

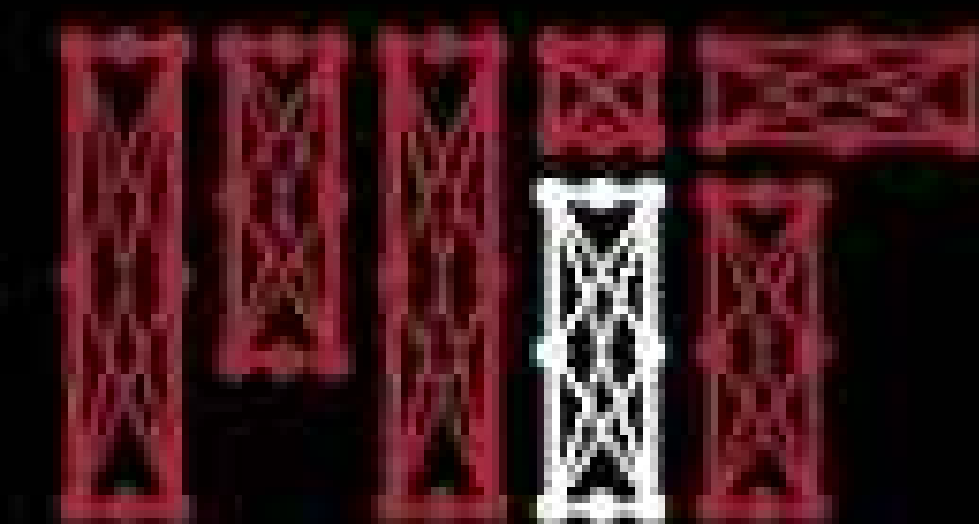
An abstract graphic on the left side of the slide, consisting of a complex network of blue lines and dots, resembling a neural network or a data structure, set against a black background.

Introduction to Deep Learning

Alexander Amini

MIT Introduction to Deep Learning

January 5, 2026



MIT Introduction to Deep Learning

🌐 introtodeeplearning.com 🐦 [@MITDeepLearning](https://twitter.com/MITDeepLearning)



“Seeing” the progress of deep learning throughout the years



2015

Goodfellow et al.



2018

Karras, Laine, Aila.



2020

MIT Intro to Deep Learning

Language has transformed the way that we interact with deep learning



2022

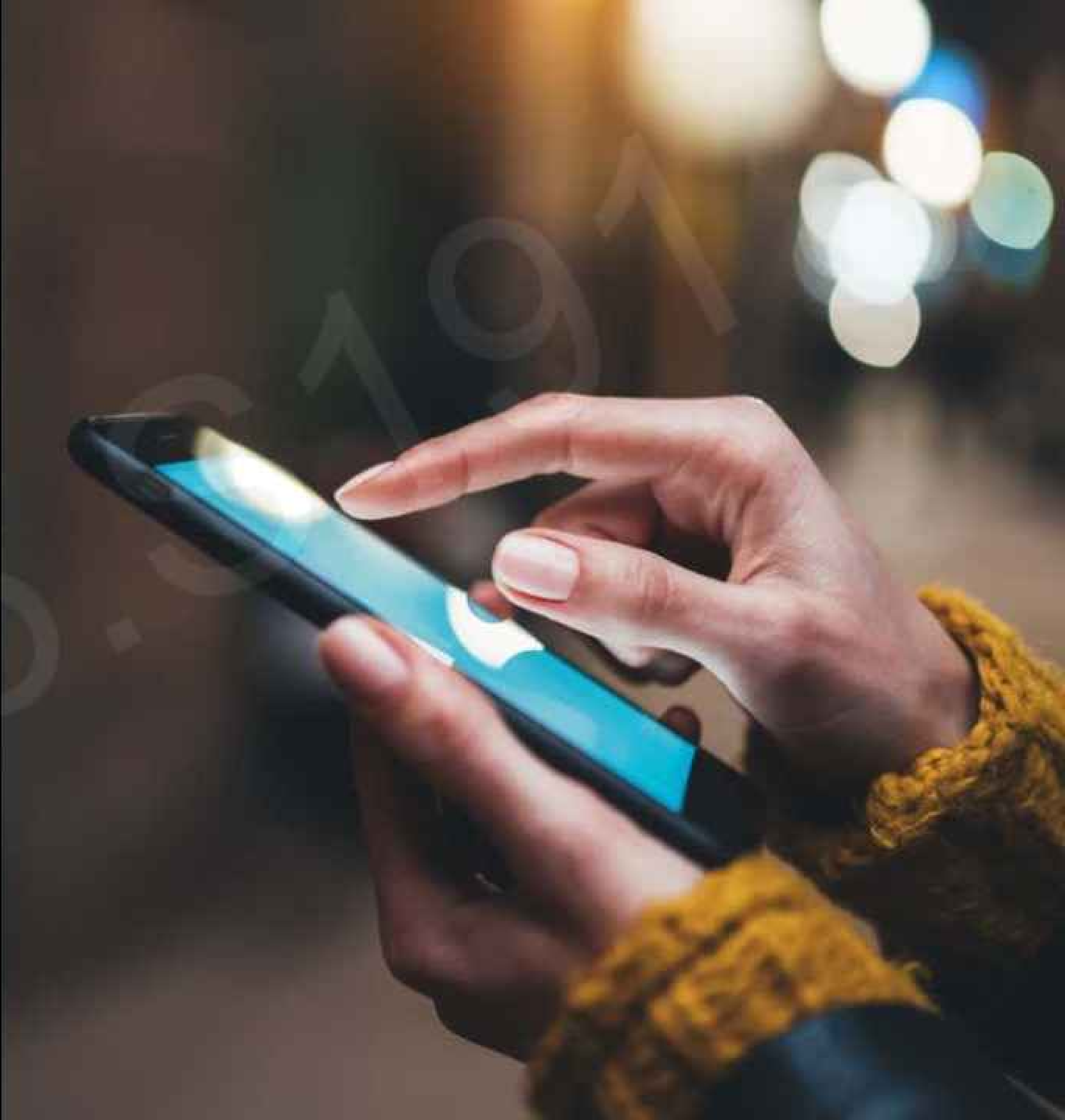
ChatGPT launches (GPT-3)



2023

The GPT-4 "wow" moment





That's pretty crazy when you think about it

One of the biggest “wow” moments with ChatGPT was the update from GPT-3.5 to GPT-4.

And now this open-source model with only 2.6B parameters that can run locally on your phone... is BETTER than the original GPT-4.

Unbelievable.

Benchmarking LFM2-2.6B-Exp vs. GPT-4

Benchmark	LFM2-2.6B-Exp (Liquid AI)	GPT-4 (Original)
IFEval (Strict Instruction Following)	88.13	79.0
Multi-IF (Complex Instructions)	73.05	65.0
IFBench (Formatting Accuracy)	44.40	24.0
GPQA (PhD-level Reasoning)	41.31	35.7
MMLU-Pro (Hard Expert Knowledge)	53.87	63.7
AIME25 (Olympiad-level Math)	22.67	15.0

 Export to Sheets



4:27 PM · Dec 25, 2025 · 253.5K Views



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



Teaching computers how to **learn a task** directly from **raw data**

Lecture Schedule



Intro to Deep Learning

Lecture 1

Jan. 5, 2026

[Slides] [Video] coming soon!



Deep Computer Vision

Lecture 3

Jan. 6, 2026

[Slides] [Video] coming soon!



Deep Reinforcement Learning

Lecture 5

Jan. 7, 2026

[Slides] [Video] coming soon!



AI for Science

Lecture 7

Jan. 8, 2026

[Info] [Slides] [Video] coming soon!



Guest Lecture

Lecture 9

Jan. 9, 2026

[Slides] [Video] coming soon!



Deep Sequence Modeling

Lecture 2

Jan. 5, 2026

[Slides] [Video] coming soon!



Deep Generative Modeling

Lecture 4

Jan. 6, 2026

[Slides] [Video] coming soon!



New Frontiers

Lecture 6

Jan. 7, 2026

[Slides] [Video] coming soon!



Secrets to Massively Parallel Training

Lecture 8

Jan. 8, 2026

[Info] [Slides] [Video] coming soon!



Guest Lecture

Lecture 10

Jan. 9, 2026

[Slides] [Video] coming soon!



Deep Learning in Python; Music Generation

Software Lab 1

[Code] coming soon!



Facial Detection Systems

Software Lab 2

[Paper] [Code] coming soon!



Fine-Tune an LLM, You Must!

Software Lab 3

[Code] coming soon!



Final Project

Work on final projects



Project Presentations

Pitch your ideas, awards, and celebration!



- Jan 5 – Jan 9, 2026
- Lecture Breakdown
- Labs: TensorFlow & PyTorch
- Competitions
- Final Projects + Prizes!

Software Labs: TensorFlow *and* PyTorch



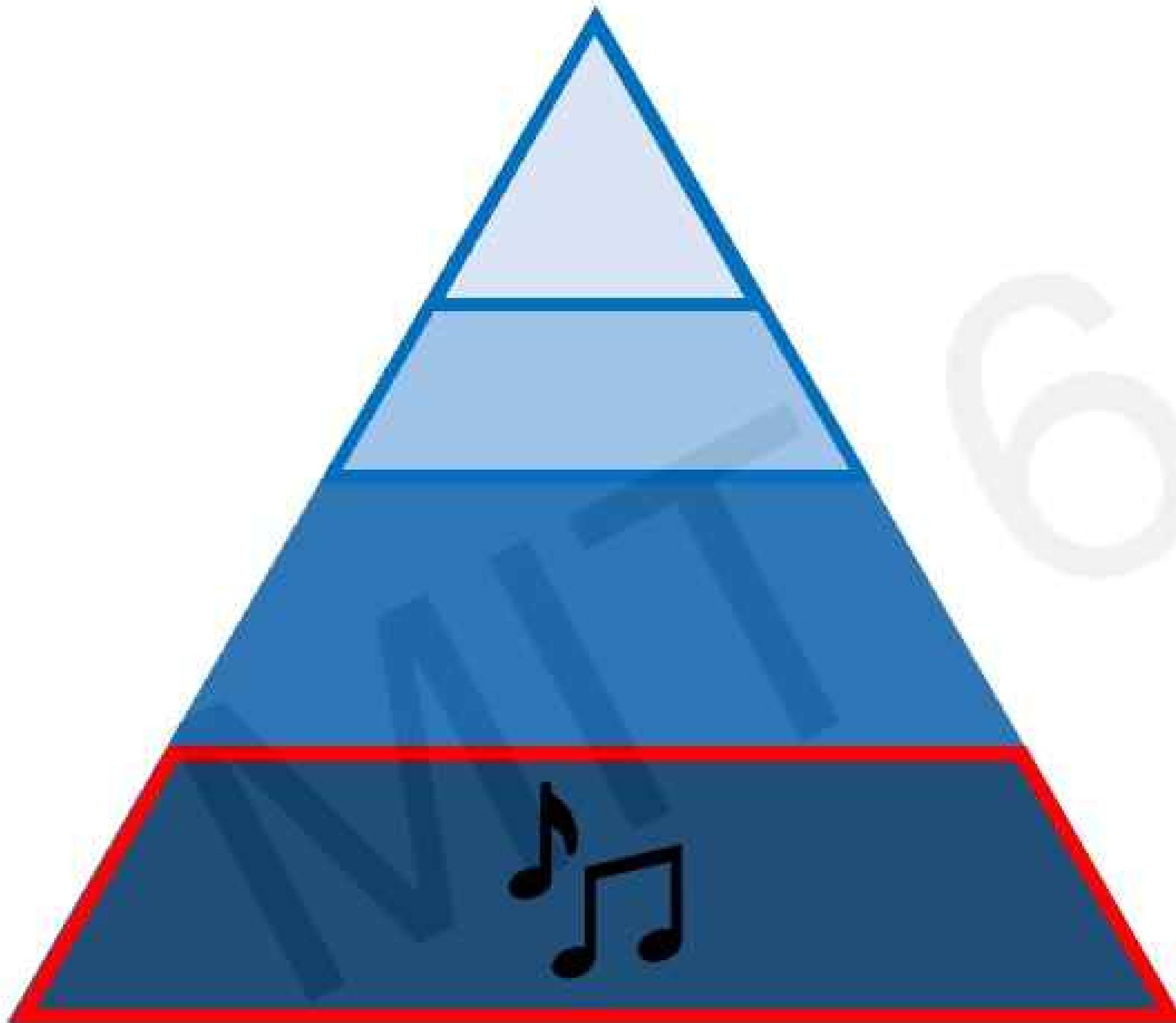
TensorFlow



PyTorch

Labs and Prizes

All due Thursday 1/8 at 11:59pm ET. Instructions: bit.ly/6s191-syllabus



Music Generation

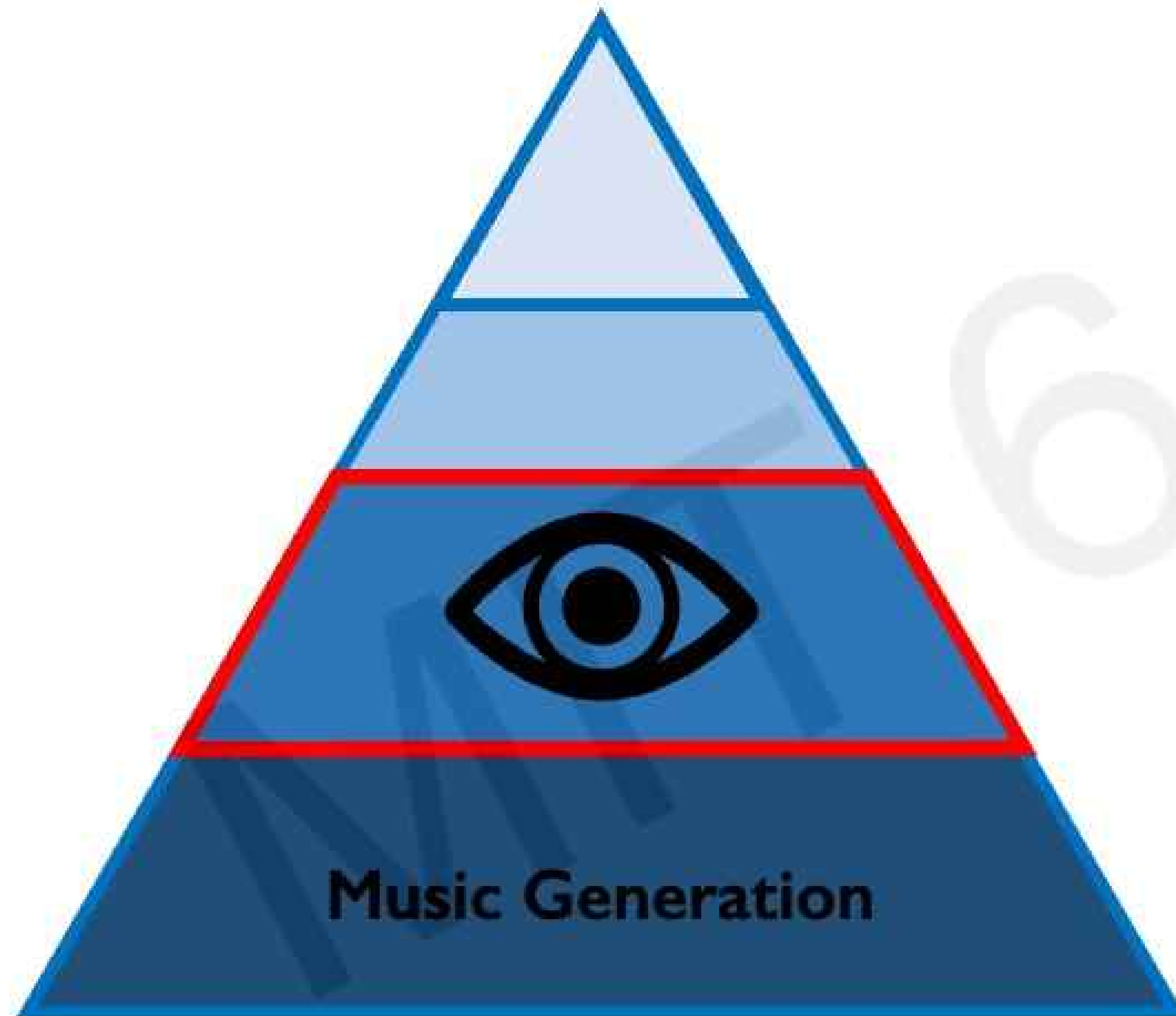
Build a neural network that can learn the genre of Irish folk songs and use it to generate brand new songs!

Prize:



Labs and Prizes

All due Thursday 1/8 at 11:59pm ET. Instructions: bit.ly/6s191-syllabus



Computer Vision

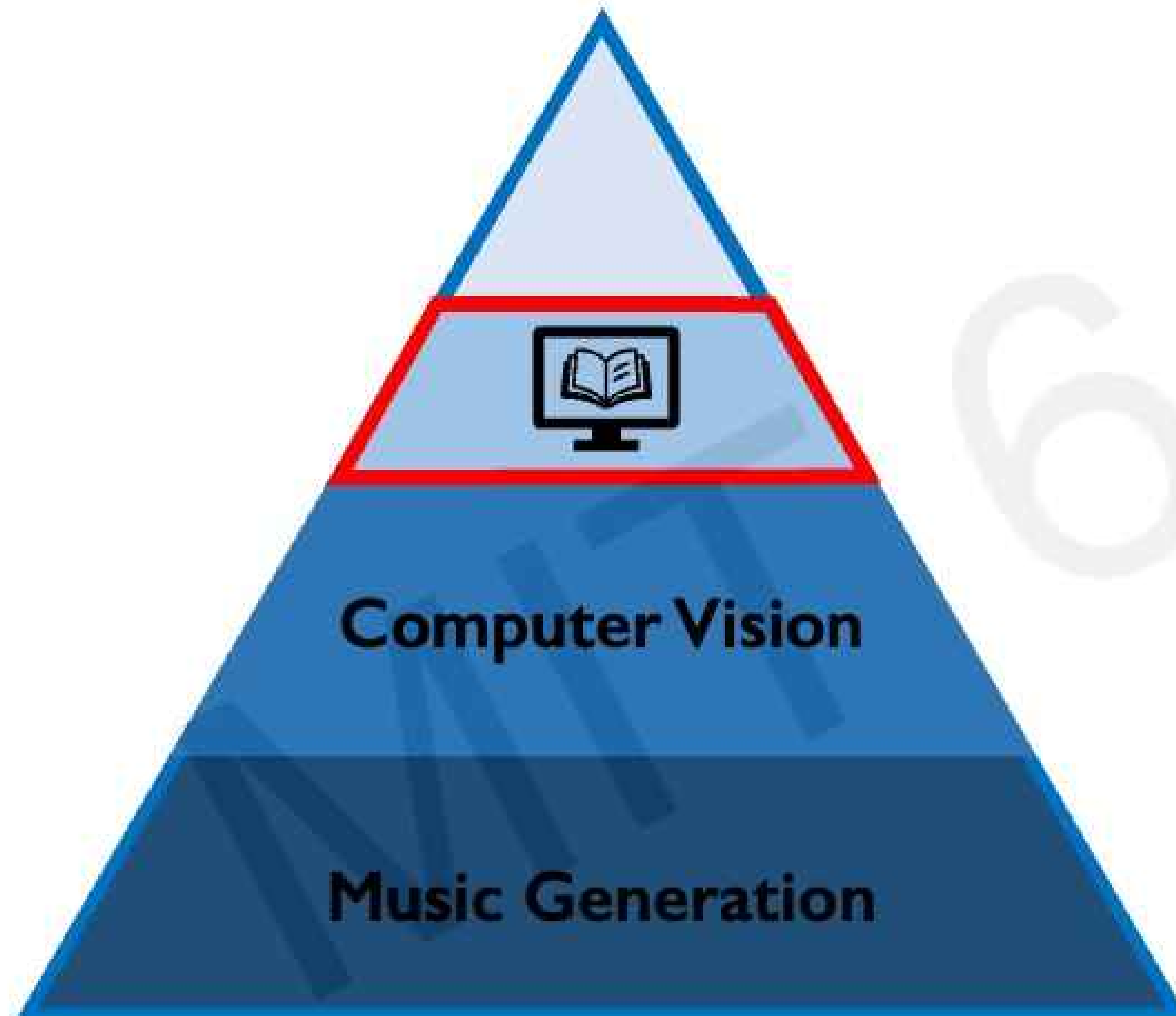
Build a neural network that can detect and mitigate biases in computer vision facial recognition systems!

Prize:



Labs and Prizes

All due Thursday 1/8 at 11:59pm ET. Instructions: bit.ly/6s191-syllabus



Large Language Models

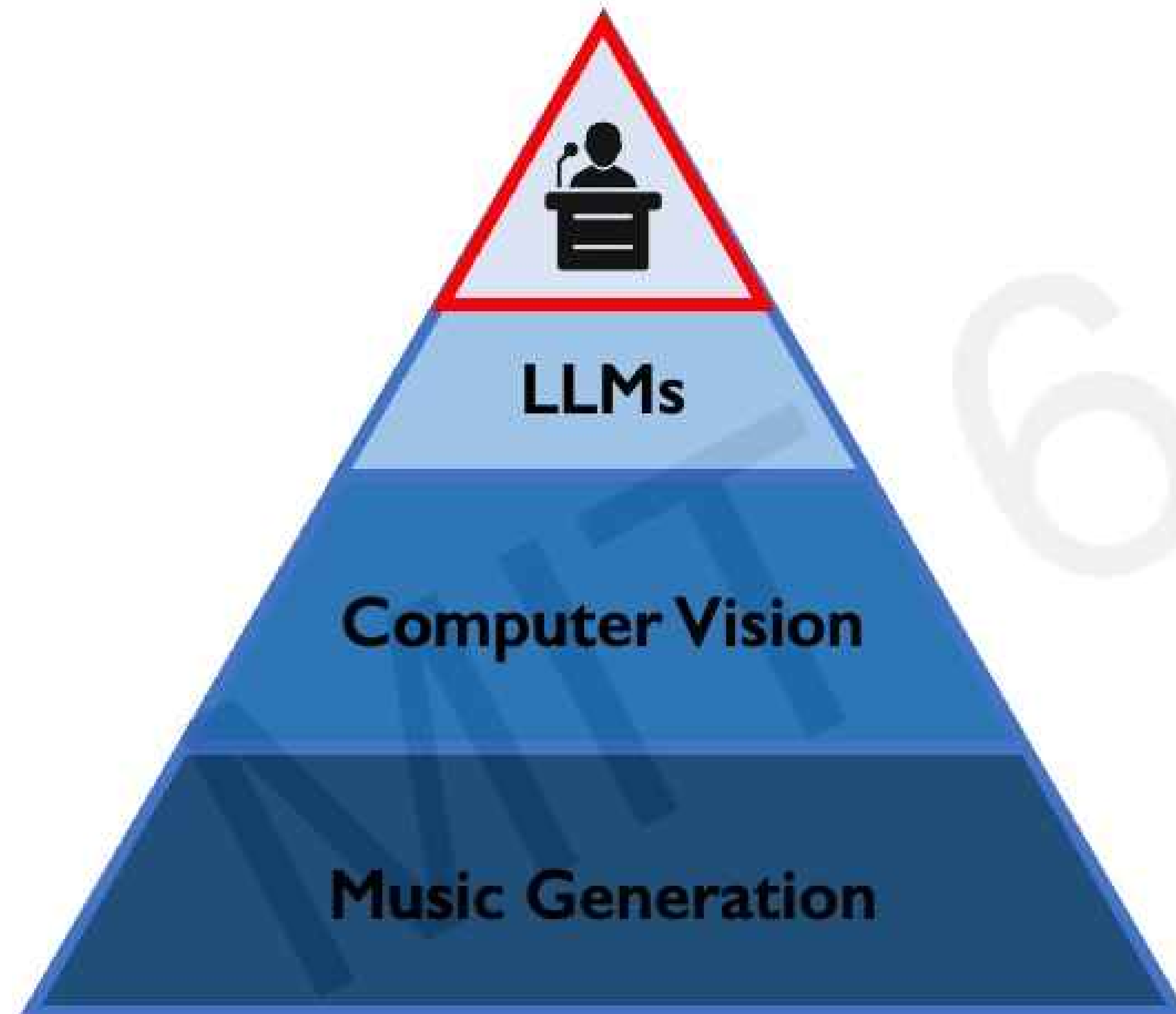
Finetune the Gemma large language model (LLM) in a mystery style and evaluate with an AI judge!

Prize:



Project Pitch Competition

Friday 1/9. Instructions: bit.ly/6s191-syllabus



Project Pitch Competition

Present a novel deep learning research idea or application (5 minutes, strict)

Presentations on **Friday, Jan 9**

Submit groups by **Wed 1/07 11:59pm ET**

Submit slides by **Thu 1/08 11:59pm ET**

Instructions: bit.ly/6s191-syllabus

Prizes:

Gold:

NVIDIA 5070 GPU



Silver:

Smartwatch



Bronze:

HD Monitor



Program Support

- All lectures will be held in person in 32-123
- Software labs + office hours in 32-123
- Piazza: piazza.com/mit/spring2026/6s191
 - Course announcements & asking questions
- Program Website: introtodeeplearning.com
 - Lecture schedule
 - Slides and lecture recordings
 - Software labs
- Syllabus: bit.ly/6s191-syllabus
- Labs: github.com/MITDeepLearning/introtodeeplearning
- Email us: introtodeeplearning-staff@mit.edu



Program Staff



Alexander Amini
Lead Instructor



Ava Amini
Lead Instructor



Anisha Parsan
Lead TA



Shrika Eddula
Lead TA

Program TAs & EECS Sponsor



Victory



Daniela Rus
Director of CSAIL



Benjamin



Vanessa



Jeannie

introtodeeplearning-staff@mit.edu

Thanks to Sponsors!



Google



Liquid



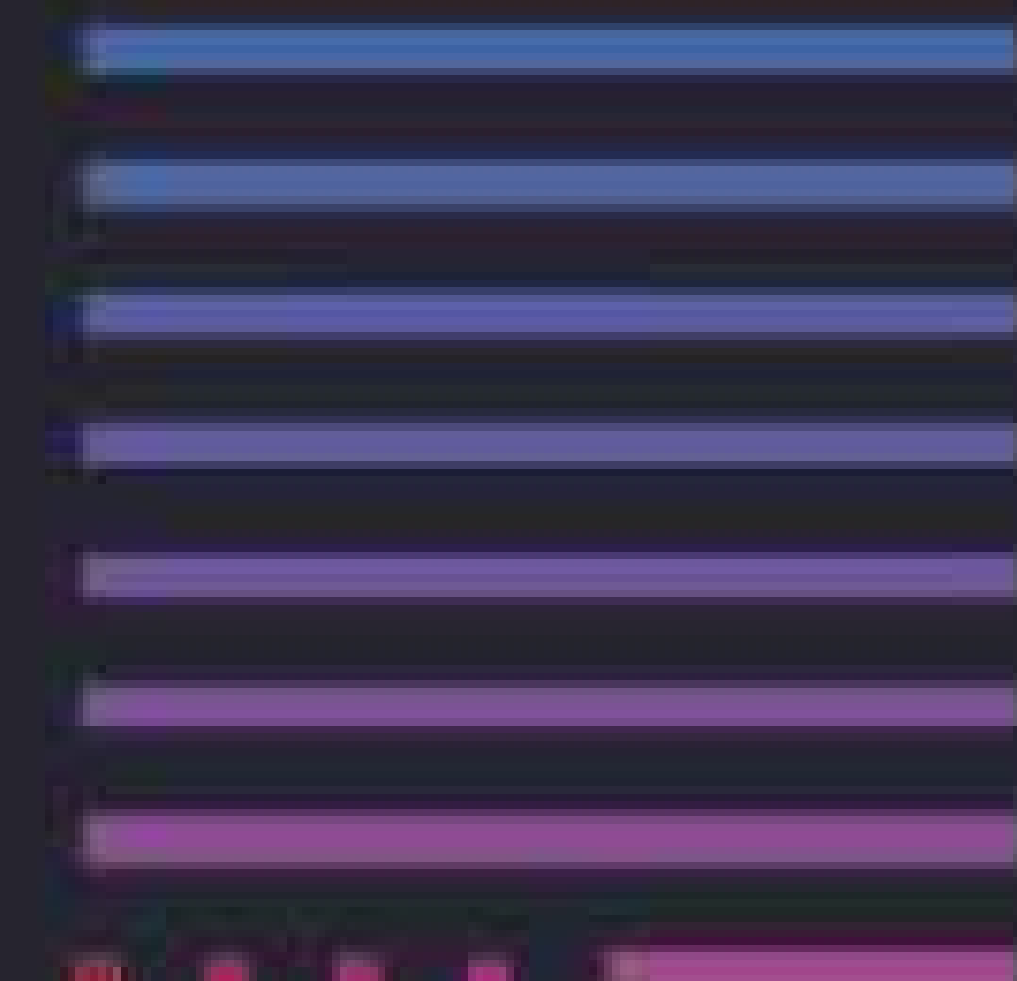
Microsoft



comet



MIT



MIT-IBM
Watson
AI Lab

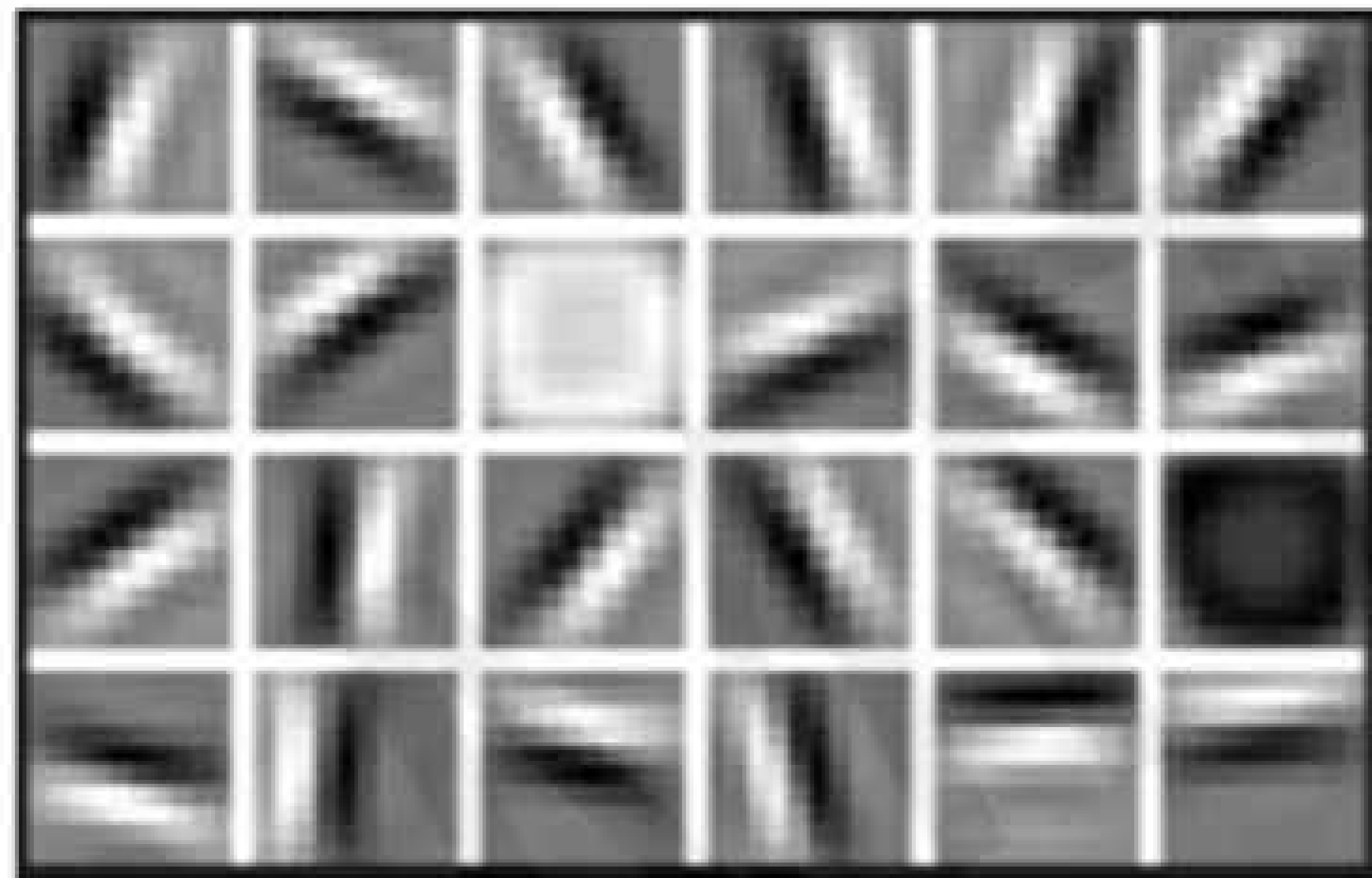
Why Deep Learning and Why Now?

