# Lecture 4 – Intro to Deep Learning

Naveh Porat

June 2022

# Table of Contents


**01**

Quick ML Brief

**02**

The Neuron

**03**

Feed Forward Neural Network

**04**

Backpropagation Algorithm

**05**

DNN Tips & Tricks

**06**

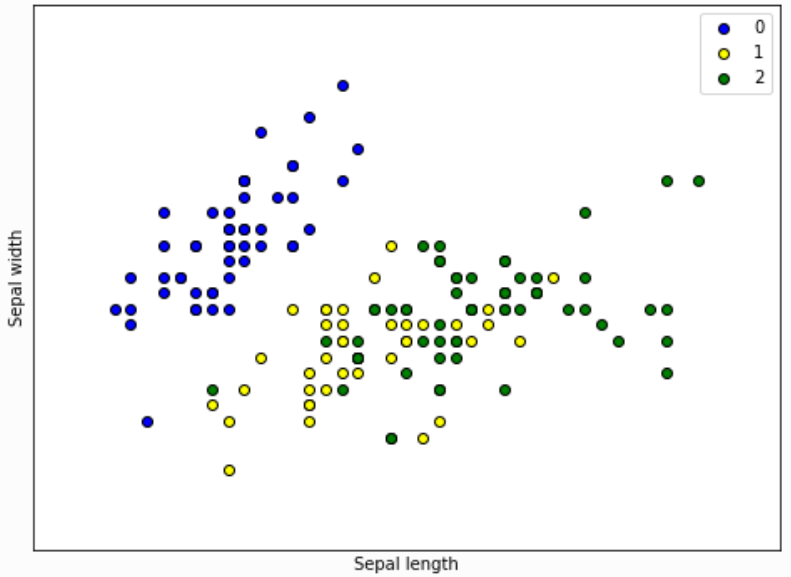
Why Deep

# Quick ML Brief

# Iris Dataset

- In the dataset there a 3 classes of Iris types, and each class contains 50 instances.
- Each instance is provided with the width and the length of its petal and its sepal.
- The dataset can be found here.


**Iris Versicolor** **Iris Setosa** **Iris Virginica**



# Components of a ML Model

A ML model can be breaked down into 5 main parts:

1. The data representation.
2. The model architecture.
3. The model's loss\objective.
4. Training algorithm.
5. Inference method.

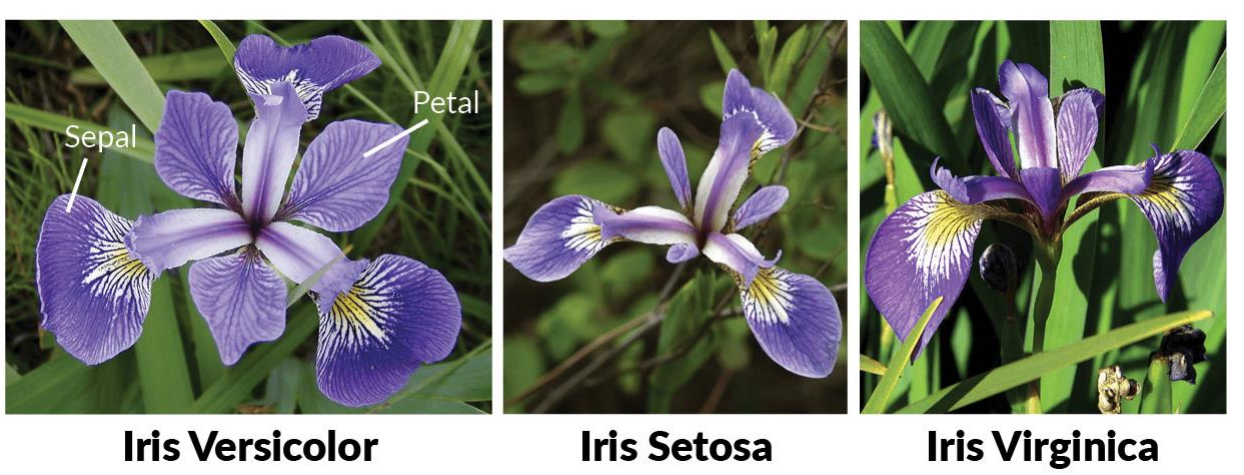
# Components of a ML Model – Data Representation

We want to classify Iris flowers. What data do we provide the model?

- Petal Length/Width
- Color
- Picture

What format is the data inputed?

- 1, 2, 5
- 0.1, 3.3
- A, B, C



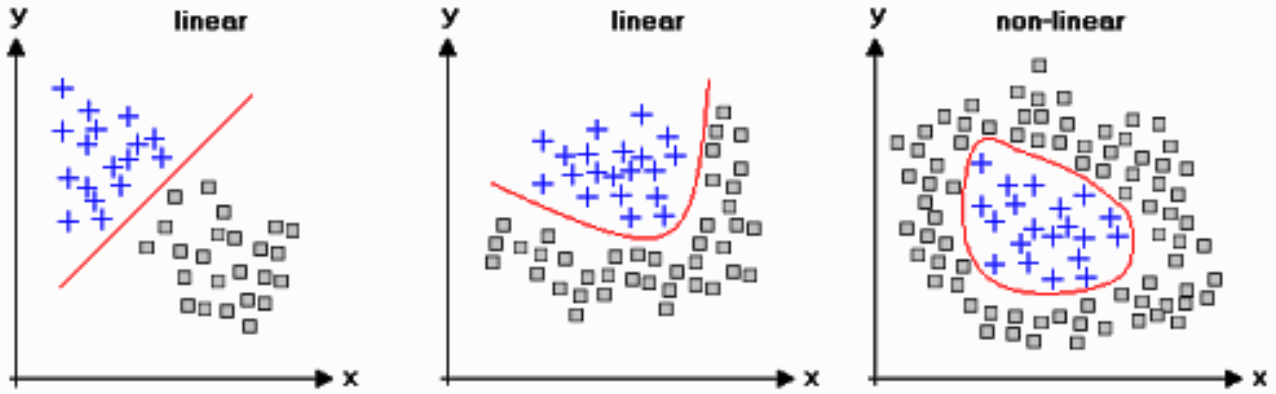
# Components of a ML Model – Architecture

What are the mathematical operations the model can use?

- Linear: $f(x) = a \cdot x + b$
- Polynomial: $f(x) = a \cdot x^3 + b \cdot x^2$
- Anything else

What are the parameters that need to be learnt?

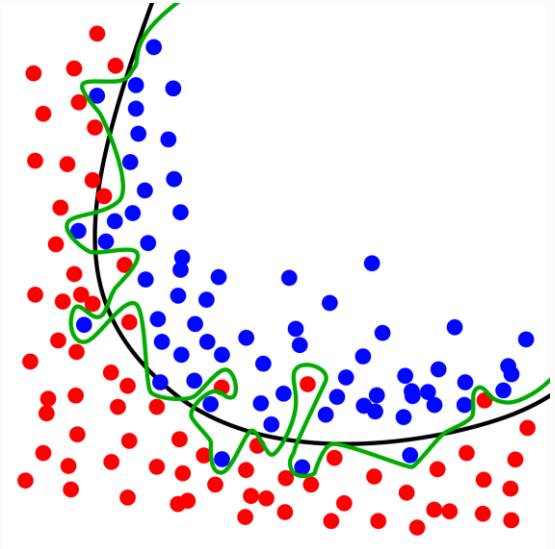
- $f(x) = a \cdot x + 8$

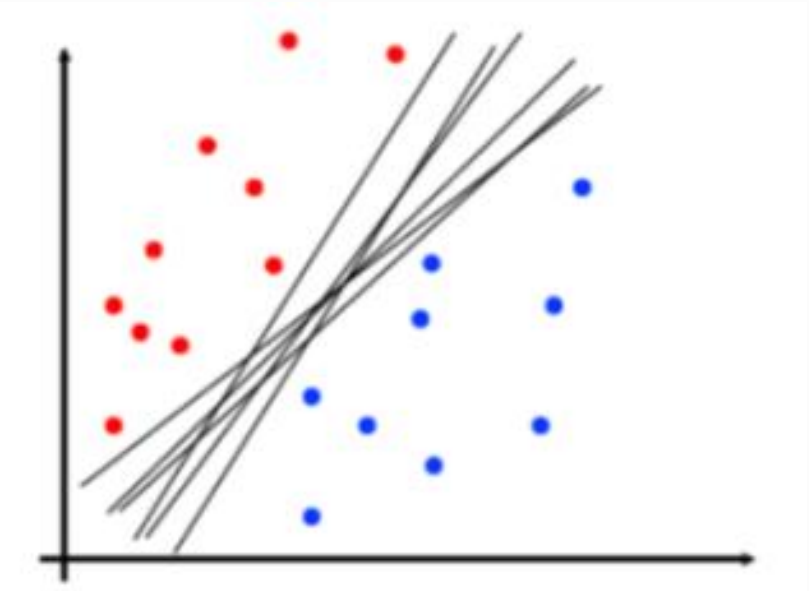


# Components of a ML Model – Loss\Objective

How do we define a “good” model? What does the model aim to achieve?

- Margin or other properties
- Recall vs. Accuracy
- Overfitting, complexity

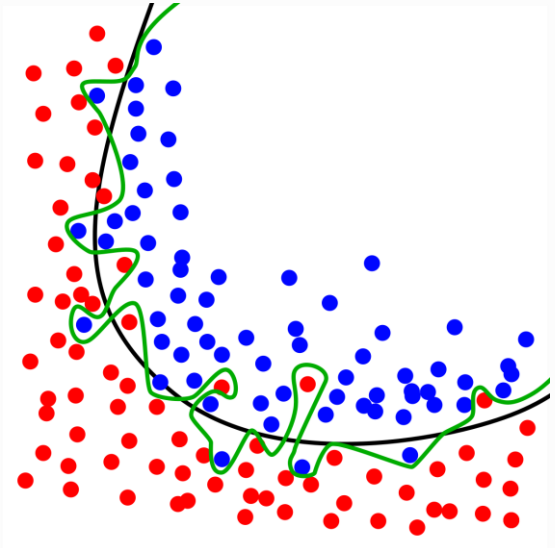


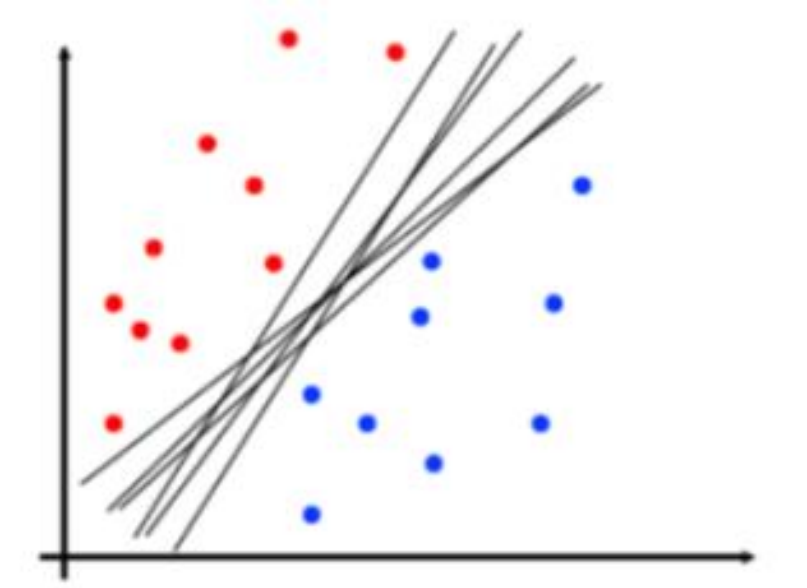


# Components of a ML Model – Training Algorithm

How does the model learn the parameters?

- Usually, we define a loss function (as we saw), and the model tries to minimize the loss.
- There are different algorithms for that.





# Gradient Descent [~1847]

- Goal: find

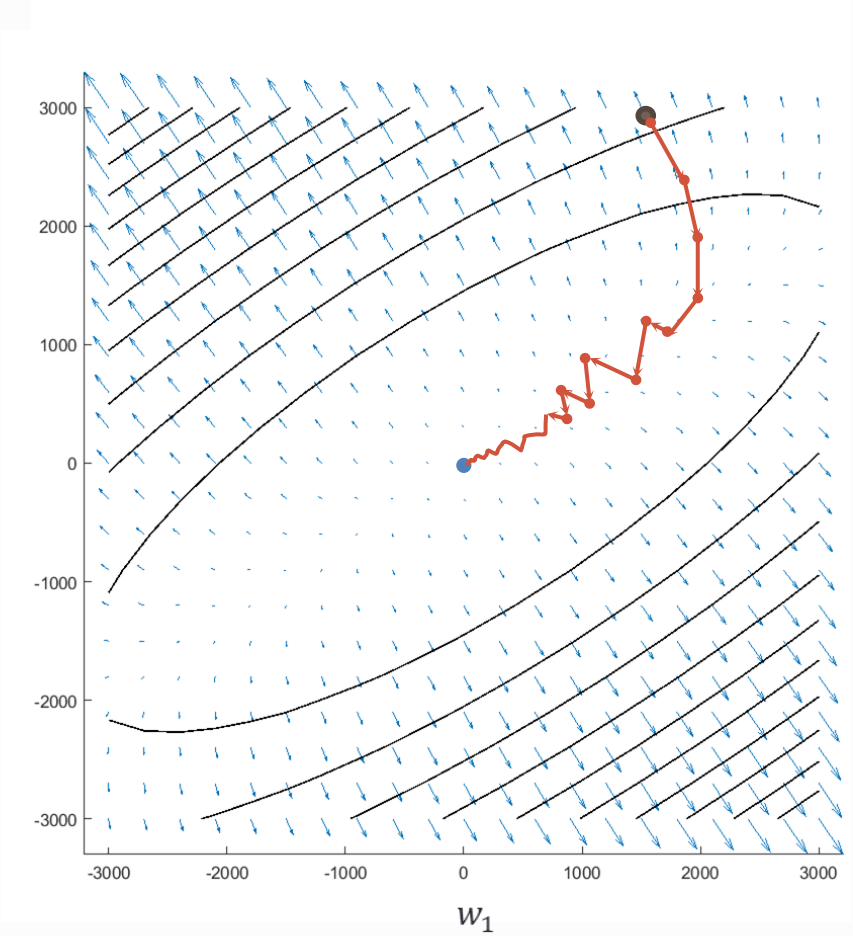
$$w^* = argmin_{w \in \mathbb{R}} \mathcal{L}(w)$$

- Idea: gradients point to steepest ascent direction in loss landscape

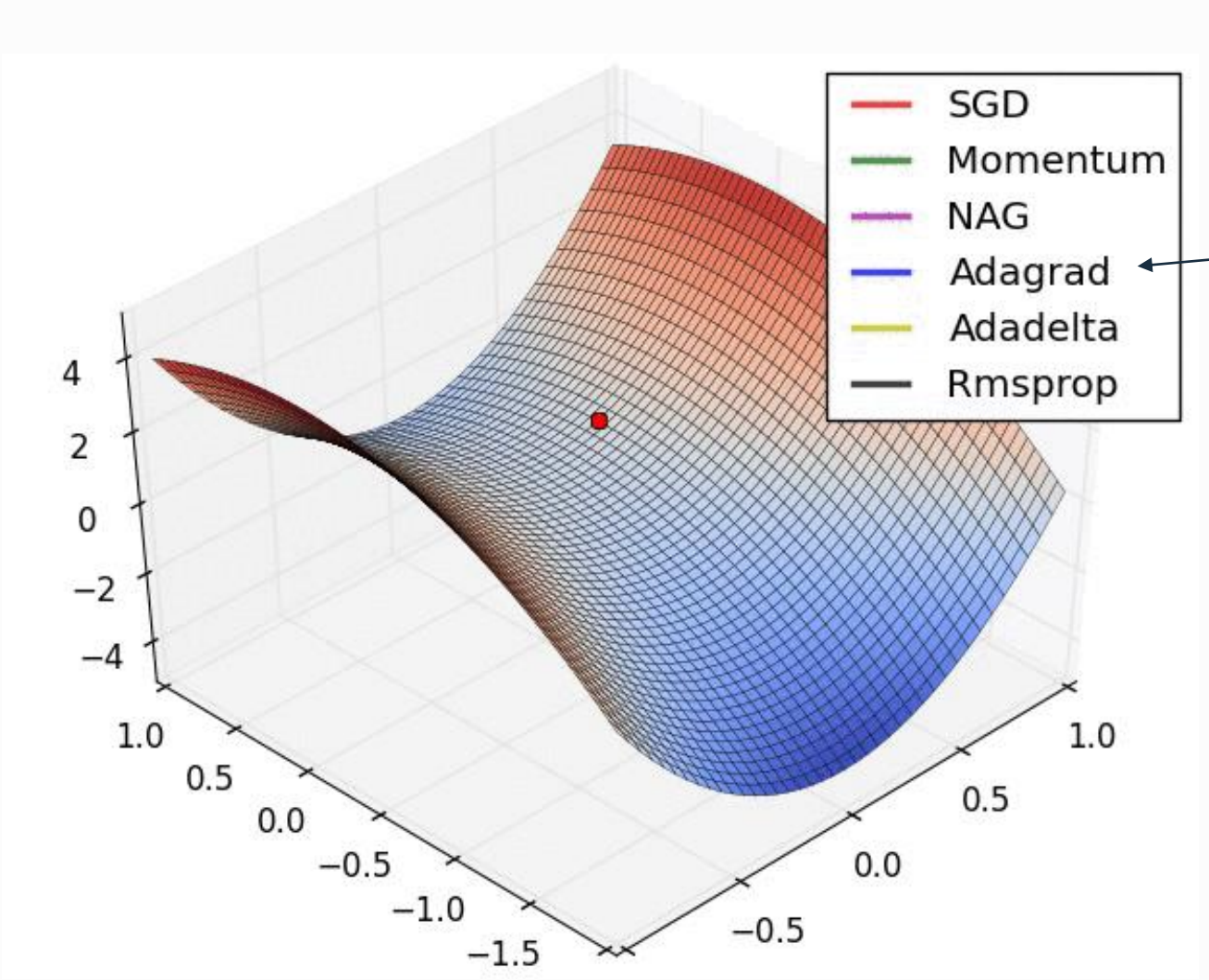
- Descend iteratively:

$$w^{k+1} = w^k - \eta \nabla_w \mathcal{L}(w^k)$$

- Guarantee: for $\eta$ small enough, converge to a local minimum



# Stochastic GD and Friends

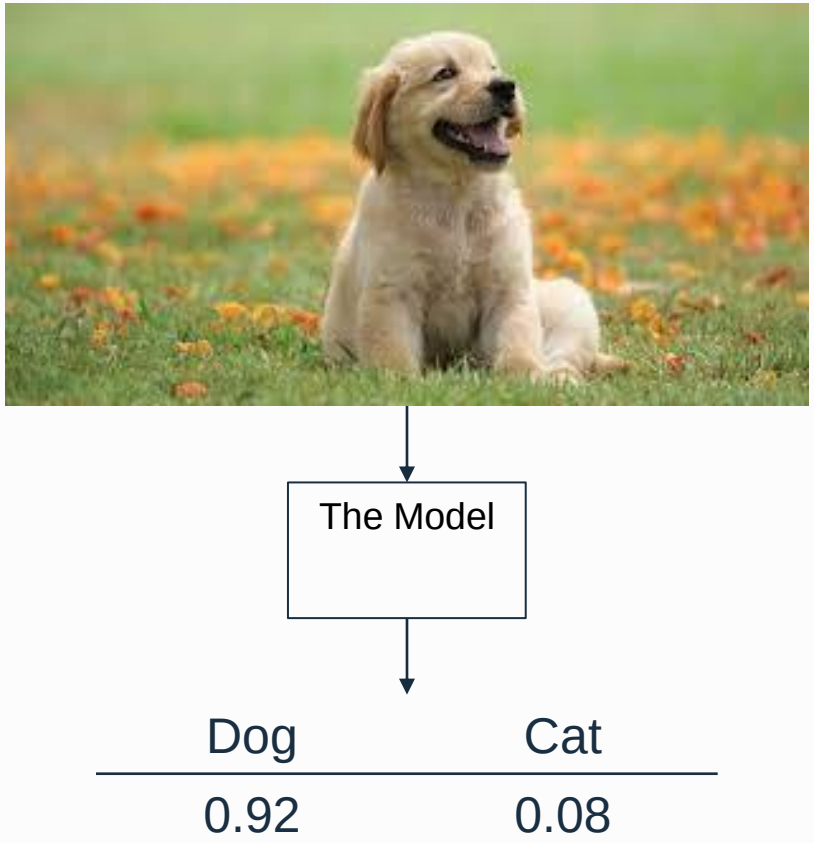



Elad Hazan!!

# Components of a ML Model – Inference

Once we have the model's predictions, how do we infer a result?

- Usually, we define a loss function (as we saw), and the model tries to minimize the loss.
- There are different algorithms for that.



# Intro to DL

# A Brief History of NLP

## Where are we on the timeline?

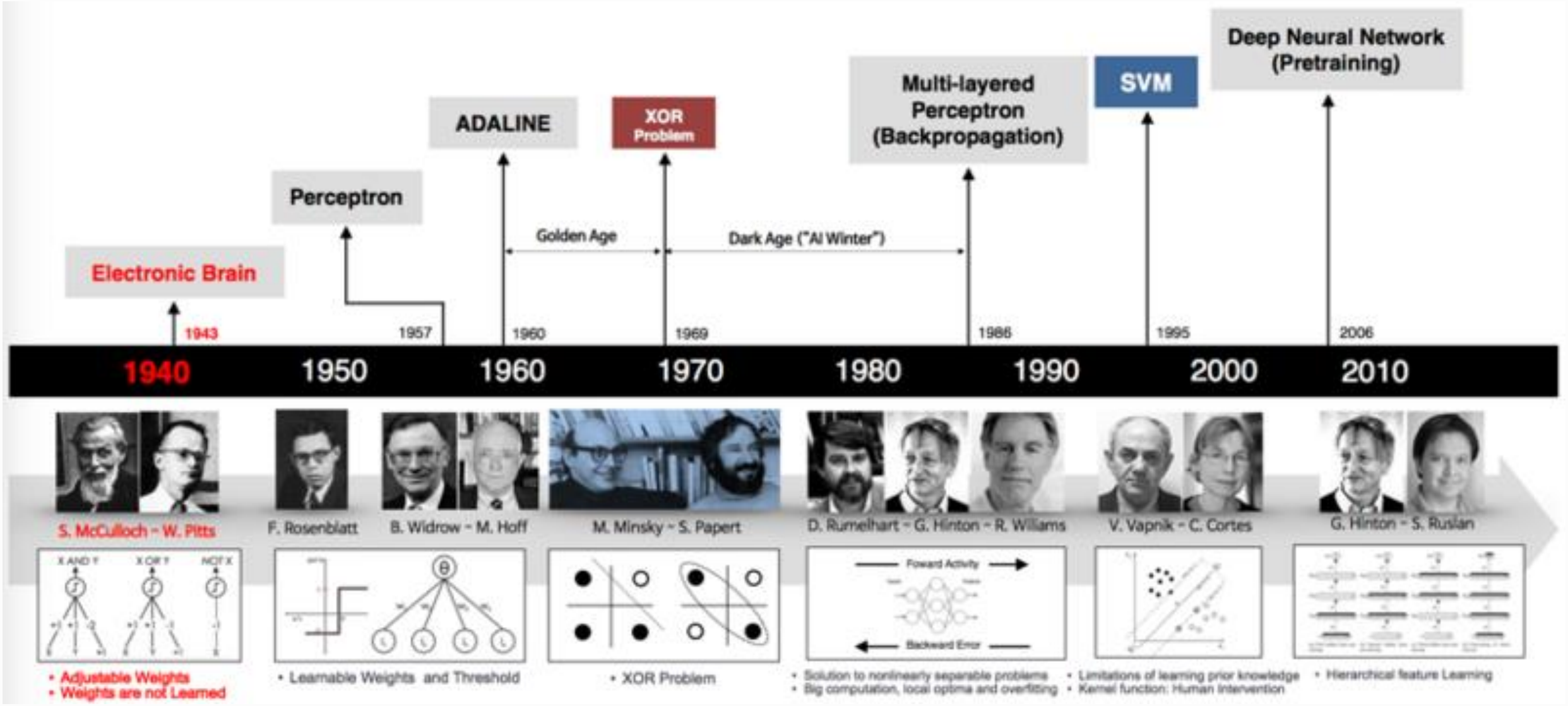
- 1950 -- ~1990s → Write many rules
- 1990 -- ~2000 → Corpus-based statistics
- 2000 -- 2013 → Supervised machine learning
- 2013 -- today → “Deep Learning!”
  - 2013-2018→ MLP(FFNN)/CNN/RNN
  - 2018 → Transformers (Attention-based models)

# A Brief History of NLP

## Where are we on the timeline?

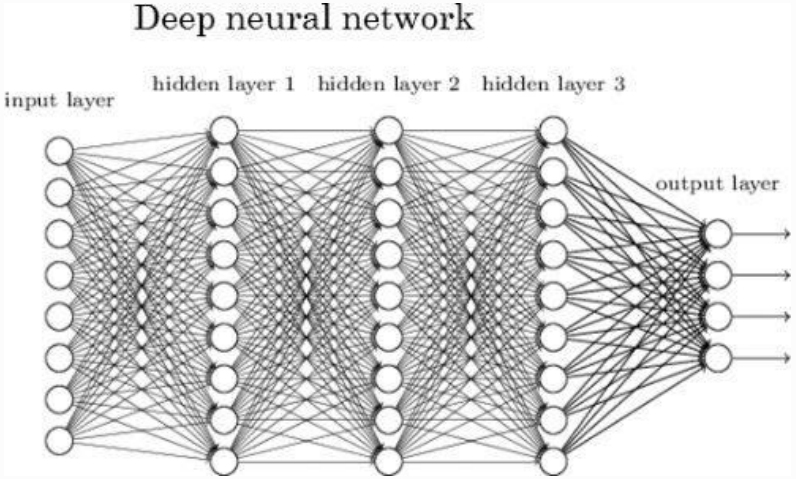
- 1950 -- ~1990s → Write many rules
- 1990 -- ~2000 → Corpus-based statistics
- 2000 -- 2013 → Supervised machine learning
- **2013 -- today → “Deep Learning!”**
  - 2013-2018→ **MLP(FFNN)**/CNN/RNN
  - 2018 → Transformers (Attention-based models)

# Another Historical Note



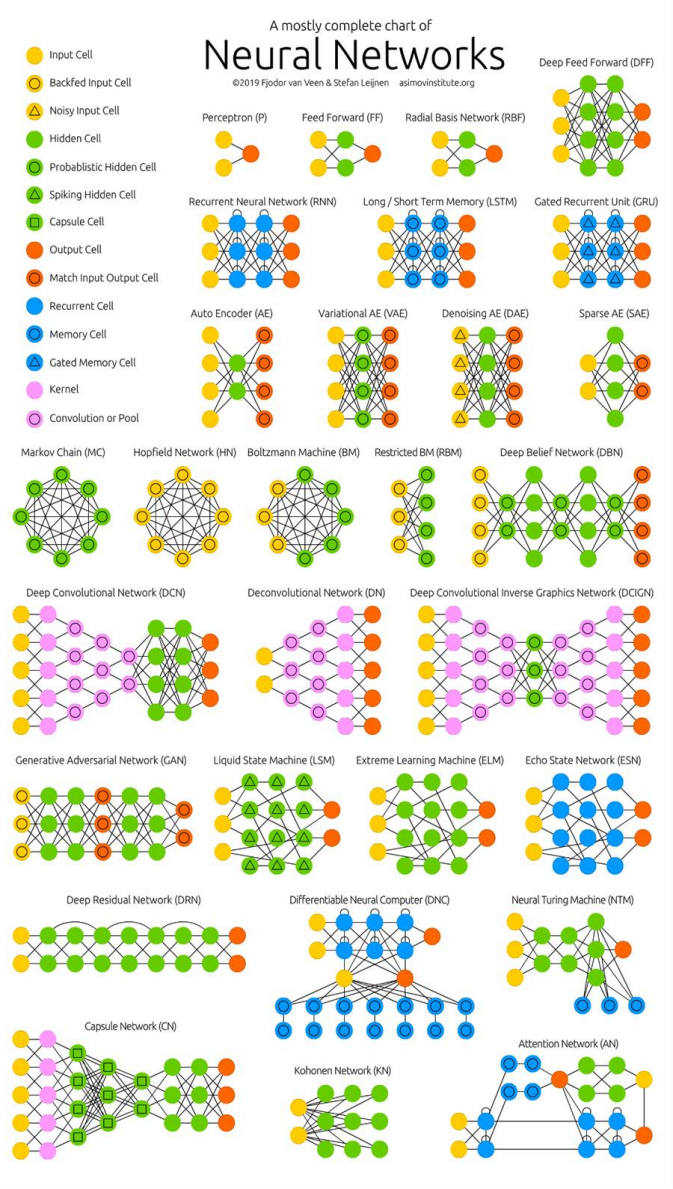
# Deep Neural Networks (DNNs)

- Neural networks are a type of **architecture**, i.e. a family of functions.
- As before, when learning we search for the best **network parameters** to minimize the empirical loss.
- This is done using some gradient descent variation.

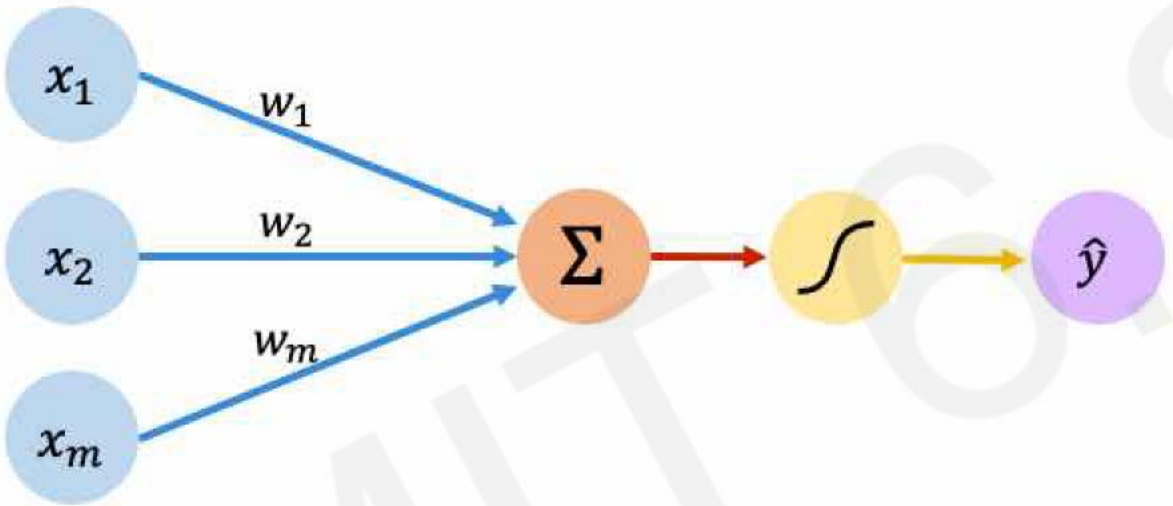


# Deep Neural Networks (DNNs)

- Neural networks, as the name suggests, are made up of stacked **neurons**.
- Neuron -> Thing the holds a (real) number
- This number depends on the example/text you feed it
- Has input and computes something w.r.t to input
- -> a function



# The Perceptron: Forward Propagation




Inputs Weights Sum Non-Linearity Output

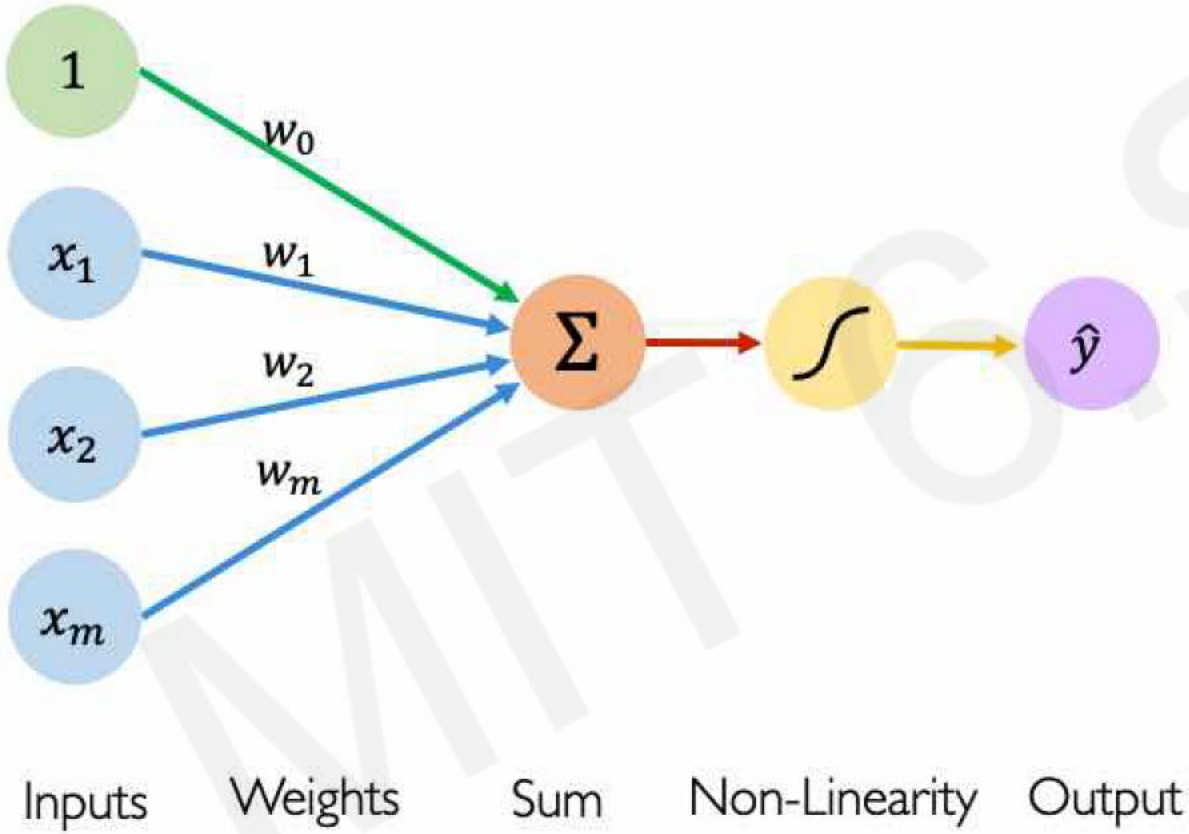
Output

Linear combination of inputs

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

Non-linear activation function

# The Perceptron: Forward Propagation




Output

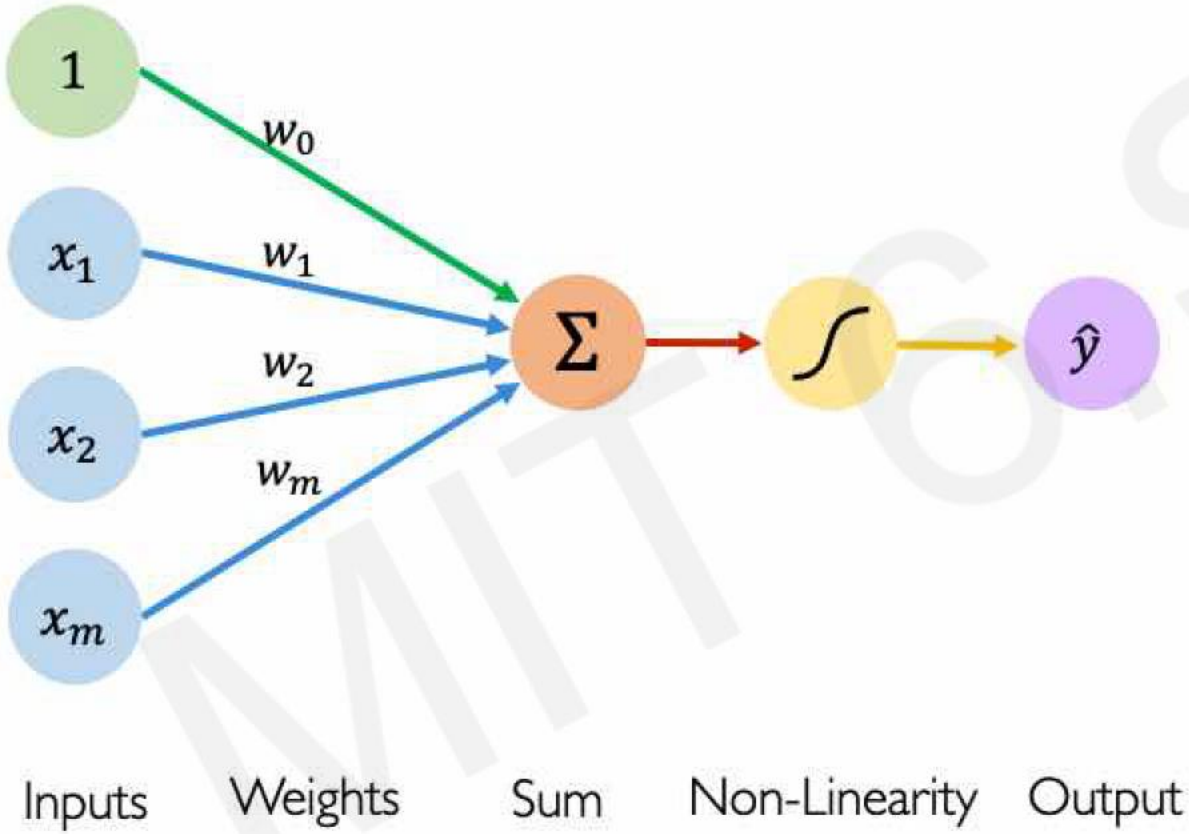
Linear combination of inputs

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

Non-linear activation function

Bias

# The Perceptron: Forward Propagation

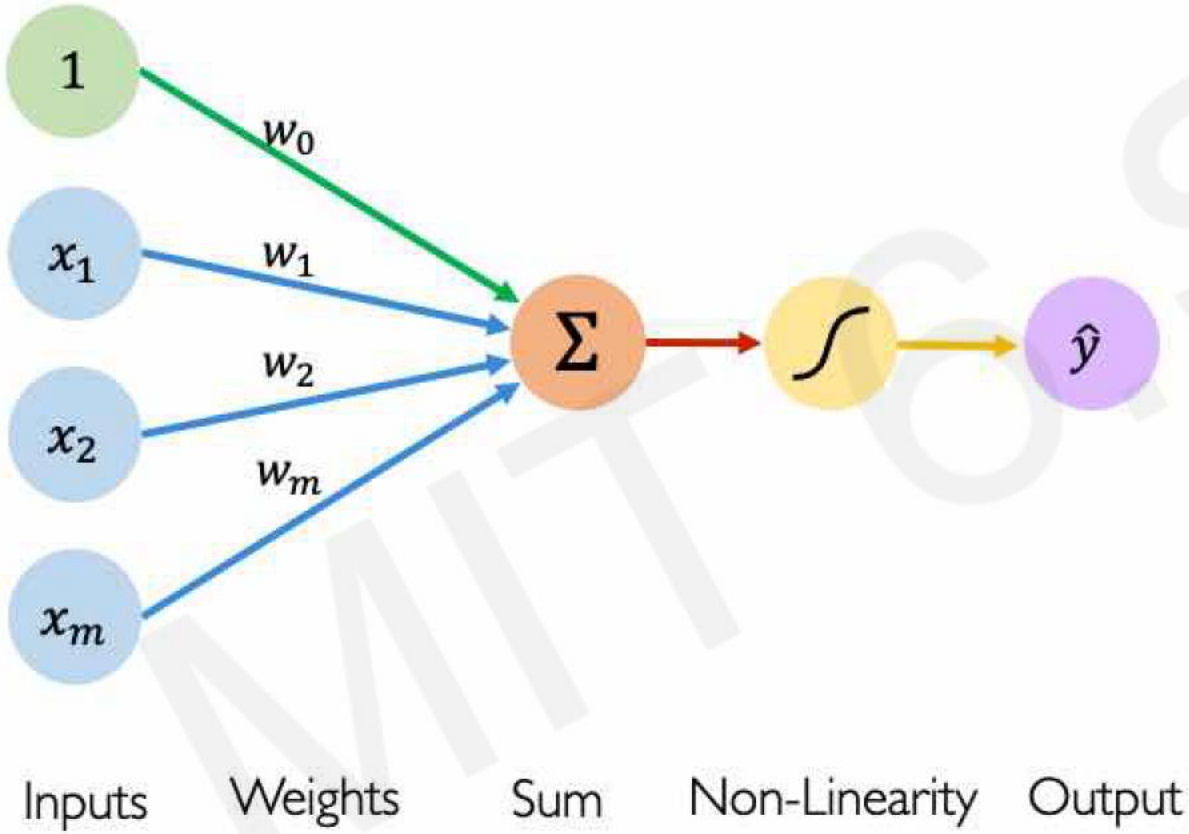



$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$

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

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

# The Perceptron: Forward Propagation

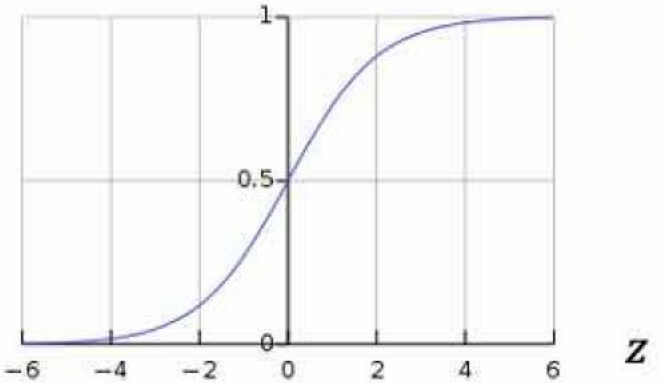



## Activation Functions

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

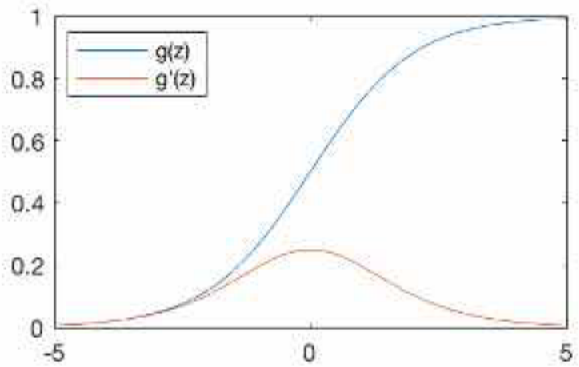
- Example: sigmoid function

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



# Common Activation Functions

## Sigmoid Function

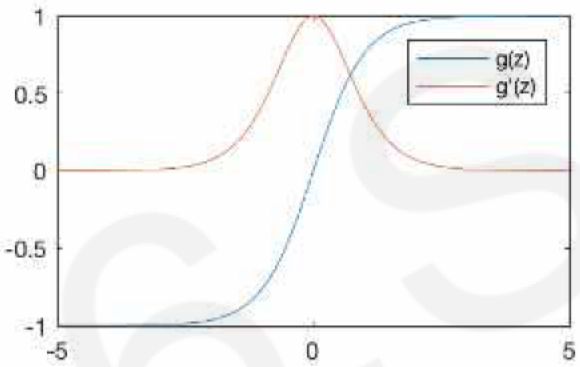



$$g(z) = \frac{1}{1 + e^{-z}}$$

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

`tf.math.sigmoid(z)`

## Hyperbolic Tangent

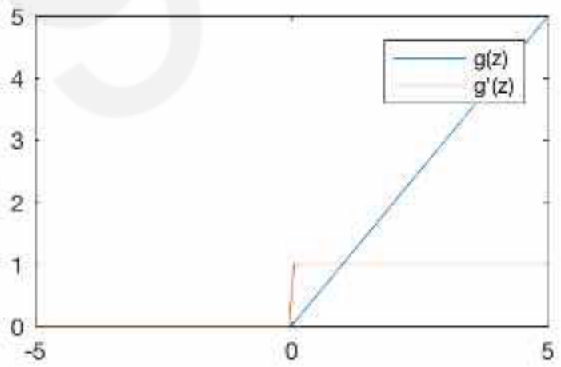



$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

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

`tf.math.tanh(z)`

## Rectified Linear Unit (ReLU)




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

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

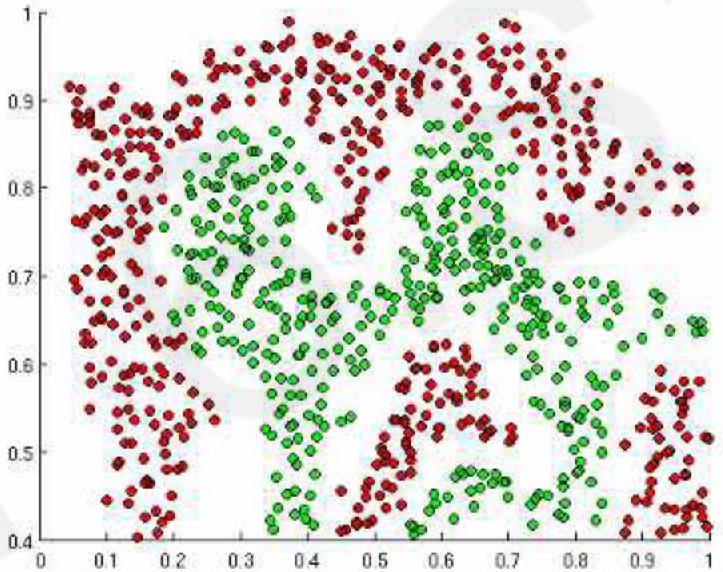
`tf.nn.relu(z)`

TensorFlow code blocks

NOTE: All activation functions are non-linear

# Importance of Activation Functions

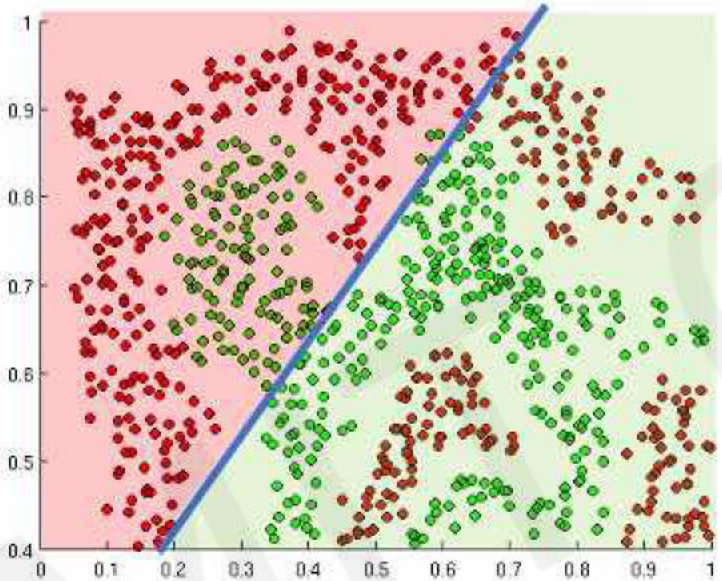
*The purpose of activation functions is to* ***introduce non-linearities*** *into the network*




