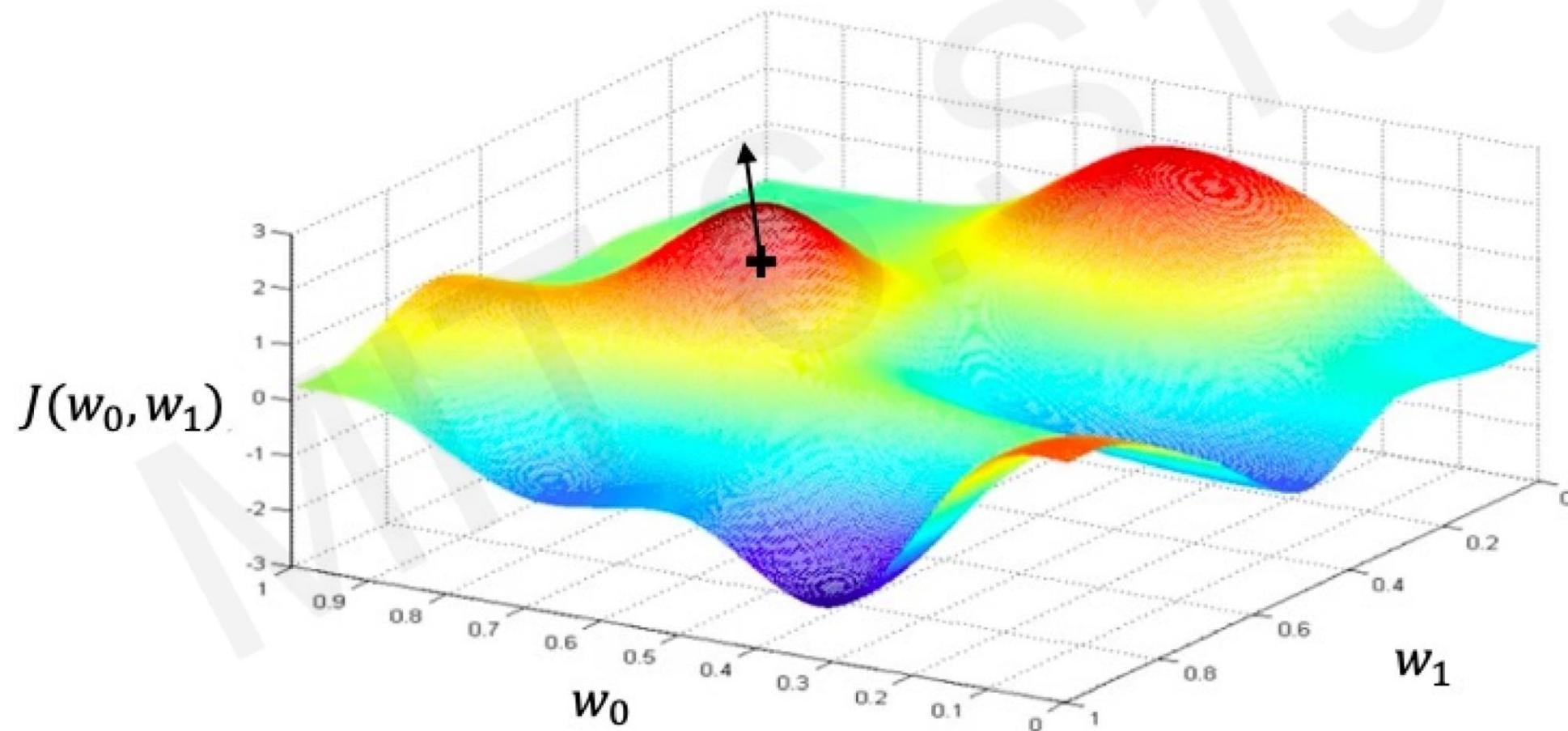
# Loss Optimization

Randomly pick an initial  $(w_0, w_1)$



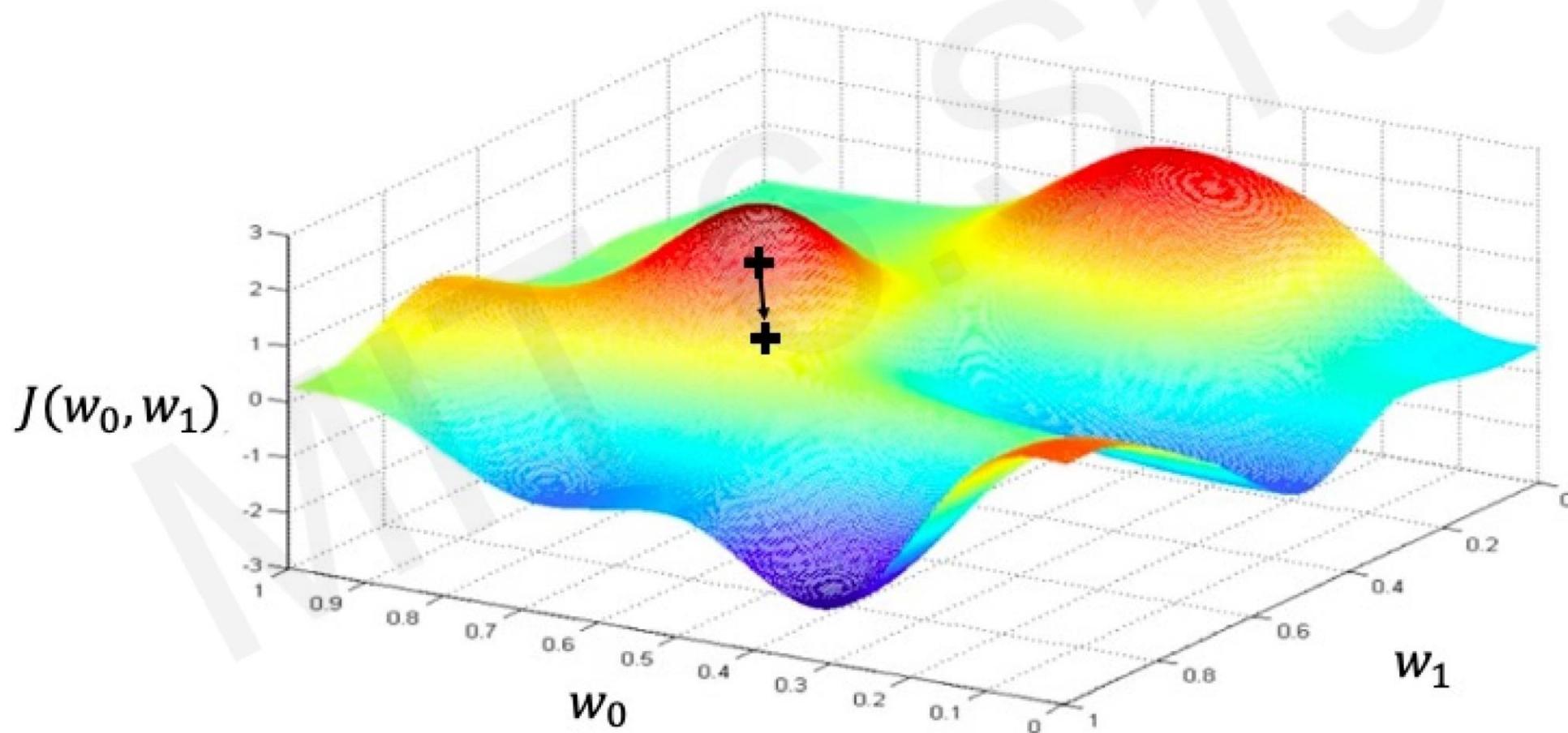
# Loss Optimization

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



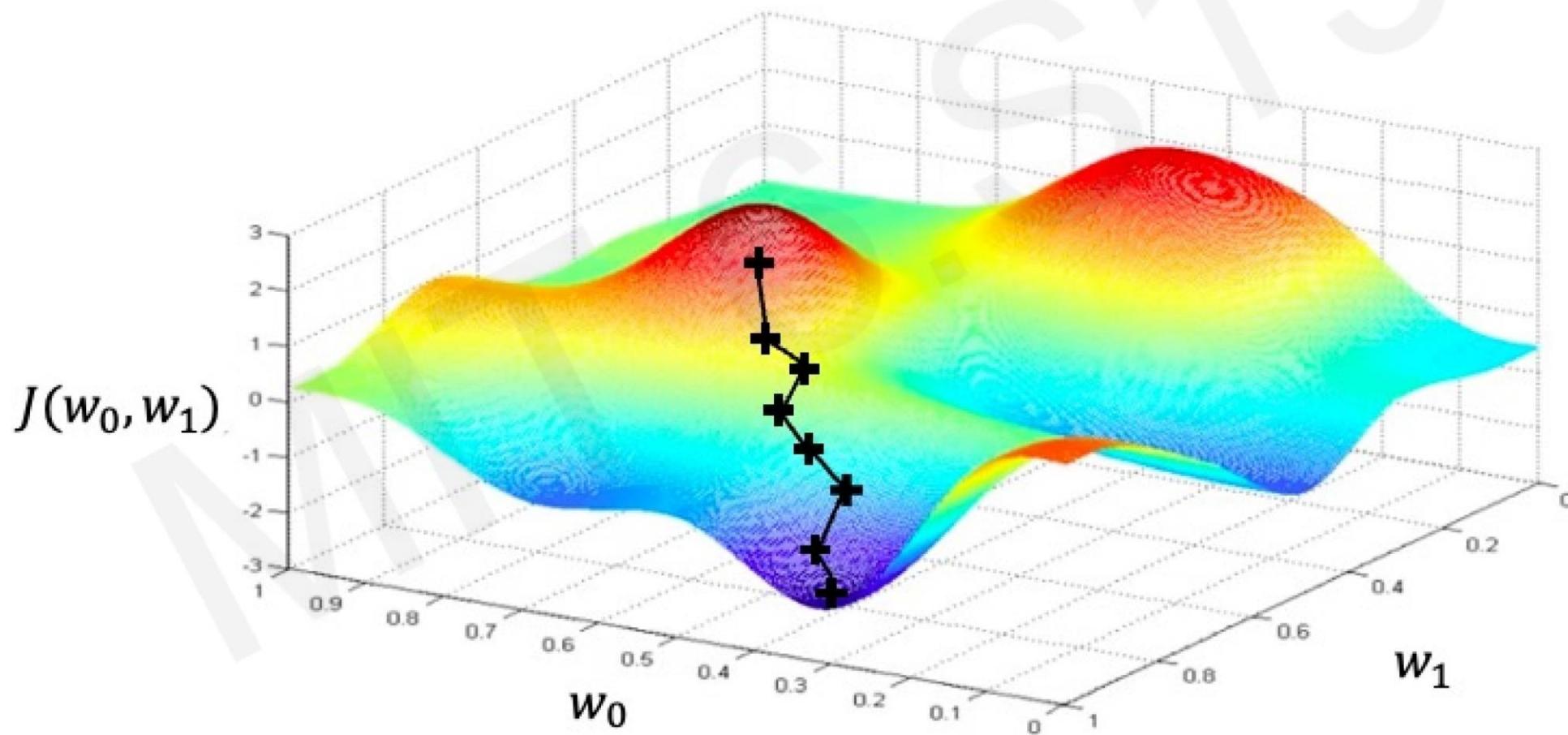