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

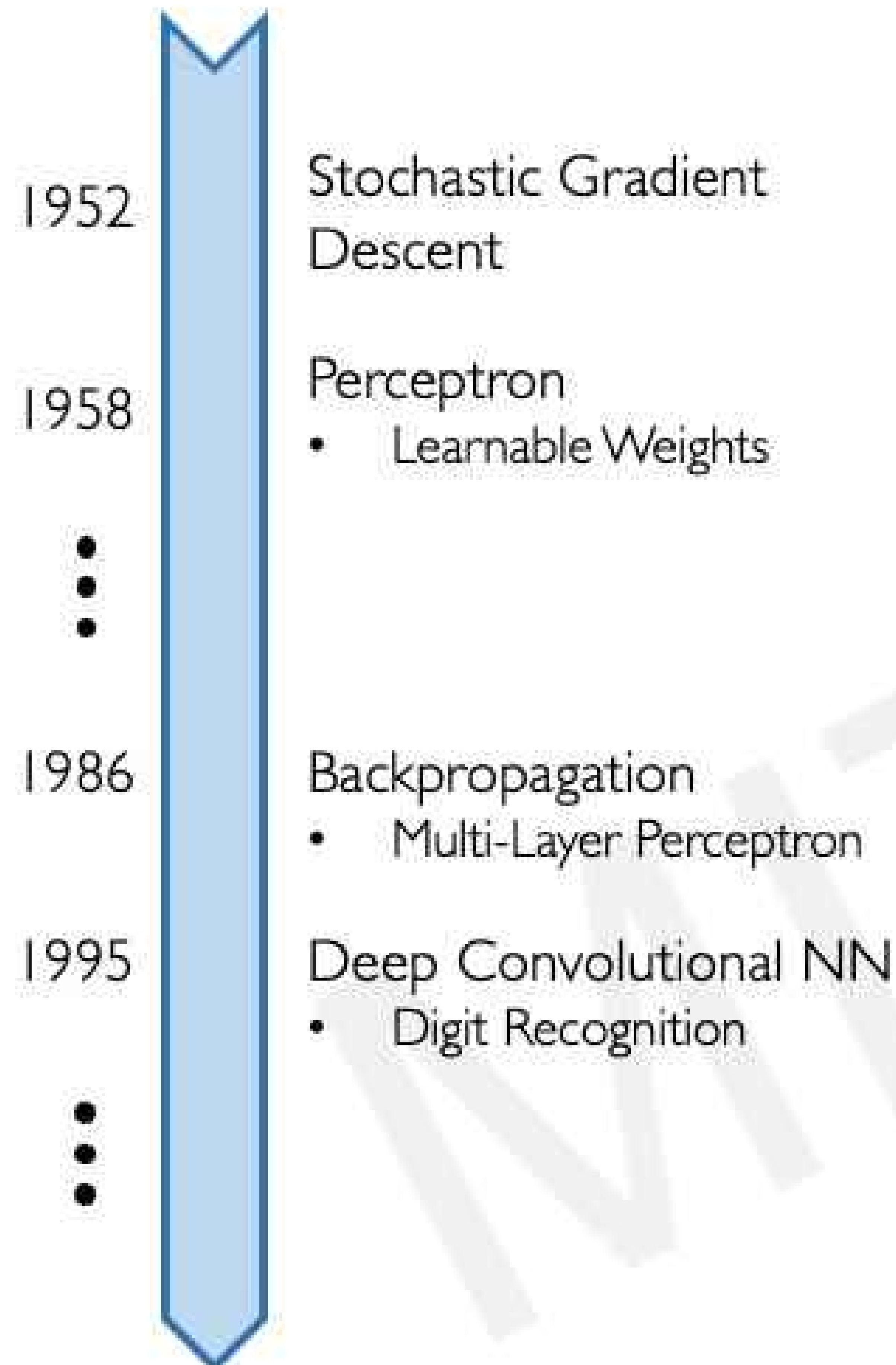
High Level Features



Facial Structure

Why Now?

Neural Networks date back decades, so why the dominance?



1. Big Data

- Larger Datasets
- Easier Collection & Storage

IMAGENET



WIKIPEDIA
The Free Encyclopedia



2. Hardware

- Graphics Processing Units (GPUs)
- Massively Parallelizable



3. Software

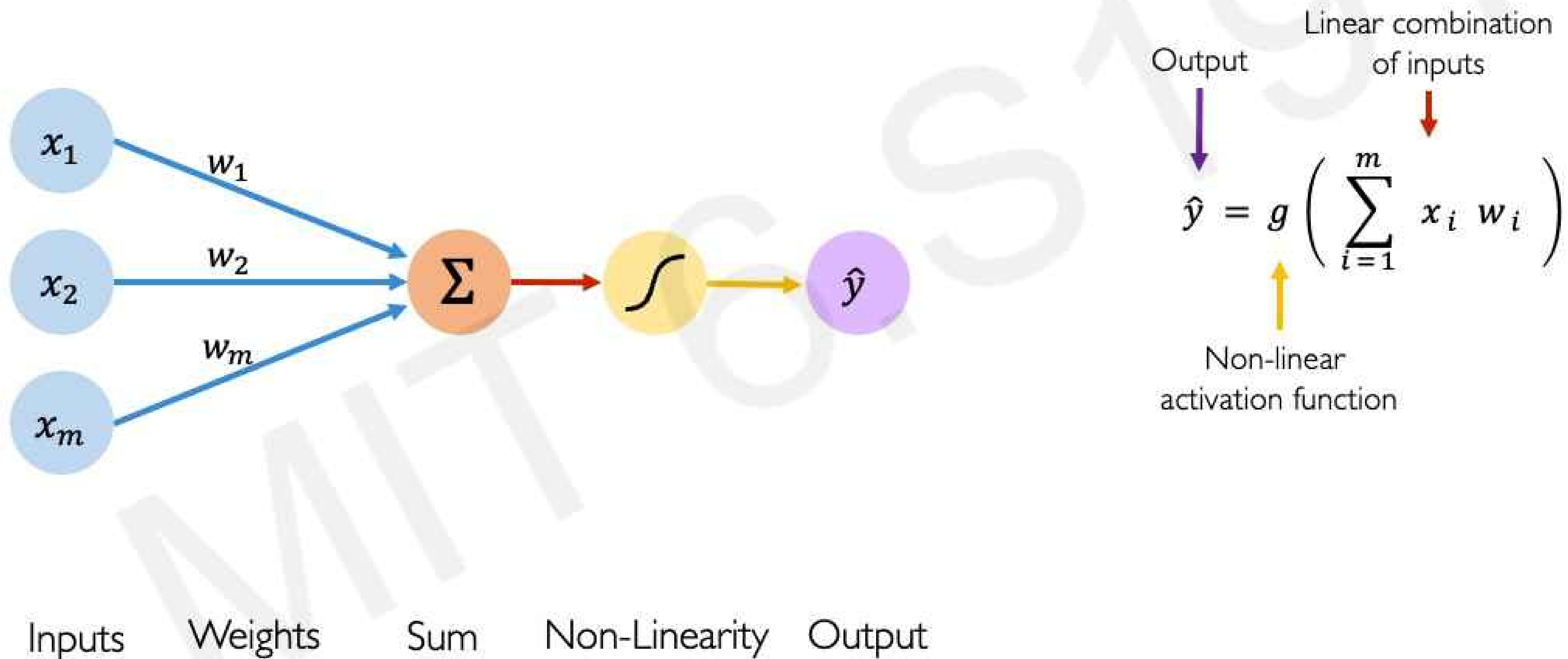
- Improved Techniques
- New Models
- Toolboxes



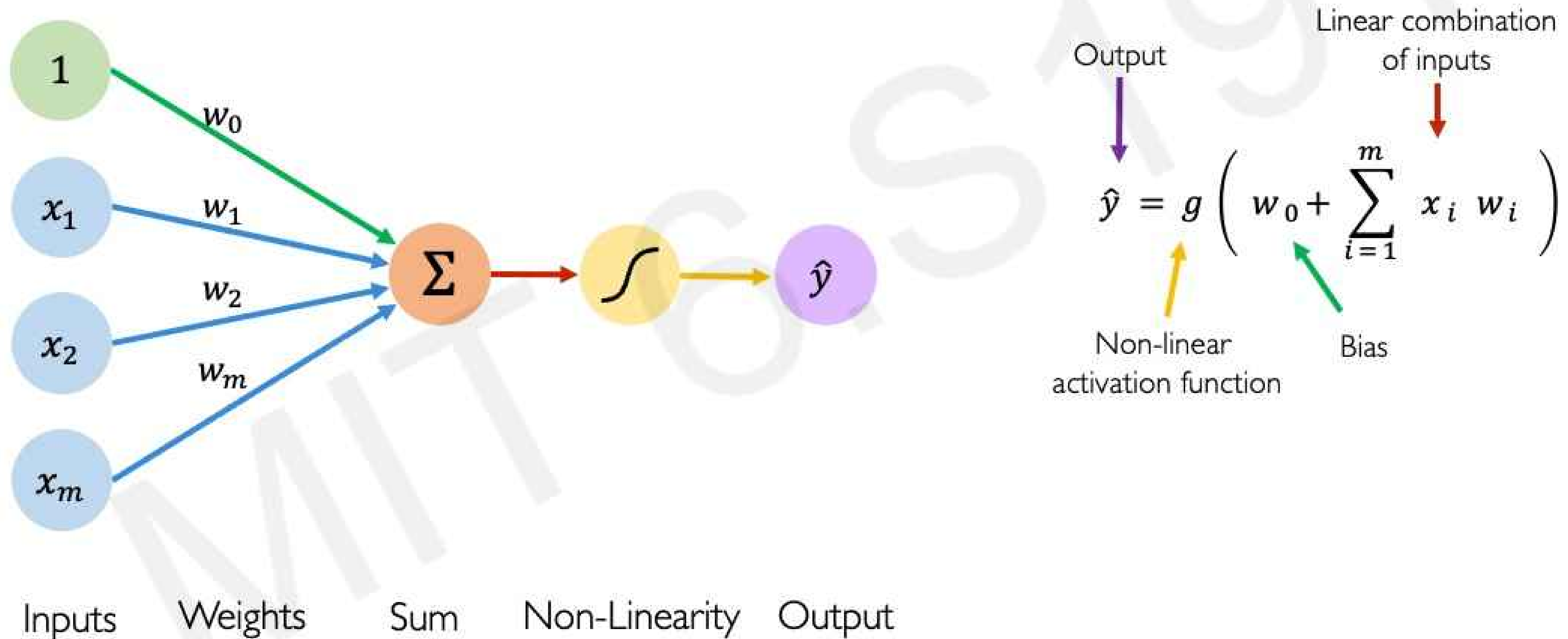
The Perceptron

The structural building block of deep learning

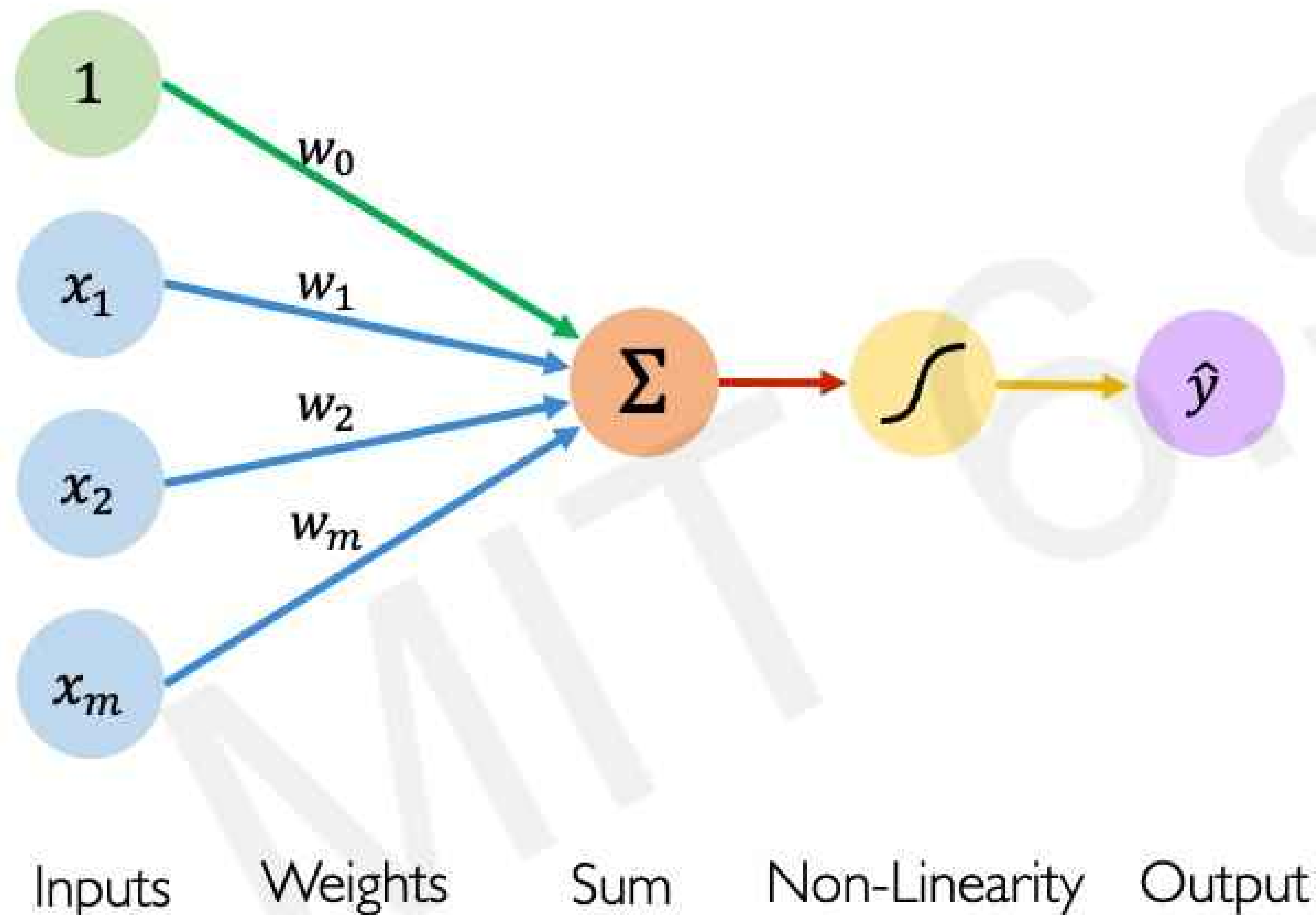
The Perceptron: Forward Propagation



The Perceptron: Forward Propagation



The Perceptron: Forward Propagation

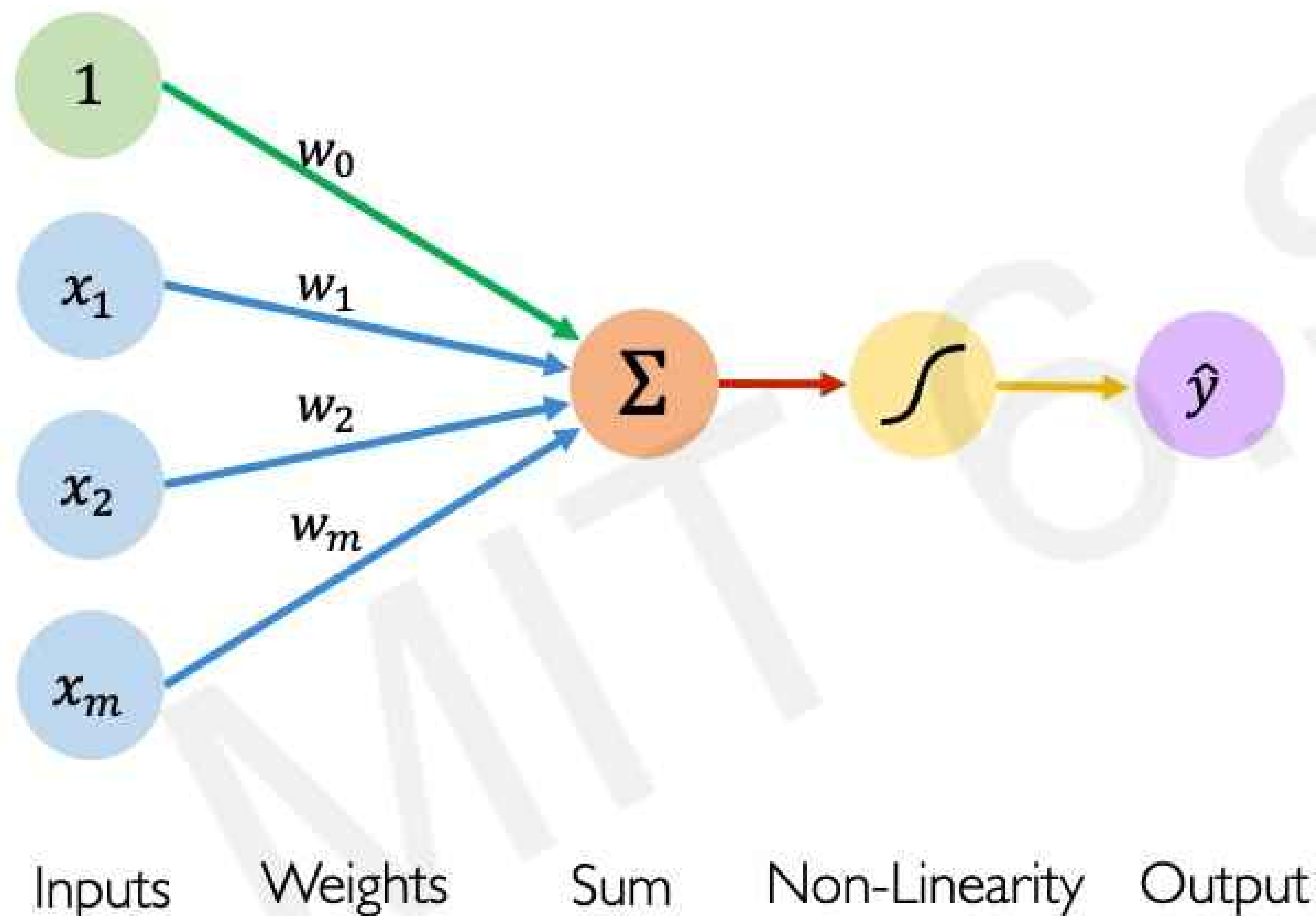


$$\hat{y} = g \left(w_0 + \sum_{i=1}^m x_i w_i \right)$$

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

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

The Perceptron: Forward Propagation

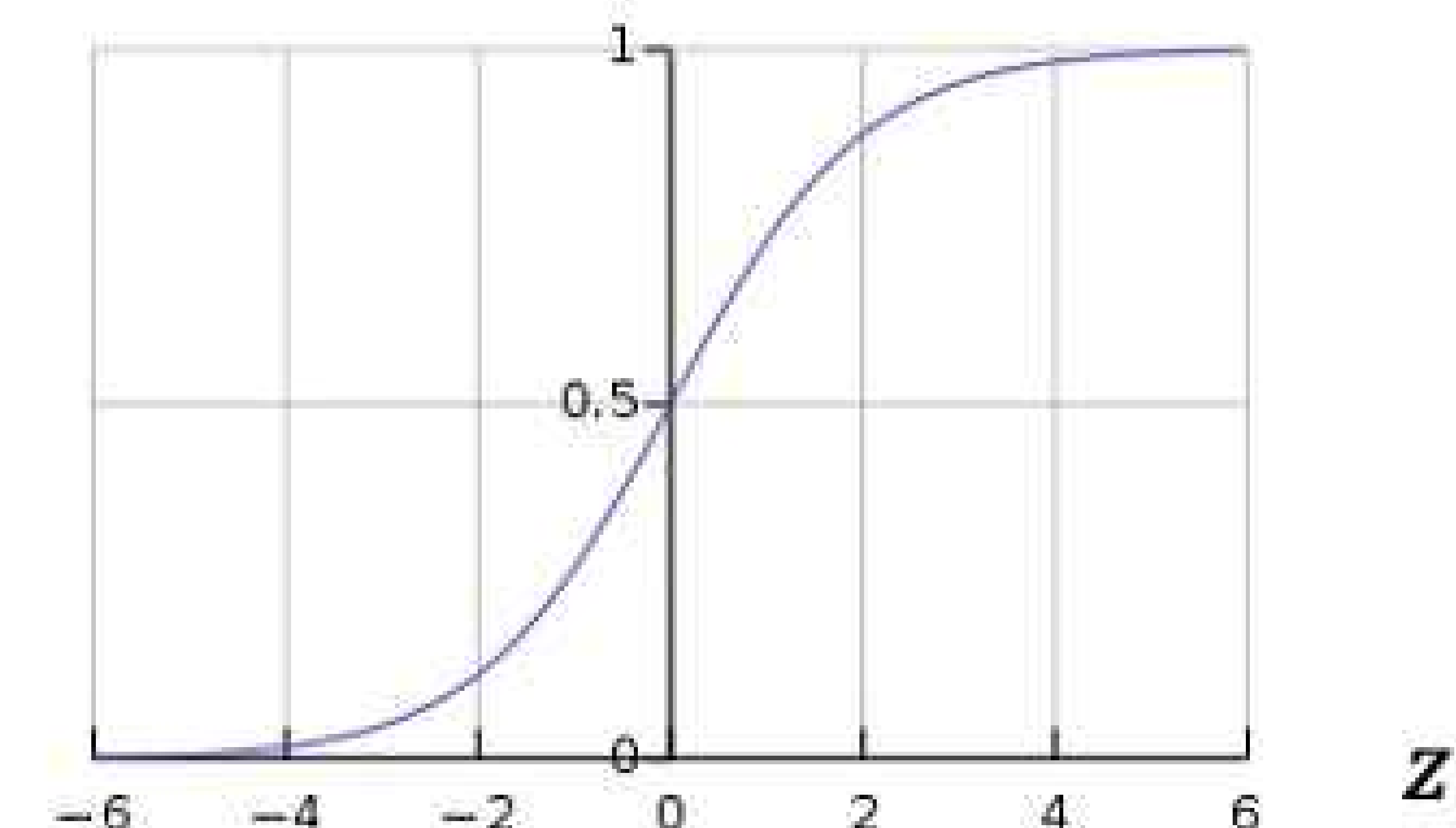


Activation Functions

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

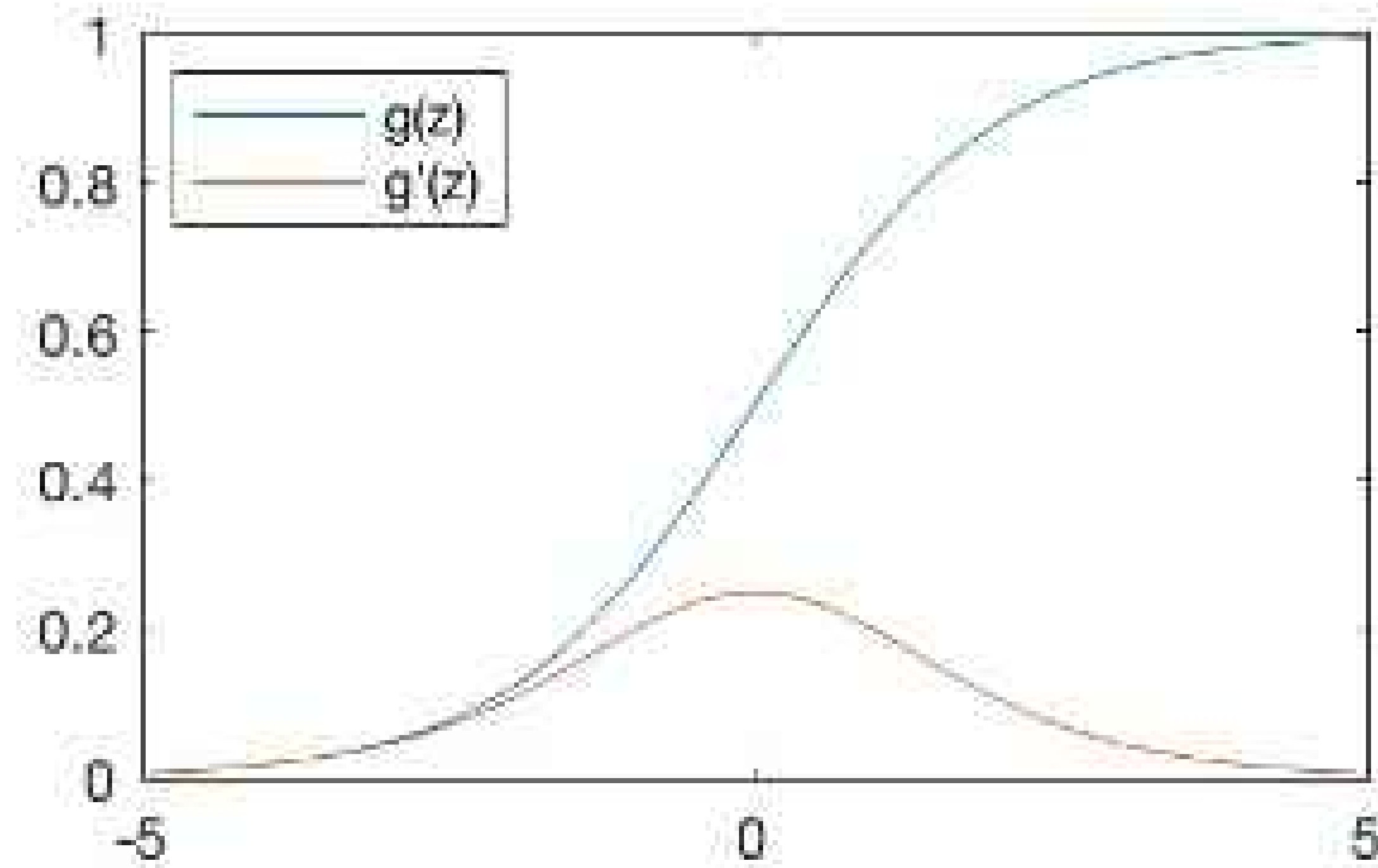
- Example: sigmoid function

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



Common Activation Functions

Sigmoid Function



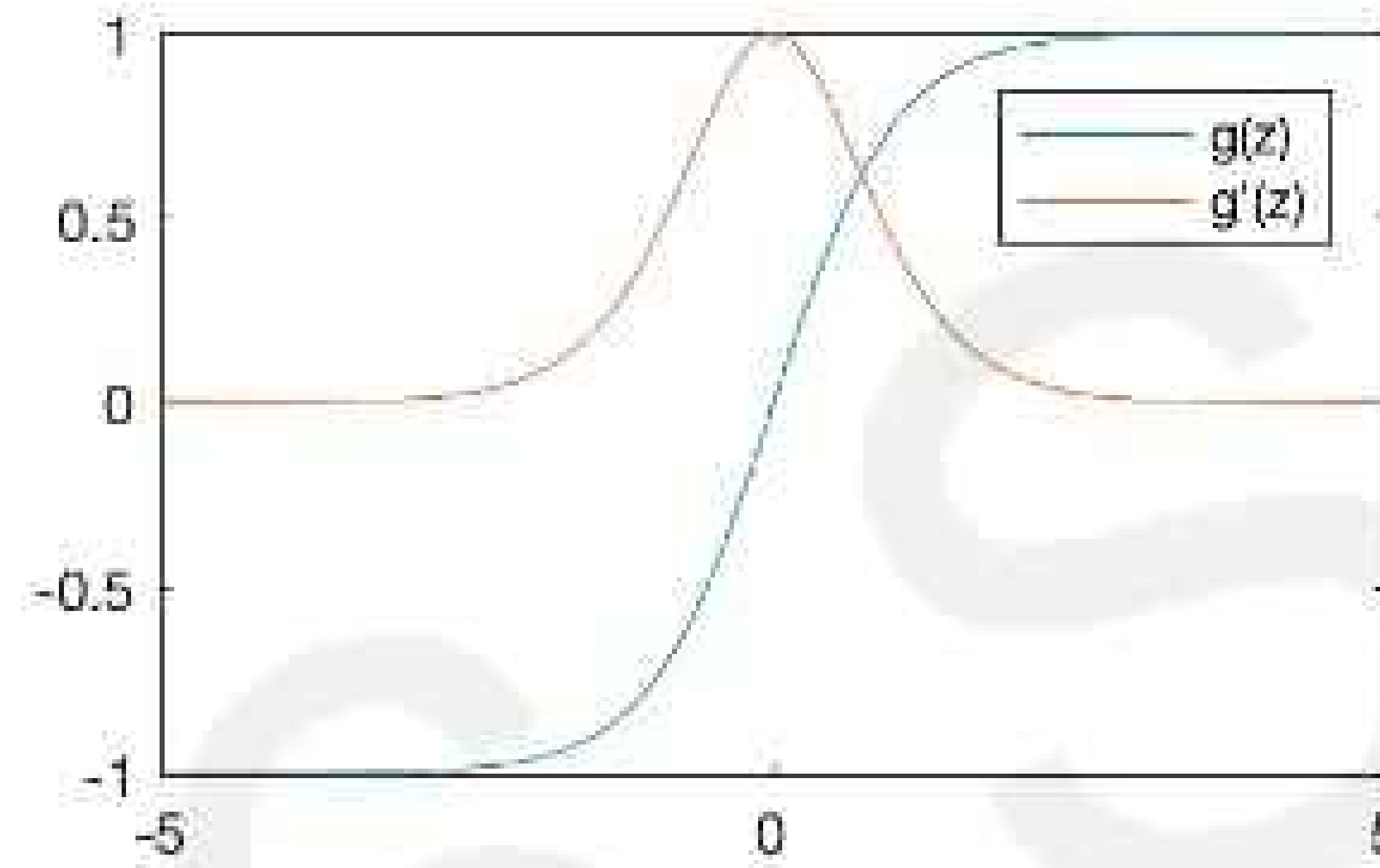
$$g(z) = \frac{1}{1 + e^{-z}}$$

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

 `tf.math.sigmoid(z)`

 `torch.sigmoid(z)`

Hyperbolic Tangent



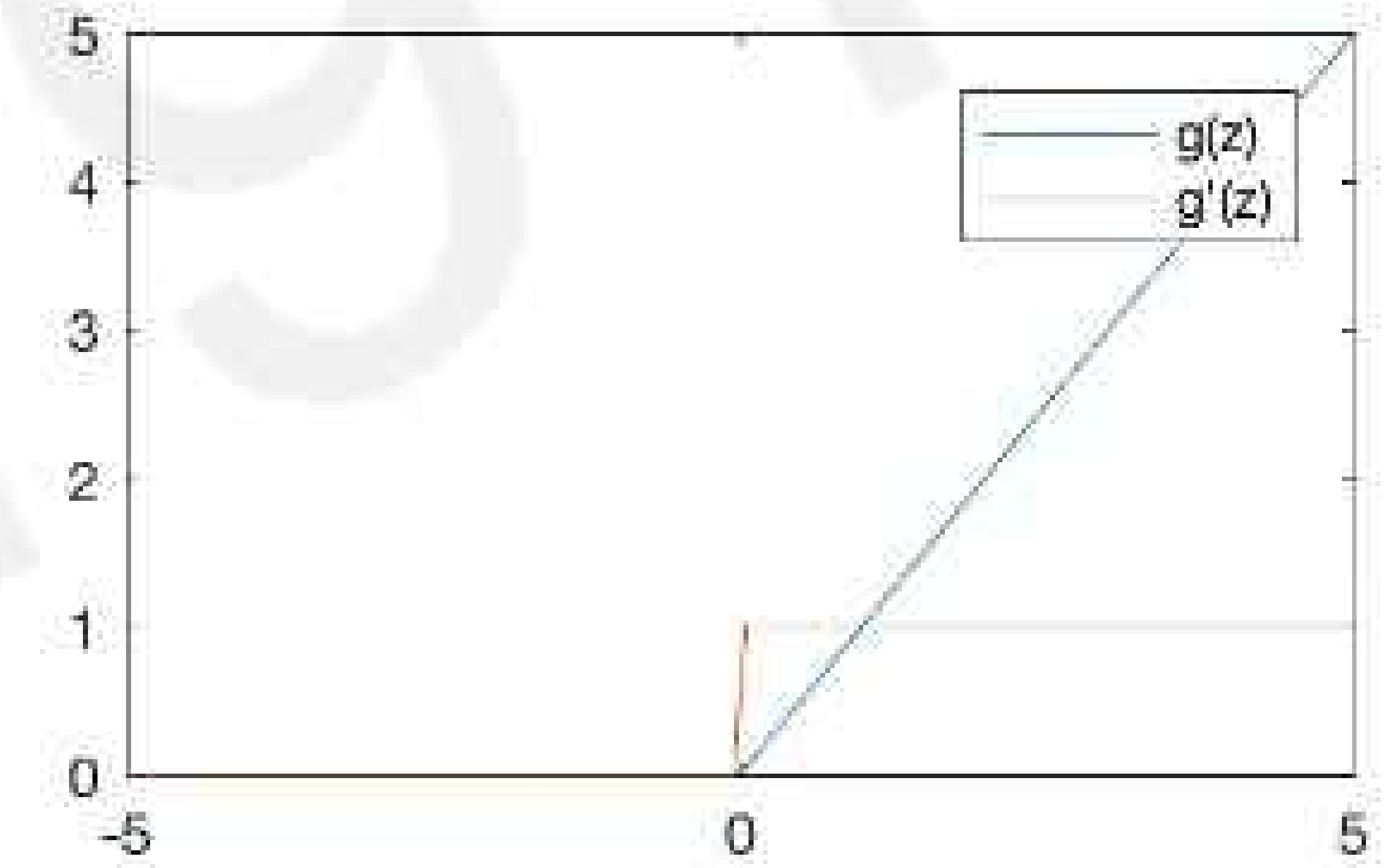
$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

