Introduction to Deep Learning
MIT 6.S191

Alexander Amini

What is Deep Learning?

Artificial Intelligence

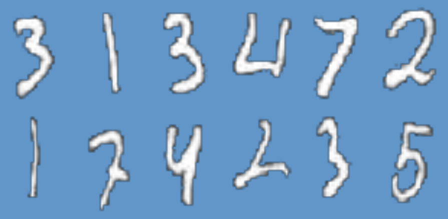
Any technique that enables computers to mimic human behavior

Machine Learning

Ability to learn without explicitly being programmed

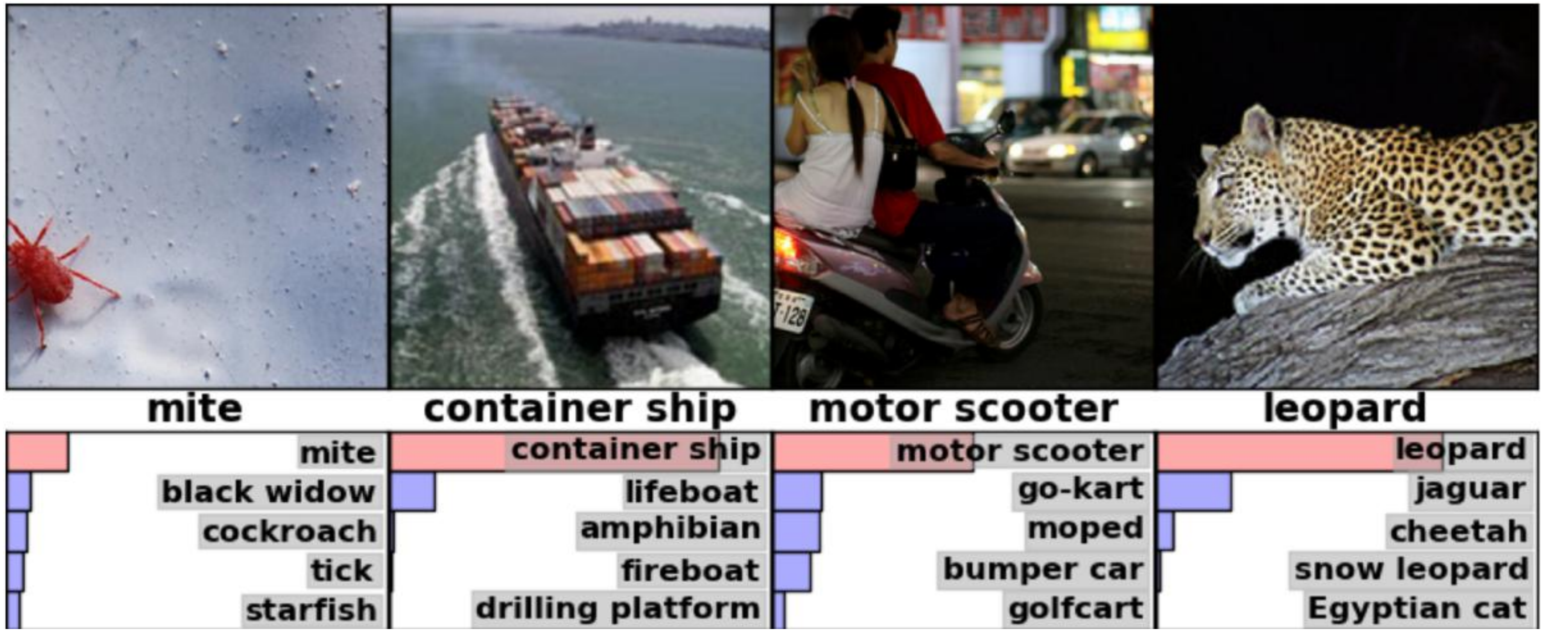
Deep Learning

Learn underlying features in data using neural networks



Deep Learning Success: Vision

Image Recognition



Deep Learning Success: Vision

6.S191 Lab!

Detect pneumothorax in real X-Ray scans

Deep Learning Success: Audio

6.S191 Lab!

Music Generation

Deep Learning Success

And so many more…

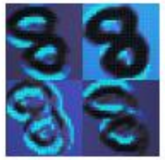
6.S191 Goals

Fundamentals	Practical Skills	Advancements	Community @ MIT

Knowledge, intuition, know-how, and community to do
deep learning research and development

Administrative Information

- Meets Mon Jan 29 – Fri Feb 2
- 10:30 am – 1:30pm
- Lecture + Lab Breakdown
- Graded P/D/F; 3 Units
- 1 Final Assignment

Session	Part 1	Part 2	Lab
1	 Introduction to Deep Learning [**Slides**] [**Video**] *coming soon!*	 Deep Sequence Modeling [**Slides**] [**Video**] *coming soon!*	 Intro to TensorFlow, Music Generation with RNNs [**Code**] *coming soon!*
2	 Deep Computer Vision [**Slides**] [**Video**] *coming soon!*	 Deep Generative Models [**Slides**] [**Video**] *coming soon!*	 Disease Detection from Human X-Ray Scans [**Code**] *coming soon!*
3	 Deep Reinforcement Learning [**Slides**] [**Video**] *coming soon!*	 Open Challenges and New Frontiers [**Slides**] [**Video**] *coming soon!*	 Work time for paper reviews/project proposals
4	Guest Lecture: Google [Info] [**Slides**] [**Video**] *coming soon!*	Guest Lecture: NVIDIA [Info] [**Slides**] [**Video**] *coming soon!*	 Sponsor Booths + Work time for paper reviews/project proposals
5	Guest Lecture: IBM [Info] [**Slides**] [**Video**] *coming soon!*	Guest Lecture: Tencent [Info] [**Slides**] [**Video**] *coming soon!*	 Project Proposal Presentations, Judging and Awards

Final Class Project

Option 1: Proposal Presentation

- Groups of 3 or 4
- Present a novel deep learning research idea or application
- 1 slide, 1 minute (strict)
- List of example proposals on website: introtodeeplearning.com
- Presentations on **Friday, Feb 2**
- Submit groups by **Wednesday 5pm** to be eligible
- Submit slide by **Thursday 9pm** to be eligible

- Judged by a panel of industry judges
- Top winners are awarded:

1x NVIDIA Titan Xp
MSRP: $1200

2x NVIDIA Titan TX2
MSRP: $600

3x Google Home
MSRP: $300

Final Class Project

Option 1: Proposal Presentation

- Groups of 3 or 4
- Present a novel deep learning research idea or application
- 1 slide, 1 minute (strict)
- List of example proposals on website: introtodeeplearning.com
- Presentations on **Friday, Feb 2**
- Submit groups by **Wednesday 5pm** to be eligible
- Submit slide by **Thursday 9pm** to be eligible

Option 2: Write a 1-page review of a deep learning paper

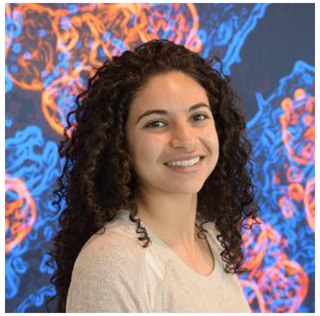
- Suggested papers listed on website: introtodeeplearning.com
- Grade is based on clarity of writing and technical communication of main ideas
- Due **Friday 10:30am** (before lecture)

Class Support

- Piazza: https://piazza.com/class/iwmlwep2fnd5uu
- Course Website: http://introtodeeplearning.com
- Lecture slides: http://introtodeeplearning.com/#schedule
- Email us: introtodeeplearning-staff@mit.edu
- Office Hours by request

Course Staff

Alexander Amini
Lead Organizer



Ava Soleimany
Lead Organizer

Harini Suresh
Co-Chair

Lex Fridman
Co-Chair

Shraman

Yilun

Abhimanyu

Jose Javier

Alex

Nikhil

Faraaz

Tim

Bhaktipriya

introtodeeplearning-staff@mit.edu

Thanks to Sponsors!

NVIDIA

Google

Tencent AI Lab

IBM Research

Microsoft

Why Deep Learning and Why Now?

Why Deep Learning?

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

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

Low Level Features **Mid Level Features** **High Level Features**

Lines & Edges Eyes & Nose & Ears 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

IMAGENET

WIKIPEDIA
The Free Encyclopedia

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

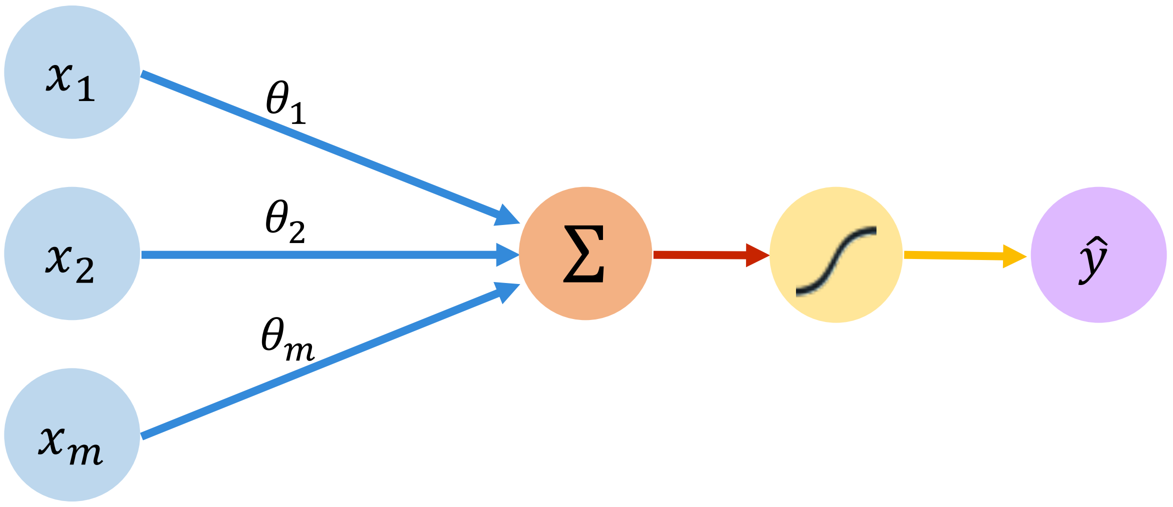
3. Software
- Improved Techniques
- New Models
- Toolboxes

TensorFlow

The Perceptron

The structural building block of deep learning

The Perceptron: Forward Propagation



Inputs Weights Sum Non-Linearity Output

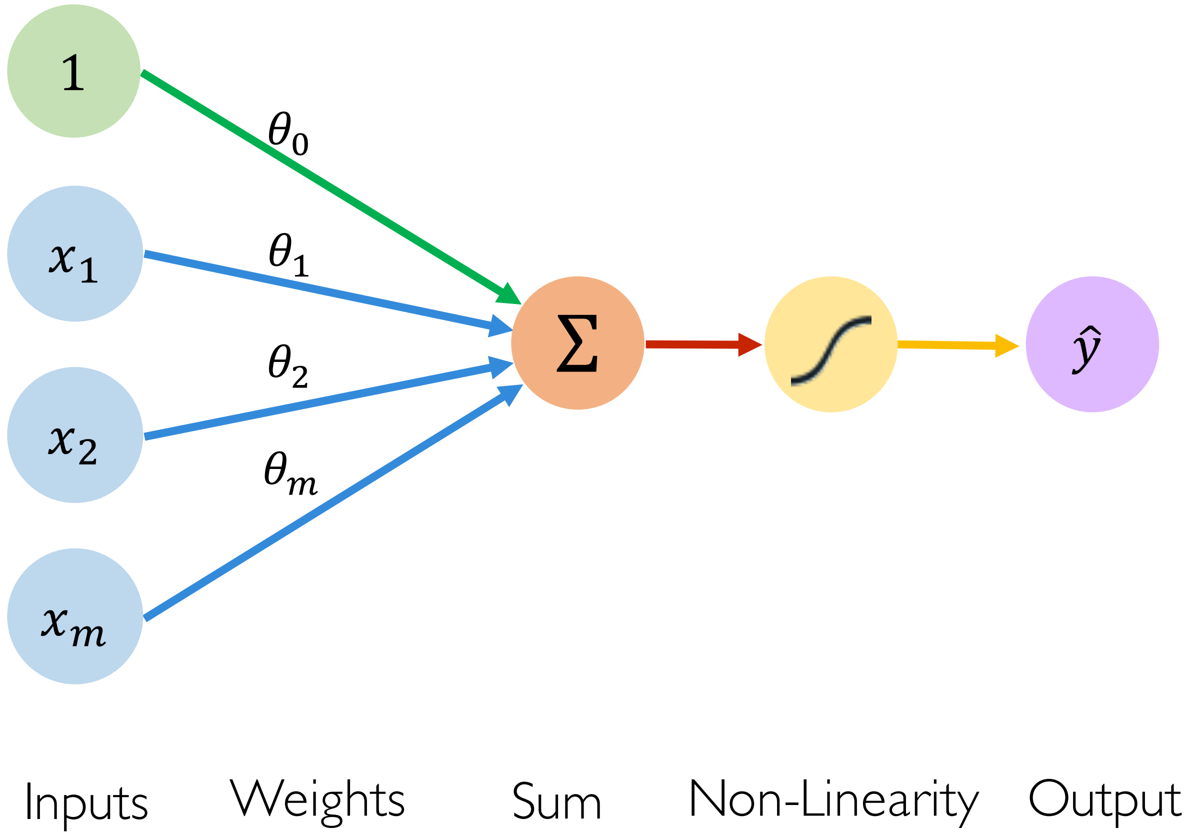
Output

Linear combination of inputs

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

Non-linear activation function

The Perceptron: Forward Propagation



Output

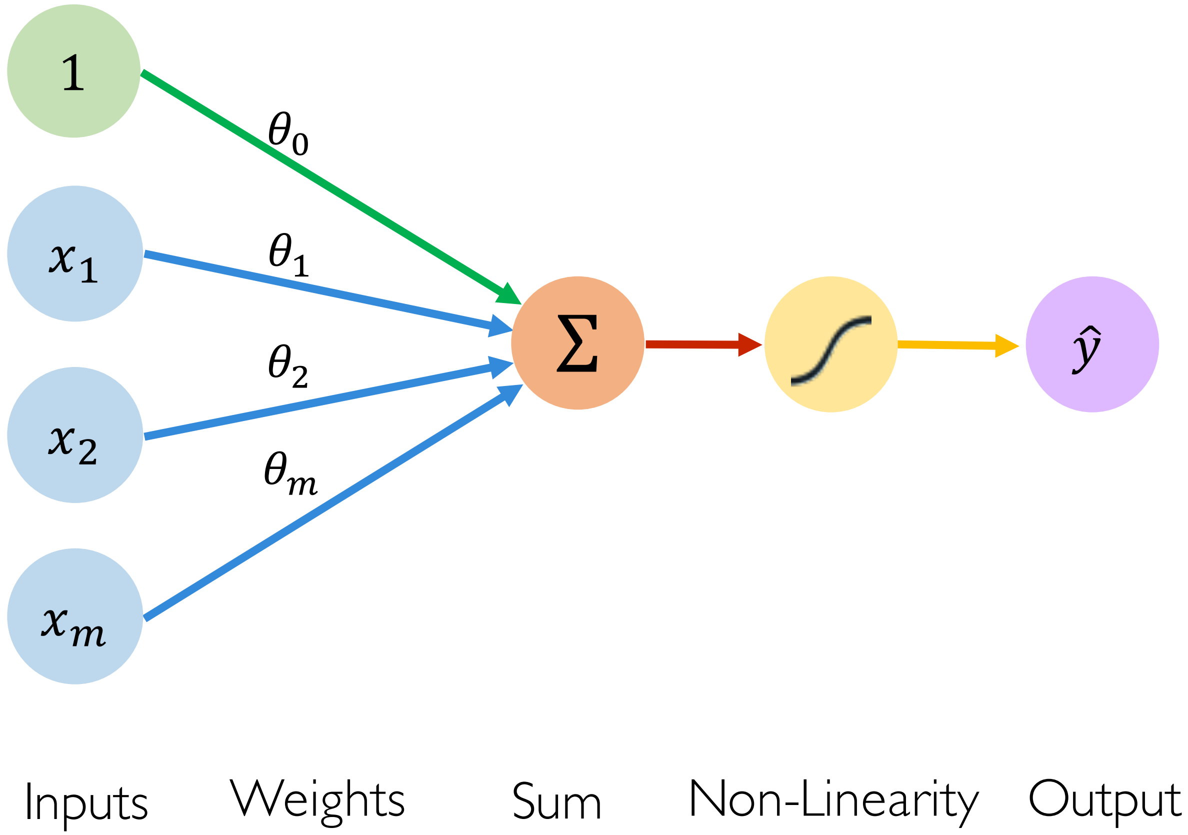
Linear combination of inputs

$$\hat{y} = g\left(\theta_0 + \sum_{i=1}^{m} x_i\,\theta_i\right)$$

Non-linear activation function

Bias

The Perceptron: Forward Propagation

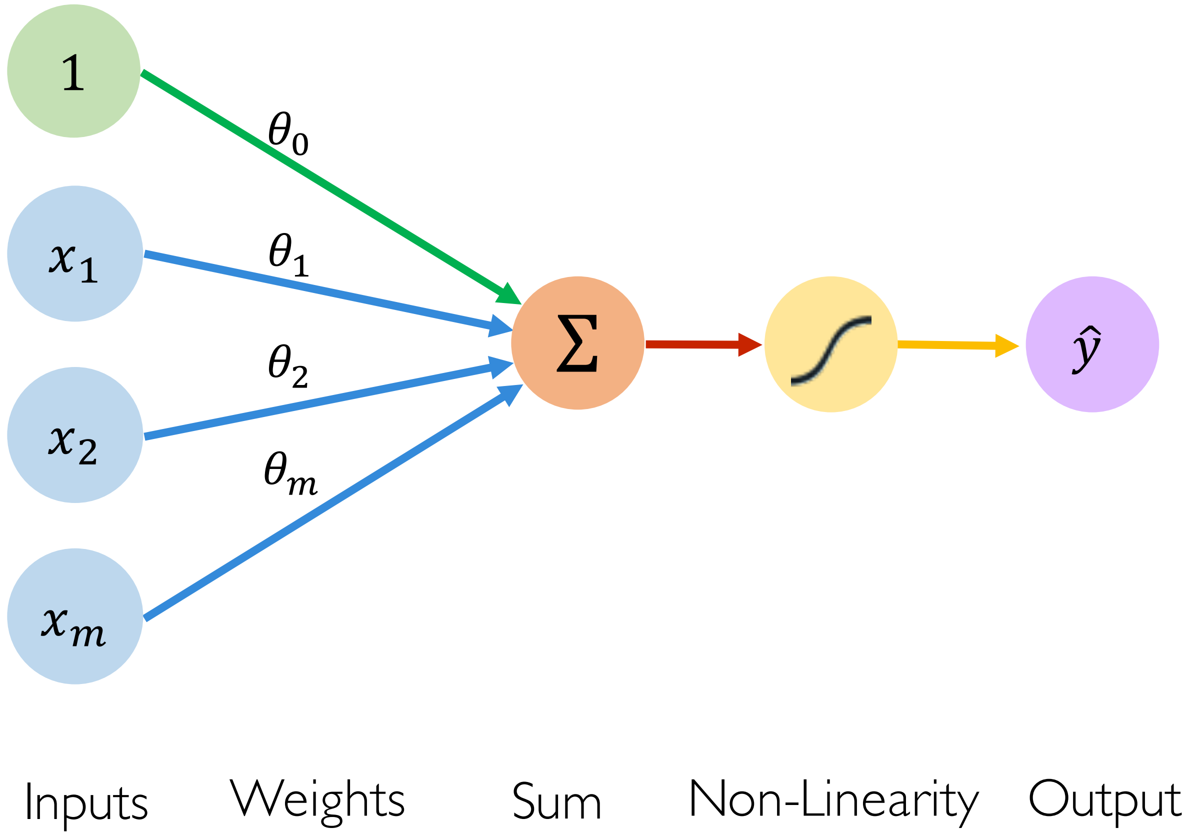


$$\hat{y} = g\left(\theta_0 + \sum_{i=1}^{m} x_i \, \theta_i\right)$$

$$\hat{y} = g\left(\theta_0 + \boldsymbol{X}^T \boldsymbol{\theta}\right)$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_m \end{bmatrix}$

The Perceptron: Forward Propagation

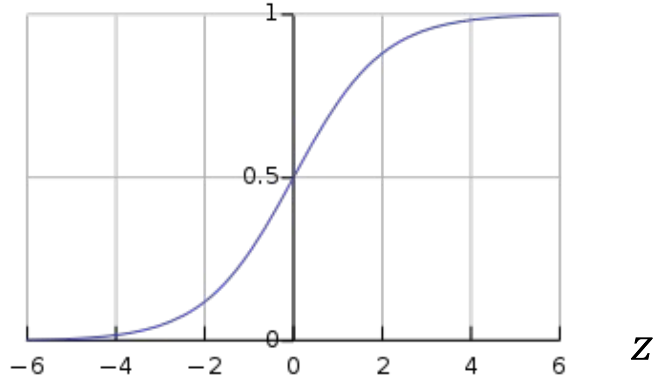


Activation Functions

$$\hat{y} = g\left(\theta_0 + \boldsymbol{X}^T \boldsymbol{\theta} \right)$$

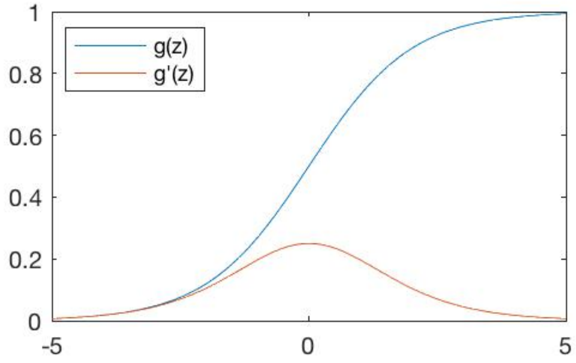
- Example: sigmoid function

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



Common Activation Functions

Sigmoid Function

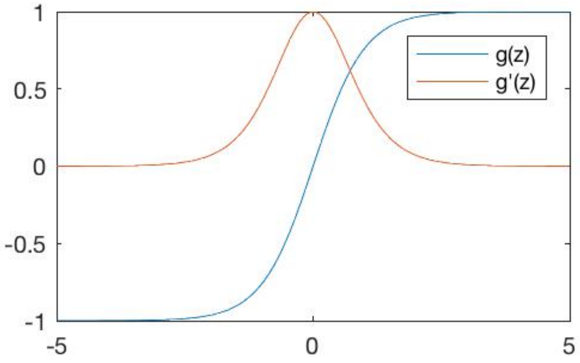


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

`tf.nn.sigmoid(z)`

Hyperbolic Tangent

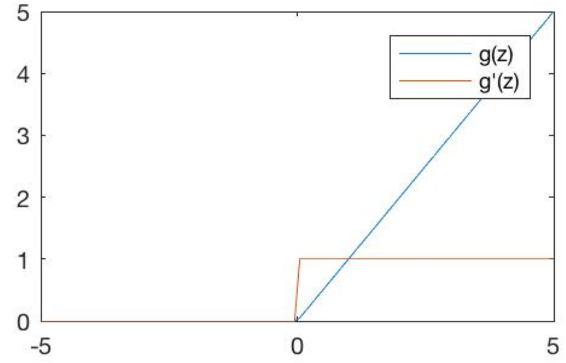


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

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

`tf.nn.tanh(z)`

Rectified Linear Unit (ReLU)



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

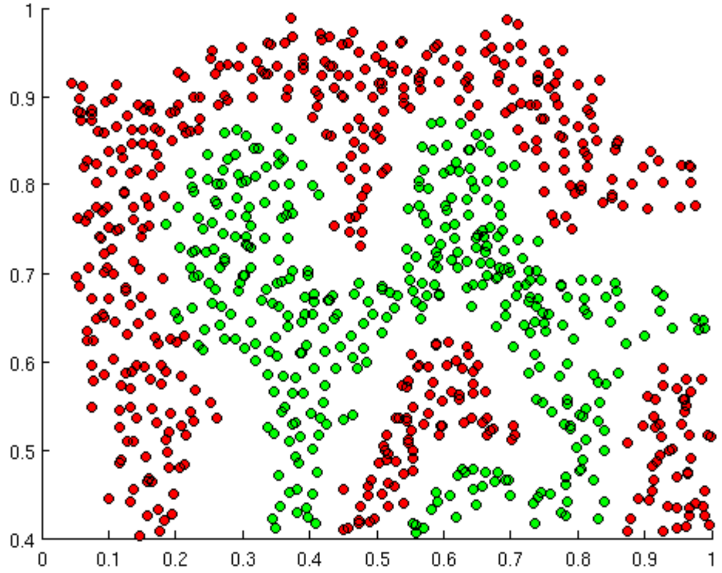
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

`tf.nn.relu(z)`

NOTE: All activation functions are non-linear

Importance of Activation Functions

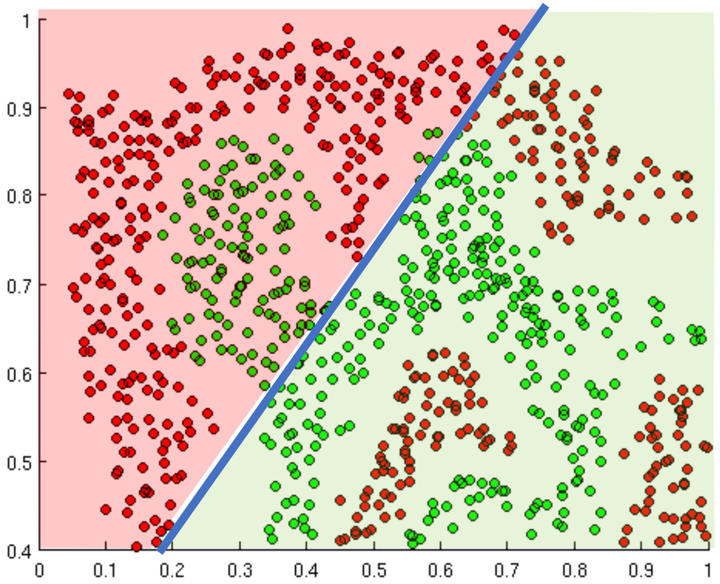
The purpose of activation functions is to ***introduce non-linearities*** *into the network*



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

Importance of Activation Functions

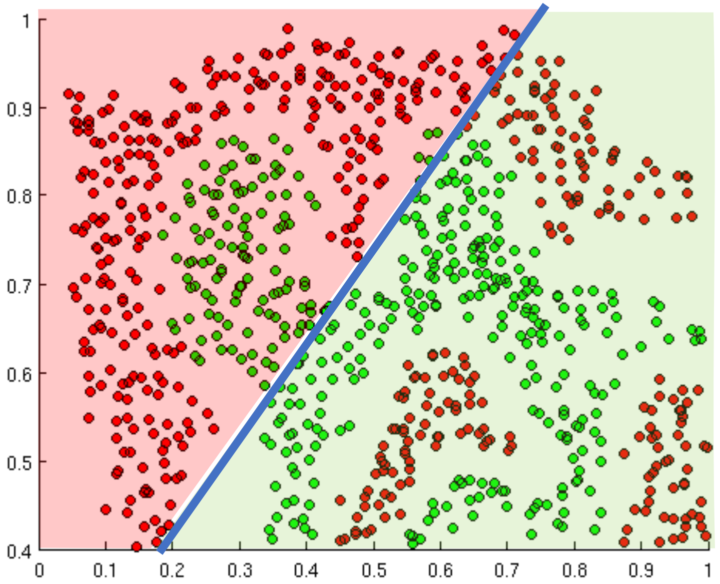
The purpose of activation functions is to ***introduce non-linearities*** *into the network*



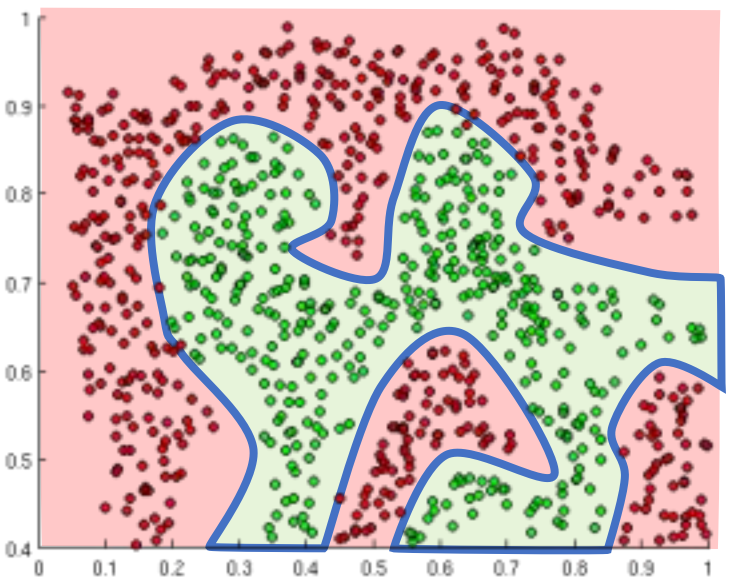
Linear Activation functions produce linear decisions no matter the network size

Importance of Activation Functions

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

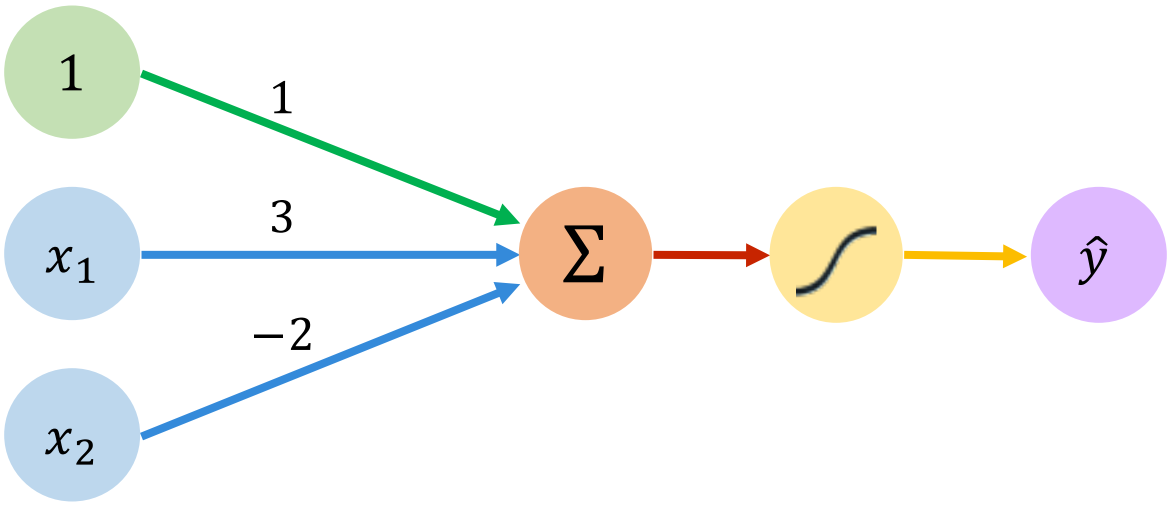


Linear Activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example



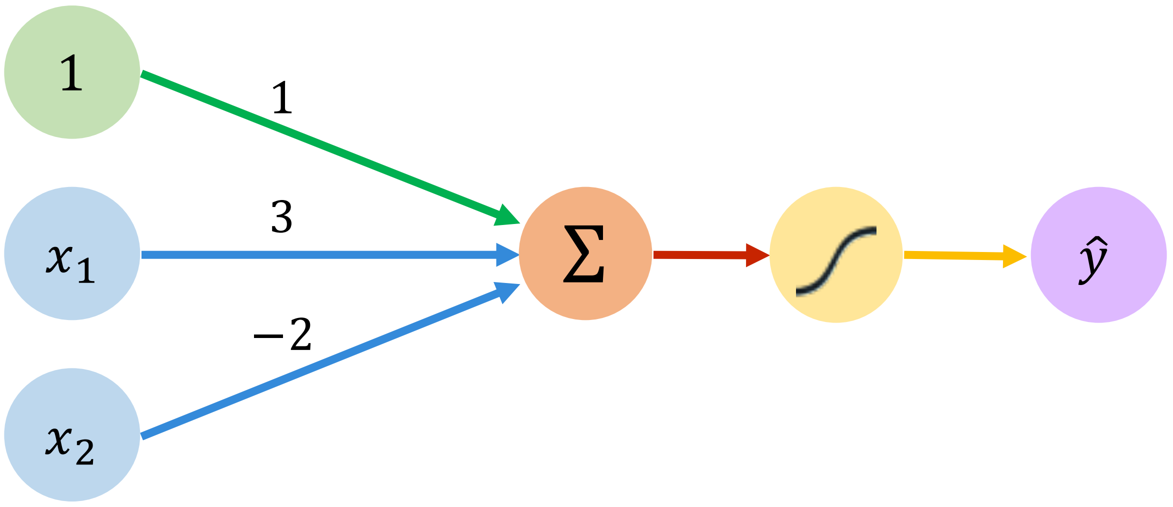
We have: $\theta_0 = 1$ and $\boldsymbol{\theta} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g\left(\theta_0 + \boldsymbol{X}^T\boldsymbol{\theta}\right)$$

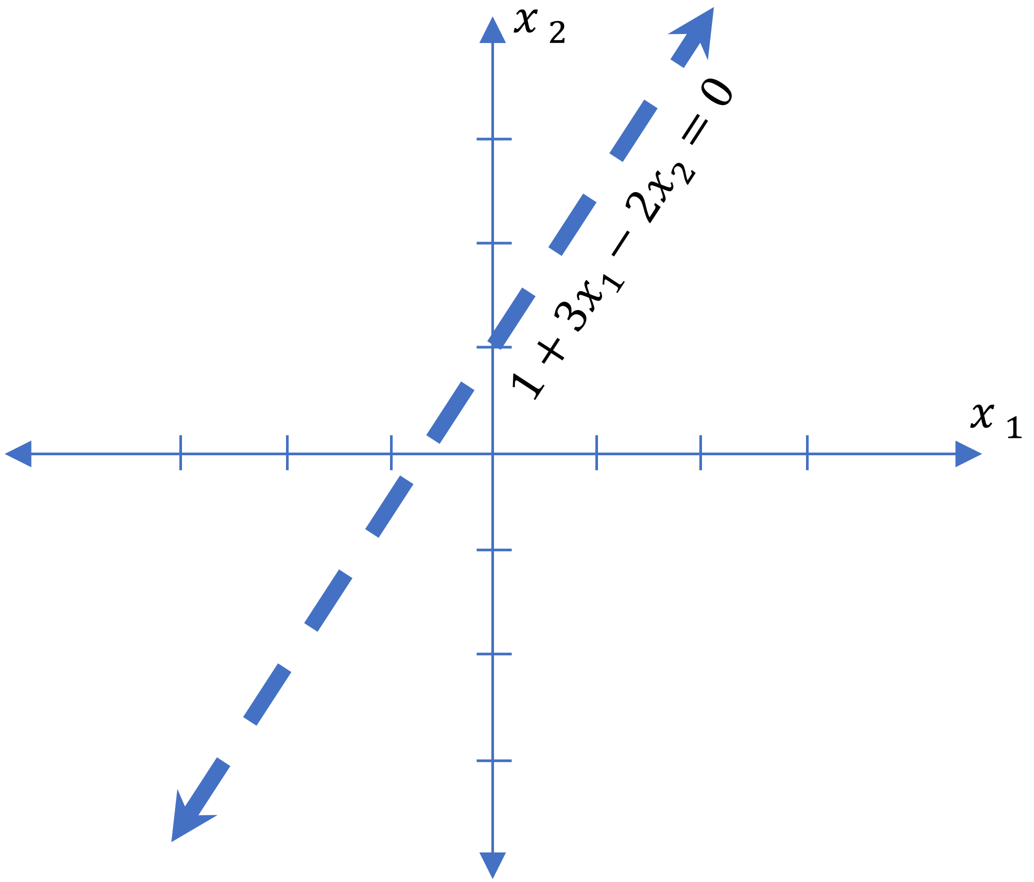
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$

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

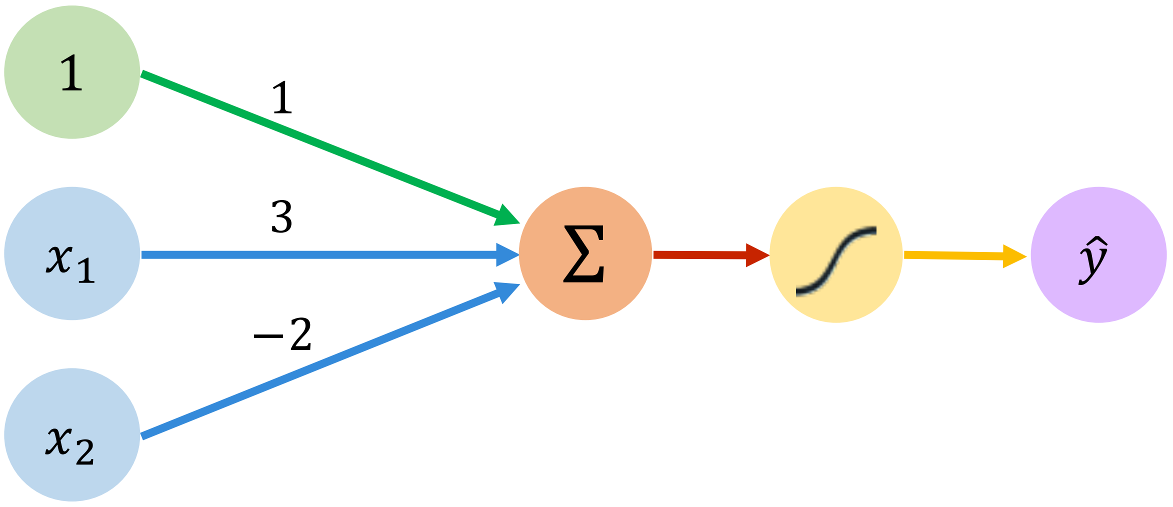
The Perceptron: Example



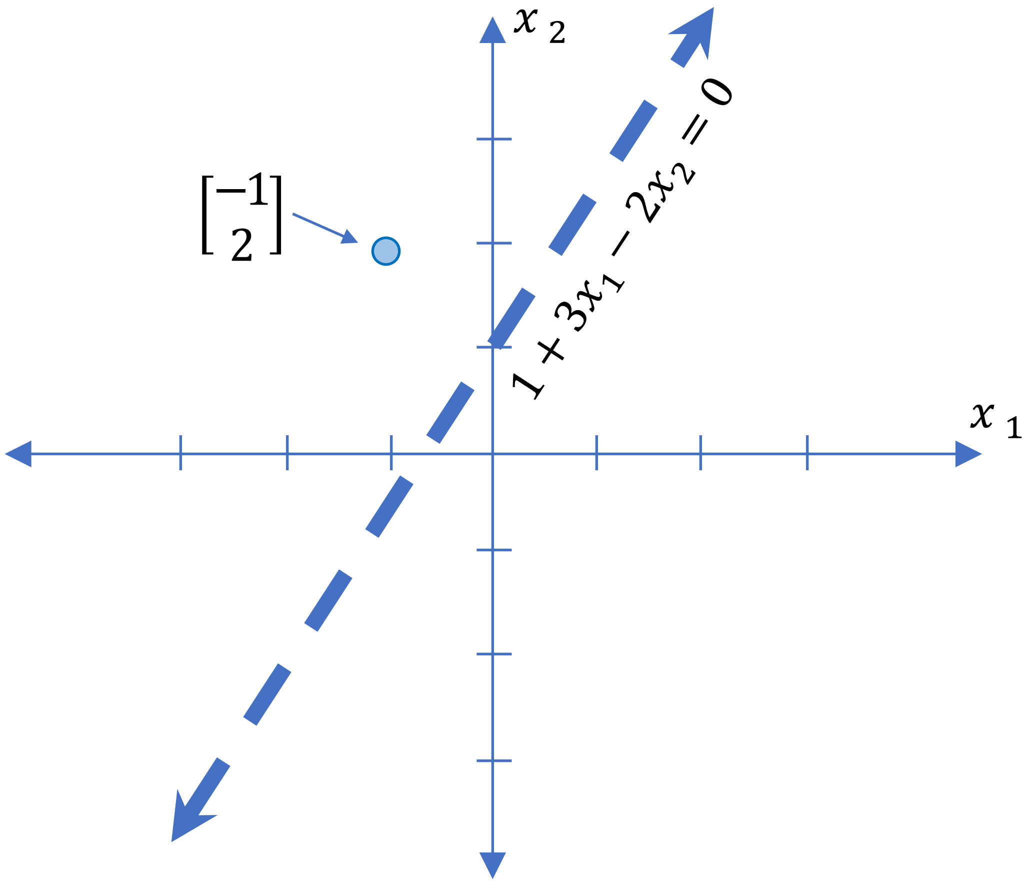
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



