# Deep Learning Foundations

**References:**

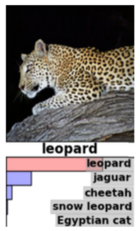
- *deeplearningbook.org*
- *https://d2l.ai/chapter_multilayer-perceptrons/index.html*
- *introtodeeplearning.com*
- *deeplearningindaba.com*
- *https://www.cs.toronto.edu/~rgrosse/courses/csc321_2018*
- *http://cs231n.github.io*
- Deep learning Tutorial, by Hung-yi Lee

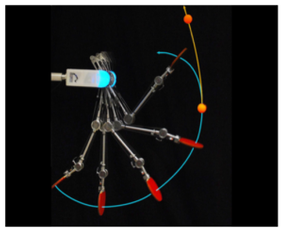
# Outline

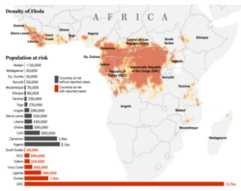
1. Definition of Machine Learning
2. Basic Structure of Perceptron
3. Building Neural Networks with Perceptrons
4. Applying Neural Networks
5. Training Neural Networks
6. Neural Networks in Practice: Optimization

# What is Machine Learning/Deep Learning

Improving a machine's **performance** at some **task** with **experience**







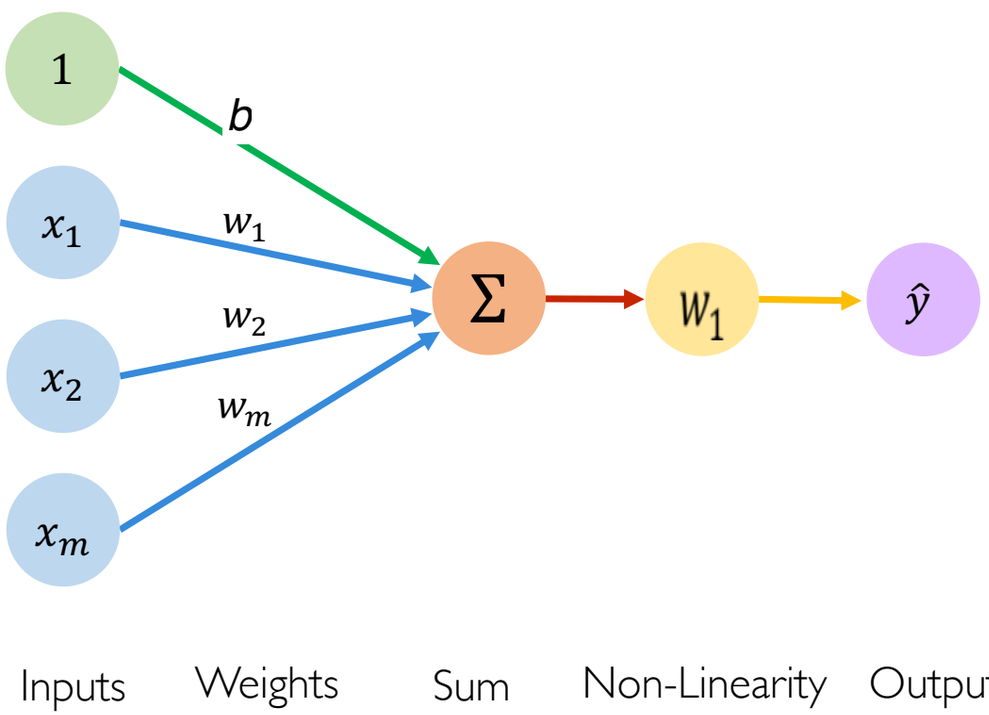

- Define task **T** - classification, regression
- Establish performance measure **P** - classification error rate, Mean Square Error (MSE)
- Provide some experience **E** - training examples

# The Perceptron

## The structural building block of deep learning

# The Perceptron: Forward Propagation

Perceptron:
- Feedforward neural networks, or
- A mathematical function mapping a set of input values to output values




Output

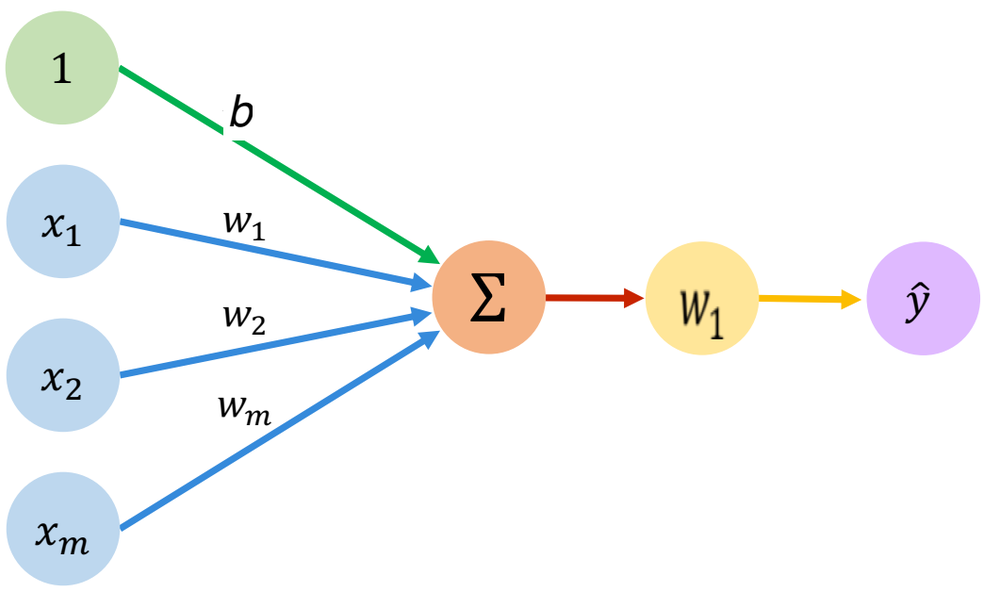
Linear combination of inputs

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

Non-linear activation function

Bias

# The Perceptron: Forward Propagation




Inputs Weights Sum Non-Linearity Output

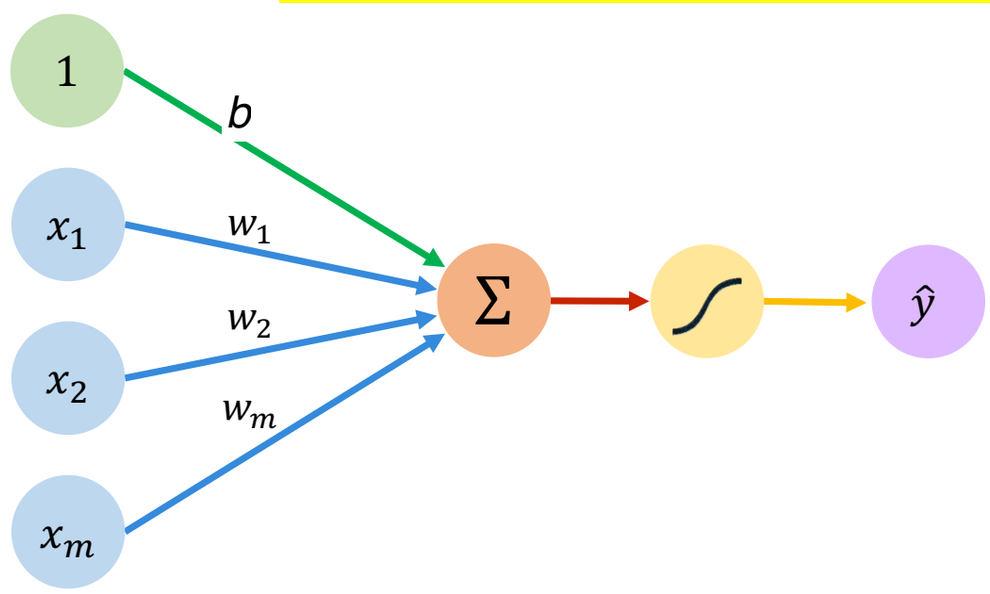
$$\hat{y} = g\left( b + \sum_{i=1}^{m} x_i \, w_i \right)$$

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

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

*Q: Why not use linear function for g?*



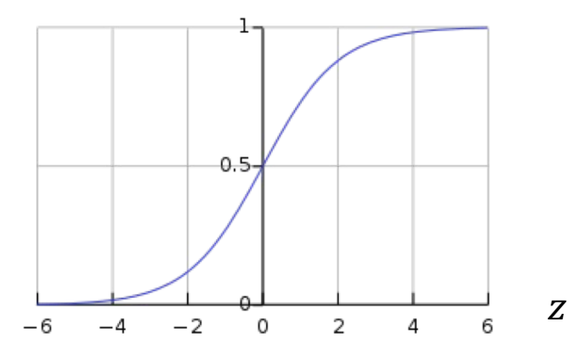

Inputs Weights Sum Non-Linearity Output

## Activation Functions

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

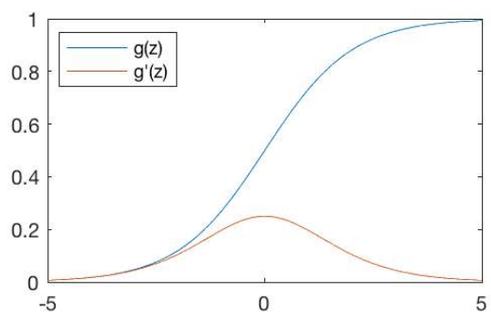
- Example: sigmoid function

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



# Common Activation Functions

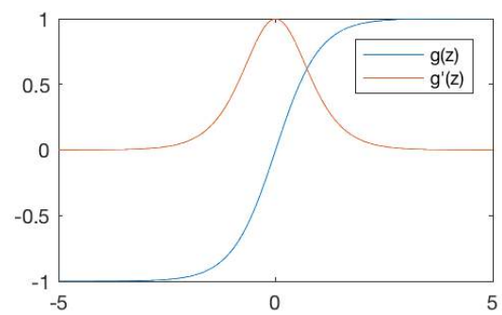
Sigmoid Function




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

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

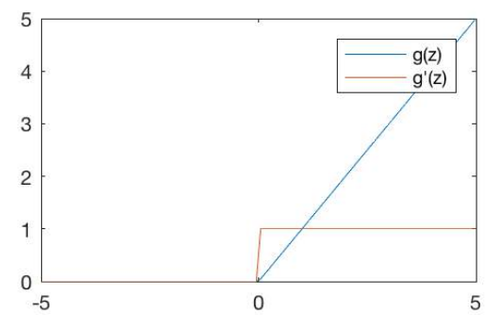
Hyperbolic Tangent




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

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

Rectified Linear Unit (ReLU)




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

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

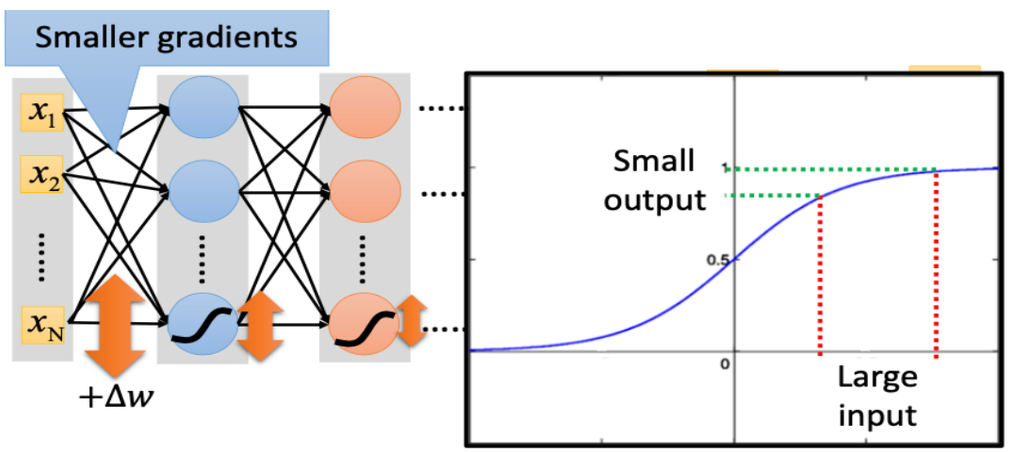
*Notice gradient vanishes when z gets small or big*

NOTE: All activation functions are non-linear



**Discussion: in a very deep multilayer perceptron (e.g. n>5), do you think it is a good idea to use sigmoid or tanh as the activation function of all hidden layers?**

# Activation Functions and Vanishing Gradient Problem



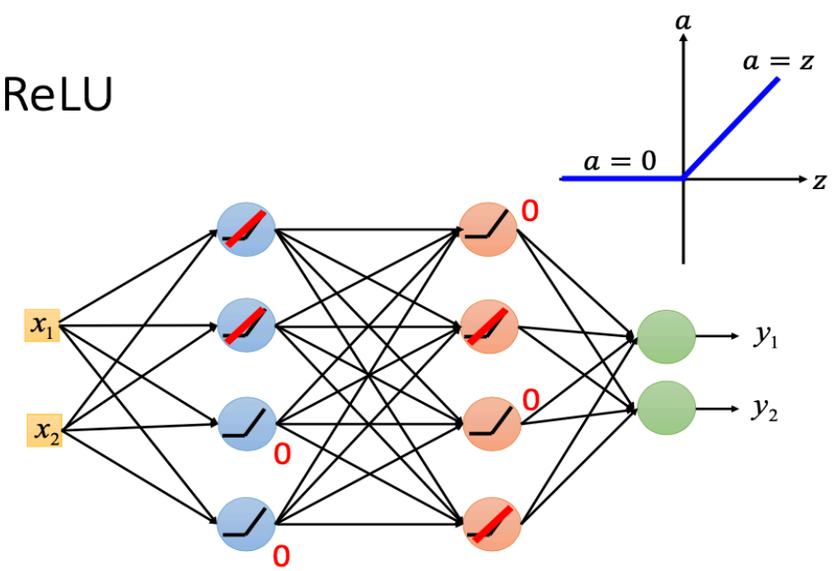

Intuitive way to compute the derivatives …

$$\frac{\partial l}{\partial w} =? \frac{\Delta l}{\Delta w}$$

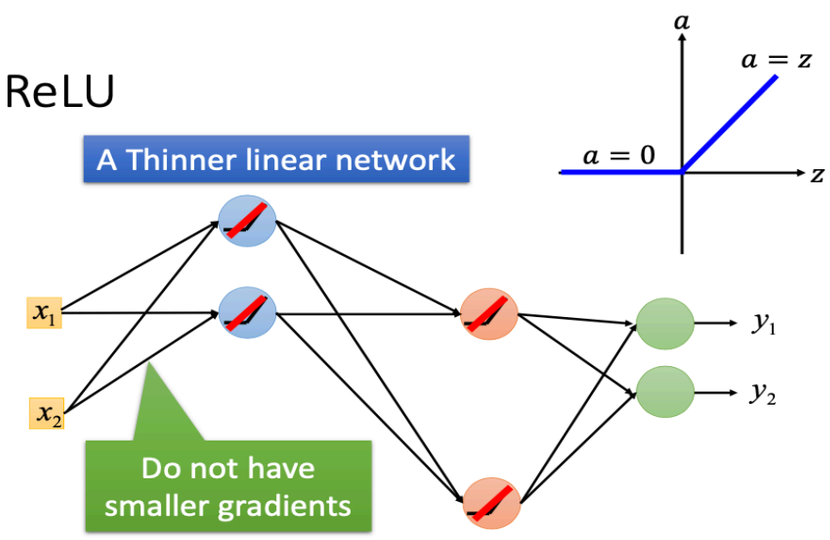
**Gradient Vanishing**:

- Small Gradients
- Learning very slow

ReLU

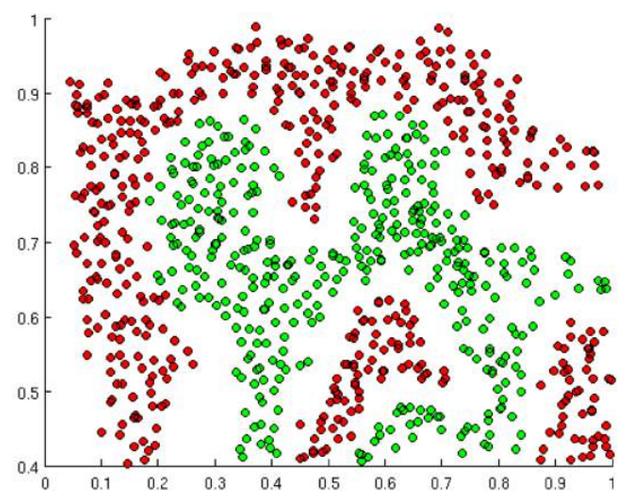



ReLU



# Importance of Activation Functions

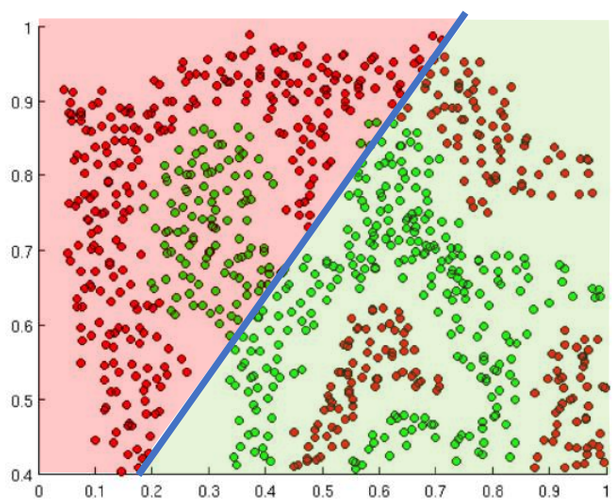
*The purpose of activation functions is to* ***introduce non-linearities*** *into the network*




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

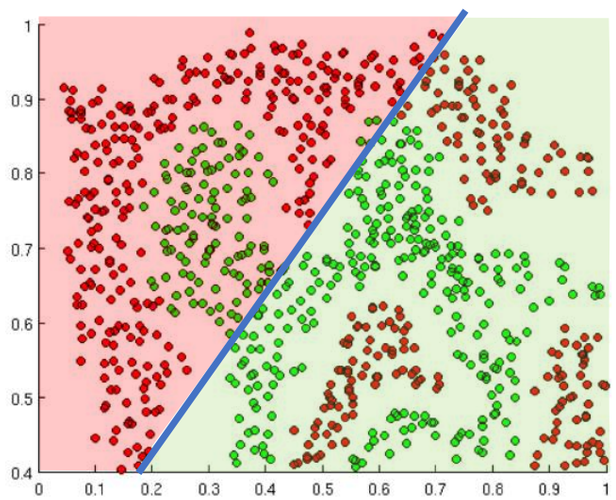
**Universal Approximation Theorem**

Regardless of what function we are trying to learn, there exists a large multllayer perceptron (MLP) which can represent this function

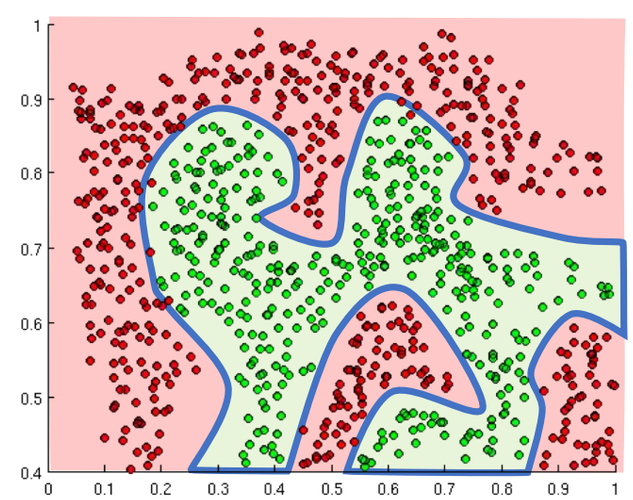
However, it is not guaranteed that a training algorithm will be able to learn that function

# Importance of Activation Functions

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

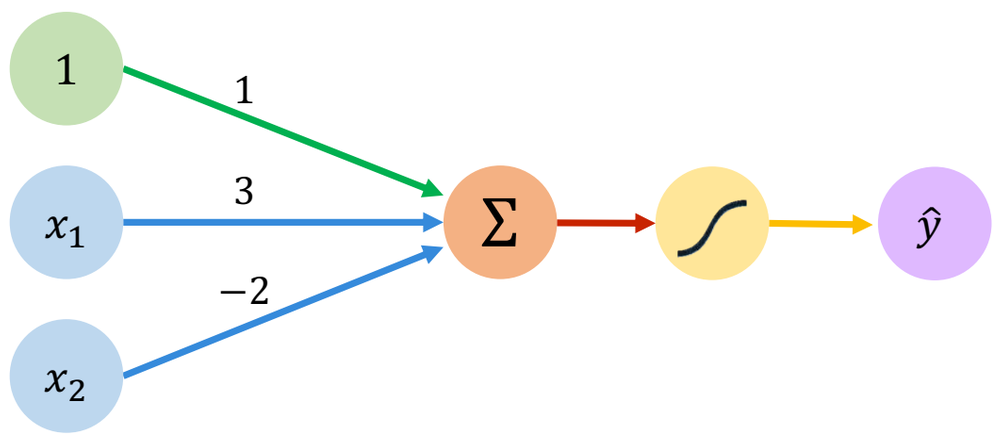



Linear Activation functions produce linear decisions no matter the network size




Non-linearities allow us to approximate arbitrarily complex functions

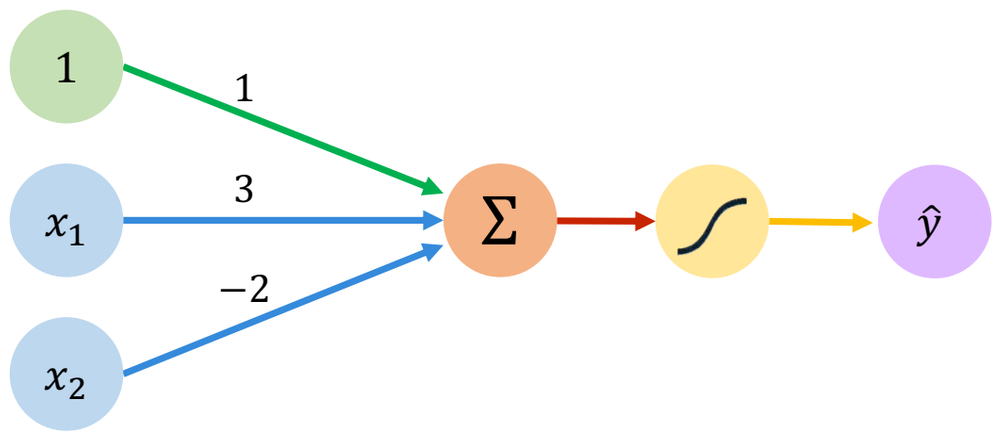
# The Perceptron: Example



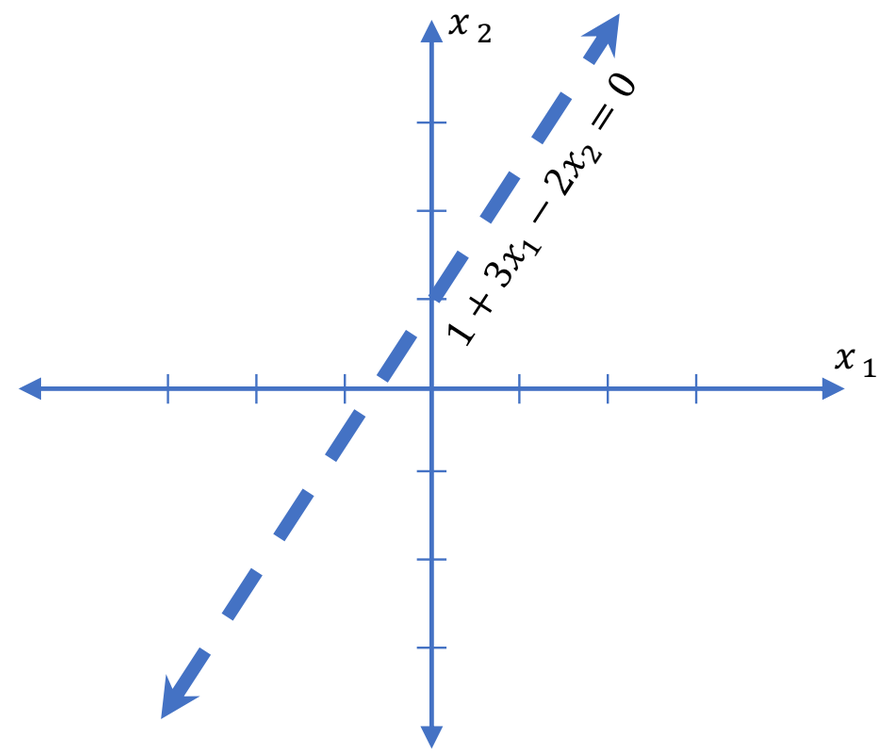

We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g(b + \boldsymbol{X}^T \boldsymbol{W})$$
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$
$$\hat{y} = g(\underbrace{1 + 3x_1 - 2x_2}_{\text{This is just a line in 2D!}})$$