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

 `tf.math.tanh(z)`

 `torch.tanh(z)`

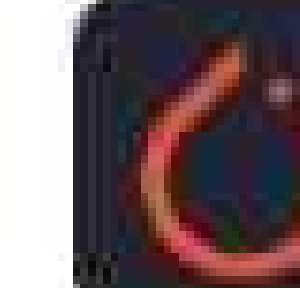
Rectified Linear Unit (ReLU)



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

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

 `tf.nn.relu(z)`

 `torch.nn.ReLU(z)`



TensorFlow code blocks

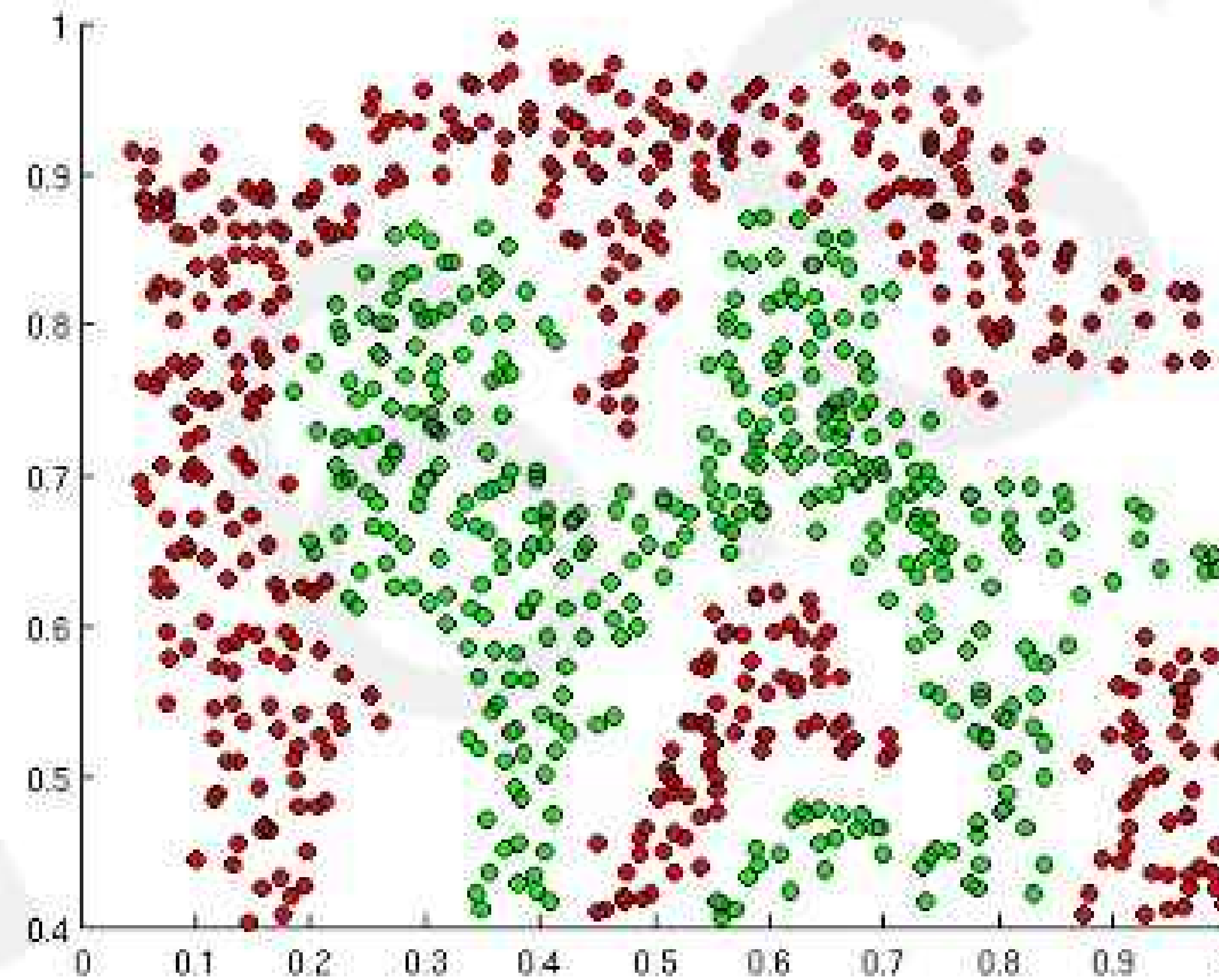
NOTE: All activation functions are non-linear

PyTorch code blocks



Importance of Activation Functions

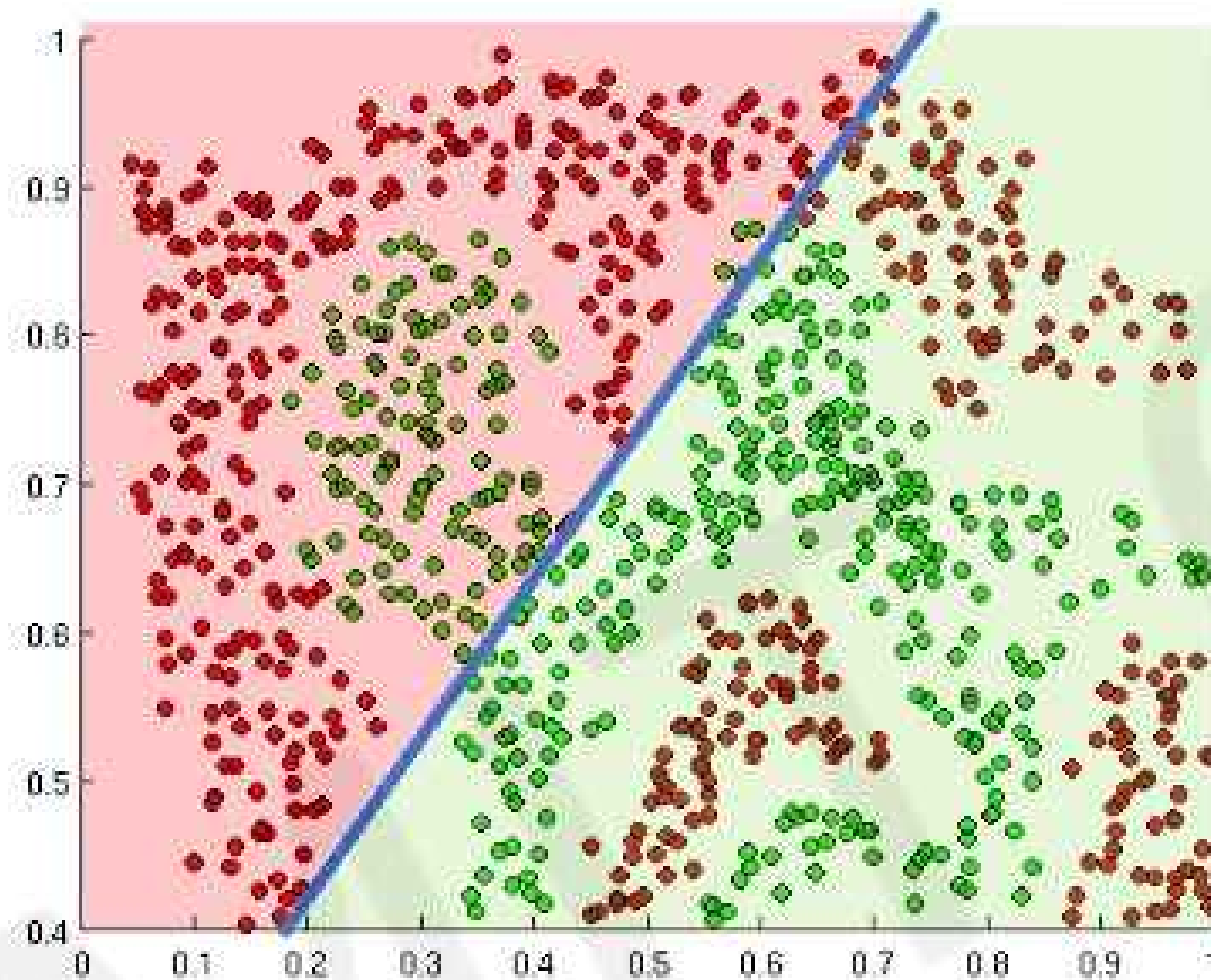
The purpose of activation functions is to *introduce non-linearities* into the network



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

Importance of Activation Functions

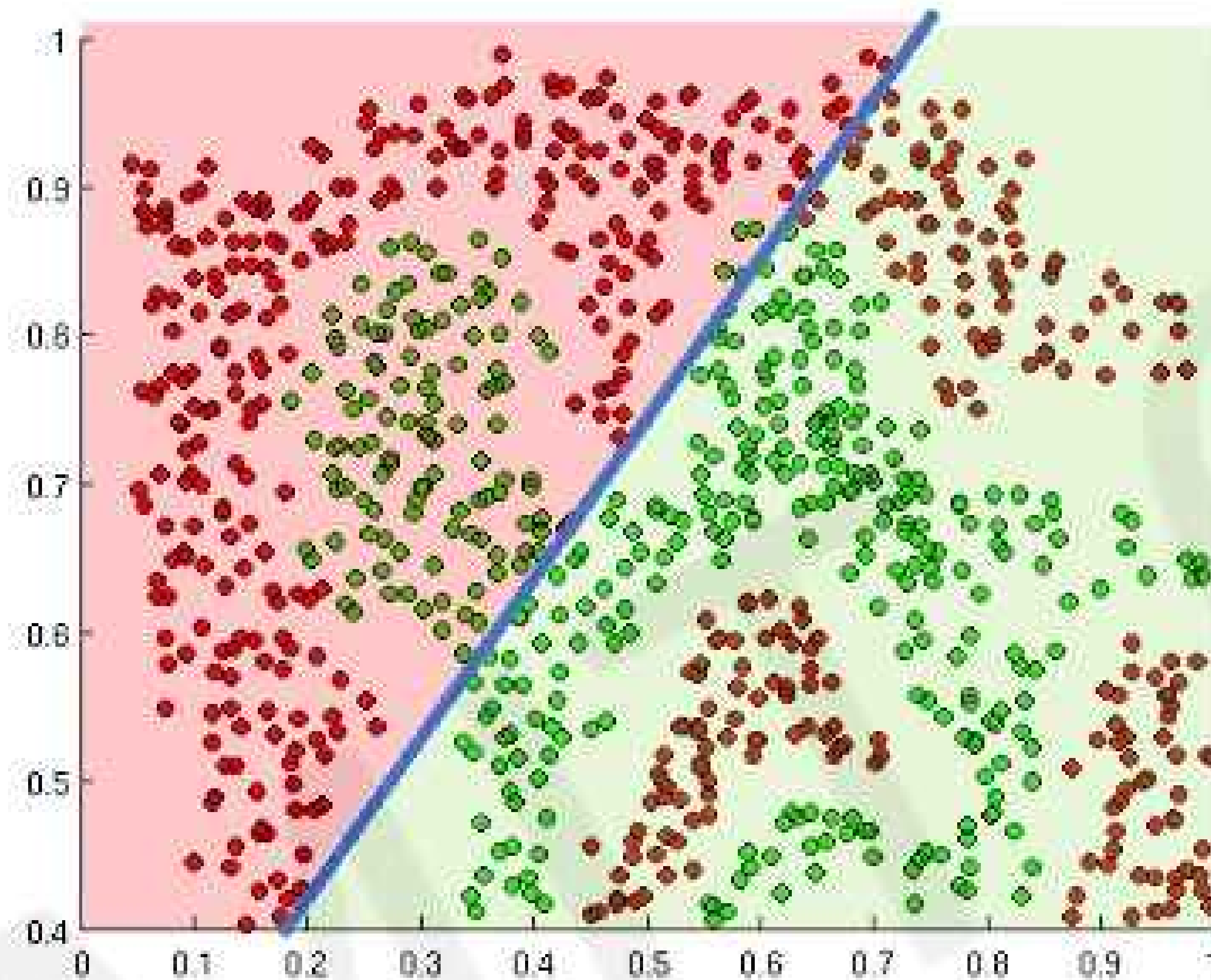
The purpose of activation functions is to *introduce non-linearities* into the network



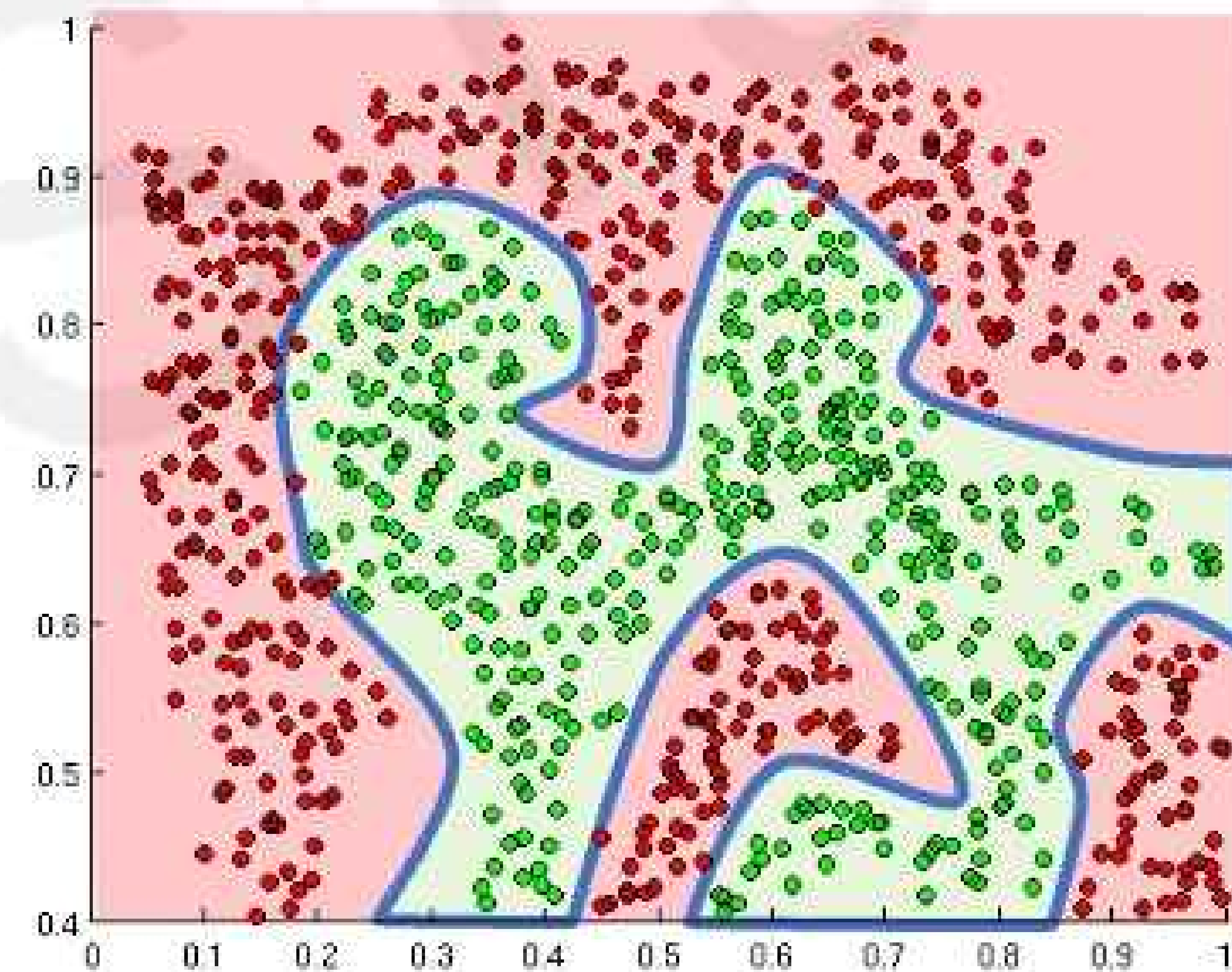
Linear activation functions produce linear decisions no matter the network size

Importance of Activation Functions

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

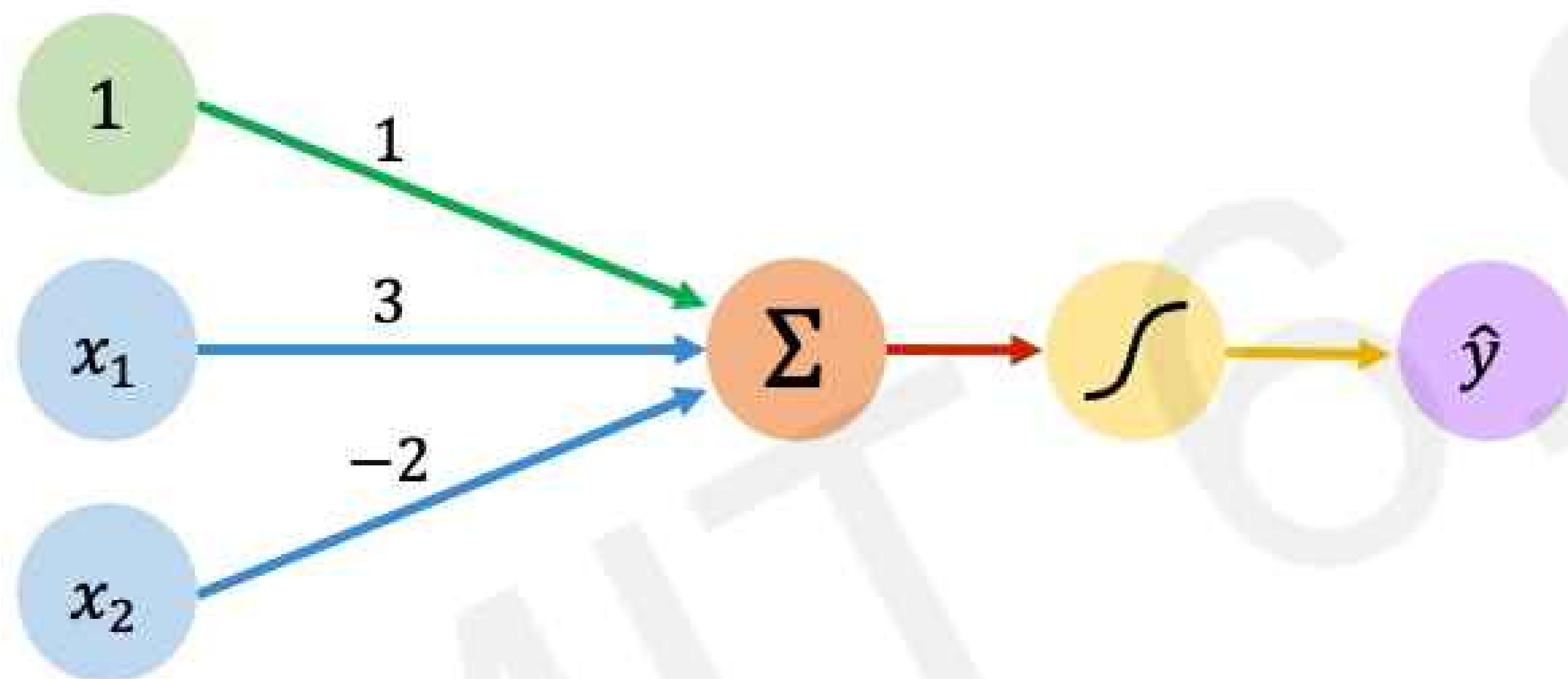


Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example

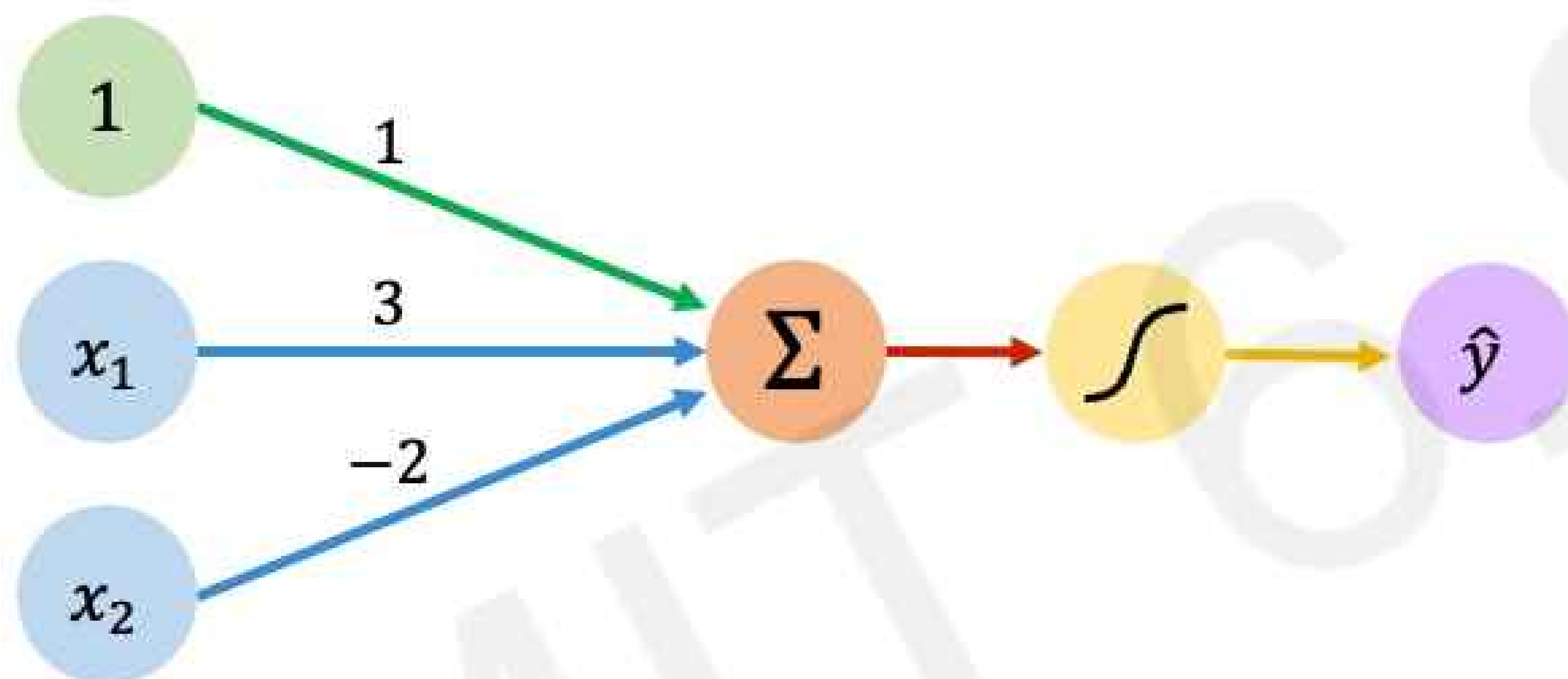


We have: $w_0 = 1$ and $\mathbf{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

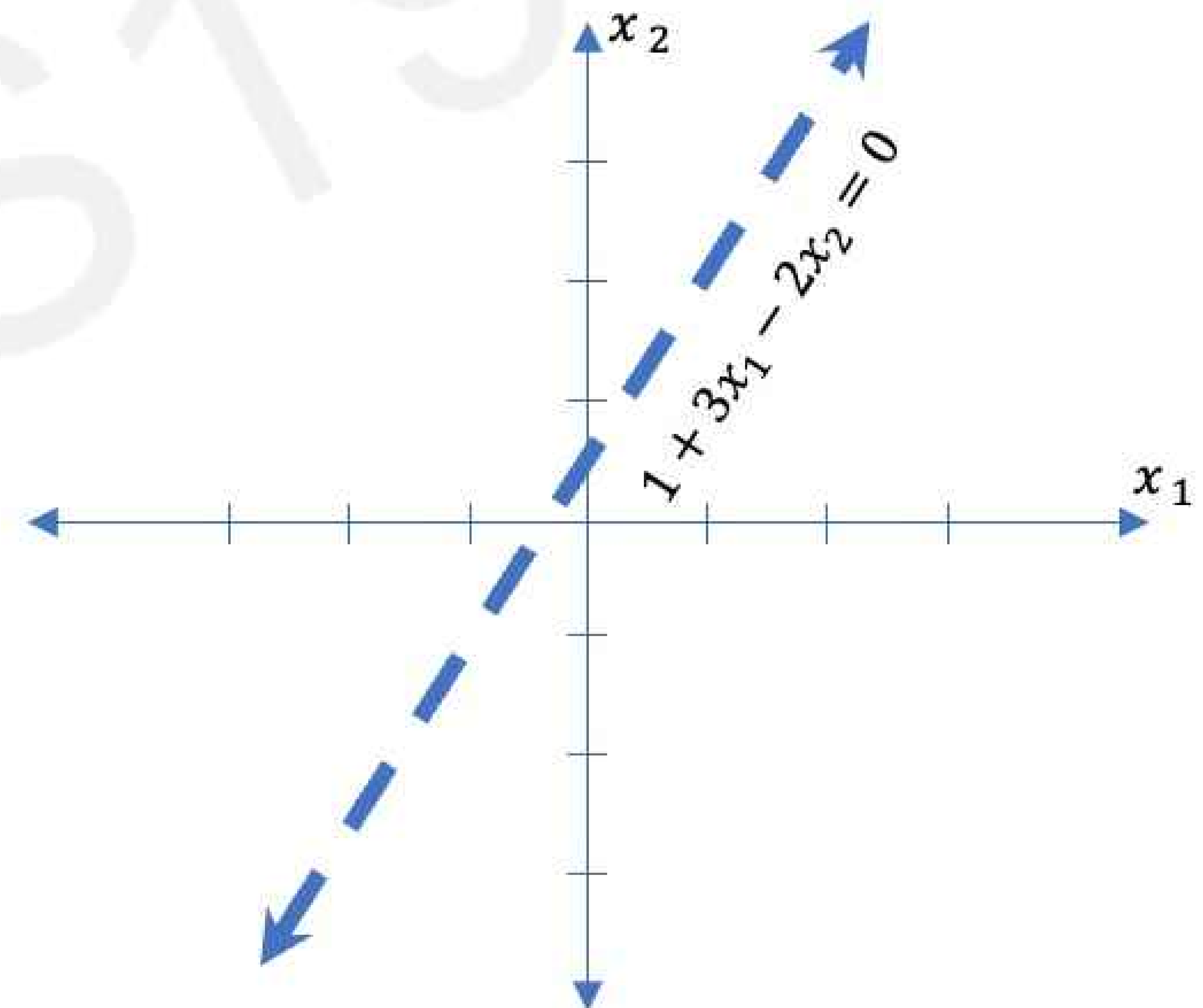
$$\begin{aligned}\hat{y} &= g(w_0 + \mathbf{X}^T \mathbf{W}) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g(1 + 3x_1 - 2x_2)\end{aligned}$$

This is just a line in 2D!

