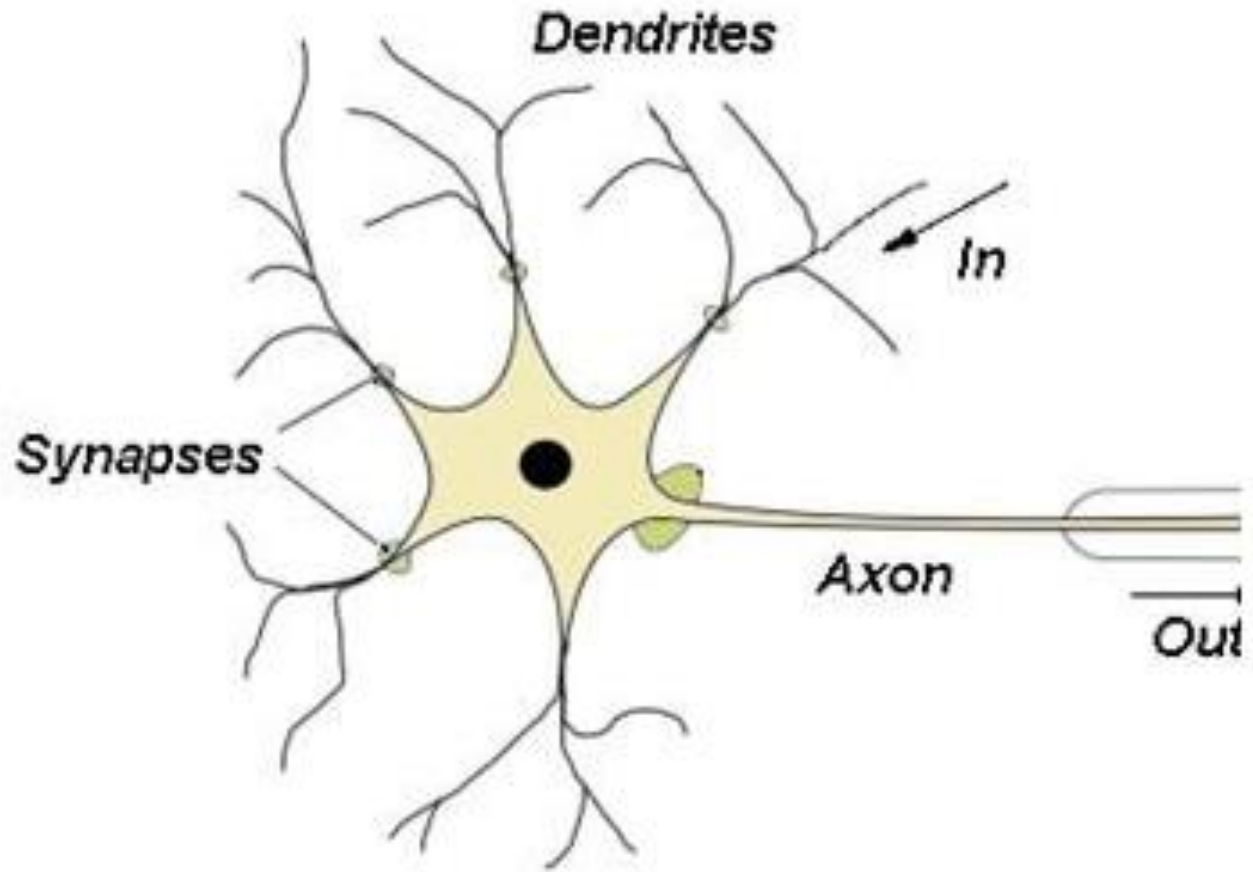
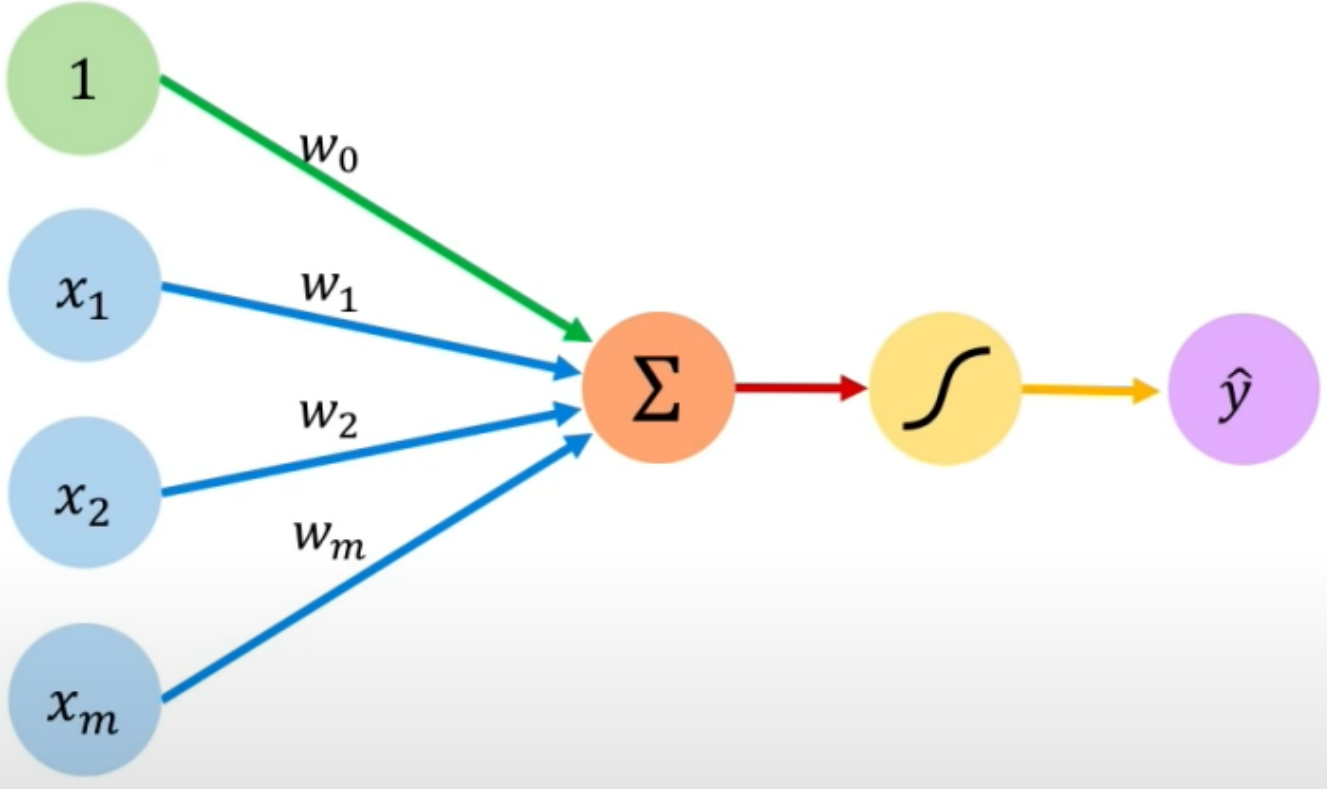
Introduction to Deep Learning

Pr. Samira Douzi

Human Neuron



The Perceptron: Forward Propagation



Inputs Weights Sum Non-Linearity Output

Output

Linear combination of inputs

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

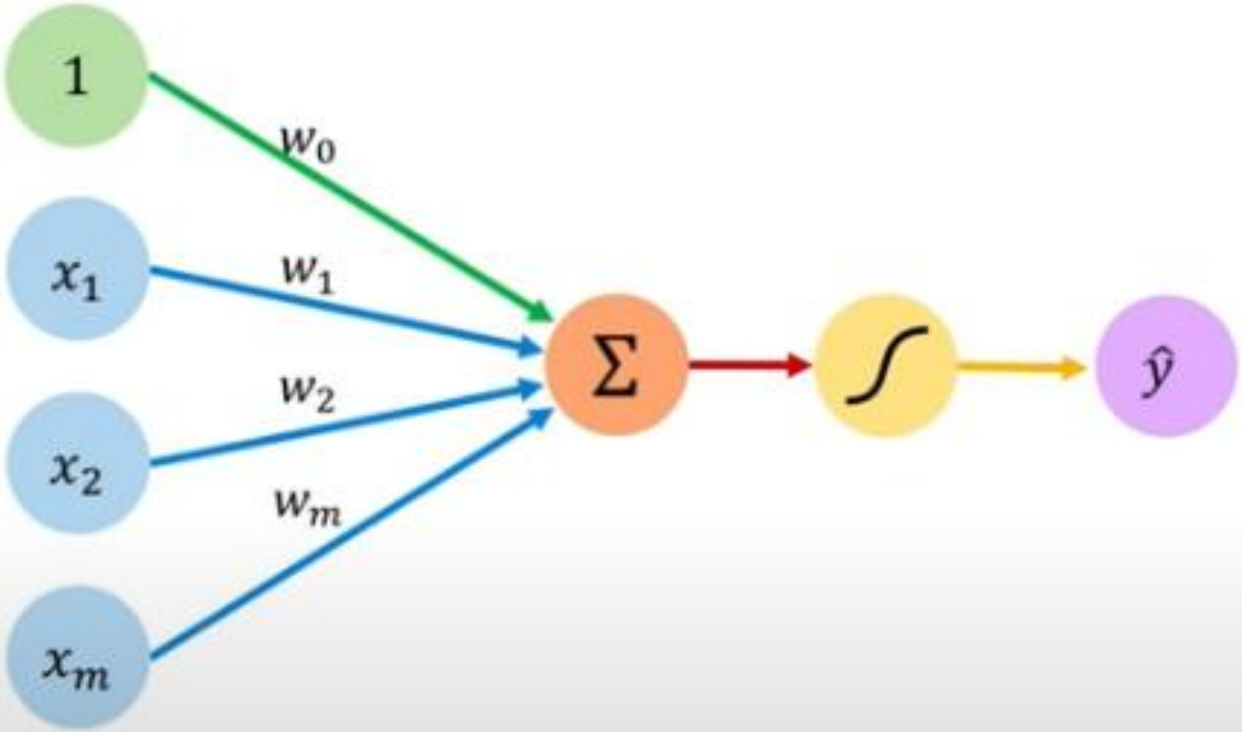
Non-linear activation function

Bias

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

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

The Perceptron: Forward Propagation



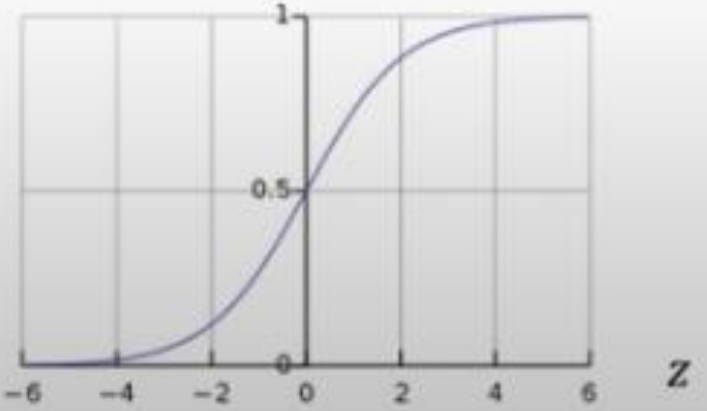
Inputs Weights Sum Non-Linearity Output

Activation Functions

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

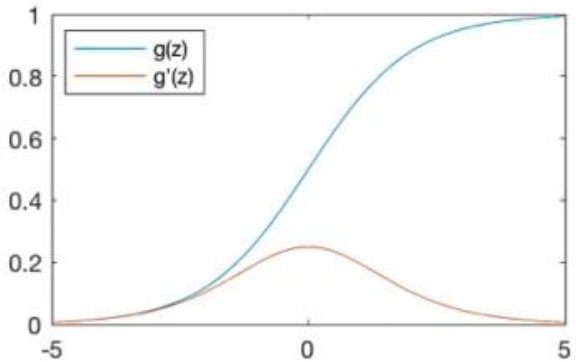
- Example: sigmoid function

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



Common Activation Functions

Sigmoid Function

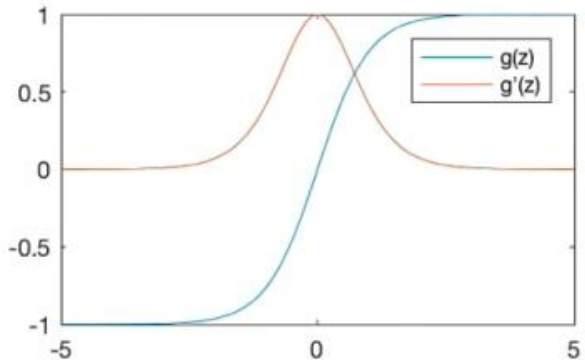


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

`tf.math.sigmoid(z)`

Hyperbolic Tangent

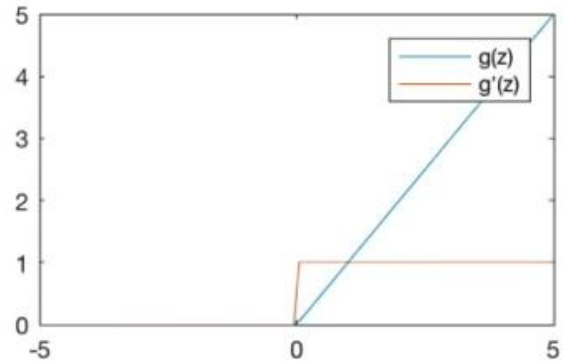


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

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

`tf.math.tanh(z)`

Rectified Linear Unit (ReLU)