# Loss Optimization

Take small step in opposite direction of gradient



# Gradient Descent

Repeat until convergence



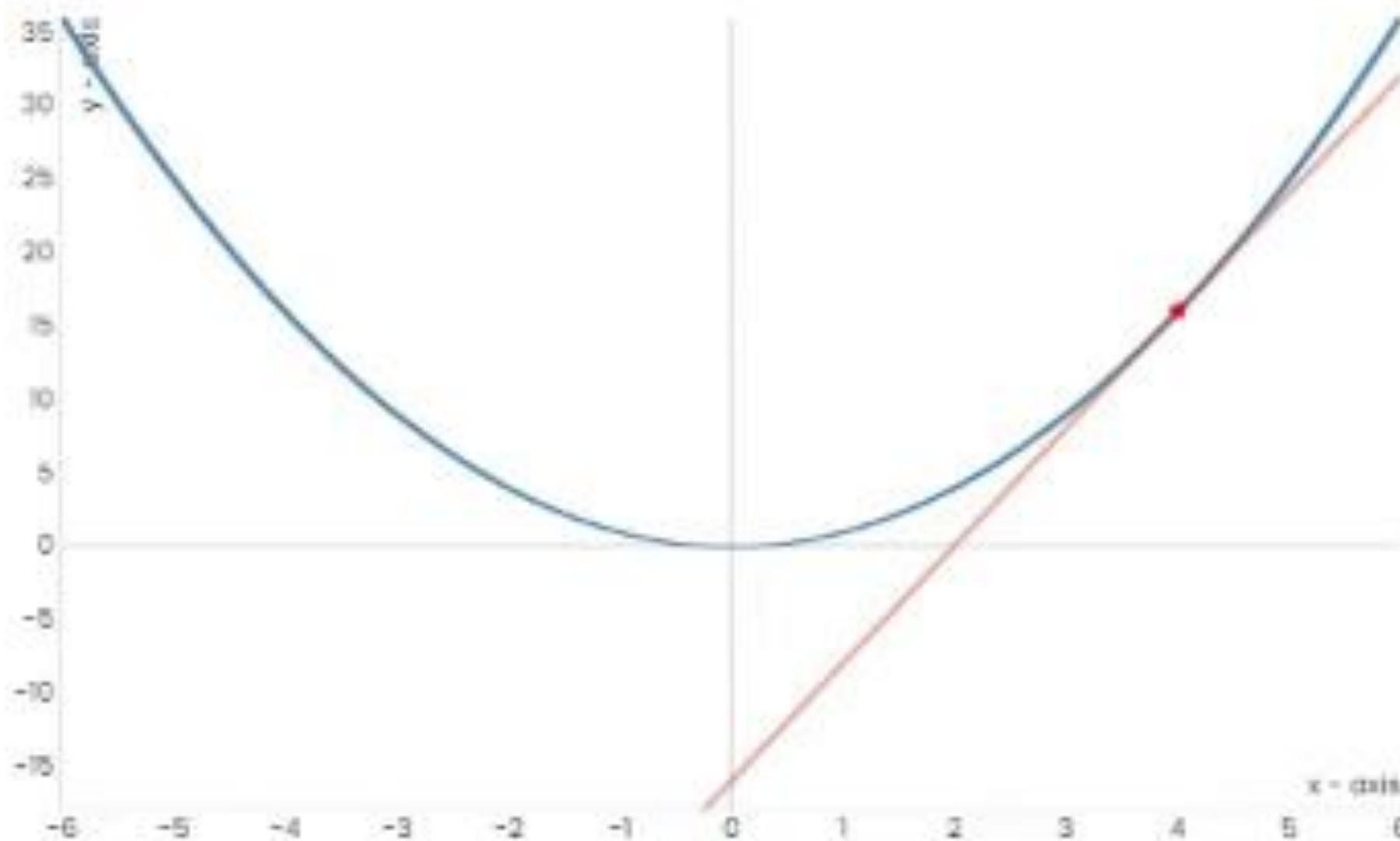
# Gradient Descent

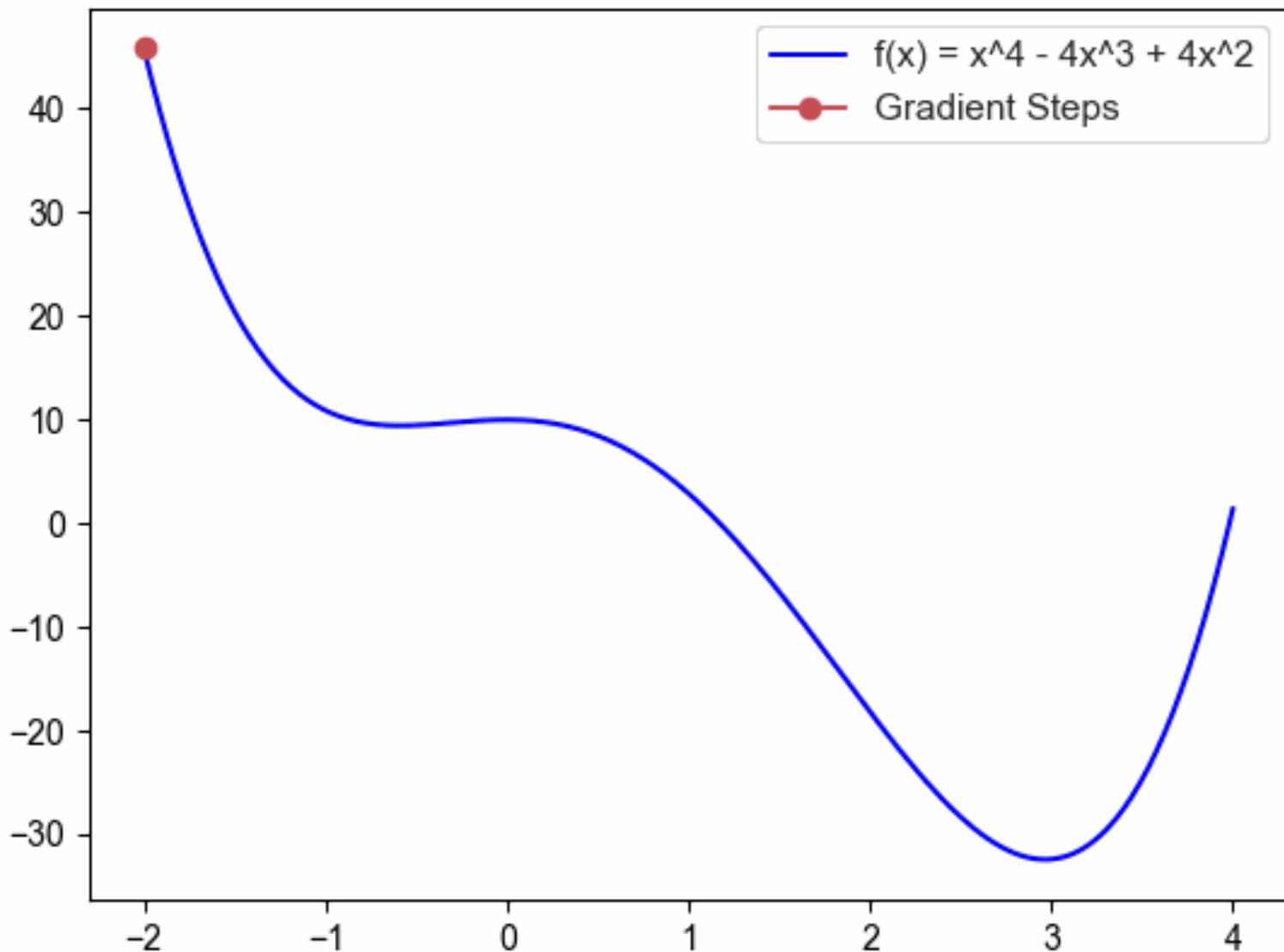