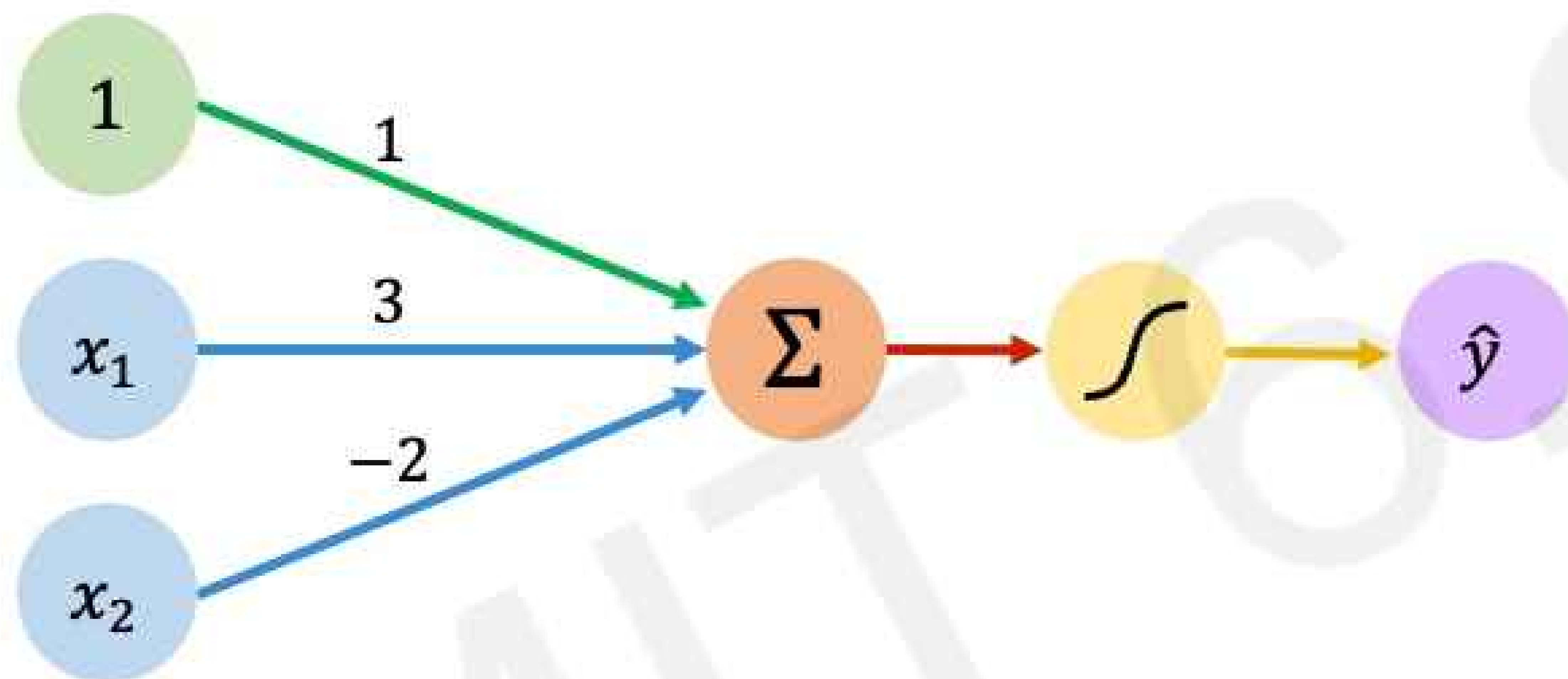
The Perceptron: Example



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

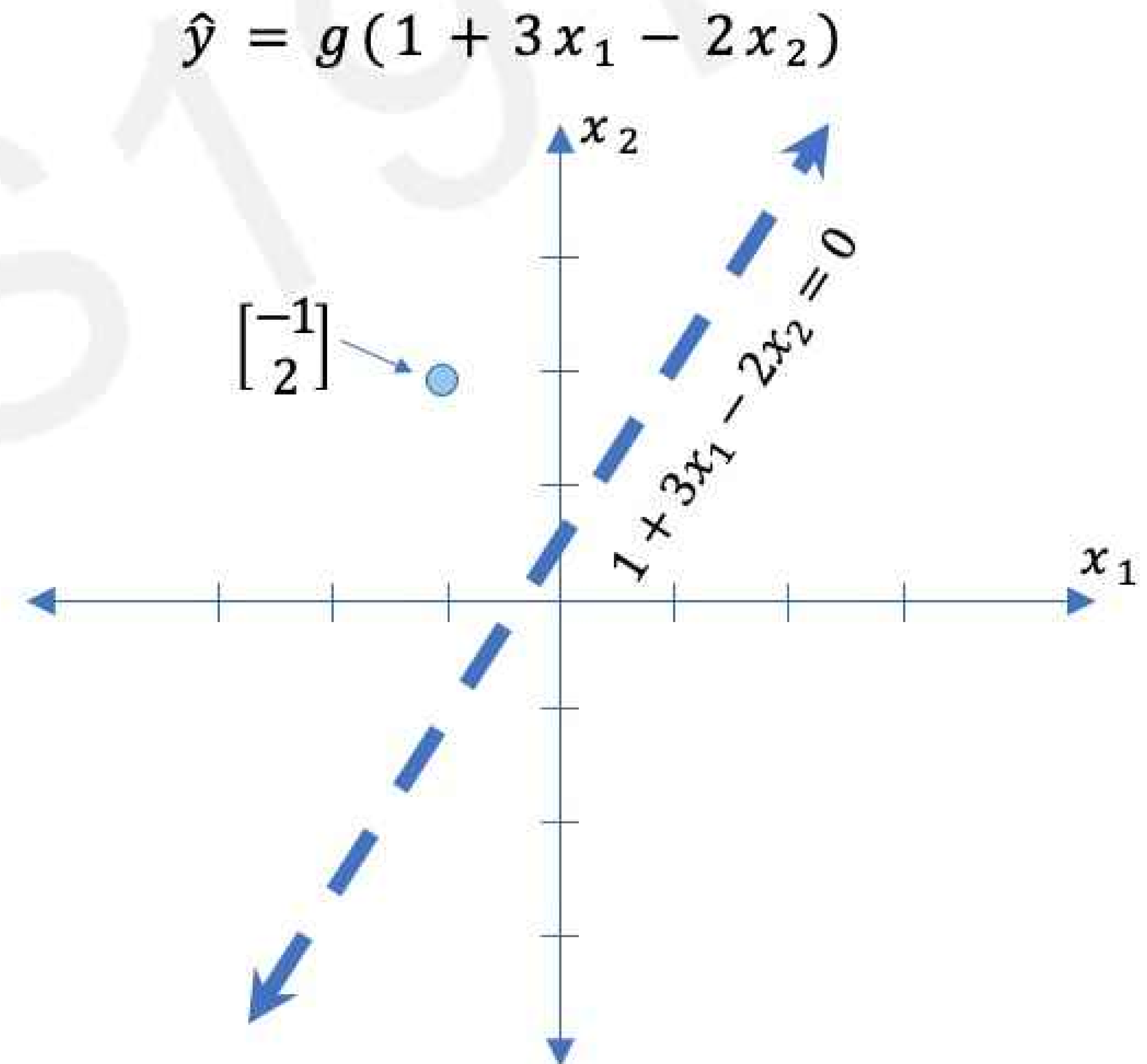


The Perceptron: Example

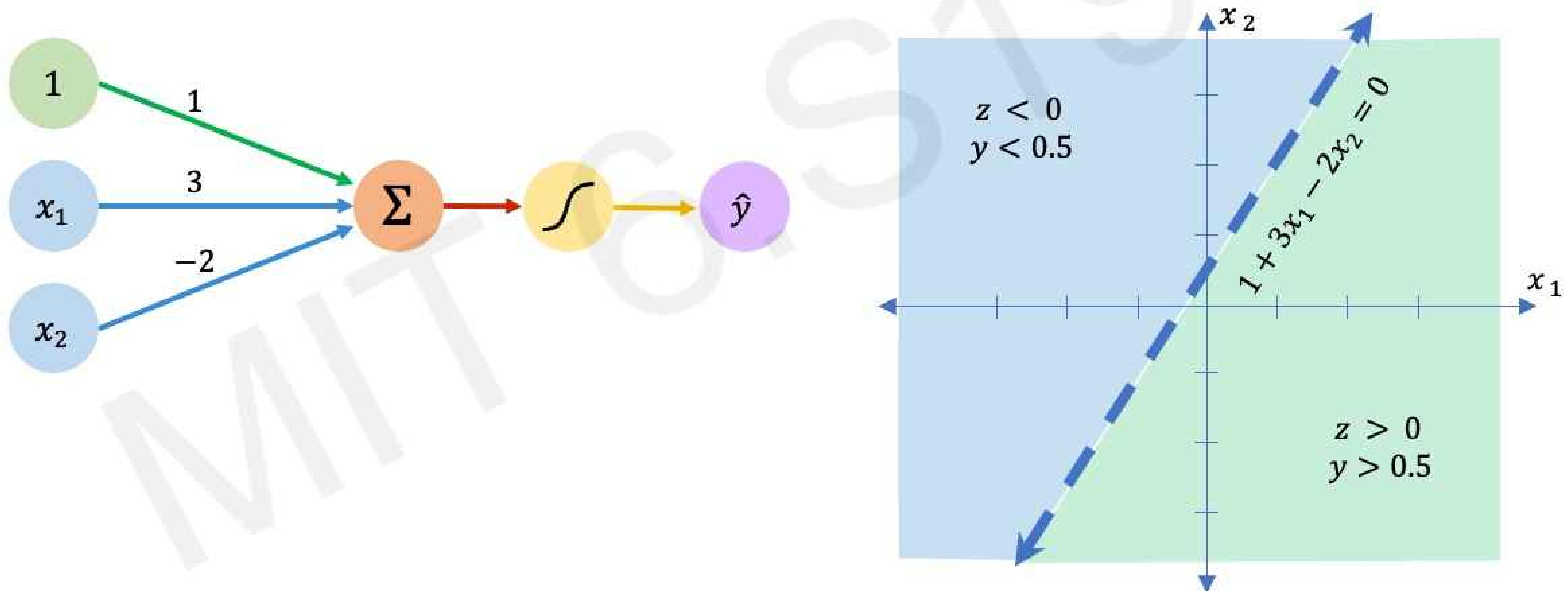


Assume we have input: $\mathbf{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

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



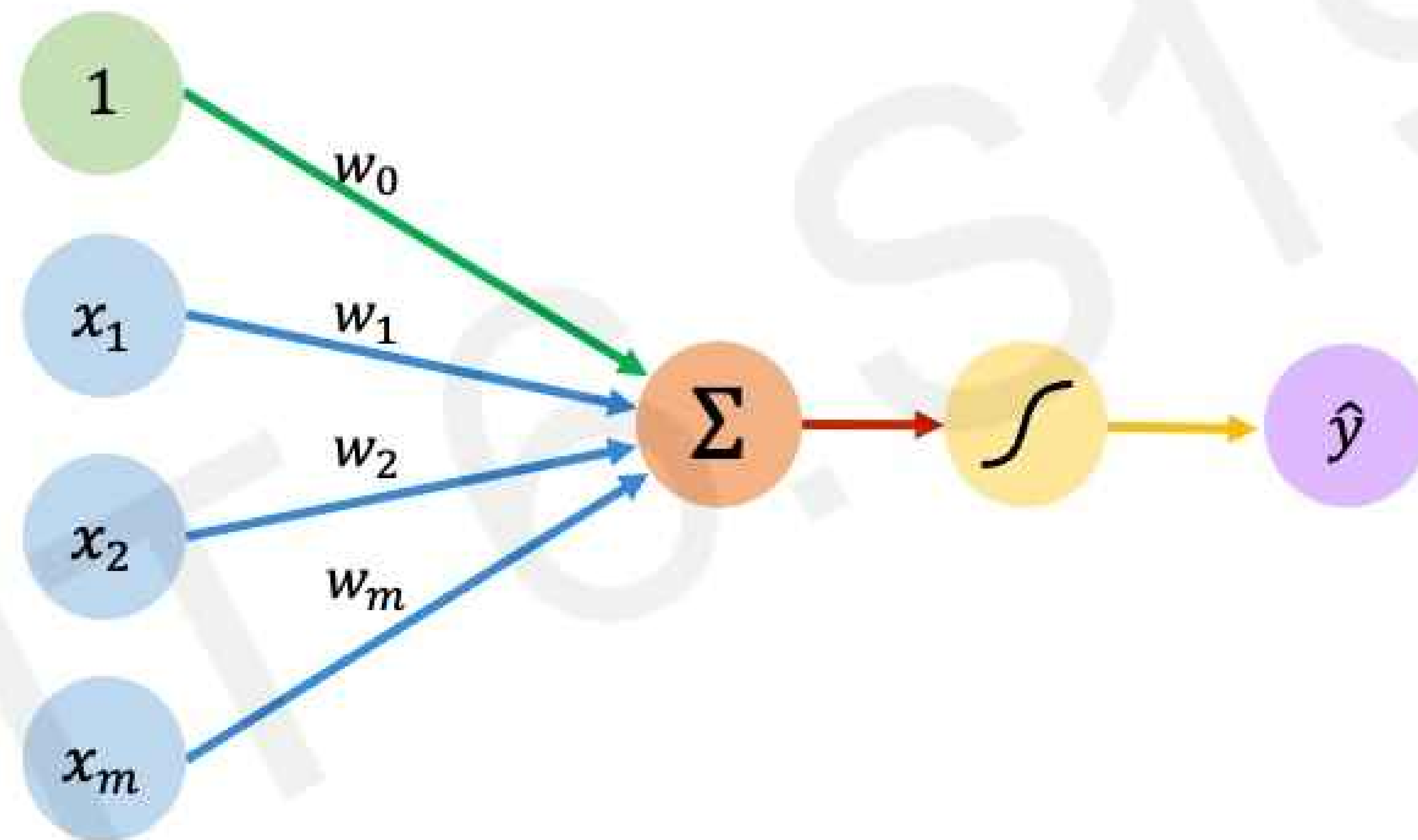
The Perceptron: Example



Building Neural Networks with Perceptrons

The Perceptron: Simplified

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



Inputs

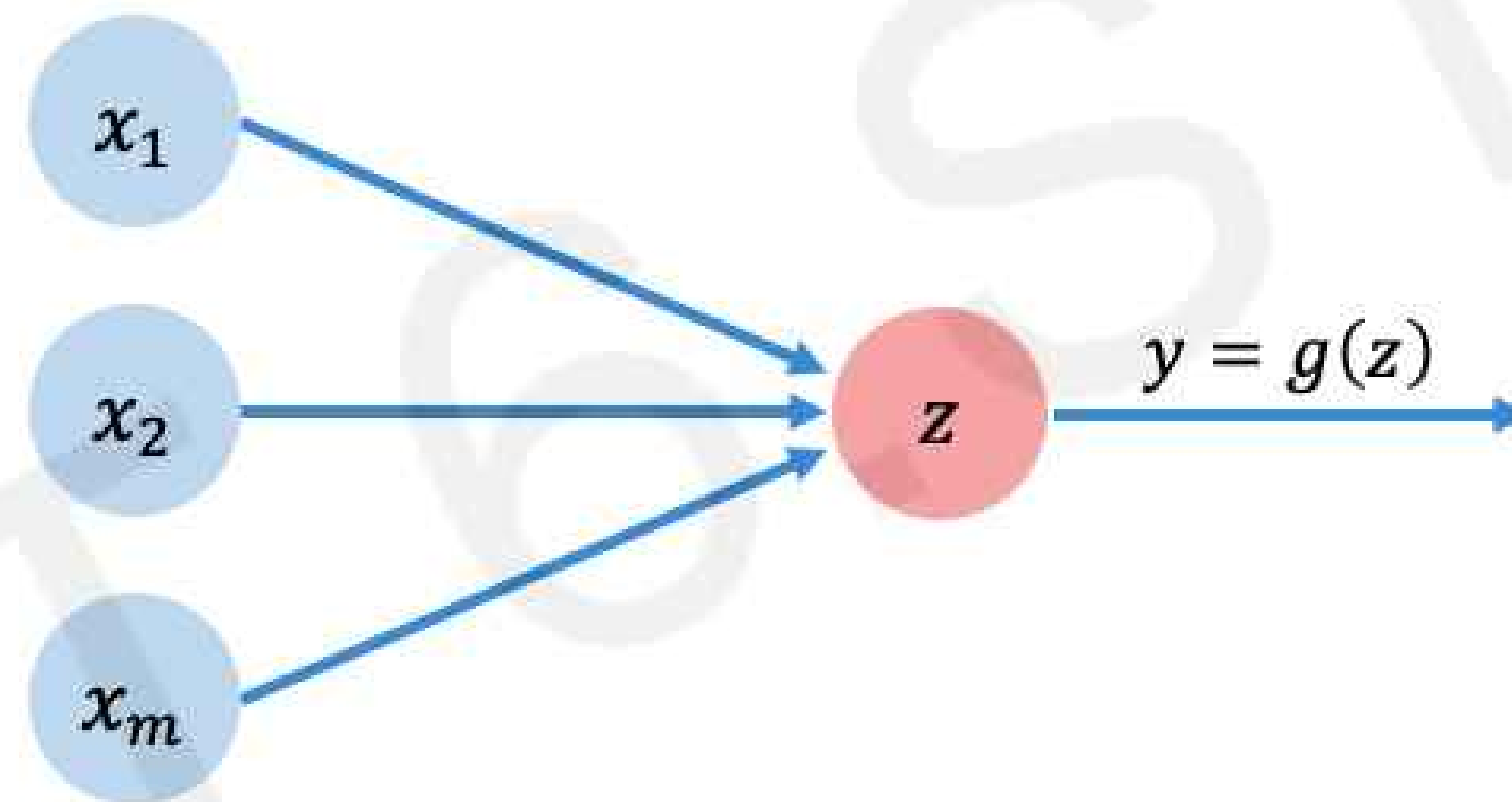
Weights

Sum

Non-Linearity

Output

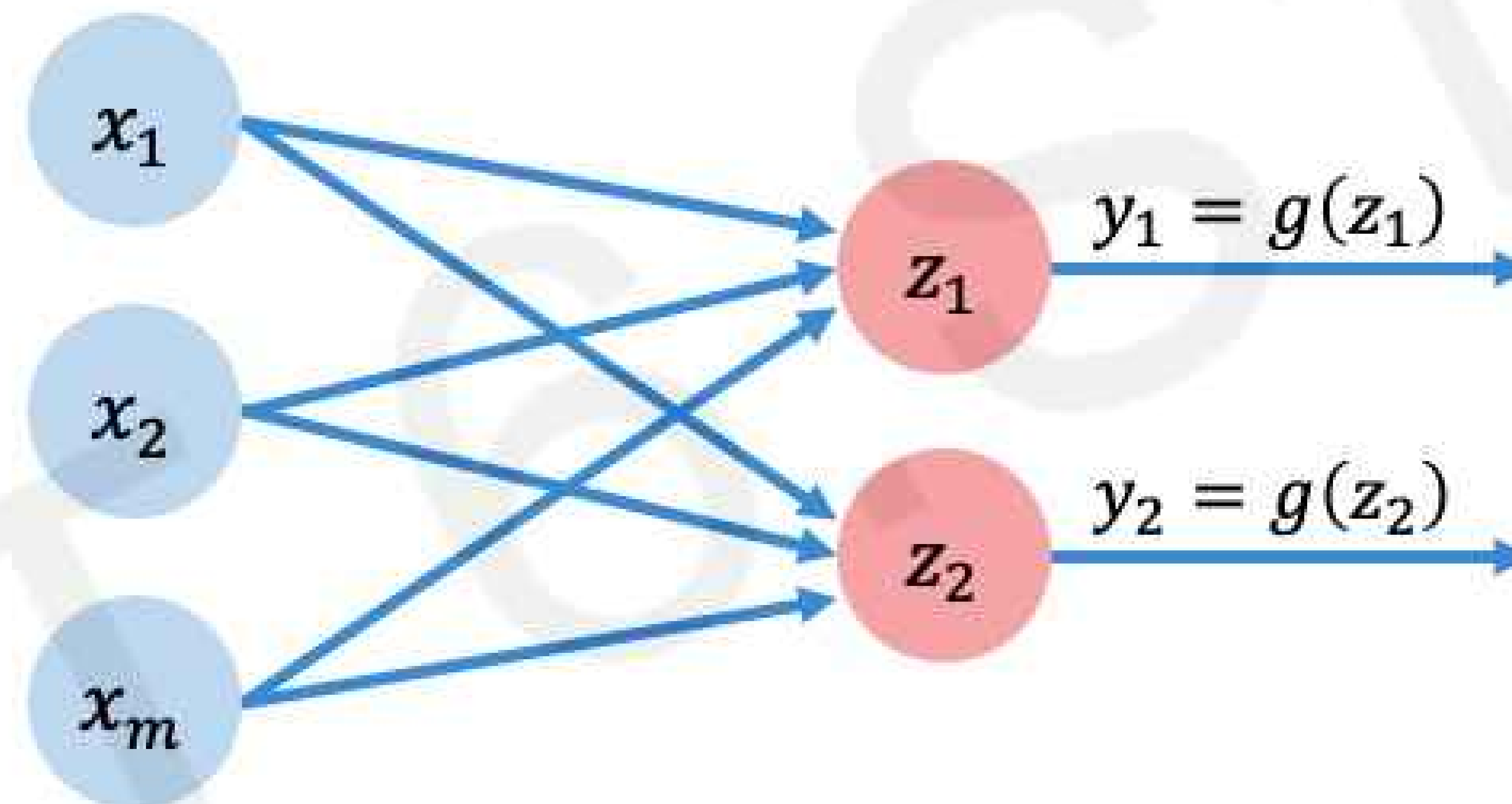
The Perceptron: Simplified



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

Multi Output Perceptron

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
```

```
class MyDenseLayer(nn.Module):  
    def __init__(self, input_dim, output_dim):  
        super(MyDenseLayer, self).__init__()
```

```
        # Initialize weights and bias  
        self.W = nn.Parameter(torch.randn(input_dim,  
                                           output_dim, requires_grad=True))  
        self.b = nn.Parameter(torch.randn(1, output_dim,  
                                           requires_grad=True))
```

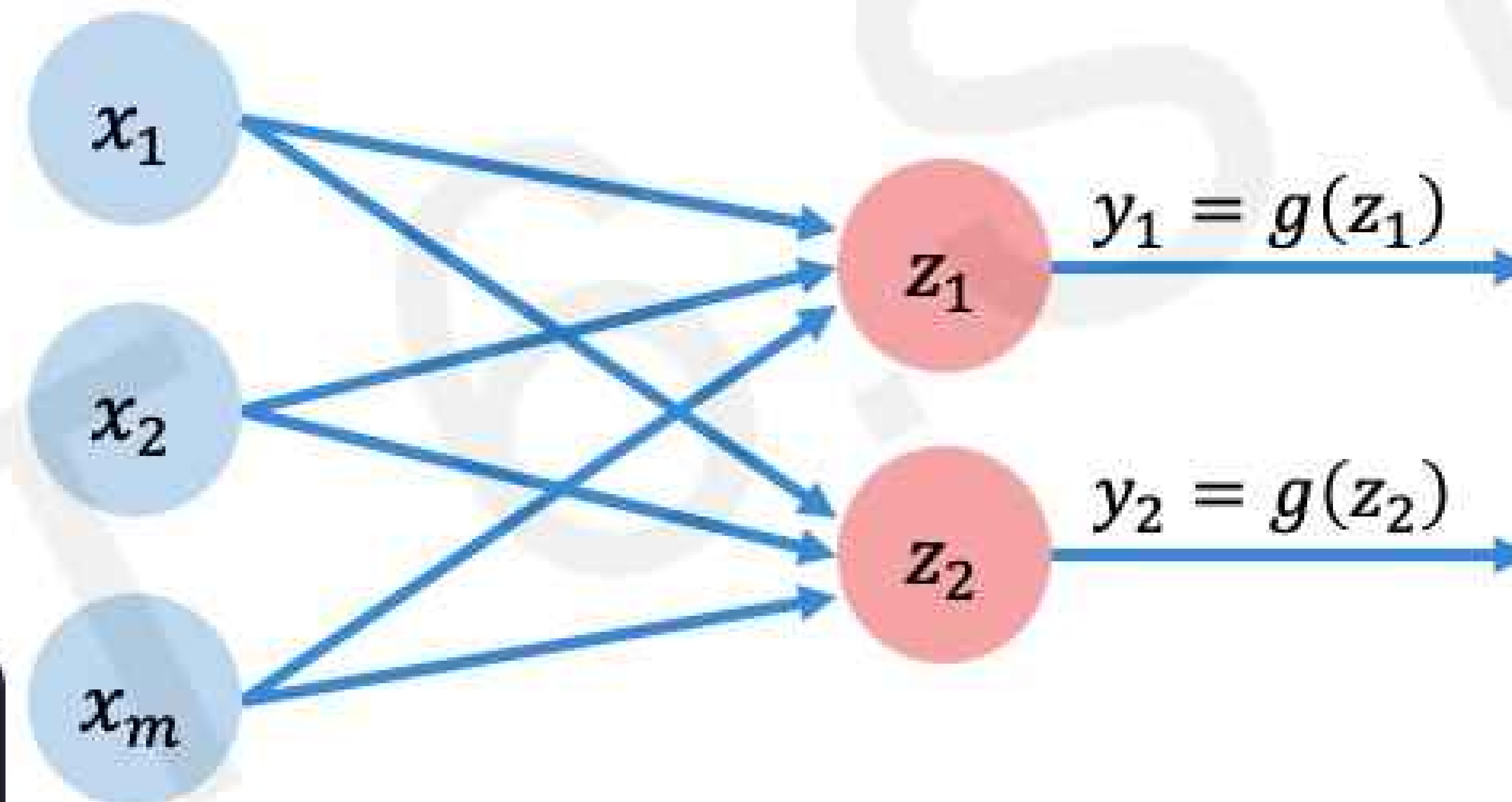
```
    def forward(self, inputs):
```

```
        # Forward propagate the inputs  
        z = torch.matmul(inputs, self.W) + self.b
```

```
        # Feed through a non-linear activation  
        output = torch.sigmoid(z)  
        return output
```

Multi Output Perceptron

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

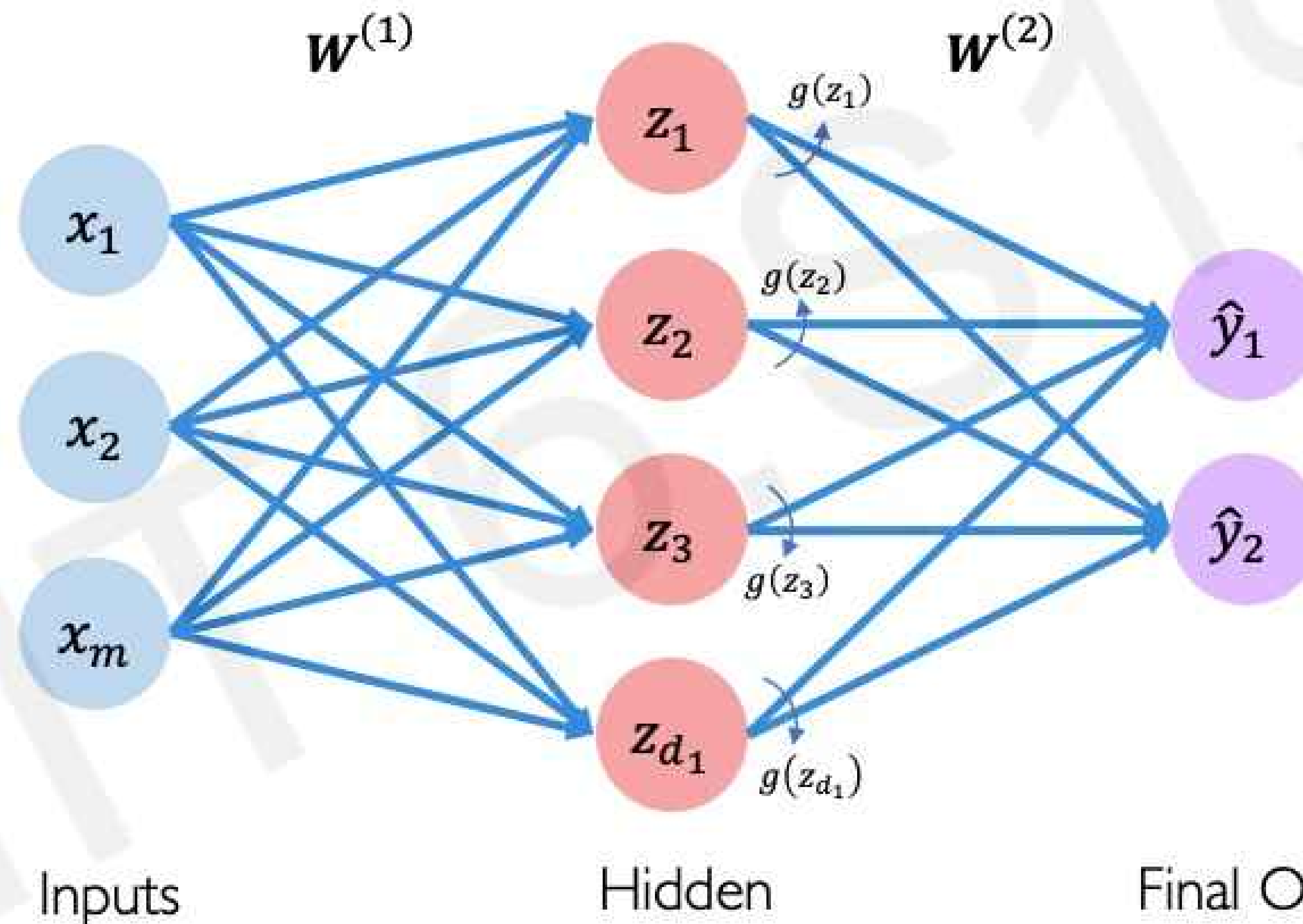