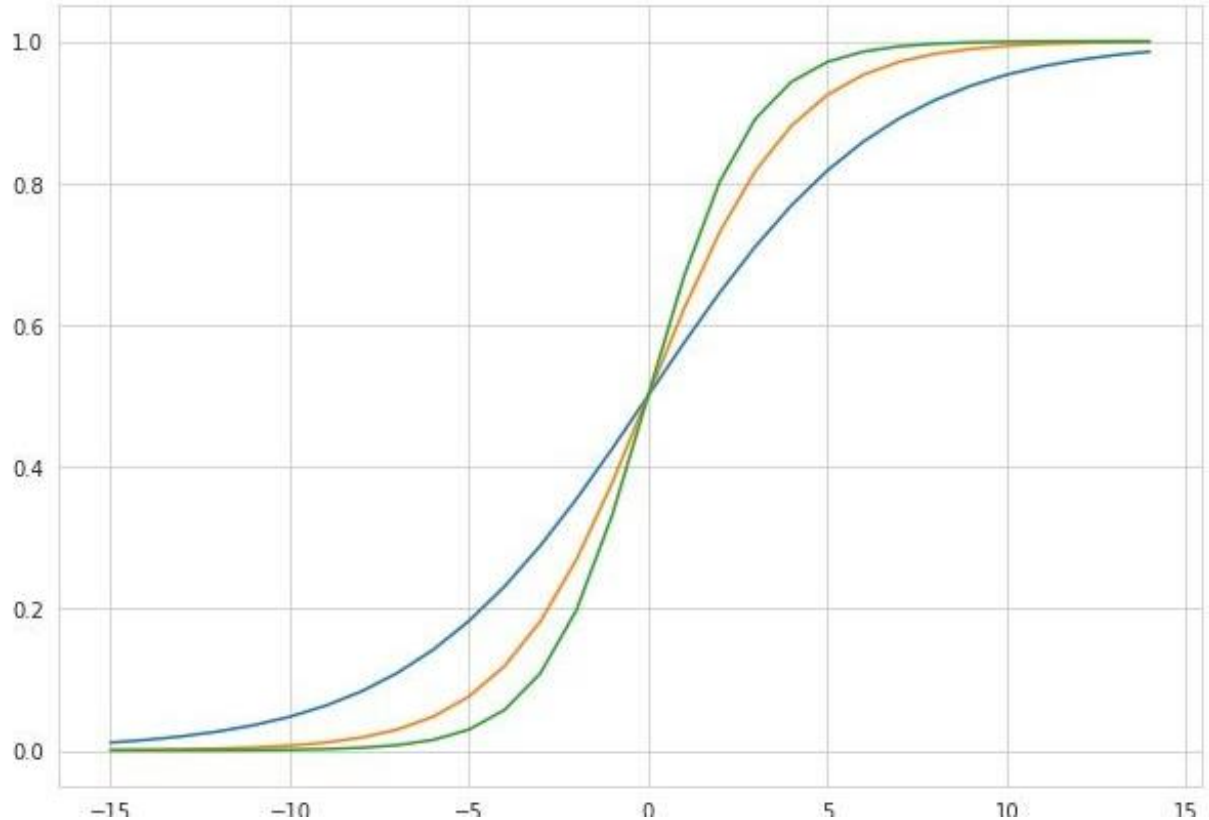


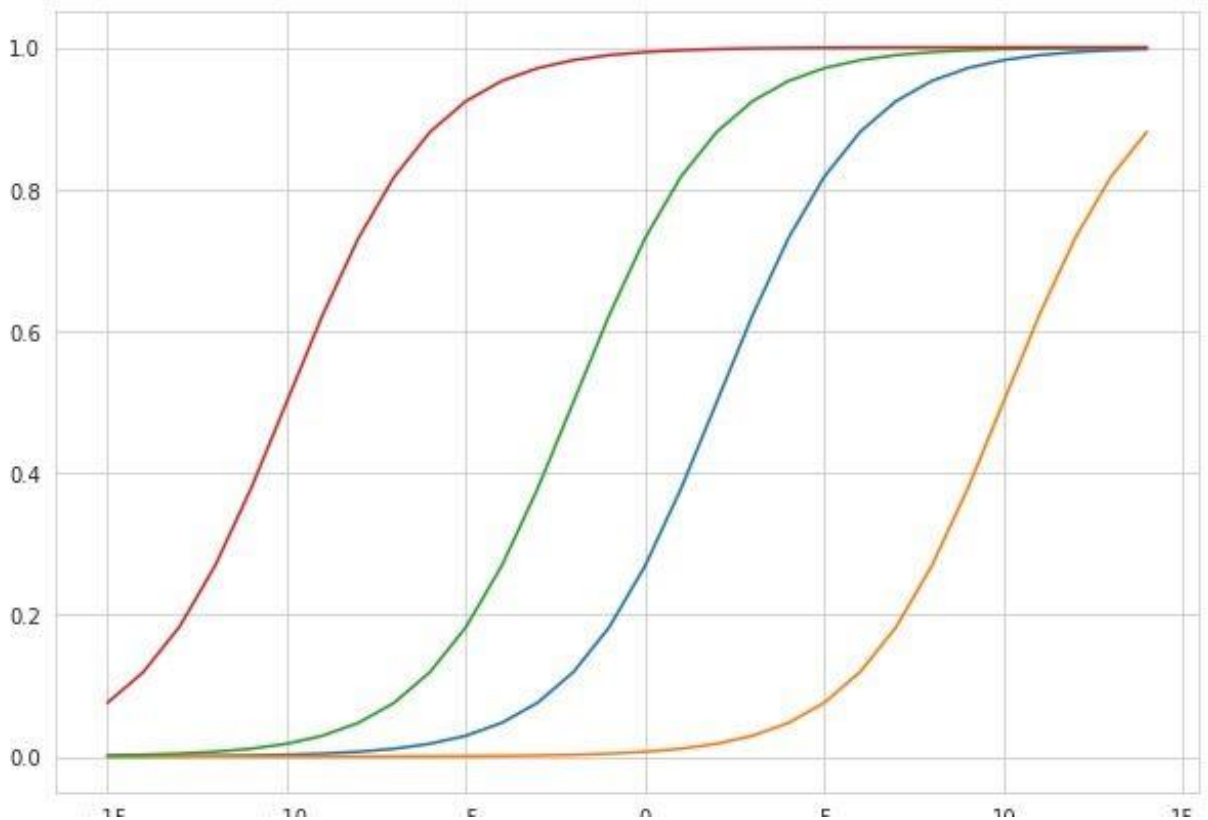
$$g(z) = \max(0, z)$$

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

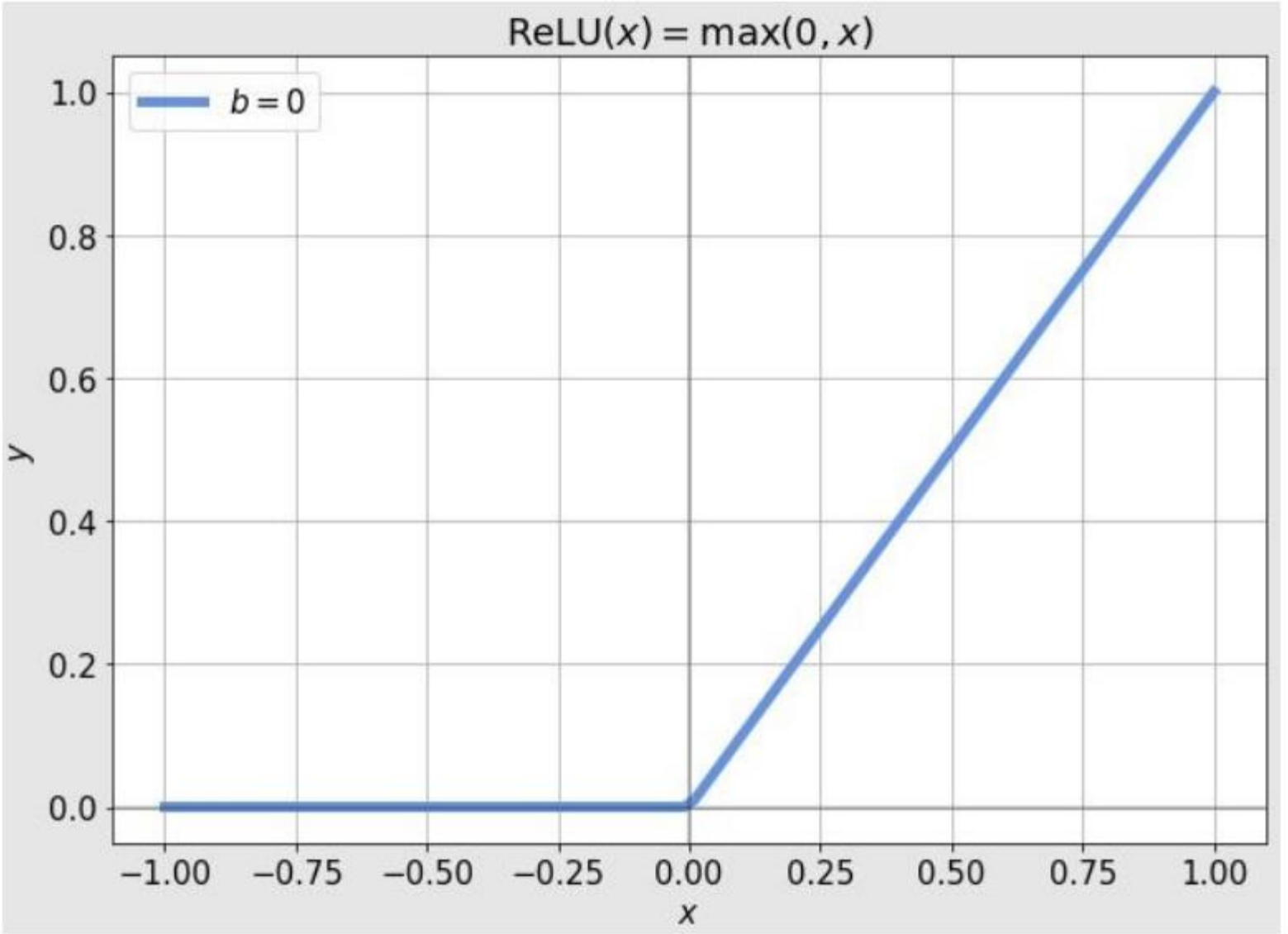
`tf.nn.relu(z)`

What Is the Necessity of Bias in Neural Networks?

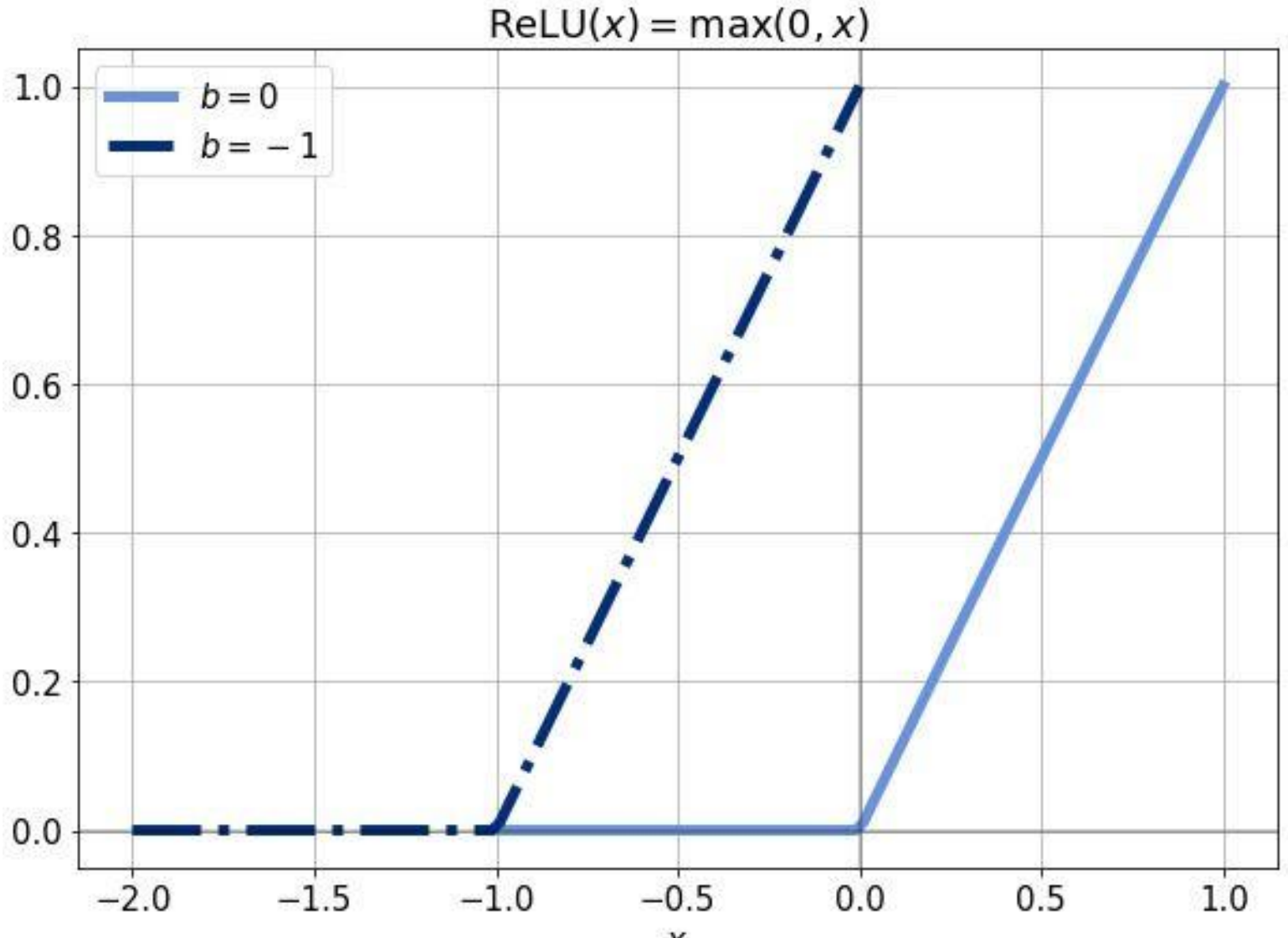




What Is the Necessity of Bias in Neural Networks?



What Is the Necessity of Bias in Neural Networks?



JAMA Internal Medicine | Special Communication

Potential Biases in Machine Learning Algorithms Using Electronic Health Record Data

Milena A. Gianfrancesco, PhD, MPH; Suzanne Tamang, PhD, MS; Jinoos Yazdany, MD, MPH; Gabriela Schmajuk, MD, MS

A promise of machine learning in health care is the avoidance of biases in diagnosis and treatment; a computer algorithm could objectively synthesize and interpret the data in the medical record. Integration of machine learning with clinical decision support tools, such as computerized alerts or diagnostic support, may offer physicians and others who provide health care targeted and timely information that can improve clinical decisions. Machine learning algorithms, however, may also be subject to biases. The biases include those related to missing data and patients not identified by algorithms, sample size and underestimation, and misclassification and measurement error. There is concern that biases and deficiencies in the data used by machine learning algorithms may contribute to socioeconomic disparities in health care. This Special Communication outlines the potential biases that may be introduced into machine learning-based clinical decision support tools that use electronic health record data and proposes potential solutions to the problems of overreliance on automation, algorithms based on biased data, and algorithms that do not provide information that is clinically meaningful. Existing health care disparities should not be amplified by thoughtless or excessive reliance on machines.

JAMA Intern Med. doi:10.1001/jamainternmed.2018.3763
Published online August 20, 2018.

Author Affiliations: Division of Rheumatology, Department of Medicine, University of California, San Francisco (Gianfrancesco, Yazdany, Schmajuk); Center for Population Health Sciences, Stanford University, Palo Alto, California (Tamang); Veterans Affairs Medical Center, San Francisco, California (Schmajuk).

Corresponding Author: Milena A. Gianfrancesco, PhD, MPH, Division of Rheumatology, Department of Medicine, University of California, San Francisco, 513 Parnassus Ave, San Francisco, CA 94143 (milena.gianfrancesco@ucsf.edu).

A promise of machine learning in health care is the avoidance of biases in diagnosis and treatment. Practitioners can have bias in their diagnostic or therapeutic decision making that might be circumvented if a computer algorithm could objectively synthesize and interpret the data in the medical record and offer clinical decision support to aid or guide diagnosis and treatment. Although all statistical models exist along a continuum of fully human-guided vs fully machine-guided data analyses,[1] machine learning algorithms in general tend to rely less on human specification (ie, defining a set of variables to be included a priori) and instead allow the algorithm to decide which variables are important to include in the model. Classic machine learning algorithms involve techniques such as decision trees and association rule learning, including market basket analysis (ie, customers who bought Y also bought Z). Deep learning, a subset of machine learning that includes neural networks, attempts to model brain architecture by using multiple, overlaying models. Machine learning has generated substantial advances in medical imaging, for example, through improved detection of colonic polyps, cerebral microbleeding, and diabetic retinopathy.[2] Predictive modeling with electronic health records using deep learning can accurately predict in-hospital mortality, 30-day unplanned readmission, prolonged length of stay, and final discharge diagnoses.[3] Integration of machine learning with clinical decision support tools, such as computerized alerts or diagnostic support, may offer physicians and others who provide health care with targeted and timely information that can improve clinical decisions.

However, machine learning as applied to clinical decision support may be subject to important biases. Outside medicine, there is concern that machine learning algorithms used in the legal and judicial systems, advertisements, computer vision, and language models could make social or economic disparities worse.[4-6] For example, word-embedding models, which are used in website searches and machine translation, reflect societal biases, associating searches for jobs that included the terms *female* and *woman* with suggestions for openings in the arts and humanities professions, whereas searches that included the terms *male* and *man* suggested math and engineering occupations.[7]

As the use of machine learning in health care increases, the underlying data sources and methods of data collection should be examined. Could these algorithms worsen or perpetuate existing health inequalities? All types of observational studies and traditional statistical modeling may be biased; however, the data that are available for analysis in health care have the potential to affect clinical decision support tools that are based on machine learning in unexpected ways. Biases that may be introduced through reliance on data derived from the electronic health record are listed in the **Table**.

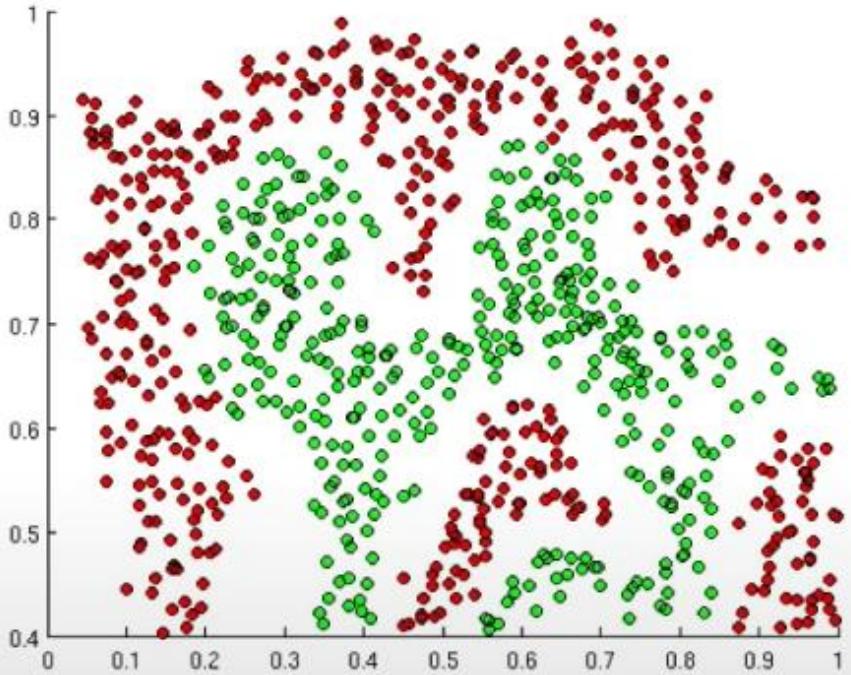
Missing Data and Patients Not Identified by Algorithms

One of the advantages of machine learning algorithms is that the computer can use all the data available in the electronic health record, which may be cumbersome or impossible for a person to review in its entirety. Conversely, these algorithms will use only the data available in the electronic health record or data derived from communicating sources (eg, sensor data, patient-reported data) that may

- A 2019 research paper on ‘Potential Biases in Machine Learning Algorithms Using Electronic Health Record Data’ discusses how bias can impact deep learning algorithms used in the field of healthcare.