The Perceptron: Example



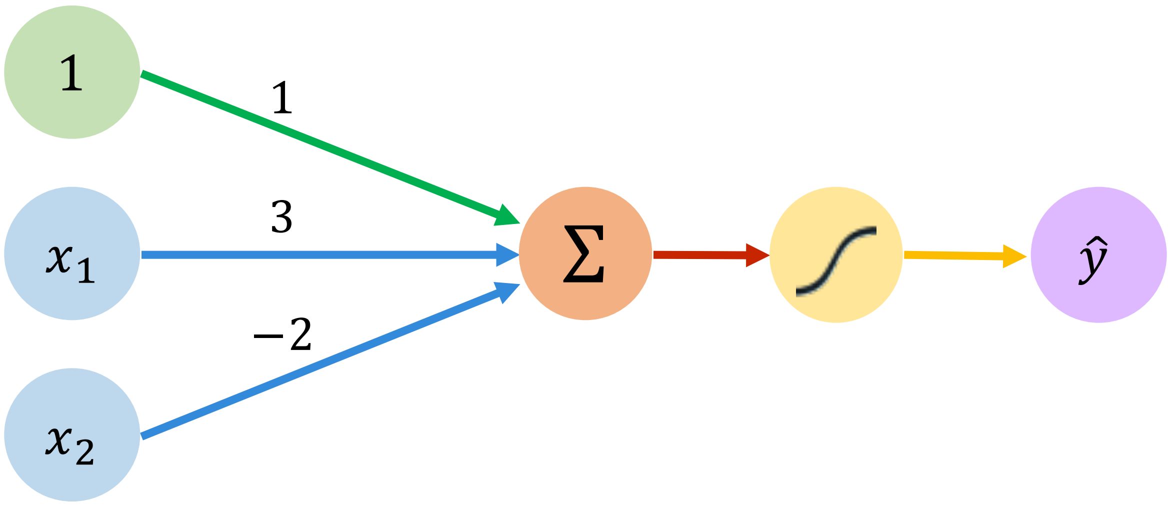
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



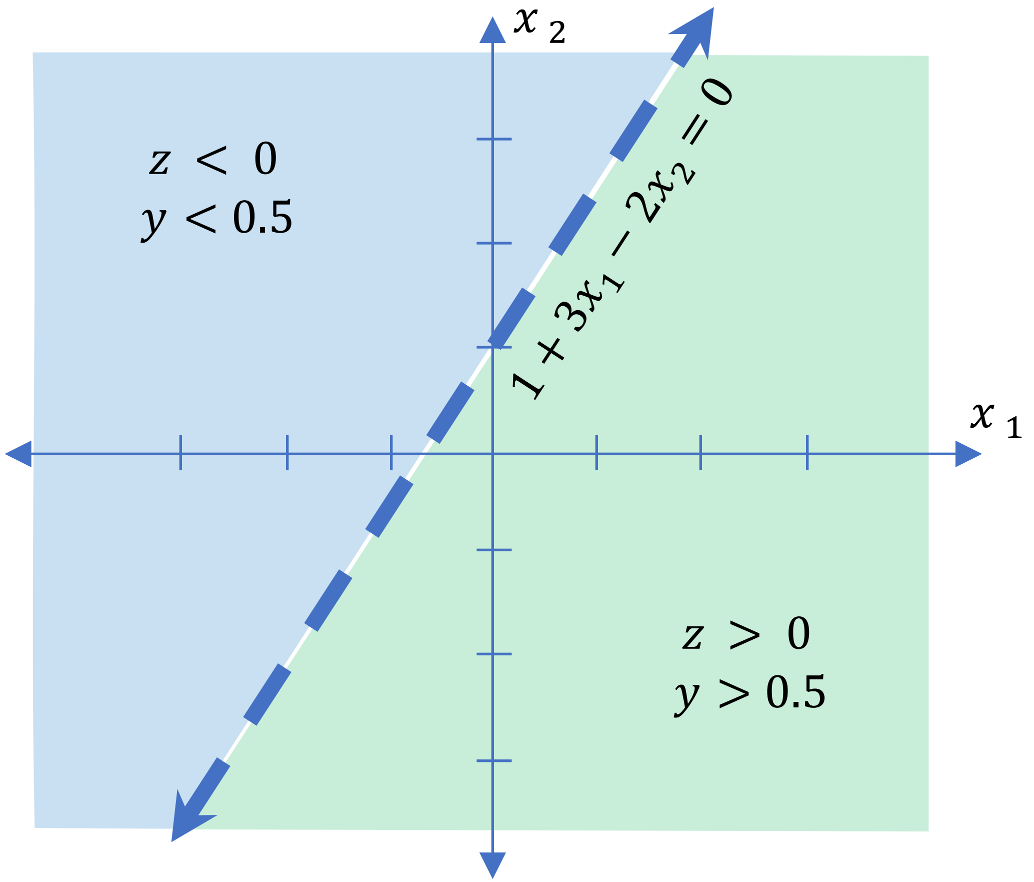
Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

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

The Perceptron: Example

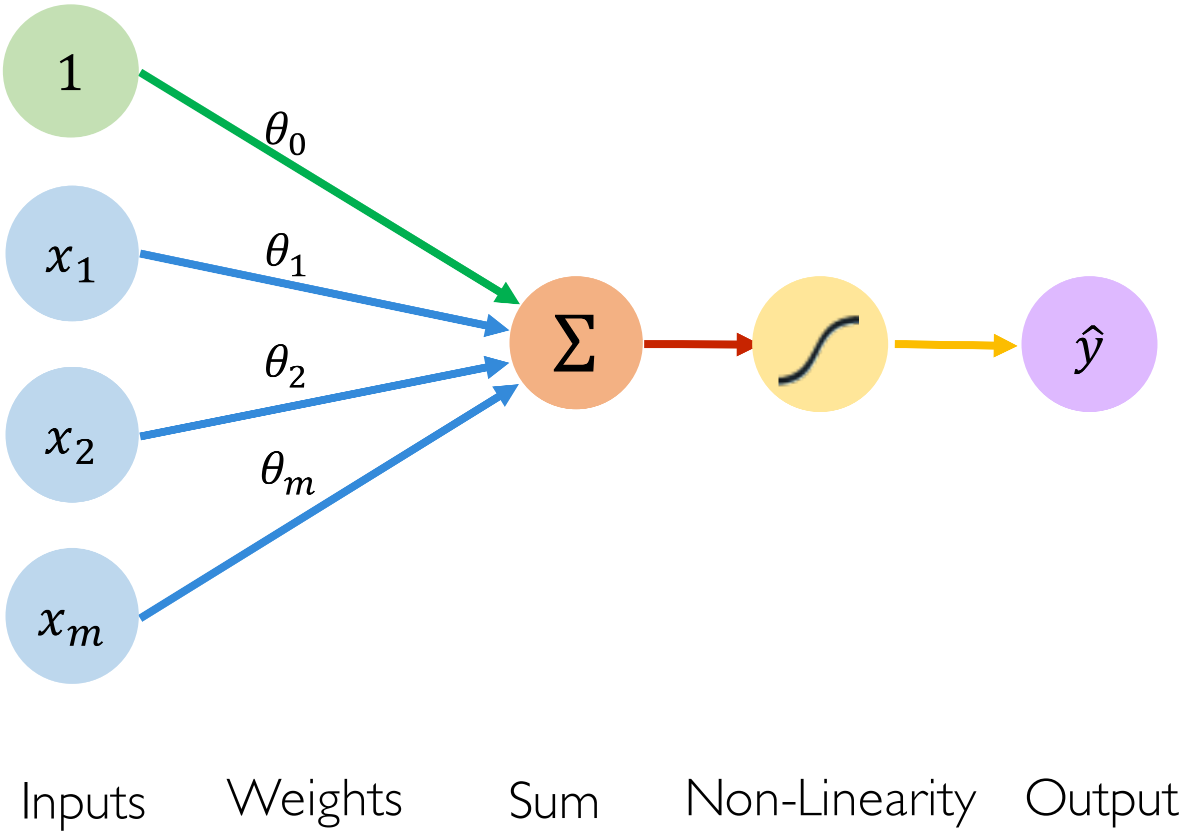


$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

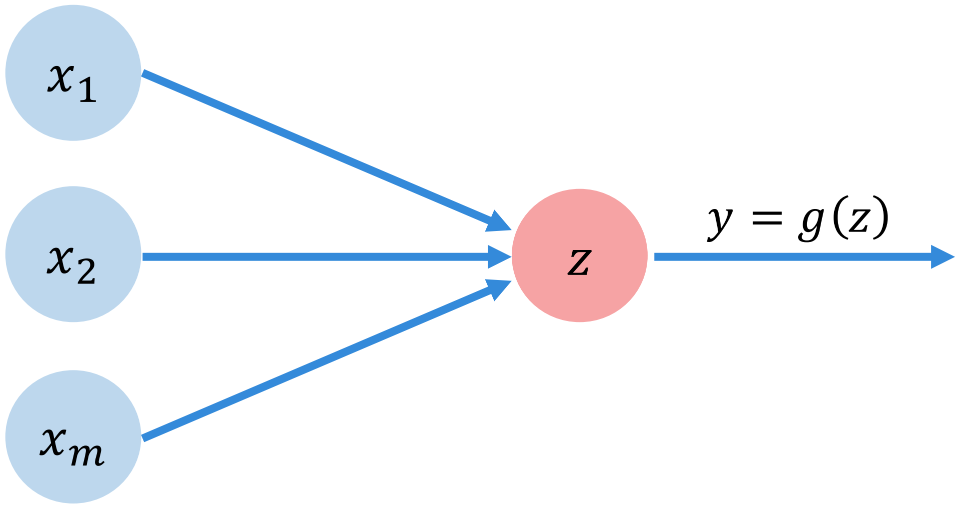


Building Neural Networks with Perceptrons

The Perceptron: Simplified

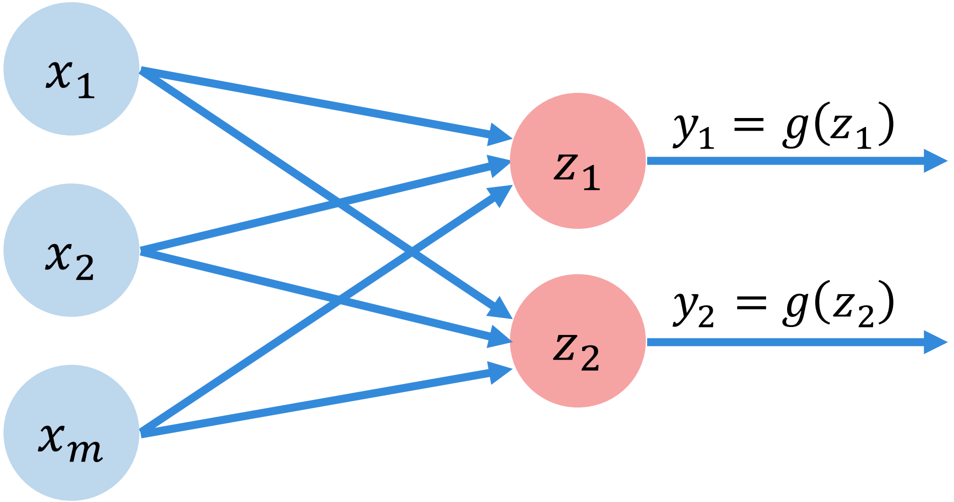


The Perceptron: Simplified



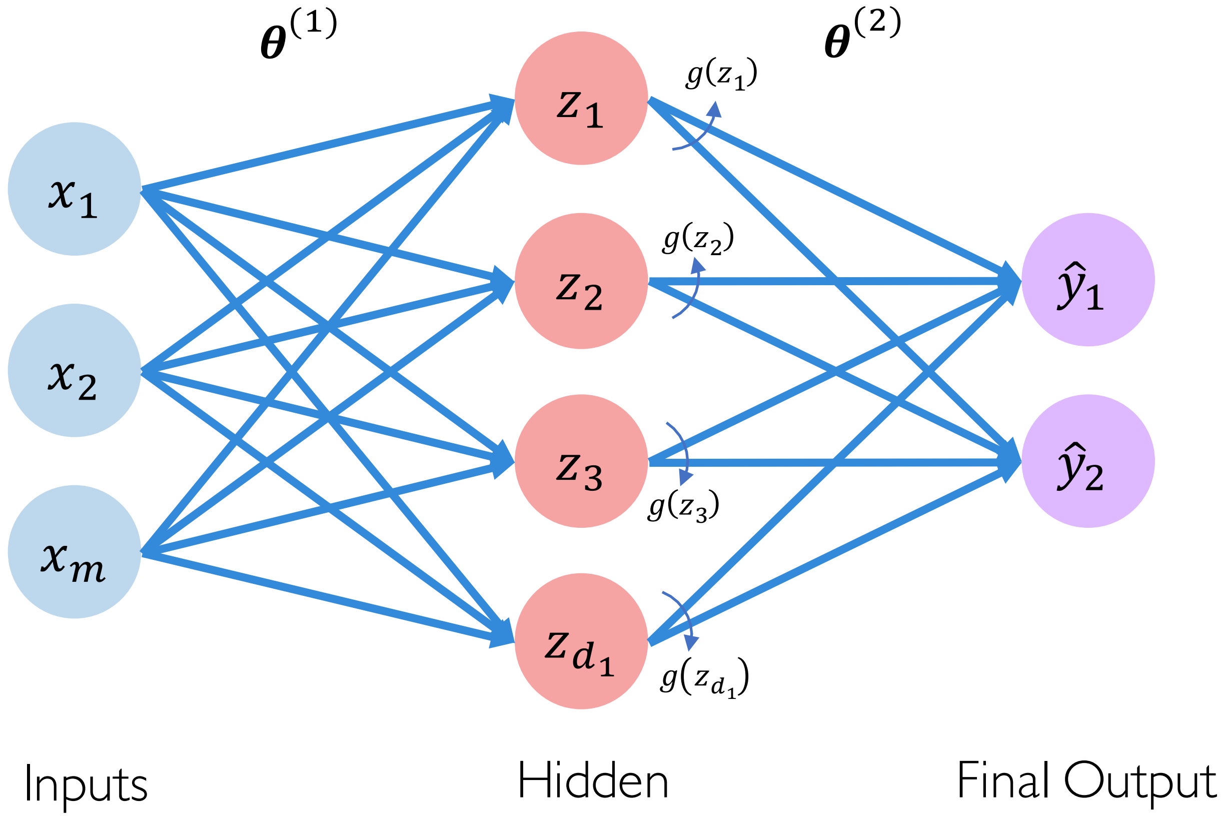
$$z = \theta_0 + \sum_{j=1}^{m} x_j \, \theta_j$$

Multi Output Perceptron



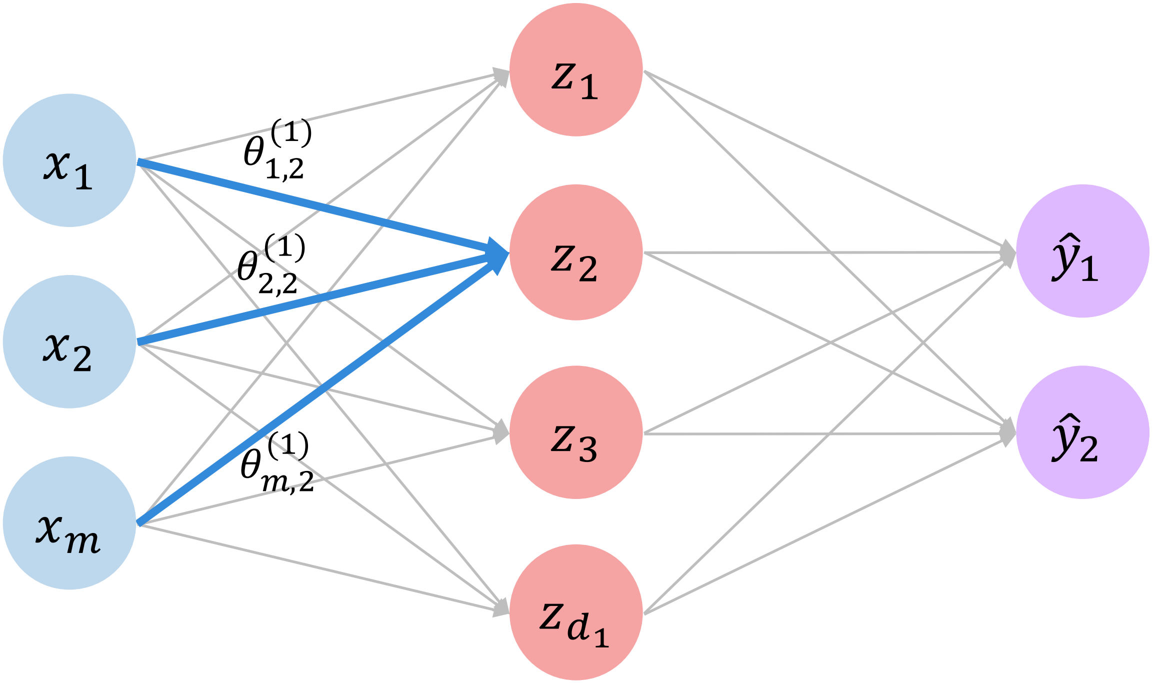
$$z_i = \theta_{0,i} + \sum_{j=1}^{m} x_j \, \theta_{j,i}$$

Single Layer Neural Network



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

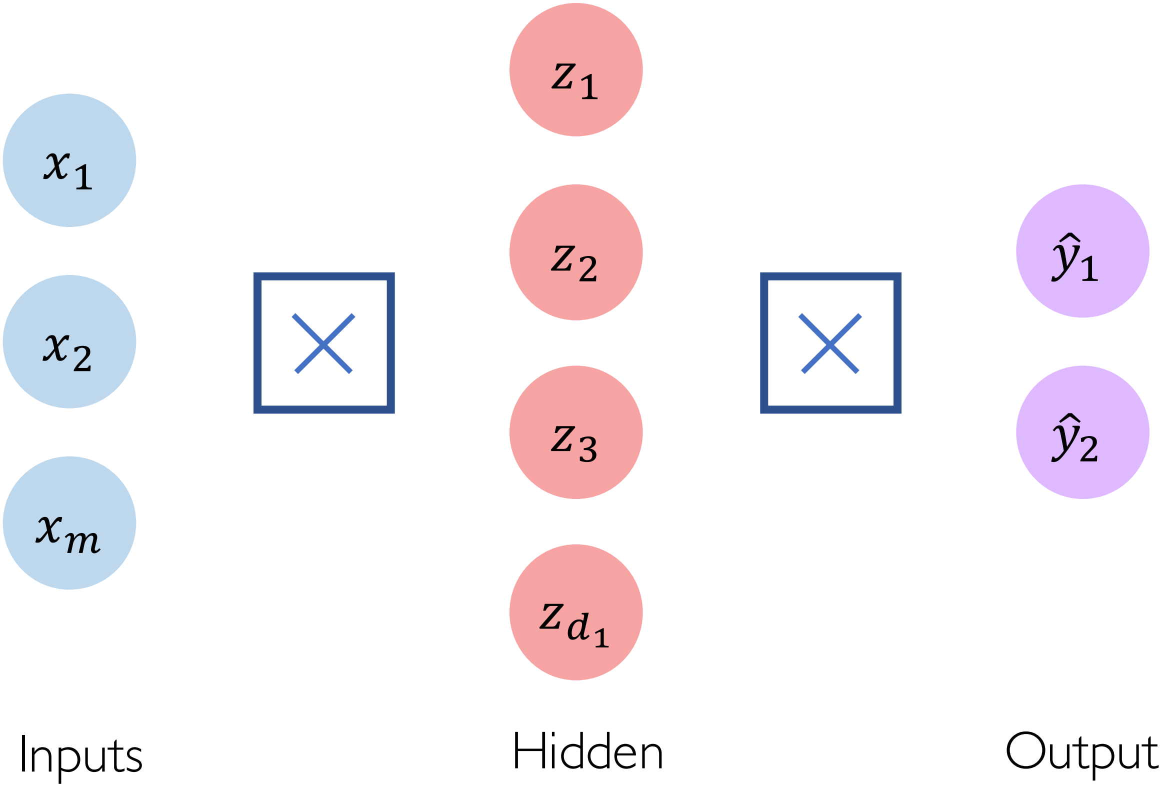
Single Layer Neural Network



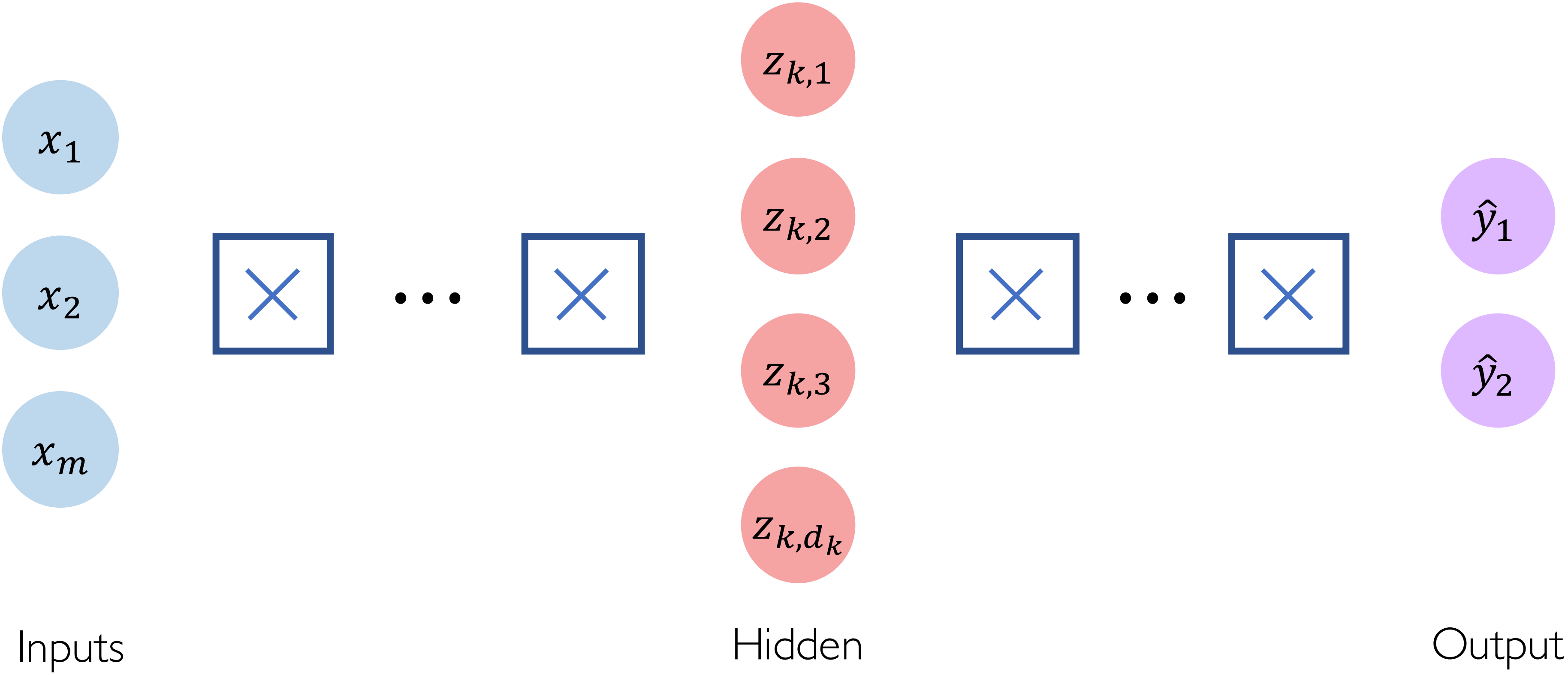
$$z_2 = \theta_{0,2}^{(1)} + \sum_{j=1}^{m} x_j\, \theta_{j,2}^{(1)}$$

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

Multi Output Perceptron



Deep Neural Network



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

Applying Neural Networks

Example Problem

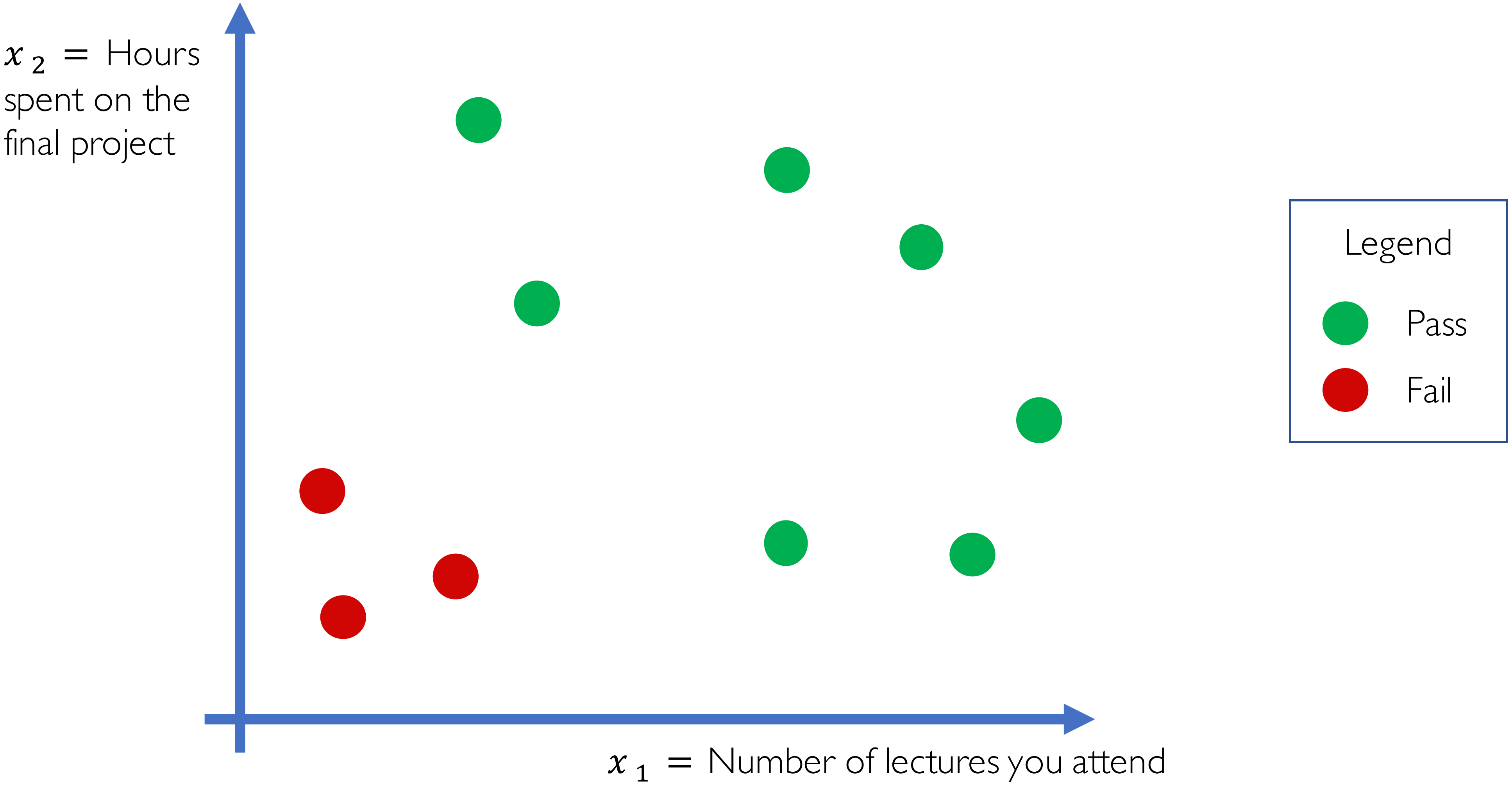
Will I pass this class?

Let's start with a simple two feature model

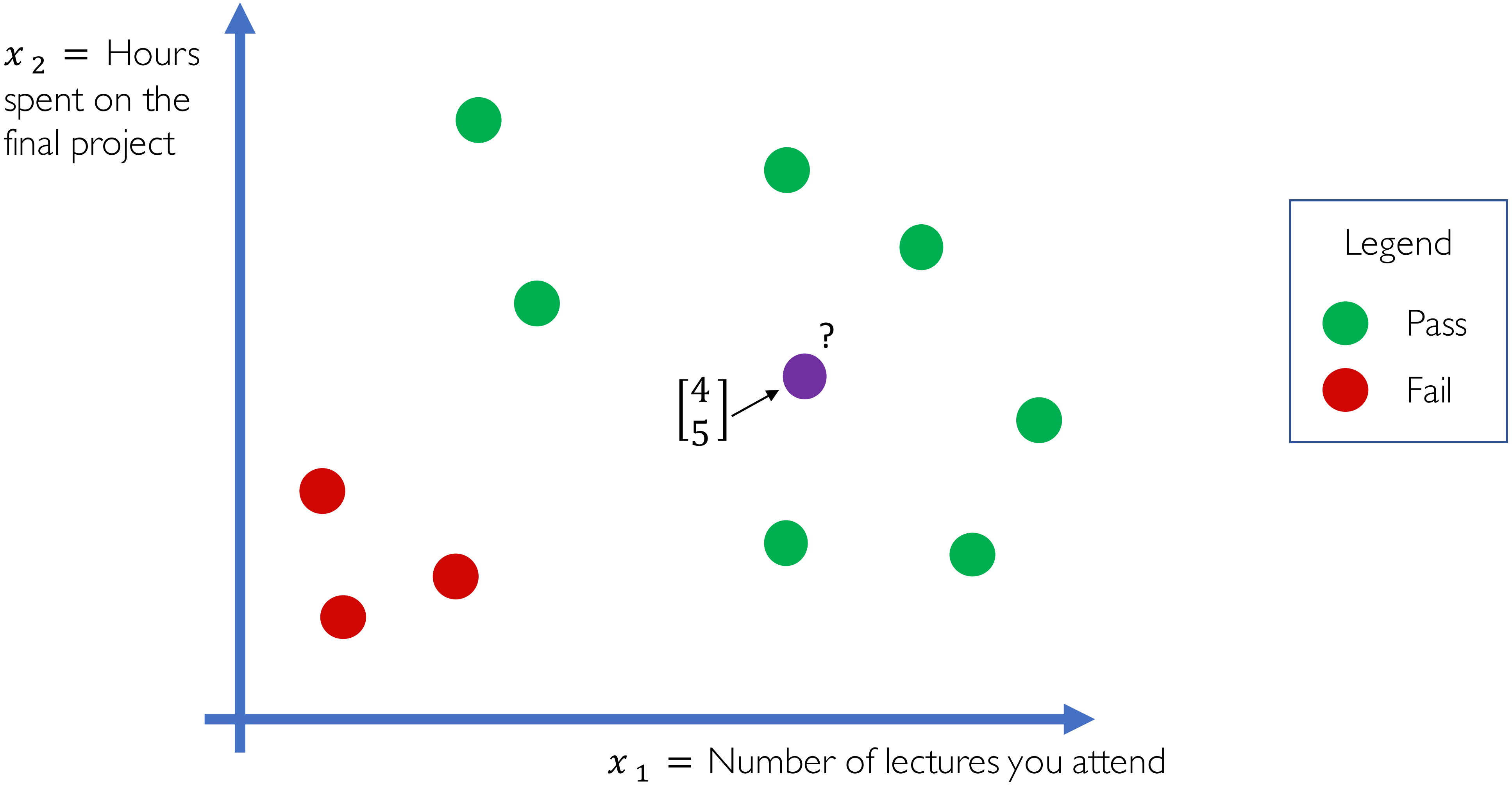
x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

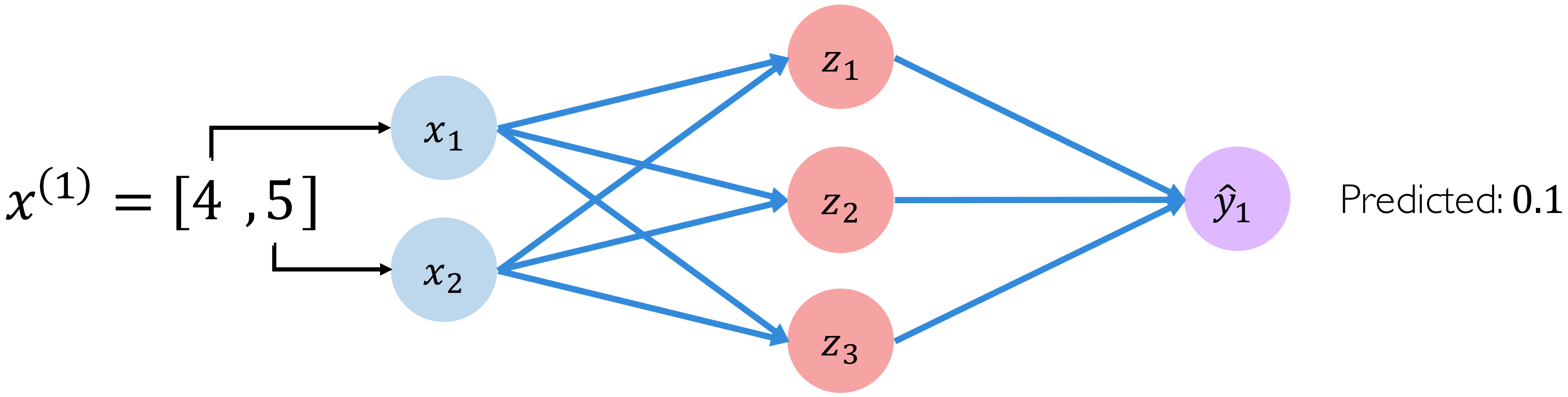
Example Problem: Will I pass this class?



Example Problem: Will I pass this class?

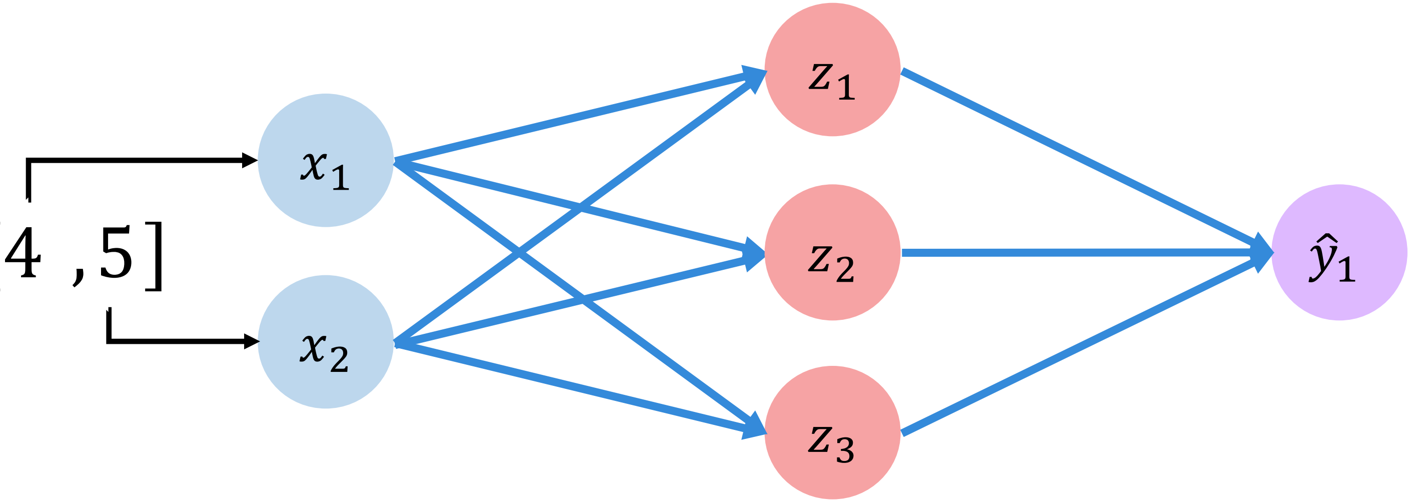


Example Problem: Will I pass this class?



Example Problem: Will I pass this class?

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

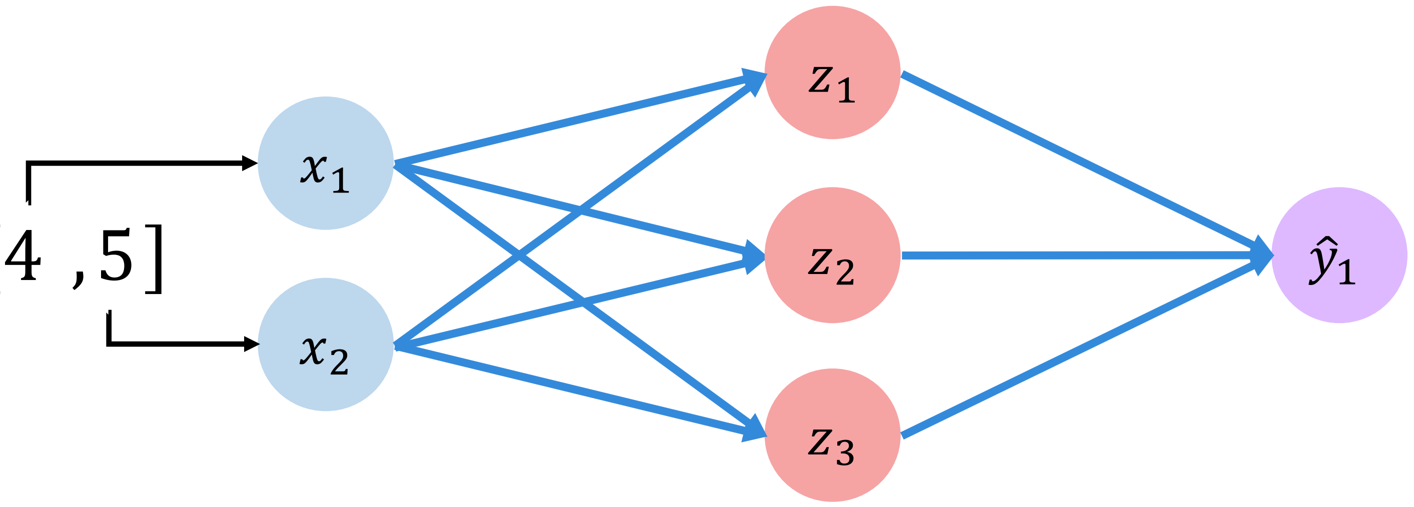


Predicted: **0.1**
Actual: **1**

Quantifying Loss

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

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

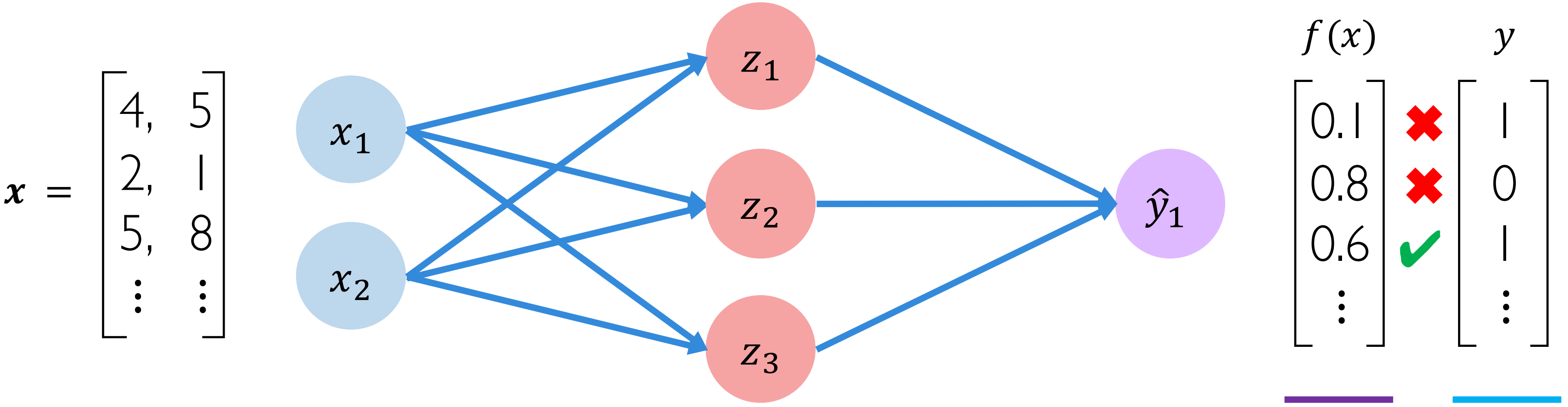


Predicted: **0.1**
Actual: **1**

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

Empirical Loss

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



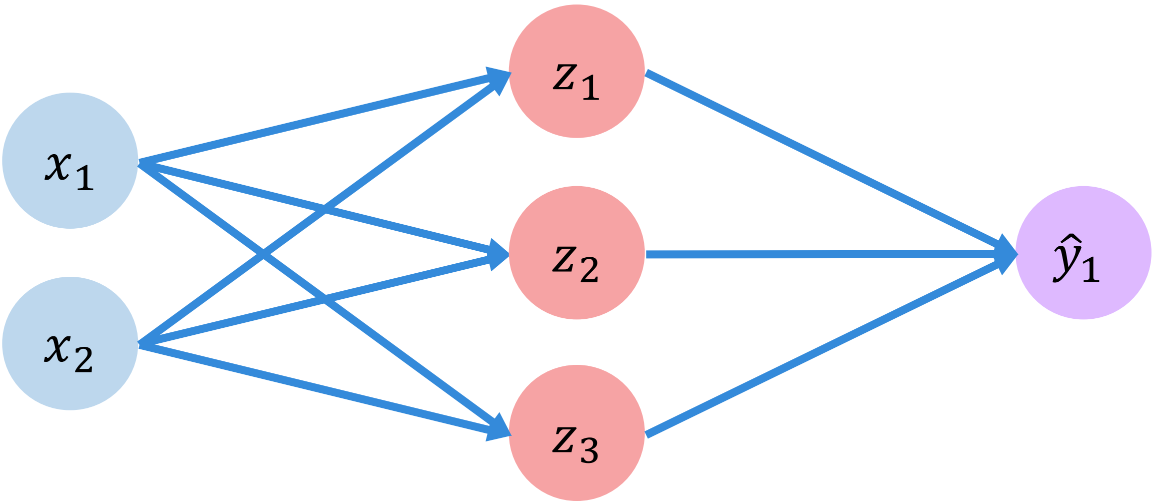
Also known as:
- Objective function
- Cost function
- Empirical Risk

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

Binary Cross Entropy Loss

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

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



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

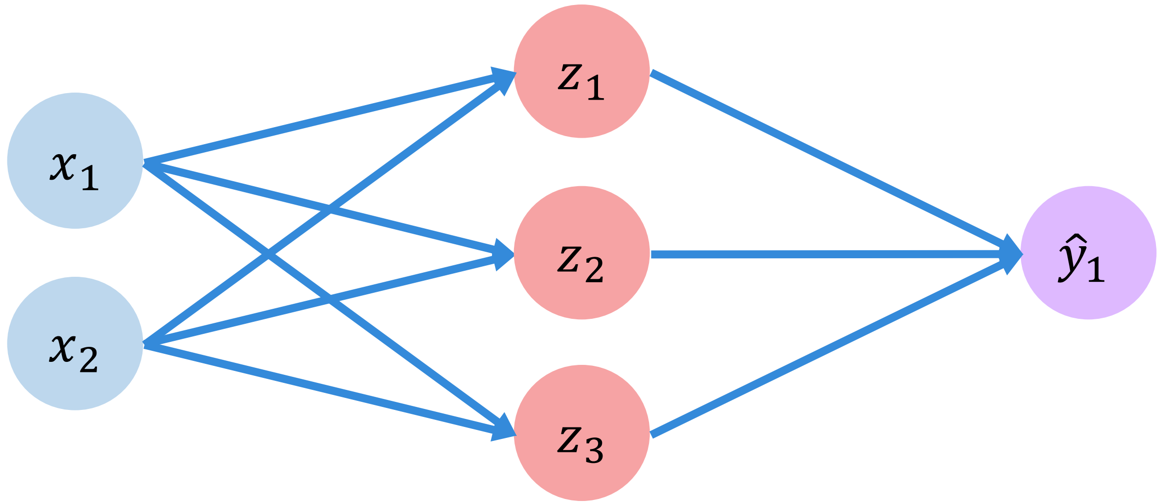
$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\theta\big)}_{\text{Predicted}}\Big) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\Big(1 - \underbrace{f\big(x^{(i)};\theta\big)}_{\text{Predicted}}\Big)$$

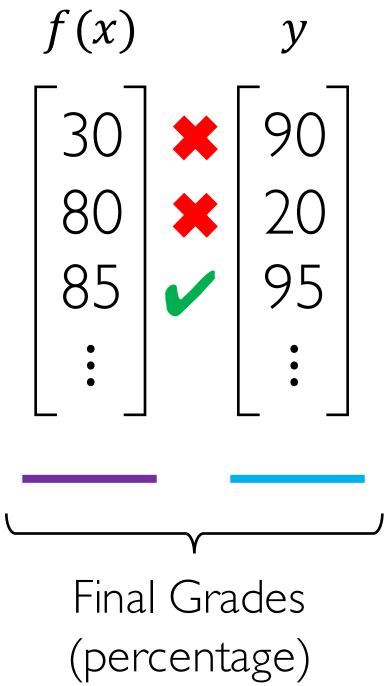
```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(model.y, model.pred) )
```

Mean Squared Error Loss

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

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





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

```
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***

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

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

Loss Optimization

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

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

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

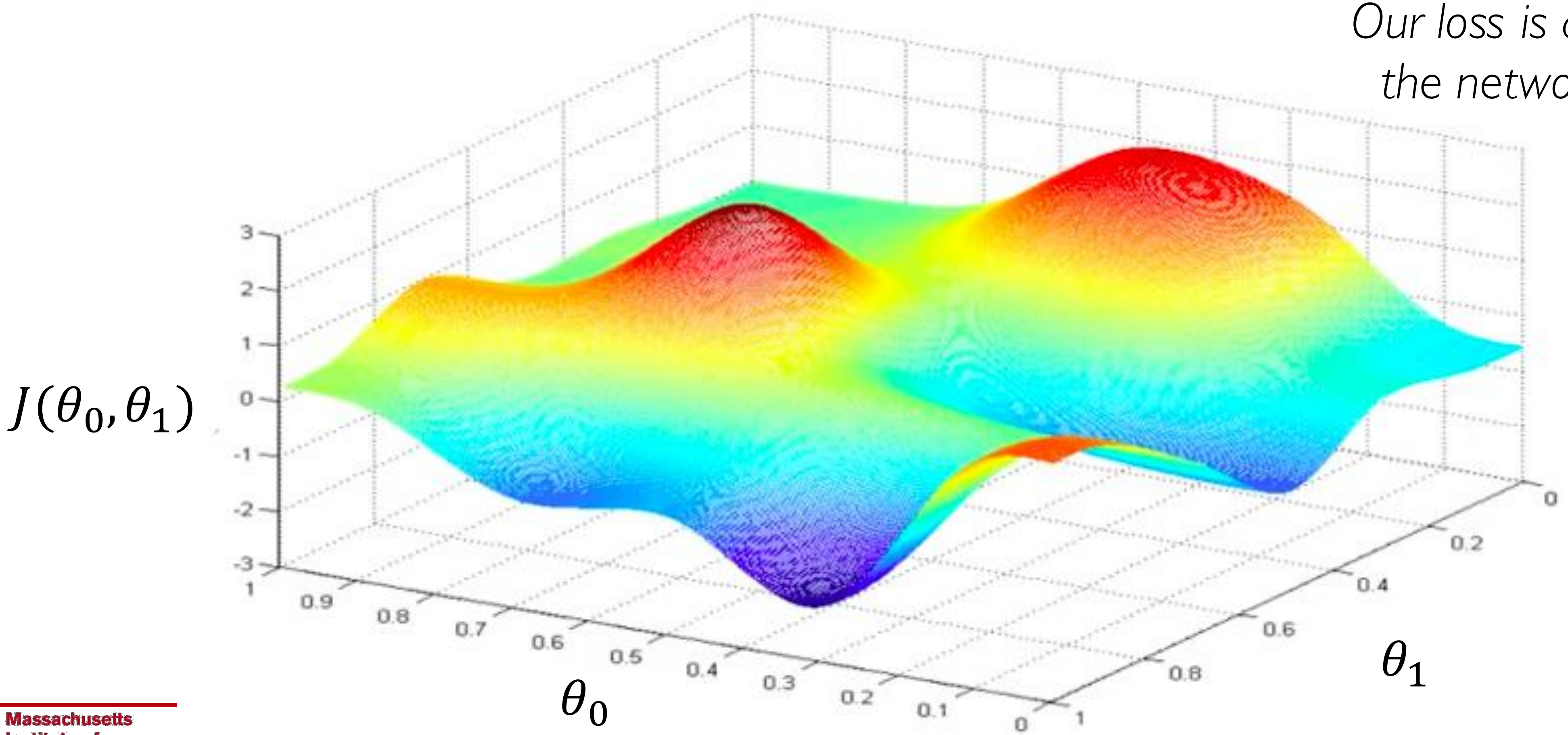
Remember:

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

Loss Optimization

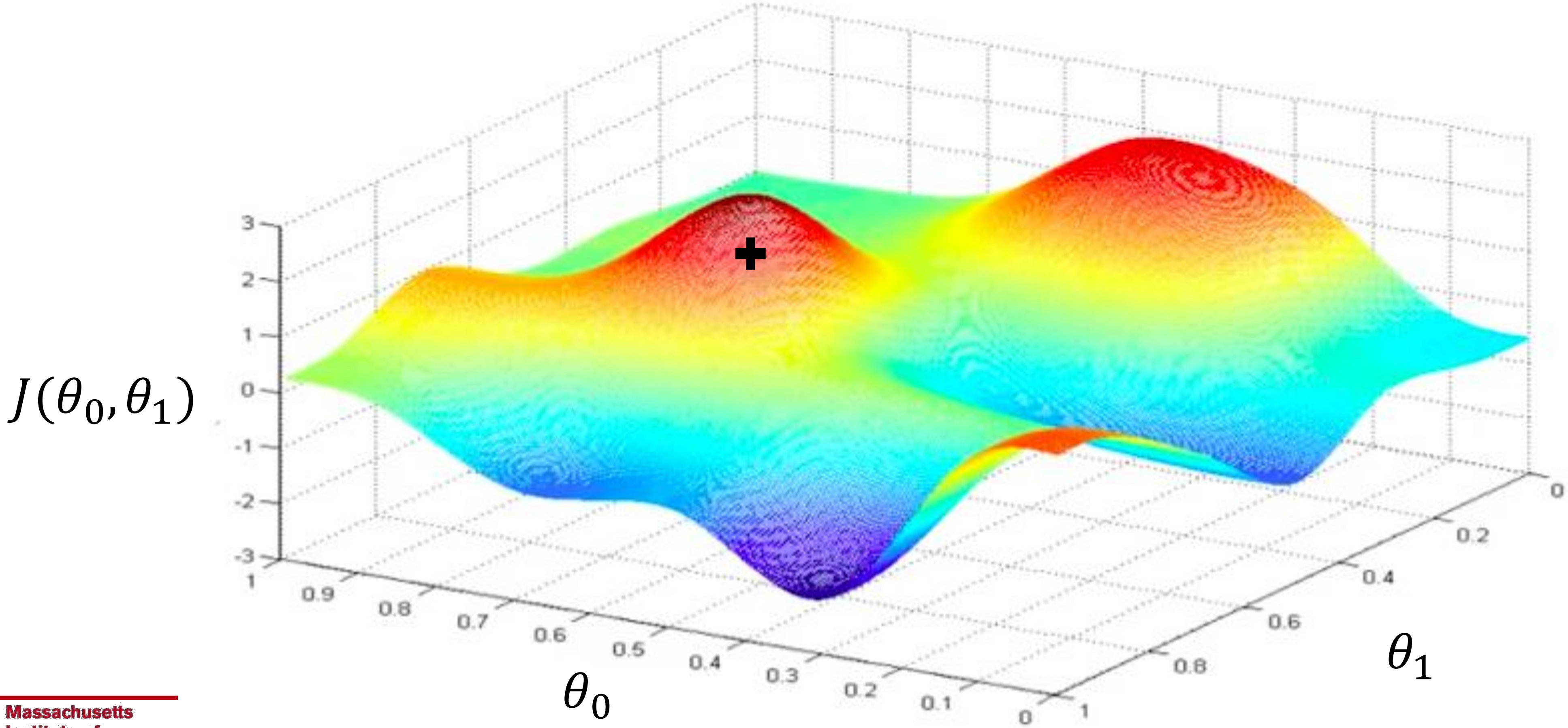
$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J(\boldsymbol{\theta})$$

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



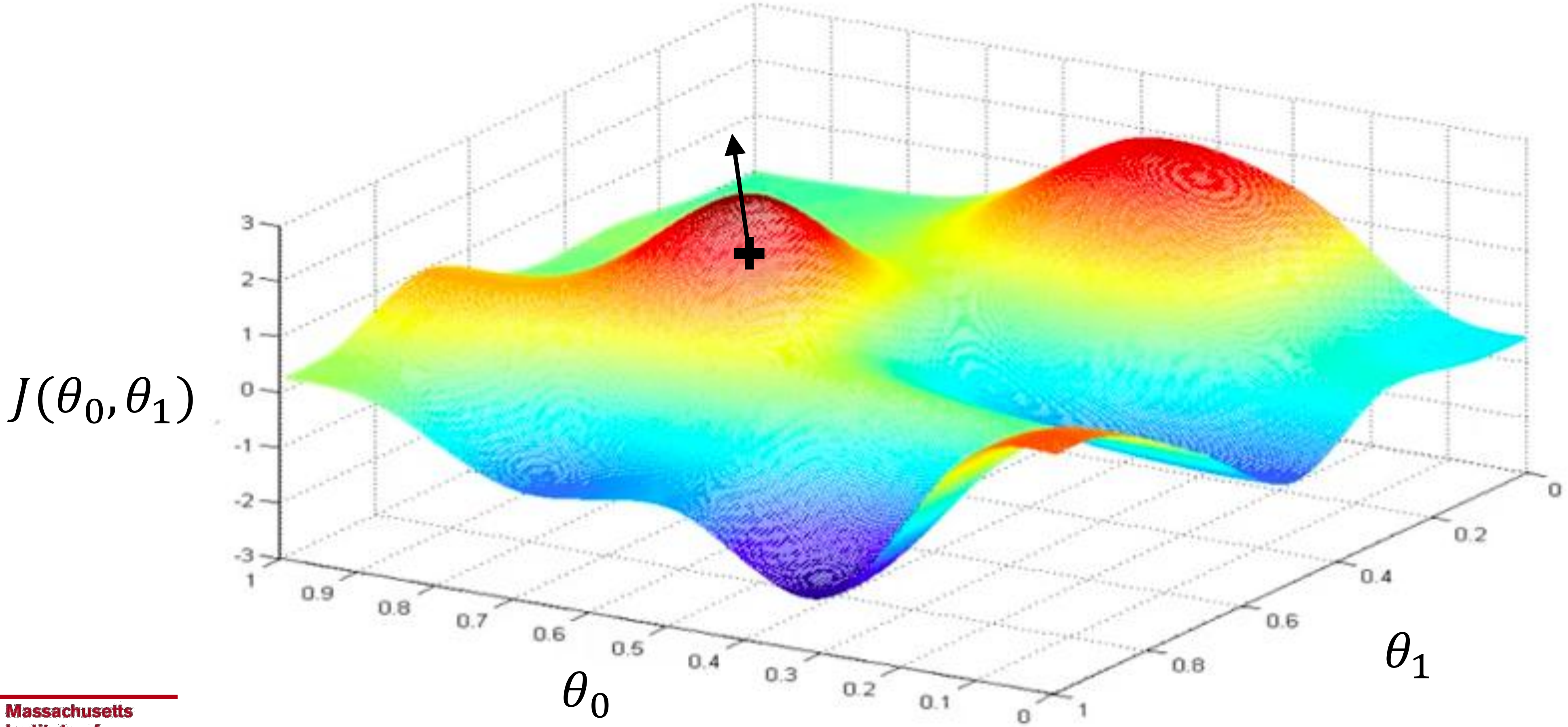
Loss Optimization

Randomly pick an initial (θ_0, θ_1)



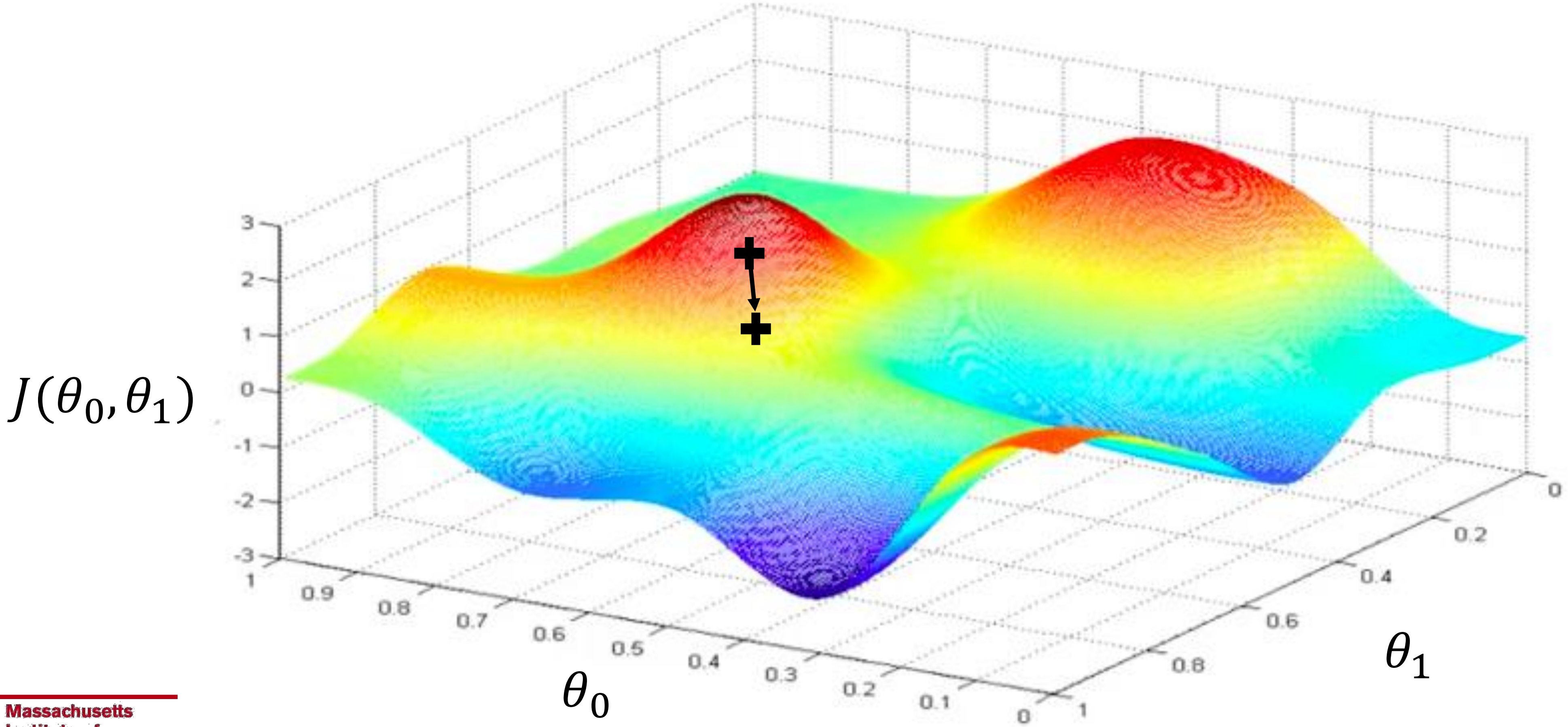
Loss Optimization

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



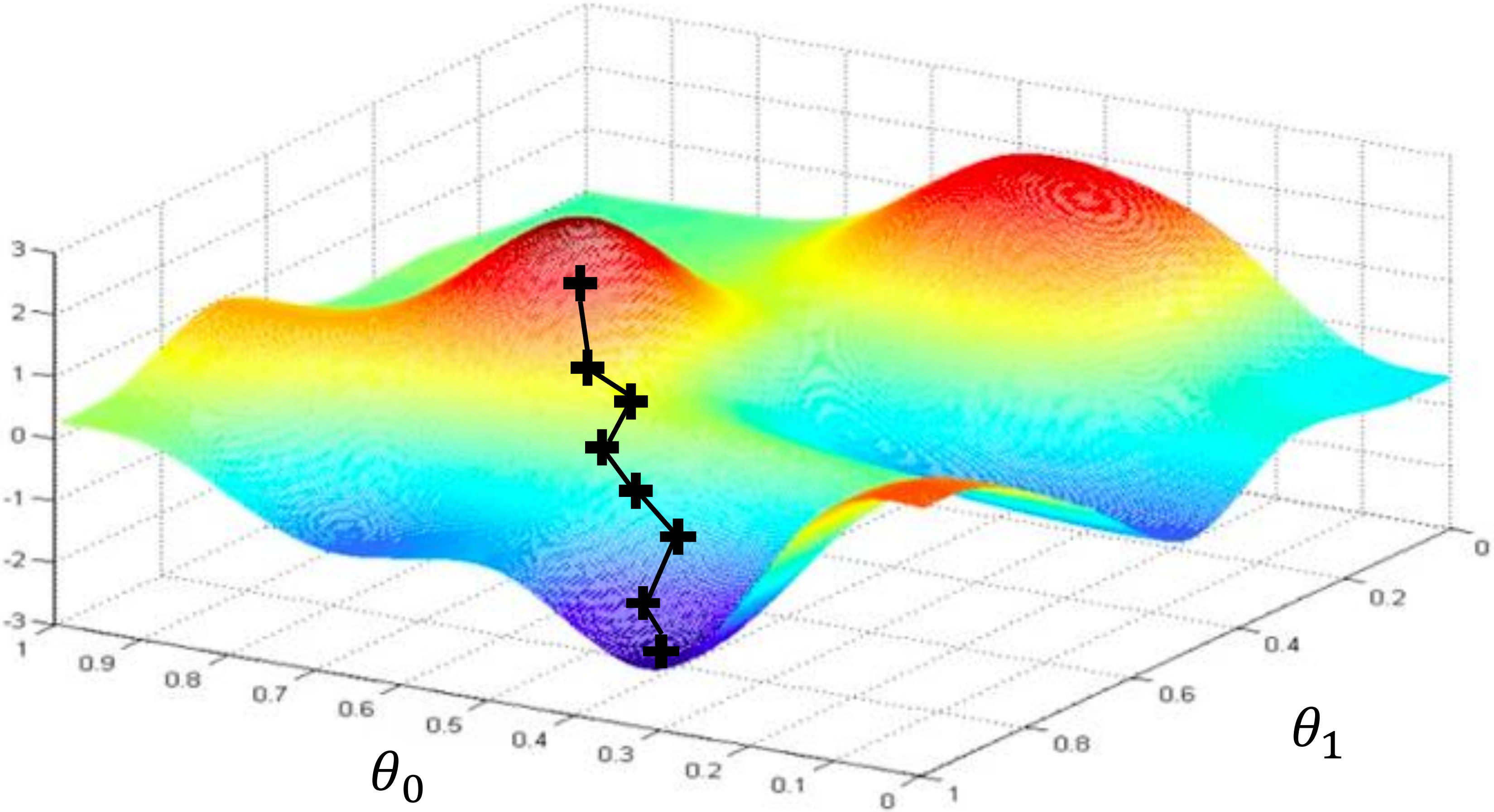
Loss Optimization

Take small step in opposite direction of gradient



Gradient Descent

Repeat until convergence



$J(\theta_0, \theta_1)$

θ_0

θ_1

Gradient Descent

Algorithm

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

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

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

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

Gradient Descent

Algorithm

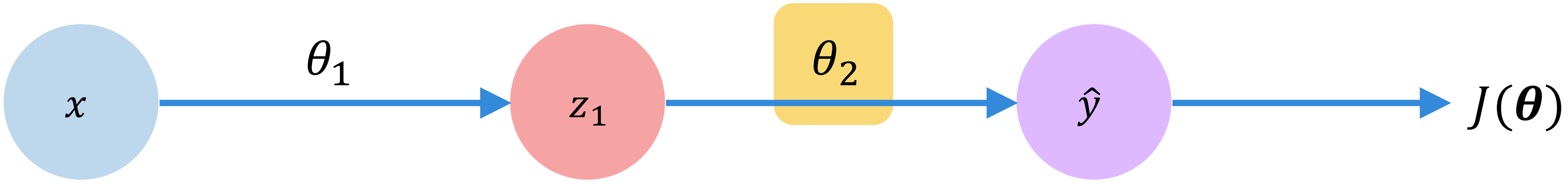
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights

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

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

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

Computing Gradients: Backpropagation



How does a small change in one weight (ex. $\boldsymbol{\theta_2}$) affect the final loss $\boldsymbol{J(\theta)}$?

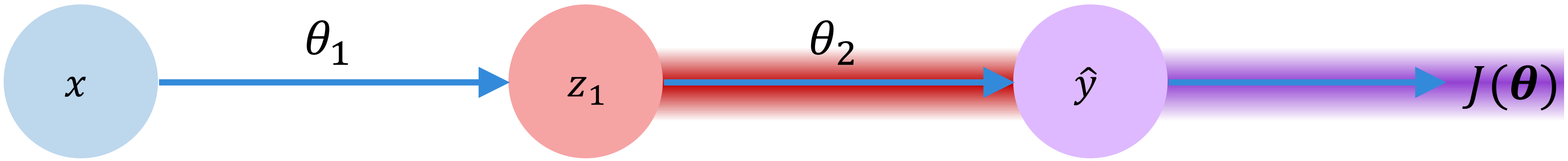
Computing Gradients: Backpropagation

$$x \xrightarrow{\theta_1} z_1 \xrightarrow{\boxed{\theta_2}} \hat{y} \longrightarrow J(\boldsymbol{\theta})$$

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boxed{\theta_2}} =$$

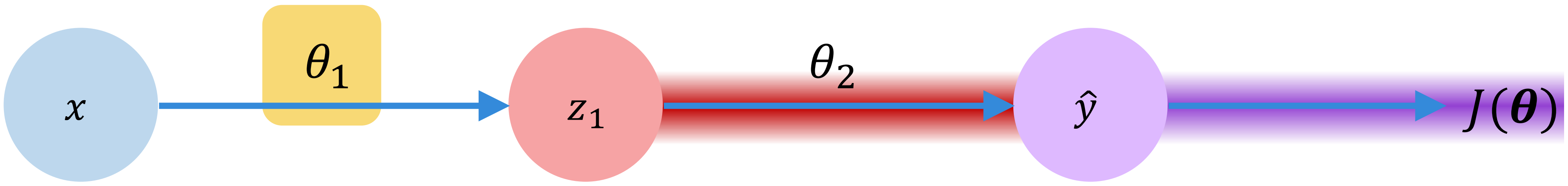
Let's use the chain rule!

Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_2} = \underbrace{\frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}}} * \underbrace{\frac{\partial \hat{y}}{\partial \theta_2}}$$

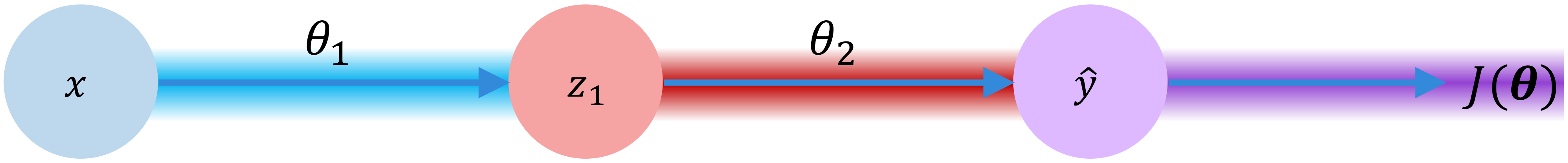
Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} = \underbrace{\frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}}}_{} * \underbrace{\frac{\partial \hat{y}}{\partial \theta_1}}_{}$$

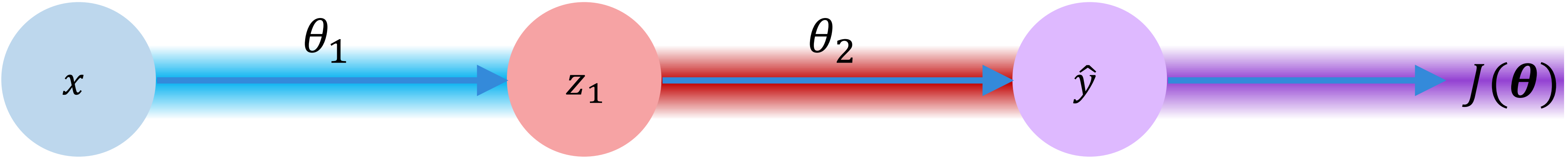
Apply chain rule! Apply chain rule!

Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} = \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial \theta_1}$$

Computing Gradients: Backpropagation

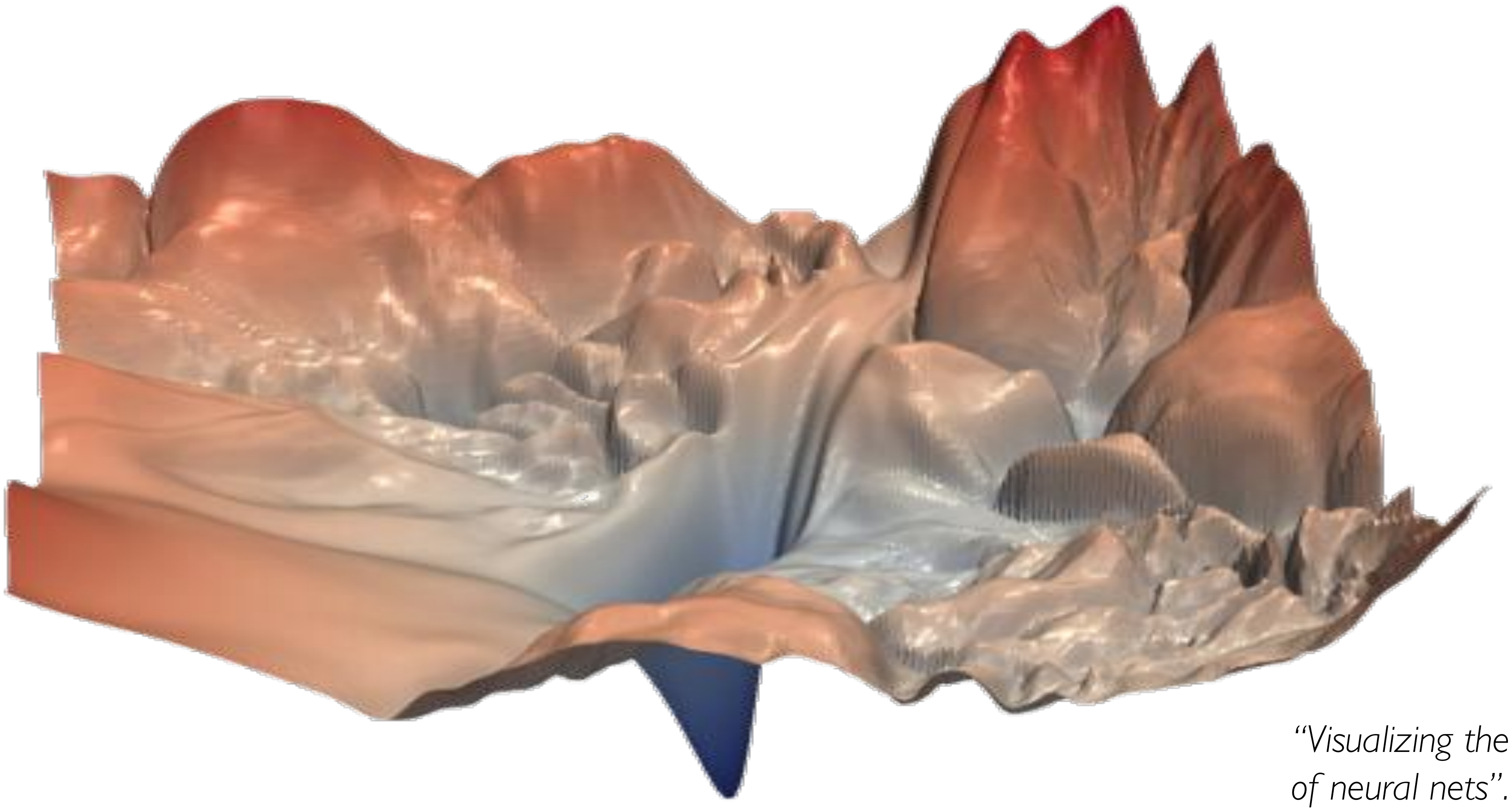


$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} = \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial \theta_1}$$

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

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

Loss Functions Can Be Difficult to Optimize

Remember:

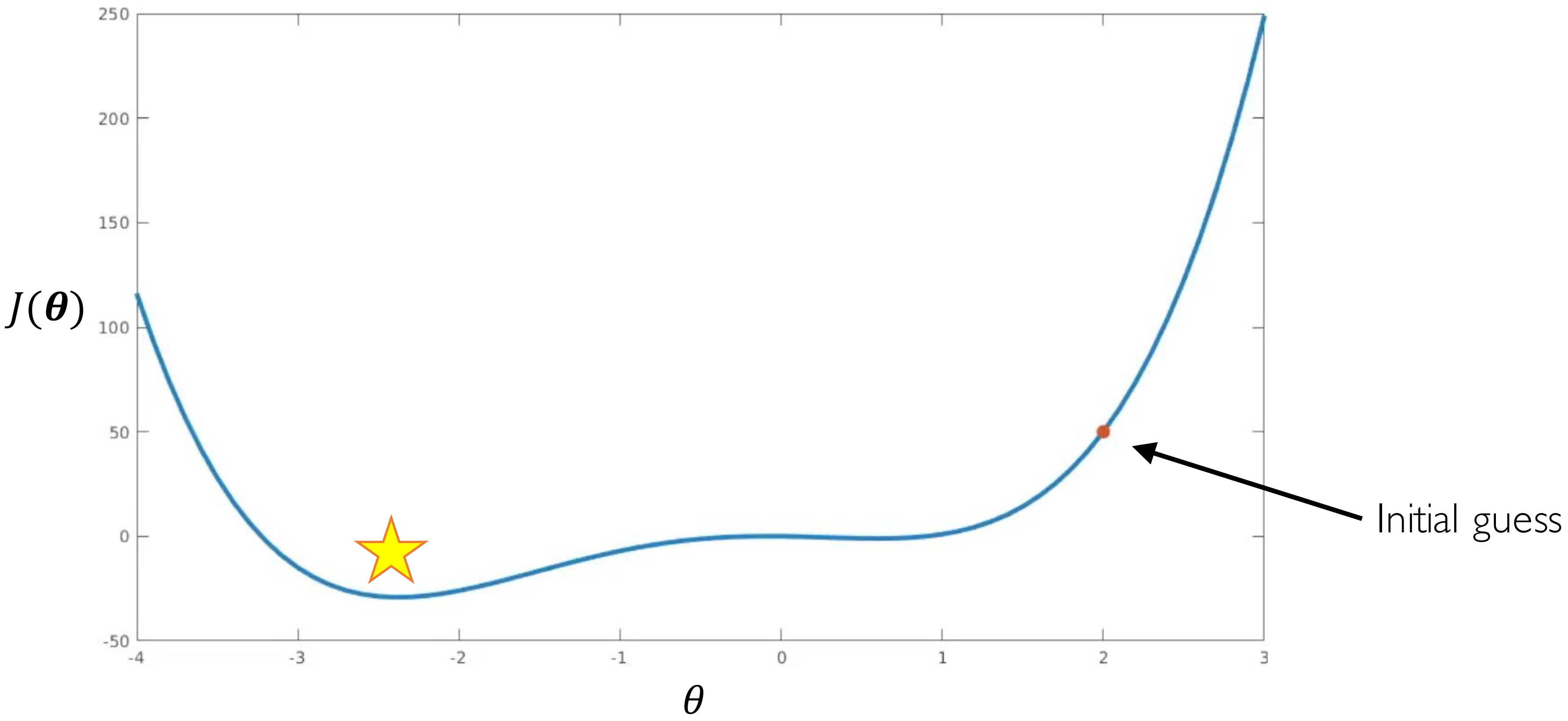
Optimization through gradient descent

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

How can we set the learning rate?

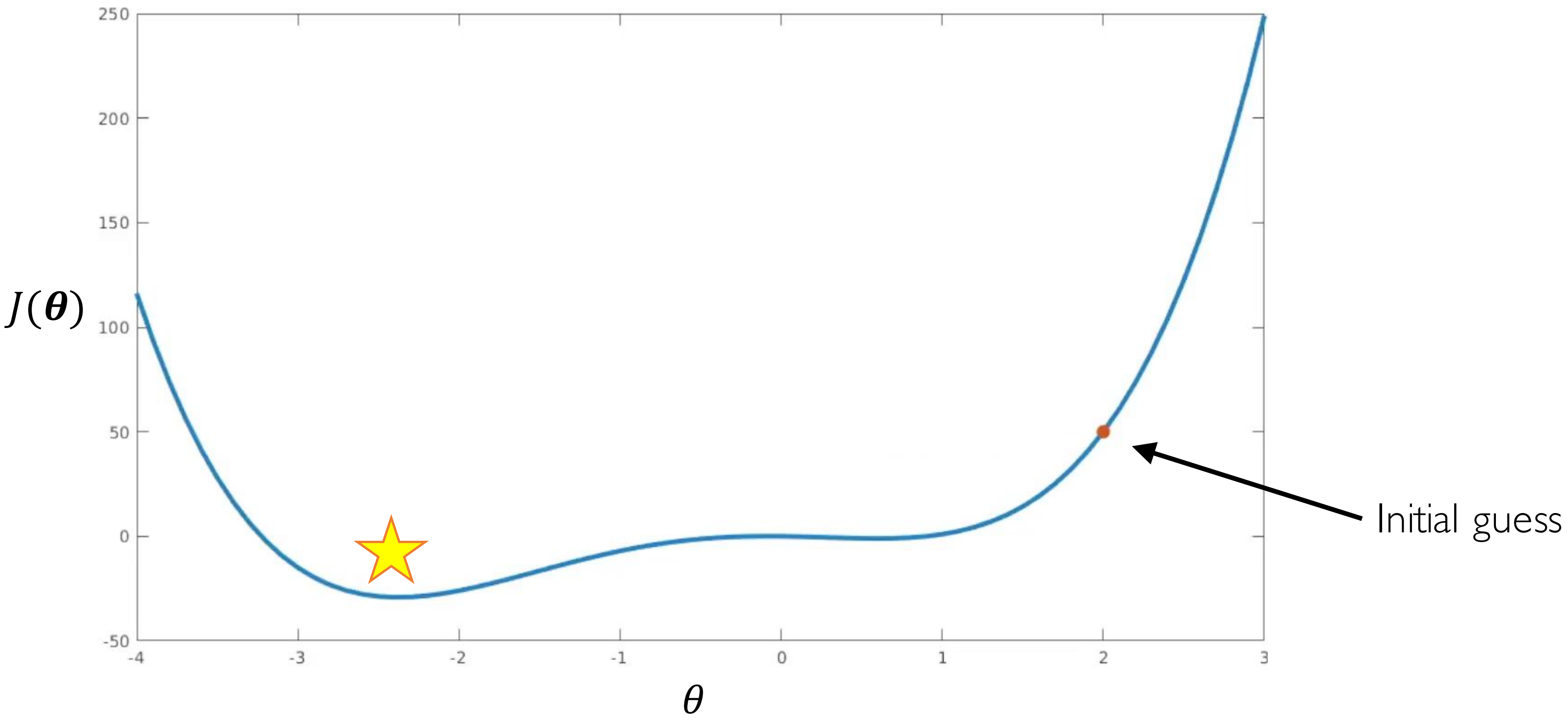
Setting the Learning Rate

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



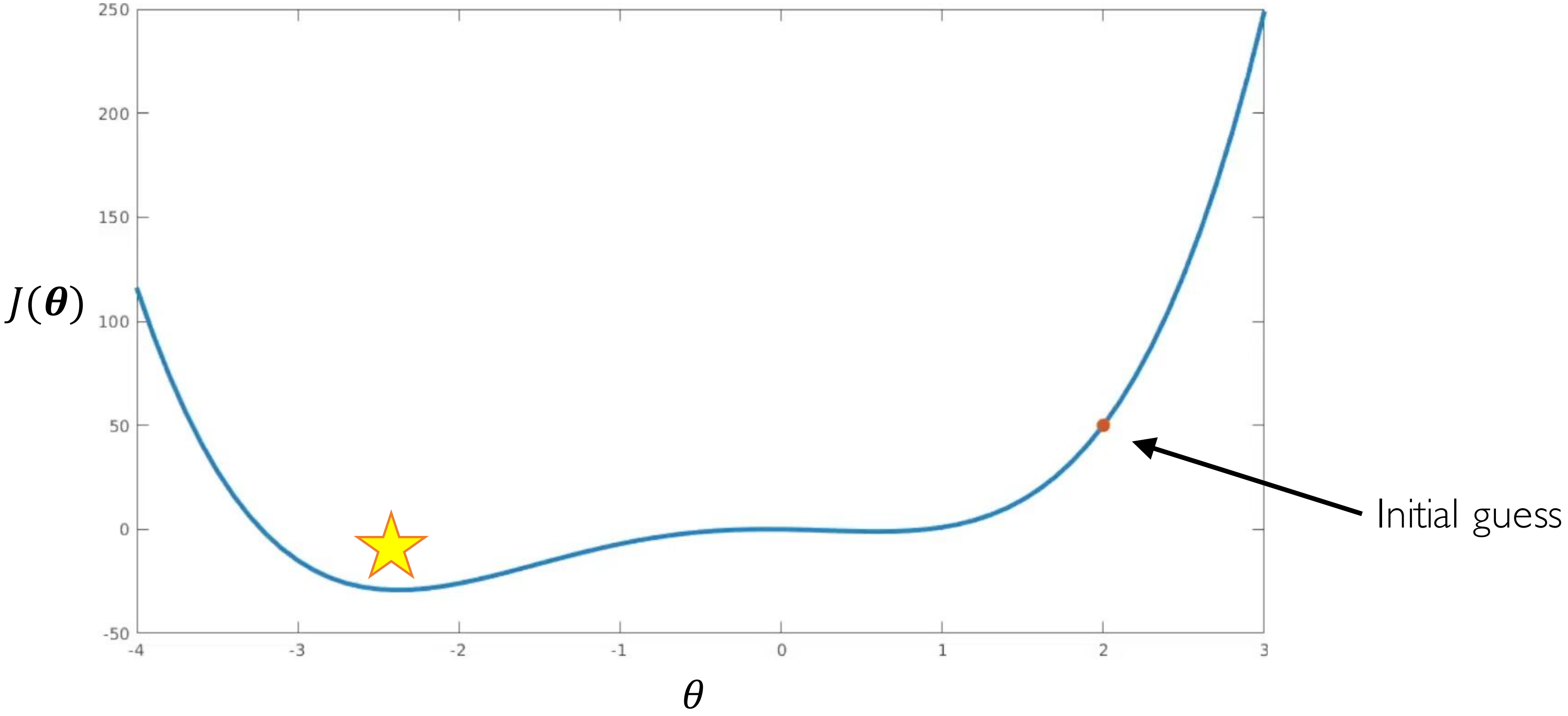
Setting the Learning Rate

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.

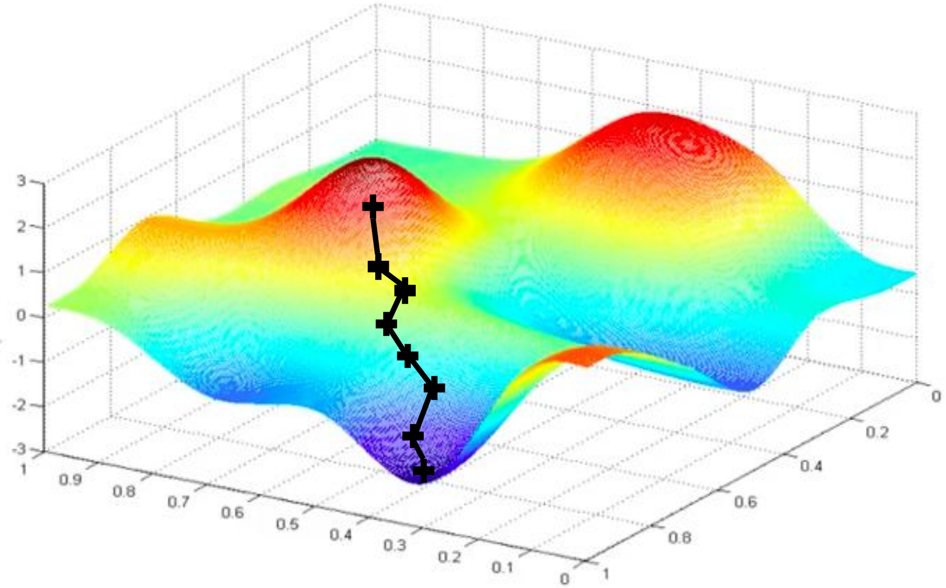
Additional details: http://ruder.io/optimizing-gradient-descent/

Neural Networks in Practice:
Mini-batches

Gradient Descent

Algorithm

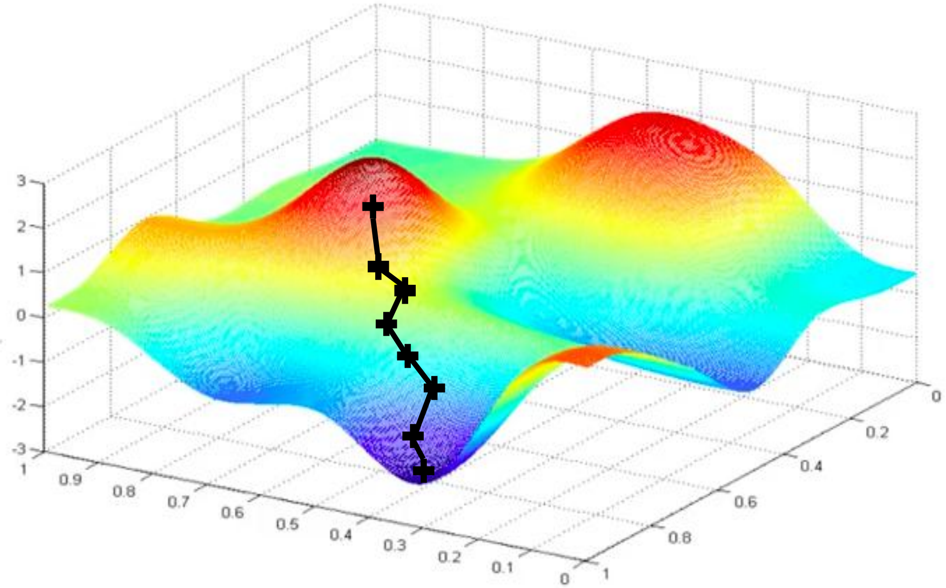
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights



Gradient Descent

Algorithm

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

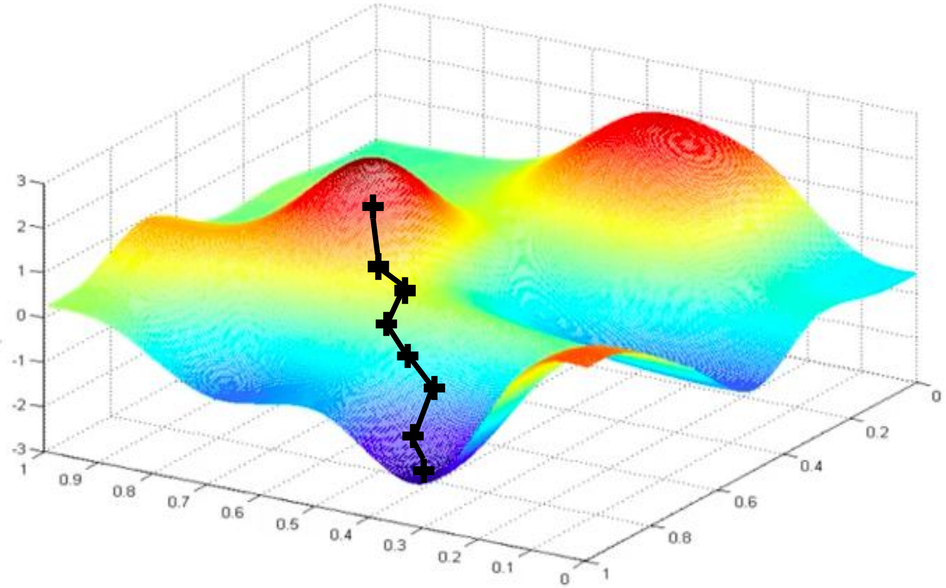


Can be very computational to compute!

Stochastic Gradient Descent

Algorithm

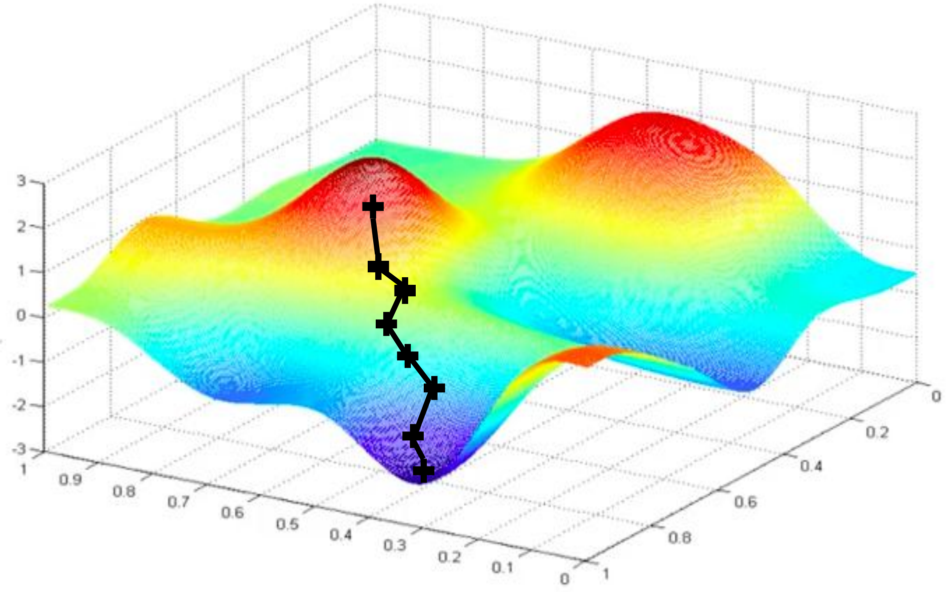
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(\theta)}{\partial \theta}$
5. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
6. Return weights



Stochastic Gradient Descent

Algorithm

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

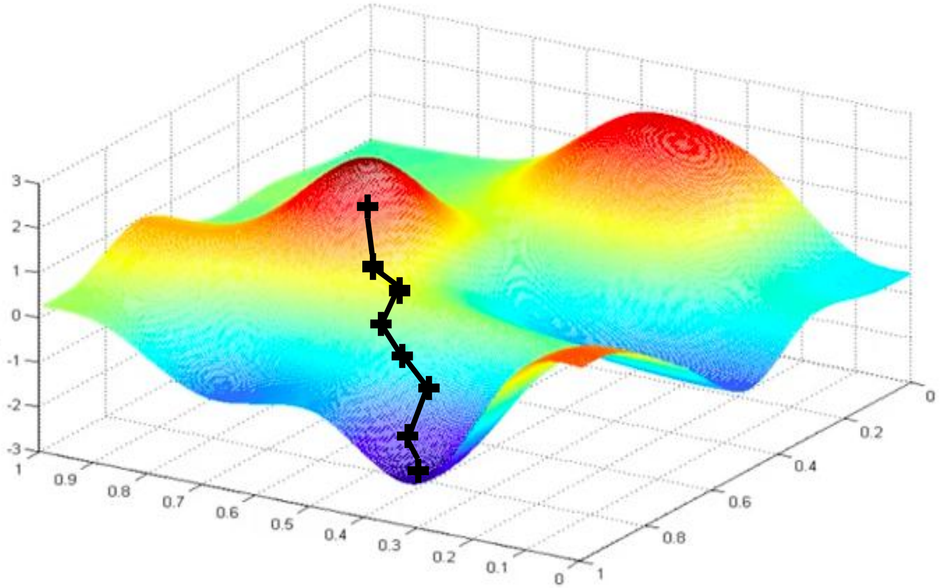


Easy to compute but
very noisy
(stochastic)!

Stochastic Gradient Descent

Algorithm

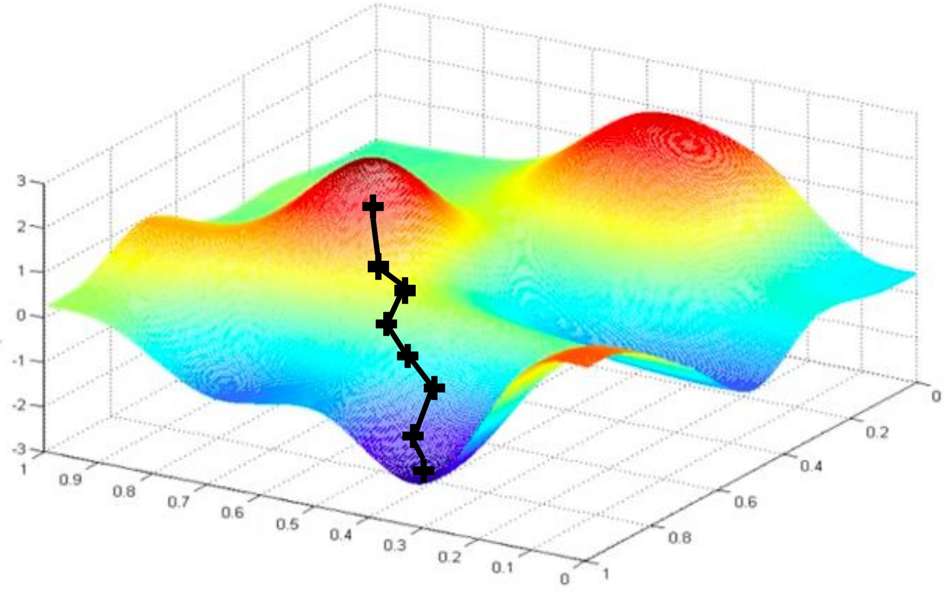
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{B}\sum_{k=1}^{B}\frac{\partial J_k(\theta)}{\partial \theta}$
5. Update weights, $\theta \leftarrow \theta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

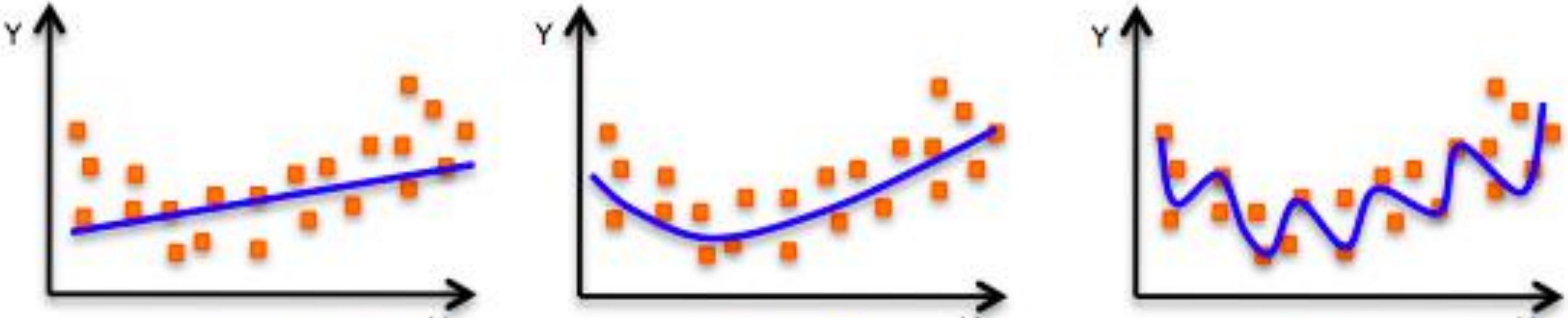
Allows for larger learning rates

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?

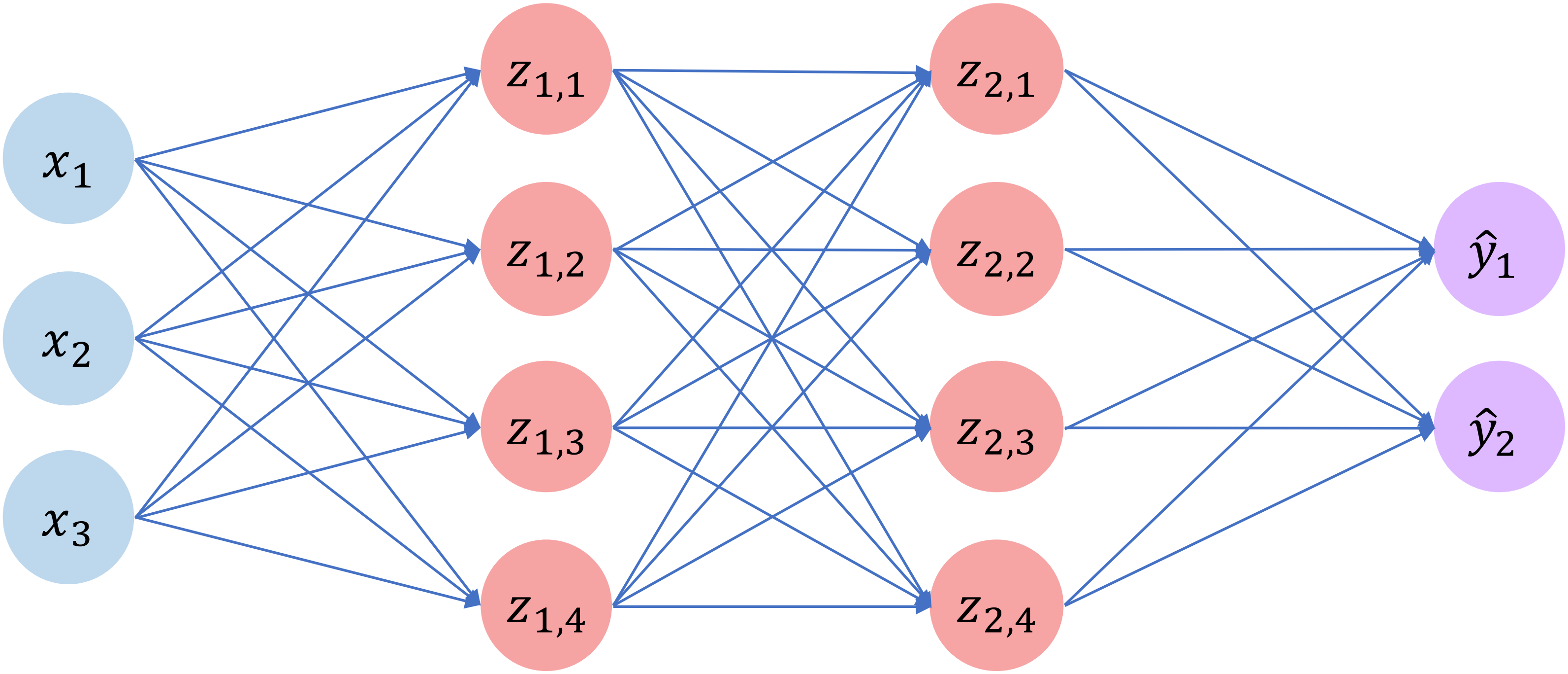
Technique that constrains our optimization problem to discourage complex models

Why do we need it?

Improve generalization of our model on unseen data

Regularization 1: Dropout

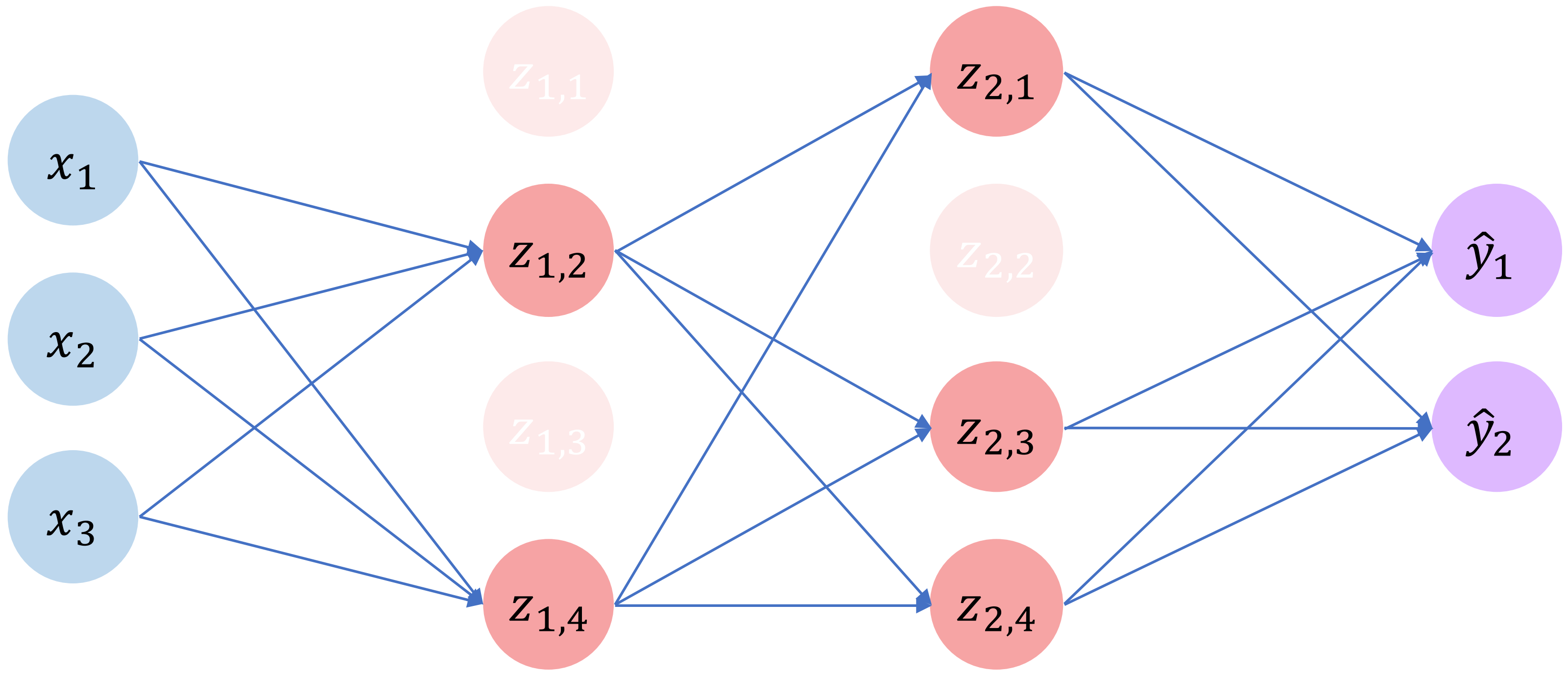
- During training, randomly set some activations to 0



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

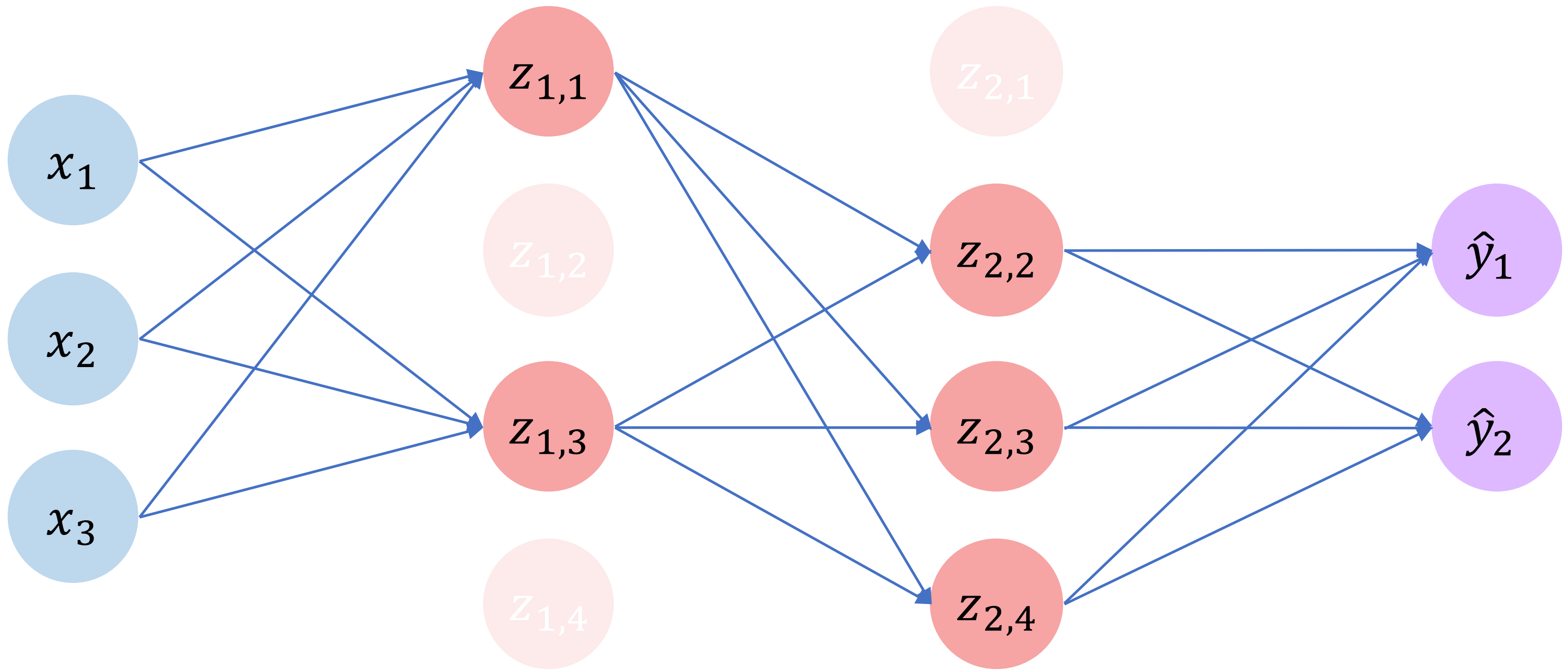
```
tf.nn.dropout(hiddenLayer, p=0.5)
```



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.nn.dropout(hiddenLayer, p=0.5)
```



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

Regularization 2: Early Stopping

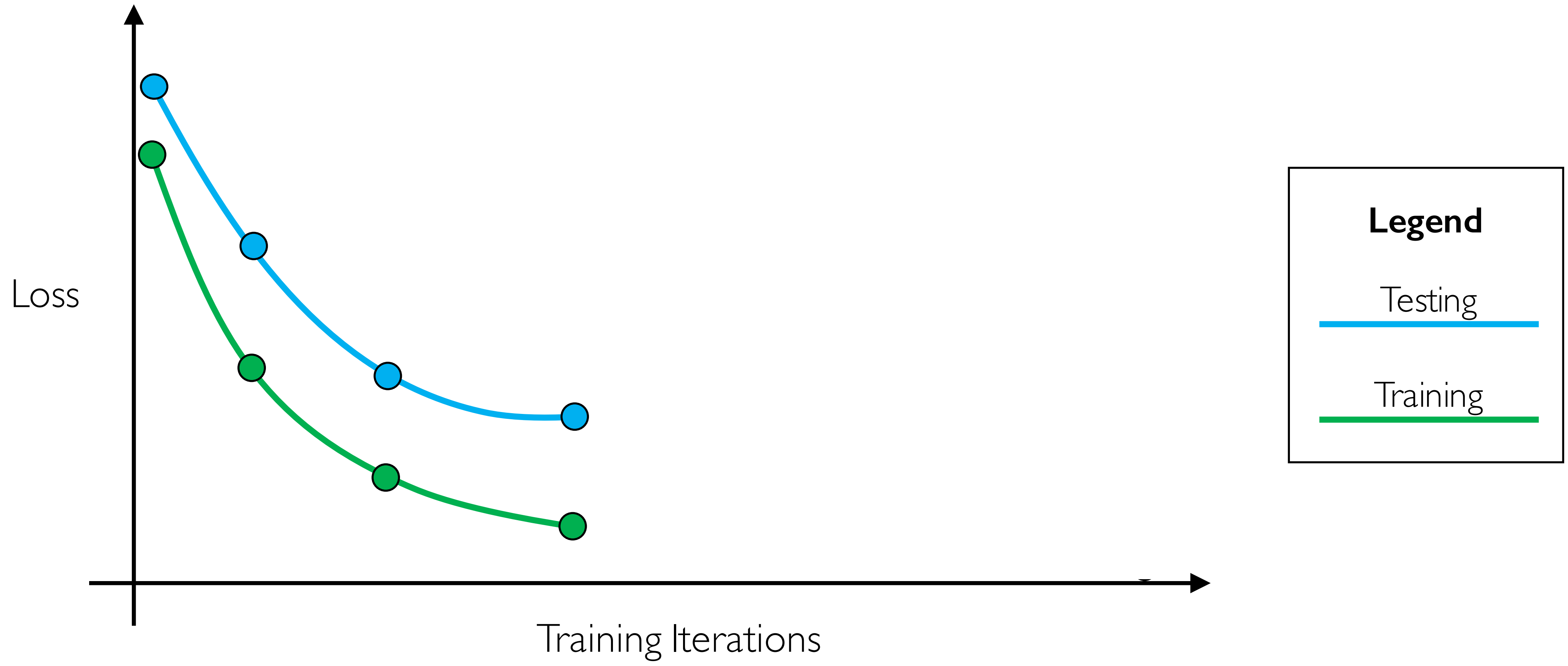
- Stop training before we have a chance to overfit

Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

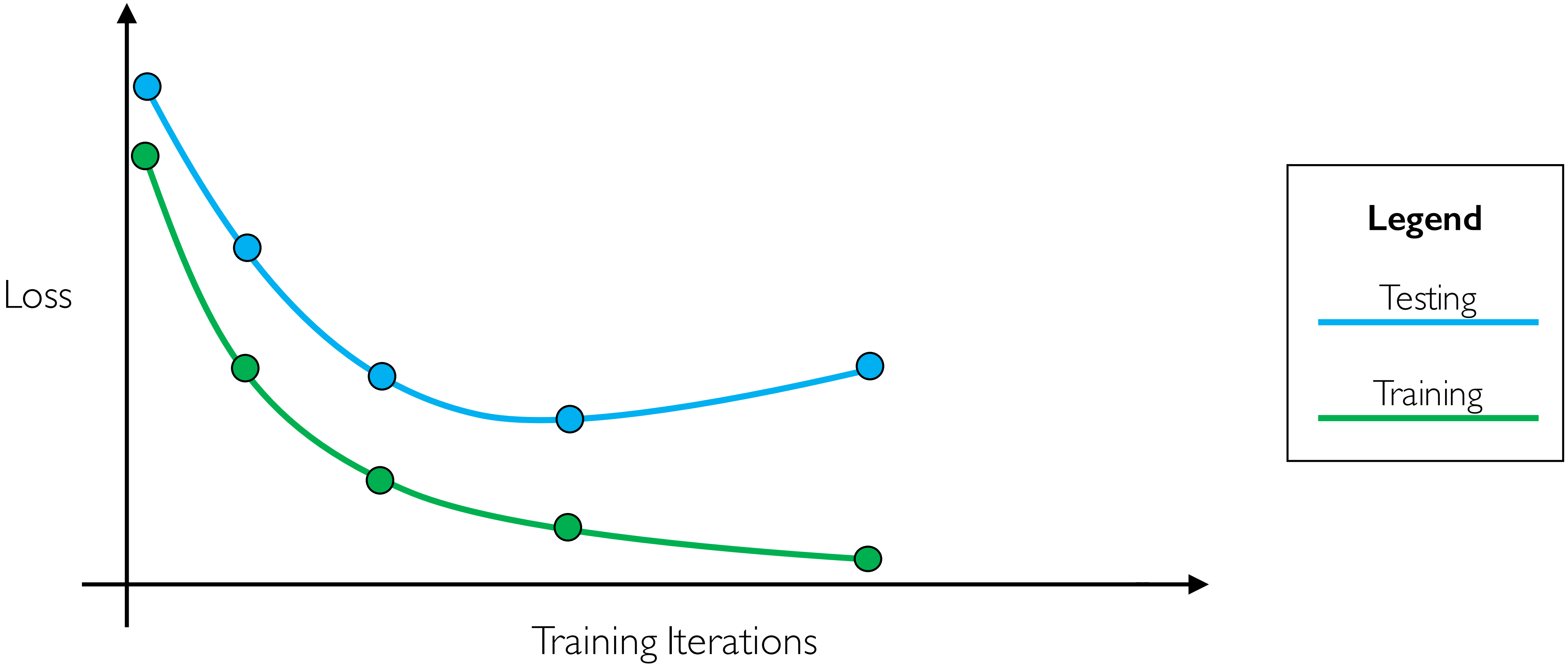
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



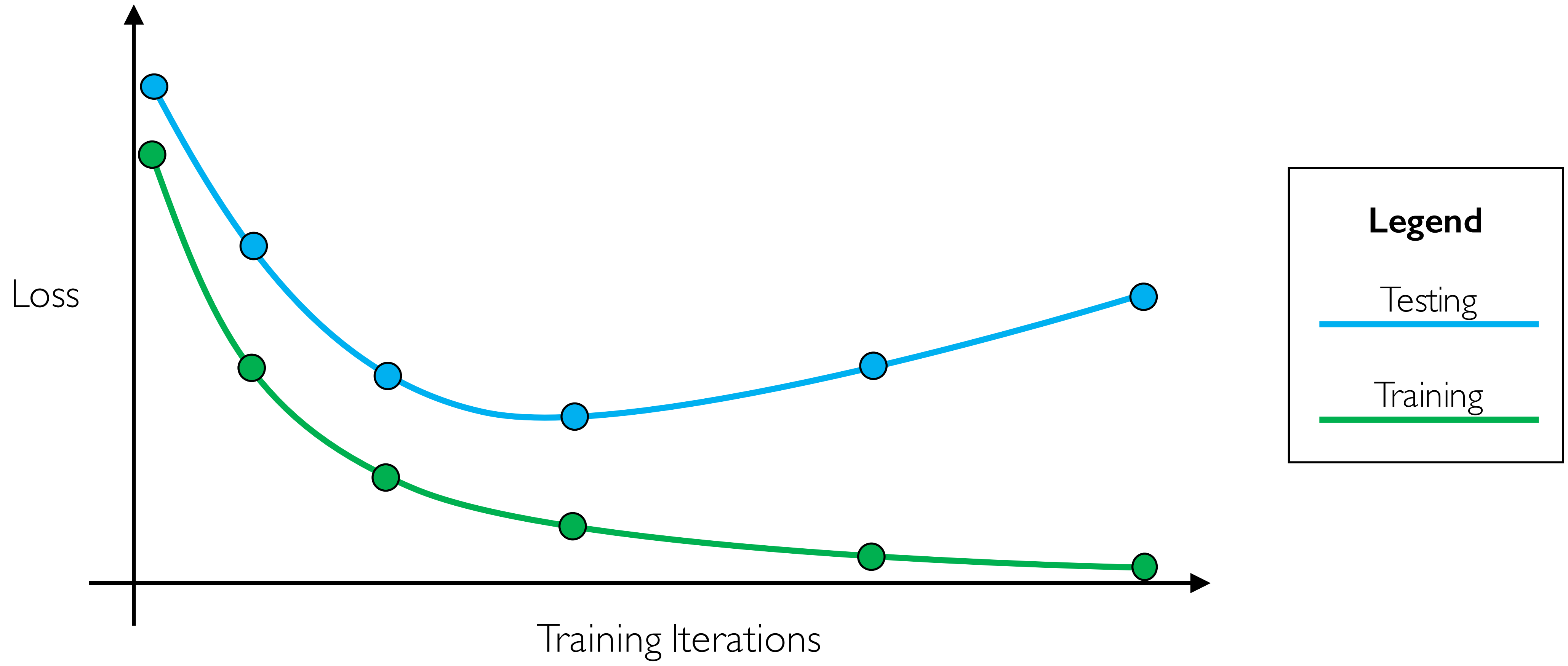
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



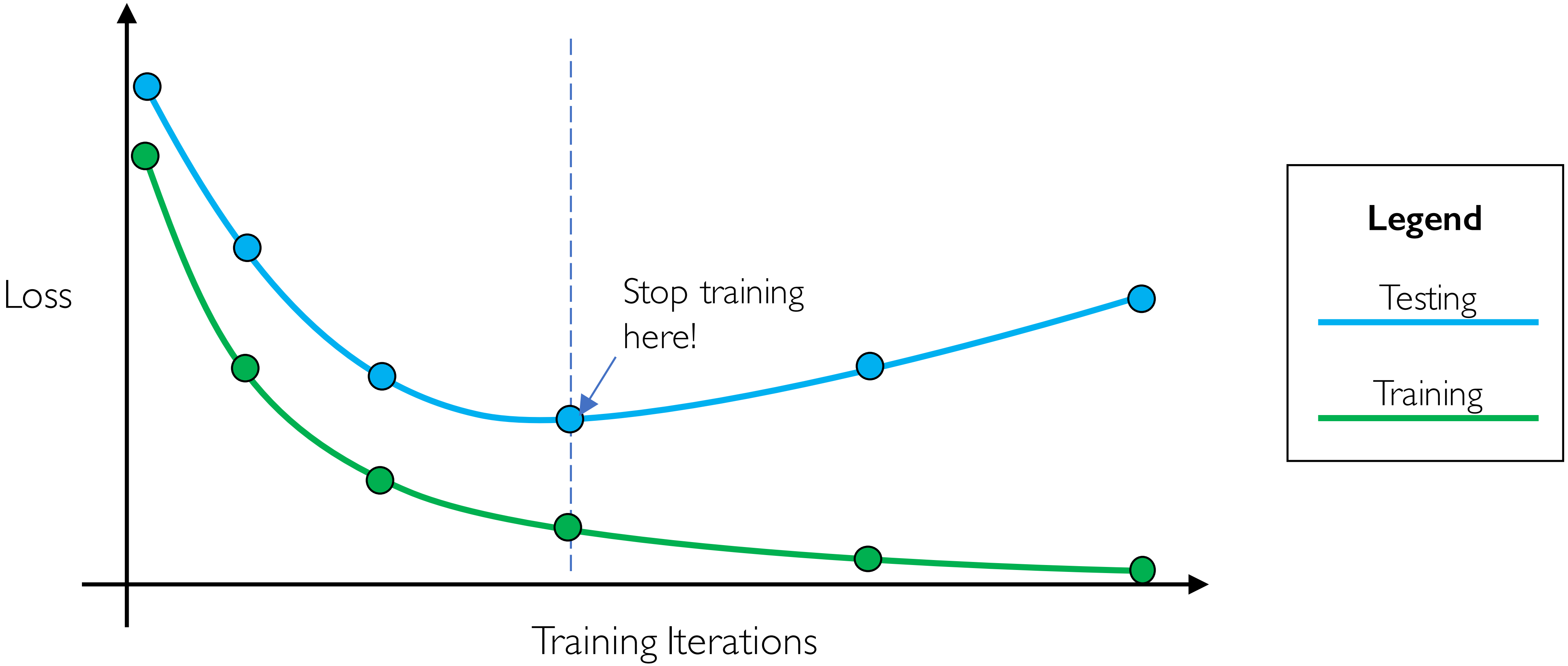
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



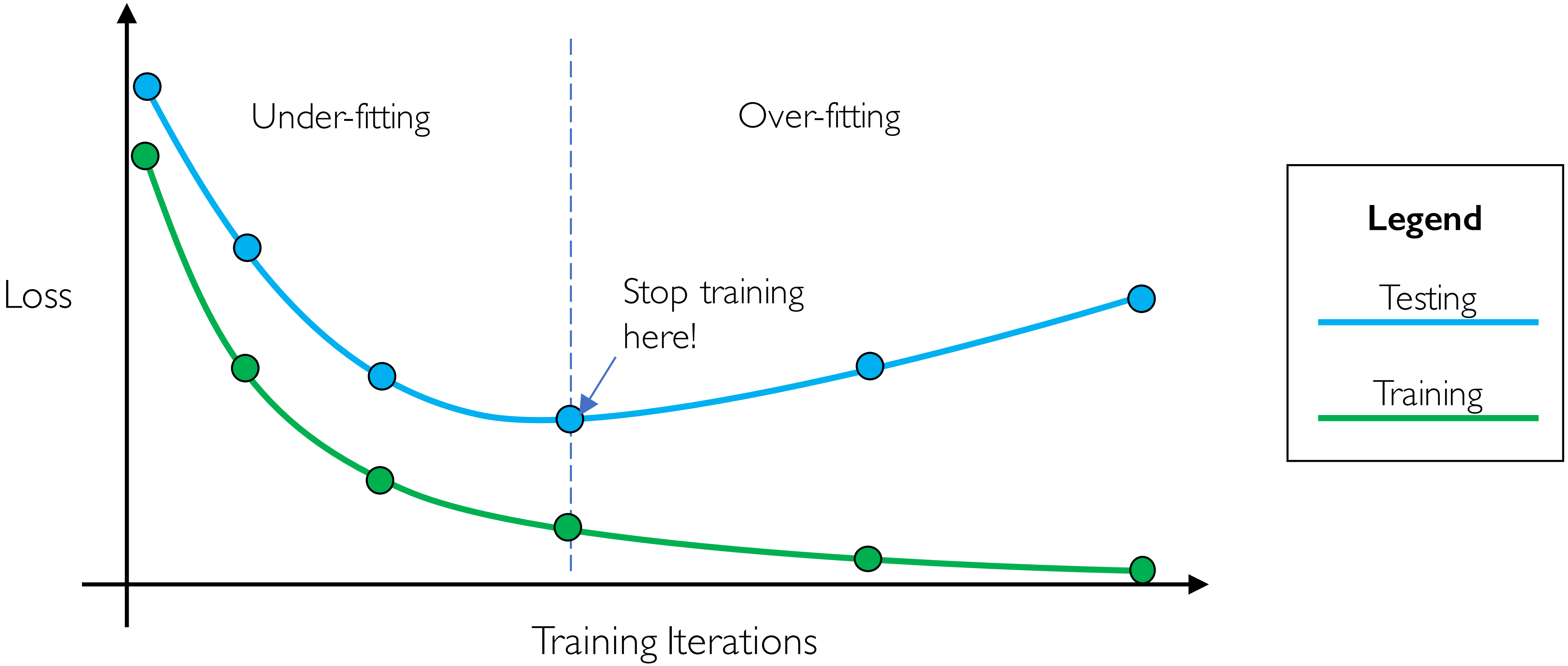
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



Regularization 2: Early Stopping

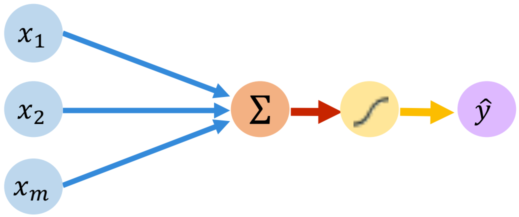
- Stop training before we have a chance to overfit



Core Foundation Review

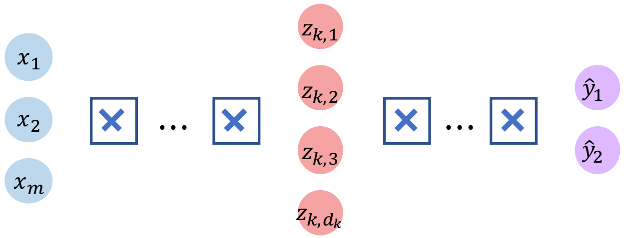
The Perceptron

- Structural building blocks
- Nonlinear activation functions



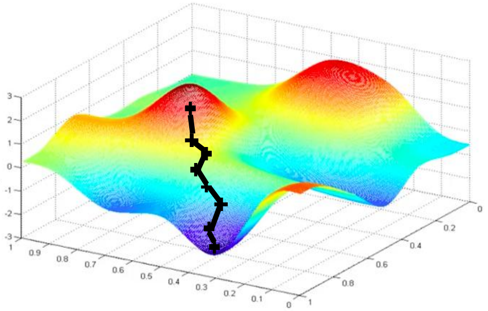
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization



Questions?