

Introduction to Deep Learning

6.S191 Introduction to Deep Learning

introtodeeplearning.com @MITDeepLearning



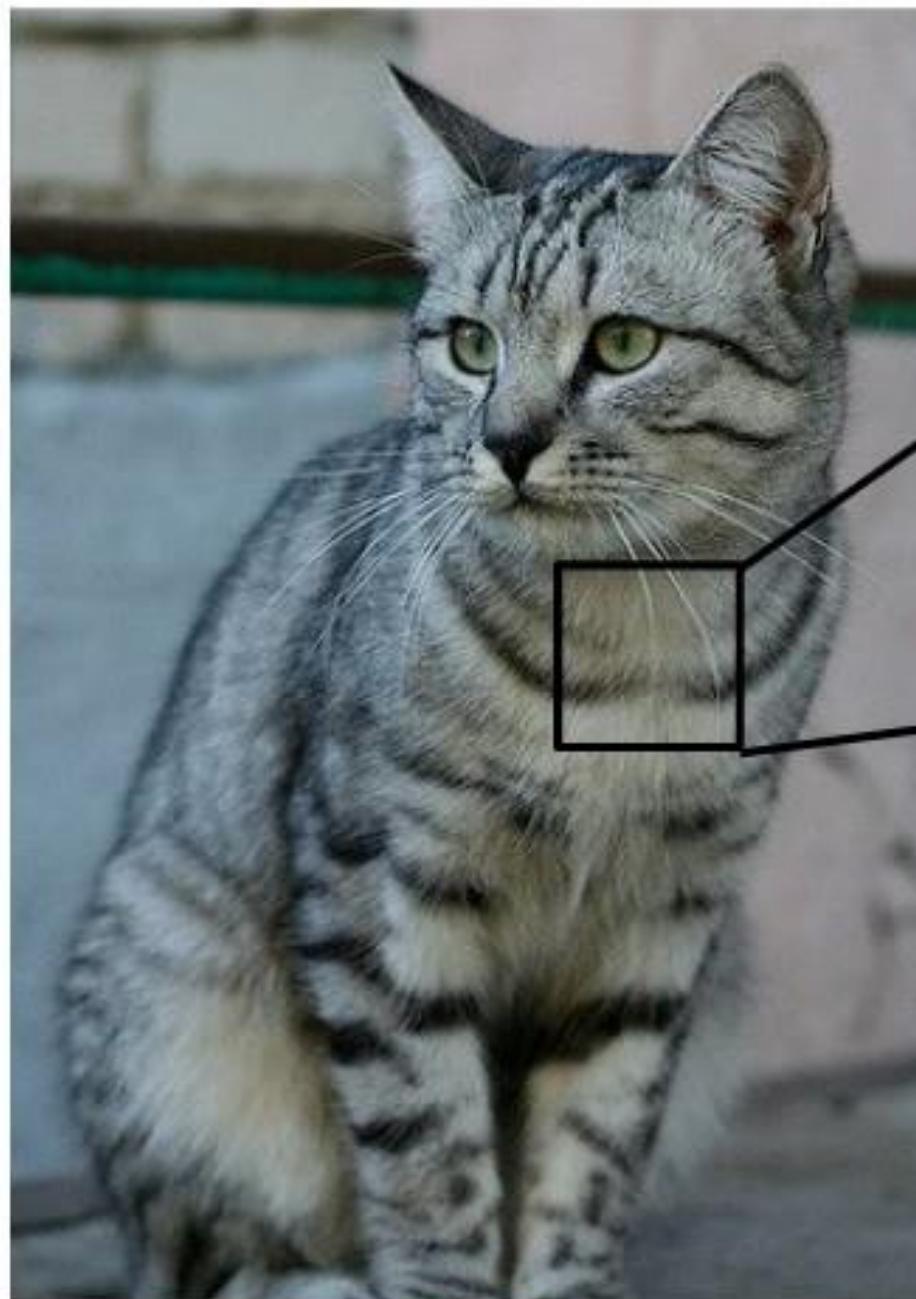
Image Classification: A core task in Computer Vision



cat

[This image](#) by [Nikita](#) is
licensed under [CC-BY 2.0](#)

The Problem: Semantic Gap



[105 112 108 111 104 99 106 99 96 103 112 119 104 97 93 87]
[91 98 102 106 104 79 98 103 99 105 123 136 110 105 94 85]
[76 85 98 105 128 105 87 96 95 99 115 112 106 103 99 85]
[99 81 81 93 120 131 127 100 95 98 102 99 96 93 101 94]
[106 91 61 64 69 91 88 85 101 107 109 98 75 84 96 95]
[114 108 85 55 55 69 64 54 64 87 112 129 98 74 84 91]
[133 137 147 103 65 81 80 65 52 54 74 84 102 93 85 82]
[128 137 144 140 109 95 86 70 62 65 63 63 60 73 86 101]
[125 133 148 137 119 121 117 94 65 79 80 65 54 64 72 98]
[127 125 131 147 133 127 126 131 111 96 89 75 61 64 72 84]
[115 114 109 123 150 148 131 118 113 109 100 92 74 65 72 78]
[89 93 90 97 108 147 131 118 113 114 113 109 106 95 77 80]
[63 77 86 81 77 79 102 123 117 115 117 125 125 130 115 87]
[62 65 82 89 78 71 80 101 124 126 119 101 107 114 131 119]
[63 65 75 88 89 71 62 81 120 138 135 105 81 98 110 118]
[87 65 71 87 106 95 69 45 76 130 126 107 92 94 105 112]
[118 97 82 86 117 123 116 66 41 51 95 93 89 95 102 107]
[164 146 112 80 82 120 124 104 76 48 45 66 88 101 102 109]
[157 170 157 120 93 86 114 132 112 97 69 55 70 82 99 94]
[130 128 134 161 139 100 109 118 121 134 114 87 65 53 69 86]
[128 112 96 117 150 144 120 115 104 107 102 93 87 81 72 79]
[123 107 96 86 83 112 153 149 122 109 104 75 80 107 112 99]
[122 121 102 80 82 86 94 117 145 148 153 102 58 78 92 107]
[122 164 148 103 71 56 78 83 93 103 119 139 102 61 69 84]]

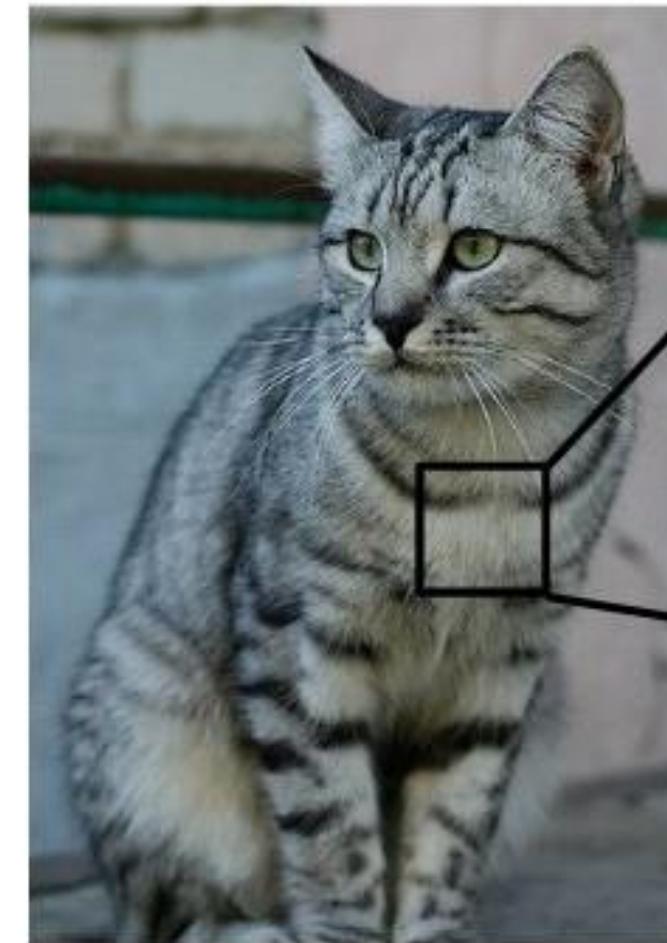
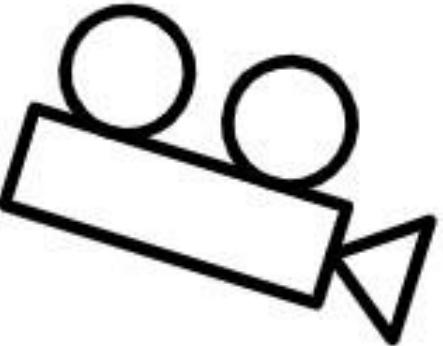
What the computer sees

An image is a tensor of integers between [0, 255]:

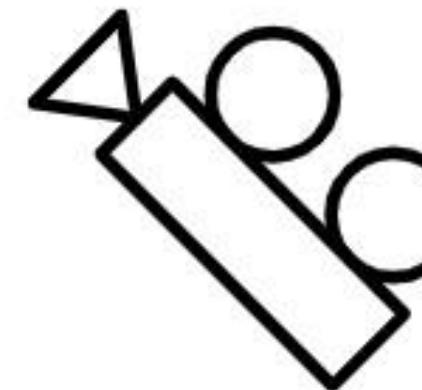
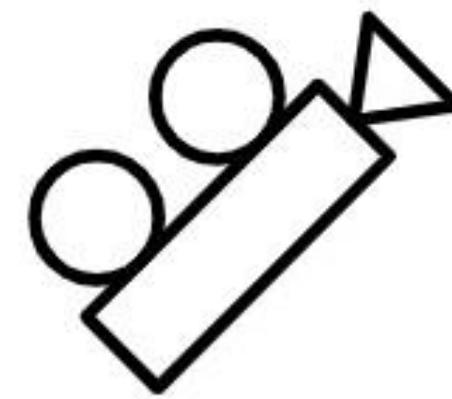
e.g. 800 x 600 x 3
(3 channels RGB)

[This image](#) by [Nikita](#) is
licensed under [CC-BY 2.0](#)

Challenges: Viewpoint variation



[185 112 108 111 104 99 186 99 96 183 112 119 104 97 93 87]
[91 98 102 105 104 79 90 183 99 185 123 136 110 105 94 85]
[76 85 98 105 128 105 87 96 95 99 115 112 106 103 99 85]
[99 81 81 93 128 131 127 186 95 98 182 99 96 93 101 94]
[106 91 61 64 60 91 88 85 181 187 189 98 75 84 96 95]
[114 108 85 55 55 59 64 54 64 87 112 129 98 74 84 91]
[133 137 147 105 65 81 88 65 52 54 74 84 102 93 85 82]
[128 137 144 140 109 95 86 78 62 65 63 63 60 73 85 101]
[125 133 148 137 119 121 117 94 65 79 80 65 54 64 72 98]
[127 125 131 147 133 127 126 131 111 96 89 75 61 64 72 84]
[115 114 109 123 158 148 131 118 113 189 180 92 74 65 72 78]
[89 93 98 97 108 147 131 118 113 114 113 189 105 95 77 88]
[63 77 86 81 77 79 102 123 117 115 117 125 125 130 115 87]
[62 65 82 80 78 71 88 101 124 126 119 101 107 114 131 119]
[63 65 75 88 89 71 62 81 128 138 135 105 81 98 118 118]
[87 65 71 87 106 95 60 45 76 130 126 107 92 94 105 112]
[118 97 82 86 117 123 116 66 41 51 95 93 89 95 102 107]
[164 146 112 88 82 128 124 184 76 48 45 66 88 101 102 100]
[157 170 157 128 93 86 114 132 112 97 69 55 70 82 99 94]
[138 128 134 161 139 100 100 118 121 134 114 87 65 53 69 86]
[128 112 96 117 158 144 128 115 184 187 102 93 87 81 72 79]
[123 107 96 86 83 112 153 149 122 189 184 75 80 107 112 99]
[122 121 102 88 82 86 94 117 145 148 153 102 58 78 92 107]
[122 164 148 103 71 56 78 83 93 183 119 139 102 61 69 84]



All pixels change when
the camera moves!

Challenges: Background Clutter



[This image is CC0 1.0 public domain](#)



[This image is CC0 1.0 public domain](#)

Challenges: Illumination



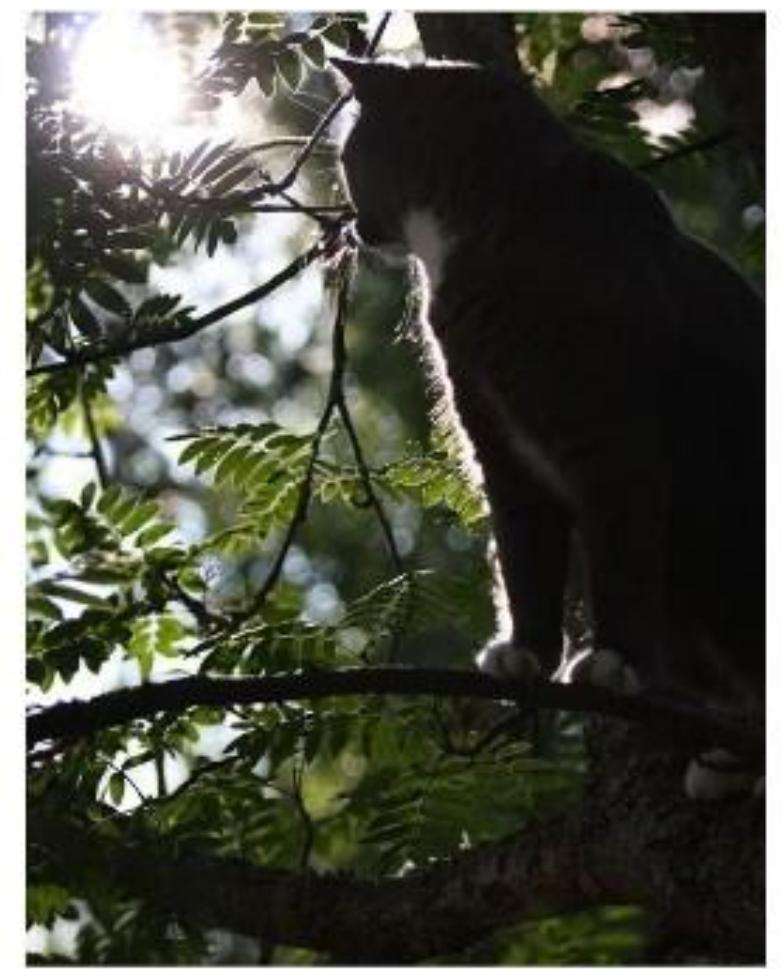
[This image is CC0 1.0 public domain](#)



[This image is CC0 1.0 public domain](#)



[This image is CC0 1.0 public domain](#)



[This image is CC0 1.0 public domain](#)

Challenges: Occlusion



[This image](#) is CC0 1.0 public domain

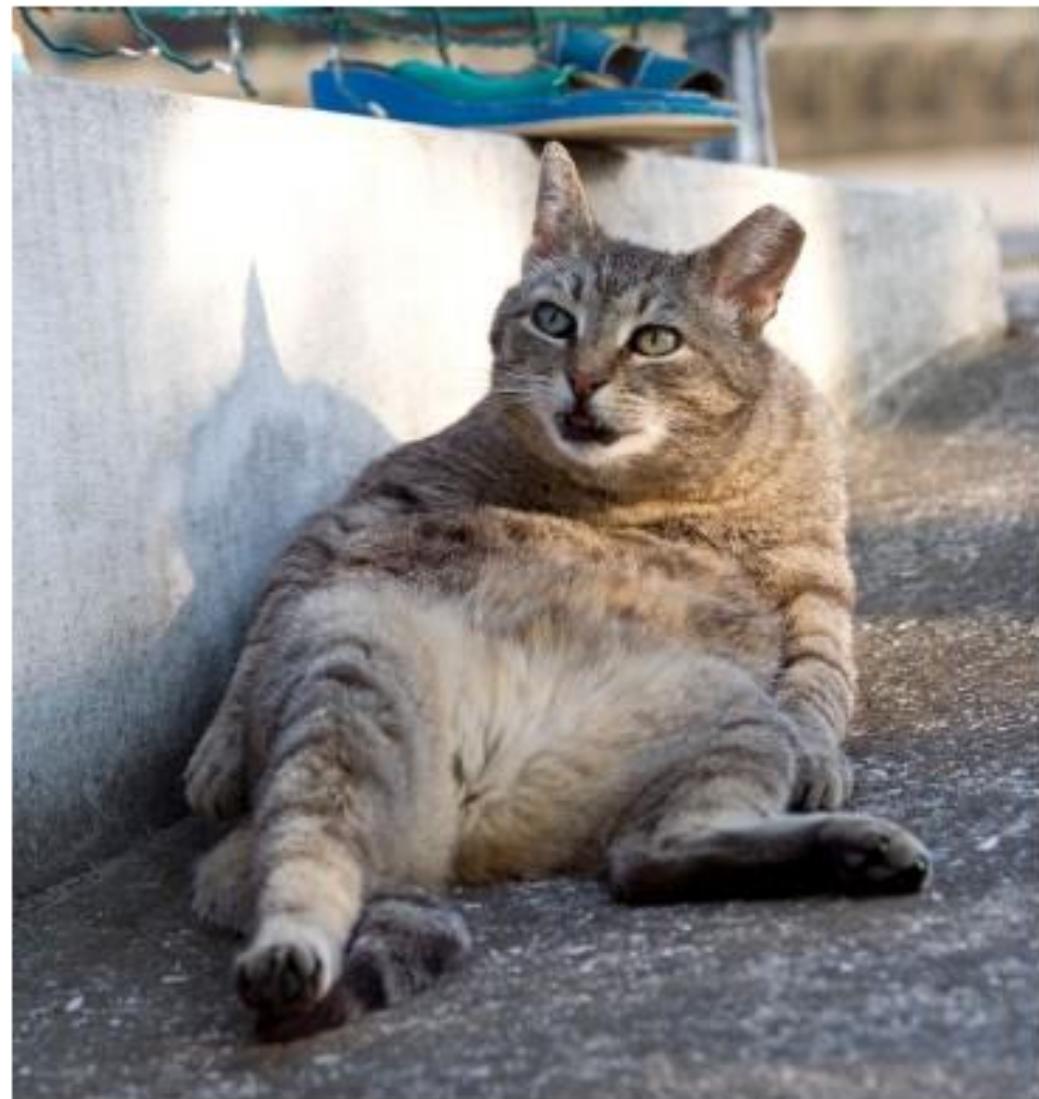


[This image](#) is CC0 1.0 public domain

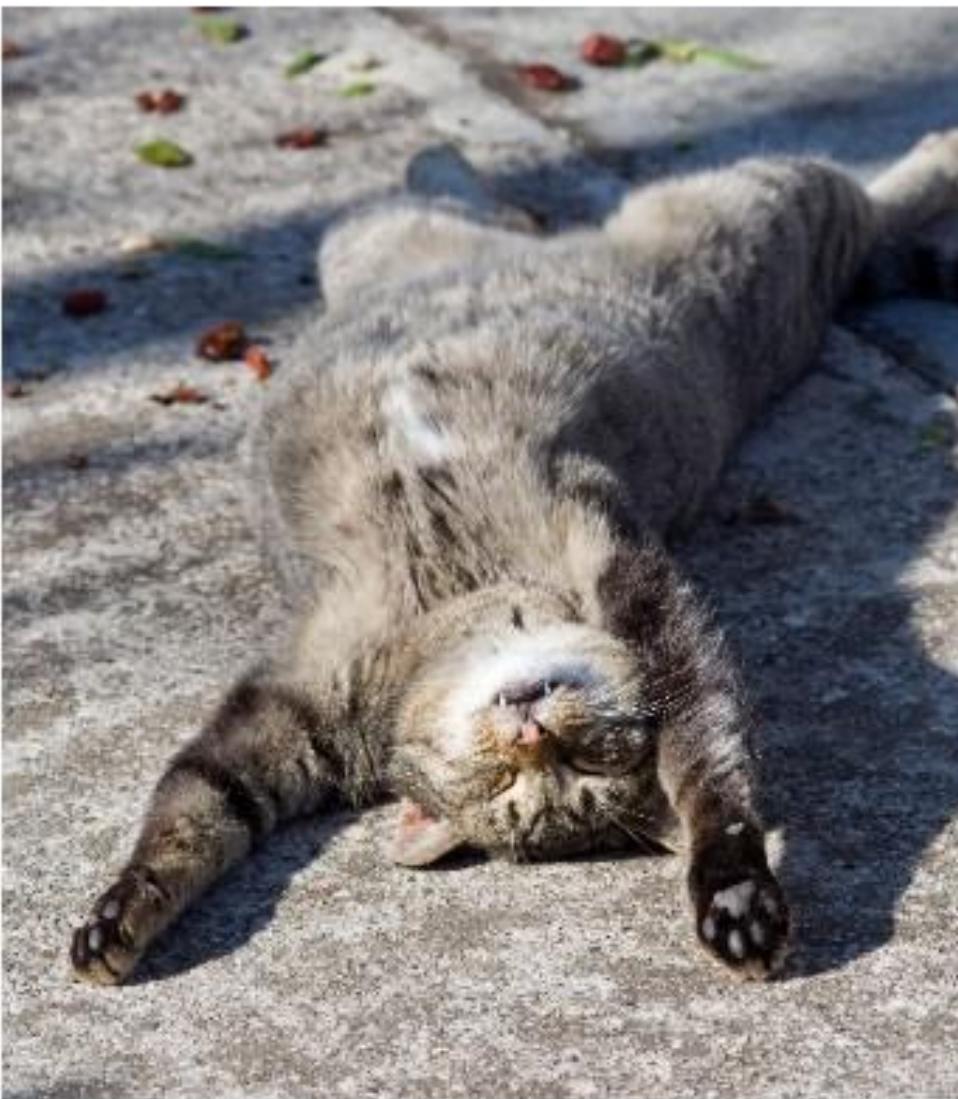


[This image](#) by [jonsson](#) is licensed under CC-BY 2.0

Challenges: Deformation



[This image](#) by [Umberto Salvagnin](#)
is licensed under [CC-BY 2.0](#)



[This image](#) by [Umberto Salvagnin](#)
is licensed under [CC-BY 2.0](#)



[This image](#) by [sare bear](#) is
licensed under [CC-BY 2.0](#)



[This image](#) by [Tom Thai](#) is
licensed under [CC-BY 2.0](#)

Challenges: Intraclass variation



[This image is CC0 1.0 public domain](#)

An image classifier

```
def classify_image(image):  
    # Some magic here?  
    return class_label
```

Unlike e.g. sorting a list of numbers,
no obvious way to hard-code the algorithm for
recognizing a cat, or other classes.

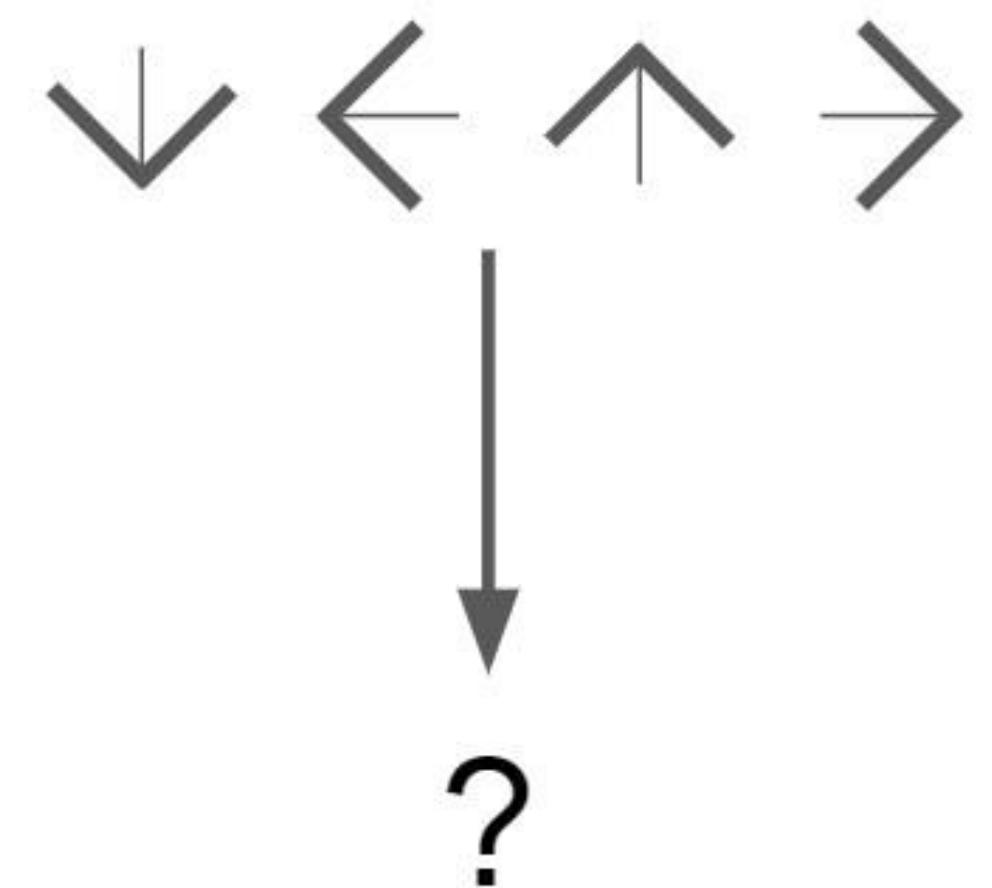
Attempts have been made



Find edges



Find corners



Machine Learning: Data-Driven Approach

1. Collect a dataset of images and labels
2. Use Machine Learning algorithms to train a classifier
3. Evaluate the classifier on new images

Example training set

```
def train(images, labels):
    # Machine learning!
    return model

def predict(model, test_images):
    # Use model to predict labels
    return test_labels
```

airplane



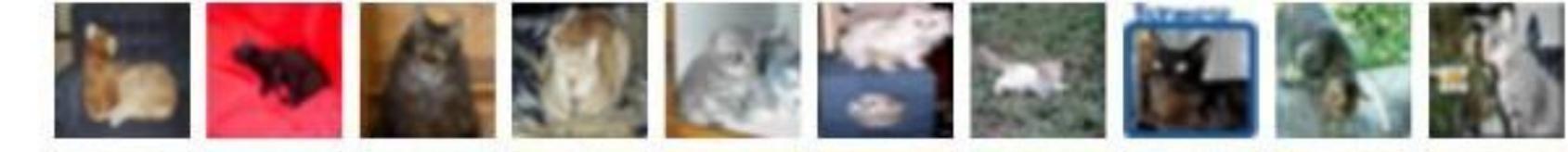
automobile



bird



cat



deer



What is Deep Learning?

ARTIFICIAL INTELLIGENCE

Any technique that enables computers to mimic human behavior



MACHINE LEARNING

Ability to learn without explicitly being programmed



DEEP LEARNING

Extract patterns from data using neural networks



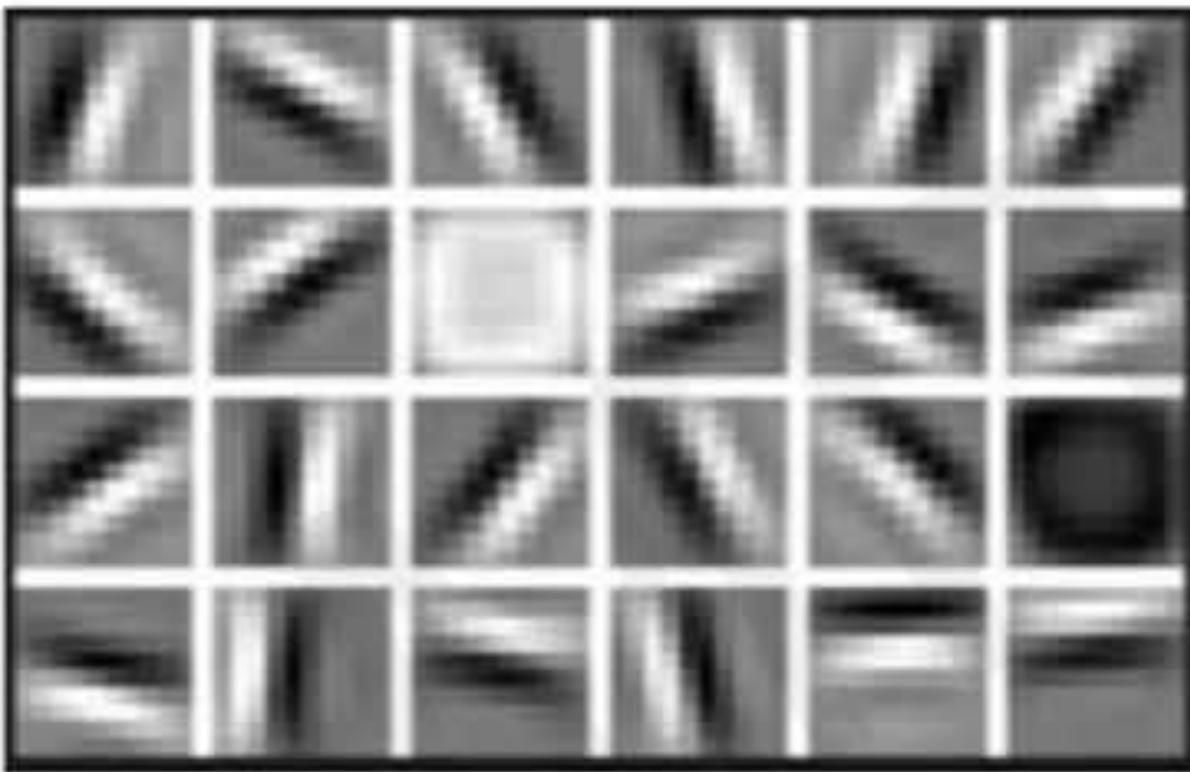
Why Deep Learning and Why Now?

Why Deep Learning?