Importance of Activation Functions

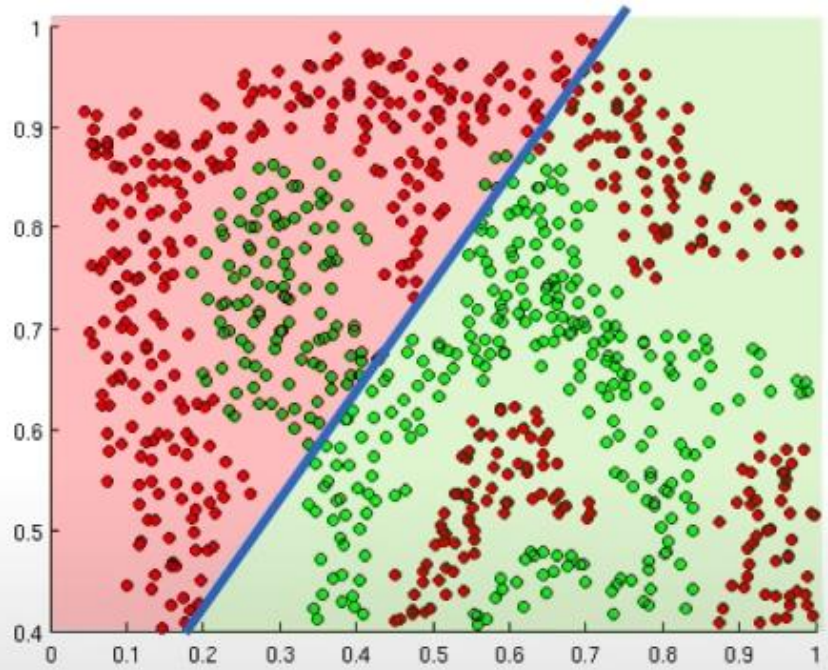
The purpose of activation functions is to ***introduce non-linearities*** *into the network*



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

Importance of Activation Functions

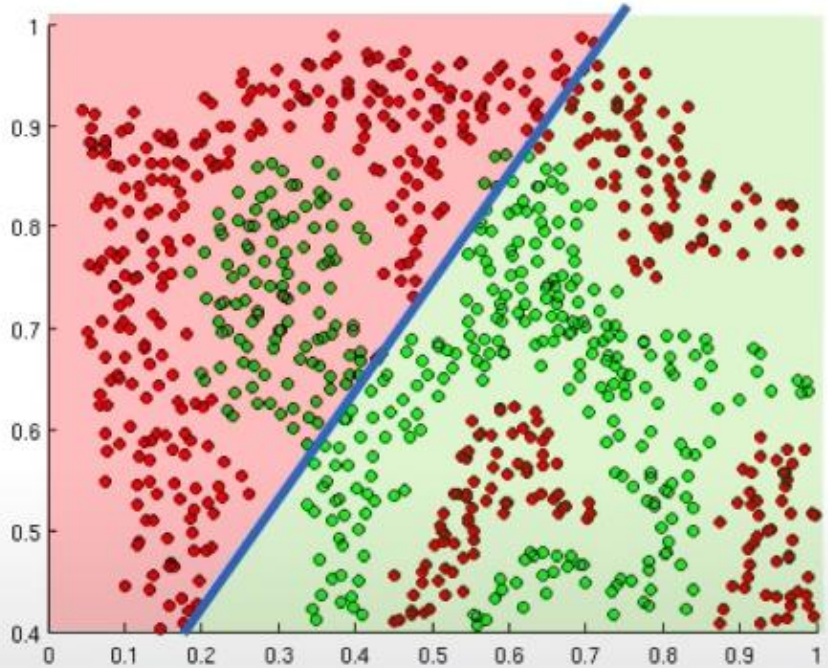
The purpose of activation functions is to ***introduce non-linearities*** *into the network*



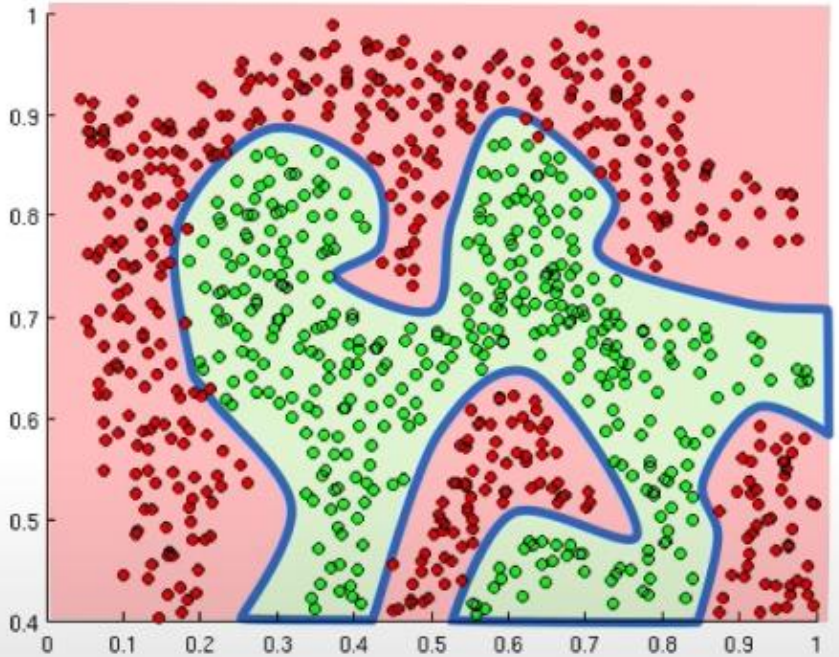
Linear activation functions produce linear decisions no matter the network size

Importance of Activation Functions

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

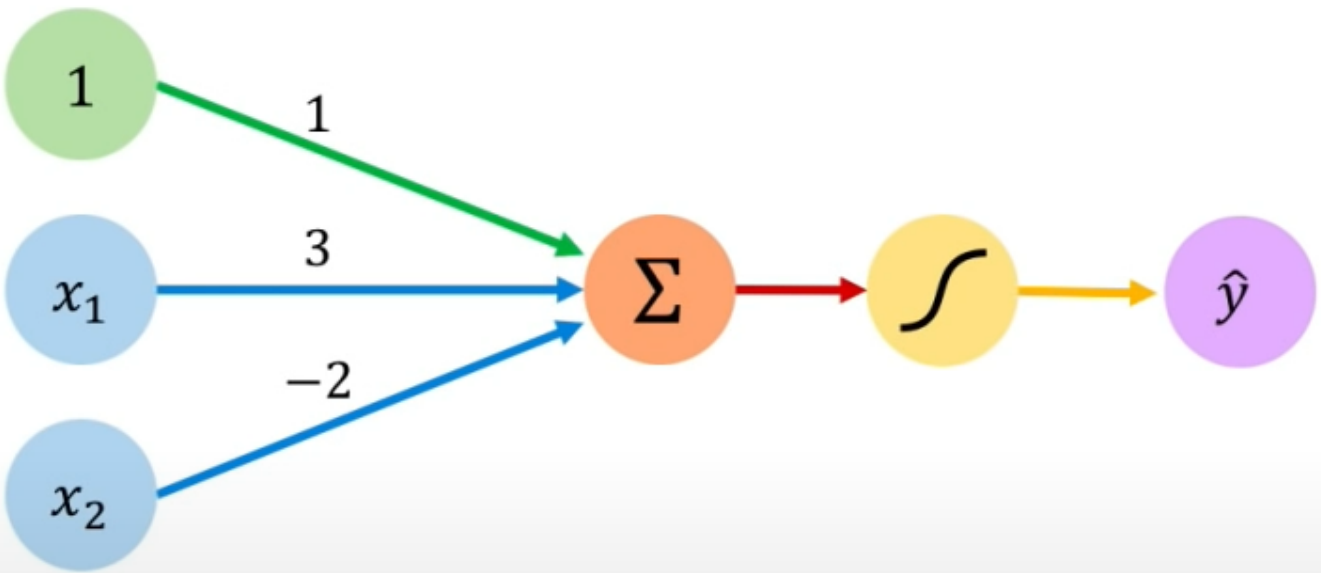


Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example



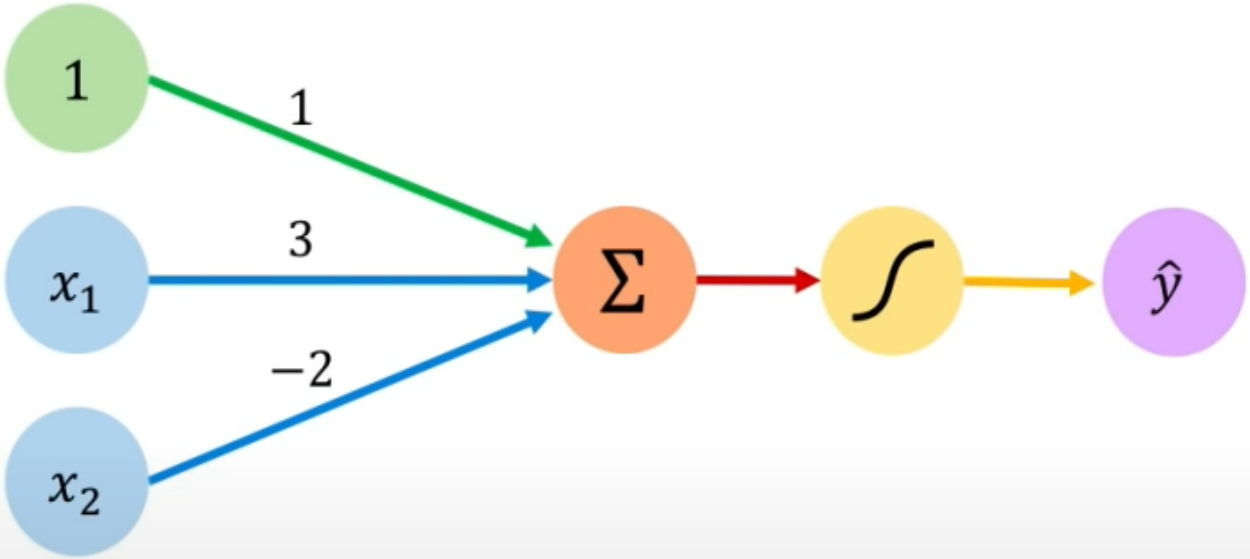
We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

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

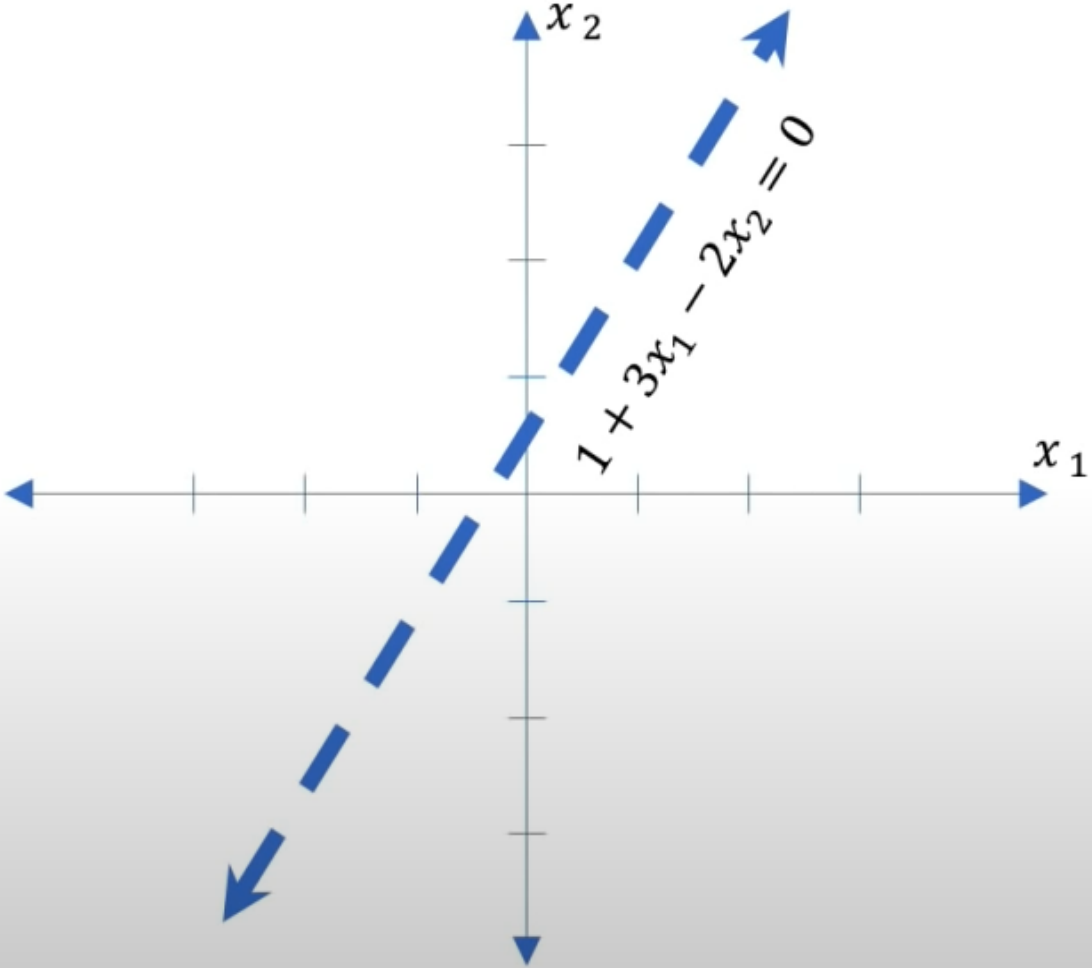
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$

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

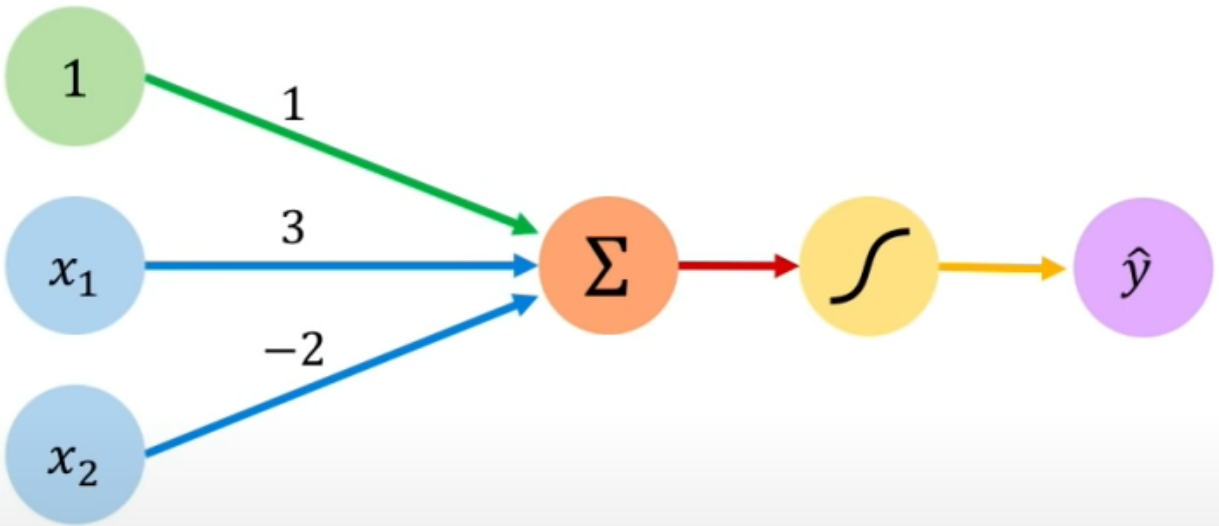
The Perceptron: Example



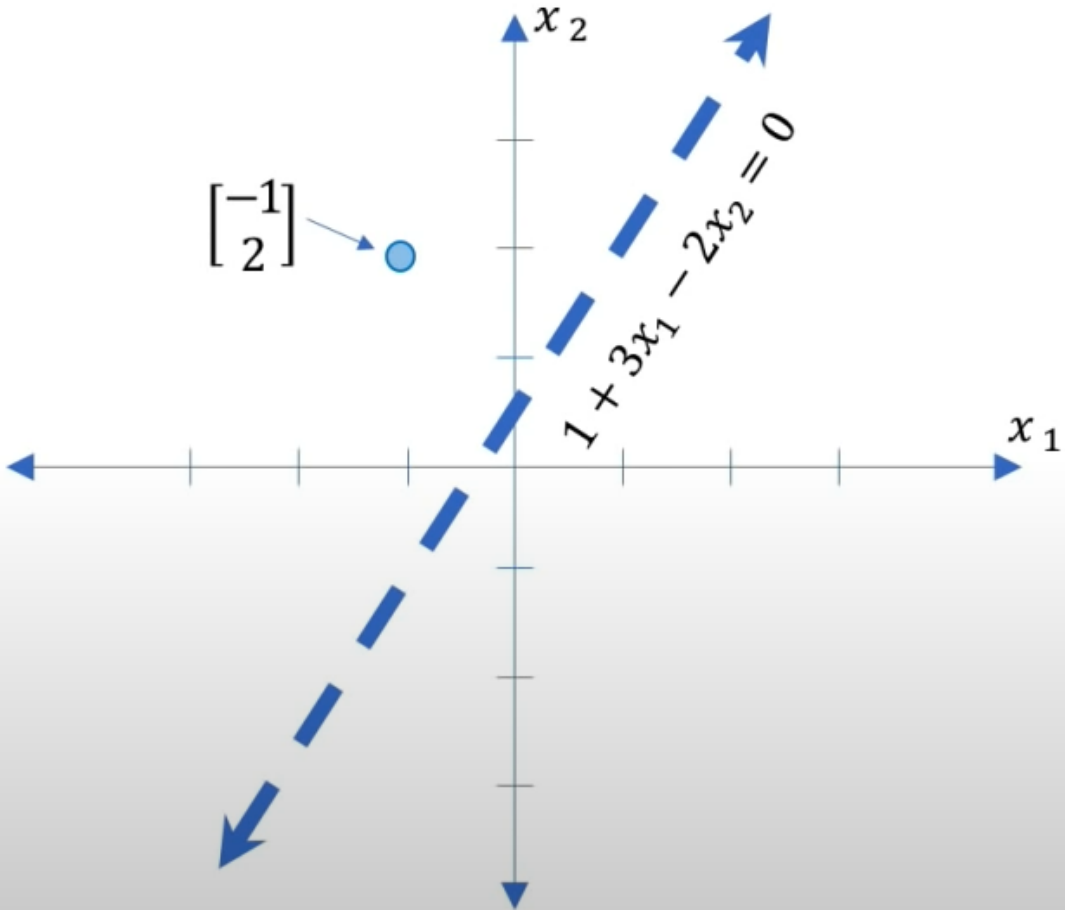
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