## Algorithm

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

# iteration 0

-0.479, 0.228





# Optimization

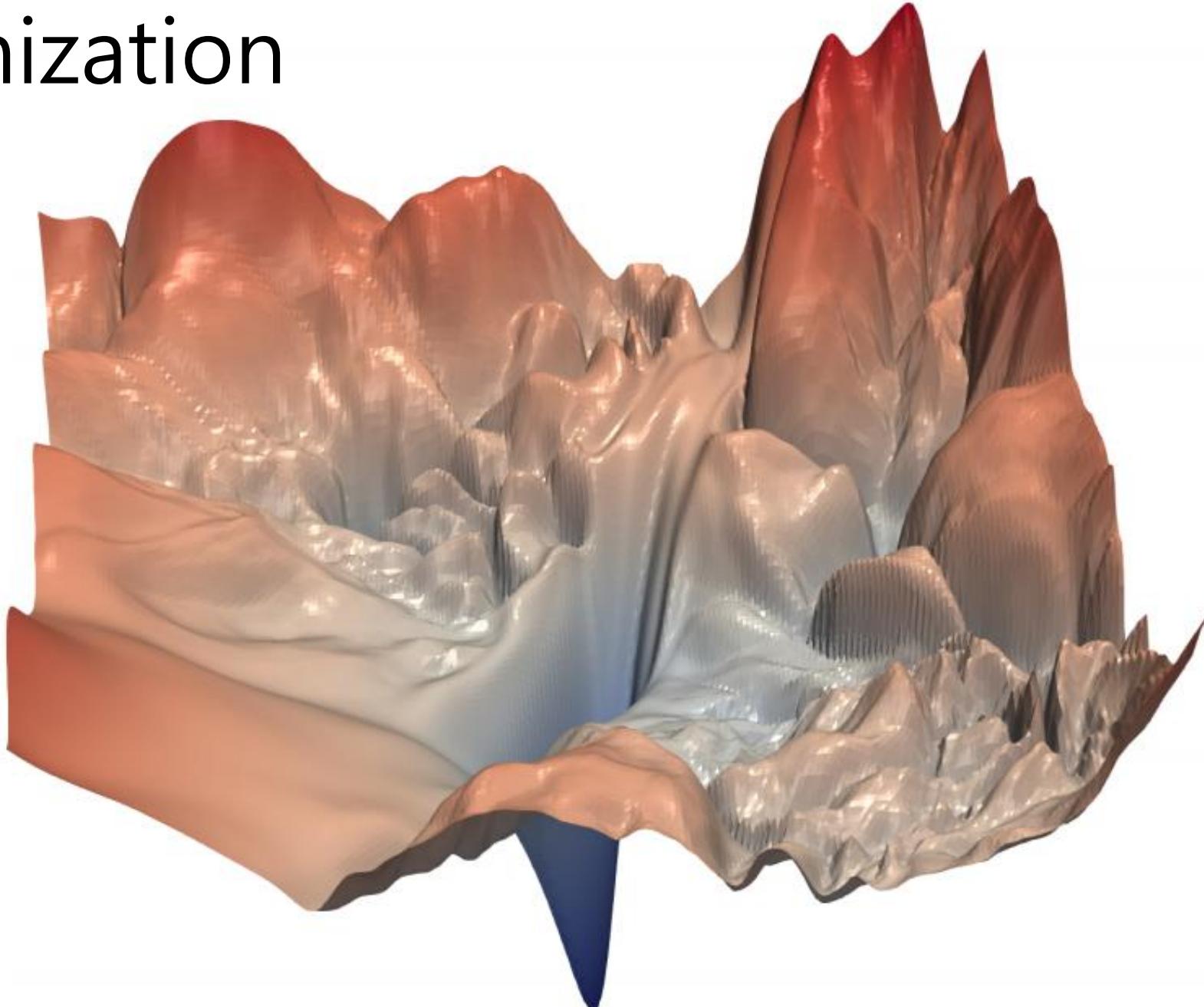
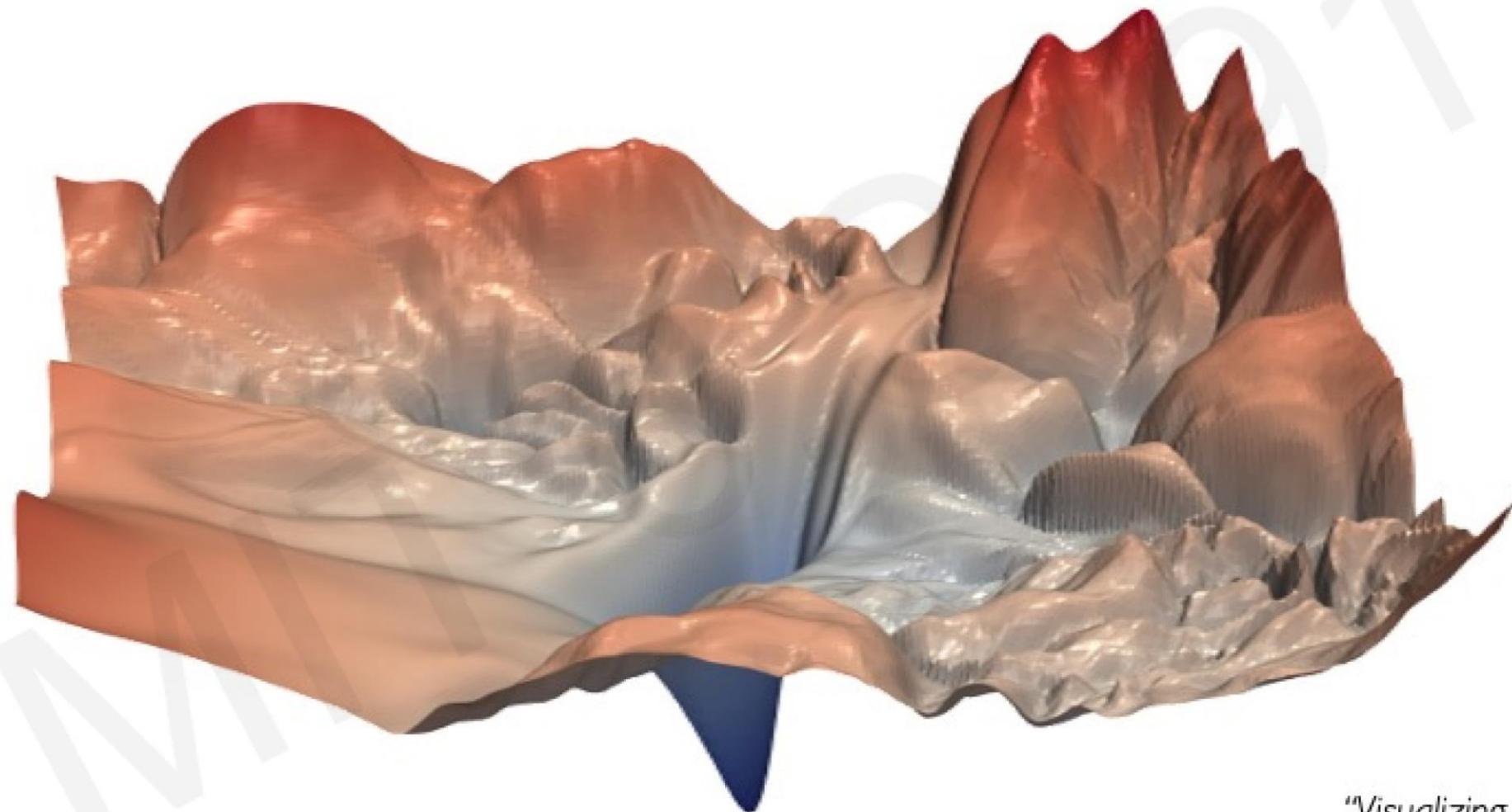


Image from: Li et al. "Visualizing the Loss Landscape of Neural Nets", arXiv:1712.09913