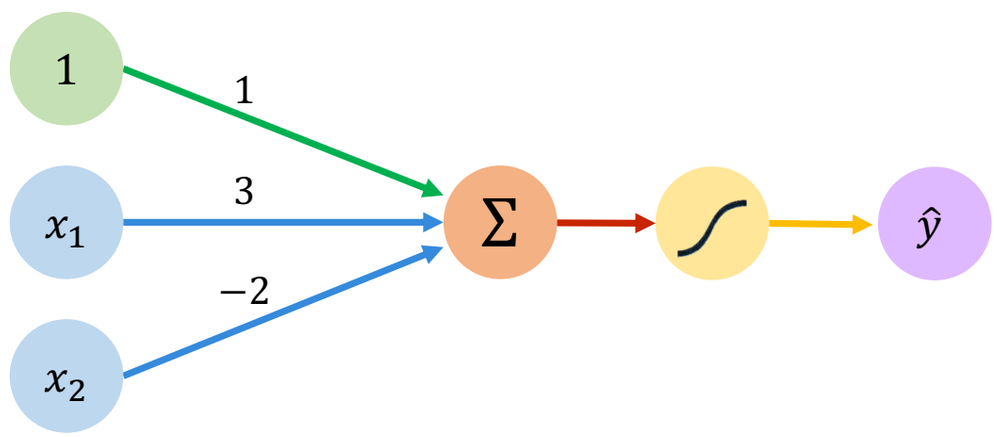
# The Perceptron: Example



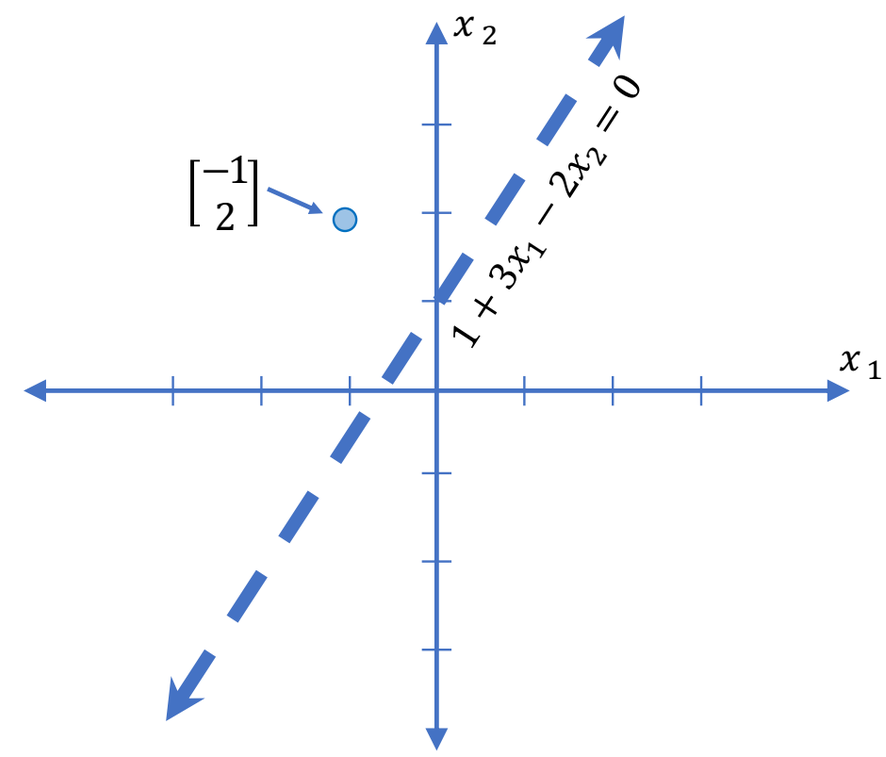

$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



# The Perceptron: Example



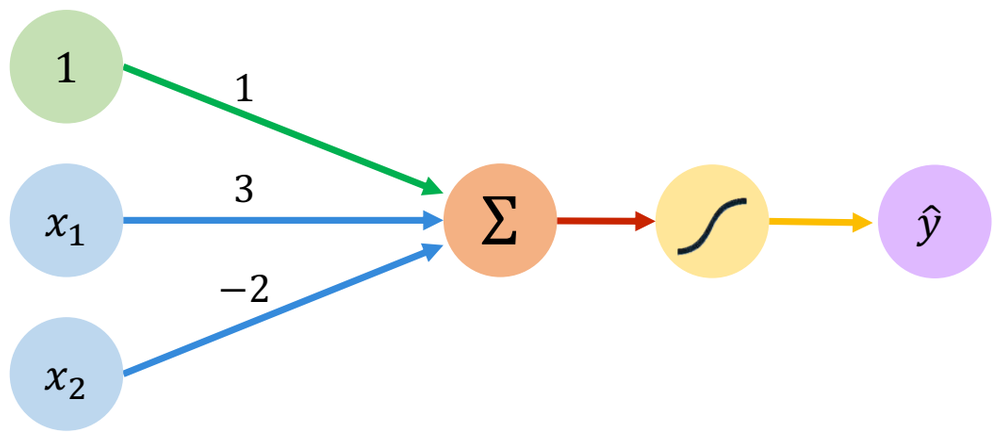

$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



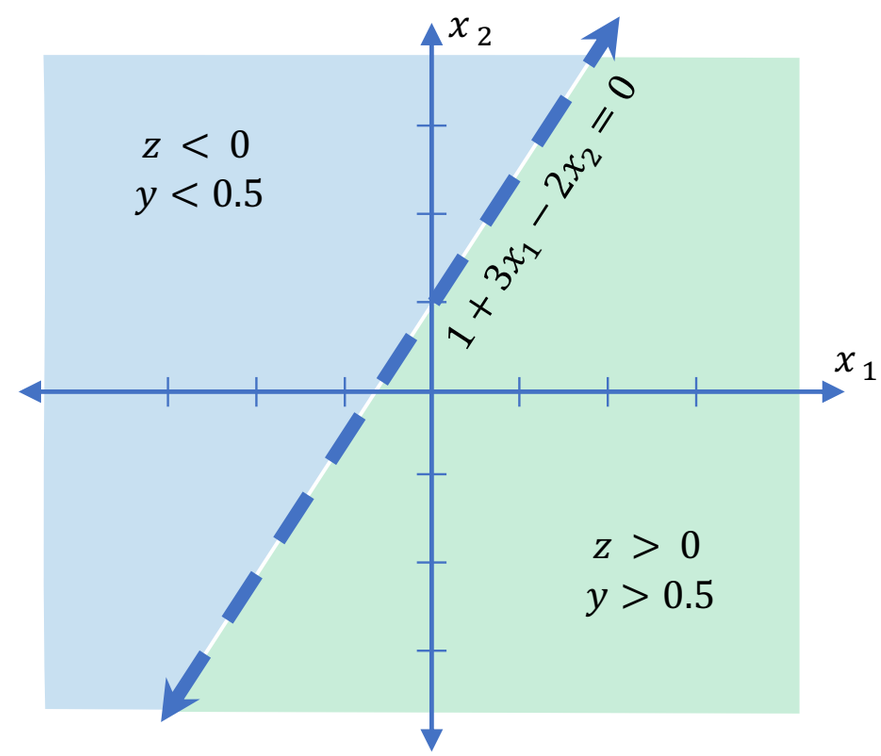

Assume we have input: $\boldsymbol{X} = \begin{bmatrix}-1\\2\end{bmatrix}$

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

# The Perceptron: Example

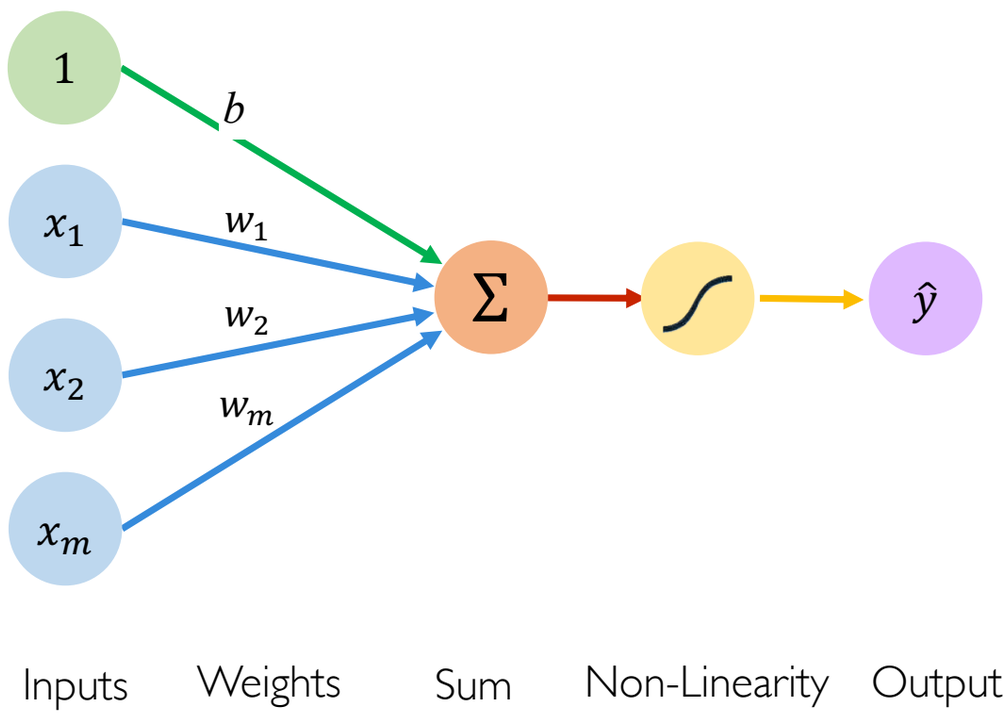



$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

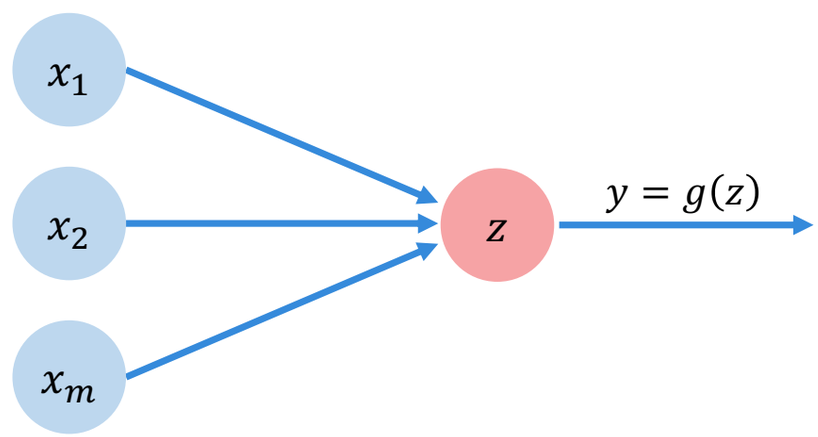


# Building Neural Networks with Perceptrons

# The Perceptron: Simplified

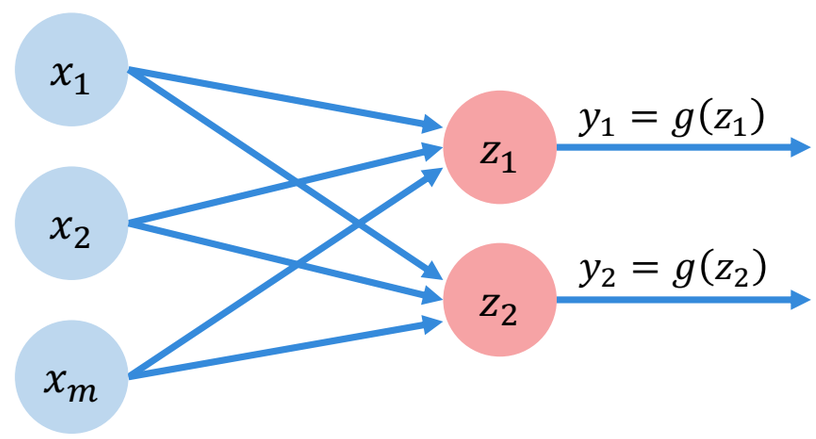


# The Perceptron: Simplified



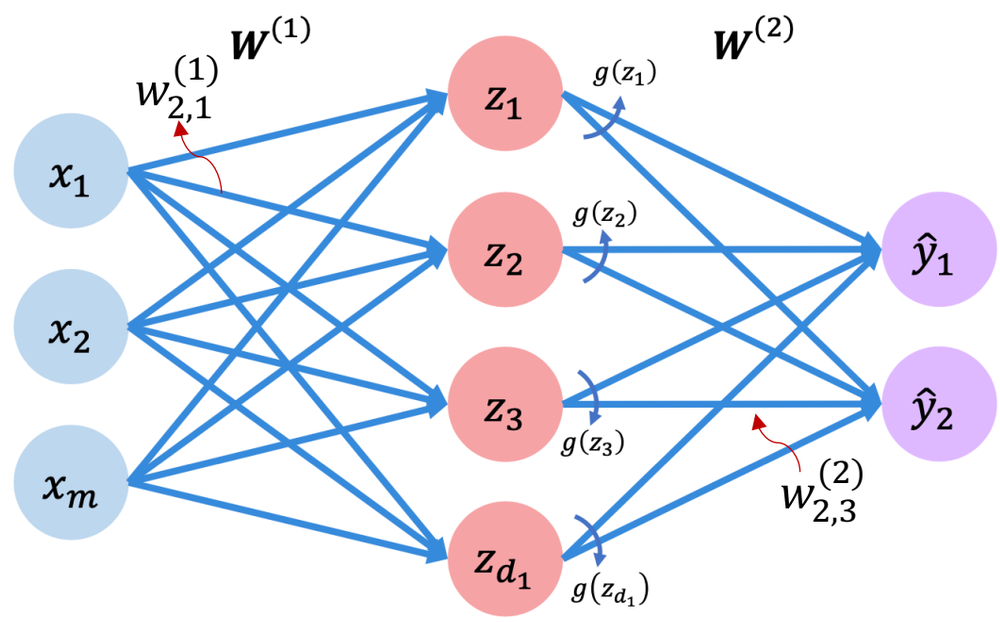

$$z = \quad b + \sum_{j=1}^{m} x_j \, w_j$$

# Multi Output Perceptron




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

# Single Layer Neural Network




$g$: Activation Function

Sigmoid vs. Softmax

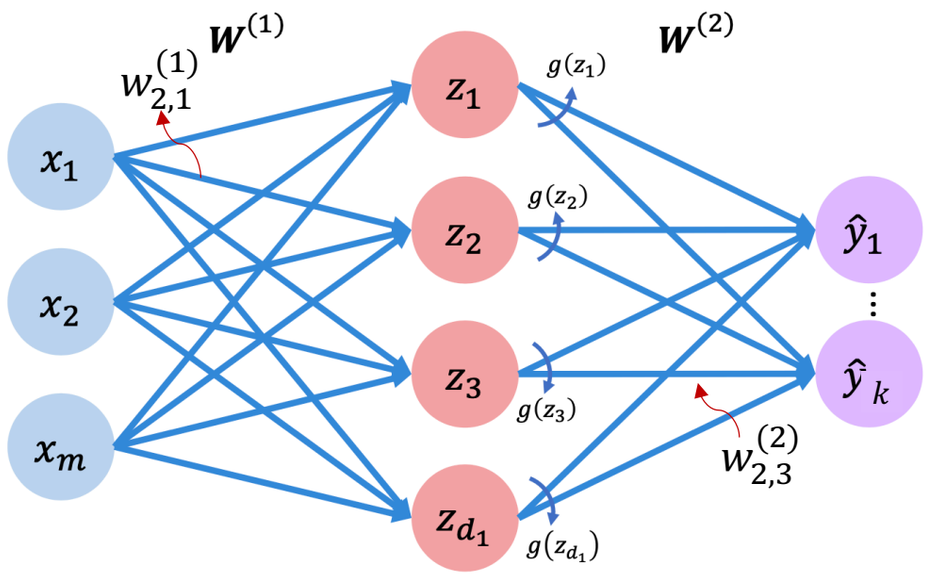
$$z_i = g^{(1)}(b_i^{(1)} + \sum_{j=1}^{m} x_j\, w_{i,j}^{(1)})$$

$$e.g.\ z_2 = ReLU(b_2^{(1)} + w_{2,1}^{(1)} x_1 + w_{2,2}^{(1)} x_2 + w_{2,m}^{(1)} x_m)$$

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

$$\hat{y}_2 = \sigma\,(b_2^{(2)} + w_{2,1}^{(2)} z_1 + w_{2,2}^{(2)} z_2 + w_{2,3}^{(2)} z_3 + w_{2,d_1}^{(2)} z_{d_1})$$

# Sigmoid vs. Softmax




Let $Z = [z_1, z_2, z_3, \dots,]$,

$$W_i = \left[w_{i,1}^{(2)}, w_{i,2}^{(2)}, w_{i,3}^{(2)}, \dots,\right],$$

**(1) Sigmoid**

e.g.

$$\hat{y}_1 = \frac{1}{1 + e^{-(W_1^T Z + b_1)}} \qquad 0 < \hat{y}_1 < 1 \qquad \textbf{0.88}$$

$$\hat{y}_k = \frac{1}{1 + e^{-(W_k^T Z + b_k)}} \qquad 0 < \hat{y}_k < 1 \qquad \textbf{0.65}$$

- With shared input, each y makes independent output
- Use it for **Multi-label classification**, e.g., classify an article into multiple possible topics

**(2) Softmax**

e.g.

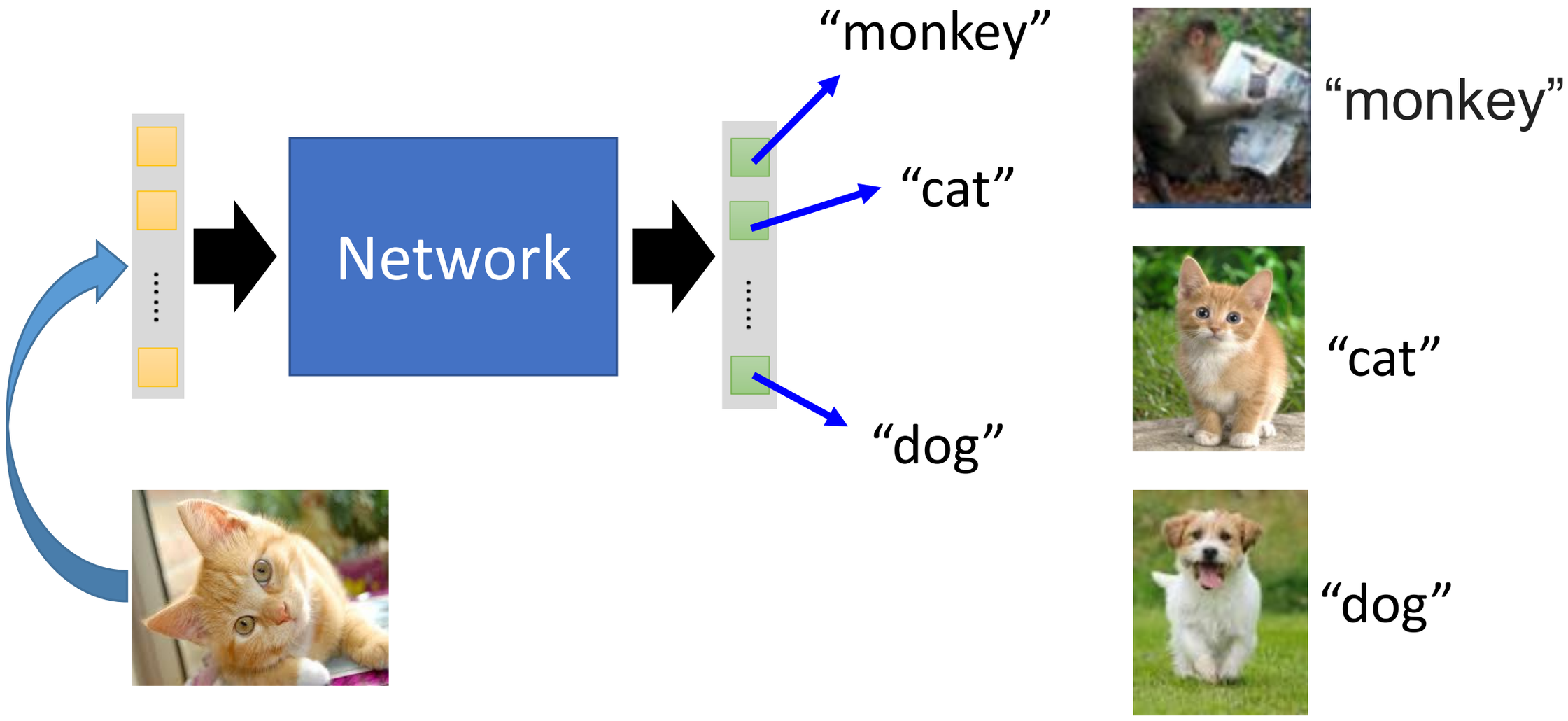
$$\hat{y}_1 = \frac{e^{-(W_1^T Z + b_1)}}{\sum_i e^{-(W_i^T Z + b_i)}} \qquad 0 < \hat{y}_1 < 1 \qquad \textbf{0.75}$$

$$\hat{y}_k = \frac{e^{-(W_k^T Z + b_k)}}{\sum_i e^{-(W_i^T Z + b_i)}} \qquad 0 < \hat{y}_k < 1 \qquad \textbf{0.25}$$

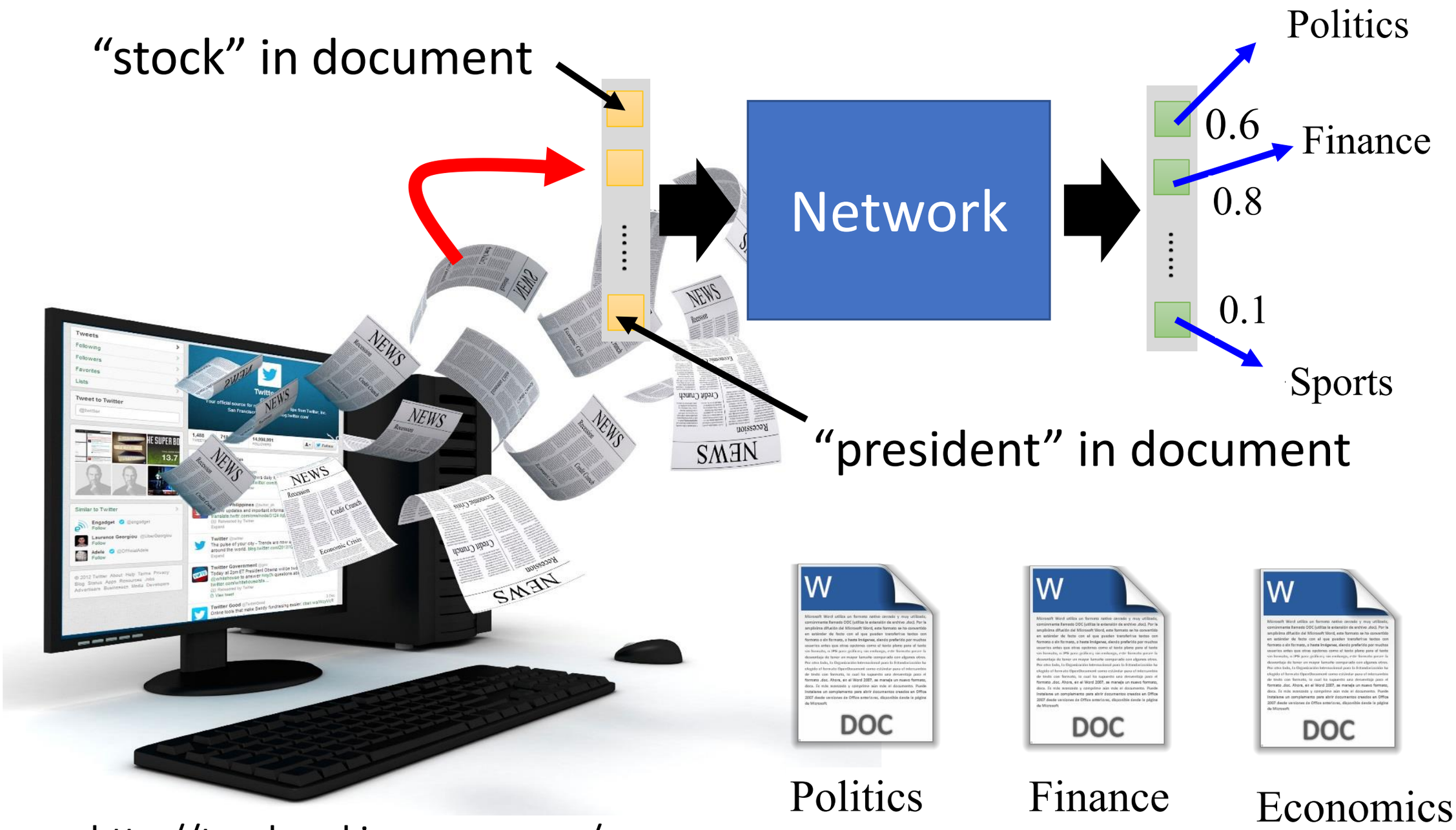
- $\sum_k \hat{y}_k = \mathbf{1}$
- Use it for **Single-label classification** (but more than 2 classes), e.g., classify an article into a unique topic
- The final label can be assigned to the y with largest value

# For example, you can do …….

- Image Recognition

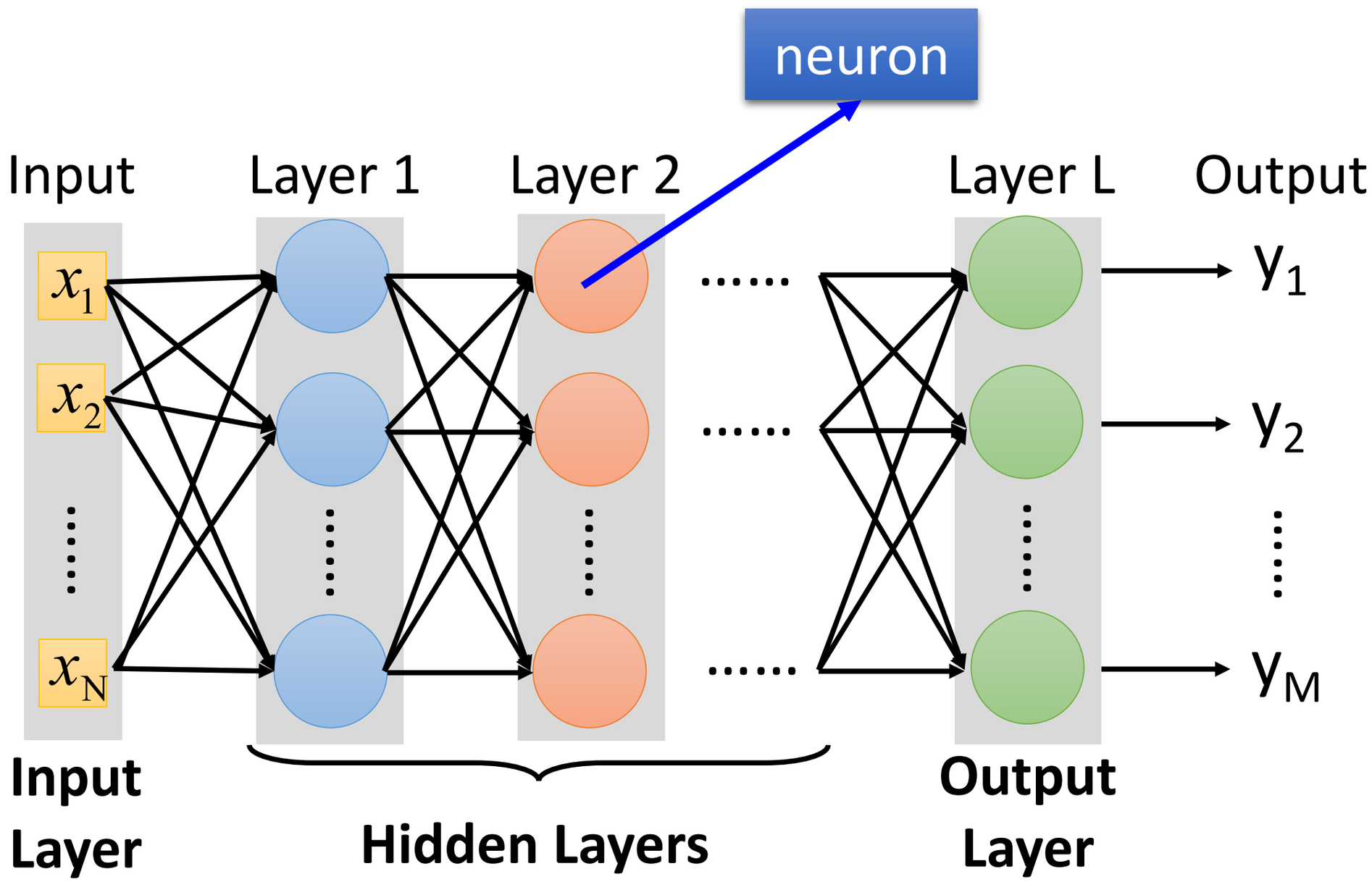


# For example, you can do …….




http://top-breaking-news.com/

# Deep Neural Network




Deep means many hidden layers

# Applying Neural Networks

# Example Problem

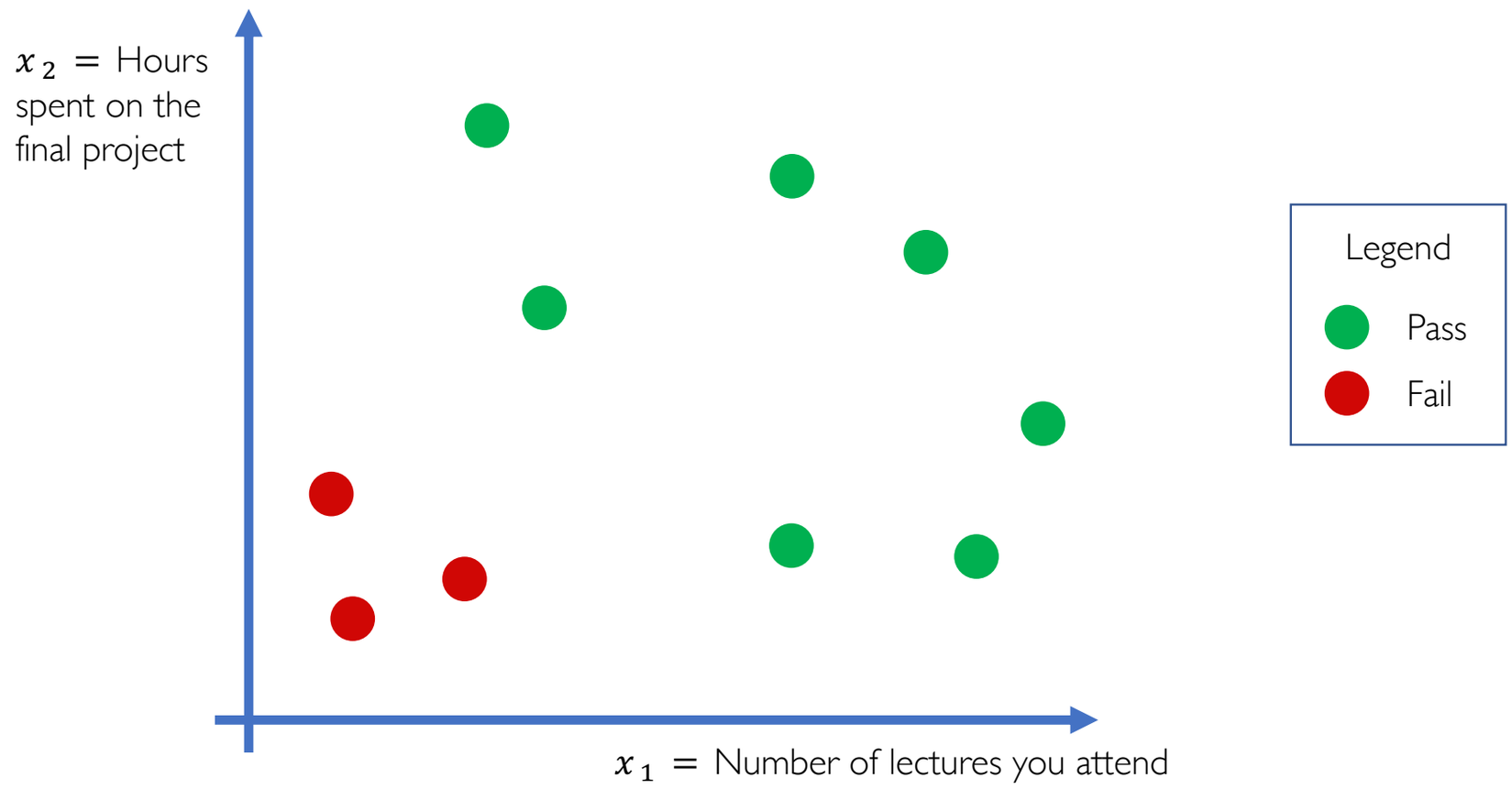
Will I pass this class?

Let's start with a simple two feature model

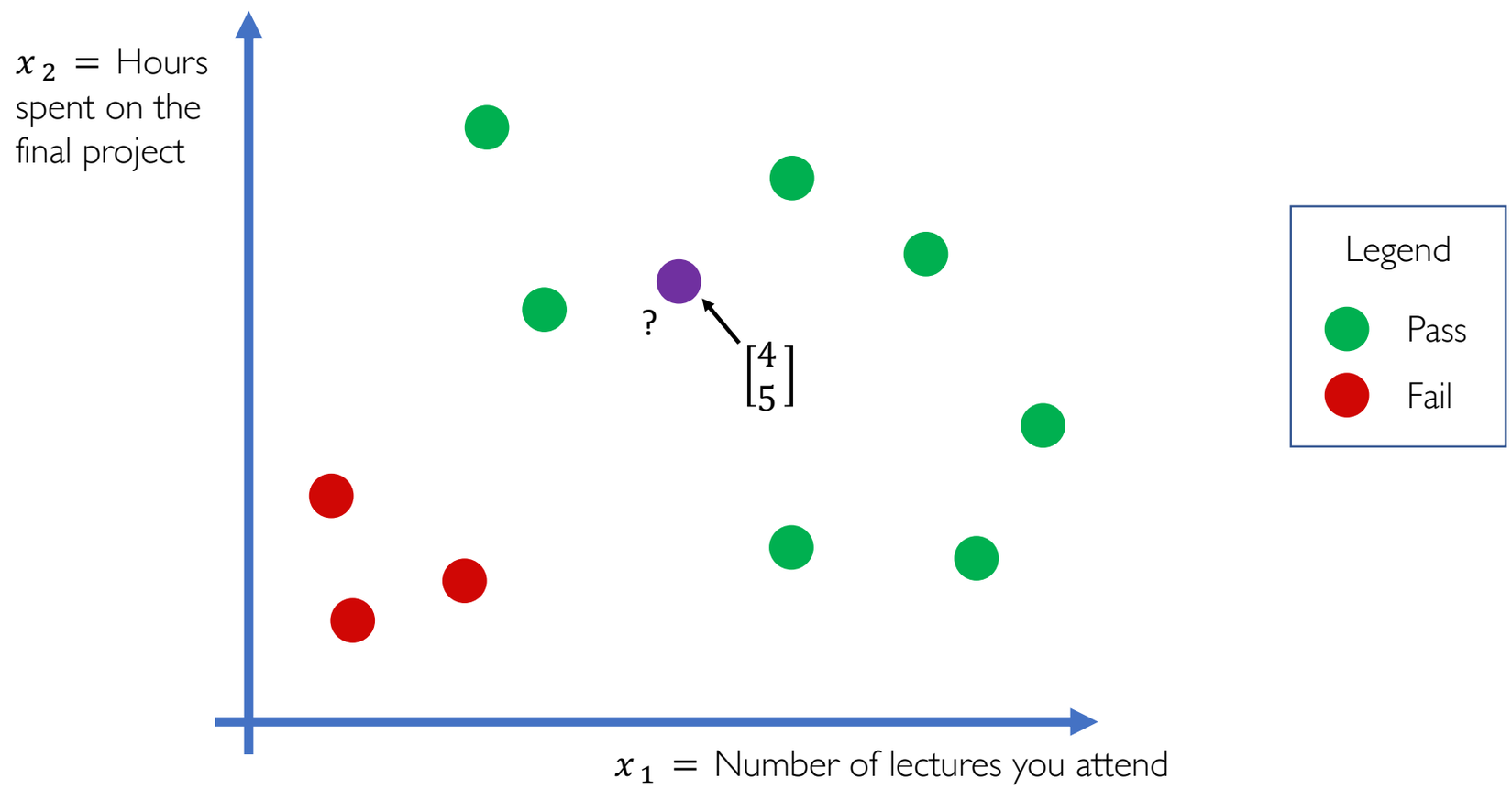
$x_1$ = Number of lectures you attend

$x_2$ = Hours spent on the final project

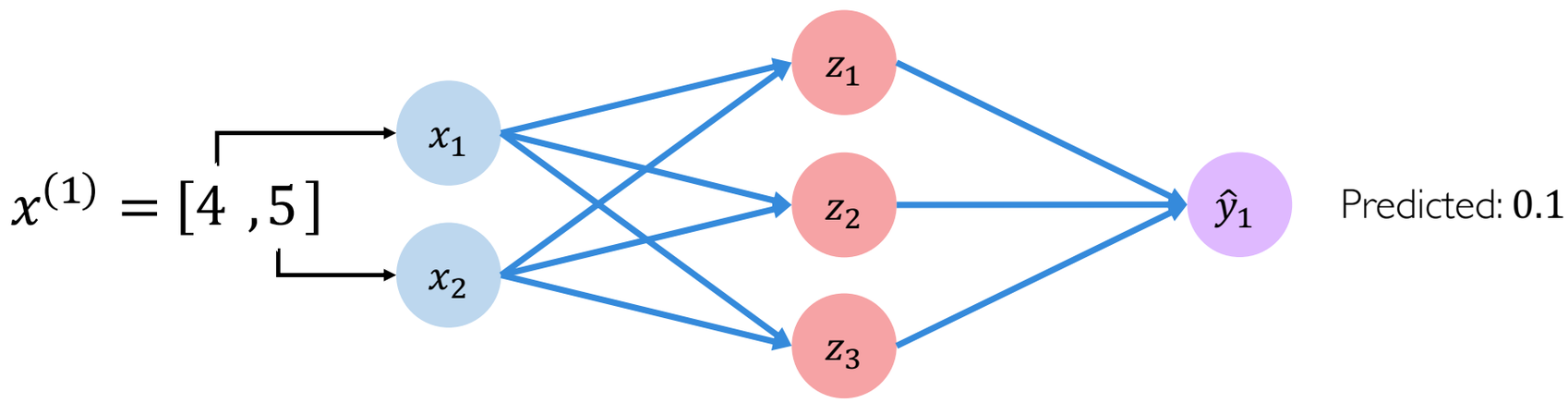
# Example Problem: Will I pass this class?