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

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

Low Level Features



Lines & Edges

Mid Level Features



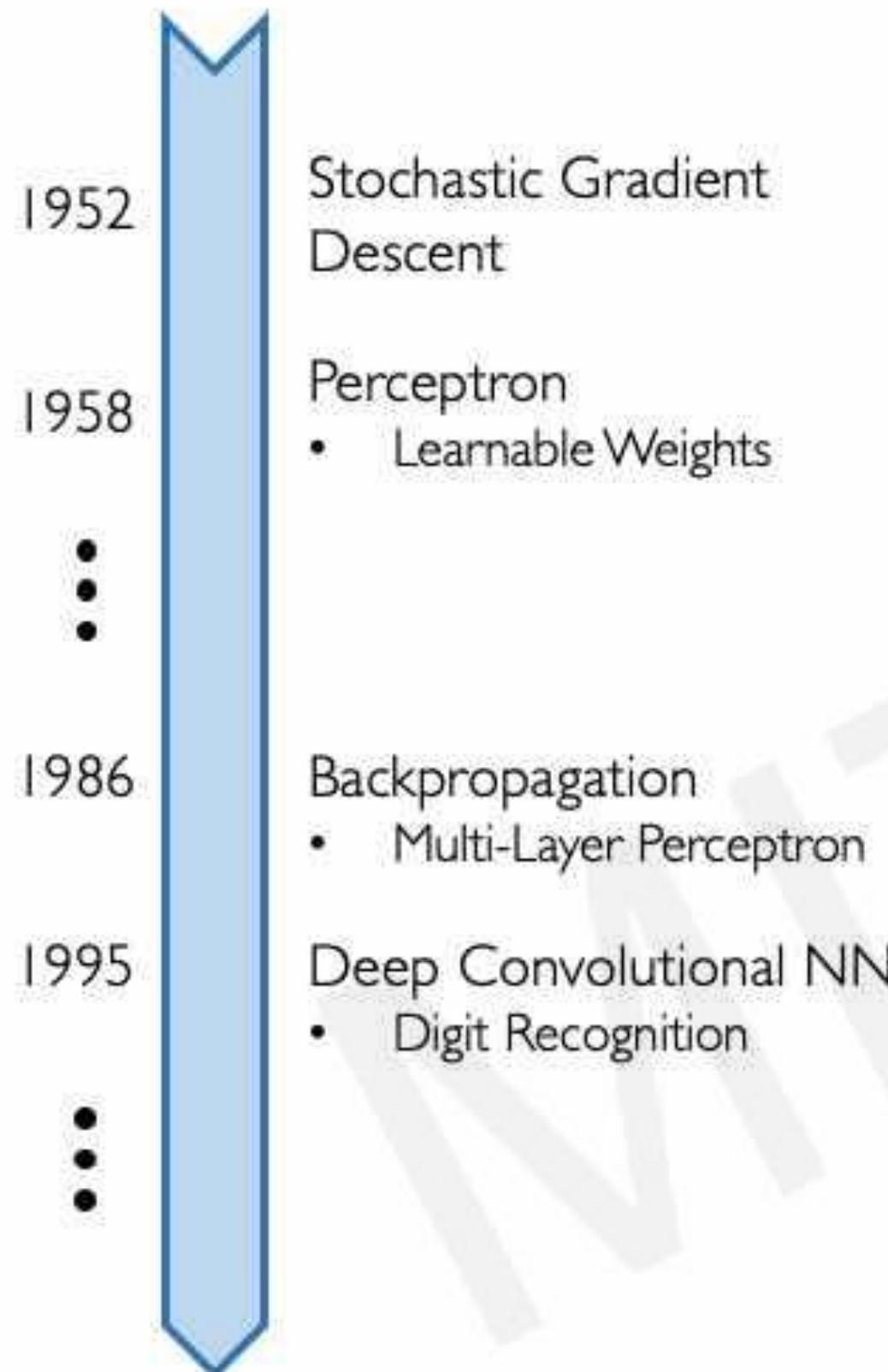
Eyes & Nose & Ears

High Level Features



Facial Structure

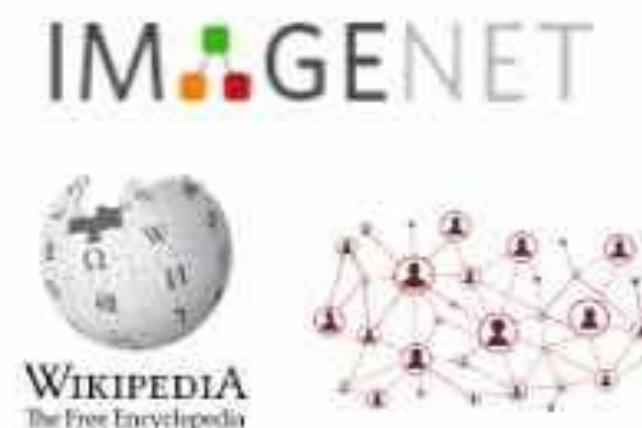
Why Now?



Neural Networks date back decades, so why the resurgence?

I. Big Data

- Larger Datasets
- Easier Collection & Storage



2. Hardware

- Graphics Processing Units (GPUs)
- Massively Parallelizable



3. Software

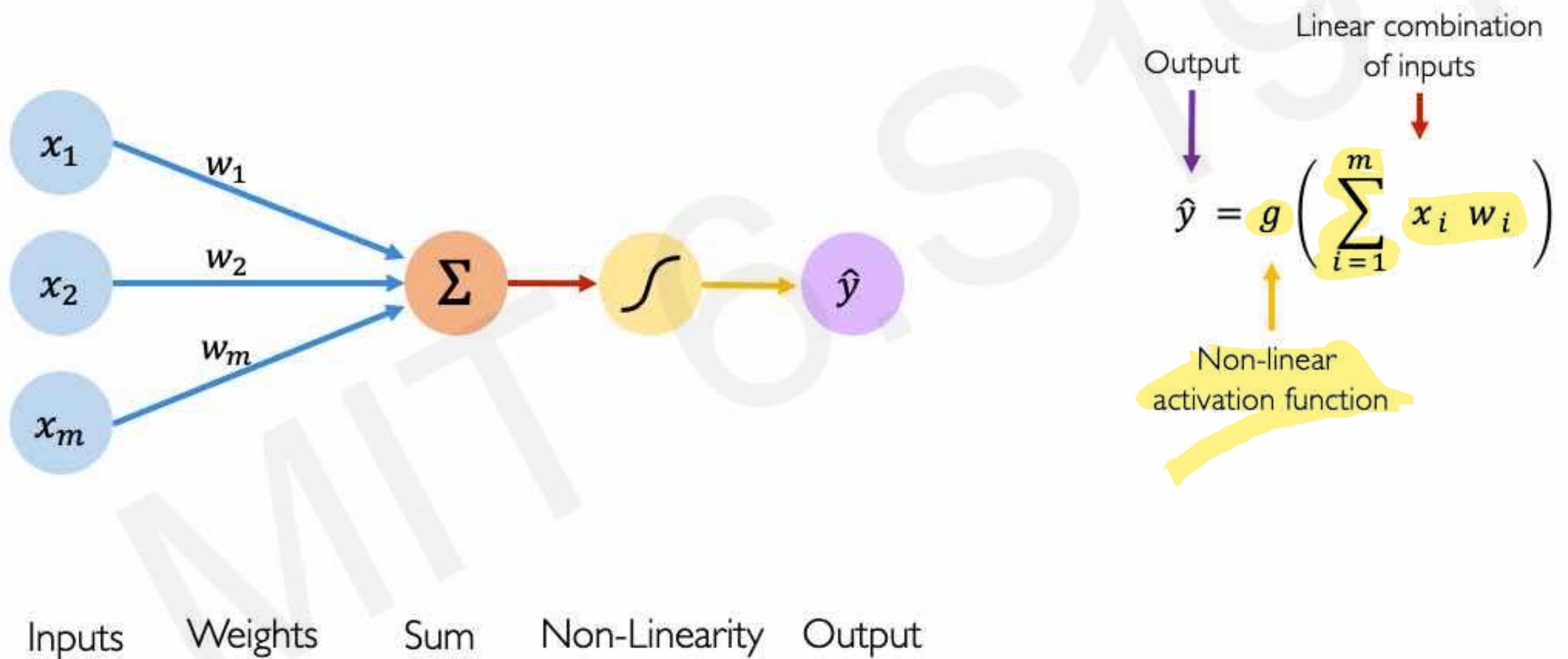
- Improved Techniques
- New Models
- Toolboxes



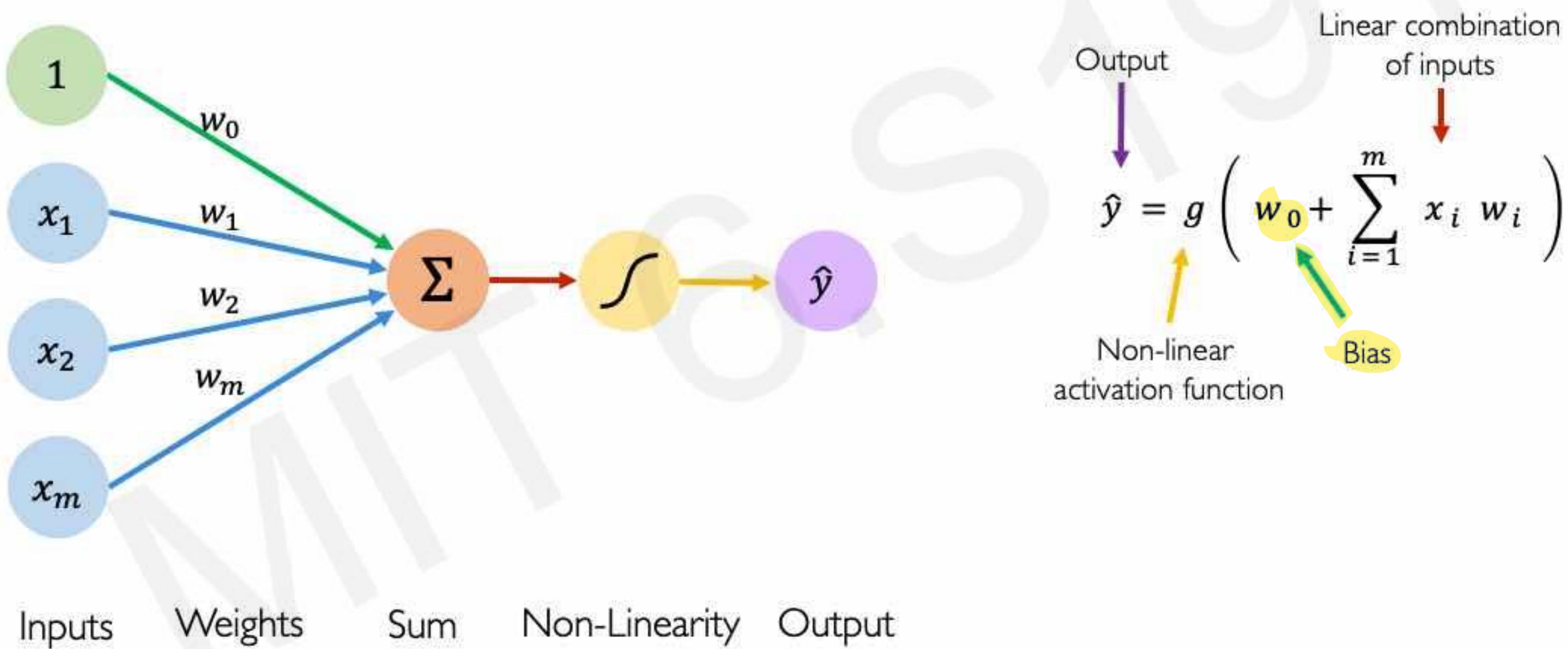
The Perceptron

The structural building block of deep learning

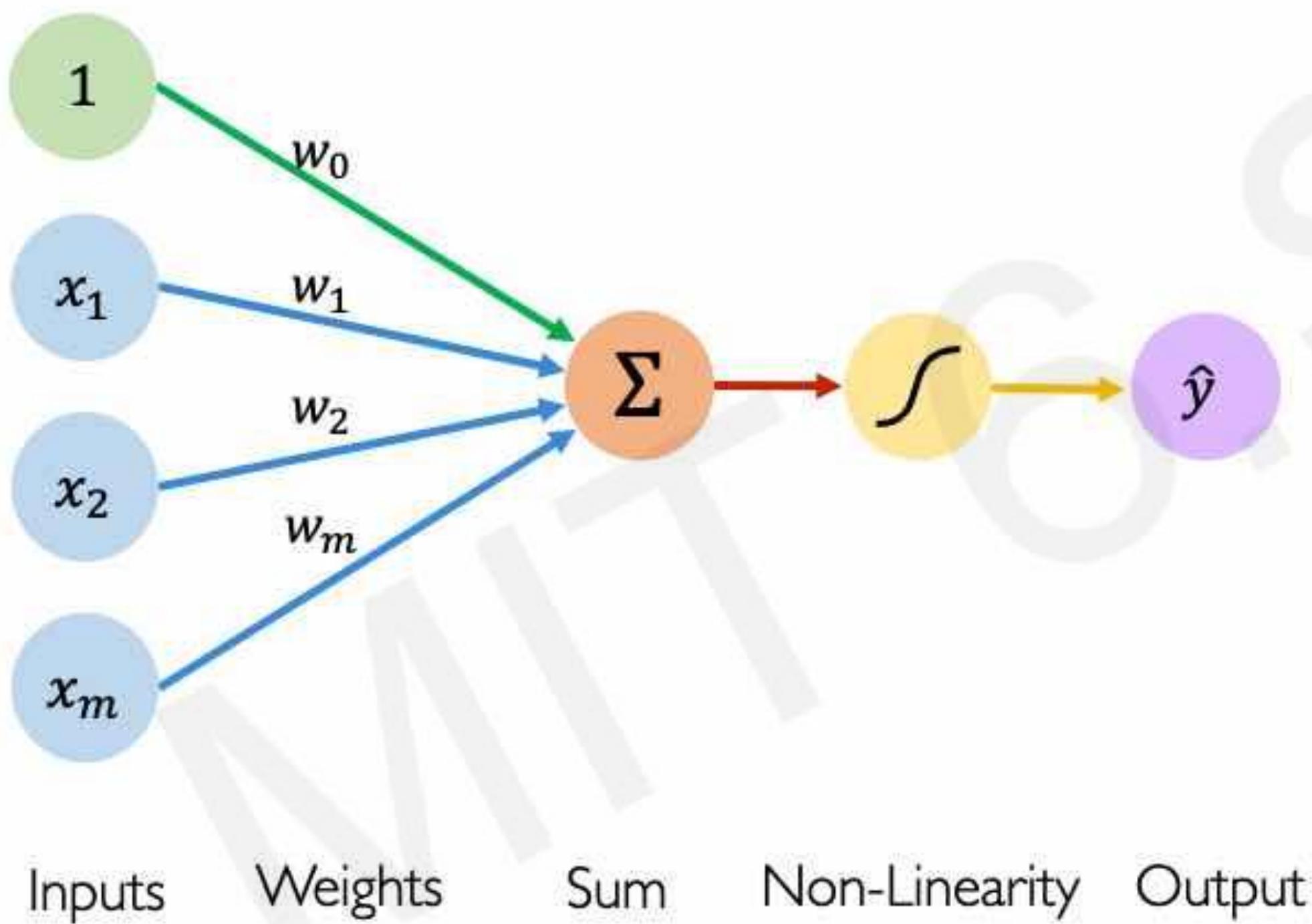
The Perceptron: Forward Propagation



The Perceptron: Forward Propagation



The Perceptron: Forward Propagation

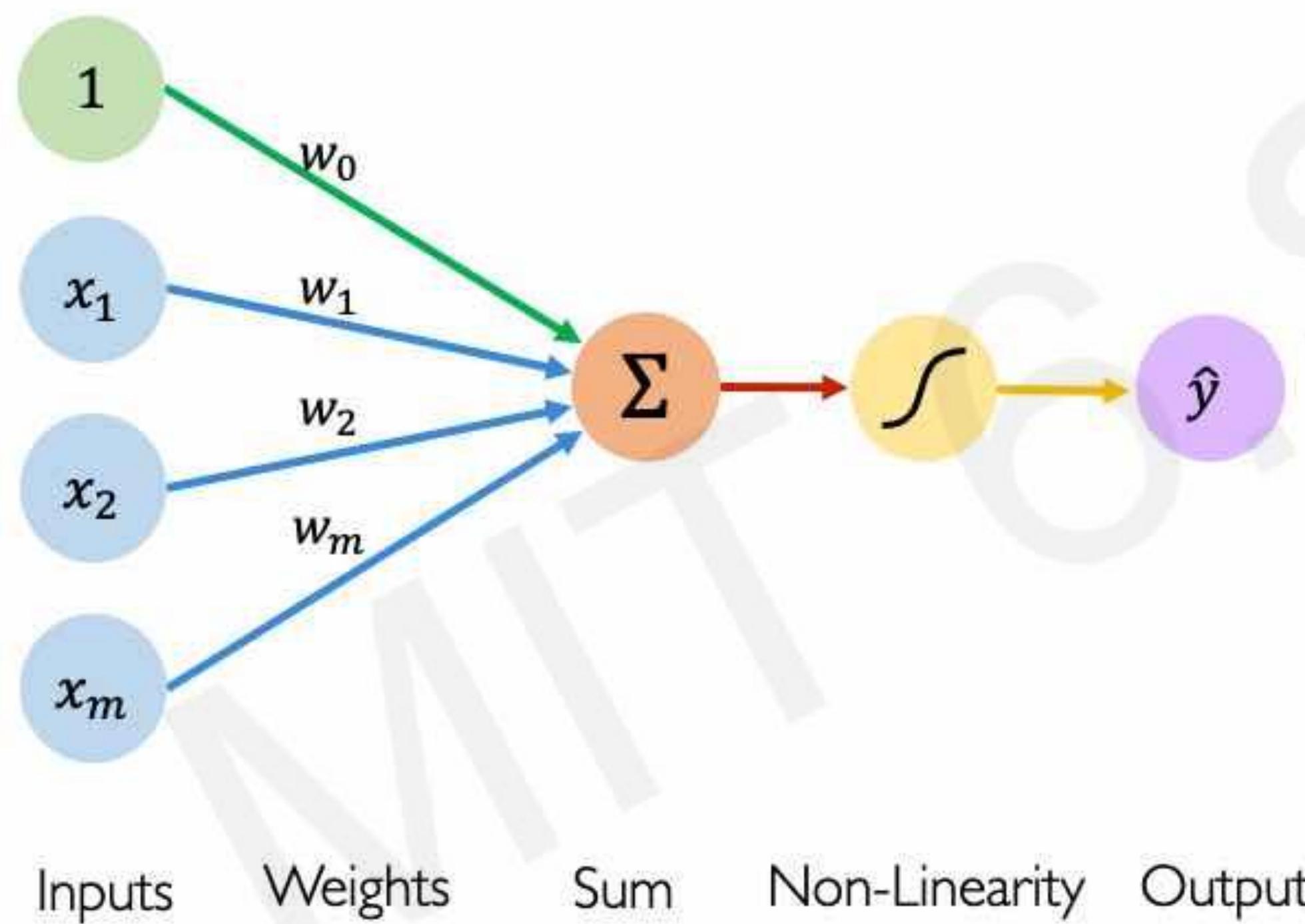


$$\hat{y} = g \left(w_0 + \sum_{i=1}^m x_i w_i \right)$$

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

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

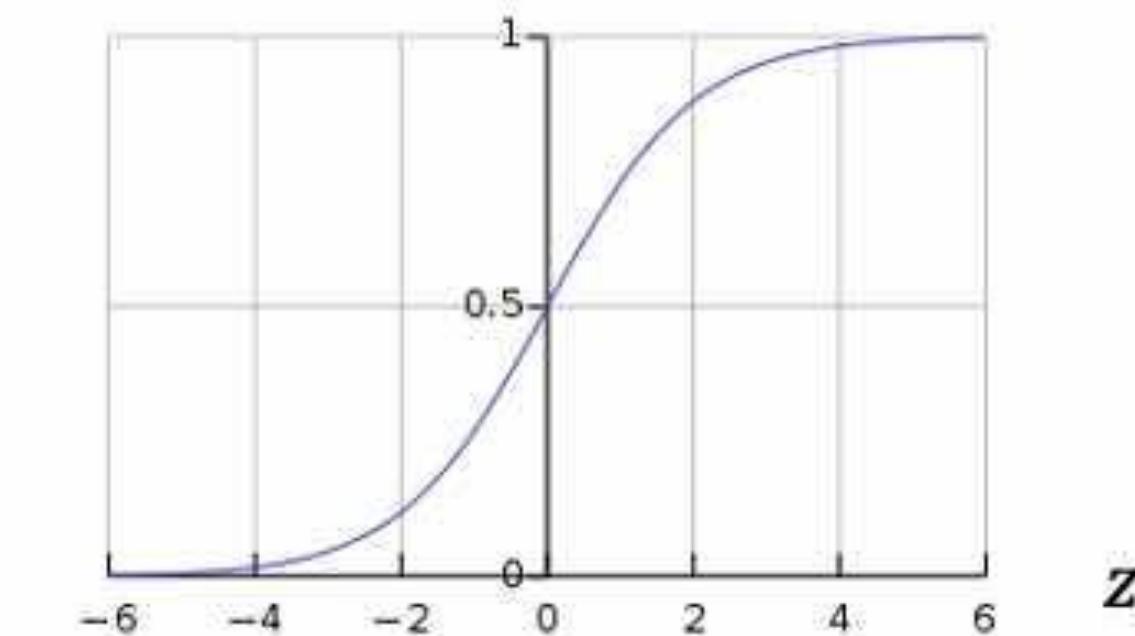


Activation Functions

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

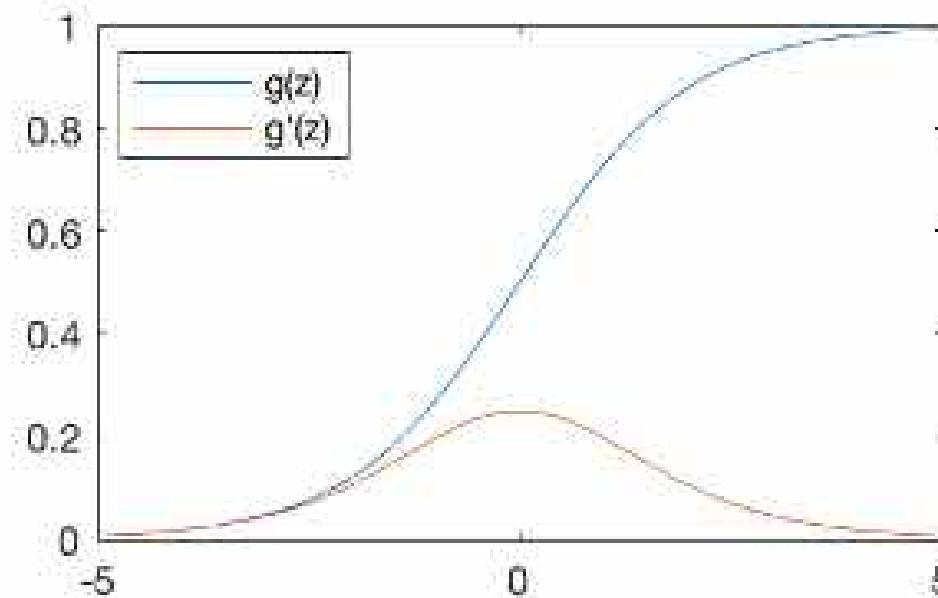
- Example: sigmoid function

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



Common Activation Functions

Sigmoid Function



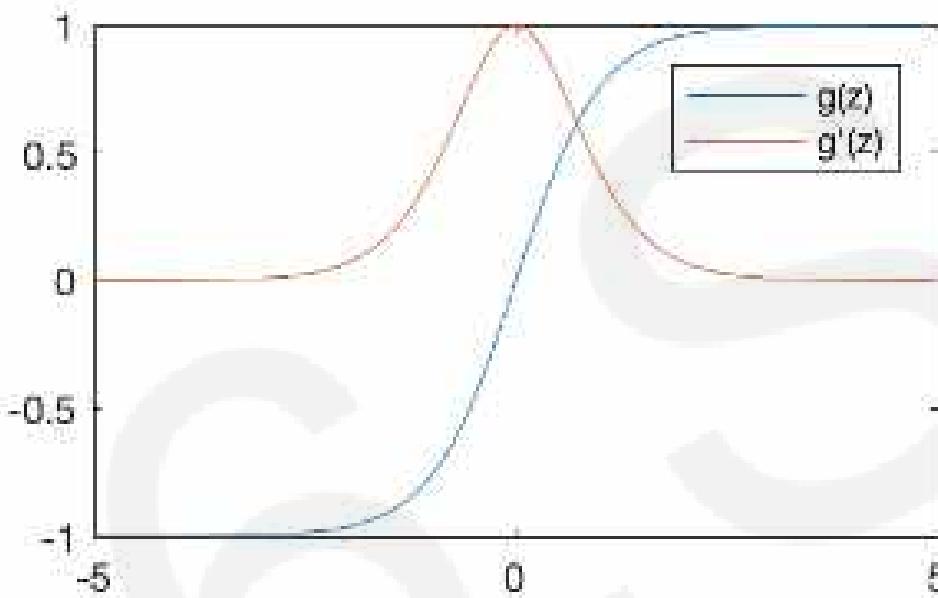
$$g(z) = \frac{1}{1 + e^{-z}}$$

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



`tf.math.sigmoid(z)`

Hyperbolic Tangent



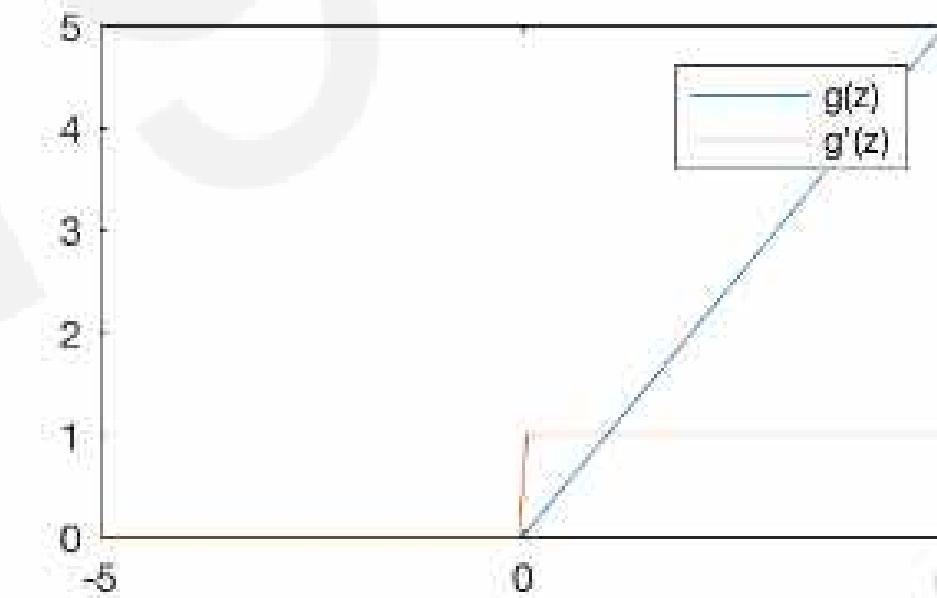
$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

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



`tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



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

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



`tf.nn.relu(z)`



TensorFlow code blocks

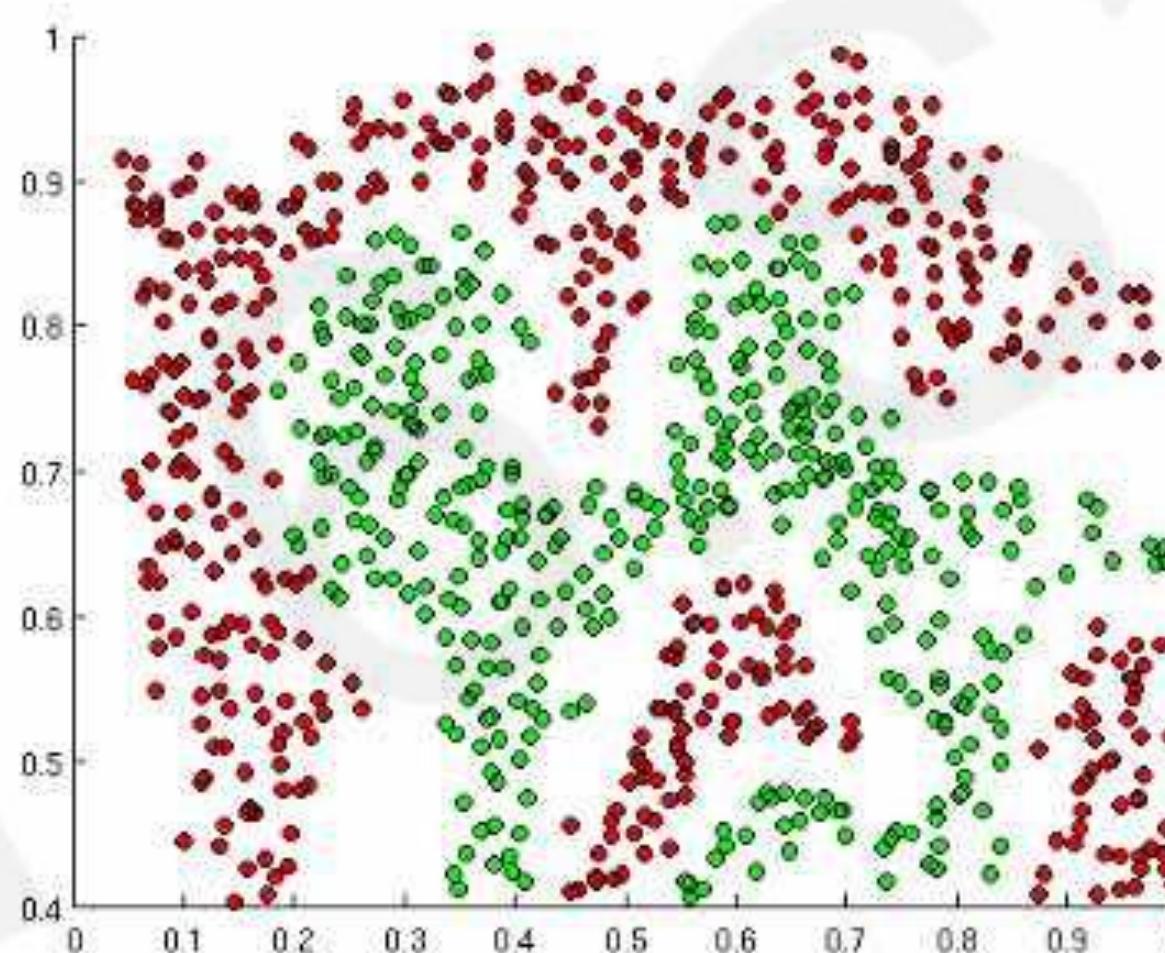


Massachusetts
Institute of
Technology

NOTE: All activation functions are non-linear

Importance of Activation Functions

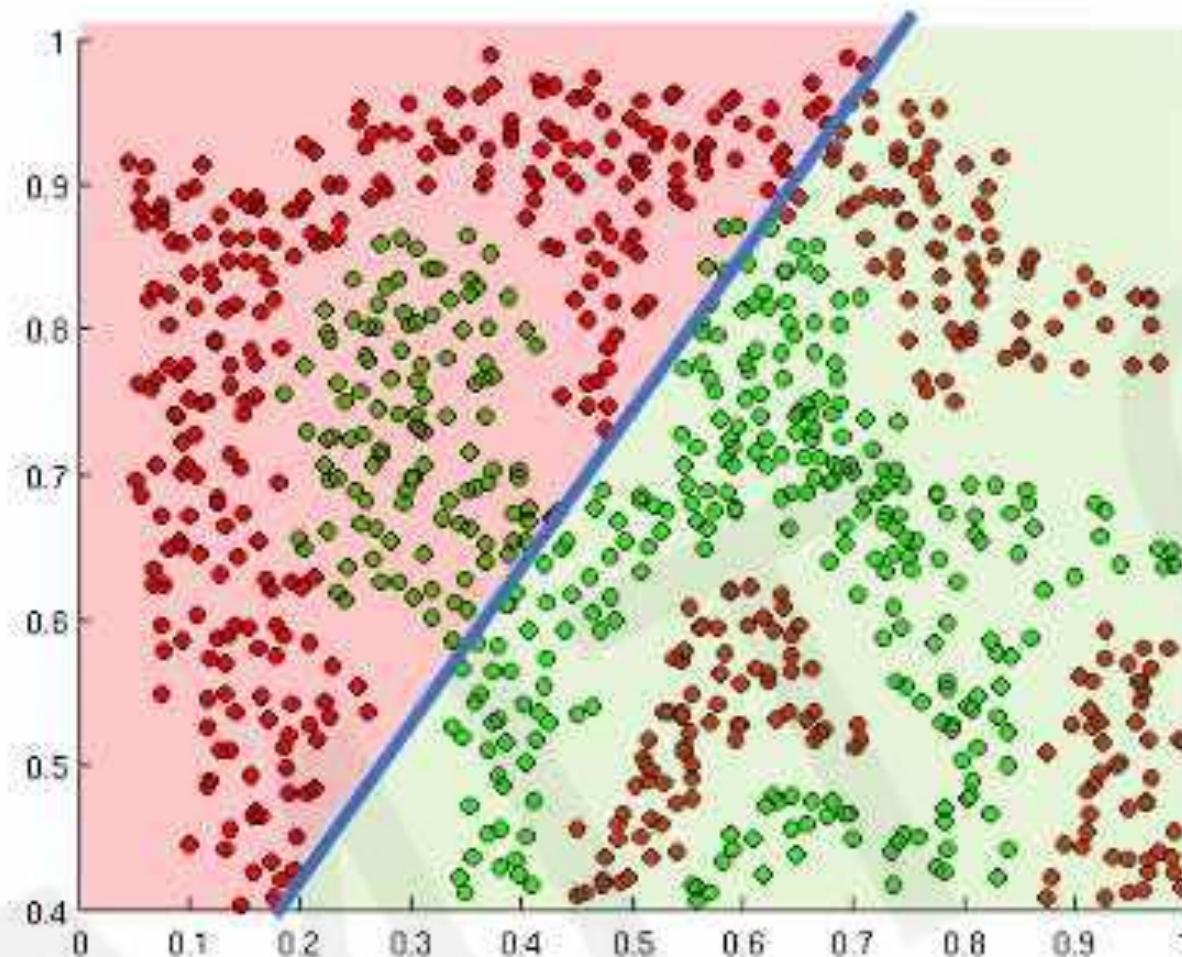
The purpose of activation functions is to **introduce non-linearities** into the network