# Neural Networks in Practice: Optimization

# Training Neural Networks is Difficult



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

# Loss Functions Can Be Difficult to Optimize

**Remember:**

Optimization through gradient descent

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

# Loss Functions Can Be Difficult to Optimize

**Remember:**

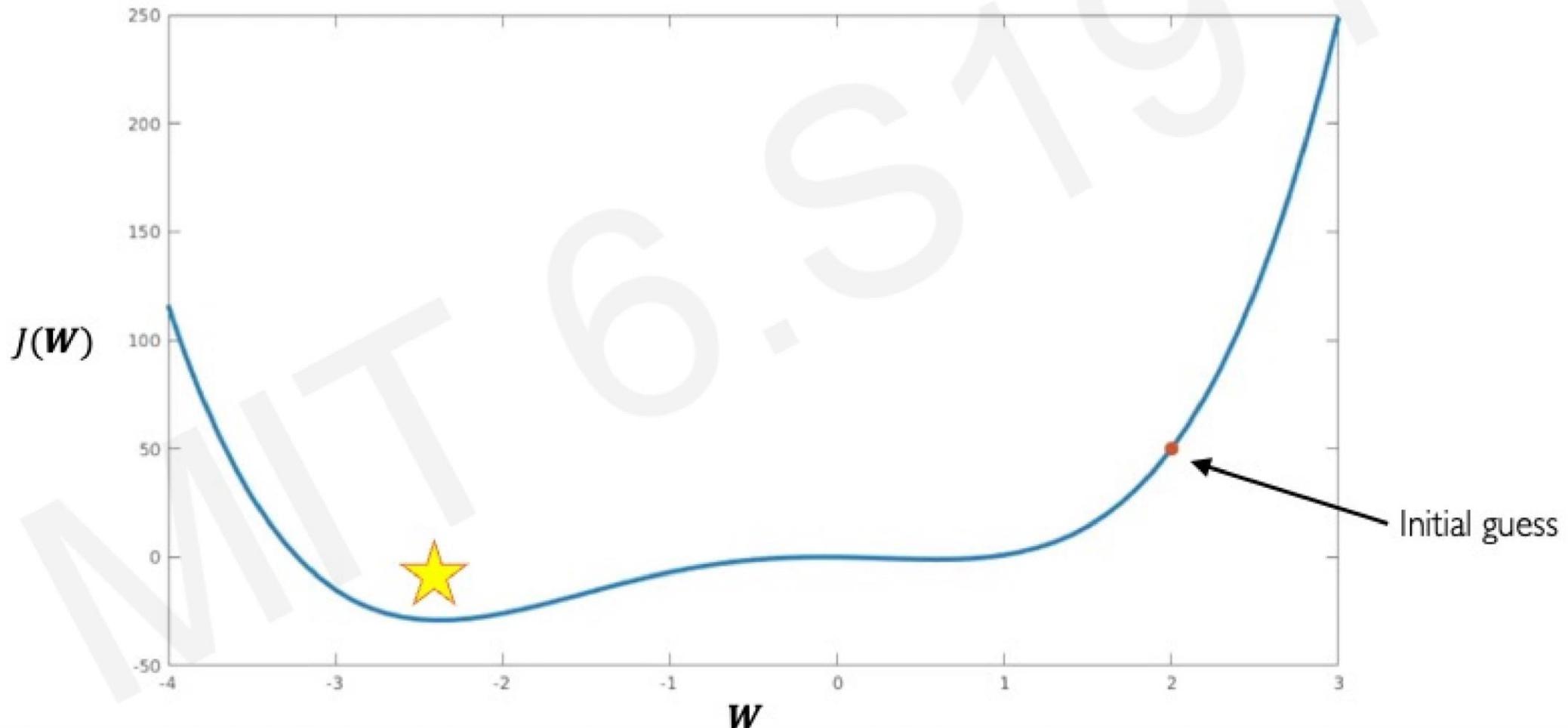
Optimization through gradient descent

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

How can we set the  
learning rate?