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

# Importance of Activation Functions

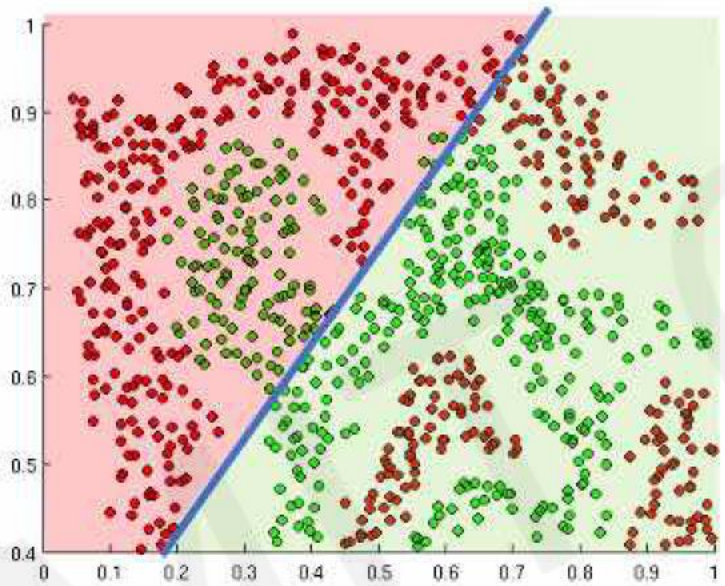
*The purpose of activation functions is to* ***introduce non-linearities*** *into the network*



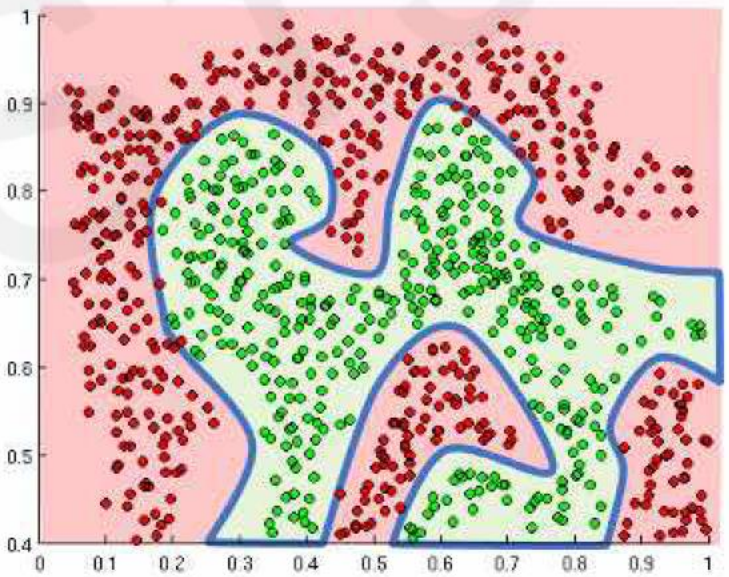

Linear activation functions produce linear decisions no matter the network size

# Importance of Activation Functions

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

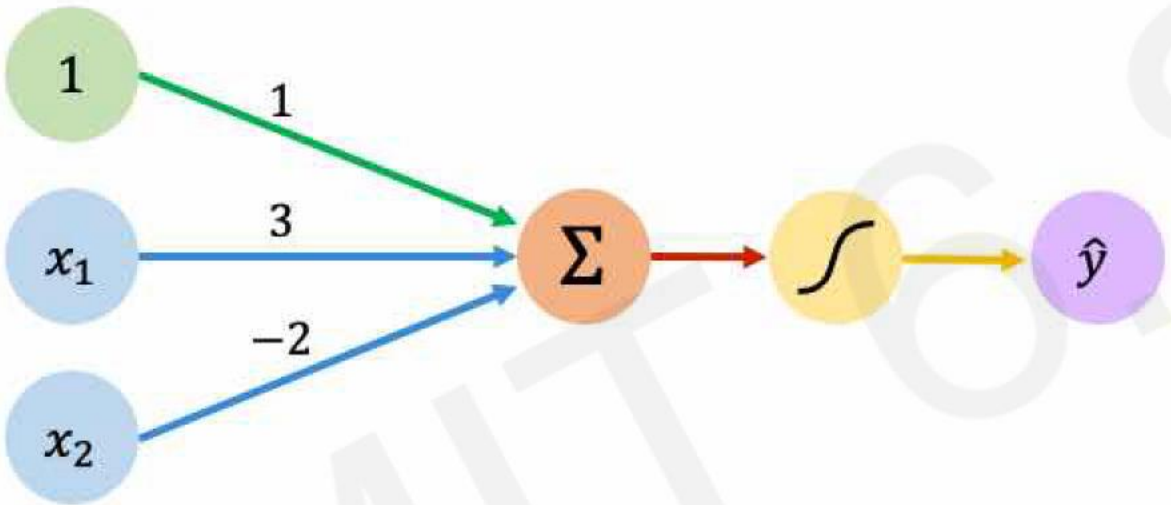



Linear activation functions produce linear decisions no matter the network size




Non-linearities allow us to approximate arbitrarily complex functions

# The Perceptron: Example



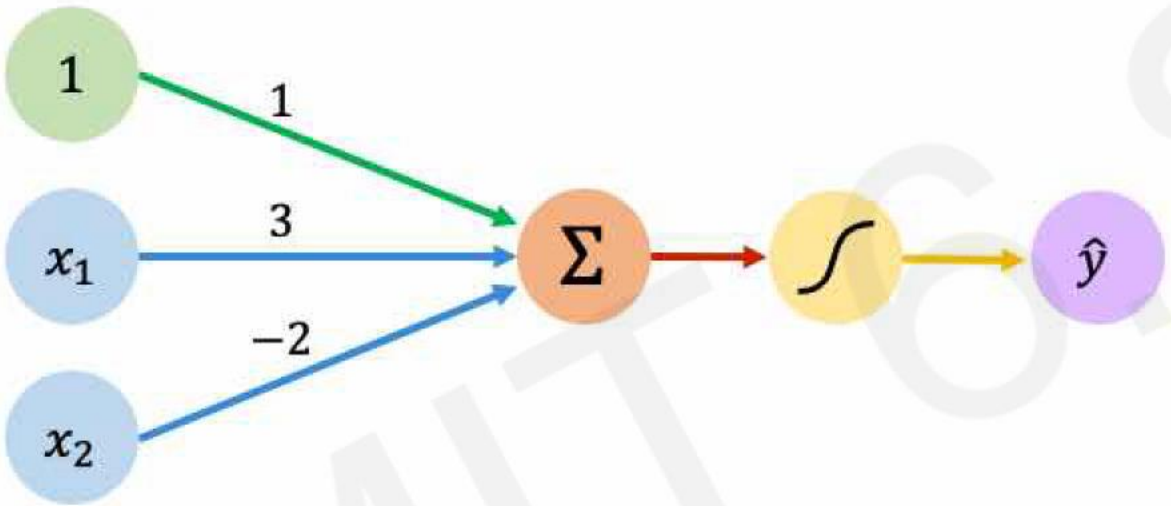

We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

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

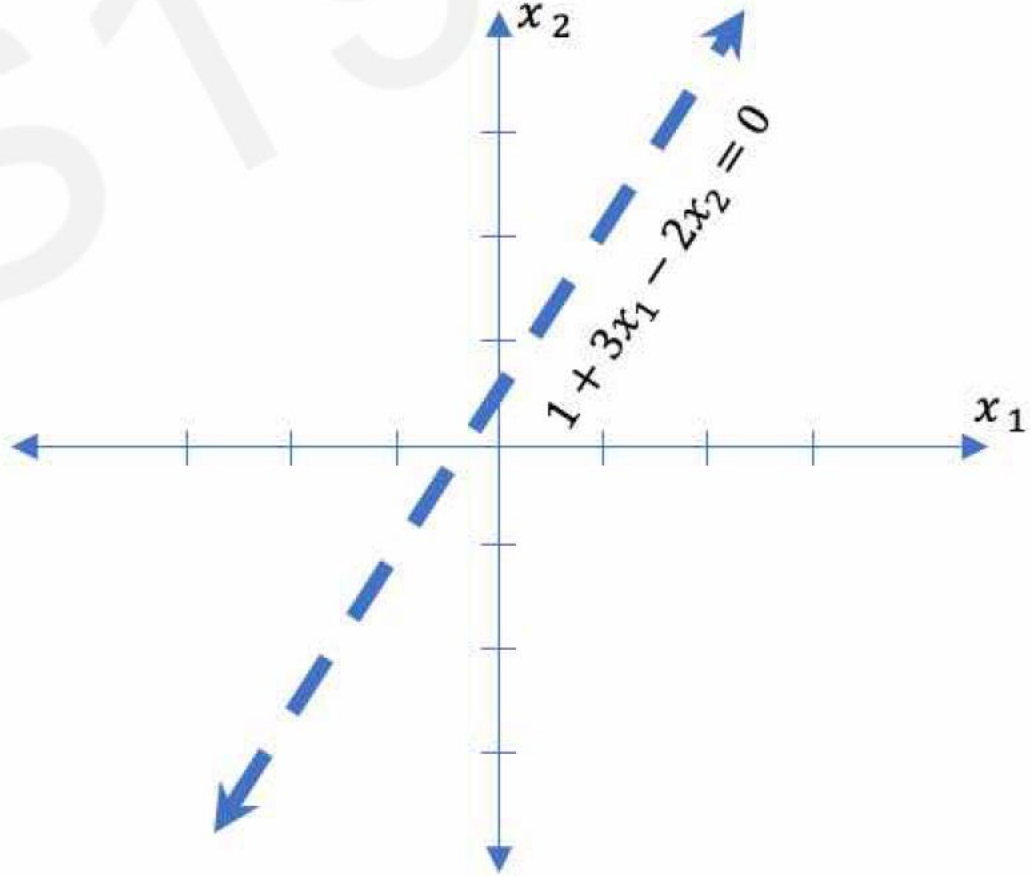
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$

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

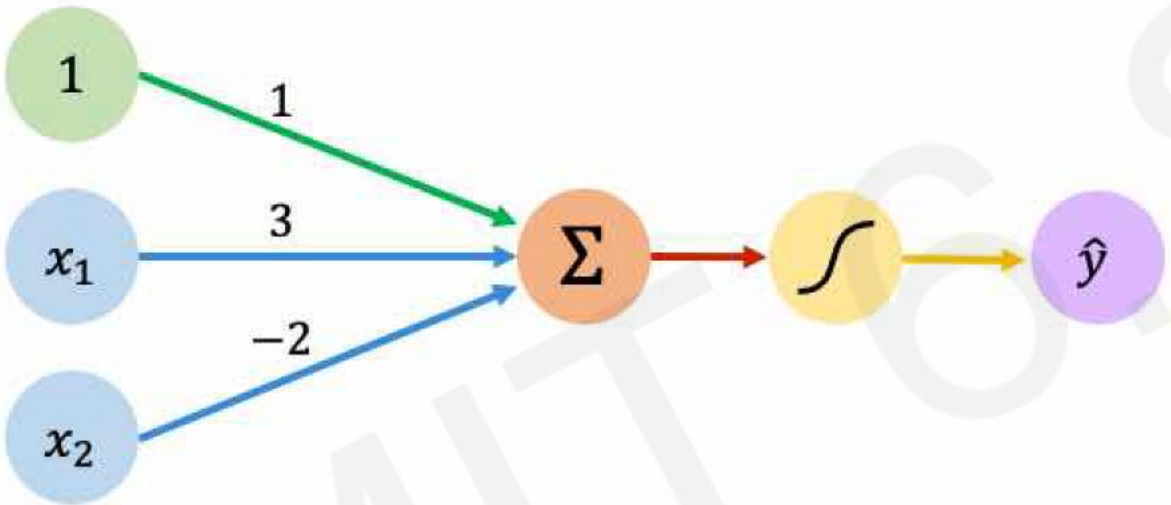
# The Perceptron: Example



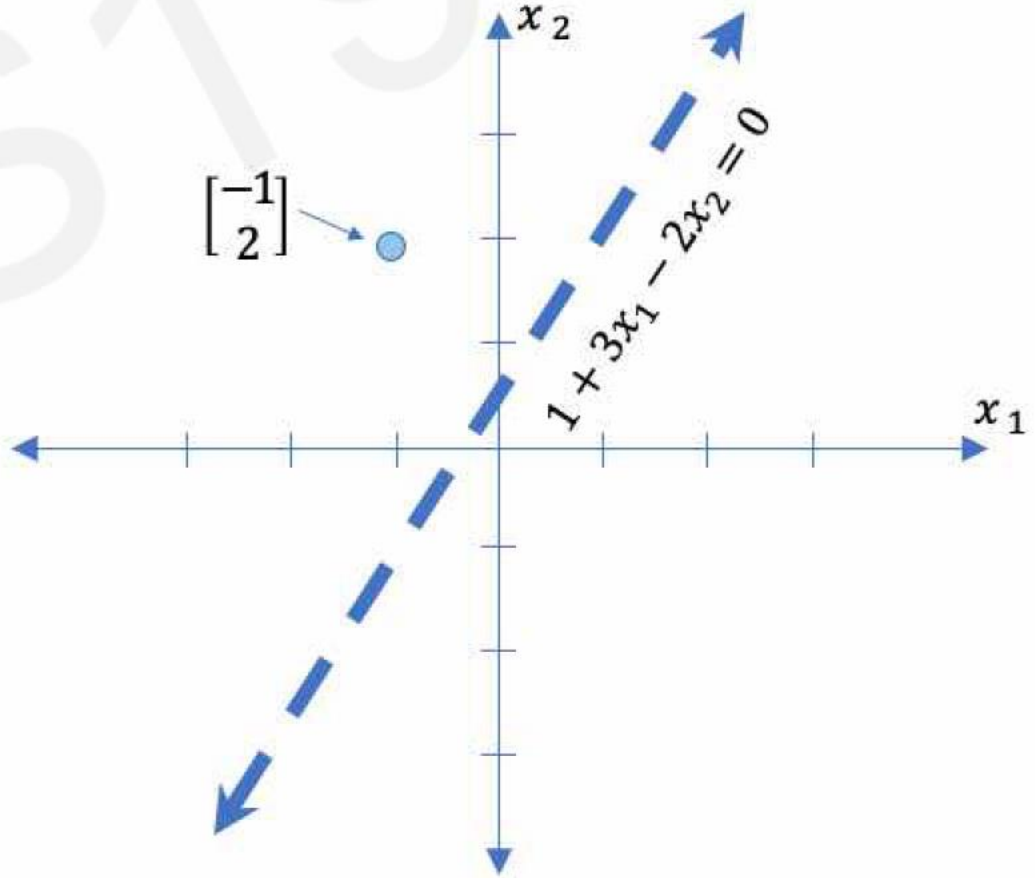

$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



# The Perceptron: Example



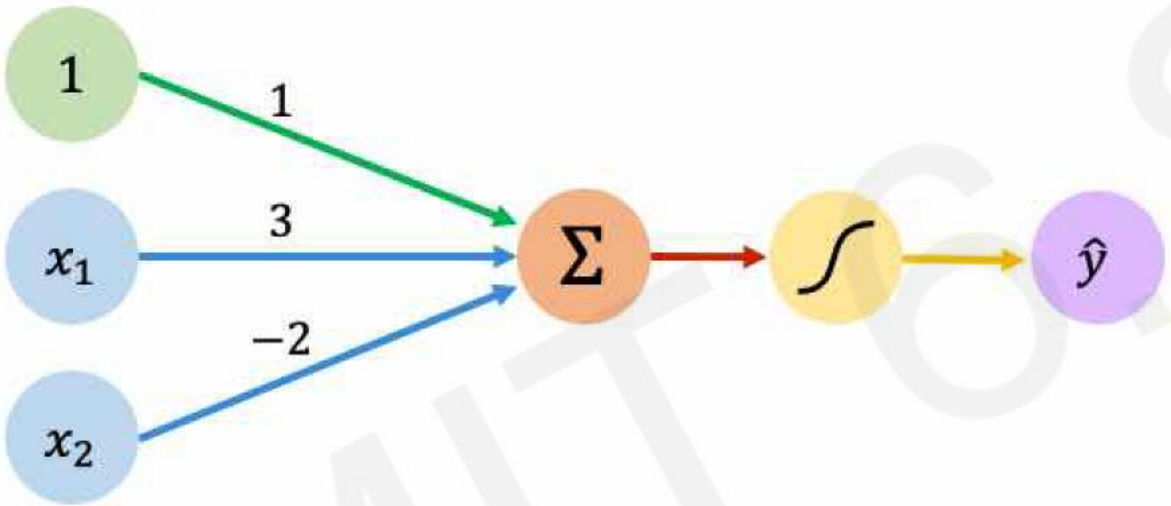

$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



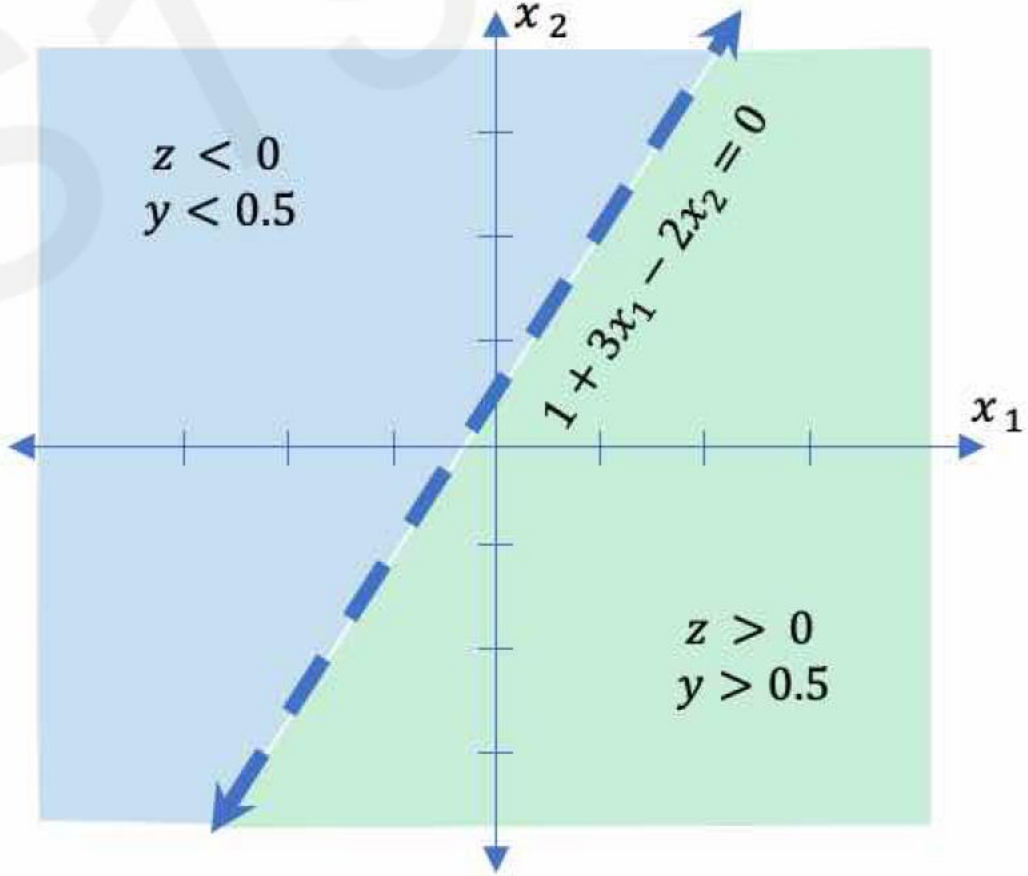

Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

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

# The Perceptron: Example



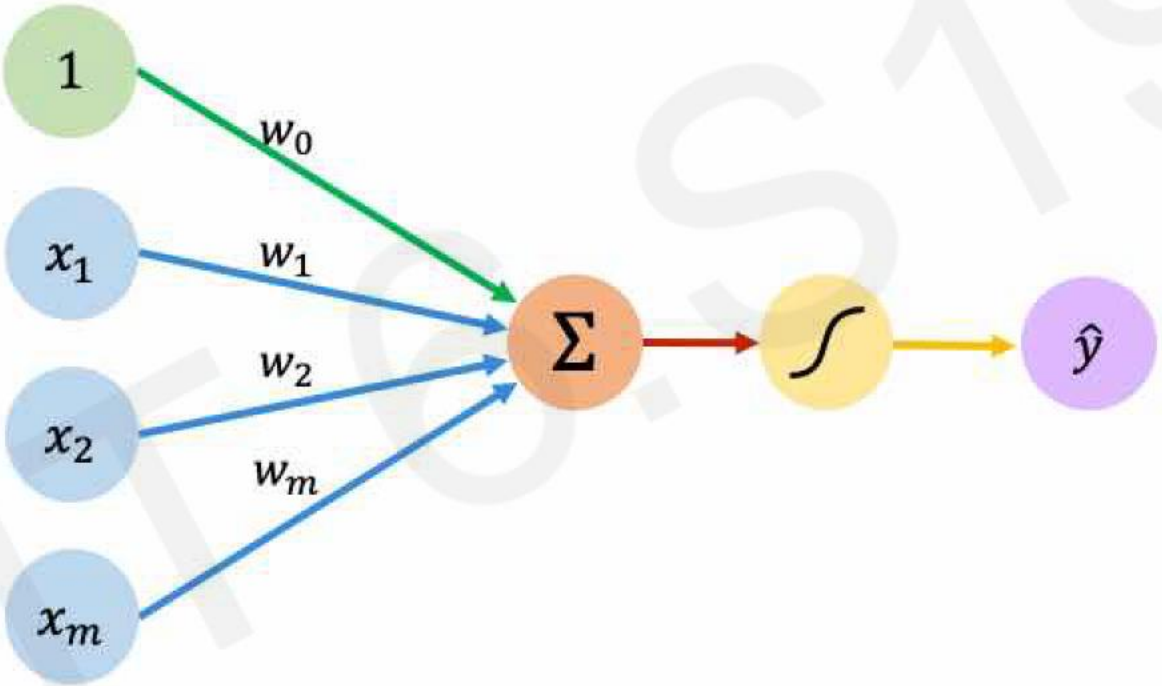

$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



# Feed Forward Neural Networks

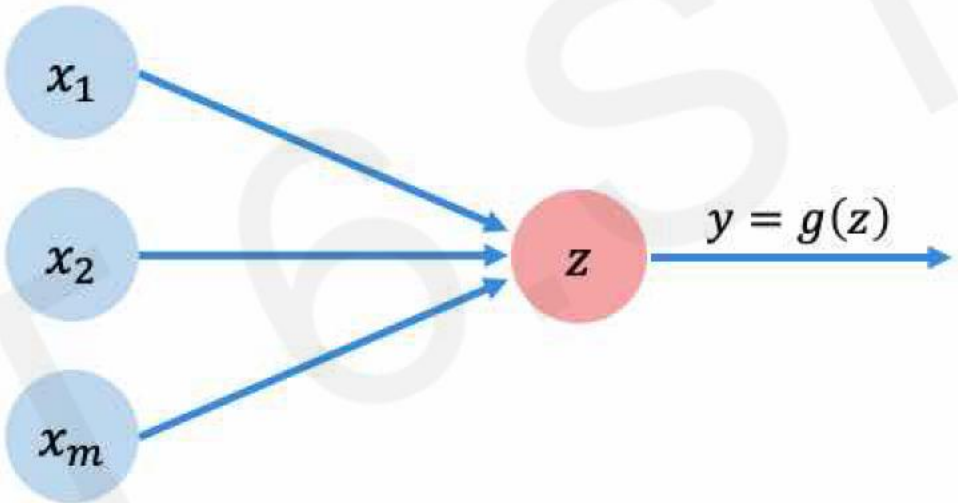
# The Perceptron: Simplified

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




Inputs Weights Sum Non-Linearity Output

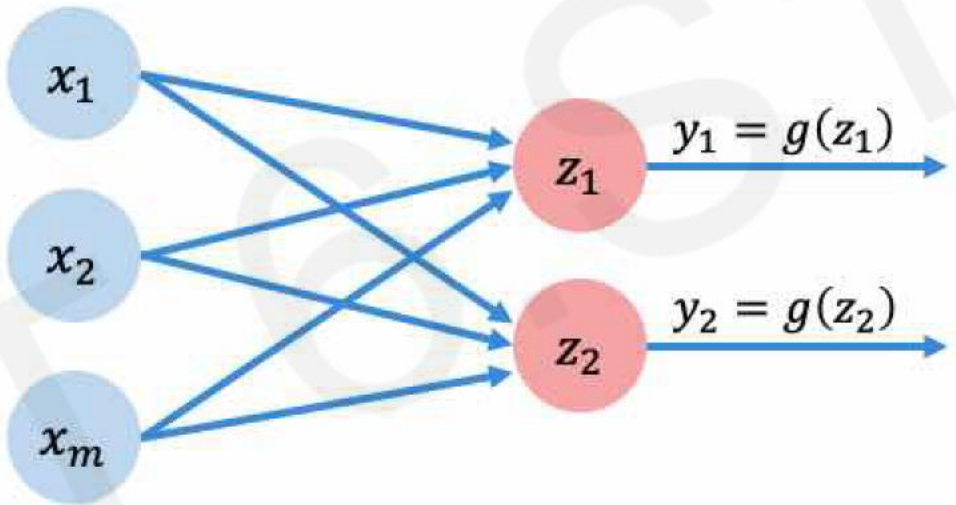
# The Perceptron: Simplified




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

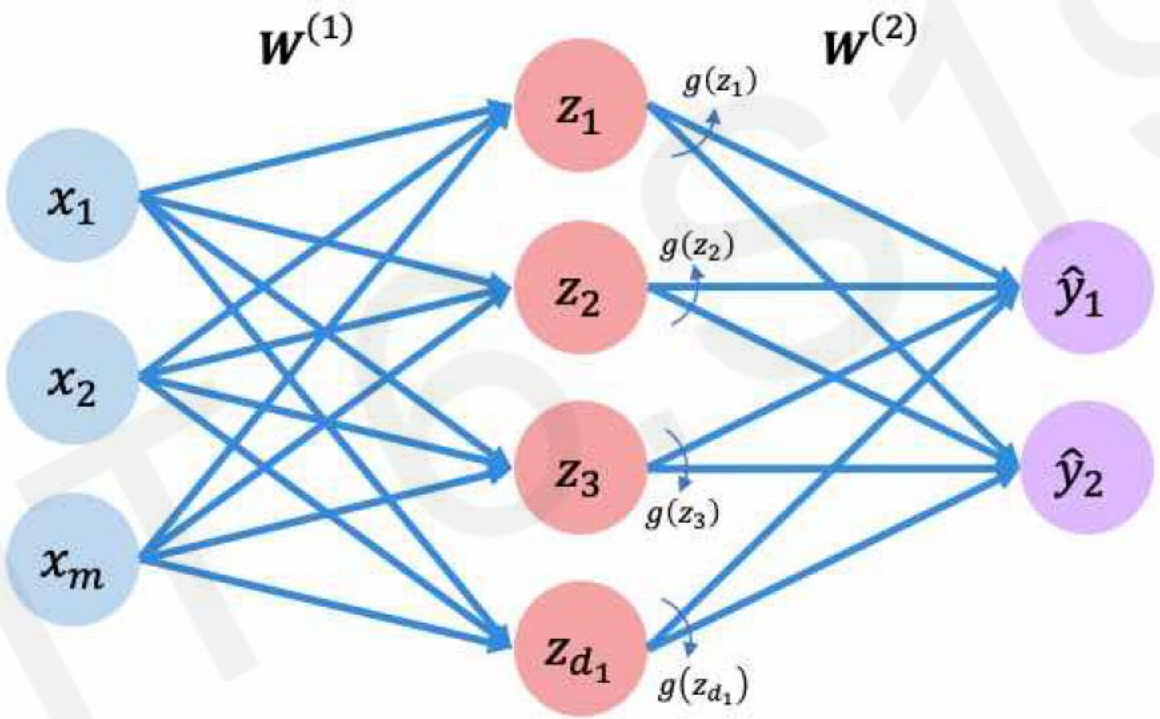
# Multi Output Perceptron

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



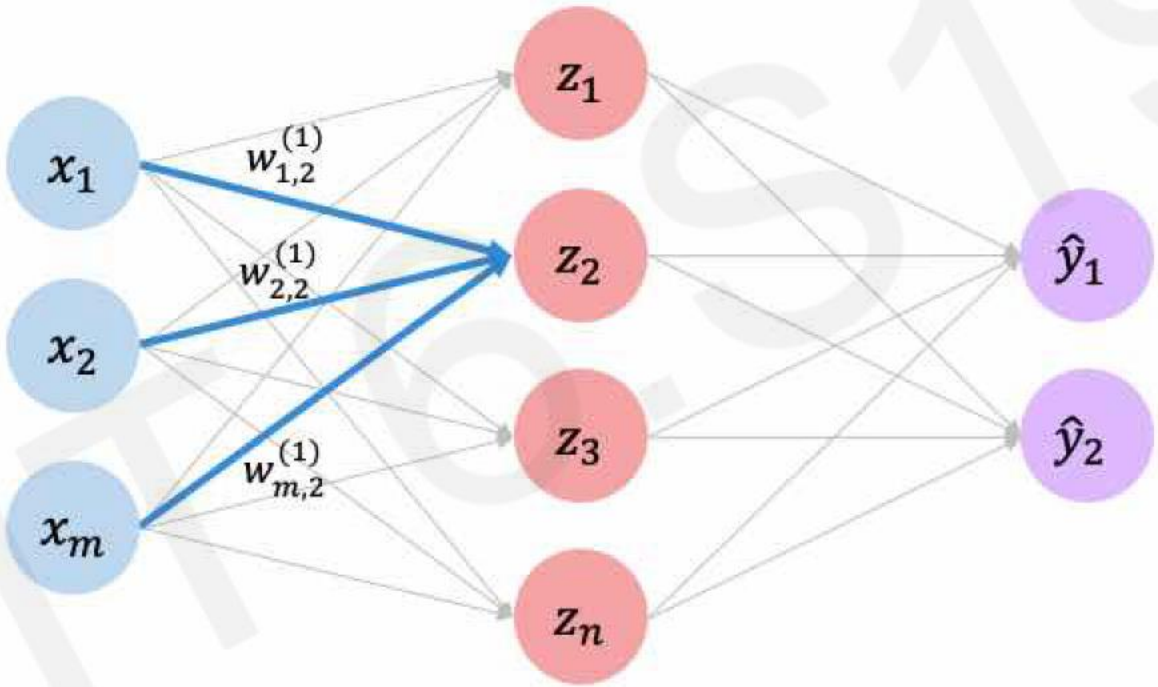

$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

# Single Layer Neural Network




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

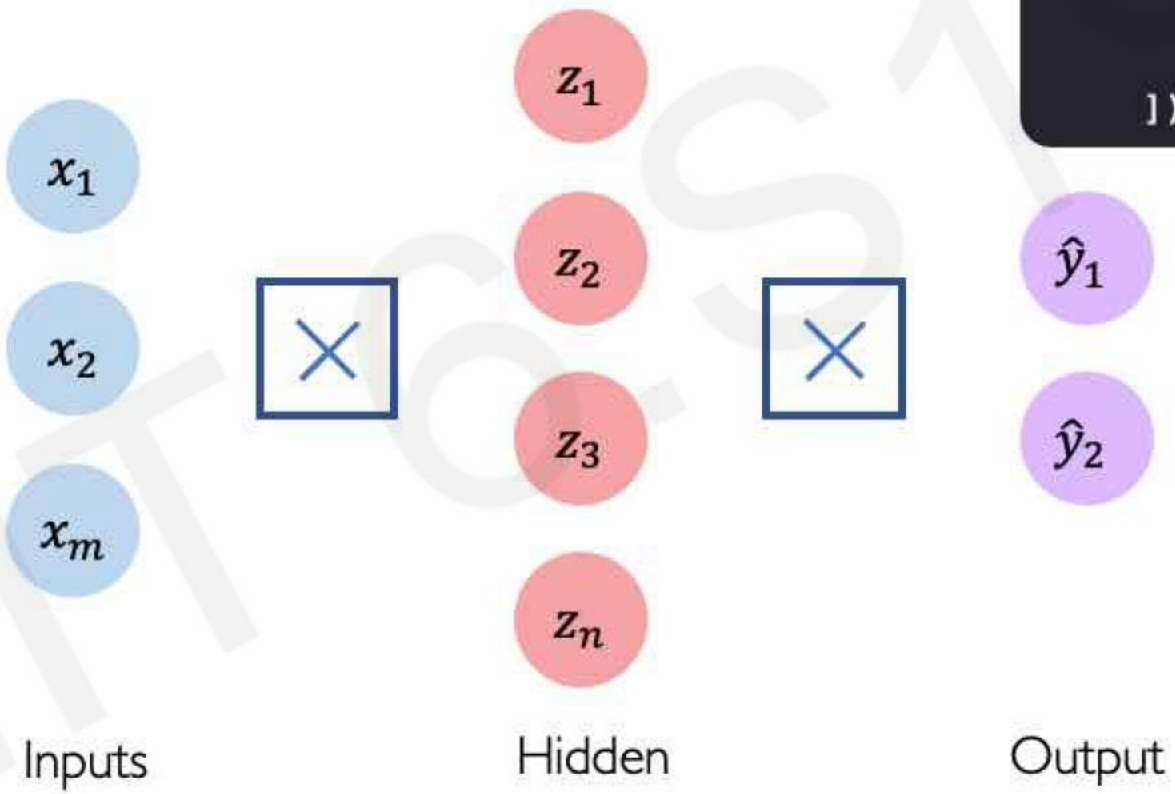
# Single Layer Neural Network




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

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

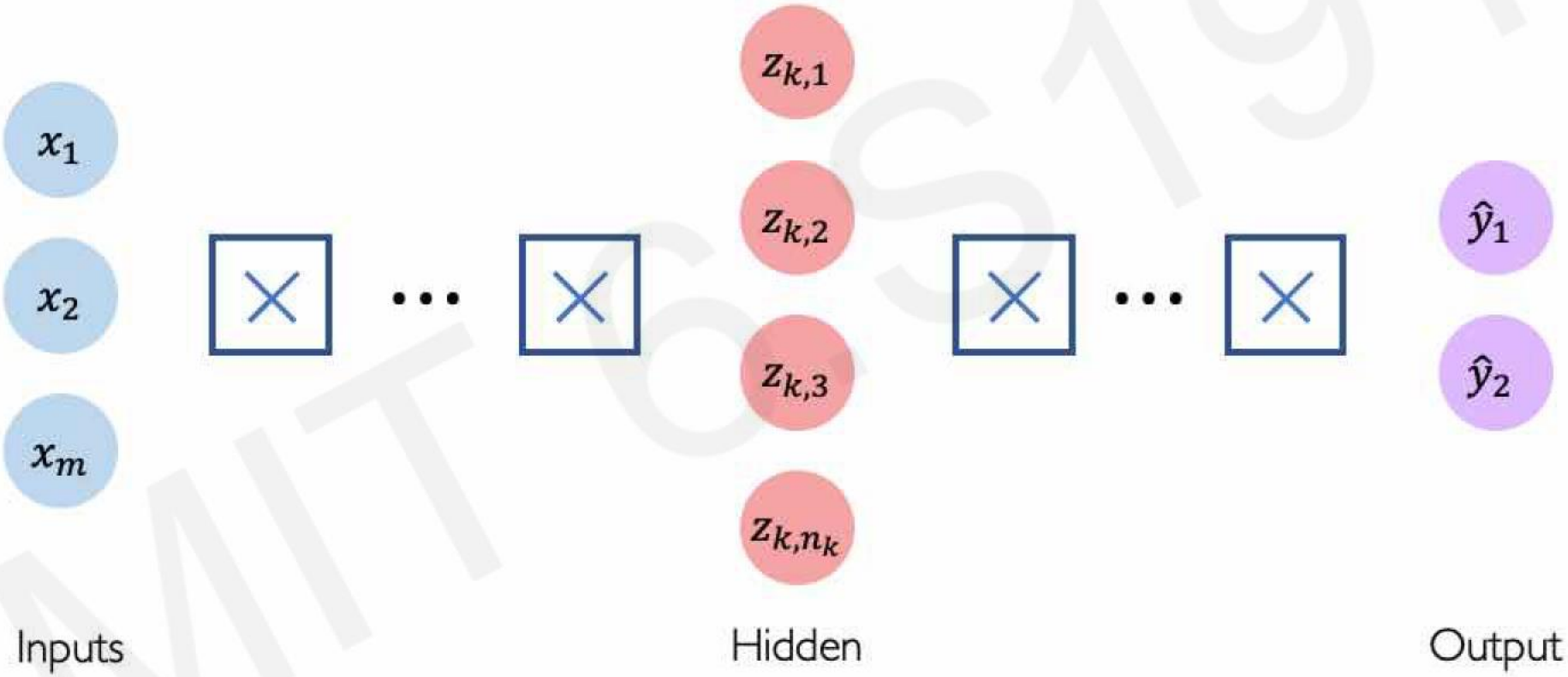
# Multi Output Perceptron




```tensorflow
import tensorflow as tf

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

# Deep Neural Network




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

# Applying Neural Networks

# Example Problem

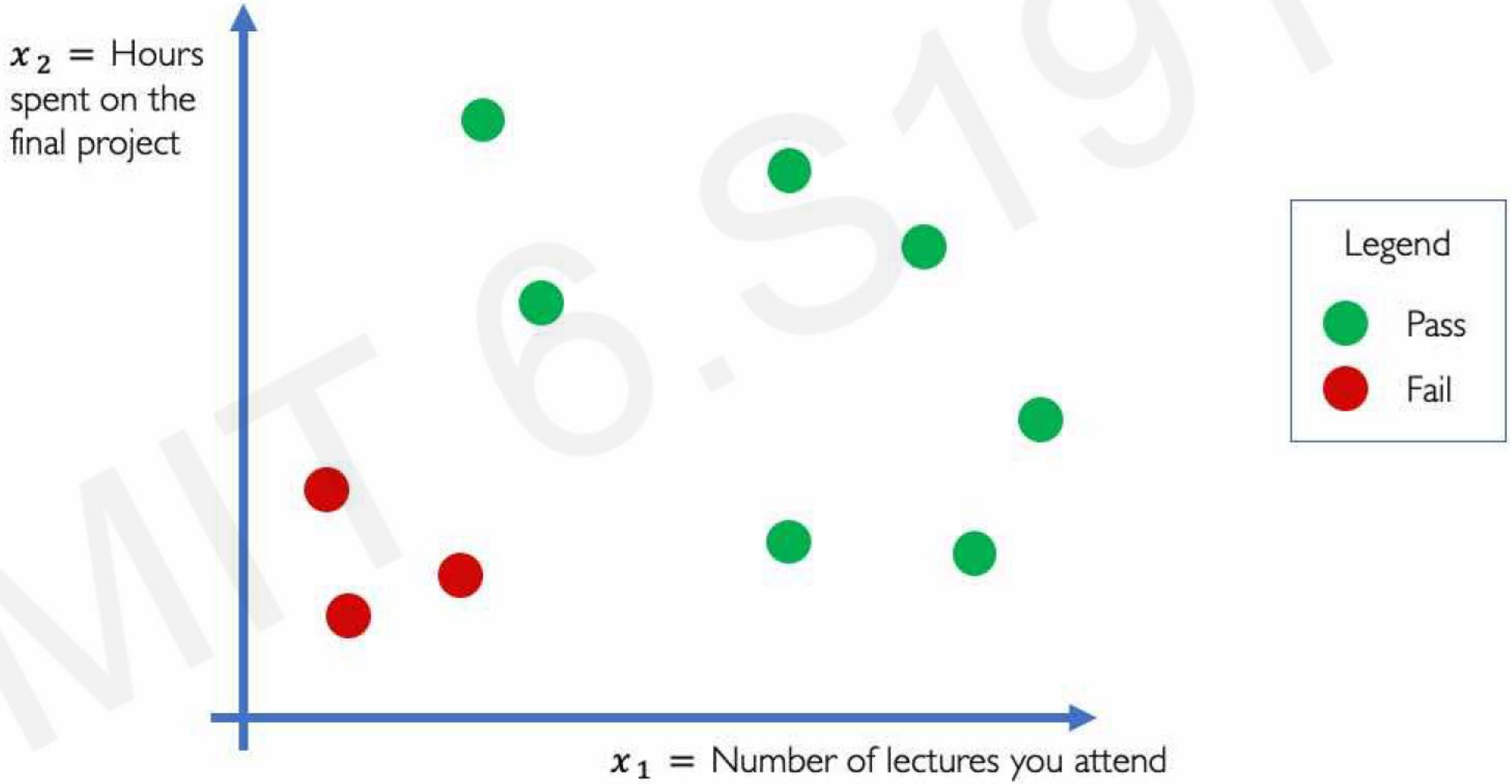
Will I pass this class?

Let's start with a simple two feature model

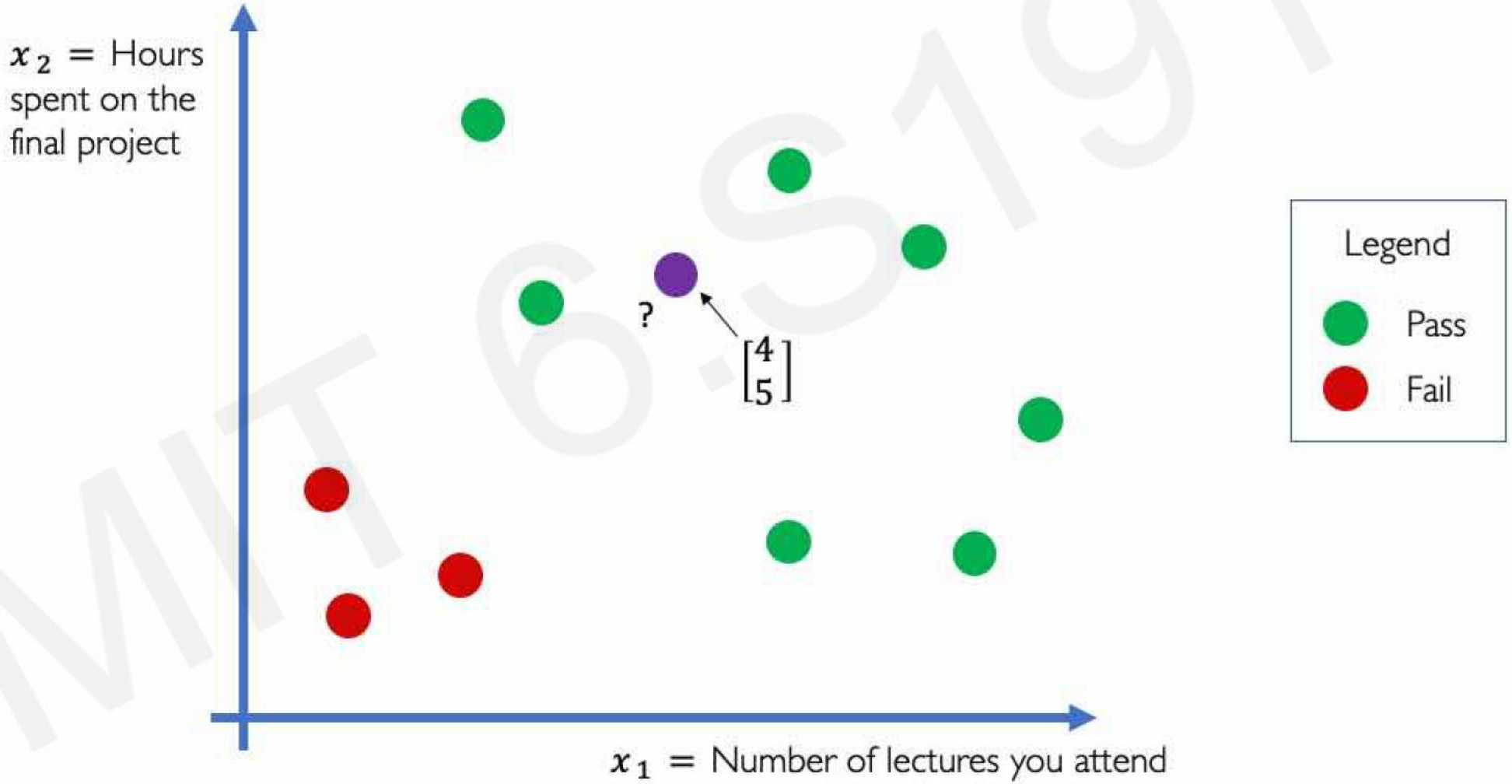
$x_1$ = Number of lectures you attend

$x_2$ = Hours spent on the final project

# Example Problem: Will I pass this class?

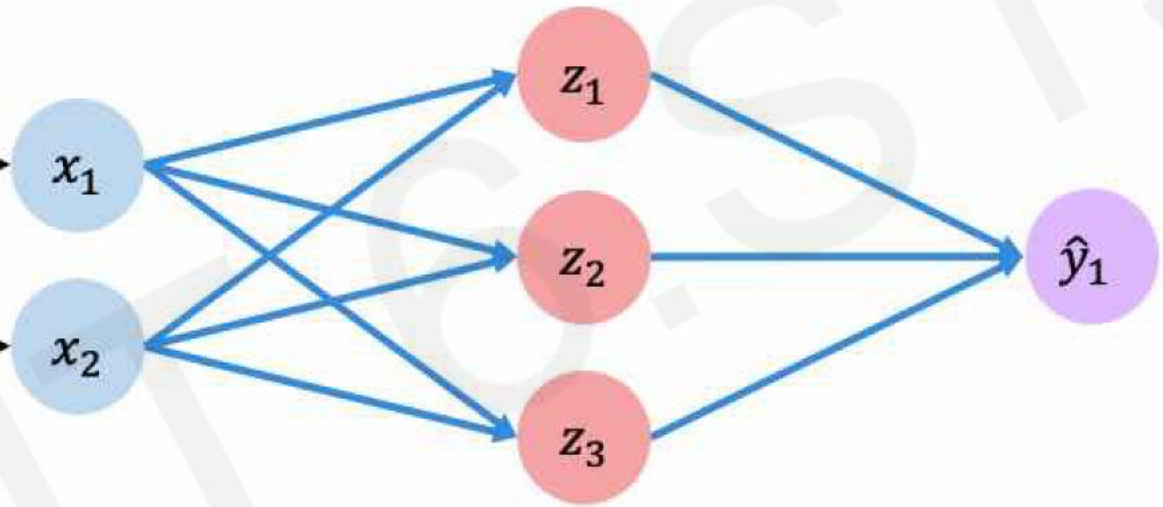


# Example Problem: Will I pass this class?



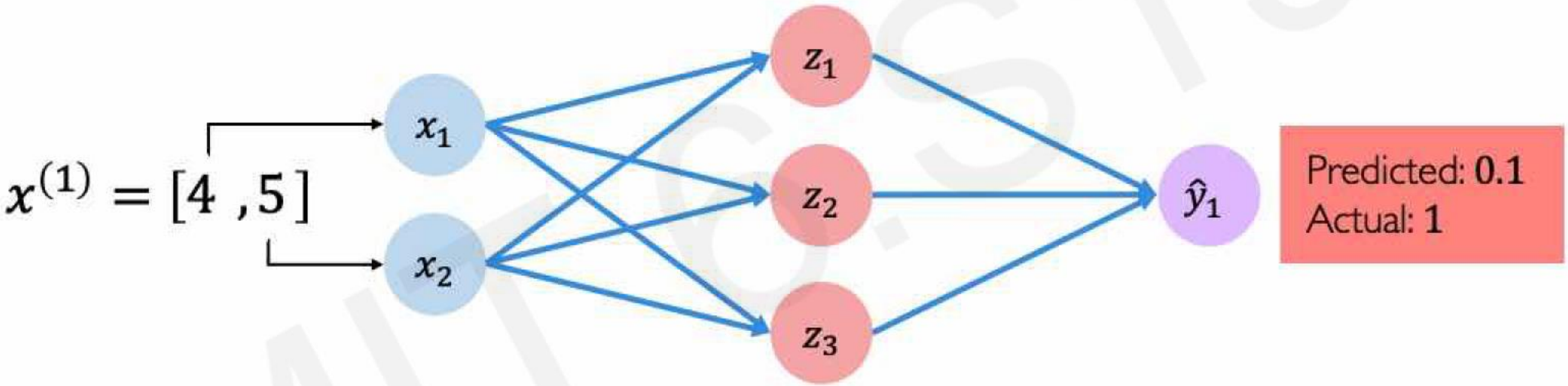
# Example Problem: Will I pass this class?

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



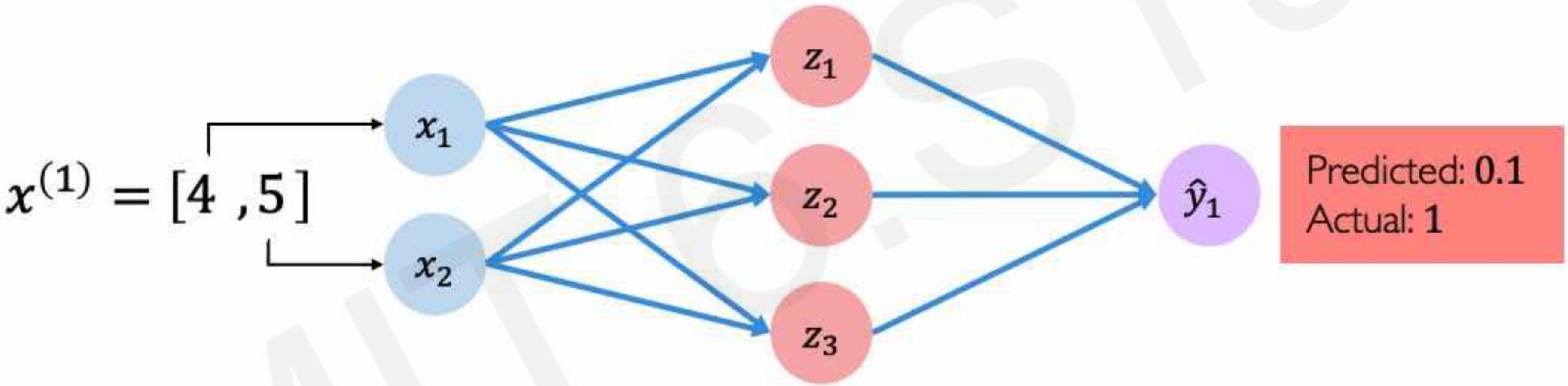

Predicted: 0.1

# Example Problem: Will I pass this class?



# Quantifying Loss

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



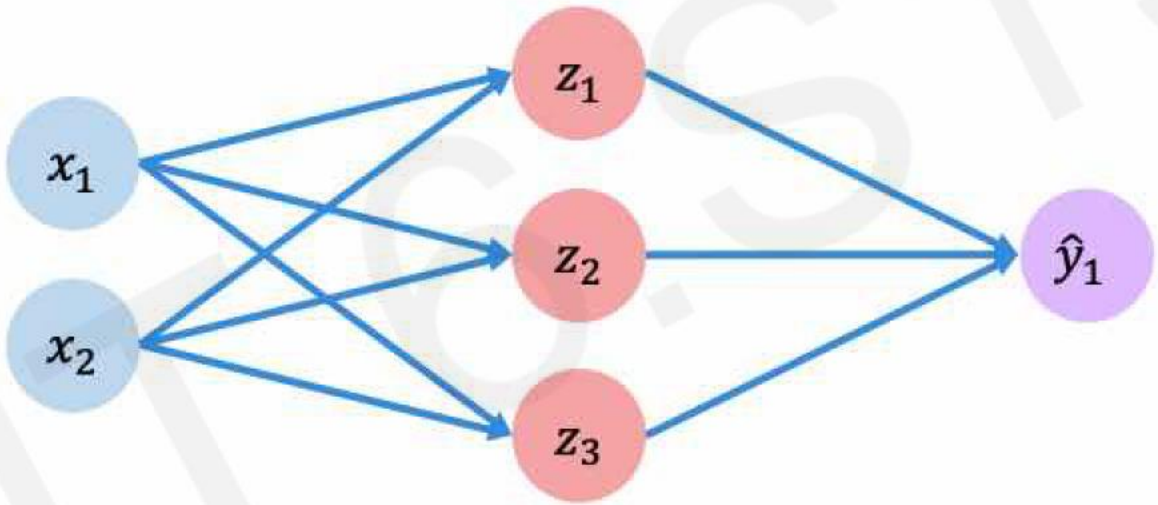

$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), \boldsymbol{y}^{(i)}\right)$$

Predicted Actual

# Empirical Loss

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

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




$f(x)$ $y$

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

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

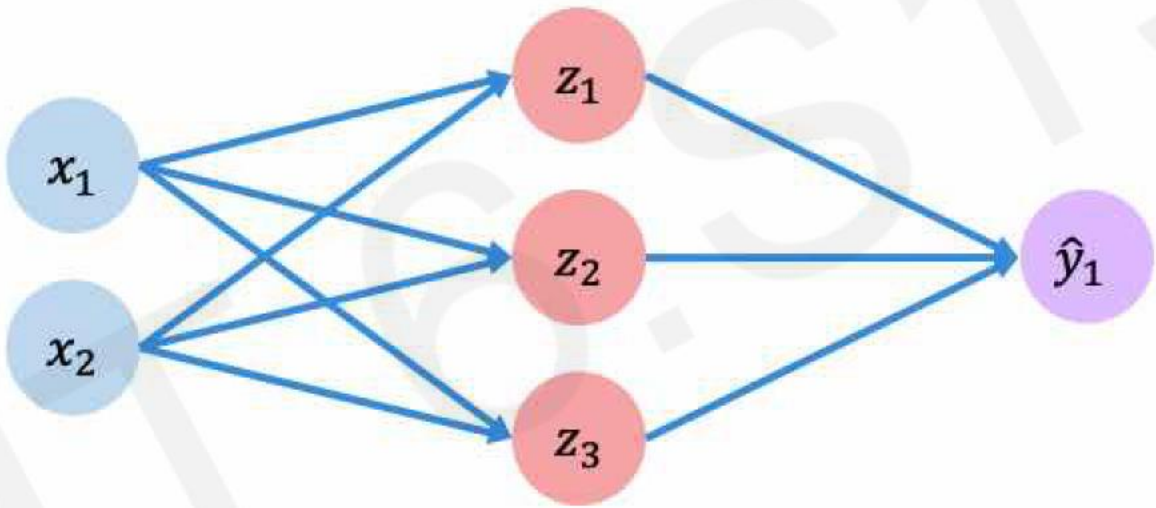
Predicted Actual

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*

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




$f(x)$ $y$

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

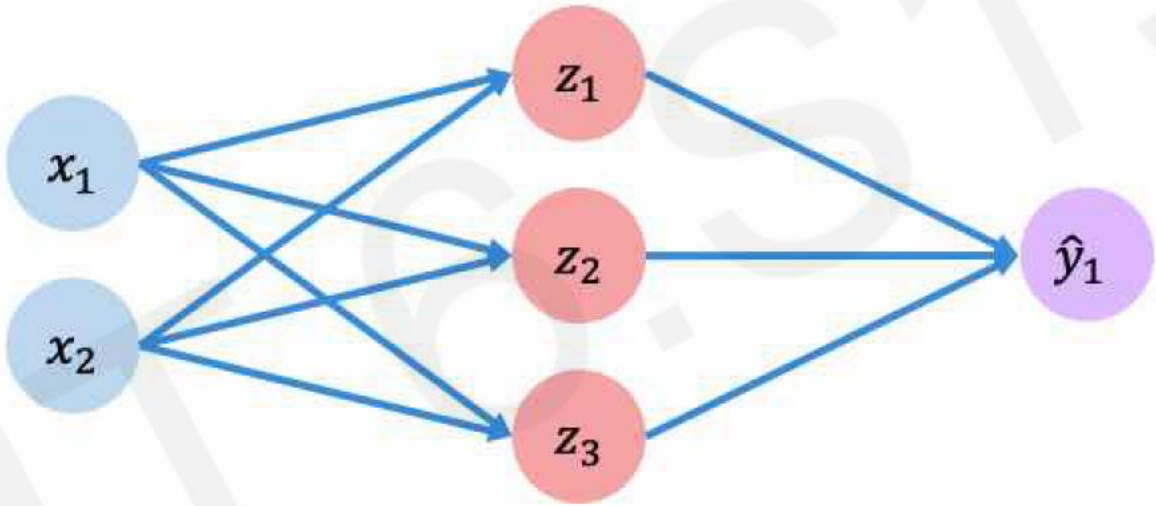
$$J(\boldsymbol{W}) = -\frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\Big(1 - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

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

# Mean Squared Error Loss

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

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




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

Actual Predicted

$f(x)$ $y$

$$\begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix} \begin{matrix} \times \\ \times \\ \checkmark \\ \end{matrix} \begin{bmatrix} 90 \\ 20 \\ 95 \\ \vdots \end{bmatrix}$$

Final Grades (percentage)

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

# The Backpropagation Algorithm

# Loss Optimization

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

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

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

# Loss Optimization

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

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

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

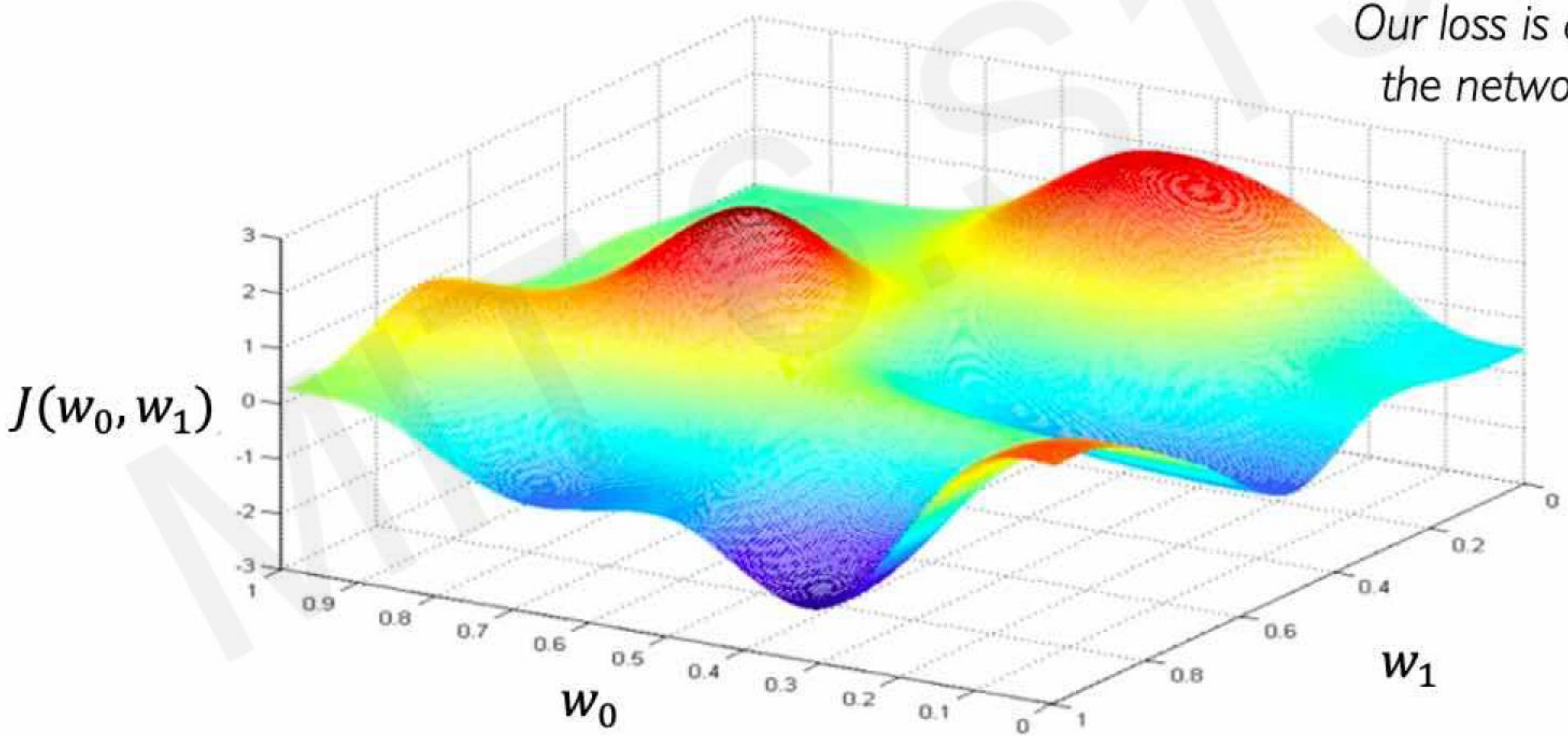
Remember:

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

# Loss Optimization

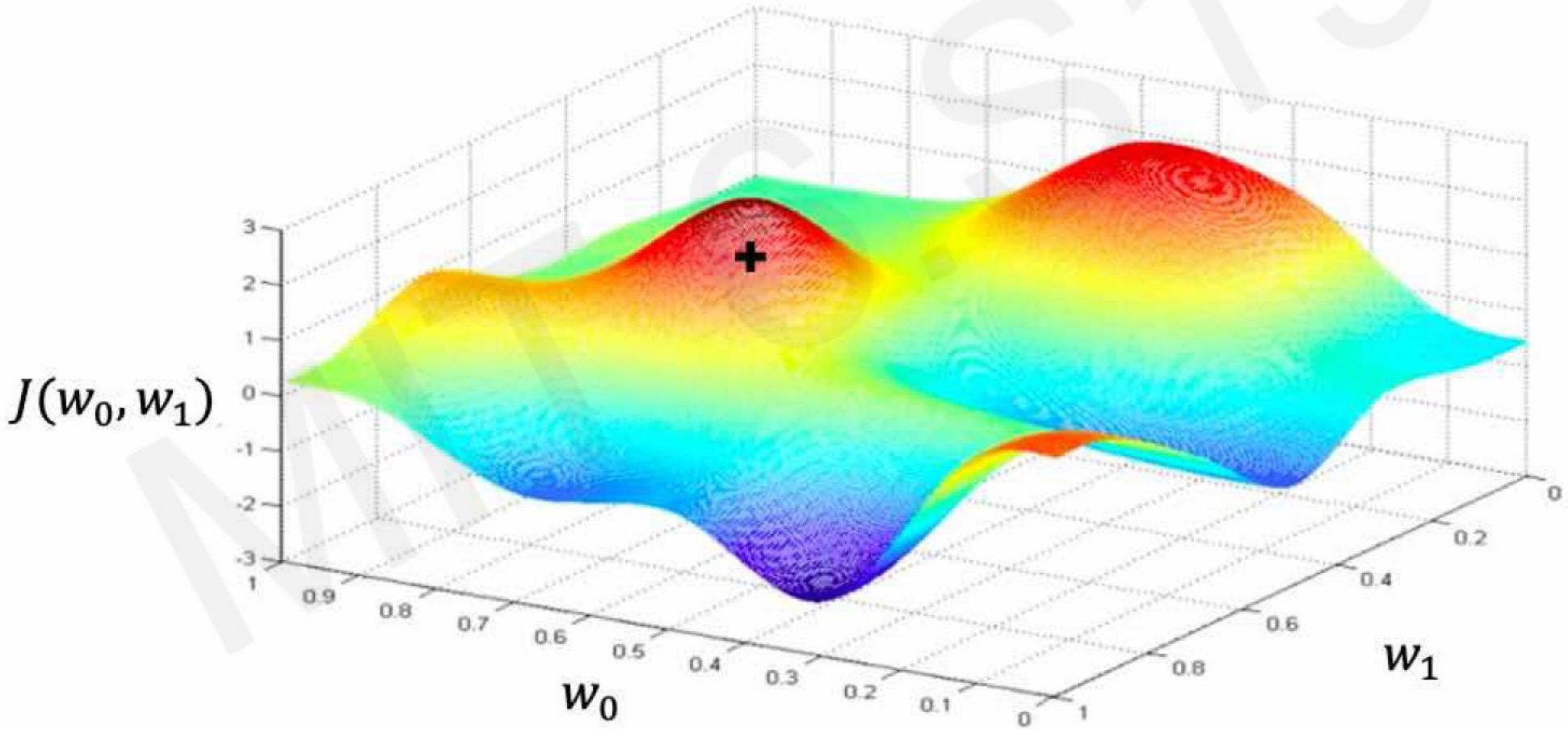
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

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



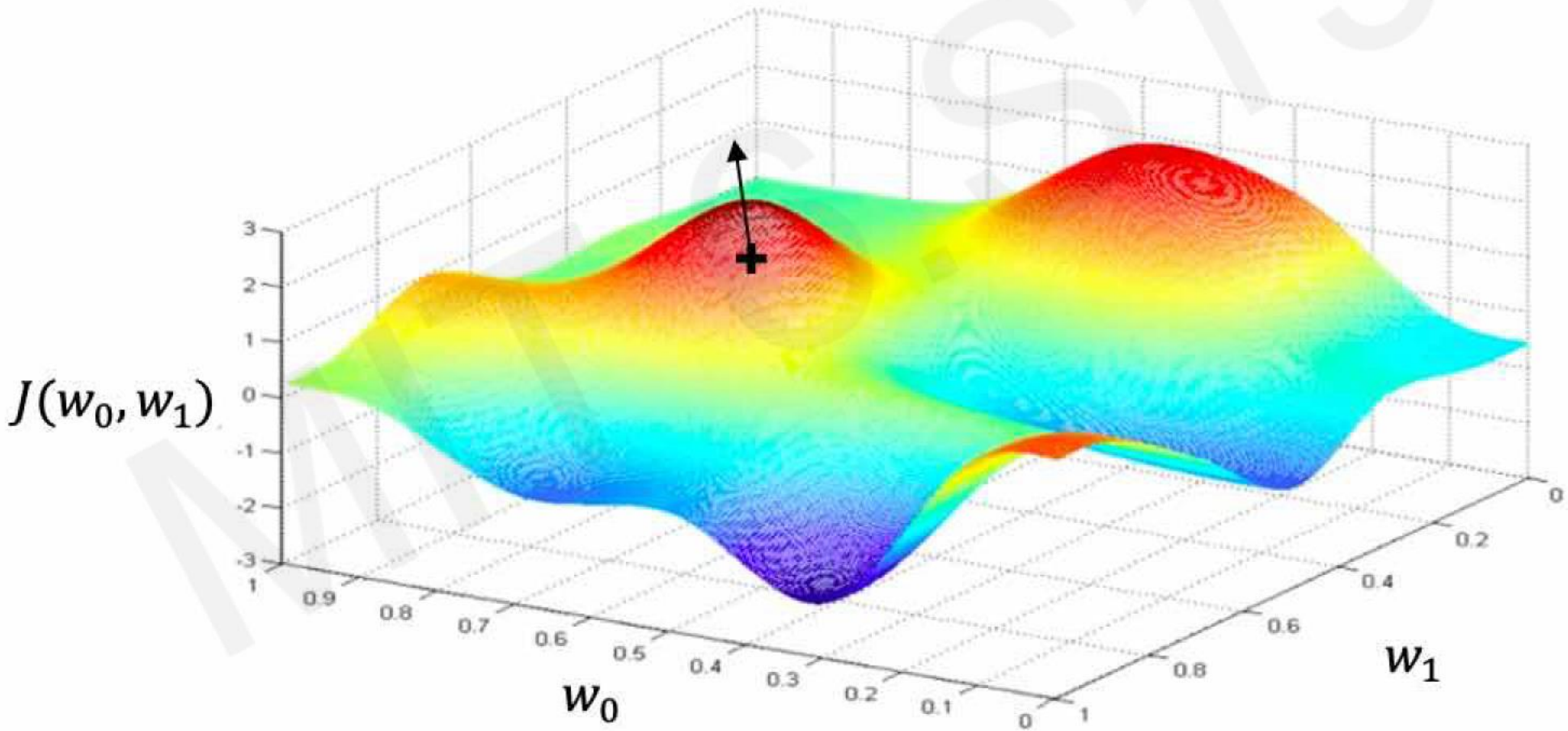
# Loss Optimization

Randomly pick an initial $(w_0, w_1)$



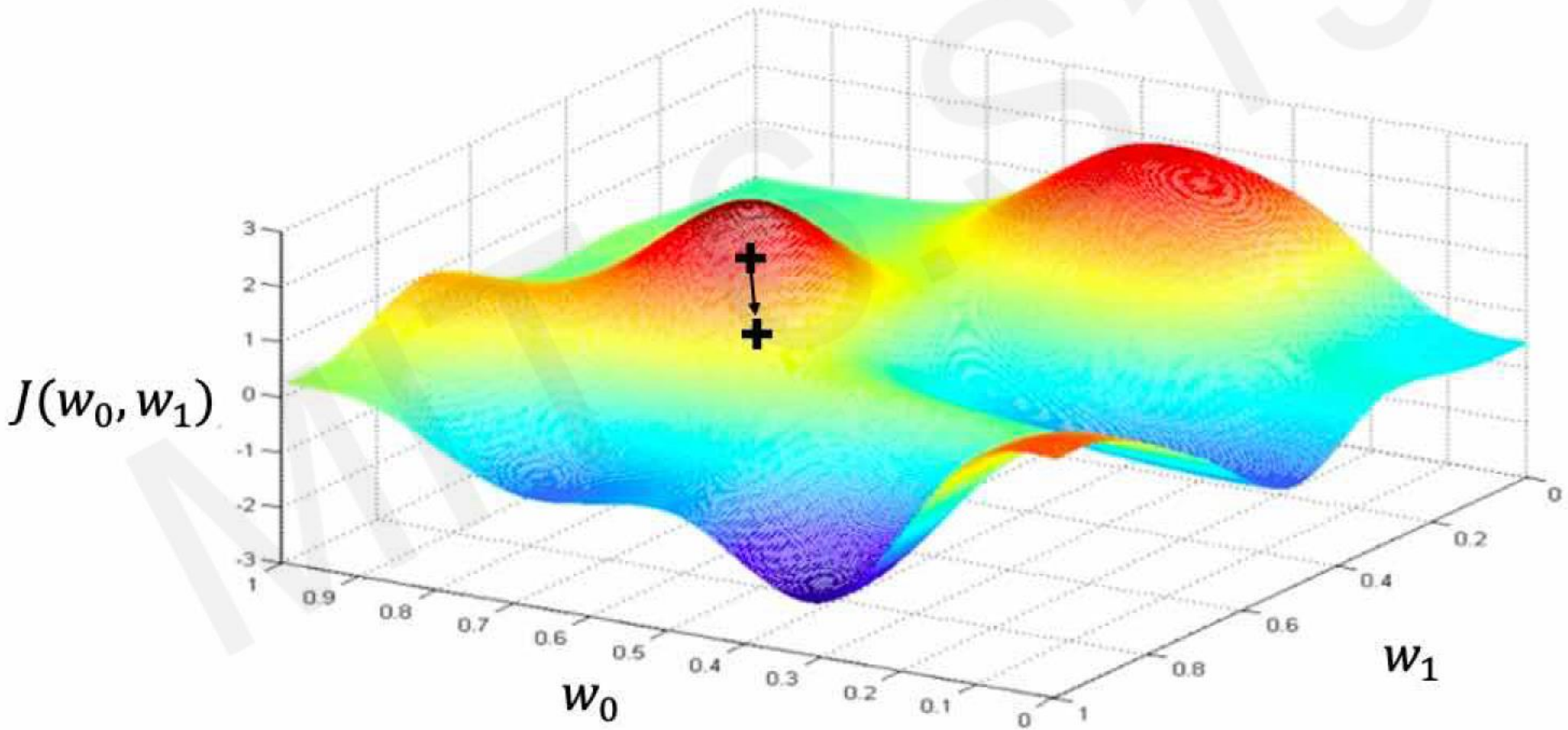
# Loss Optimization

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



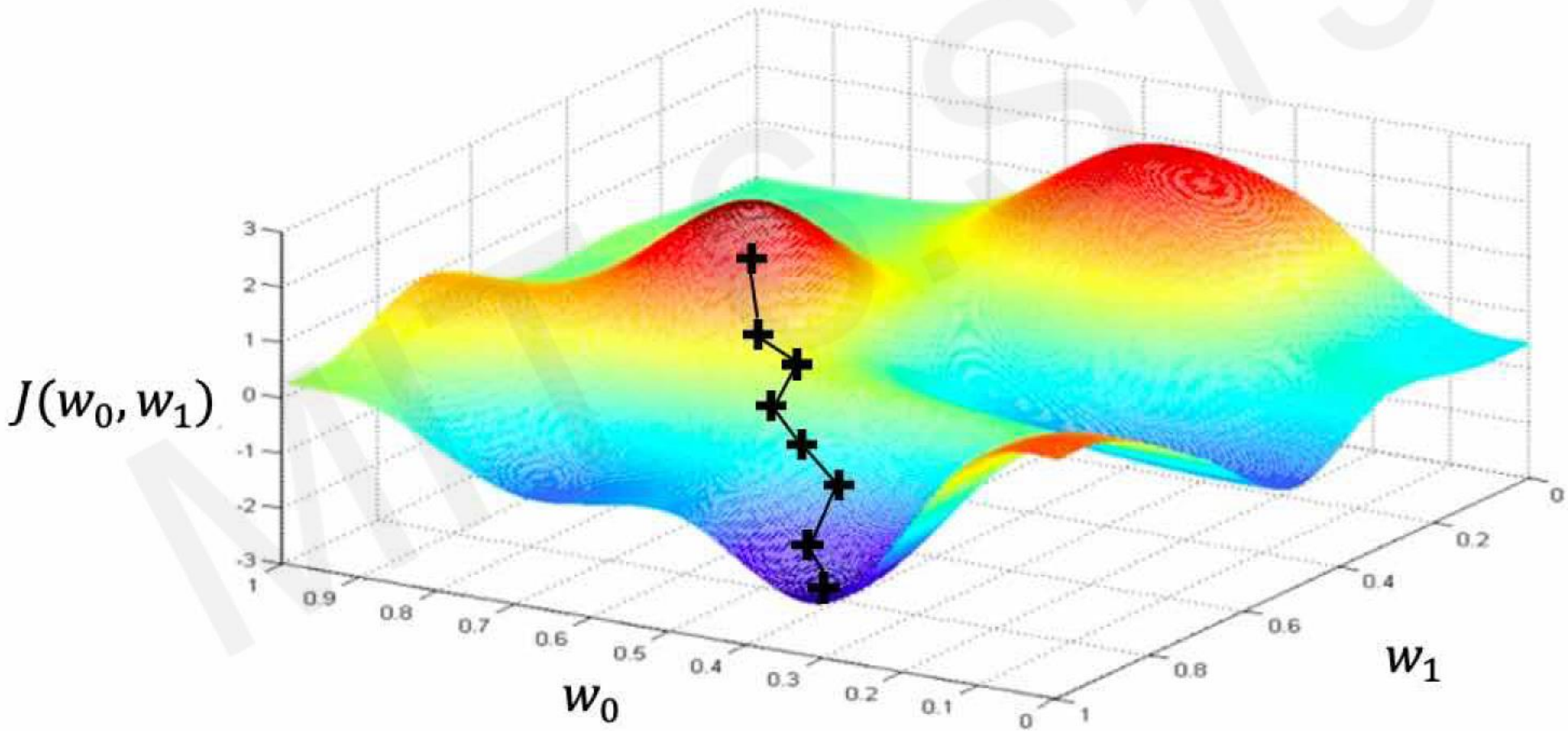
# Loss Optimization

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

# Computing Gradients: Backpropagation

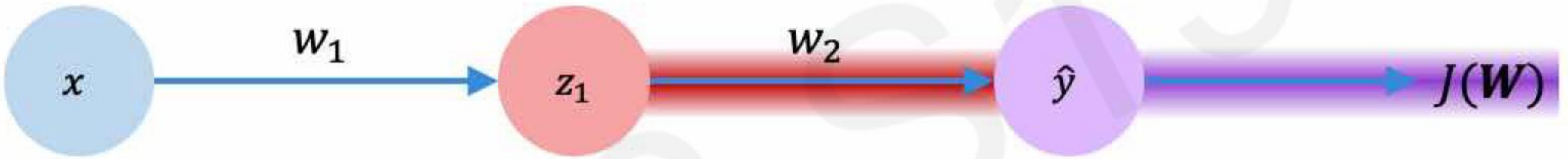

*How does a small change in one weight (ex. $\boldsymbol{w_2}$) affect the final loss $\boldsymbol{J(W)}$?*

# Computing Gradients: Backpropagation


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

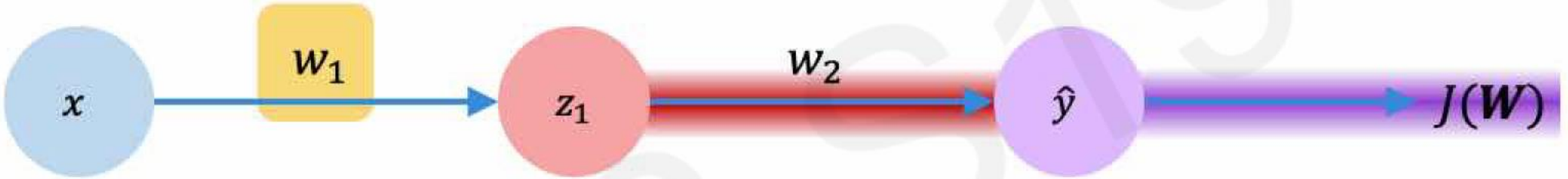
Let's use the chain rule!

# 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

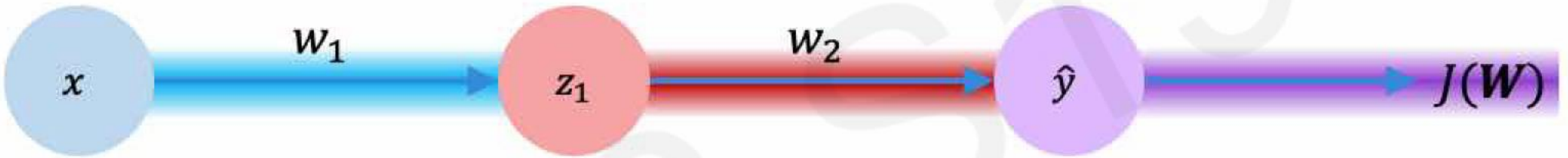



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

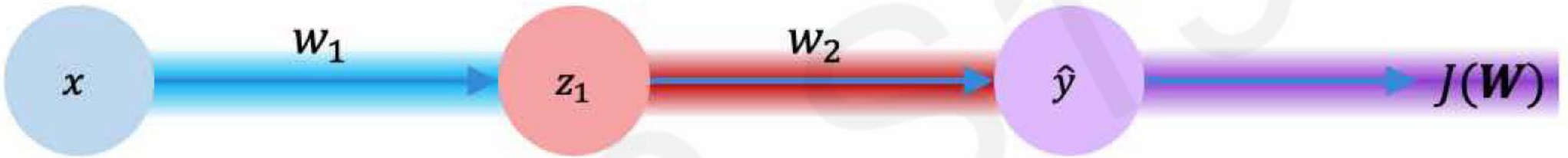
Apply chain rule!

# Computing Gradients: Backpropagation




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

# Computing Gradients: Backpropagation

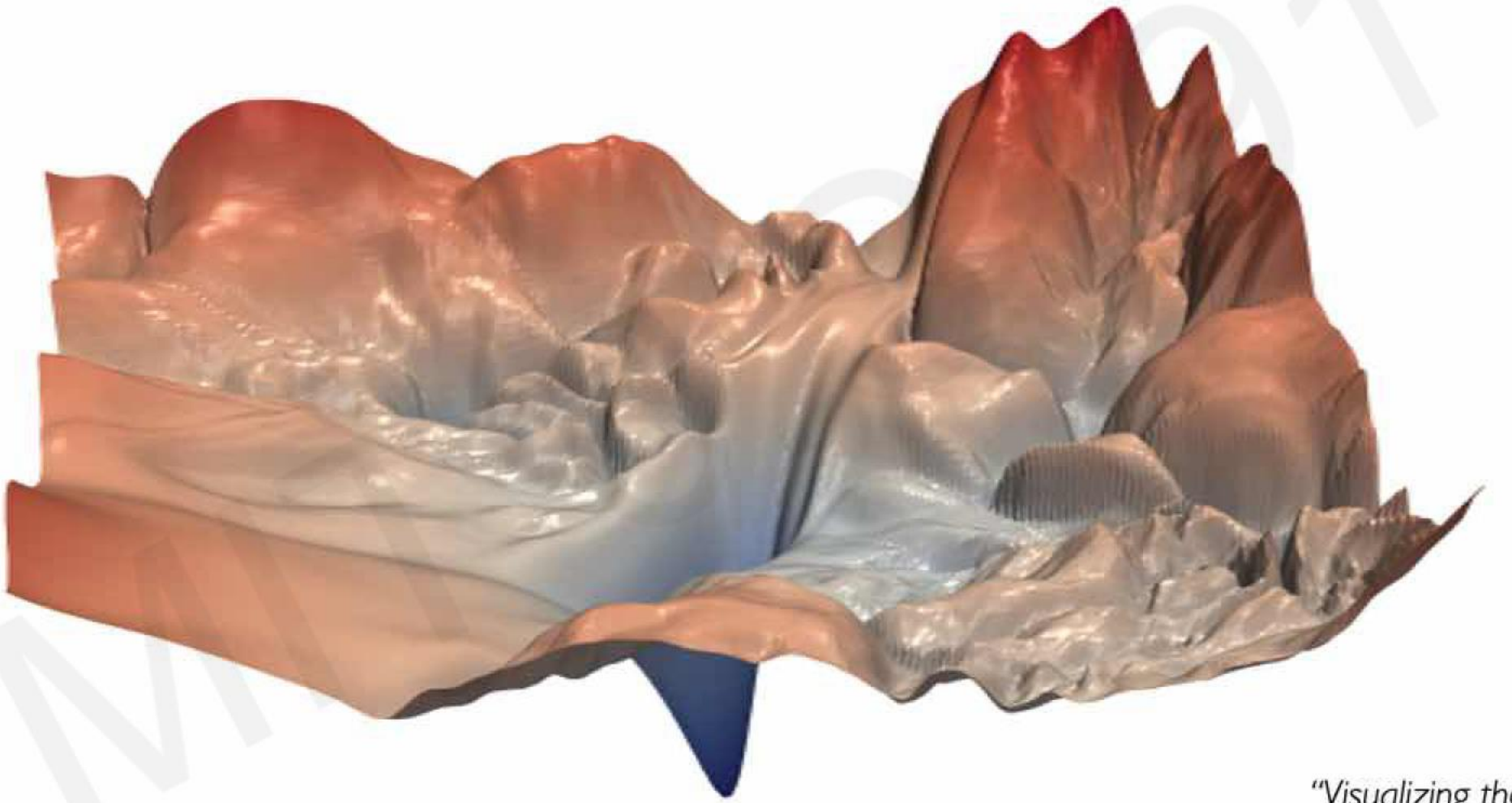



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

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

# DNN Tips & Tricks

# Training Neural Networks is Difficult



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

# Loss Functions Can Be Difficult to Optimize

**Remember:**

Optimization through gradient descent

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

# Loss Functions Can Be Difficult to Optimize

**Remember:**

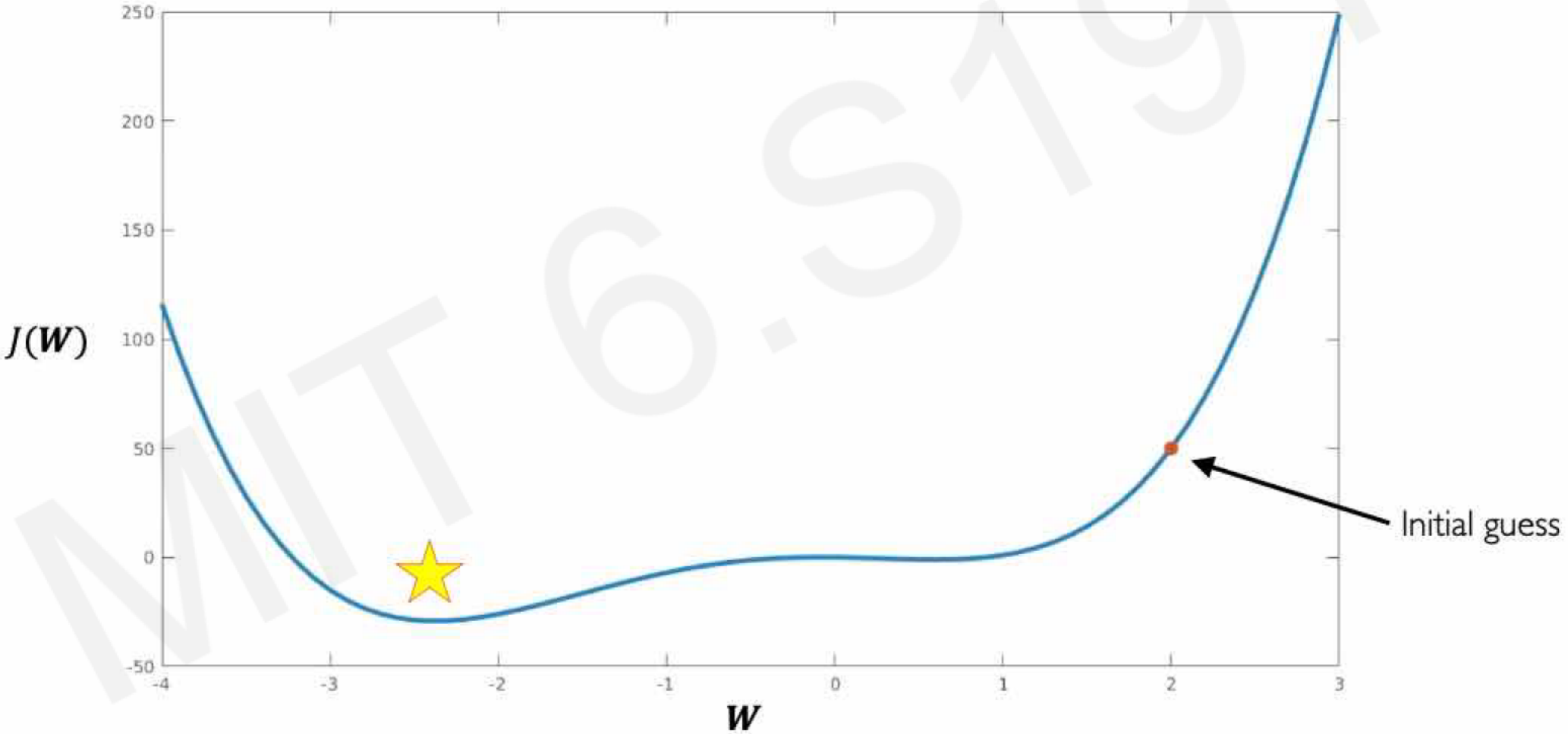
Optimization through gradient descent

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

How can we set the learning rate?

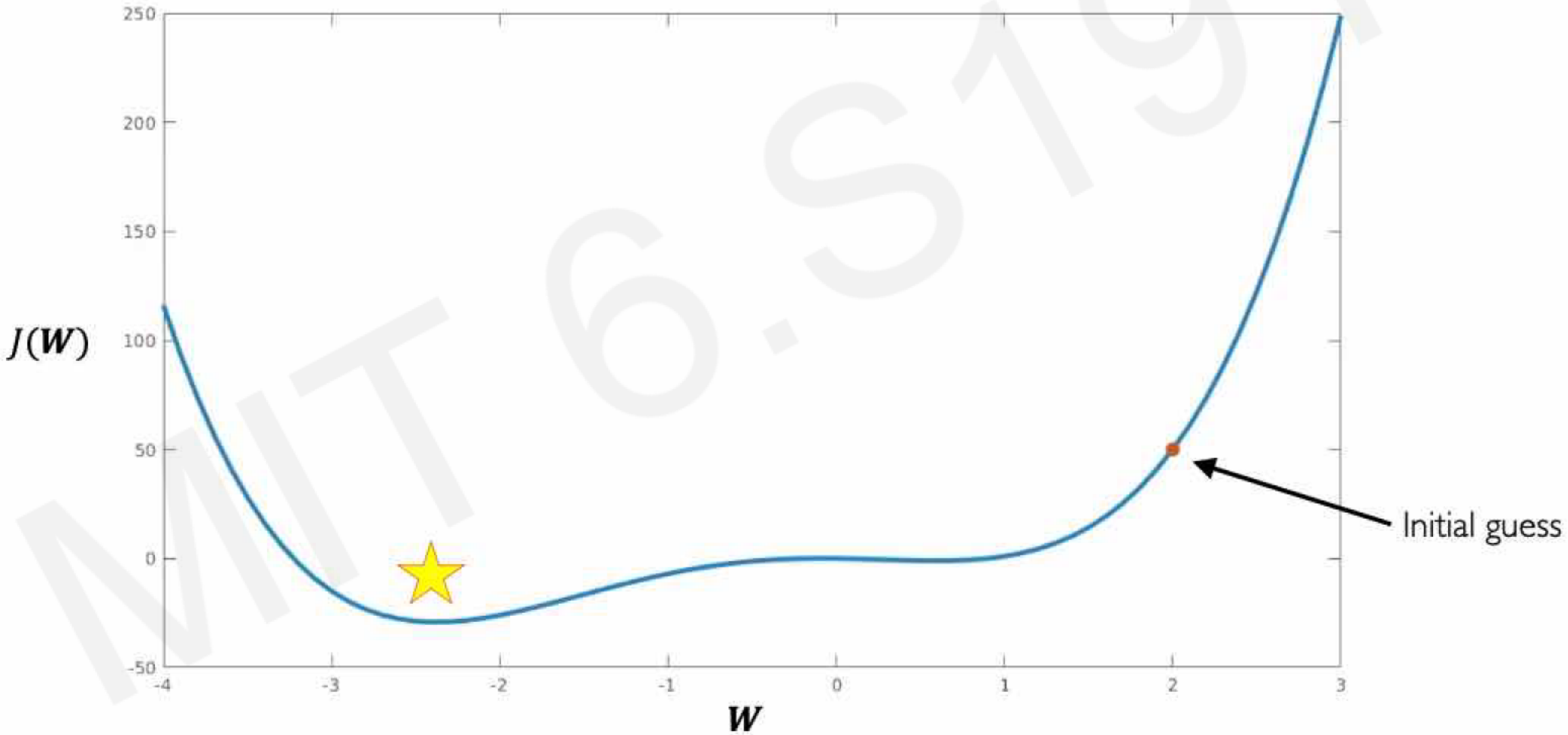
# Setting the Learning Rate

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



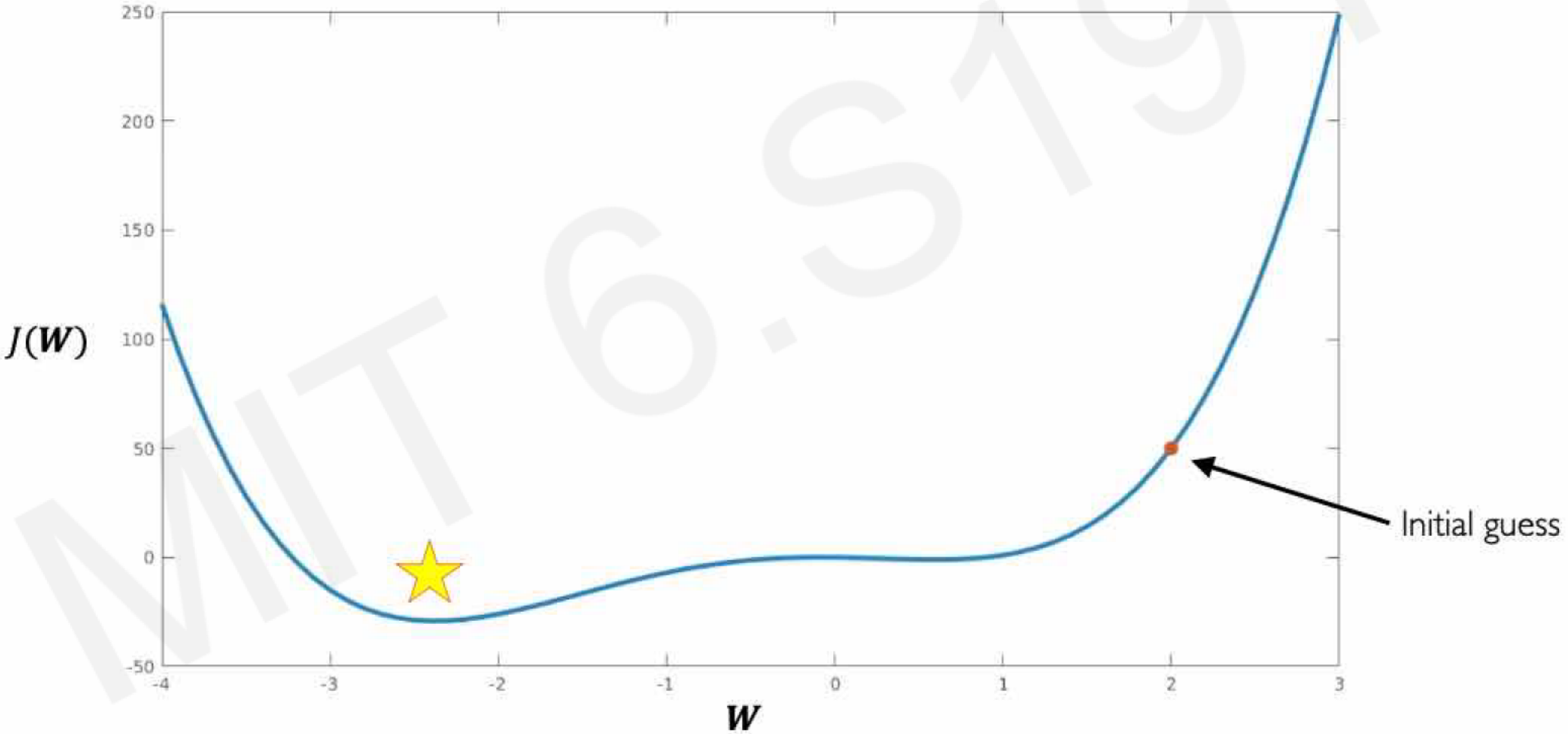
# Setting the Learning Rate

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

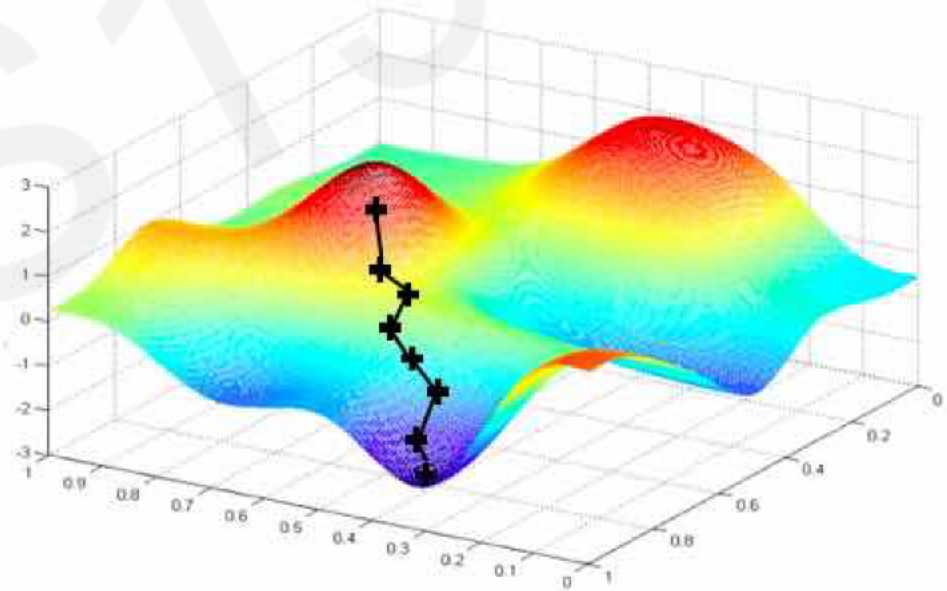
Additional details: http://ruder.io/optimizing-gradient-descent/

# Neural Networks in Practice: Mini-batches

# Gradient Descent

**Algorithm**

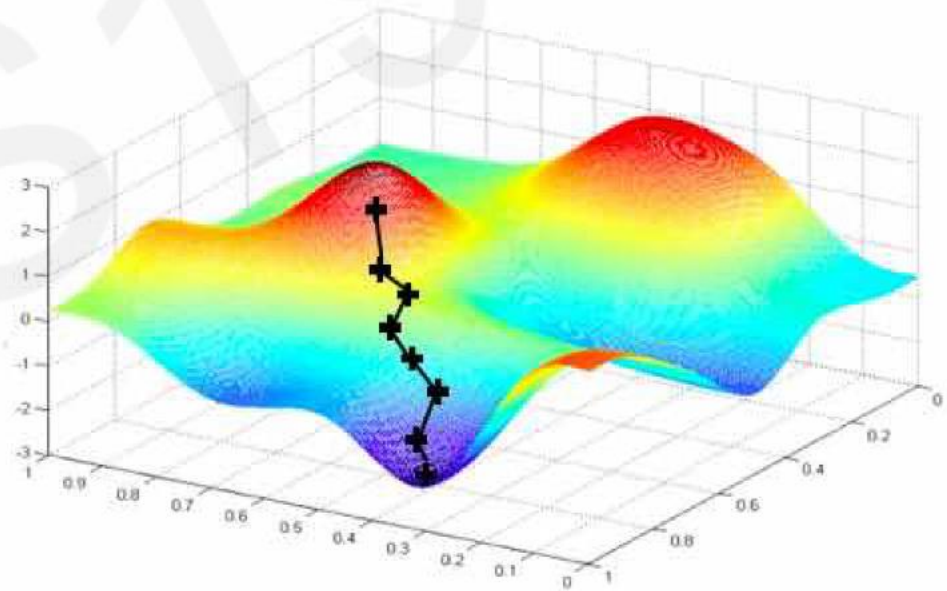
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5.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



# Stochastic Gradient Descent

**Algorithm**

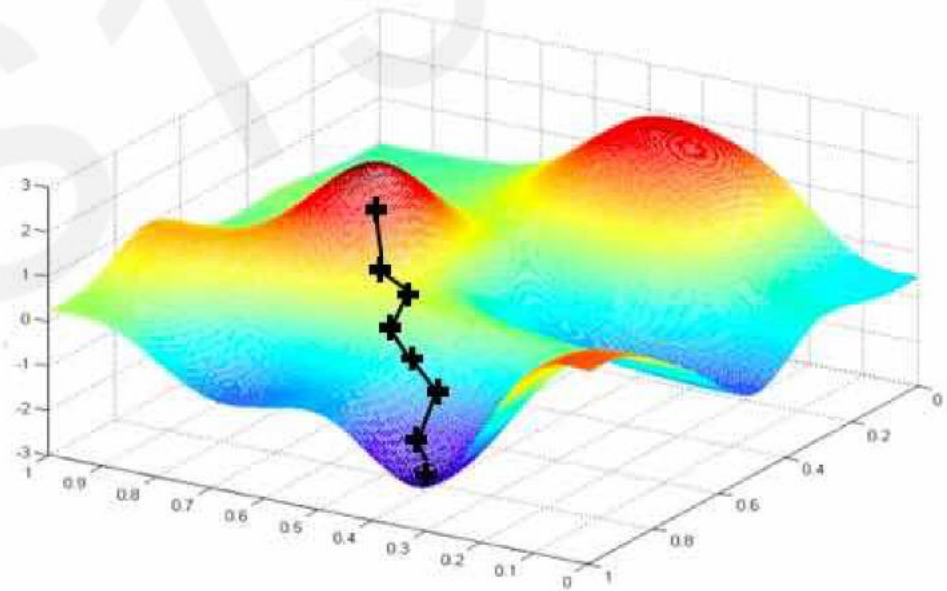
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point $\boldsymbol{i}$
4. Compute gradient, $\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial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



# Stochastic Gradient Descent

**Algorithm**

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

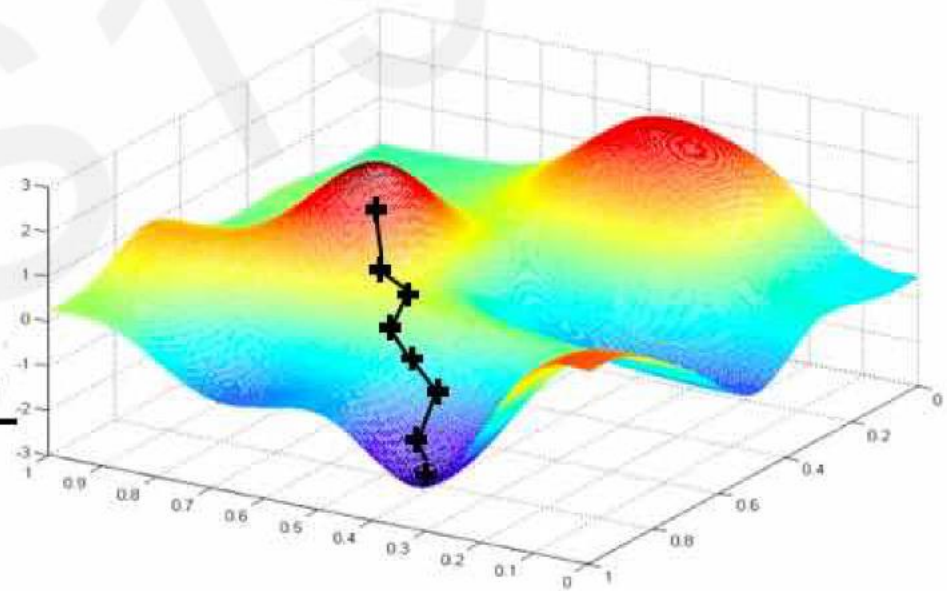


Easy to compute but
**very noisy** (stochastic)!

# Stochastic Gradient Descent

**Algorithm**

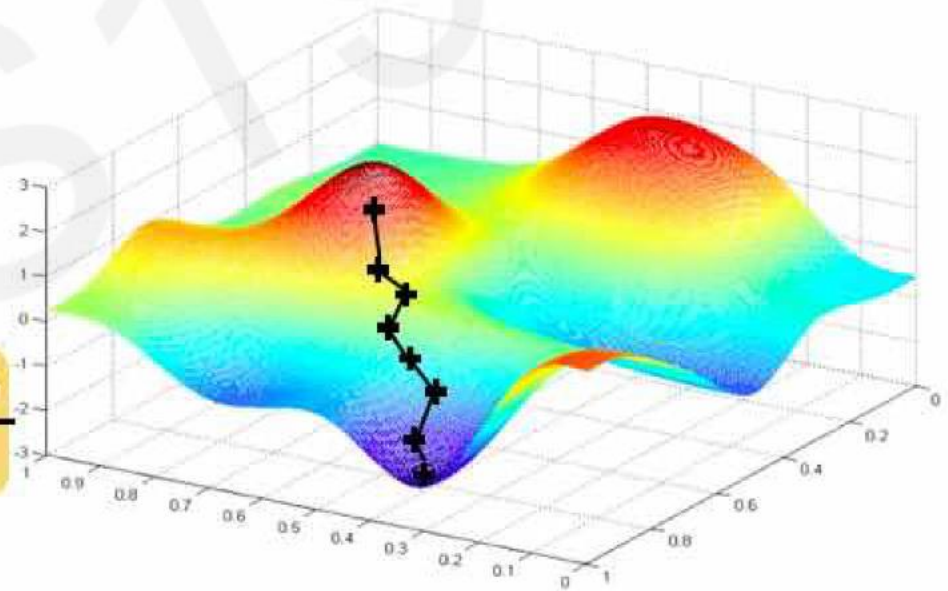
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of $B$ data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update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



# 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



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

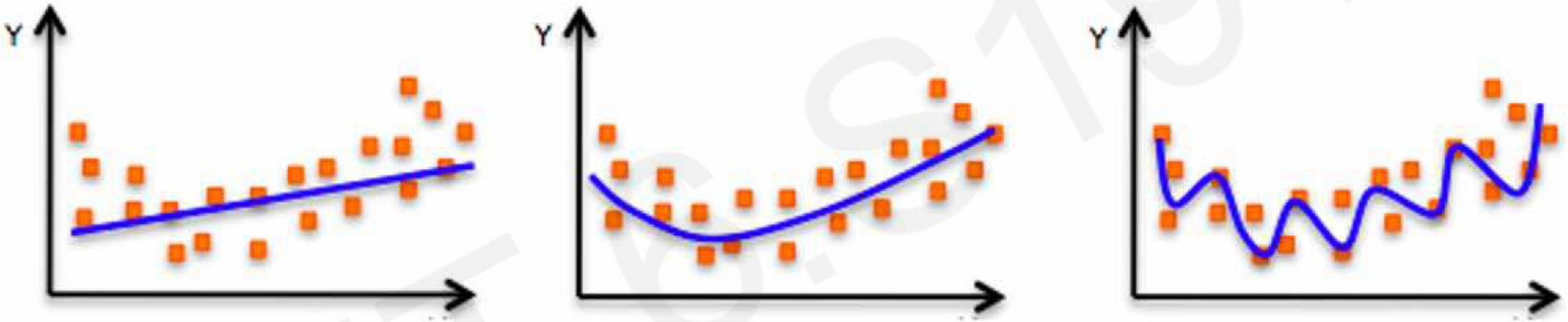
Allows for larger learning rates

**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?*

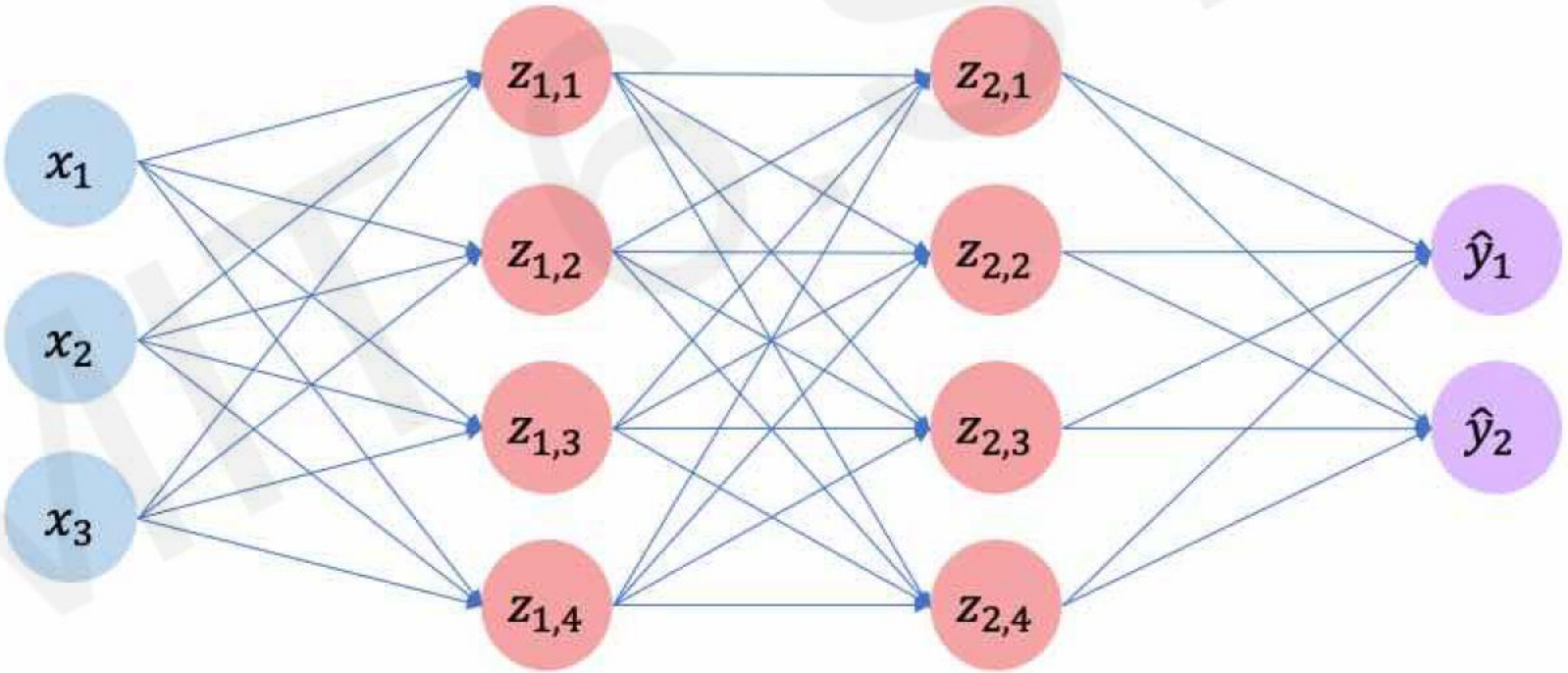
*Technique that constrains our optimization problem to discourage complex models*

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

*Improve generalization of our model on unseen data*

# Regularization 1: Dropout

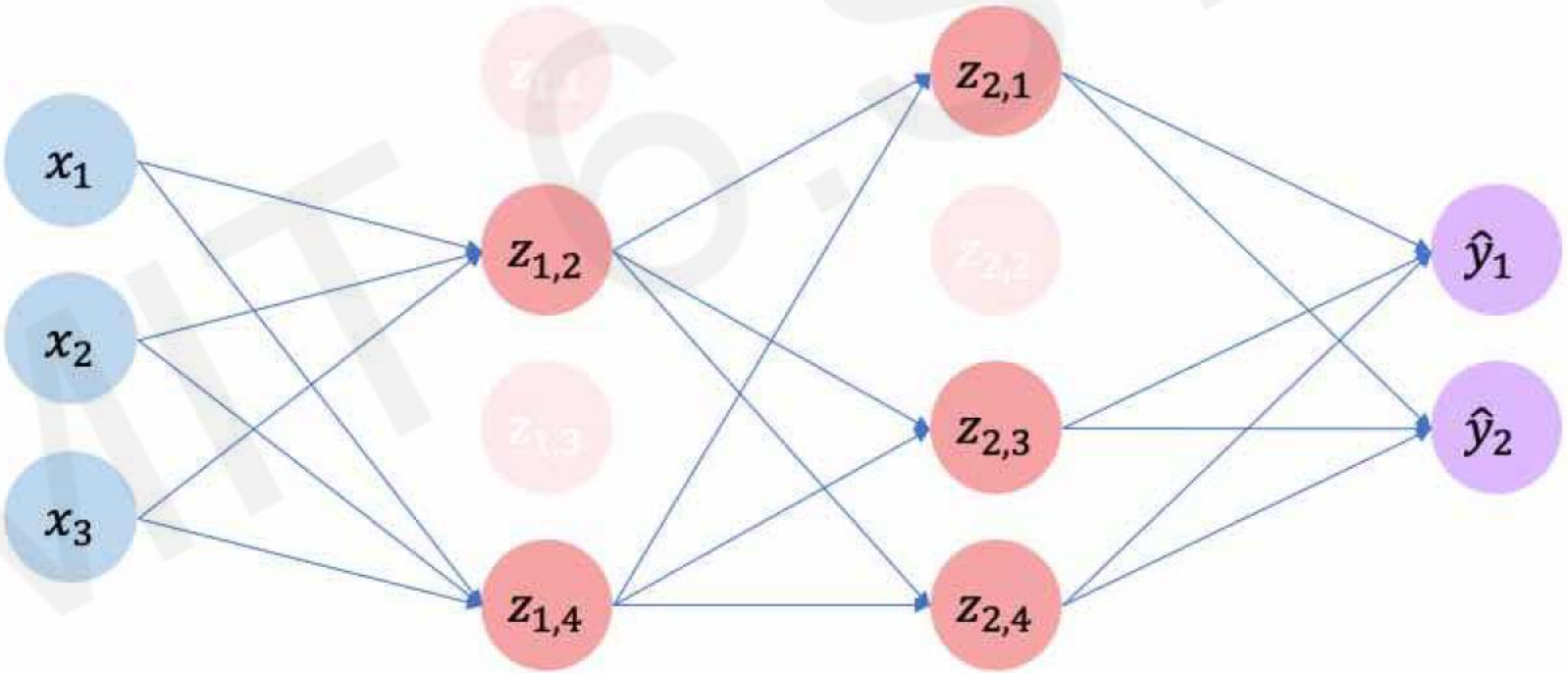
- During training, randomly set some activations to 0



# 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

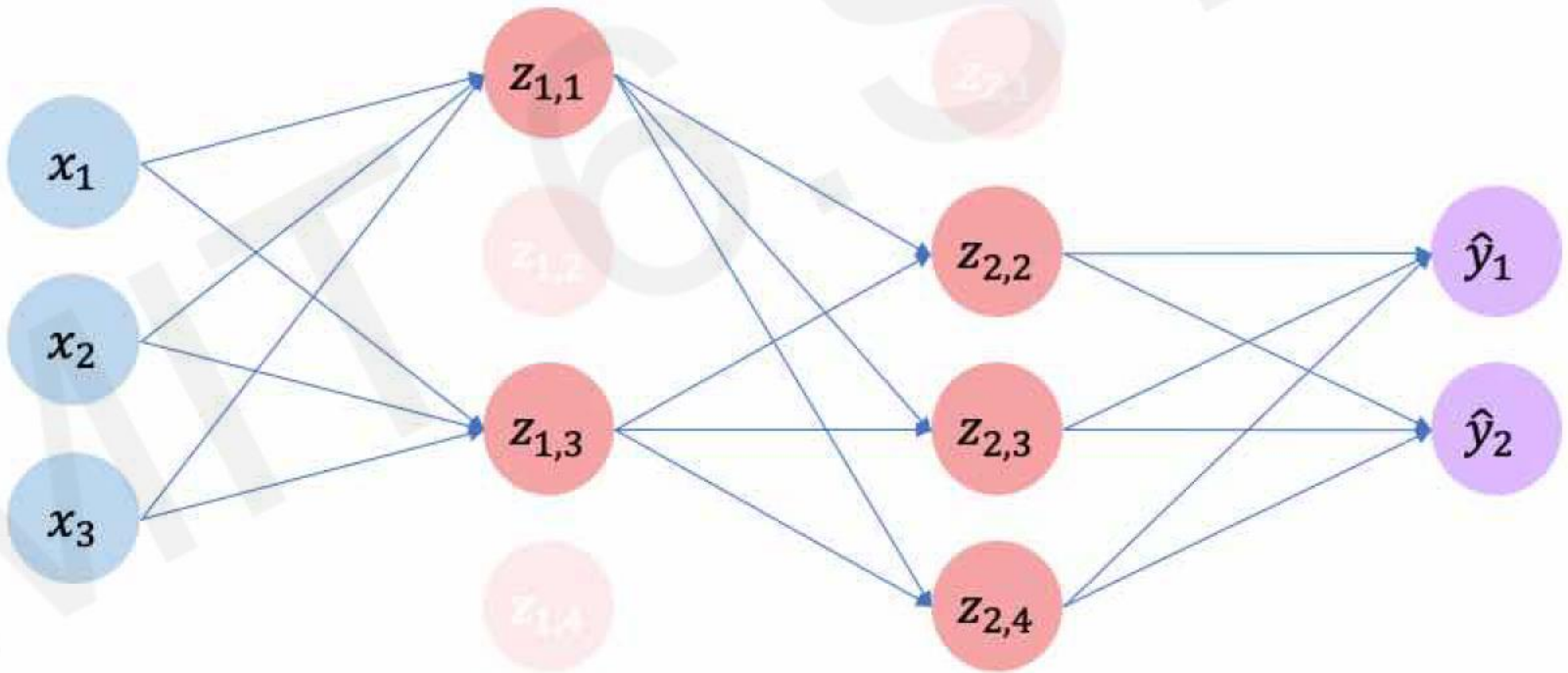
```
tf.keras.layers.Dropout(p=0.5)
```



# Regularization 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

# Regularization 2: Early Stopping

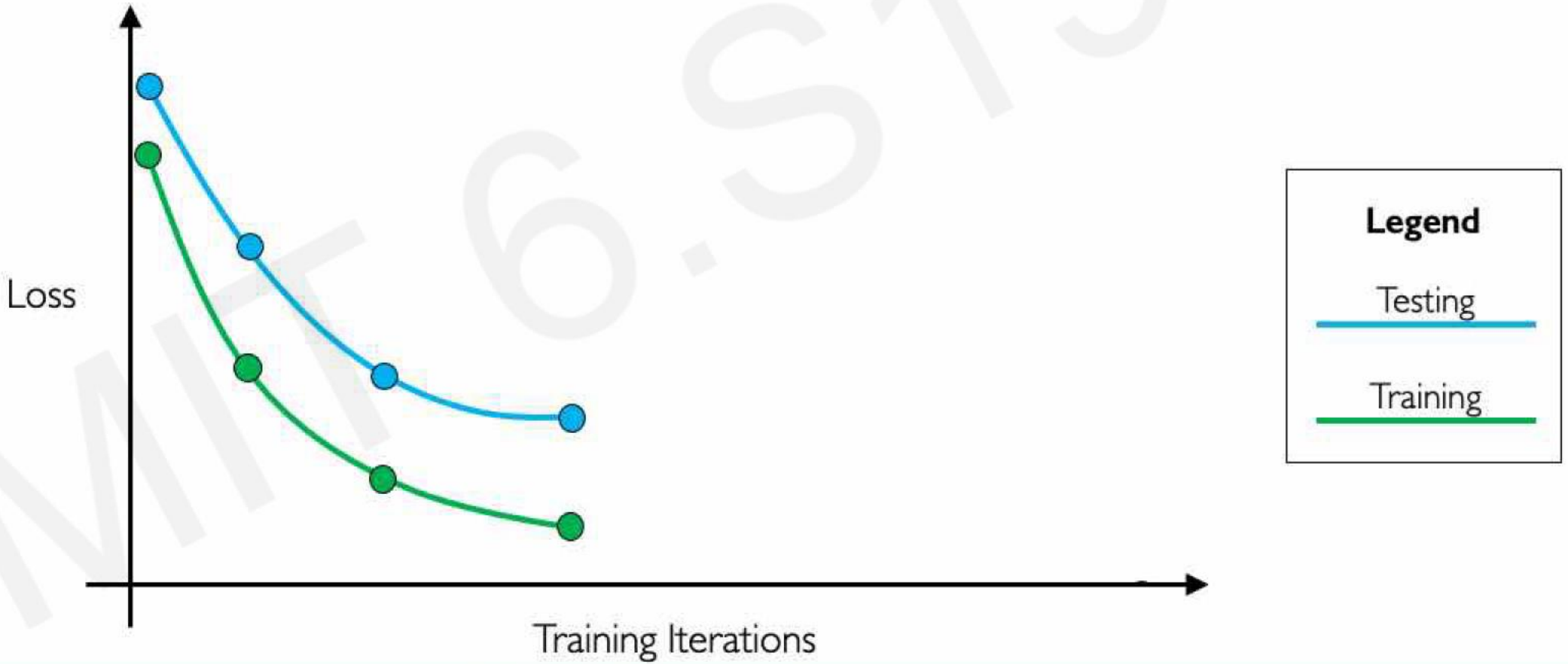
- Stop training before we have a chance to overfit

# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

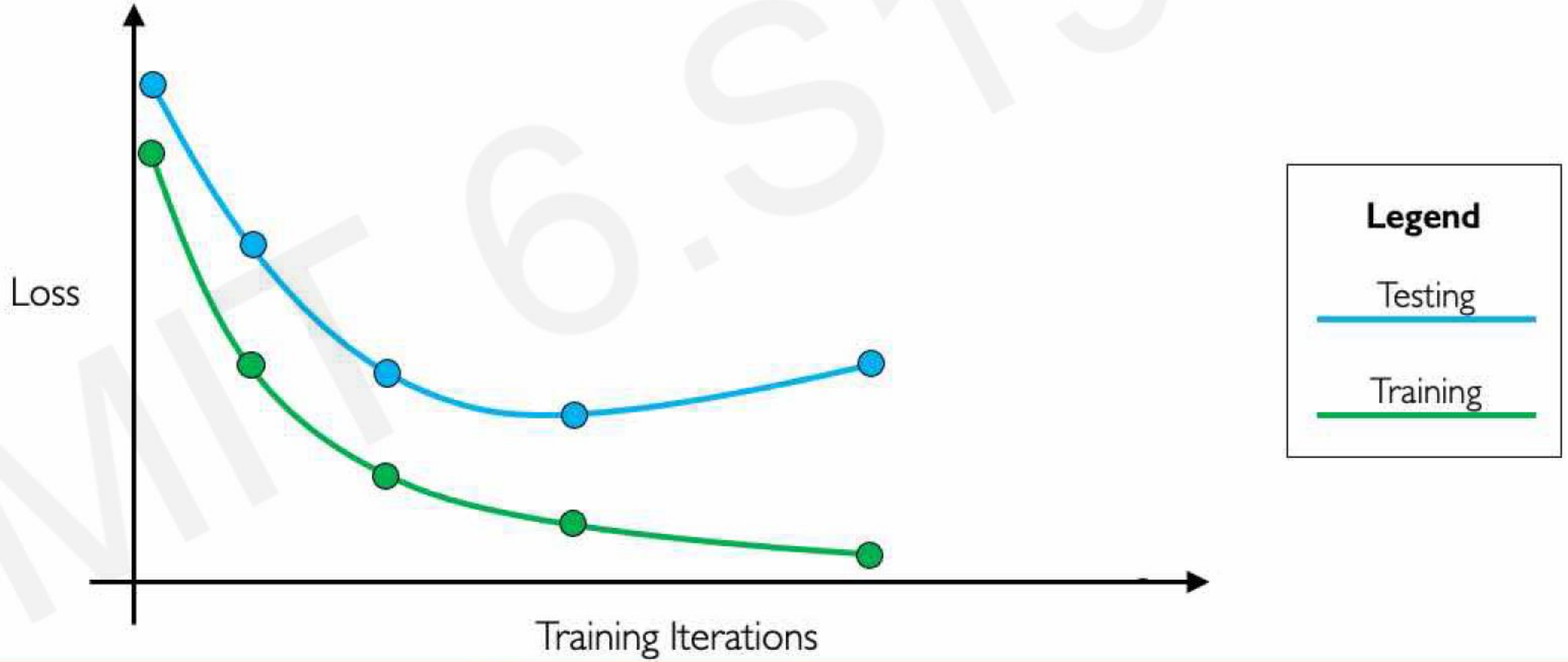
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



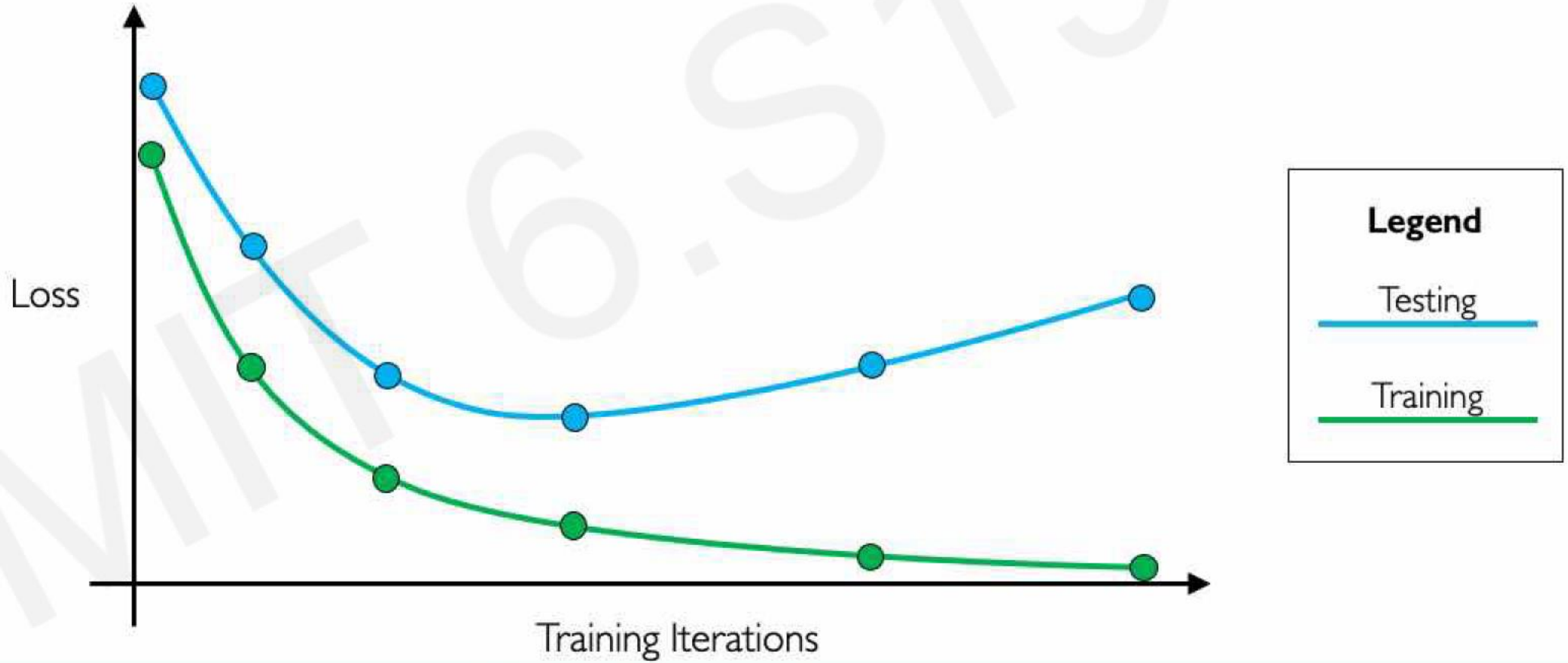
# Regularization 2: Early Stopping

- 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

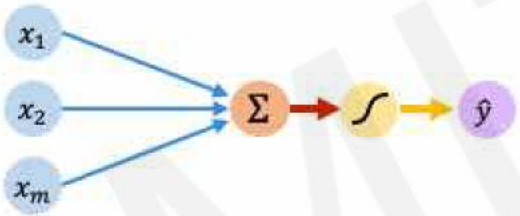
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

# Core Foundation Review

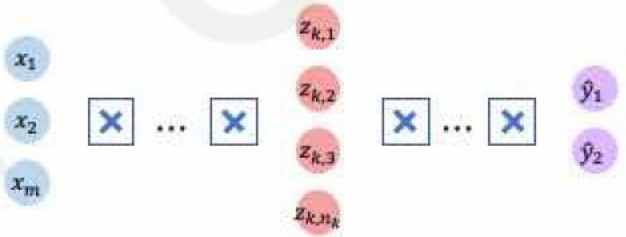
## The Perceptron

- Structural building blocks
- Nonlinear activation functions



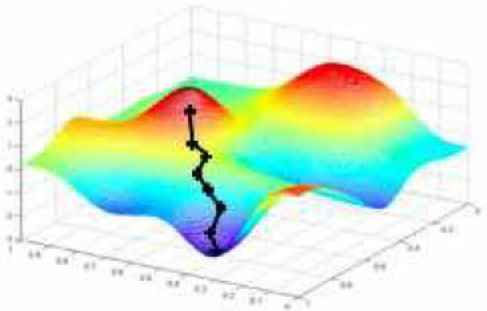

## Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



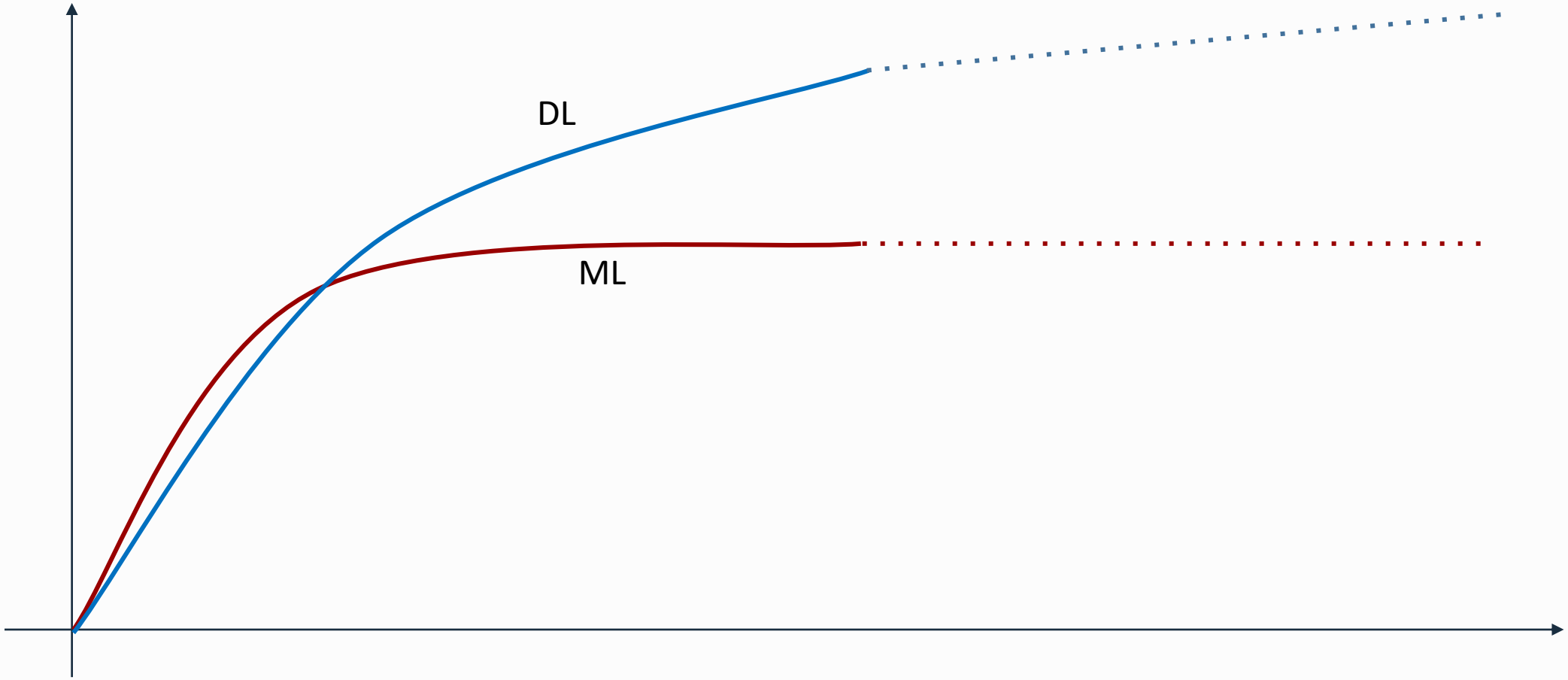

## Training in Practice

- Adaptive learning
- Batching
- Regularization



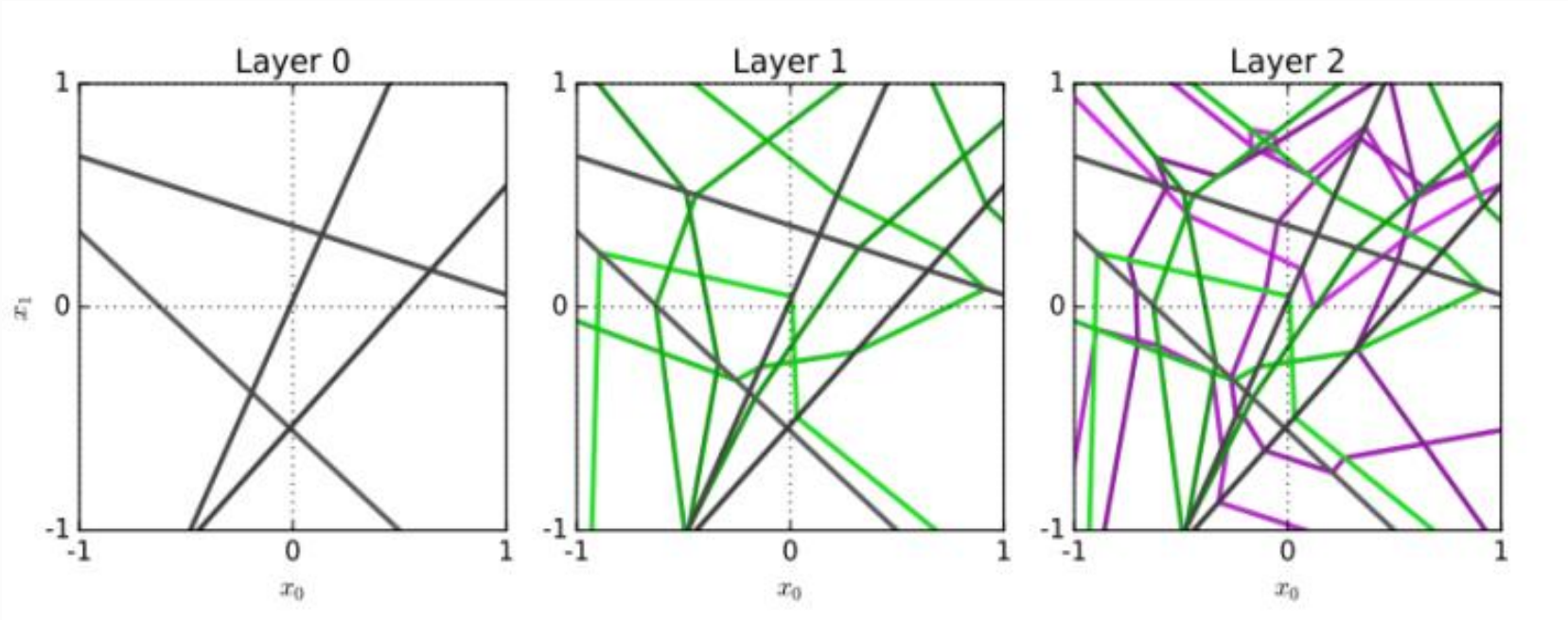
# Why Deep

# Why deep, comparatively



# Why deep, intuitively

- Intuitively, depth exponentiates capacity (vs width, which grows polynomially)
- (Width remains important! It sets the “embedded dimension”)
- Depth is great, but adds structure (or constraints, or symmetries):



# Why deep, in NLP

- In NLP, everything* can be improved with DL (at least w.r.t representation)
- High-dimensional input is hard to represent w/o DL
- Many weird interactions in language => expressivity is key
- => ALL of our models are DL-variants (RNN, CNN, GNN, Transformers)

*Well, almost...

# Why do we need DNNs in NLP?

"The Technion is the best [MASK] in Israel, much better than Tel Aviv University."

Can you predict [MASK]?

# Questions?