```
import tensorflow as tf  
  
layer = tf.keras.layers.Dense(  
    units=2)
```

```
import torch.nn as nn  
  
layer = nn.Linear(in_features=m,  
    out_features=2)
```

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

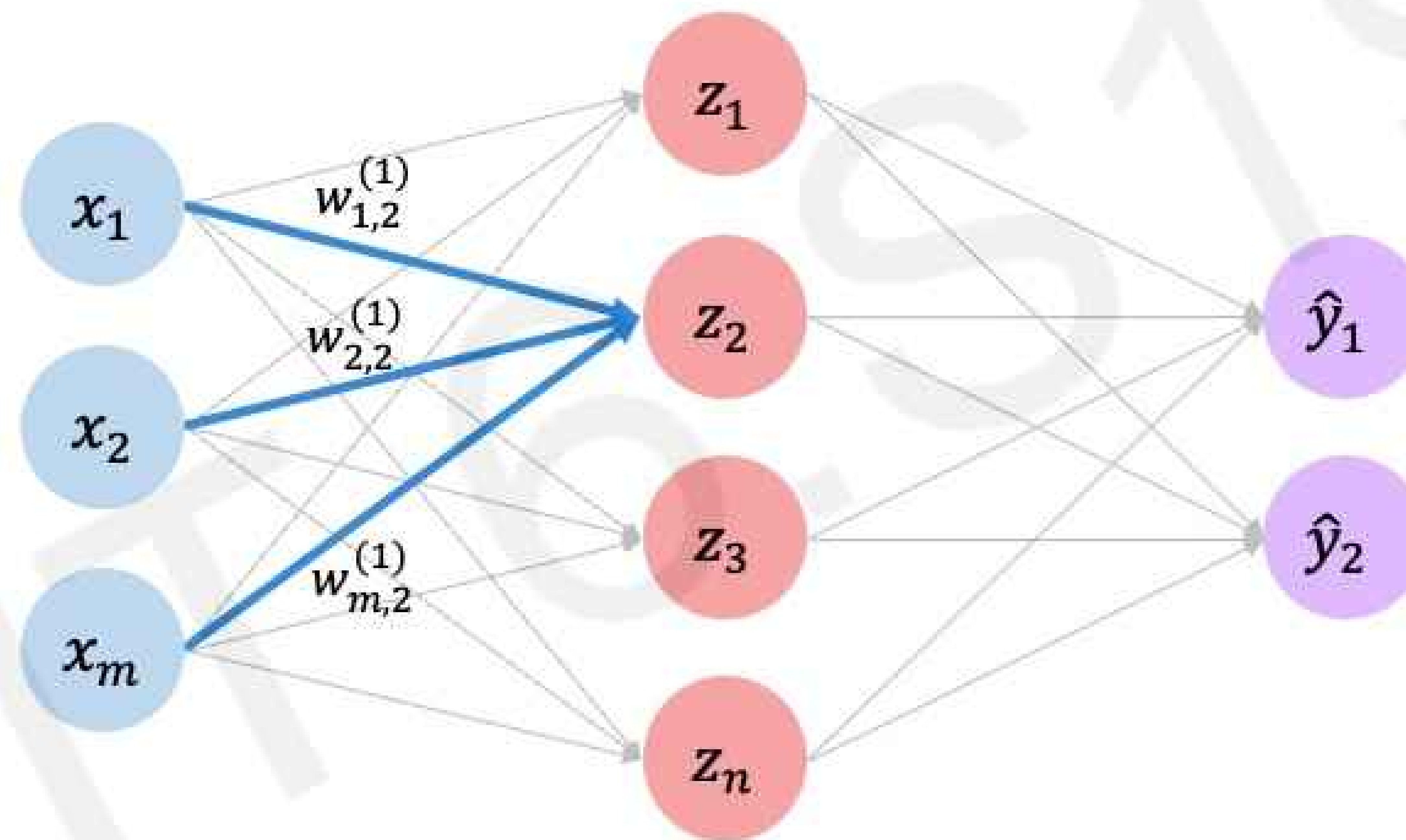
Single Layer Neural Network



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

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

Single Layer Neural Network

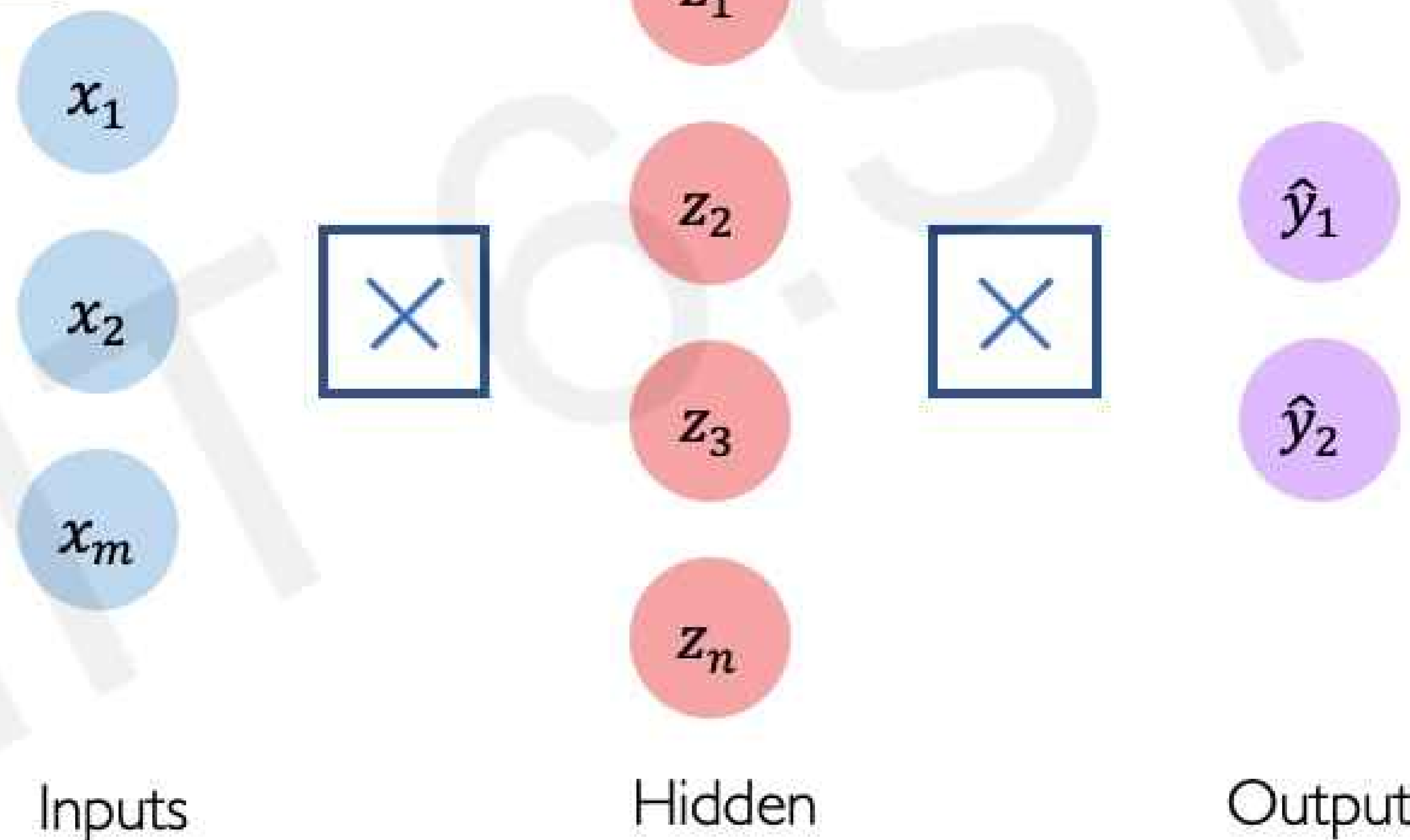


$$\begin{aligned} z_2 &= w_{0,2}^{(1)} + \sum_{j=1}^m x_j w_{j,2}^{(1)} \\ &= w_{0,2}^{(1)} + x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + x_m w_{m,2}^{(1)} \end{aligned}$$

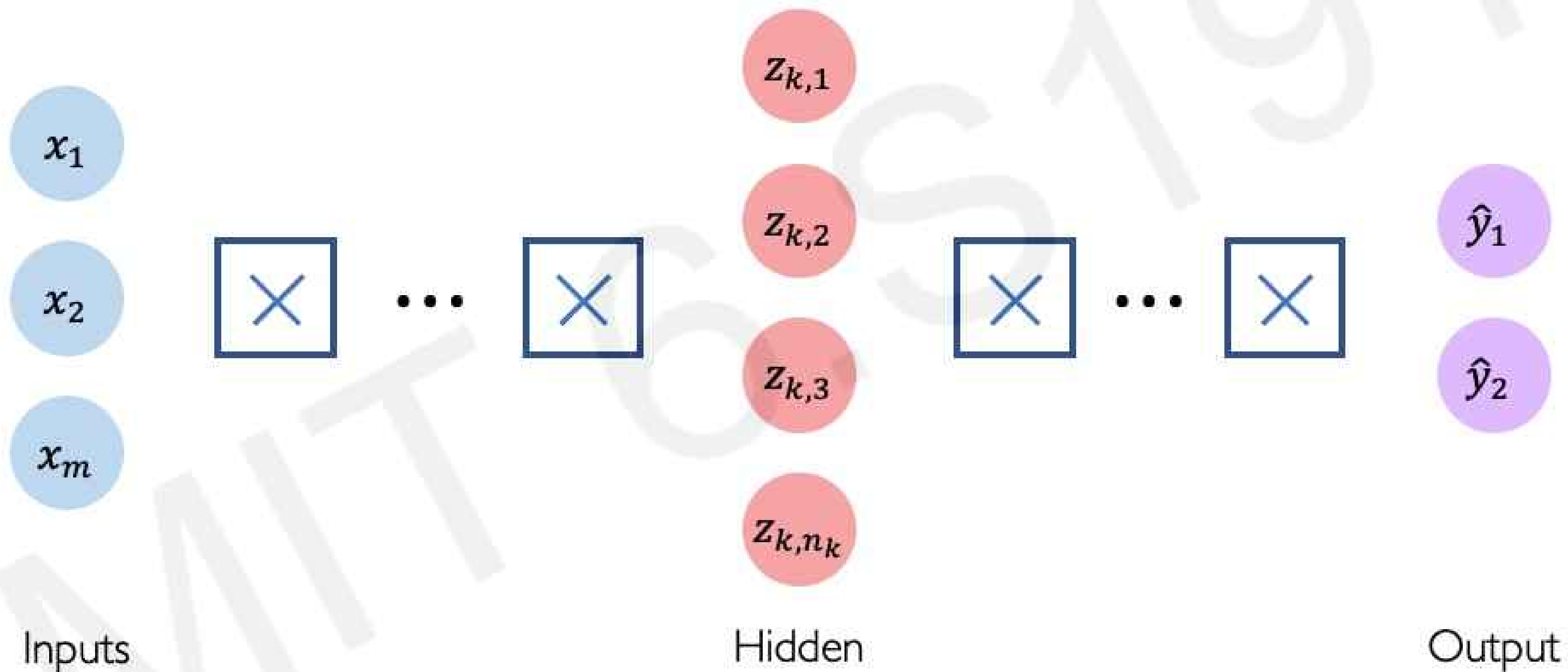
Multi Output Perceptron

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

```
from torch import nn  
  
model = nn.Sequential(  
    nn.Linear(m, n),  
    nn.ReLU(),  
    nn.Linear(n, 2)  
)
```

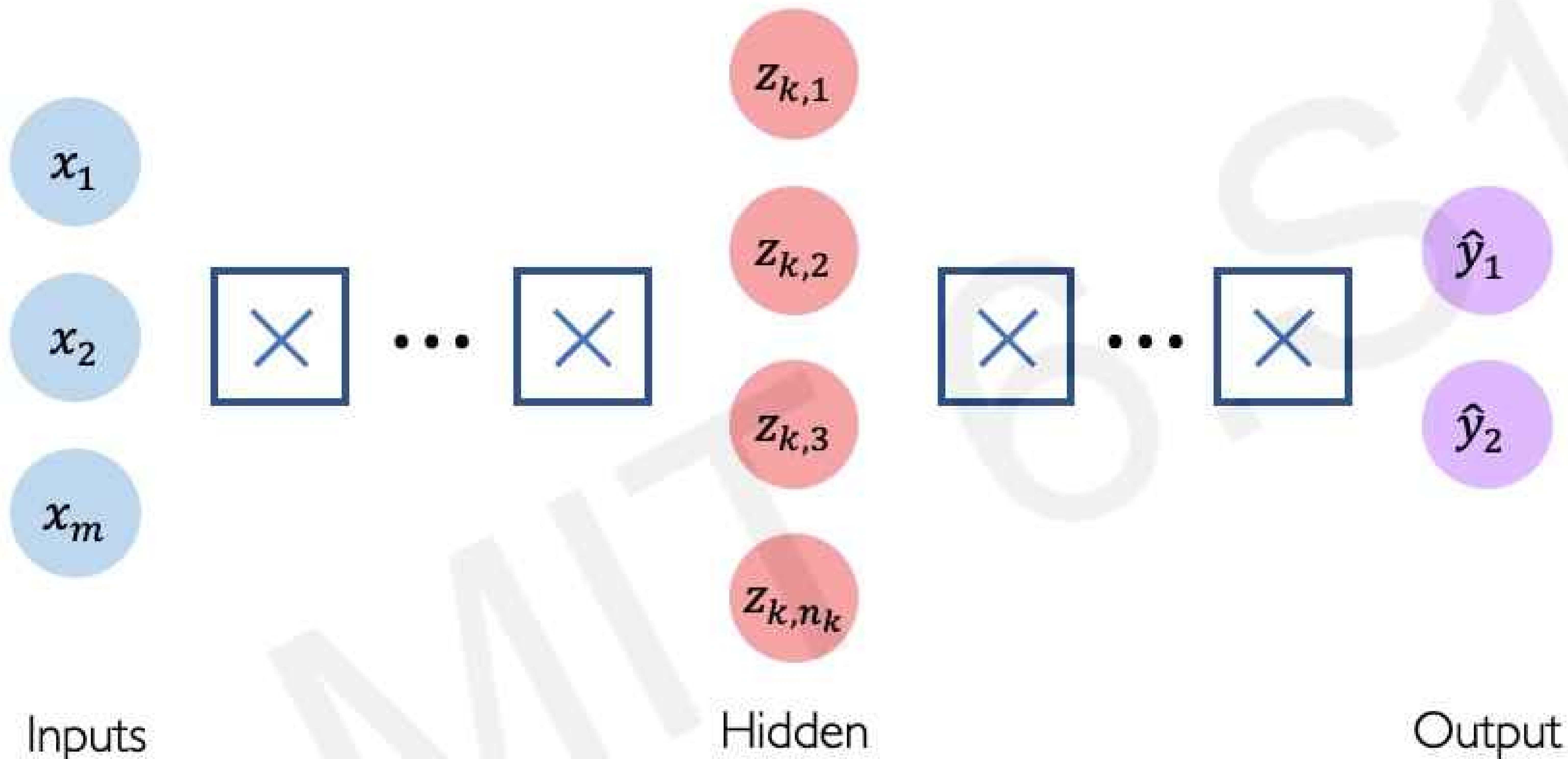


Deep Neural Network



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

Deep Neural Network



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



```
import tensorflow as tf

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



```
from torch import nn

model = nn.Sequential(
    nn.Linear(m, n1),
    nn.ReLU(),
    ...
    nn.ReLU(),
    nn.Linear(nK, 2)
)
```

Applying Neural Networks

Example Problem

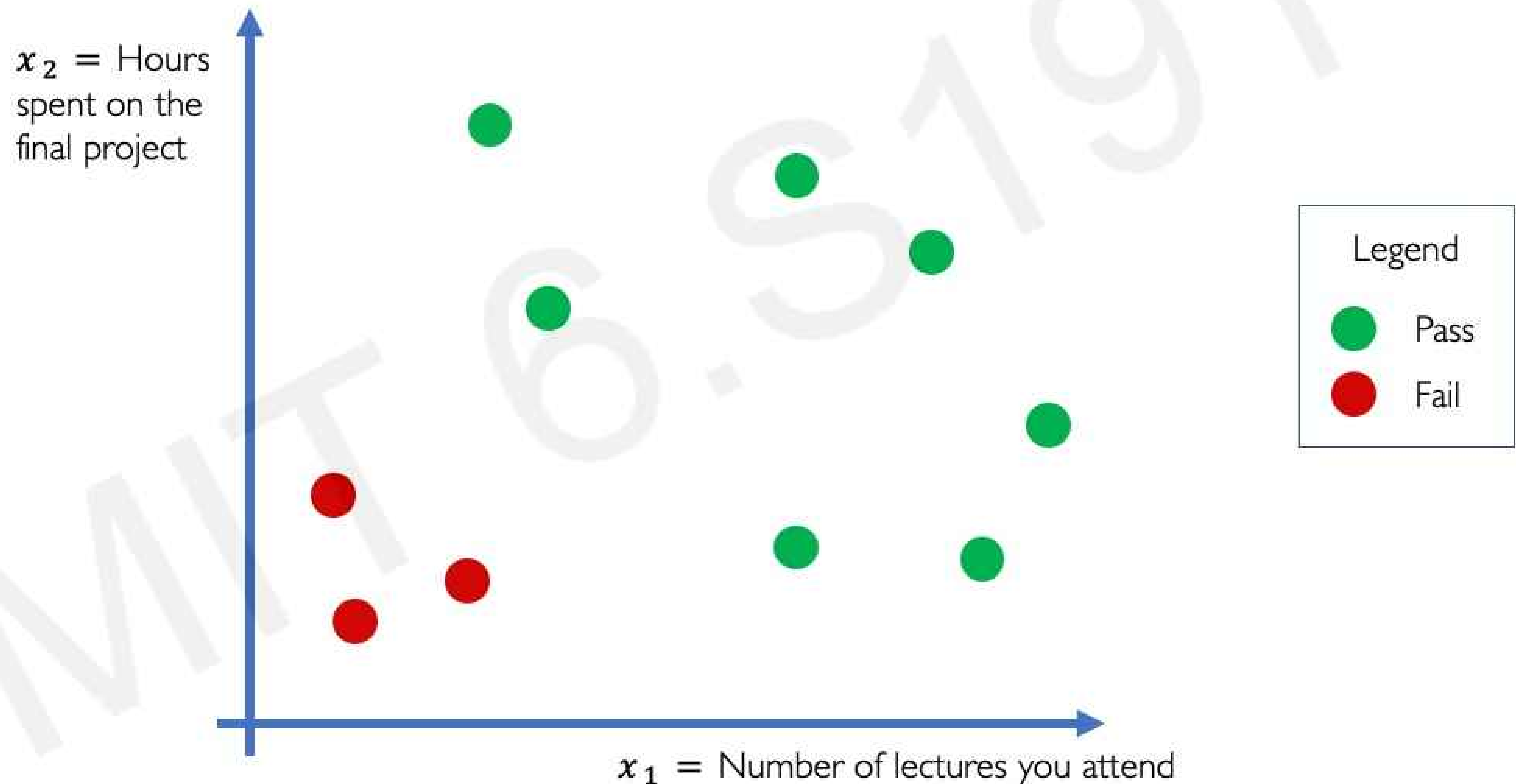
Will I pass this class?

Let's start with a simple two feature model

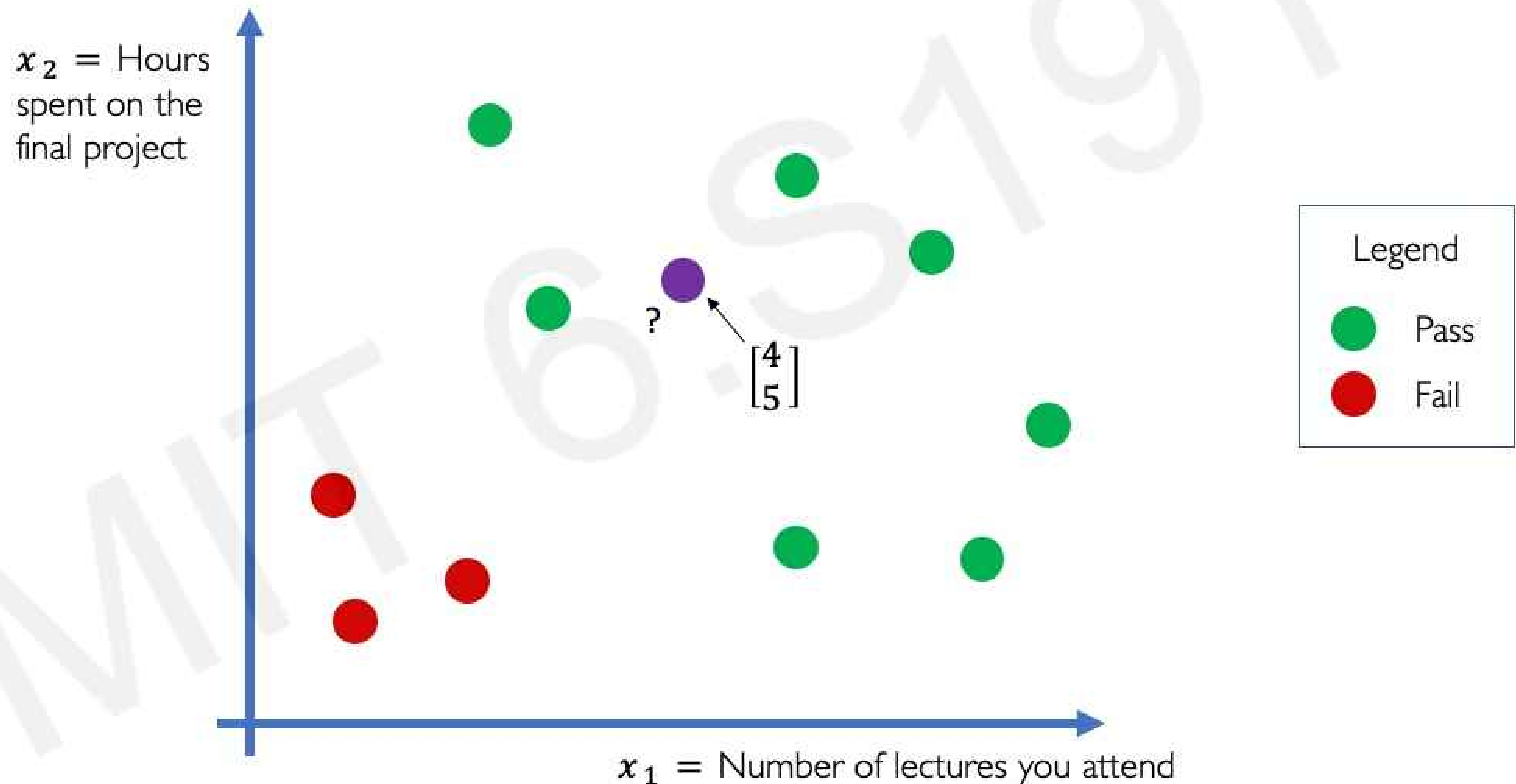
x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

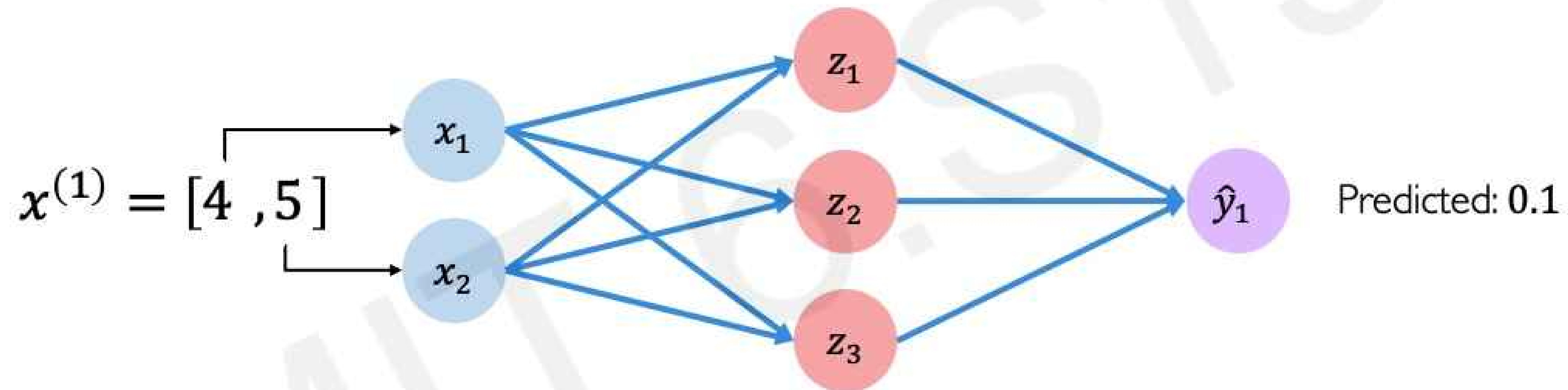
Example Problem: Will I pass this class?



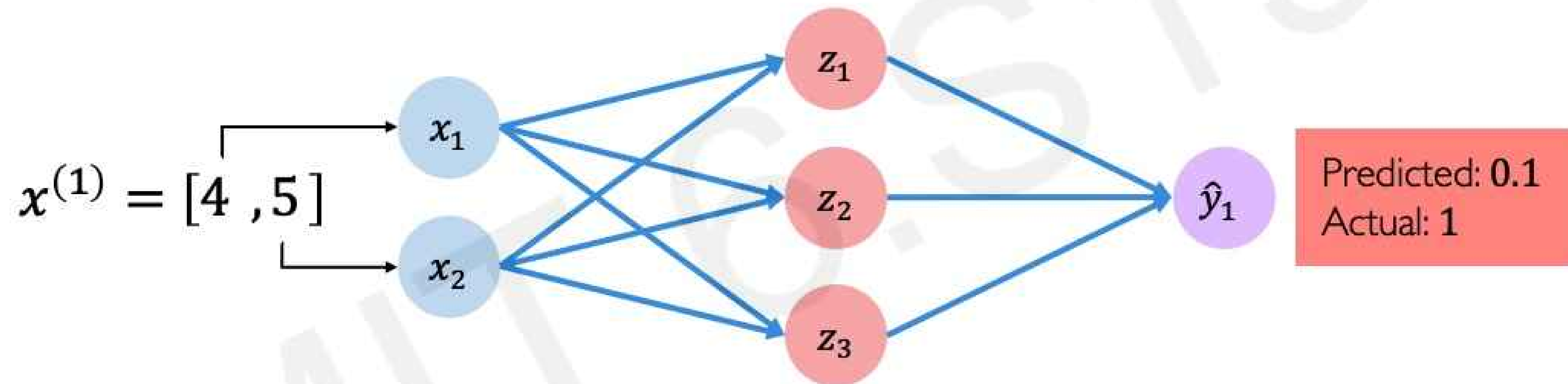
Example Problem: Will I pass this class?



Example Problem: Will I pass this class?

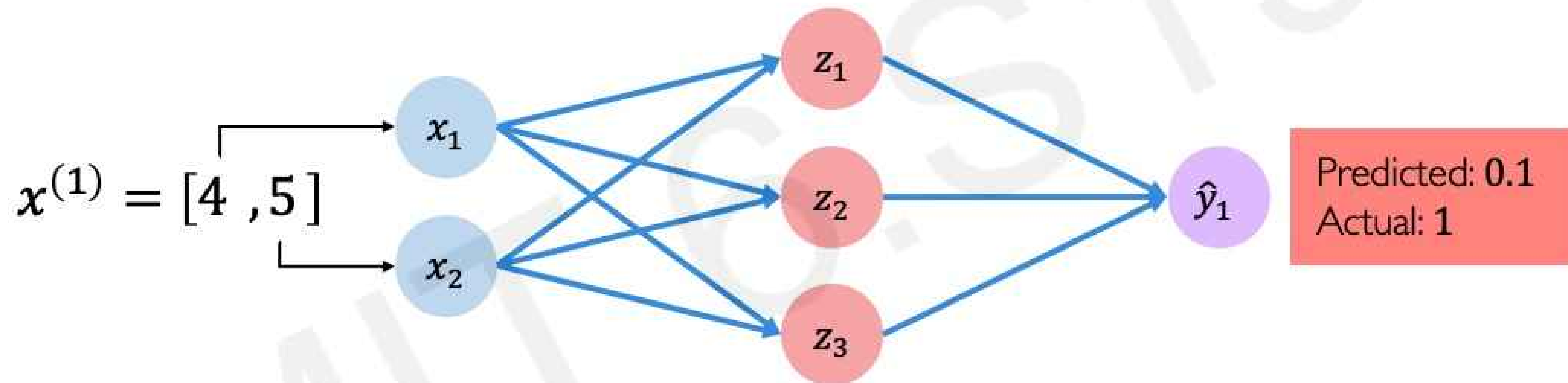


Example Problem: Will I pass this class?



Quantifying Loss

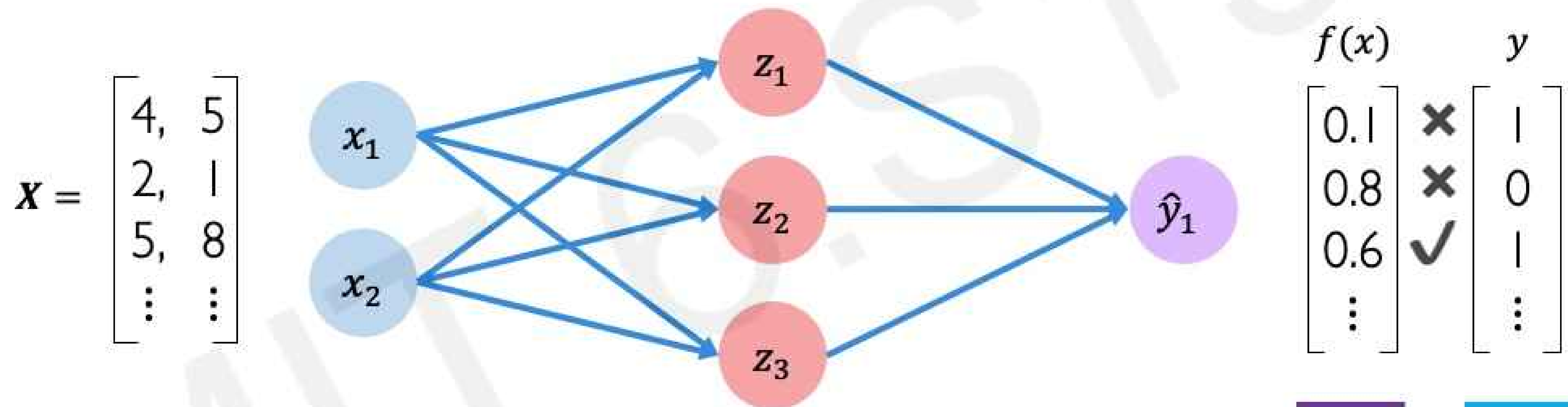
The **loss** of our network measures the cost incurred from incorrect predictions



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

Empirical Loss

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



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

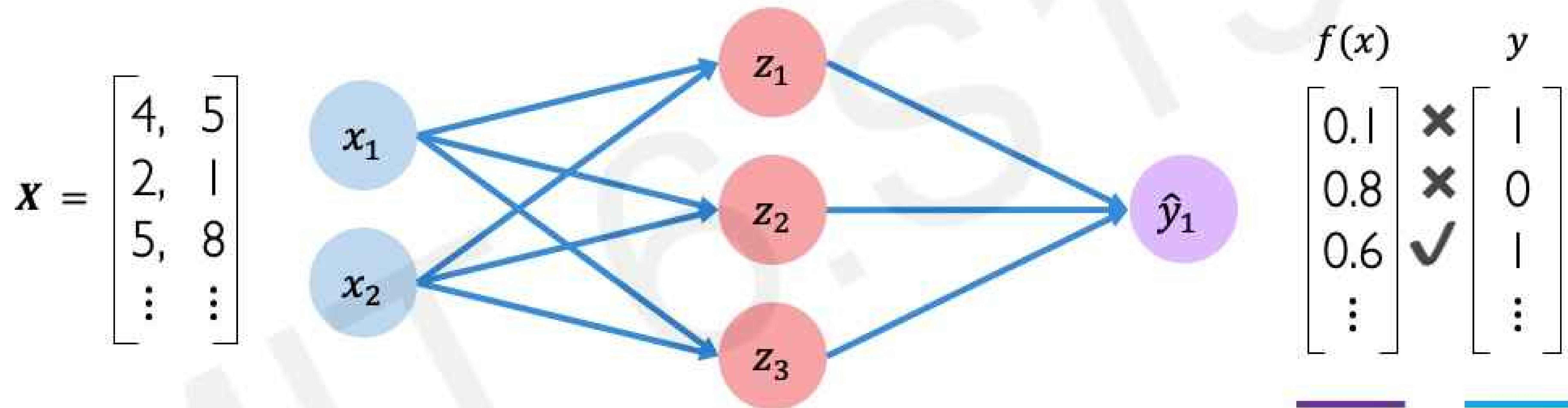
$$J(W) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\underbrace{f(x^{(i)}; W)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Predicted

Actual

Binary Cross Entropy Loss

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



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



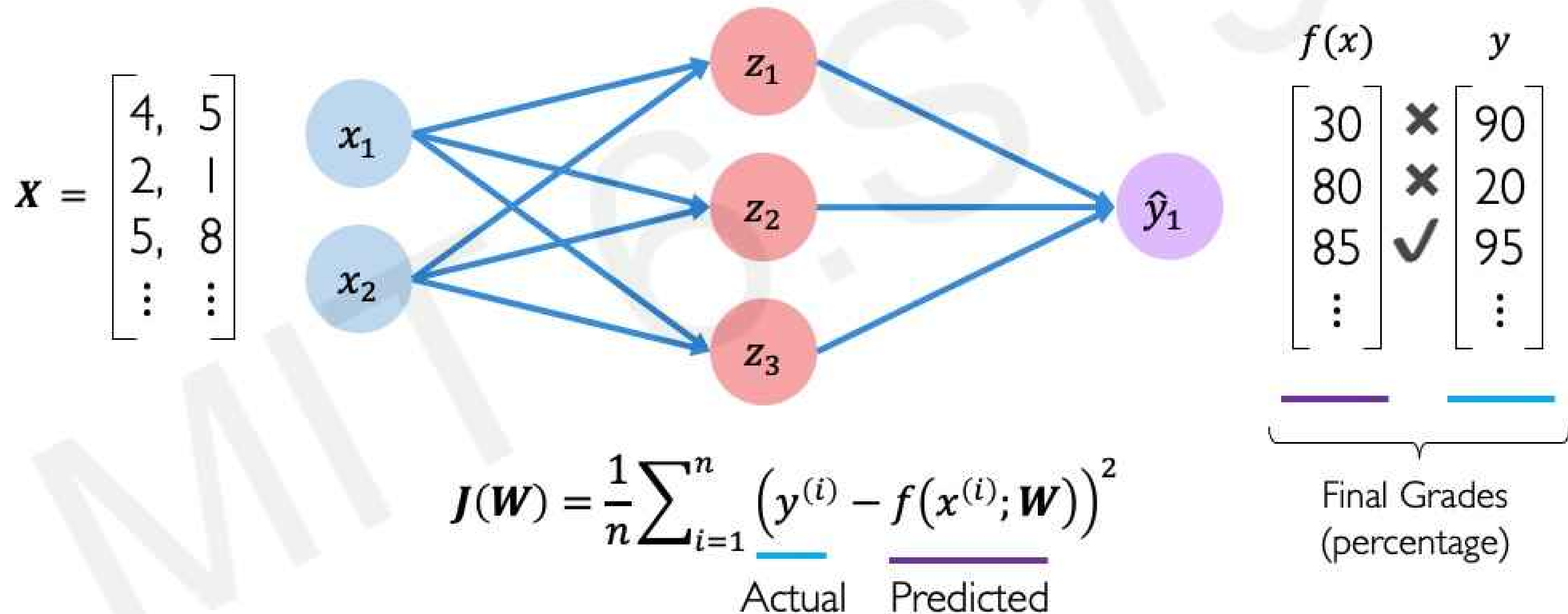
```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```

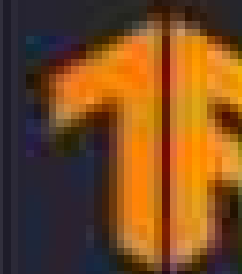


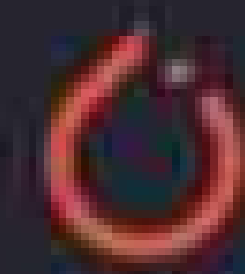
```
loss = torch.nn.functional.cross_entropy( predicted, y )
```

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 )
```

```
loss = torch.nn.functional.mse_loss(  
    predicted, y )  