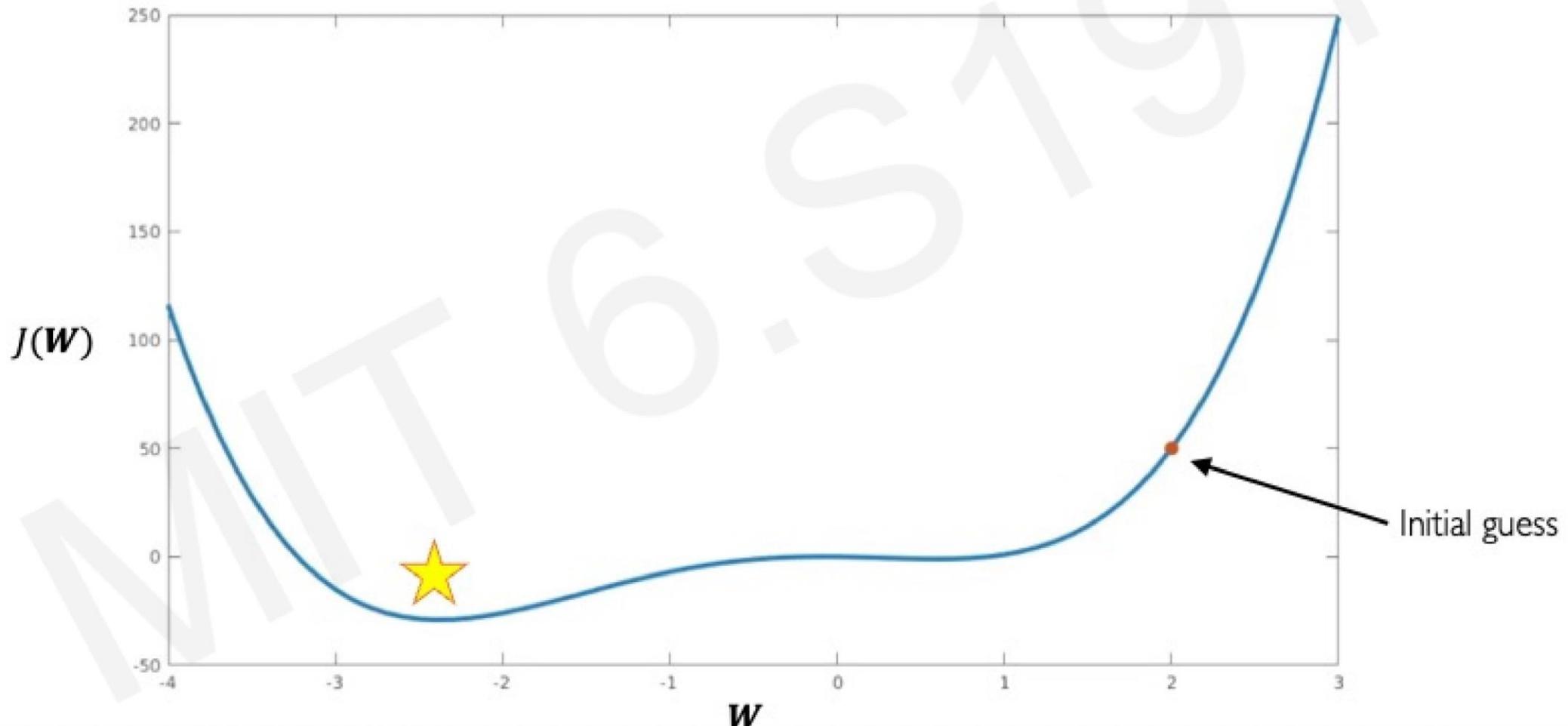
# Setting the Learning Rate

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



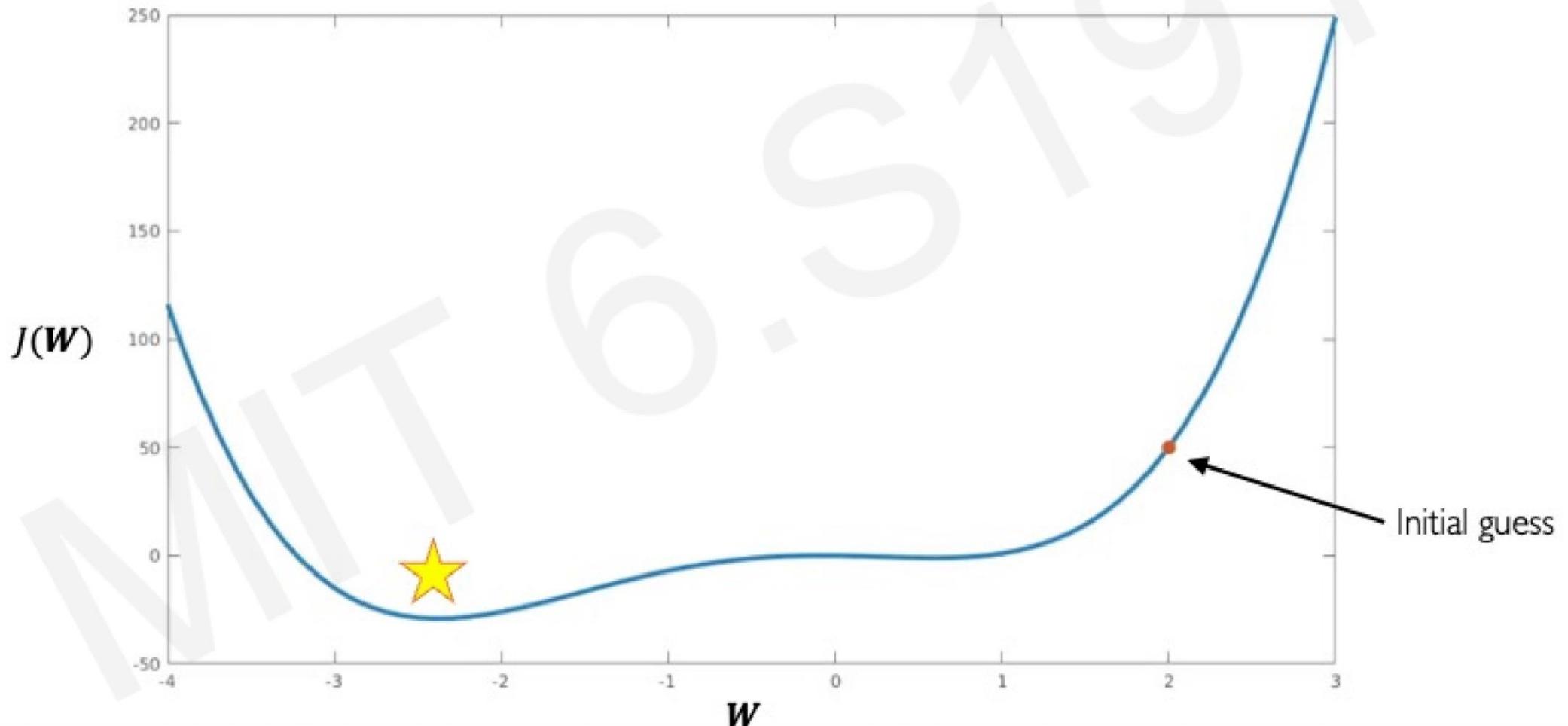
# Setting the Learning Rate

*Large learning rates* overshoot, become unstable and diverge



# Setting the Learning Rate

*Stable learning rates* converge smoothly and avoid local minima



# 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.

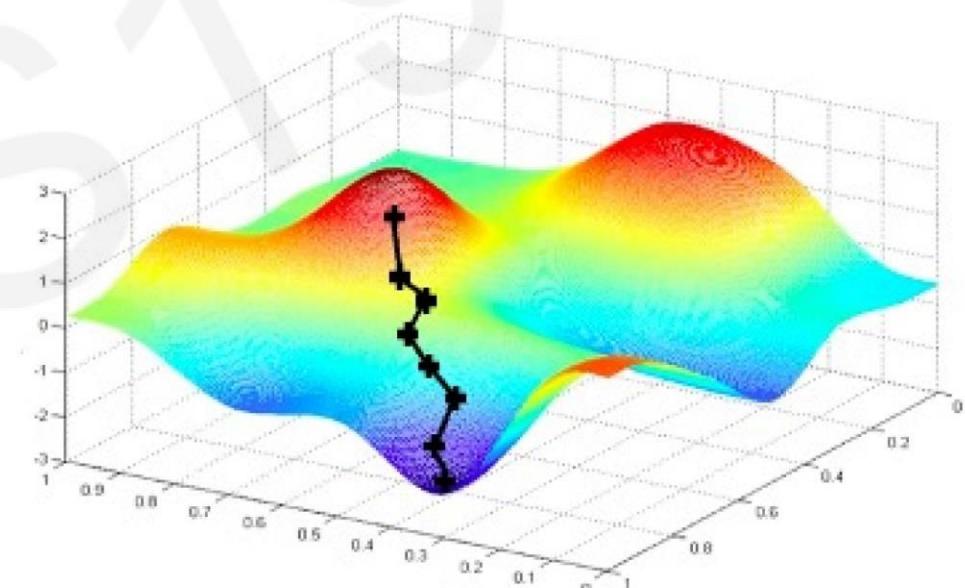
Additional details: <http://ruder.io/optimizing-gradient-descent/>

# Neural Networks in Practice: Mini-batches

# Gradient Descent

## Algorithm

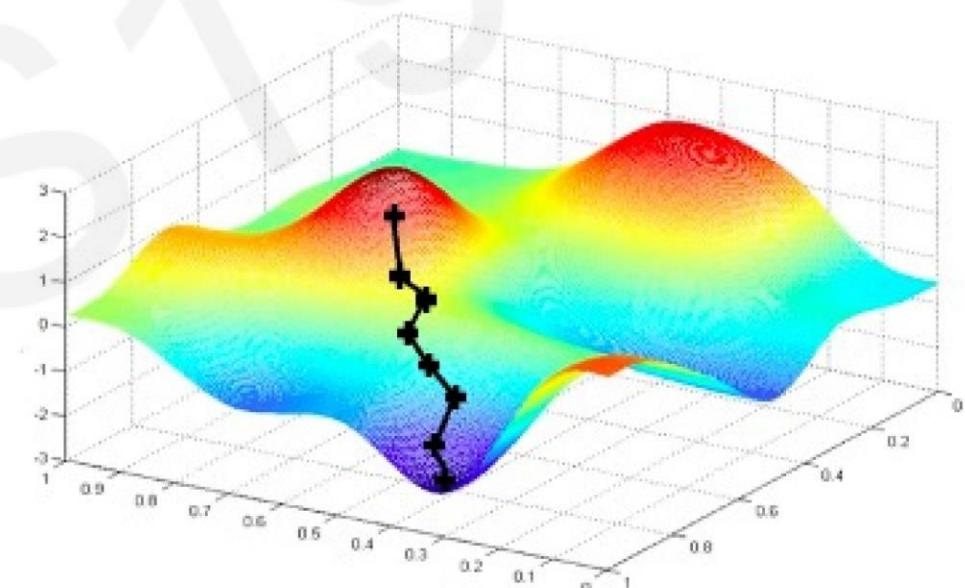
1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient,  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights



# Gradient Descent

## Algorithm

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

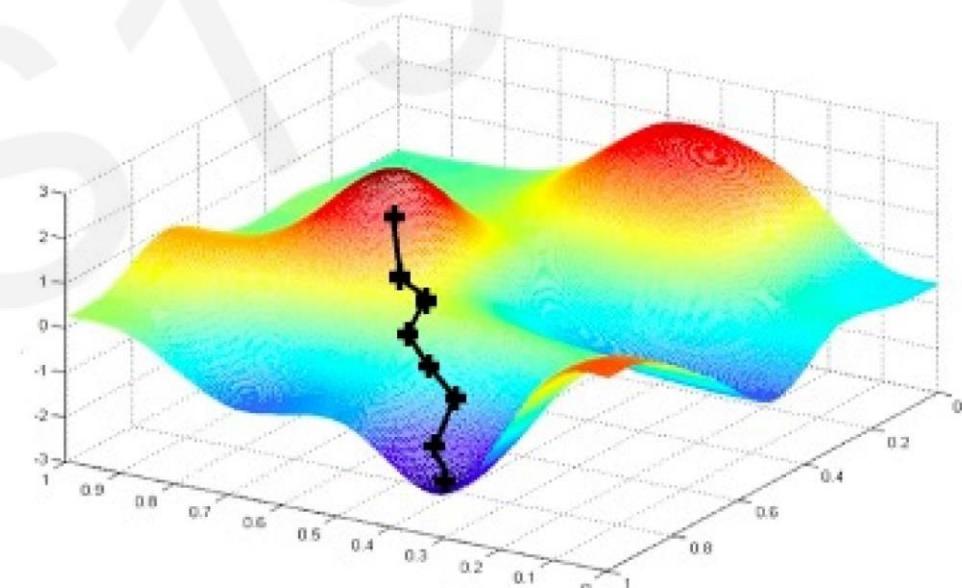


Can be very  
**computationally**  
intensive to compute!

# Stochastic Gradient Descent

## Algorithm

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

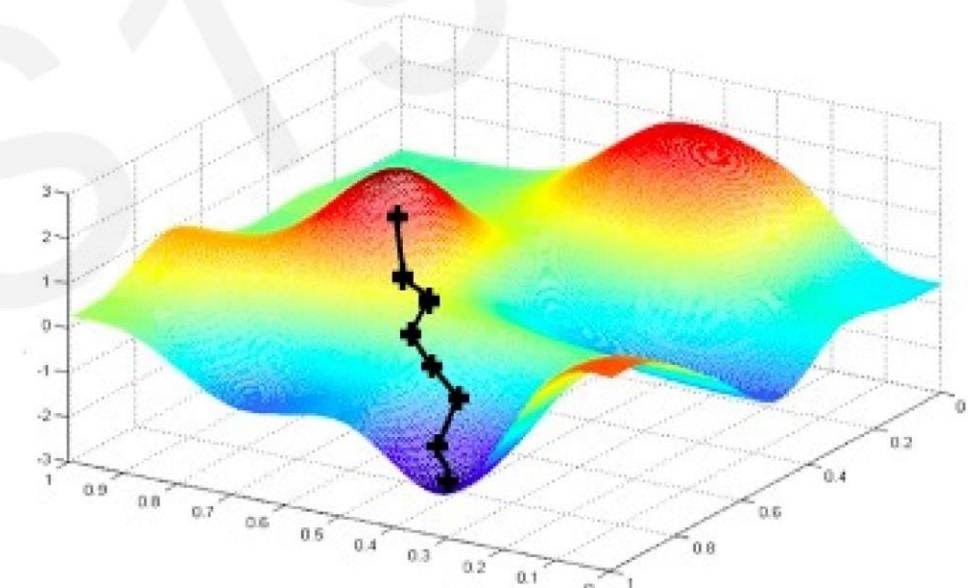


# Stochastic Gradient Descent

## Algorithm

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

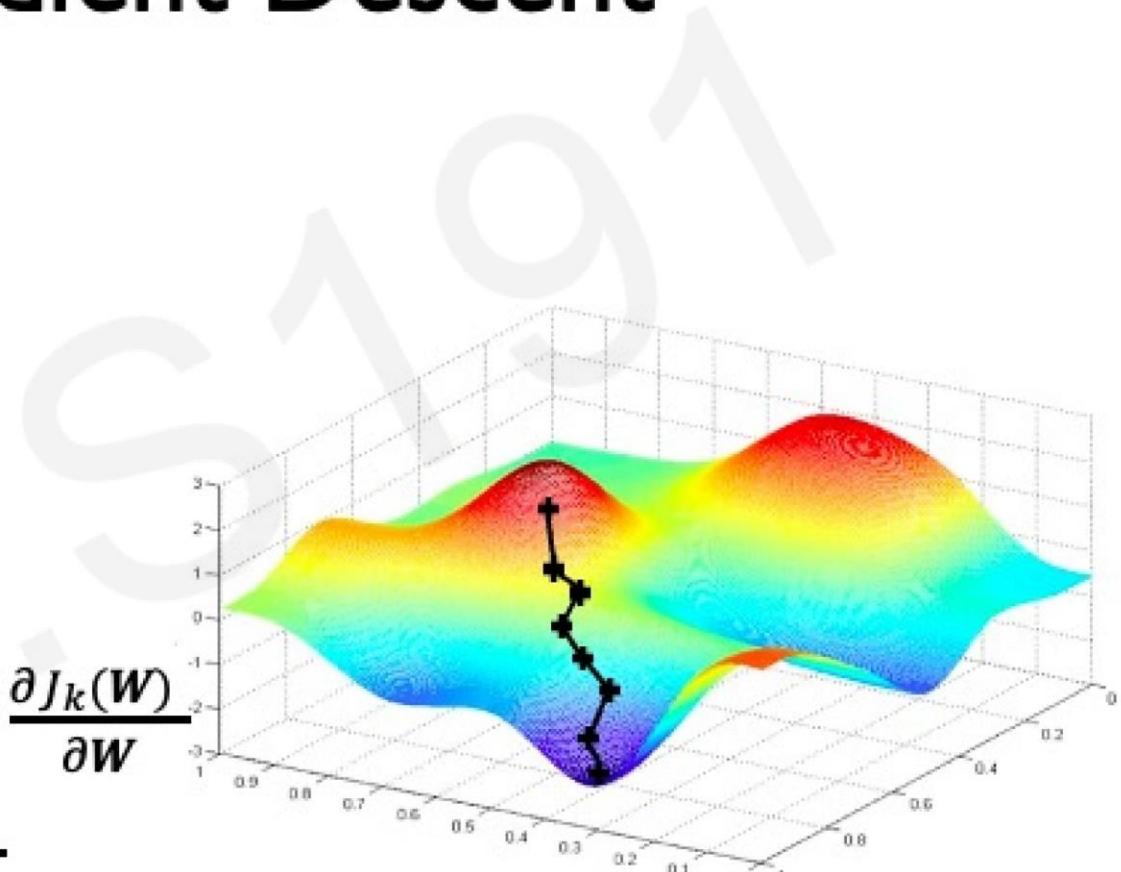
Easy to compute but  
**very noisy** (stochastic)!



# Stochastic Gradient Descent

## Algorithm

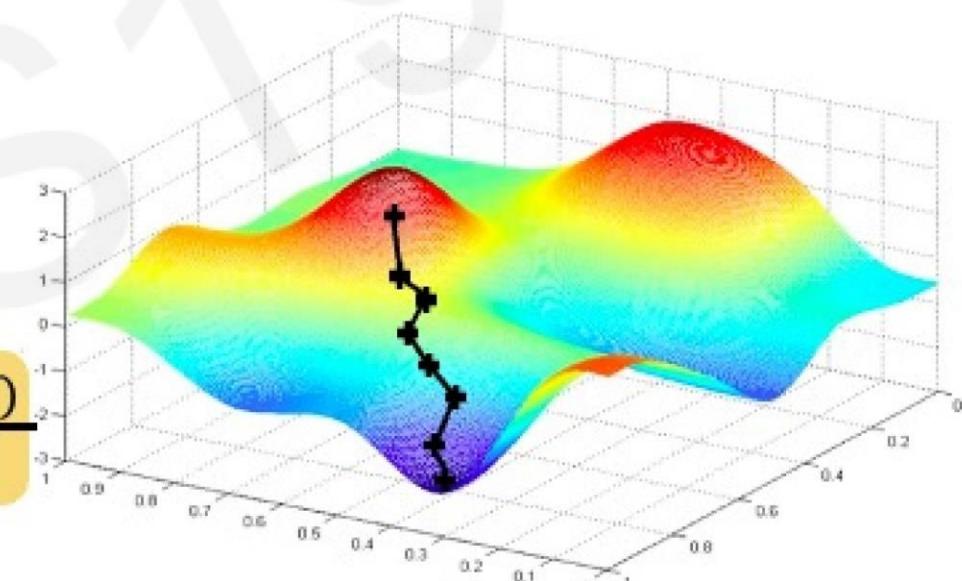
1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of  $B$  data points
4. Compute gradient,  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights, 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



# Stochastic Gradient Descent

## Algorithm

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



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

# Mini-batches while training

## More accurate estimation of gradient

Smoother convergence

Allows for larger learning rates

# Mini-batches while training

**More accurate estimation of gradient**

Smoother convergence

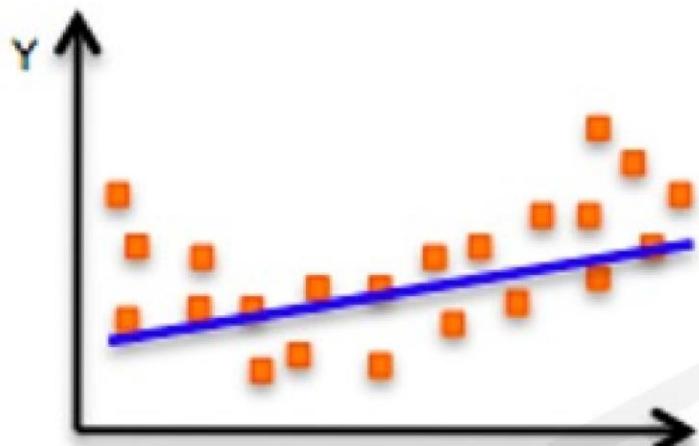
Allows for larger learning rates

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

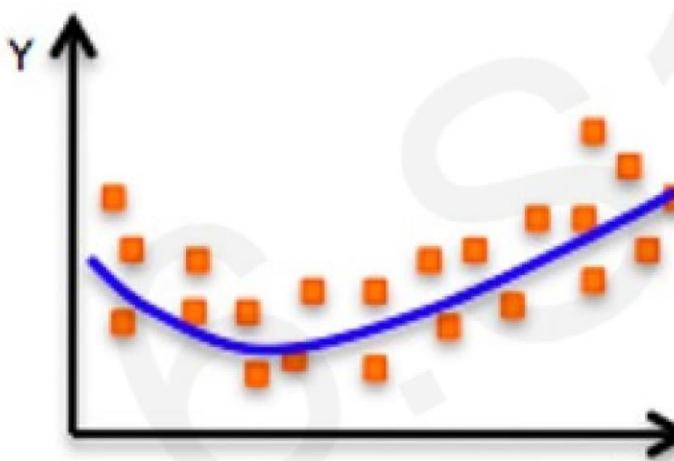
Can parallelize computation + achieve significant speed increases on GPU's