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

Importance of Activation Functions

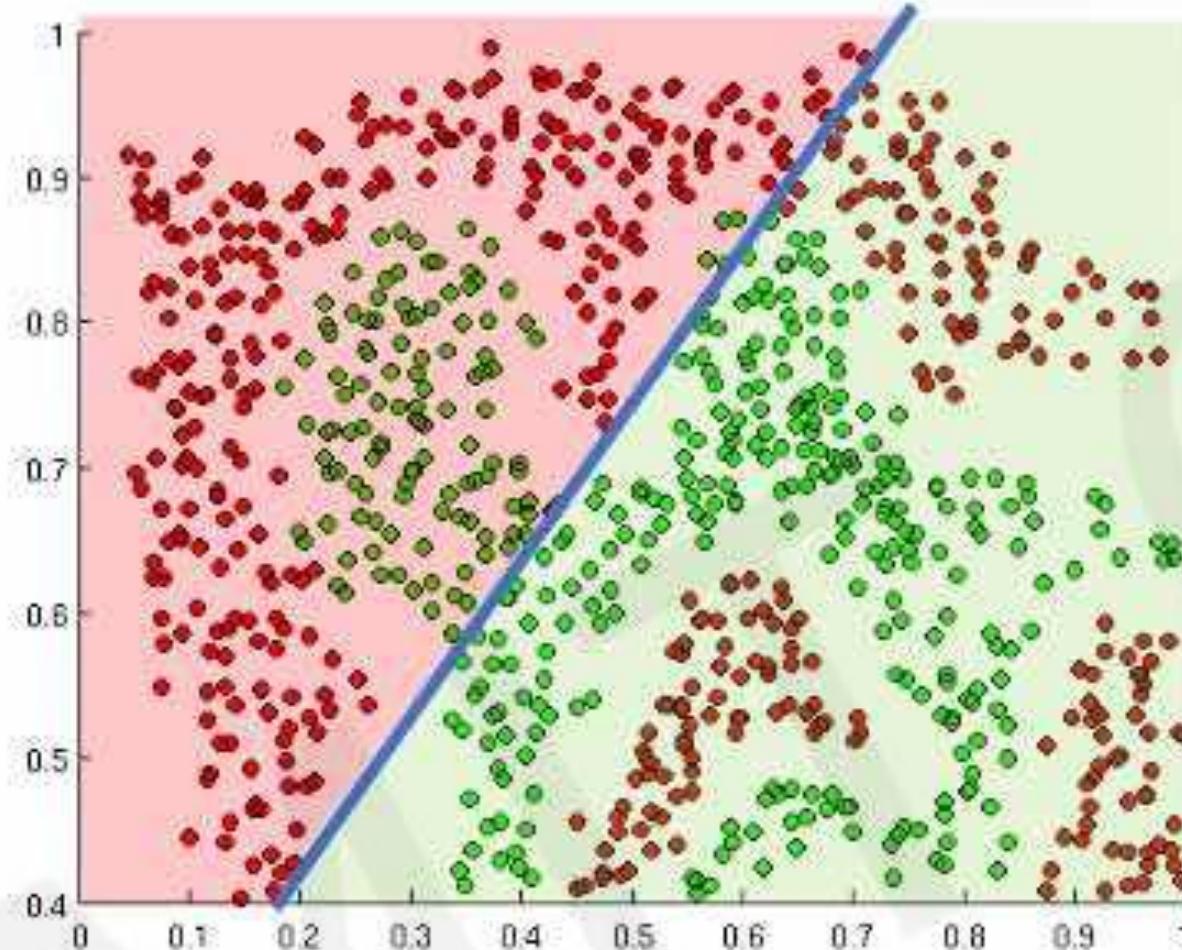
The purpose of activation functions is to **introduce non-linearities** into the network



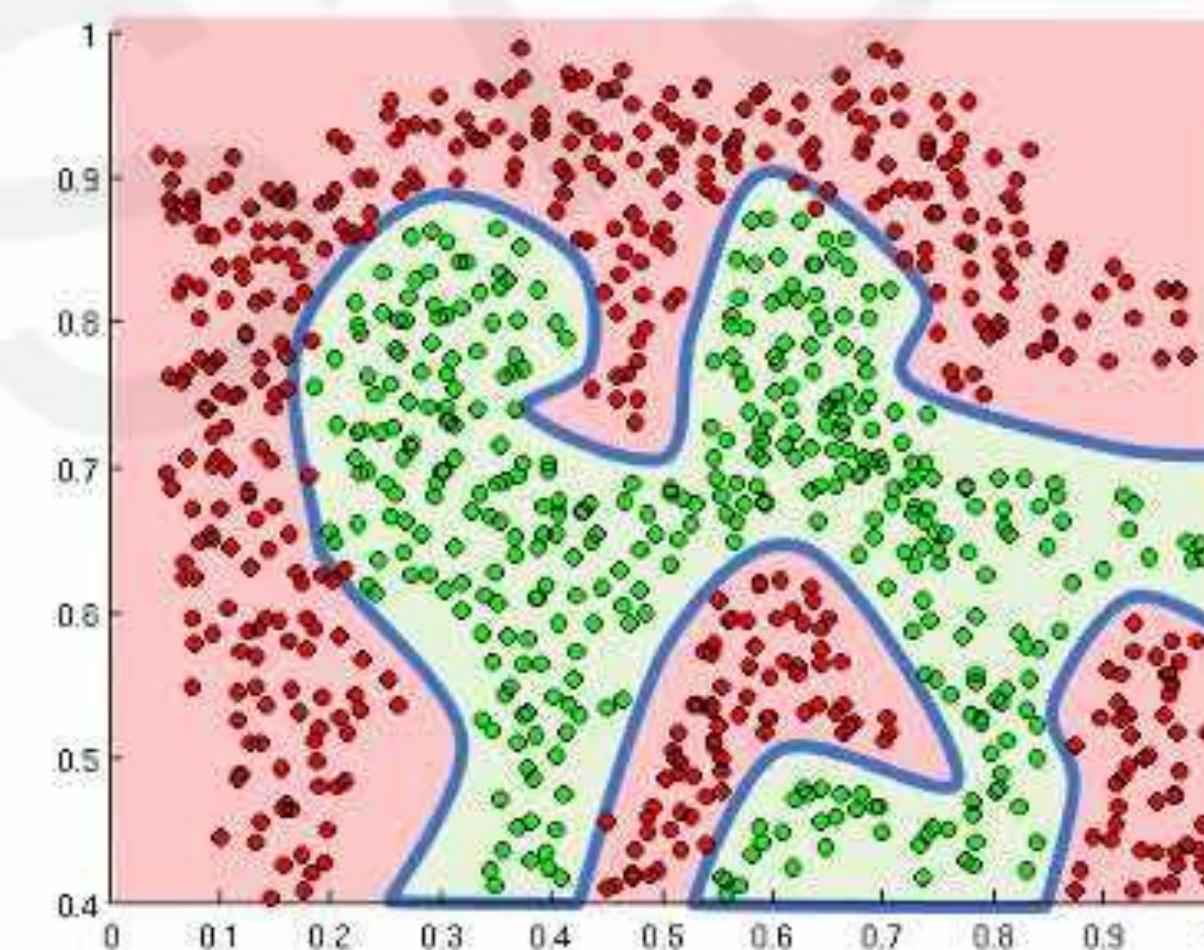
Linear activation functions produce linear decisions no matter the network size

Importance of Activation Functions

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

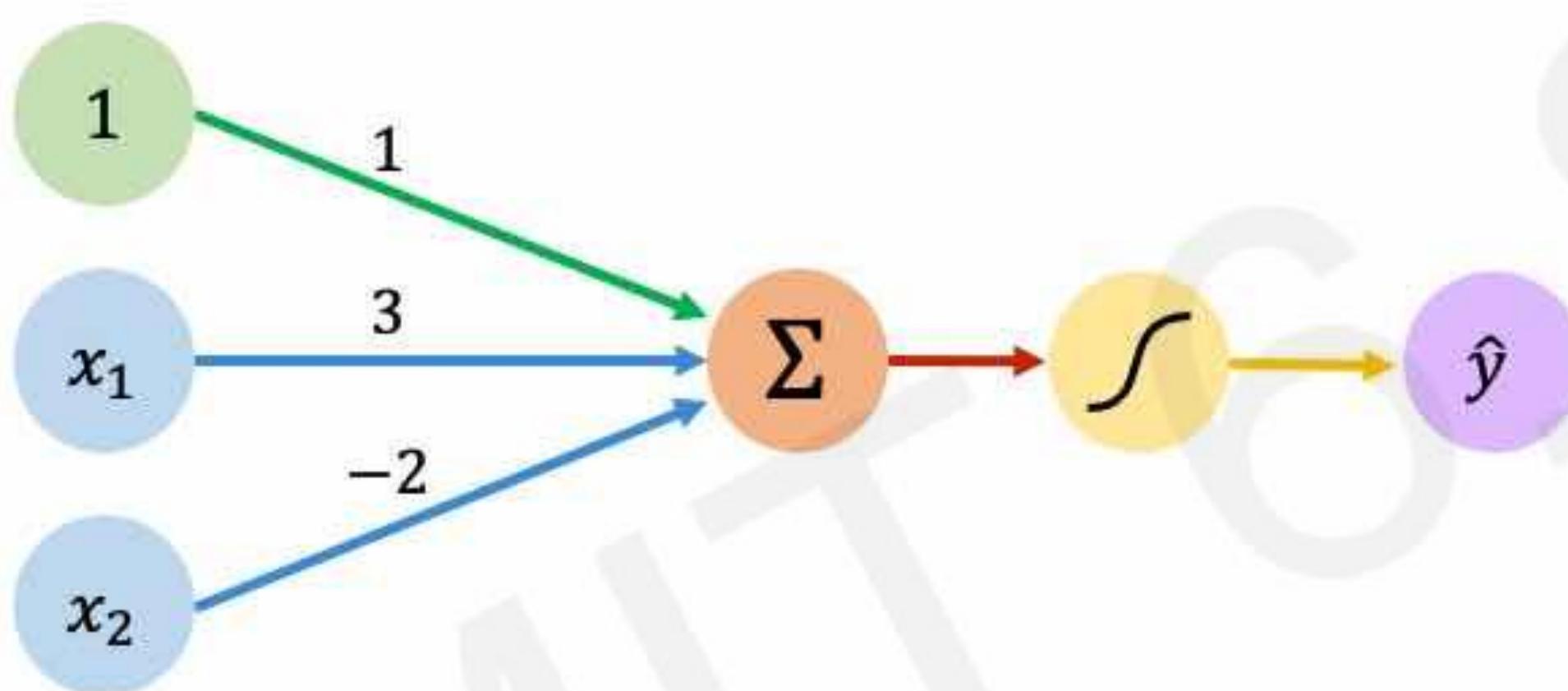


Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example

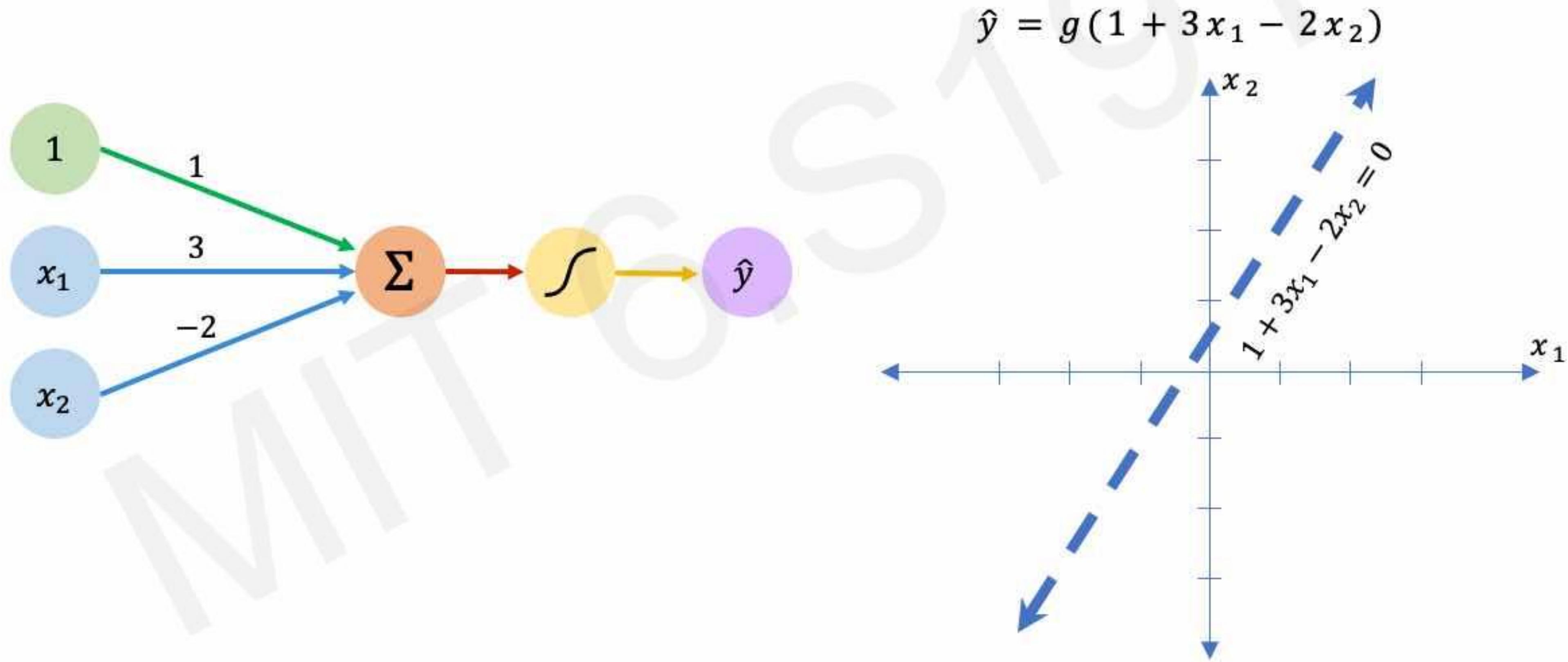


We have: $w_0 = 1$ and $\mathbf{w} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

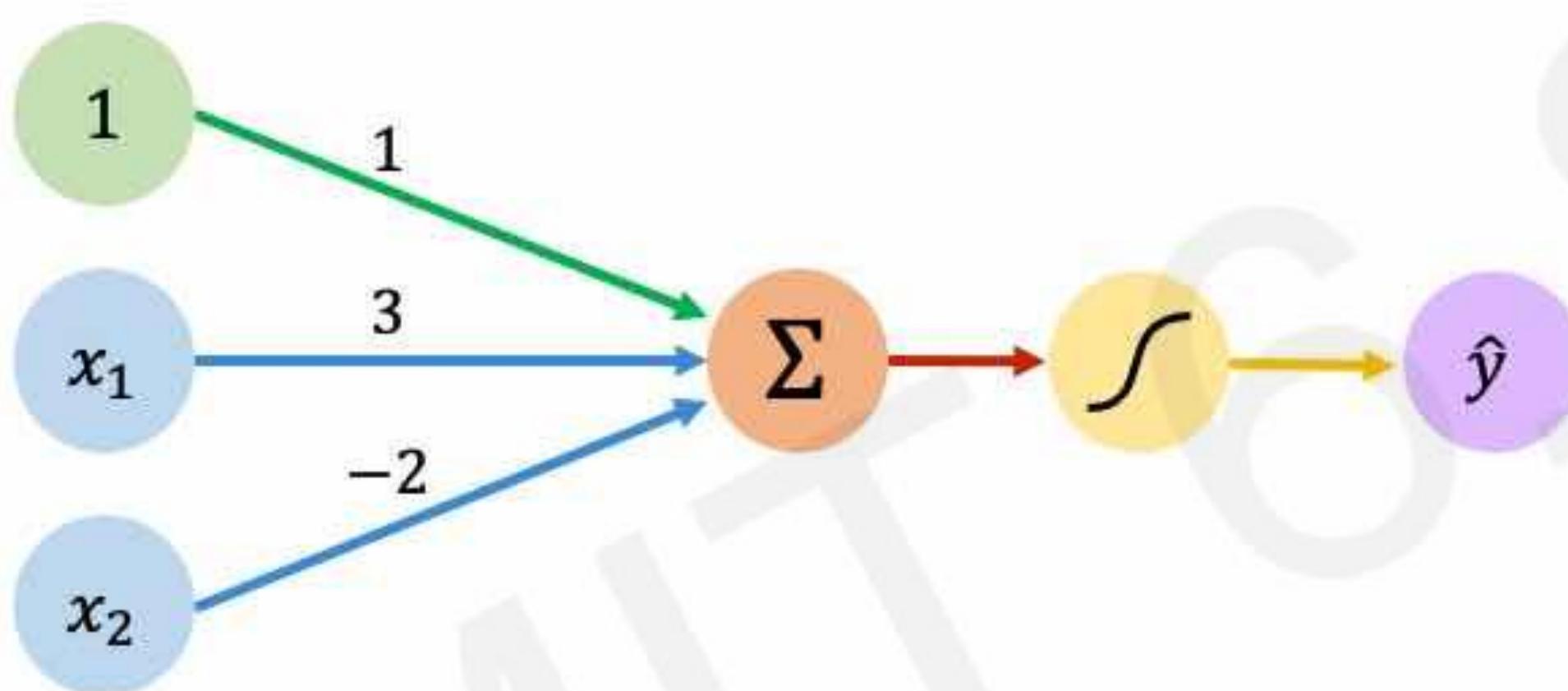
$$\begin{aligned}\hat{y} &= g(w_0 + \mathbf{X}^T \mathbf{w}) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g\left(1 + 3x_1 - 2x_2\right)\end{aligned}$$

This is just a line in 2D!

The Perceptron: Example



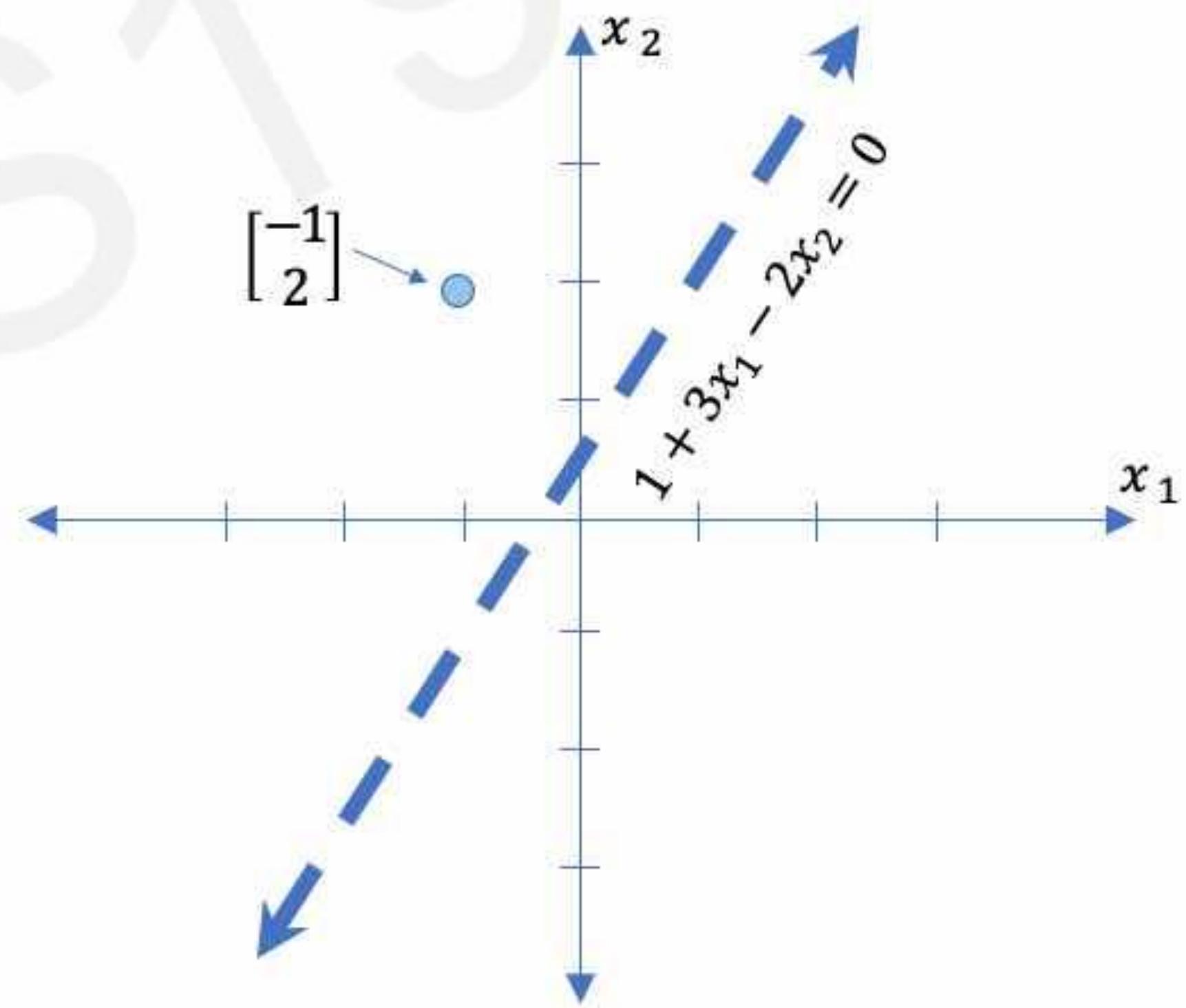
The Perceptron: Example



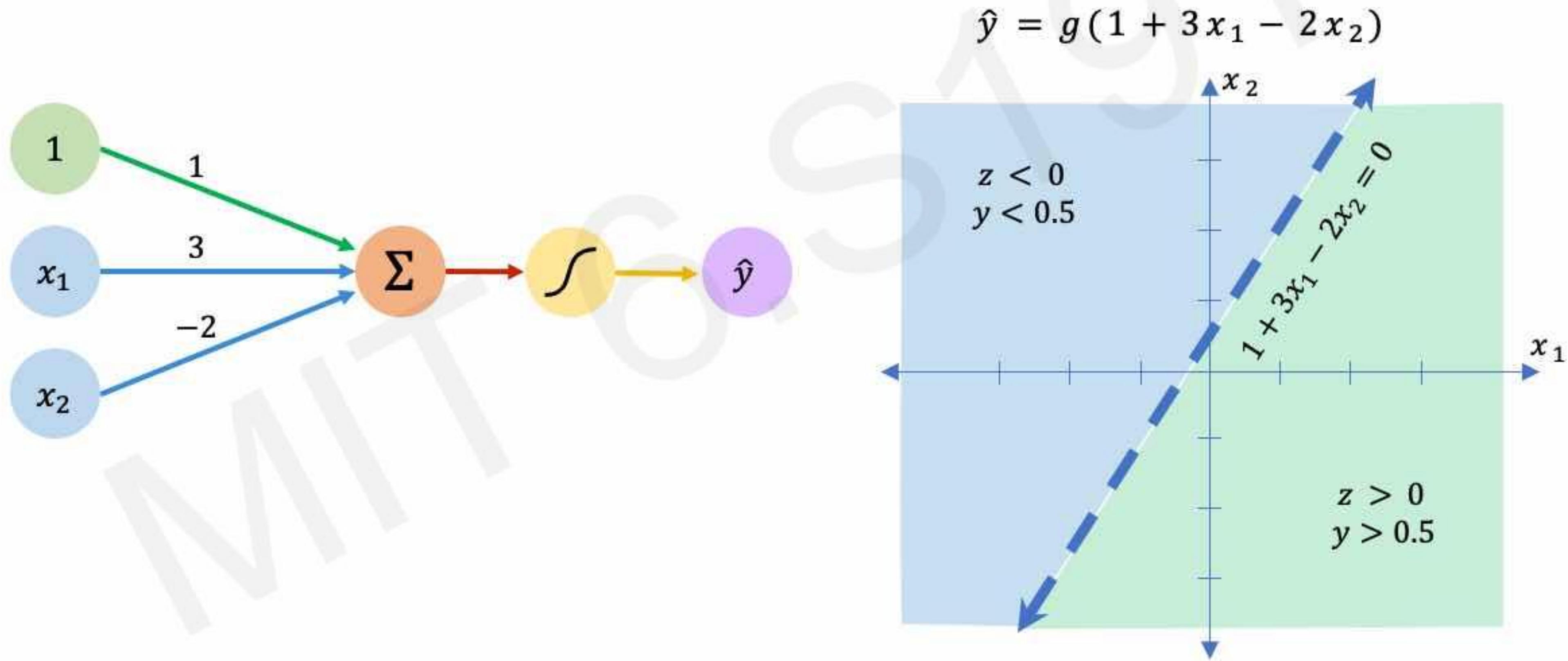
Assume we have input: $\mathbf{x} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

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

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



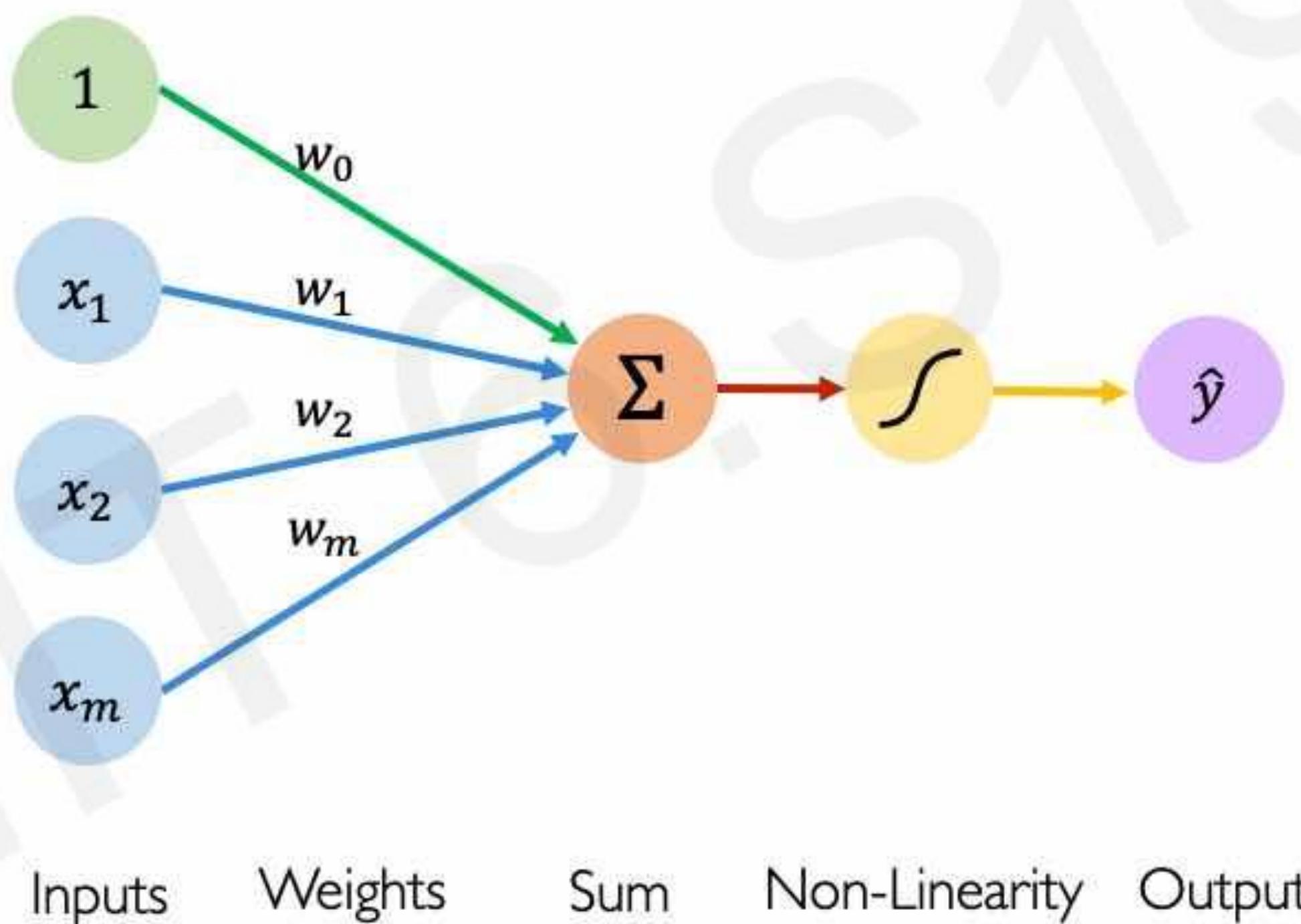
The Perceptron: Example



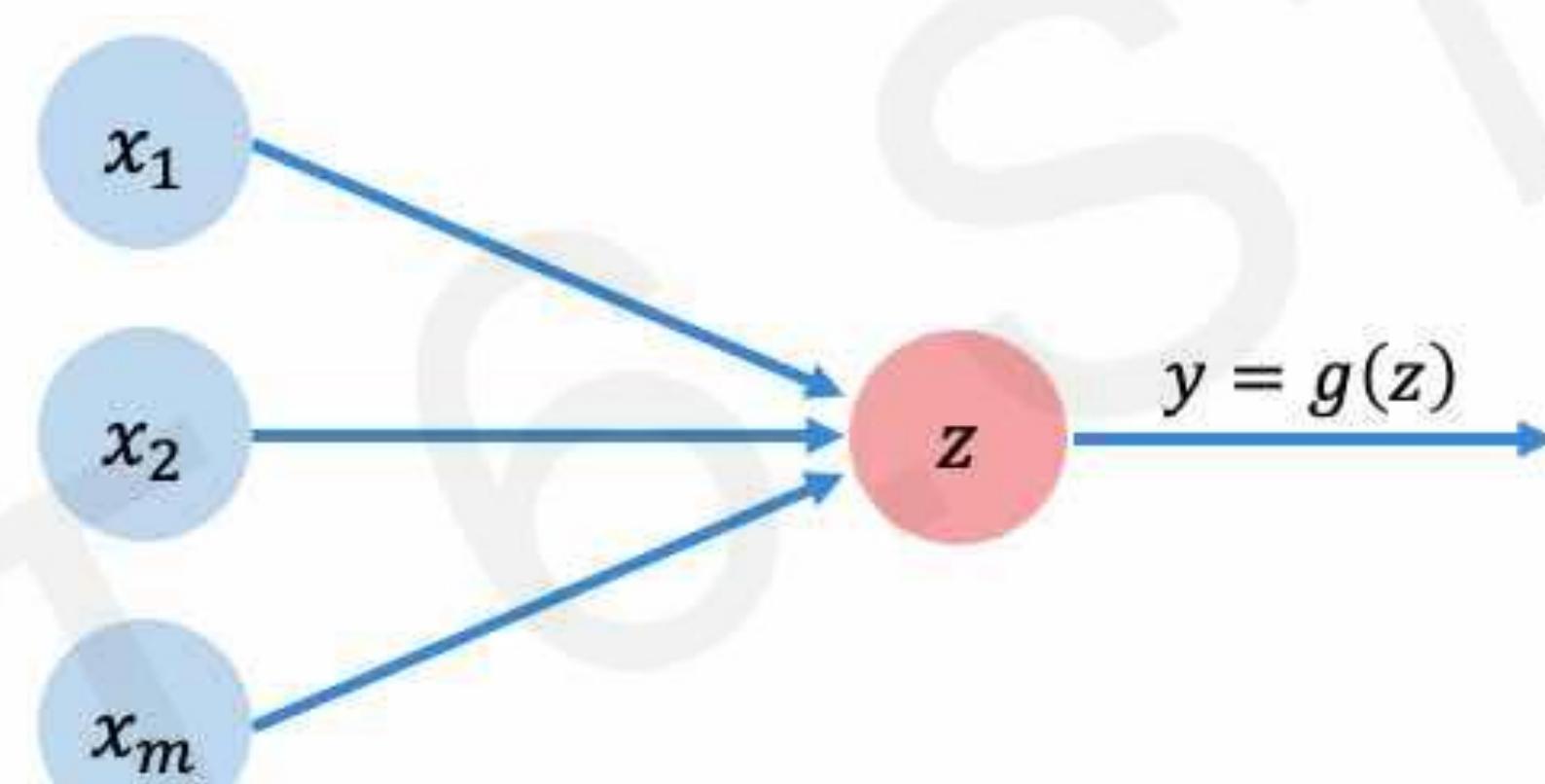
Building Neural Networks with Perceptrons