```

Training Neural Networks

Loss Optimization

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

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

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

Loss Optimization

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

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

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

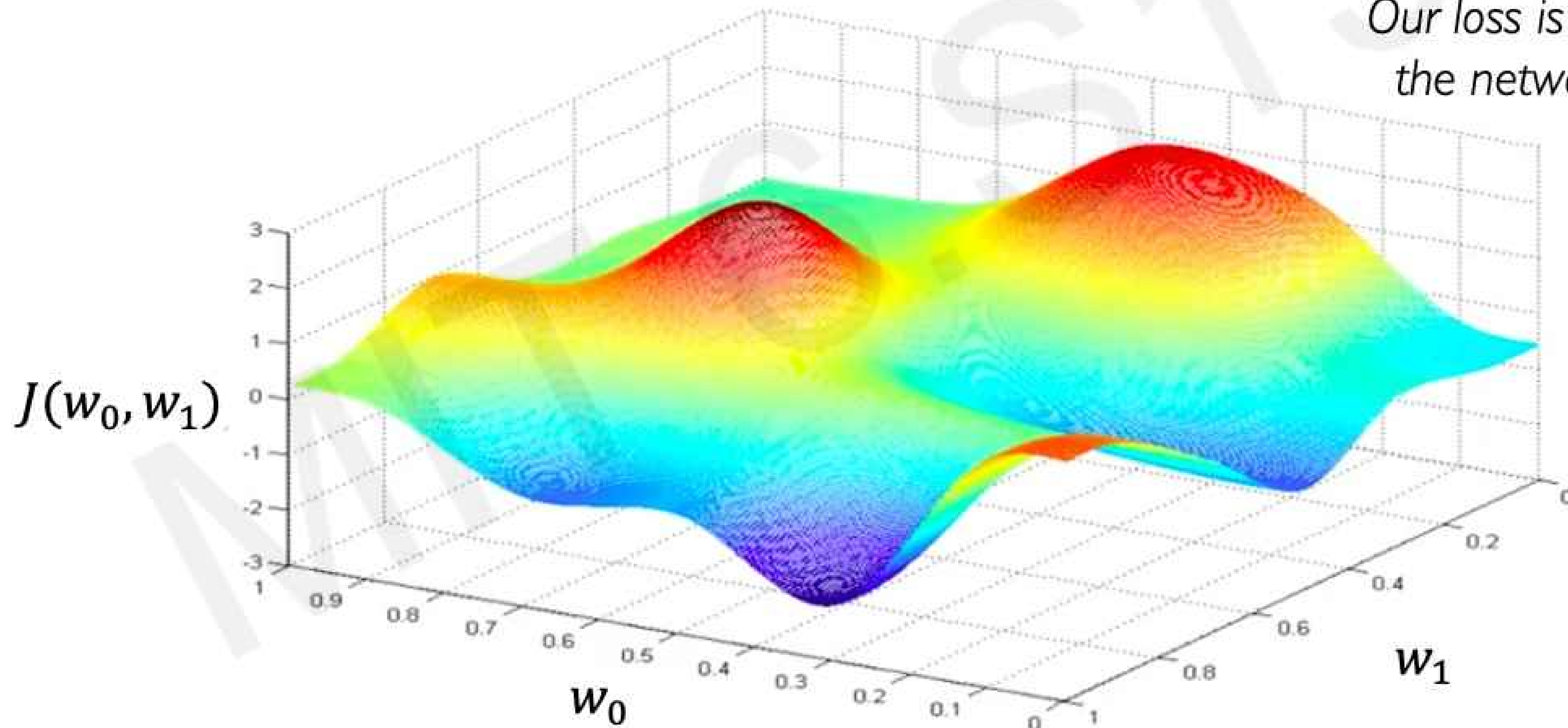
Remember:

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

Loss Optimization

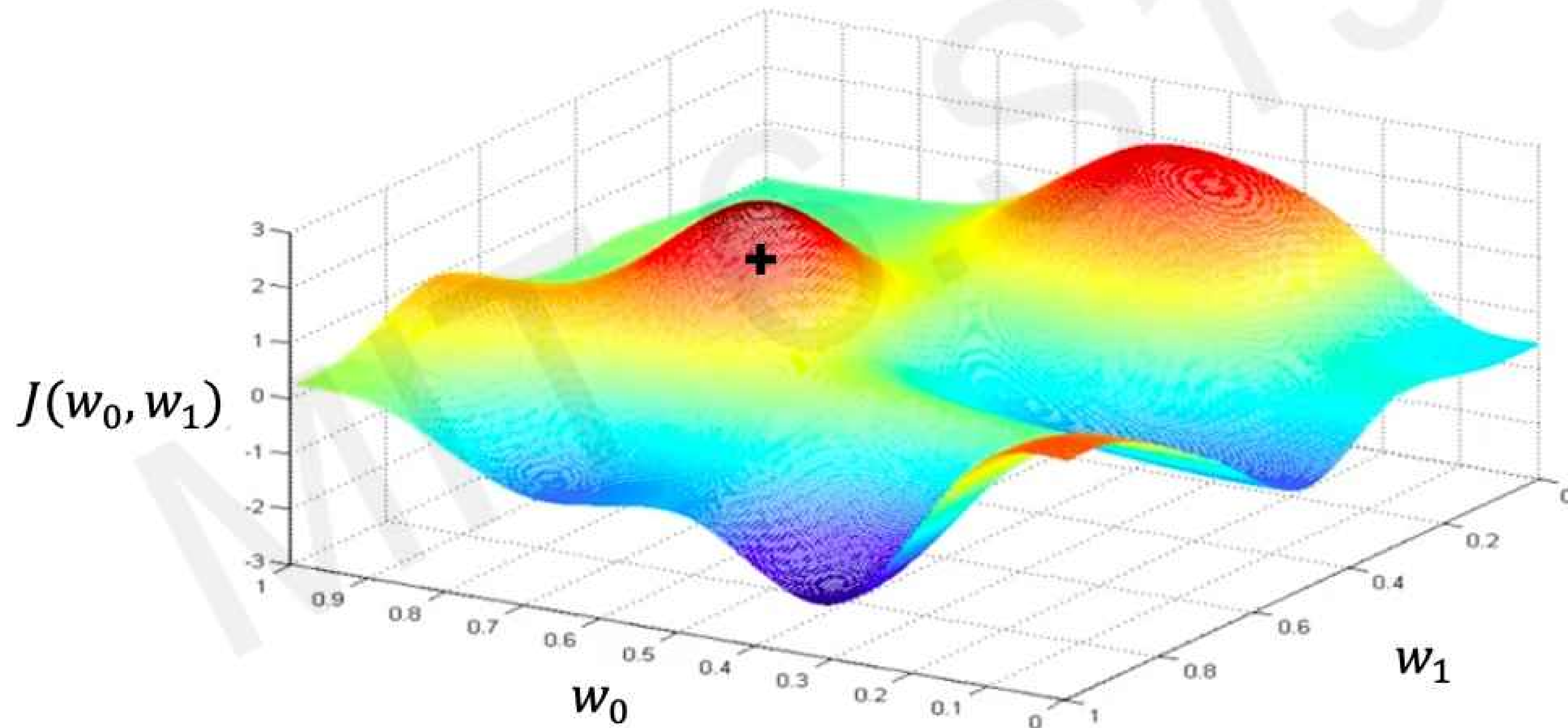
$$W^* = \underset{W}{\operatorname{argmin}} J(W)$$

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



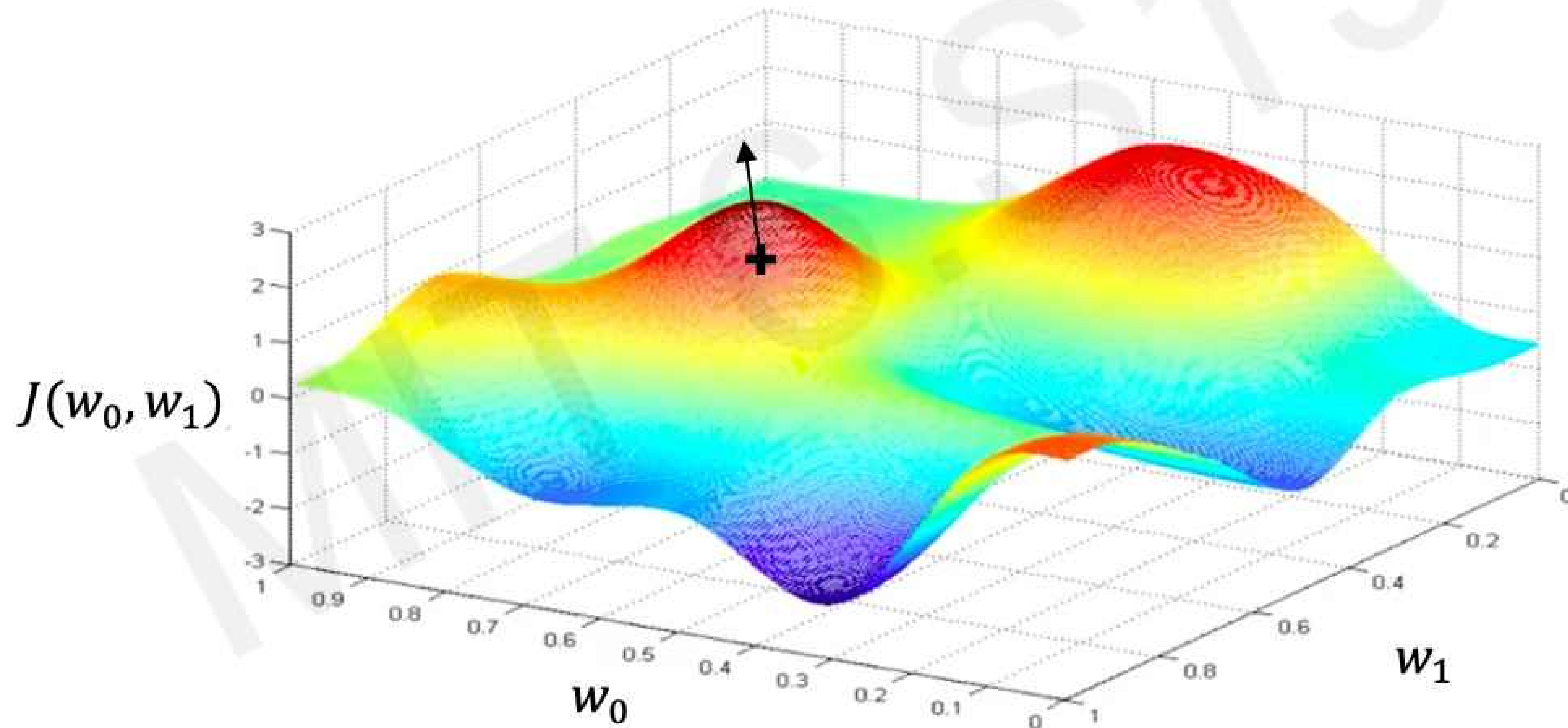
Loss Optimization

Randomly pick an initial (w_0, w_1)



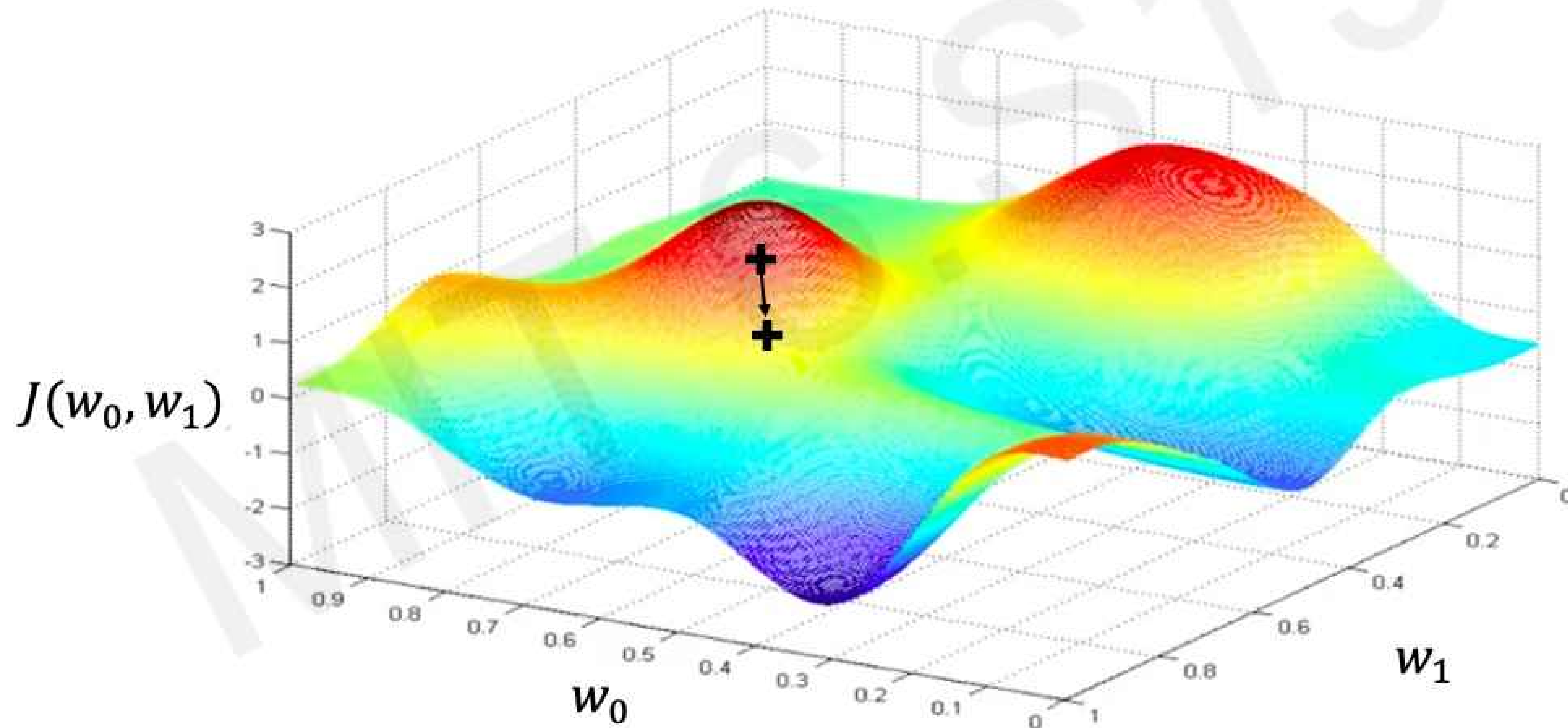
Loss Optimization

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



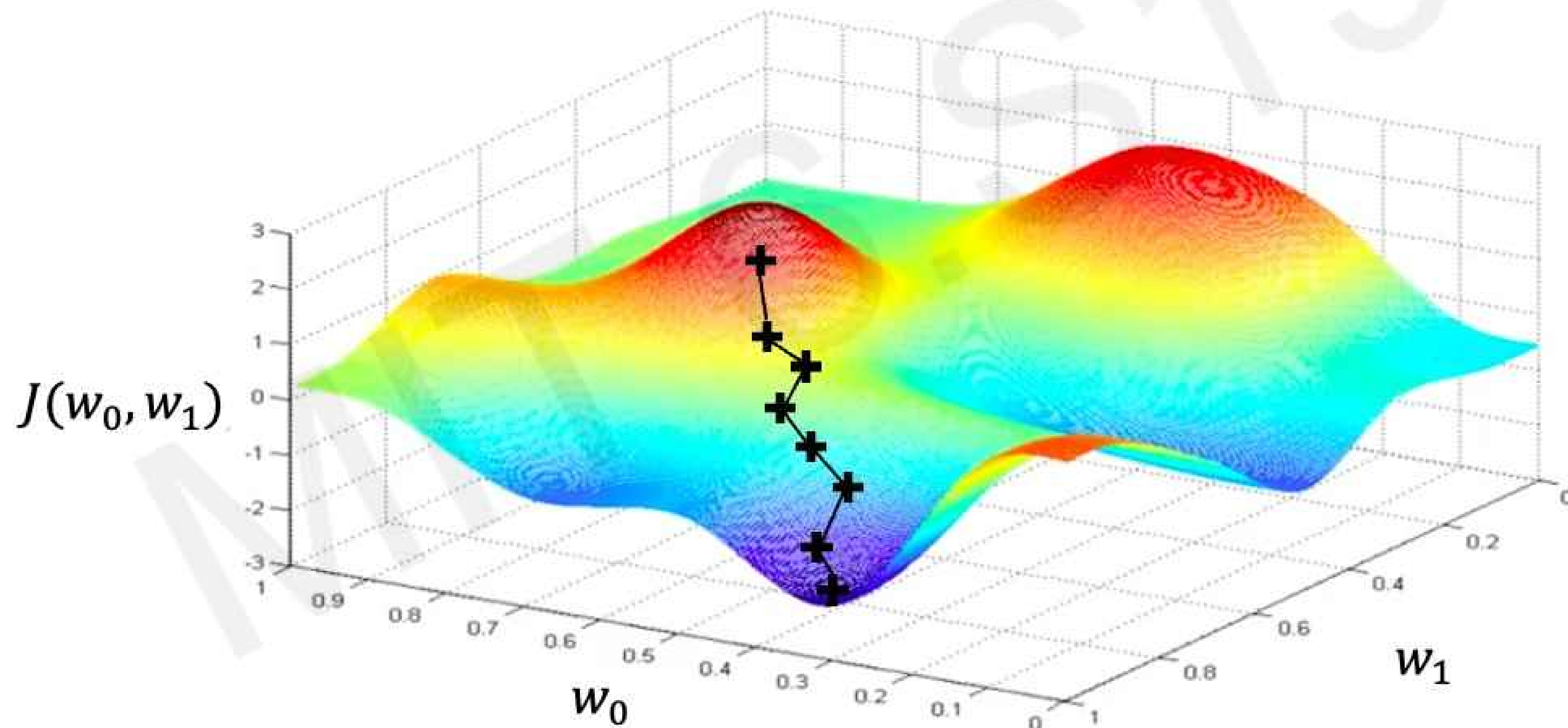
Loss Optimization

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

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

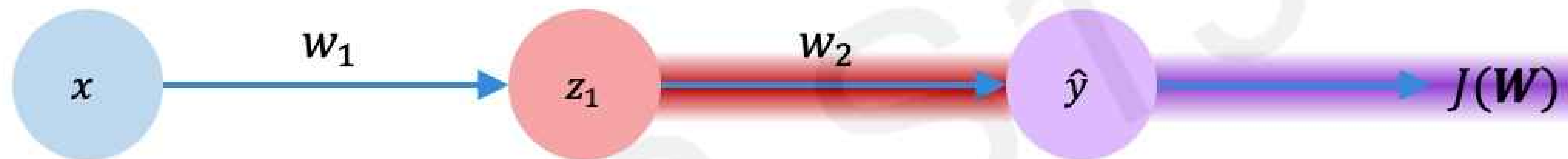
Computing Gradients: Backpropagation



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

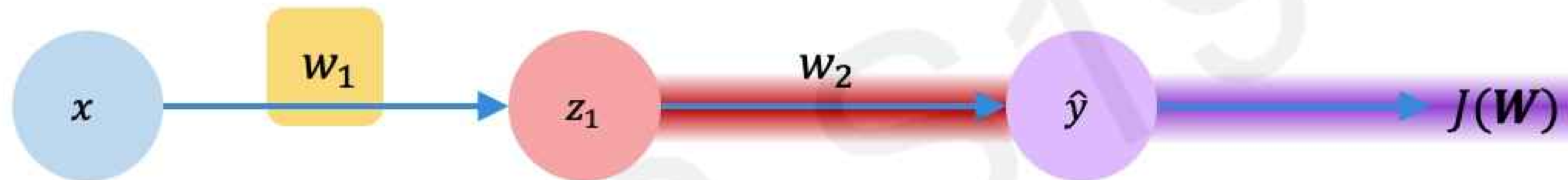
Let's use the chain rule!

Computing Gradients: Backpropagation



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

Computing Gradients: Backpropagation

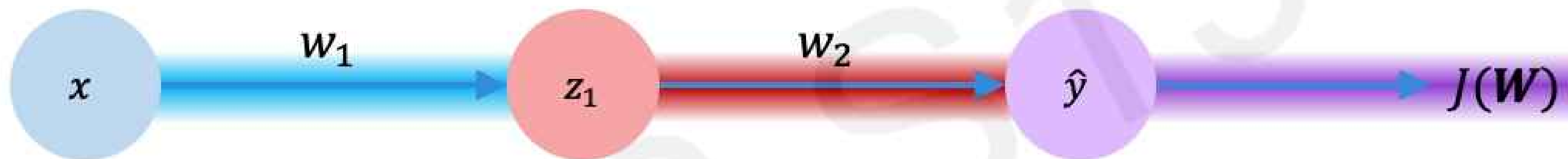


$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

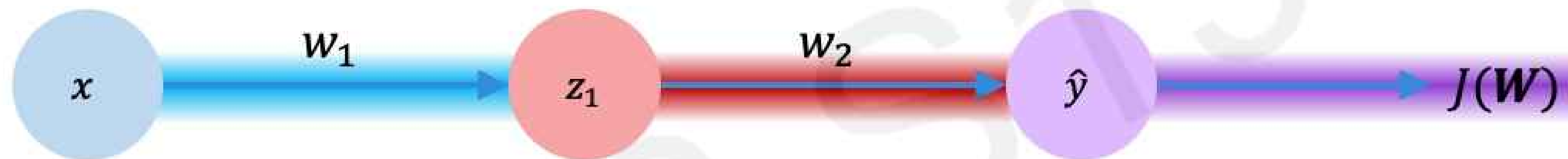
Apply chain rule!

Computing Gradients: Backpropagation



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

Computing Gradients: Backpropagation

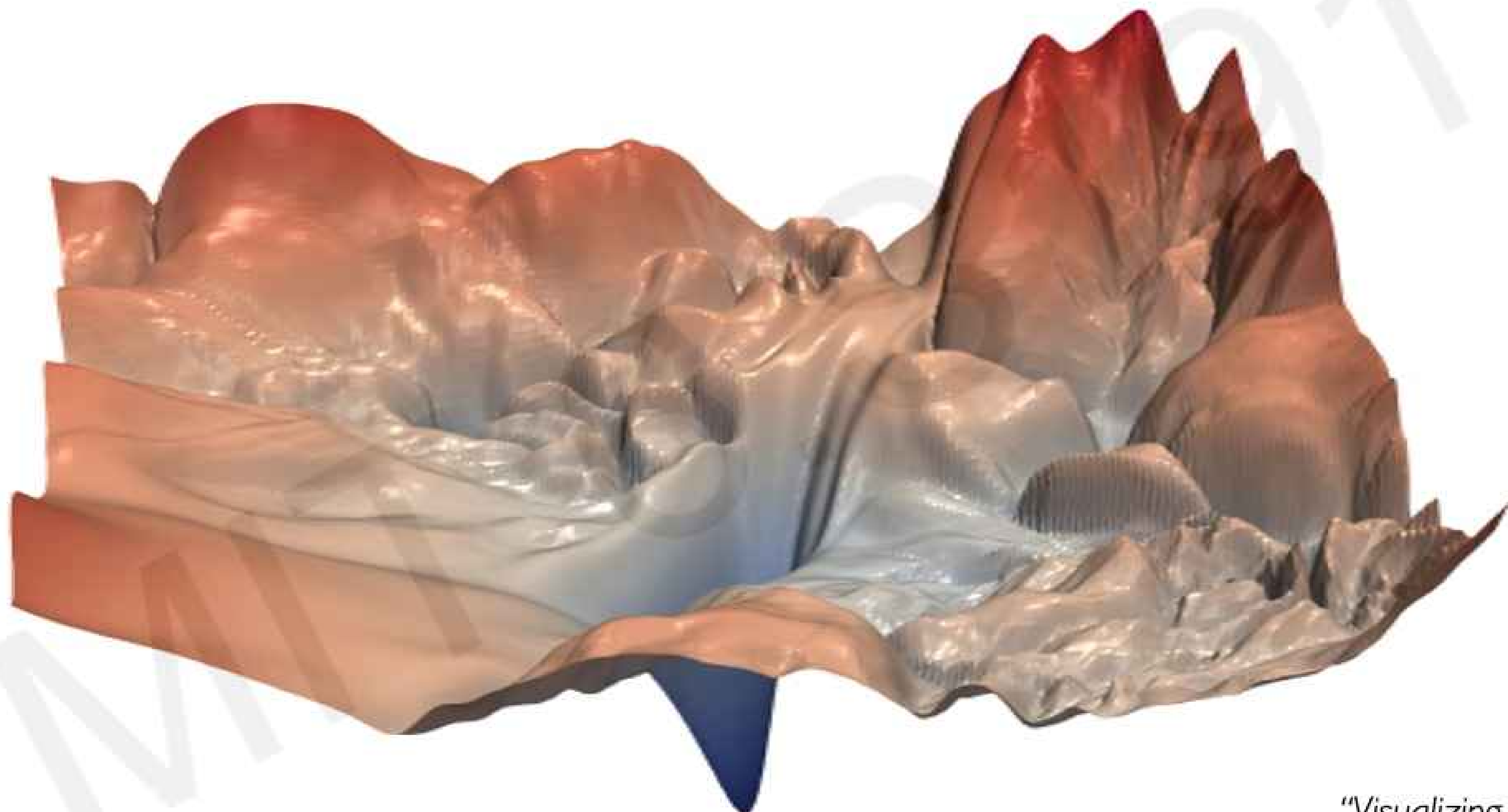


$$\frac{\partial J(W)}{\partial w_1} = \underbrace{\frac{\partial J(W)}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue}}$$

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

Neural Networks in Practice: Optimization

Training Neural Networks is Difficult



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

Loss Functions Can Be Difficult to Optimize

Remember:

Optimization through gradient descent

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

Loss Functions Can Be Difficult to Optimize

Remember:

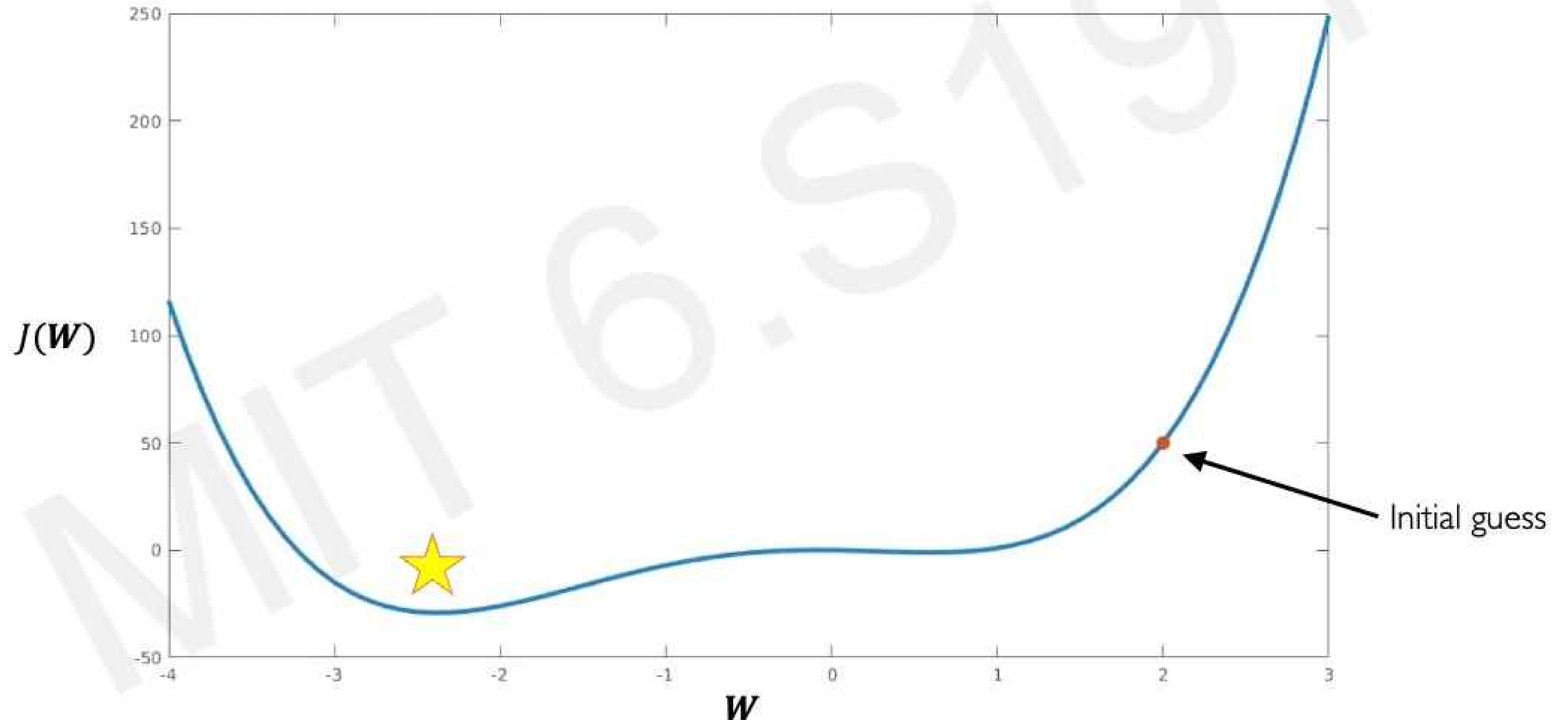
Optimization through gradient descent

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

How can we set the
learning rate?

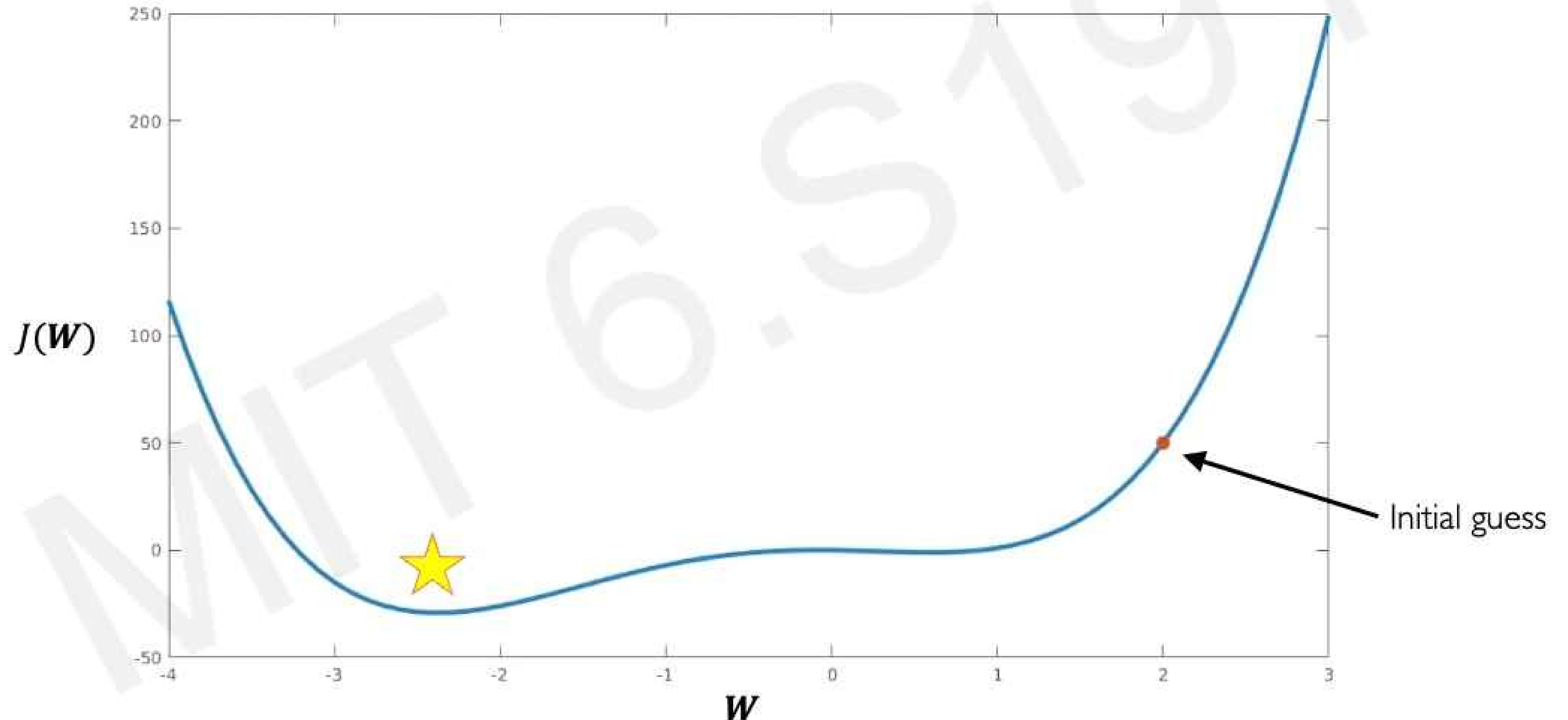
Setting the Learning Rate

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



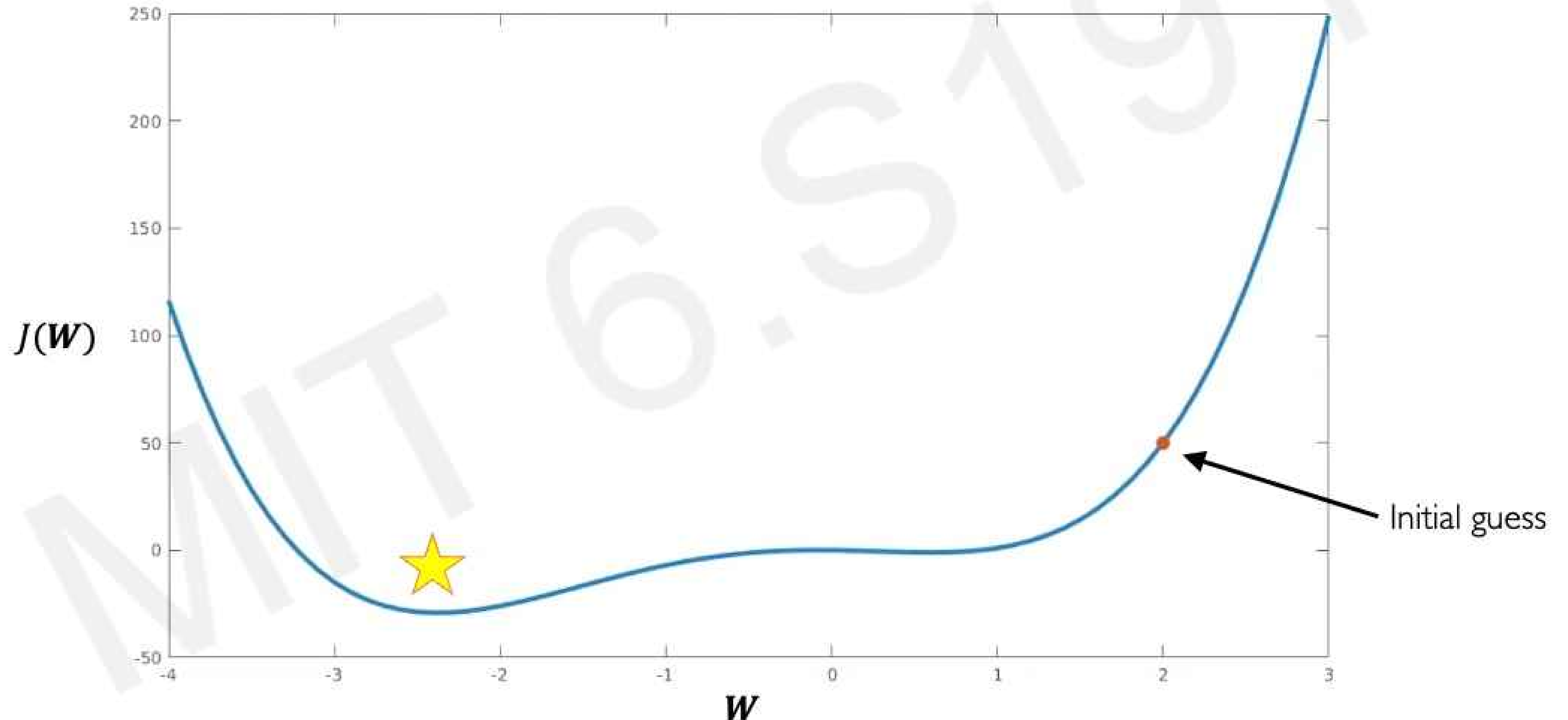
Setting the Learning Rate

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 1:

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

How to deal with this?

Idea 1:

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

Idea 2:

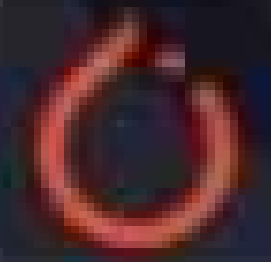
Do something smarter!

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

Adaptive Learning Rates

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

Gradient Descent Algorithms

Algorithm	TF Implementation	Torch Implementation	Reference
• SGD	 <code>tf.keras.optimizers.SGD</code>	 <code>torch.optim.SGD</code>	Kiefer & Wolfowitz, 1952.
• Adam	 <code>tf.keras.optimizers.Adam</code>	 <code>torch.optim.Adam</code>	Kingma et al., 2014.
• Adadelta	 <code>tf.keras.optimizers.Adadelta</code>	 <code>torch.optim.Adadelta</code>	Zeiler et al., 2012.
• Adagrad	 <code>tf.keras.optimizers.Adagrad</code>	 <code>torch.optim.Adagrad</code>	Duchi et al., 2011.
• RMSProp	 <code>tf.keras.optimizers.RMSProp</code>	 <code>torch.optim.RMSProp</code>	

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))
```



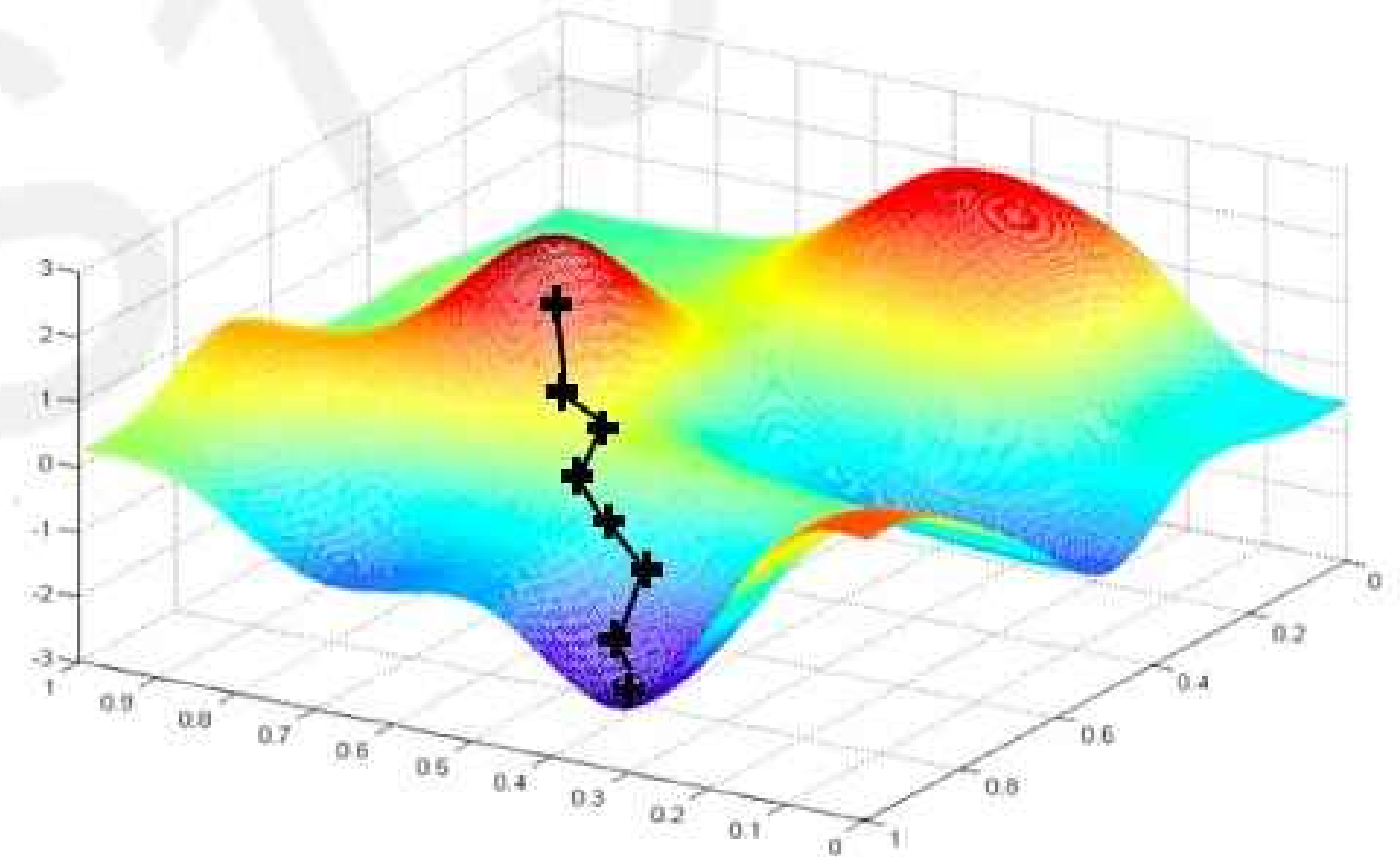
Can replace with
any TensorFlow
optimizer!

Neural Networks in Practice: Mini-batches

Gradient Descent

Algorithm

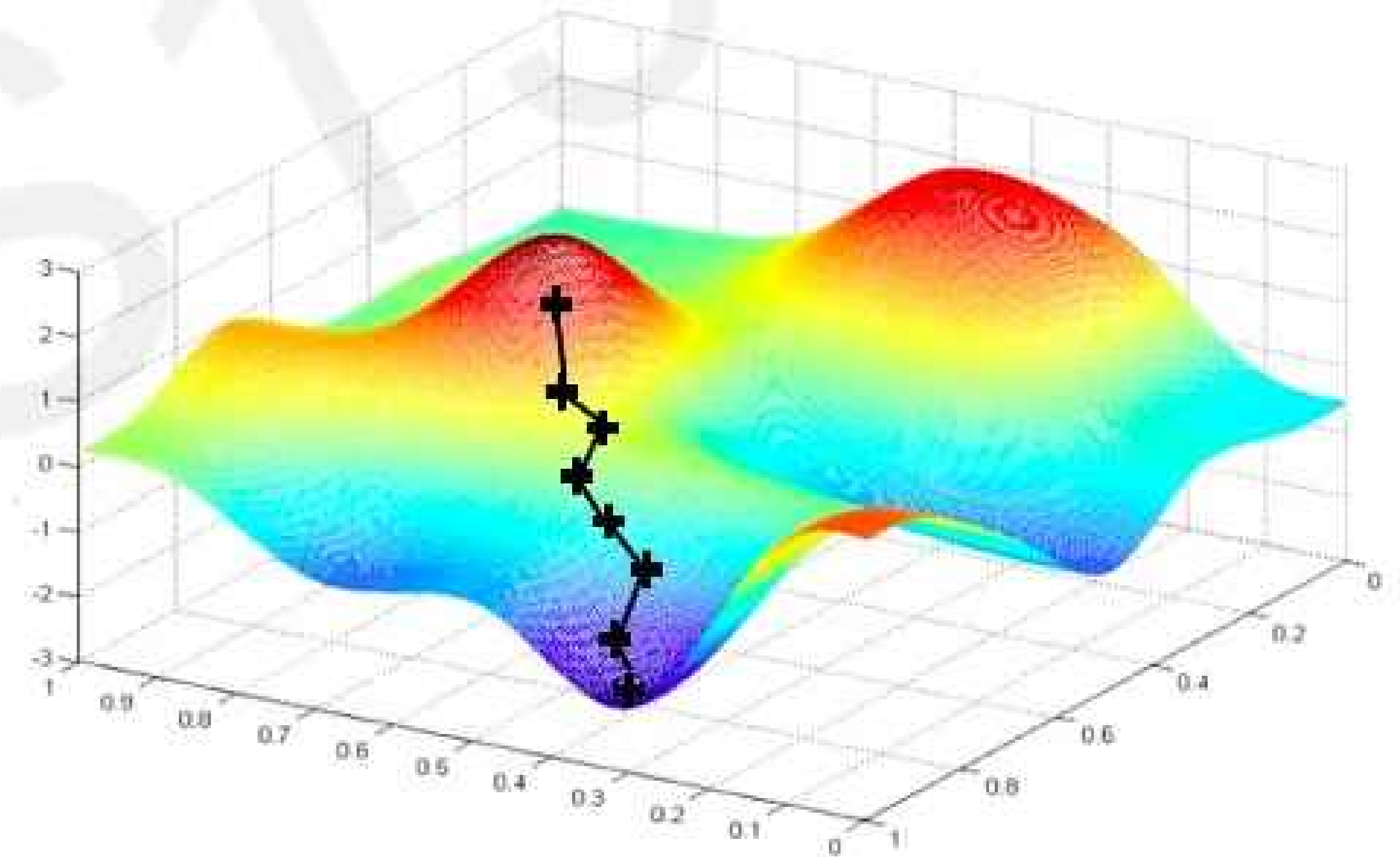
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights



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, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

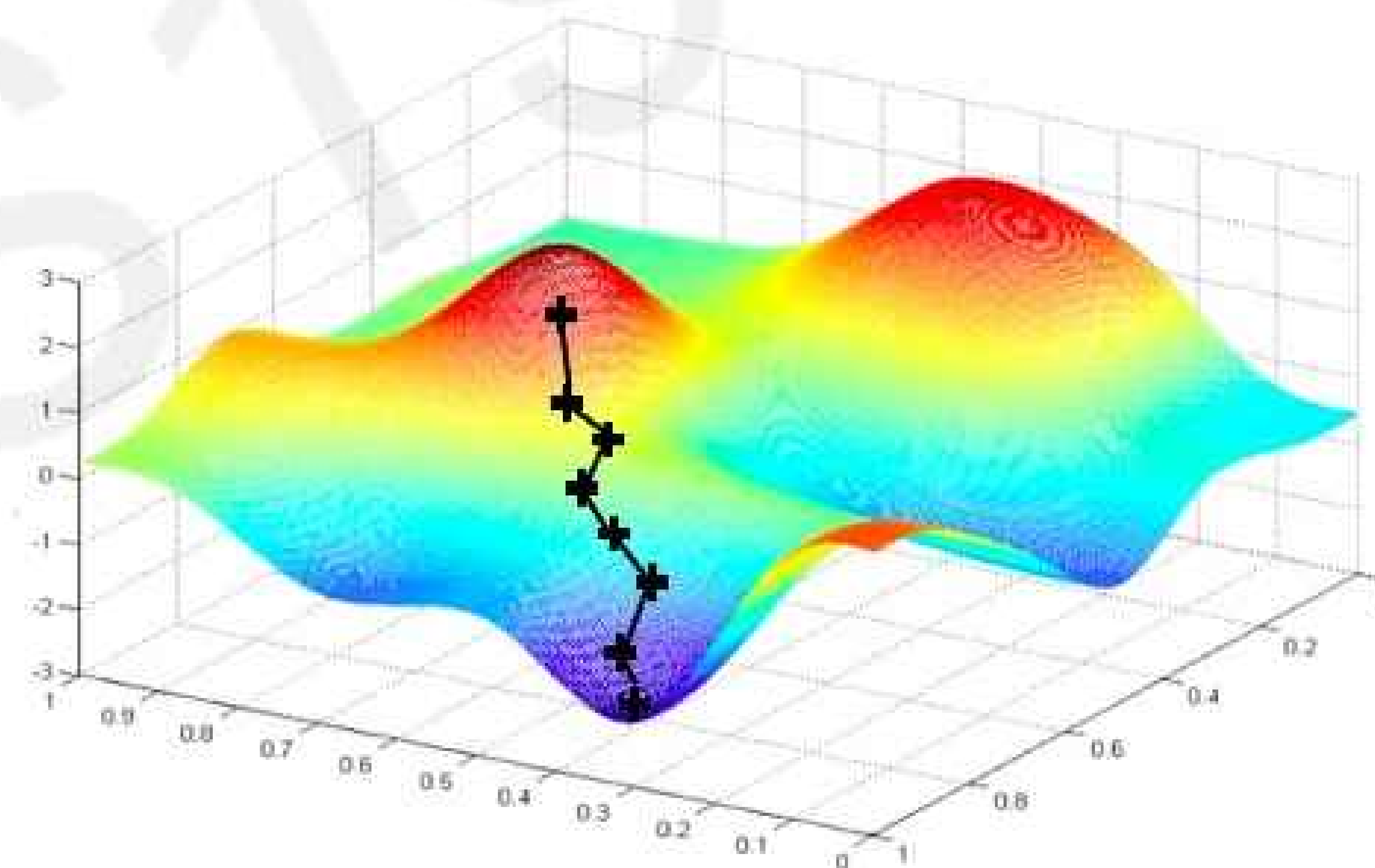


Can be very
computationally
intensive to compute!

Stochastic Gradient Descent

Algorithm

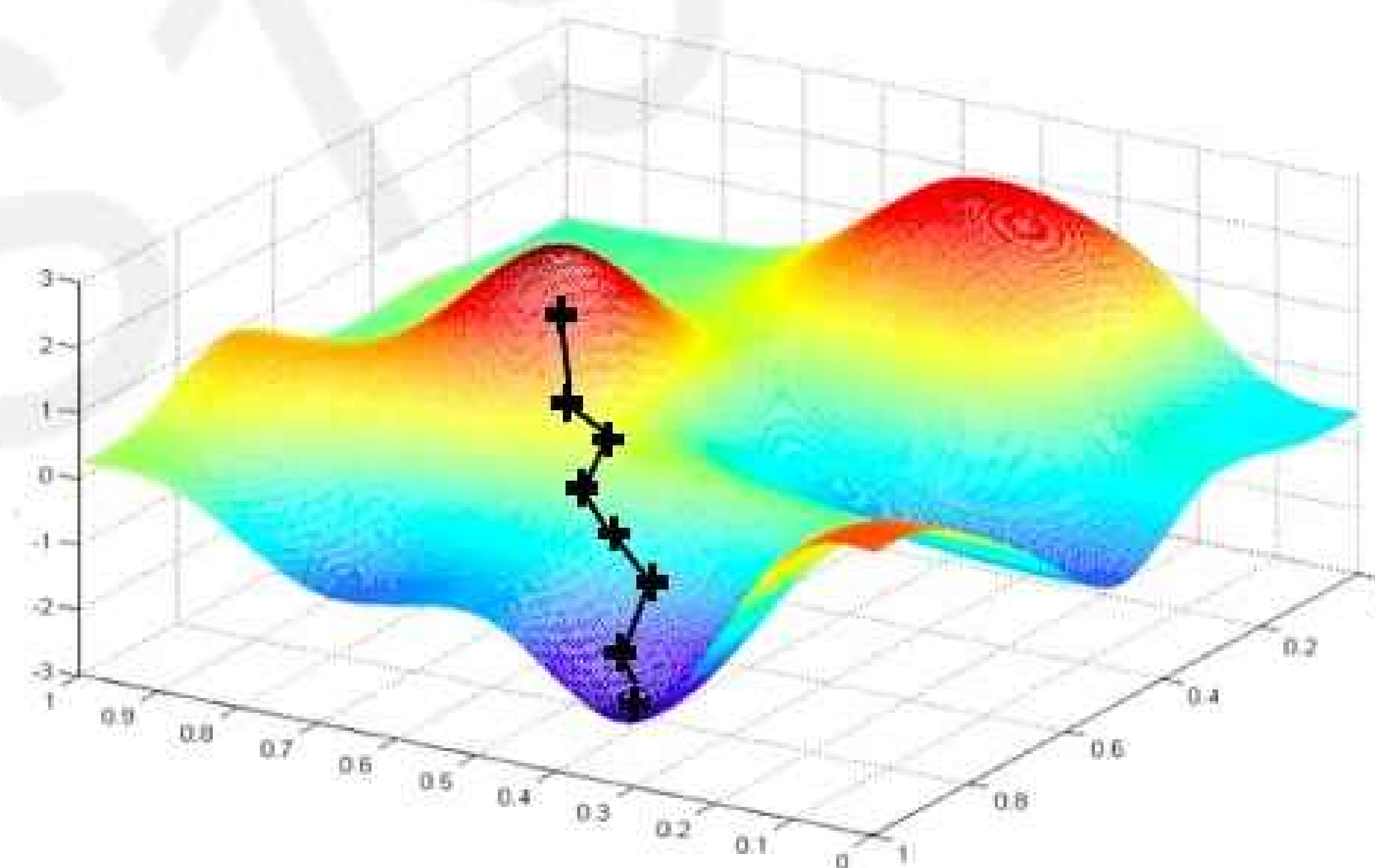
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(W)}{\partial W}$
5. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(W)}{\partial W}$
6. Return weights



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, $\frac{\partial J_i(W)}{\partial W}$
5. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(W)}{\partial W}$
6. Return weights

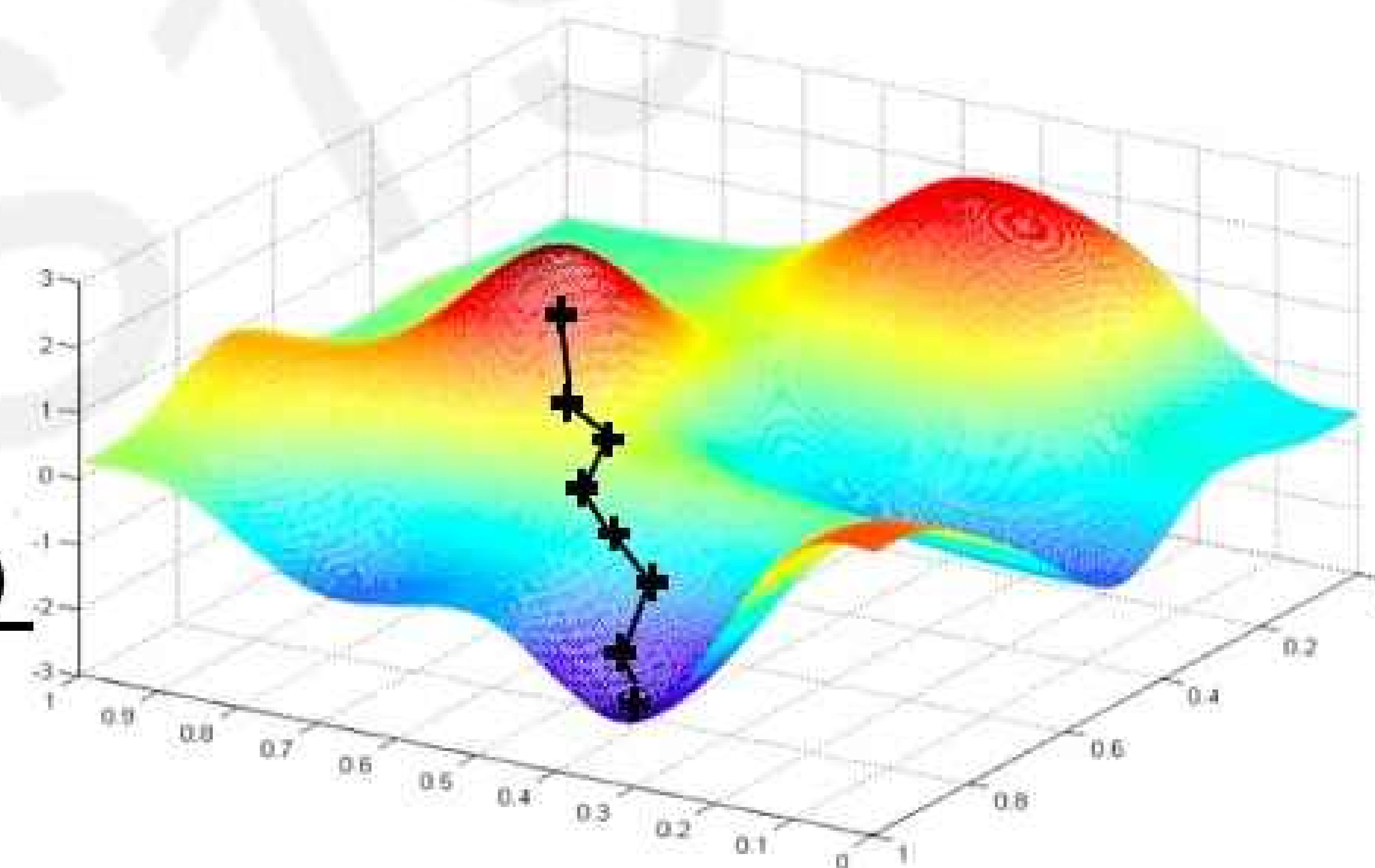


Easy to compute but
very noisy (stochastic)!

Stochastic Gradient Descent

Algorithm

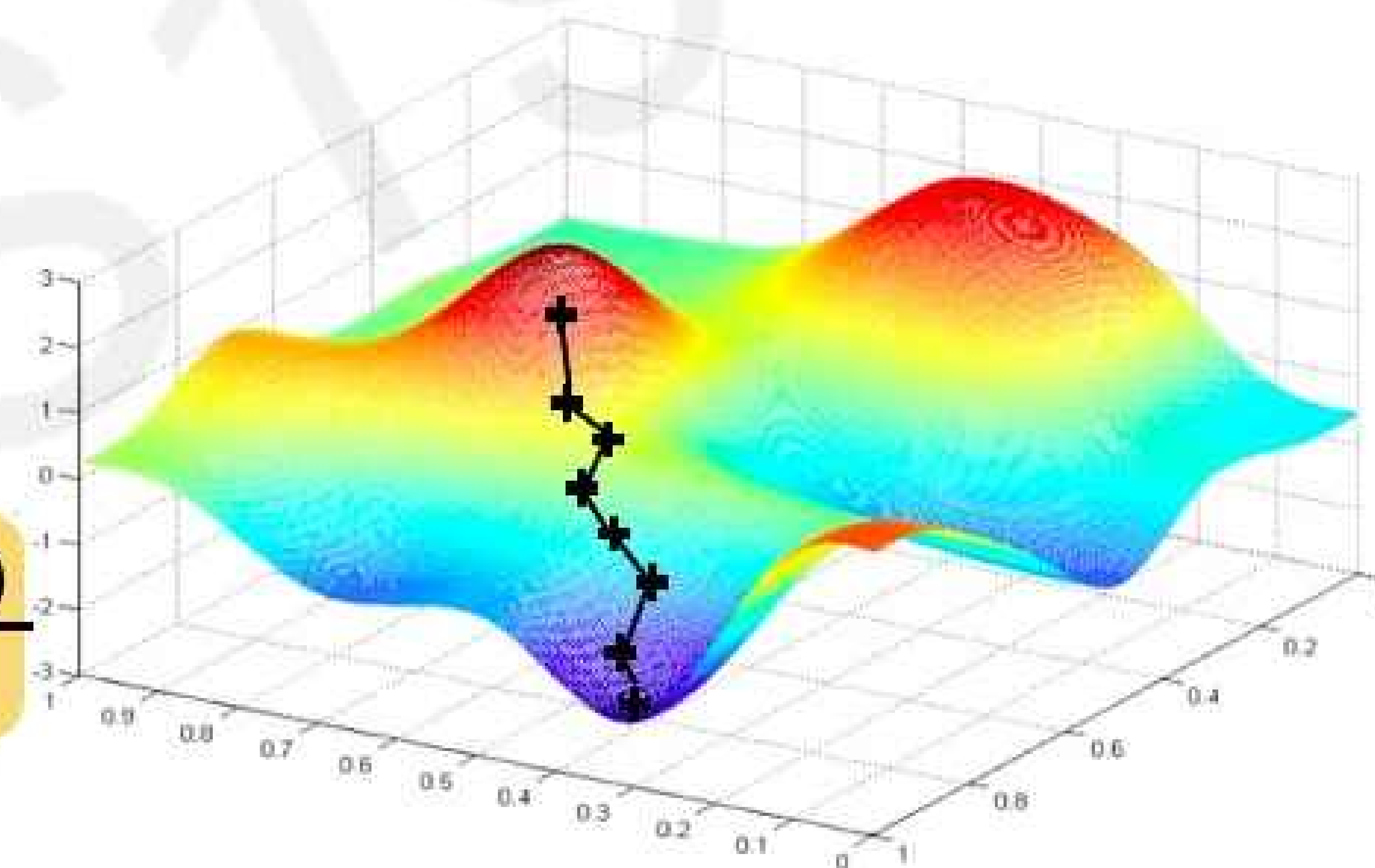
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights, $\mathbf{W}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



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

Mini-batches while training

More accurate estimation of gradient

Smother convergence
Allows for larger learning rates

Mini-batches while training

More accurate estimation of gradient

Smoother convergence

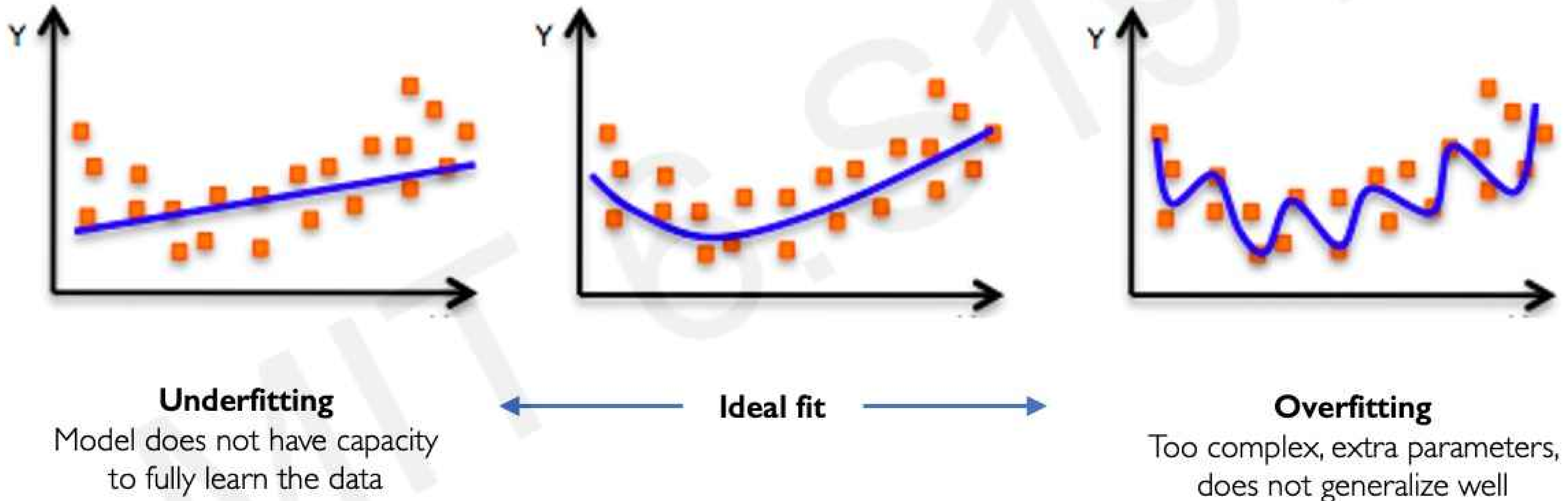
Allows for larger learning rates

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



Regularization

What is it?

Technique that constrains our optimization problem to discourage complex models

Regularization

What is it?

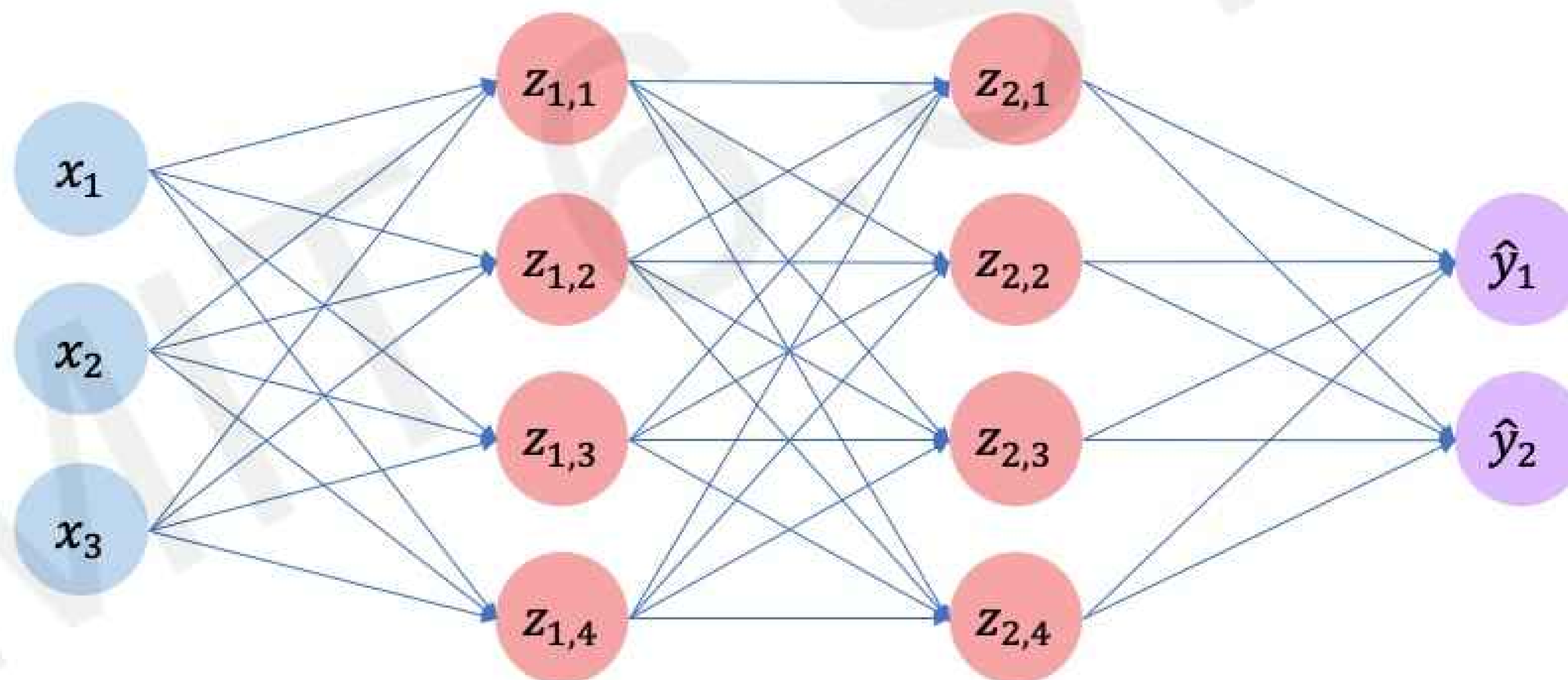
Technique that constrains our optimization problem to discourage complex models

Why do we need it?

Improve generalization of our model on unseen data

Regularization I: Dropout

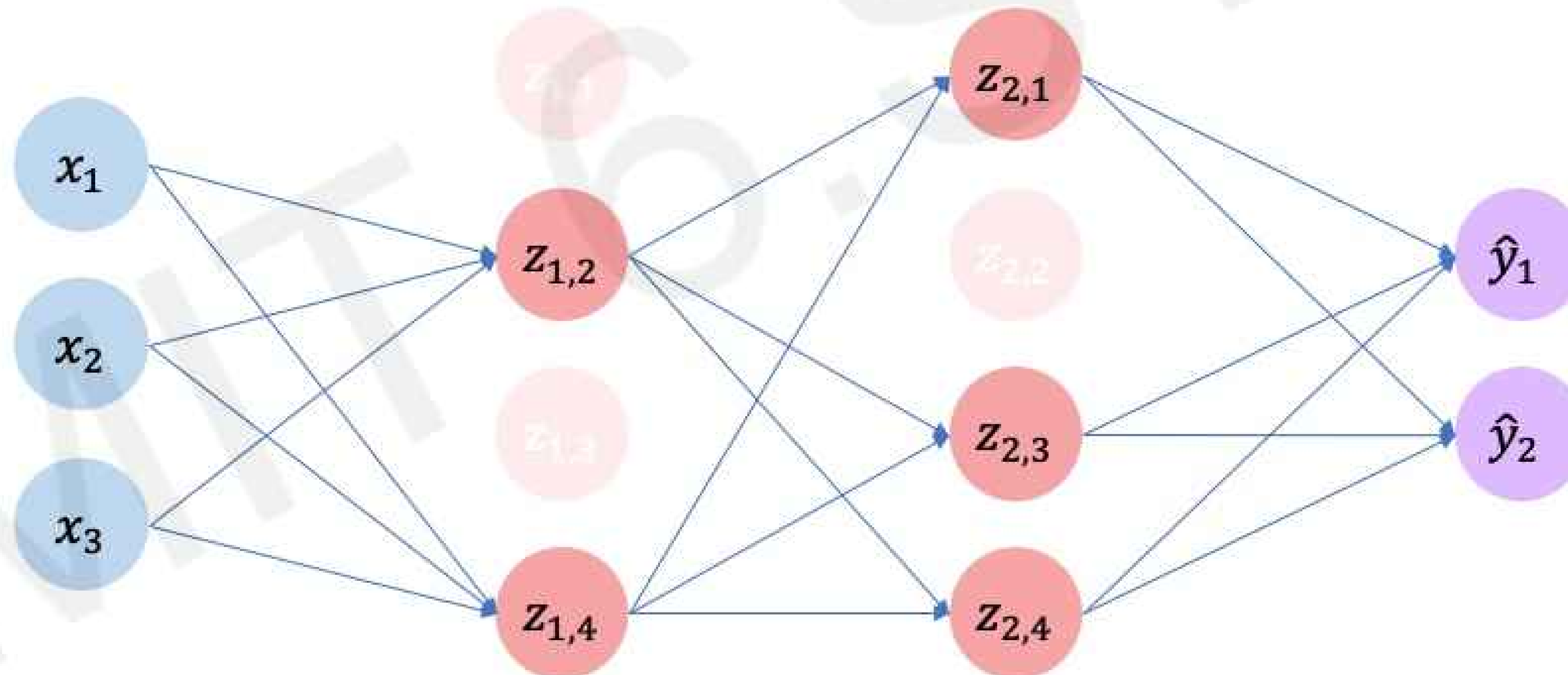
- During training, randomly set some activations to 0



Regularization I: Dropout

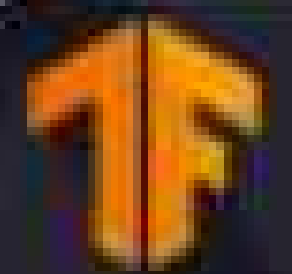
- During training, randomly set some activations to 0
 - Typically 'drop' 50% of activations in layer
 - Forces network to not rely on any 1 node

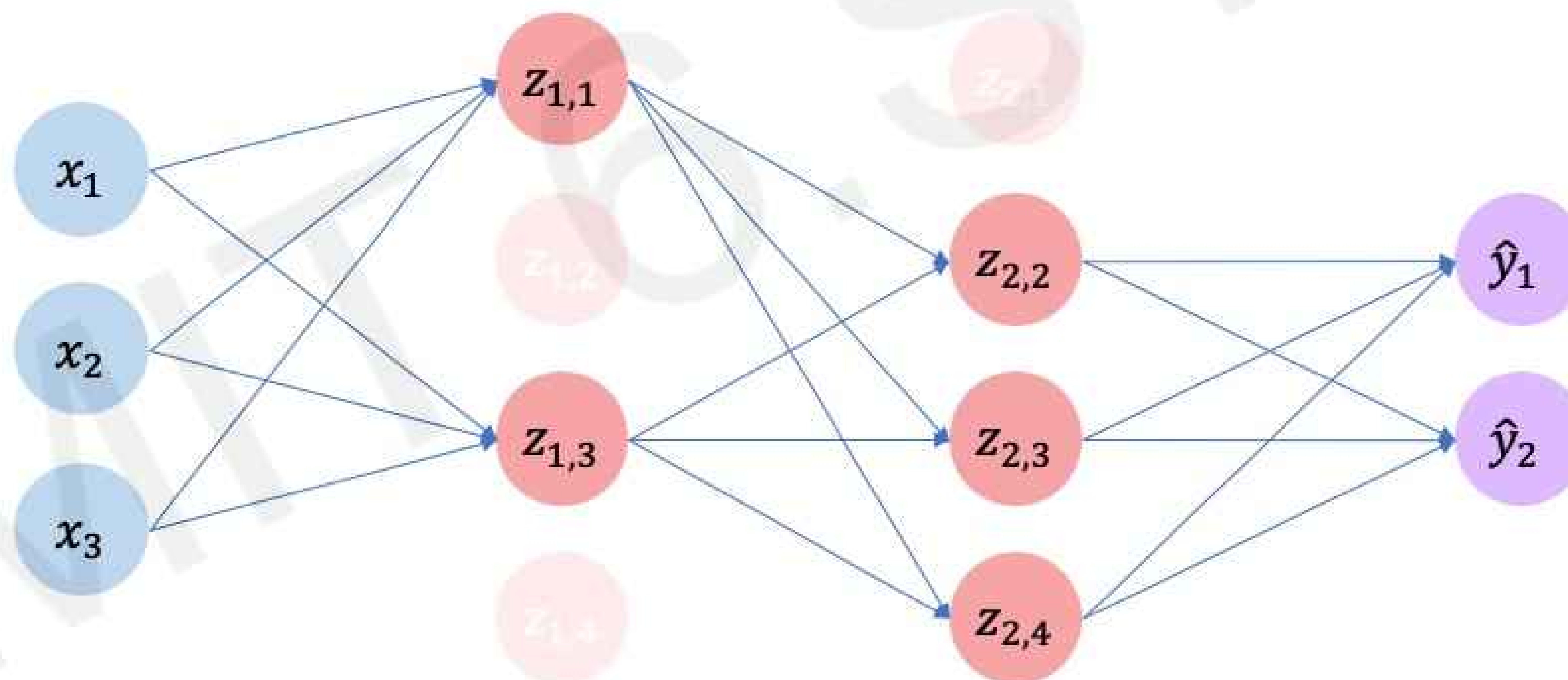
 `tf.keras.layers.Dropout(p=0.5)`
 `torch.nn.Dropout(p=0.5)`



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)`
 `torch.nn.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



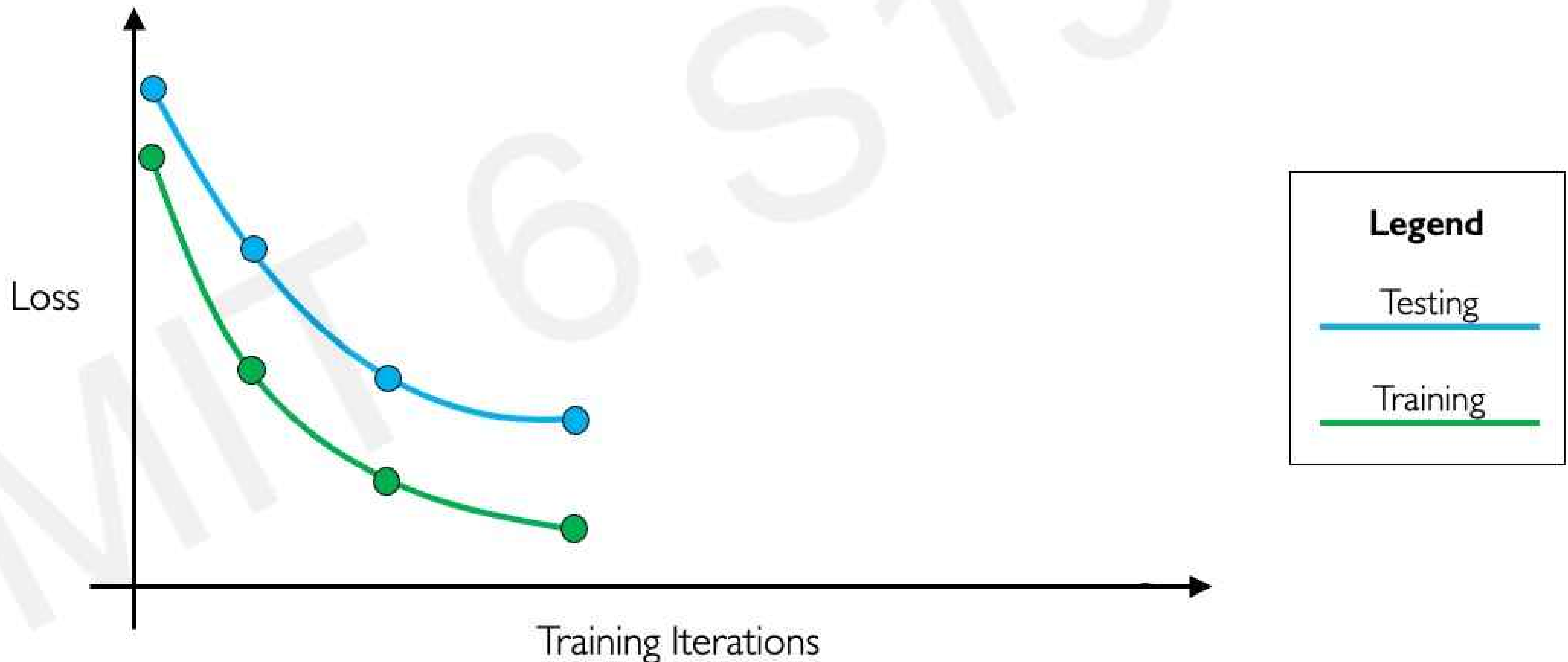
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



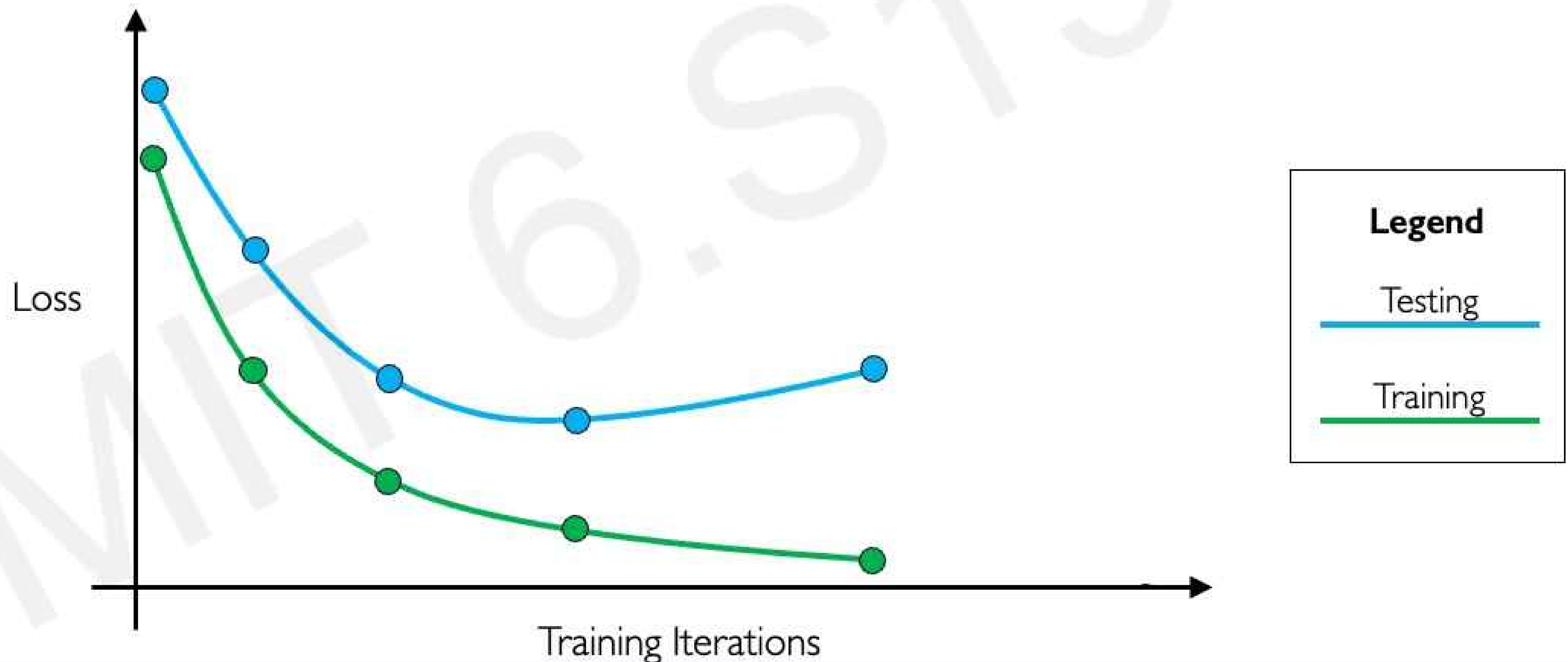
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



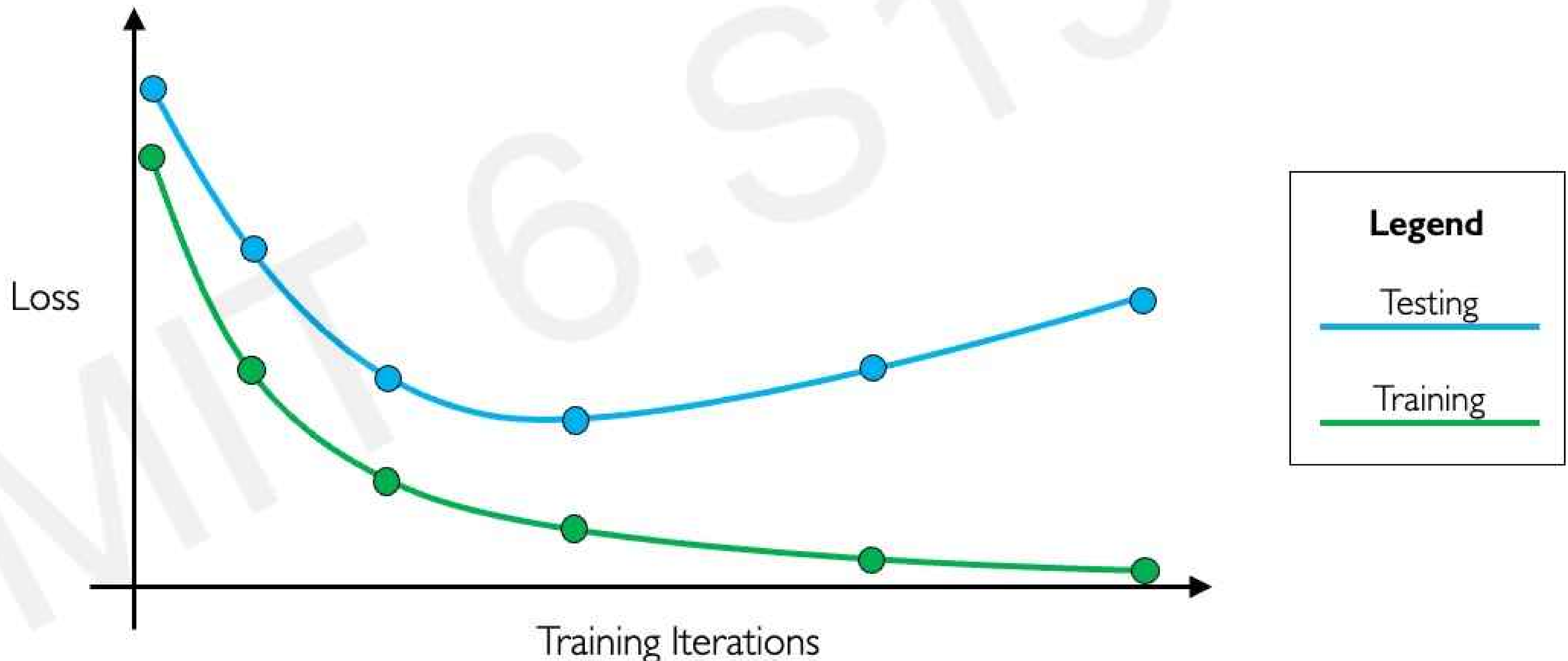
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



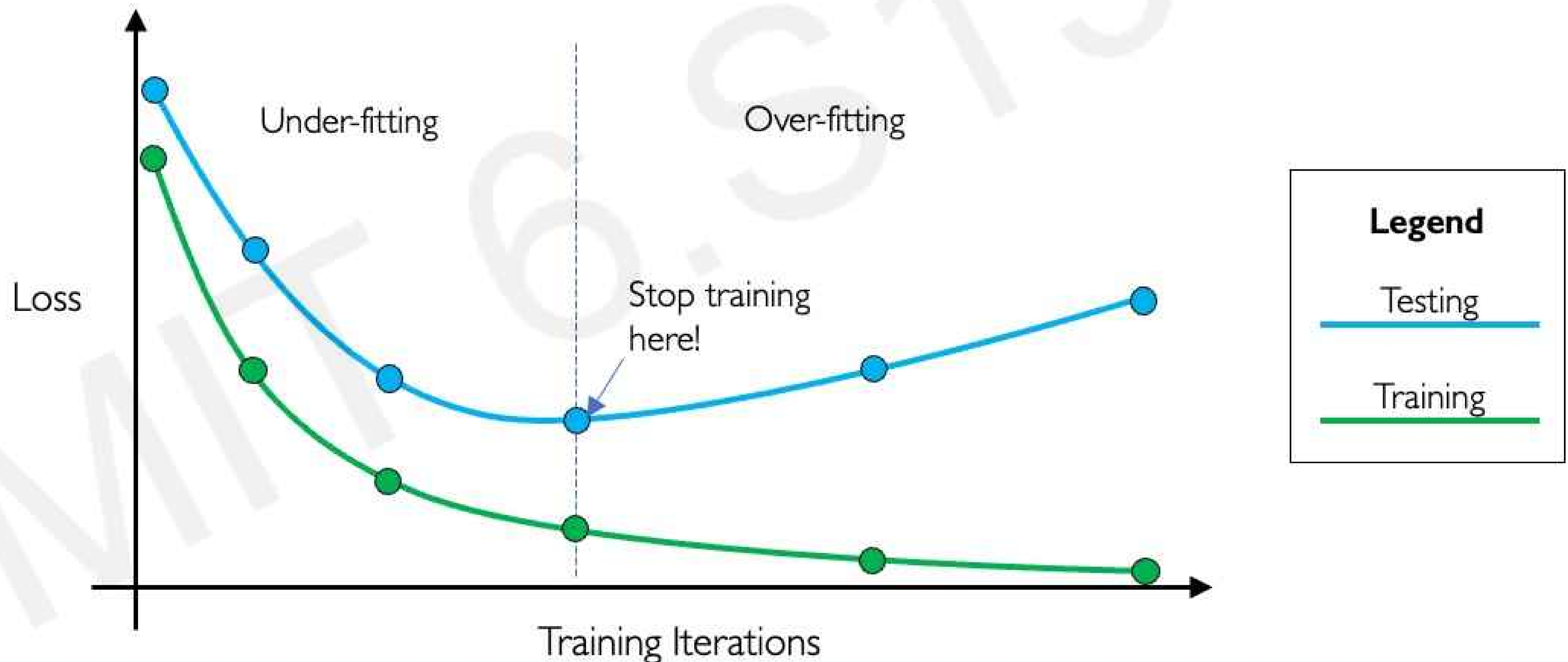
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



Regularization 2: Early Stopping

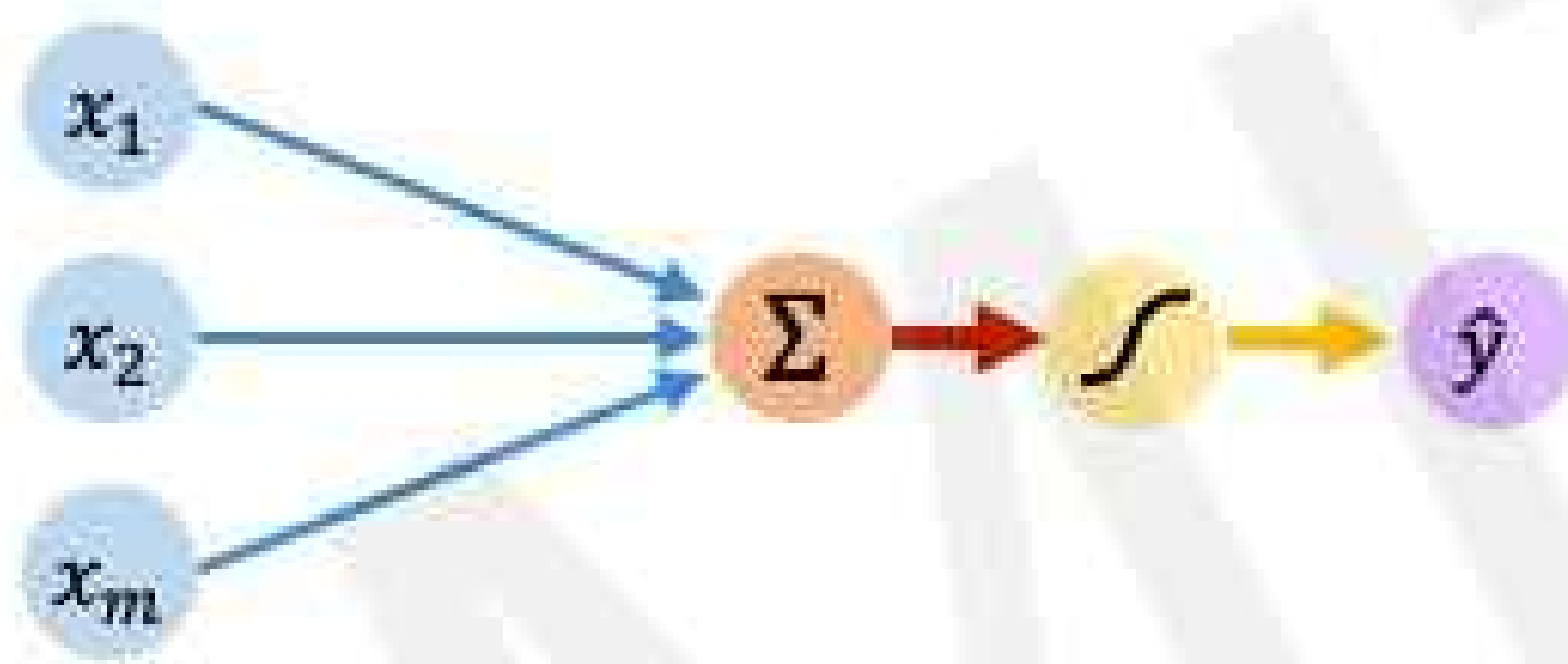
- Stop training before we have a chance to overfit



Core Foundation Review

The Perceptron

- Structural building blocks
- Nonlinear activation functions



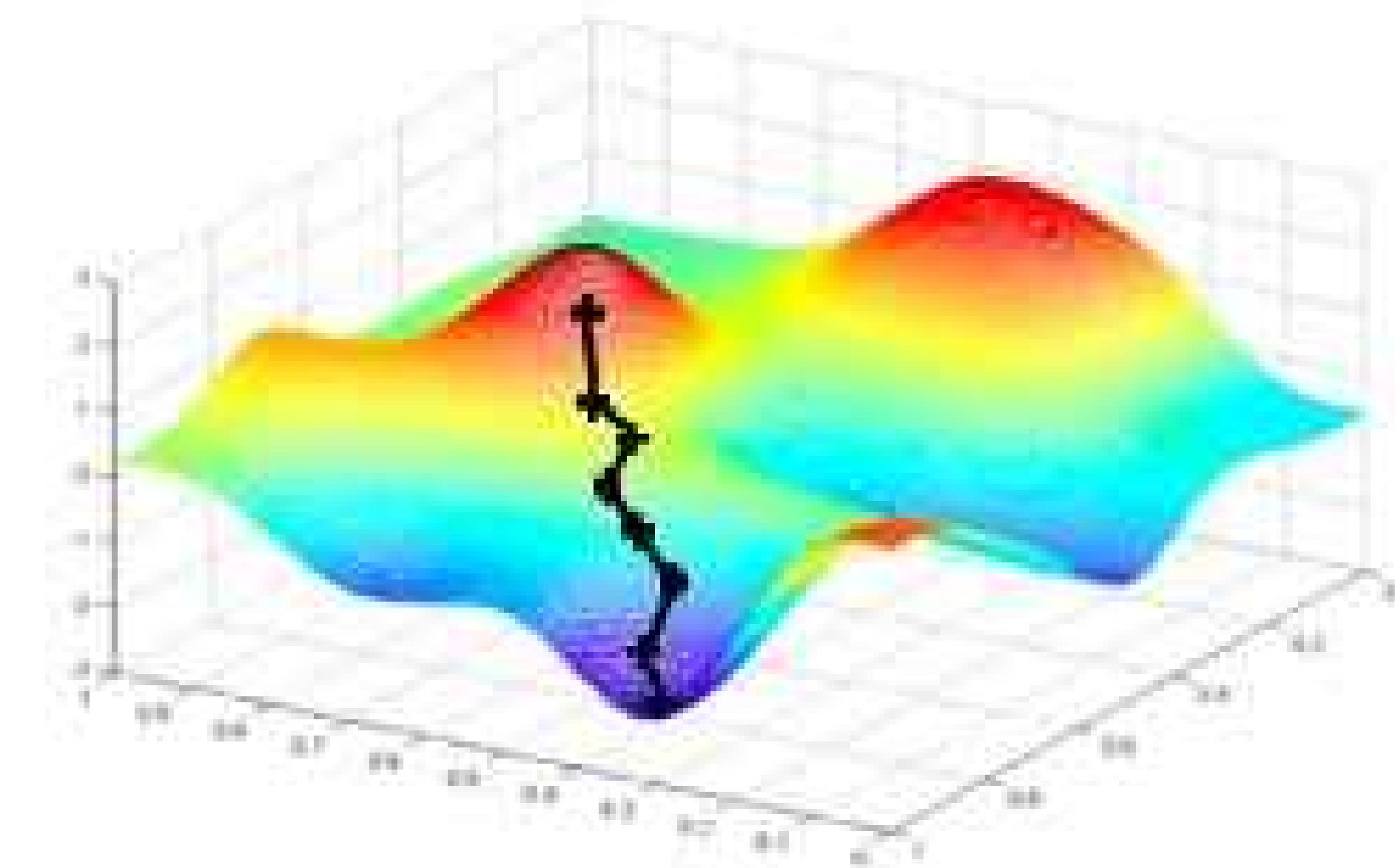
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization



MIT Introduction to Deep Learning

Lab 1: Introduction to Deep Learning in Python 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!