The Perceptron: Example



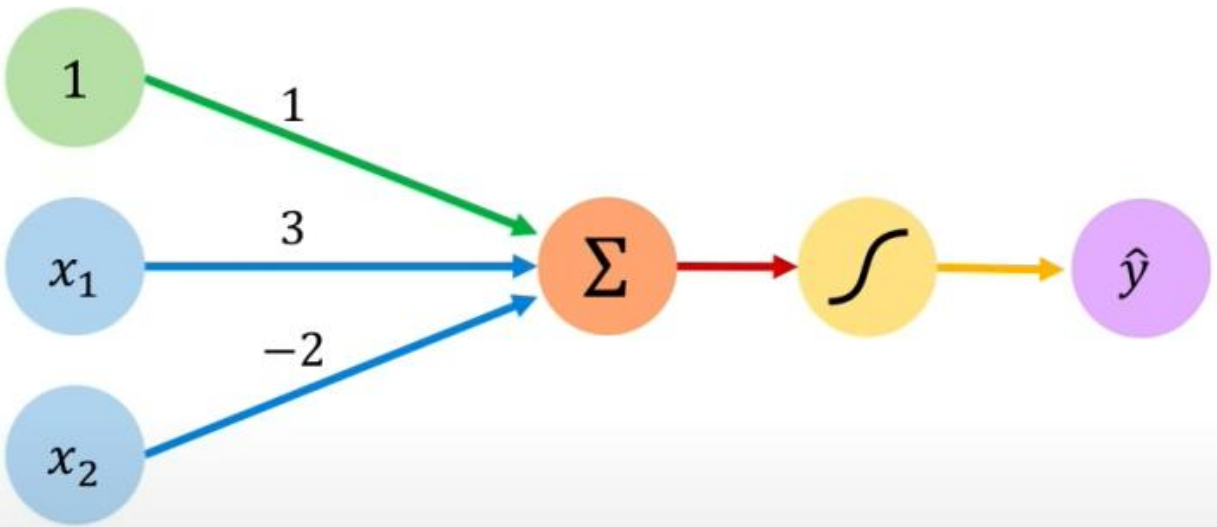
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



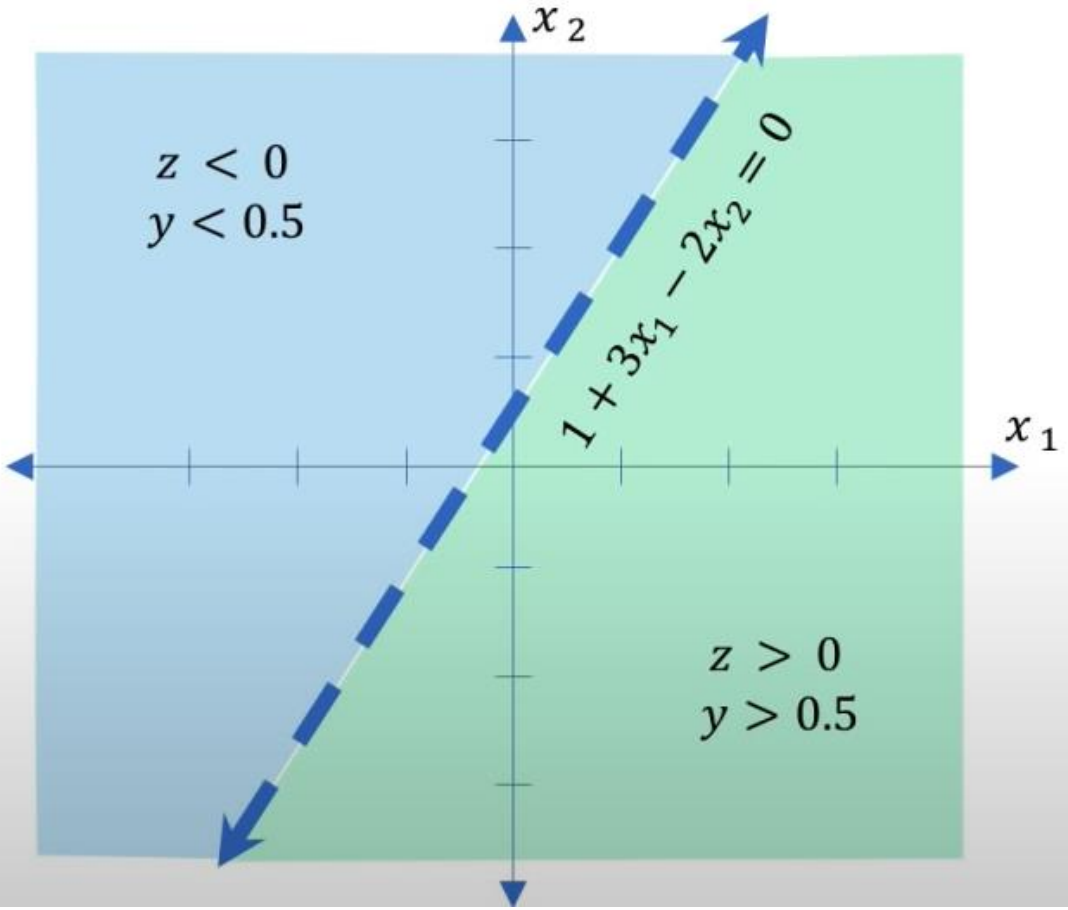
Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

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

The Perceptron: Example



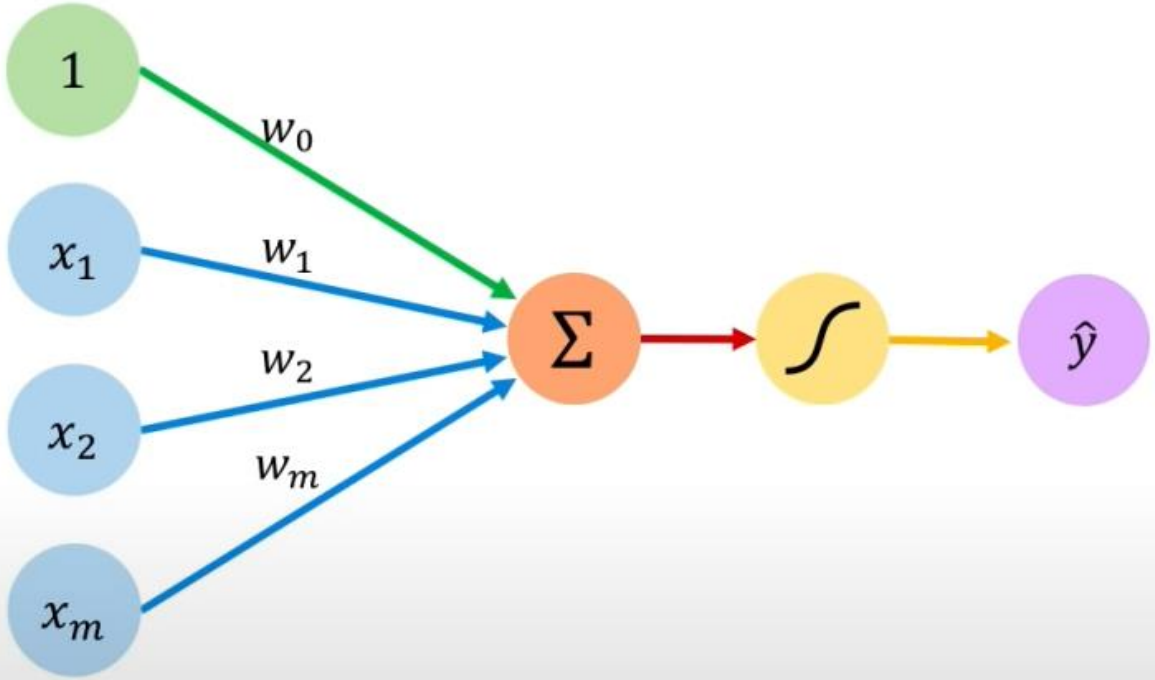
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



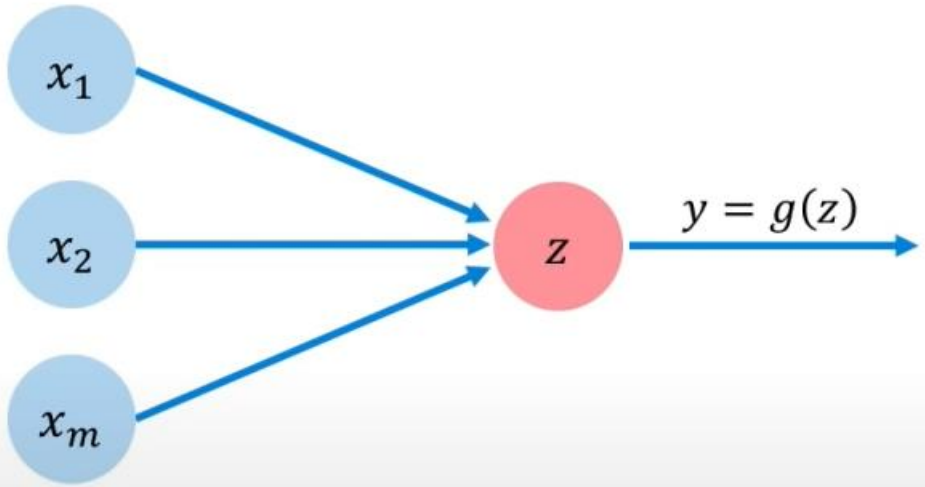
Building Neural Networks with Perceptrons

The Perceptron: Simplified

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



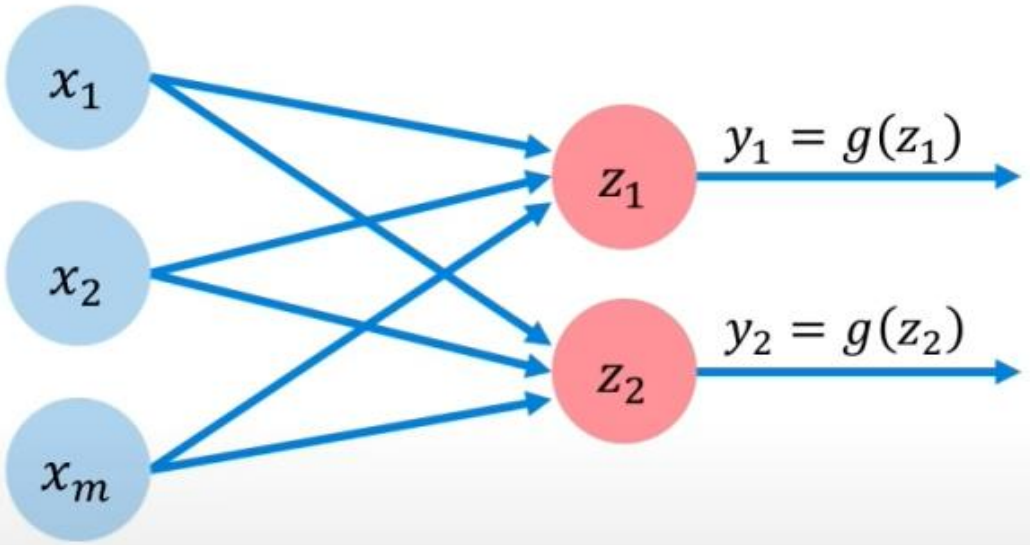
The Perceptron: Simplified



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

Multi Output Perceptron

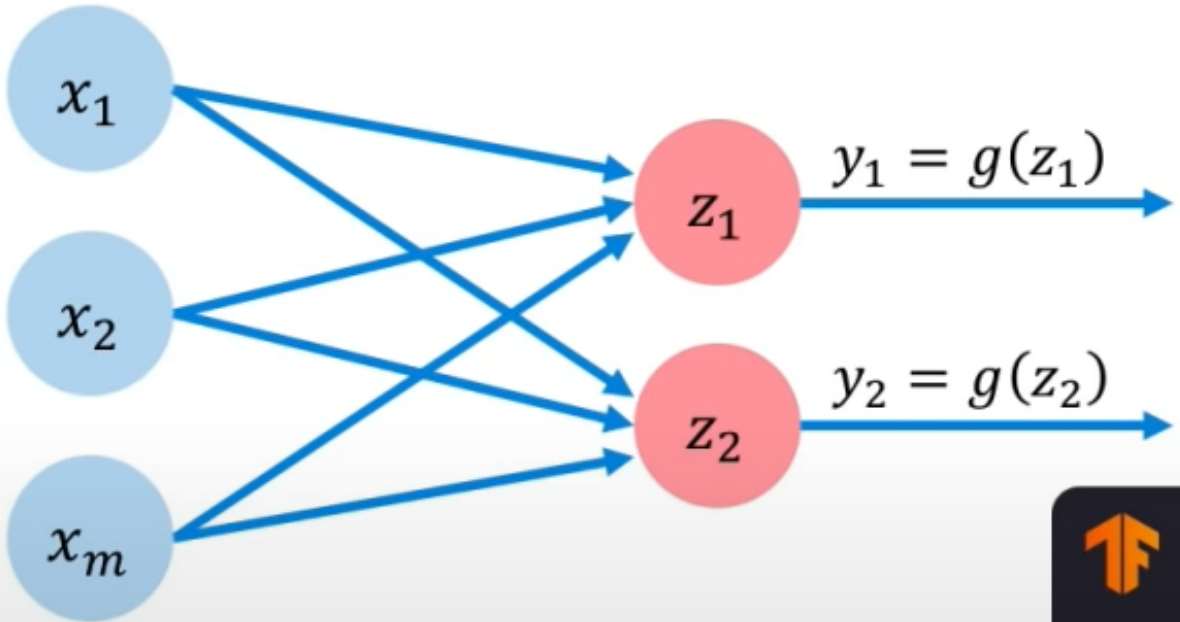
Because all inputs are densely connected to all outputs, these layers are called **Dense** layers