# Neural Networks in Practice: Overfitting

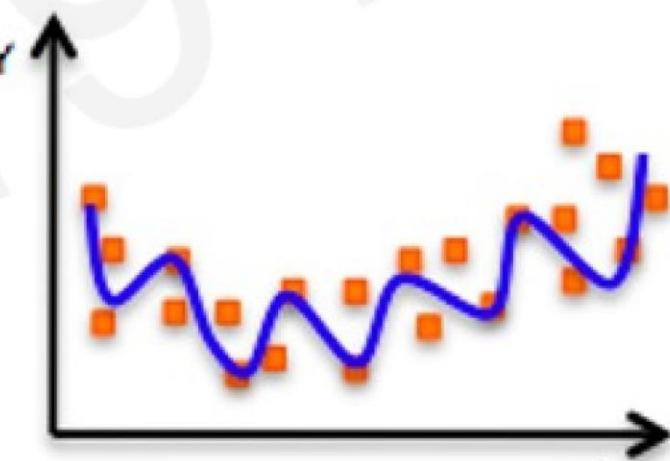
# The Problem of Overfitting



**Ideal fit**



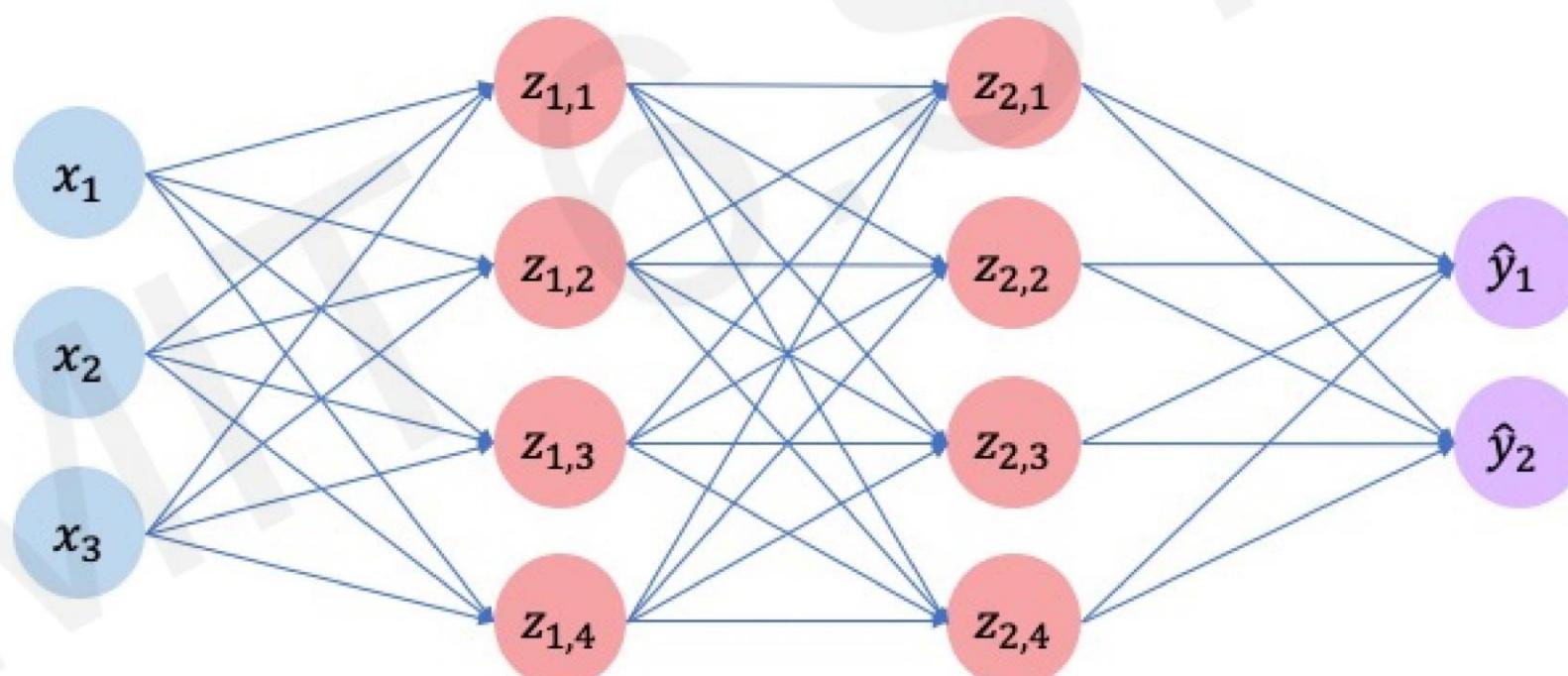
**Underfitting**  
Model does not have capacity  
to fully learn the data



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

# Regularization I: Dropout

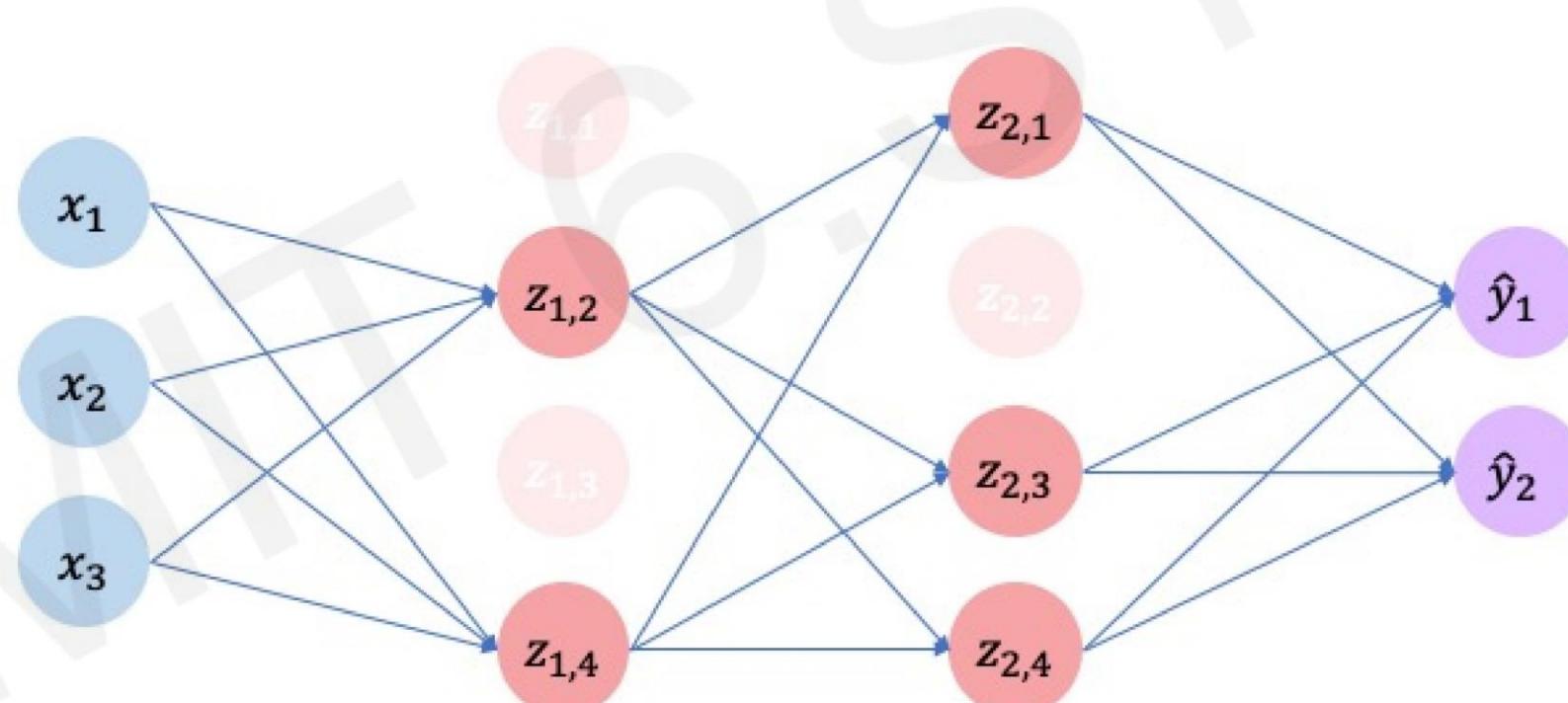
- During training, randomly set some activations to 0



# 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

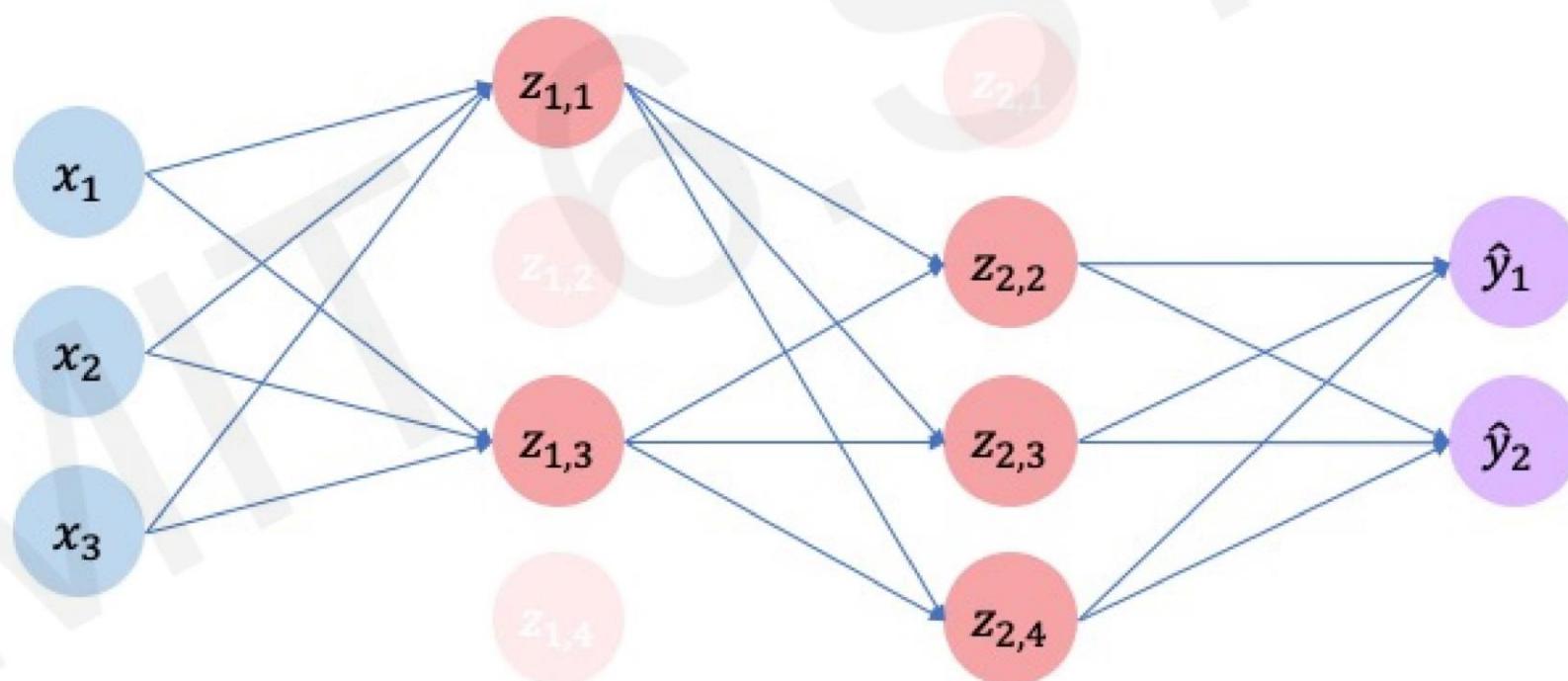
 `tf.keras.layers.Dropout(p=0.5)`



# 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)`



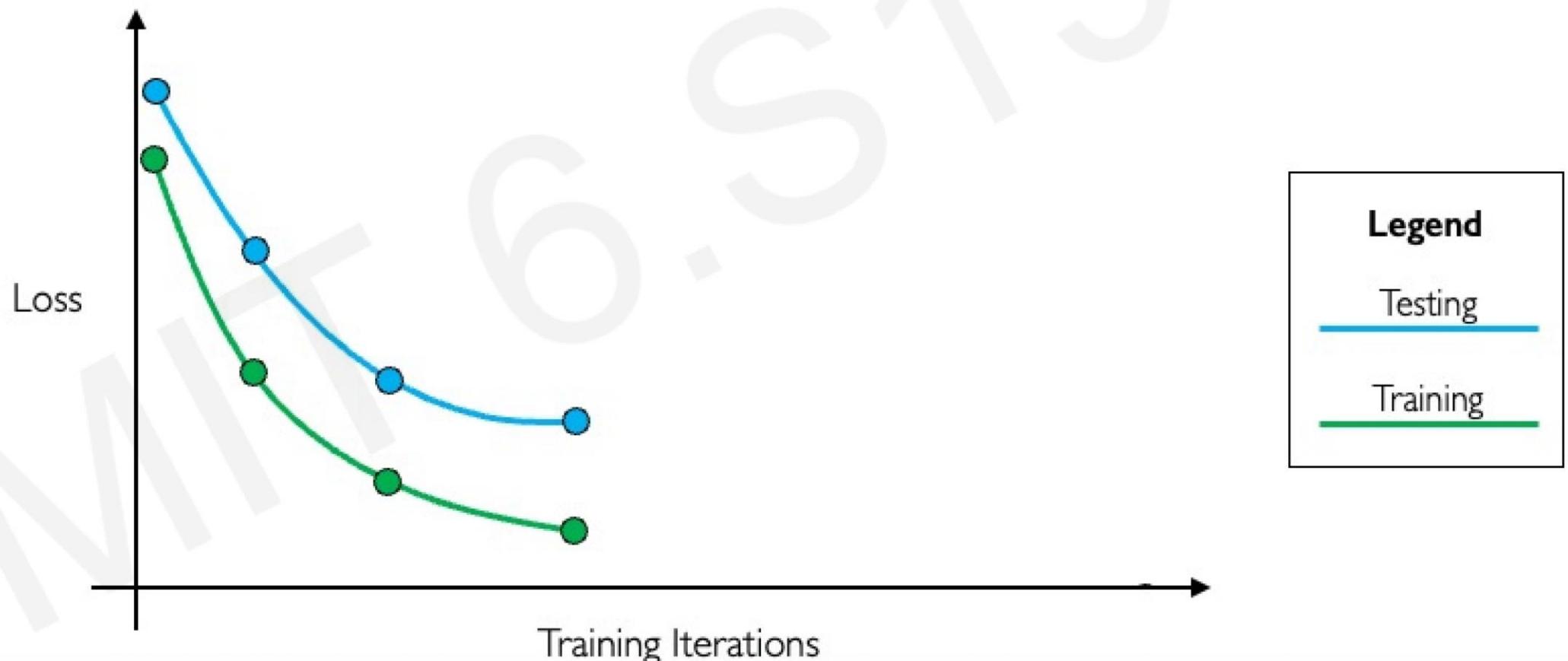
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



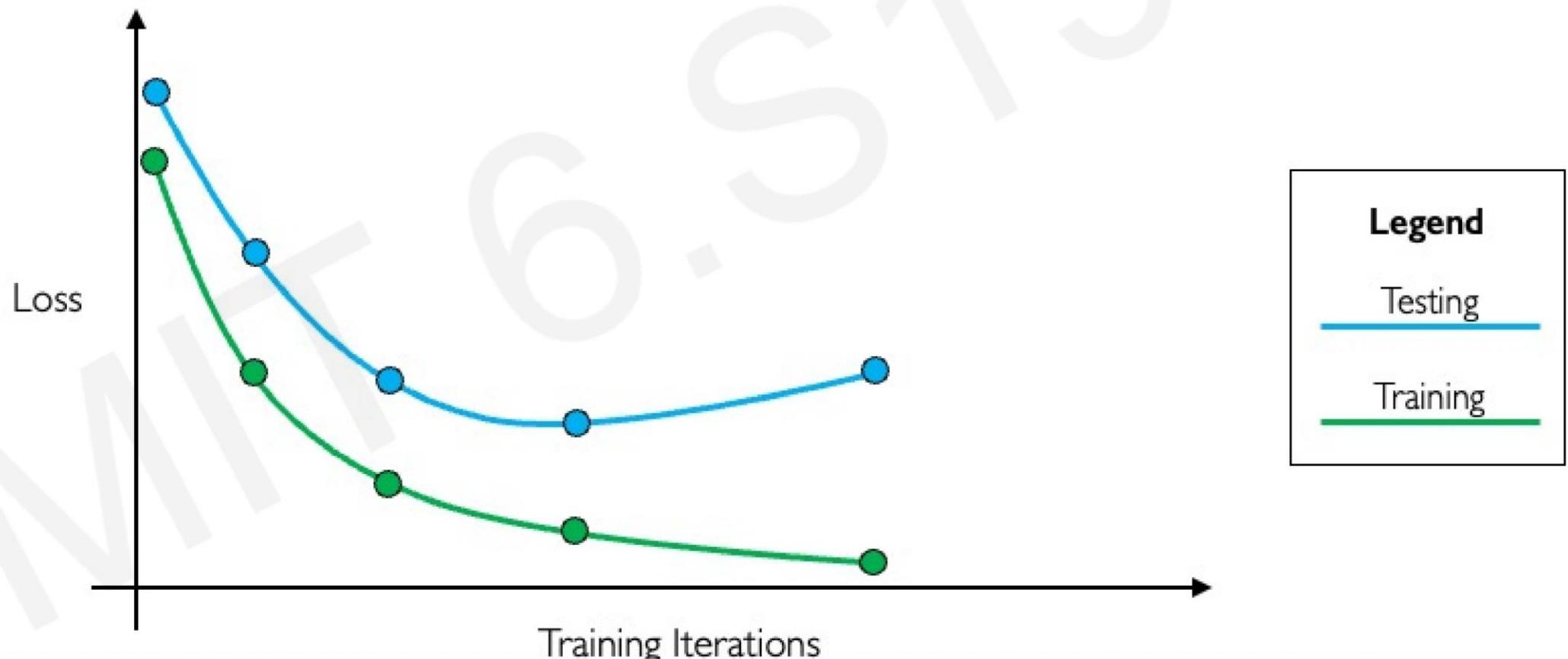
# Regularization 2: Early Stopping

- 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