The Perceptron: Simplified

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



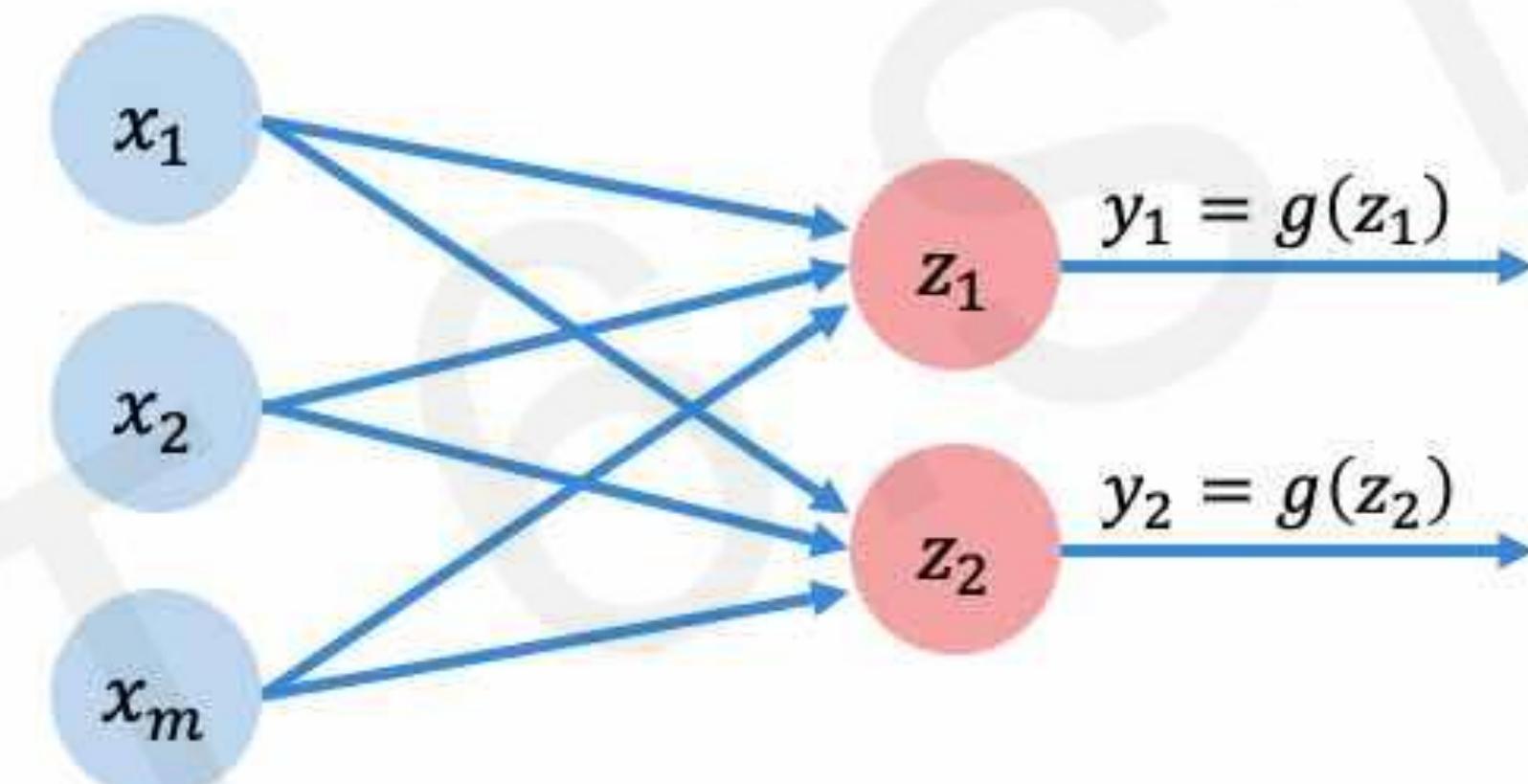
The Perceptron: Simplified



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

Multi Output Perceptron

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



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



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])

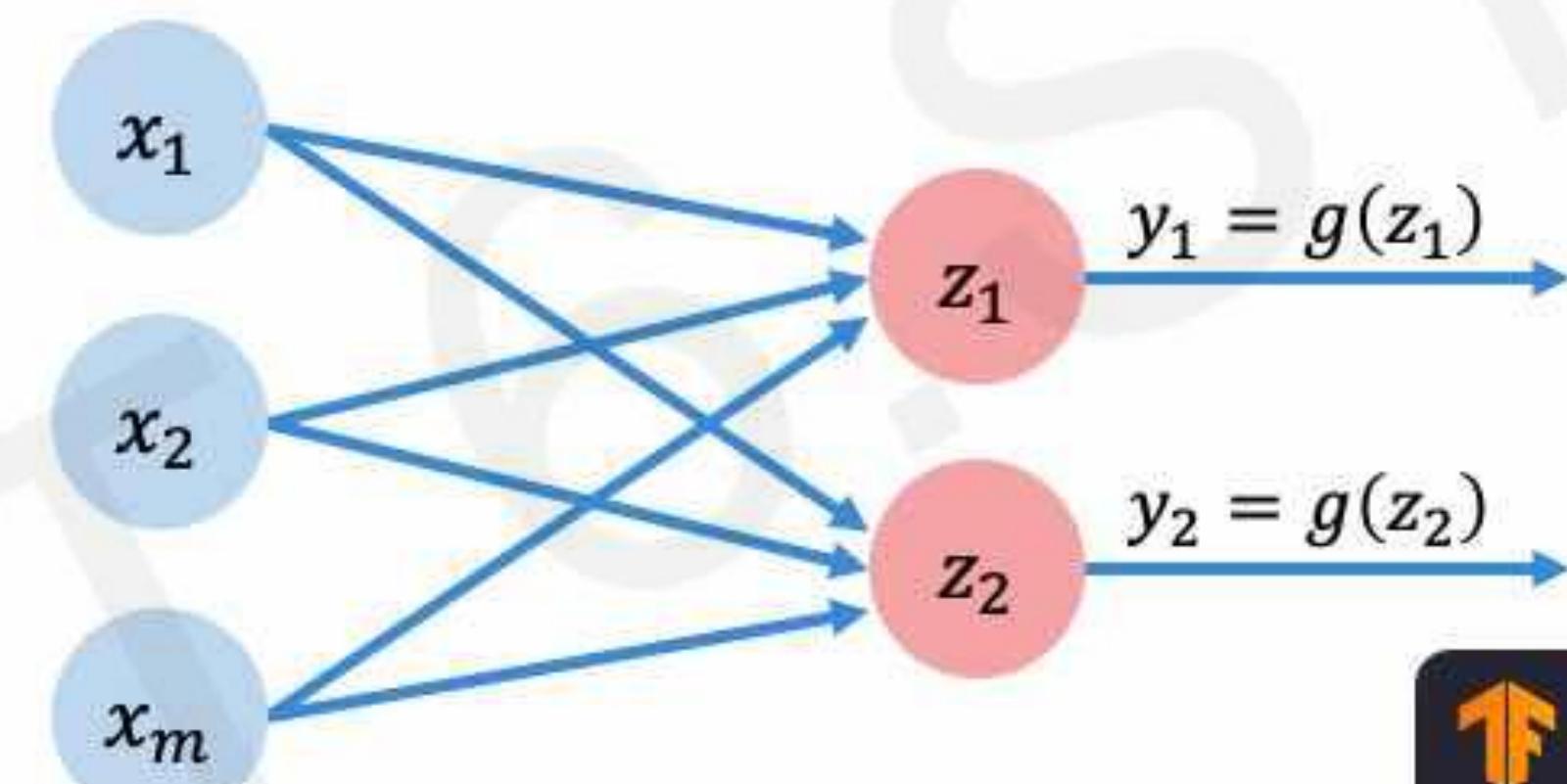
    def call(self, inputs):
        # Forward propagate the inputs
        z = tf.matmul(inputs, self.W) + self.b

        # Feed through a non-linear activation
        output = tf.math.sigmoid(z)

    return output
```

Multi Output Perceptron

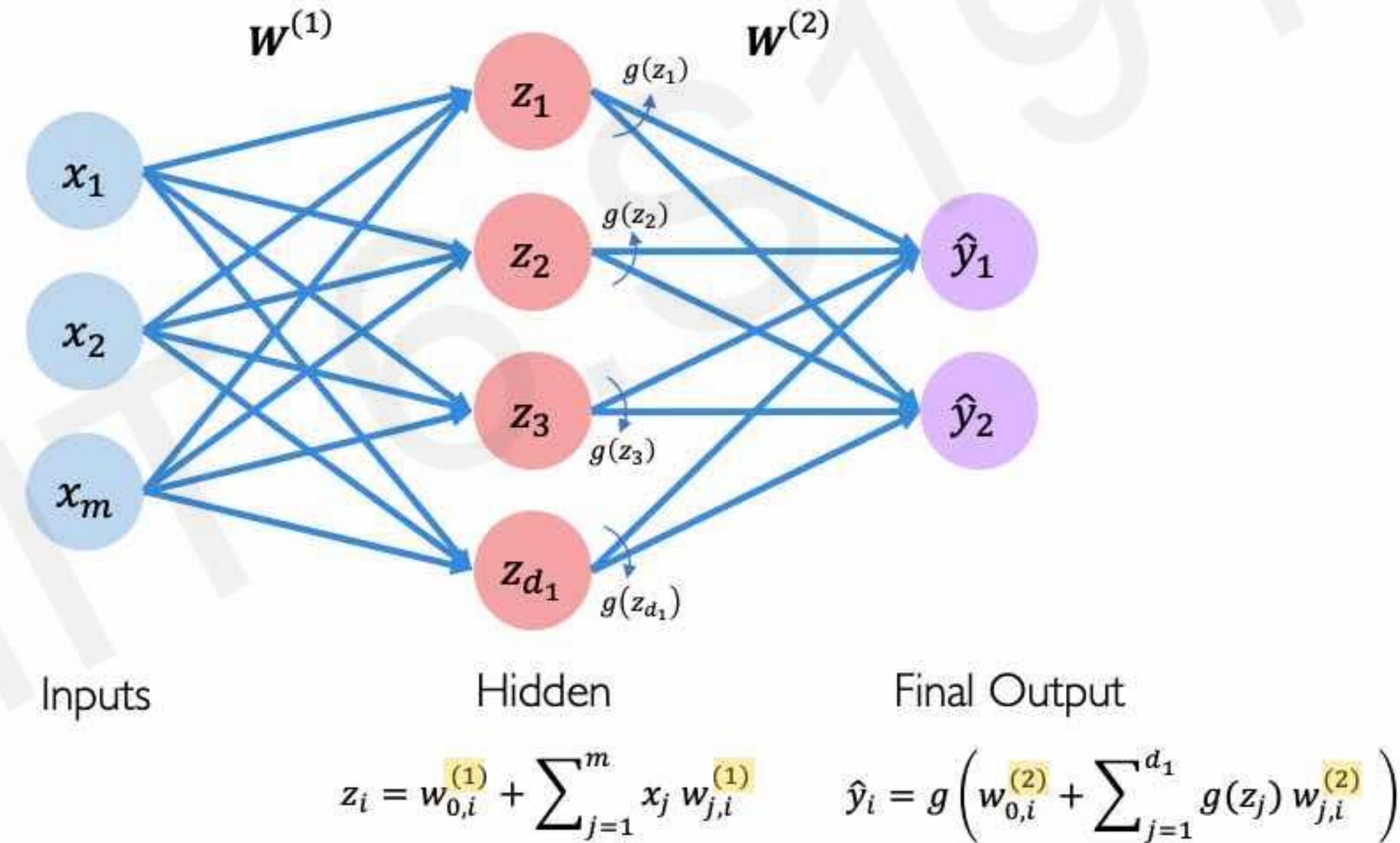
Because all inputs are densely connected to all outputs, these layers are called **Dense** layers



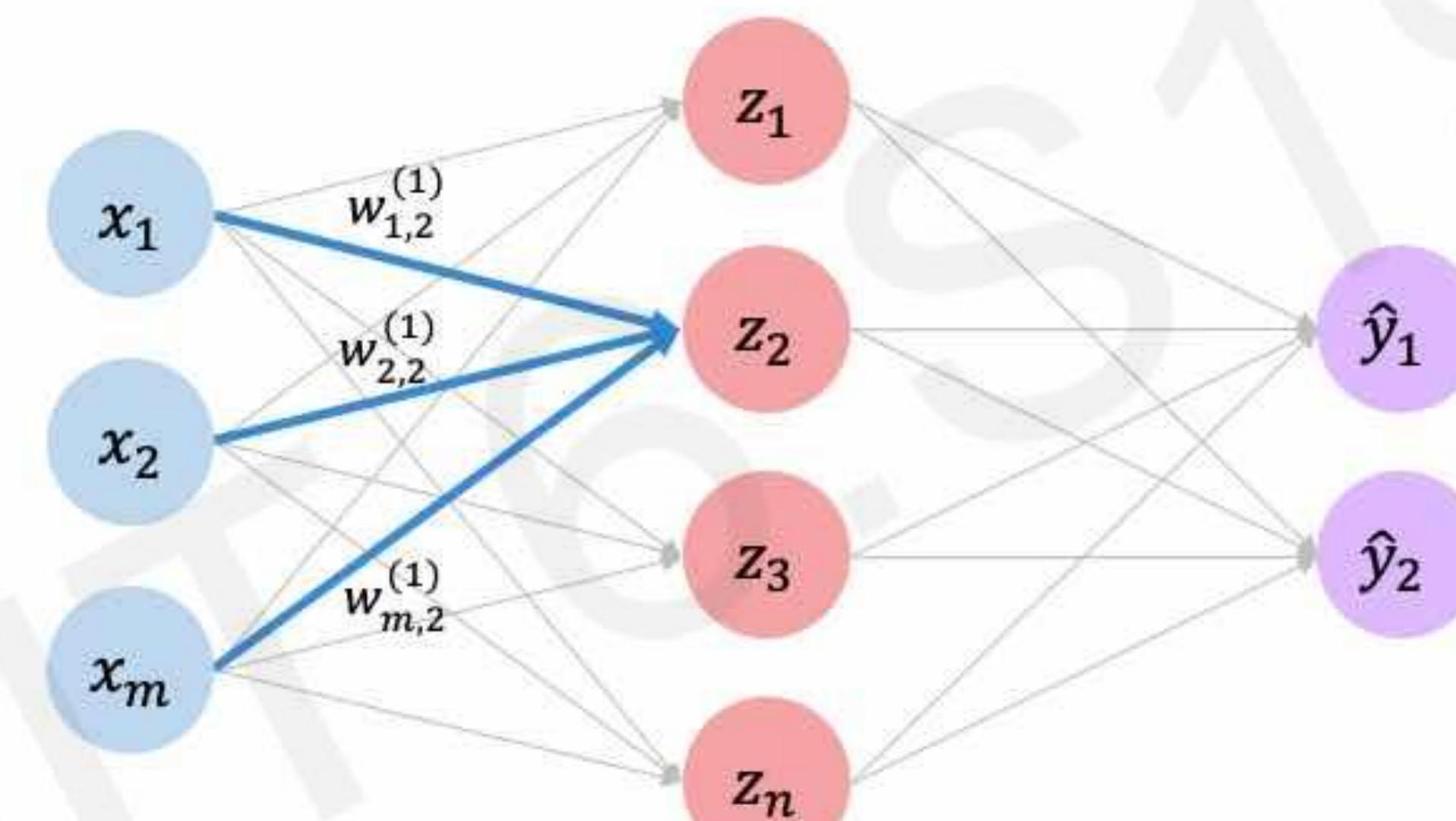
```
import tensorflow as tf  
layer = tf.keras.layers.Dense(  
    units=2)
```

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

Single Layer Neural Network

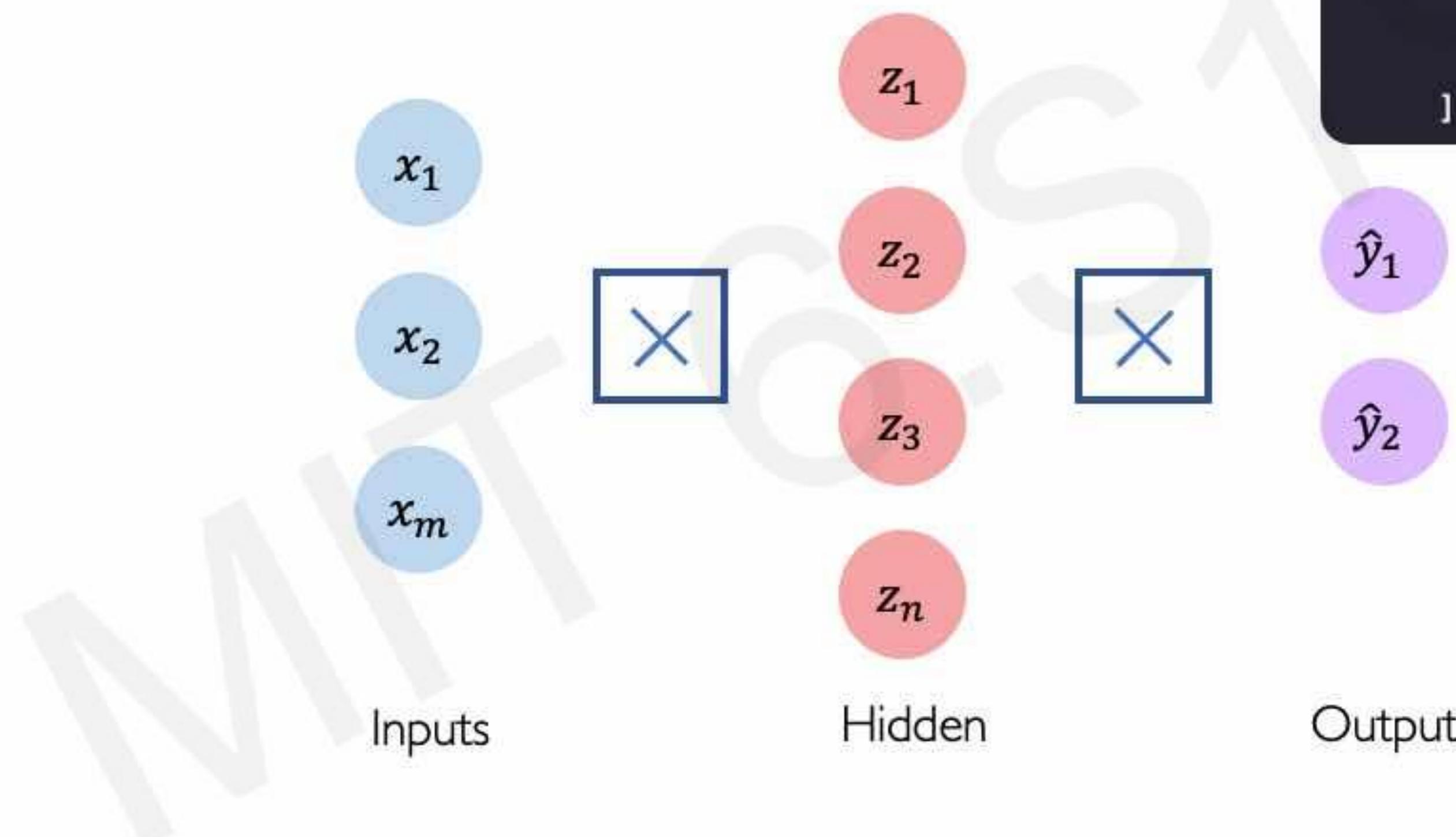


Single Layer Neural Network



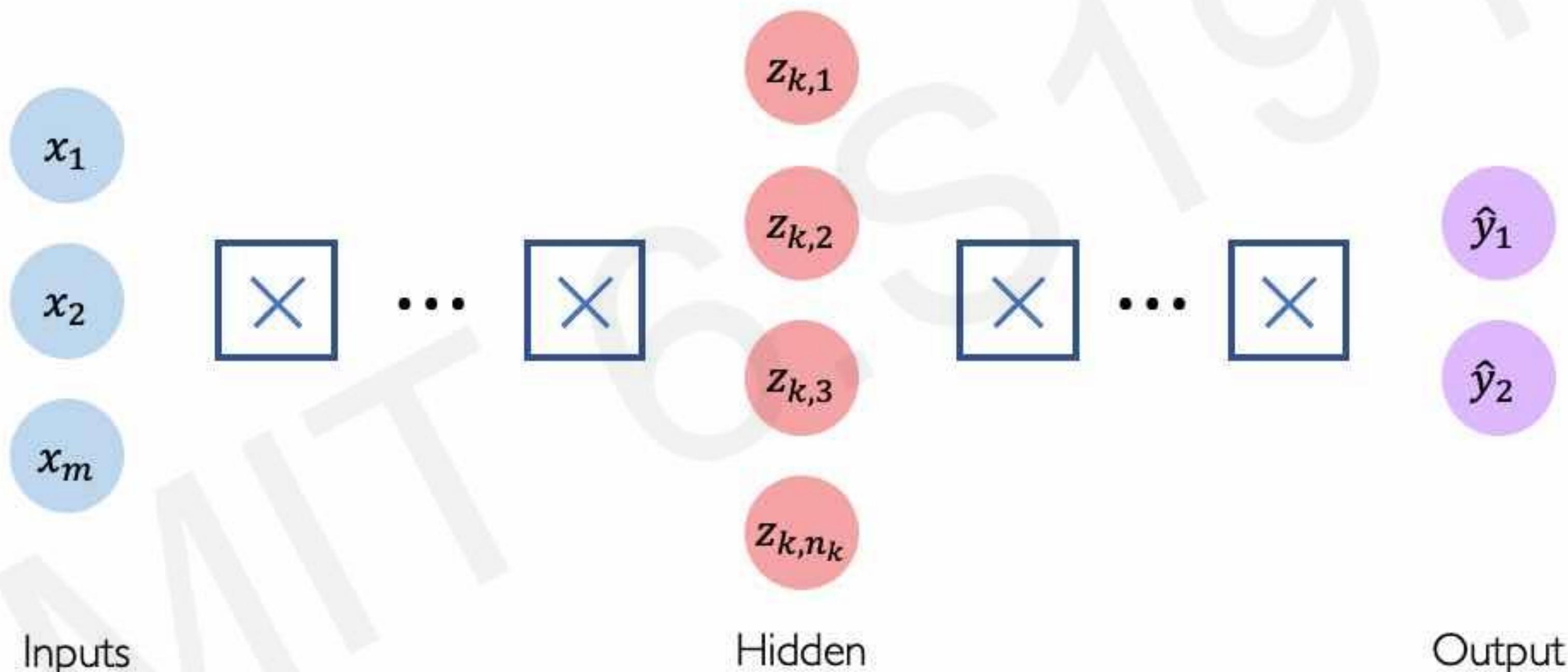
$$\begin{aligned} z_2 &= w_{0,2}^{(1)} + \sum_{j=1}^m x_j w_{j,2}^{(1)} \\ &= w_{0,2}^{(1)} + x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + \dots + x_m w_{m,2}^{(1)} \end{aligned}$$

Multi Output Perceptron



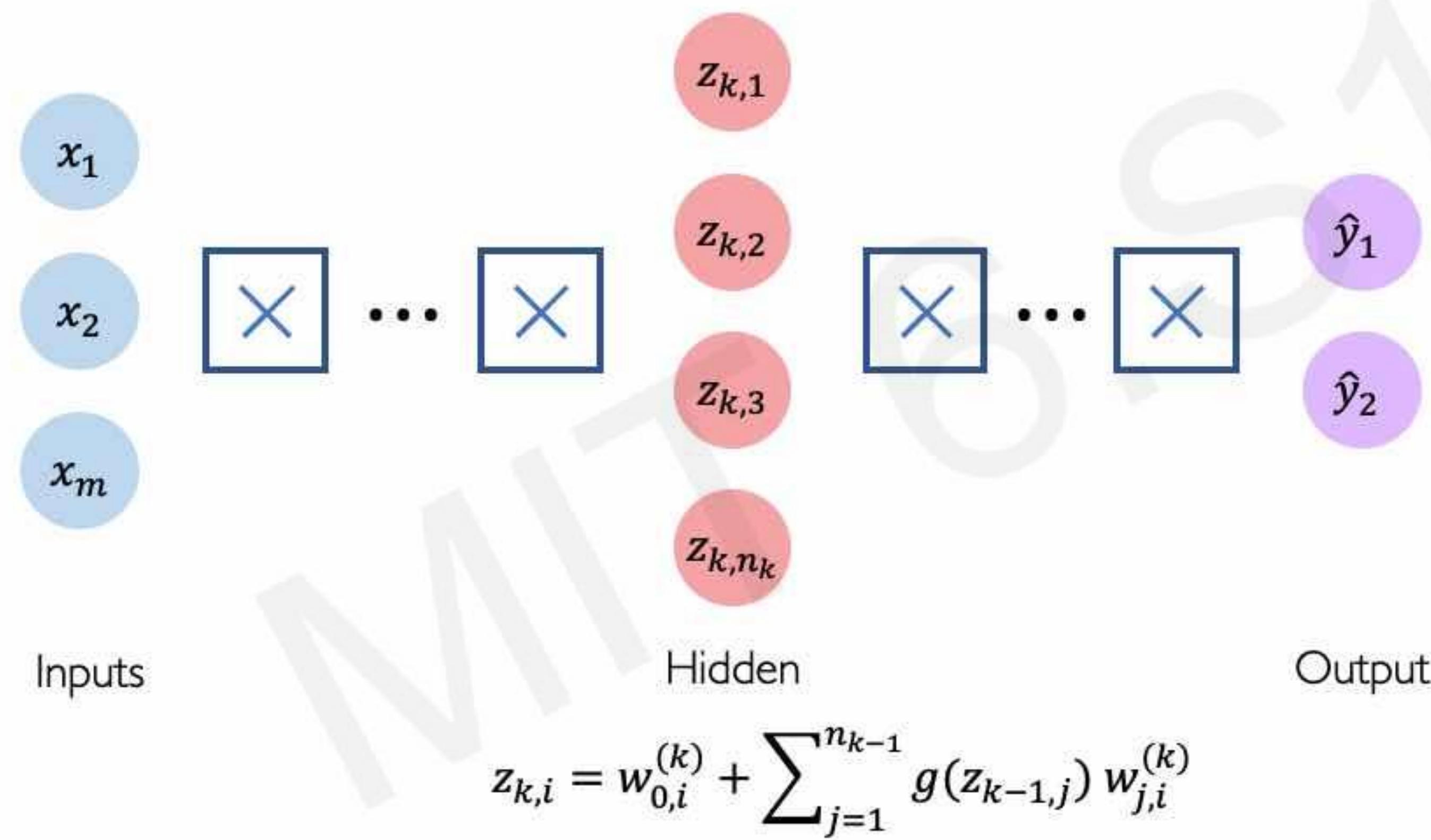
```
import tensorflow as tf  
  
model = tf.keras.Sequential([  
    tf.keras.layers.Dense(n),  
    tf.keras.layers.Dense(2)  
])
```

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



