



Introduction to Deep Learning

MIT 6.S191

Alexander Amini



What is Deep Learning?

ARTIFICIAL INTELLIGENCE

Any technique that enables computers to mimic human behavior



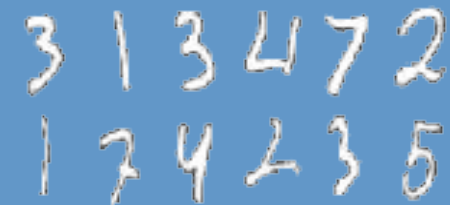
MACHINE LEARNING

Ability to learn without explicitly being programmed



DEEP LEARNING

Learn underlying features in data using neural networks



Deep Learning Success: Vision

Image Recognition

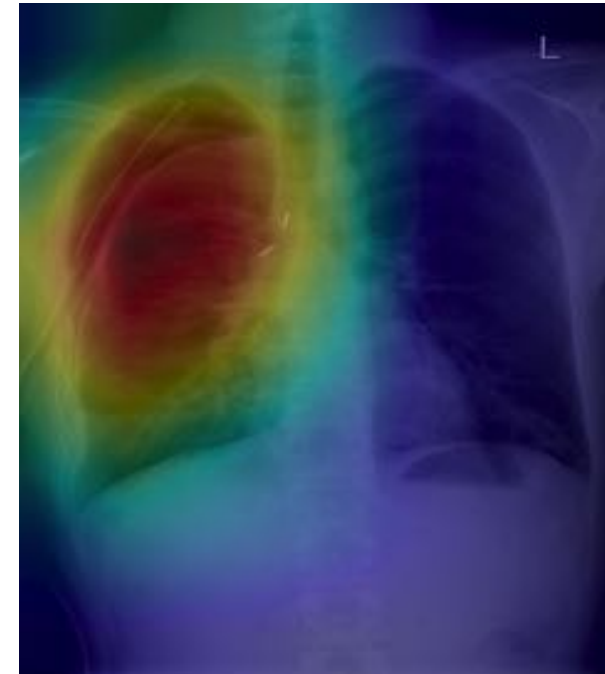
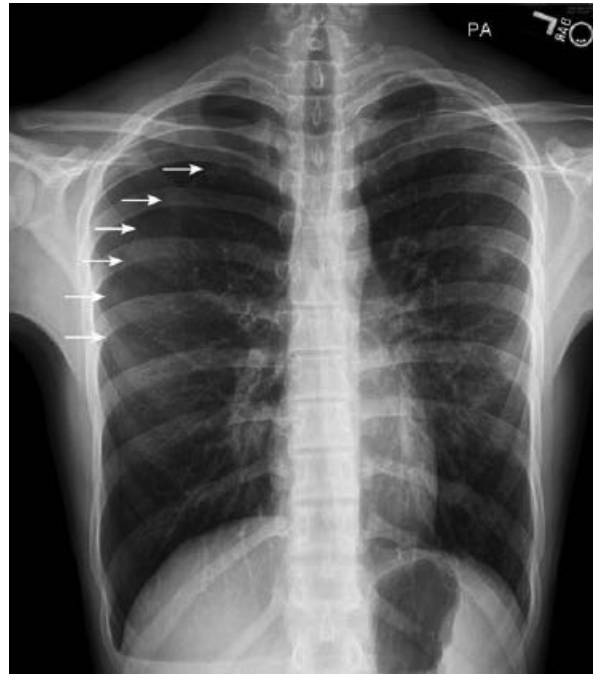
IMAGENET



Deep Learning Success: Vision



Detect pneumothorax in real X-Ray scans



Deep Learning Success: Audio



Other sequences-model applications:

- predict stock price
- machine translation
- ...

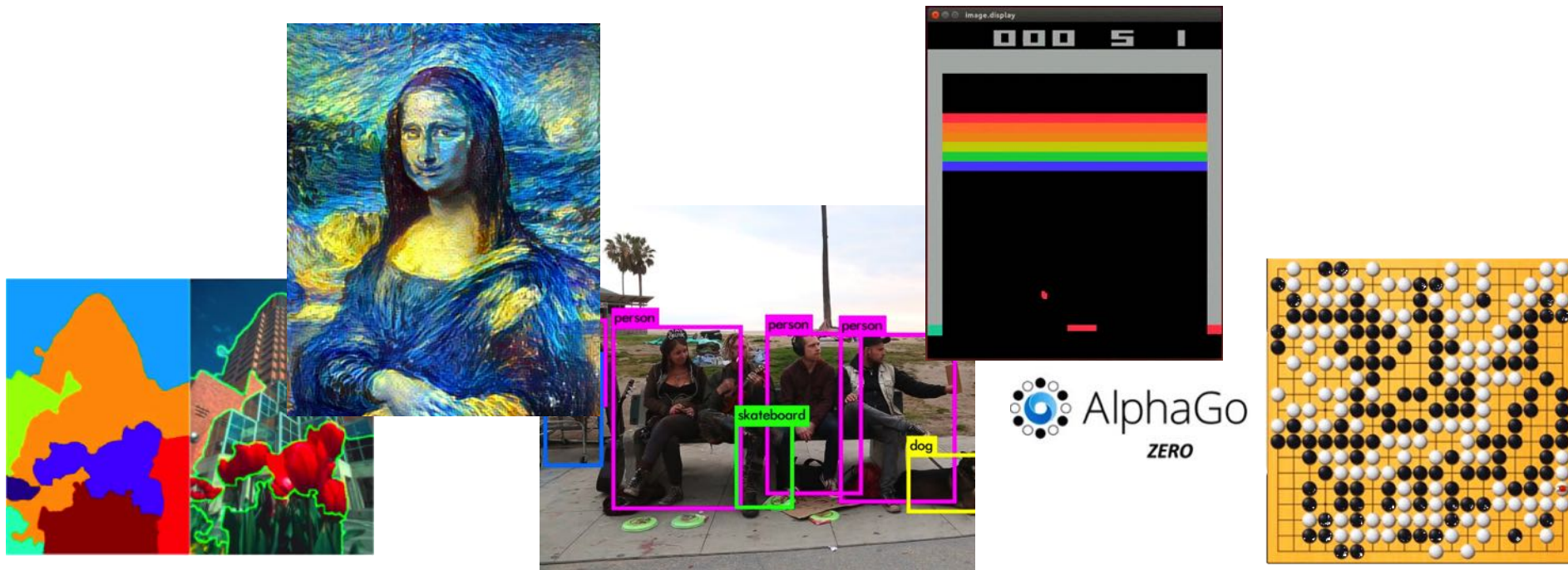
Music Generation

Temporal dependence



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

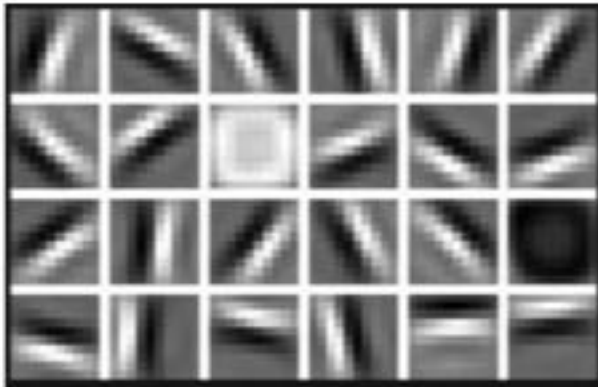
Why Deep Learning and Why Now?

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



Lines & Edges

Mid Level Features



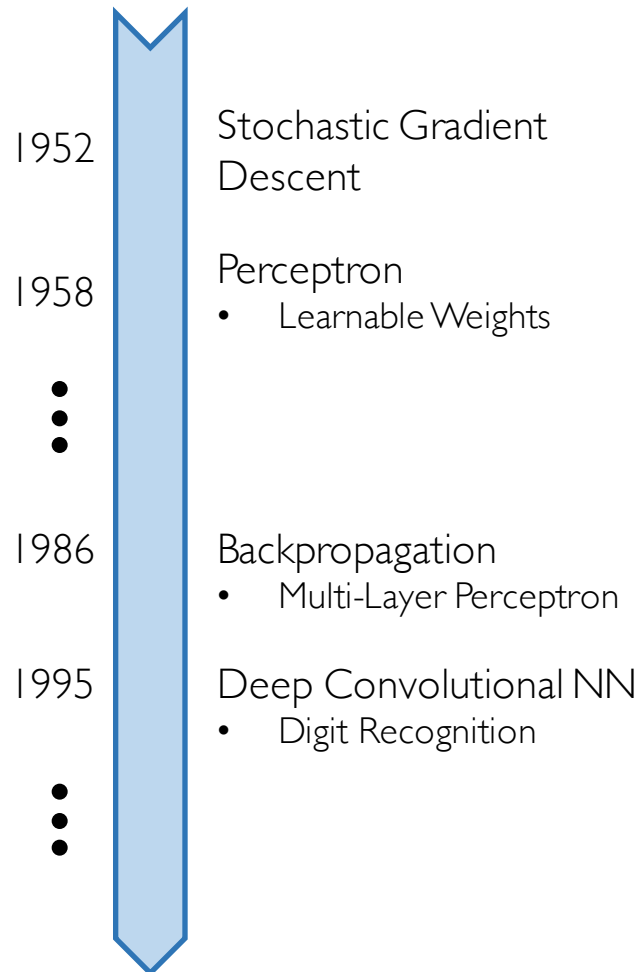
Eyes & Nose & Ears

High Level Features



Facial Structure

Why Now?



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

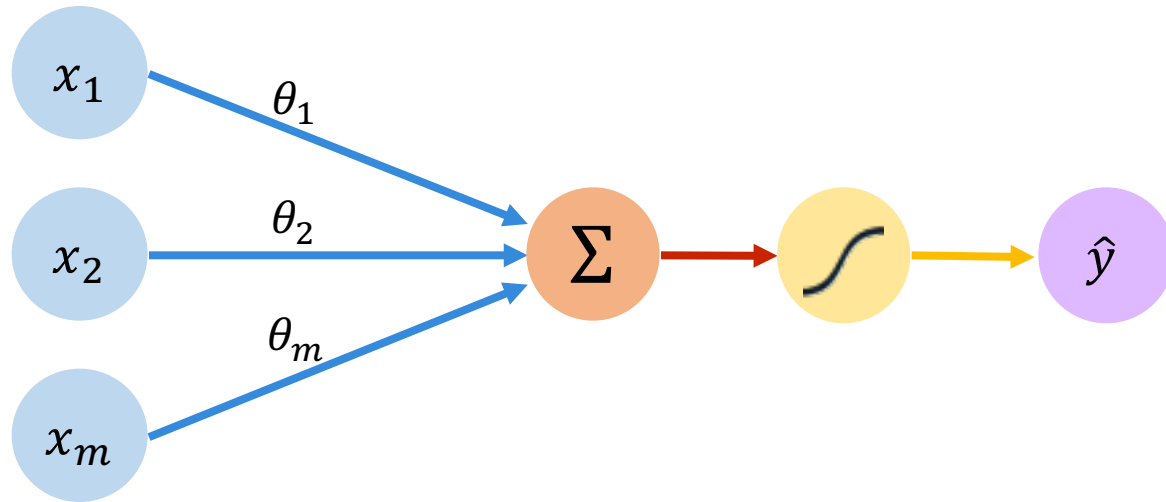
- Improved Techniques
- New Models
- Toolboxes



The Perceptron

The structural building block of deep learning

The Perceptron: Forward Propagation



Output

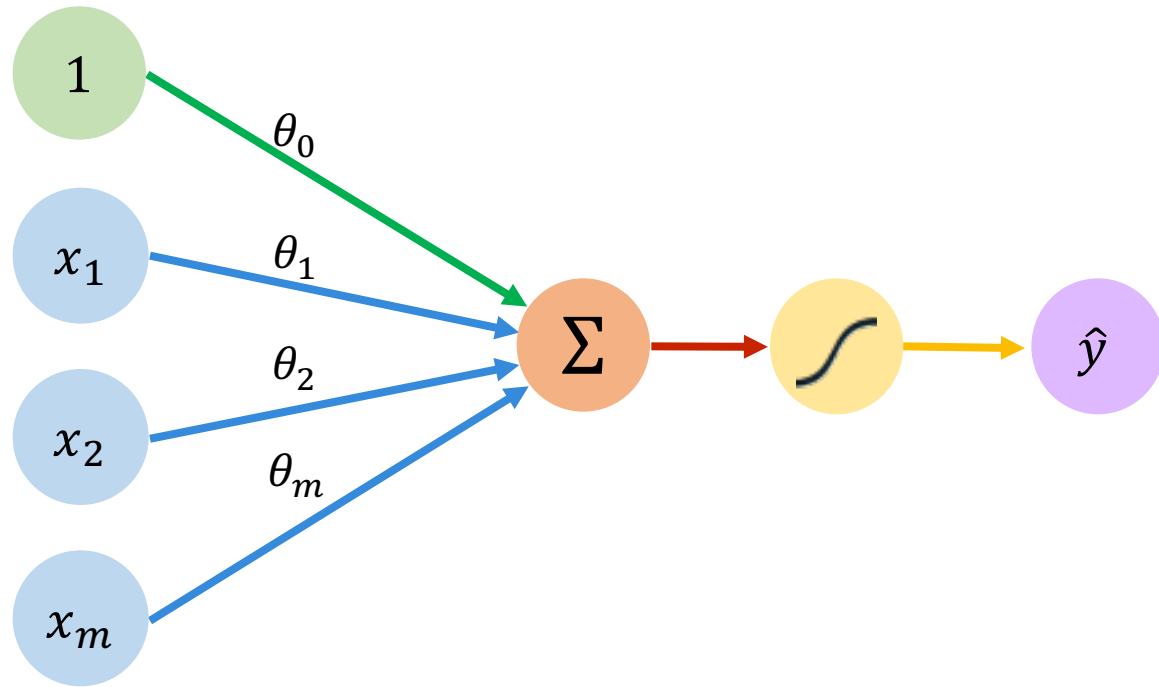
Linear combination of inputs

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

Non-linear activation function

Inputs Weights Sum Non-Linearity Output

The Perceptron: Forward Propagation



Inputs Weights Sum Non-Linearity Output

Output

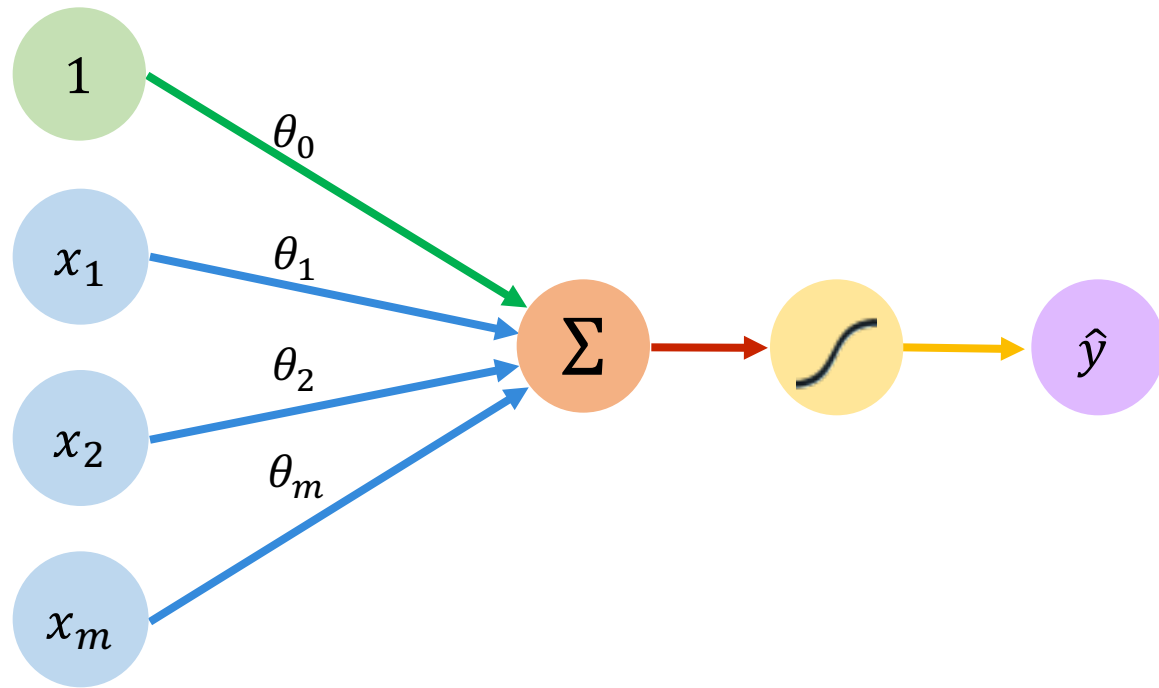
Linear combination of inputs

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

Non-linear activation function

Bias

The Perceptron: Forward Propagation



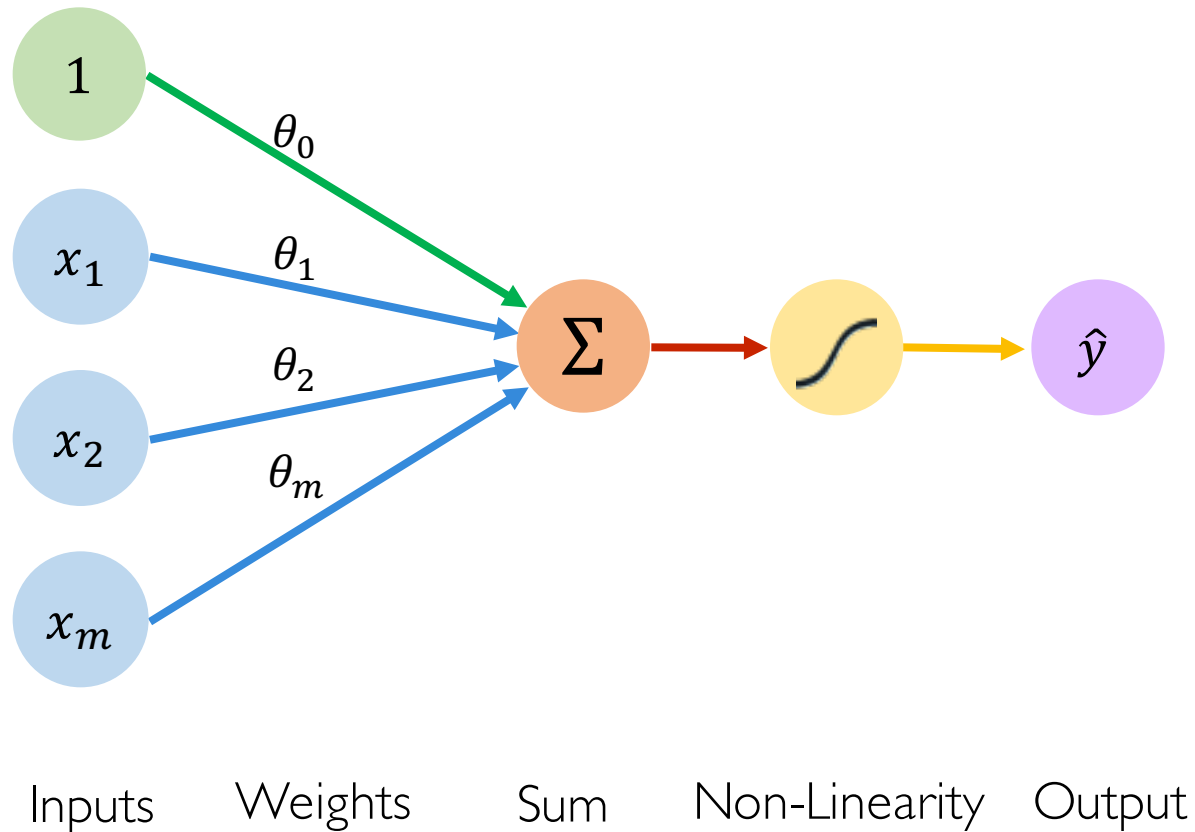
$$\hat{y} = g \left(\theta_0 + \sum_{i=1}^m x_i \theta_i \right)$$