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

Dense layer from scratch

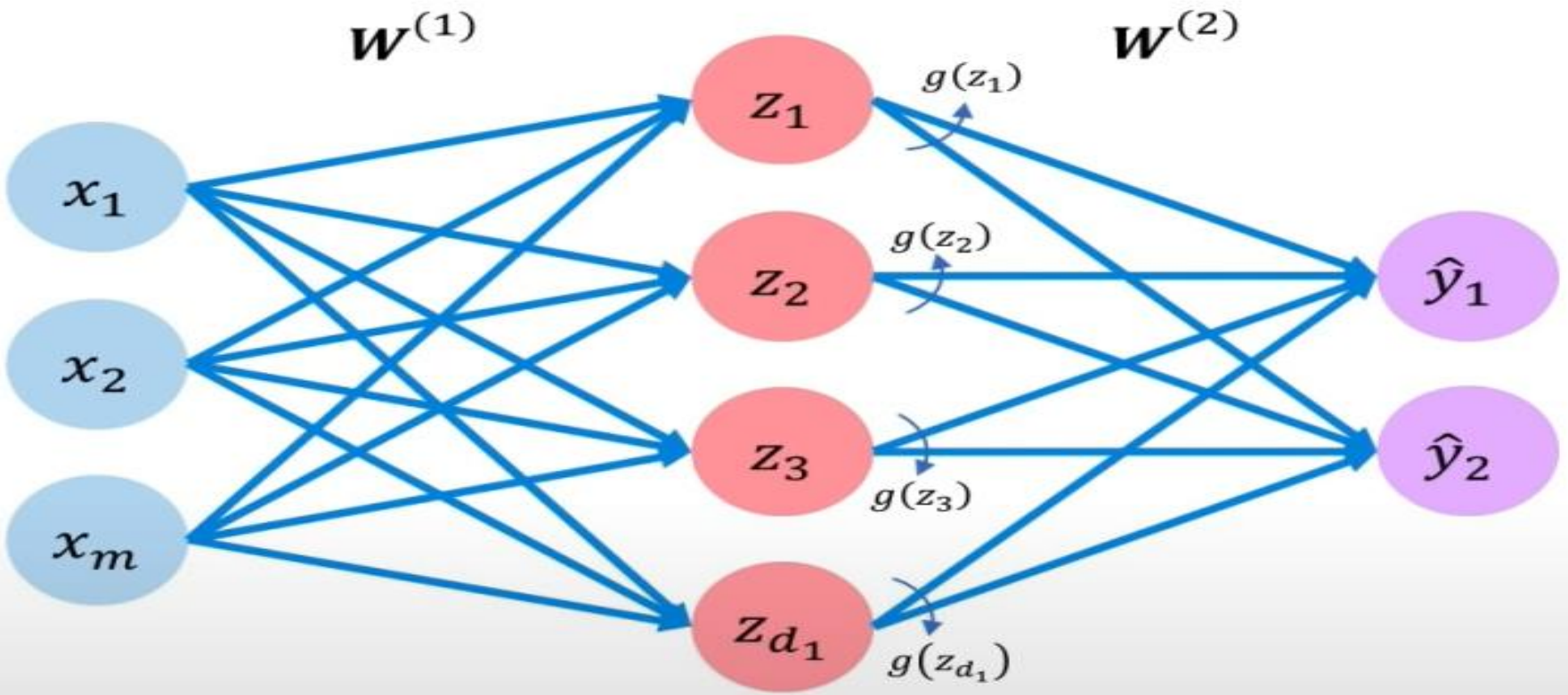
```
class MyDenseLayer(tf.keras.layers.Layer):
  def __init__(self, input_dim, output_dim):
    super(MyDenseLayer, self).__init__()

    # 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
```




```python
import tensorflow as tf

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

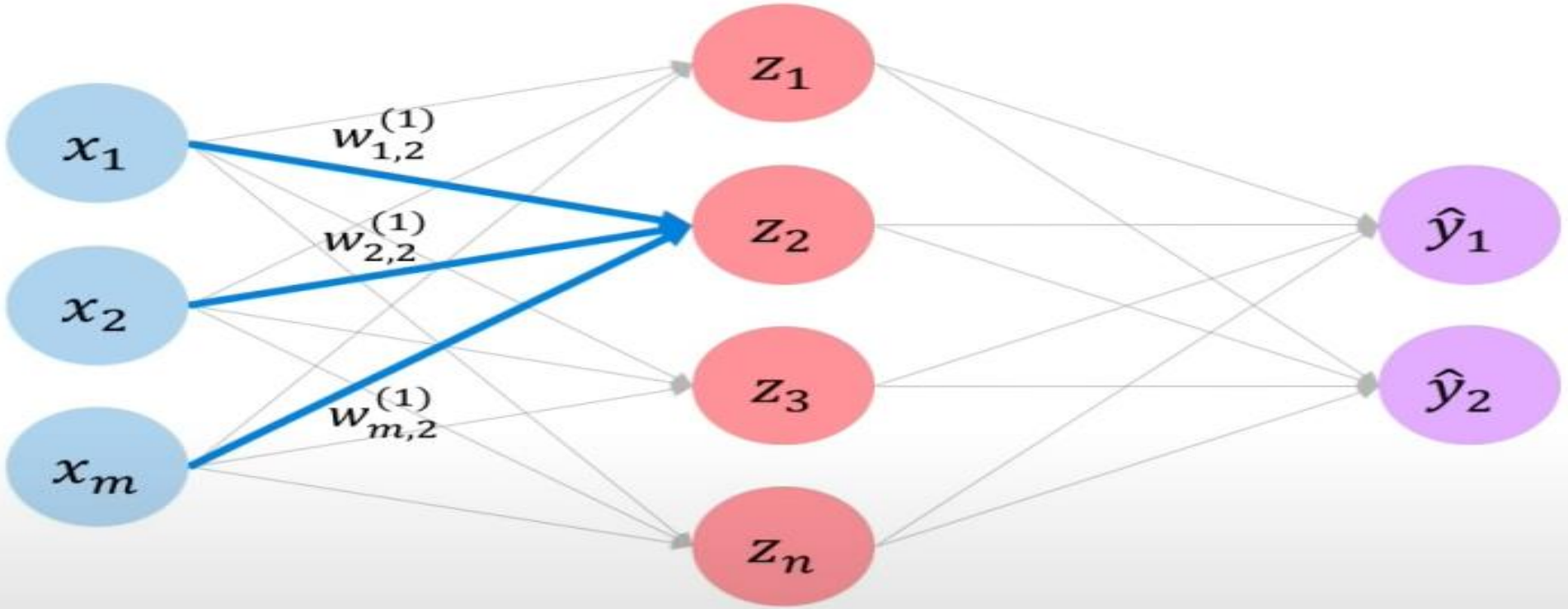
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

Single Layer Neural Network



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

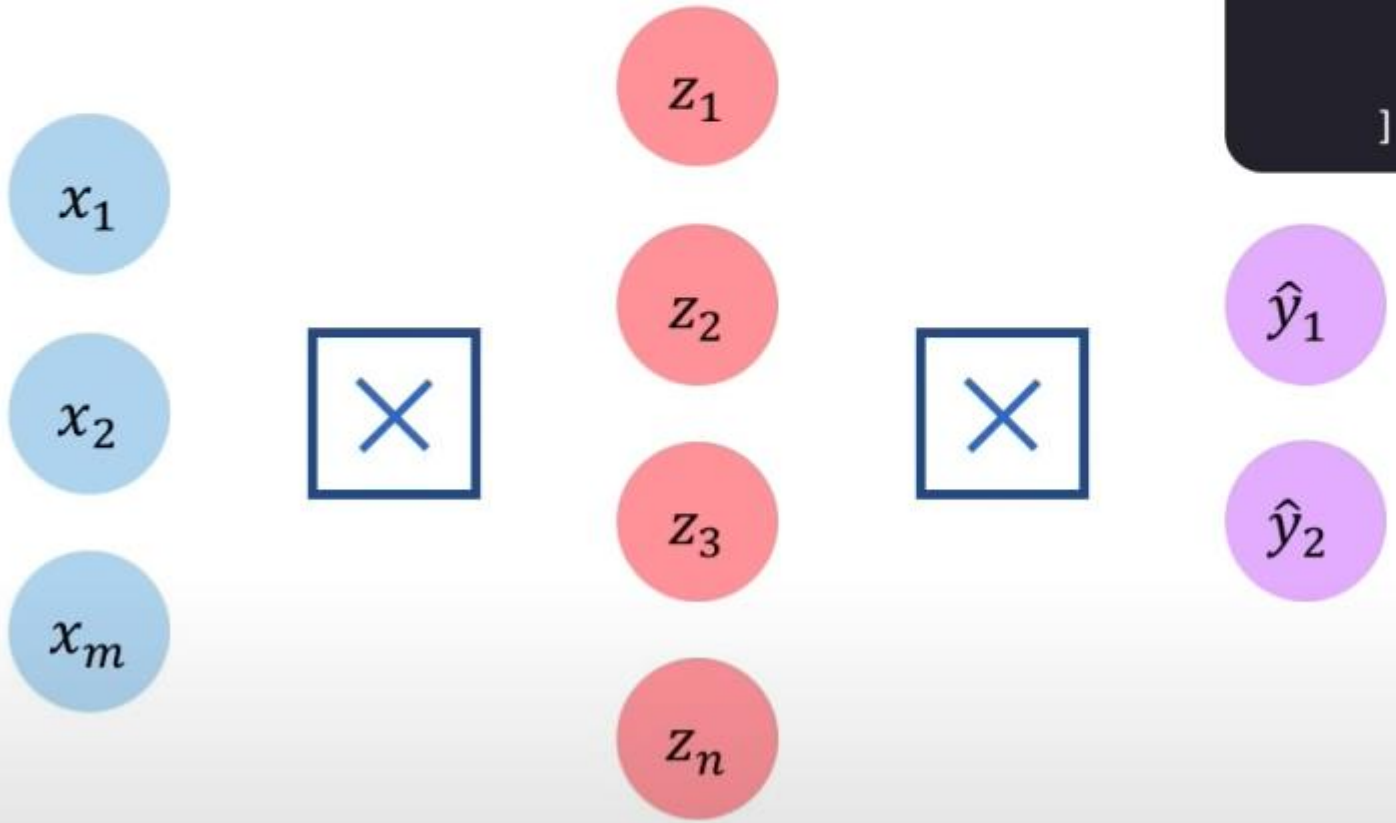
Single Layer Neural Network



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

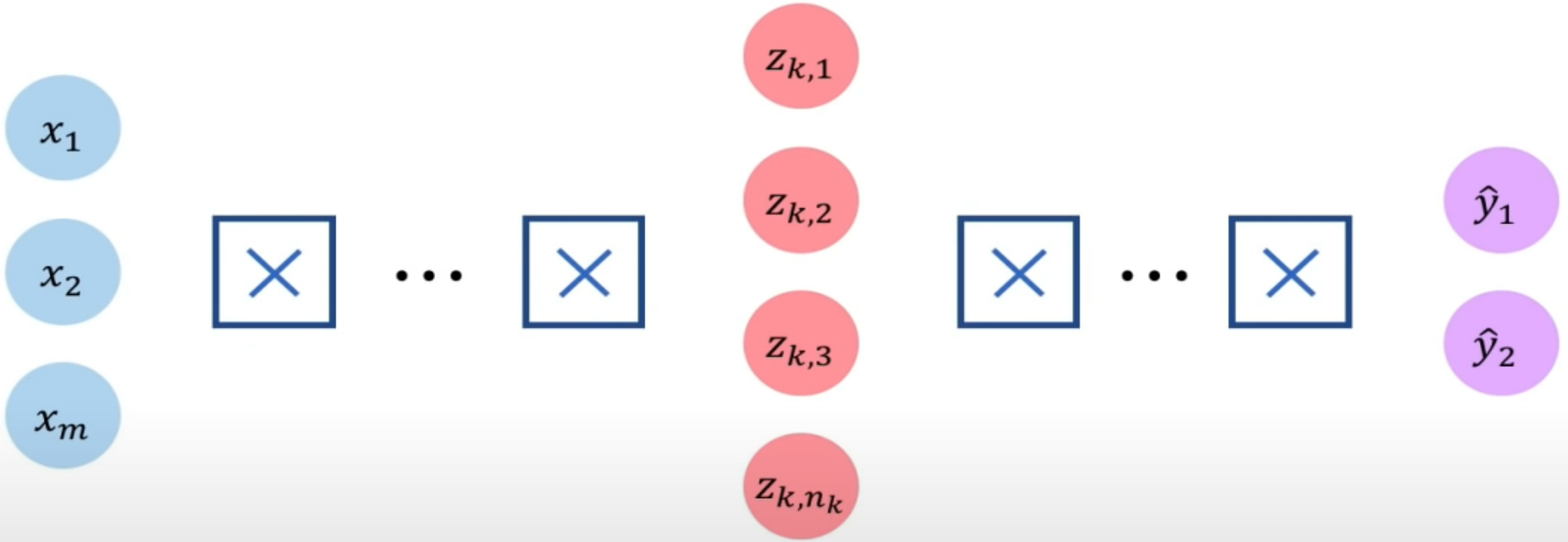
$$= w_{0,2}^{(1)} + x_1 \, w_{1,2}^{(1)} + x_2 \, w_{2,2}^{(1)} + x_m \, w_{m,2}^{(1)}$$

Multi Output Perceptron



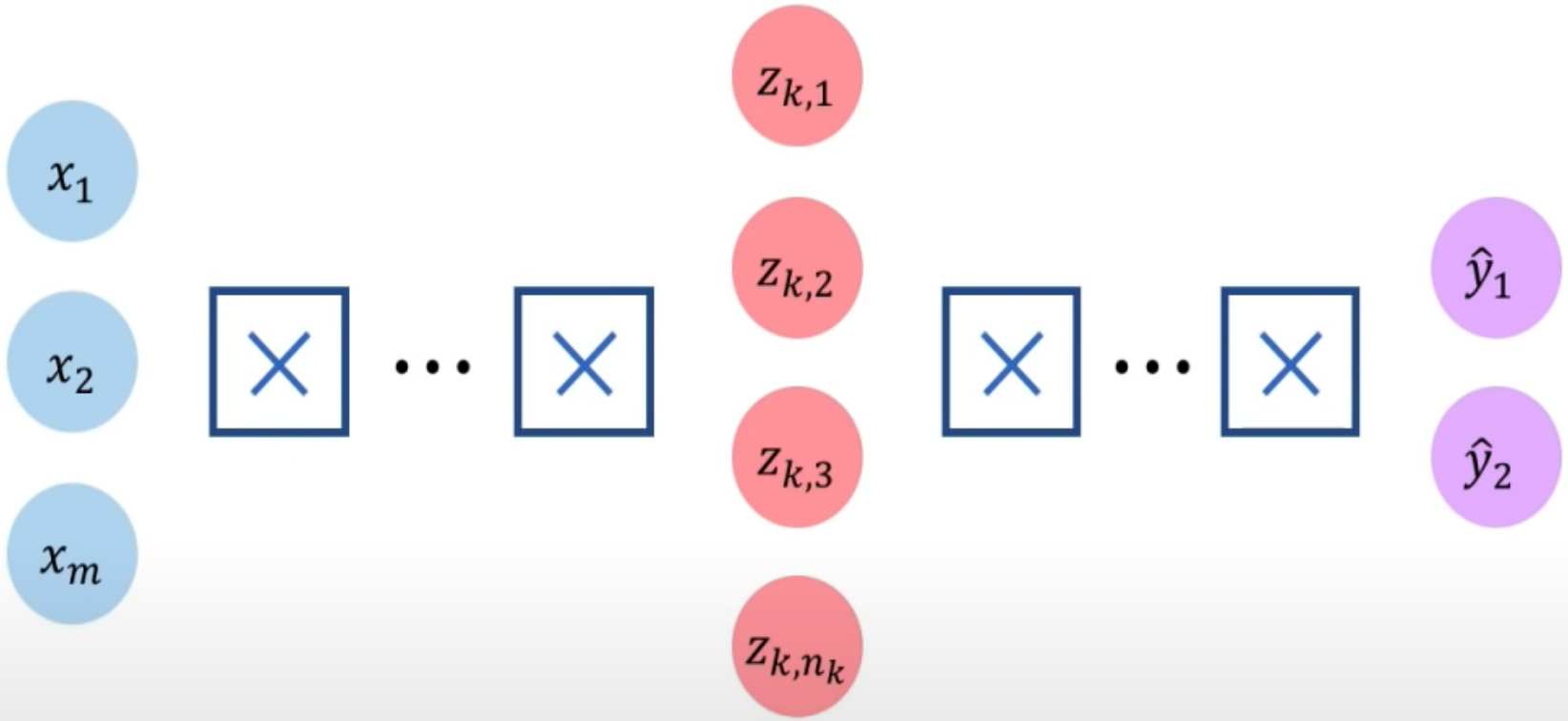