```
TensorFlow logo  
import tensorflow as tf  
  
model = tf.keras.Sequential([  
    tf.keras.layers.Dense(n1),  
    tf.keras.layers.Dense(n2),  
    ...  
    tf.keras.layers.Dense(2)  
])
```

Applying Neural Networks

Example Problem

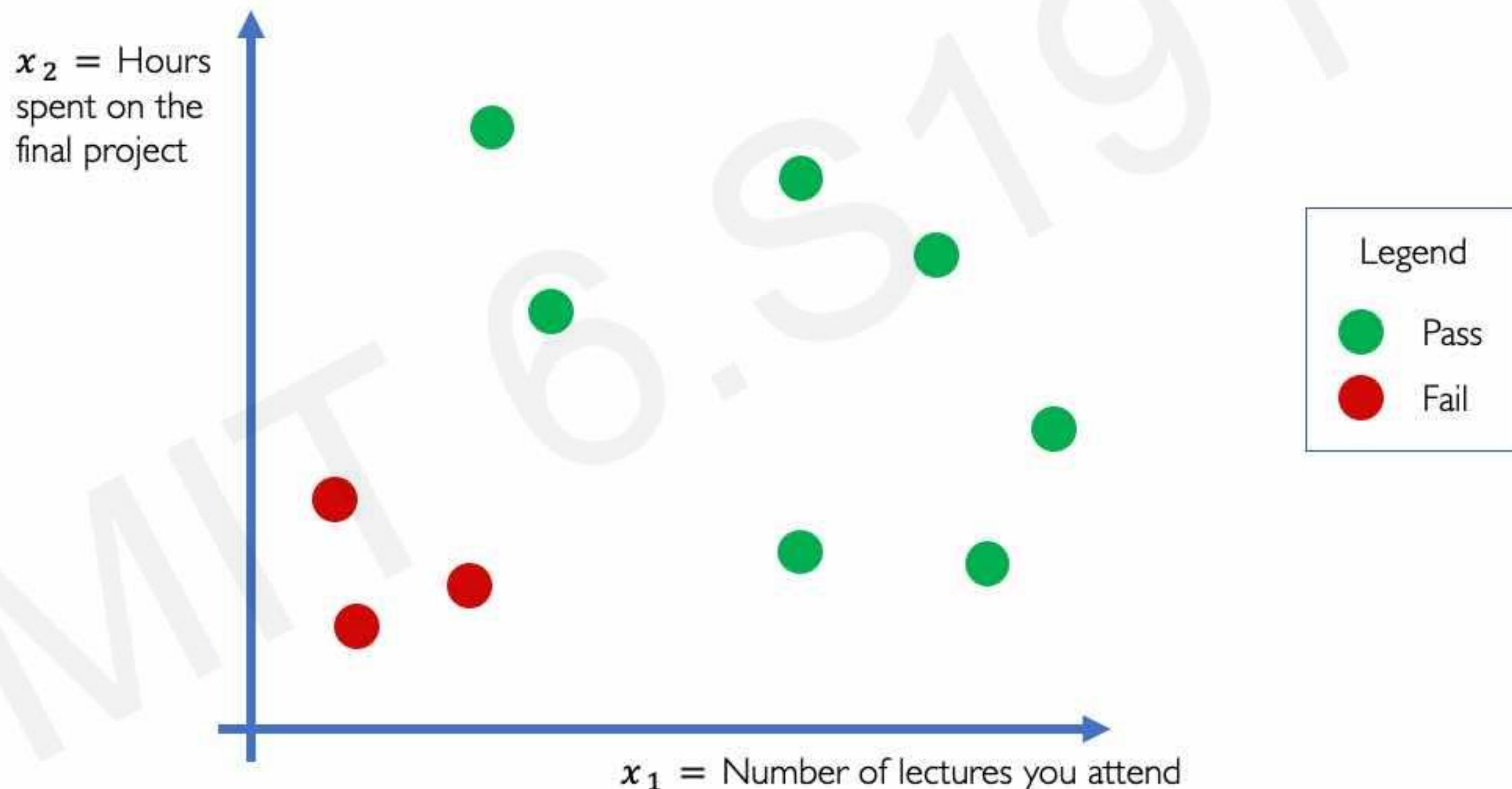
Will I pass this class?

Let's start with a simple two feature model

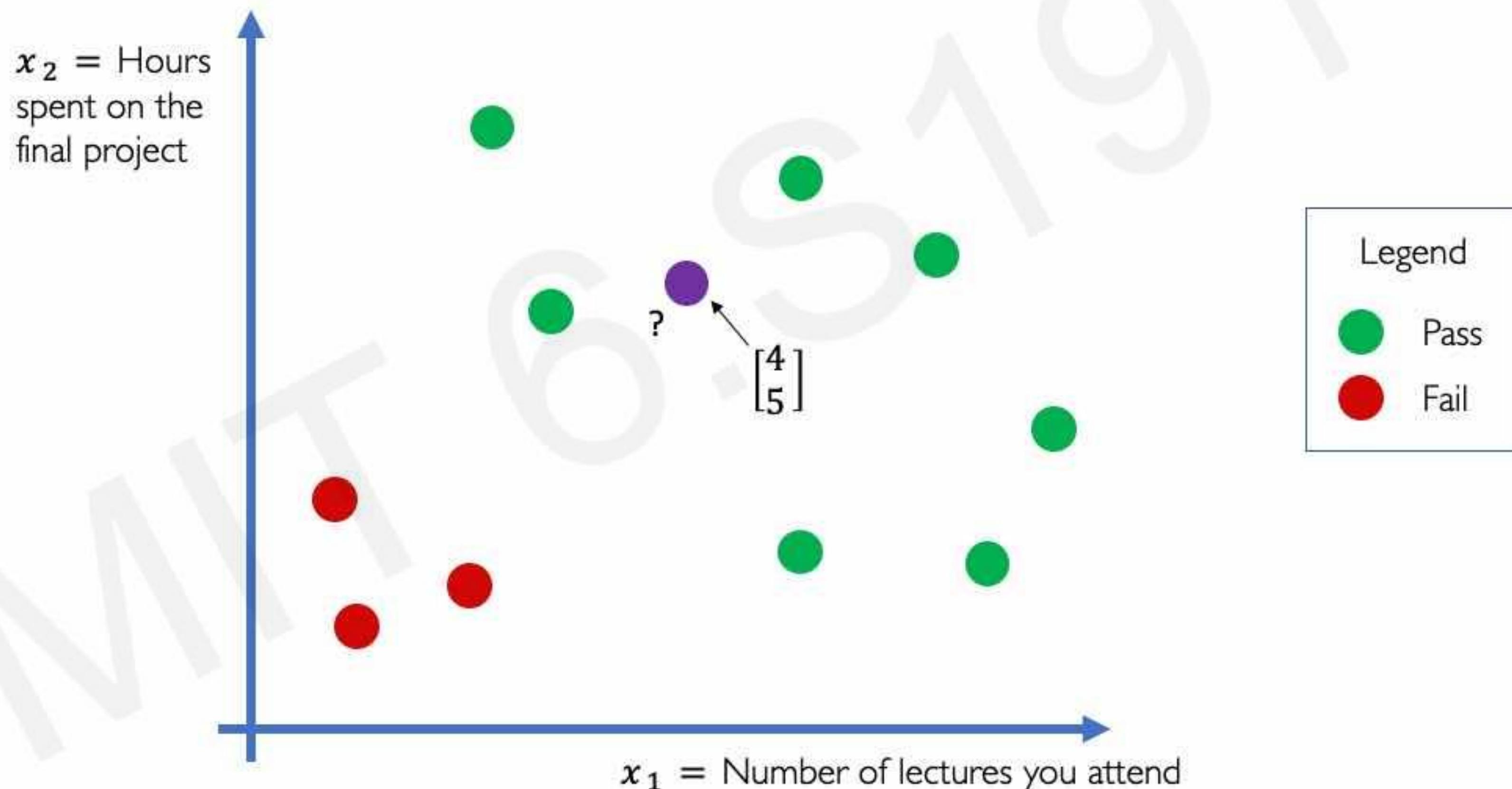
x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

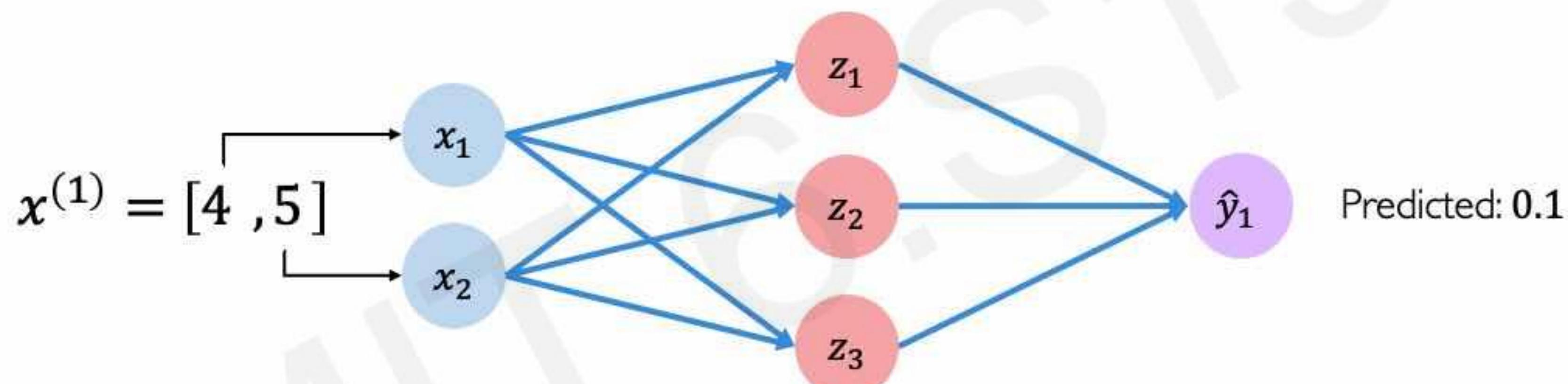
Example Problem: Will I pass this class?



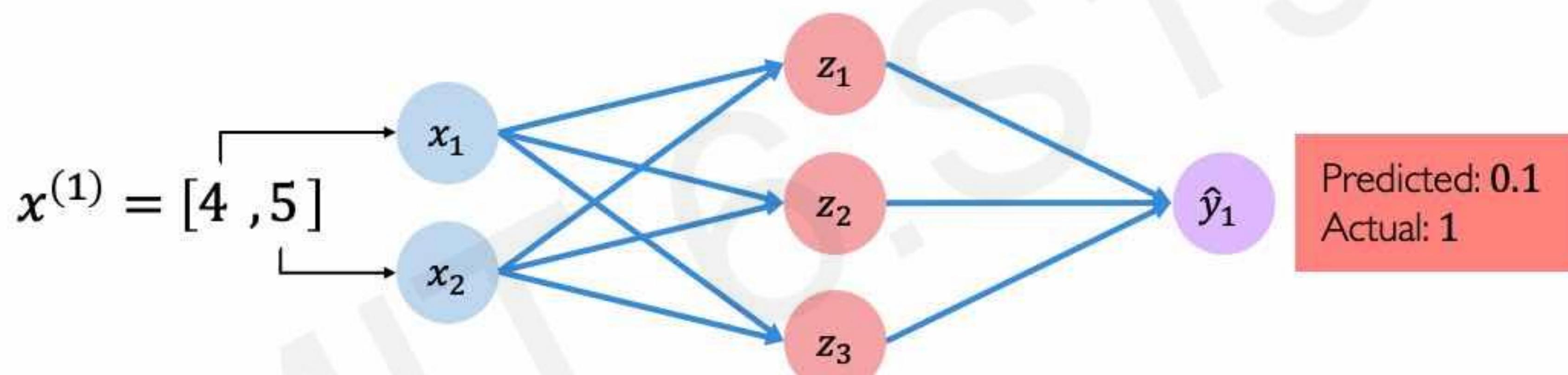
Example Problem: Will I pass this class?



Example Problem: Will I pass this class?

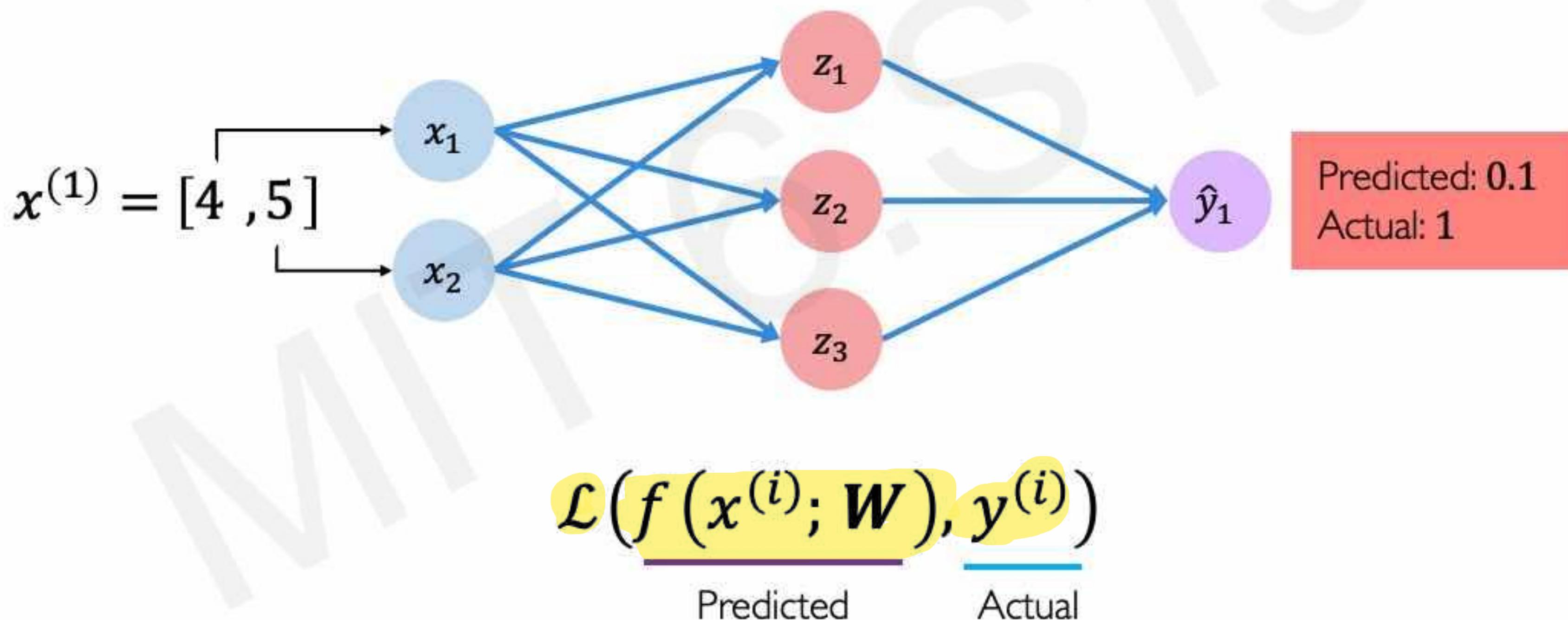


Example Problem: Will I pass this class?



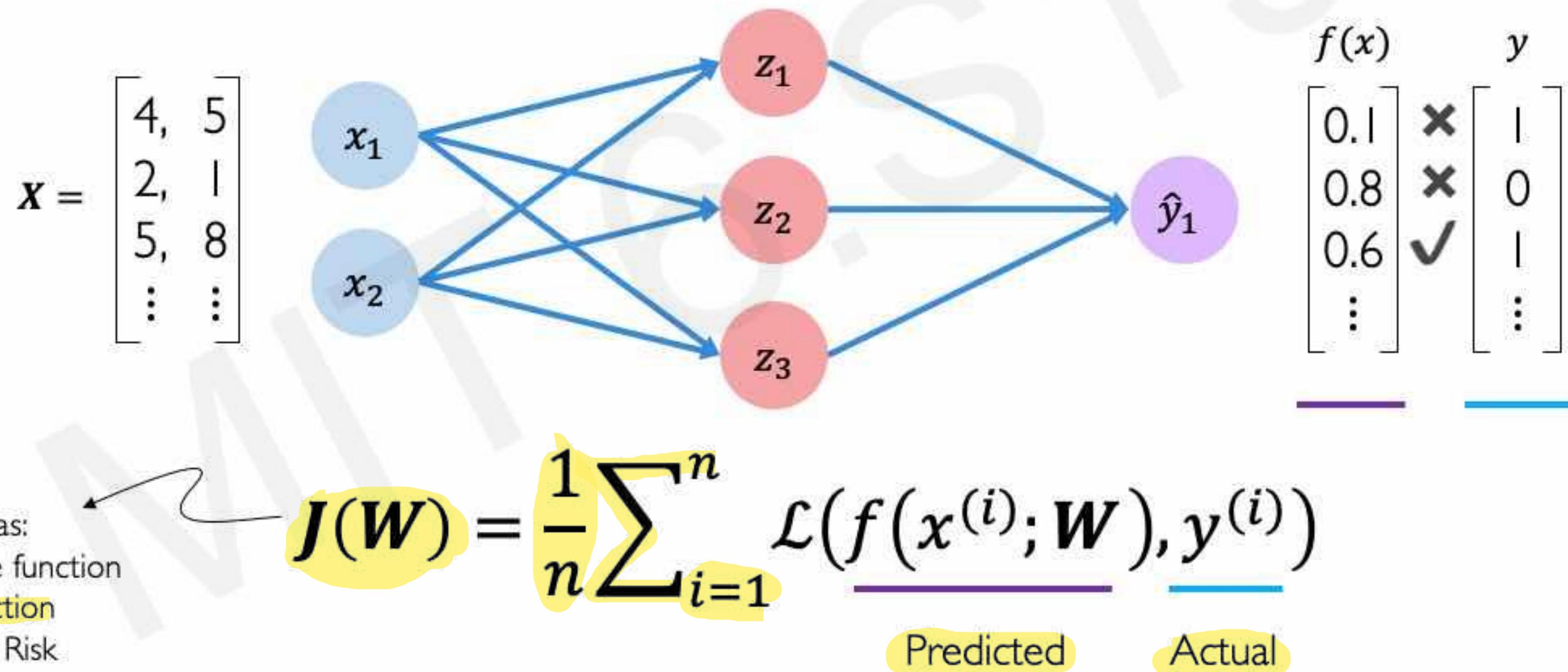
Quantifying Loss

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



Empirical Loss

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

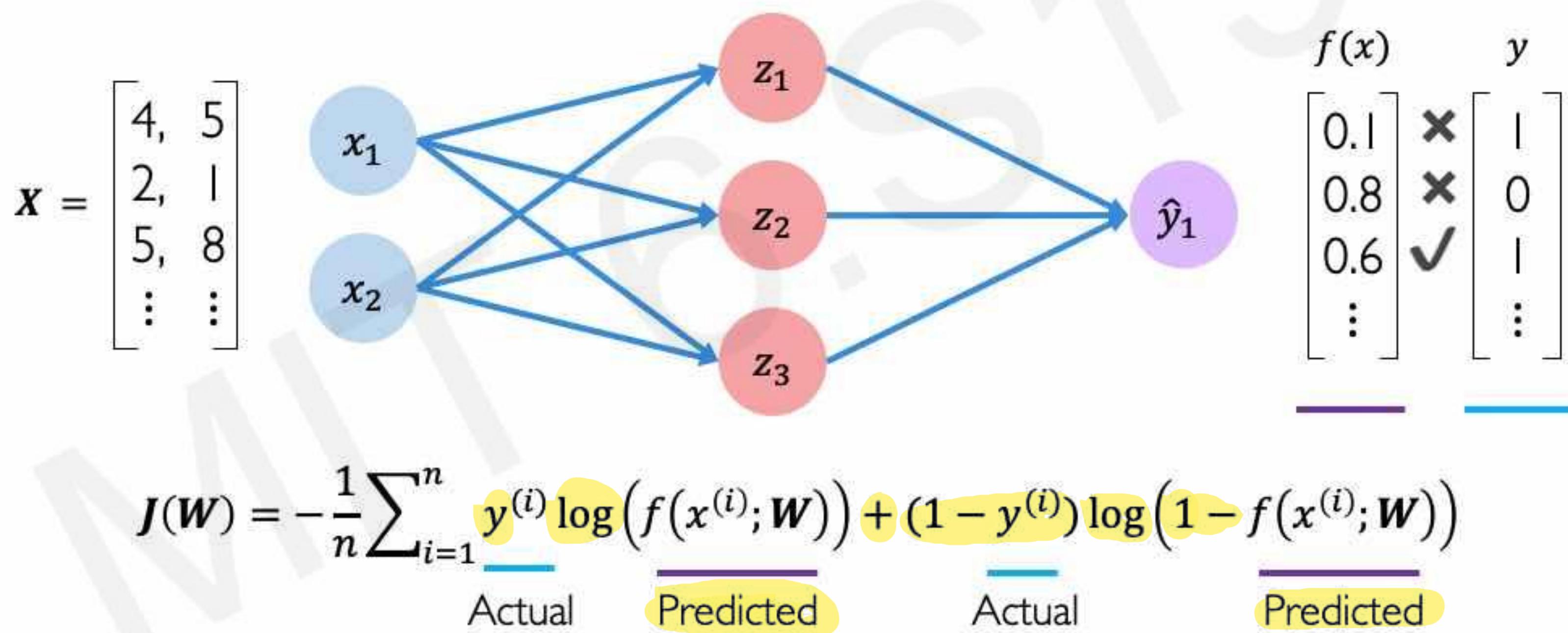


Also known as:

- Objective function
- Cost function
- Empirical Risk

Binary Cross Entropy Loss

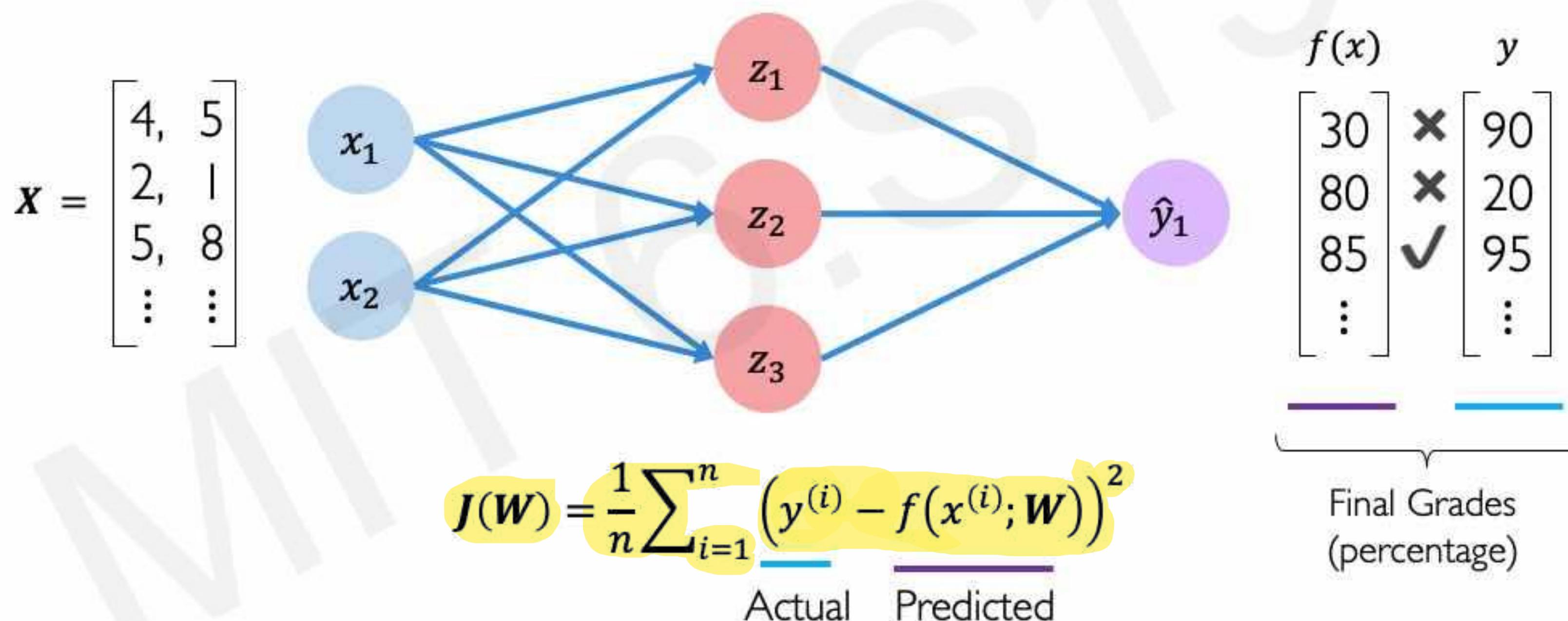
Cross entropy loss can be used with models that output a probability between 0 and 1



```
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**

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

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

Loss Optimization

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

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

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

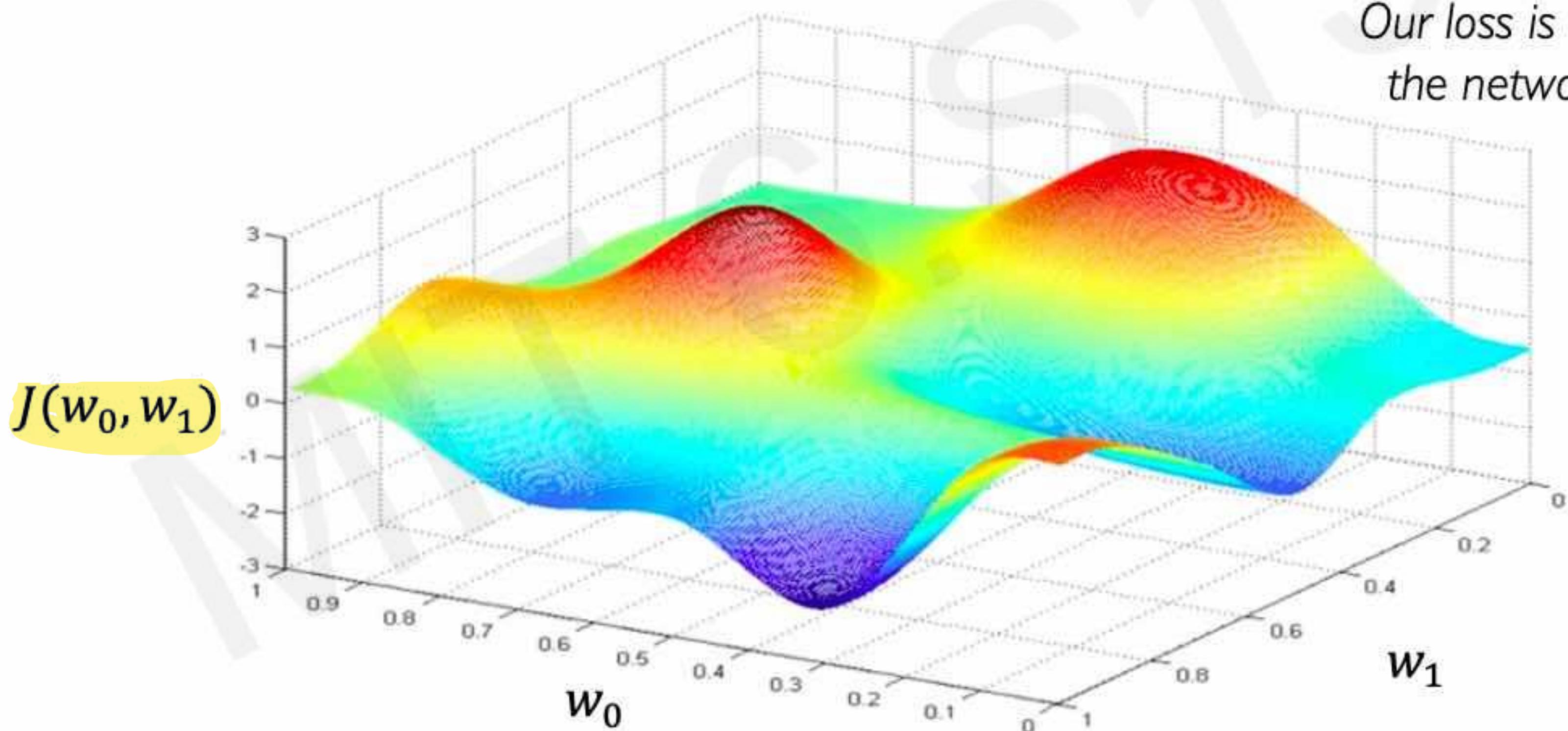


Remember:

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

Loss Optimization

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

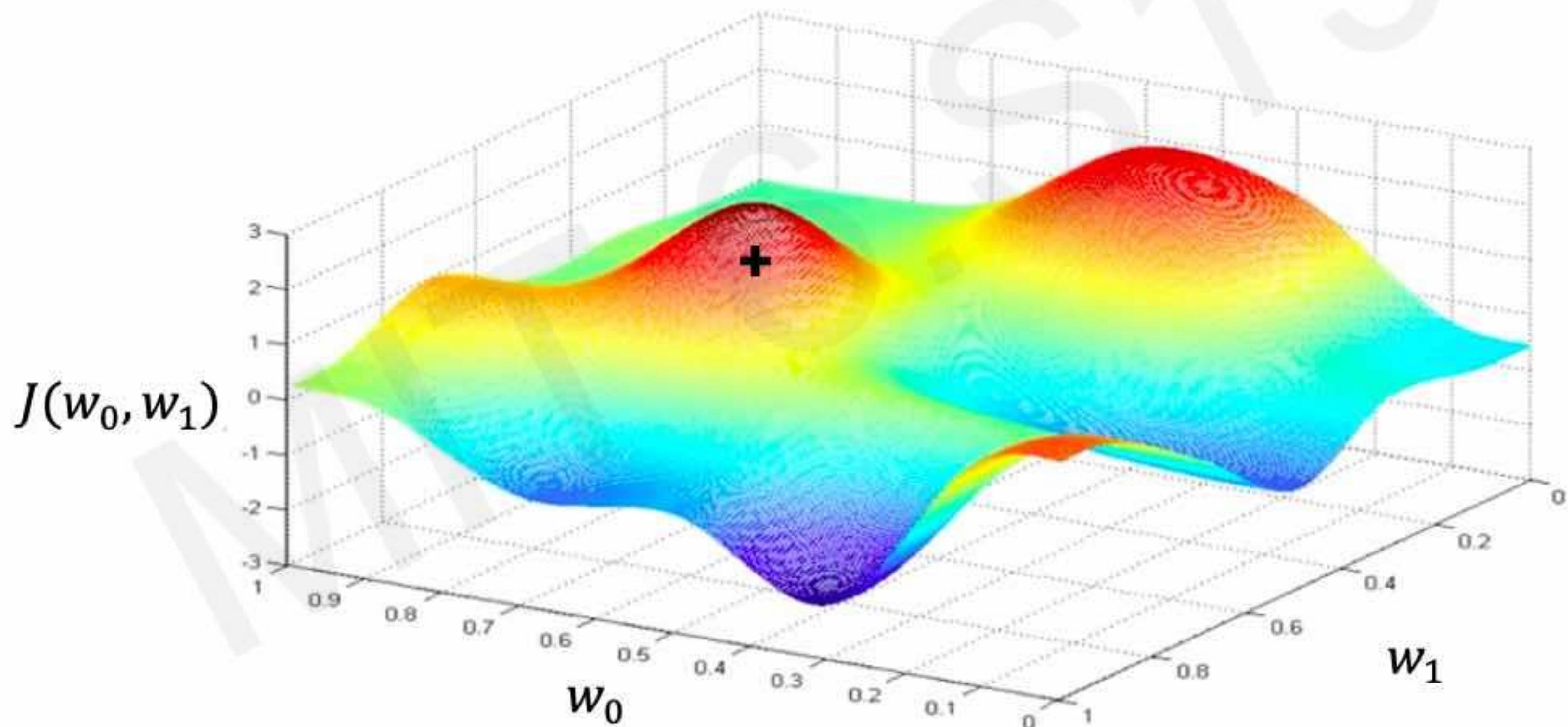


Remember:

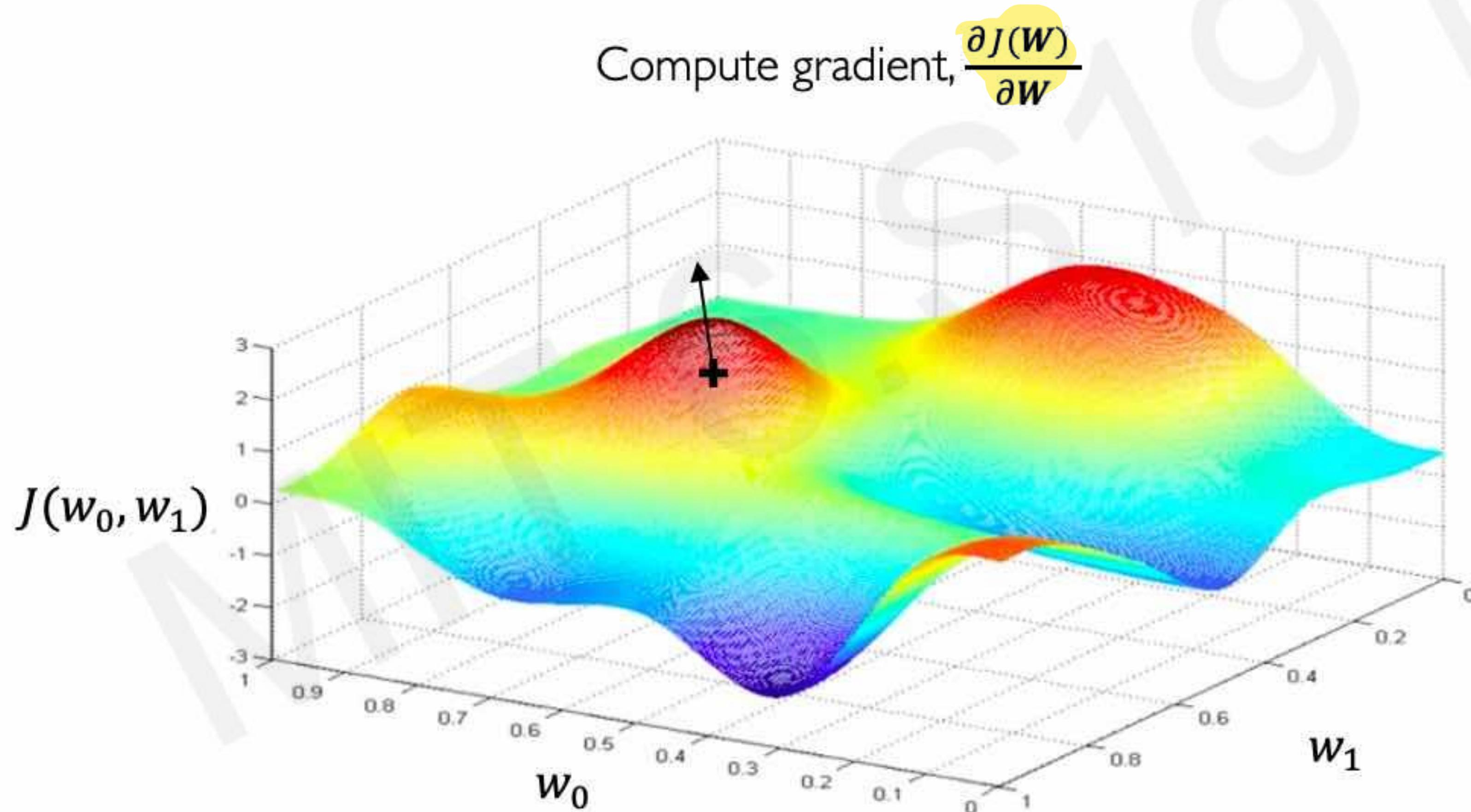
*Our loss is a function of
the network weights!*

Loss Optimization

Randomly pick an initial (w_0, w_1)

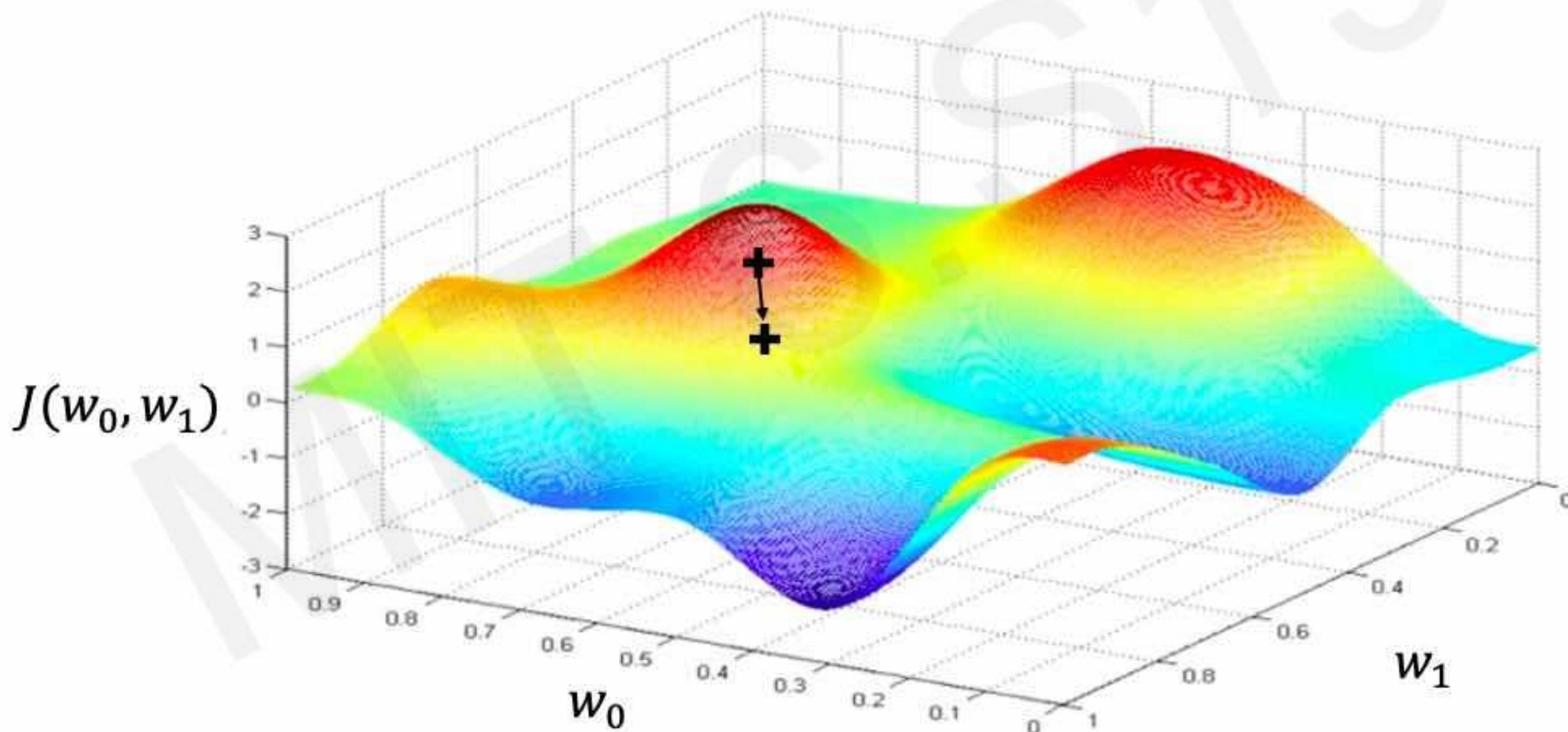


Loss Optimization



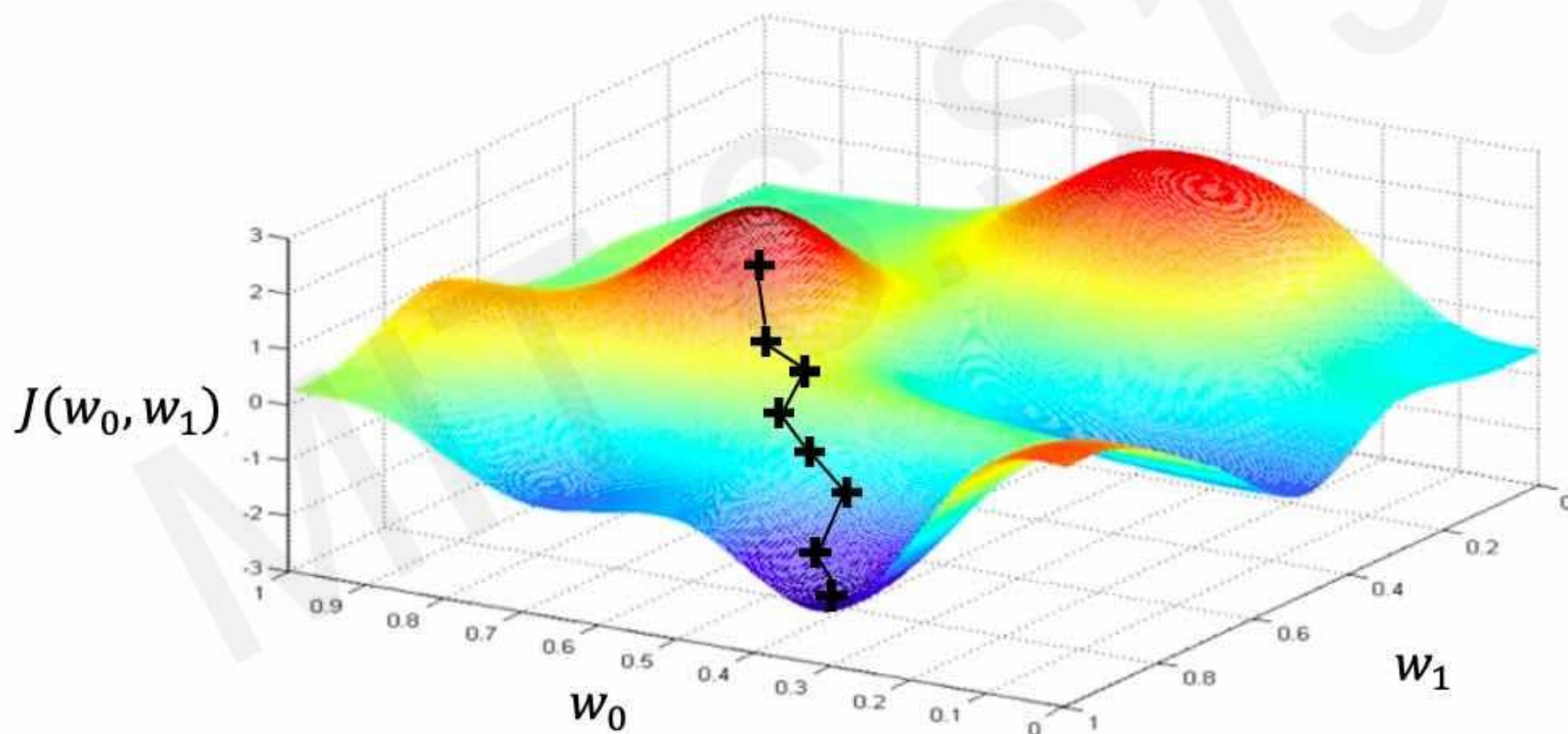
Loss Optimization

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



Gradient Descent

Algorithm

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

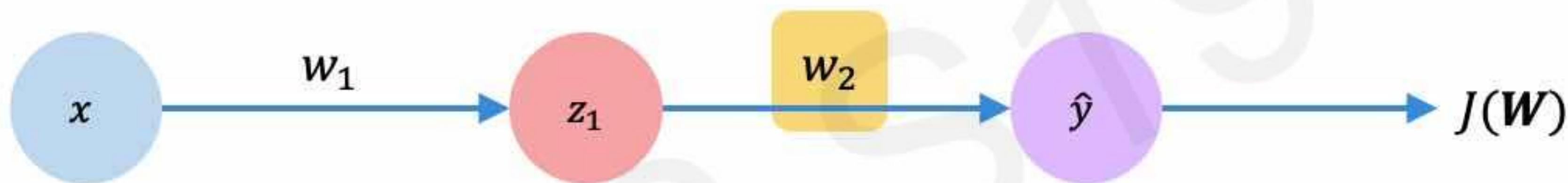
Algorithm

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

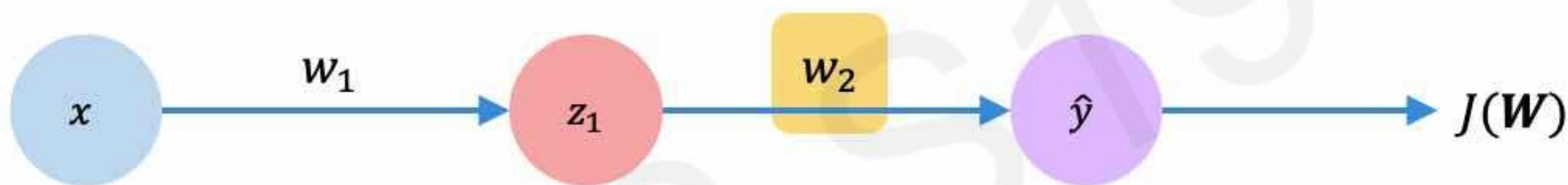


Computing Gradients: Backpropagation



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

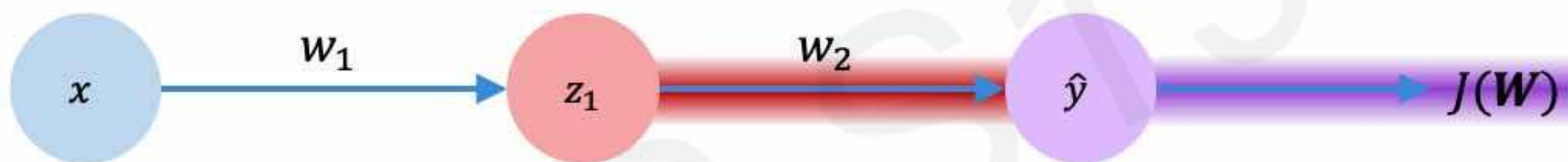
Computing Gradients: Backpropagation



$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{w}_2} =$$

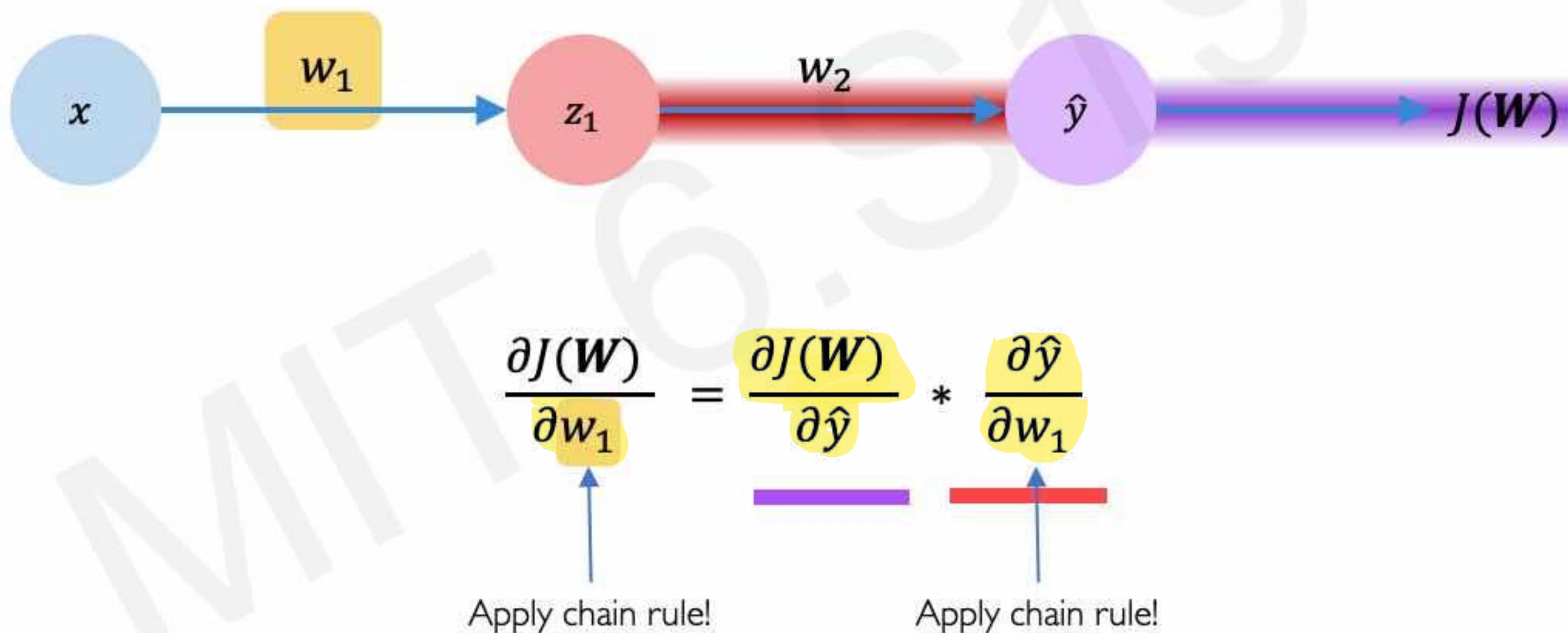
Let's use the chain rule!

Computing Gradients: Backpropagation

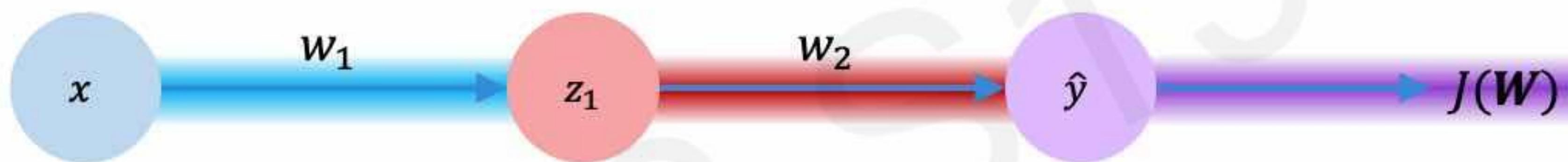


$$\frac{\partial J(\mathbf{W})}{\partial w_2} = \underbrace{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}}_{\text{purple bar}} * \underbrace{\frac{\partial \hat{y}}{\partial w_2}}_{\text{red bar}}$$

Computing Gradients: Backpropagation

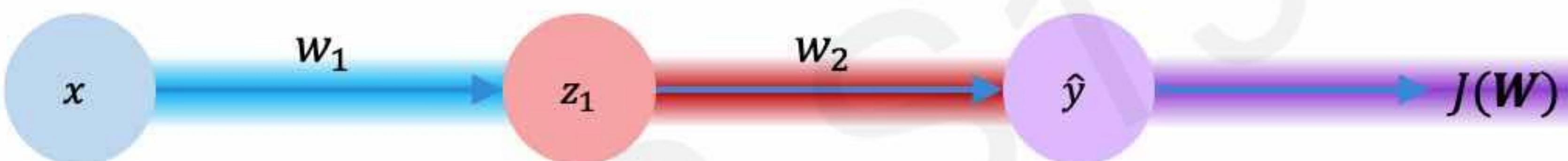


Computing Gradients: Backpropagation



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

Computing Gradients: Backpropagation

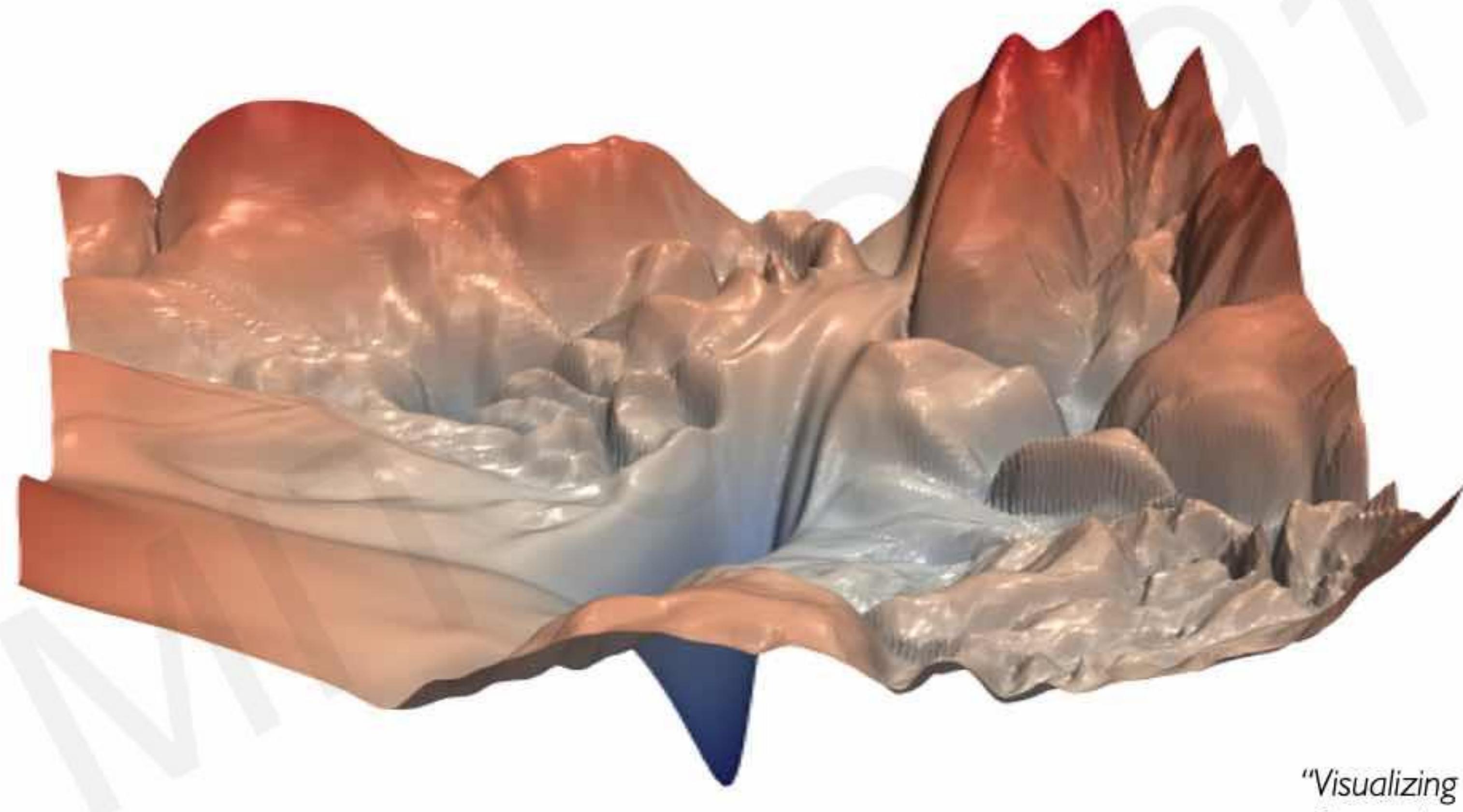


$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \underbrace{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}}_{\text{purple bar}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red bar}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue bar}}$$

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

Neural Networks in Practice: Optimization

Training Neural Networks is Difficult



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

Loss Functions Can Be Difficult to Optimize

Remember:

Optimization through gradient descent

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

Loss Functions Can Be Difficult to Optimize

Remember:

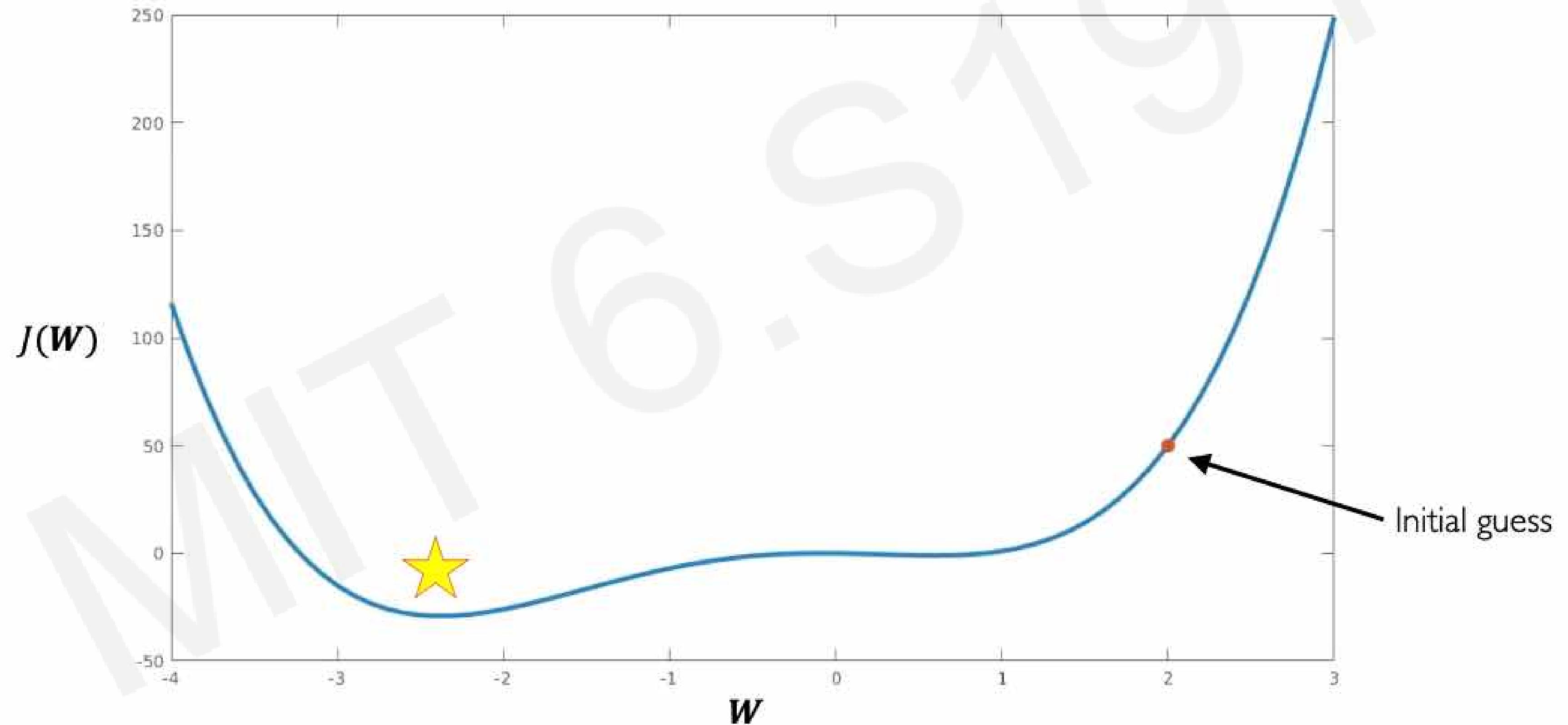
Optimization through gradient descent

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

How can we set the
learning rate?

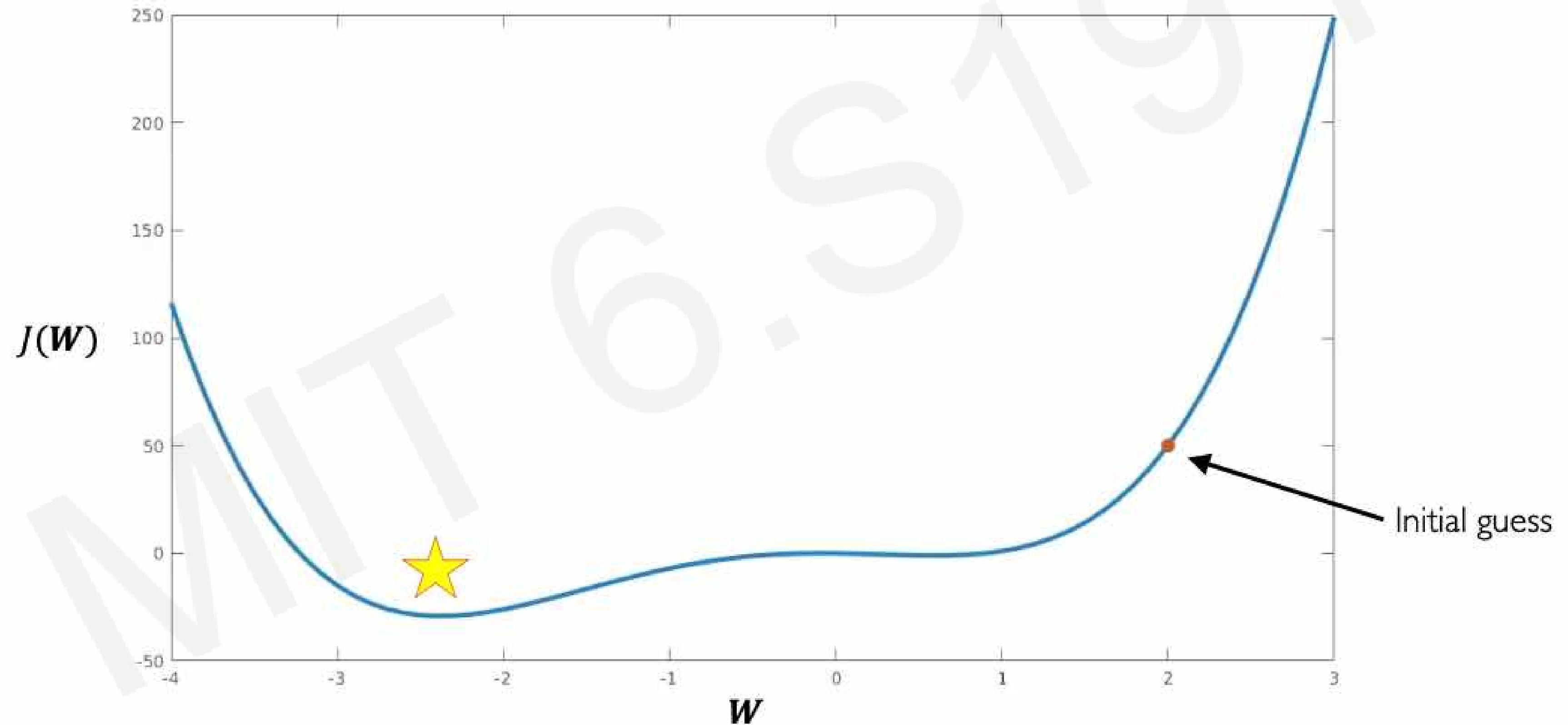
Setting the Learning Rate

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



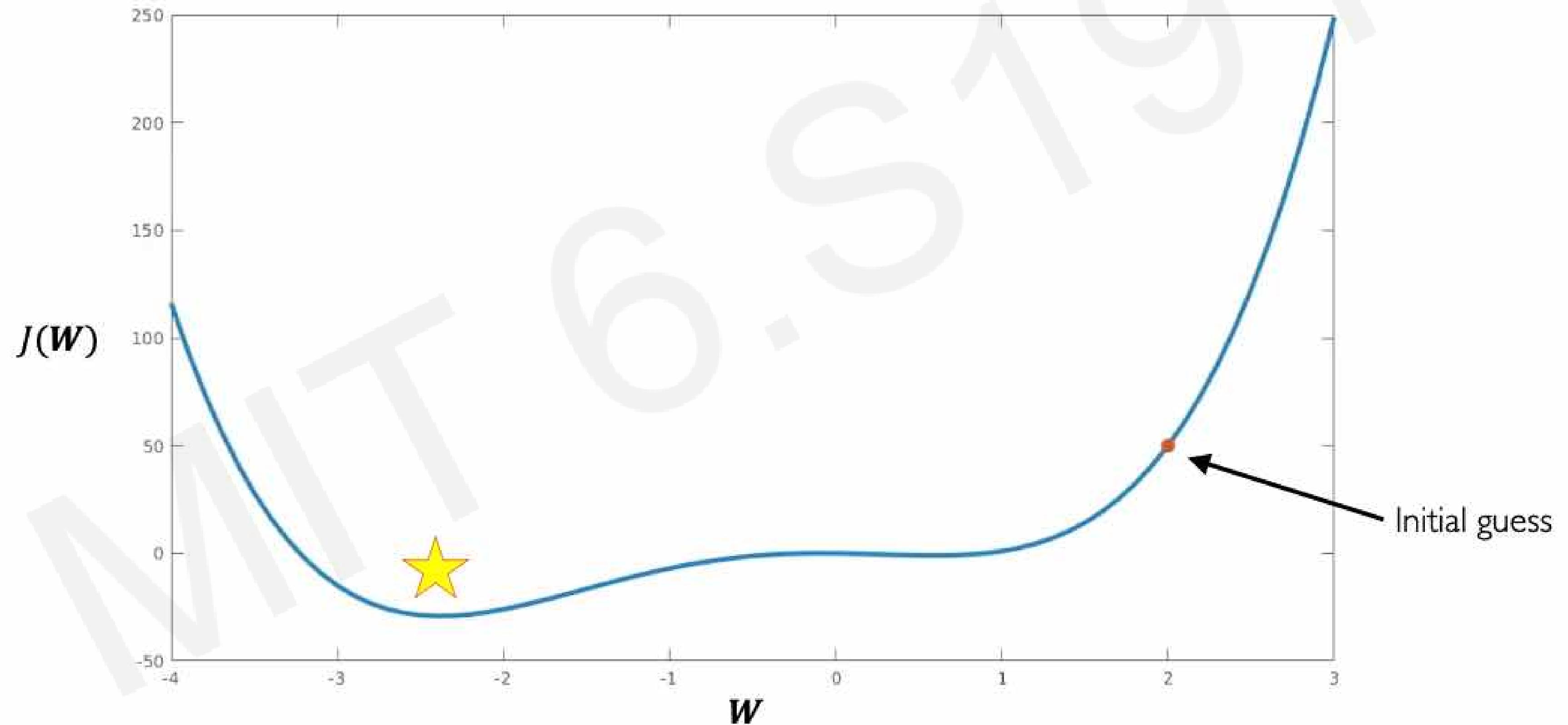
Setting the Learning Rate

Large learning rates overshoot, become unstable and diverge



Setting the Learning Rate

Stable learning rates converge smoothly and avoid local minima



How to deal with this?

Idea I:

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

How to deal with this?

Idea 1:

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

Idea 2:

Do something smarter!

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

Adaptive Learning Rates

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

Gradient Descent Algorithms

Algorithm

- SGD
- Adam
- Adadelta
- Adagrad
- RMSProp

TF Implementation



`tf.keras.optimizers.SGD`



`tf.keras.optimizers.Adam`



`tf.keras.optimizers.Adadelta`



`tf.keras.optimizers.Adagrad`



`tf.keras.optimizers.RMSProp`

Reference

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

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

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

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

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))
```

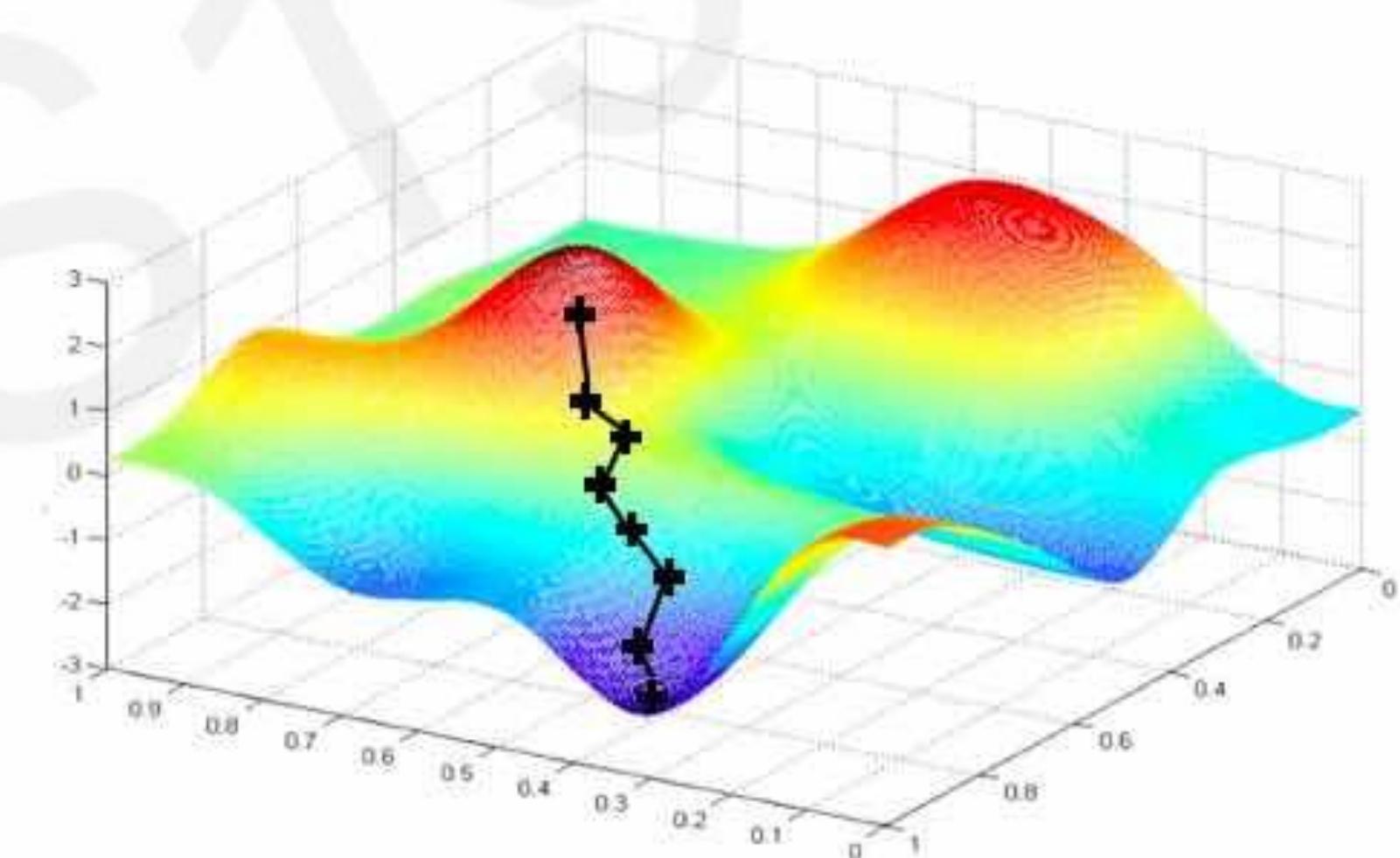


Neural Networks in Practice: **Mini-batches**

Gradient Descent

Algorithm

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

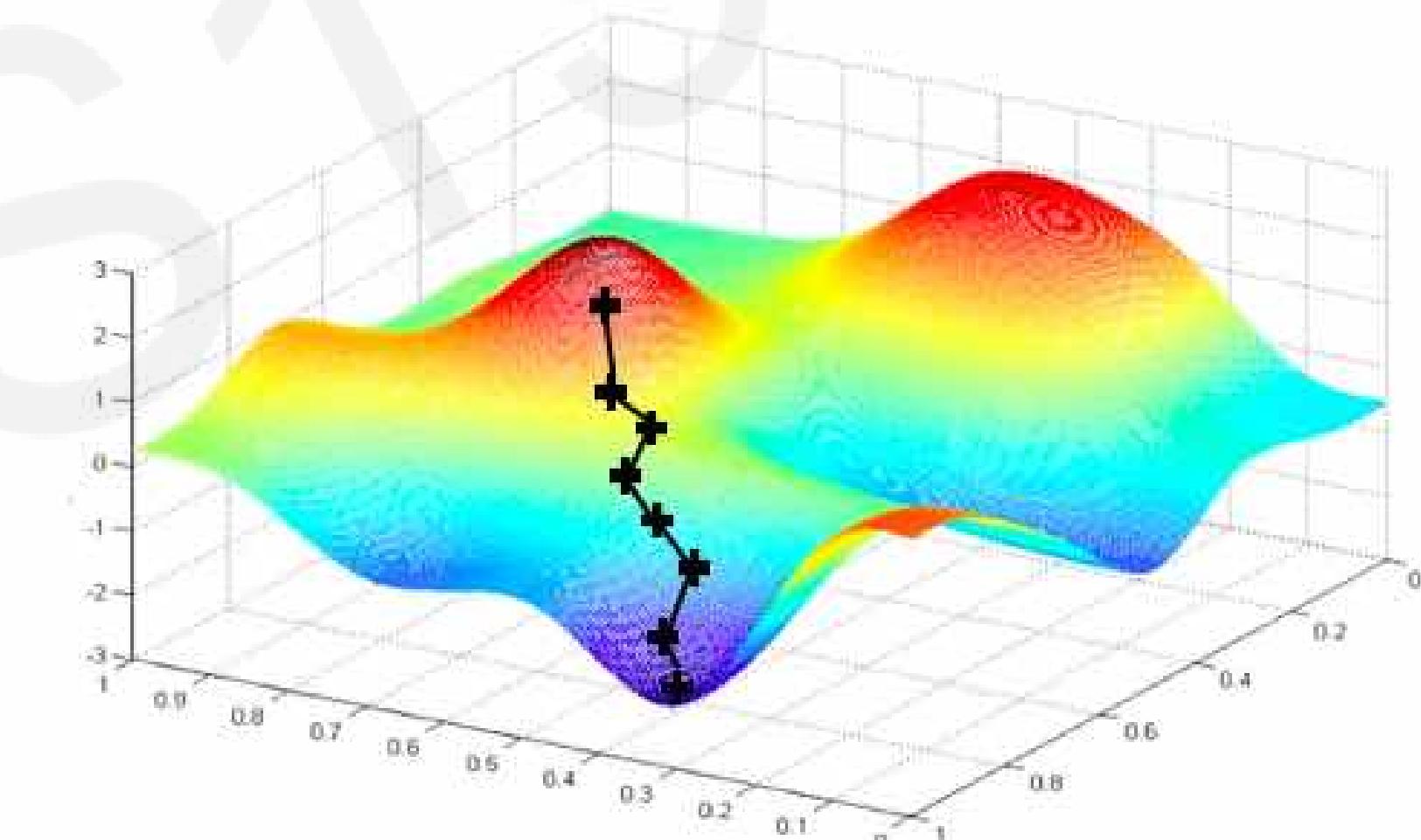


Gradient Descent

Algorithm

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

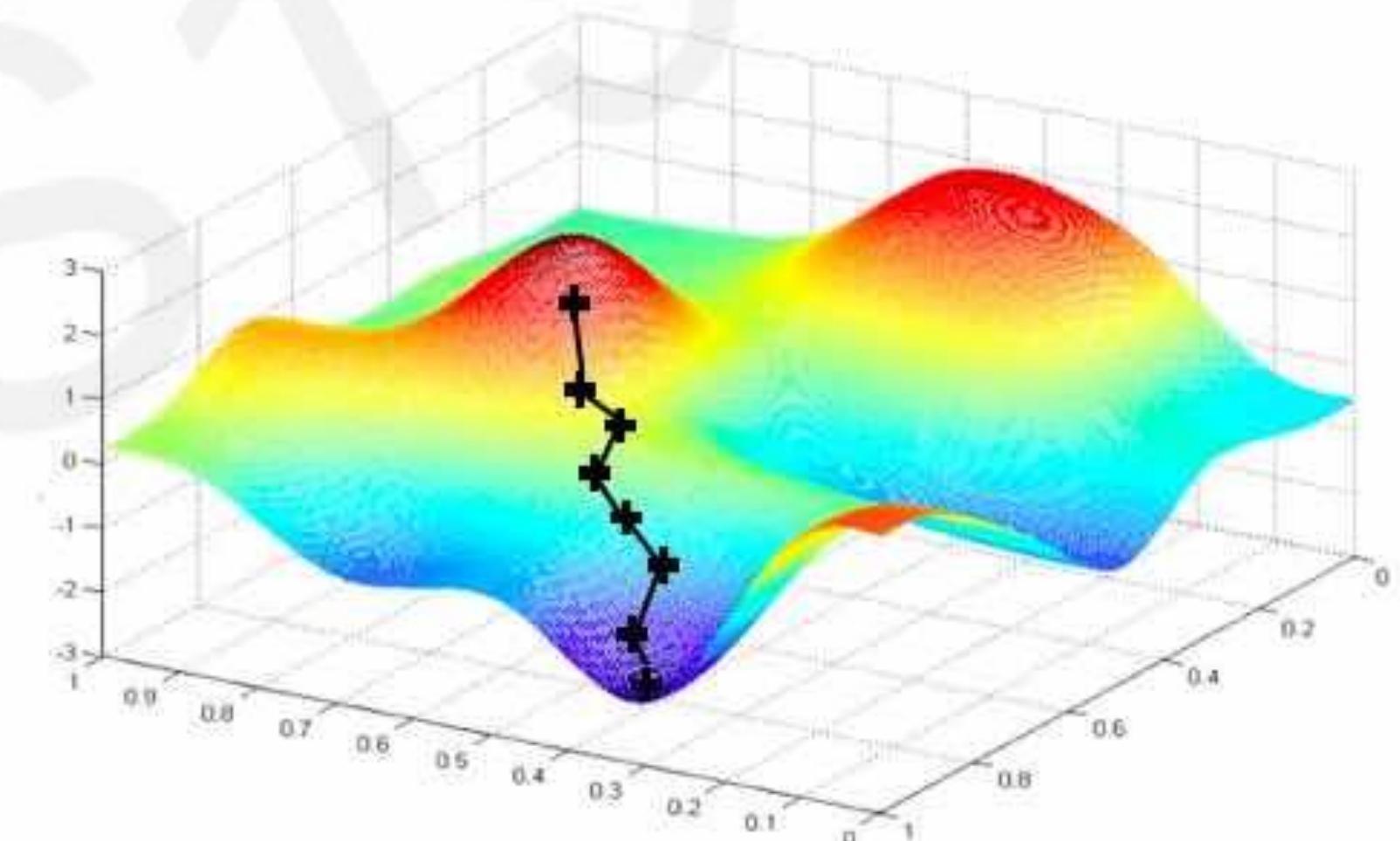
Can be very
computationally
intensive to compute!



Stochastic Gradient Descent

Algorithm

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

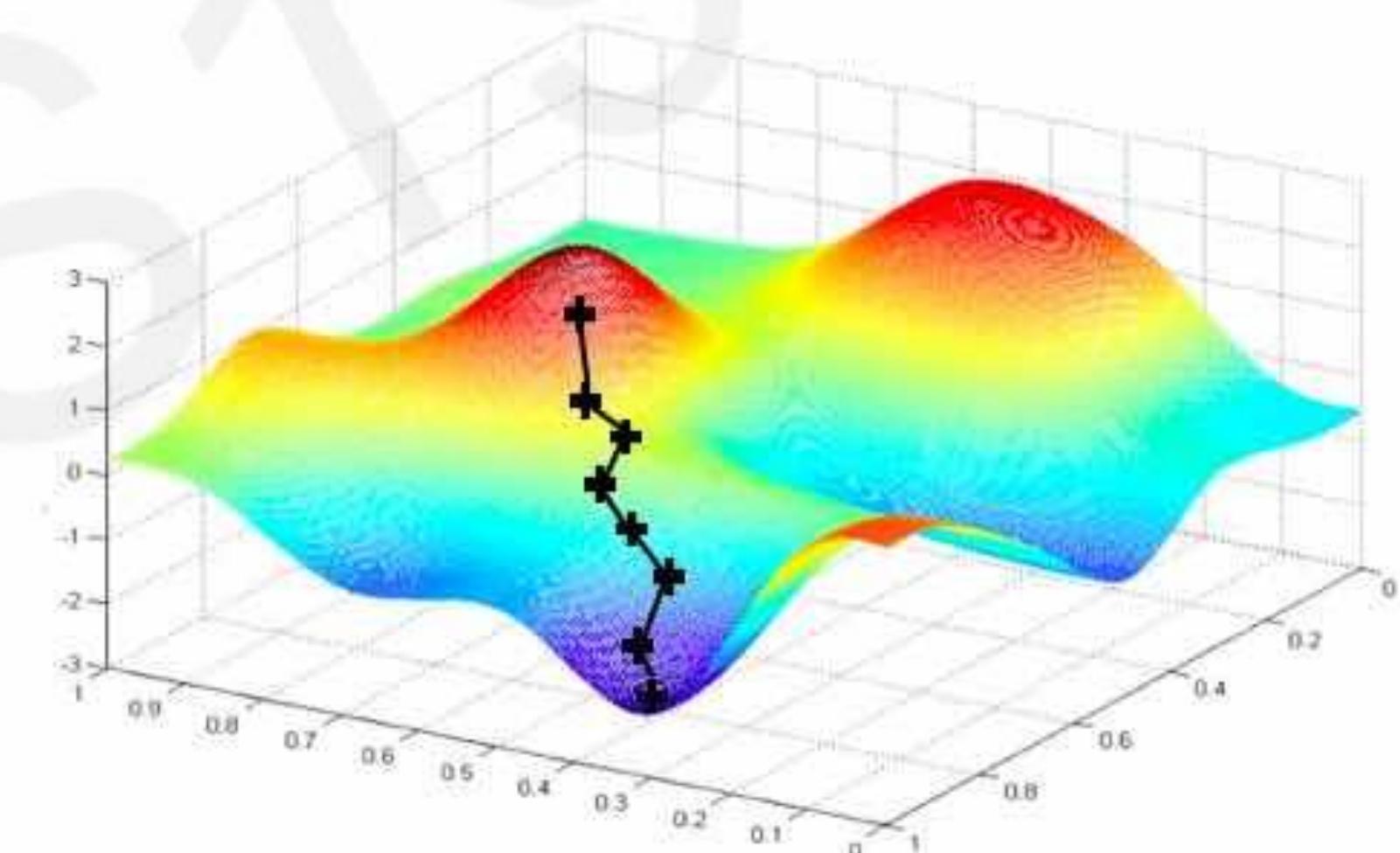


Stochastic Gradient Descent

Algorithm

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

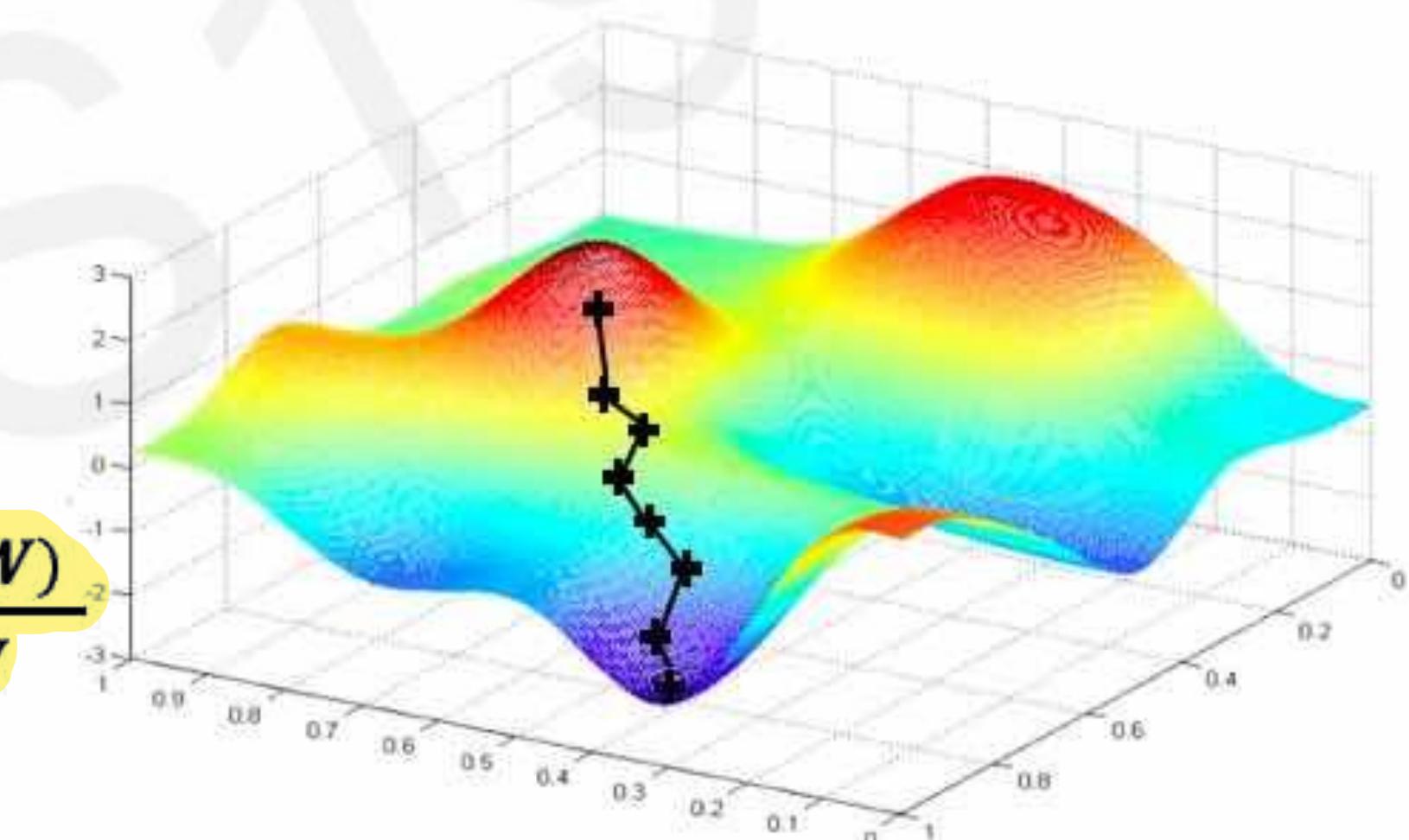
Easy to compute but
very noisy (stochastic)!



Stochastic Gradient Descent

Algorithm

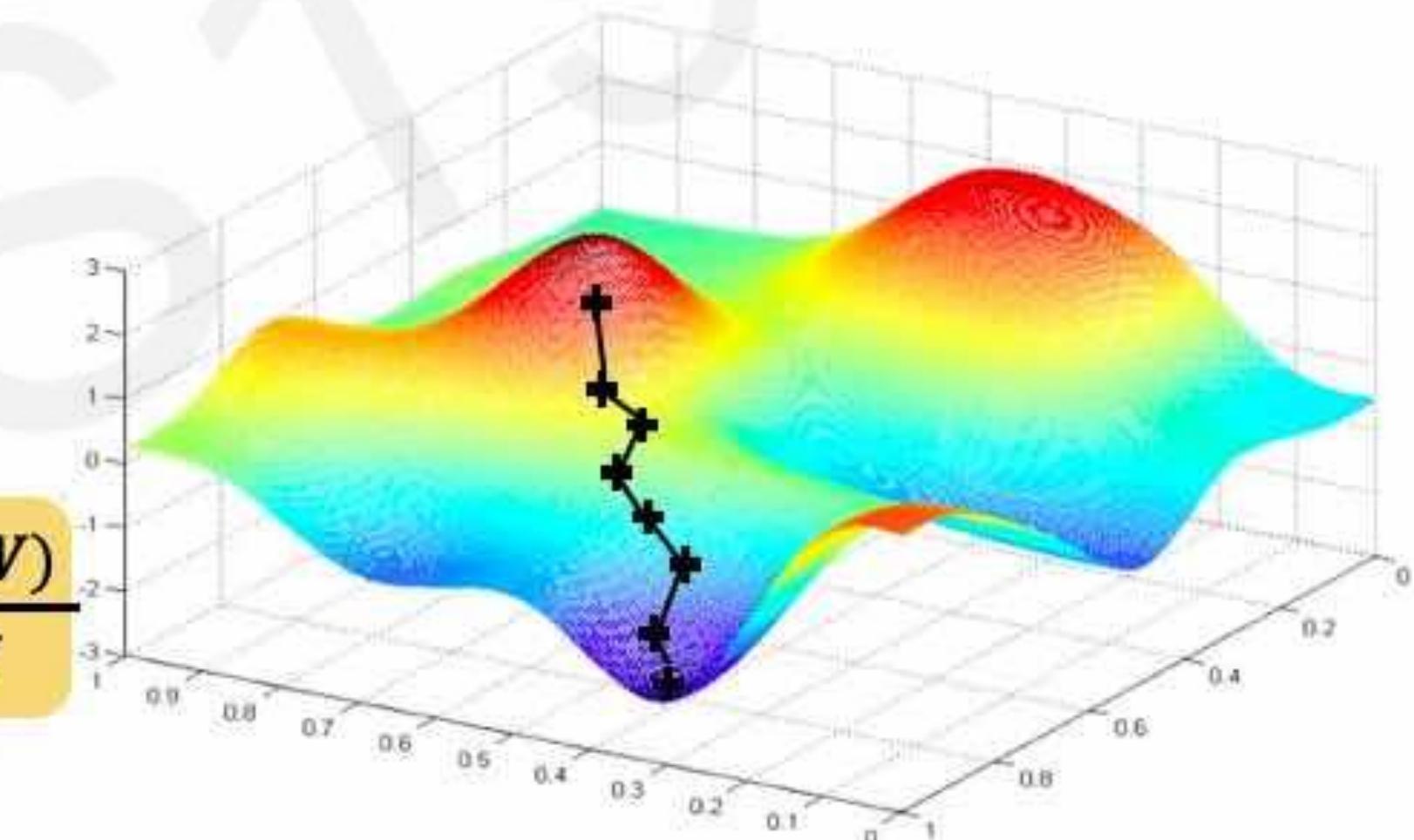
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights, $\mathbf{W}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



Stochastic Gradient Descent

Algorithm

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



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

Mini-batches while training

More accurate estimation of gradient

Smoother convergence

Allows for larger learning rates

Mini-batches while training

More accurate estimation of gradient

Smoother convergence

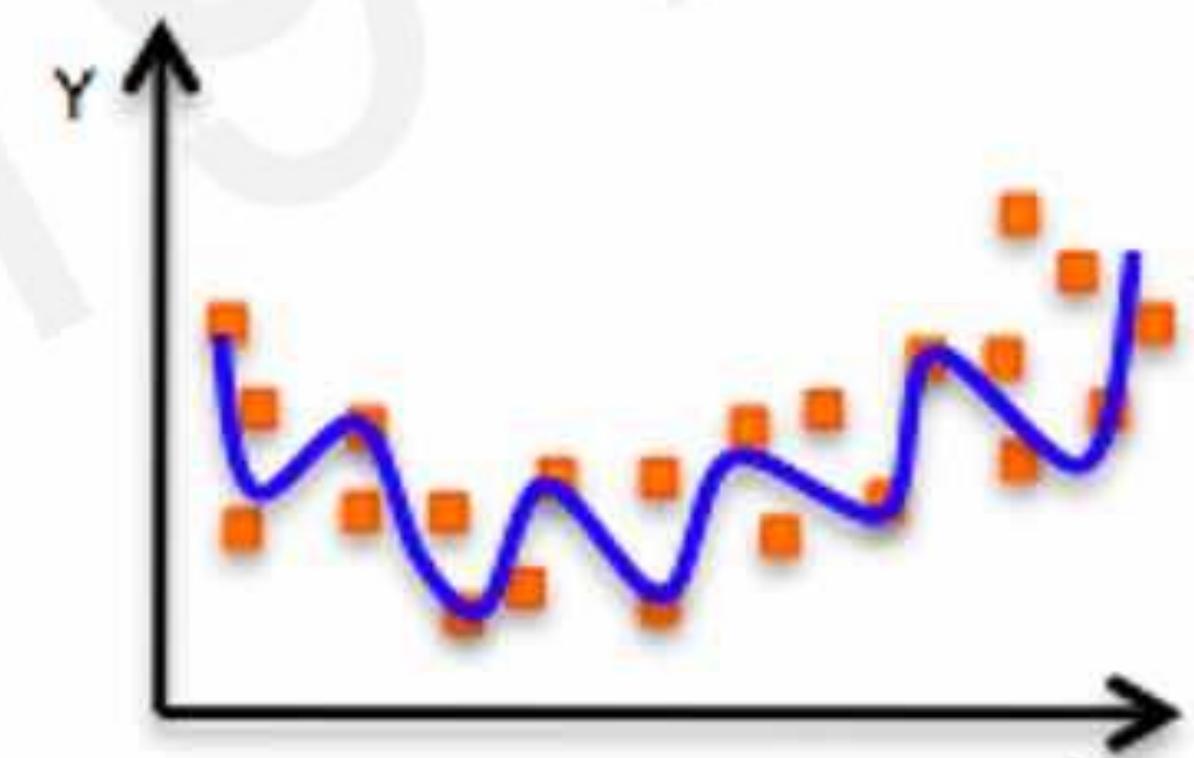
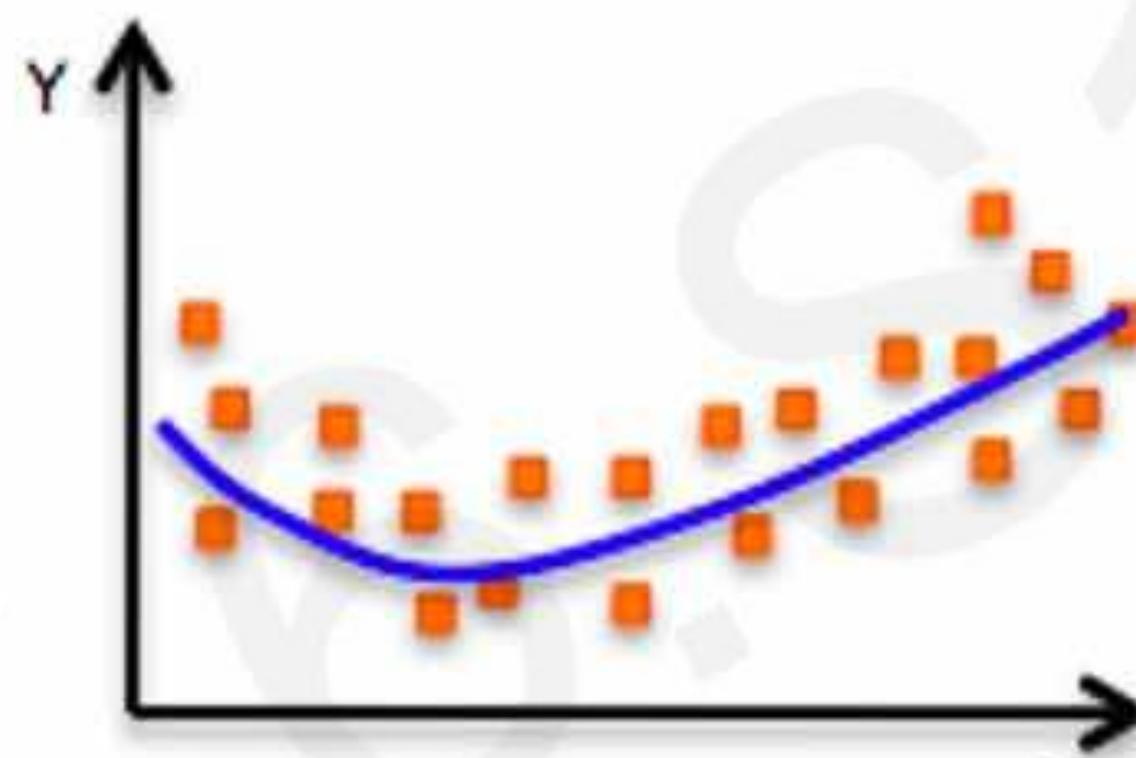
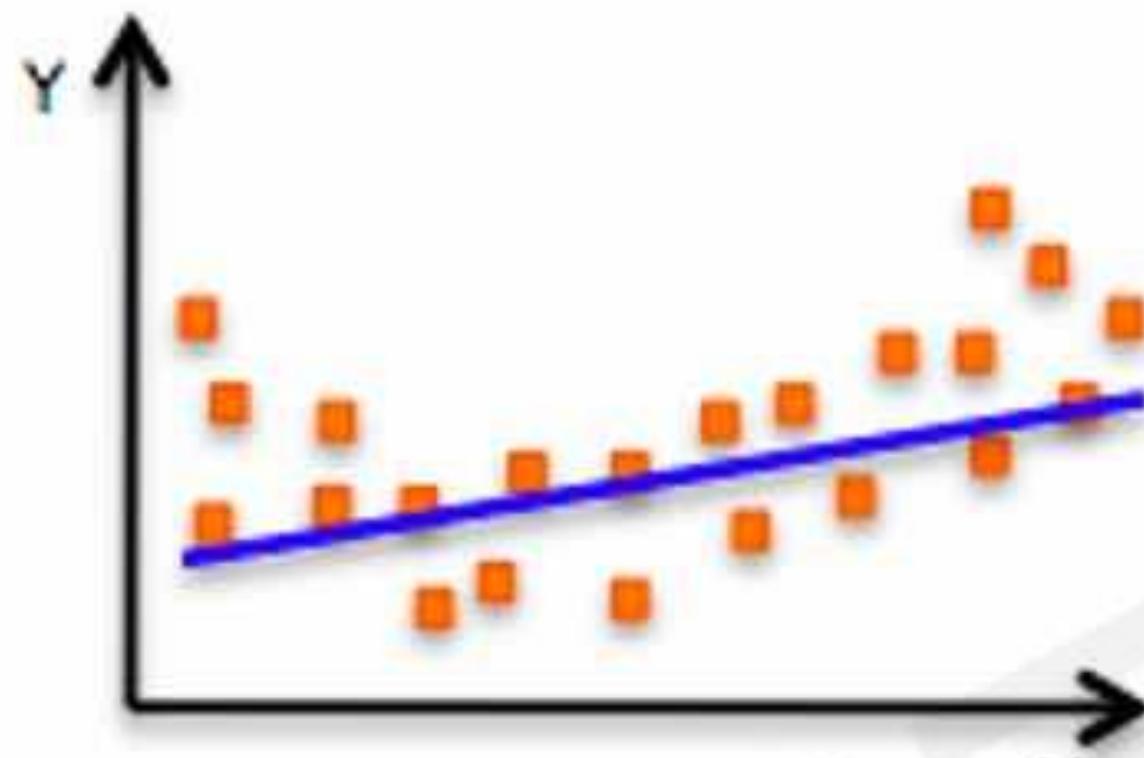
Allows for larger learning rates

Mini-batches lead to fast training!

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

Neural Networks in Practice: Overfitting

The Problem of Overfitting



Underfitting

Model does not have capacity to fully learn the data

Ideal fit

Overfitting

Too complex, extra parameters, does not generalize well

Regularization

What is it?

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

Regularization

What is it?

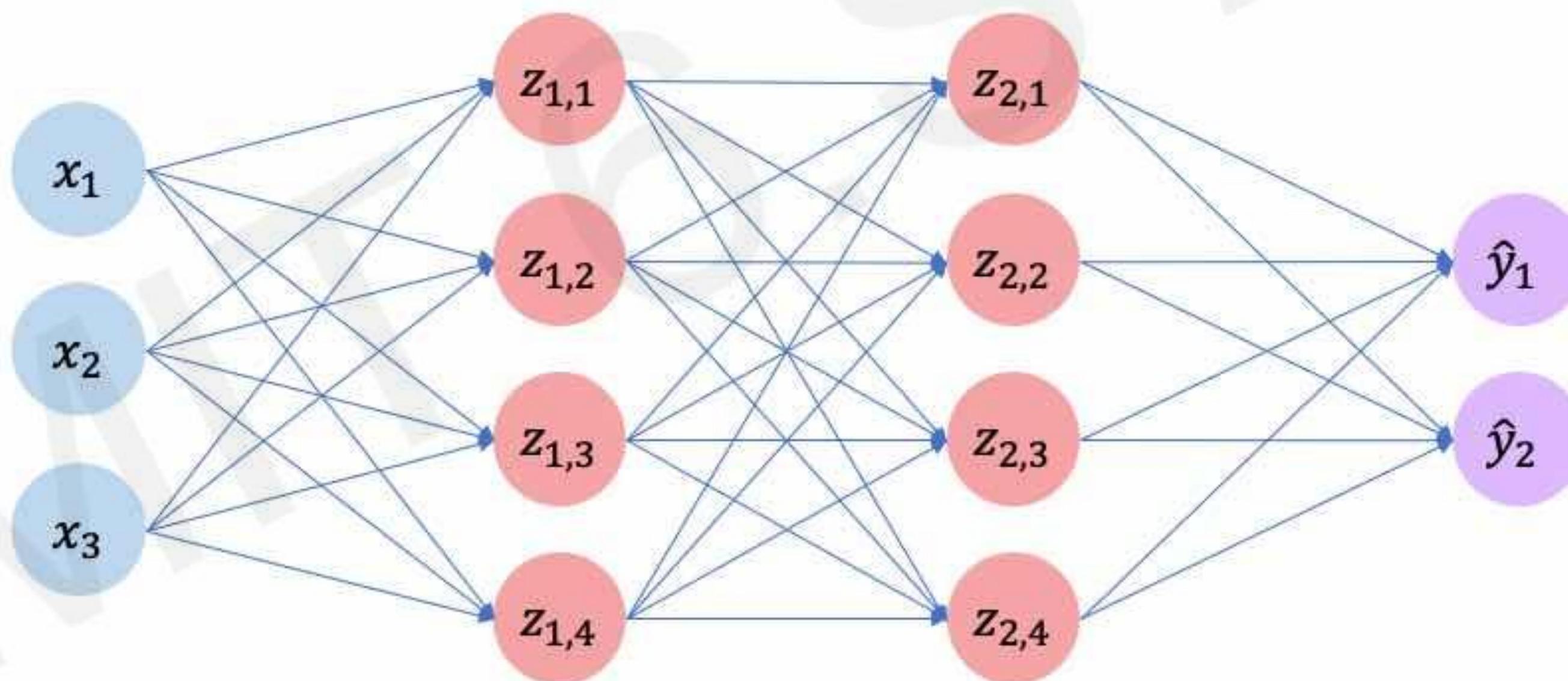
Technique that constrains our optimization problem to discourage complex models

Why do we need it?

Improve generalization of our model on unseen data

Regularization I: Dropout

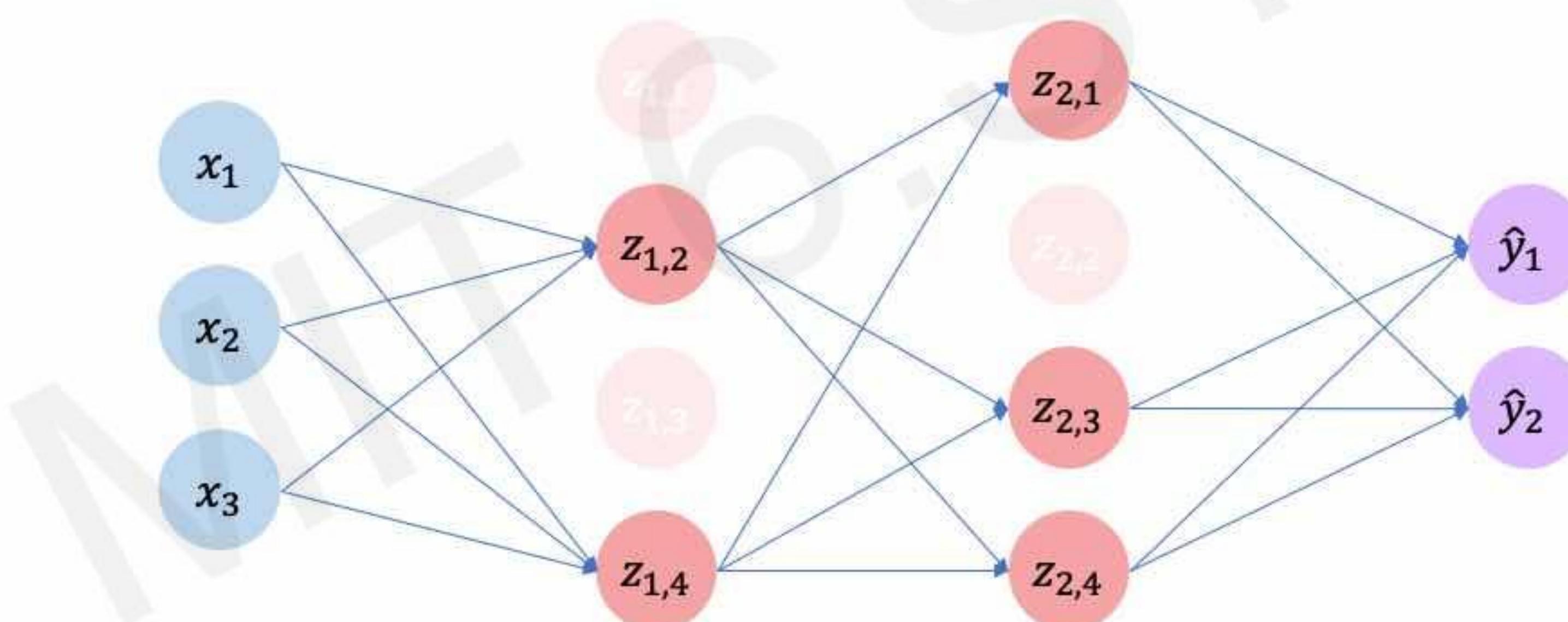
- During training, randomly set some activations to 0



Regularization I: Dropout

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

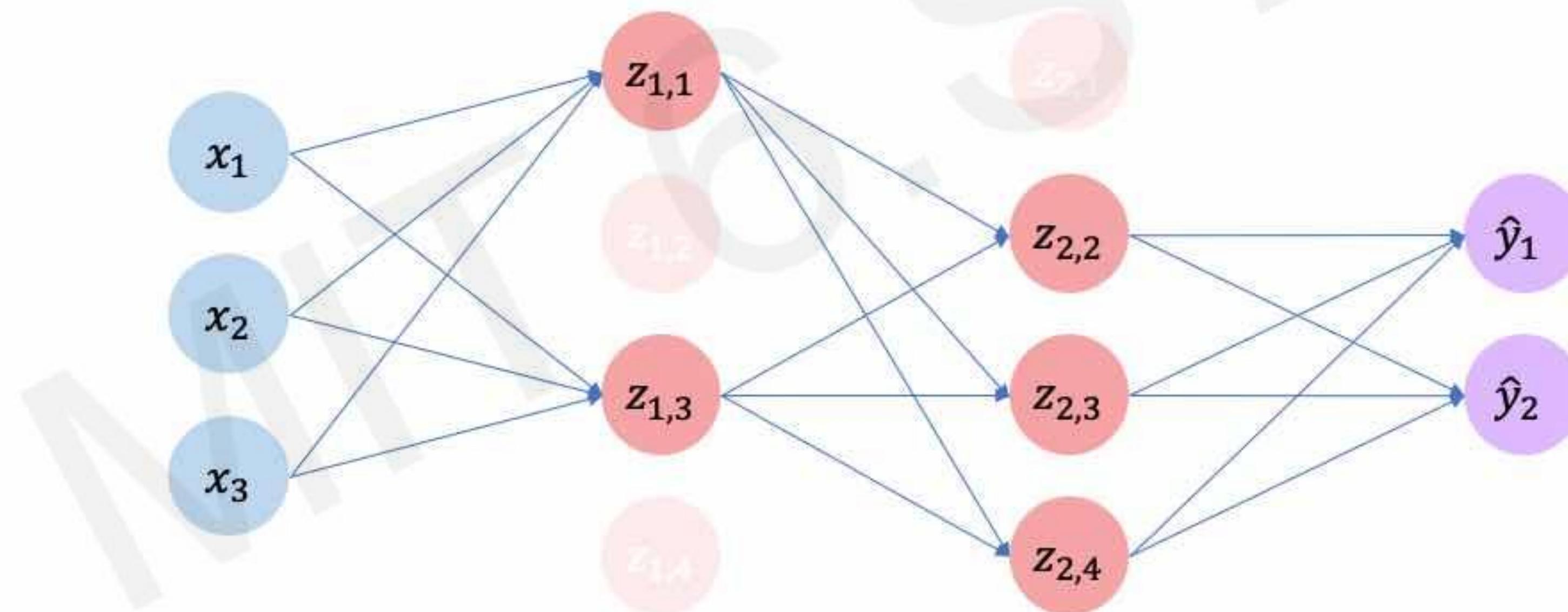
 `tf.keras.layers.Dropout(p=0.5)`



Regularization I: Dropout

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

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



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



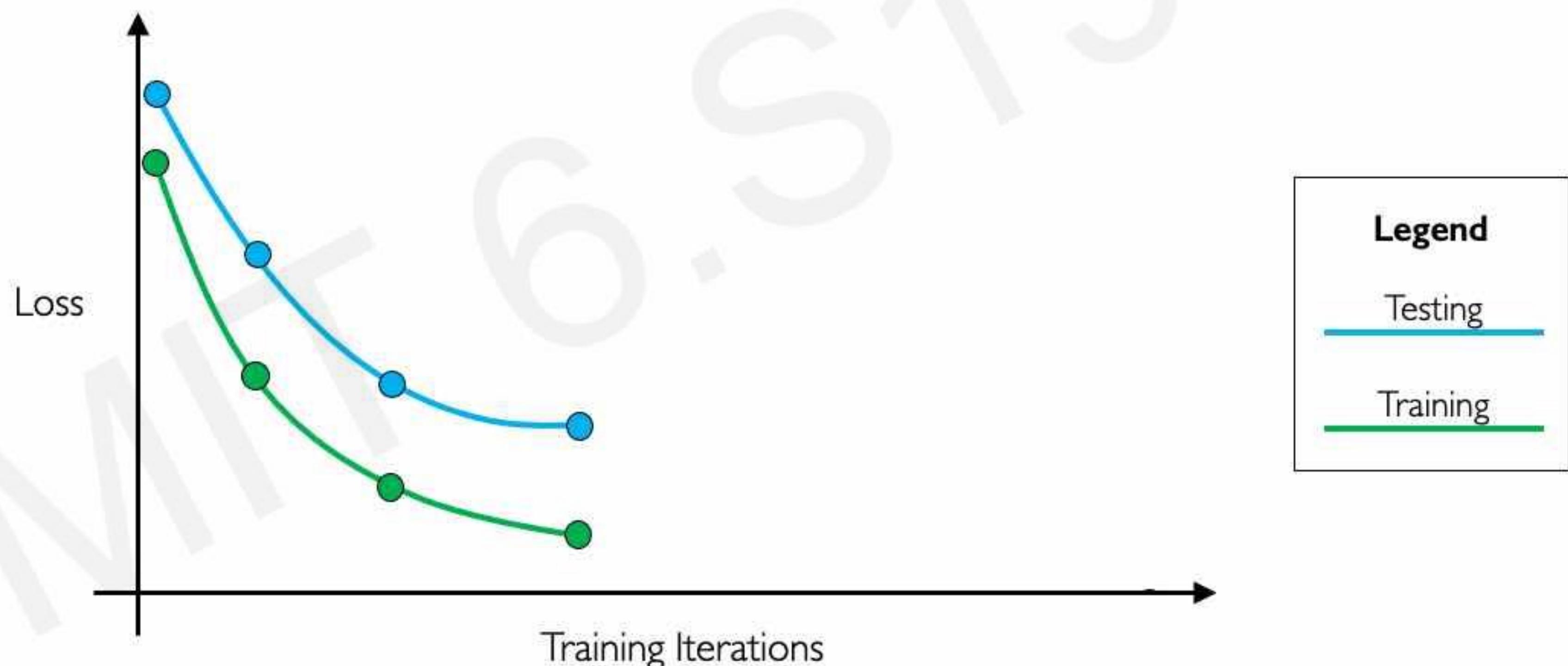
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



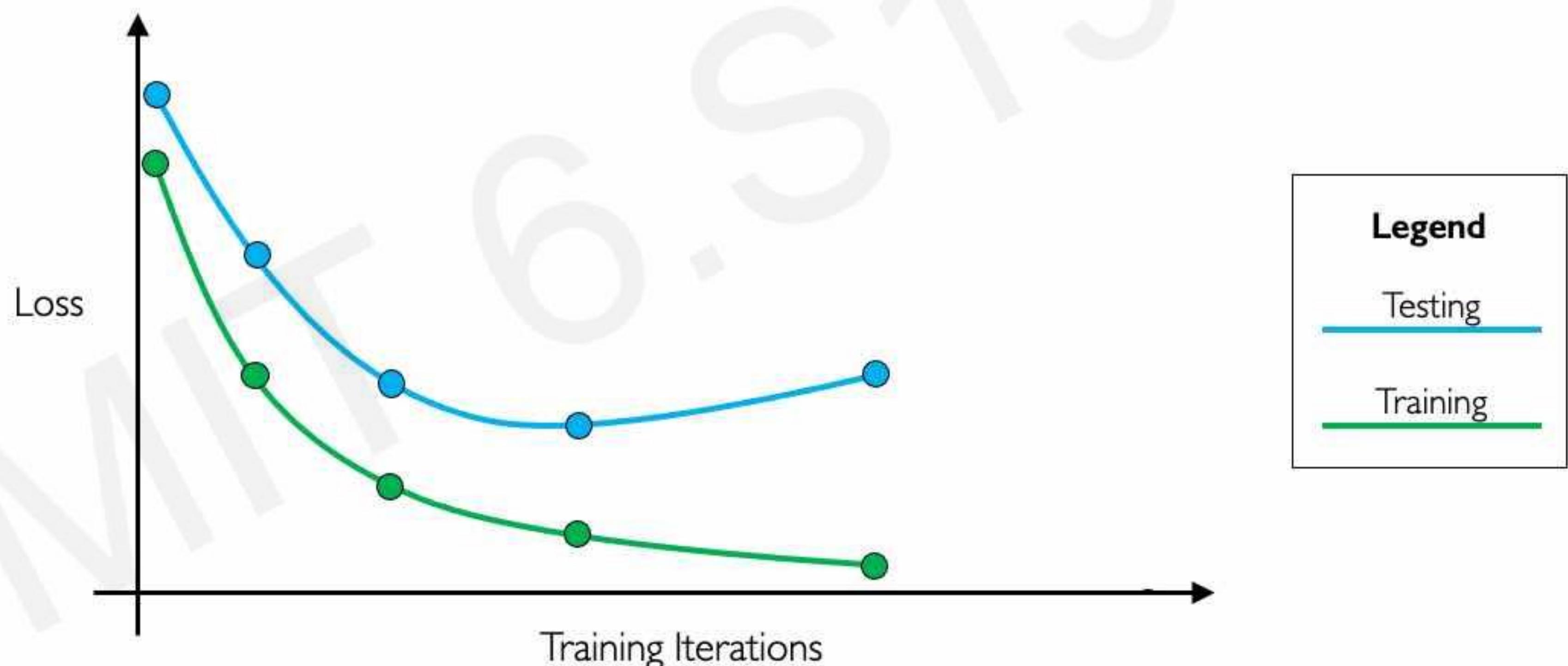
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



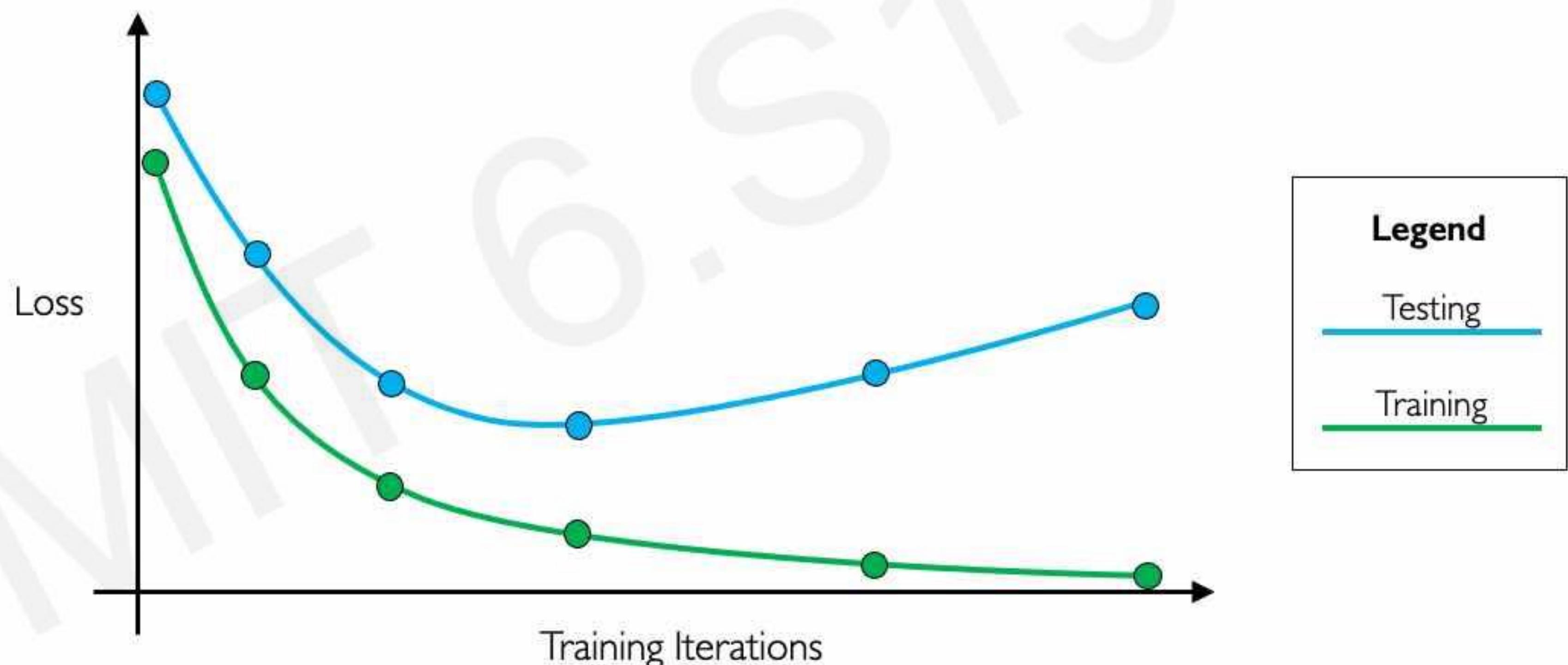
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



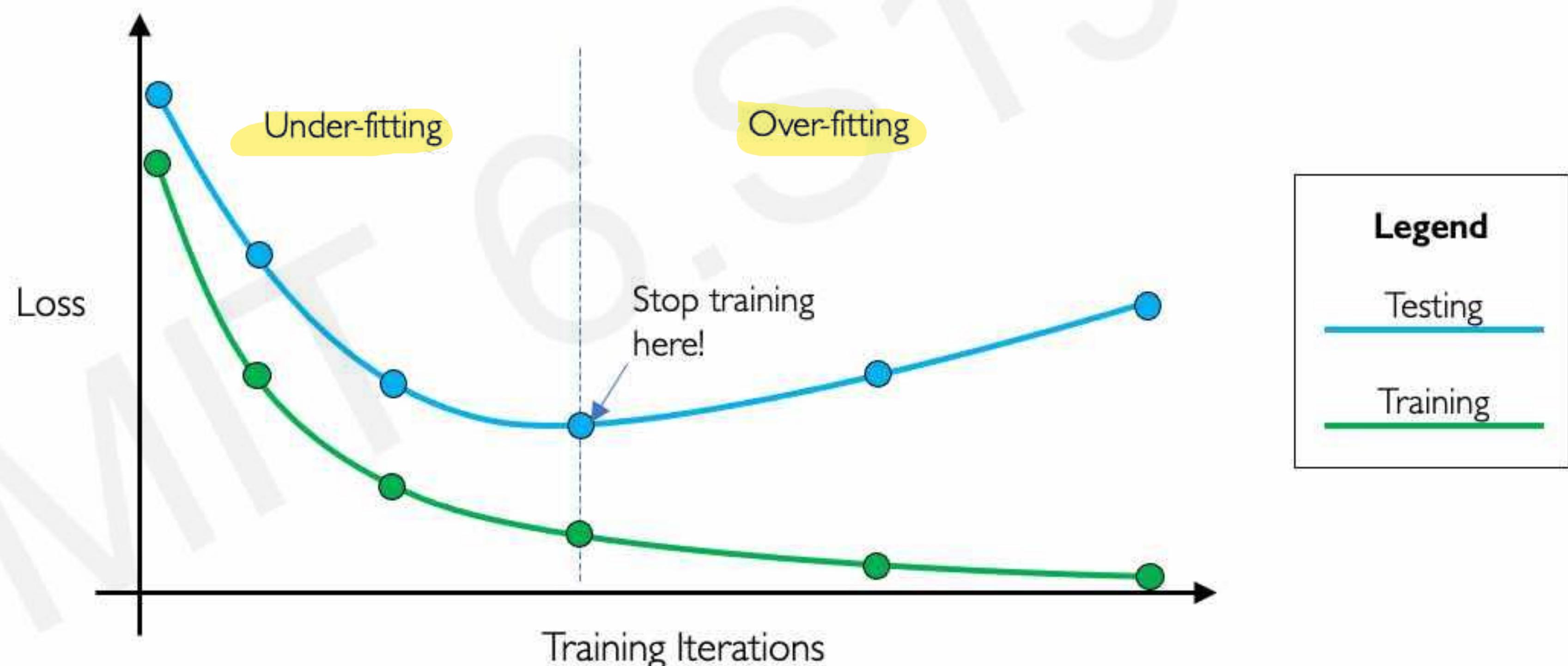
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



Regularization 2: Early Stopping

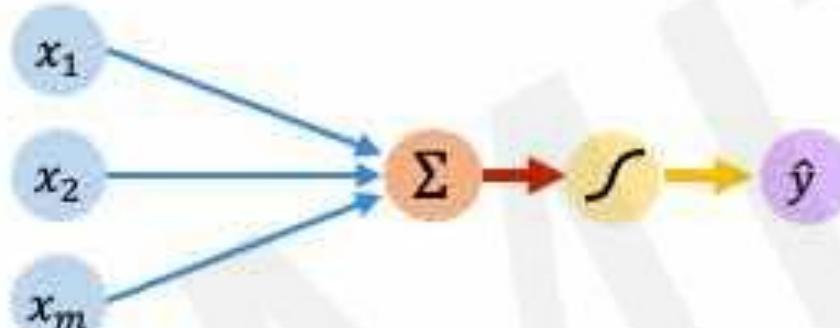
- Stop training before we have a chance to overfit



Core Foundation Review

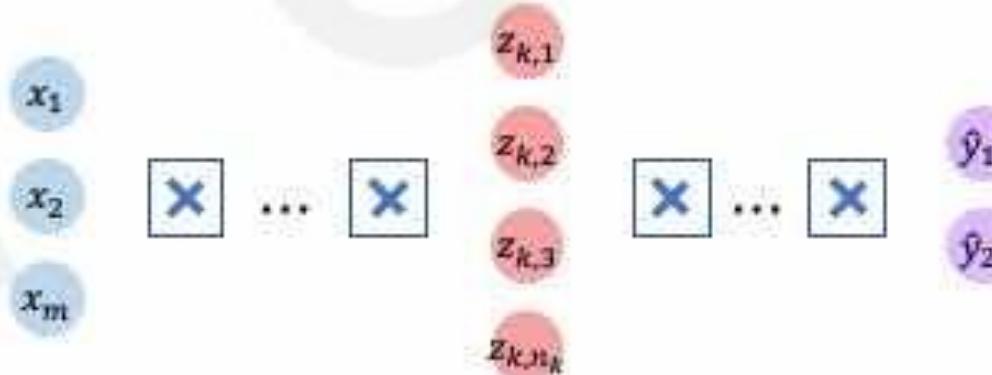
The Perceptron

- Structural building blocks
- Nonlinear activation functions



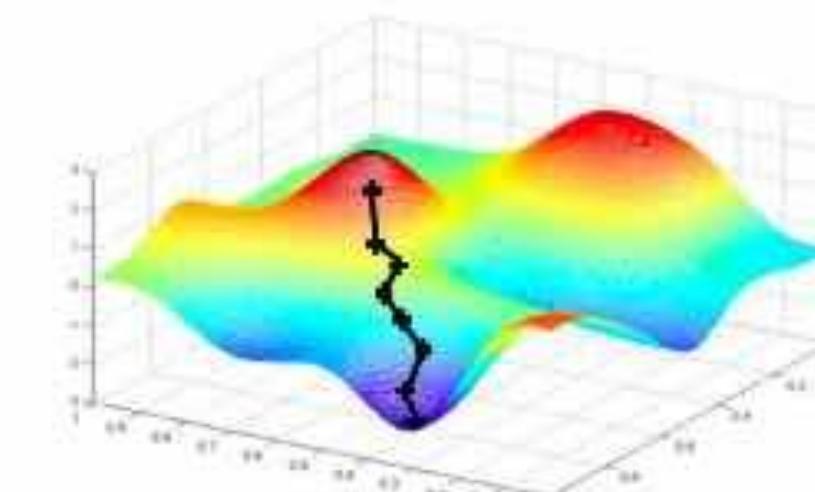
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization



6.S191: 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 the class Gather.Town or 10-250!