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

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

Inputs Weights Sum Non-Linearity Output

The Perceptron: Forward Propagation

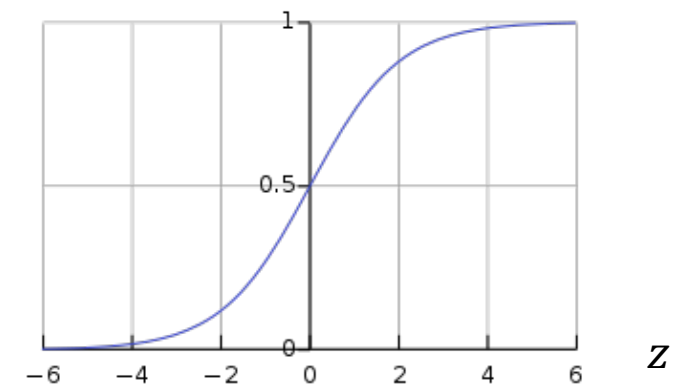


Activation Functions

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

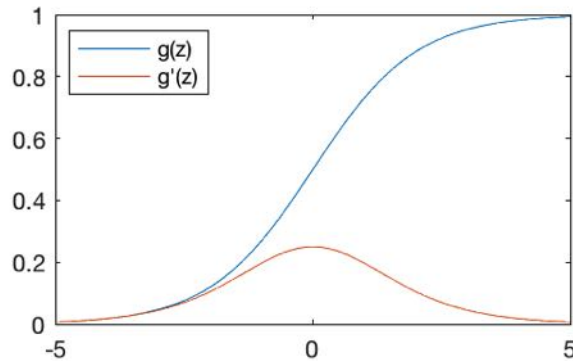
- Example: sigmoid function

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



Common Activation Functions

Sigmoid Function

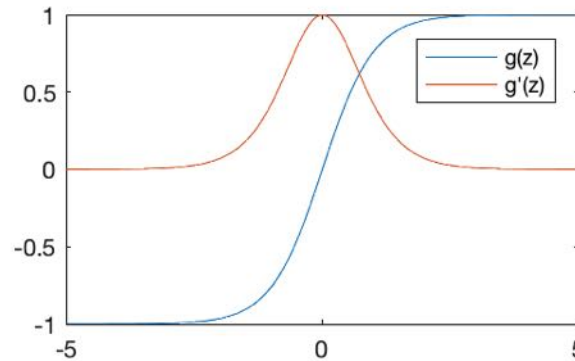


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

 `tf.nn.sigmoid(z)`

Hyperbolic Tangent

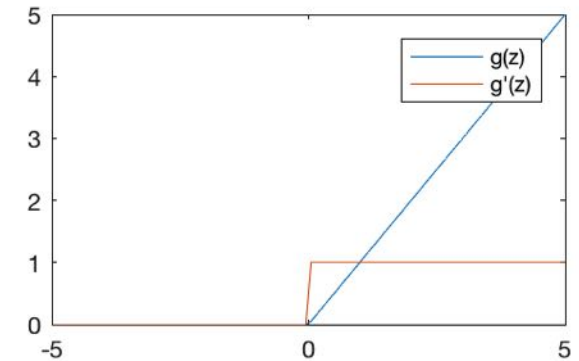


$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

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

 `tf.nn.tanh(z)`

Rectified Linear Unit (ReLU)



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

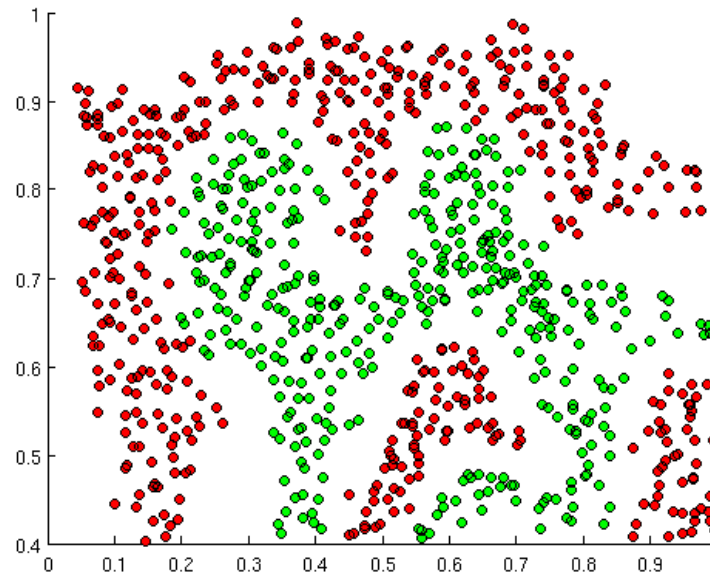
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

 `tf.nn.relu(z)`

NOTE: All activation functions are **non-linear**

Importance of Activation Functions

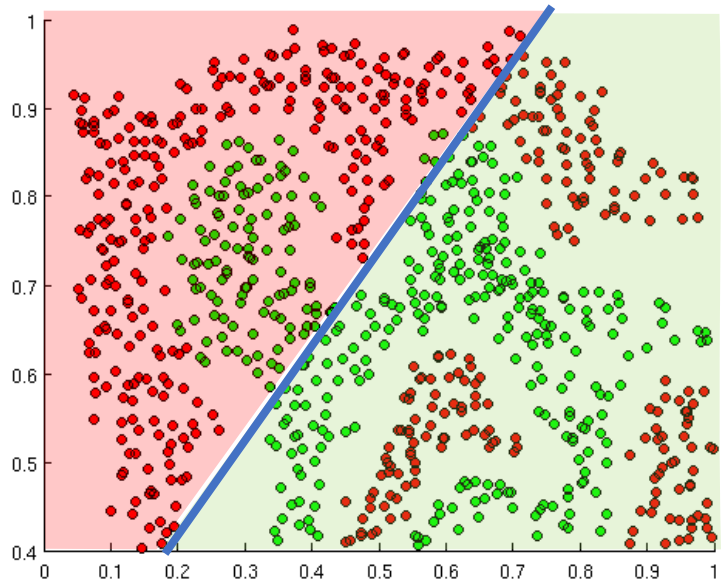
*The purpose of activation functions is to **introduce non-linearities** into the network*



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

Importance of Activation Functions

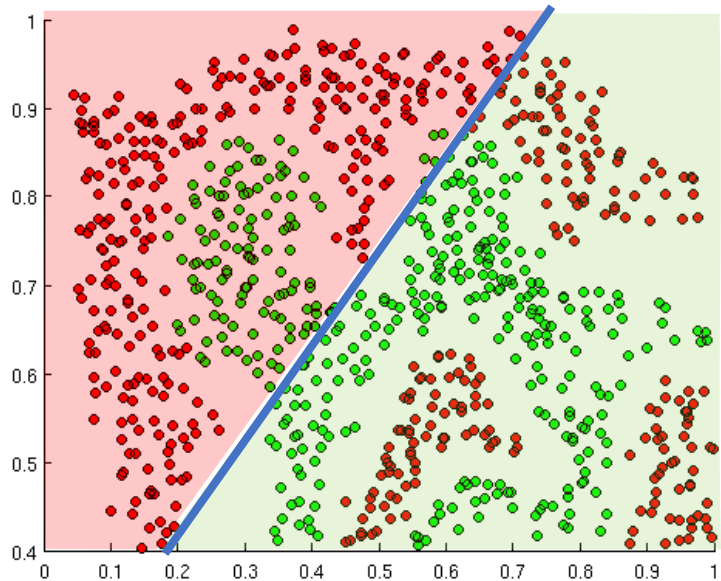
The purpose of activation functions is to **introduce non-linearities** into the network



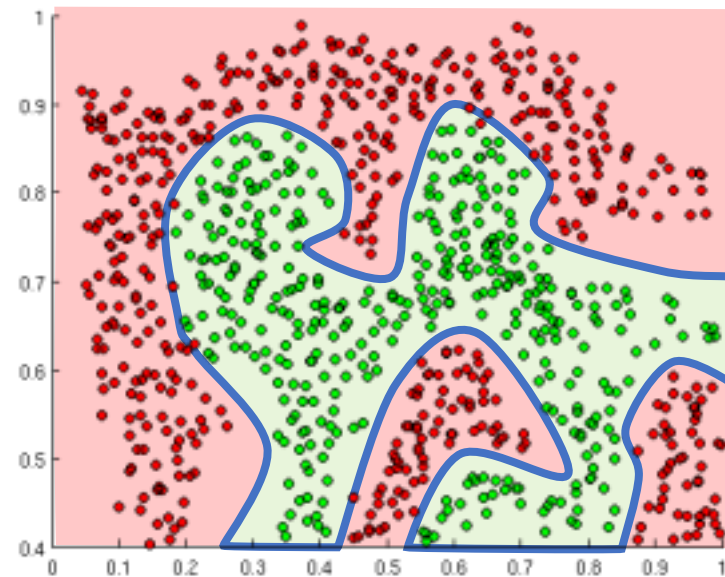
Linear Activation functions produce linear decisions no matter the network size

Importance of Activation Functions

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

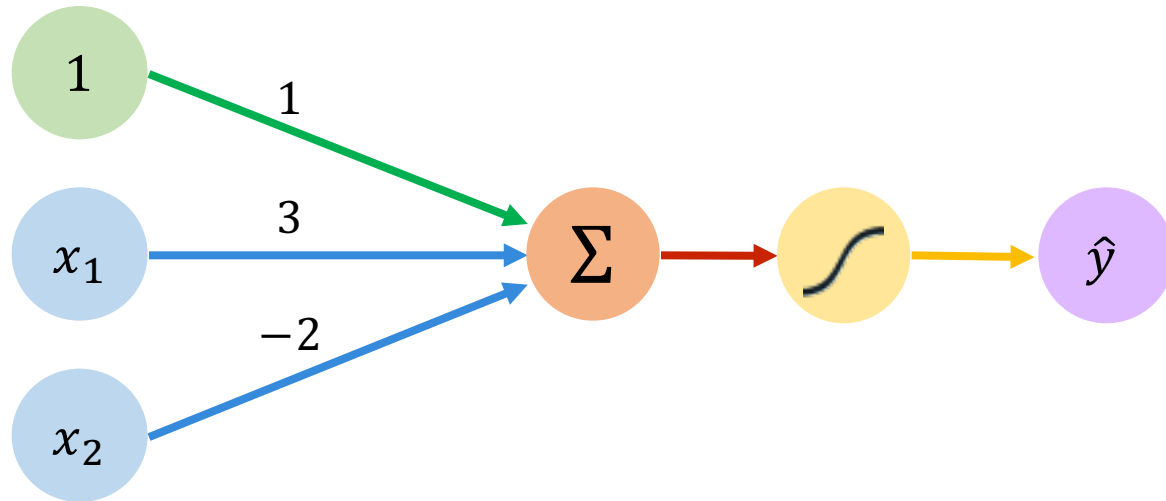


Linear Activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example

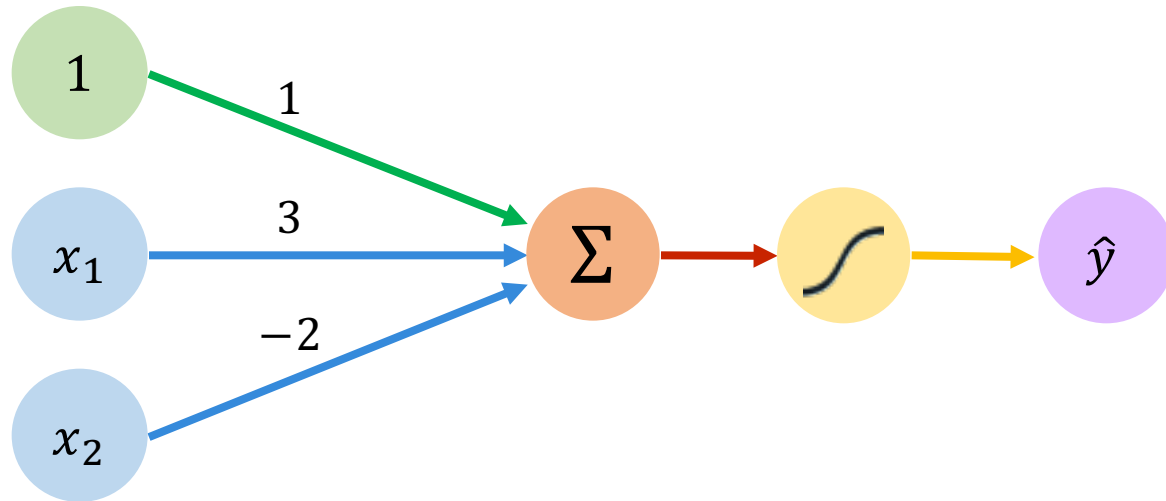


We have: $\theta_0 = 1$ and $\boldsymbol{\theta} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

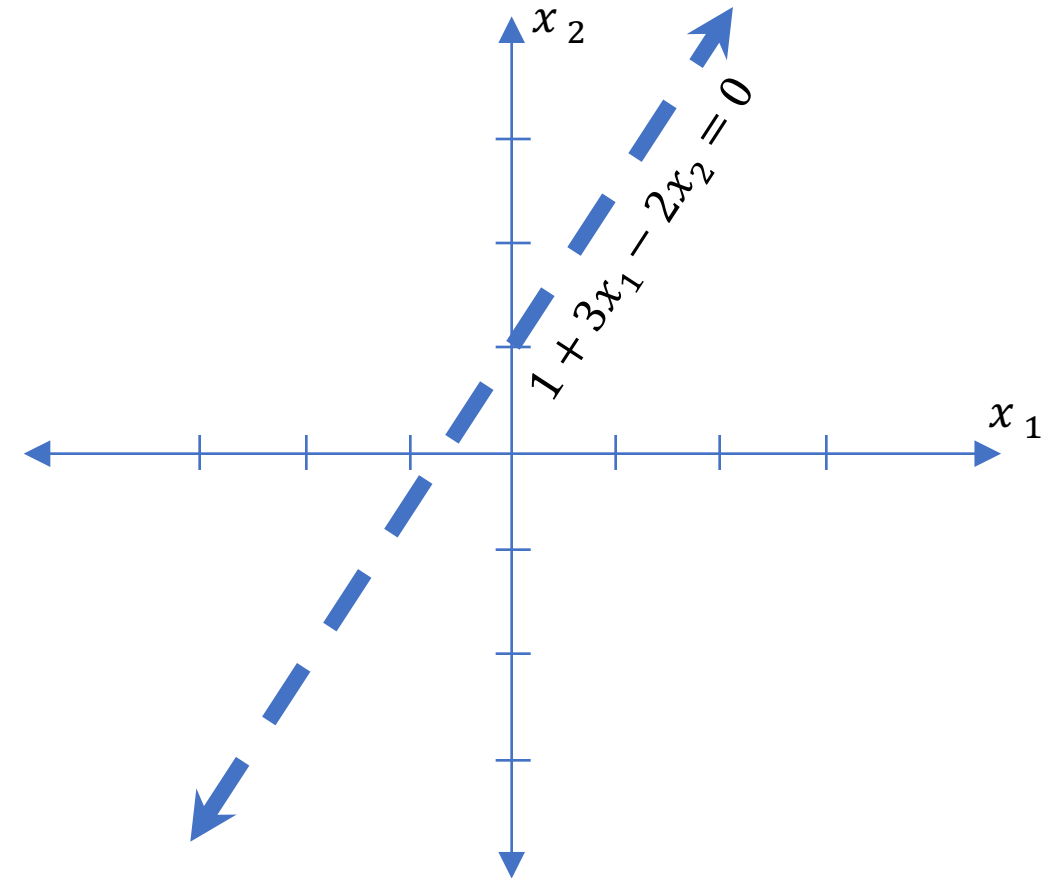
$$\begin{aligned}\hat{y} &= g(\theta_0 + \mathbf{X}^T \boldsymbol{\theta}) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g(1 + 3x_1 - 2x_2)\end{aligned}$$

This is just a line in 2D!