```python
import tensorflow as tf

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




```python
import tensorflow as tf

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

Inputs Hidden Output

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

Applying Neural Networks

Example Problem

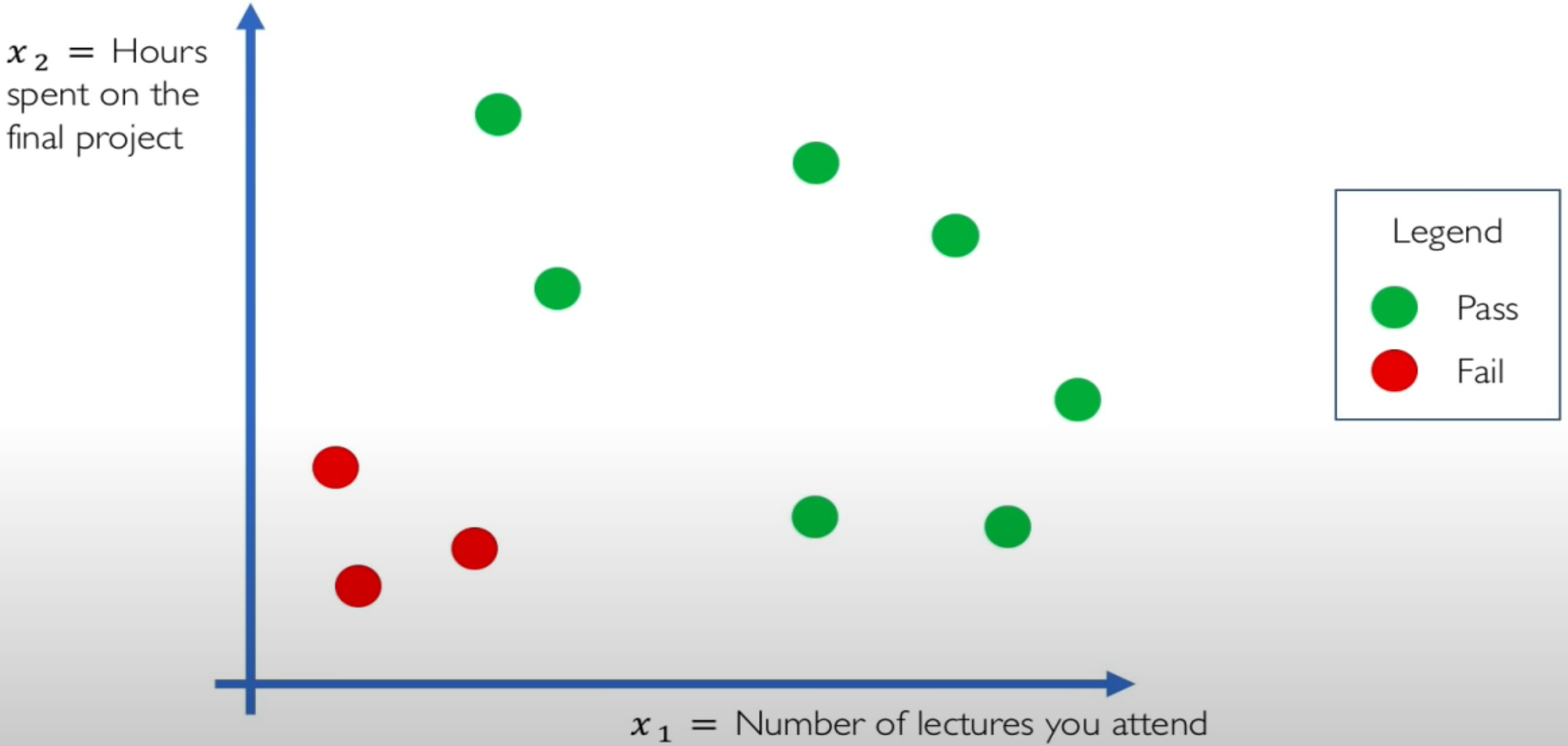
Will I pass this class?

Let's start with a simple two feature model

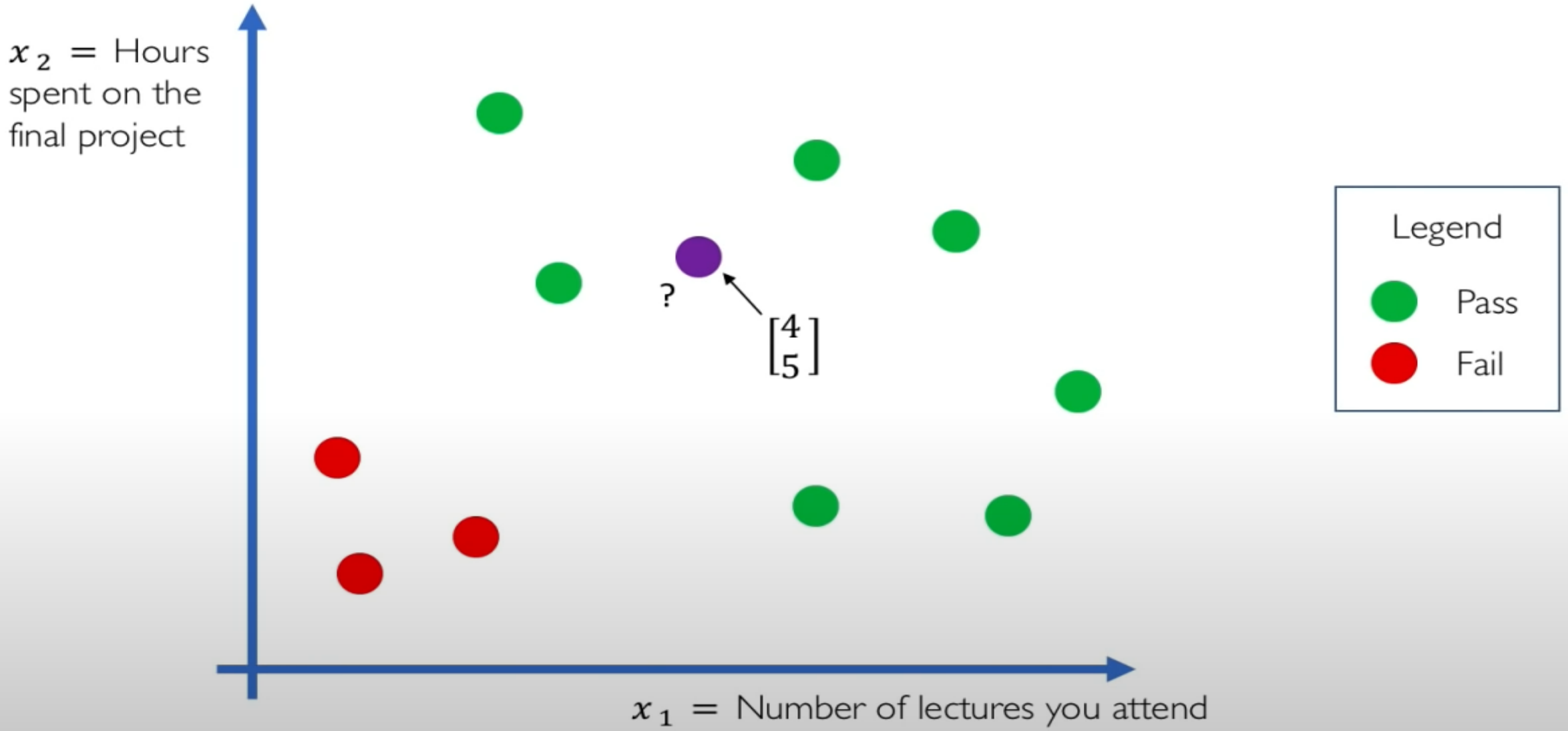
x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

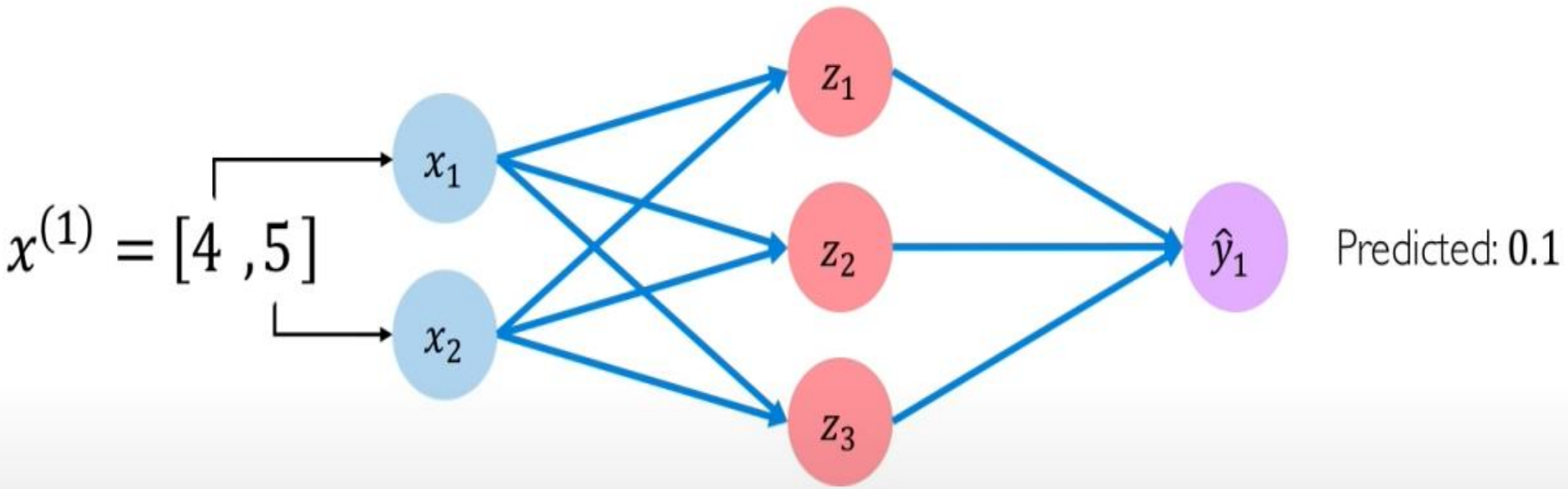
Example Problem: Will I pass this class?



Example Problem: Will I pass this class?

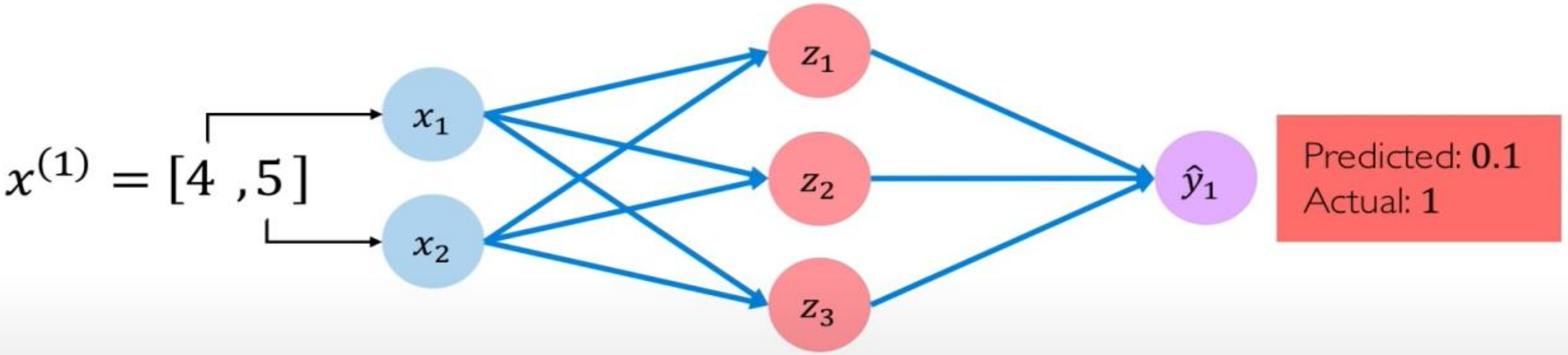


Example Problem: Will I pass this class?



Quantifying Loss

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



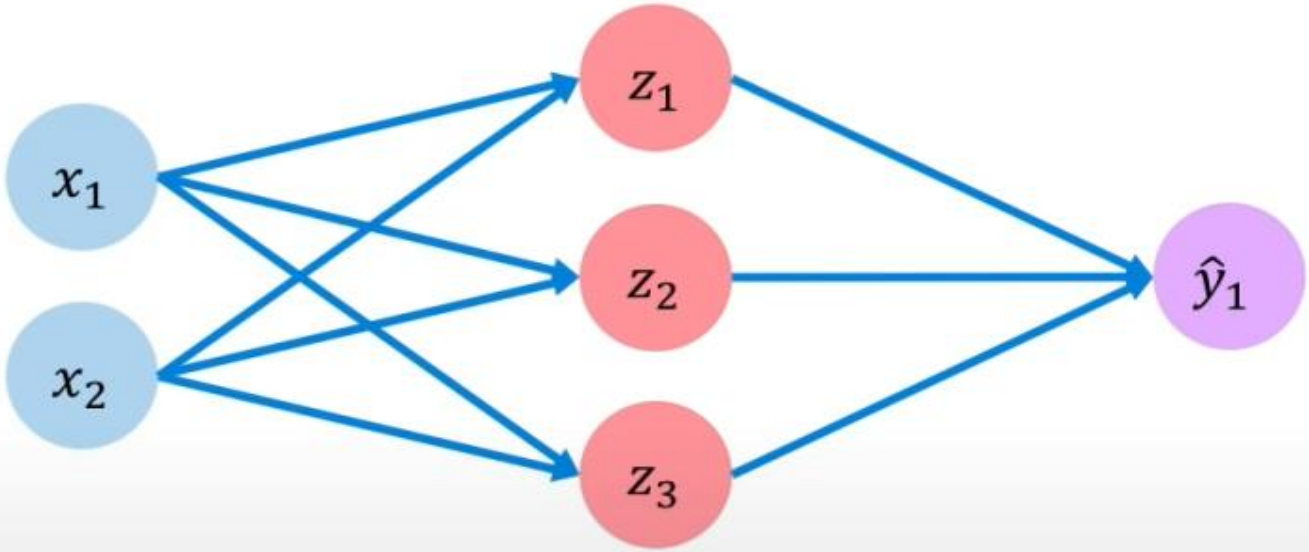
$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Empirical Loss

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

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



$f(x)$ y

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \begin{matrix} \times \\ \times \\ \checkmark \\ \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

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

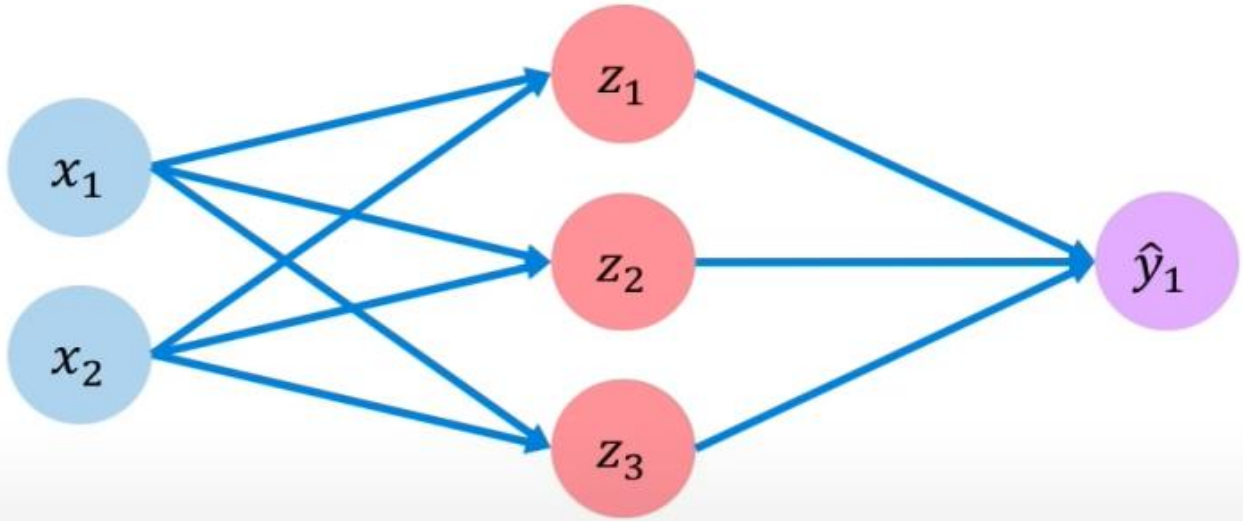
$$\boldsymbol{J(W)} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Binary Cross Entropy Loss

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

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



$f(x)$		y
0.1	✗	1
0.8	✗	0
0.6	✓	1
⋮		⋮

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

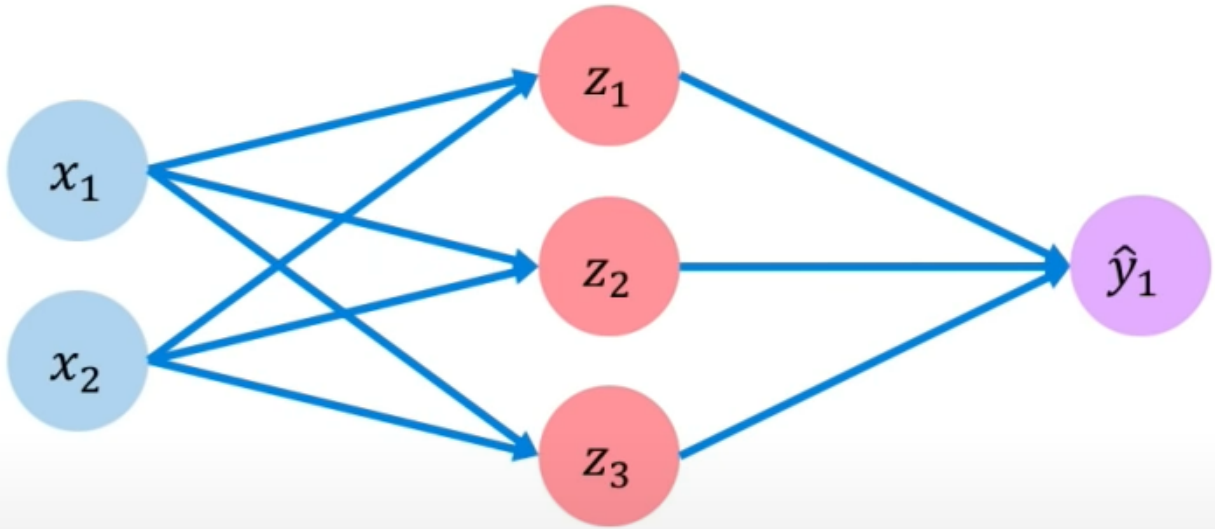
Actual — Predicted — Actual — Predicted

```
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*

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



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

Actual Predicted

$f(x)$		y
30	✗	90
80	✗	20
85	✓	95
⋮		⋮

Final Grades (percentage)

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

Training Neural Networks

Loss Optimization

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

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

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

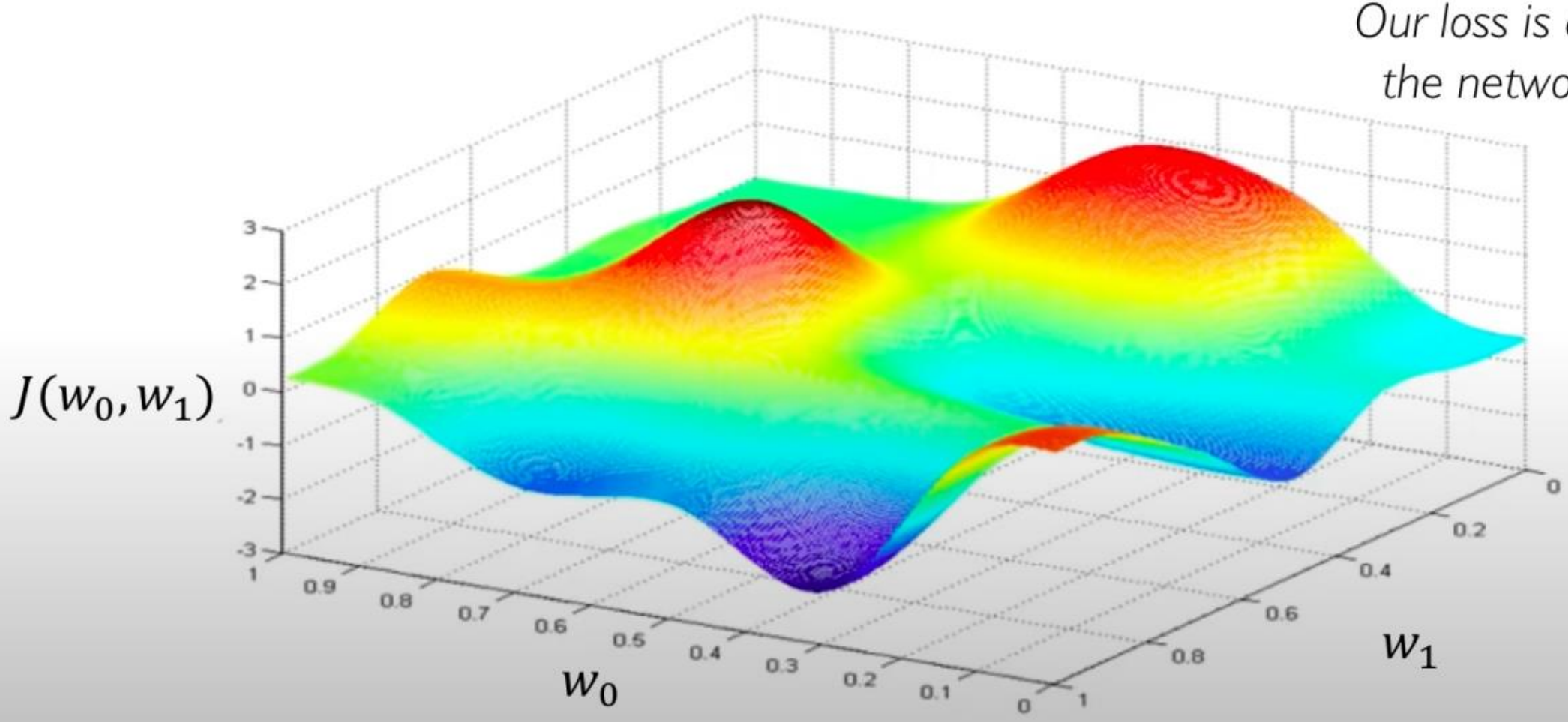
Remember:

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

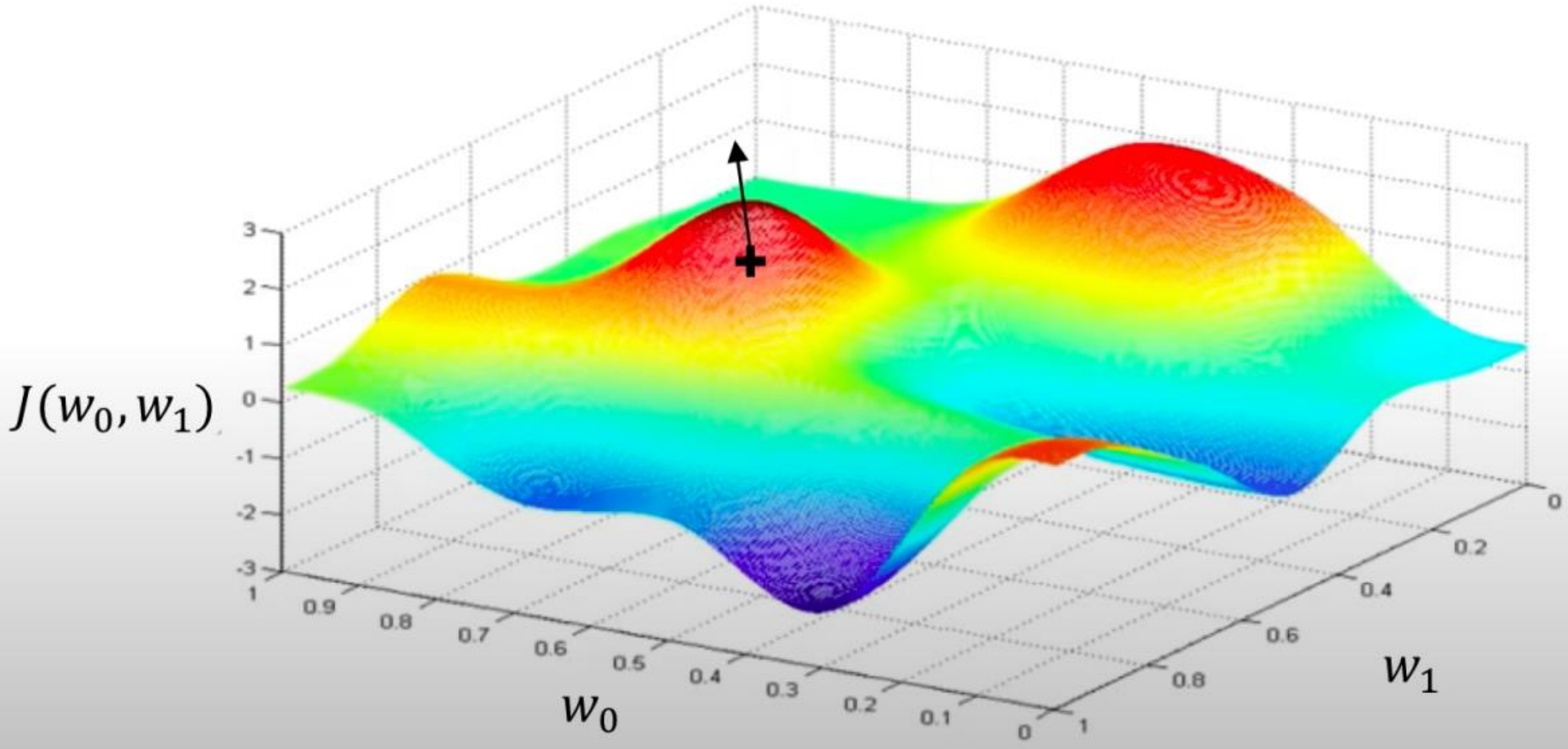
Loss Optimization

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

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

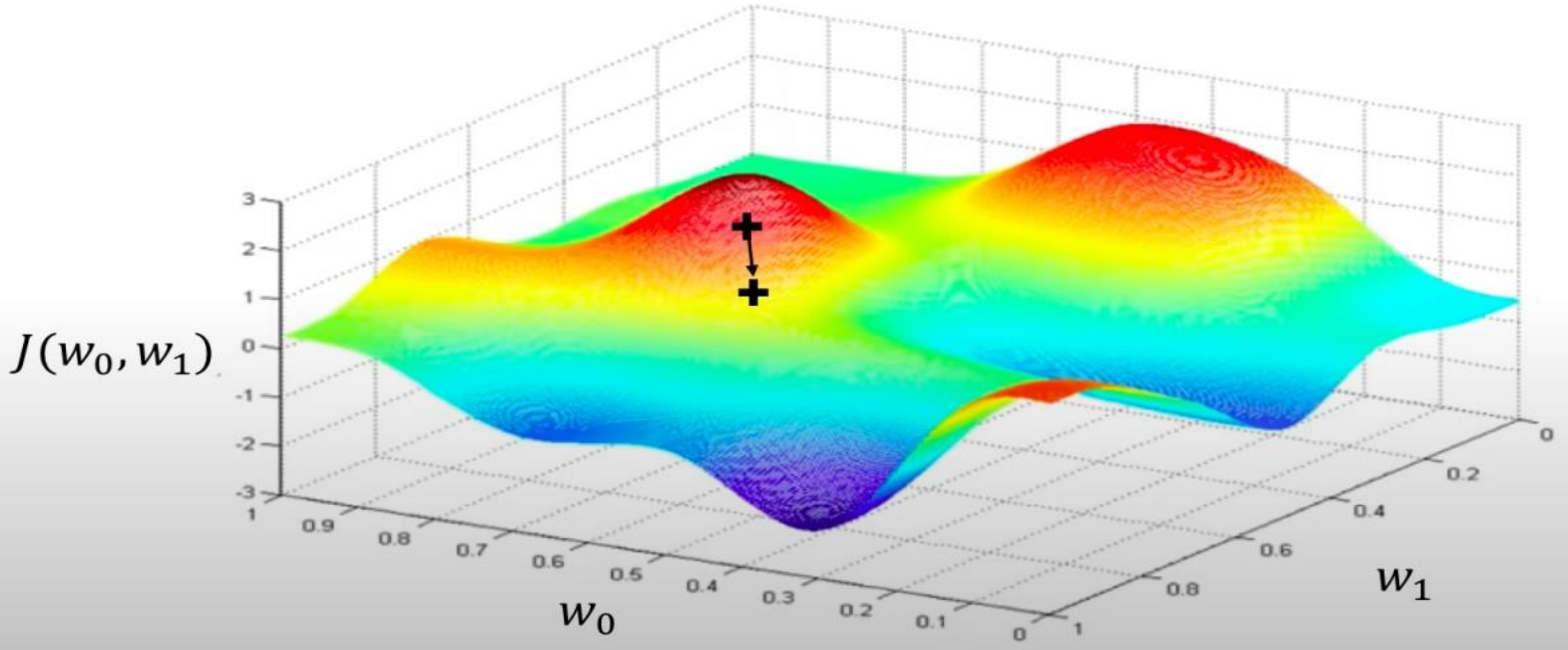


Randomly pick an initial (w_0, w_1) ⇒ Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$



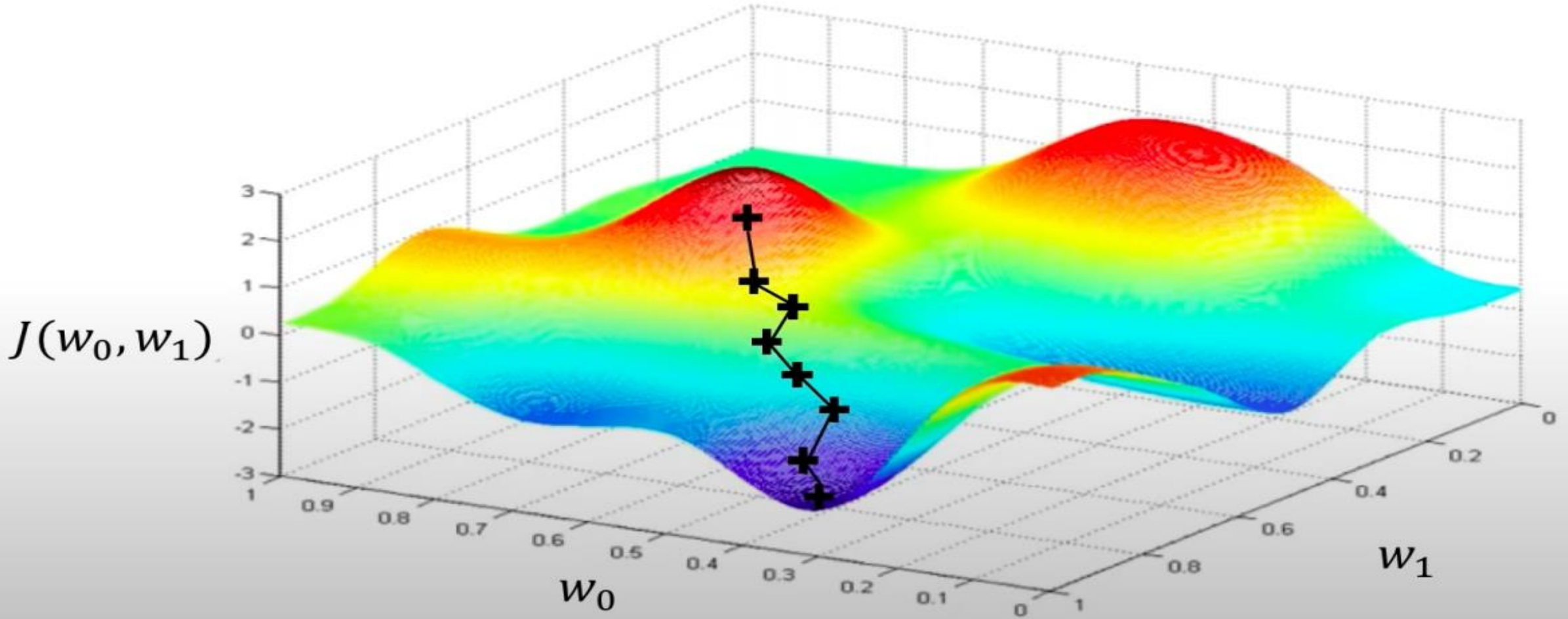
Loss Optimization

Take small step in opposite direction of gradient



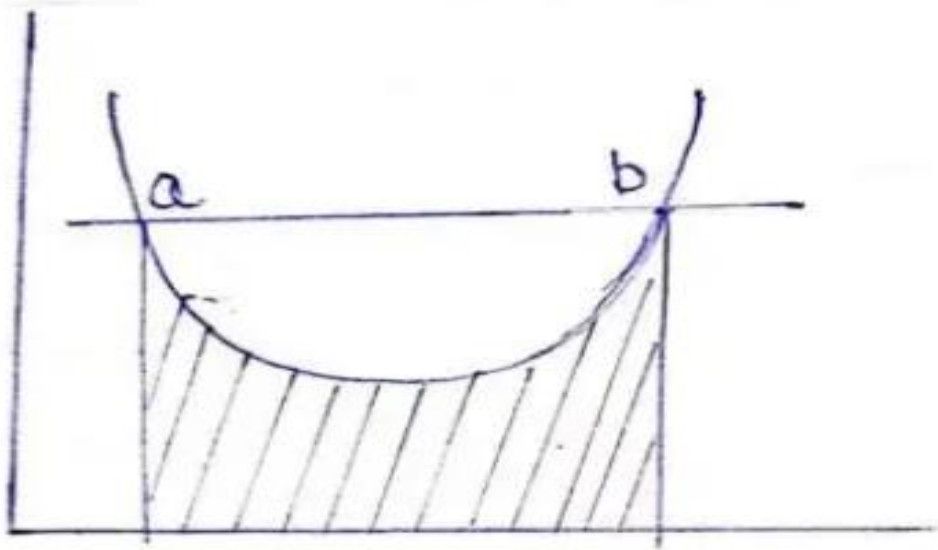
Gradient Descent

Repeat until convergence

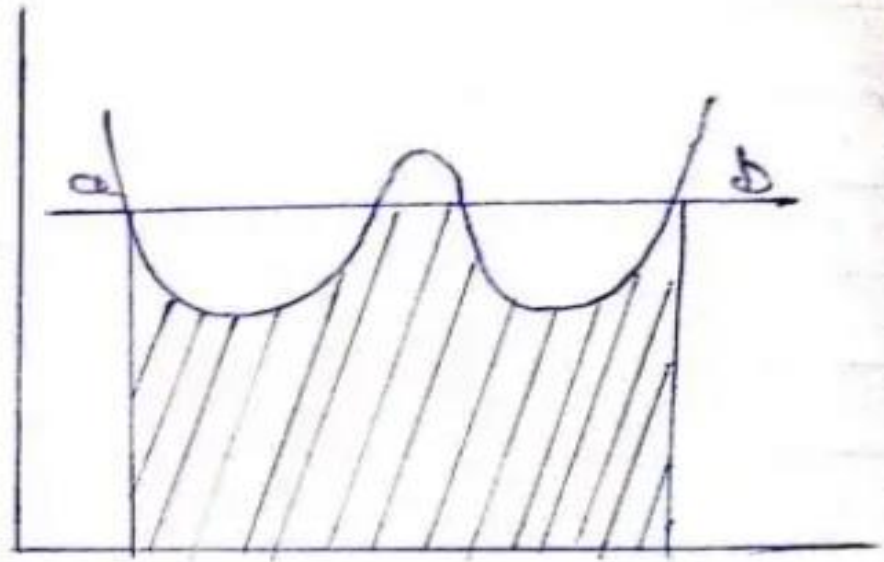


Gradient Descent

Gradient Descent is an optimizing algorithm used in Machine/ Deep Learning algorithms. The goal of Gradient Descent is to minimize the objective convex function f(x) using iteration.



convex function



not convex function

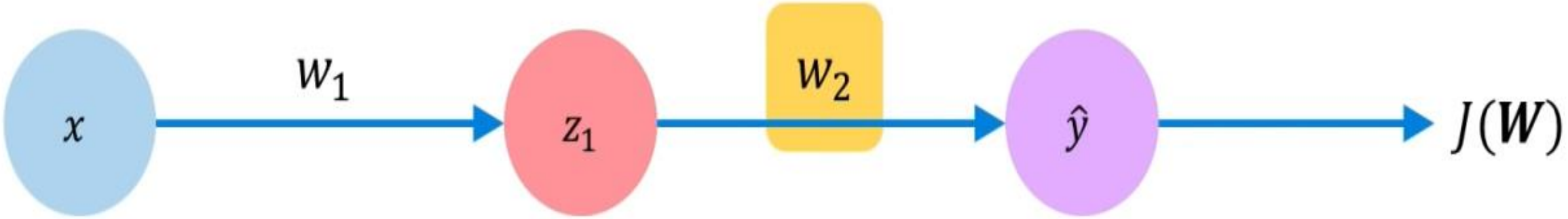
Gradient Descent

Algorithm

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

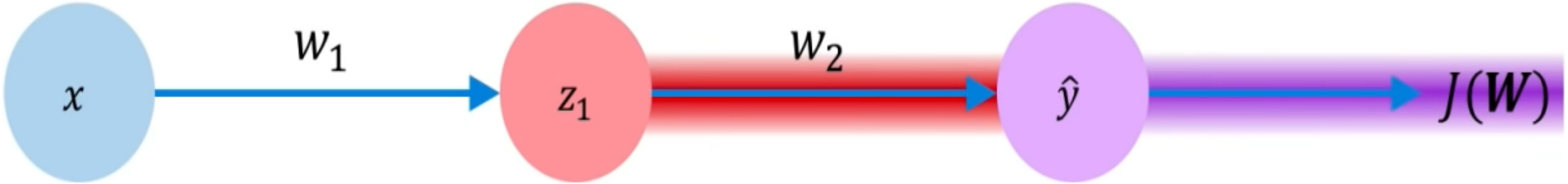
```python
import tensorflow as tf

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

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

    weights = weights - lr * gradient
```

Computing Gradients: Backpropagation



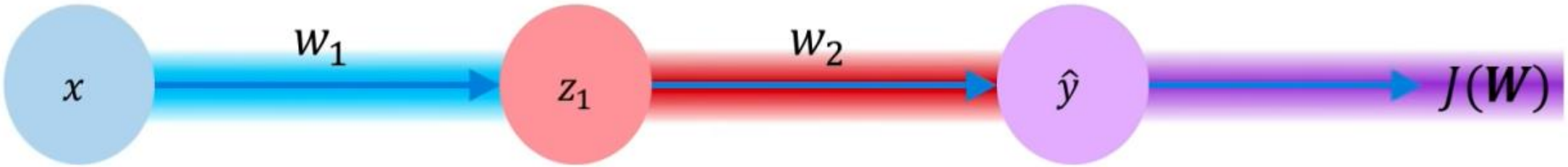
How does a small change in one weight (ex. $\boldsymbol{w_2}$) affect the final loss $\boldsymbol{J(W)}$?

Computing Gradients: Backpropagation



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

Computing Gradients: Backpropagation



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

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

Neural Networks in Practice: Optimization

Loss Functions Can Be Difficult to Optimize

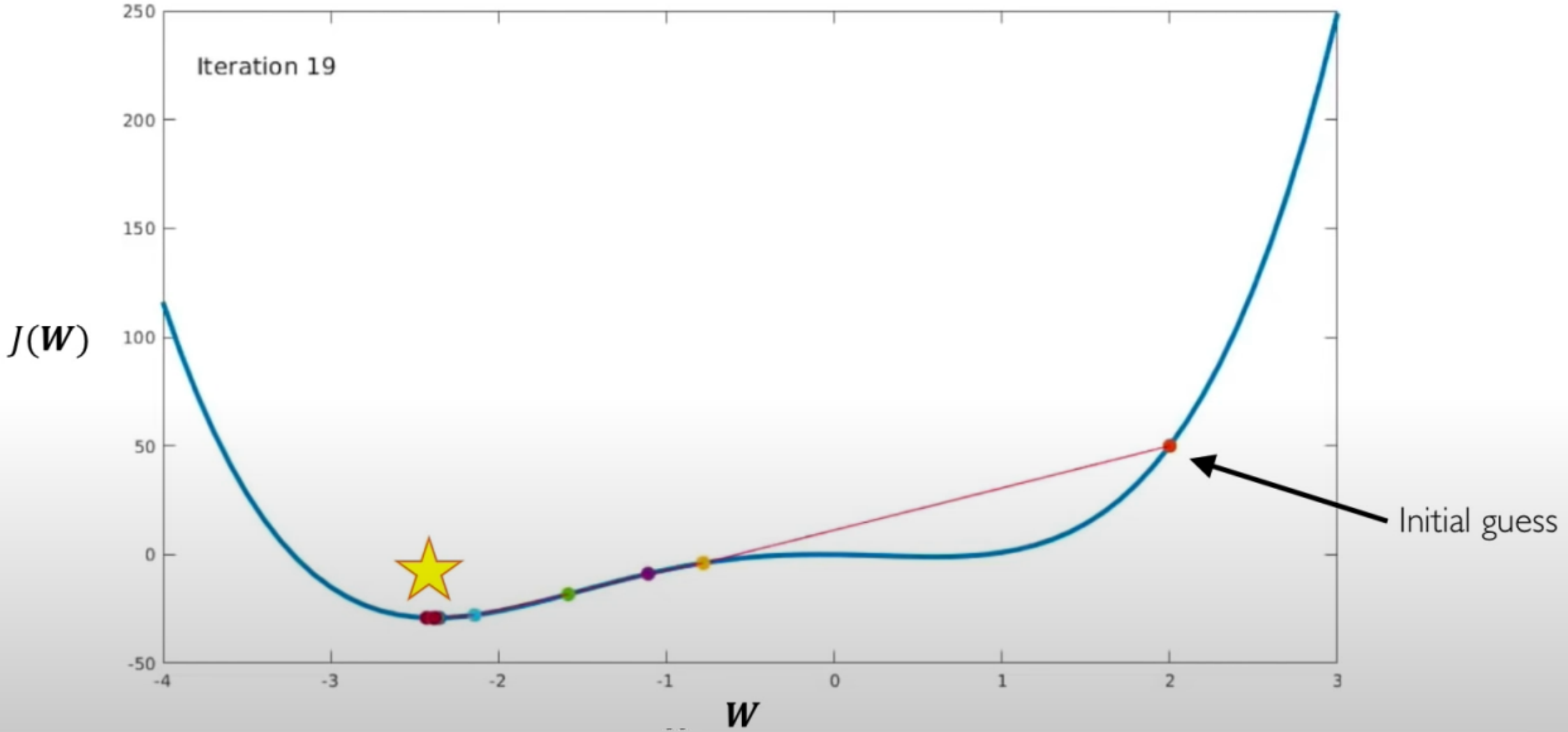
Remember:

Optimization through gradient descent

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

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"

Idea 2:

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...

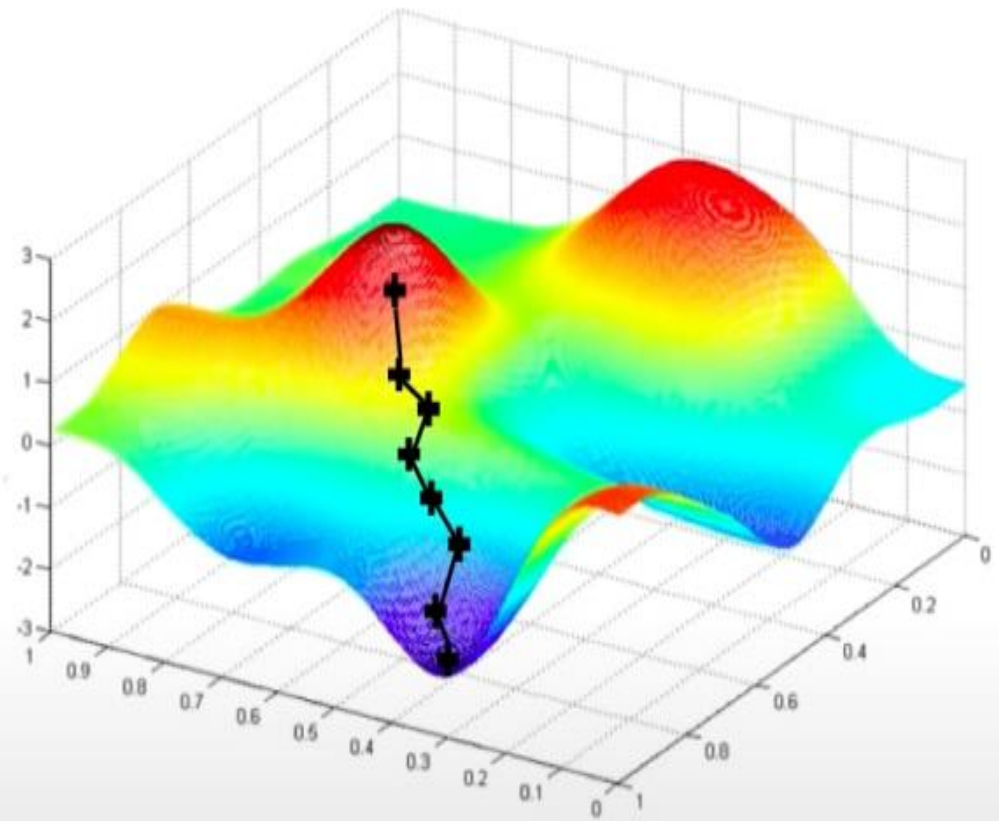
Gradient Descent Algorithms

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

Gradient Descent

Algorithm

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

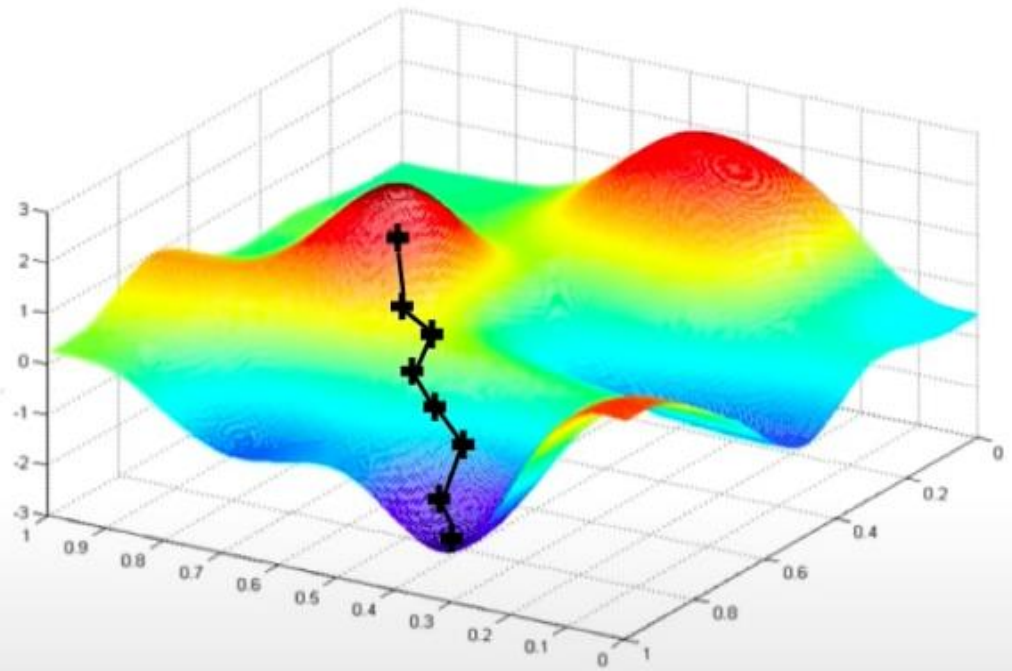


Stochastic Gradient Descent

Algorithm

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

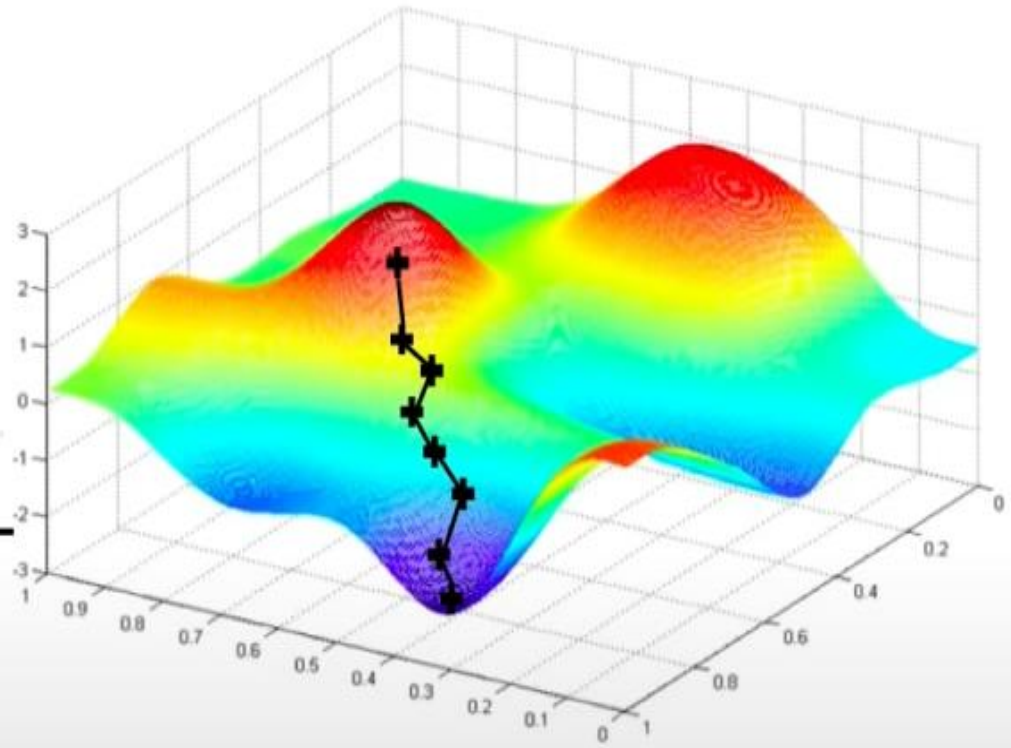
Easy to compute but
very noisy (stochastic)!



Stochastic Gradient Descent

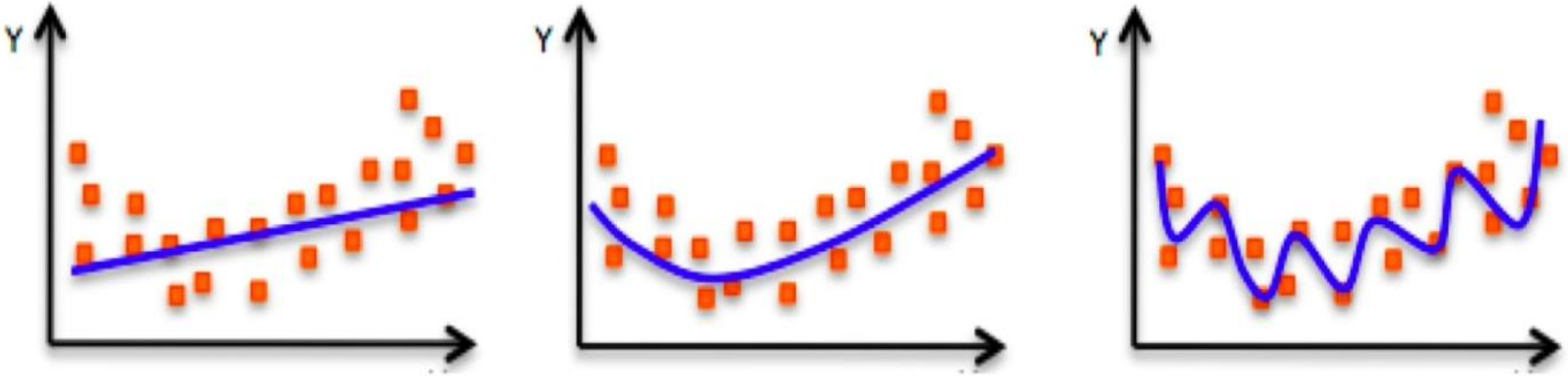
Algorithm

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



Neural Networks in Practice: Overfitting

The Problem of Overfitting



Underfitting
Model does not have capacity to fully learn the data

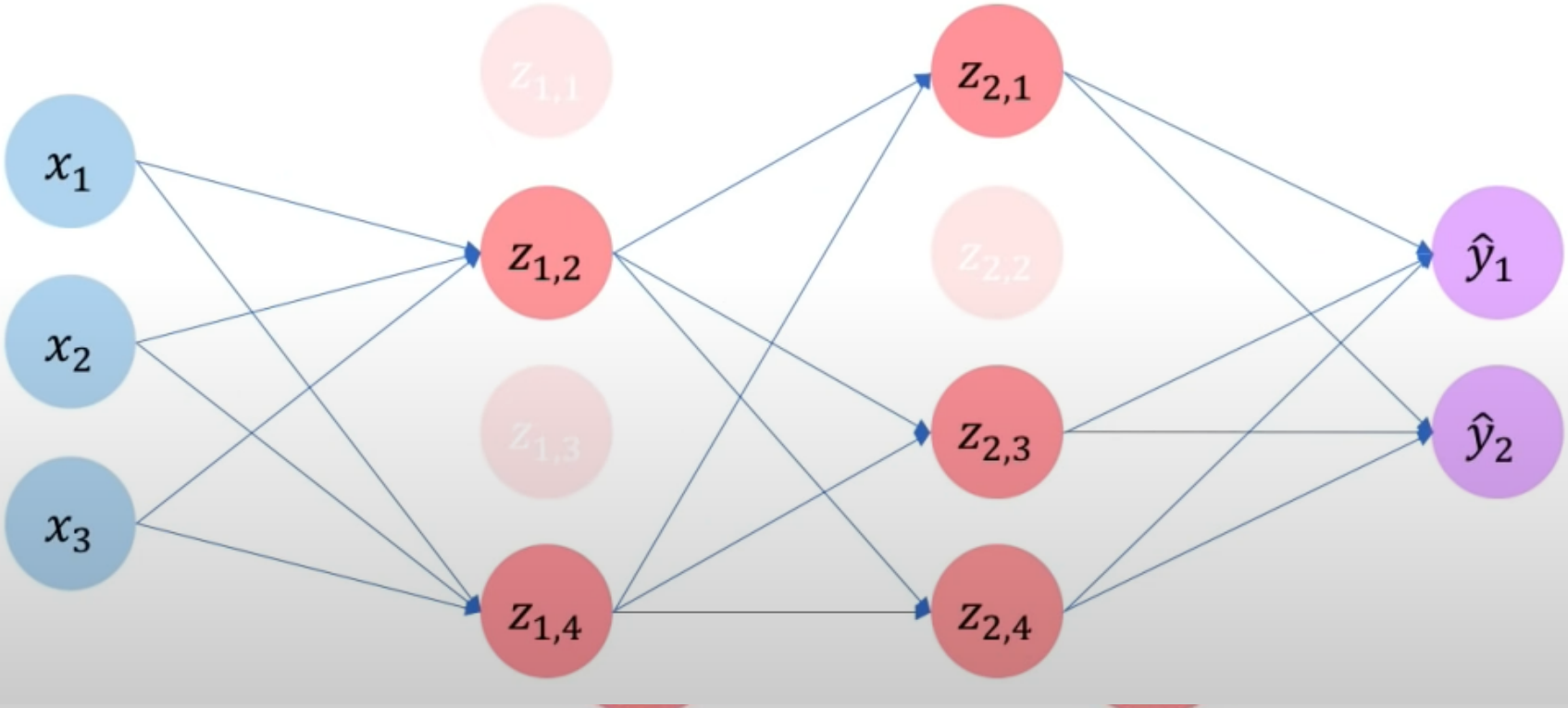
Ideal fit

Overfitting
Too complex, extra parameters, does not generalize well

Regularization 1: Dropout

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

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

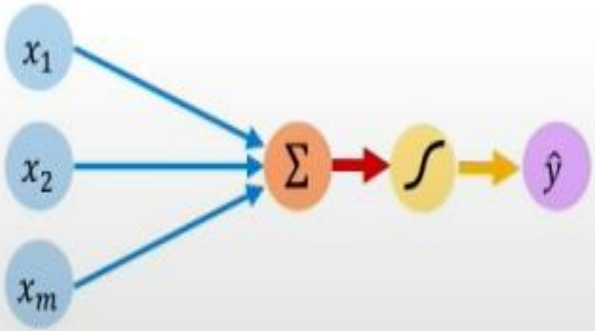


Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

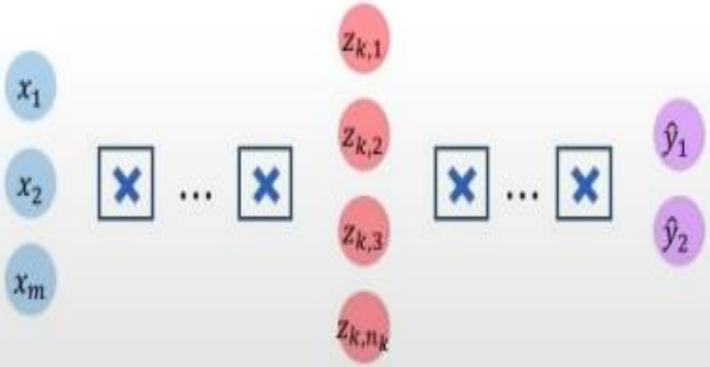
The Perceptron

- Structural building blocks
- Nonlinear activation functions



Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization