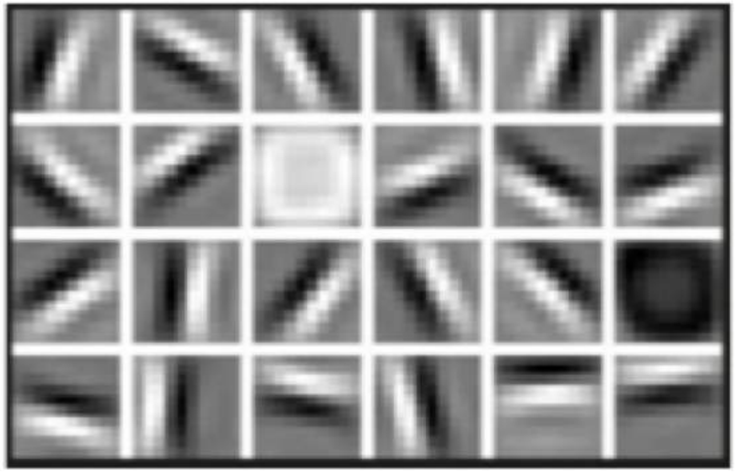
# Neural Network and Deep Learning

# Why Deep Learning?

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

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

**Low Level Features**



Lines & Edges

**Mid Level Features**

Eyes & Nose & Ears

**High Level Features**

Facial Structure

# Why Now?

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

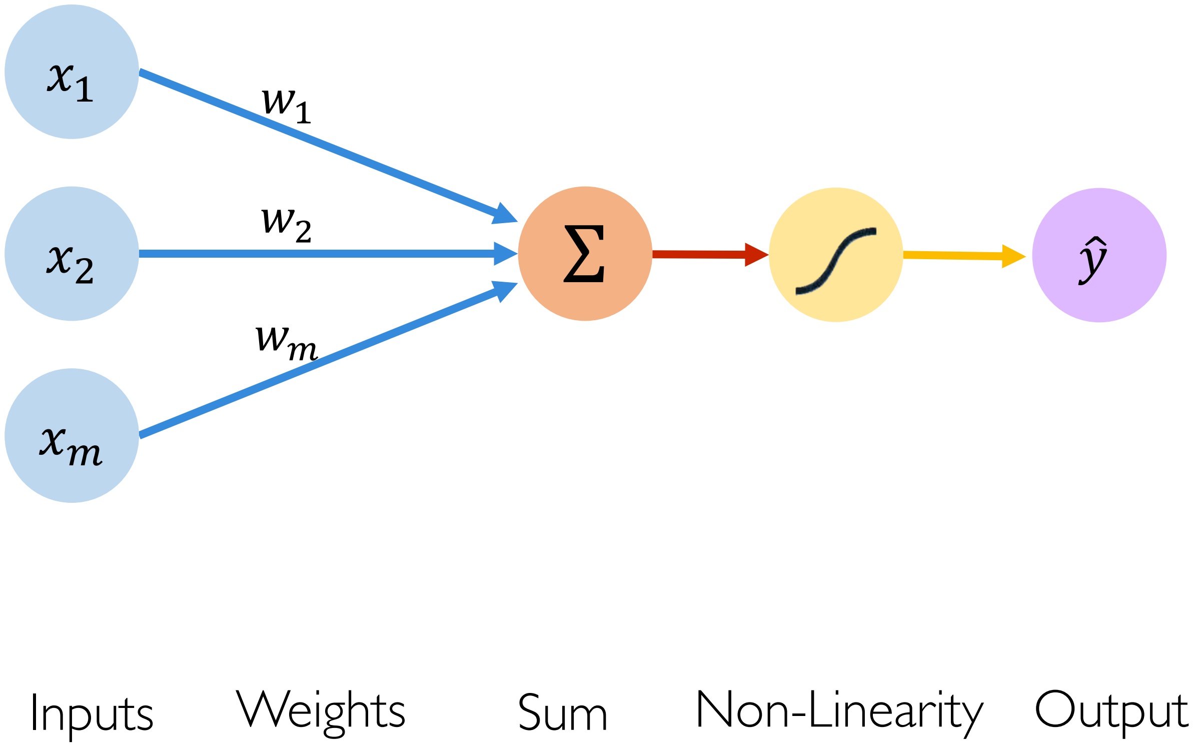
## 2. Hardware
- Graphics Processing Units (GPUs)
- Massively Parallelizable

## 3. Software
- Improved Techniques
- New Models
- Toolboxes

# The Perceptron

## The structural building block of deep learning

# The Perceptron: Forward Propagation



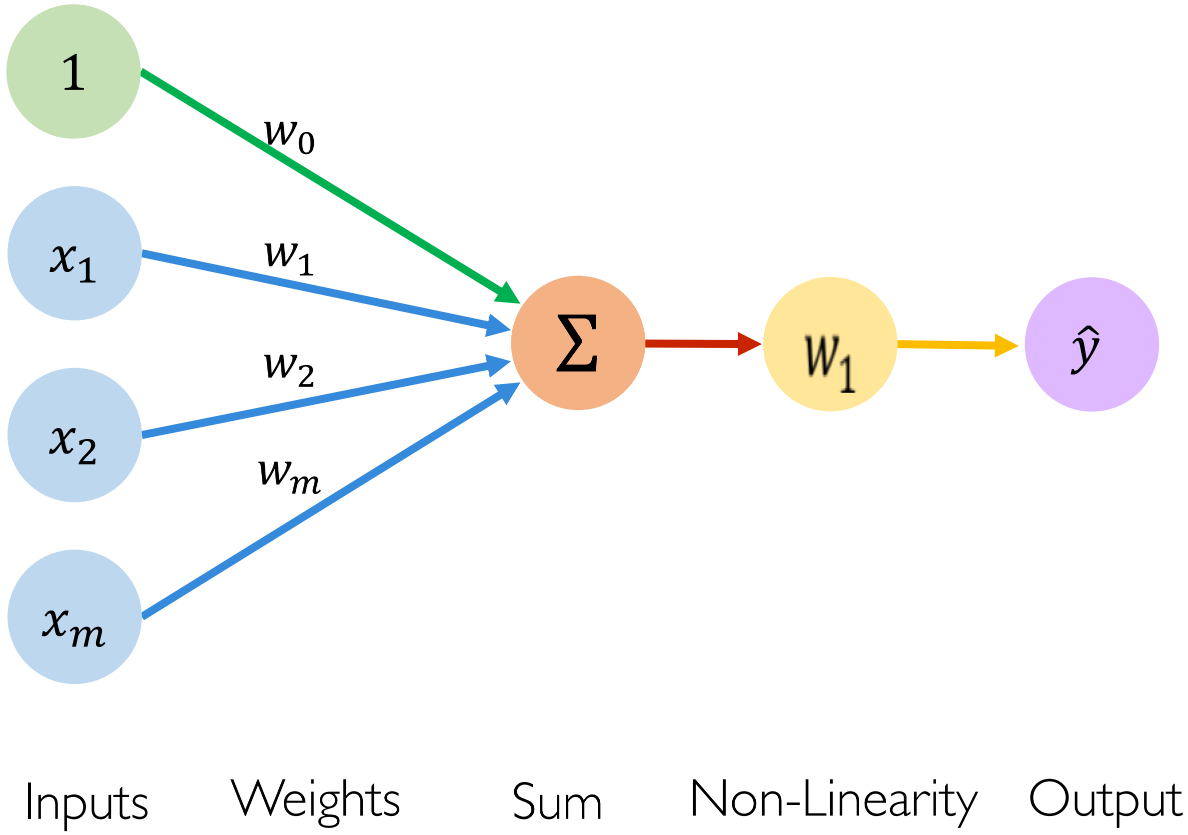

Output

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

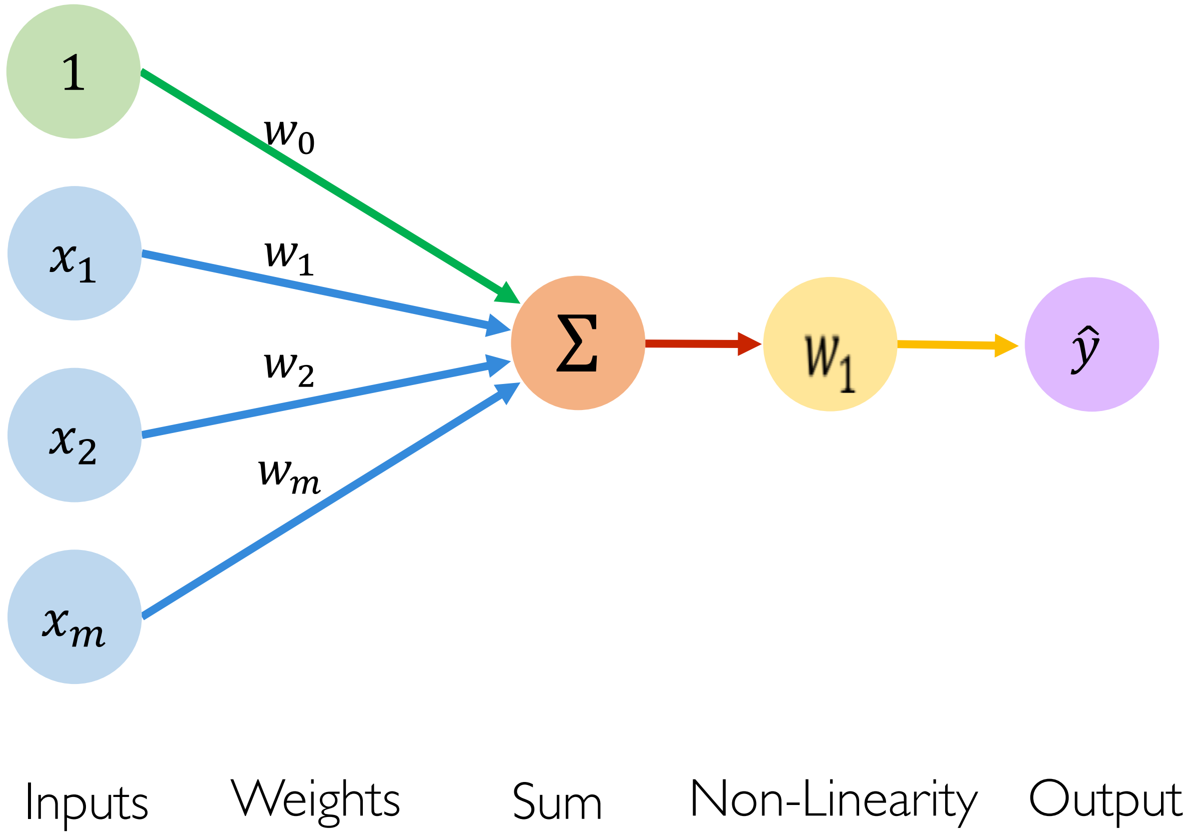
Linear combination of inputs

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

Non-linear activation function

Bias

# The Perceptron: Forward Propagation

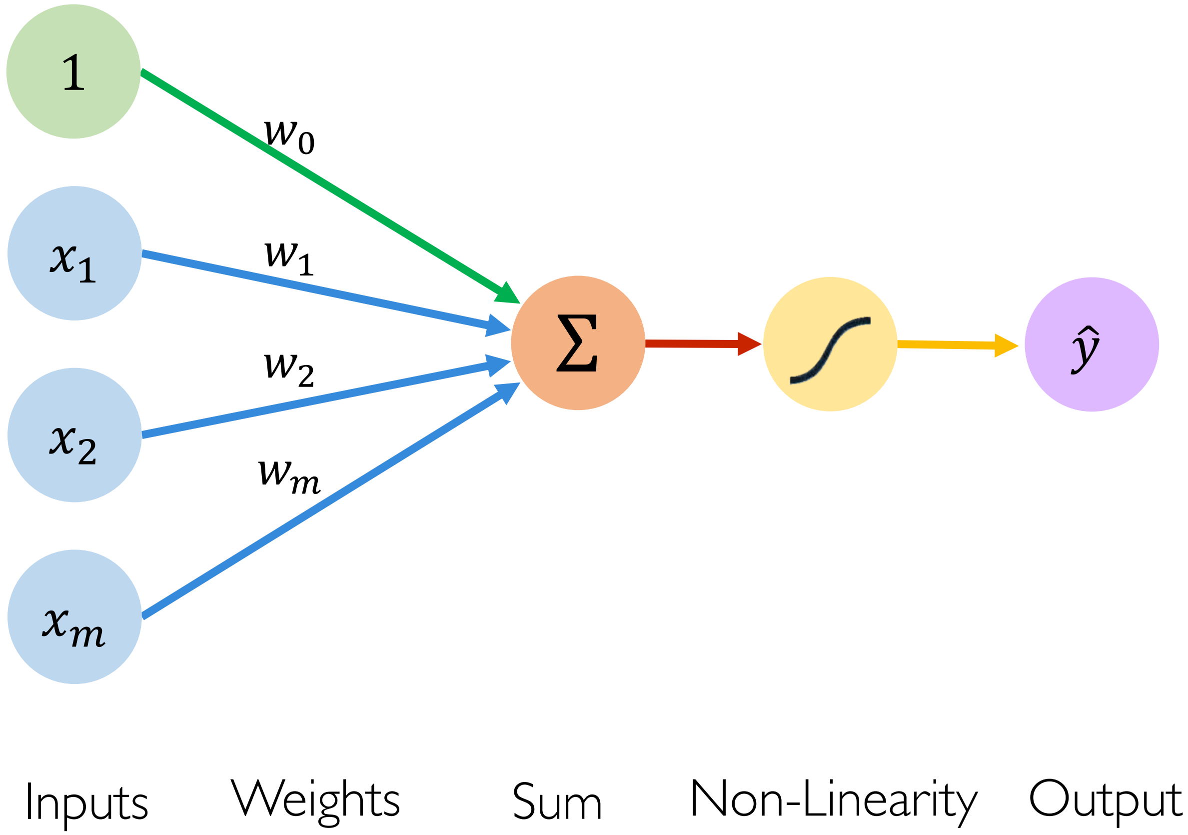



$$\hat{y} = g\left( w_0 + \sum_{i=1}^{m} x_i \, w_i \right)$$

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

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

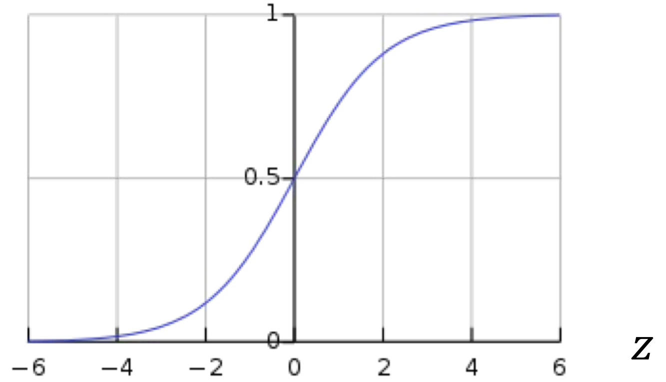



## Activation Functions

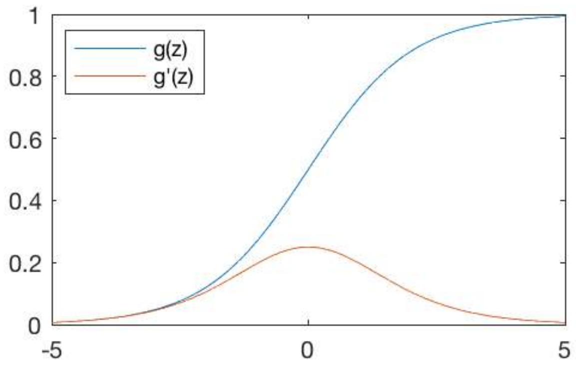
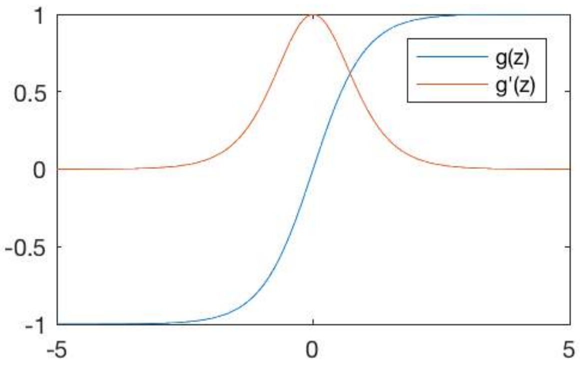
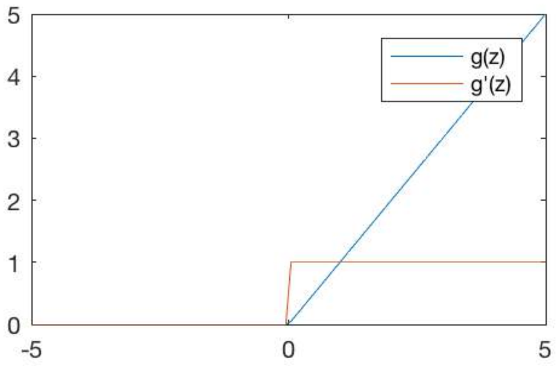
$$\hat{y} = g\left( w_0 + \boldsymbol{X}^T \boldsymbol{W} \right)$$

- Example: sigmoid function

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



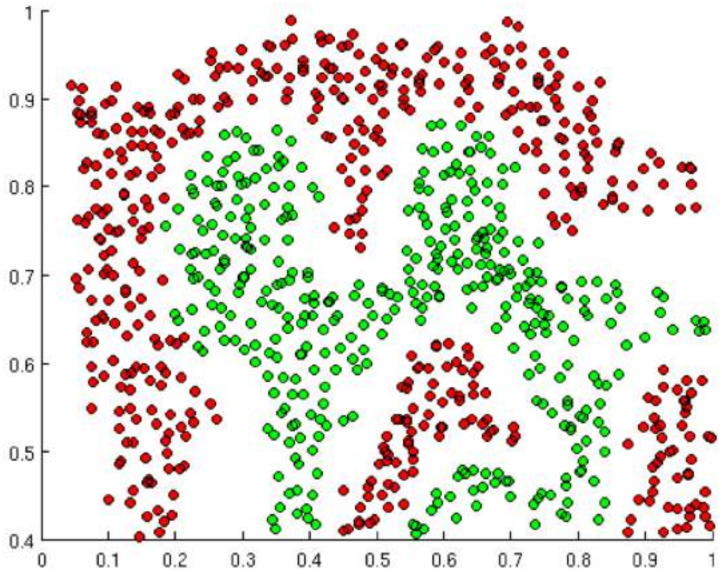
# Common Activation Functions

| Sigmoid Function | Hyperbolic Tangent | Rectified Linear Unit (ReLU) |
|---|---|---|
|  |  |  |
| $g(z) = \frac{1}{1 + e^{-z}}$ | $g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$ | $g(z) = \max(0, z)$ |
| $g'(z) = g(z)(1 - g(z))$ | $g'(z) = 1 - g(z)^2$ | $g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$ |
| `tf.nn.sigmoid(z)` | `tf.nn.tanh(z)` | `tf.nn.relu(z)` |

NOTE: All activation functions are non-linear

# Importance of Activation Functions

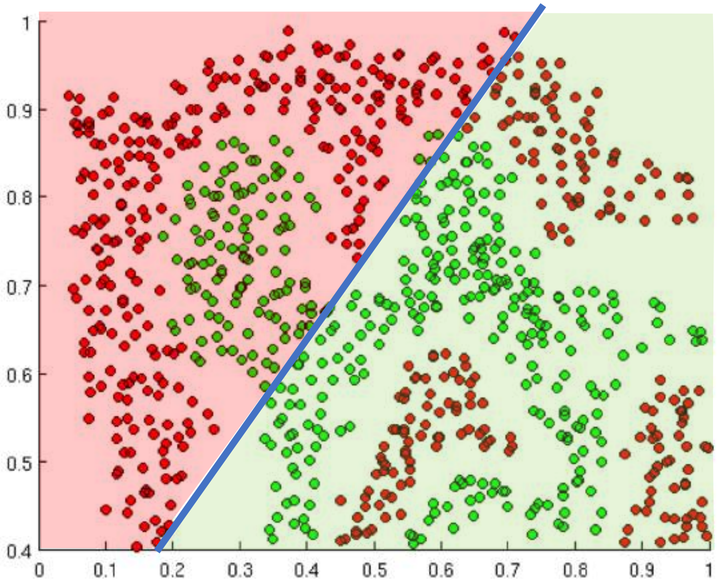
*The purpose of activation functions is to* ***introduce non-linearities*** *into the network*




What if we wanted to build a Neural Network to
distinguish green vs red points?

# Importance of Activation Functions

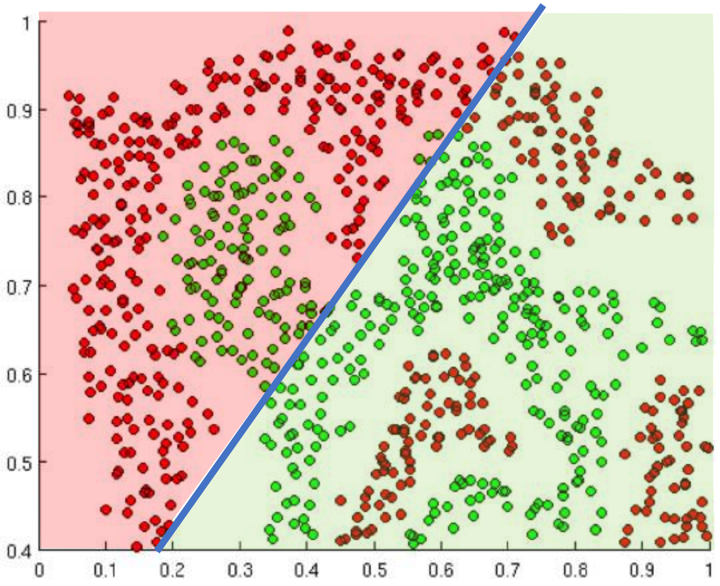
*The purpose of activation functions is to* ***introduce non-linearities*** *into the network*



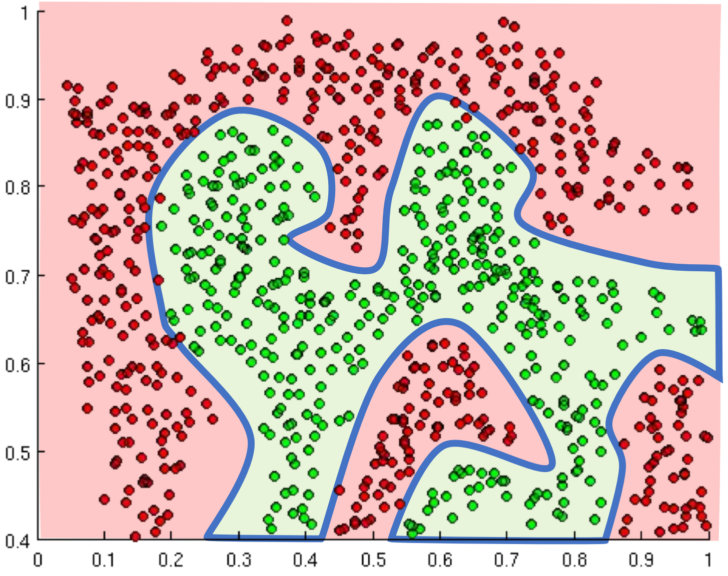

Linear Activation functions produce linear decisions no matter the network size

# Importance of Activation Functions

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

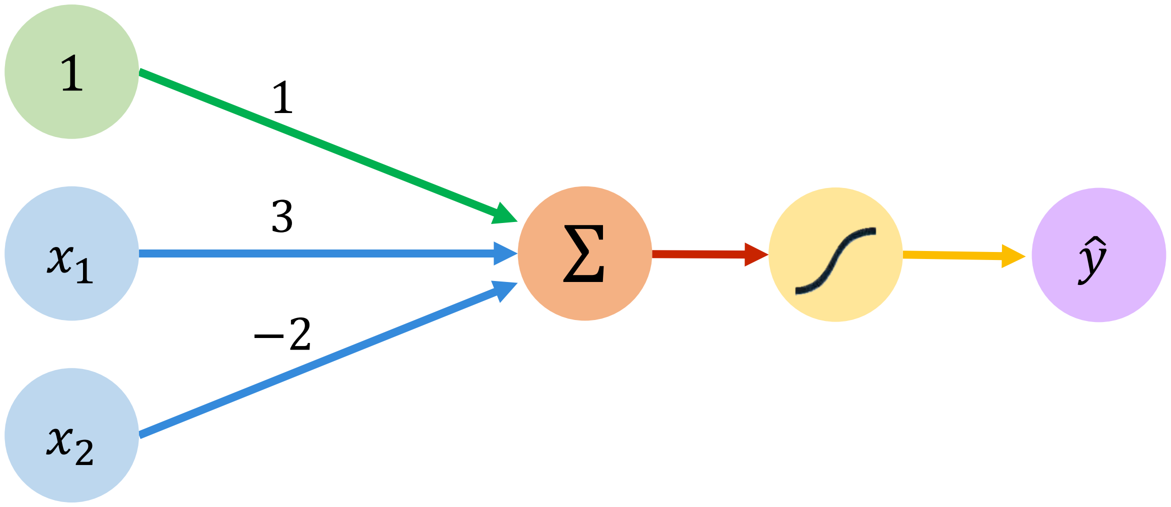



Linear Activation functions produce linear decisions no matter the network size




Non-linearities allow us to approximate arbitrarily complex functions

# The Perceptron: Example



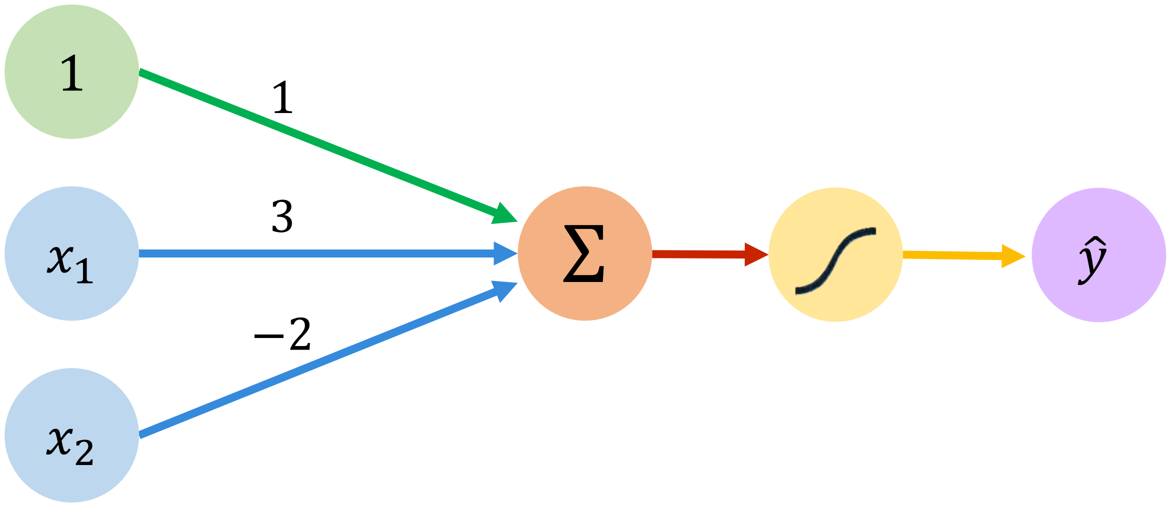

We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

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

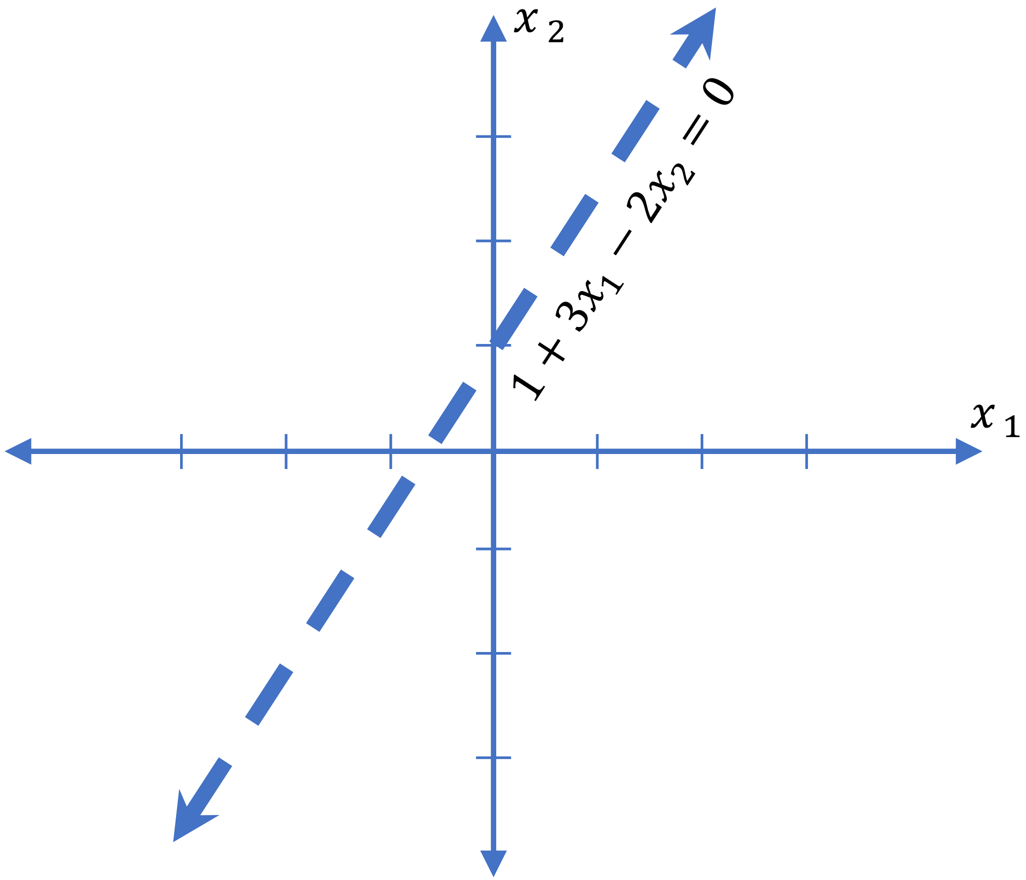
$$= g\left( 1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix} \right)$$

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

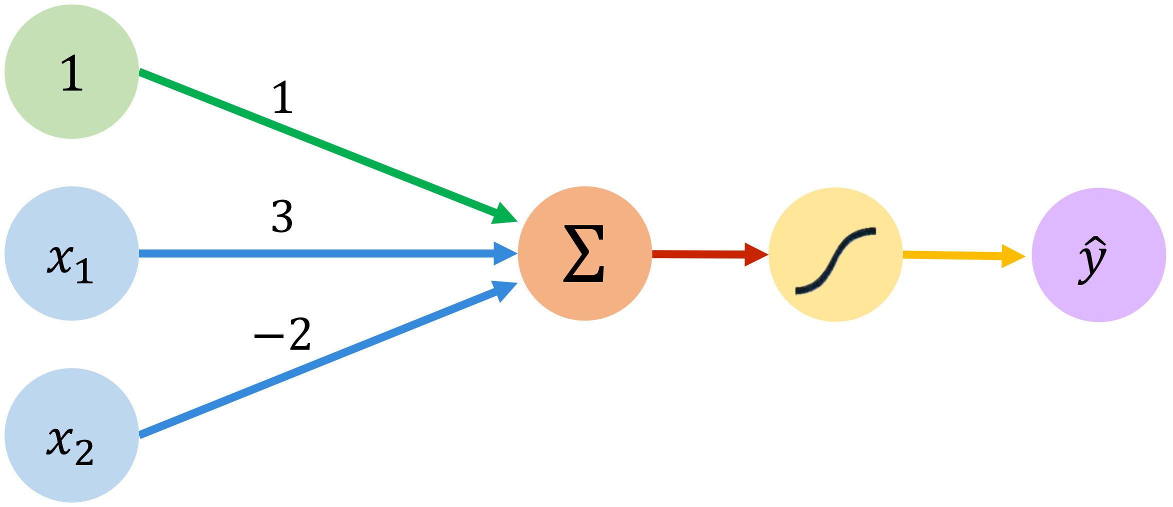
# The Perceptron: Example




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



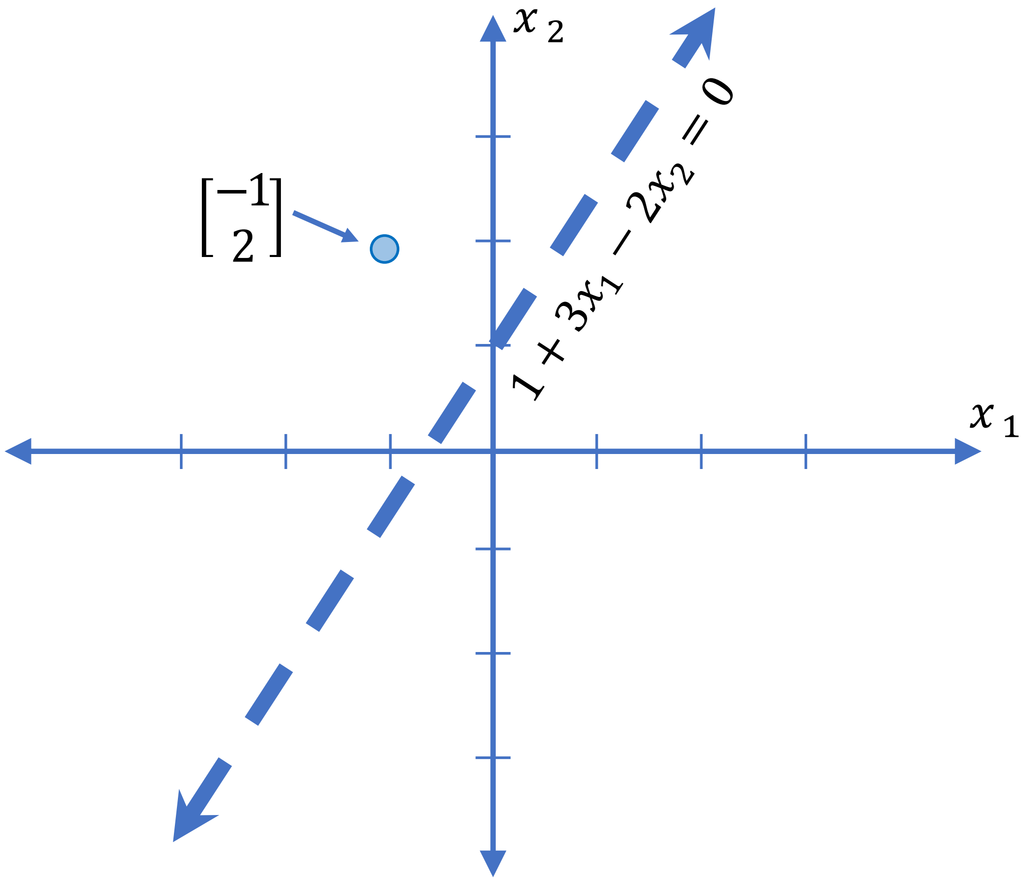
# The Perceptron: Example



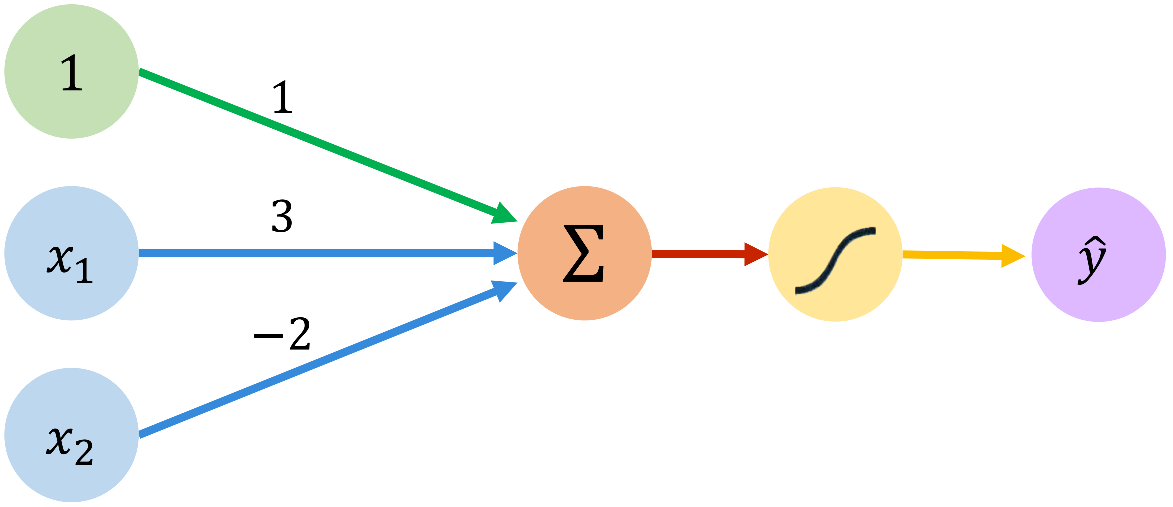

Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

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

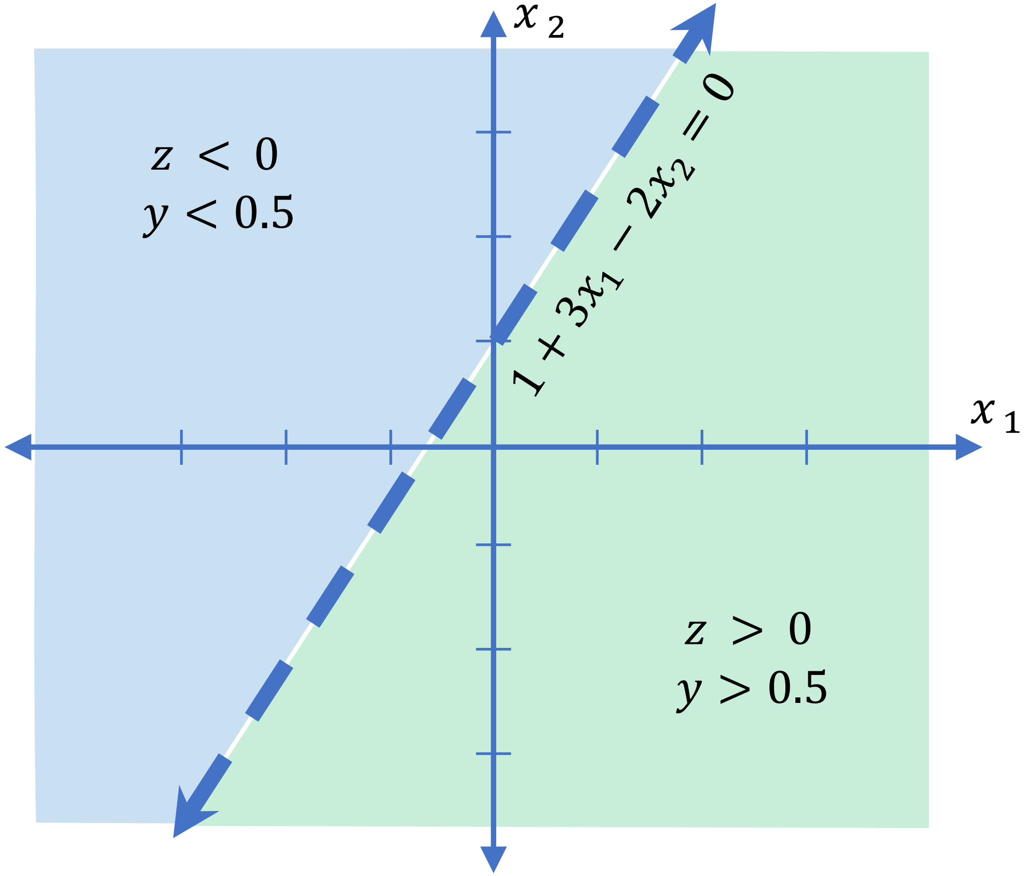
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



# The Perceptron: Example

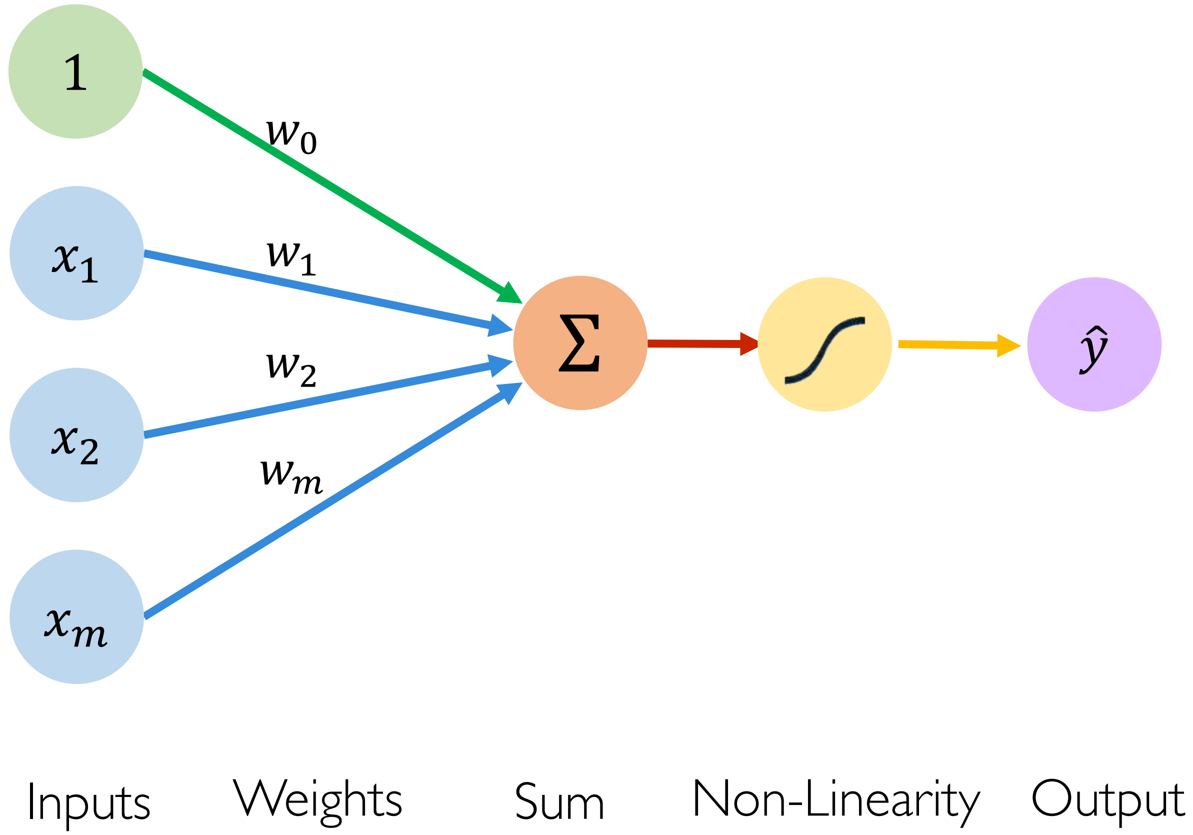



$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

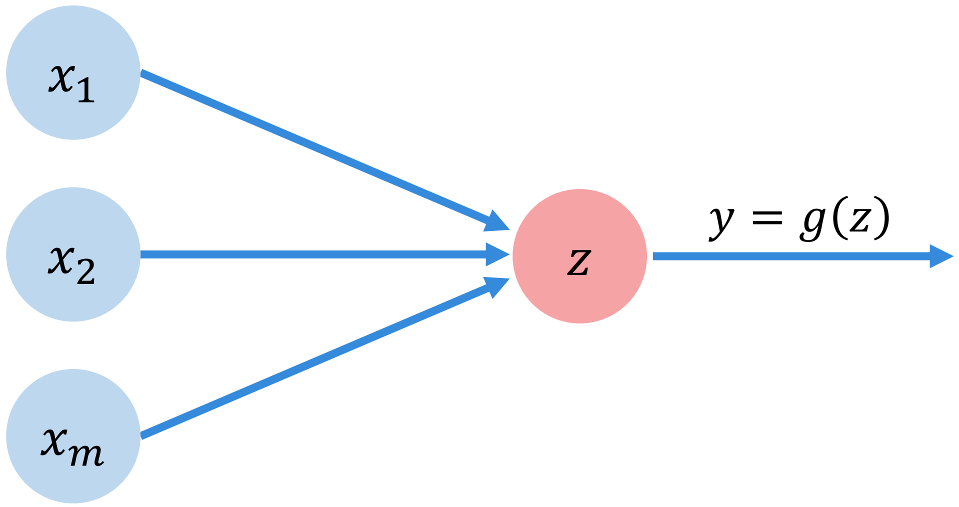


# Building Neural Networks with Perceptrons

# The Perceptron: Simplified

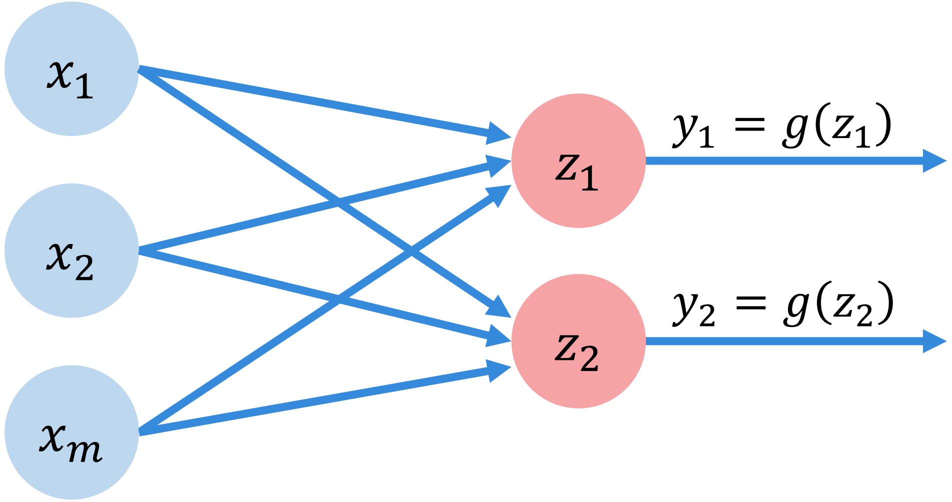


# The Perceptron: Simplified



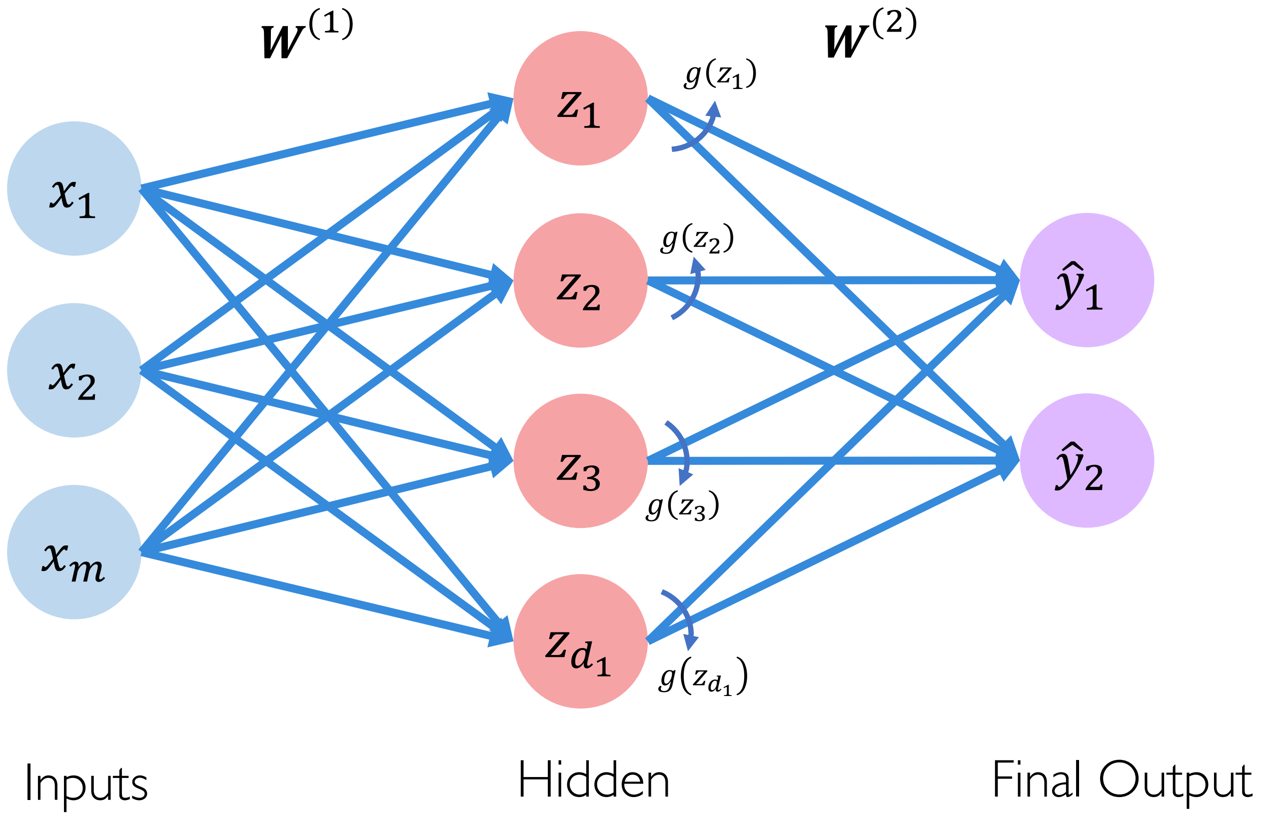

$$z = w_0 + \sum_{j=1}^{m} x_j \, w_j$$

# Multi Output Perceptron



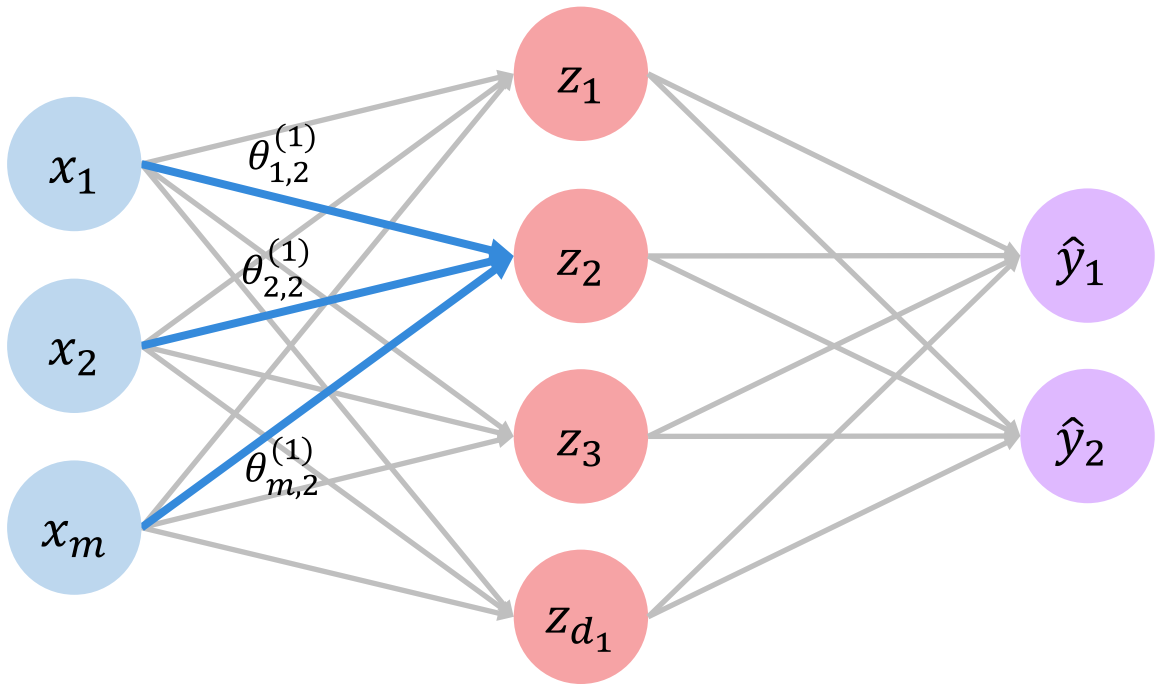

$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

# 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



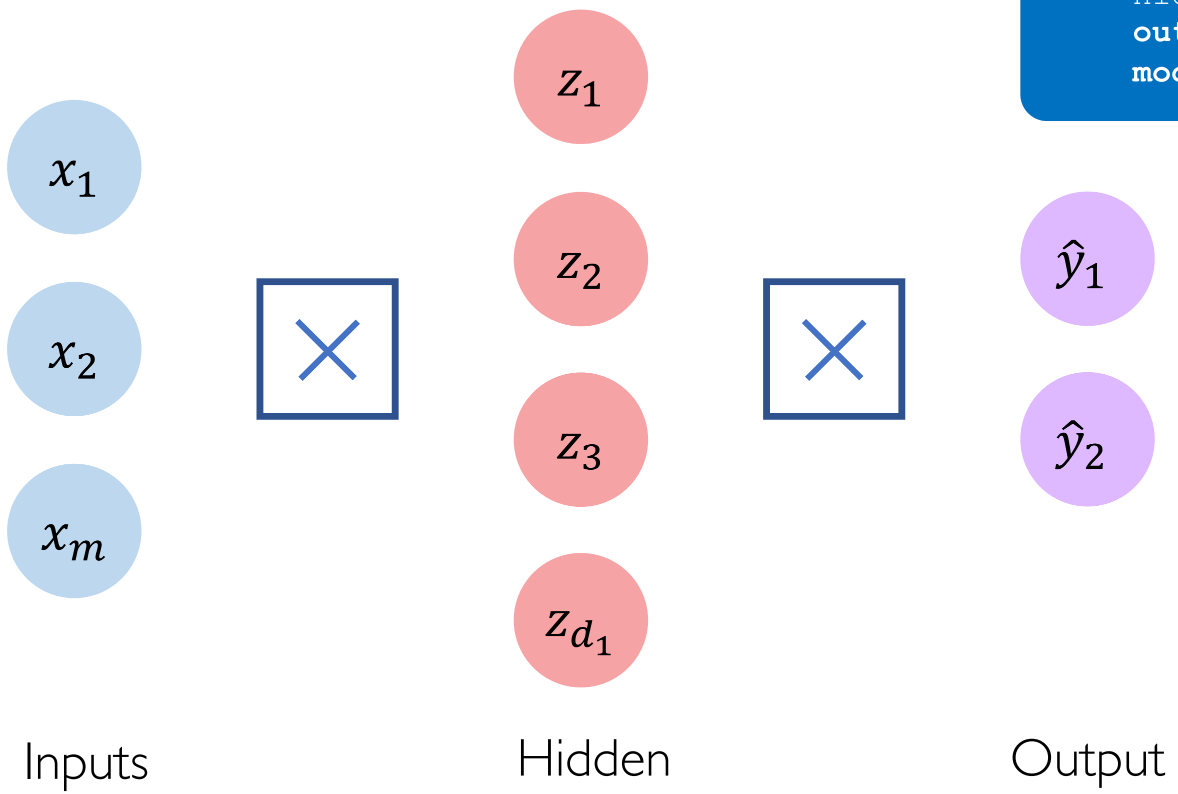

$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,2}^{(1)}$$

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

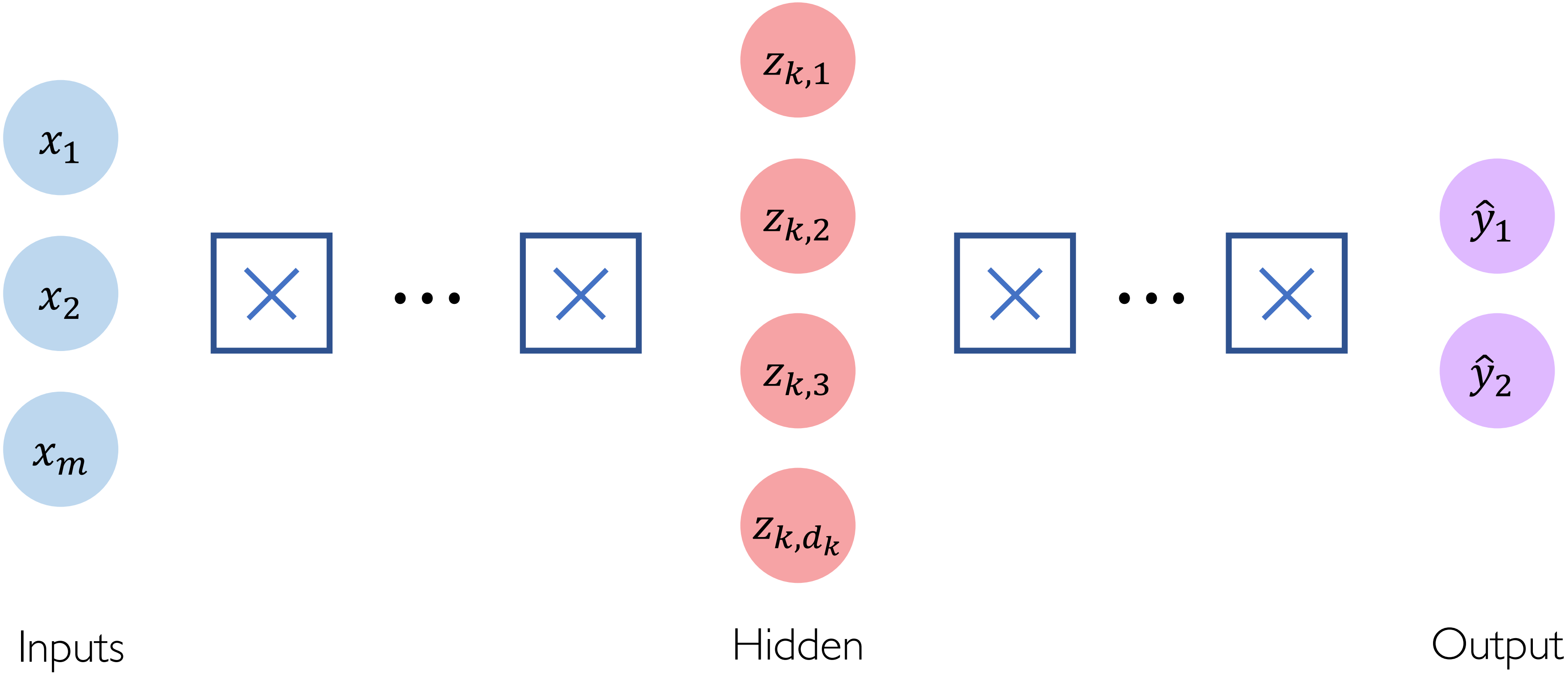
# Multi Output Perceptron

```
from tf.keras.layers import *

inputs = Inputs(m)
hidden = Dense(d1)(inputs)
outputs = Dense(2)(hidden)
model = Model(inputs, outputs)
```



# Deep Neural Network




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

# Applying Neural Networks

# Example Problem

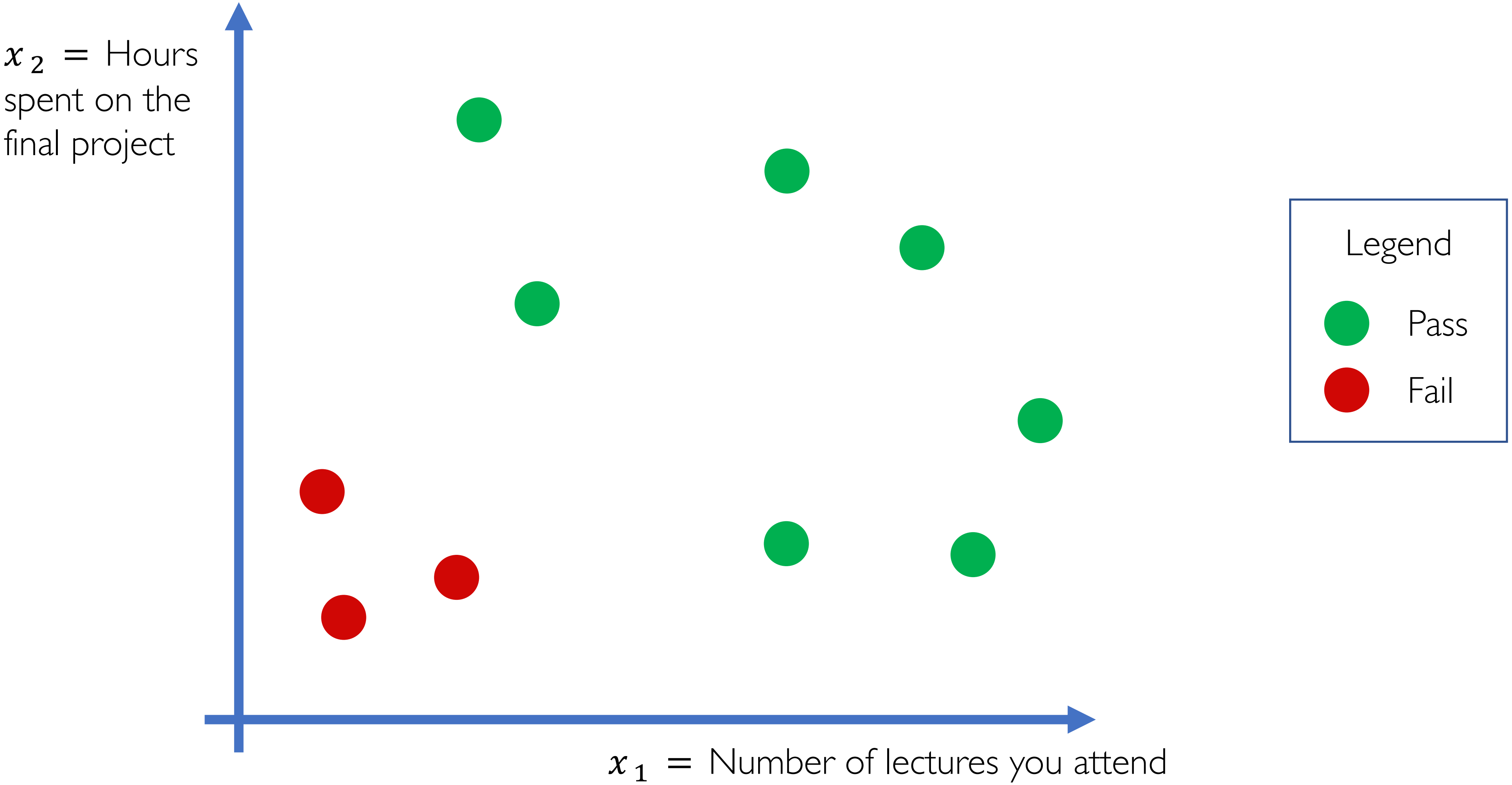
## Will I pass this class?

Let's start with a simple two feature model

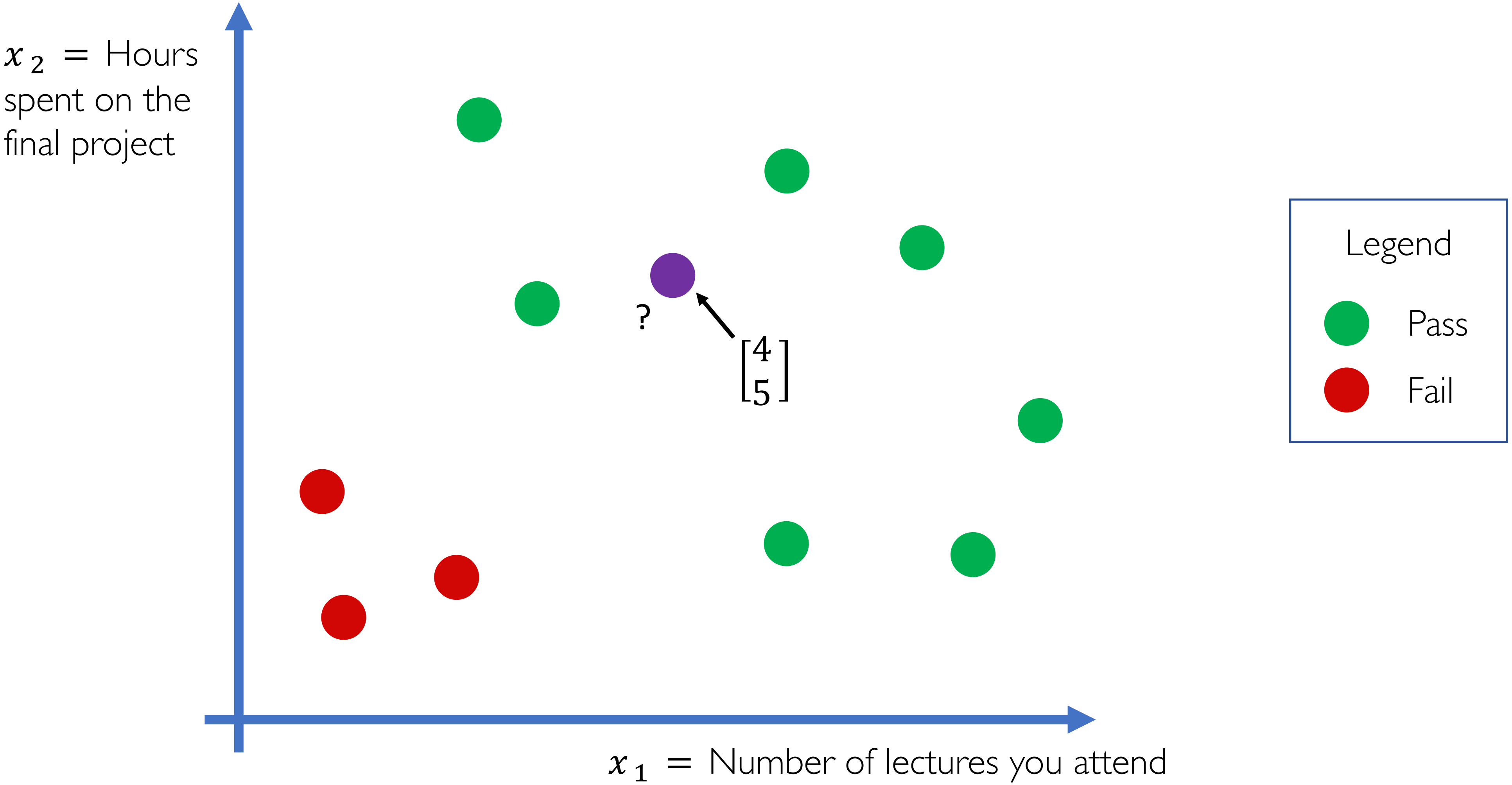
$x_1$ = Number of lectures you attend

$x_2$ = Hours spent on the final project

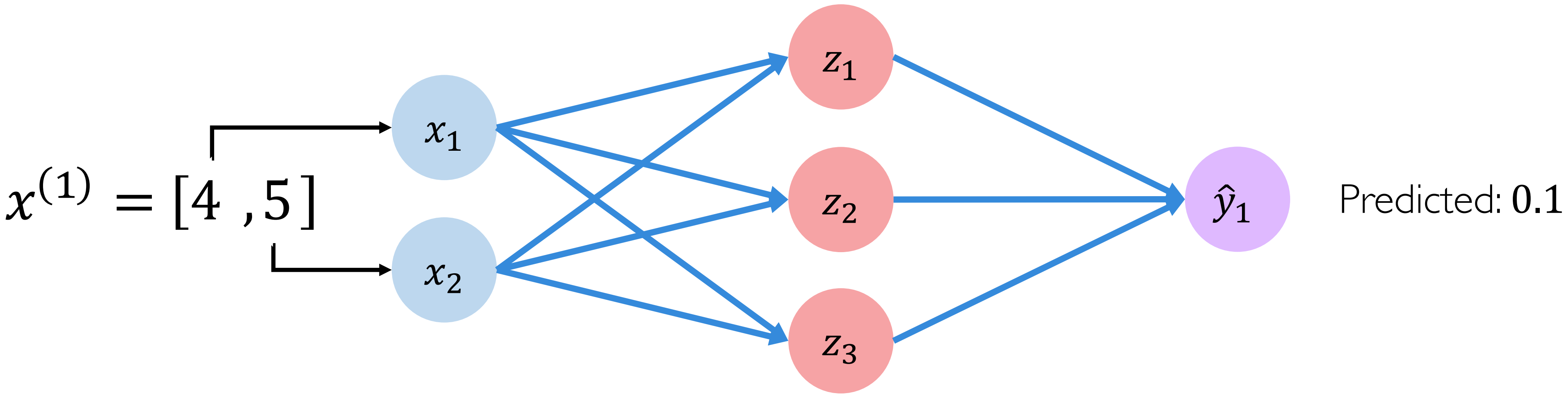
# Example Problem: Will I pass this class?



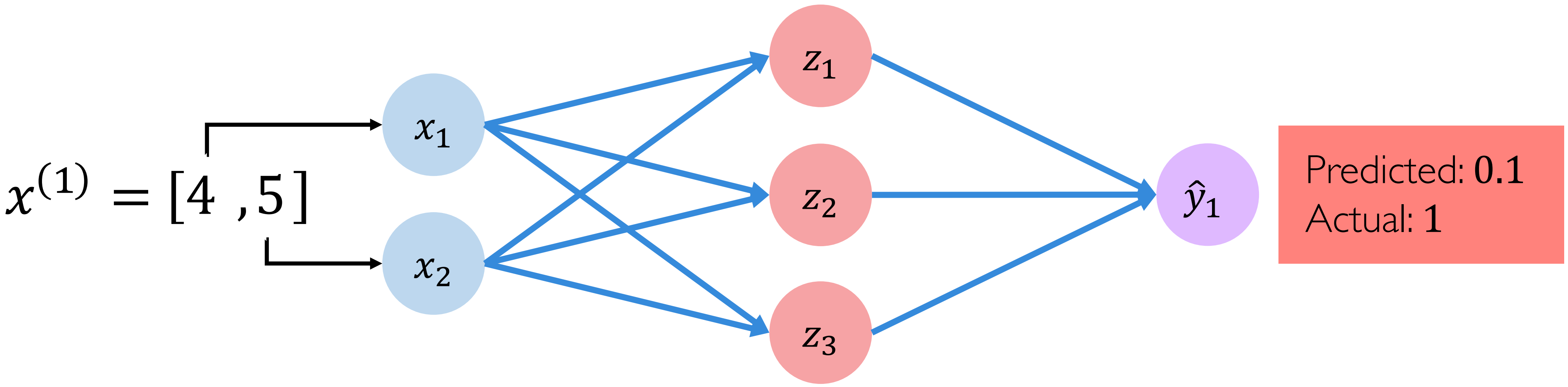
# Example Problem: Will I pass this class?



# Example Problem: Will I pass this class?

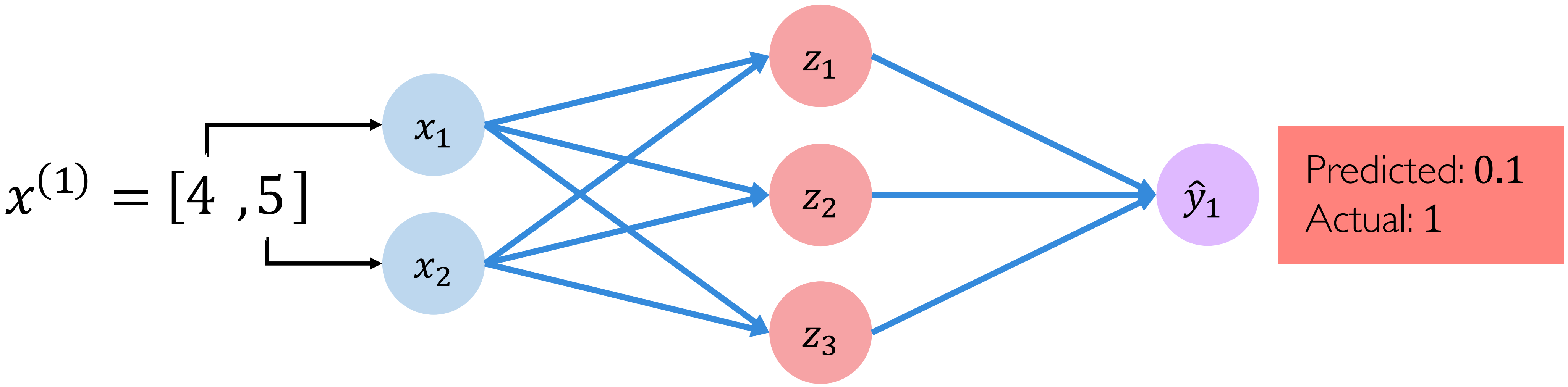


# Example Problem: Will I pass this class?



# Quantifying Loss

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

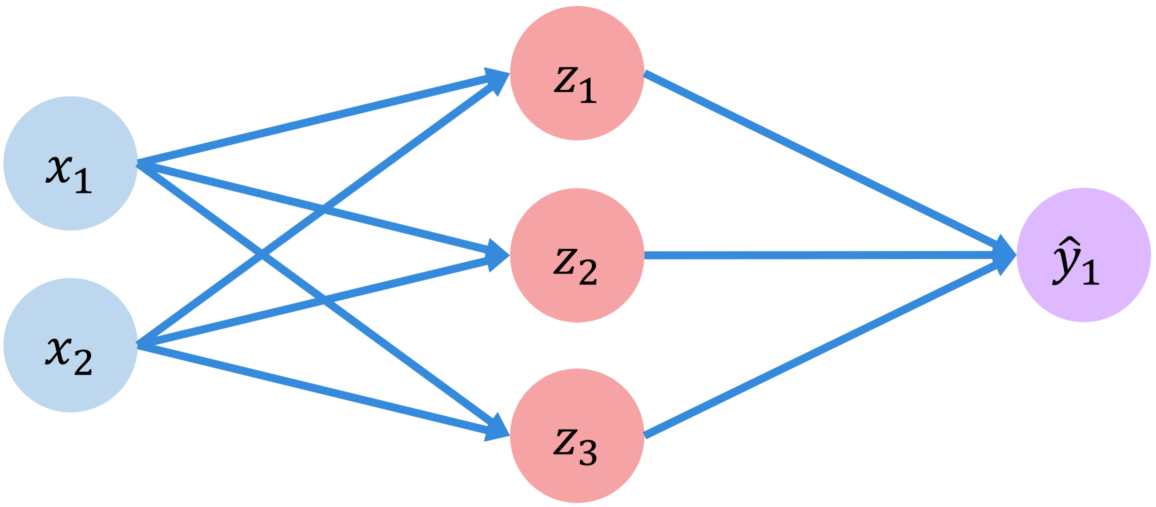



$$\mathcal{L}\left(\underbrace{f\left(x^{(i)}; \boldsymbol{W}\right)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\right)$$

# Empirical Loss

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

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




$$\begin{array}{cc} f(x) & y \\ \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} & \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix} \end{array}$$

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

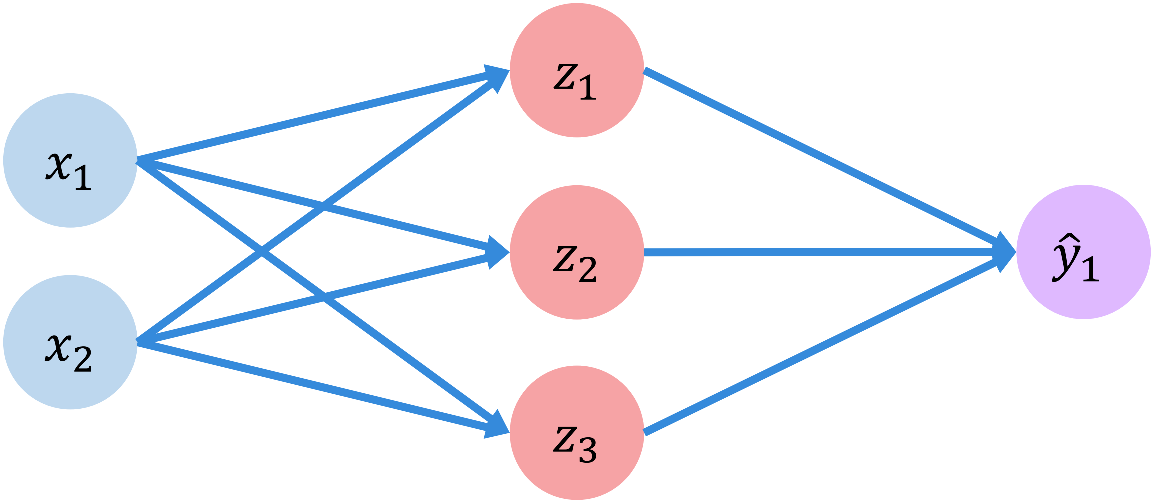
Also known as:

- 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*

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




$$f(x) = \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \qquad y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

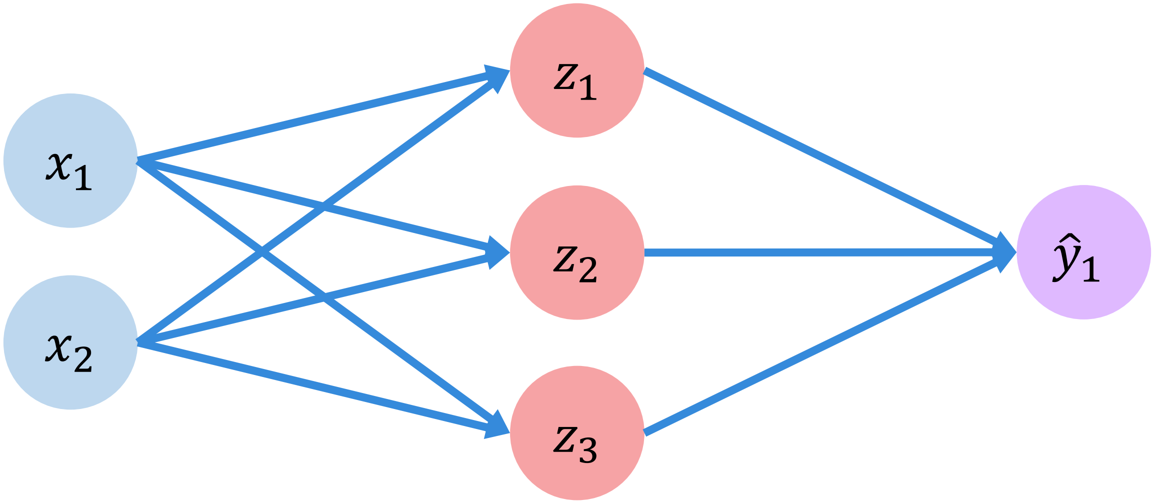
$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + \big(1 - \underbrace{y^{(i)}}_{\text{Actual}}\big) \log\Big(1 - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

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

# Mean Squared Error Loss

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

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




$$\underbrace{\underline{\begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix}}}_{f(x)} \quad \underline{\begin{bmatrix} 90 \\ 20 \\ 95 \\ \vdots \end{bmatrix}}_{y}$$

Final Grades (percentage)

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\Big(\underbrace{y^{(i)}}_{\text{Actual}} - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)^2$$

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

# 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***

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

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

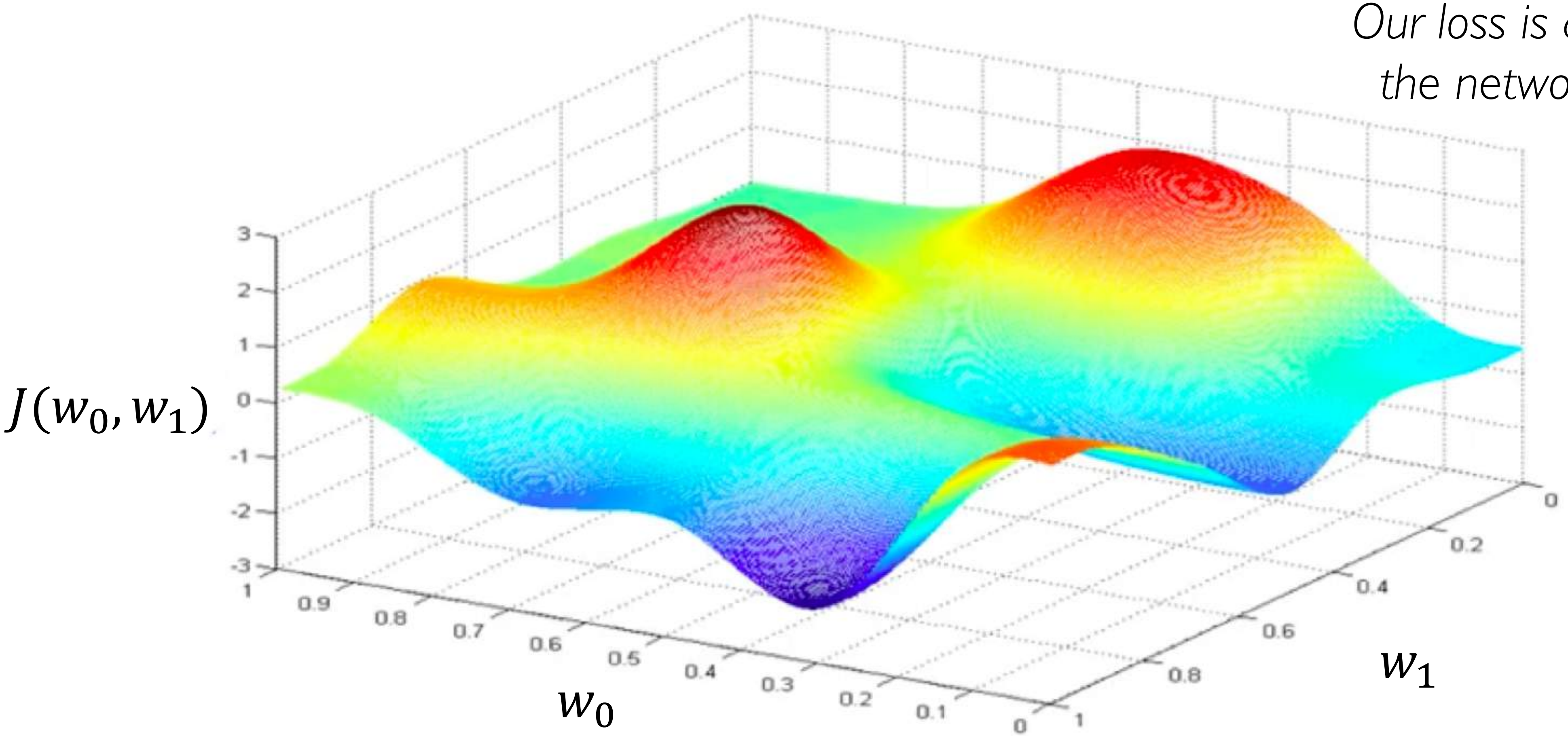
Remember:

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

# Loss Optimization

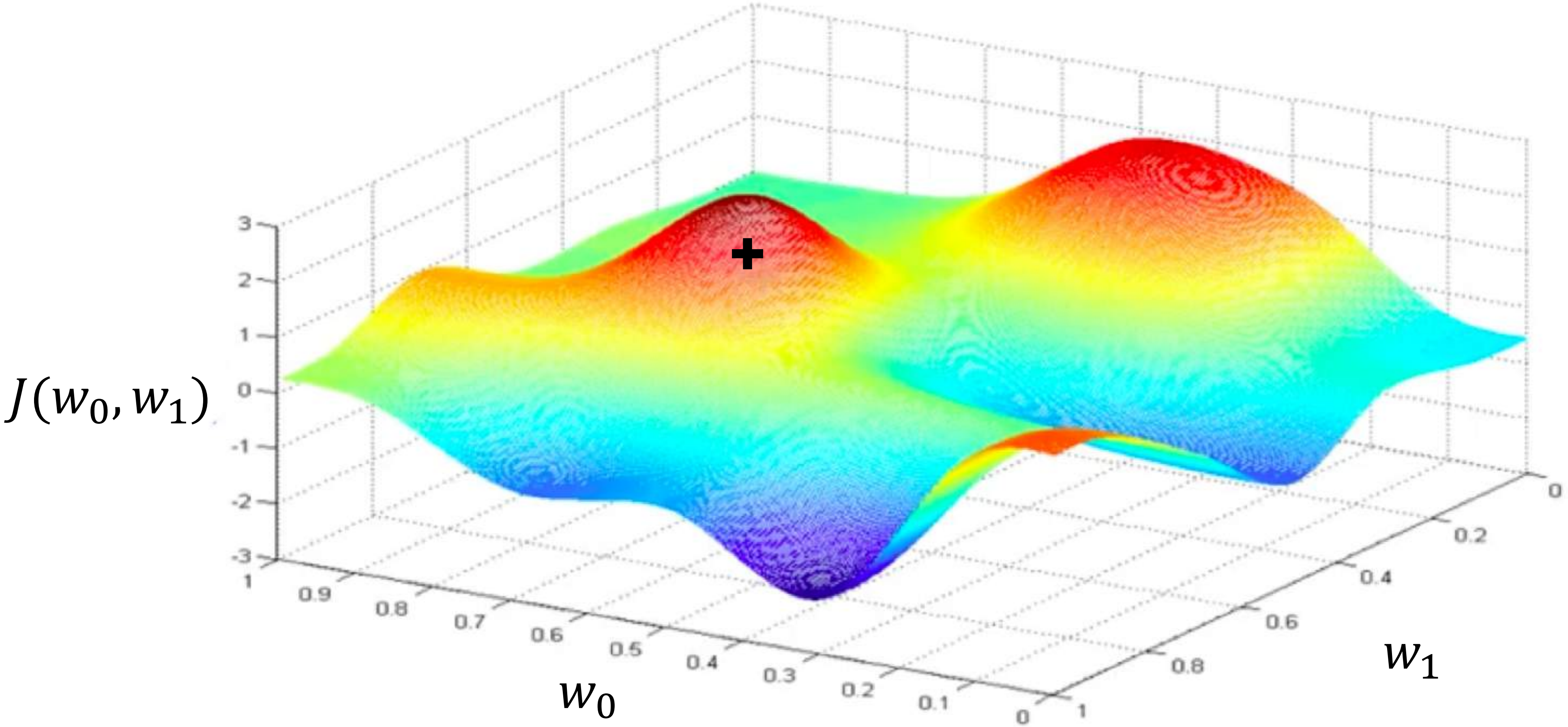
$$W^* = \underset{W}{\operatorname{argmin}} J(W)$$

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



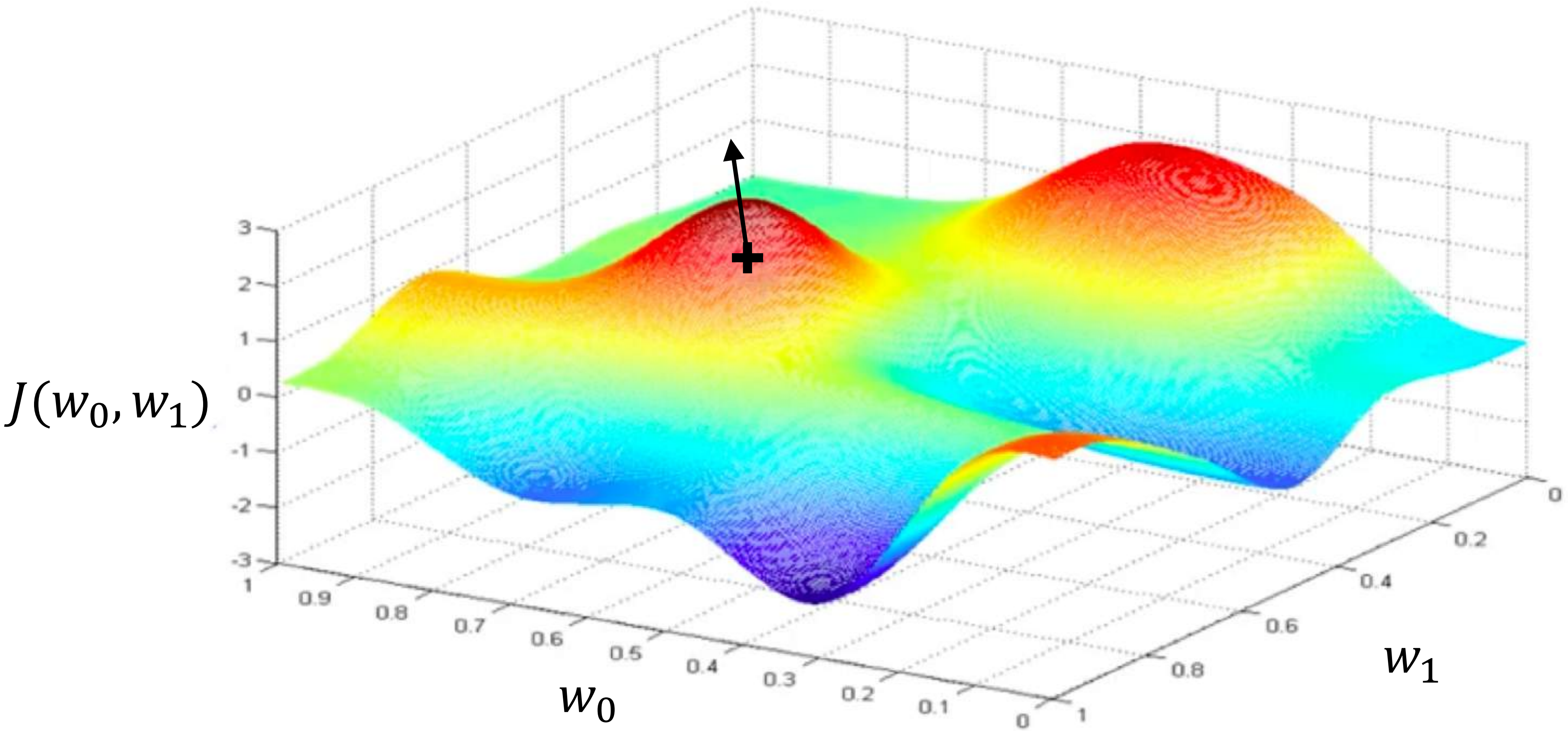
# Loss Optimization

Randomly pick an initial $(w_0, w_1)$



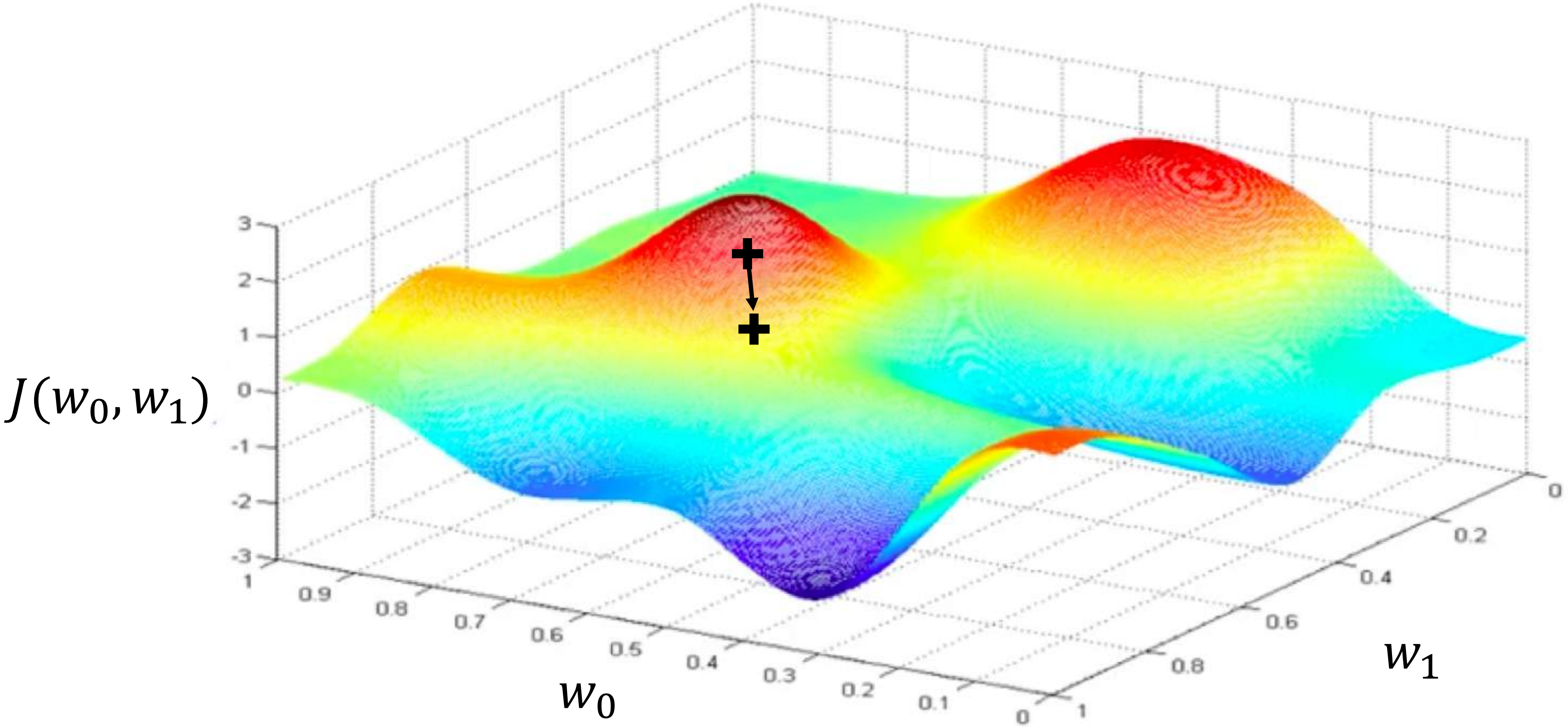
# Loss Optimization

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



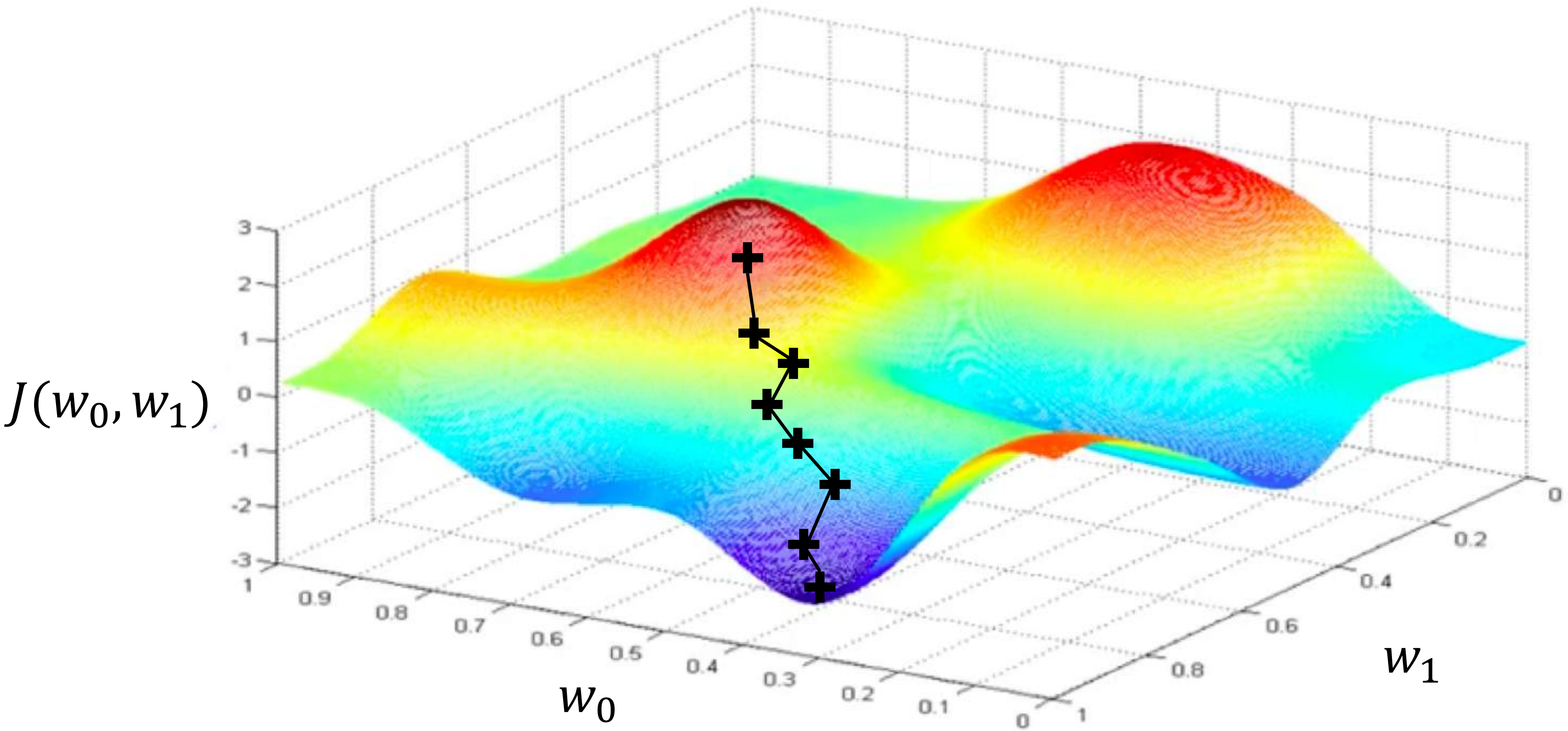
# Loss Optimization

Take small step in opposite direction of gradient



# Gradient Descent

Repeat until convergence



# Gradient Descent

**Algorithm**

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$

```
weights = tf.random_normal(shape, stddev=sigma)
```

2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

```
grads = tf.gradients(ys=loss, xs=weights)
```

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

```
weights_new = weights.assign(weights - lr * grads)
```

5. Return weights

# Gradient Descent

**Algorithm**

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$

```
weights = tf.random_normal(shape, stddev=sigma)
```

2. Loop until convergence:
3. Compute gradient, $\boxed{\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}}$

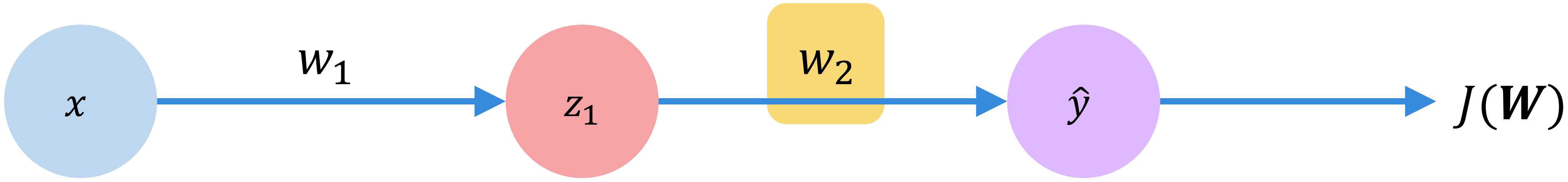
```
grads = tf.gradients(ys=loss, xs=weights)
```

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

```
weights_new = weights.assign(weights - lr * grads)
```

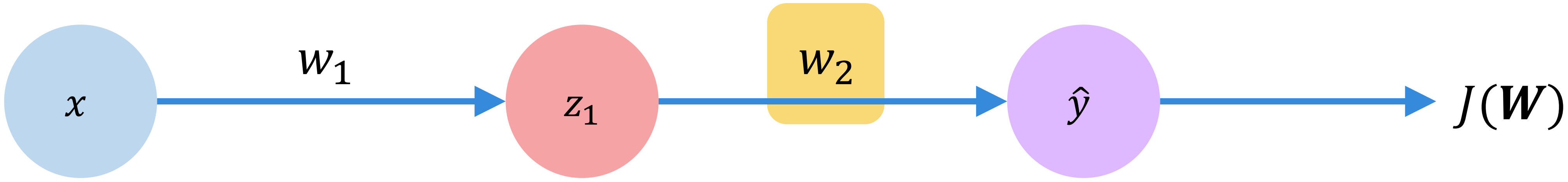
5. Return weights

# Computing Gradients: Backpropagation




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

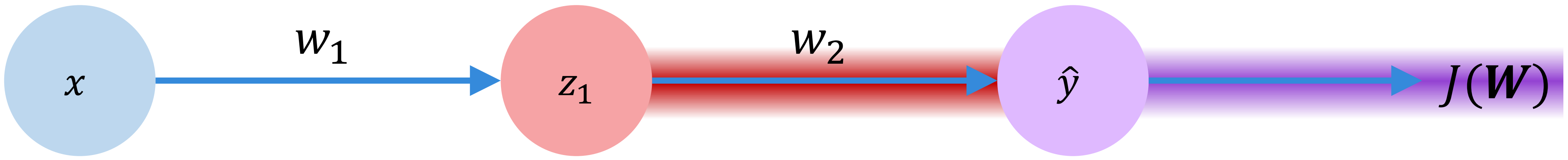
# Computing Gradients: Backpropagation




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

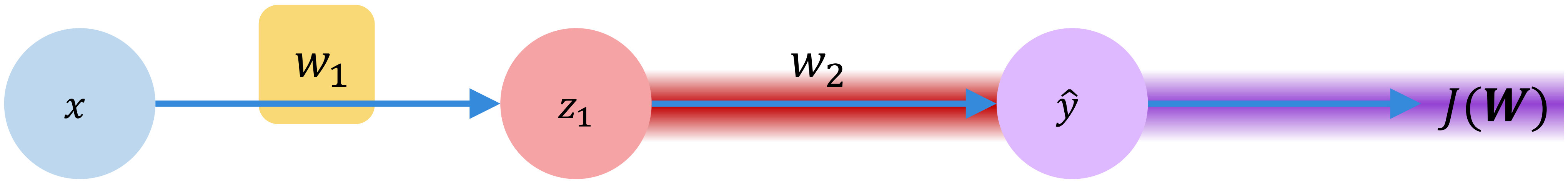
Let's use the chain rule!

# Computing Gradients: Backpropagation




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

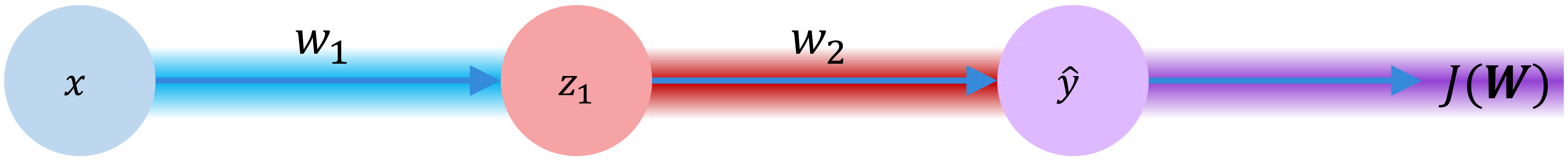
# Computing Gradients: Backpropagation




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

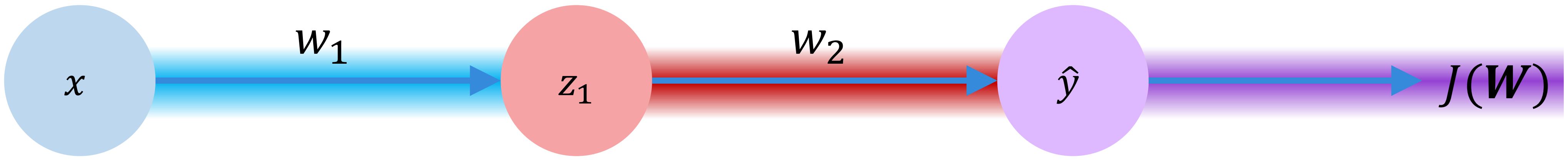
Apply chain rule! Apply chain rule!

# 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

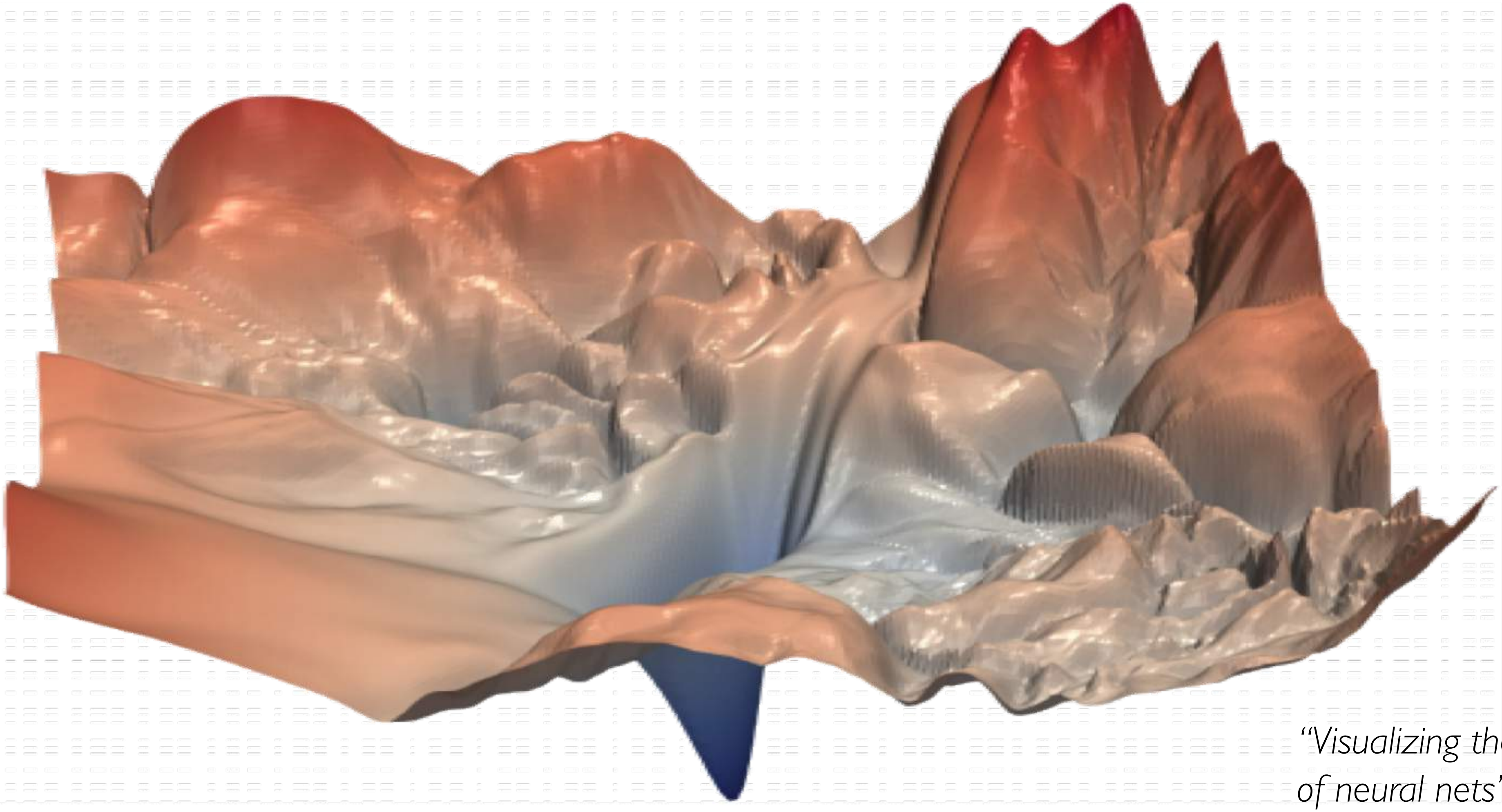



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \underbrace{\frac{\partial J(\boldsymbol{W})}{\partial \hat{y}}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}} * \underbrace{\frac{\partial z_1}{\partial w_1}}$$

*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:**

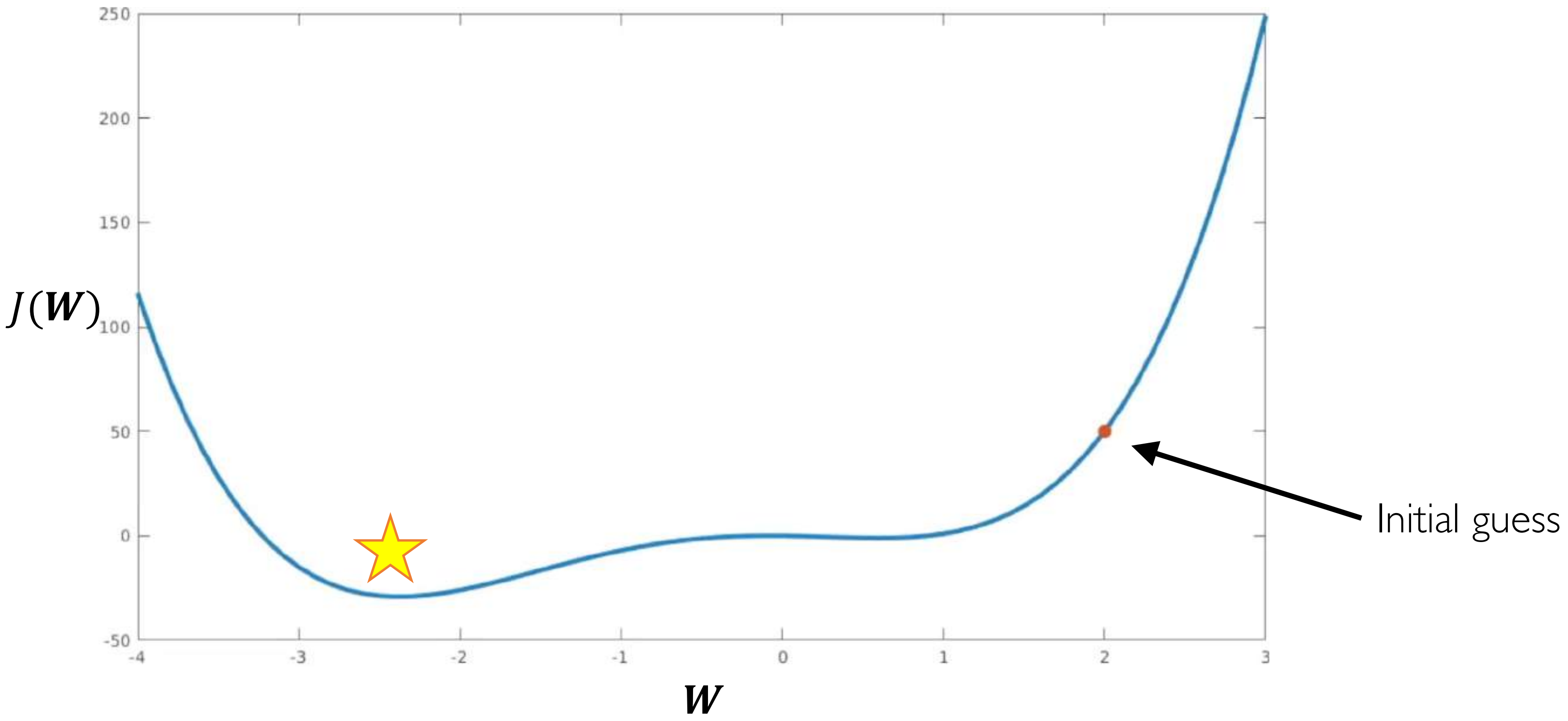
Optimization through gradient descent

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

How can we set the learning rate?

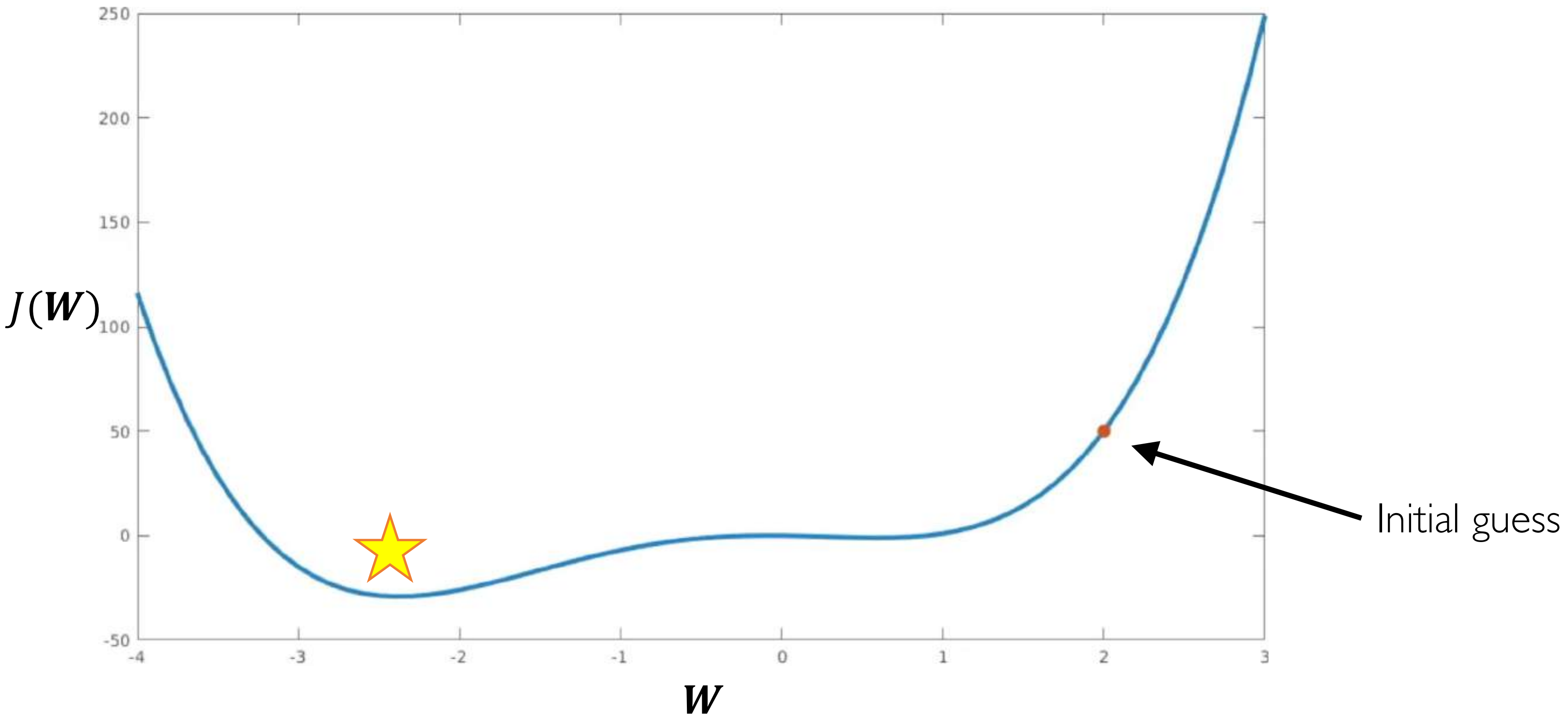
# Setting the Learning Rate

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



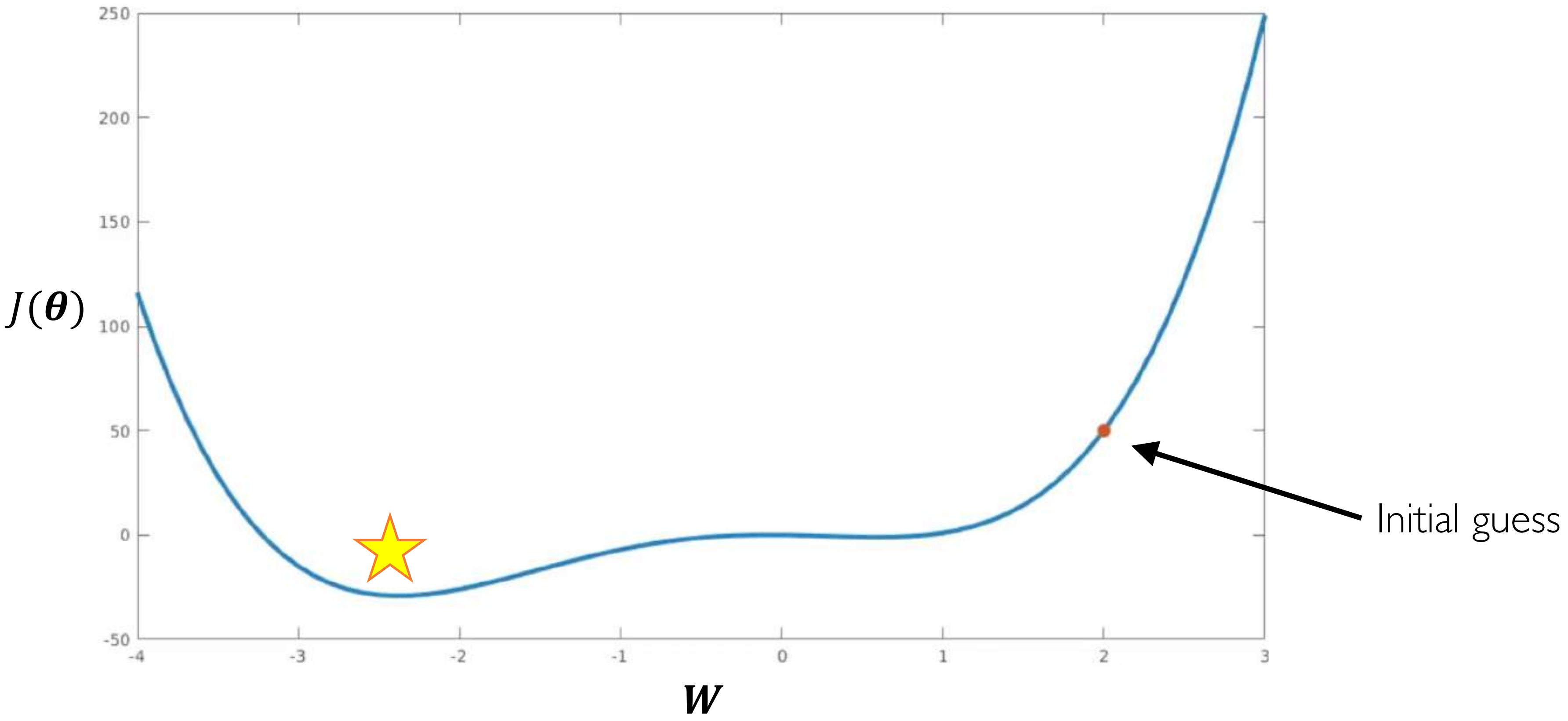
# Setting the Learning Rate

***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...

# Adaptive Learning Rate Algorithms

- Momentum
- Adagrad
- Adadelta
- Adam
- RMSProp

`tf.train.MomentumOptimizer`

`tf.train.AdagradOptimizer`

`tf.train.AdadeltaOptimizer`

`tf.train.AdamOptimizer`

`tf.train.RMSPropOptimizer`


Qian et al. "On the momentum term in gradient descent learning algorithms." 1999.

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

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

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

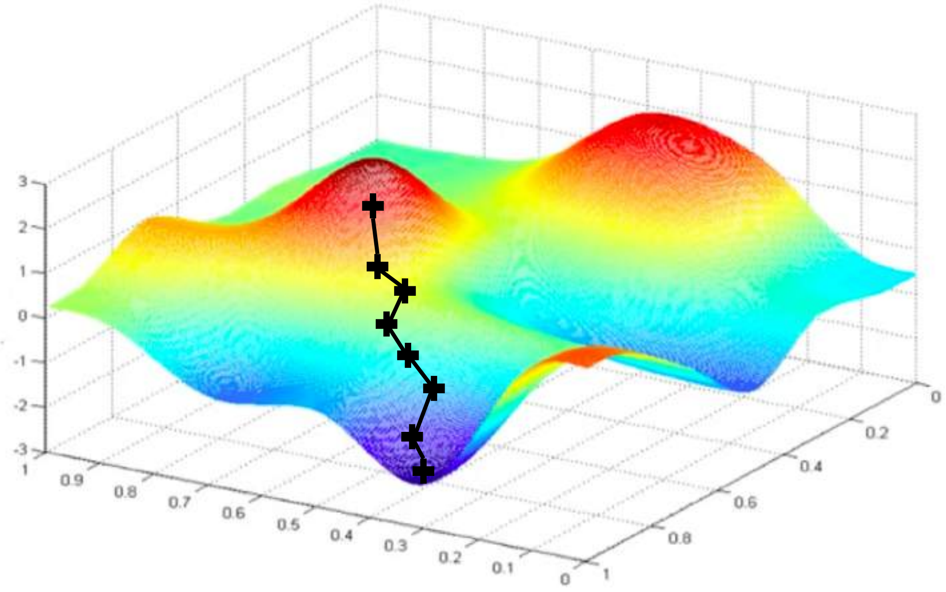

Additional details: http://ruder.io/optimizing-gradient-descent/

# 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

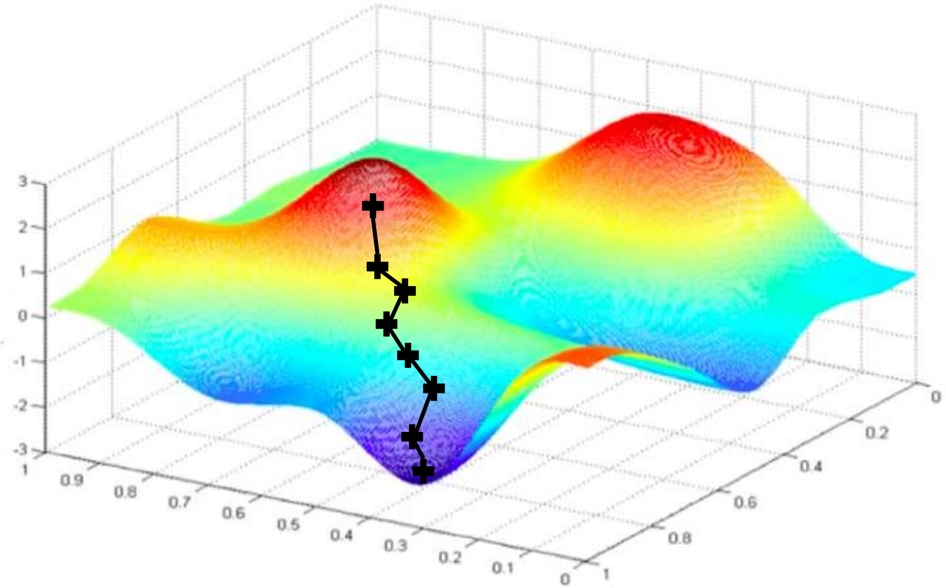


# Gradient Descent

**Algorithm**

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

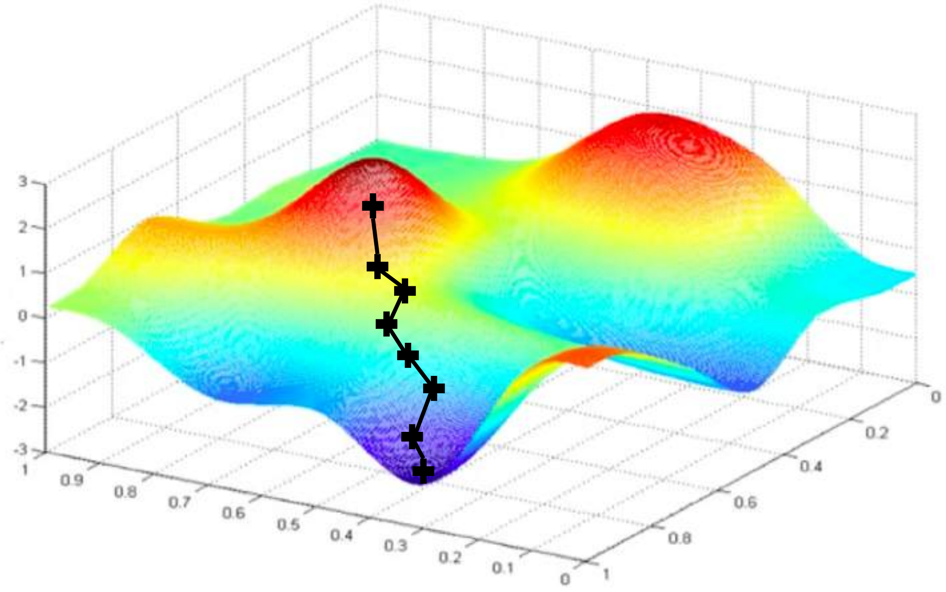
Can be very computational to compute!



# Stochastic Gradient Descent

**Algorithm**

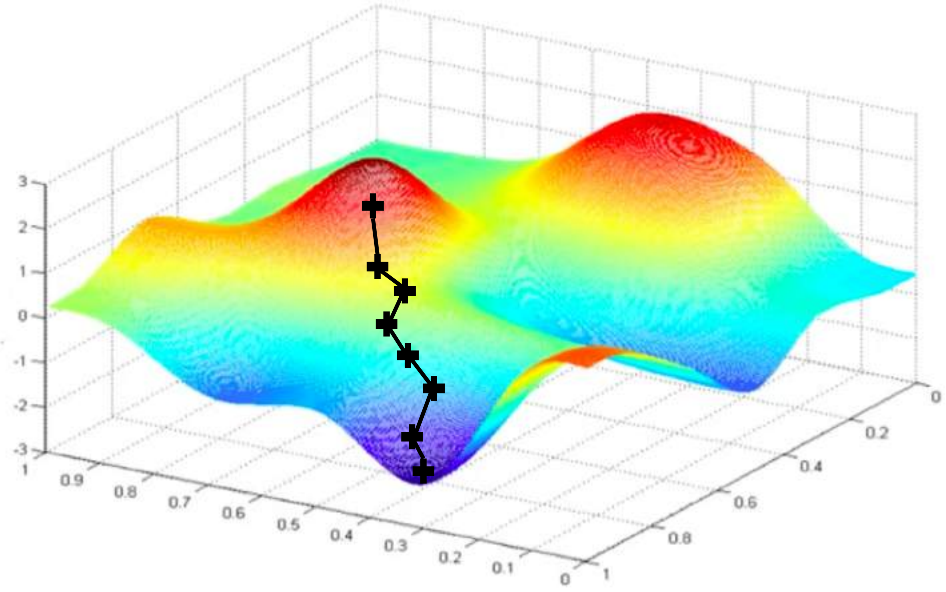
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point $\boldsymbol{i}$
4. Compute gradient, $\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial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

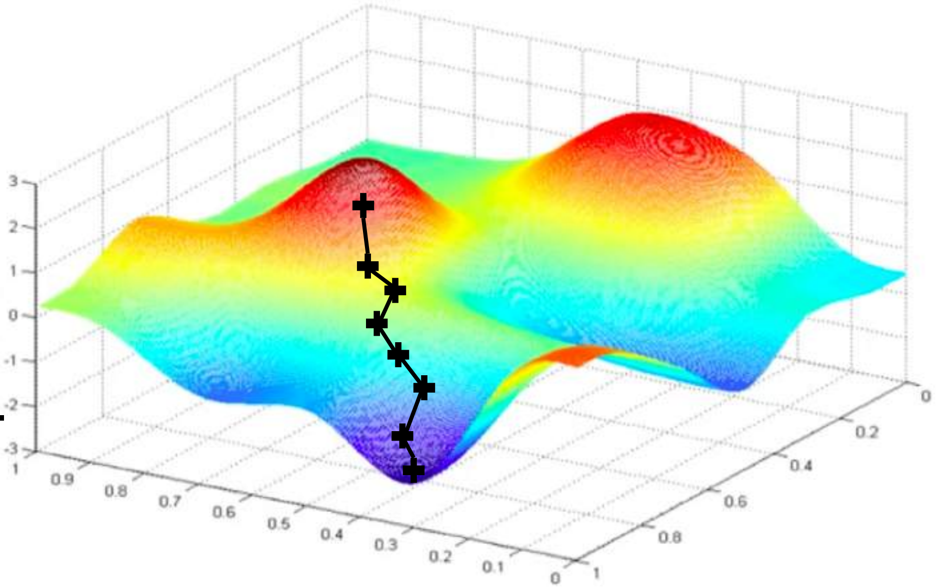


Easy to compute but
**very noisy**
(stochastic)!

# Stochastic Gradient Descent

**Algorithm**

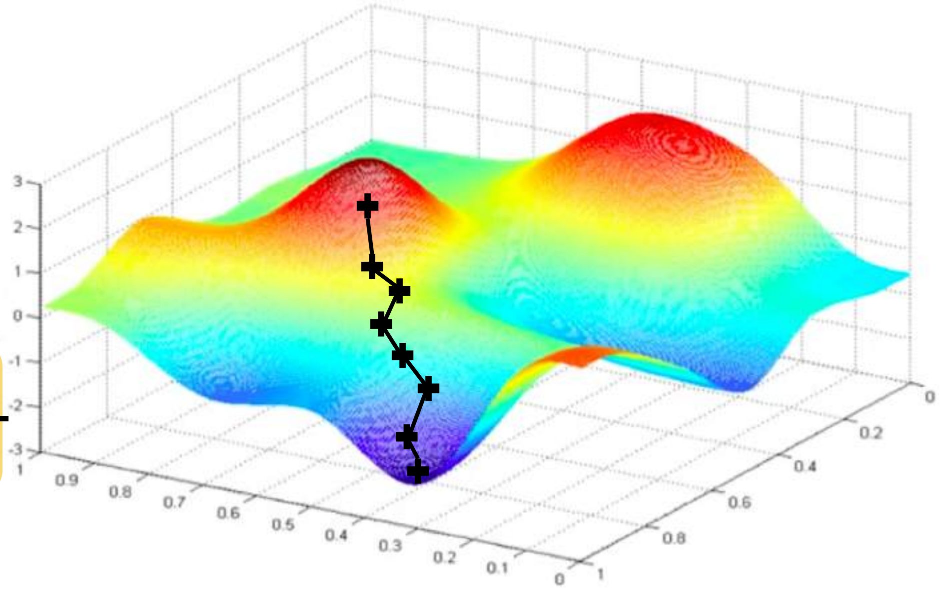
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of $B$ data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B}\frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update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

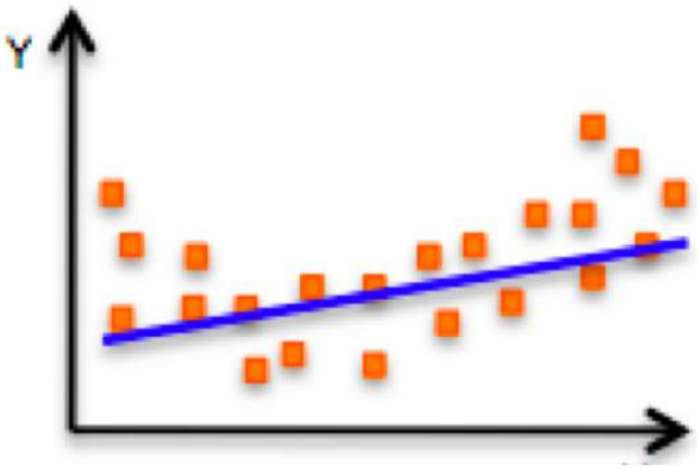
Allows for larger learning rates

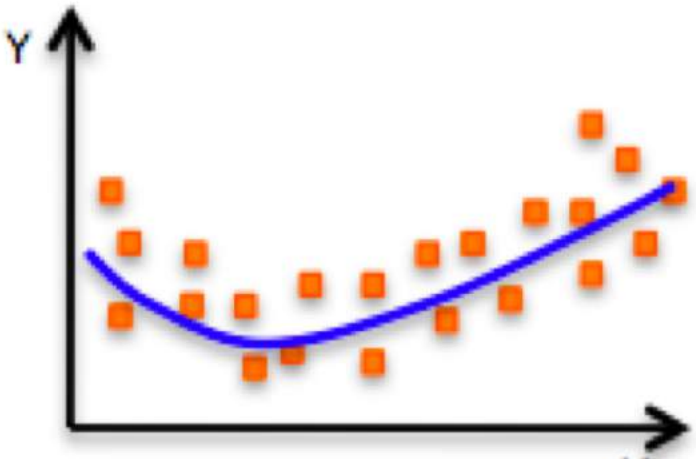
**Mini-batches lead to fast training!**

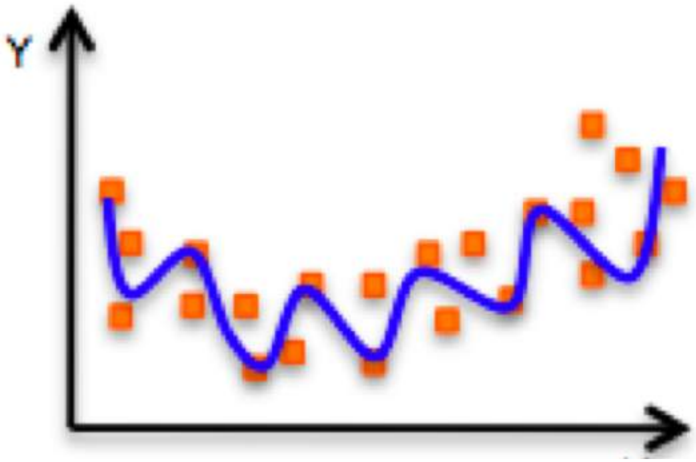
Can parallelize computation + achieve significant speed increases on GPU's

# 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?*

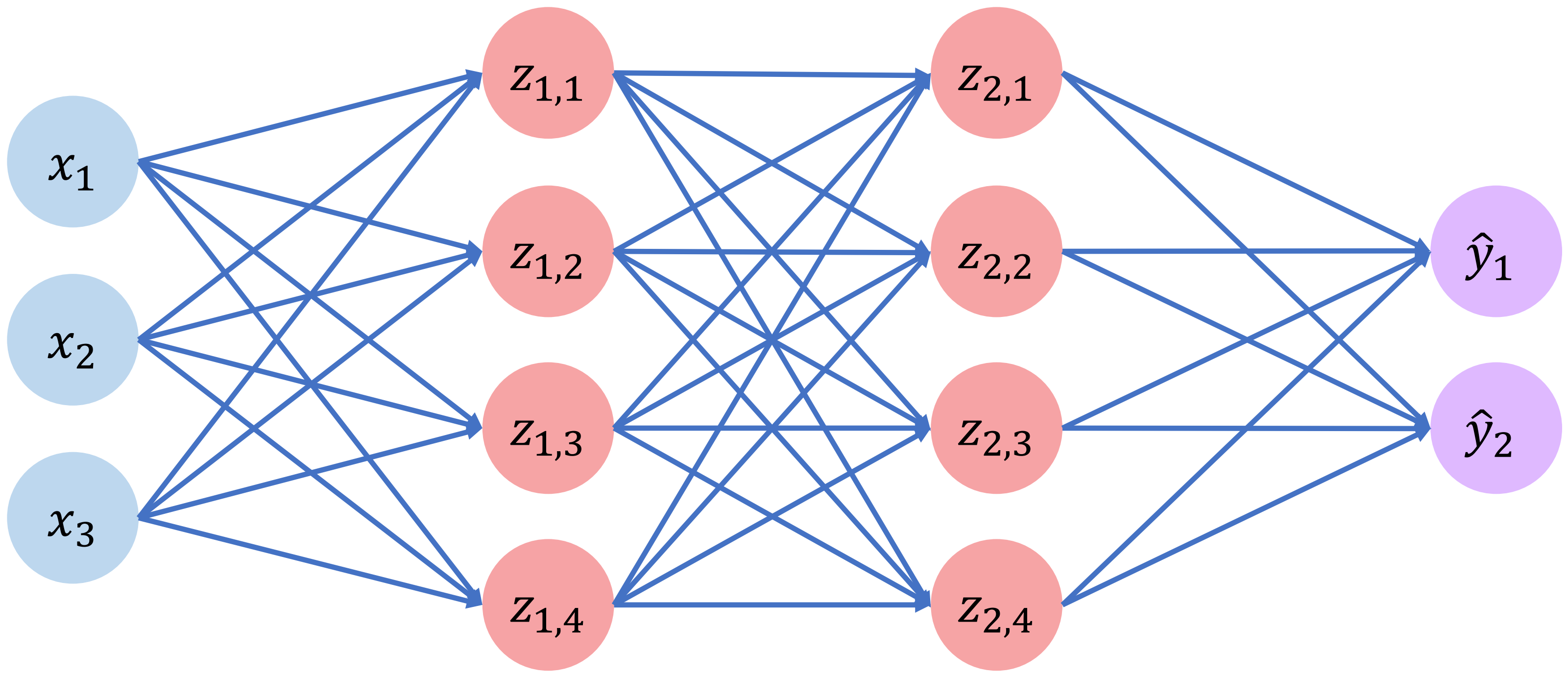
*Technique that constrains our optimization problem to discourage complex models*

*Why do we need it?*

*Improve generalization of our model on unseen data*

# Regularization I: Dropout

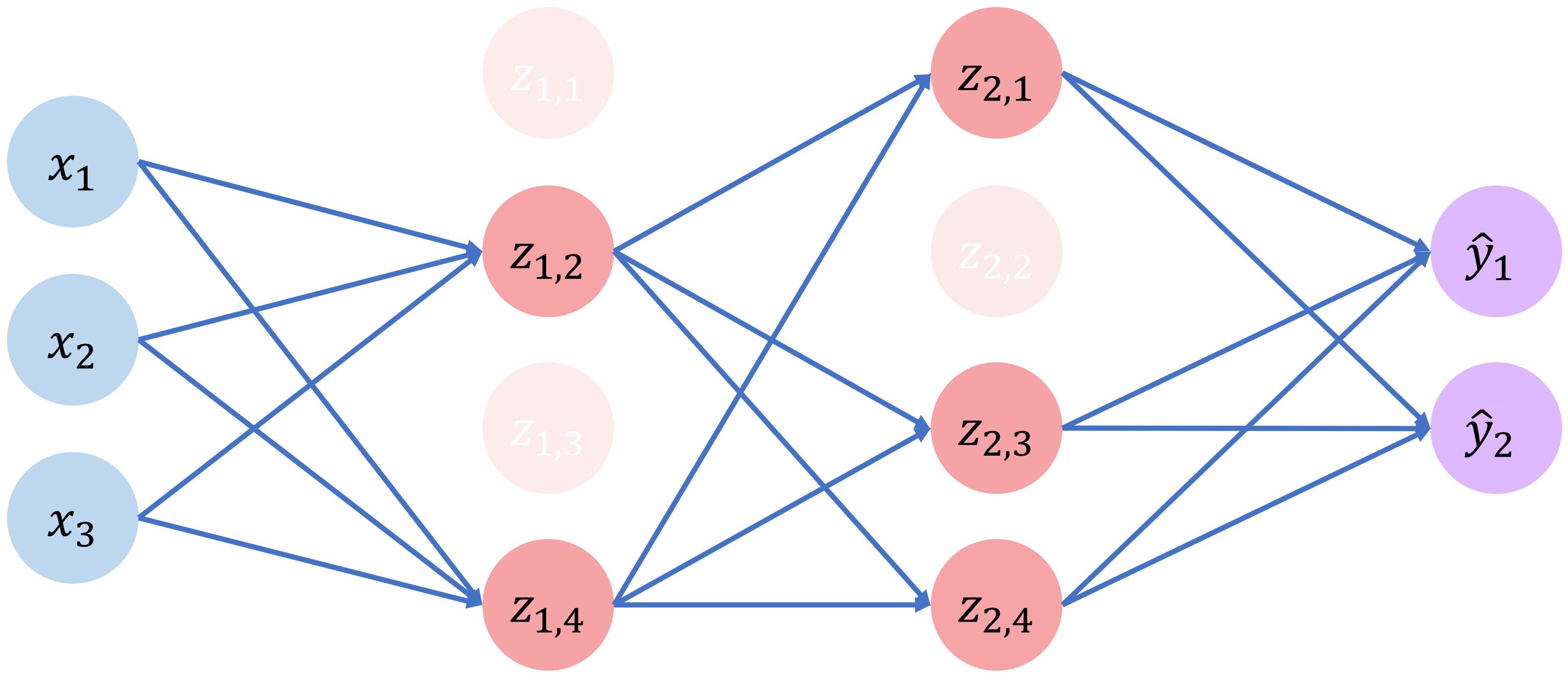
- During training, randomly set some activations to 0



# 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

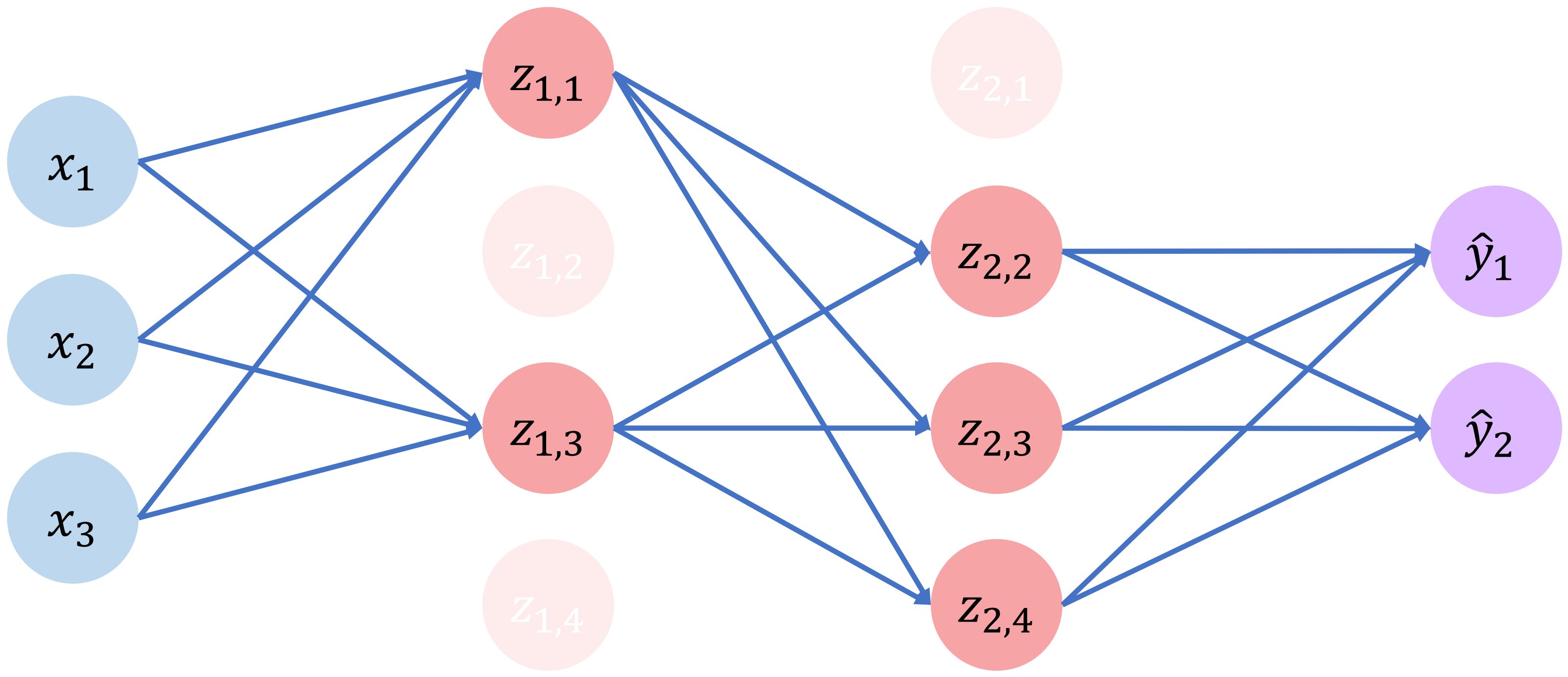
```
tf.keras.layers.Dropout(p=0.5)
```



# 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)
```

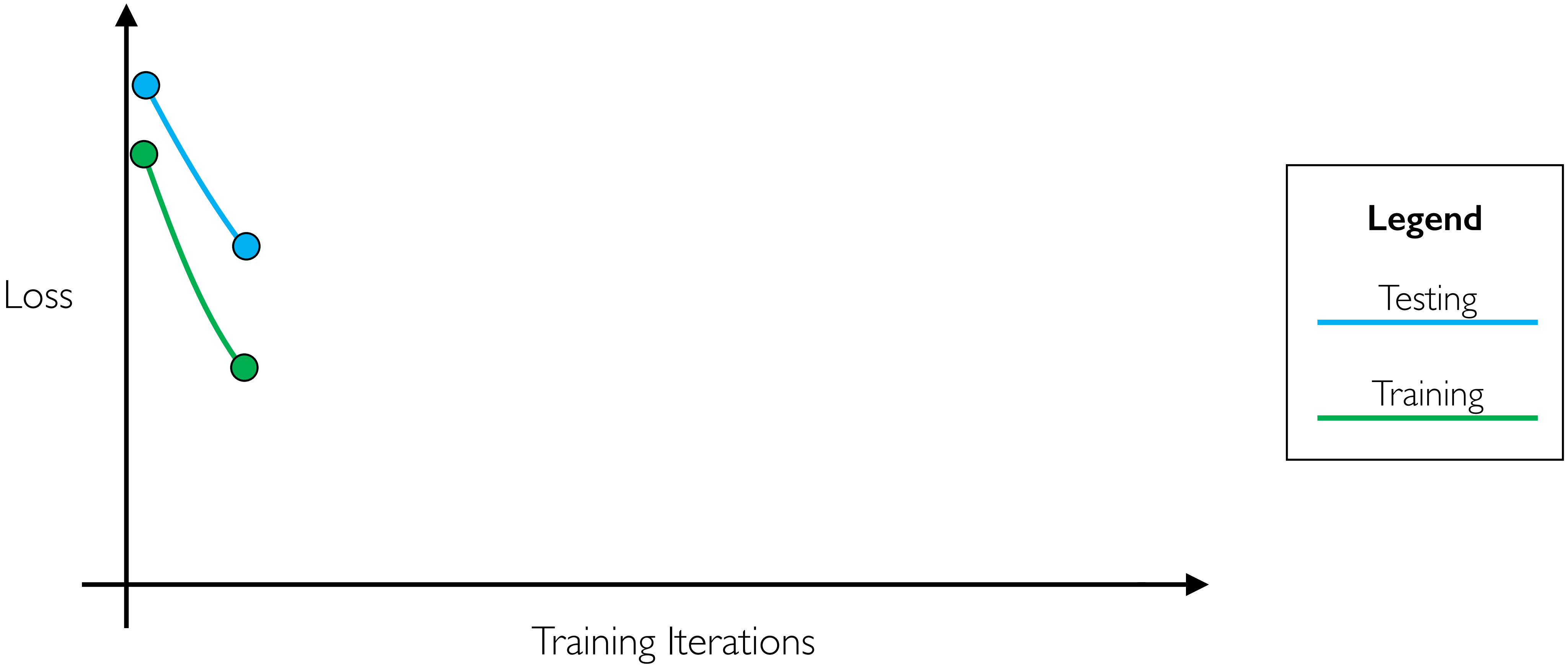


# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

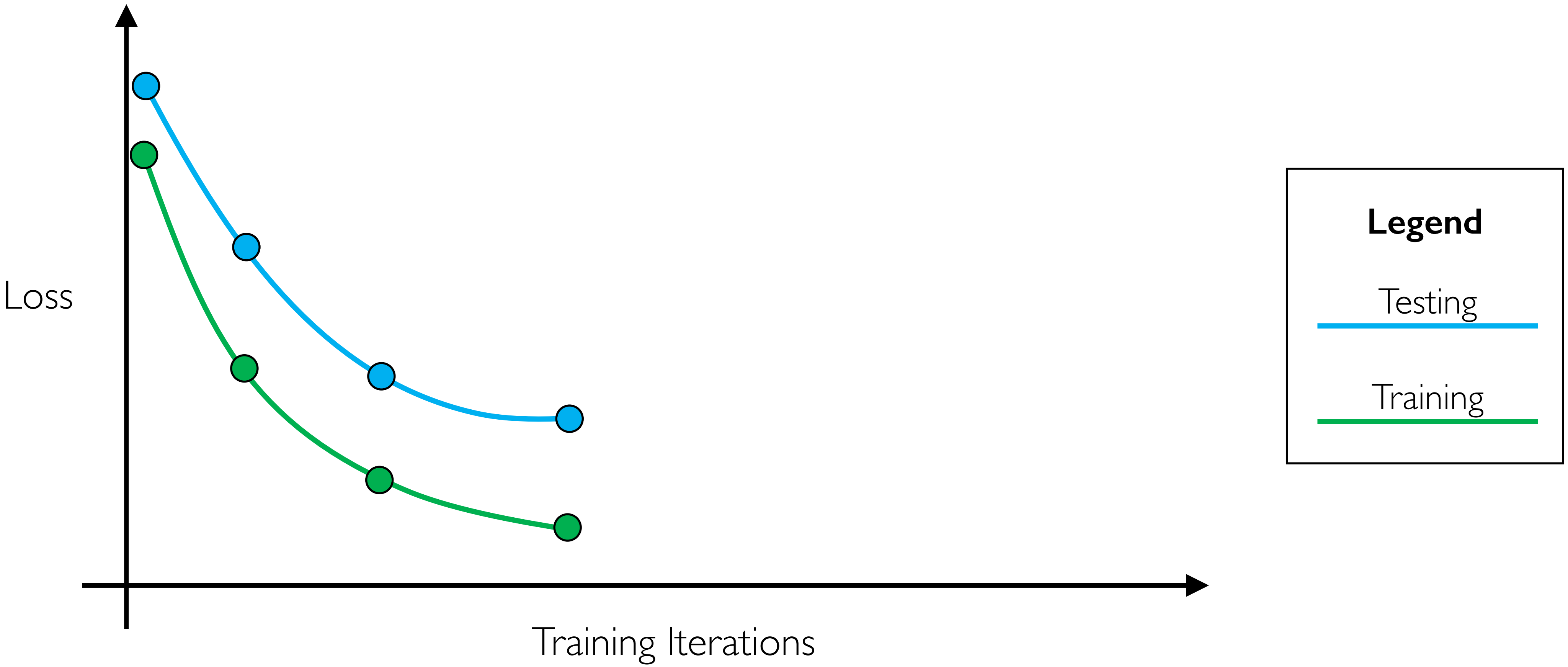


# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

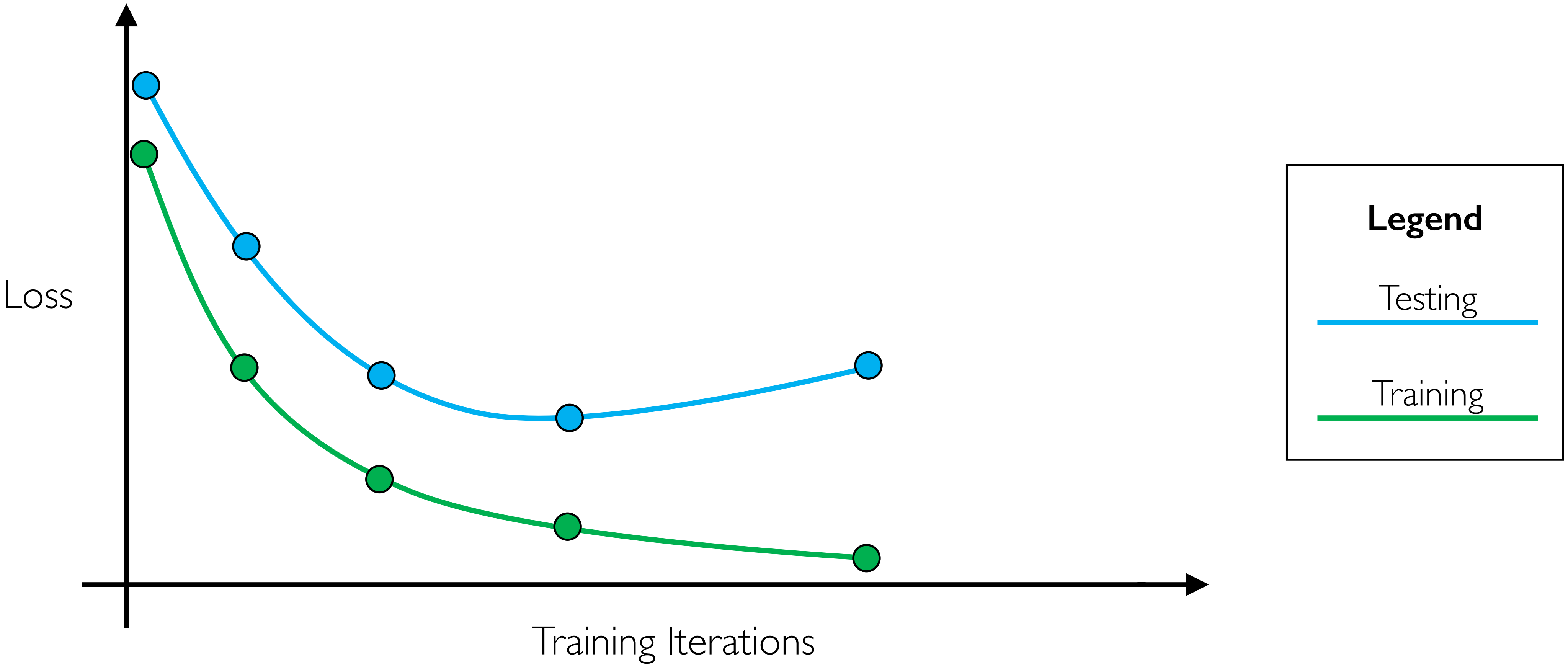
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



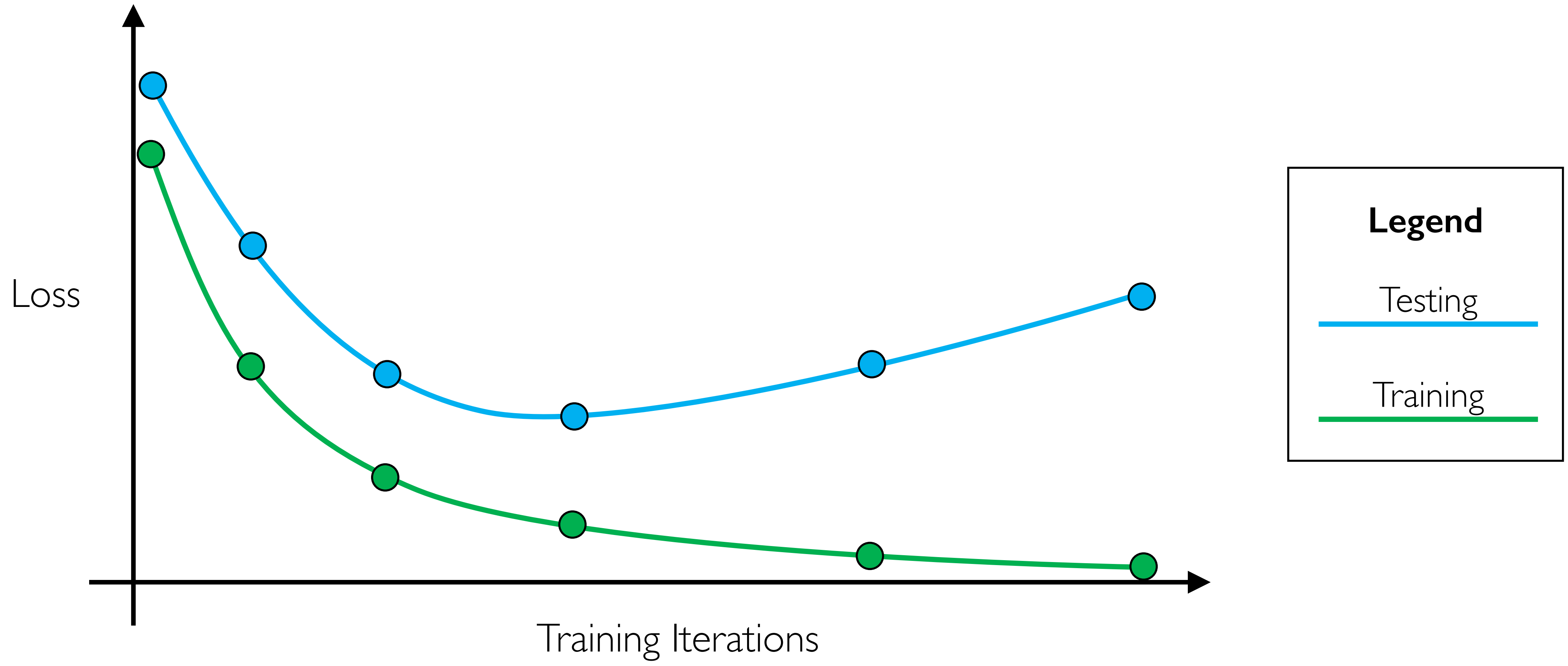
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



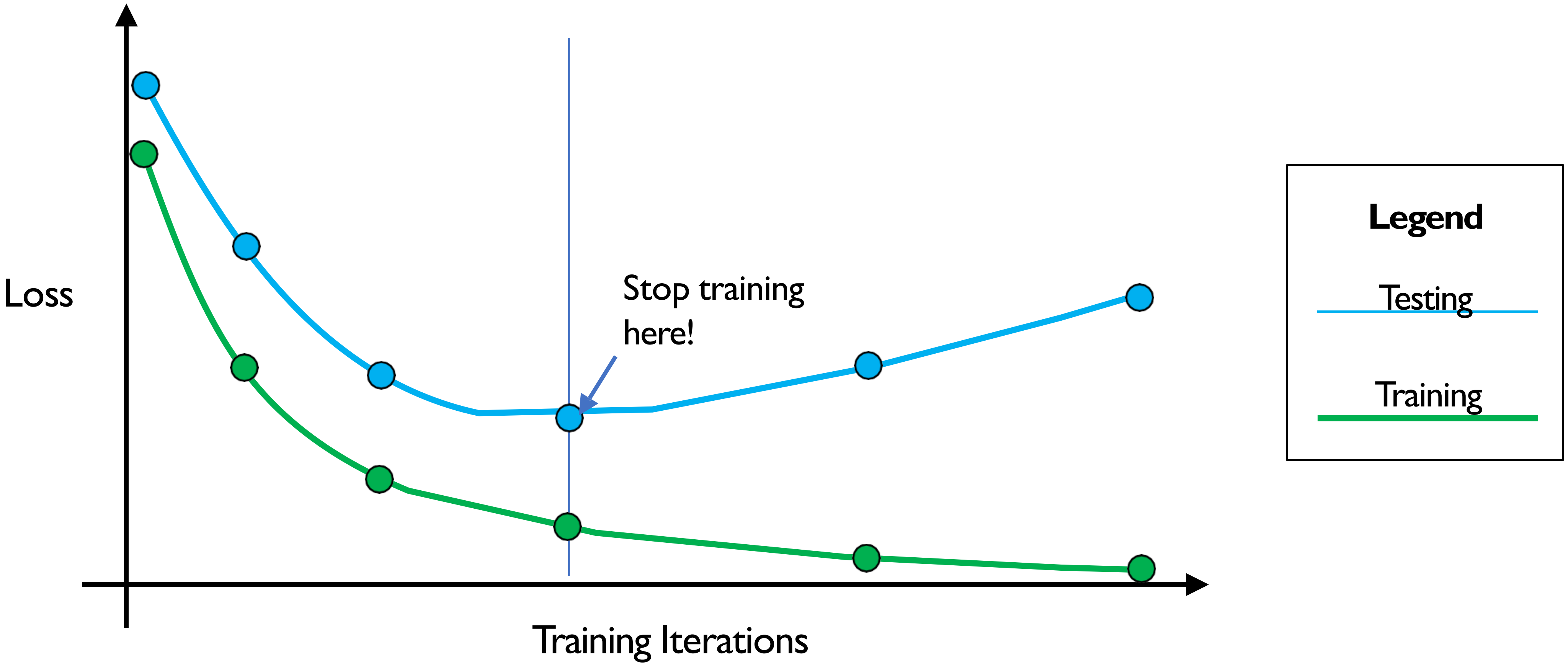
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



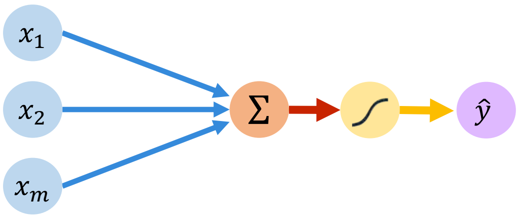
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

# Core Foundation Review

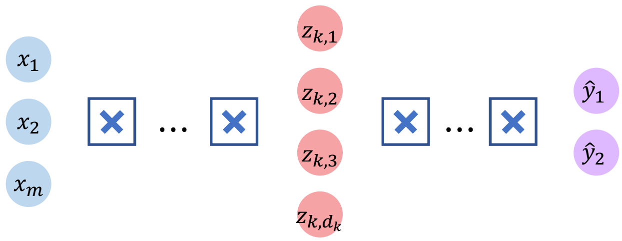
## The Perceptron

- Structural building blocks
- Nonlinear activation functions



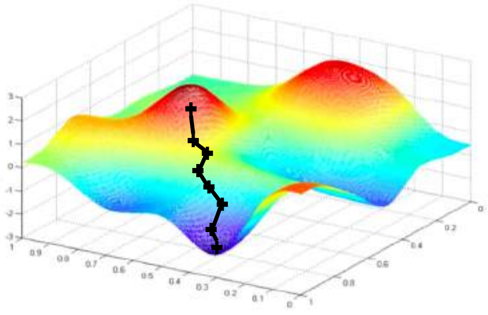

## Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation




## Training in Practice

- Adaptive learning
- Batching
- Regularization



Questions?