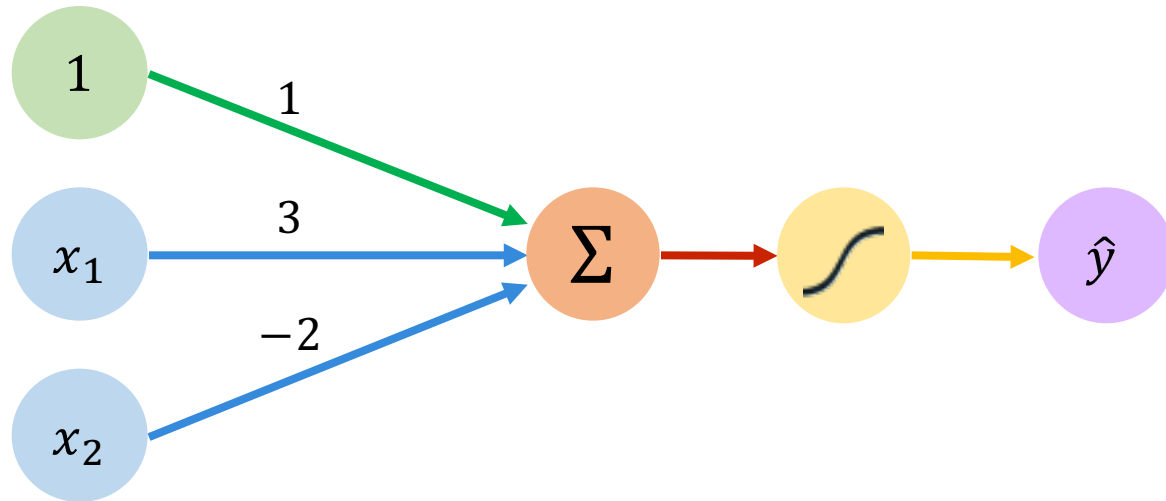
The Perceptron: Example



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



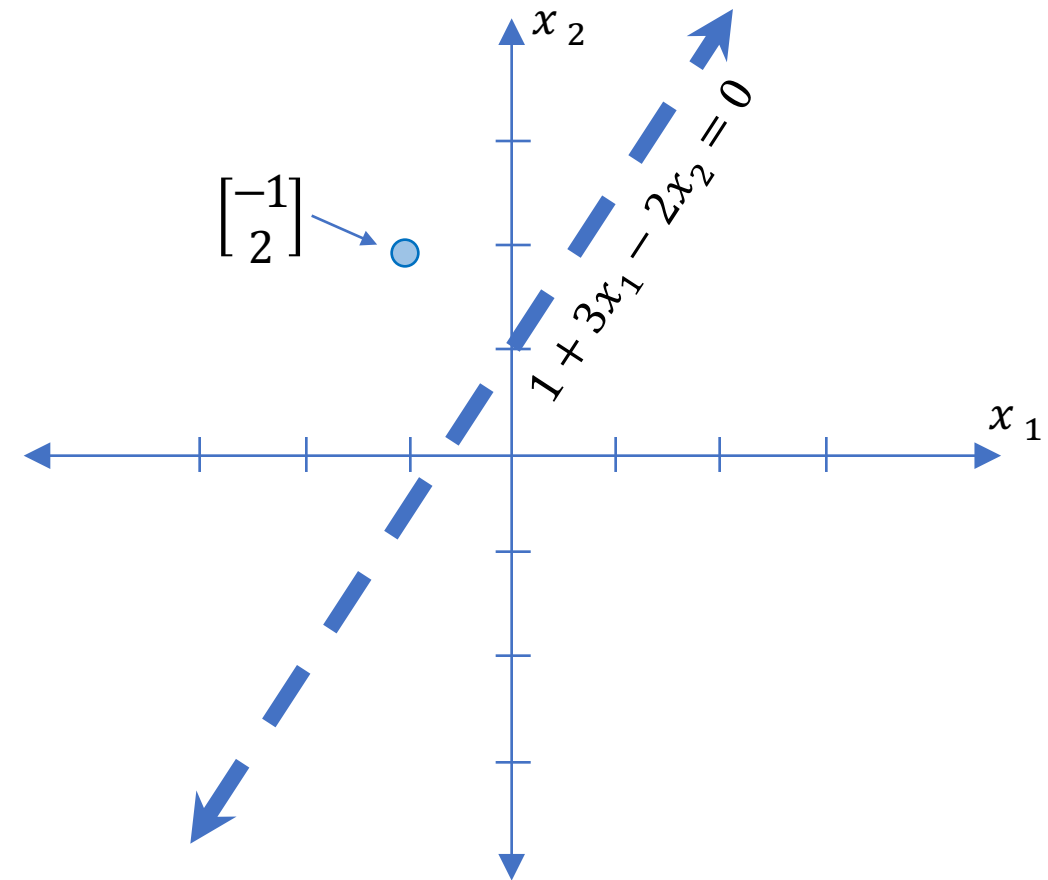
The Perceptron: Example



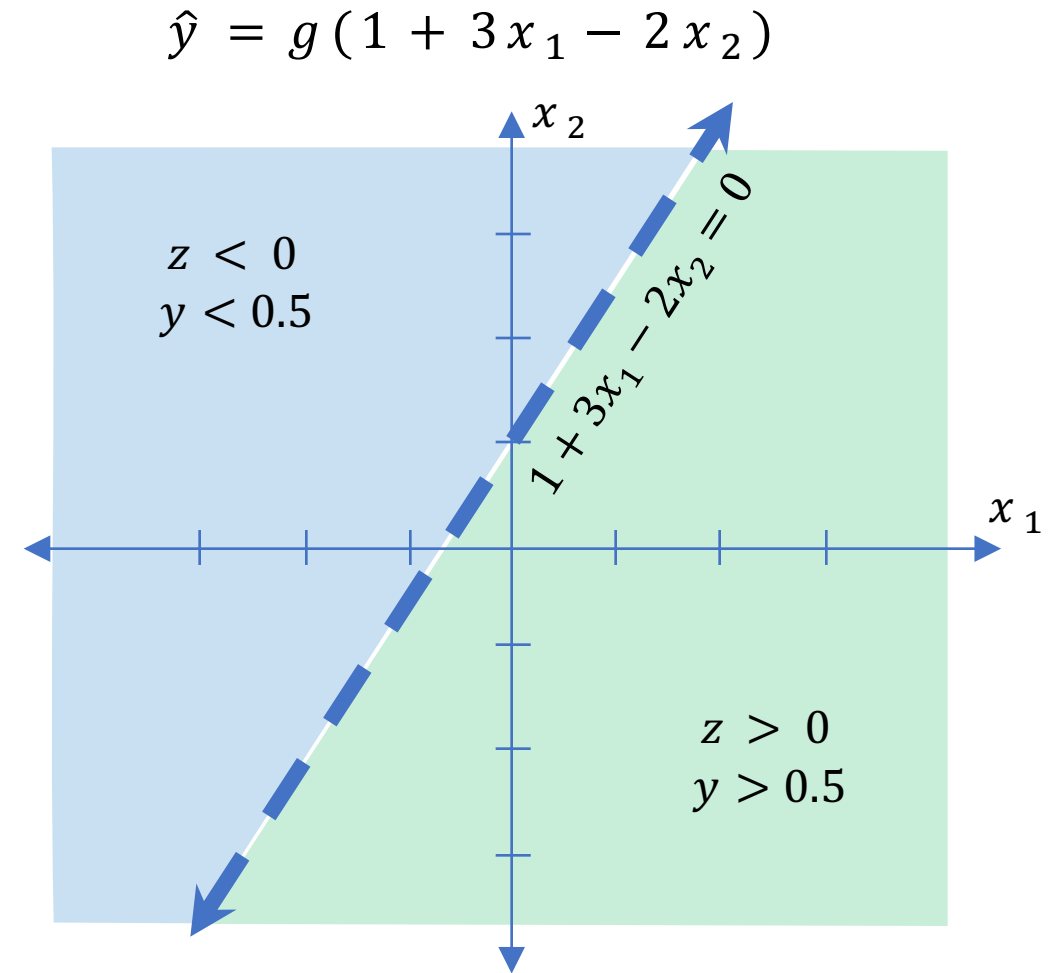
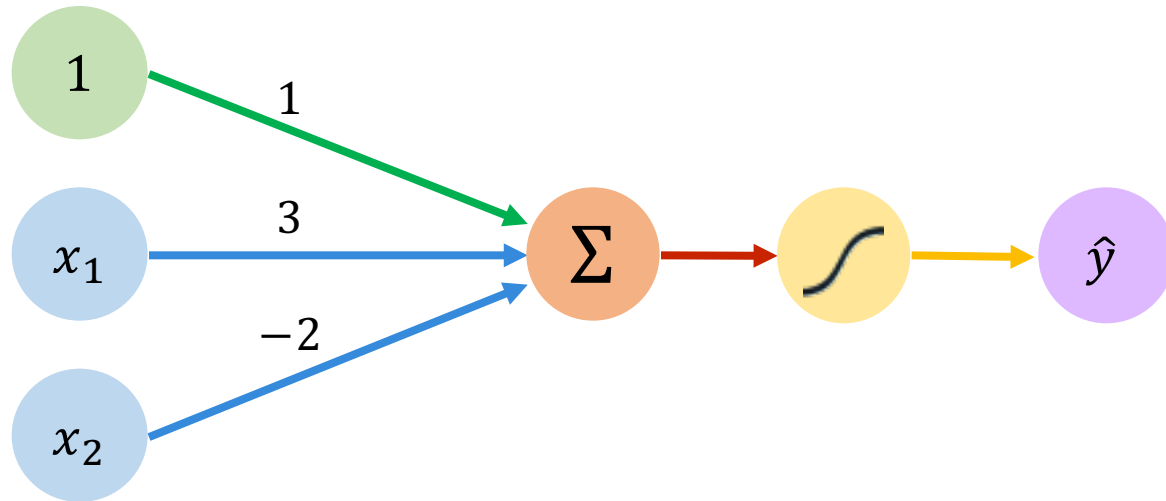
Assume we have input: $\mathbf{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

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

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

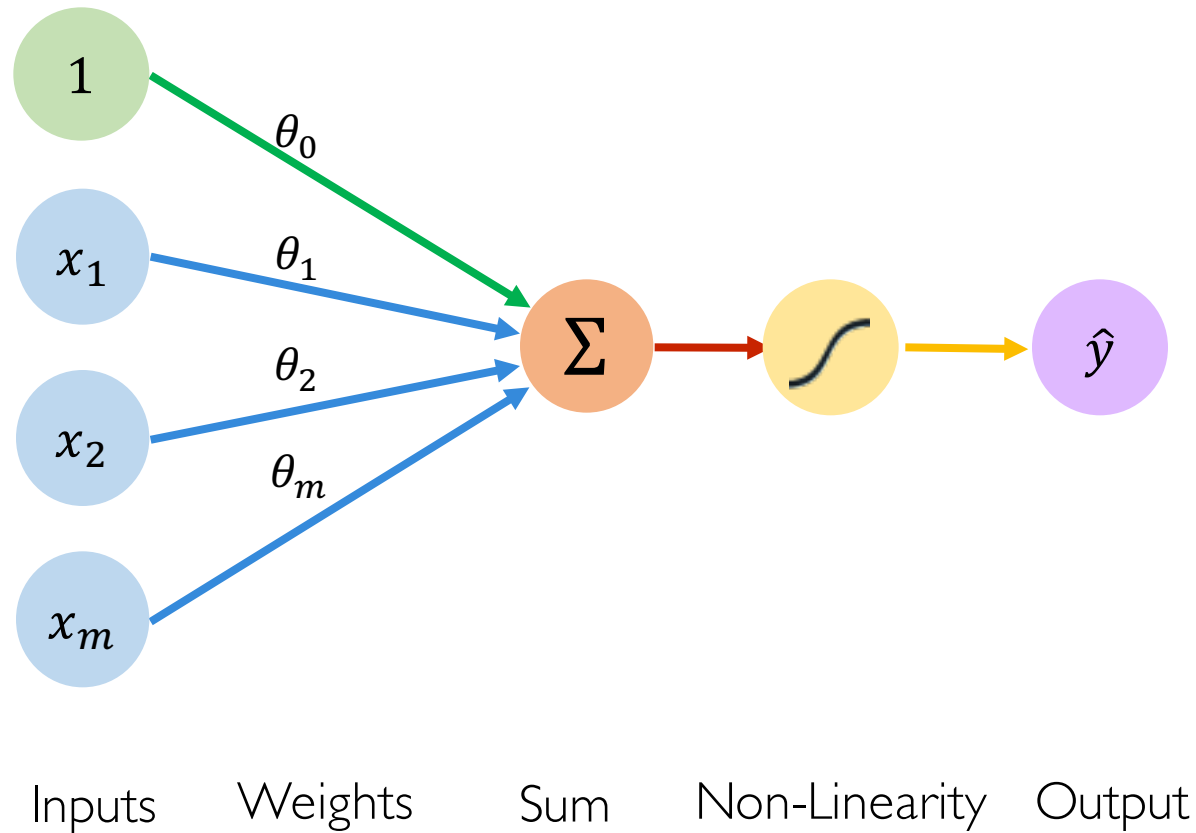


The Perceptron: Example

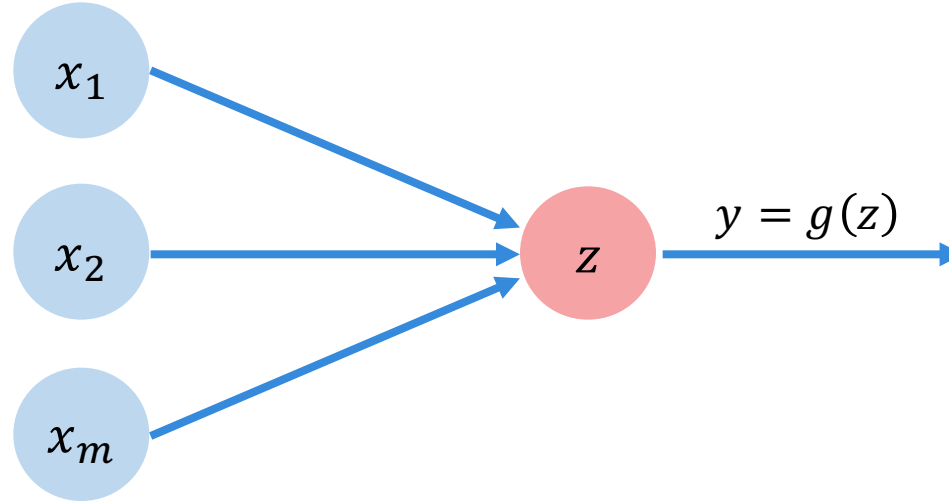


Building Neural Networks with Perceptrons

The Perceptron: Simplified

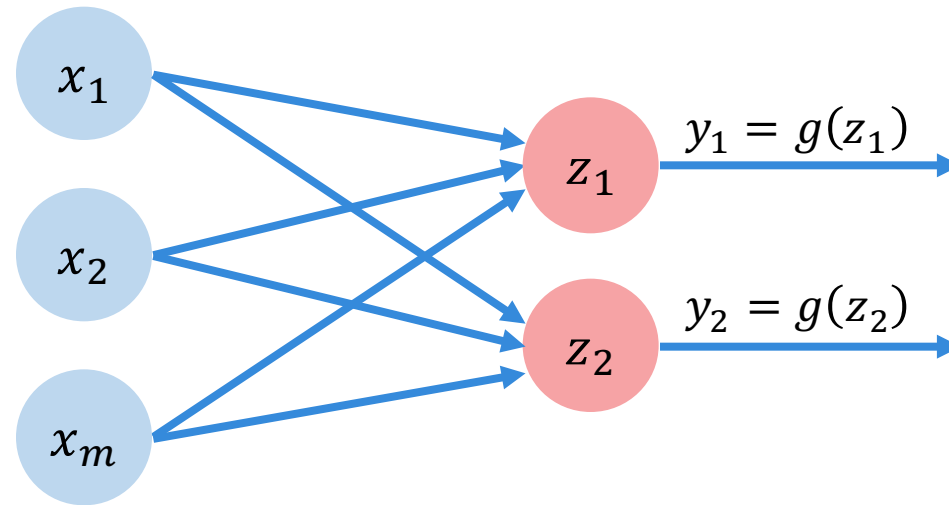


The Perceptron: Simplified



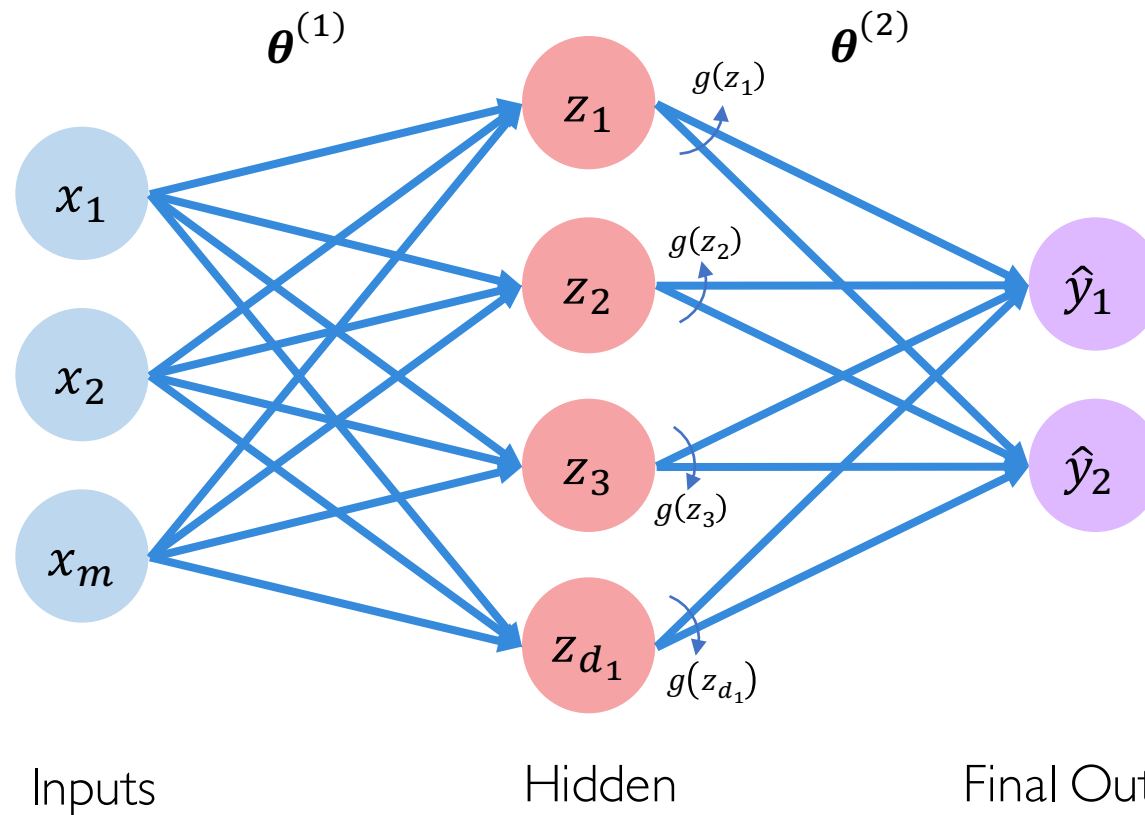
$$z = \theta_0 + \sum_{j=1}^m x_j \theta_j$$

Multi Output Perceptron



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

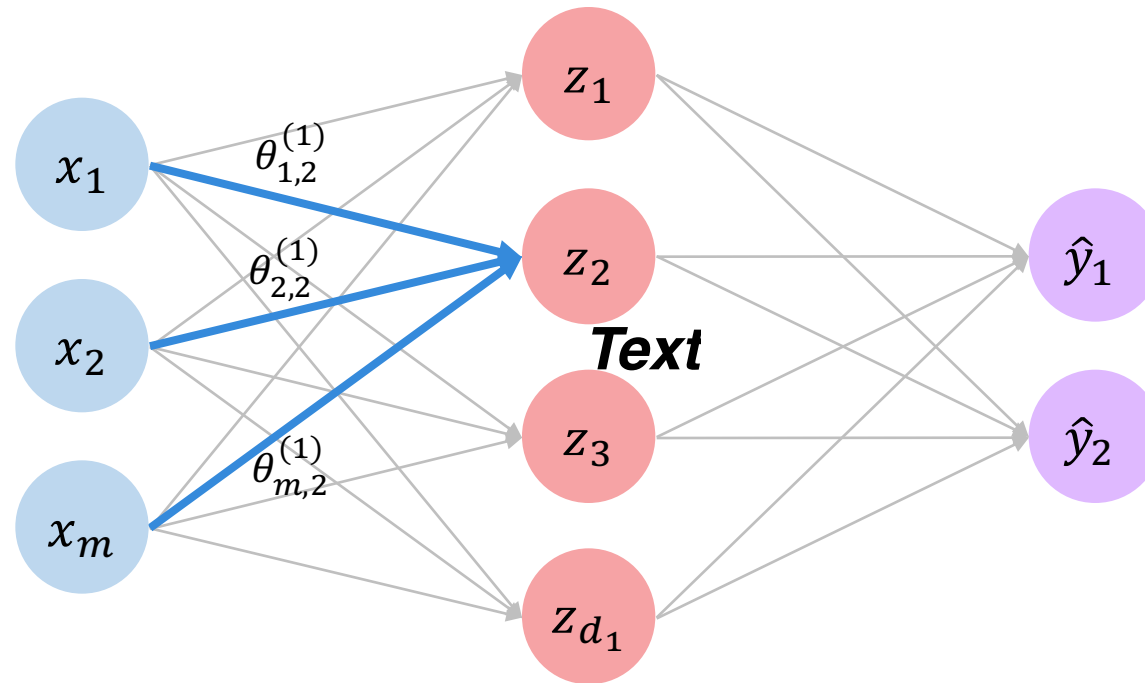
Single Layer Neural Network



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

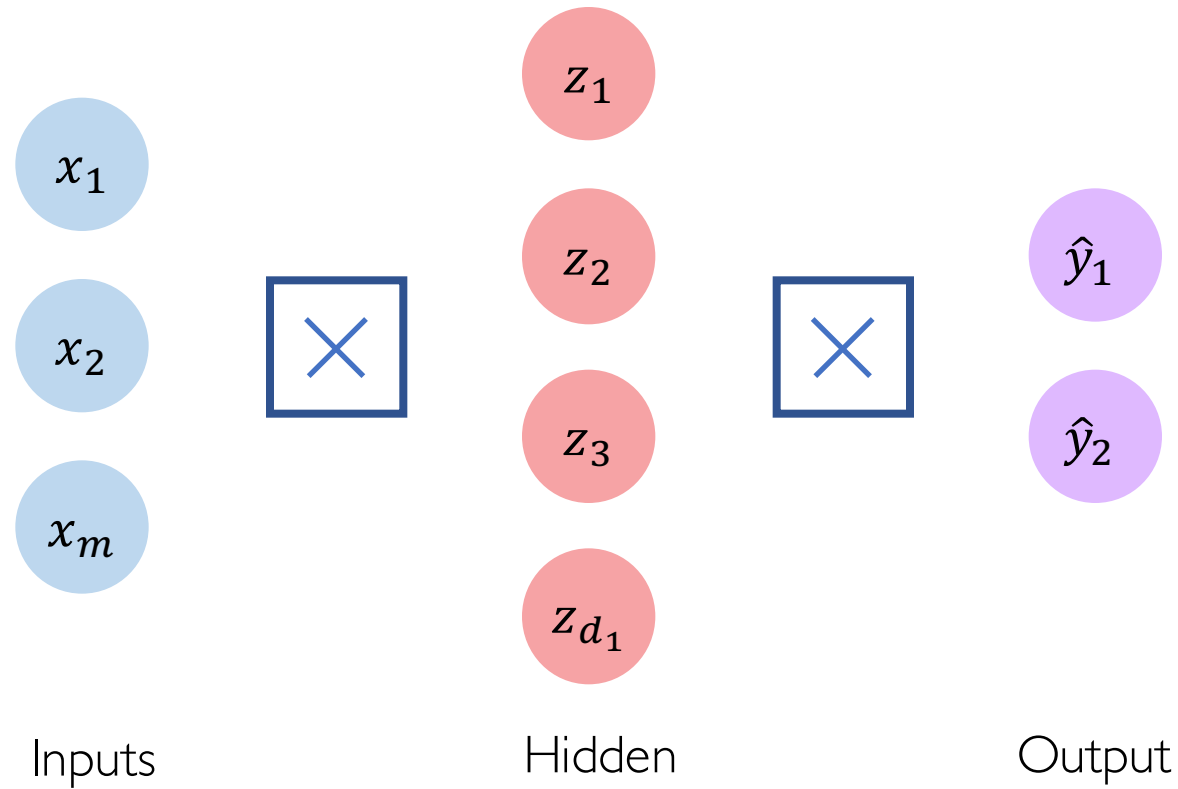
$$\hat{y}_i = \theta_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) \theta_{j,i}^{(2)}$$

Single Layer Neural Network

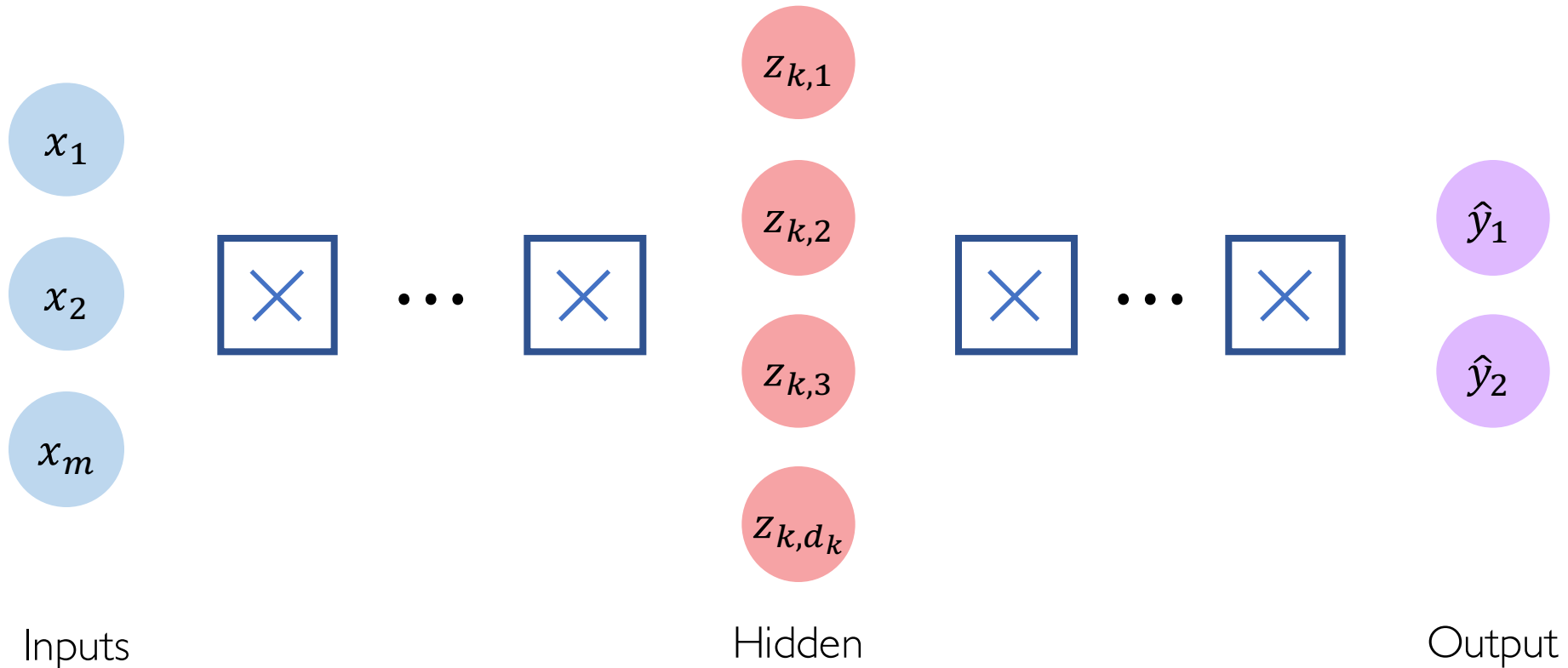


$$\begin{aligned} z_2 &= \theta_{0,2}^{(1)} + \sum_{j=1}^m x_j \theta_{j,2}^{(1)} \\ &= \theta_{0,2}^{(1)} + x_1 \theta_{1,2}^{(1)} + x_2 \theta_{2,2}^{(1)} + x_m \theta_{m,2}^{(1)} \end{aligned}$$

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

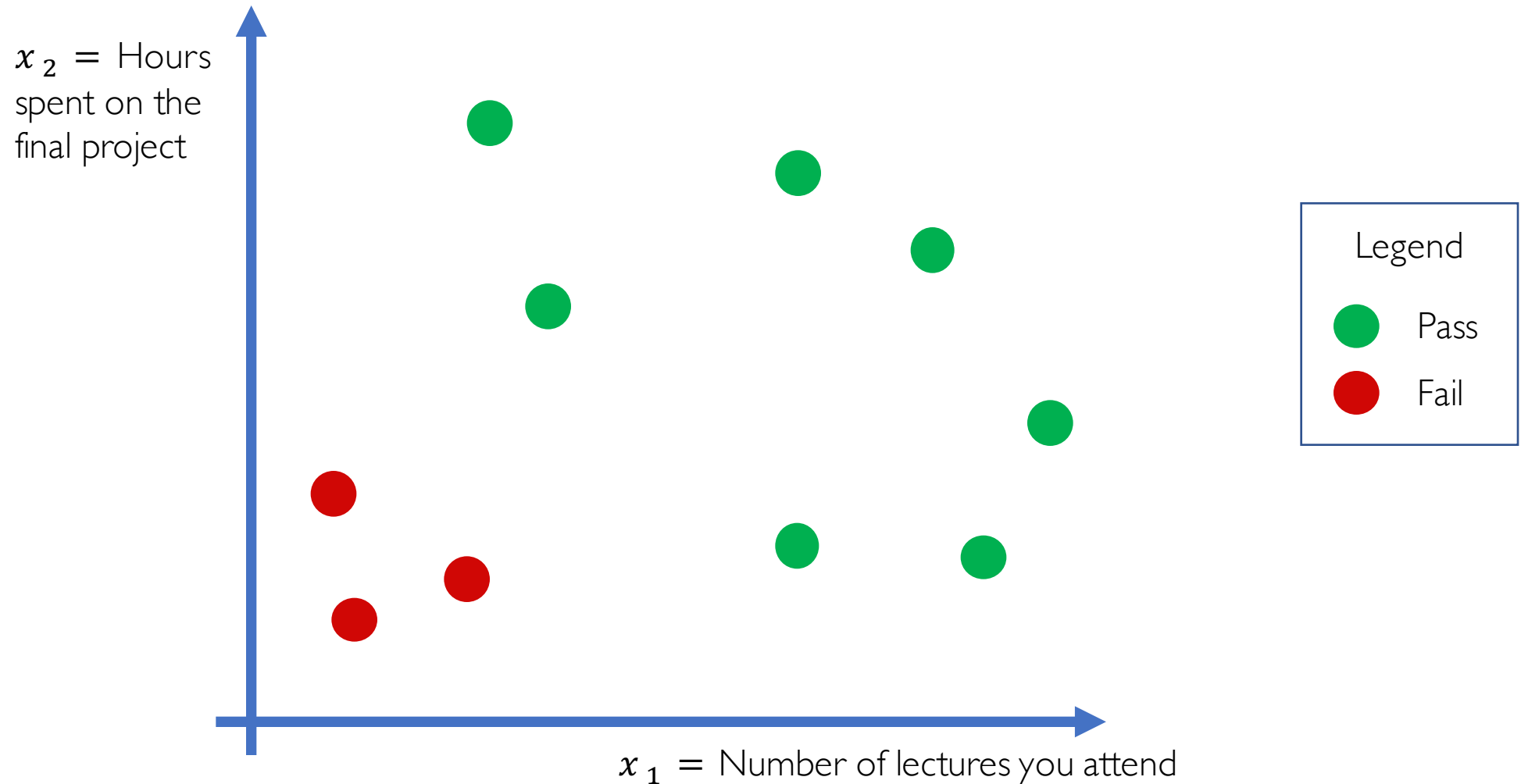
Will I pass this class?

Let's start with a simple two feature model

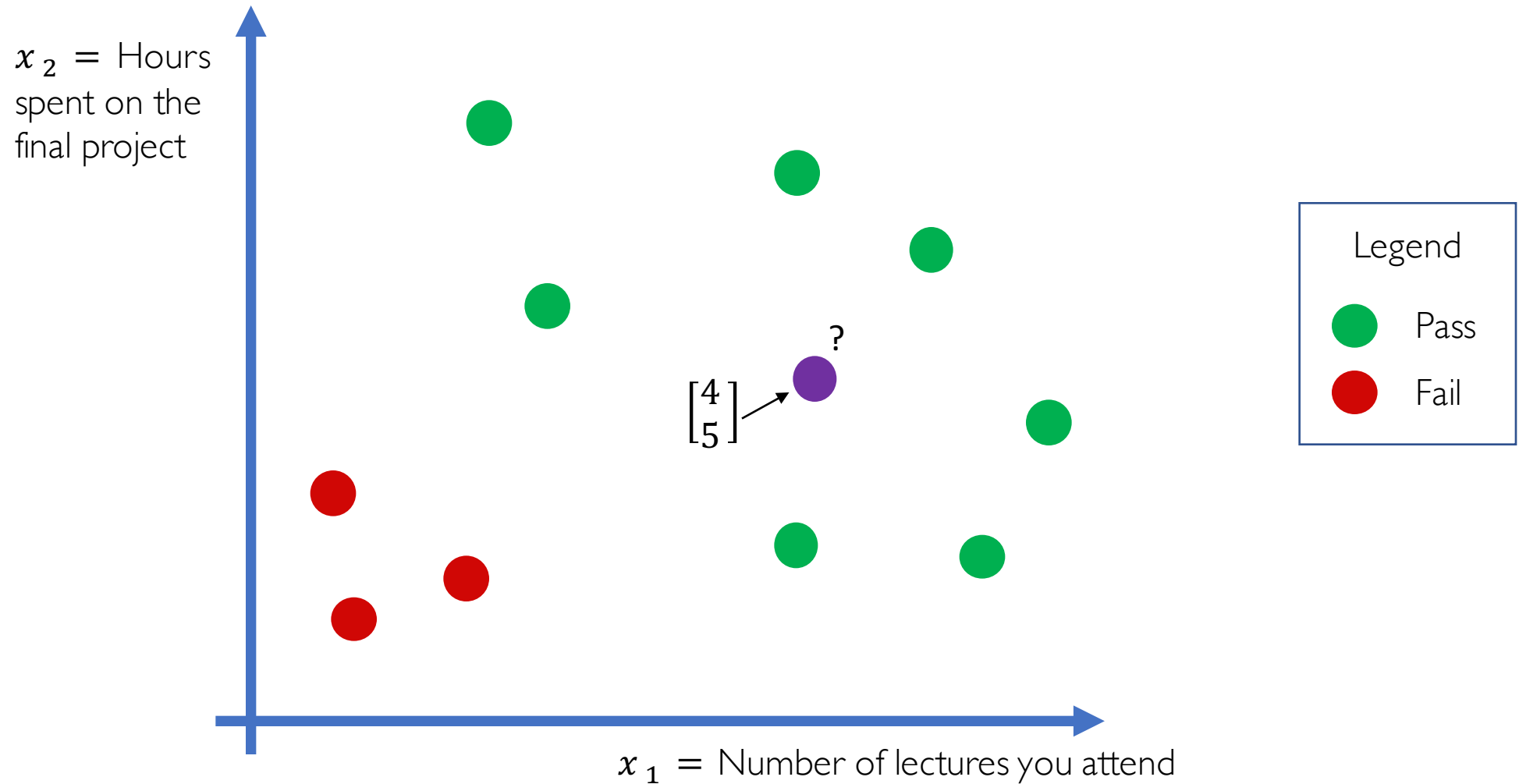
x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

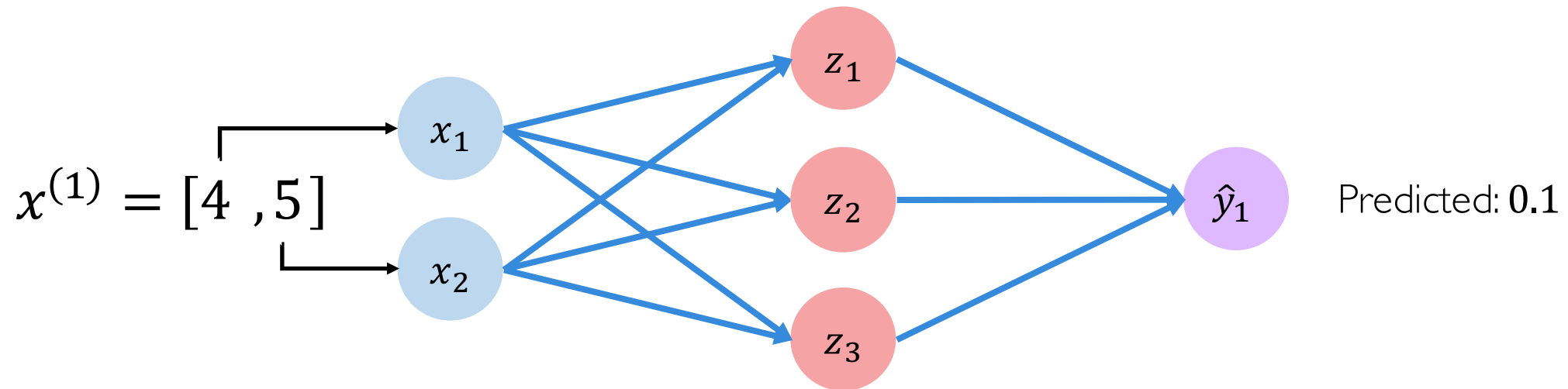
Example Problem: Will I pass this class?



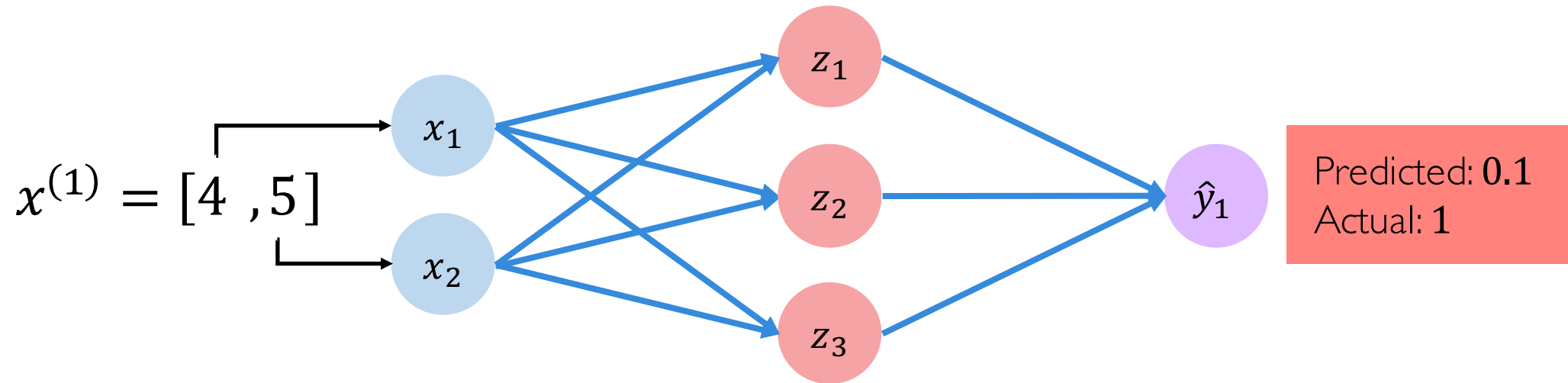
Example Problem: Will I pass this class?



Example Problem: Will I pass this class?

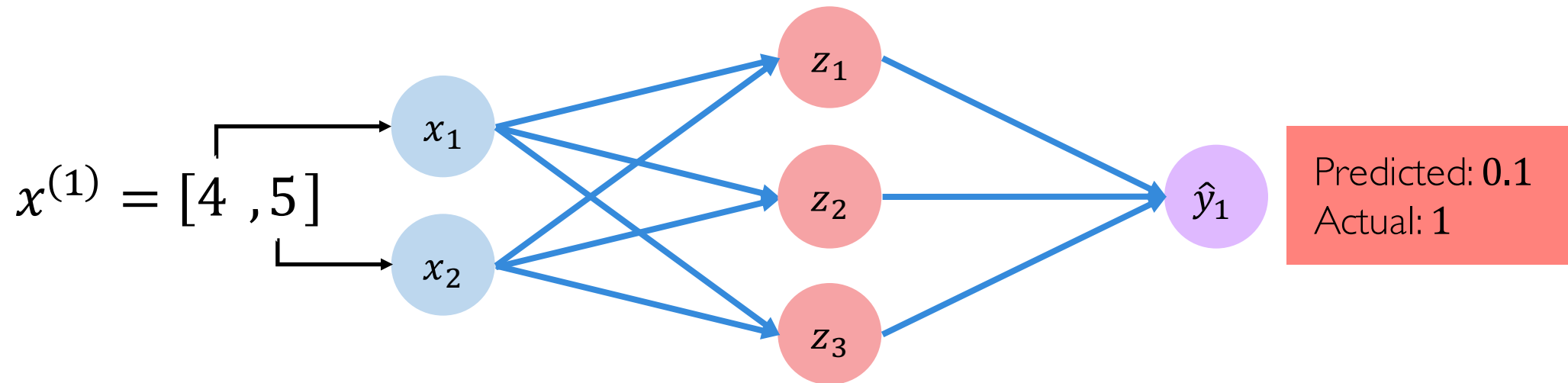


Example Problem: Will I pass this class?



Quantifying Loss

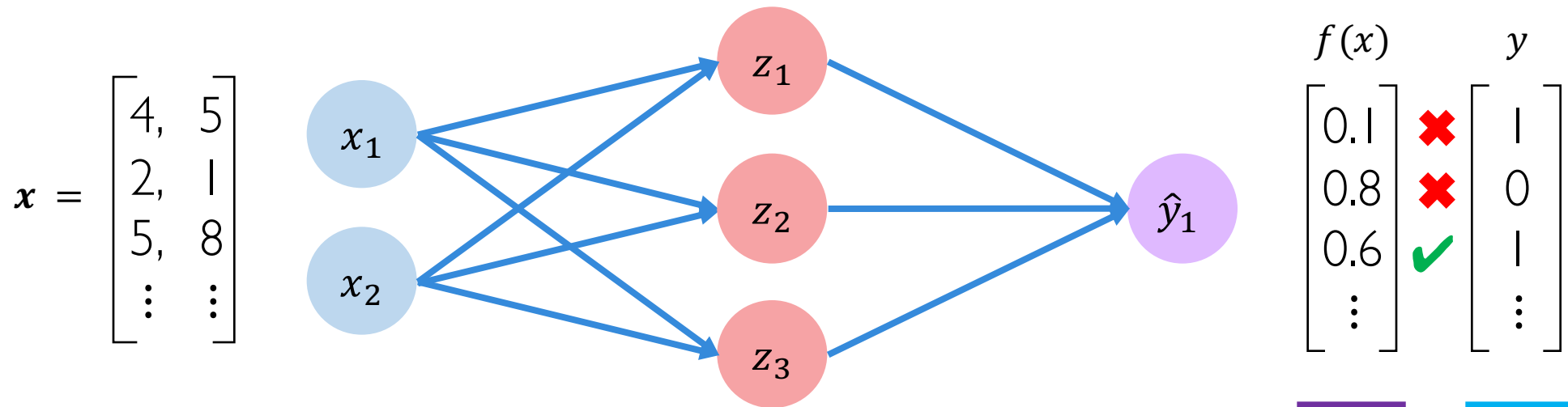
The **loss** of our network measures the cost incurred from incorrect predictions



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

Empirical Loss

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



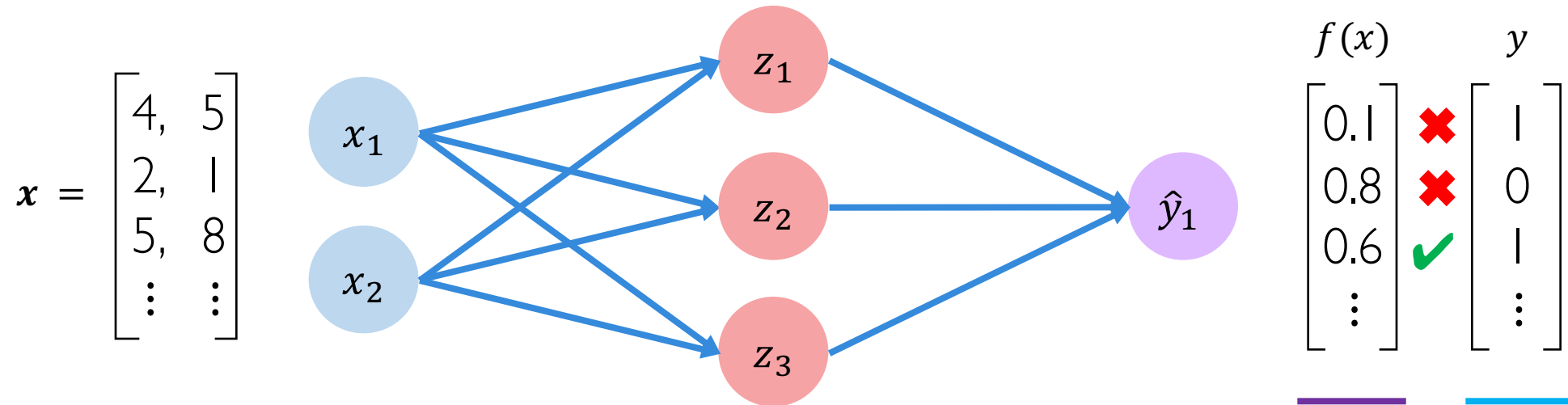
Also known as:

- Objective function
- Cost function
- Empirical Risk

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

Binary Cross Entropy Loss

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



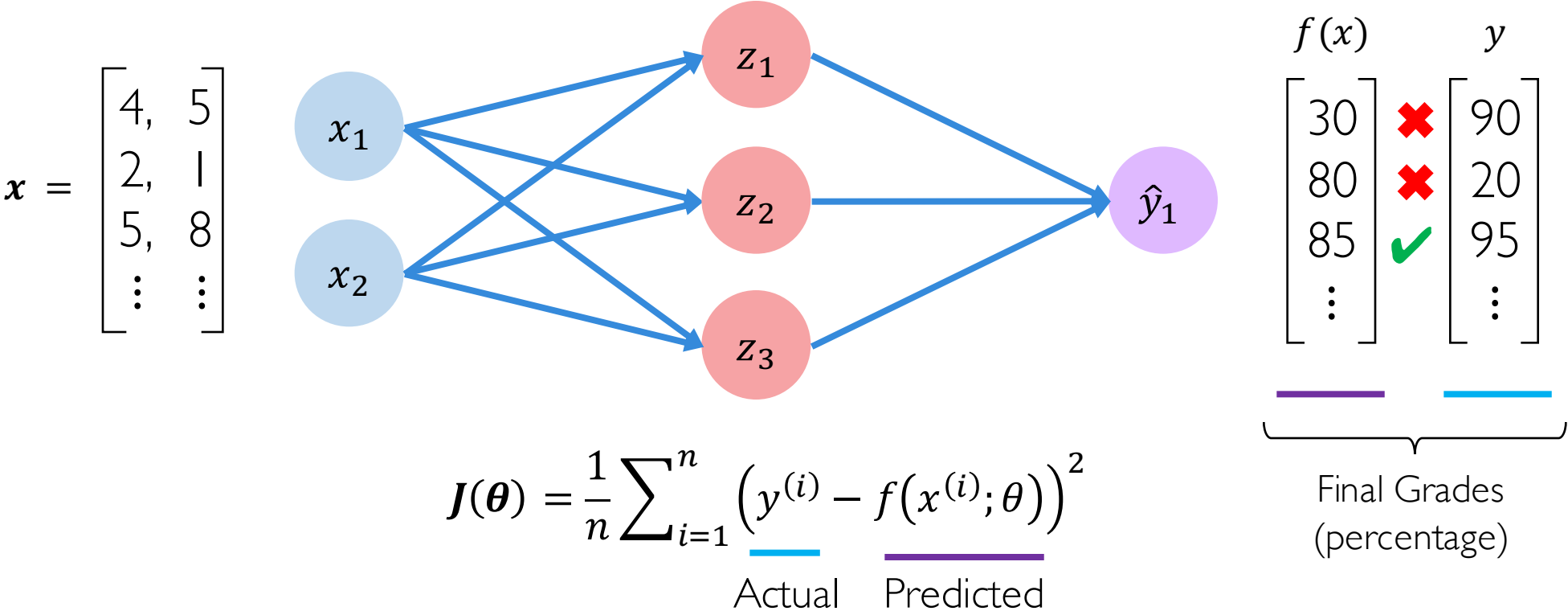
$$J(\theta) = \frac{1}{n} \sum_{i=1}^n \underbrace{y^{(i)}}_{\text{Actual}} \log \left(\underbrace{f(x^{(i)}; \theta)}_{\text{Predicted}} \right) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log \left(1 - \underbrace{f(x^{(i)}; \theta)}_{\text{Predicted}} \right)$$



```
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



```
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**

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

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

Loss Optimization

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

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

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

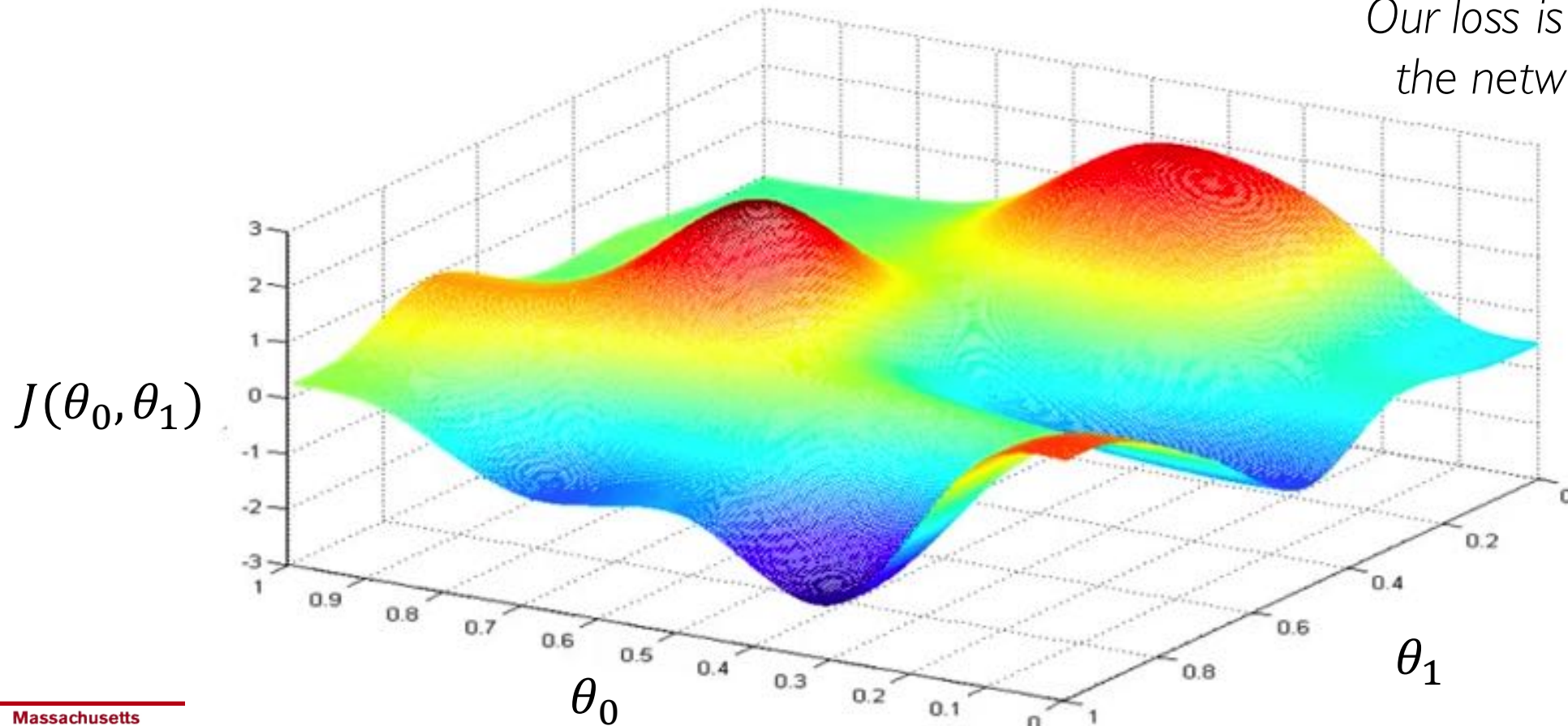
Remember:

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

Loss Optimization

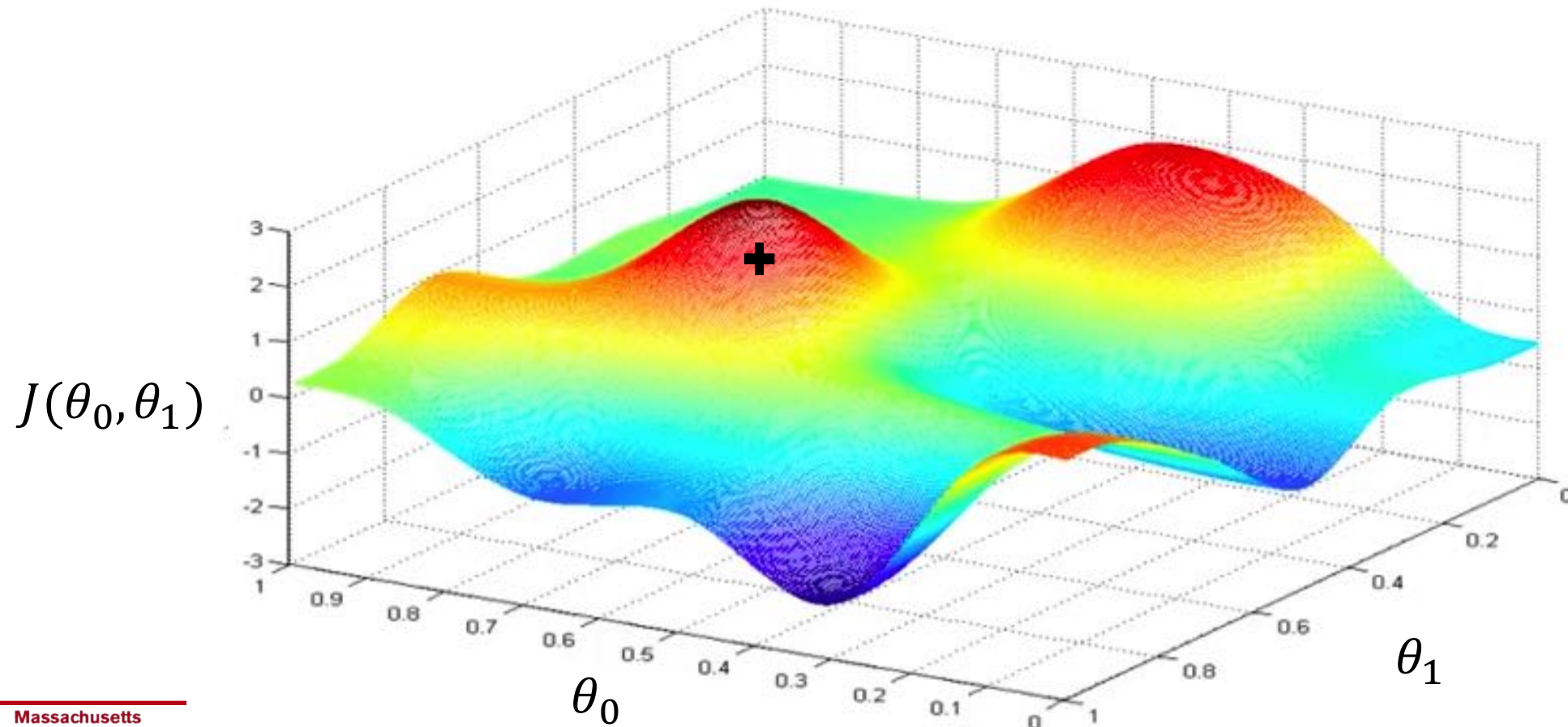
$$\theta^* = \underset{\theta}{\operatorname{argmin}} J(\theta)$$

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



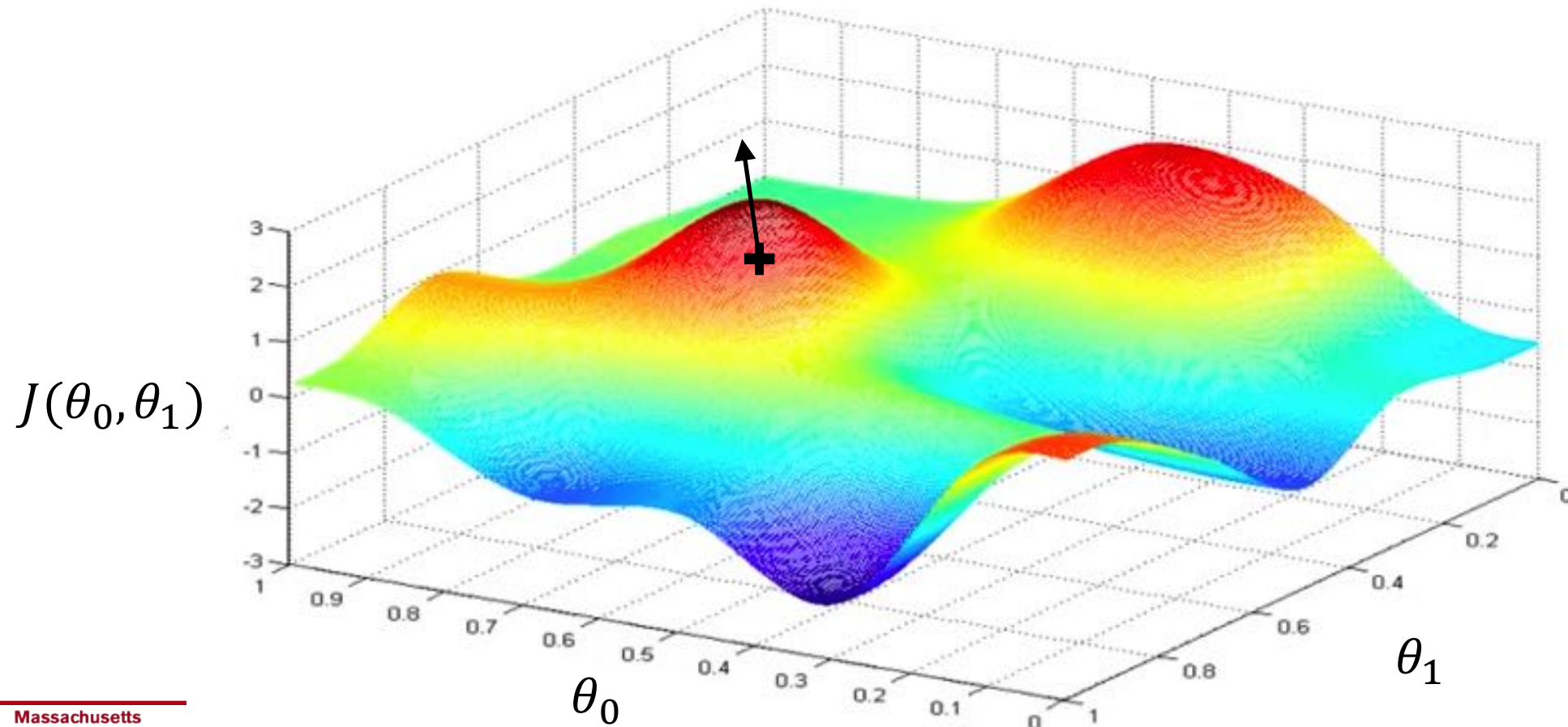
Loss Optimization

Randomly pick an initial (θ_0, θ_1)



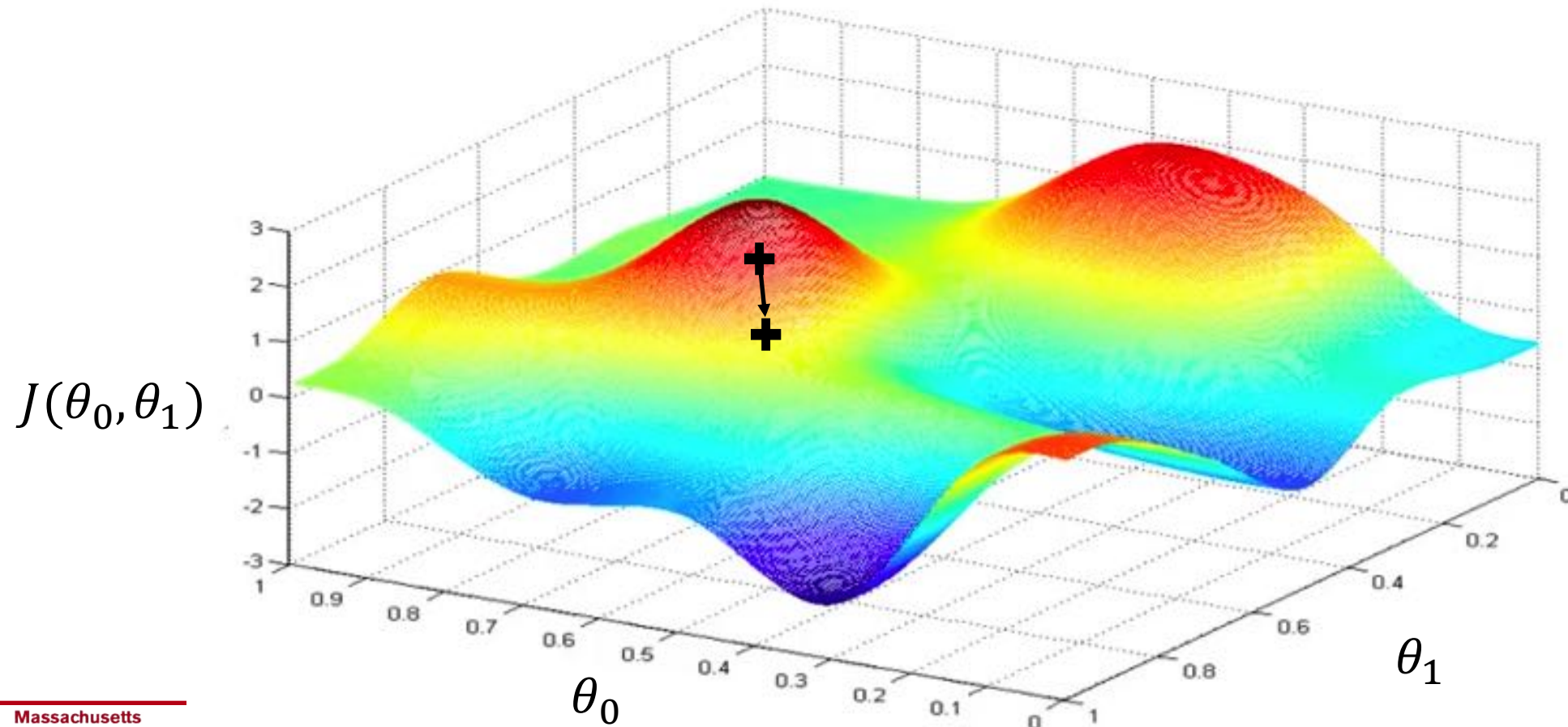
Loss Optimization

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



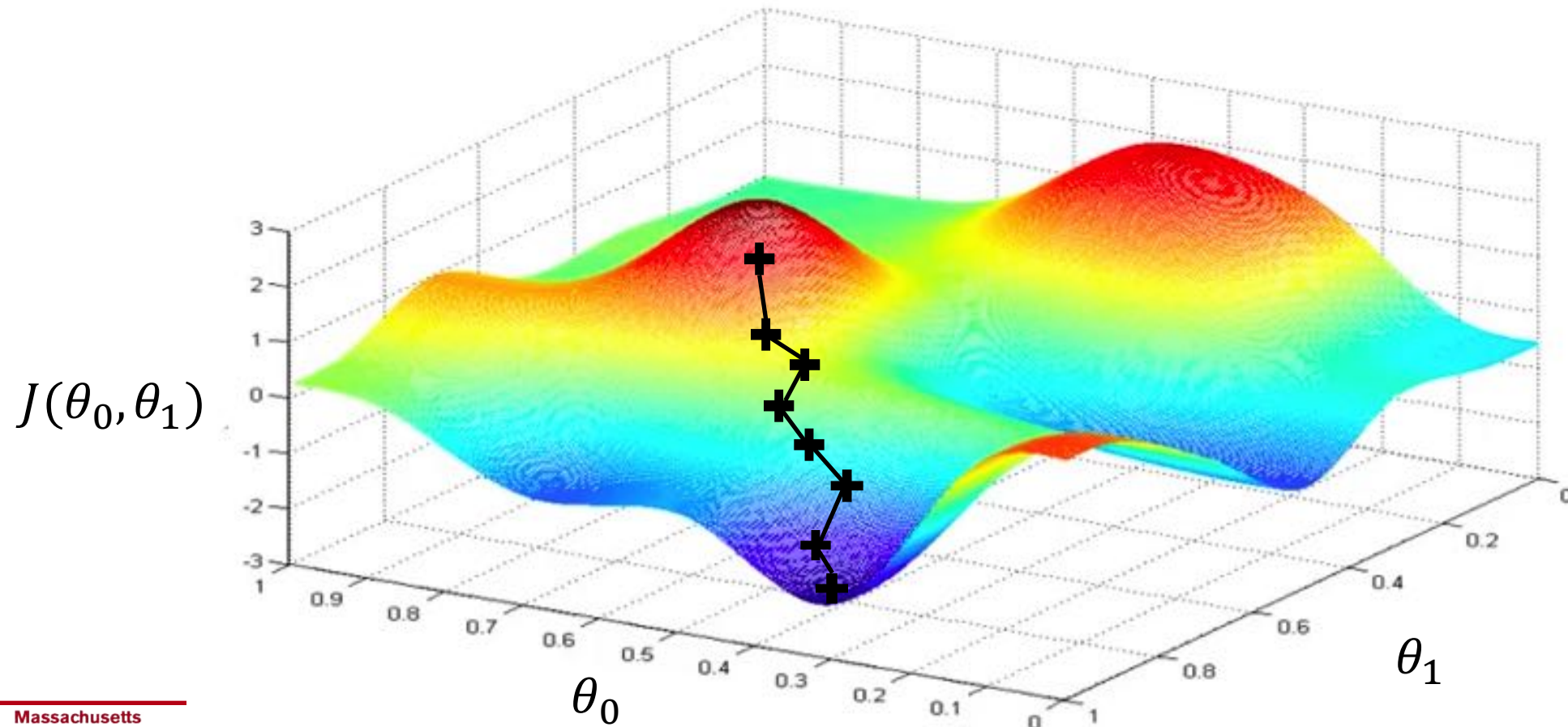
Loss Optimization

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, $\frac{\partial J(\theta)}{\partial \theta}$
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

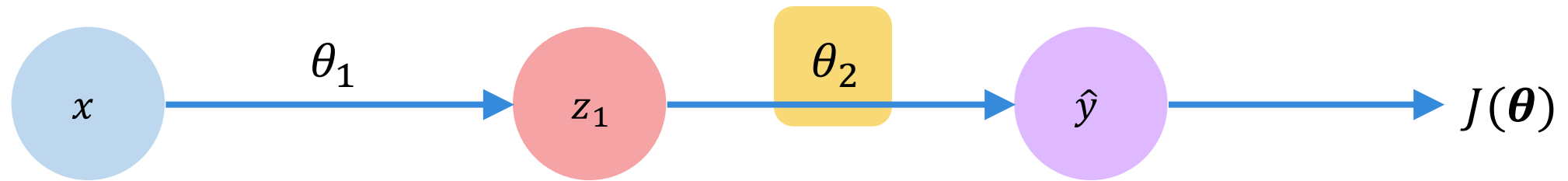
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights

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

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

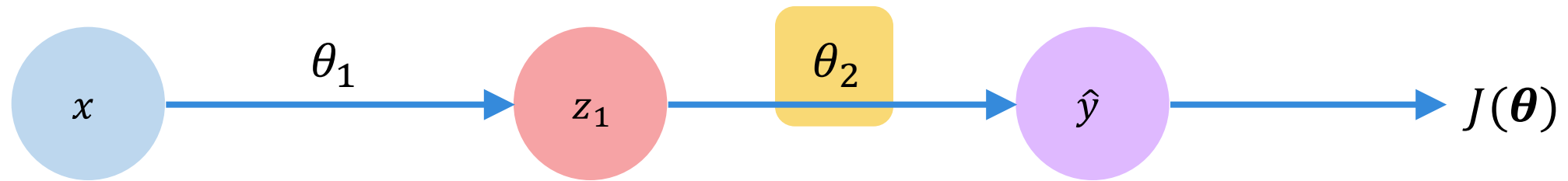
```
 weights_new = weights.assign(weights - lr * grads)
```

Computing Gradients: Backpropagation



How does a small change in one weight (ex. θ_2) affect the final loss $J(\theta)$?

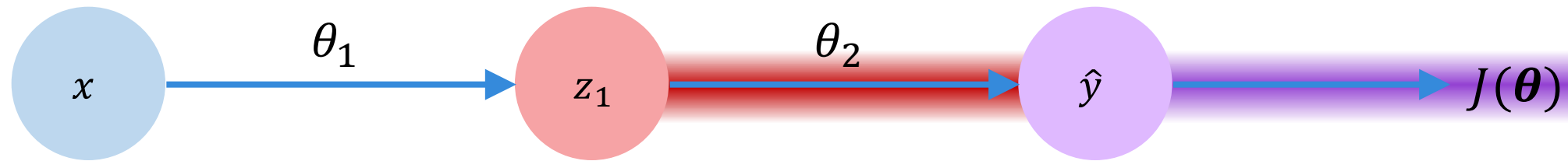
Computing Gradients: Backpropagation



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

Let's use the chain rule!

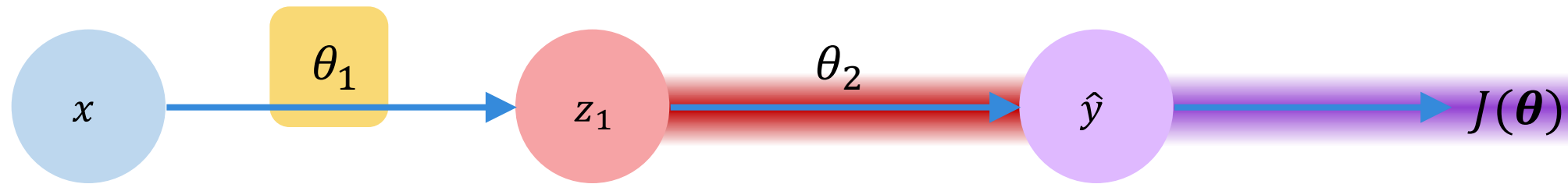
Computing Gradients: Backpropagation



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

A purple horizontal bar is positioned under the fraction $\frac{\partial J(\theta)}{\partial \hat{y}}$, and a red horizontal bar is positioned under the fraction $\frac{\partial \hat{y}}{\partial \theta_2}$.

Computing Gradients: Backpropagation

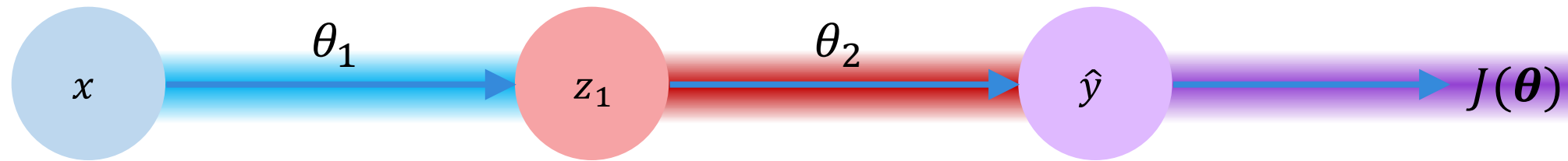


$$\frac{\partial J(\theta)}{\partial \theta_1} = \frac{\partial J(\theta)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \theta_1}$$

Apply chain rule!

Apply chain rule!

Computing Gradients: Backpropagation



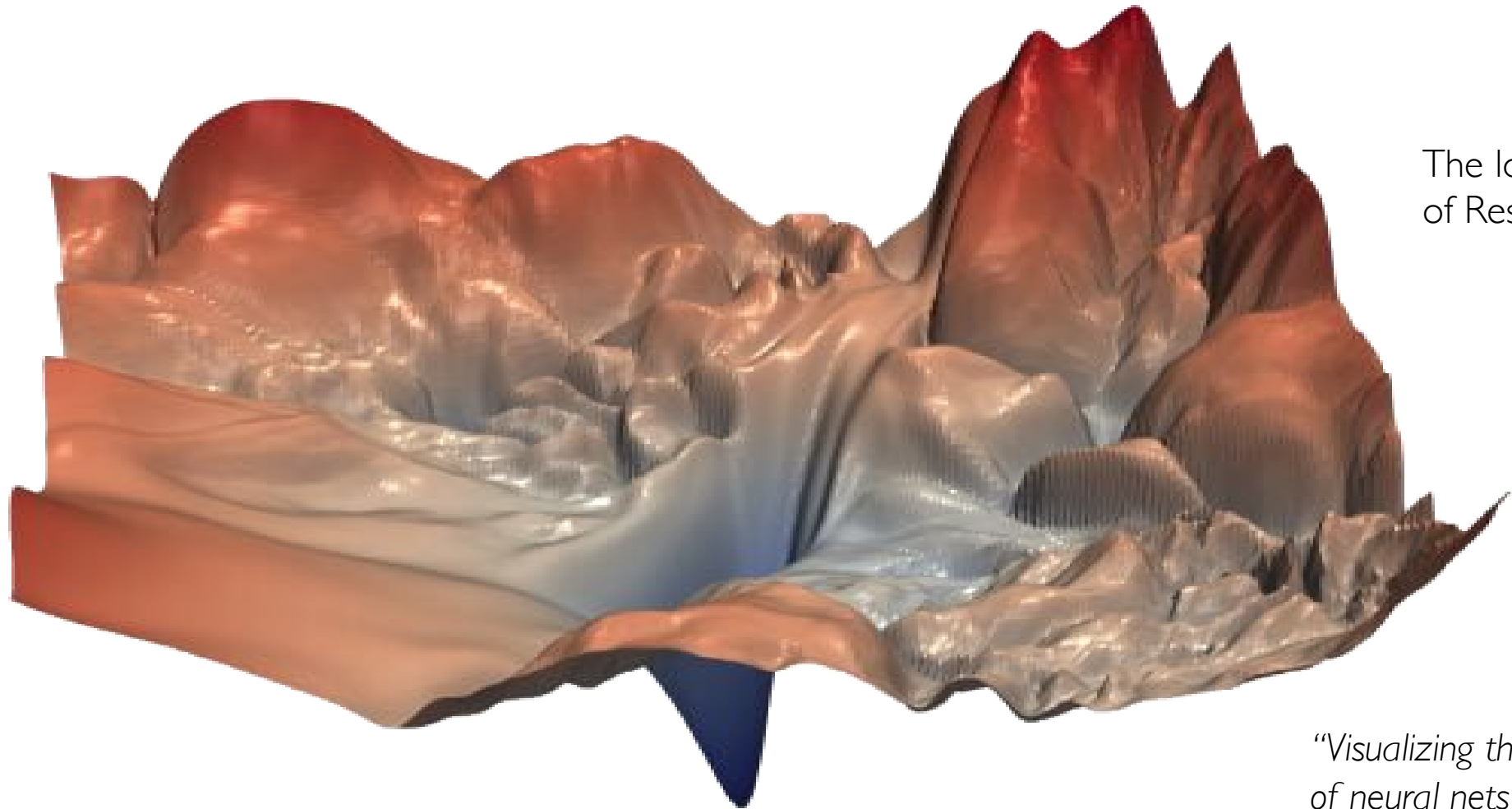
$$\frac{\partial J(\theta)}{\partial \theta_1} = \frac{\partial J(\theta)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial \theta_1}$$



Neural Networks in Practice: Optimization

Training Neural Networks is Difficult

non-convex
local minima



The loss surfaces
of Res-Net56

*“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

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

How can we set the
learning rate?

Setting the Learning Rate

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.

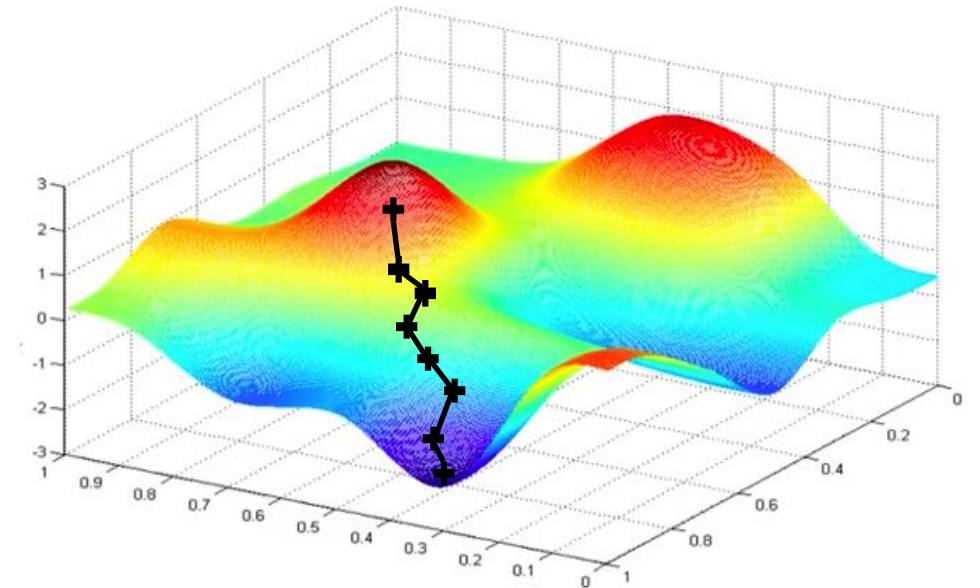
Additional details: <http://ruder.io/optimizing-gradient-descent/>

Neural Networks in Practice: Mini-batches

Gradient Descent

Algorithm

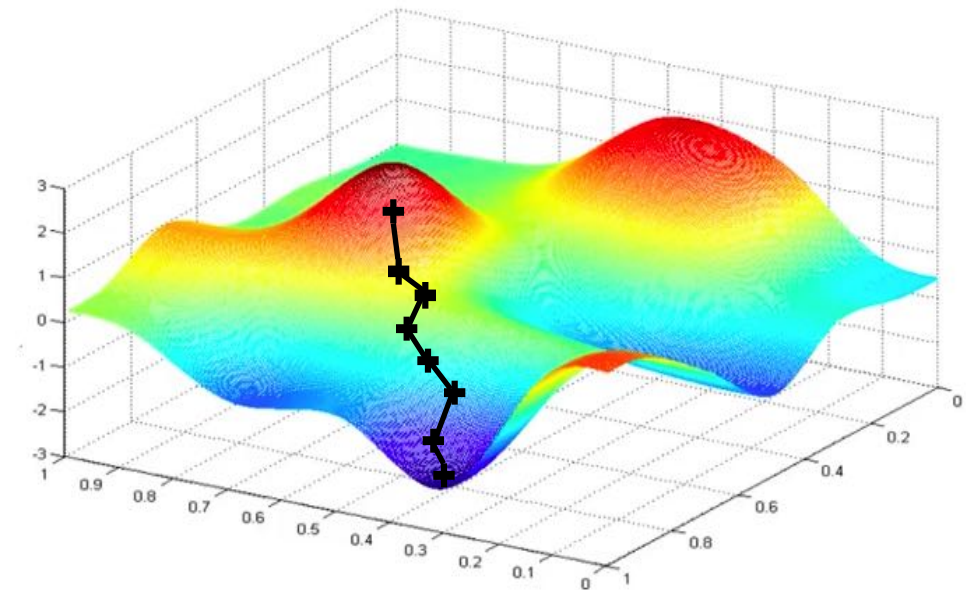
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights



Gradient Descent

Algorithm

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

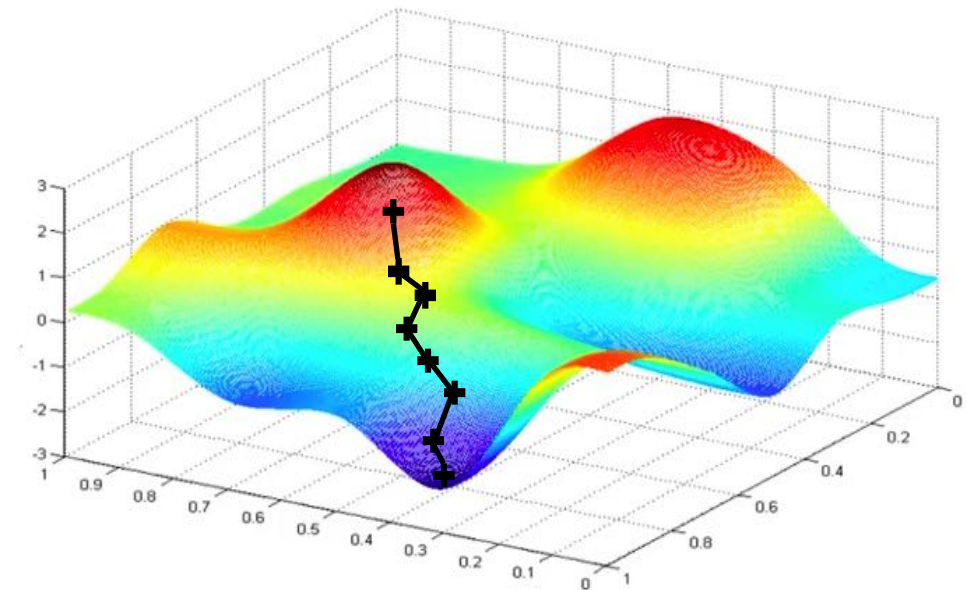


Can be very
computational to
compute!

Stochastic Gradient Descent

Algorithm

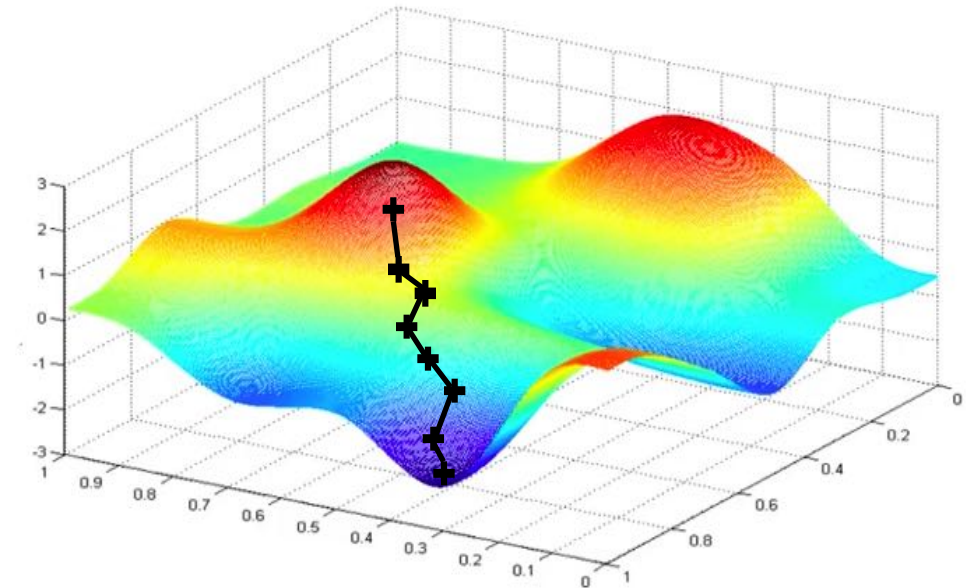
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(\theta)}{\partial \theta}$
5. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
6. Return weights



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, $\frac{\partial J_i(\theta)}{\partial \theta}$
5. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
6. Return weights

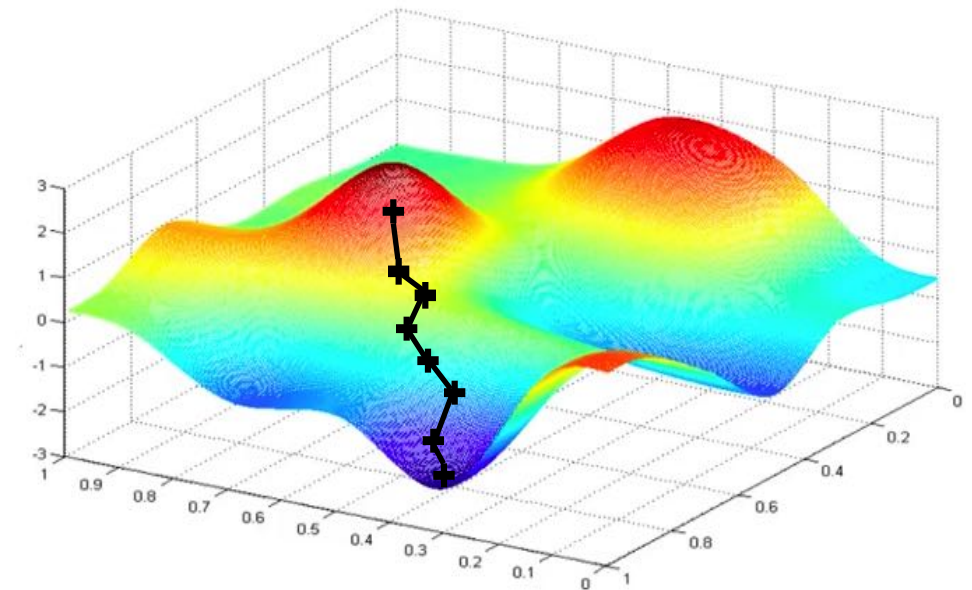


Easy to compute but
very noisy
(stochastic)!

Stochastic Gradient Descent

Algorithm

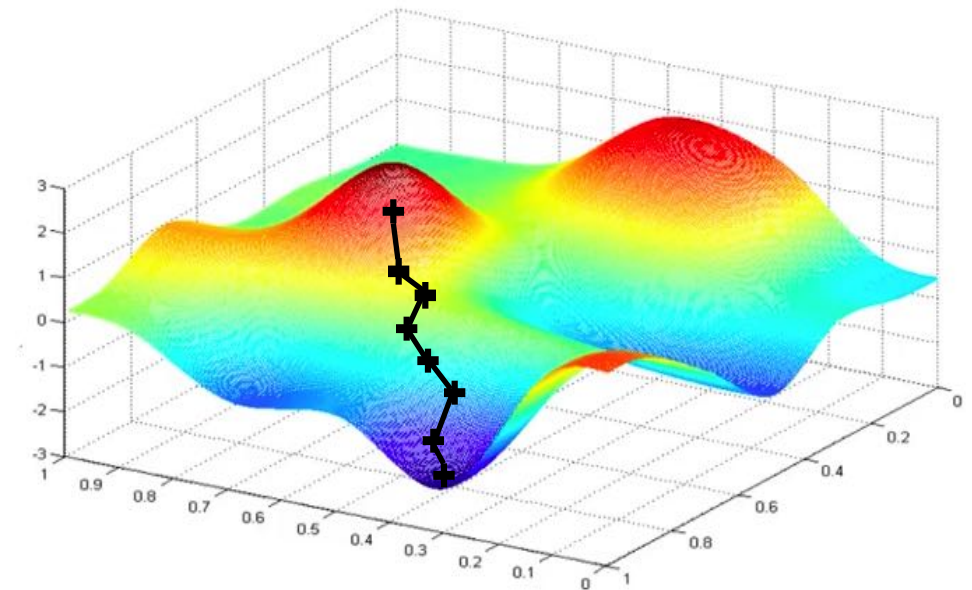
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\theta)}{\partial \theta}$
5. Update weights, $\theta \leftarrow \theta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

Smother convergence
Allows for larger learning rates

Mini-batches while training

More accurate estimation of gradient

Smoother convergence

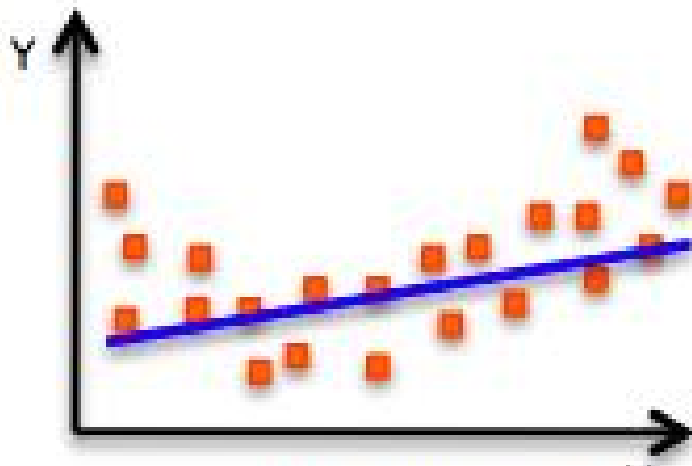
Allows for larger learning rates

Mini-batches lead to fast training!

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

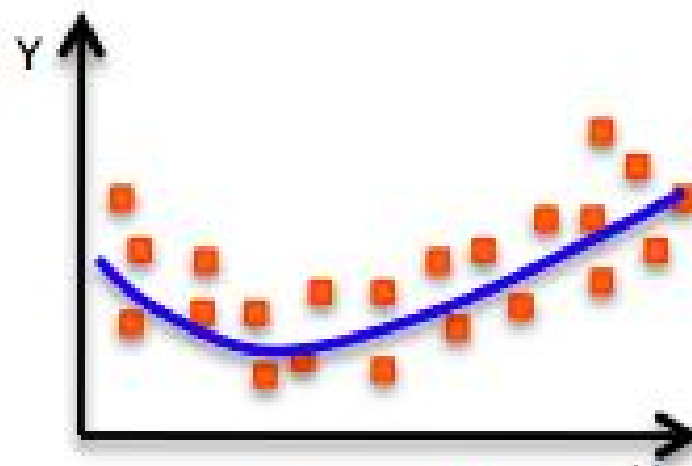
Neural Networks in Practice: Overfitting

The Problem of Overfitting

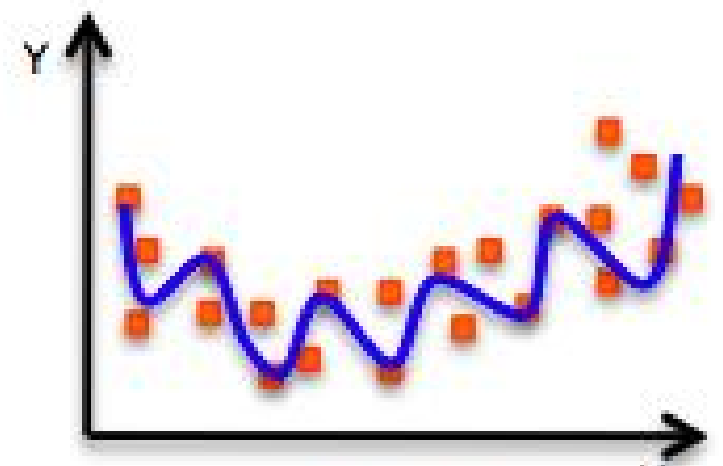


Underfitting

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?

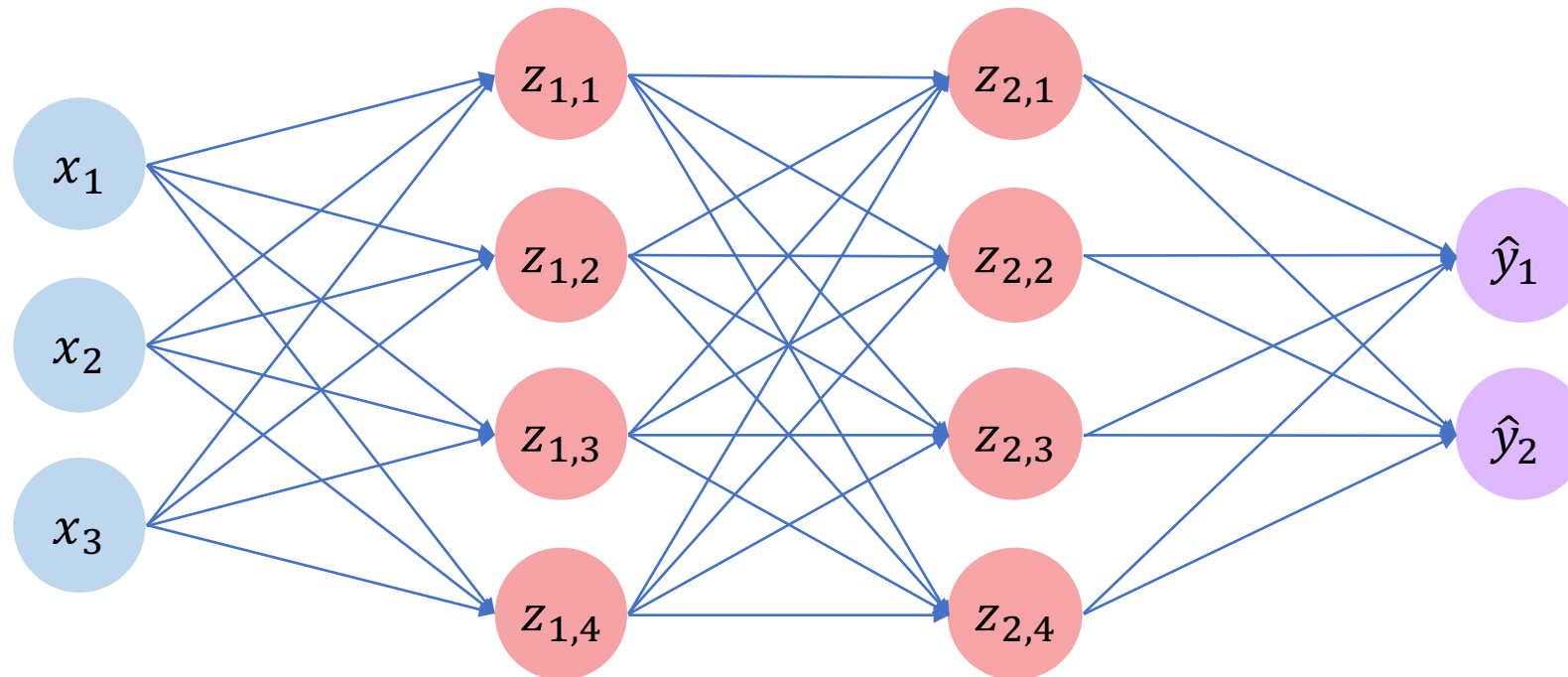
Technique that constrains our optimization problem to discourage complex models

Why do we need it?

Improve generalization of our model on unseen data

Regularization I: Dropout

- During training, randomly set some activations to 0

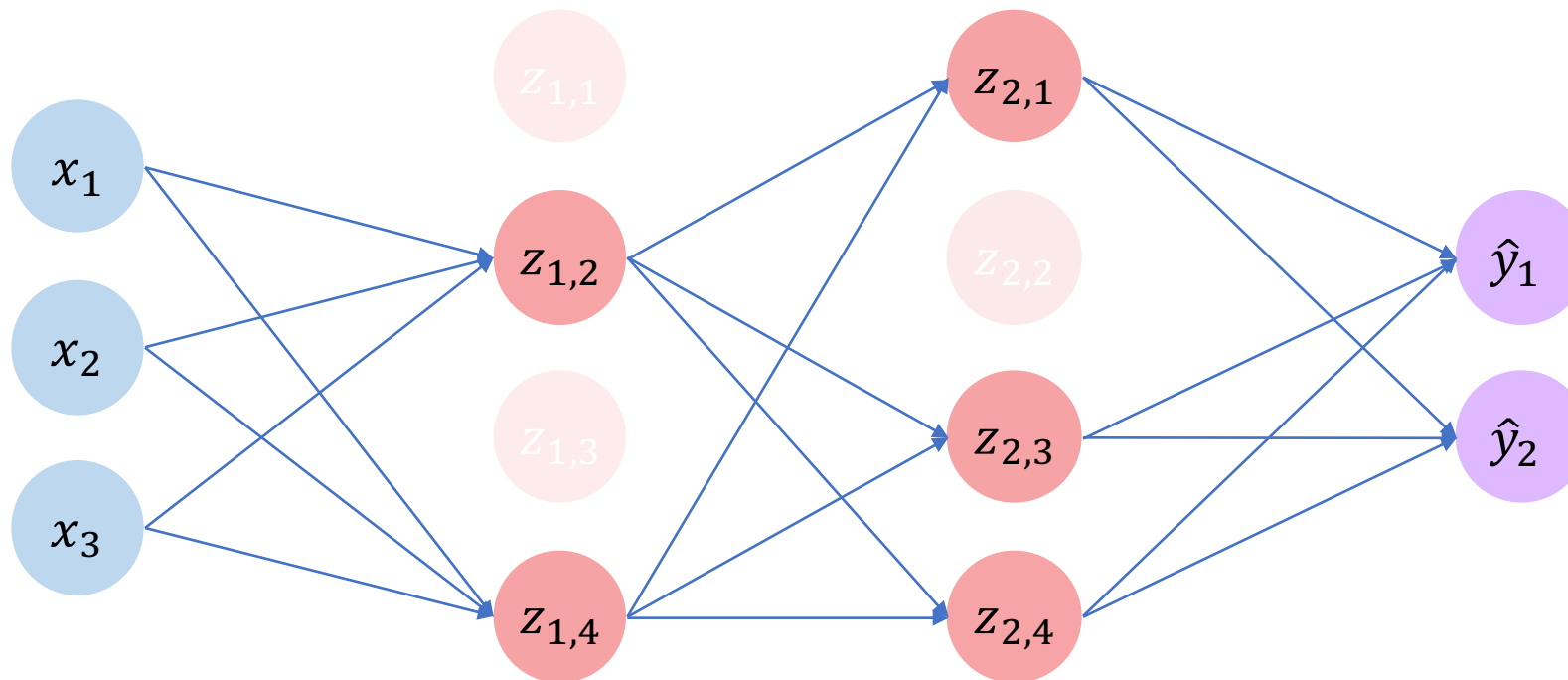


Regularization I: Dropout

- 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)
```

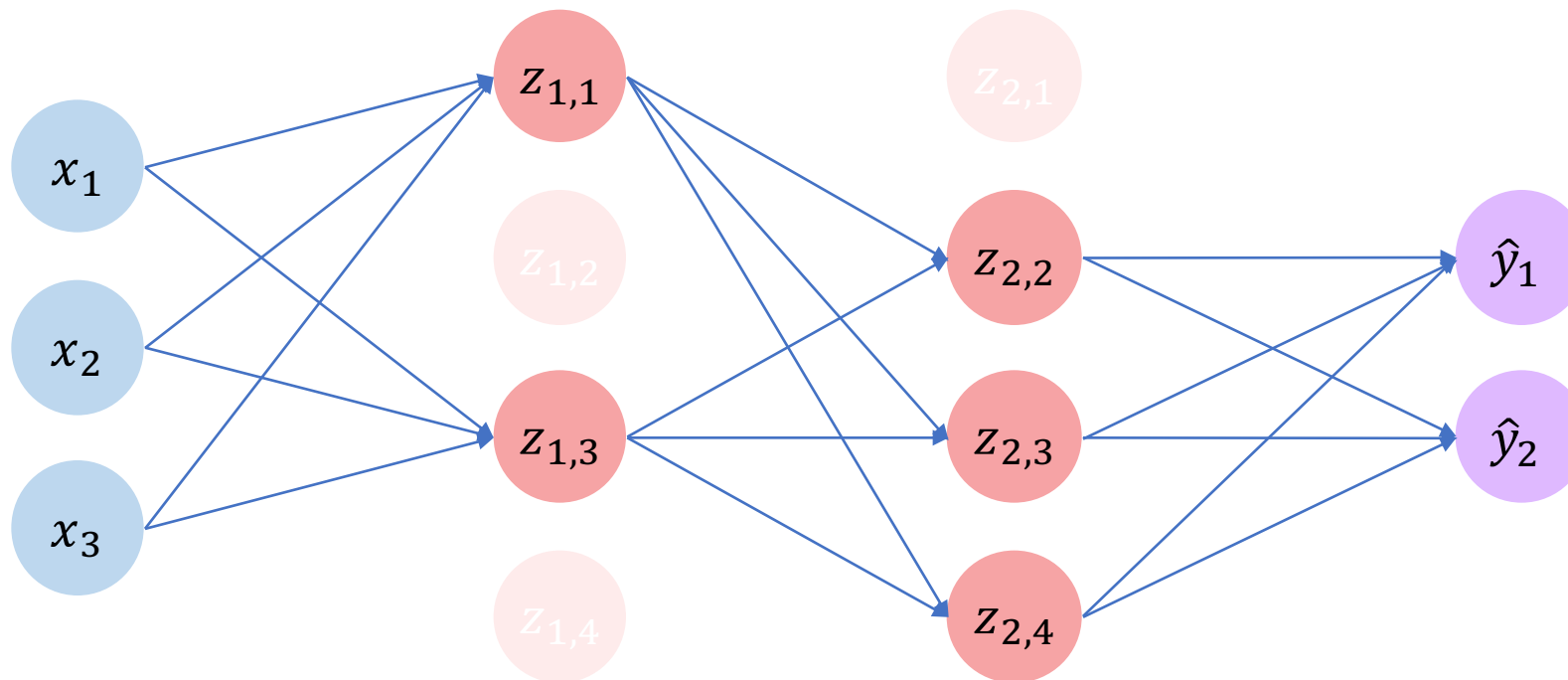


Regularization I: Dropout

- 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



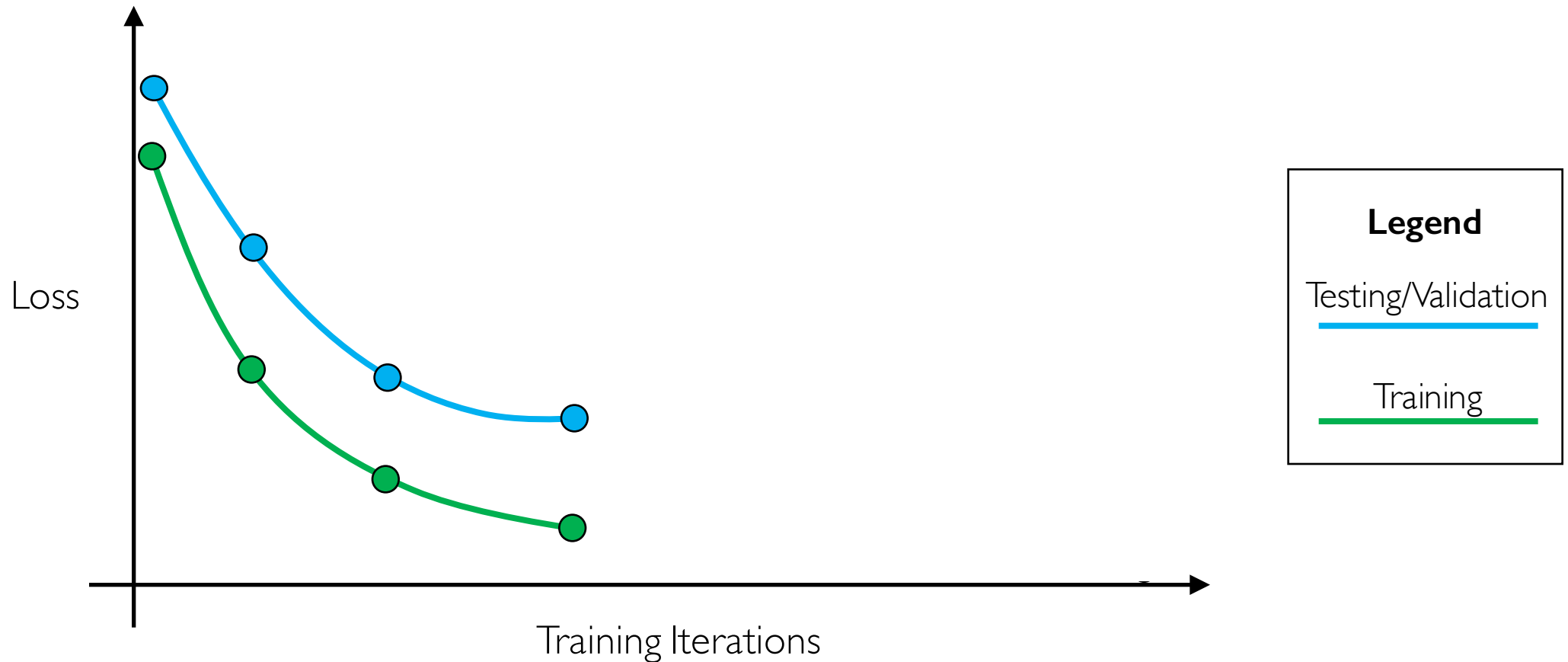
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



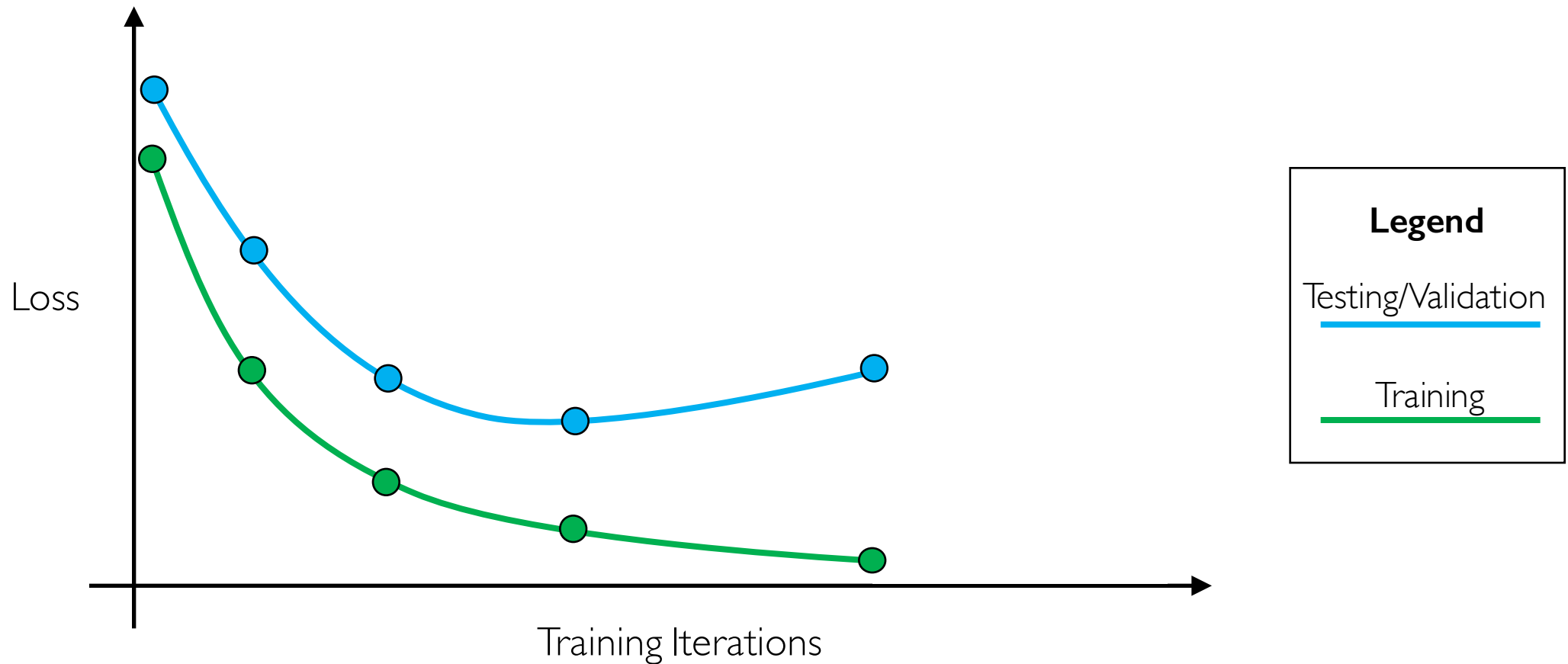
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



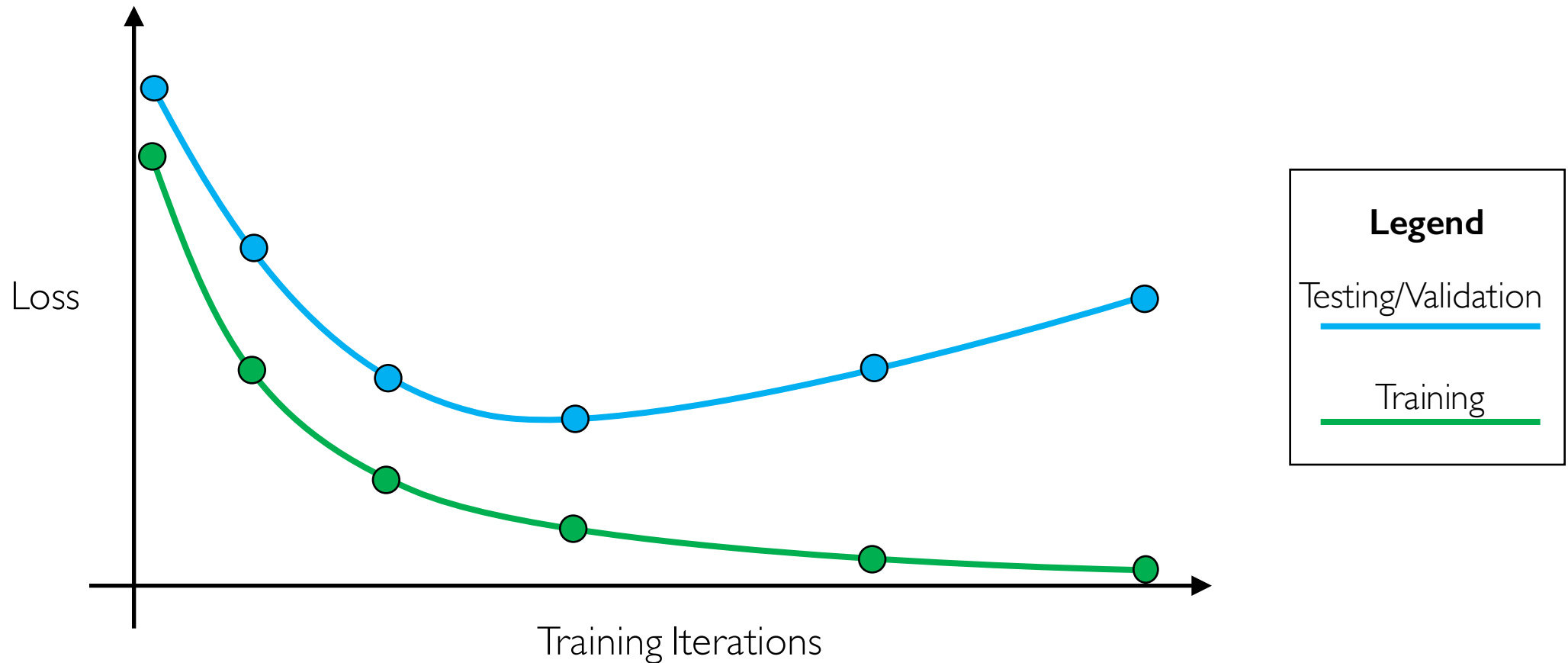
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



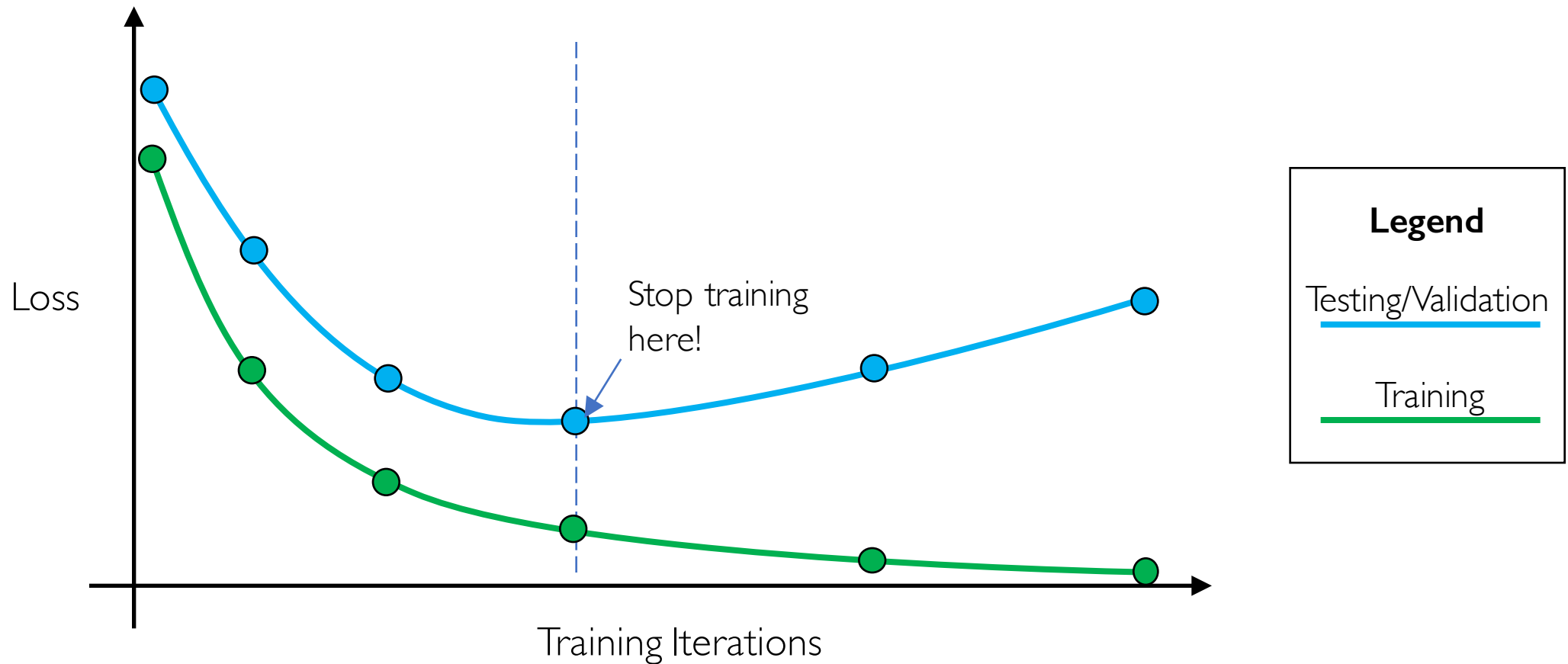
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



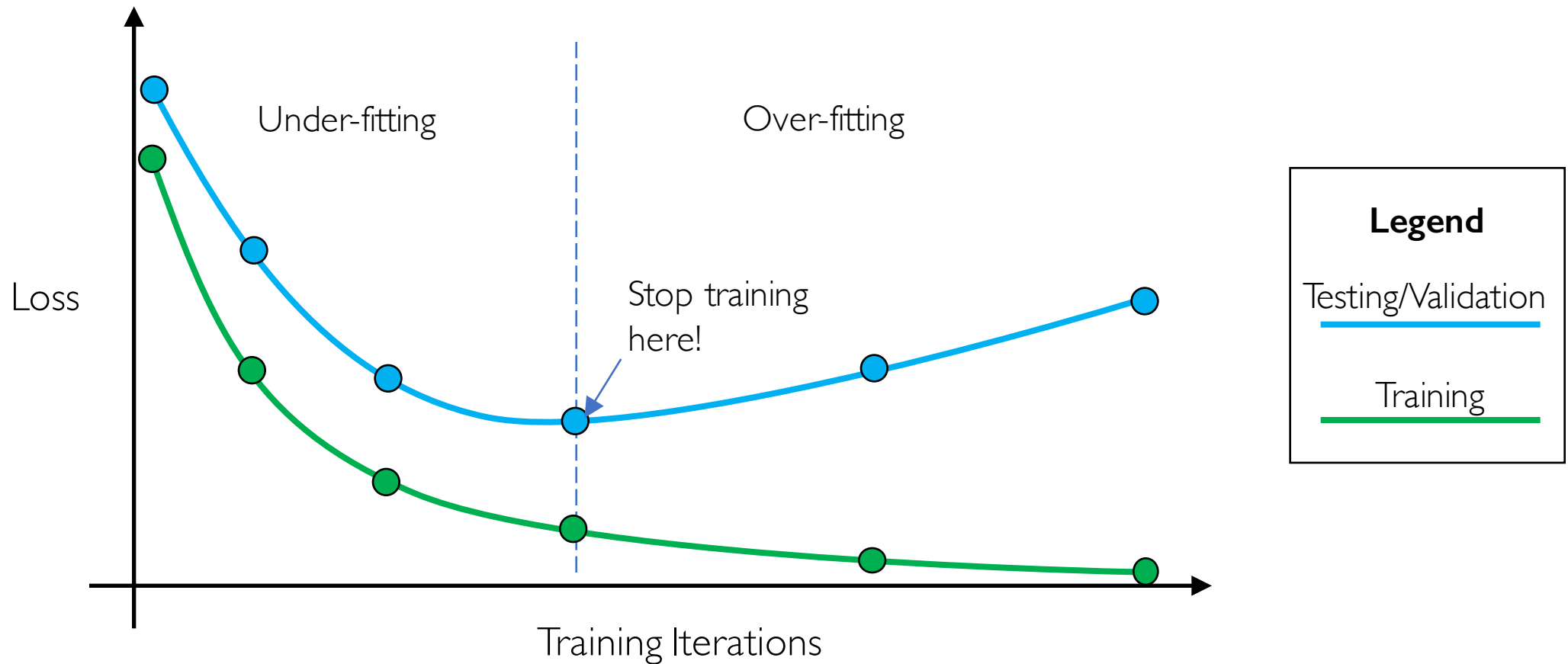
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



Regularization 2: Early Stopping

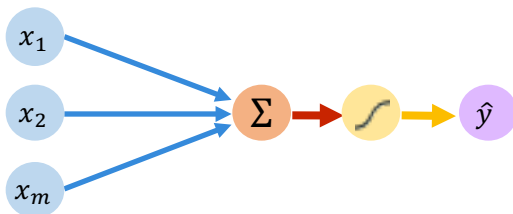
- Stop training before we have a chance to overfit



Core Foundation Review

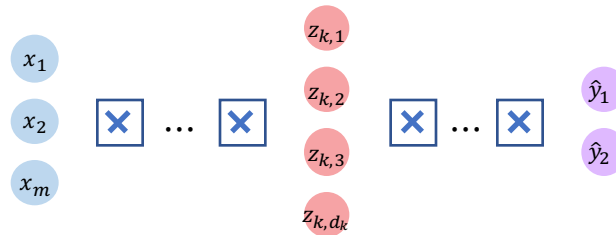
The Perceptron

- Structural building blocks
- Nonlinear activation functions



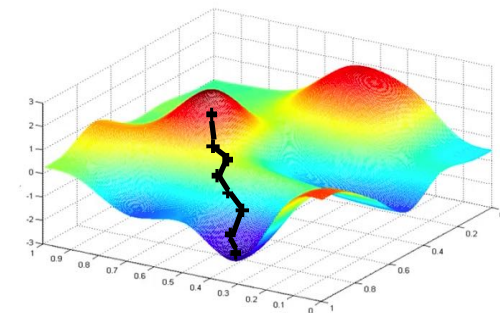
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization



Questions?