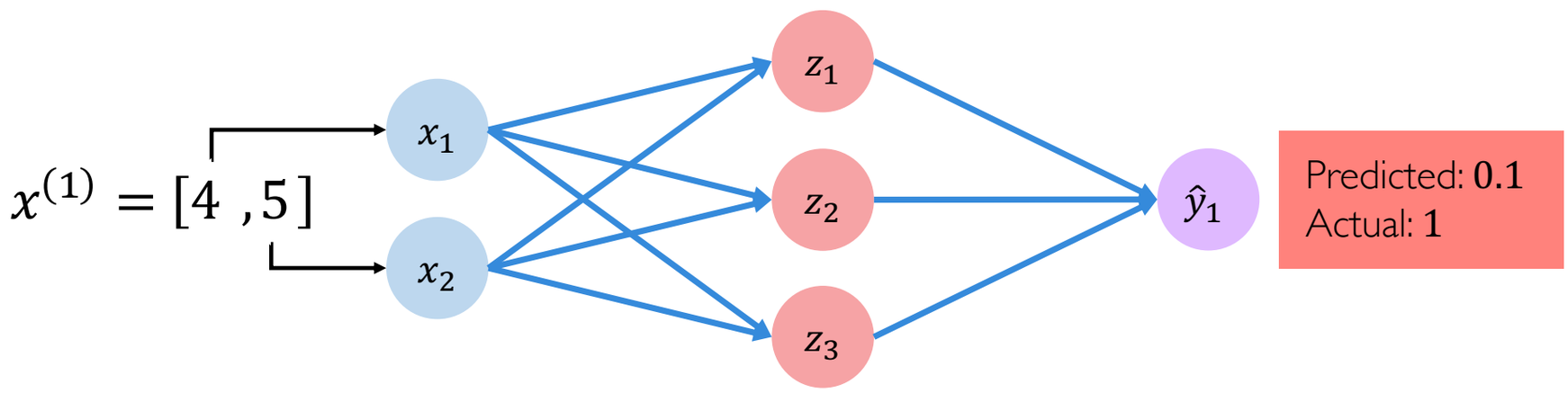
# Example Problem: Will I pass this class?



# Example Problem: Will I pass this class?

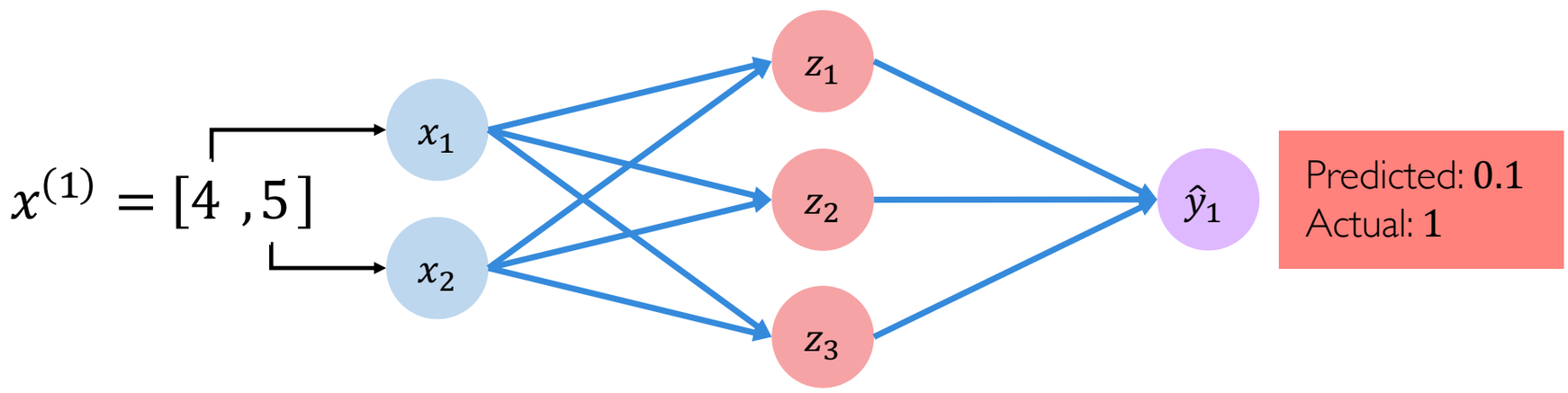


# Example Problem: Will I pass this class?



# Quantifying Loss

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

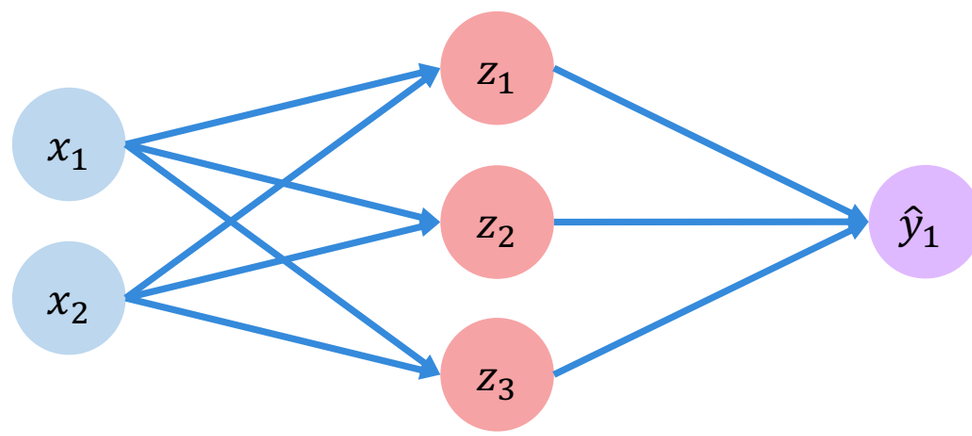



$$\mathcal{L}\left(\underbrace{f\left(x^{(i)}; \boldsymbol{W}\right)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\right)$$

# Empirical Loss

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

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




$$f(x) \qquad y$$

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \qquad \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

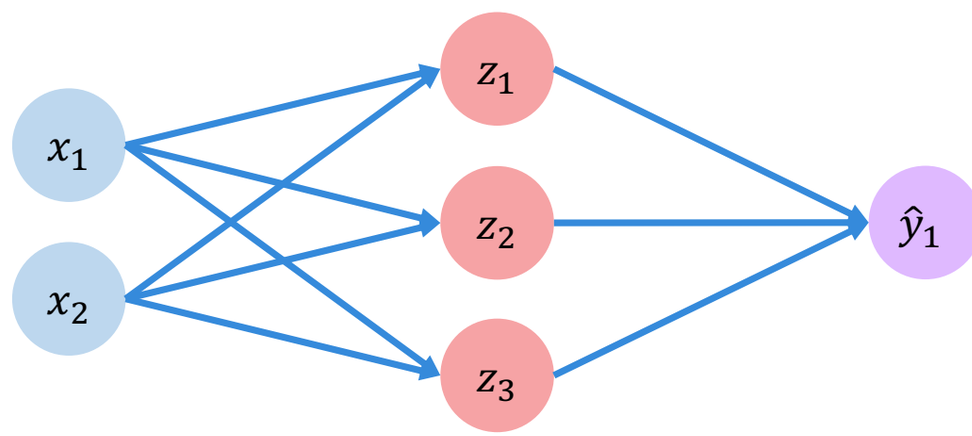
$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(\underbrace{f\left(x^{(i)}; \boldsymbol{W}\right)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\right)$$

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

# Mean Squared Error Loss

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

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




$$\begin{array}{cc} f(x) & y \\ \begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix} & \begin{bmatrix} 90 \\ 20 \\ 95 \\ \vdots \end{bmatrix} \end{array}$$

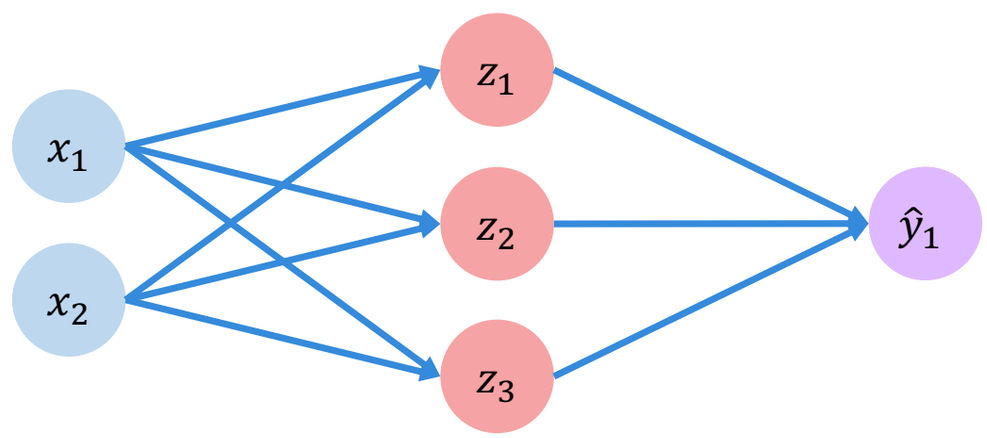
Final Grades (percentage)

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

# Binary Cross Entropy Loss

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

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

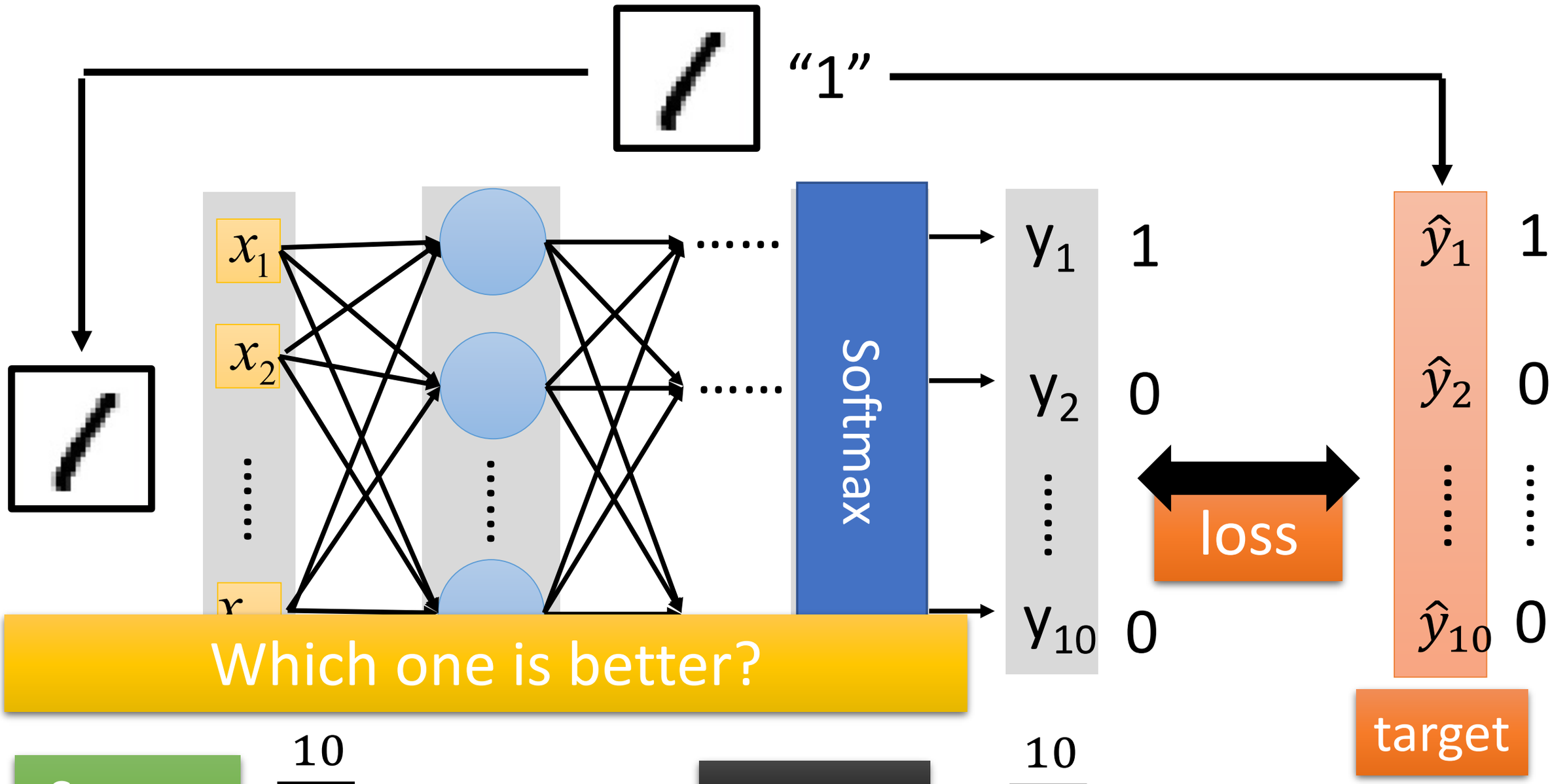



$$f(x) \qquad y$$

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \qquad \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

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

# Multiple Classes: Discrete Cross Entropy




Which one is better?

Square Error $\sum_{i=1}^{10}(y_i - \hat{y}_i)^2$ =0

Cross Entropy $-\sum_{i=1}^{10}\hat{y}_i \ln y_i$ =0

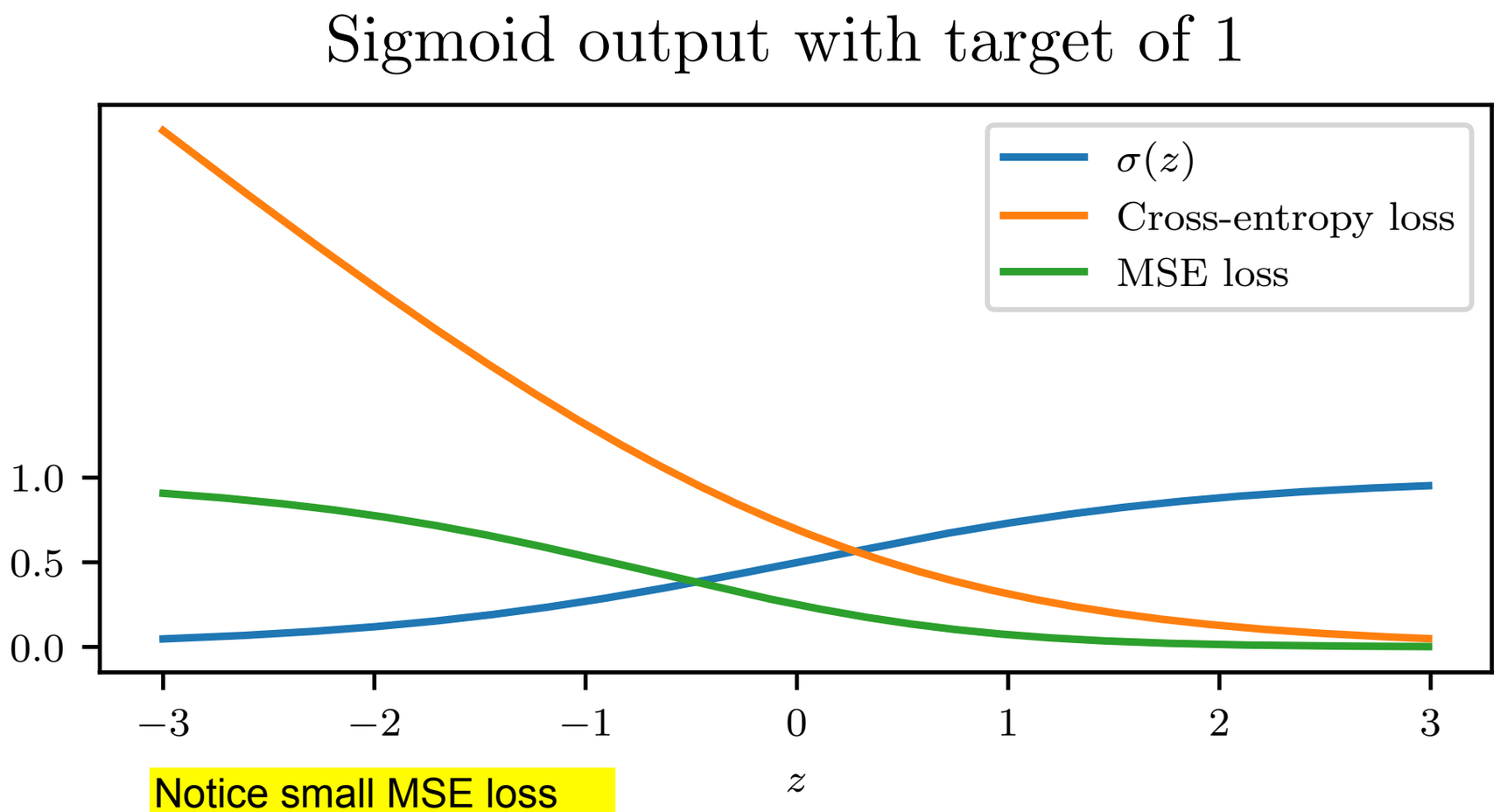
# Output Types

| Output Type | Output Distribution | Output Layer | Cost Function |
|---|---|---|---|
| Binary | Bernoulli | Sigmoid | Binary cross-entropy |
| Discrete | Multinoulli | Softmax | Discrete cross-entropy |
| Continuous | Gaussian | Linear | Gaussian cross-entropy (MSE) |
| Continuous | Mixture of Gaussian | Mixture Density | Cross-entropy |
| Continuous | Arbitrary | See part III: GAN, VAE, FVBN | Various |

**Why this combination is recommended, not Sigmoid + MSE ?**

(Goodfellow 2017)

# Don't mix and match




Notice small MSE loss changes slowly even prediction is wildly wrong

(Goodfellow 2017)

# 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***

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

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

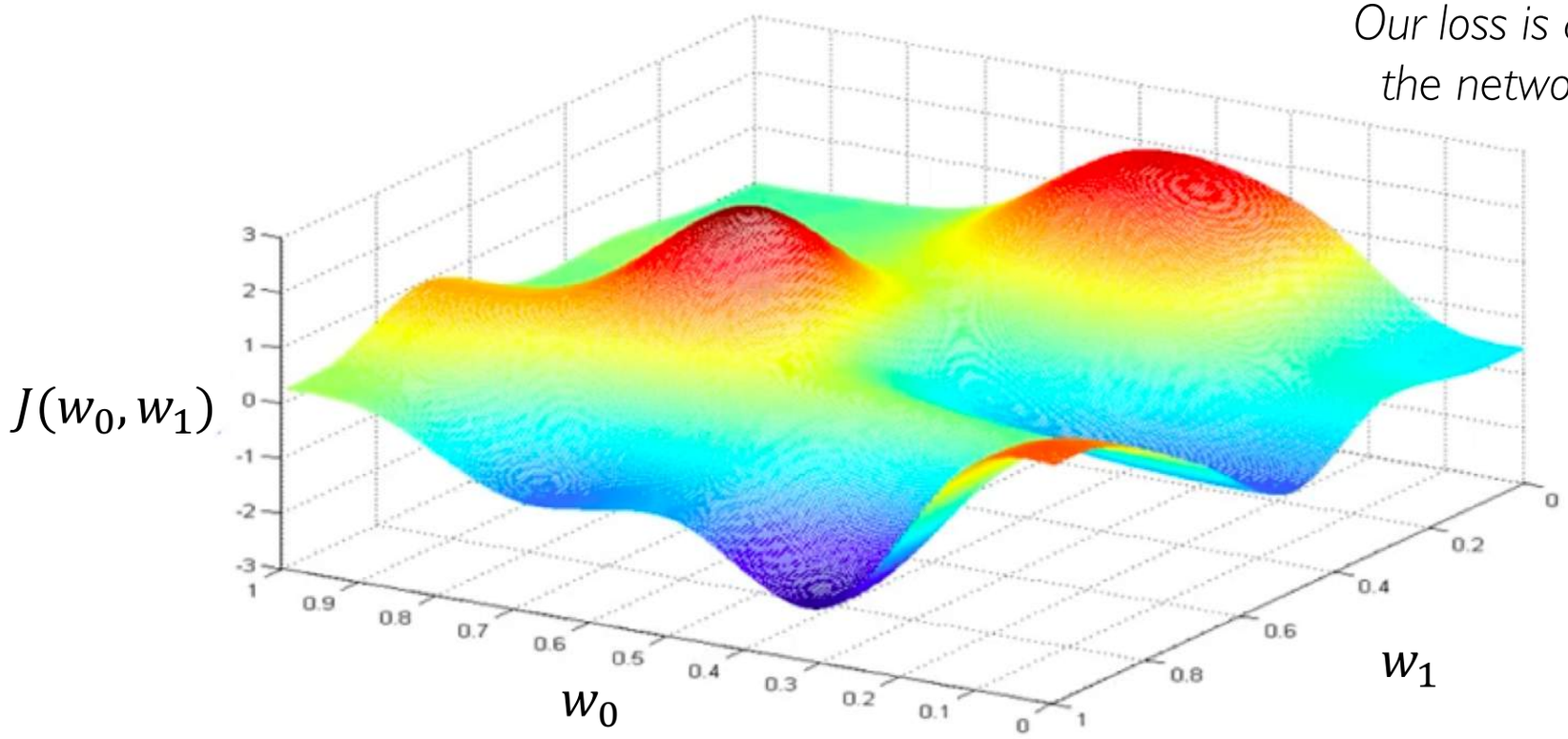
Remember:

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

# Loss Optimization

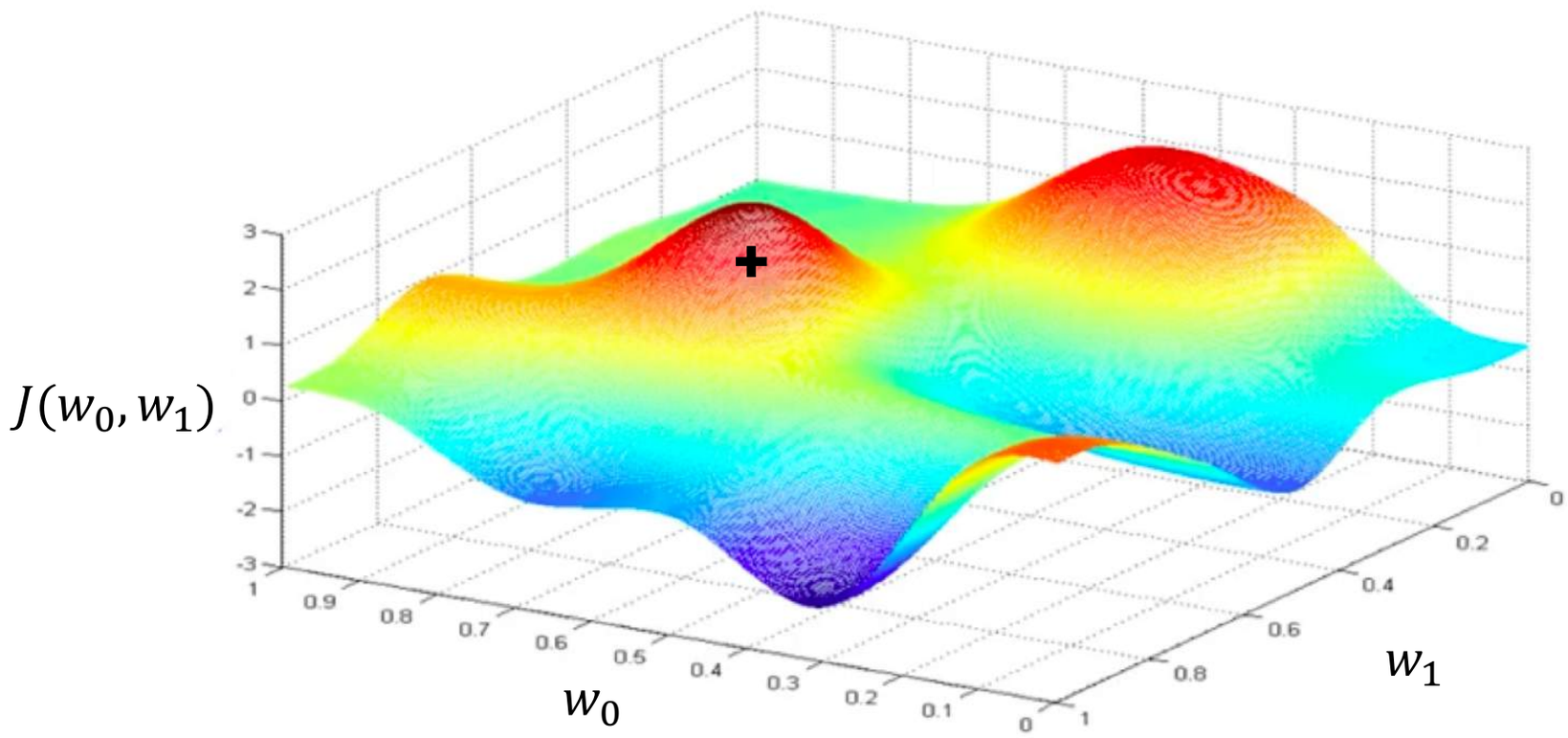
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}}\, J(\boldsymbol{W})$$

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



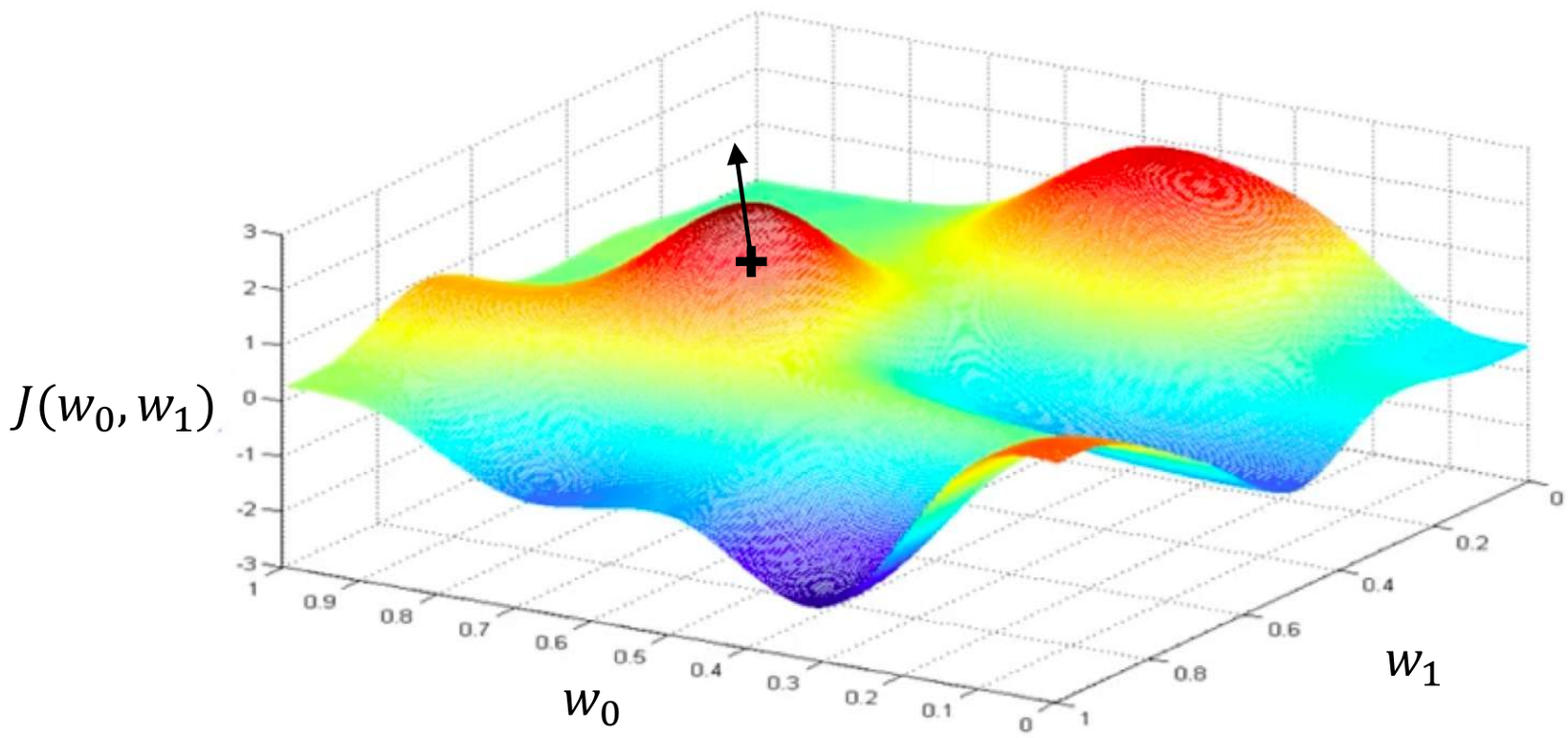
# Loss Optimization

Randomly pick an initial $(w_0, w_1)$



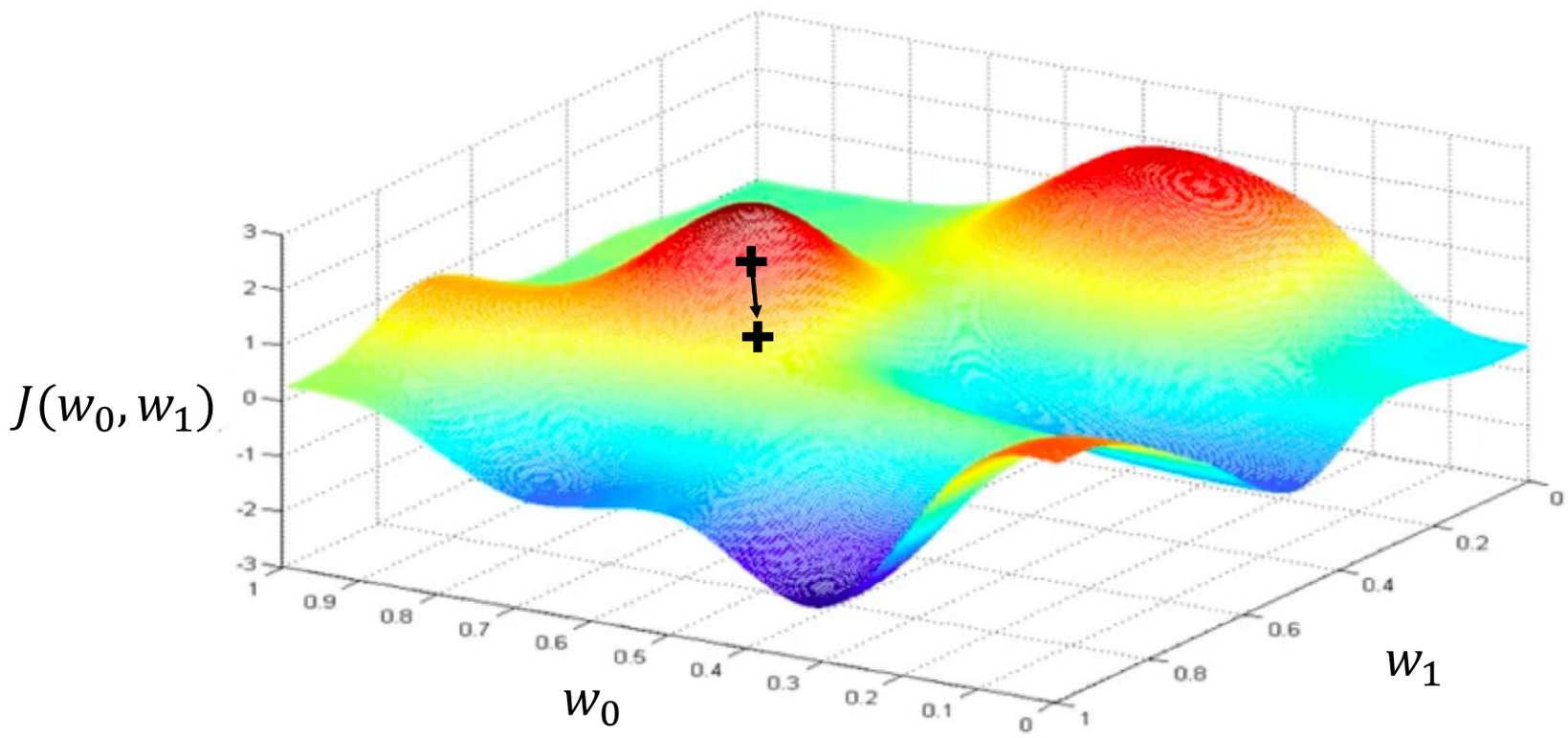
# Loss Optimization

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



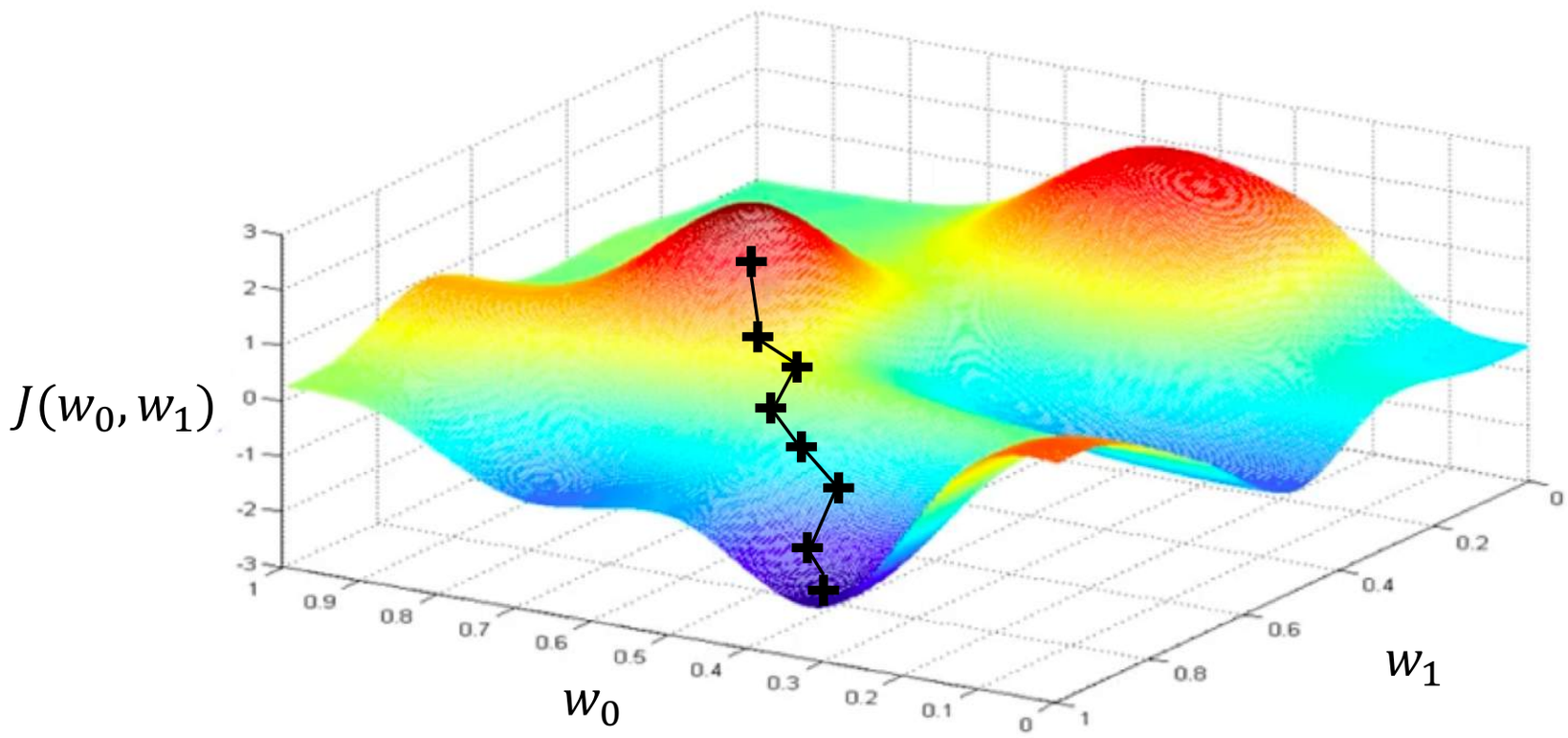
# Loss Optimization

Take small step in opposite direction of gradient



# Gradient Descent

Repeat until convergence



# Gradient Descent

**Algorithm**

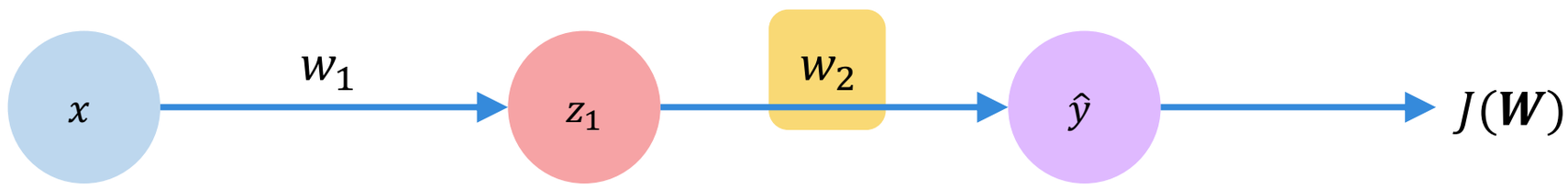
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5.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

# Gradient Descent

**Algorithm**

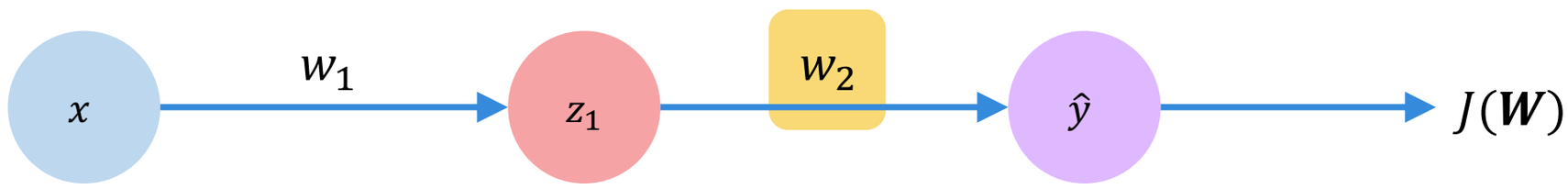
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5.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

# Computing Gradients: Backpropagation




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

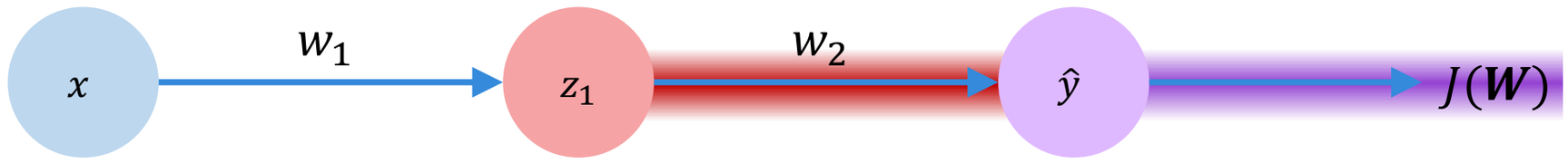
# Computing Gradients: Backpropagation




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

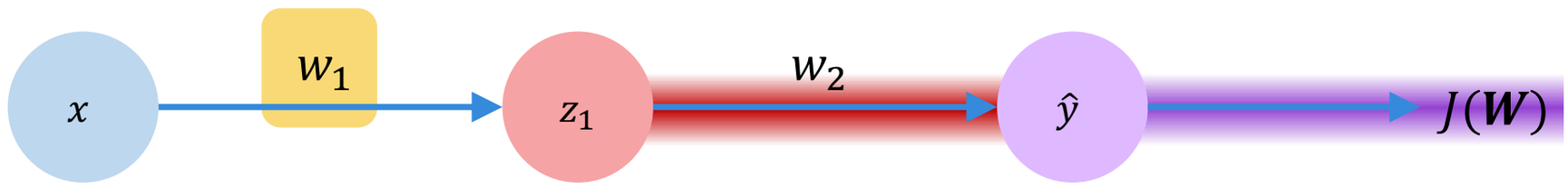
Let's use the chain rule!

# Computing Gradients: Backpropagation




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

# Computing Gradients: Backpropagation

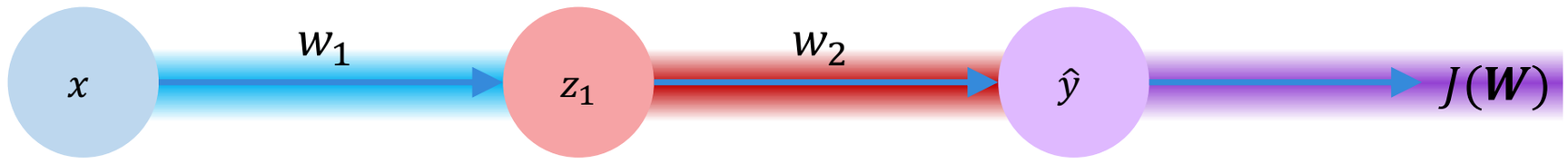



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

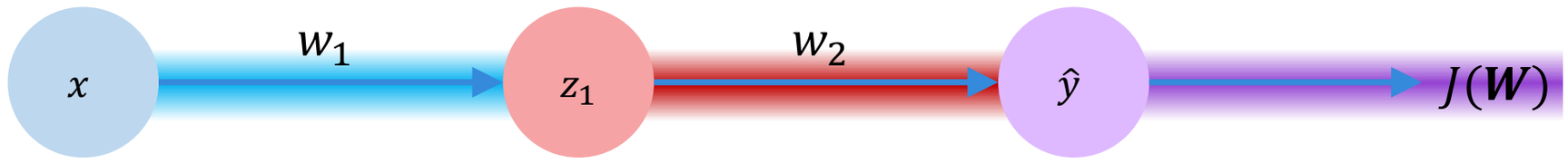
Apply chain rule!

# Computing Gradients: Backpropagation




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

# Computing Gradients: Backpropagation

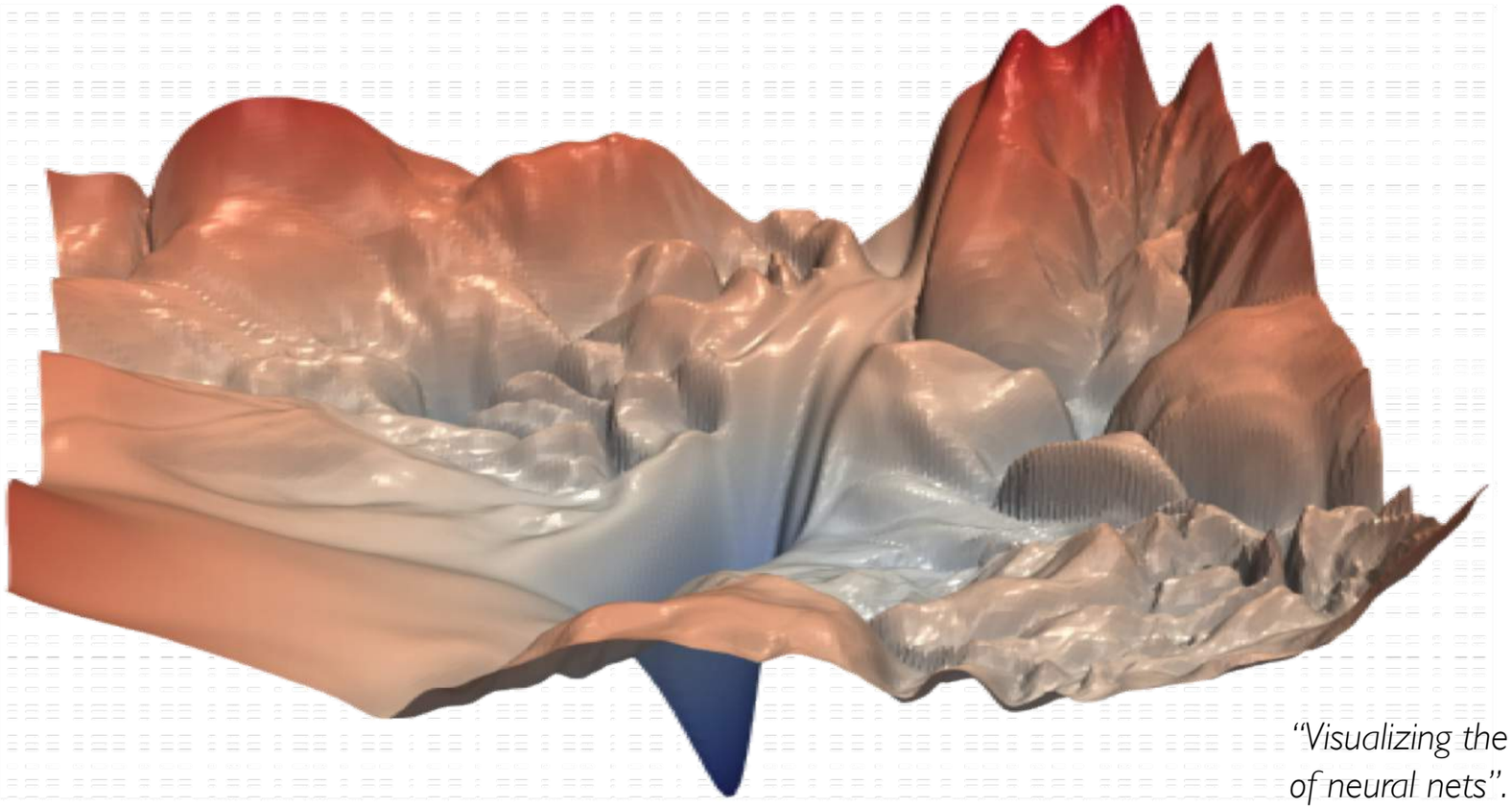



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

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

# Neural Networks in Practice: Optimization

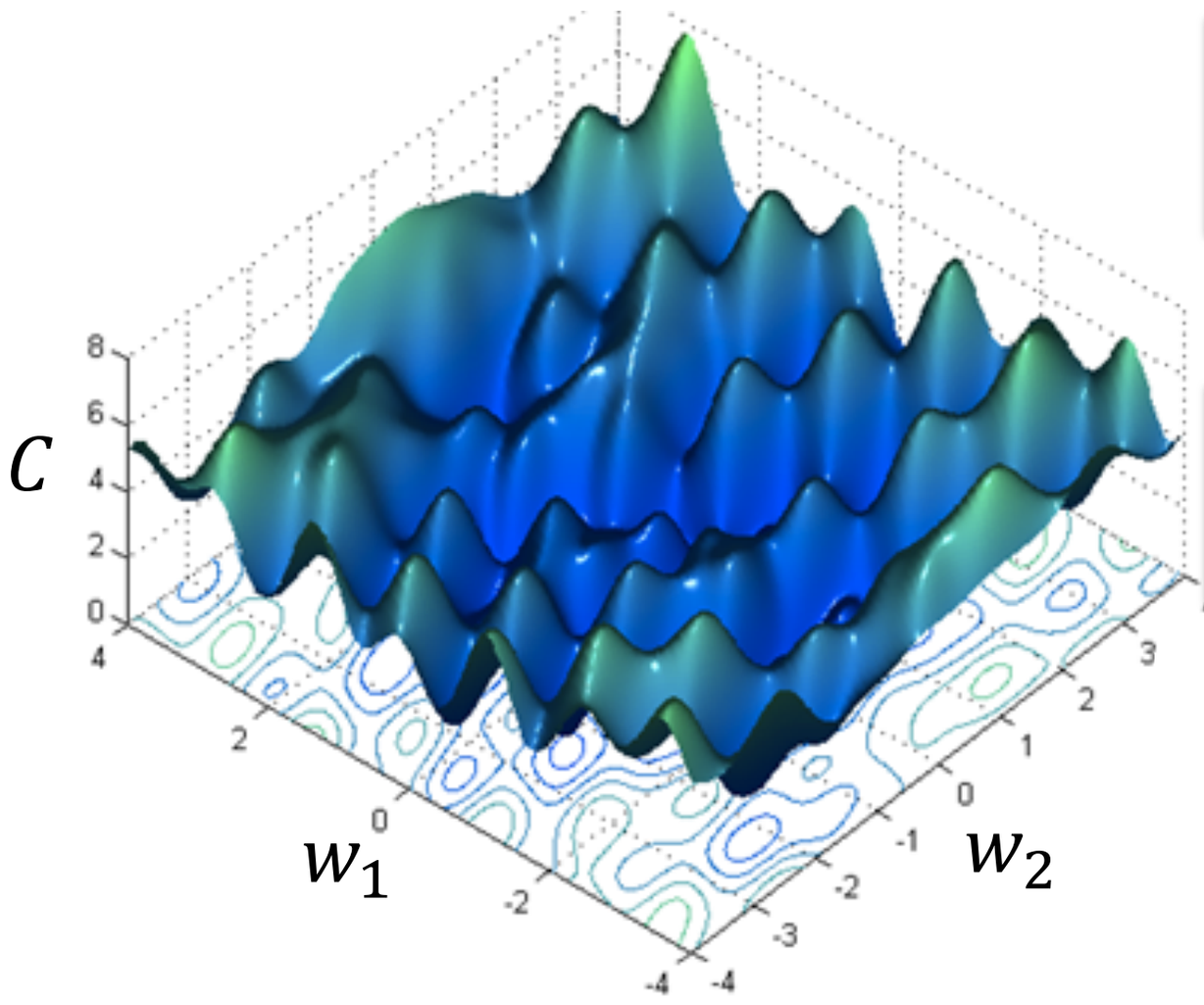
# Training Neural Networks is Difficult



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

# Local Minima

- Gradient descent never guarantee global minima




Different initial point $\theta^0$

Reach different minima, so different results

**Cost function may be non-convex!**

# Loss Functions Can Be Difficult to Optimize

**Remember:**

Optimization through gradient descent

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

# Batch Gradient Descent

**Algorithm 1** Batch Gradient Descent at Iteration $k$

**Require:** Learning rate $\epsilon_k$

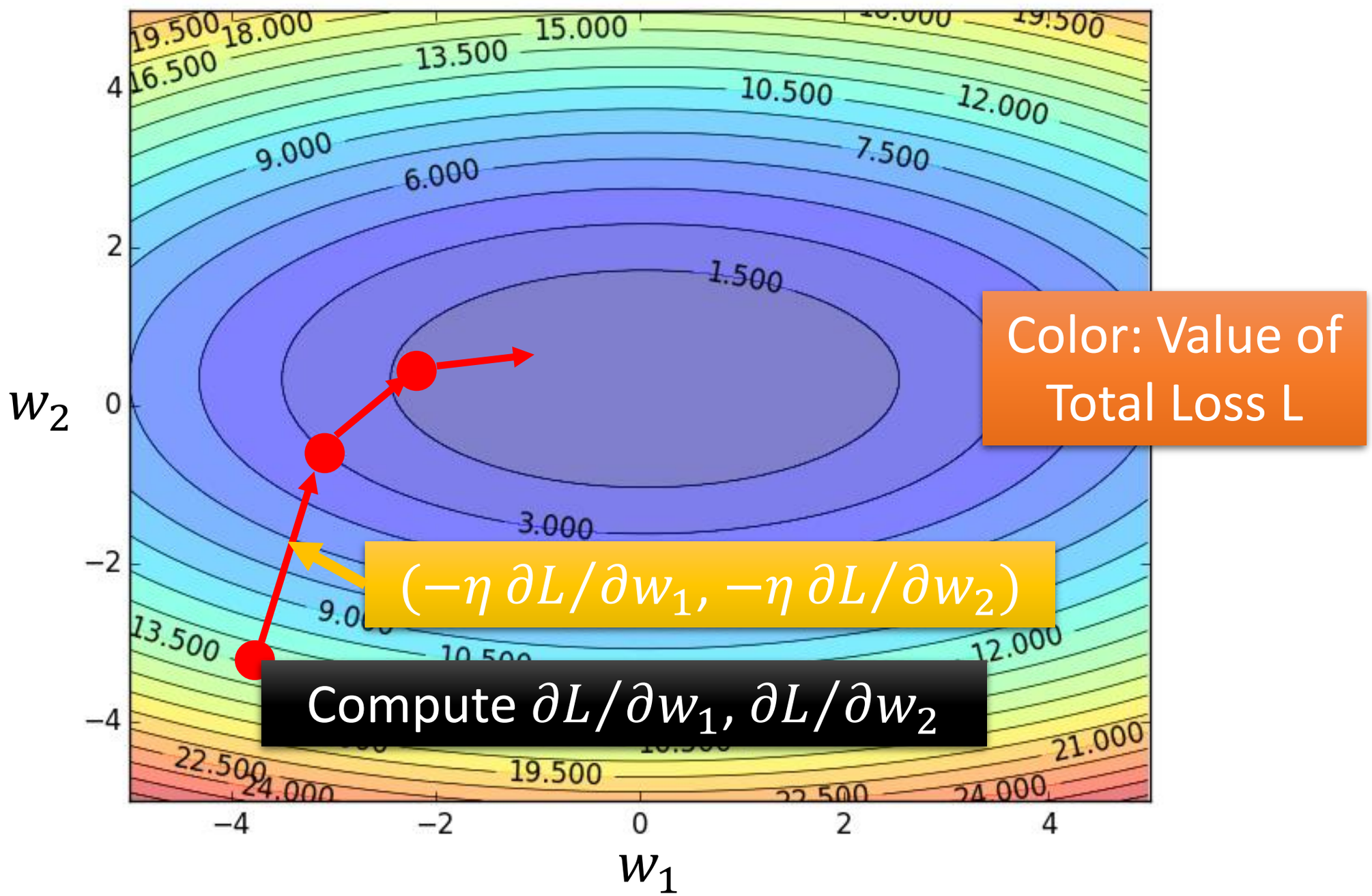
**Require:** Initial Parameter $\theta$

```
while stopping criteria not met do
    Compute gradient estimate over N examples:
    ĝ ← +(1/N) ∇_θ Σ_i L(f(x^(i); θ), y^(i))
    Apply Update: θ ← θ − ε ĝ
end while
```

- Positive: Gradient estimates are stable
- Negative: Need to compute gradients over the entire training for one update

# Gradient Descent

Hopfully, we would reach a minima .....



# Stochastic Gradient Descent

**Algorithm 2** Stochastic Gradient Descent at Iteration $k$

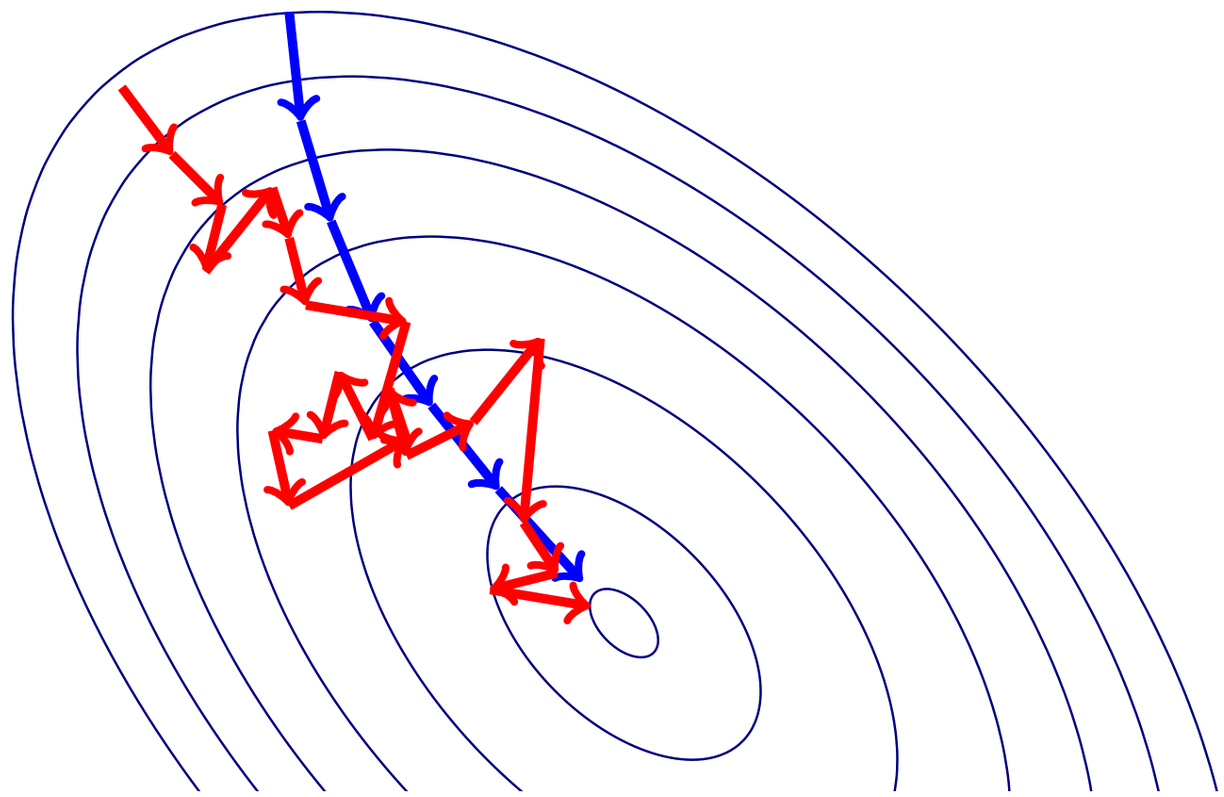
**Require:** Learning rate $\epsilon_k$
**Require:** Initial Parameter $\theta$

```
while stopping criteria not met do
    Sample example (x^(i), y^(i)) from training set
    Compute gradient estimate:
    ĝ ← +∇_θ L(f(x^(i); θ), y^(i))
    Apply Update: θ ← θ − ε ĝ
end while
```

- $\epsilon_k$ is learning rate at step $k$
- Sufficient condition to guarantee convergence:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \text{ and } \sum_{k=1}^{\infty} \epsilon_k^2 < \infty$$

# Stochastic Gradient Descent



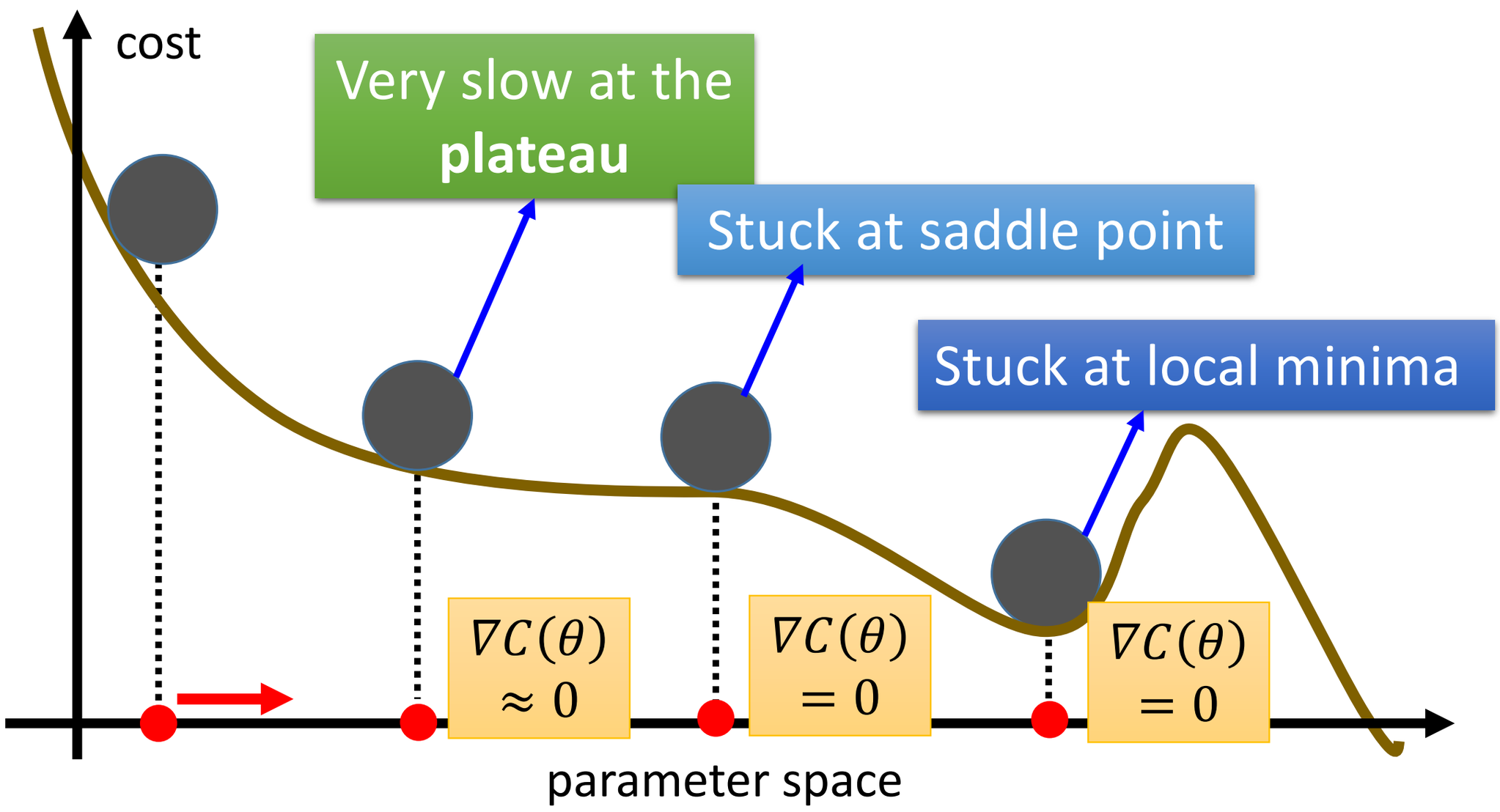
# Minibatching

Intuition: You don't have to use all training samples to find a right direction. A mini- batch may give good estimate

- Potential Problem: Gradient estimates can be very noisy
- Obvious Solution: Use larger mini-batches
- Advantage: Computation time per update does not depend on number of training examples $N$
- This allows convergence on extremely large datasets
- See: Large Scale Learning with Stochastic Gradient Descent by Leon Bottou

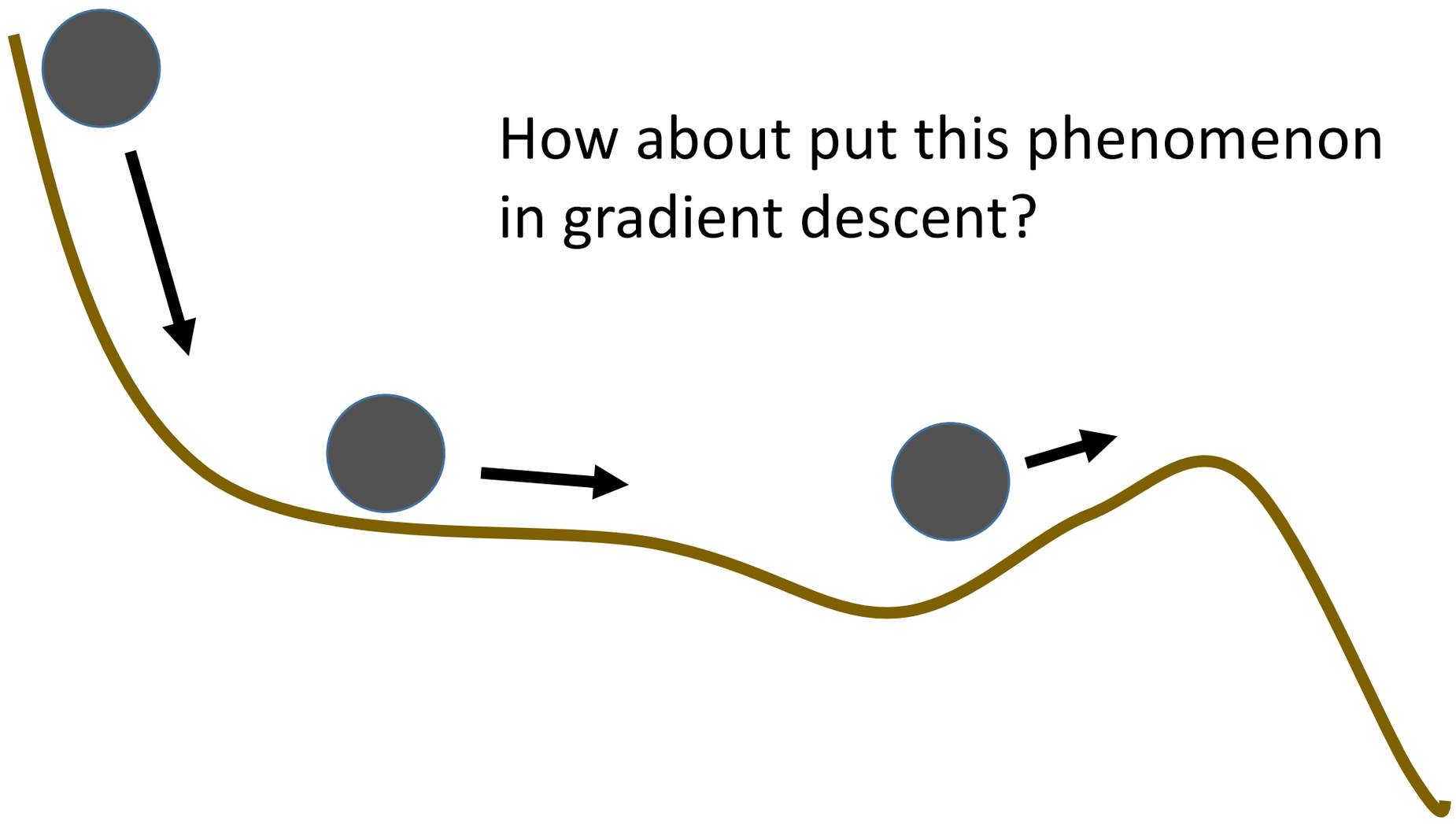
Hyper parameters: batch size, epoch

# Besides local minima ……



# In physical world ……

- Momentum

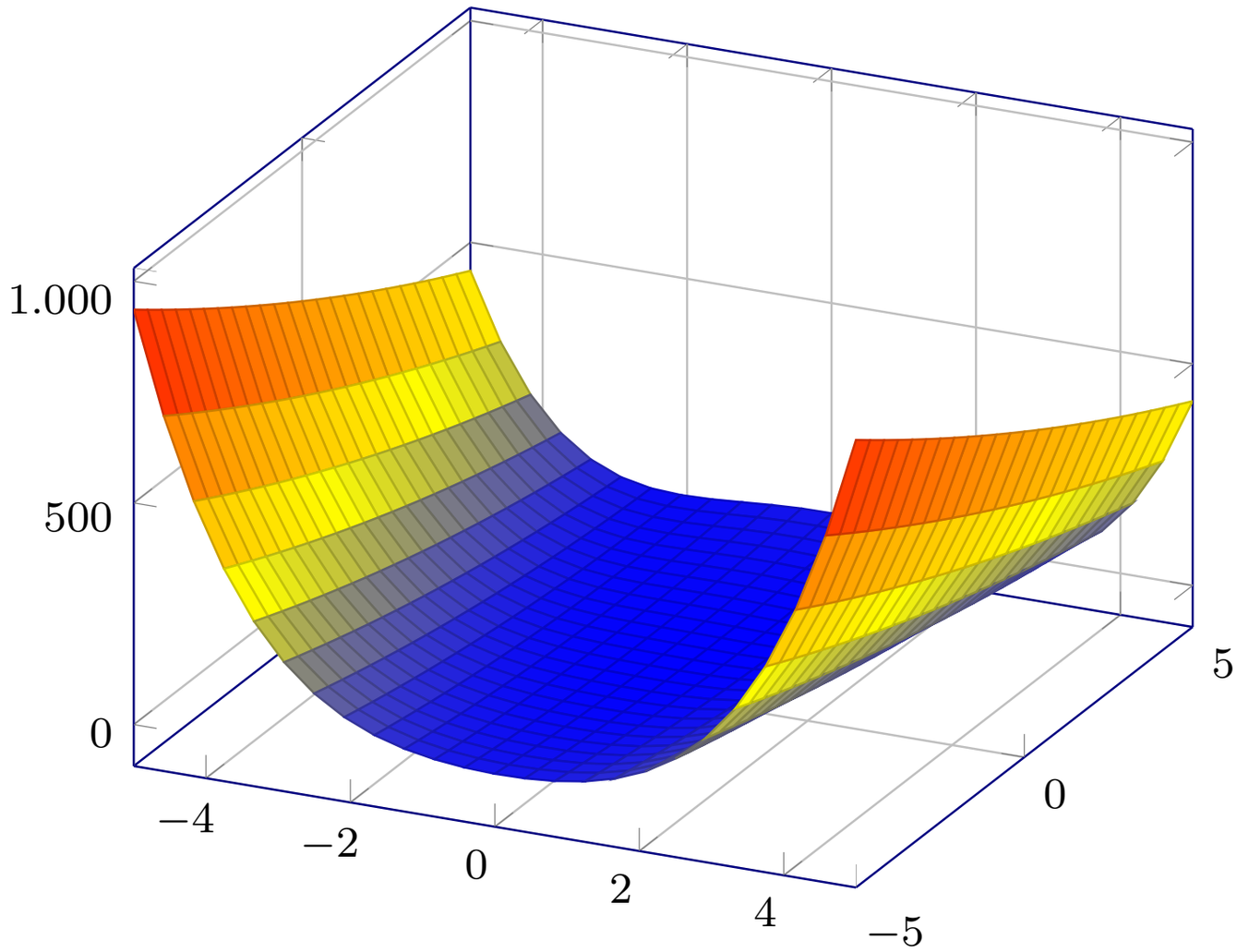


How about put this phenomenon in gradient descent?

# Momentum

- The Momentum method is a method to accelerate learning using SGD
- In particular SGD suffers in the following scenarios:
  - Error surface has high curvature
  - We get small but consistent gradients
  - The gradients are very noisy

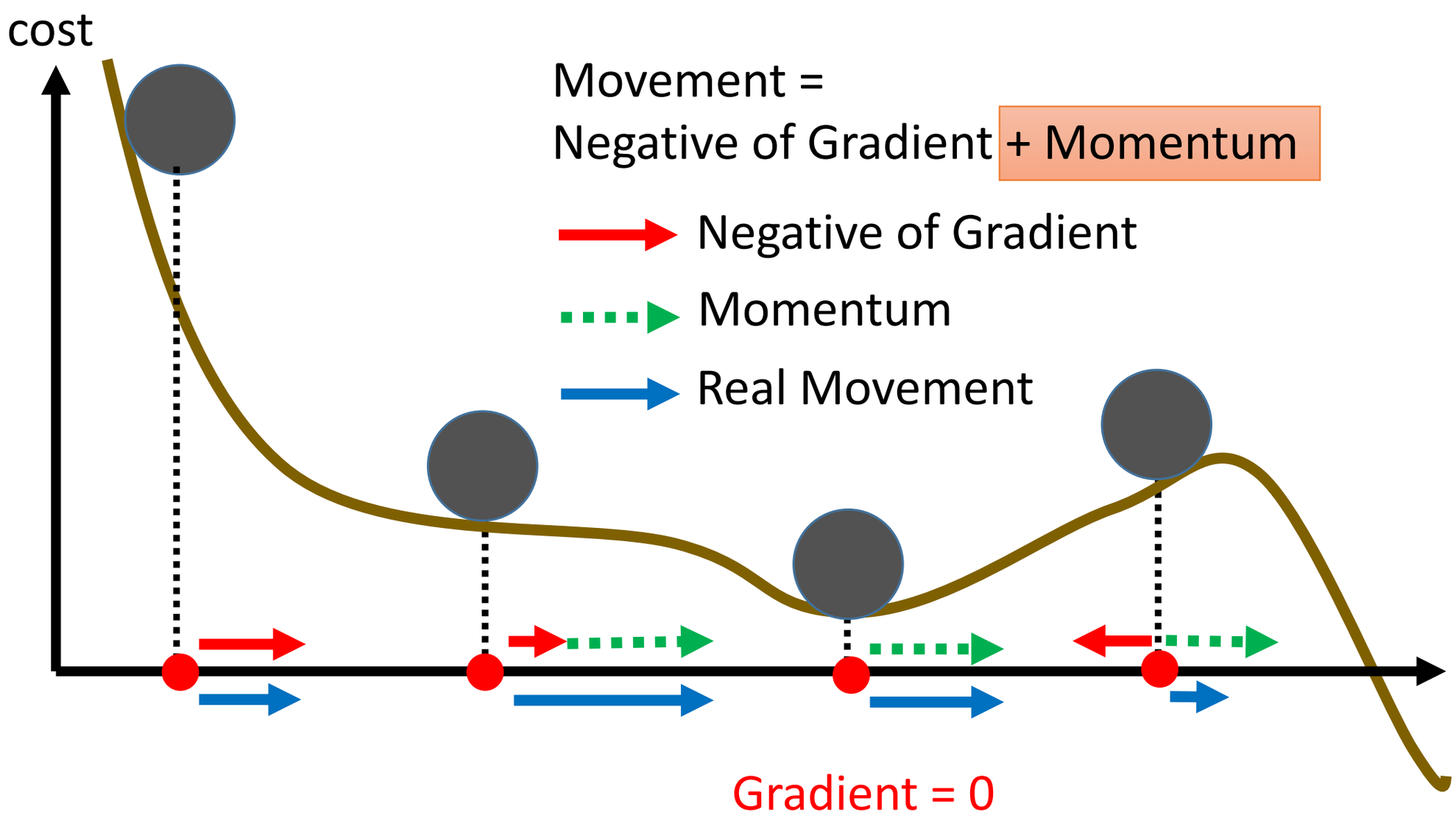
# Momentum




- Gradient Descent would move quickly down the walls, but very slowly through the valley floor

# Momentum

Still not guarantee reaching global minima, but give some hope ……



# Momentum

- How do we try and solve this problem?
- Introduce a new variable $\mathbf{v}$, the velocity
- We think of $\mathbf{v}$ as the direction and speed by which the parameters move as the learning dynamics progresses
- The velocity is an exponentially decaying moving average of the negative gradients

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left( L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}) \right)$$

- $\alpha \in [0, 1)$Update rule: $\theta \leftarrow \theta + \mathbf{v}$

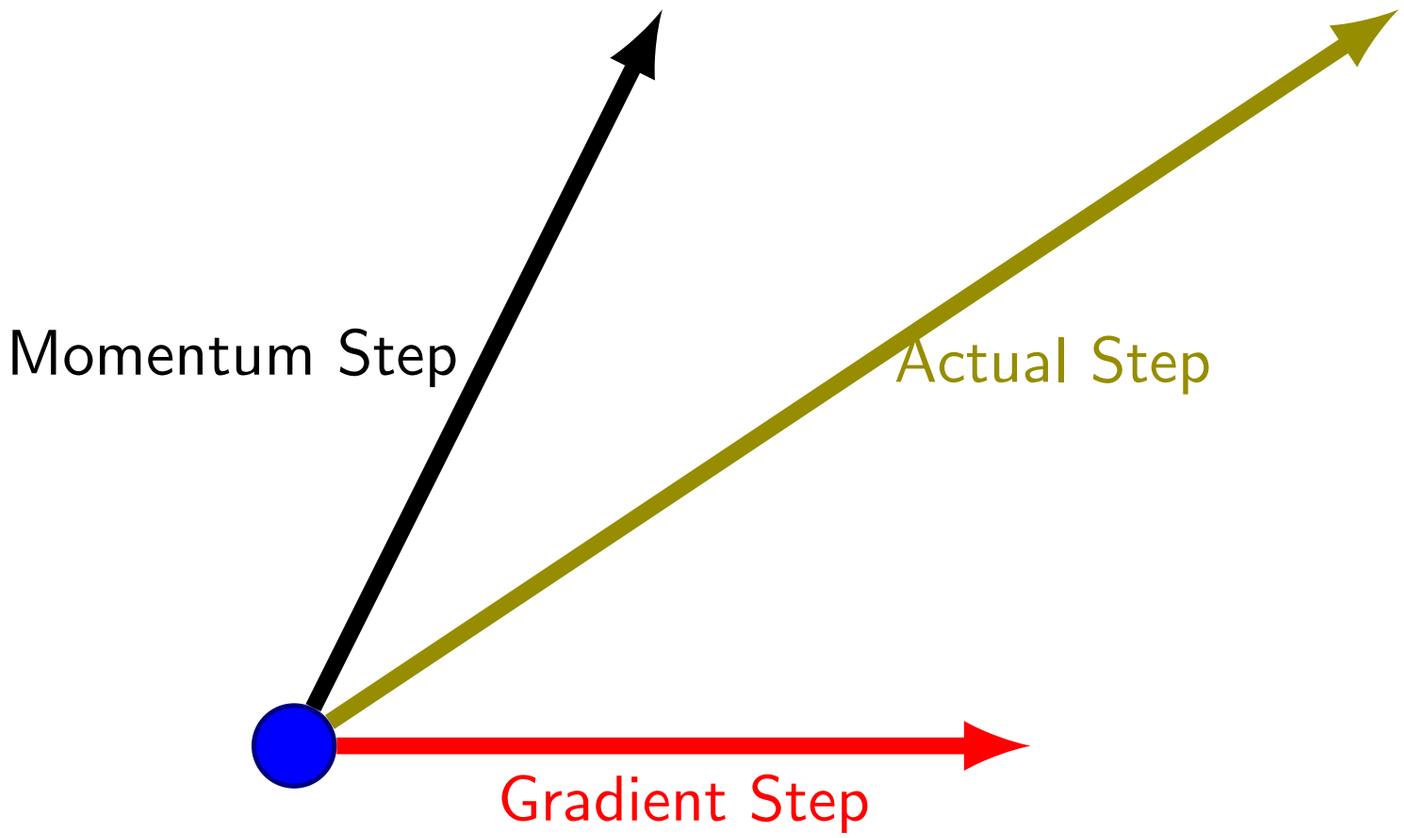
# Momentum

- Let's look at the velocity term:

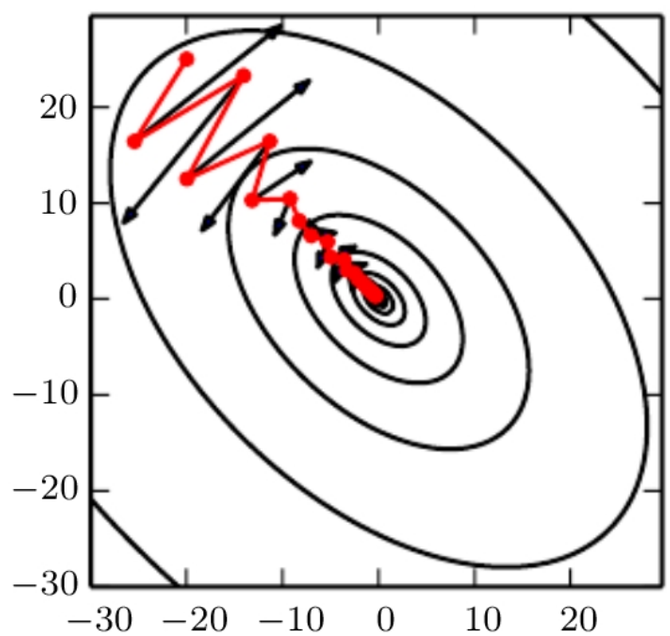
$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left( L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}) \right)$$

- The velocity accumulates the previous gradients
- What is the role of $\alpha$?
  - If $\alpha$ is larger than $\epsilon$ the current update is more affected by the previous gradients
  - Usually values for $\alpha$ are set high $\approx 0.8, 0.9$

# Momentum



# Momentum




Loss Contour Plot

Illustration of how momentum traverses such an error surface better compared to Gradient Descent

# SGD with Momentum

**Algorithm 2** Stochastic Gradient Descent with Momentum

**Require:** Learning rate $\epsilon_k$
**Require:** Momentum Parameter $\alpha$
**Require:** Initial Parameter $\theta$
**Require:** Initial Velocity $\mathbf{v}$

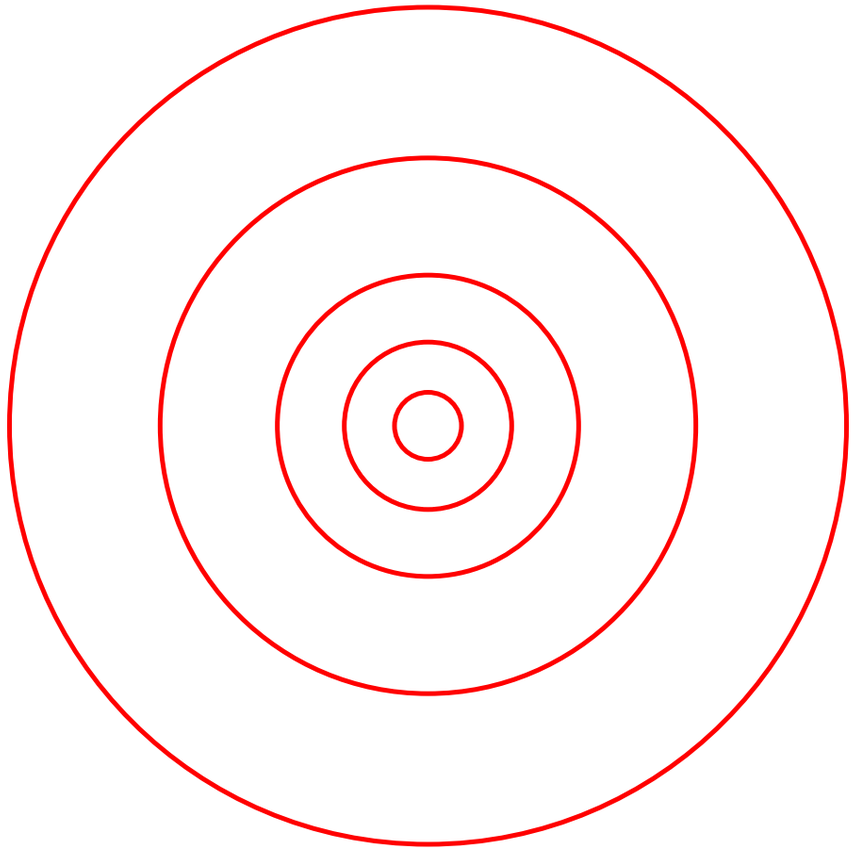
```
while stopping criteria not met do
    Sample example (x^(i), y^(i)) from training set
    Compute gradient estimate:
    ĝ ← +∇_θ L(f(x^(i); θ), y^(i))
    Compute the velocity update:
    v ← αv − εĝ
    Apply Update: θ ← θ + v
end while
```

# Adaptive Learning Rate Methods

# Motivation

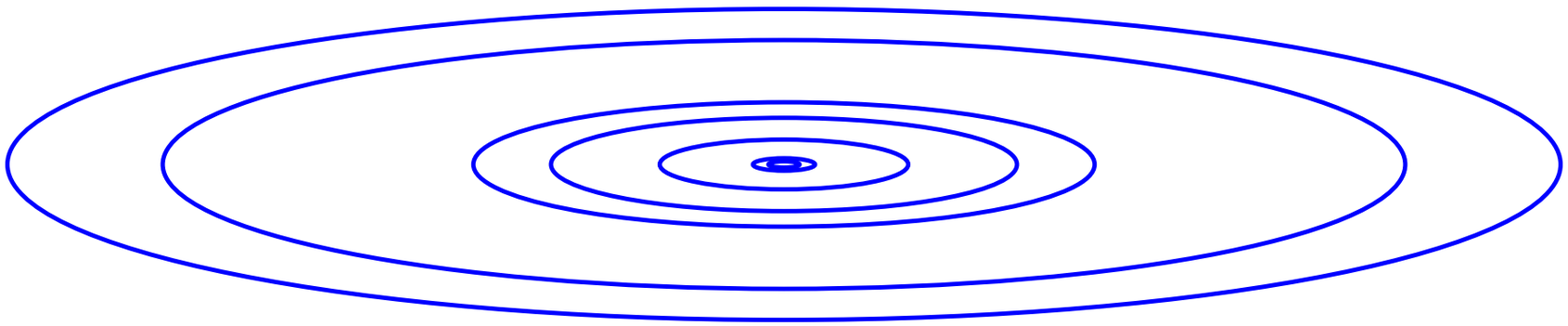
- Till now we assign the same learning rate to all features
- If the features vary in importance and frequency, why is this a good idea?
- It's probably not!

# Motivation



Nice (all features are equally important)

# Motivation



Harder!

# AdaGrad

- Idea: Downscale a model parameter by square-root of sum of squares of all its historical values
- Parameters that have large partial derivative of the loss – learning rates for them are rapidly declined

# AdaGrad

**Algorithm 4** AdaGrad

**Require:** Global Learning rate $\epsilon$, Initial Parameter $\theta$, $\delta$

Initialize $\mathbf{r} = 0$

**while** stopping criteria not met **do**

  Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

  Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

  Accumulate: $\mathbf{r} \leftarrow \mathbf{r} + \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$

  Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$

  Apply Update: $\theta \leftarrow \theta + \Delta\theta$

**end while**

Note:
$\odot$: Hadamard product, i.e. element-wise product

# Adagrad

$$\eta_w = \boxed{\frac{\eta}{\sqrt{\sum_{i=0}^{t}(g^i)^2}}}$$

| | $g^0$ | $g^1$ |
|---|---|---|
| $w_1$ | 0.1 | 0.2 |

| | $g^0$ | $g^1$ |
|---|---|---|
| $w_2$ | 20.0 | 10 |

Learning rate:

$$\frac{\eta}{\sqrt{0.1^2}} = \frac{\eta}{0.1}$$

$$\frac{\eta}{\sqrt{0.1^2 + 0.2^2}} = \frac{\eta}{0.22}$$

Learning rate:

$$\frac{\eta}{\sqrt{20^2}} = \frac{\eta}{20}$$

$$\frac{\eta}{\sqrt{20^2 + 10^2}} = \frac{\eta}{22}$$

***Observation:*** 1. Learning rate is smaller and smaller for all parameters

2. Smaller derivatives, larger learning rate, and vice versa

Why?

# RMSProp

- AdaGrad is good when the objective is convex.
- AdaGrad might shrink the learning rate too aggressively, we want to keep the history in mind
- We can adapt it to perform better in non-convex settings by accumulating an exponentially decaying average of the gradient
- This is an idea that we use again and again in Neural Networks

# RMSProp

**Algorithm 5** RMSProp

**Require:** Global Learning rate $\epsilon$, decay parameter $\rho$, $\delta$

Initialize $\mathbf{r} = 0$

**while** stopping criteria not met **do**

  Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

  Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

  Accumulate: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1-\rho)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$

  Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$

  Apply Update: $\theta \leftarrow \theta + \Delta\theta$

**end while**

# Adam

- We could have used RMSProp with momentum
- Use of Momentum with rescaling is not well motivated
- Adam is like RMSProp with Momentum but with bias correction terms for the first and second moments

# Adam: ADAptive Moments

**Algorithm 7** RMSProp with Nesterov

**Require:** $\epsilon$ (set to 0.0001), decay rates $\rho_1$ (set to 0.9), $\rho_2$ (set to 0.9), $\theta$, $\delta$

Initialize moments variables $\mathbf{s} = 0$ and $\mathbf{r} = 0$, time step t = 0

**while** stopping criteria not met **do**
  Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
  Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
  $t \leftarrow t + 1$
  Update: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\hat{\mathbf{g}}$
  Update: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
  Correct Biases: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\rho_1^t}, \hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-\rho_2^t}$
  Compute Update: $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}}+\delta}$
  Apply Update: $\theta \leftarrow \theta + \Delta\theta$
**end while**

# All your GRADs are belong to us!

SGD: $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

Momentum: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}}$ then $\theta \leftarrow \theta + \mathbf{v}$

Nesterov: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left( L(f(\mathbf{x}^{(i)}; \theta + \alpha \mathbf{v}), \mathbf{y}^{(i)}) \right)$ then $\theta \leftarrow \theta + \mathbf{v}$

AdaGrad: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$ then $\Delta\theta - \leftarrow \frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$ then $\theta \leftarrow \theta + \Delta\theta$

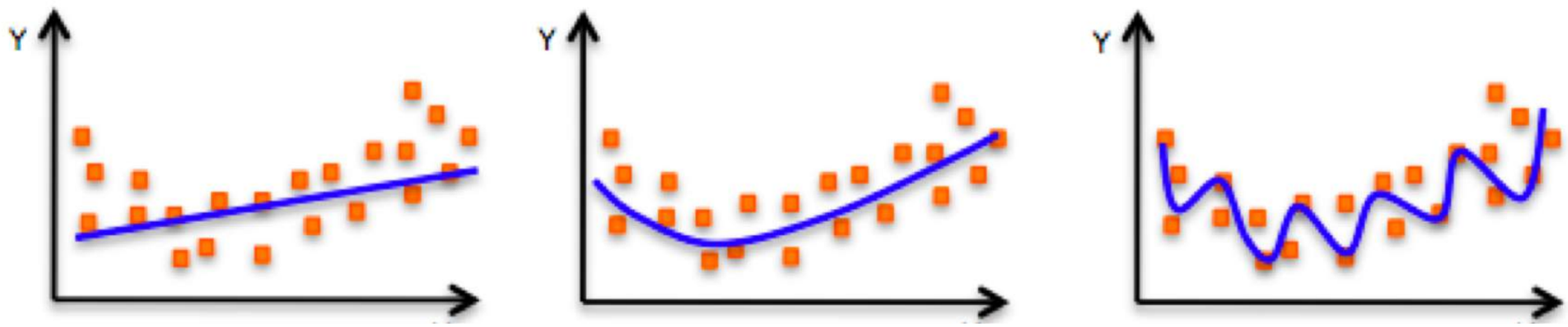
RMSProp: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$ then $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$ then $\theta \leftarrow \theta + \Delta\theta$

Adam: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}, \hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$ then $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$ then $\theta \leftarrow \theta + \Delta\theta$

**Animations of learning process dynamics:**
**http://cs231n.github.io/neural-networks-3/#anneal**

# 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?*

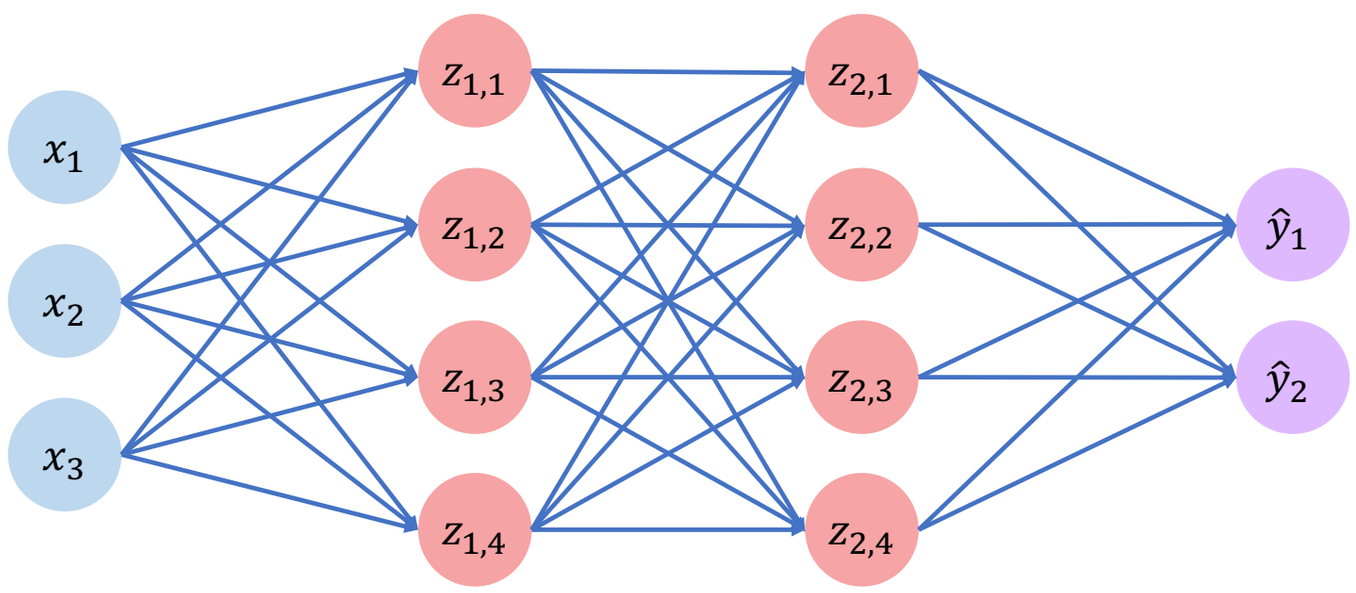
*Technique that constrains our optimization problem to discourage complex models*

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

*Improve generalization of our model on unseen data*

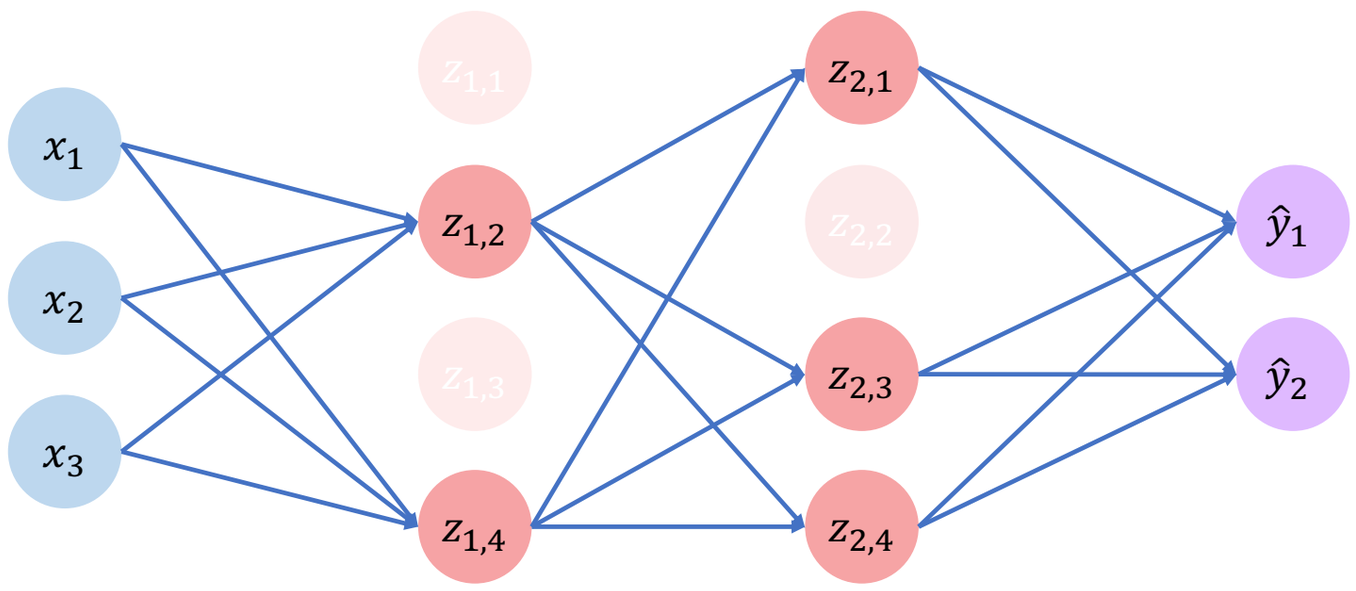
# Regularization 1: Dropout

- During training, randomly set some activations to 0



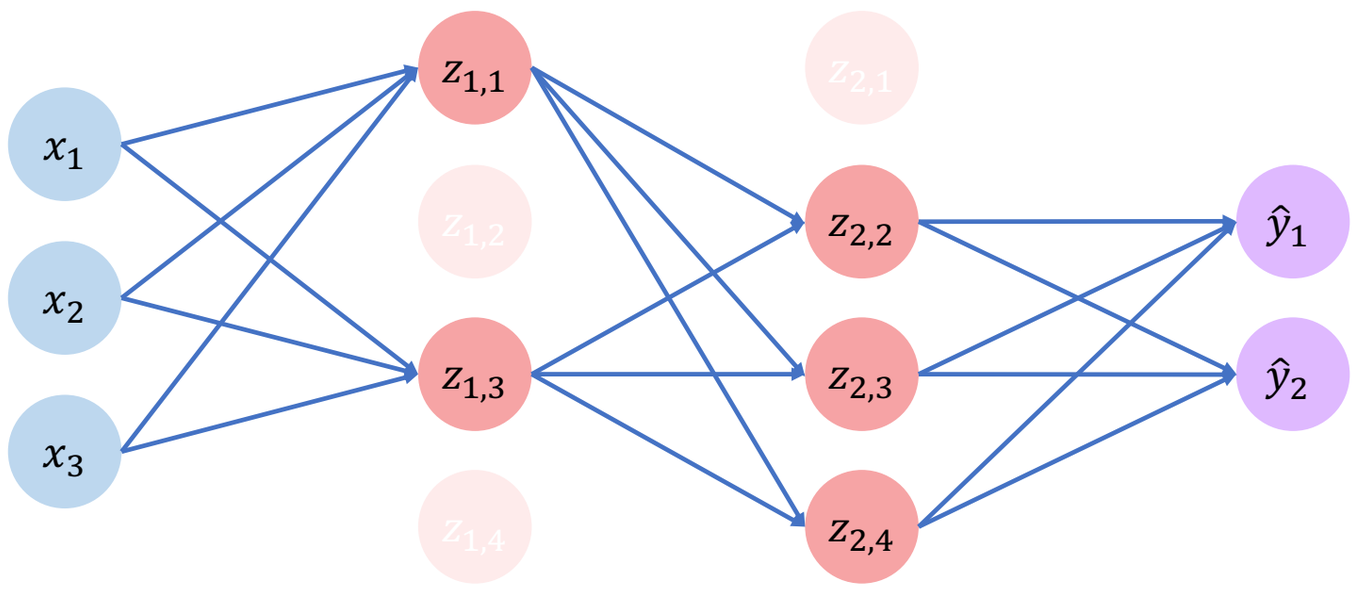
# Regularization 1: Dropout

- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node



# 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

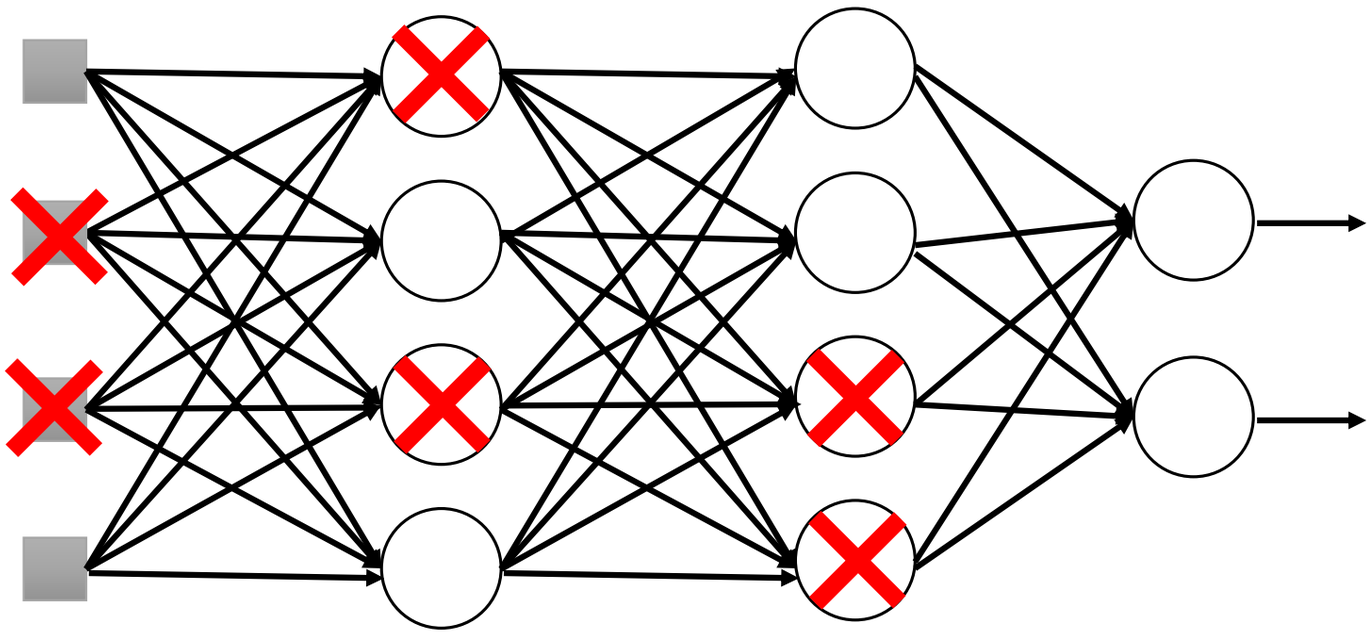


# Dropout

Pick a mini-batch

$$\theta^t \leftarrow \theta^{t-1} - \eta \nabla C(\theta^{t-1})$$

**Training:**



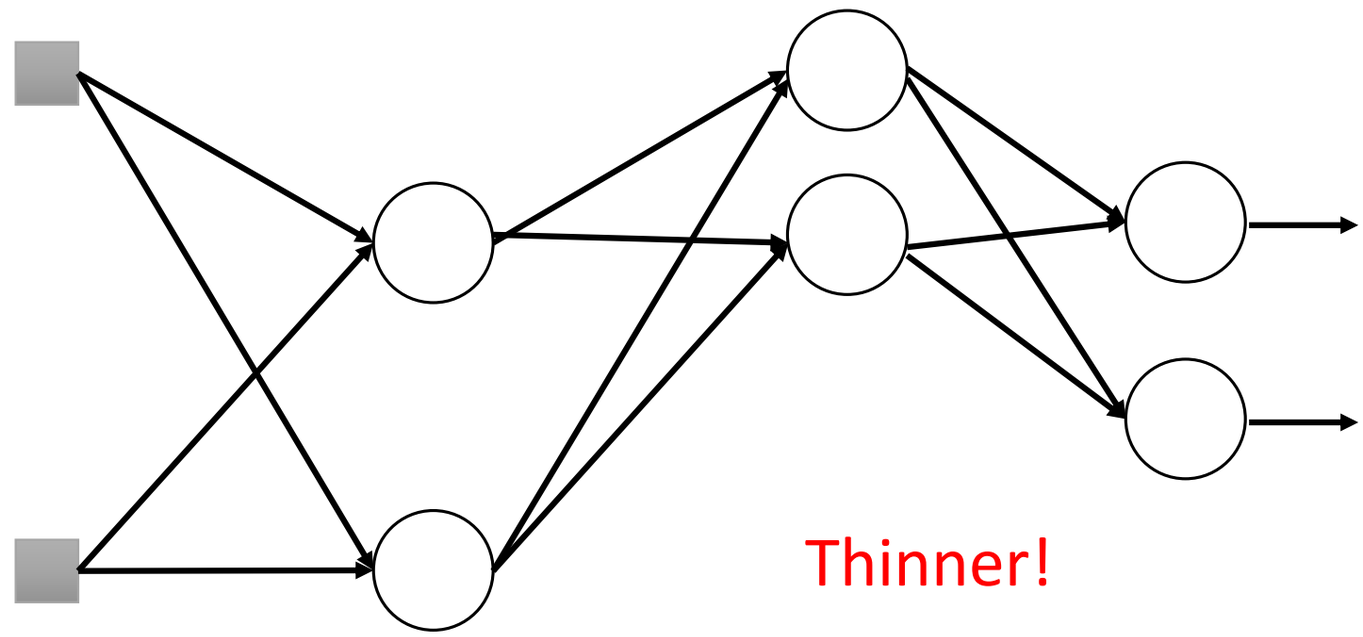
- **Each time before computing the gradients**
  - Each neuron has p% to dropout

# Dropout

Pick a mini-batch

$$\theta^t \leftarrow \theta^{t-1} - \eta \nabla C(\theta^{t-1})$$

**Training:**

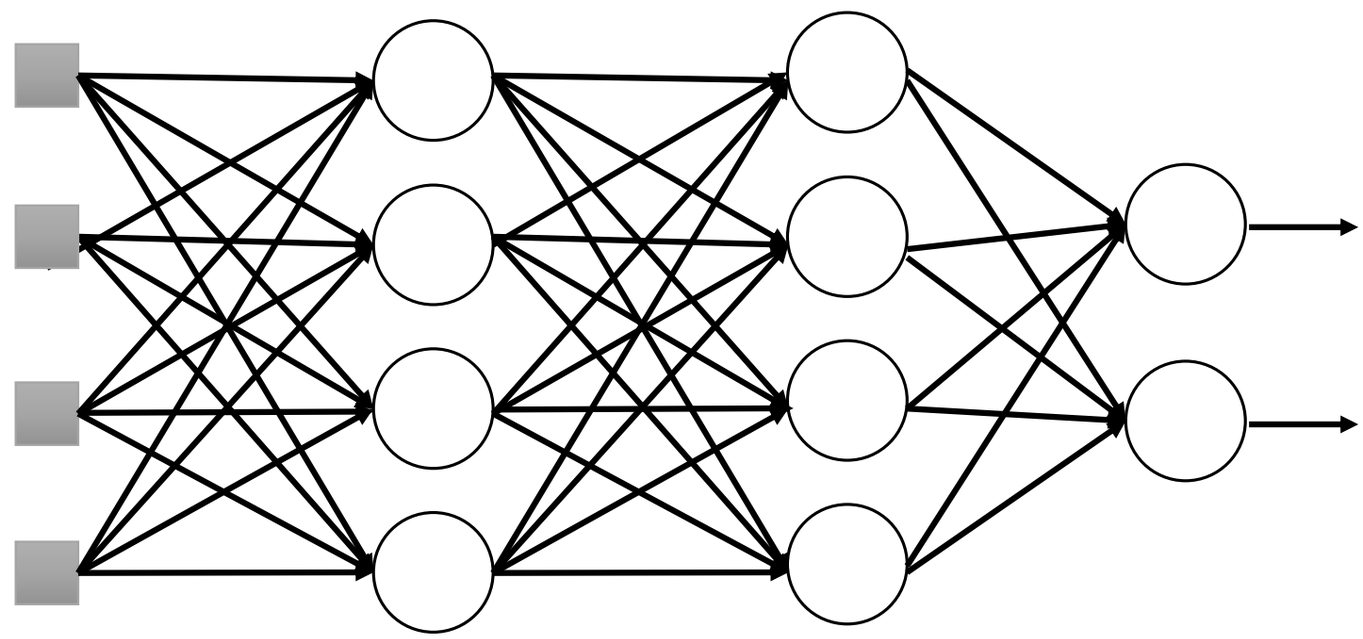



- **Each time before computing the gradients**
  - Each neuron has p% to dropout
    - ➡ **The structure of the network is changed.**
  - Using the new network for training

For each mini-batch, we resample the dropout neurons

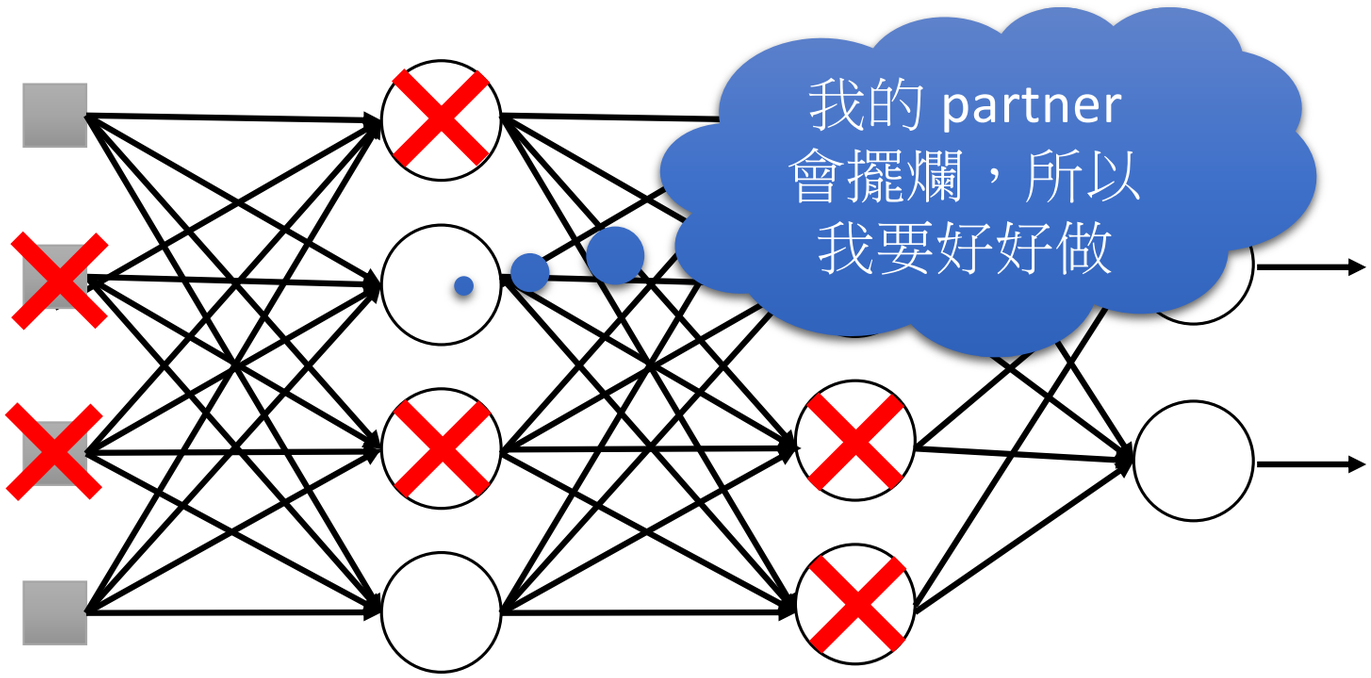
# Dropout

**Testing:**



- **No dropout**
  - If the dropout rate at training is p%,
    all the weights times (1-p)%
  - Assume that the dropout rate is 50%.
    If a weight $w = 1$ by training, set $w = 0.5$ for testing.

# Dropout - Intuitive Reason



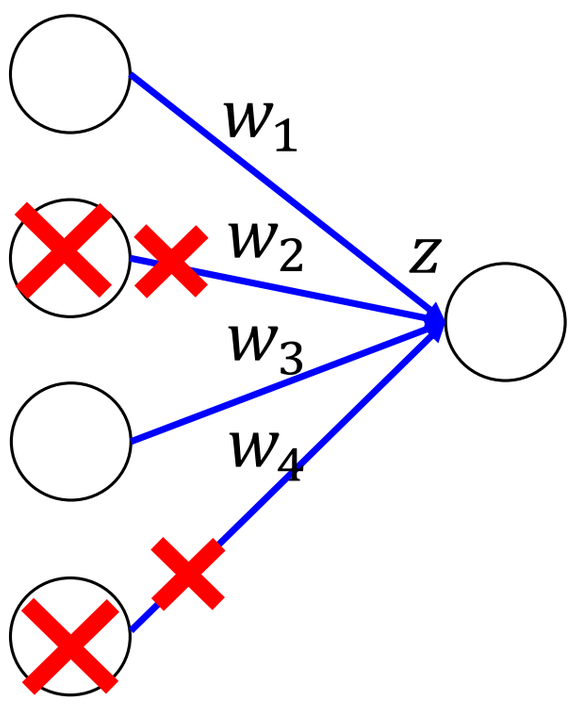

- When teams up, if everyone expect the partner will do the work, nothing will be done finally.
- However, if you know your partner will dropout, you will do better.
- When testing, no one dropout actually, so obtaining good results eventually.

# Dropout - Intuitive Reason

- Why the weights should multiply (1-p)% (dropout rate) when testing?

***Training of Dropout***

Assume dropout rate is 50%

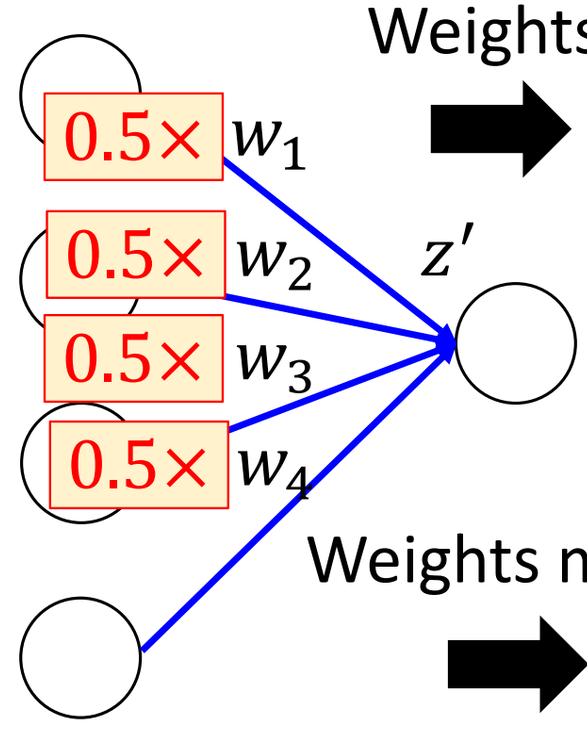



***Testing of Dropout***

No dropout

Weights from training

$z' \approx 2z$

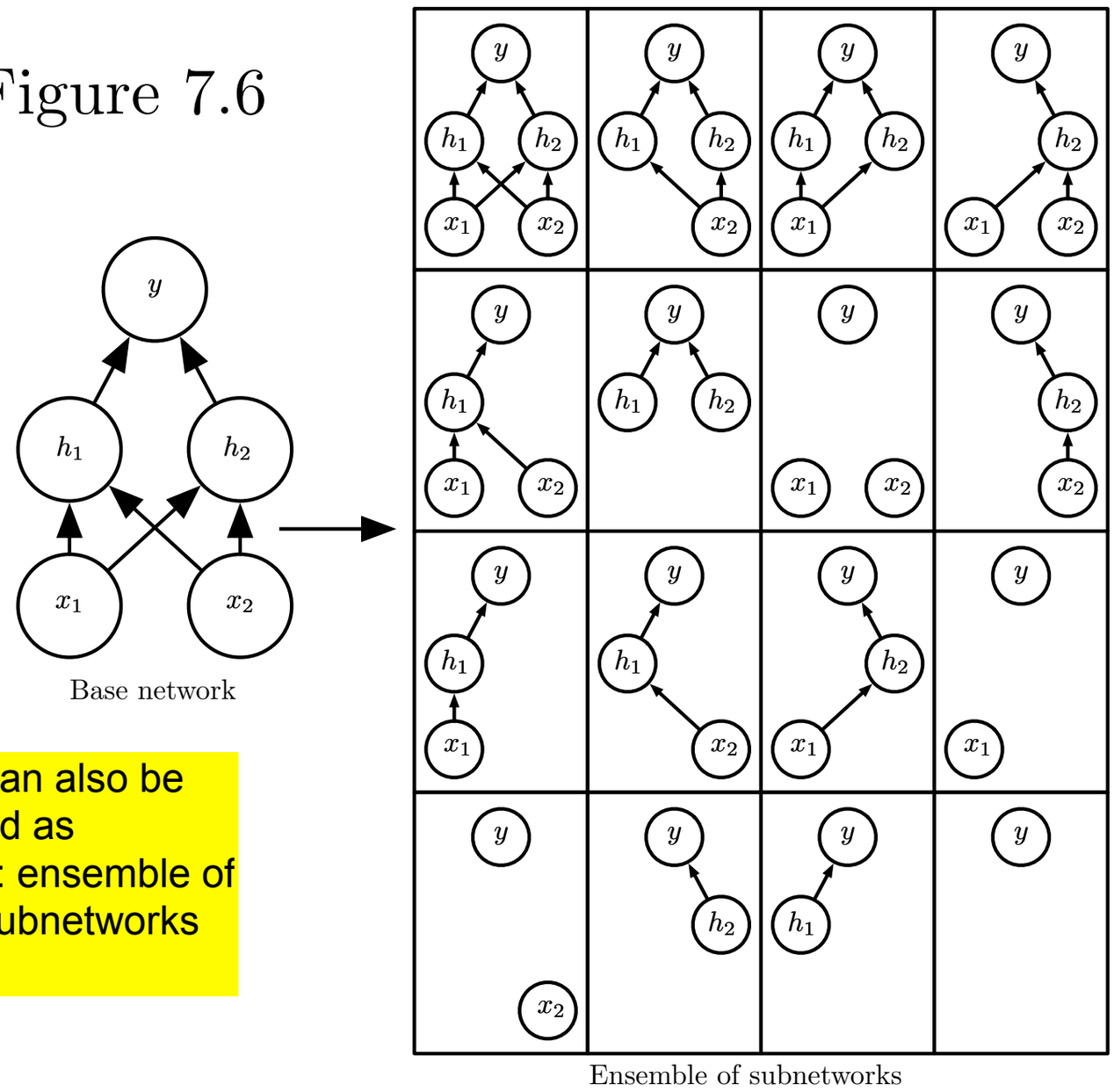



Weights multiply (1-p)%

$z' \approx z$

# Dropout

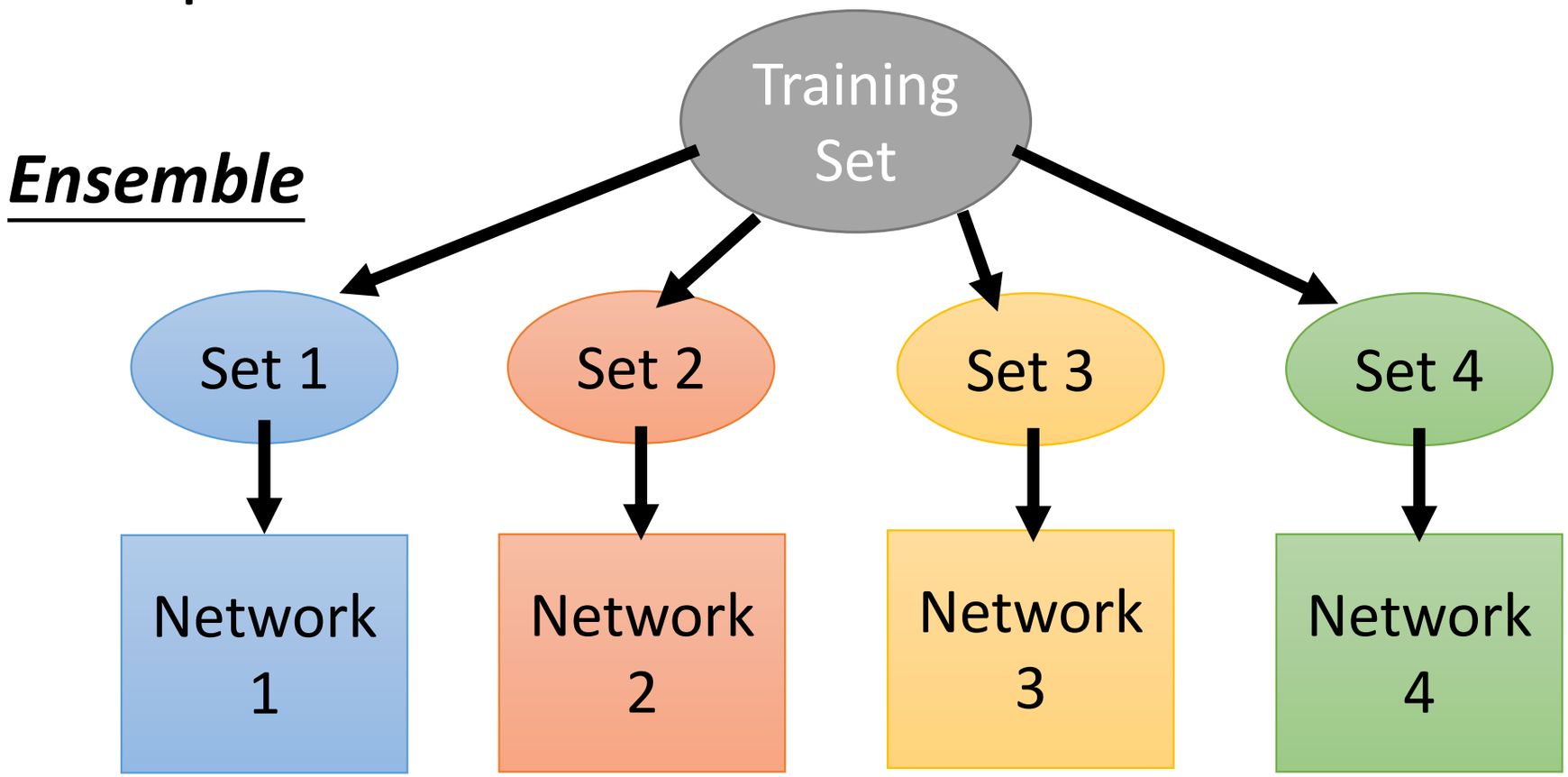
Figure 7.6




Dropout can also be considered as "bagging": ensemble of multiple subnetworks

(Goodfellow 2016)

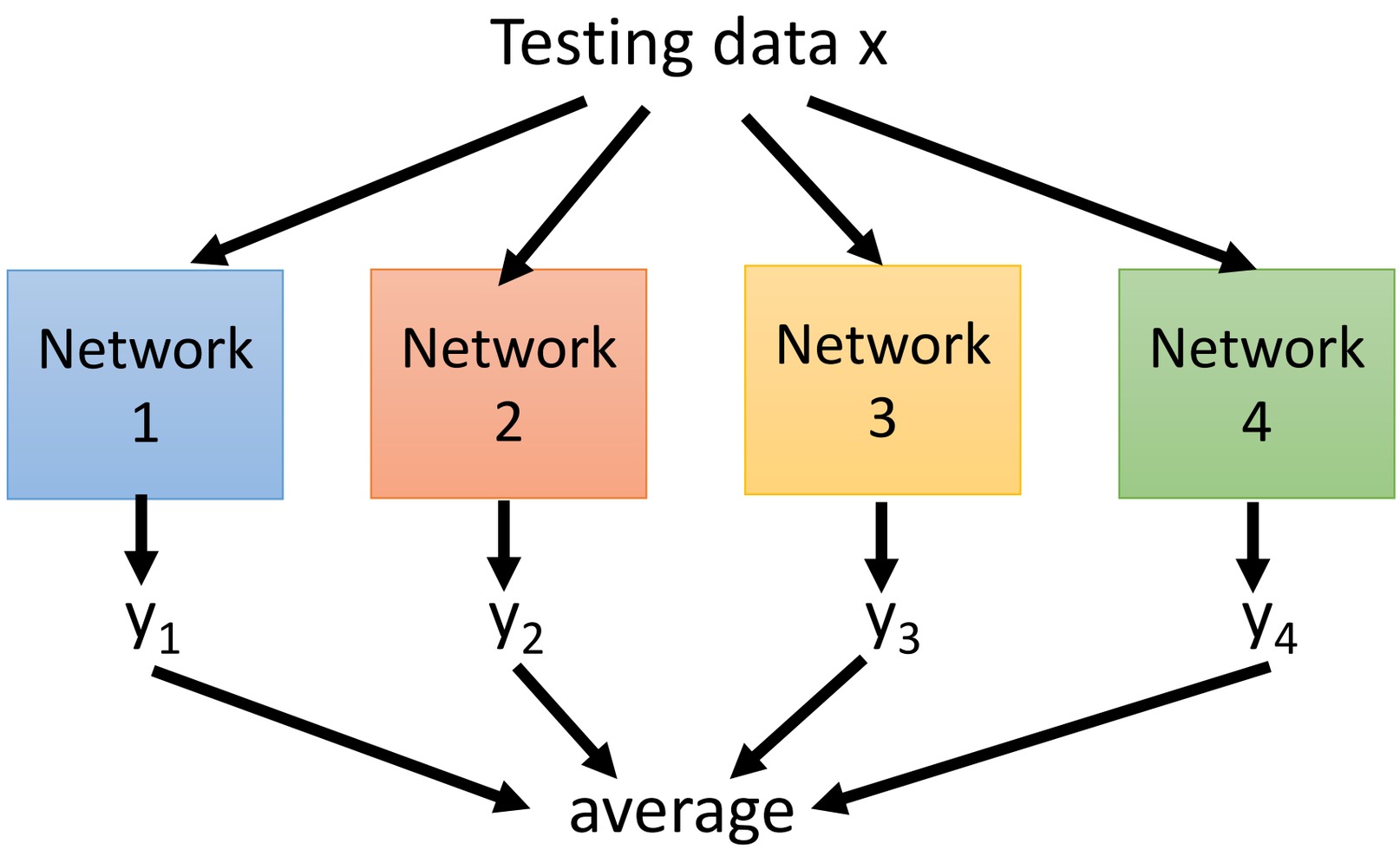
# Dropout is a kind of ensemble.



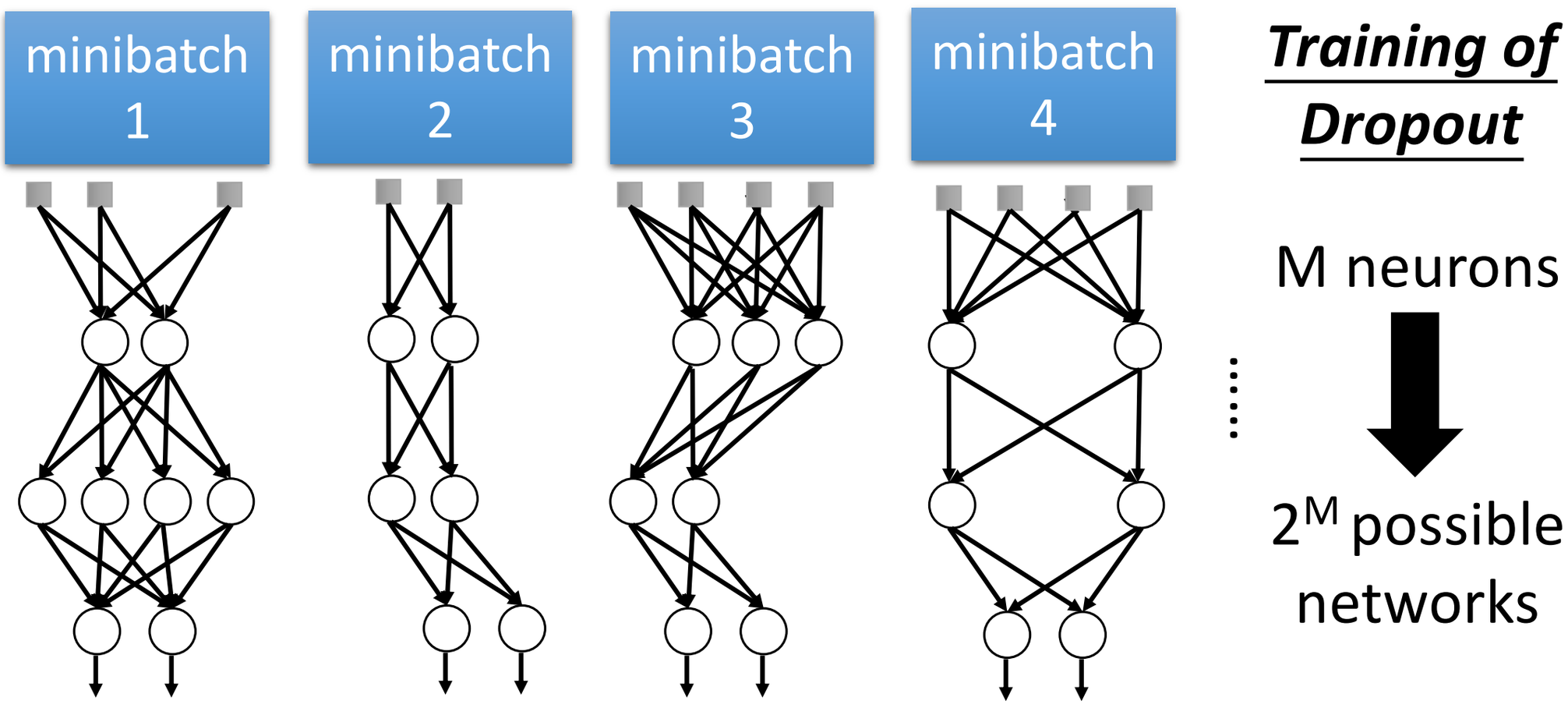

Train a bunch of networks with different structures

# Dropout is a kind of ensemble.

***Ensemble***

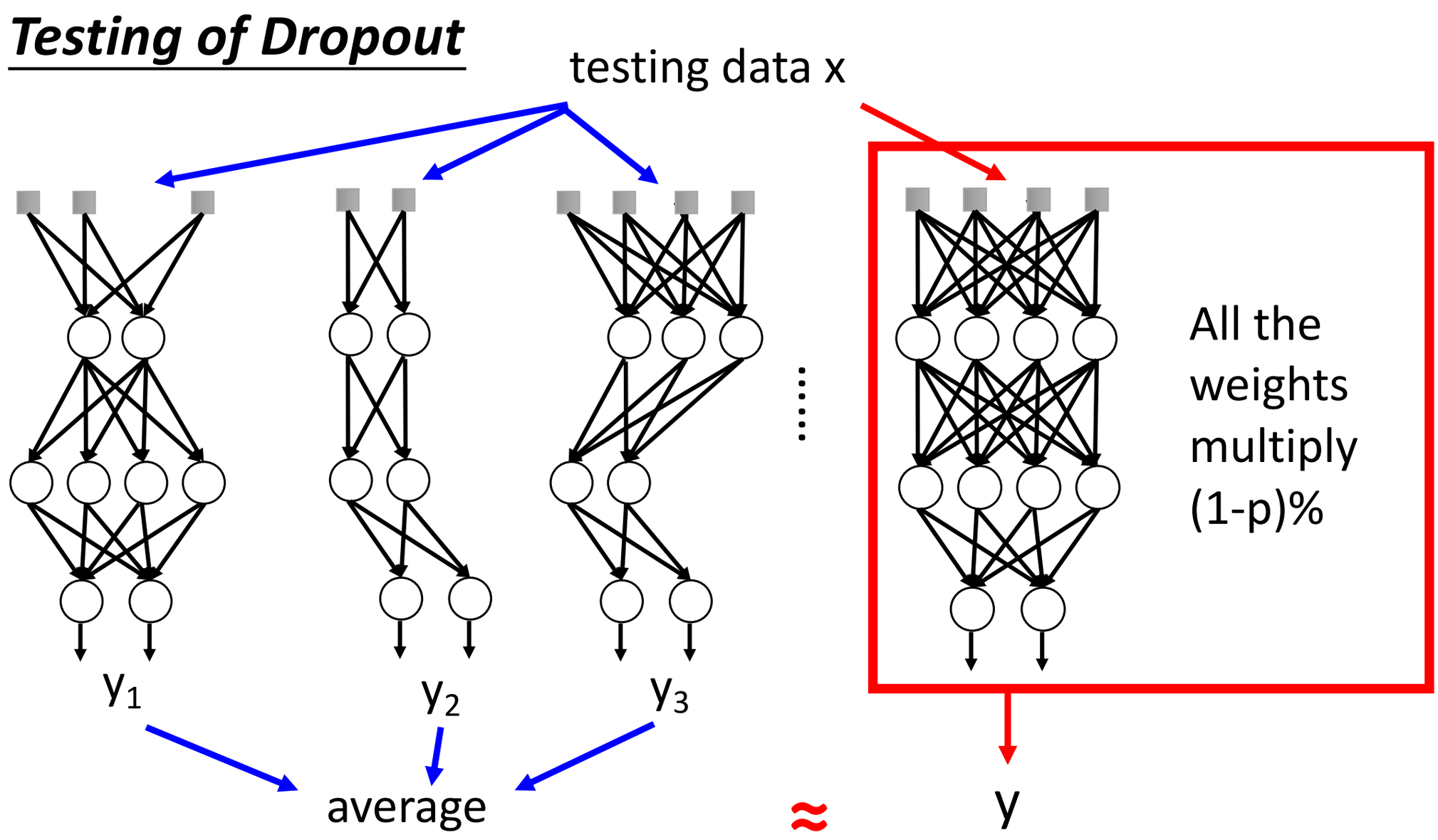


# Dropout is a kind of ensemble.




- Using one mini-batch to train one network
- Some parameters in the network are shared

# Dropout is a kind of ensemble.



# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

# Regularization 2: Early Stopping

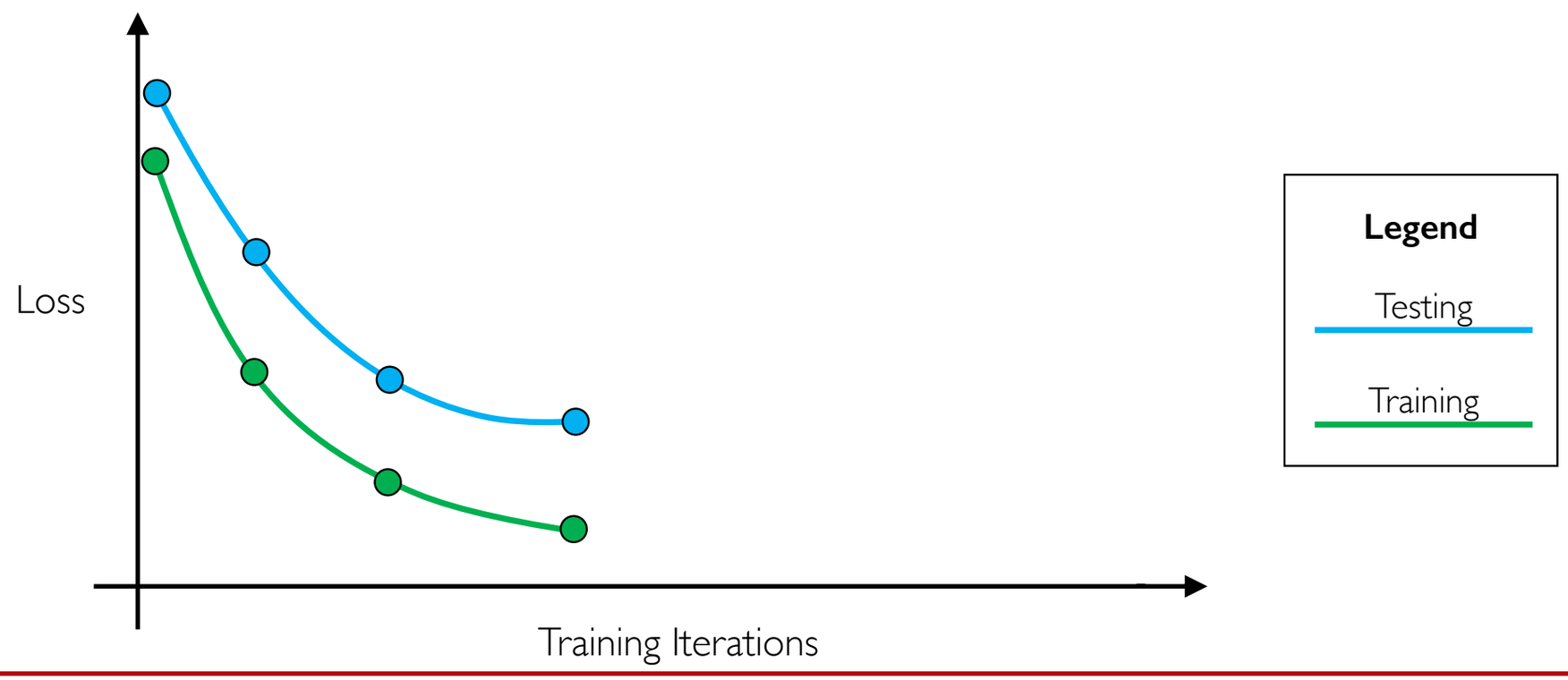
- Stop training before we have a chance to overfit

# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

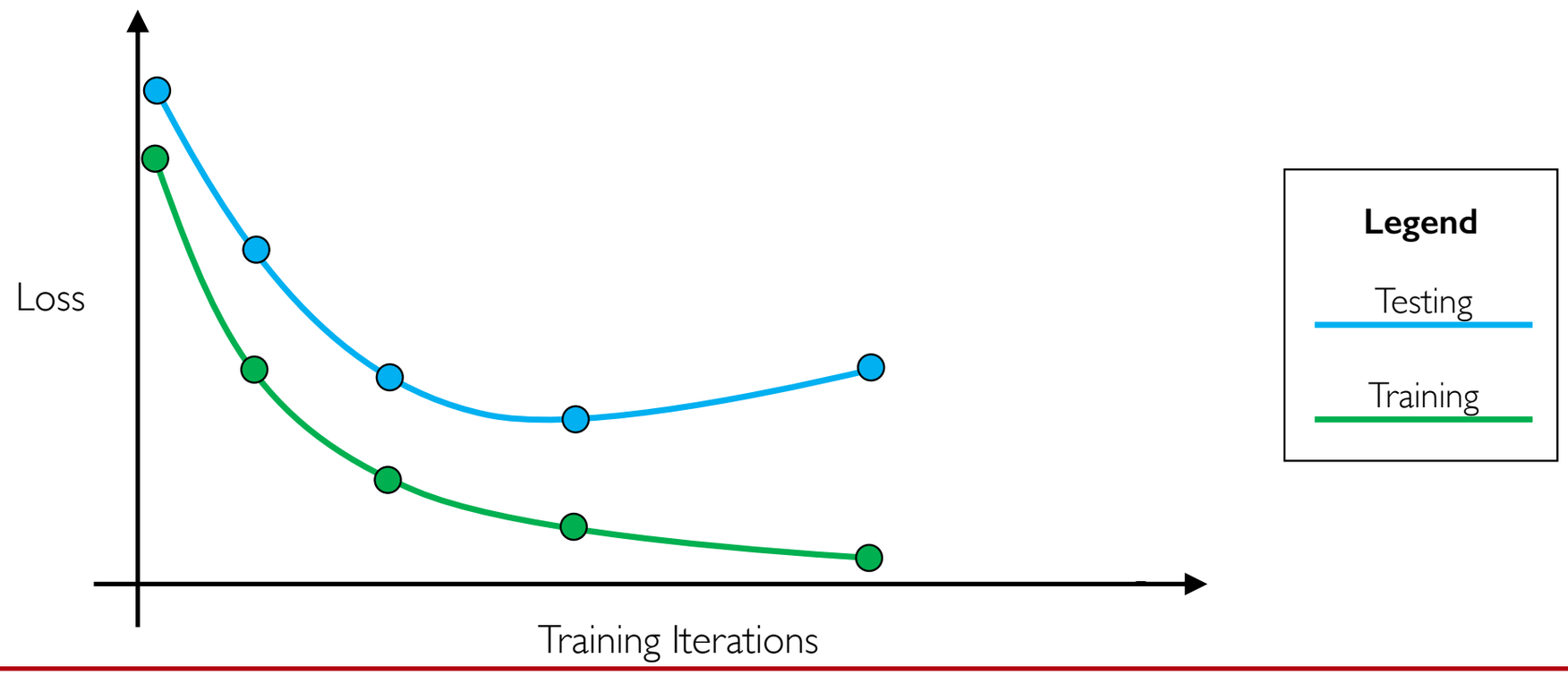
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



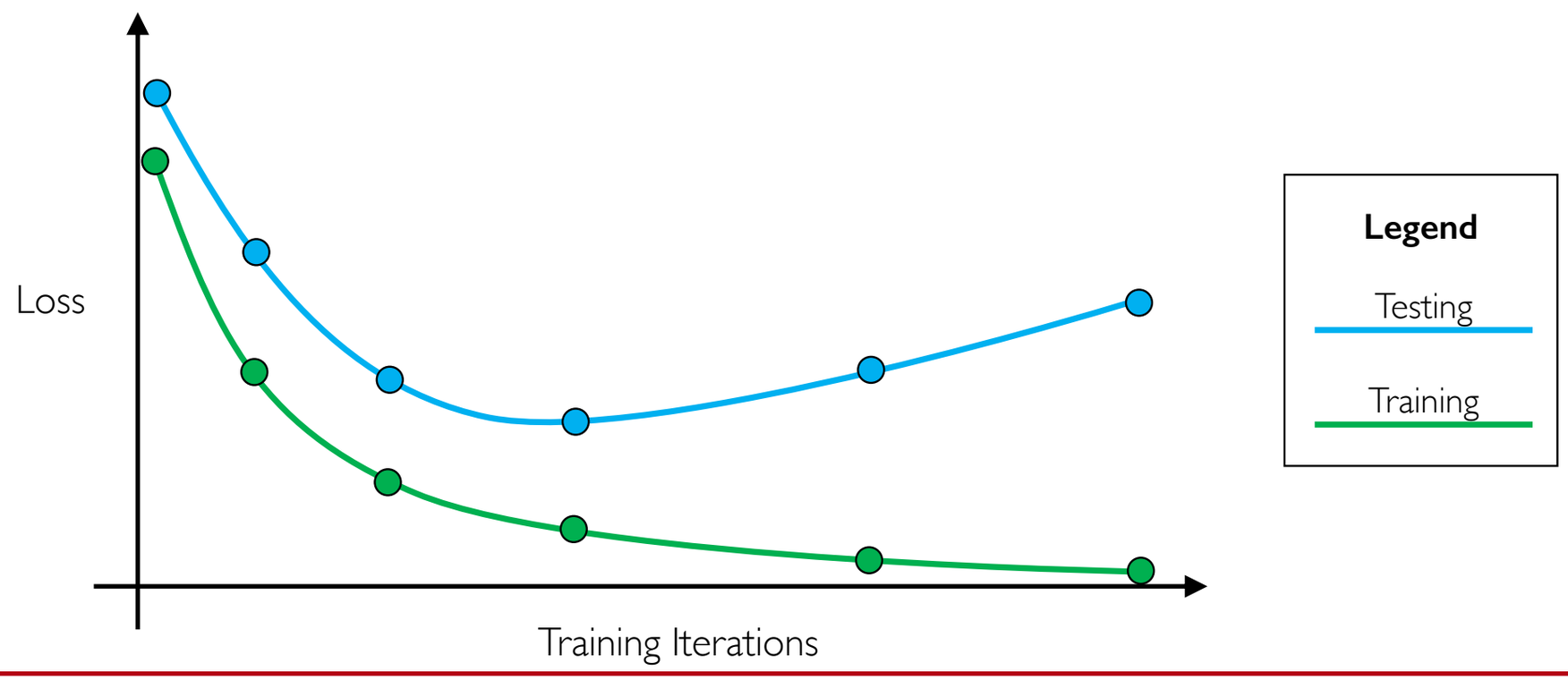
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



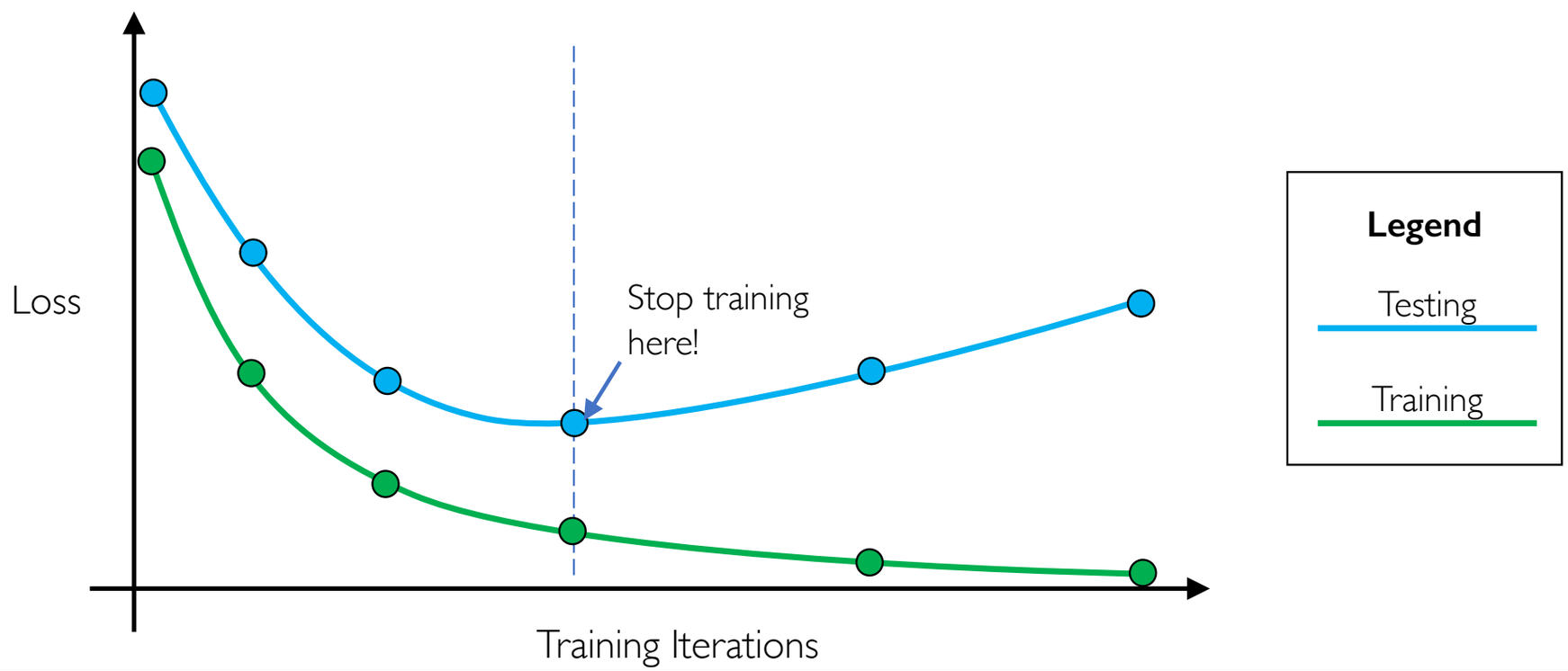
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



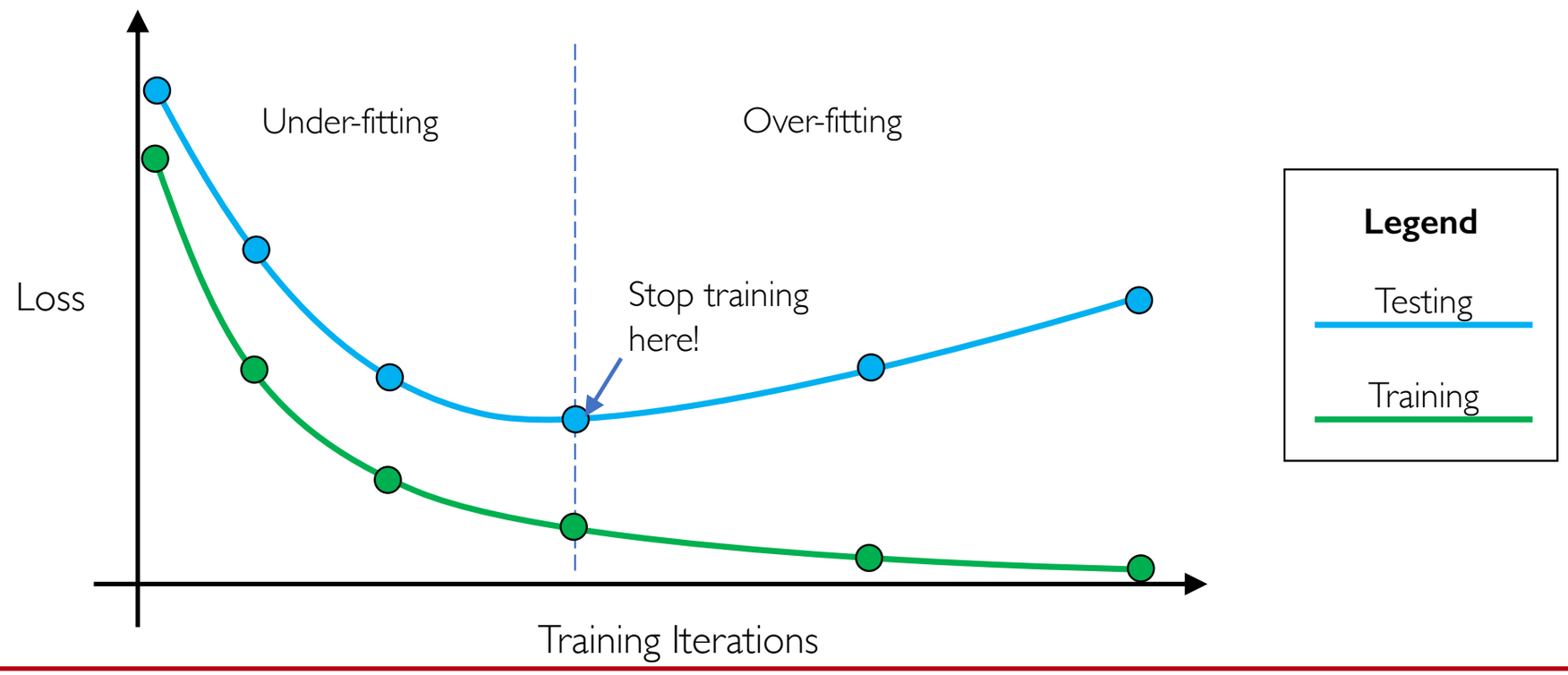
# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



# Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



# Regularization 3: Norm Penalties

- Extra penalties from weights $\boldsymbol{W}$ are added to the cost function
- Question: Why such penalties can regularize the learning? Intuition?
- **L1 Regularizer**:

$$C = \frac{1}{n}\sum_i \mathcal{L}(f(x^{(i)};\boldsymbol{\theta}),\ y^{(i)}) + \boldsymbol{\lambda}\sum_{\boldsymbol{w}\in\boldsymbol{\theta}} |\boldsymbol{w}|$$

  - Encourages sparsity, equivalent to MAP Bayesian estimation with Laplace prior on the weights

- **L2 Regularizer**:

$$C = \frac{1}{n}\sum_i \mathcal{L}(f(x^{(i)};\boldsymbol{\theta}),\ y^{(i)}) + \boldsymbol{\lambda}\sum_{\boldsymbol{w}\in\boldsymbol{\theta}} \boldsymbol{w}^2$$

  - Encourage small weights, equivalent to MAP Bayesian estimation with Gaussian prior on the weights

# Probabilistic Interpretation of Norm Penalty

- Bayesian Inference Model:
  - $y$ : observed data points ($y_1$,…, $y_N$)
  - $p\,(y|\theta)$, likelihood function, the probability of observing the data point $y$, e.g., for linear regression, $p(y|\theta) \sim N(\theta^T x, \sigma^{\mathbf{2}})$
  - $p\,(\theta)$: Prior of parameter $\theta$
  - MAP (maximum a posteriori probability estimate):

$$
\begin{aligned}
\hat{\theta}_{\mathbf{MAP}} &= \arg\max_{\theta} P(\theta|y) \\
&= \arg\max_{\theta} \frac{P(y|\theta)P(\theta)}{P(y)} \\
&= \arg\max_{\theta} P(y|\theta)P(\theta) \\
&= \arg\max_{\theta} \log(P(y|\theta)P(\theta)) \\
&= \arg\max_{\theta} \log P(y|\theta) + \log P(\theta)
\end{aligned}
$$

- Two possible priors:
  - Norm prior: $p\,(\theta) \sim N(0, \tau^2)$
  - Laplace prior: $p\,(\theta) \sim \mathrm{Laplace}(\mu, b) = \frac{1}{2b} e^{-\frac{|\theta-\mu|}{b}}$

# Probabilistic Interpretation of L2 Regularizer

- Let $\theta = (\beta_0, \beta_1, \dots, \beta_p)$
- Assume $p(y|\theta) \sim N\left(\theta^T x, \sigma^{\mathbf{2}}\right), p\,(\theta) \sim N(0, \tau^2)$

$$\hat{\theta}_{MAP} = \arg\max_{\beta} \left[ \log \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y_i - (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p}))^2}{2\sigma^2}} + \log \prod_{j=0}^{p} \frac{1}{\tau\sqrt{2\pi}} e^{-\frac{\beta_j^2}{2\tau^2}} \right]$$

$$= \arg\max_{\beta} \left[ -\sum_{i=1}^{n} \frac{(y_i - (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p}))^2}{2\sigma^2} - \sum_{j=0}^{p} \frac{\beta_j^2}{2\tau^2} \right]$$

$$= \arg\min_{\beta} \frac{1}{2\sigma^2} \left[ \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p}))^2 + \frac{\sigma^2}{\tau^2} \sum_{j=0}^{p} \beta_j^2 \right]$$

$$= \arg\min_{\beta} \left[ \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p}))^2 + \lambda \sum_{j=0}^{p} \beta_j^2 \right]$$

- Note that $\lambda = \frac{\sigma^{\mathbf{2}}}{\tau^2}$
- Very small $\tau^2$ (i.e. large $\lambda$) → coefficients will be very close to 0
- Large $\tau^2$ (i.e., small $\lambda$) → coefficients will not be affected much (no regularization!)

Reference:
http://bjlkeng.github.io/posts/probabilistic-interpretation-of-regularization/

# Probabilistic Interpretation of L1 Regularizer

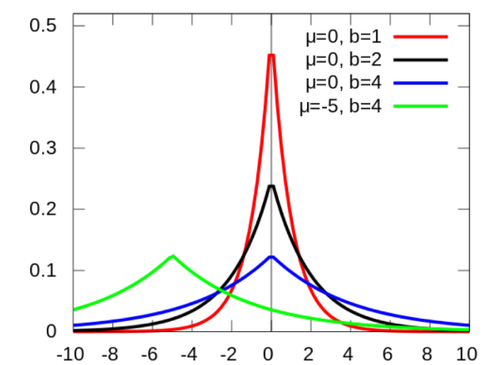
- Let $\theta = (\beta_0, \beta_1, \dots, \beta_p)$
- Assume $p(y|\theta) \sim N\left(\theta^T x, \sigma^2\right), p\,(\theta) \sim \text{Laplace}(0, b) = \frac{1}{2b} e^{-\frac{|\theta|}{b}}$

$$\hat{\theta}_{MAP} = \arg\max_{\beta} \left[ \log \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y_i - (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p}))^2}{2\sigma^2}} + \log \prod_{j=0}^{p} \frac{1}{2b} e^{-\frac{|\beta_j|}{2b}} \right]$$

$$= \arg\max_{\beta} \left[ -\sum_{i=1}^{n} \frac{(y_i - (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p}))^2}{2\sigma^2} - \sum_{j=0}^{p} \frac{|\beta_j|}{2b} \right]$$

$$= \arg\min_{\beta} \frac{1}{2\sigma^2} \left[ \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p}))^2 + \frac{\sigma^2}{b} \sum_{j=0}^{p} |\beta_j| \right]$$

$$= \arg\min_{\beta} \left[ \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_{i,1} + \dots + \beta_p x_{i,p}))^2 + \lambda \sum_{j=0}^{p} |\beta_j| \right]$$



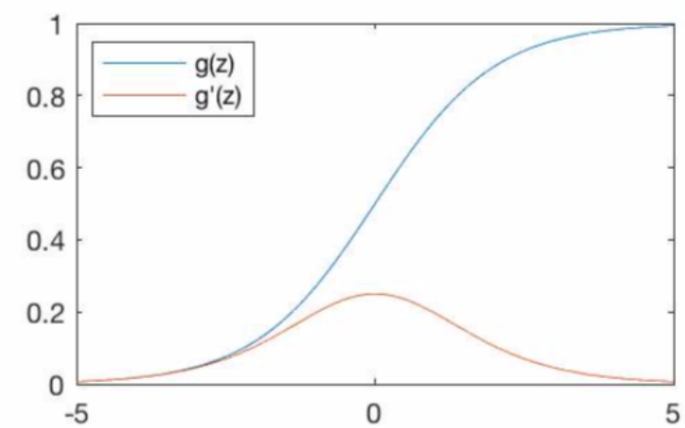

- Notice there is a sharp increase in the density at the mean in the density plot
- So, L1 promotes sparsity, i.e., zeroing out some of the coefficients.

# Regularizer 4: Batch Normalization

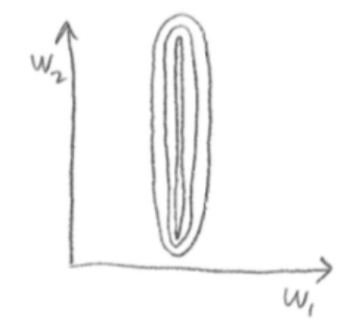
- Motivation**:**
  - Consider layers:

    $z = f_1(x, \theta_1), \ell = f_2(z, \theta_2),$

    $\theta_2 \leftarrow \theta_2 - \eta \frac{\partial f_2(z, \theta_2)}{\partial \theta_2}$
  - Reduce **Covariate Shift**:
    - Gradients depend on input values $x, z$
    - Stable distributions of $x, z$ make training more efficient
    - Reduce the need to compensate for the change in the distribution of $x, z$
  - Avoid **gradient vanish/exploding**
    - If $f_2 = \frac{1}{1+e^{-z}}$, when $|z|$ increases, $f_2{}'$ tends to 0
    - The gradient flowing down to $x$ will vanish and the model will train slowly
  - Resilience to input/parameter scale




| $x_1$ | $x_2$ | $t$ |
|---|---|---|
| 114.8 | 0.00323 | 5.1 |
| 338.1 | 0.00183 | 3.2 |
| 98.8 | 0.00279 | 4.1 |
| ⋮ | ⋮ | ⋮ |



# Batch Normalization (BN) During Training Stage

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: $\gamma$, $\beta$

**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

For each activation, learnable parameters $\gamma, \beta$ scale and shift the value to restore the original activation representation, e.g. sigmoid

# Batch Normalization (BN) During Inference Stage

- During testing or inference phase, we might pass only one sample at a time
- It doesn't make sense to find mean and variance on a single sample.
- Instead, a running averages of mean and variance of batches during training are used as mean and variance values

**for** $k = 1 \ldots K$ **do** // Note, k is the layer with batch normalization

// For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.

Process multiple training mini-batches $\mathcal{B}$, each of size $m$, and average over them:

$$\mathrm{E}[x] \leftarrow \mathrm{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$

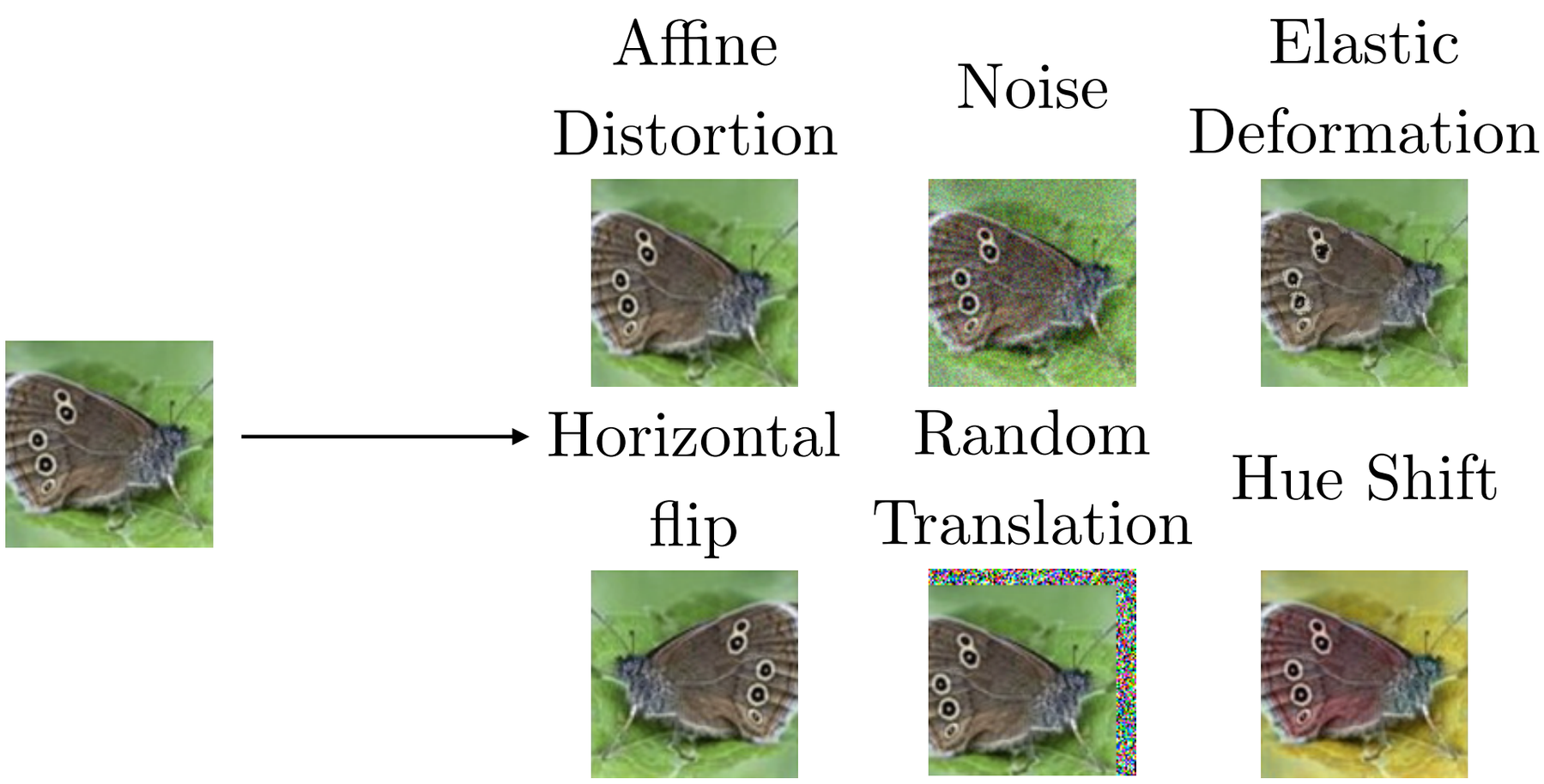
$$\mathrm{Var}[x] \leftarrow \frac{m}{m-1}\mathrm{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

In $N_{\mathrm{BN}}^{\mathrm{inf}}$, replace the transform $y = \mathrm{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}\right)$

**end for**

Other Regularizers:

# Dataset Augmentation




(Goodfellow 2016)

# Multi-Task Learning

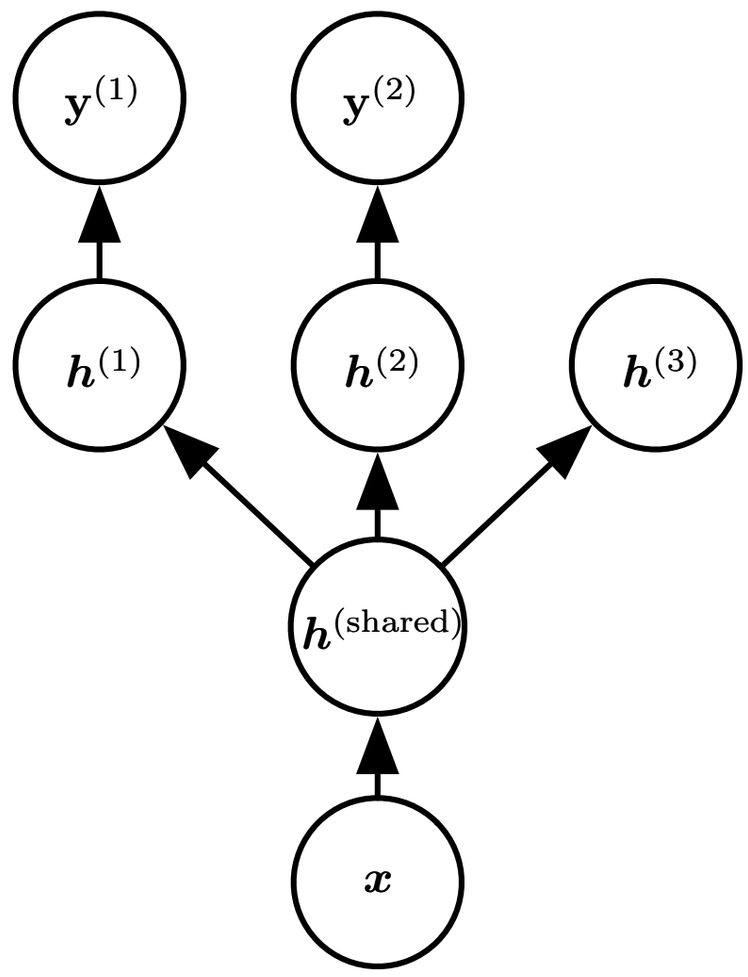
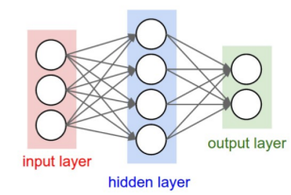



Figure 7.2

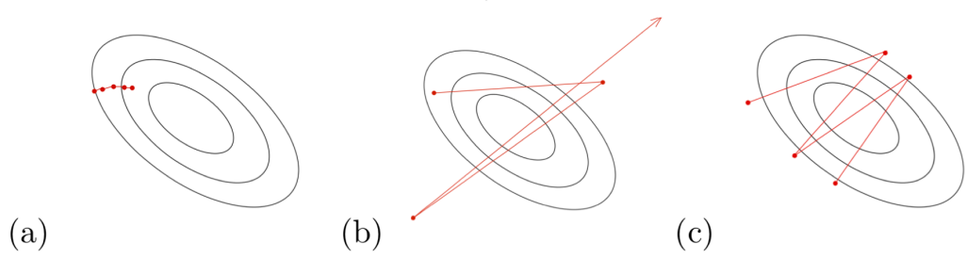
(Goodfellow 2016)

# Optimization in Practice

| Problem | Diagnostics | Workarounds |
|---|---|---|
| incorrect gradients | finite differences | fix them, or use an autodiff package |
| local optima | (hard) | random restarts |
| symmetries | visualize $\mathbf{W}$ | initialize $\mathbf{W}$ randomly |
| slow progress | slow, linear training curve | increase $\alpha$ |
| instability | cost increases | decrease $\alpha$ |
| oscillations | fluctuations in training curve | decrease $\alpha$; momentum |
| fluctuations | fluctuations in training curve | decay $\alpha$; iterate averaging |
| dead/saturated units | activation histograms | initial scale of $\mathbf{W}$; ReLU |
| badly conditioned curvature | (hard) | normalization; momentum; Adam; second-order opt. |




Q: What if initialize w, b = 0 ?




Animations of learning process dynamics:
http://cs231n.github.io/neural-networks-3/#anneal

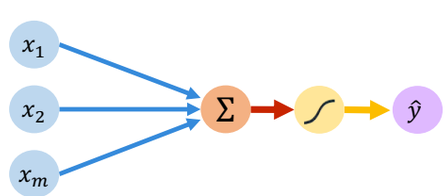
Figure 3: **(a)** Slow progress due to a small learning rate. **(b)** Instability due to a large learning rate. **(c)** Oscillations due to a large learning rate.

https://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/readings/L08%20Optimization.pdf

# Core Foundation Review

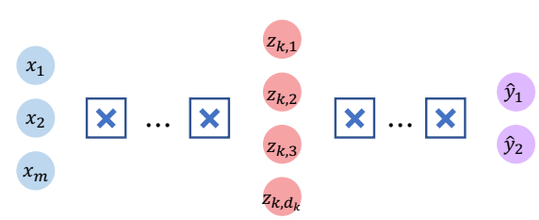
**The Perceptron**

- Structural building blocks
- Nonlinear activation functions



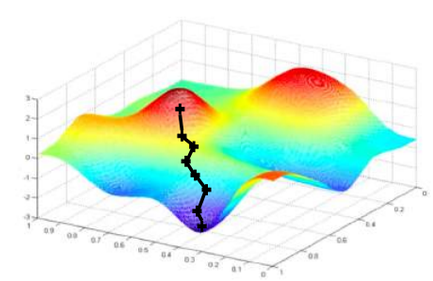

**Neural Networks**

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation




**Training in Practice**

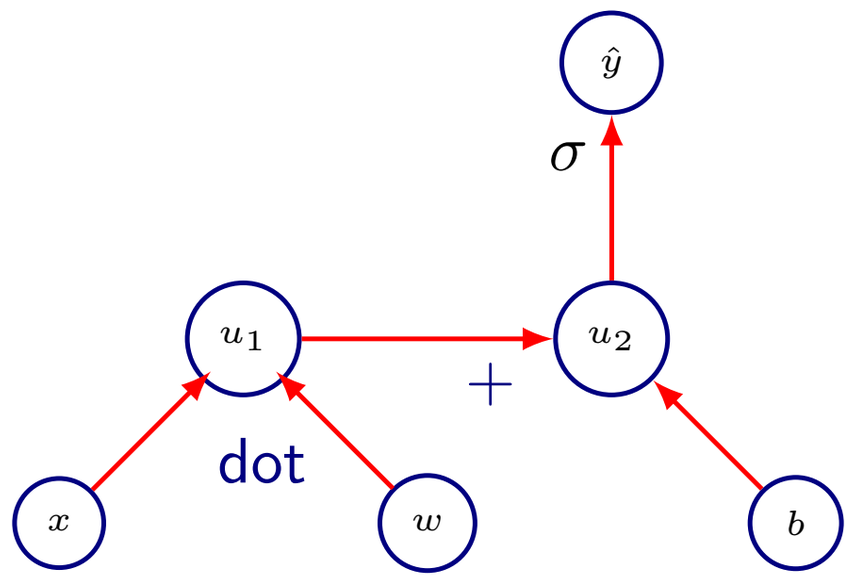
- Adaptive learning
- Batching
- Regularization



# Computational Graphs

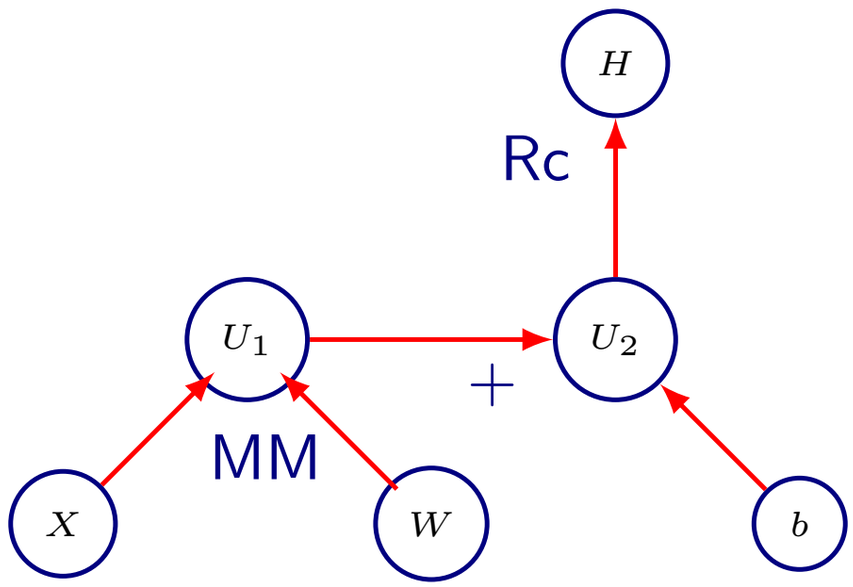
- Formalize computation as graphs
- **Nodes** indicate variables (scalar, vector, tensor or another variable)
- **Operations** are simple functions of one or more variables
- Our graph language comes with a set of **allowable** operations
- Examples:

# Logistic Regression




- Computes $\hat{y} = \sigma(\mathbf{x}^T\mathbf{w} + b)$

$$H = \max\{0, XW + b\}$$

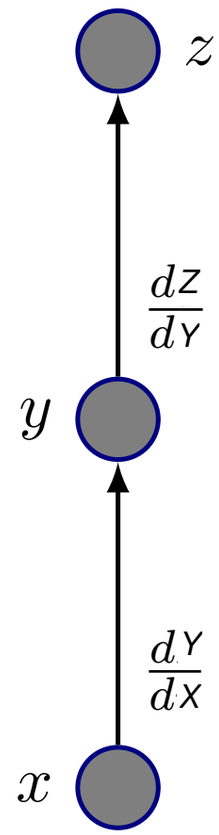



MM is matrix multiplication and Rc is ReLU activation

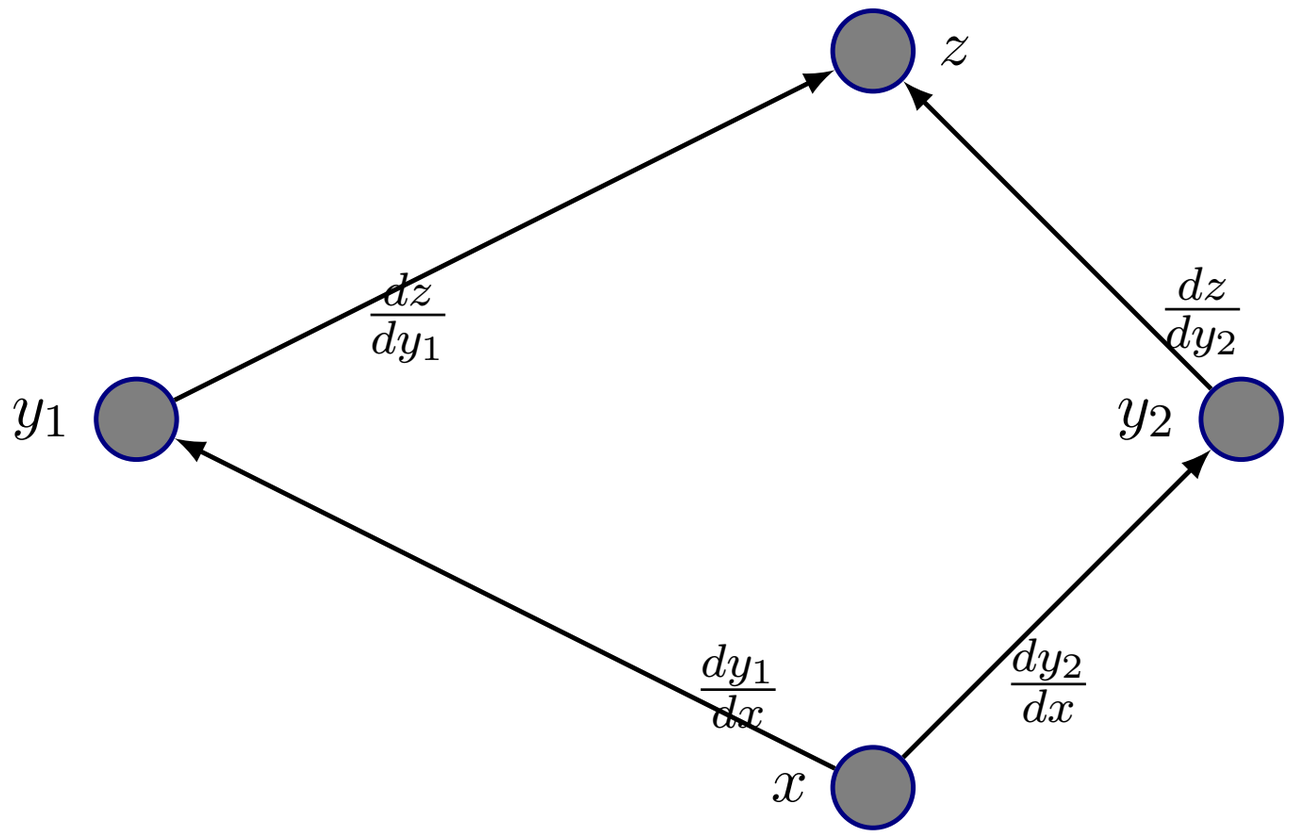
# Back to backprop: Chain Rule

- Backpropagation computes the chain rule, in a manner that is highly efficient
- Let $f, g : \mathbb{R} \to \mathbb{R}$
- Suppose $y = g(x)$ and $z = f(y) = f(g(x))$
- Chain rule:

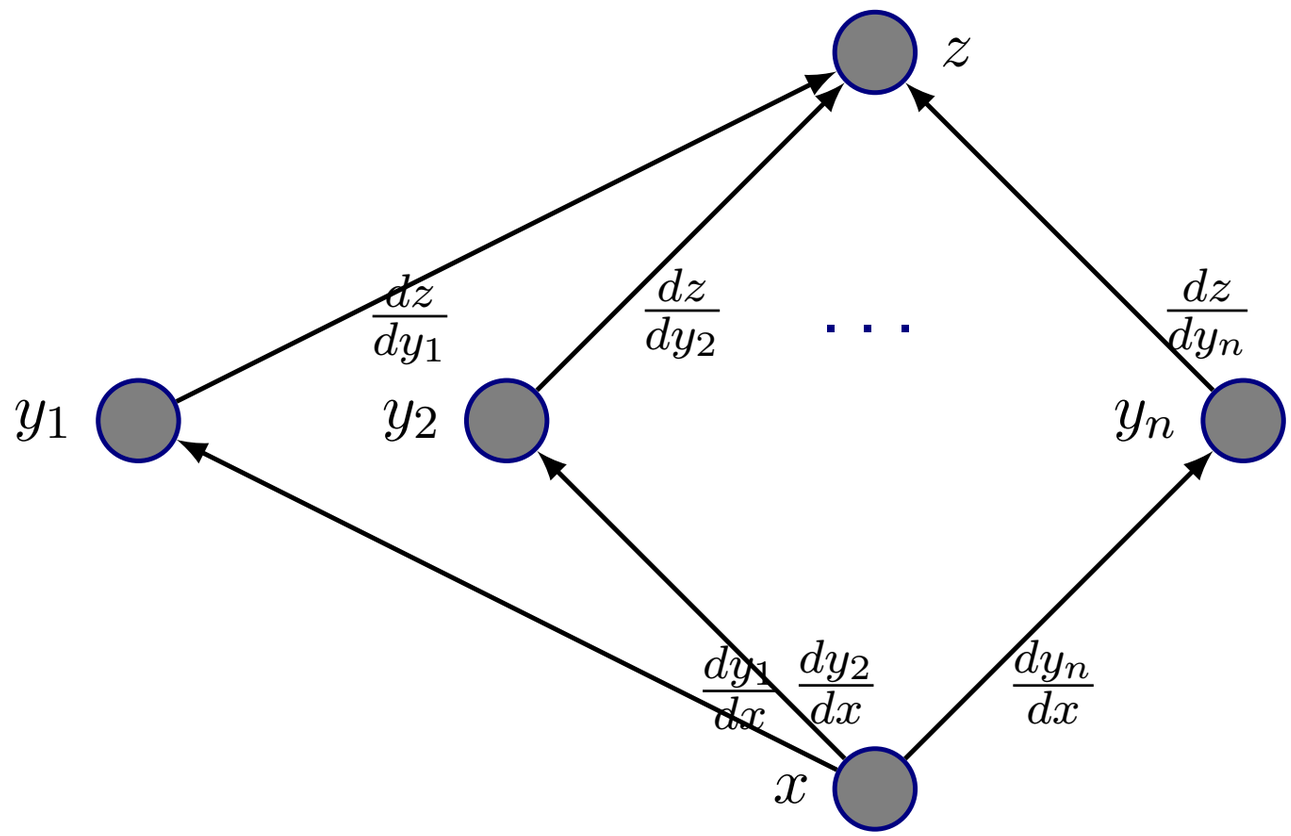
$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$




Chain rule: $\dfrac{dz}{dx} = \dfrac{dz}{dy}\dfrac{dy}{dx}$




Multiple Paths: $\frac{dz}{dx} = \frac{dz}{dy_1}\frac{dy_1}{dx} + \frac{dz}{dy_2}\frac{dy_2}{dx}$




Multiple Paths: $\frac{dz}{dx} = \sum_j \frac{dz}{dy_j} \frac{dy_j}{dx}$

# Recursive Application in a Computational Graph

- Fpropagation: Visit nodes in the order after a topological sort
- Compute the value of each node given its ancestors
- Bpropagation: Output gradient = 1
- Now visit nods in reverse order
- Compute gradient with respect to each node using gradient with respect to successors
- Successors of $x$ in previous slide $\{y_1, y_2, \ldots y_n\}$:

$$\frac{dz}{dx} = \sum_{i=1}^{n} \frac{dz}{dy_i} \frac{dy_i}{dx}$$

# Chain Rule

- We can ofcourse also write this in terms of tensors
- Let the gradient of $z$ with respect to a tensor $\mathbf{X}$ be $\nabla_{\mathbf{X}} z$
- If $\mathbf{Y} = g(\mathbf{X})$ and $z = f(\mathbf{Y})$, then:

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}$$

# Automatic Differentiation

- Computation of the gradient can be automatically inferred from the symbolic expression of fprop
- Every node type needs to know:
  - How to compute its output
  - How to compute its gradients with respect to its inputs *given* the gradient w.r.t its outputs
- Makes for rapid prototyping

# Symbol to Symbol

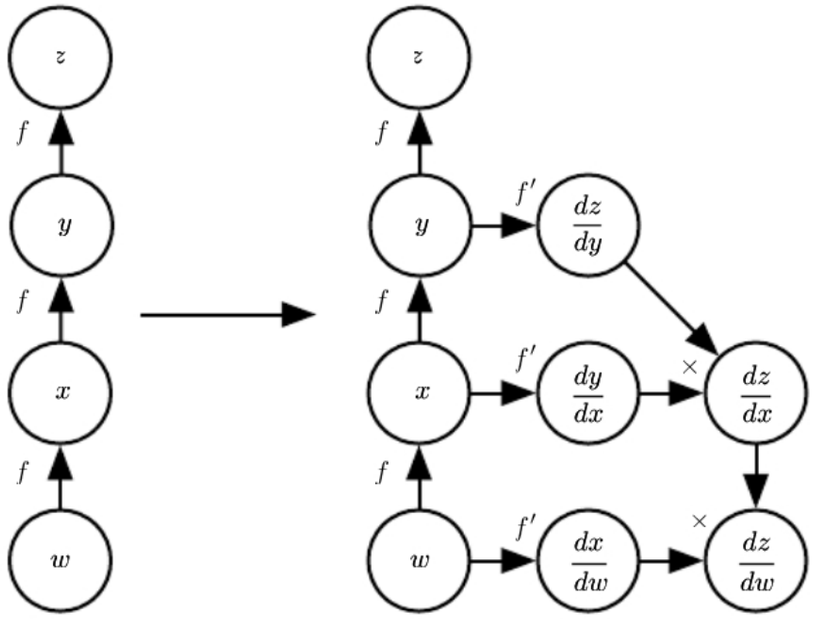
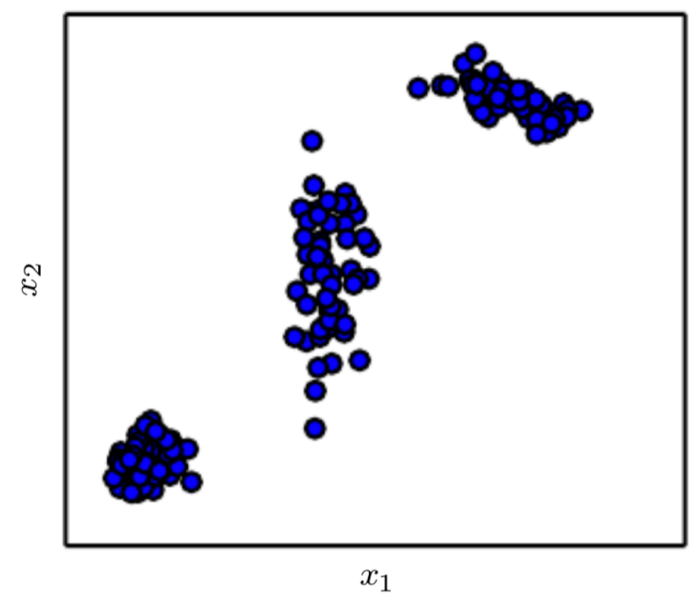



Figure: Goodfellow *et al.*

- In this approach backpropagation never accesses any numerical values
- Instead it just adds nodes to the graph that describe how to compute derivatives
- A graph evaluation engine will then do the actual computation
- Approach taken by Theano and TensorFlow

# Mixtures of Distributions

- A mixture distribution is made up of several component distributions.
- On each trial, the component which generates the sample is determined by sampling a multinoulli distribution:

$$p(x) = \sum_i p(c = i)\, p(x \mid c = i)$$

  - $p(x|c = i)$ can be Gaussians
  - $p(c = i)$ is the multinoulli distribution over components, a.k.a. **prior probability**
  - Often a **posterior probability** $p(c = i \mid x)$ is computed after is $x$ observed
- Gaussian mixture: $p(x|c = i) \sim N(\mu(i), \Sigma(i))$
- A Gaussian mixture model is a universal approximator of densities