Deep Learning

Theoretical introduction and its application for face detection, recognition and camouflage.

Raffaella Lanzarotti

Aim of the course

- Introduction to Deep Learning,
  - Theoretical
  - Practical

- We'll largely adopt the valuable material from:

  http://introtodeeplearning.com/

Schedule (tentative)

DAY 1	CLASS 1	• Introduction to Machine Learning
	LAB 1	• Tensor Flow
DAY 2	CLASS 2	• Deep Sequence Modelling
	LAB 2	• Music Generation
DAY 3	CLASS 3	• Convolutional Neural Networ
	LAB 3	• MNIST
DAY 4	CLASS 4	• Deep Generative Models
	LAB 4	• Debiasing
DAY 5	CLASS 5	• Deep Reinforcement Learning • Limits and new Frontiers

Introduction to Machine Learning

The Rise of Deep Learning

The Rise of Deep Learning

Deep learning:

- has revolutionized many areas of machine intelligence, with particular impact on image understanding tasks

- particularly effective…
  - for unstructured data
  - to learn good representations
  - to learn good "models"

What is Intelligence?

- The ability to process information, to inform future decisions

What is Deep Learning?

Artificial Intelligence

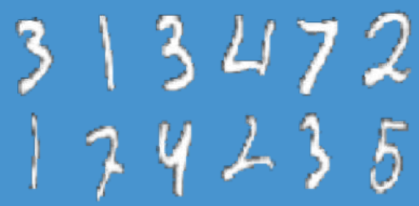
Any technique that enables computers to mimic human behavior

Machine Learning

Ability to learn without explicitly being programmed

Deep Learning

Extract patterns from data using neural networks



Why deep learning?
Why now?

Why Deep Learning?

Traditional ML:

- Hand engineered features
- LIMITS and PROBLEMS:
  - Time consuming
  - Brittle
  - Not scalable

Challenge:

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

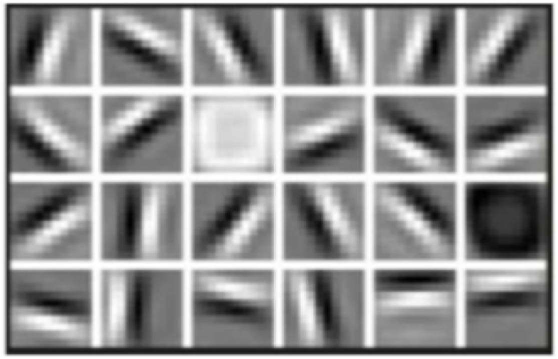
Deep Learning
learns features directly from data

Ex: Features to detect faces

- Which features characterize faces?

- They should be:
  - specific to this class
  - flexible to manage intra-class variability

Low Level Features ?



Lines & Edges

Mid Level Features?

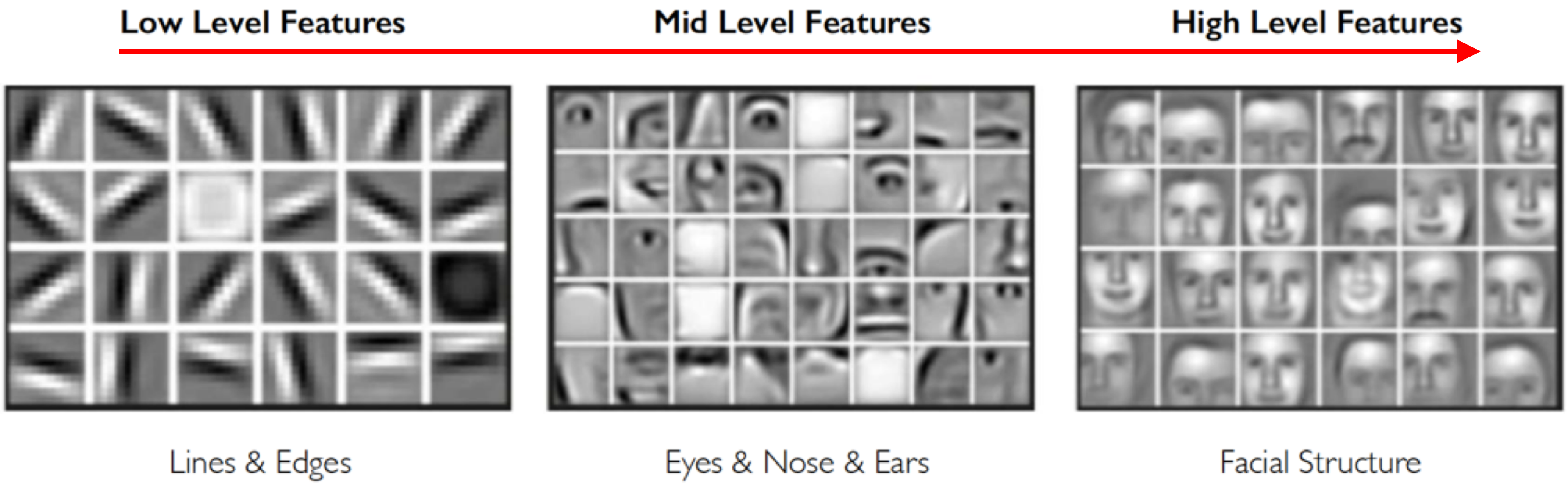
Eyes & Nose & Ears

High Level Features?

Facial Structure

Why Deep Learning?

Deep Learning
learns features in a hierarchical manner



In this course…

- We'll try to answer to this question:

HOW CAN WE GO FROM RAW DATA (e.g. pixels) TO A MORE AND MORE COMPLEX REPRESENTATION AS THE DATA FLOWS THROUGH THE MODEL?

Why Now?

1952 Stochastic Gradient Descent

1958 Perceptron
- Learnable Weights

⋮

1986 Backpropagation
- Multi-Layer Perceptron

1995 Deep Convolutional NN
- Digit Recognition

⋮

Neural Networks date back decades, so why the resurgence?

1. Big Data
- Larger Datasets
- Easier Collection & Storage

IMAGENET

WIKIPEDIA
The Free Encyclopedia

2. Hardware
- Graphics Processing Units (GPUs)
- Massively Parallelizable

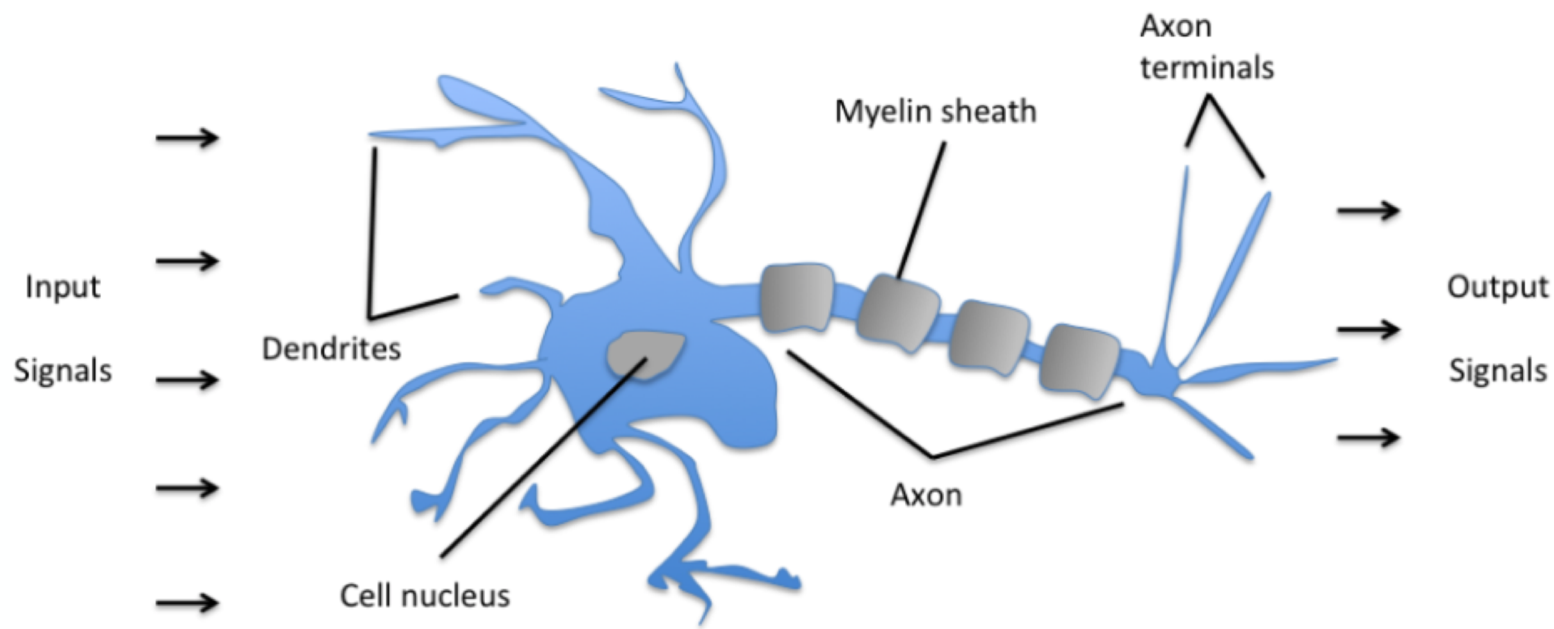
3. Software
- Improved Techniques
- New Models
- Toolboxes

TensorFlow

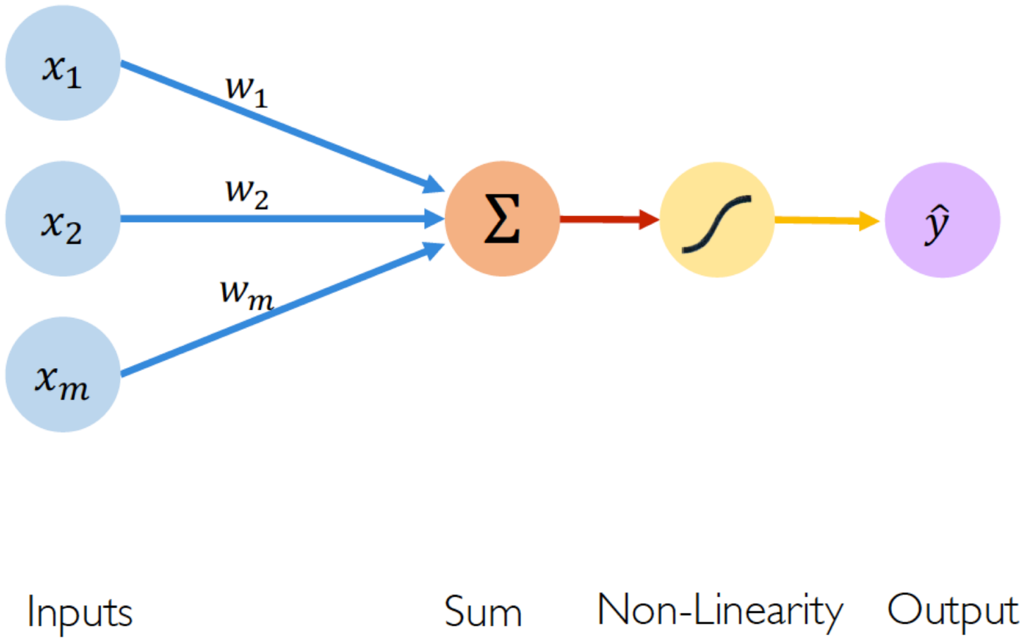
The Perceptron

The structural building block of deep learning

Biological Inspiration



The Perceptron: Forward Propagation



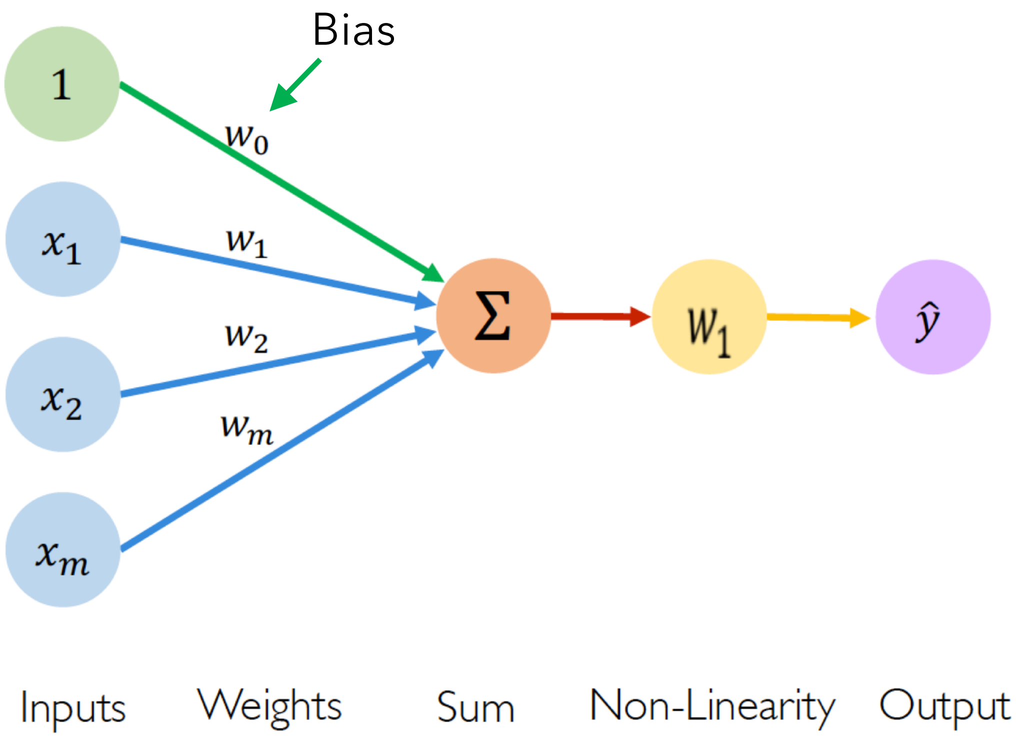
Output

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

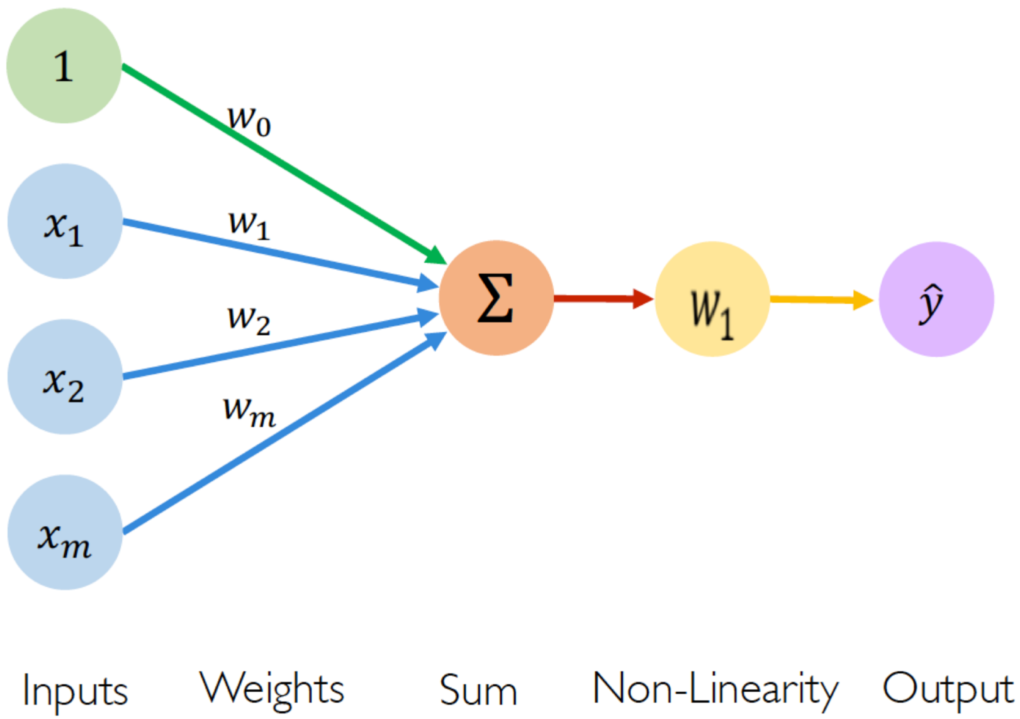
Linear combination of inputs

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

Non-linear activation function

Bias

The Perceptron: Forward Propagation



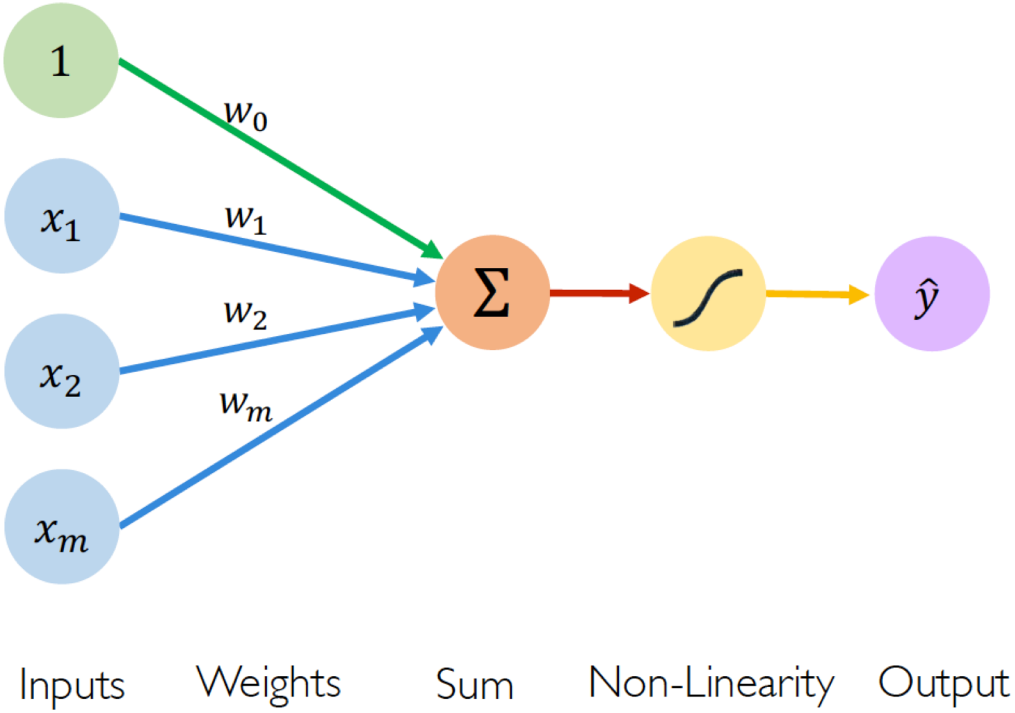
$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i \, w_i \right)$$

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

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

Using Linear Algebra…

The Perceptron: Forward Propagation

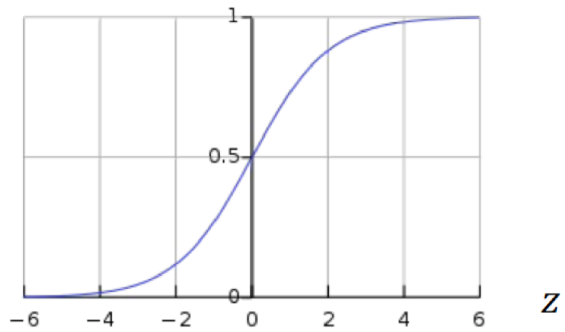


Activation Functions

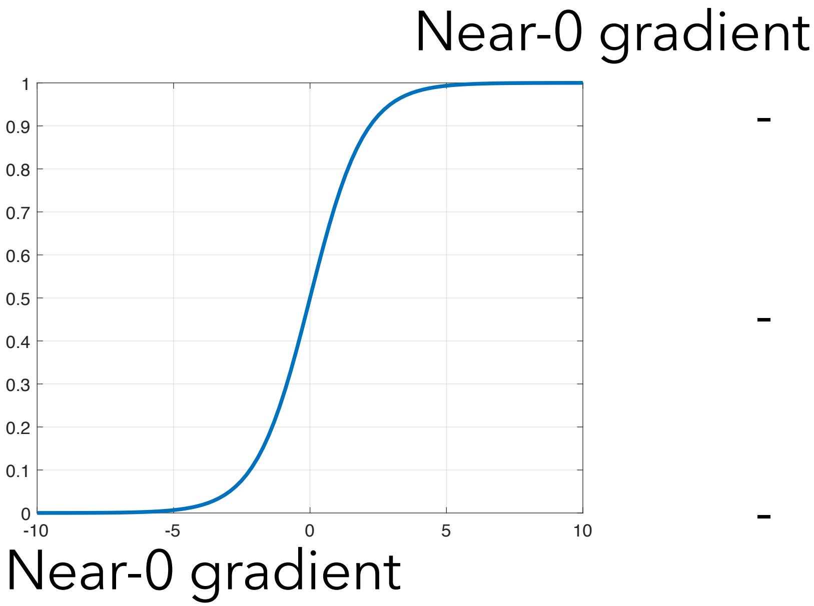
$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W} \right)$$

- Example: sigmoid function

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



Sigmoid



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

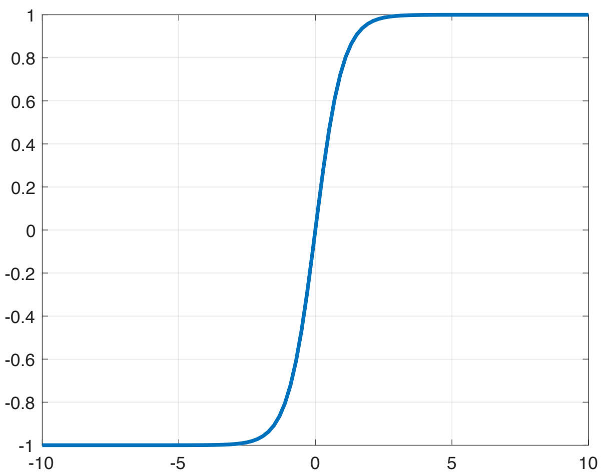
- Historically the most used for binary classification

- Useful for modelling probability, because it collapse the input between 0 and 1

- It suffers from the vanishing gradient problem

- Non-zero centered output that may cause zig-zagging

Tanh

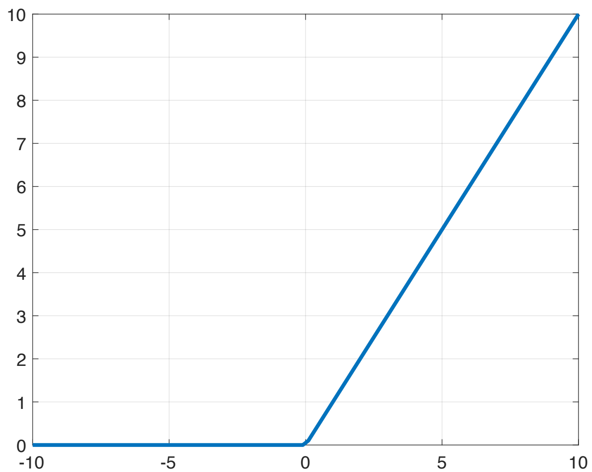


$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- It suffers from the vanishing gradient problem
- Output is zero centered, thus it has better gradient properties than sigmoid
- It is a scaled version of *Sigmoid*:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$

ReLU (Rectified Linear Unit)



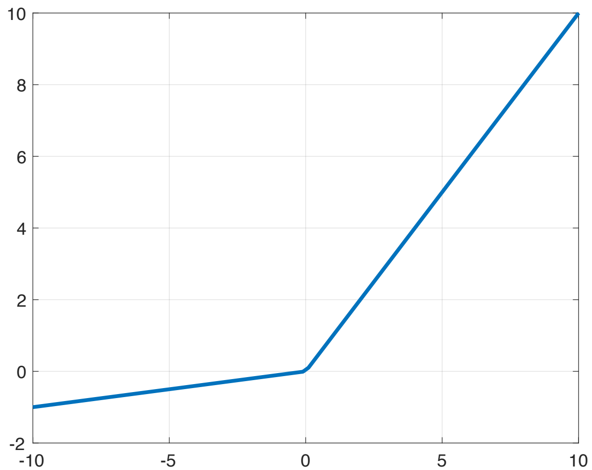
$$f(x) = max(0, x)$$

- Very popular and simple: it thresholds values below 0

- It allows for fast convergence of the optimization function

- The weight may irreversibly die

Leaky ReLU



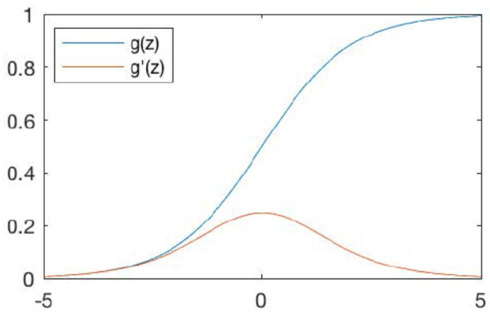
$$f(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

$$\alpha = 0.1$$

- It is aimed to <u>fix the dying ReLU problem</u>
- In a variant (called parametric ReLU) the <u>slope for negative values</u> can be learnt

Common Activation Functions

Sigmoid Function

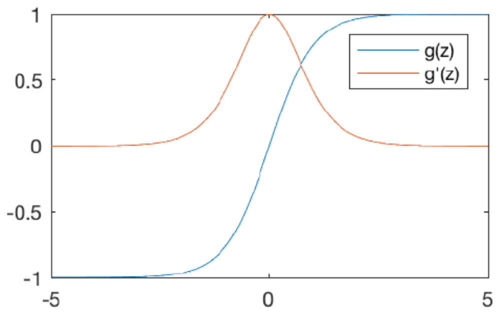


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

`tf.math.sigmoid(z)`

Hyperbolic Tangent

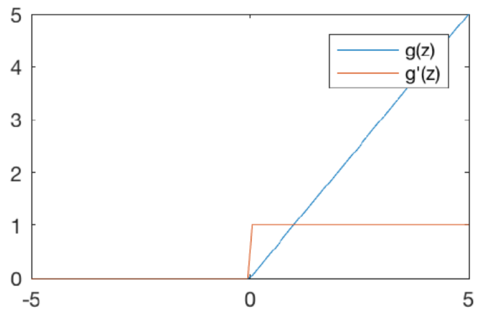


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

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

`tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



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

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

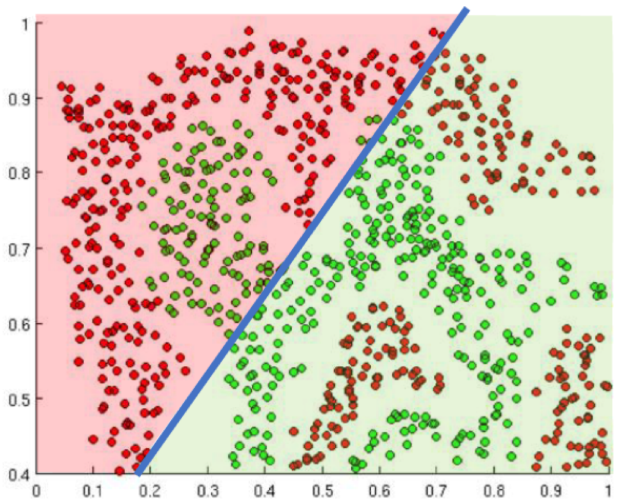
`tf.nn.relu(z)`

`TensorFlow code blocks`

NOTE: All activation functions are non-linear

Importance of Activation Functions

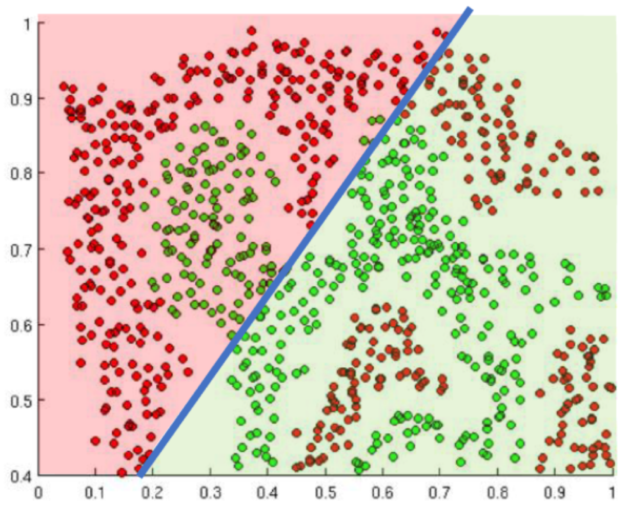
The purpose of activation functions is to ***introduce non-linearities*** *into the network*



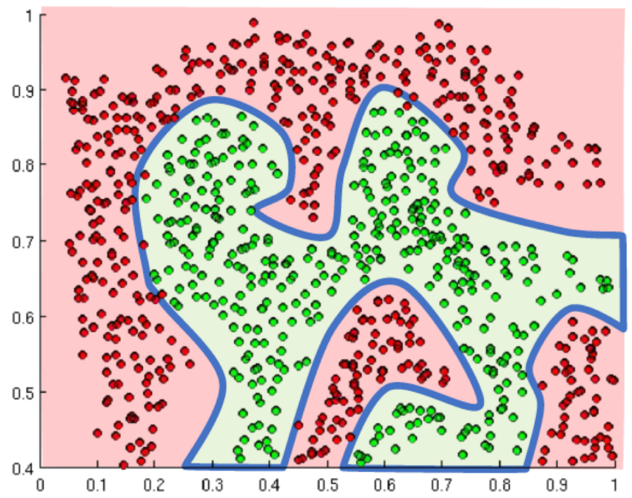
Linear Activation functions produce linear decisions no matter the network size

Importance of Activation Functions

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

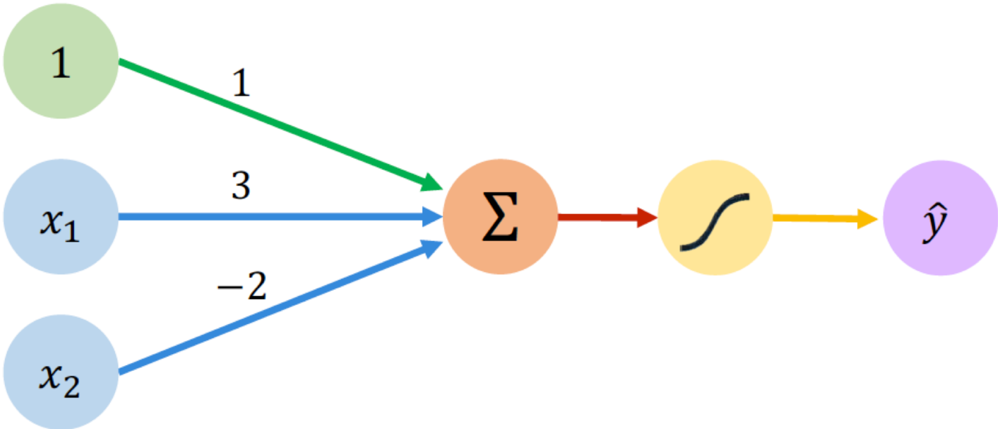


Linear Activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example

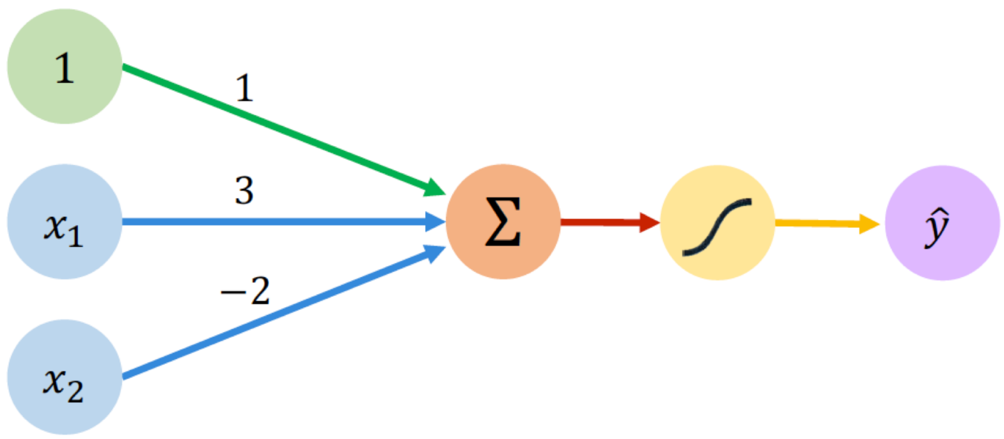


We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

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

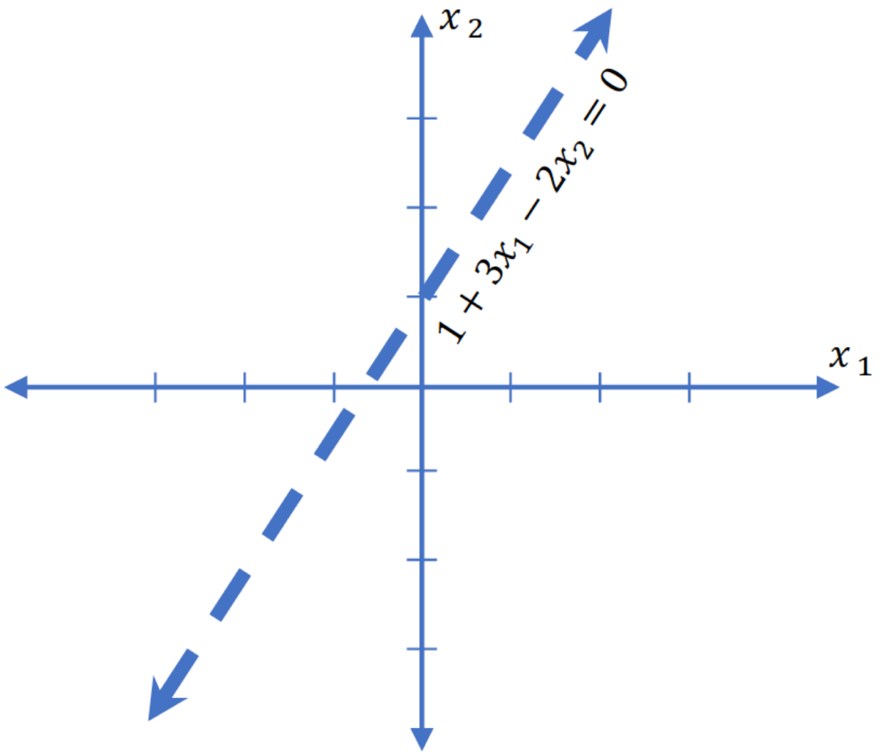
This is just a line in 2D!

The Perceptron: Example

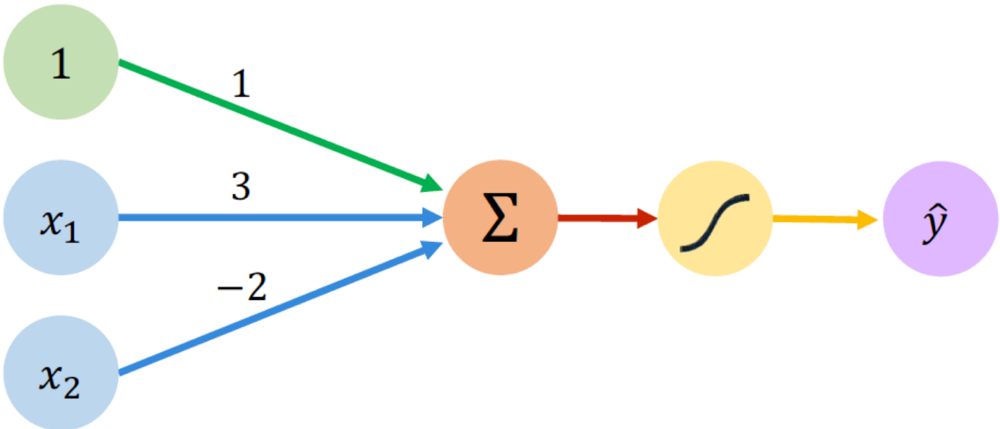


Plot this line equal to 0 in the feature space:

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



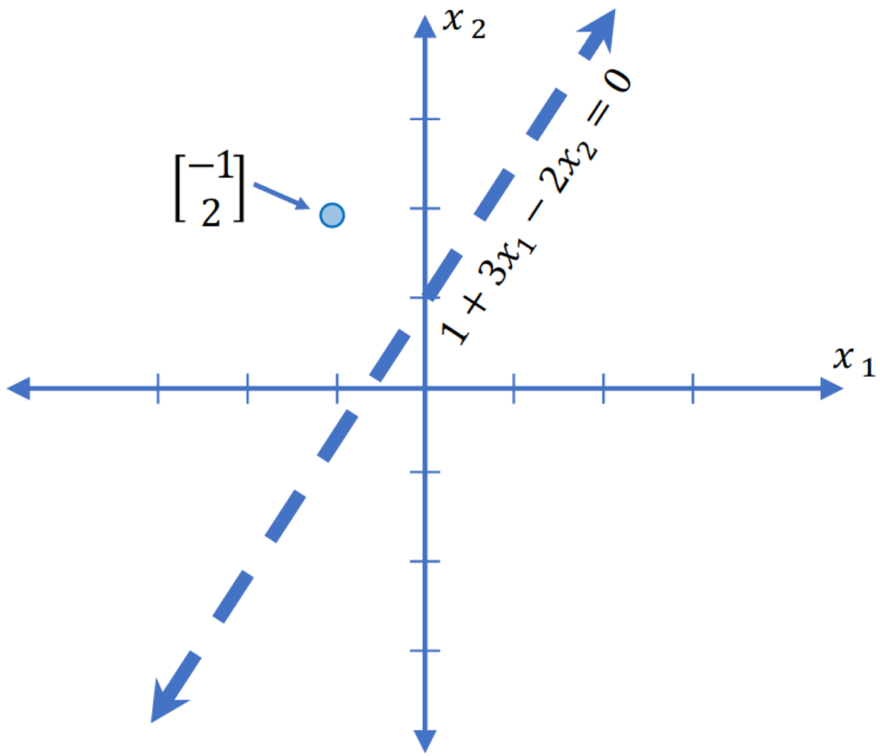
The Perceptron: Example



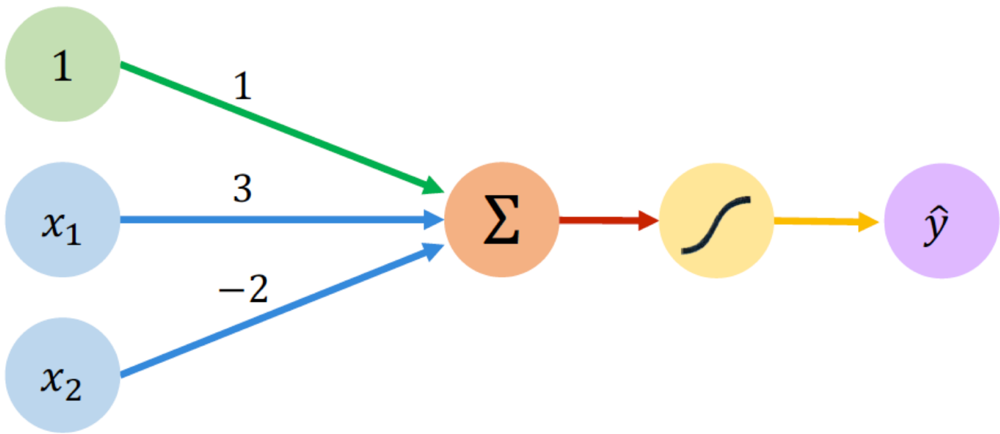
Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

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

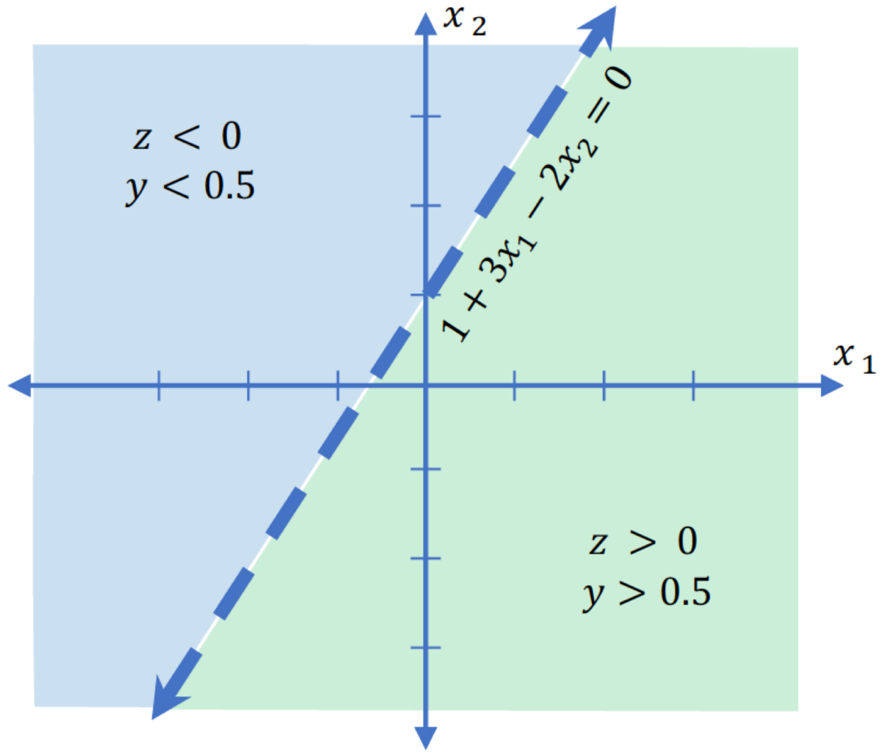
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



The Perceptron: Example

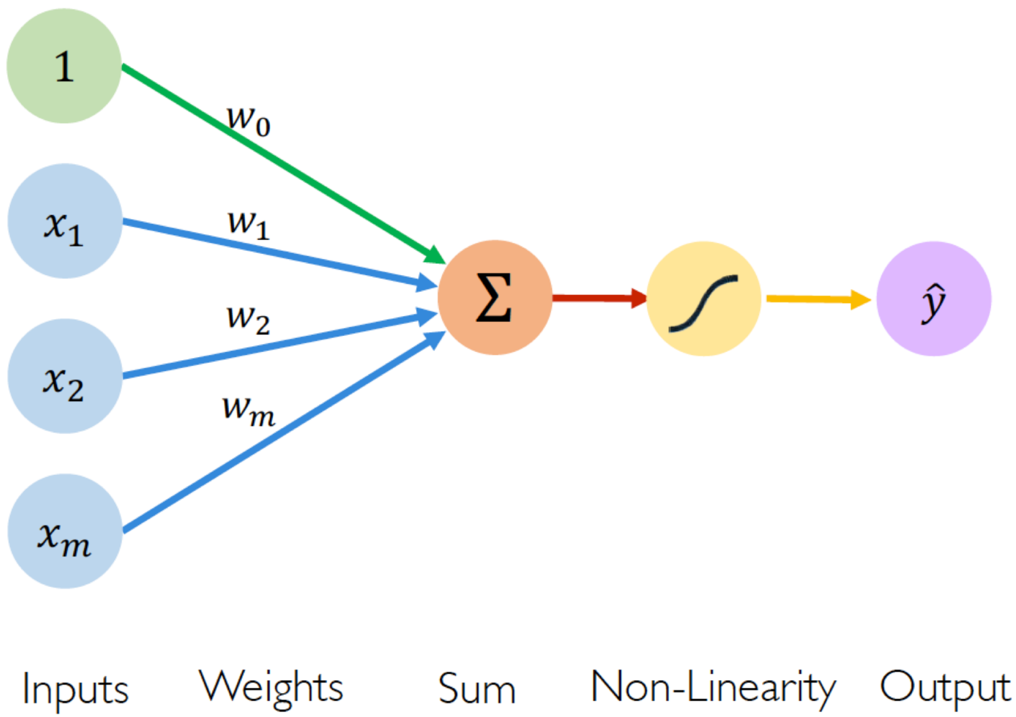


$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



Building Neural Networks with Perceptrons

The Perceptron: Simplified



The Perceptron: Simplified

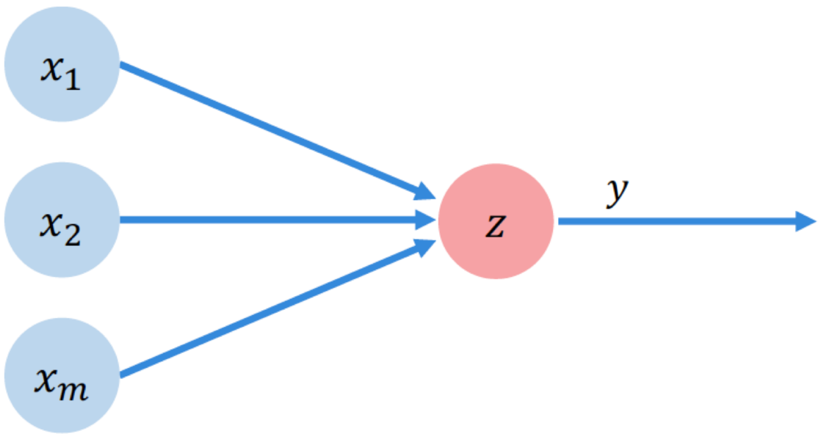
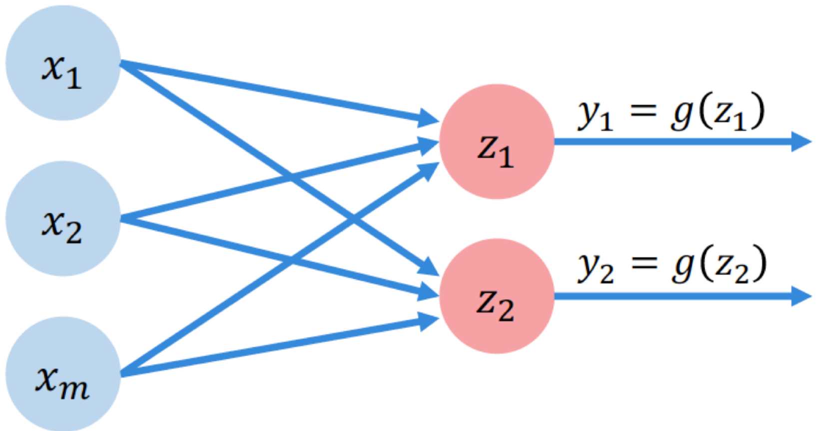


Diagram simplification
- No Bias
- No weights
- z: input to the a.f.
- y: output of the a.f.

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

Multi Output Perceptron



- Simply add a perceptron
- Same input
- Same process
- **What changes are the weights**

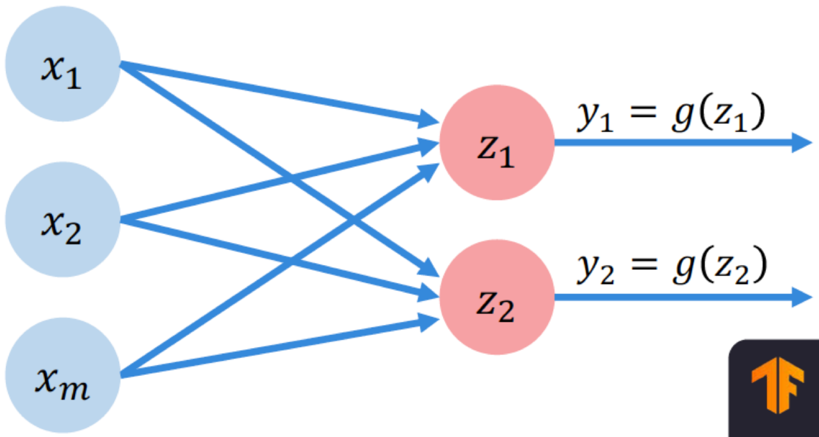
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

Dense layer from scratch

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

    # Initialize weights and bias
    self.W = self.add_weight([input_dim, output_dim])
    self.b = self.add_weight([1, output_dim])

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

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

    return output
```

Multi Output Perceptron

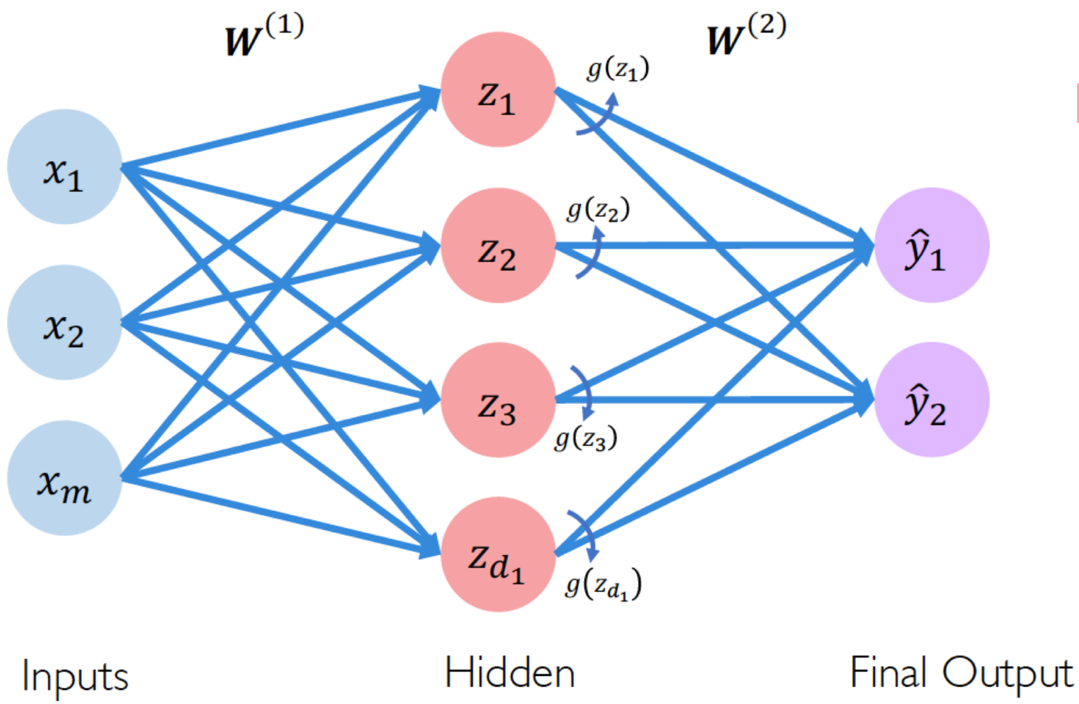
Because all inputs are densely connected to all outputs, these layers are called **Dense** layers




```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

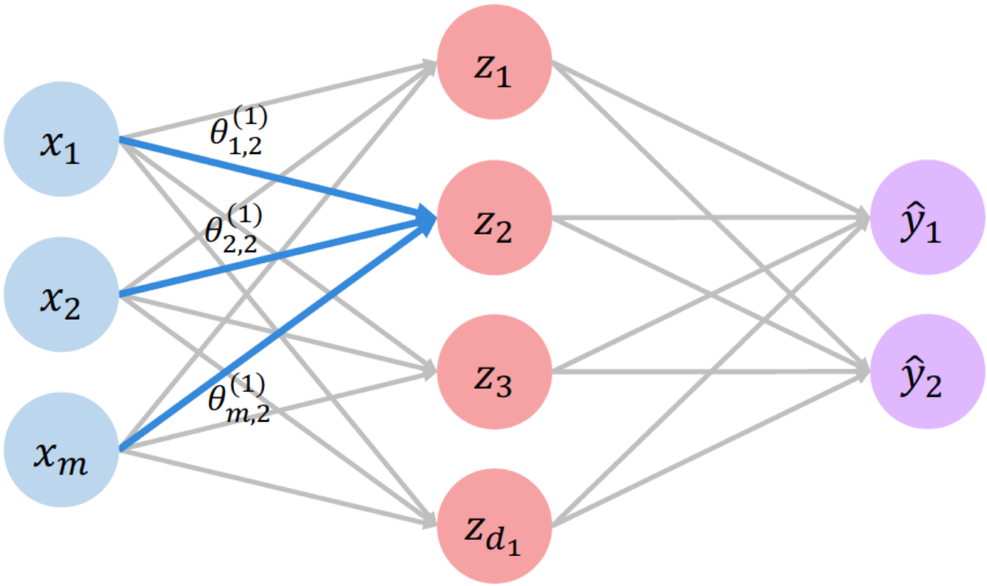


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)}\right)$$

Hidden layer(s):

- Not observable
- To be learned
- No specific behaviour enforced
- 2 weight matrices
- Same operation as before (dot product, bias, a.f.)

Single Layer Neural Network



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

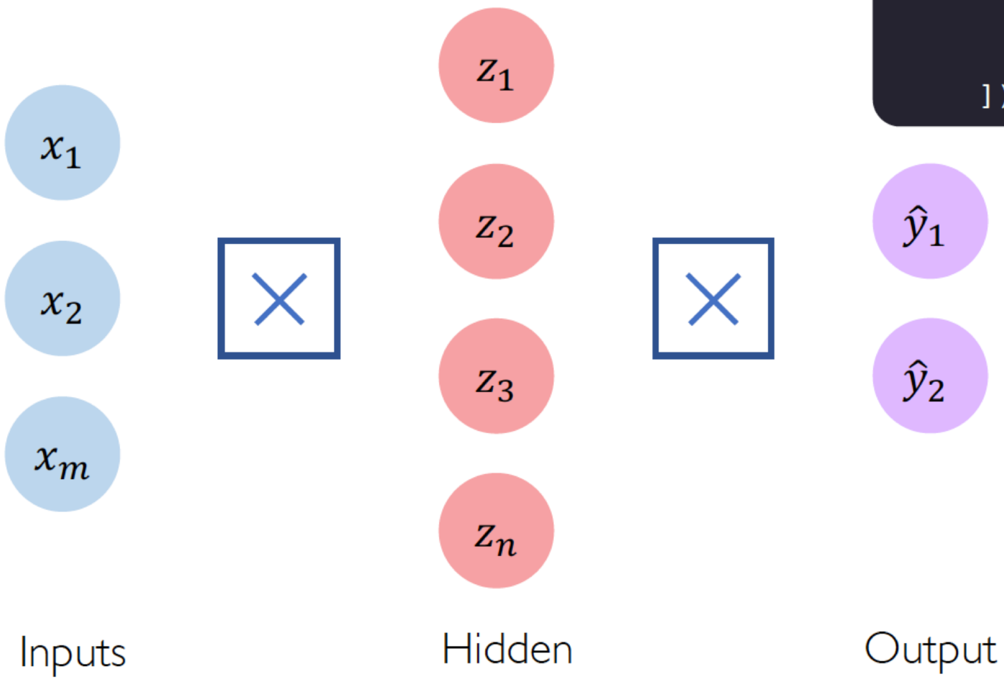
$$= w_{0,2}^{(1)} + x_1 \, w_{1,2}^{(1)} + x_2 \, w_{2,2}^{(1)} + x_m \, w_{m,2}^{(1)}$$

Zoom in into a single hidden layer, say z_2:

- Same operation as before (dot product, bias, a.f.)
- Same for z_3, what changes are the **weights**

Multi Output Perceptron

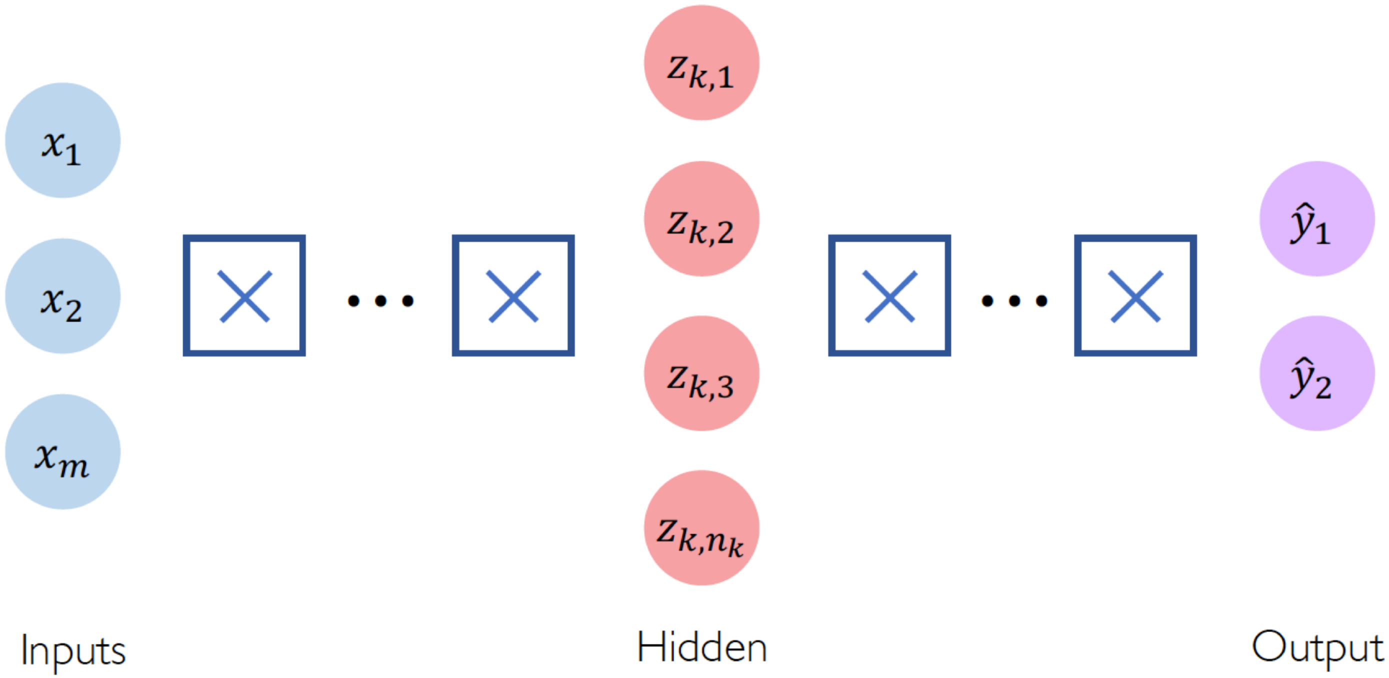
```python
import tensorflow as tf

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



 : replace connections.

Stands for:
Fully connected layer
OR
Dense layer

Deep Neural Network




```python
import tensorflow as tf

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

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

Deep Neural Network

- Stack hidden layer back to back to back to create increasingly deeper and deeper models.

- Output computed going deeper into the NN and computing these weighted sums over and over and over again with these a.f. repeatedly applied

Applying Neural Networks

Example Problem

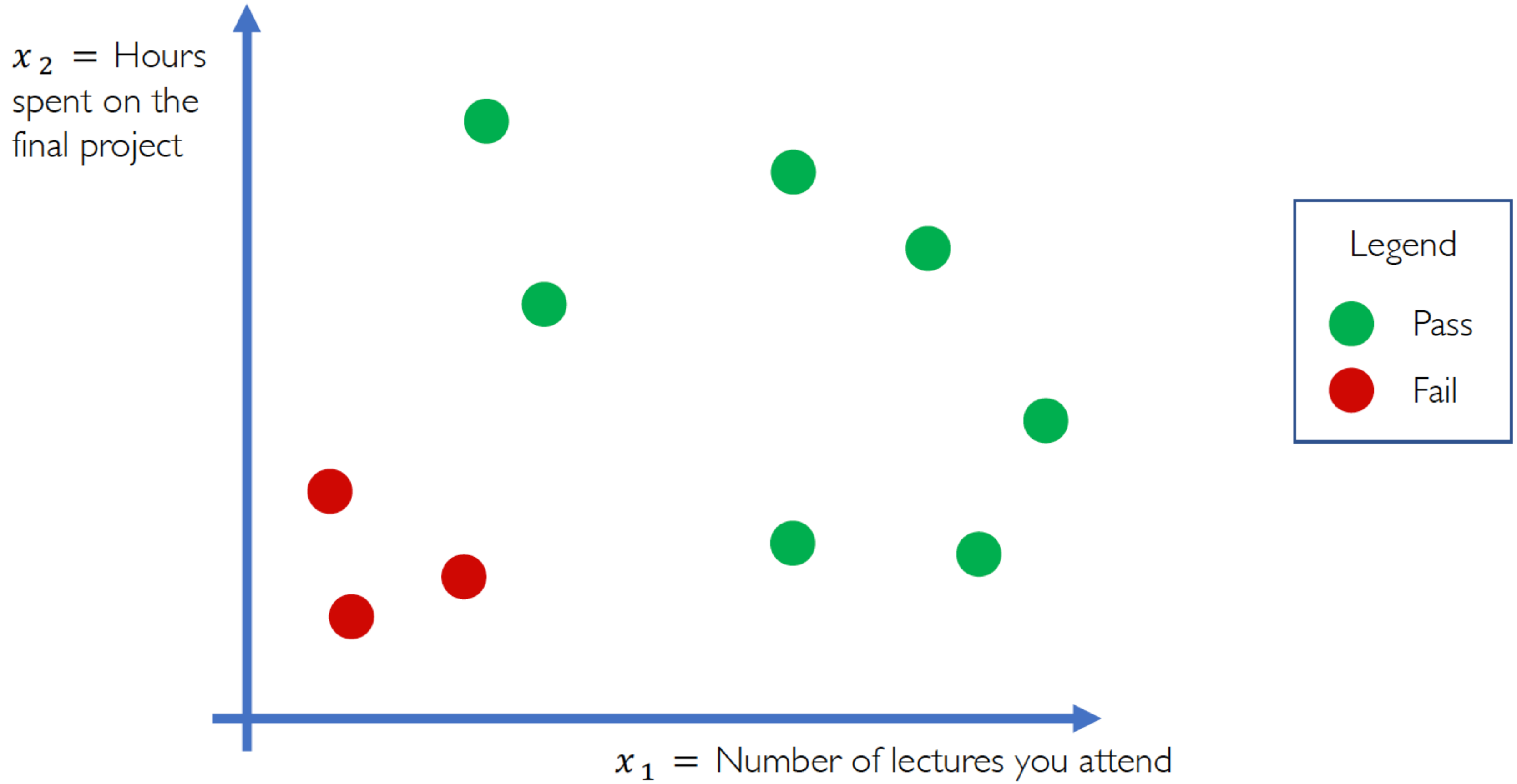
Will I pass this class?

Let's start with a simple two feature model

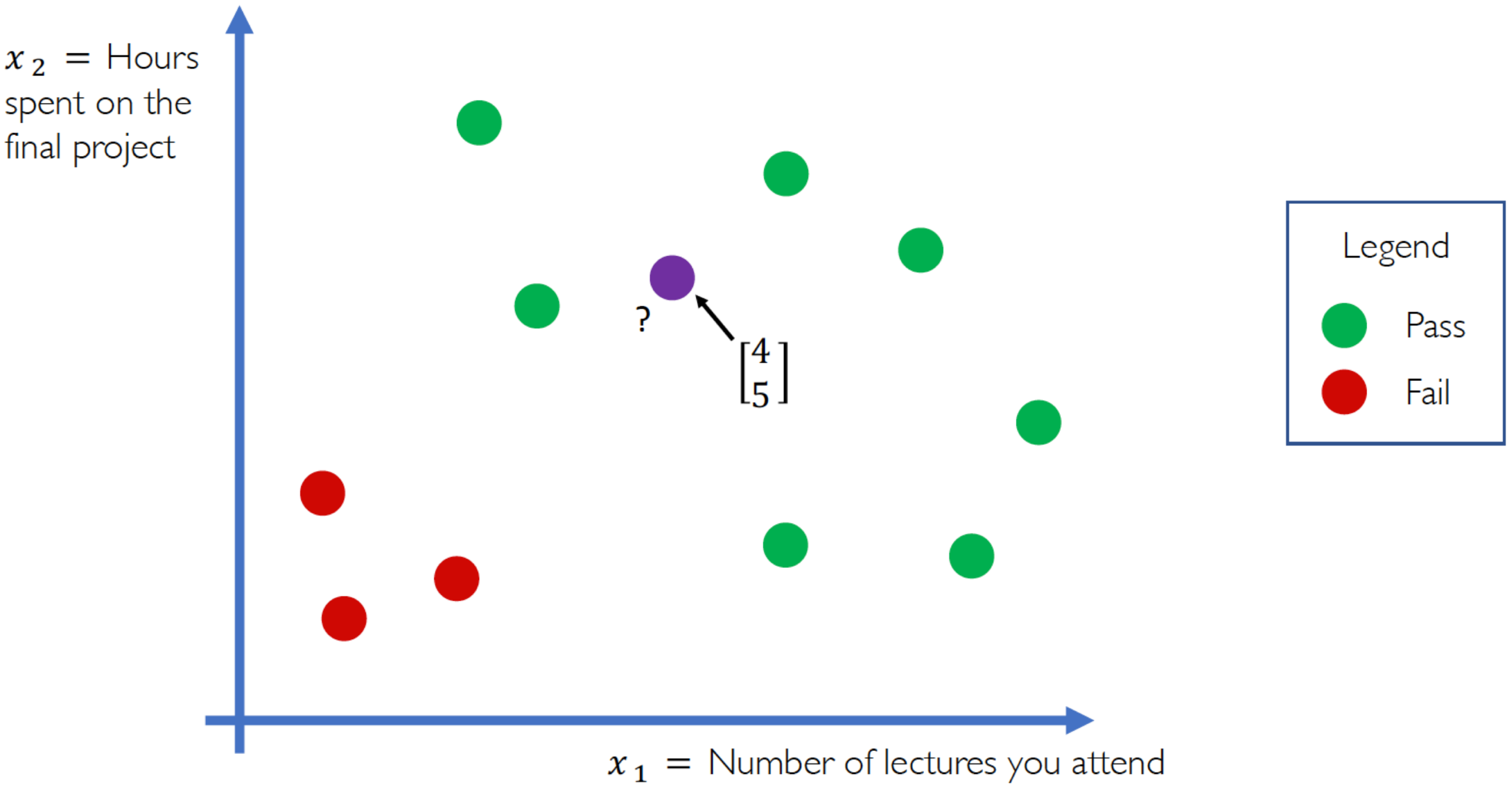
x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

Example Problem: Will I pass this class?

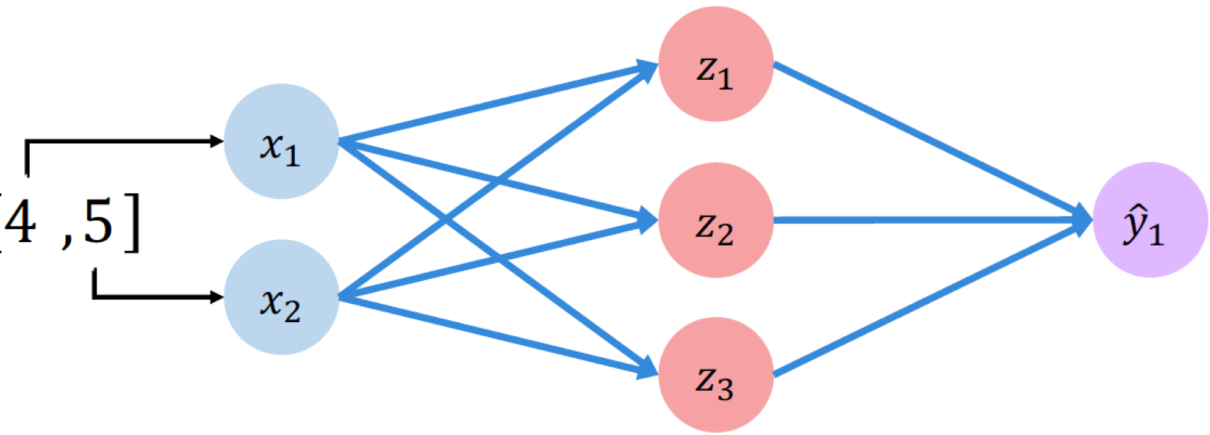


Example Problem: Will I pass this class?



Example Problem: Will I pass this class?

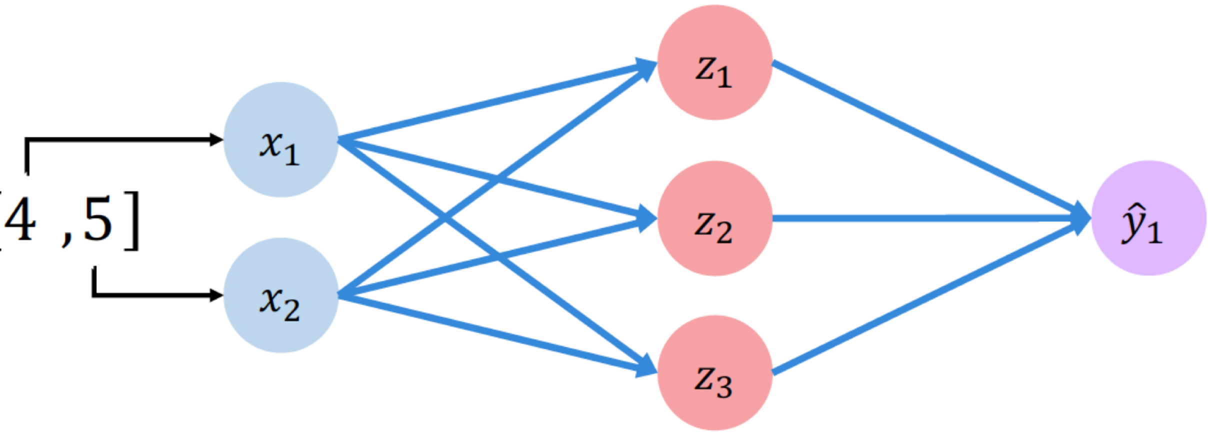
$x^{(1)} = [4\ ,5]$



Predicted: **0.1**

Example Problem: Will I pass this class?

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



Predicted: **0.1**
Actual: **1**

Example Problem: Will I pass the exam?

Why **Wrong prediction**?

- Because the network is not trained

Train a network: teach it to get the right answer. How?

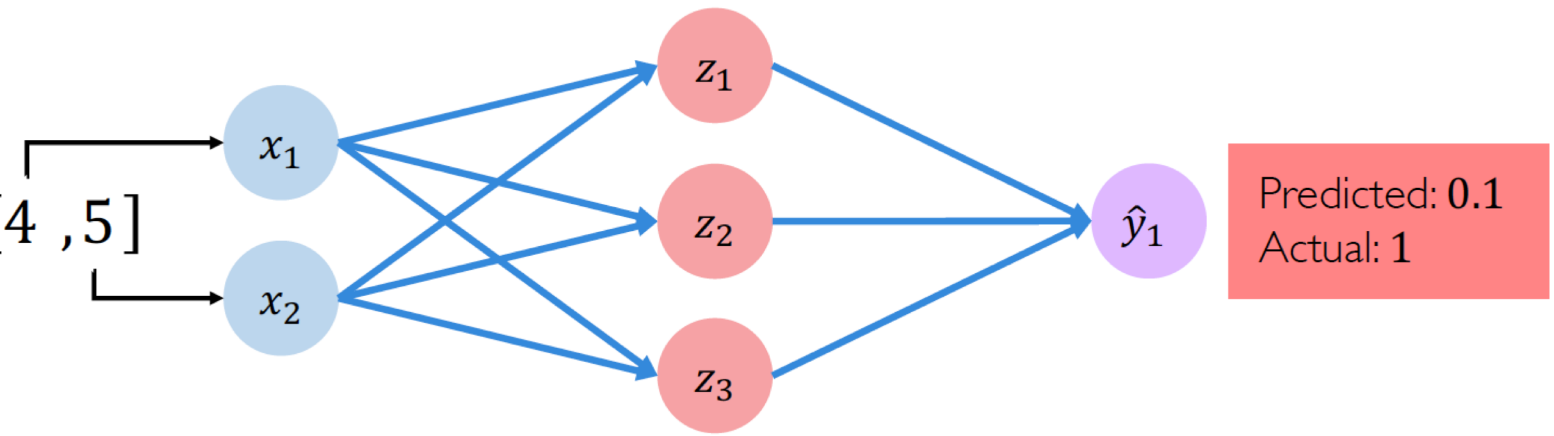
- tell it when it makes a mistake, so to correct it in the future

The **Loss** of a network is what quantify the wrong prediction

Quantifying Loss

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

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



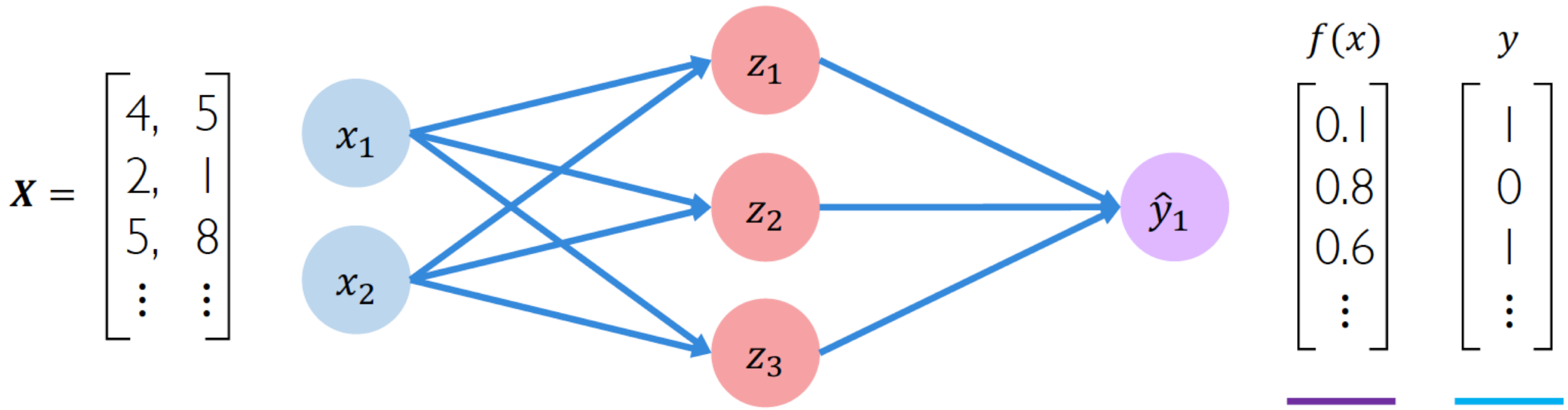
$$\mathcal{L}\left(f\left(x^{(i)};\boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Empirical Loss

When we train a network, we do not want to **minimize** the loss for a particular student, but **the loss across the entire training set**

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



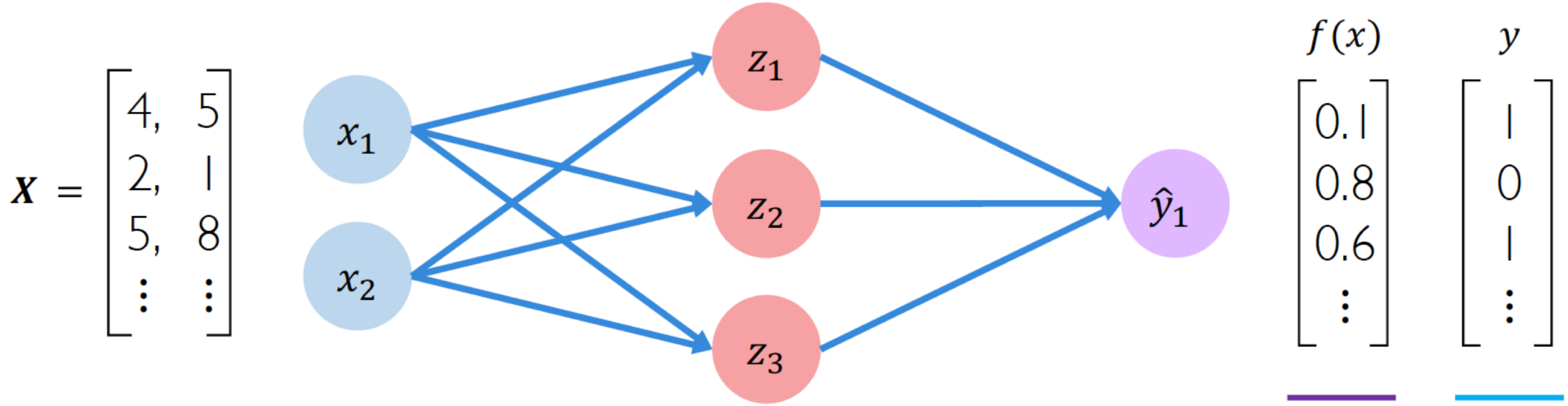
$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\big(\underbrace{f(x^{(i)};\boldsymbol{W})}_{\text{Predicted}},\underbrace{y^{(i)}}_{\text{Actual}}\big)$$

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*

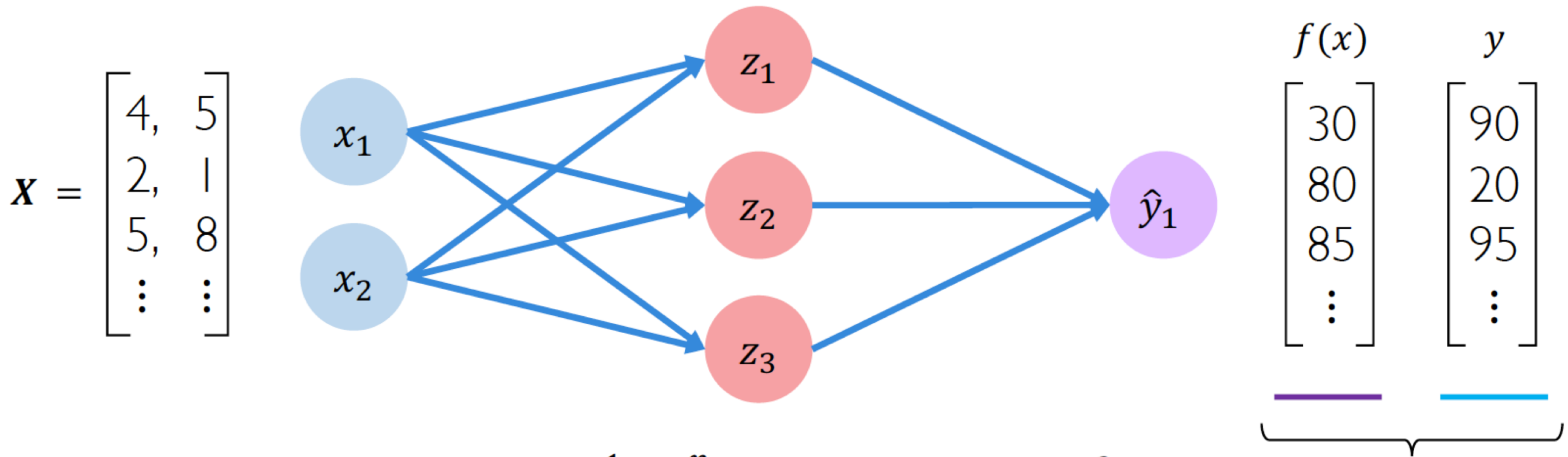


$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\Big(1 - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

```
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*



Final Grades
(percentage)

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

Actual Predicted

```
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***

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

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

Loss Optimization

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

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

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

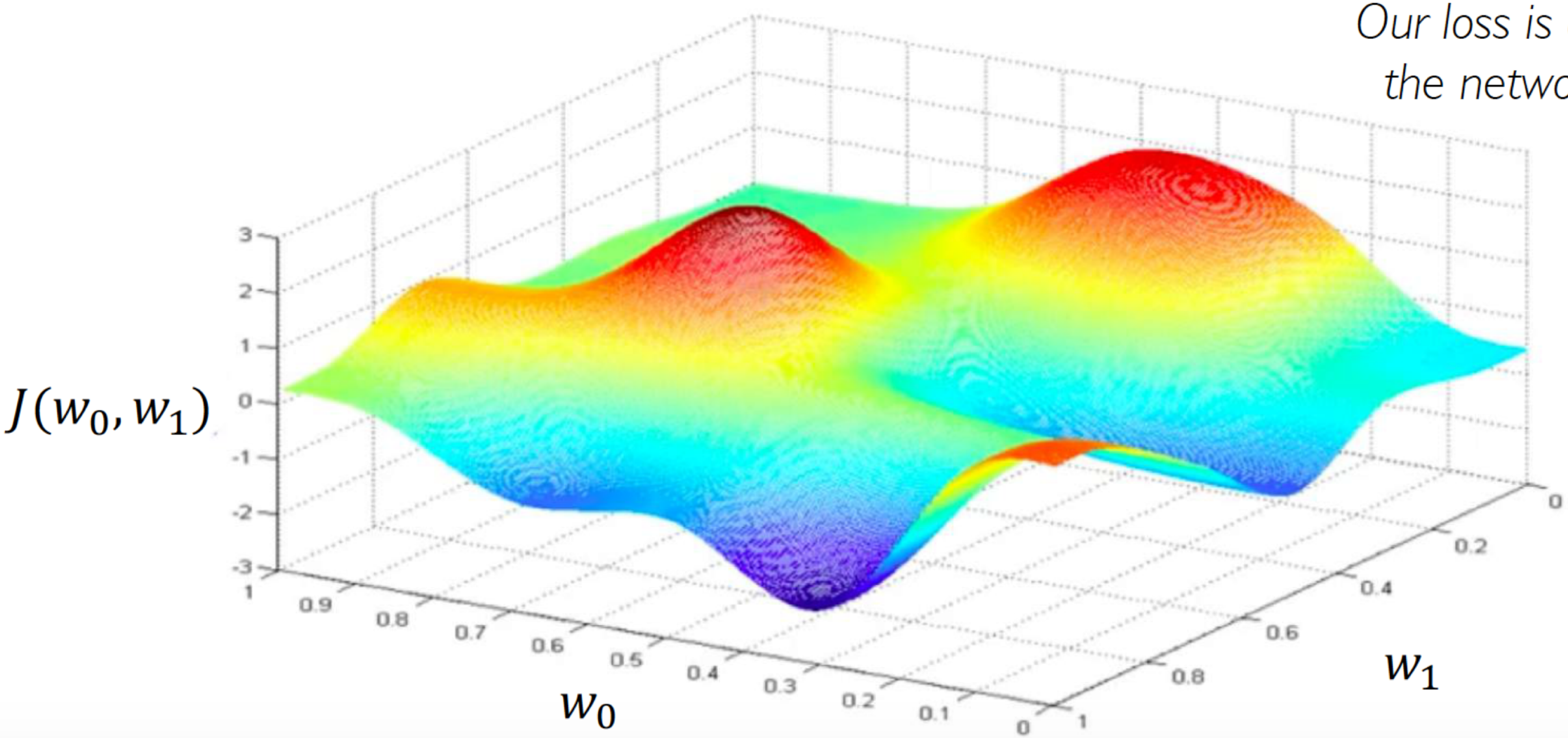
Remember:

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

Loss Optimization

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

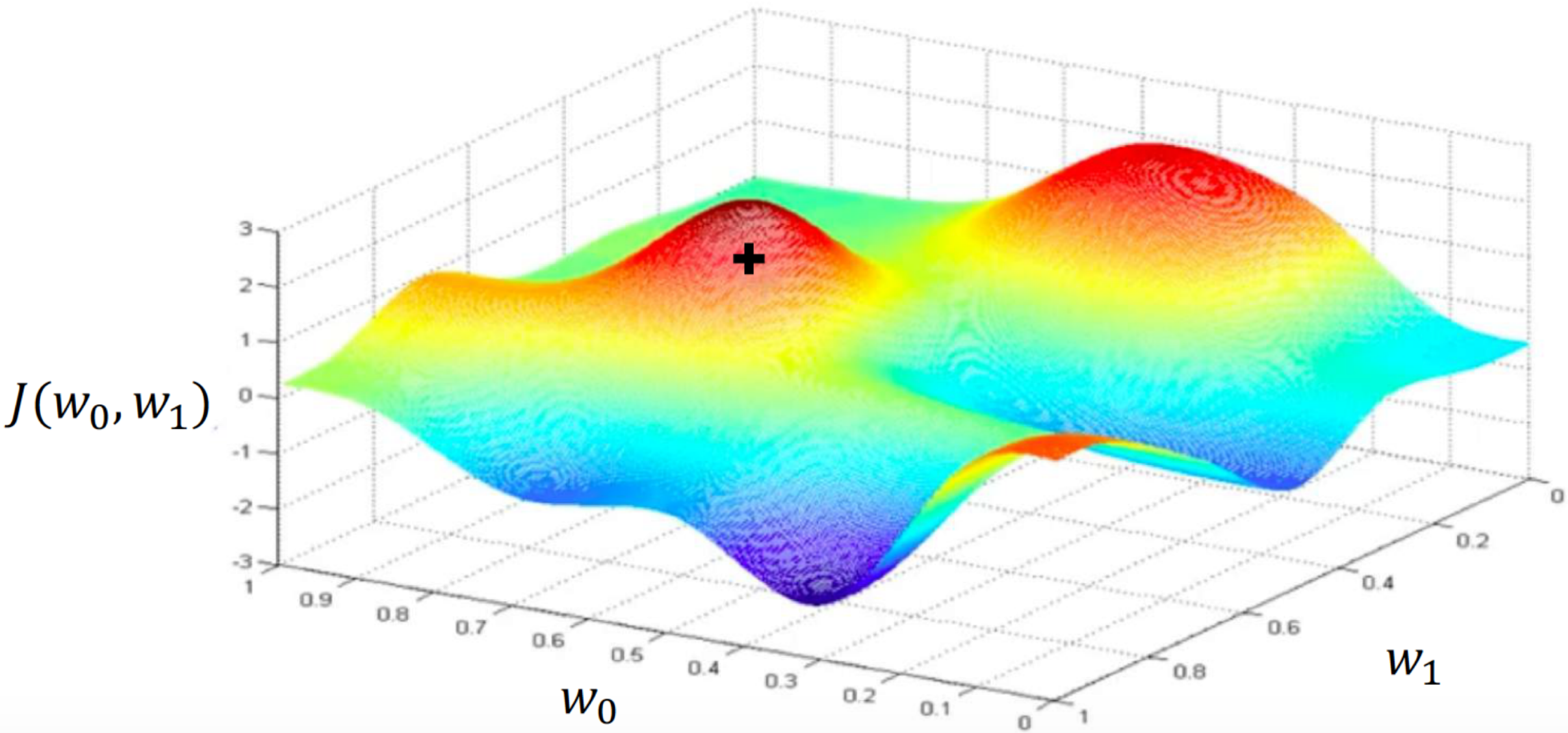
Remember:
Our loss is a function of the network weights!



Loss Optimization

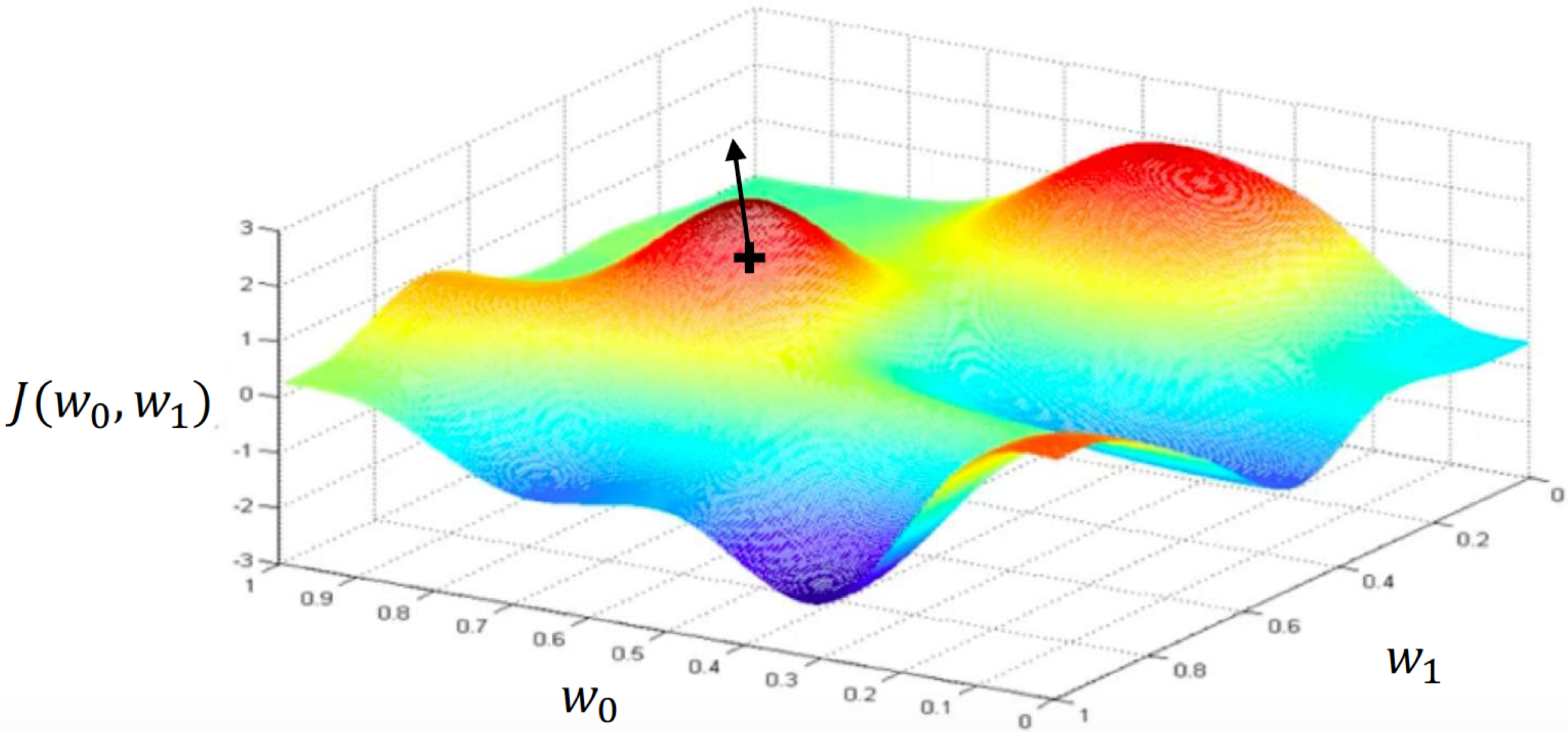
Loss optimization through **gradient descent**

Randomly pick an initial (w_0, w_1)



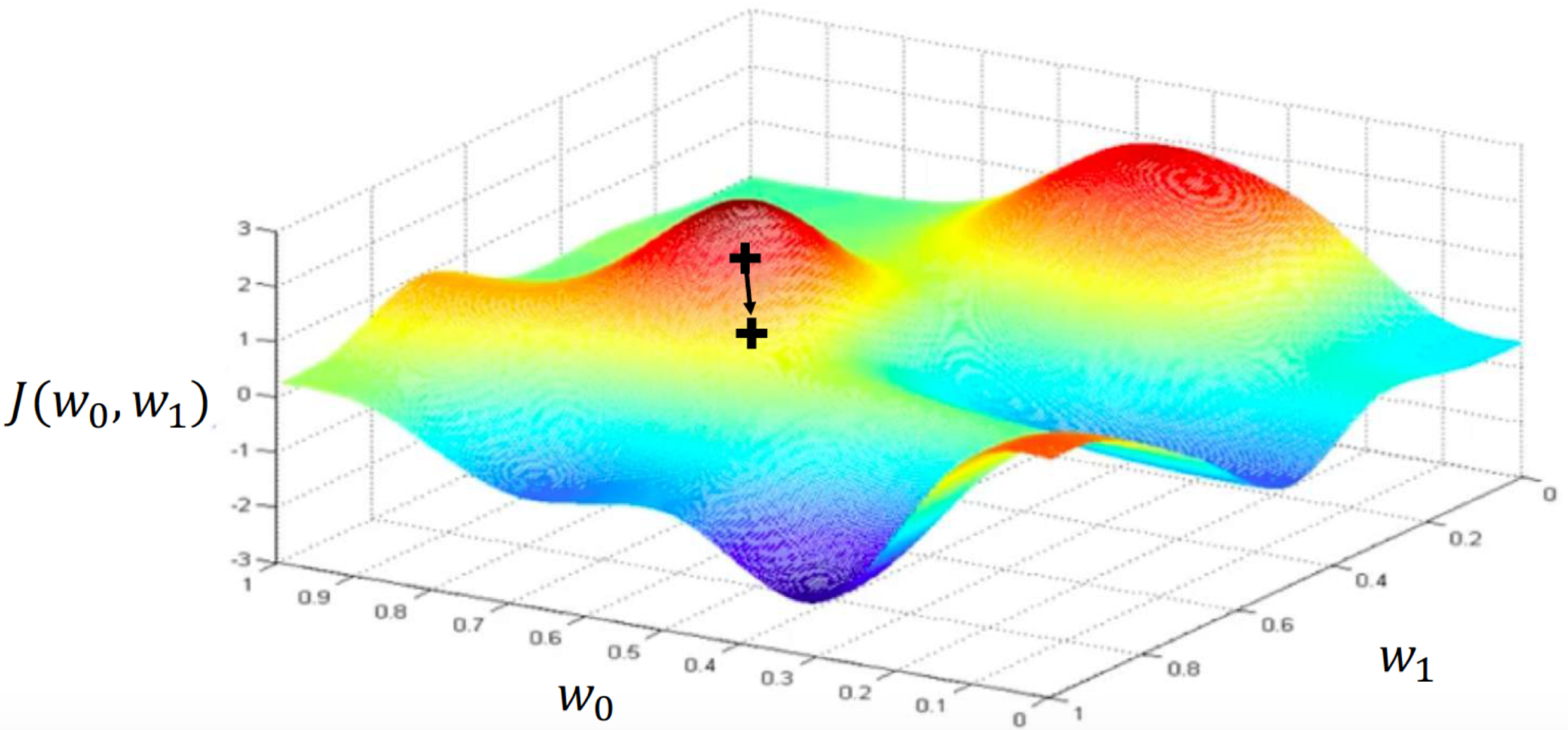
Loss Optimization

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



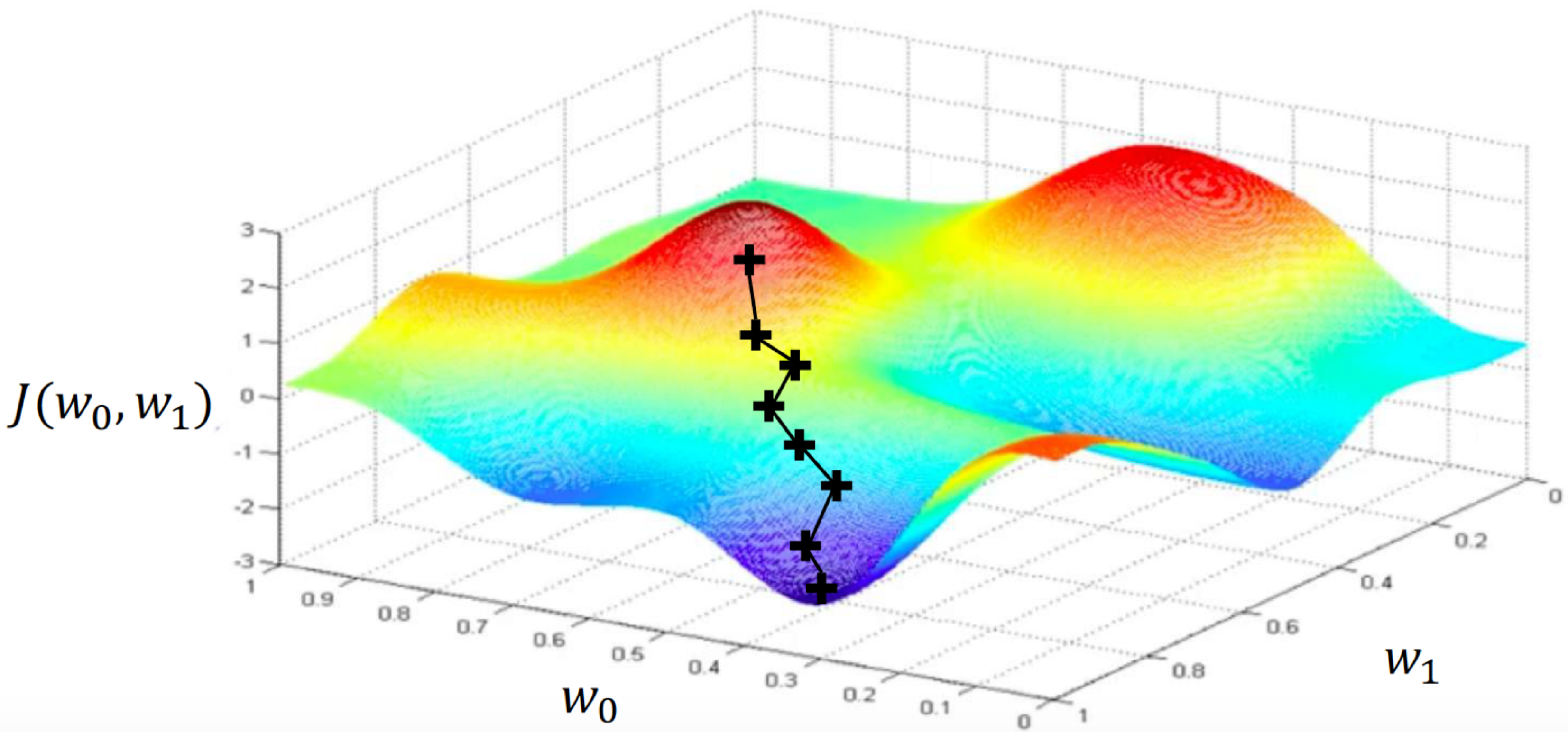
Loss Optimization

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

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

```
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
```

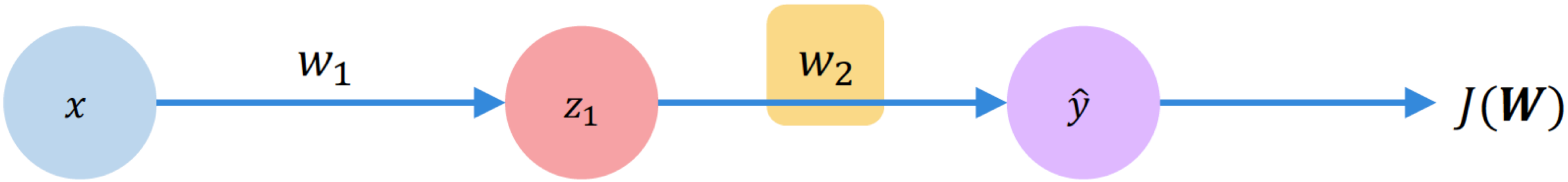
Gradient Descent

Algorithm

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

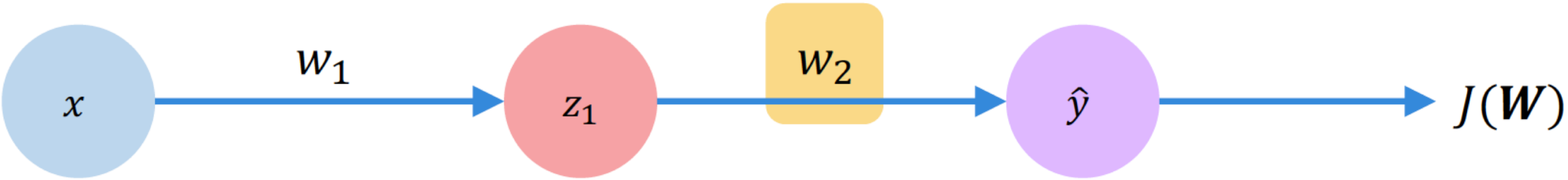
```
import tensorflow as tf

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 $J(\boldsymbol{W})$?

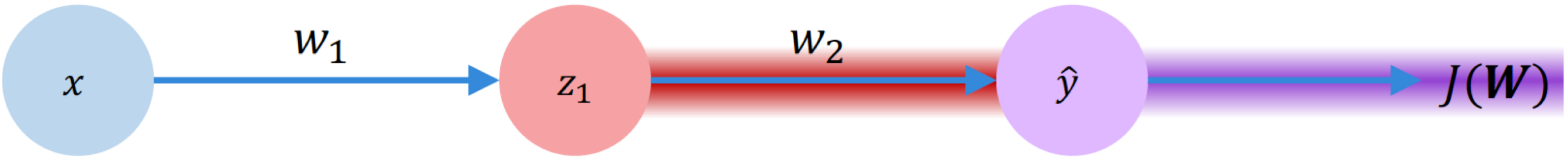
Computing Gradients: Backpropagation



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

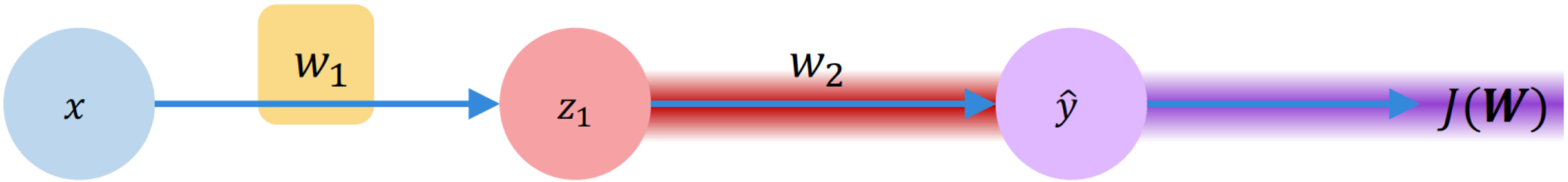
Let's use the chain rule!

Computing Gradients: Backpropagation



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

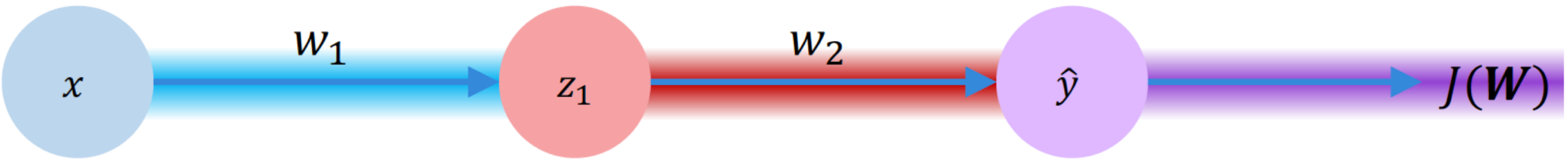
Computing Gradients: Backpropagation



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

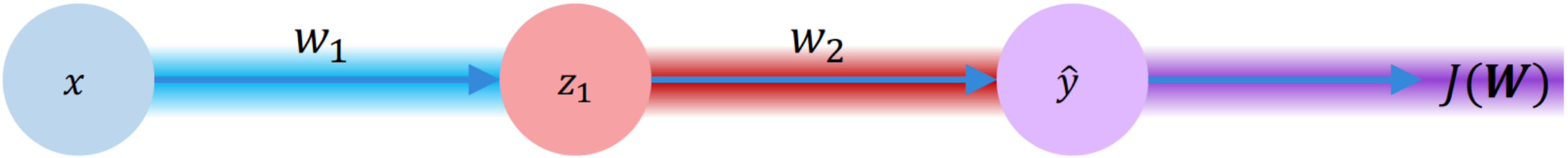
Apply chain rule! Apply chain rule!

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

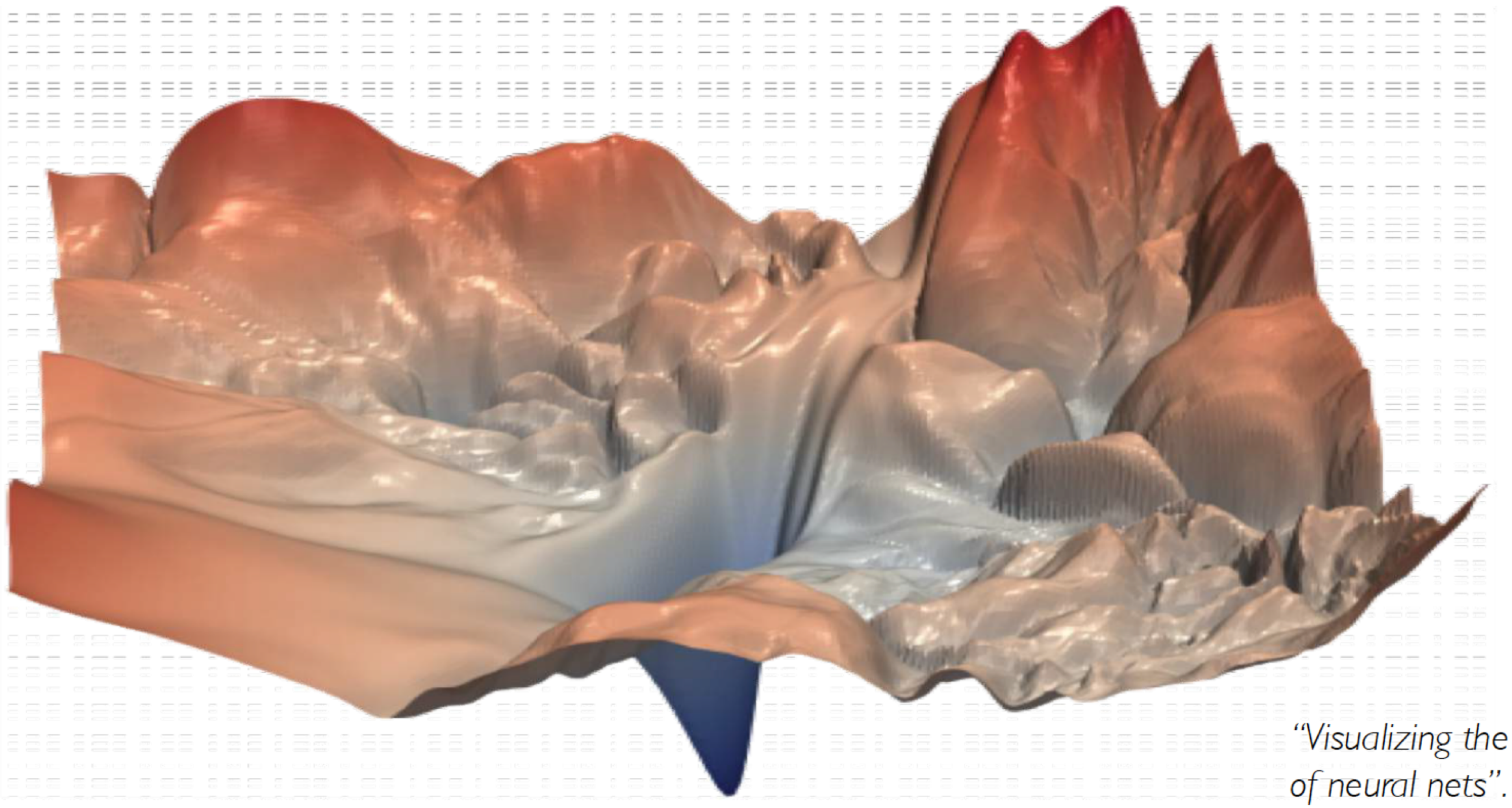


$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

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

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

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

Loss Functions Can Be Difficult to Optimize

Remember:

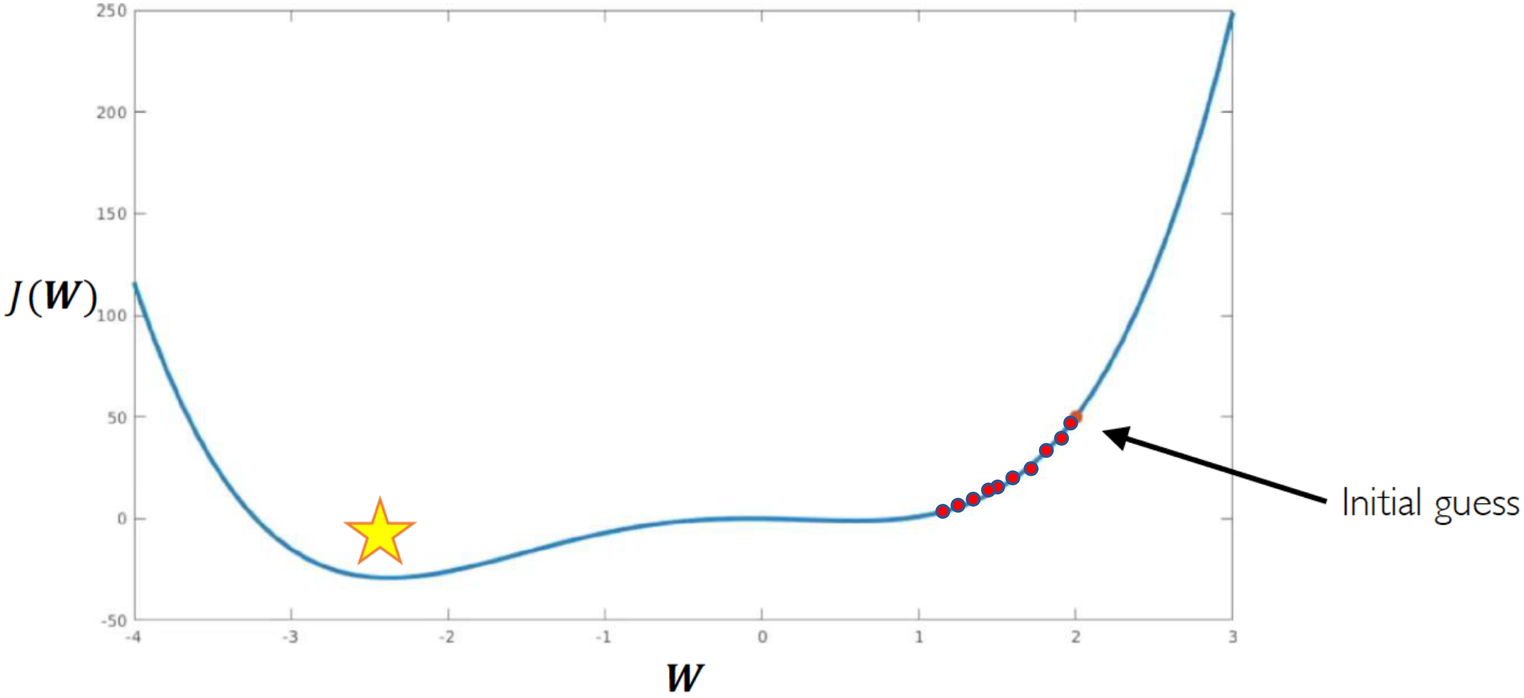
Optimization through gradient descent

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

How can we set the learning rate?

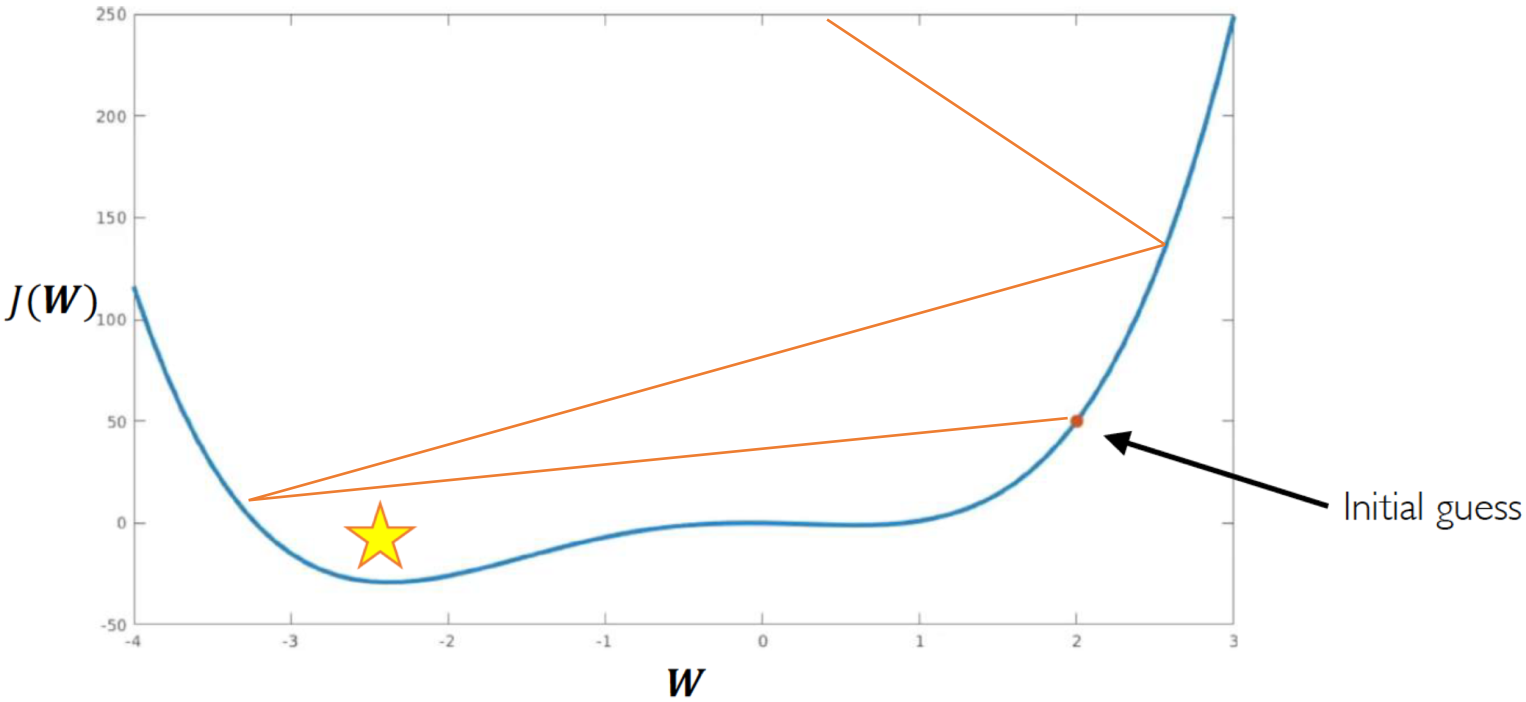
Setting the Learning Rate

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



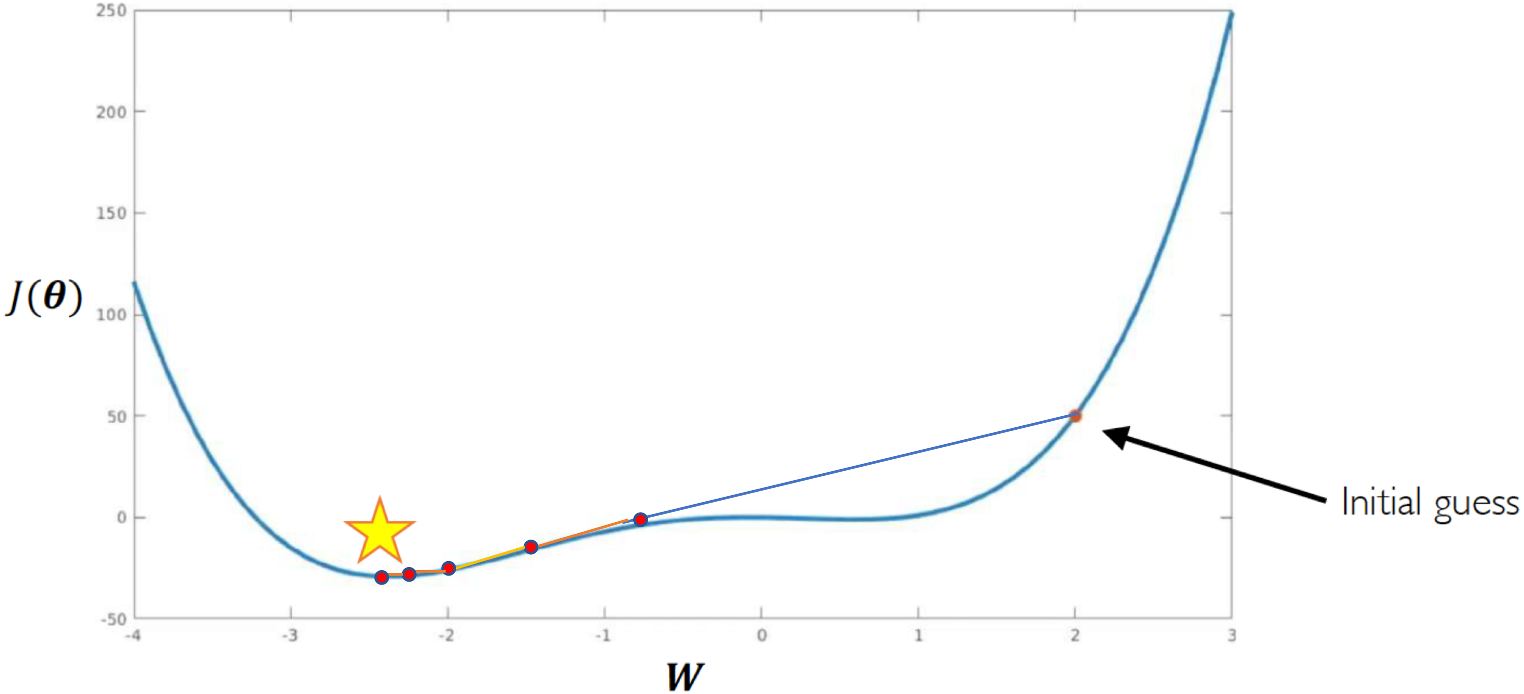
Setting the Learning Rate

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 non longer fixed
- Can be made larger or smaller depending on:
  - How large gradient is
  - How fast learning is happening
  - Size of particular weights
  - …

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/

Putting it all together

```python
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))
```

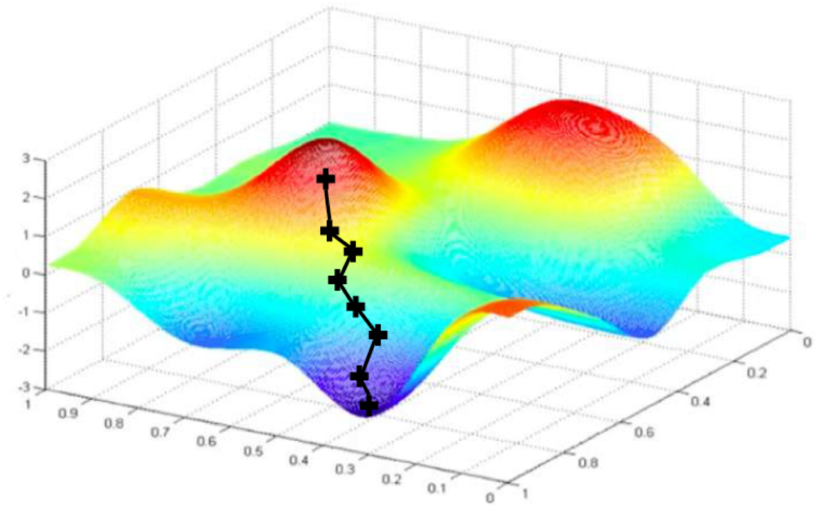
Can replace with any TensorFlow optimizer!

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

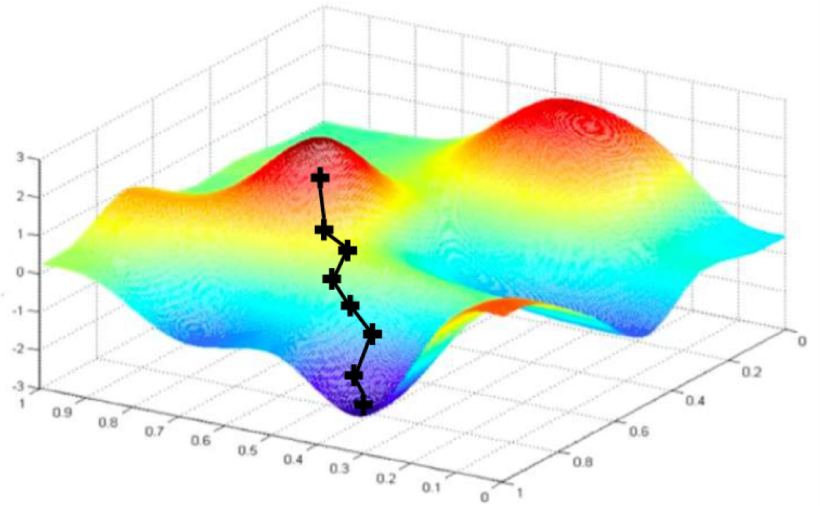


Gradient Descent

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

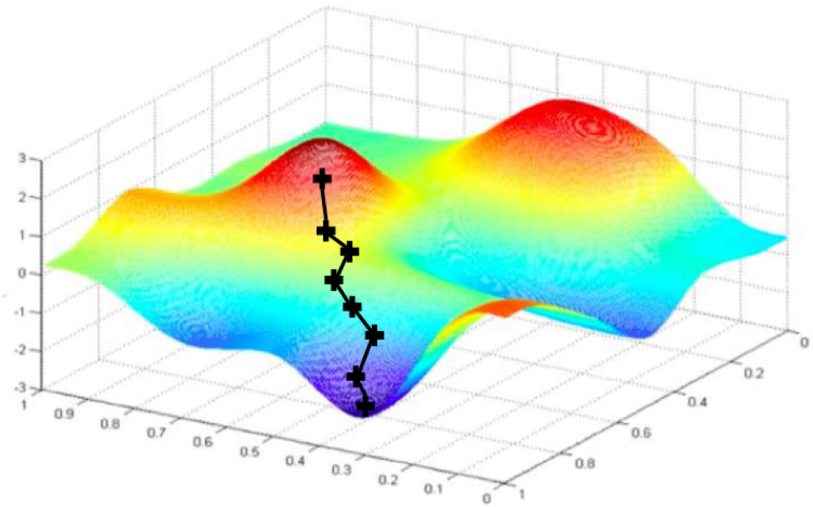
Can be very computational to compute!



Stochastic Gradient Descent

Algorithm

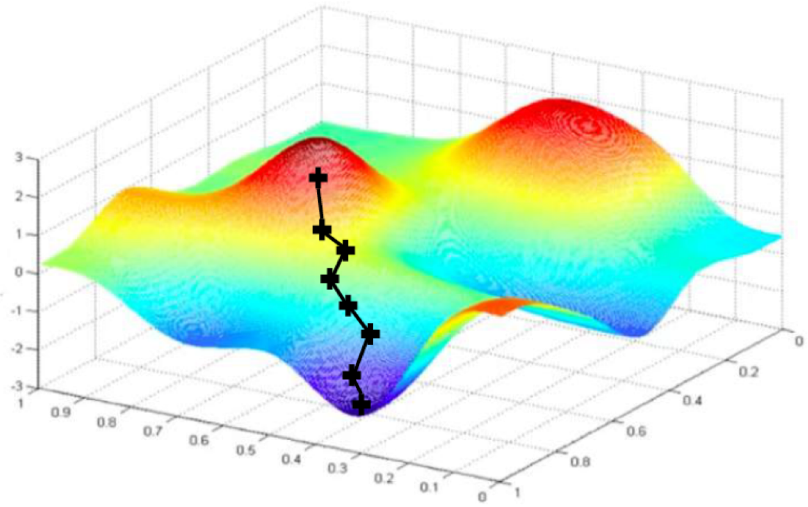
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point $\boldsymbol{i}$
4. Compute gradient, $\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial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

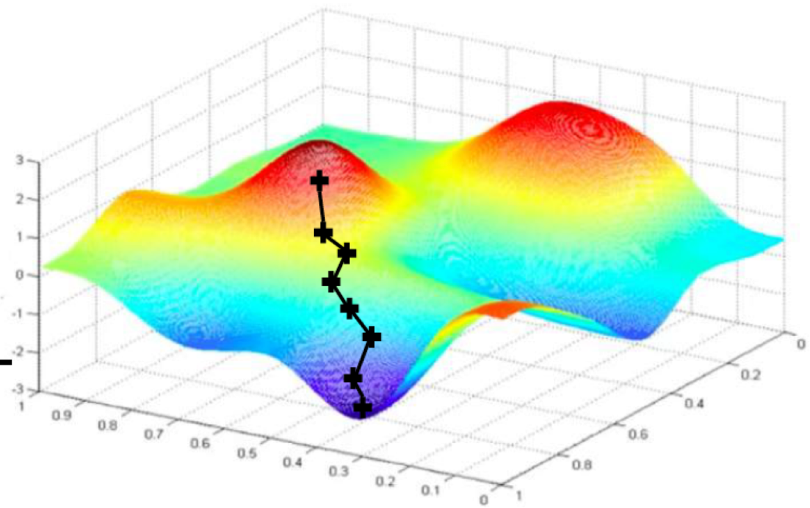


Easy to compute but **very noisy** (stochastic)!

Stochastic Gradient Descent

Algorithm

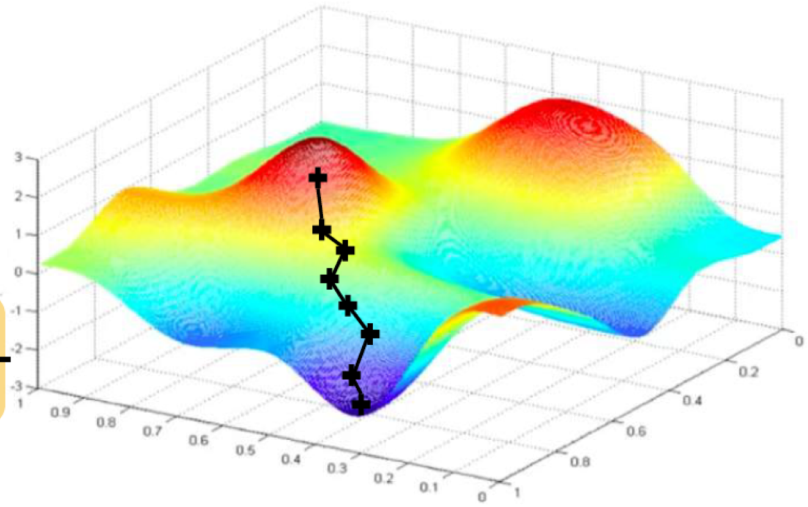
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update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

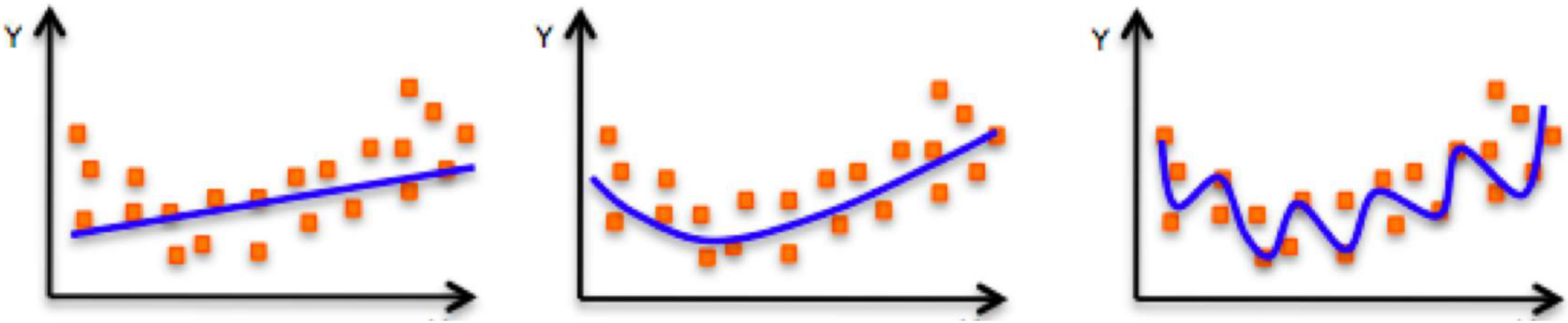
  Can parallelize computation + achieve significant speed increases on GPU's

Some terminology

- One **epoch** is when the entire dataset is passed forward and backward through the neural network only once (multiple times are usually needed)
- The **batch** size is the number of training examples in a mini-batch
- An **iteration** is the number of batches needed to complete one epoch
- Ex. For a dataset of 10000 sample with mini-batch size 1000, 10 iterations will complete 1 epoch

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?

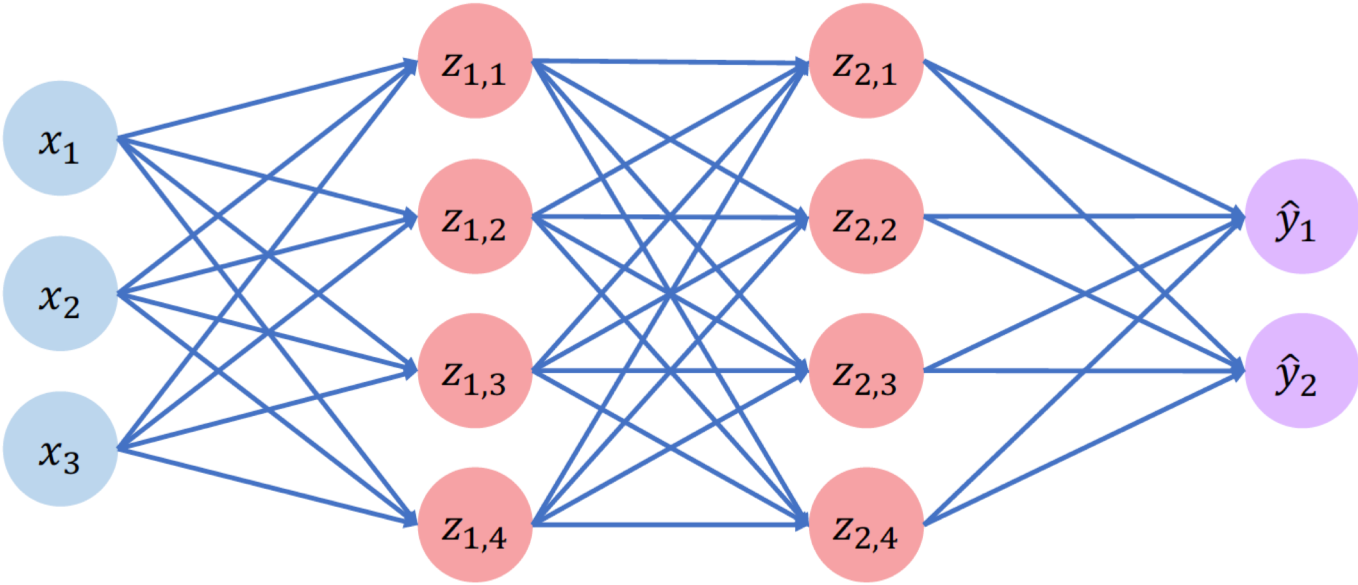
  Technique that constrains our optimization problem to discourage complex models

- Why do we need it?

  Improve generalization of our model on unseen data

Regularization 1: Dropout

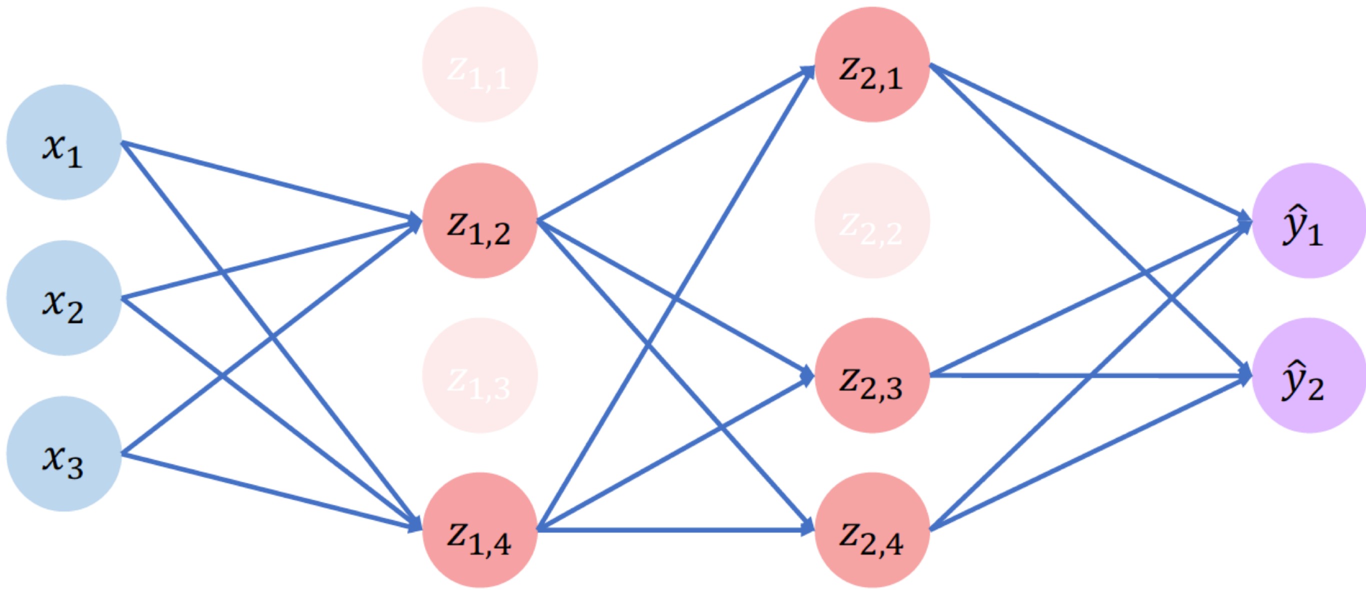
- During training, randomly set some activations to 0



Regularization 1: Dropout

- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer

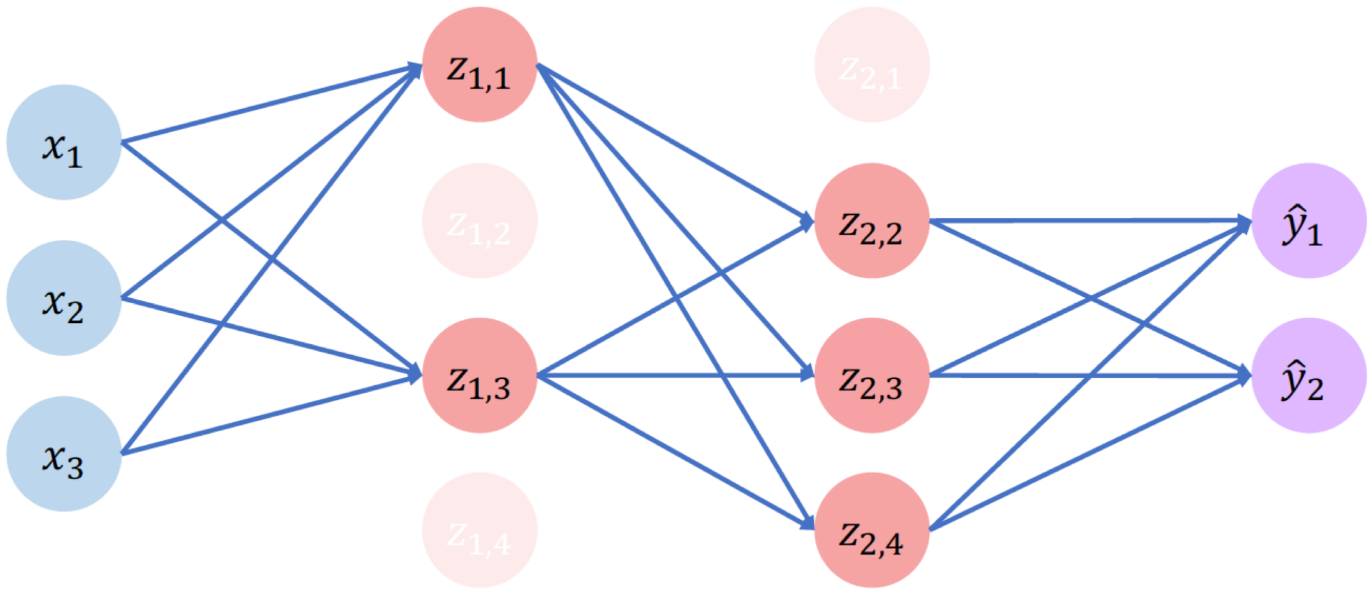
```
tf.keras.layers.Dropout(p=0.5)
```



Regularization 1: Dropout

- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer

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



Regularization I: Dropout

- the network is not going to rely too heavily on any particular path through the network

- instead it's going to **find a whole ensemble of different paths**, because it doesn't know which path is going to be dropped out at any given time

Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

Overfitting:
when a model starts to perform worse on the test (validation) set than on the training set

Regularization 2: Early Stopping

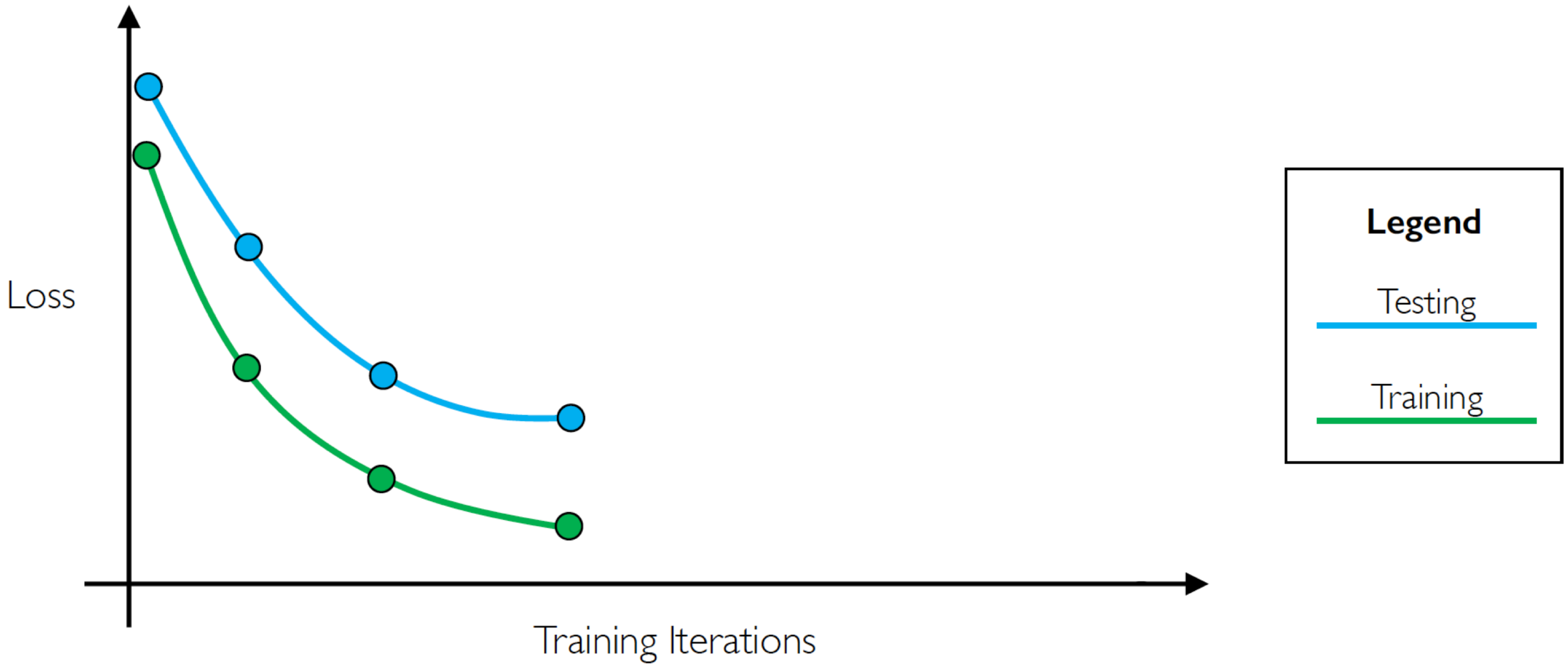
- Stop training before we have a chance to overfit

Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

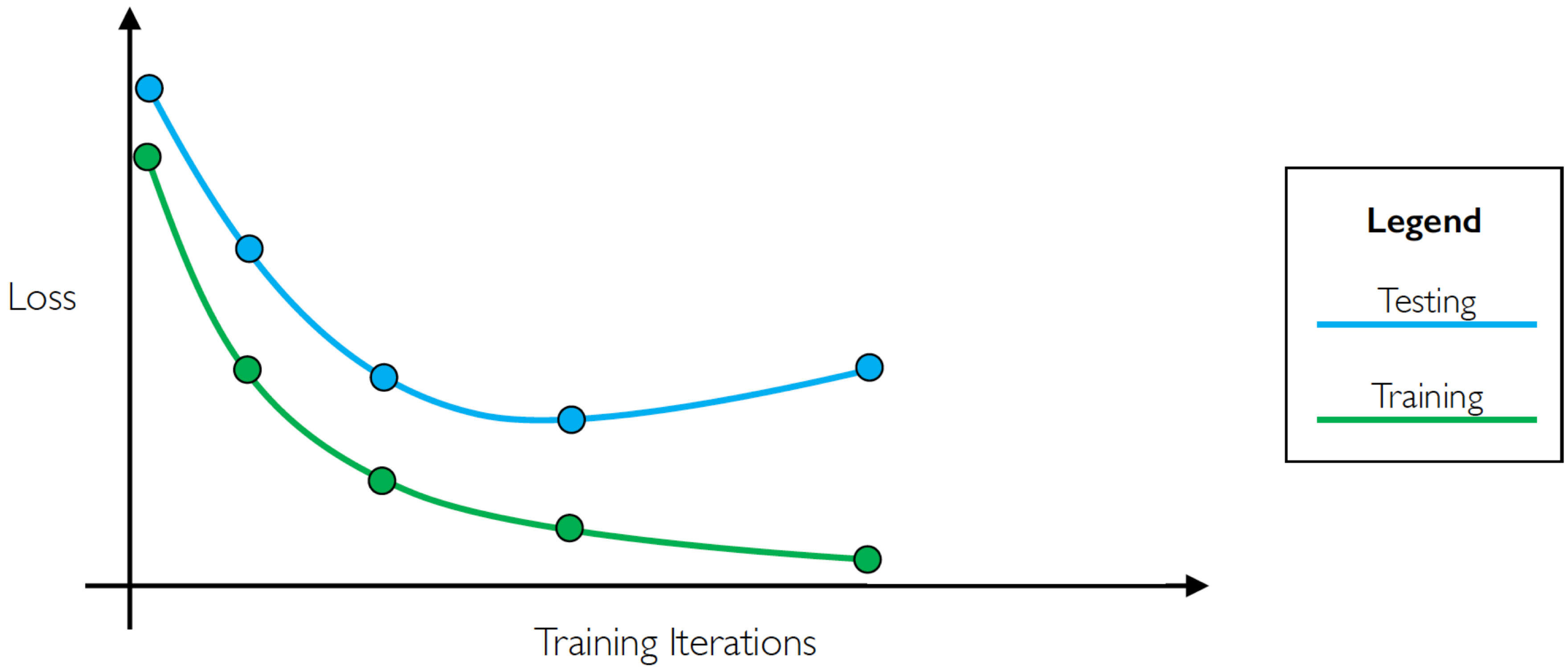
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



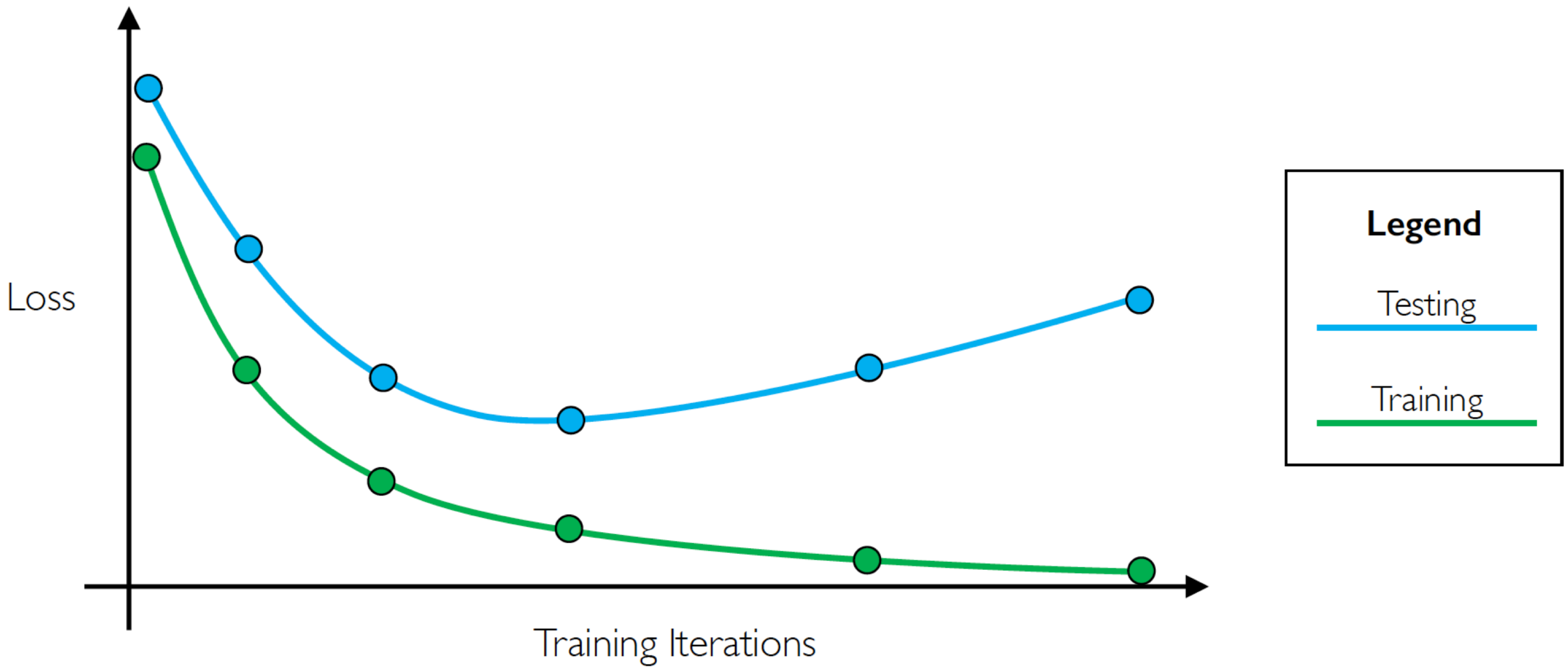
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

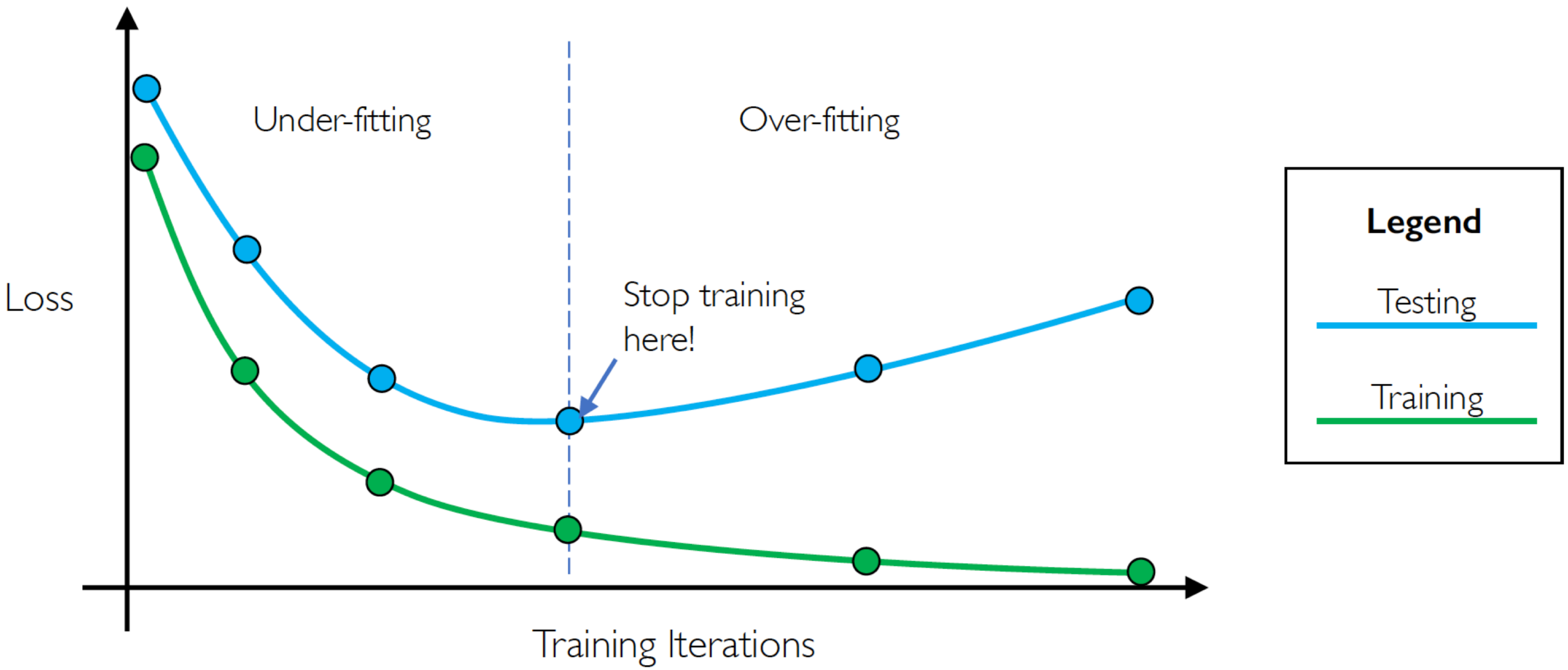


Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

Regularization 2: Early Stopping

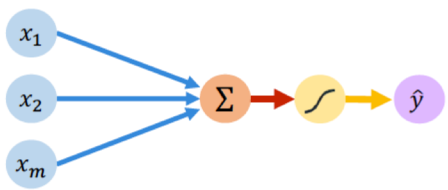
- Stop training before we have a chance to overfit



Core Foundation Review

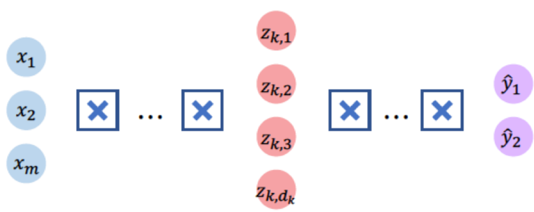
The Perceptron

- Structural building blocks
- Nonlinear activation functions



Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization