LEVEL 2: Neural Networks and Deep Learning

Day 3

Course Outline

- Machine Learning Limitations
- Neural Networks and Deep Learning
- Vectorization
- Activation functions
- Backpropagation
- Weights Initialization
- Optimization
- Learning rate

Learning Objectives

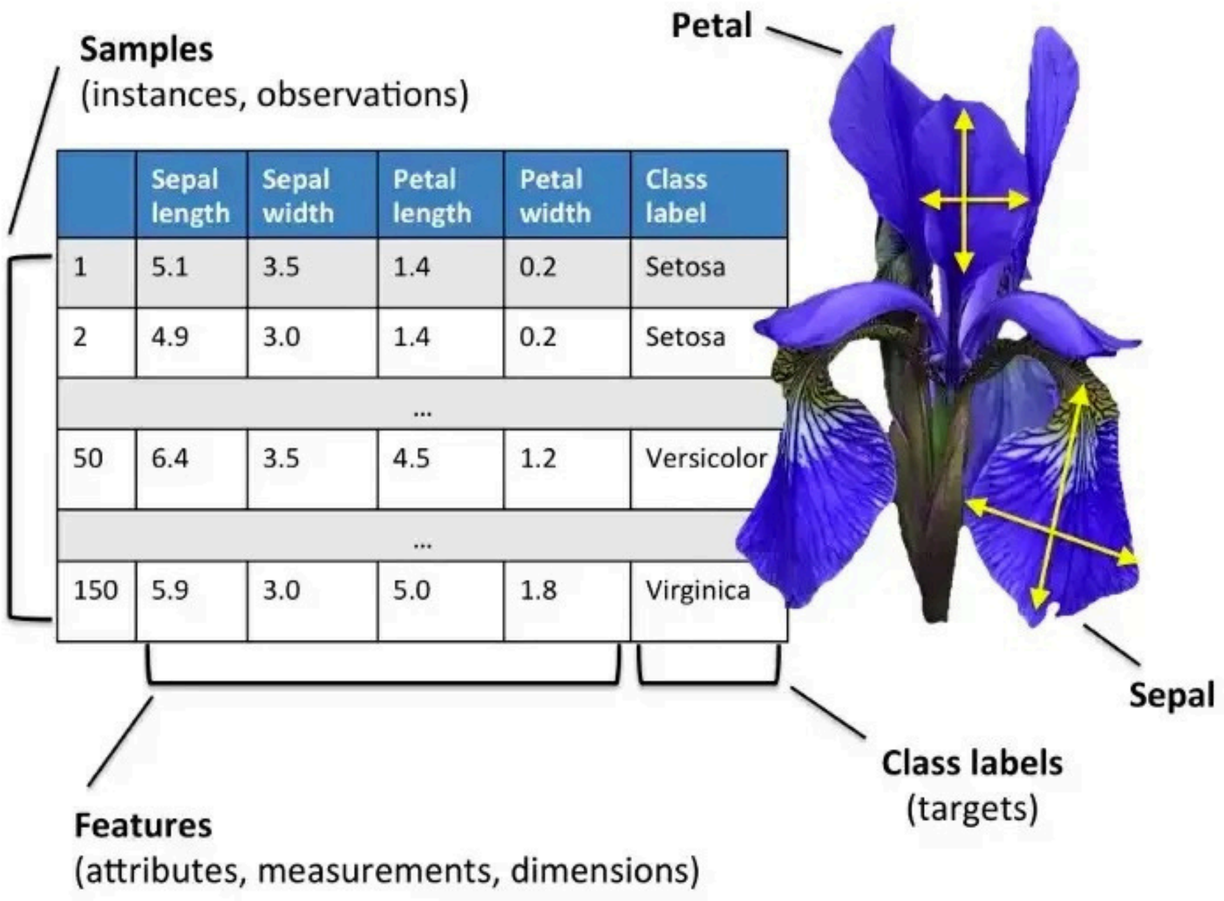
- Understand the key limitations of traditional Machine Learning models
- Explain how Neural Networks and Deep Learning address these limitations
- Apply vectorization to efficiently process multiple inputs
- Describe the role of activation functions in introducing non-linearity
- Understand the core idea of backpropagation for training neural networks
- Recognize the importance of proper weight initialization
- Identify common optimization methods used in deep learning
- Explain how the learning rate affects convergence and training stability

Machine Learning Limitations

Machine Learning Limitations

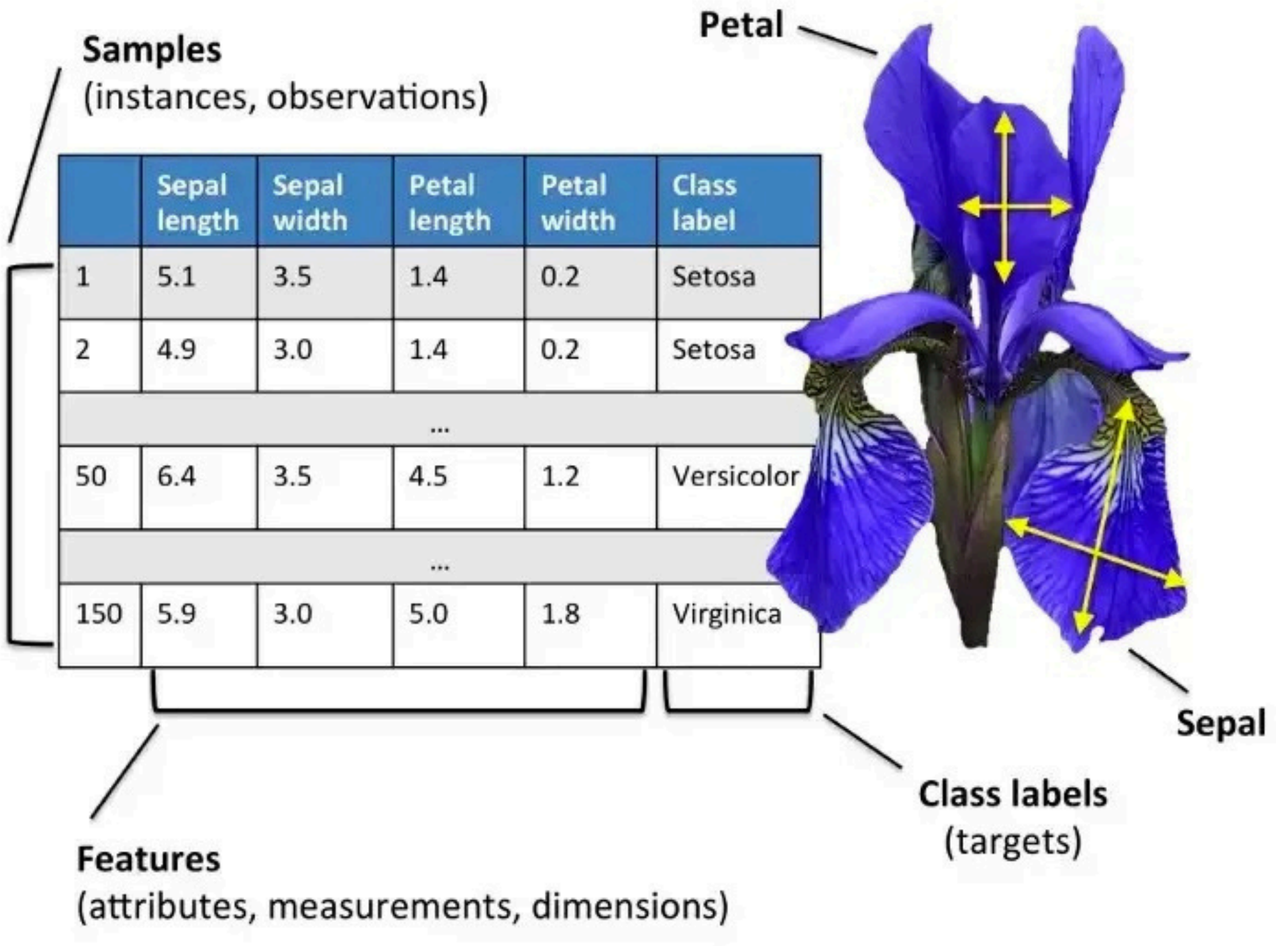
- How to classify these flowers?

Machine Learning Limitations



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

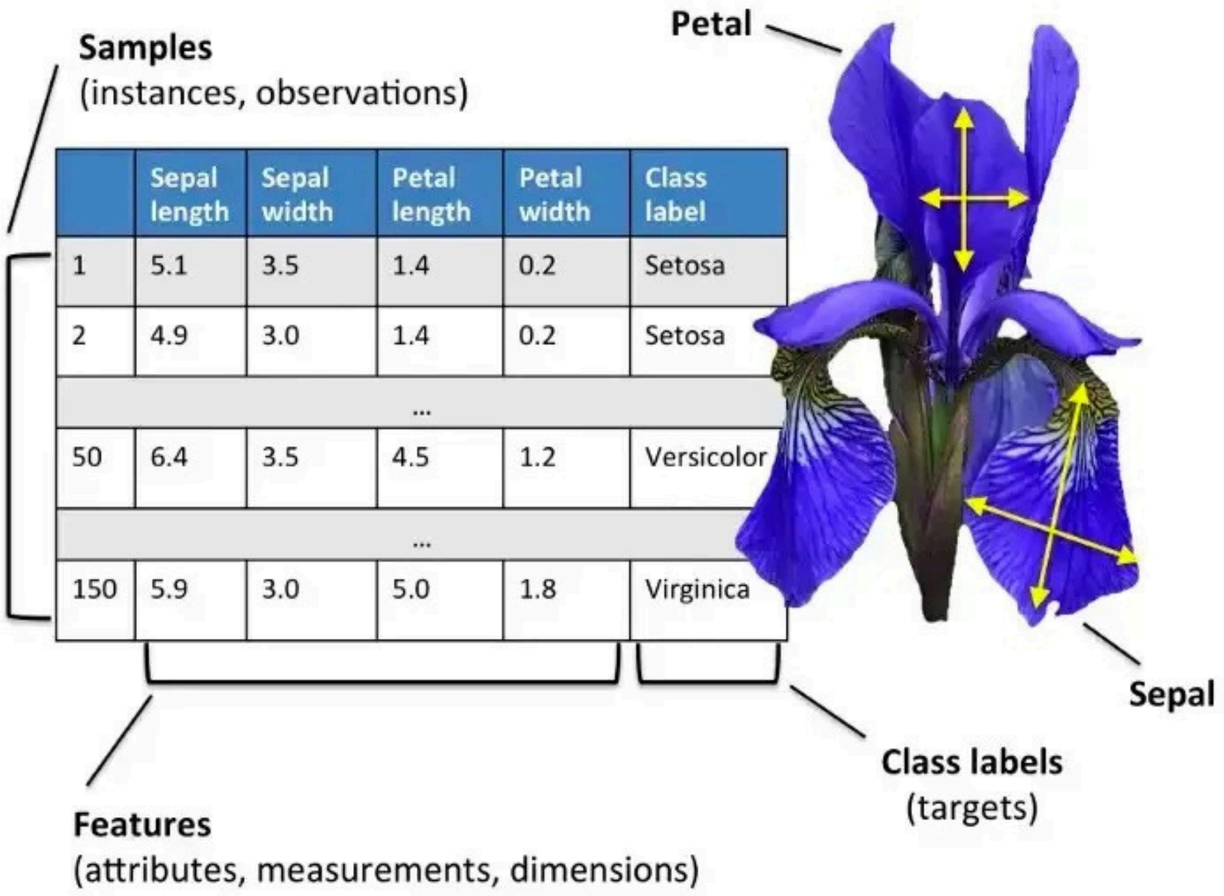
Machine Learning Limitations



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

What's the problem?

Machine Learning Limitations



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

- Human-designed features capture only simple, linear patterns.

Machine Learning Limitations

512 × 512 pixels

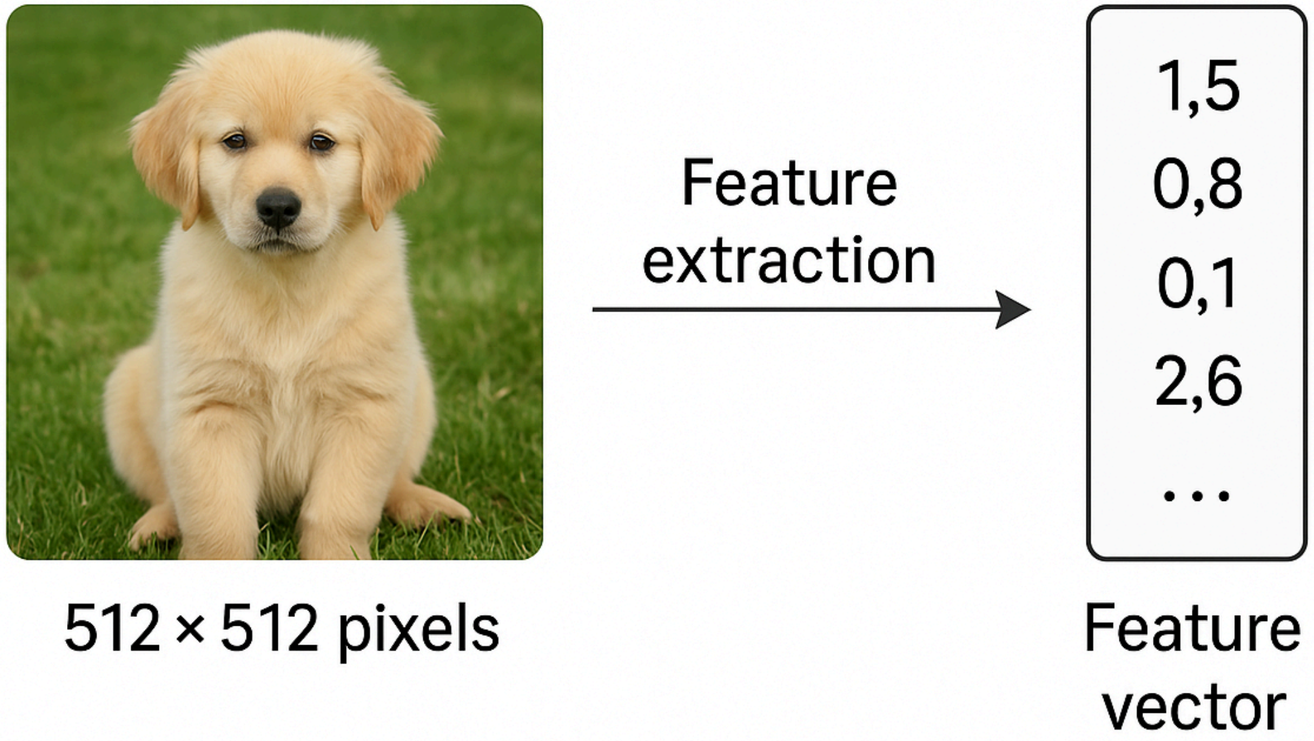
Feature extraction

1,5
0,8
0,1
2,6
...

Feature vector

What's the problem here?

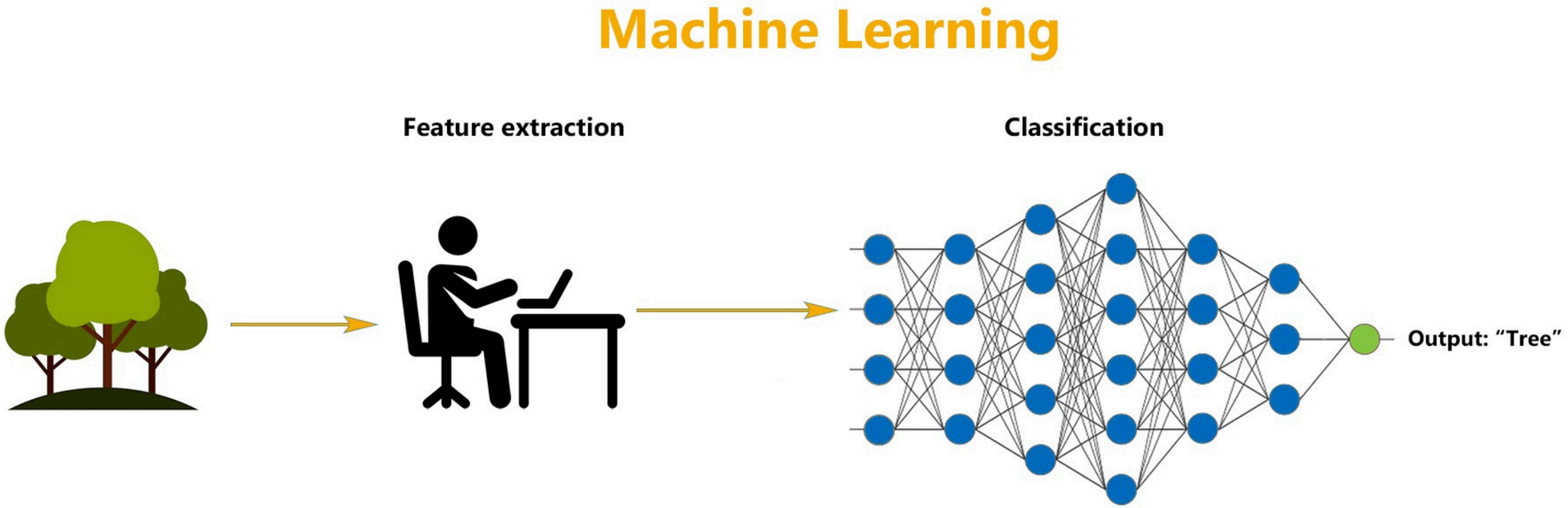
Machine Learning Limitations



- Most raw data information is lost when reduced to a few features.

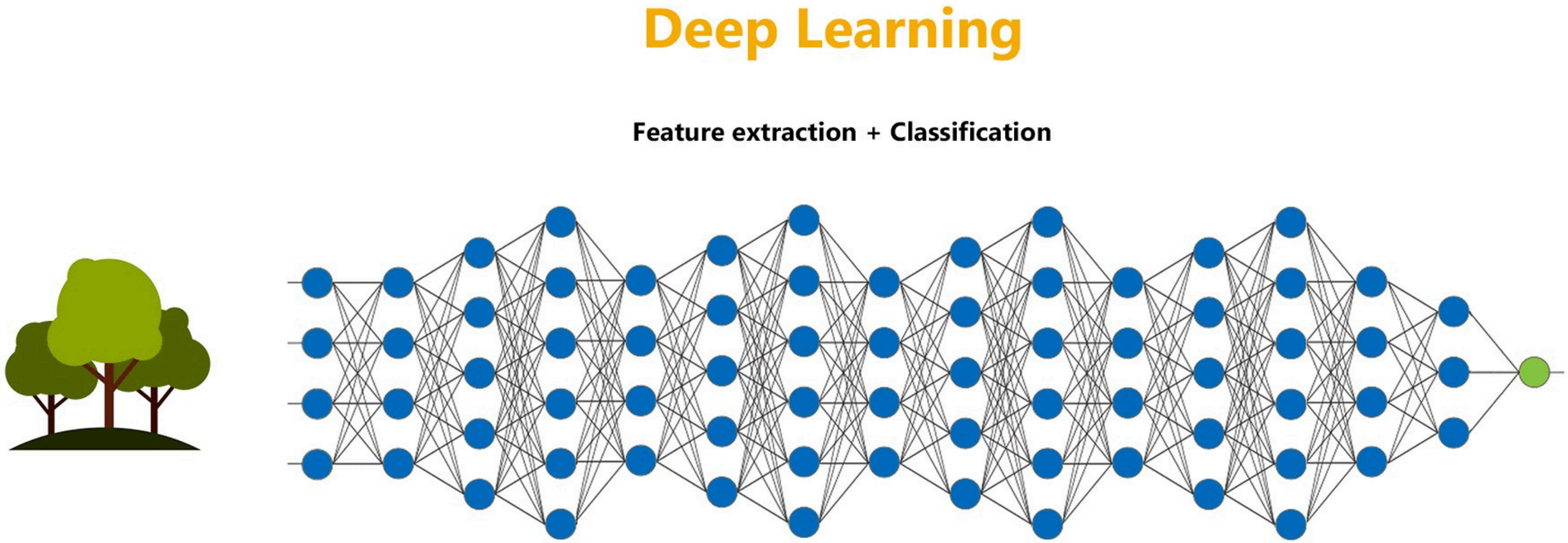
Machine Learning Limitations

- Traditional ML needs manually extracted features, while deep learning learns features directly from images automatically.



Machine Learning Limitations

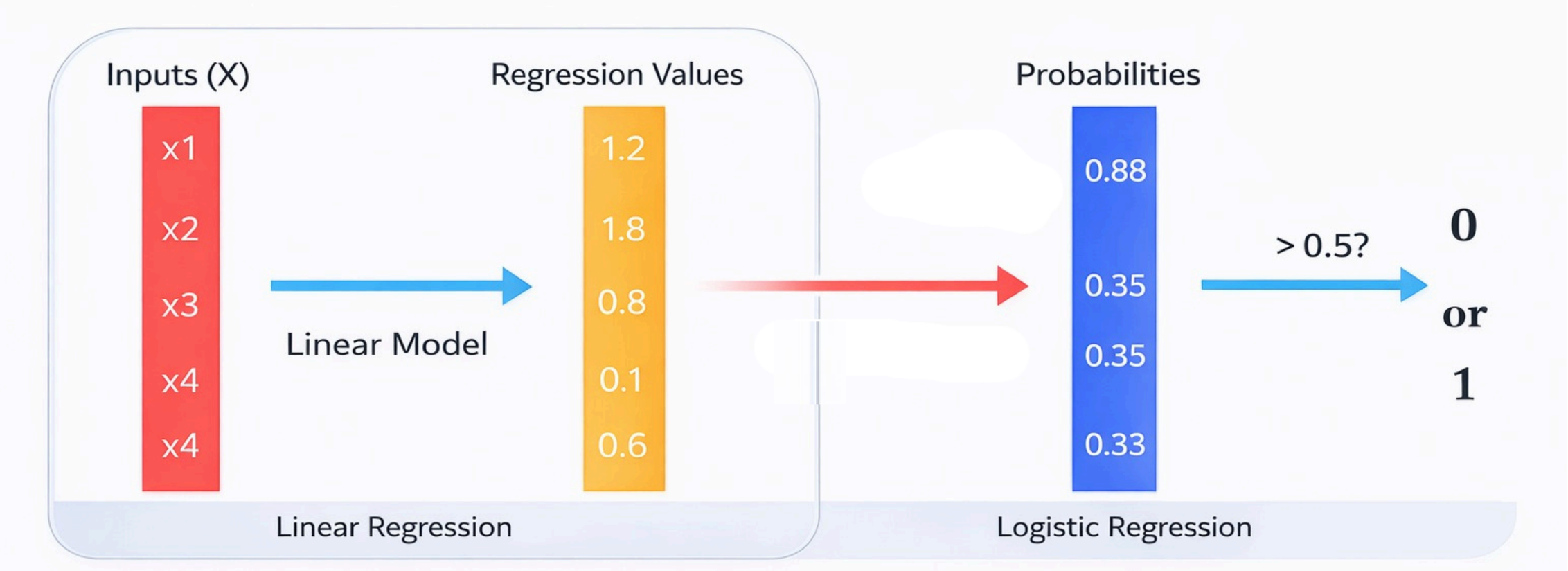
- Traditional ML needs manually extracted features, while deep learning learns features directly from images automatically.



Neural Networks and Deep Learning

Neural Networks

Logistic Regression



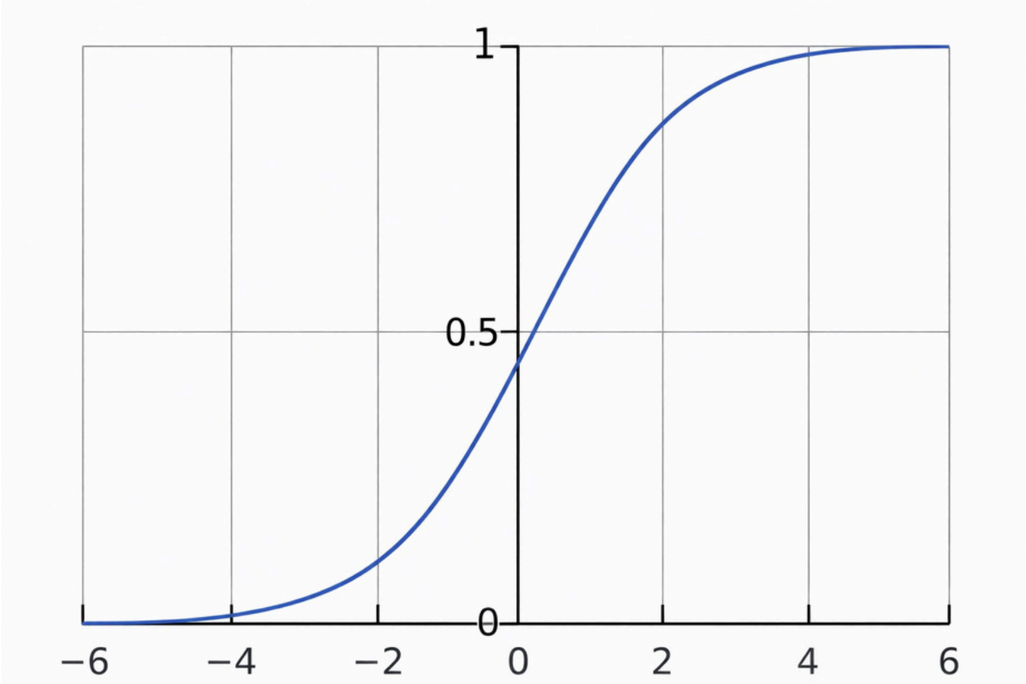
Neural Networks

- Logistic Regression
  - Sigmoid maps real values to probabilities

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

  - Suitable for gradient descent

$$\sigma'(z) = \sigma(z)\,(1 - \sigma(z))$$



$$\lim_{z \to -\infty} \sigma(z) = 0 \qquad \lim_{z \to +\infty} \sigma(z) = 1$$

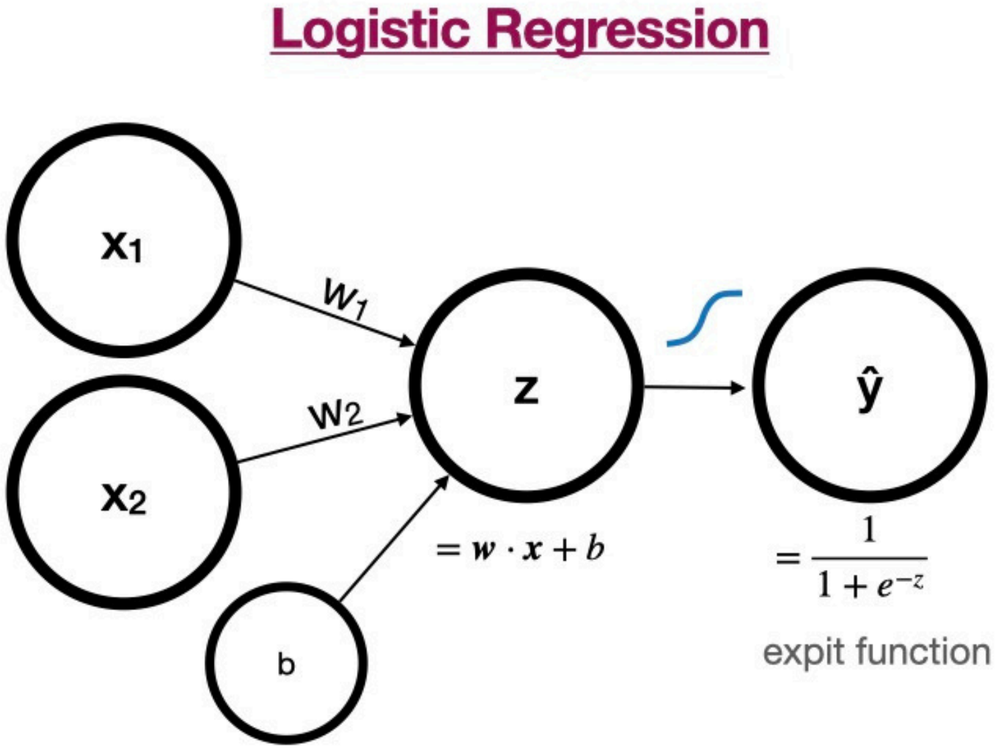
Neural Networks

- Logistic Regression
  - Logistic Regression computes:

$$y = \sigma(w^T x + b)$$

$$z = w^T x + b$$

$$y = \sigma(z)$$



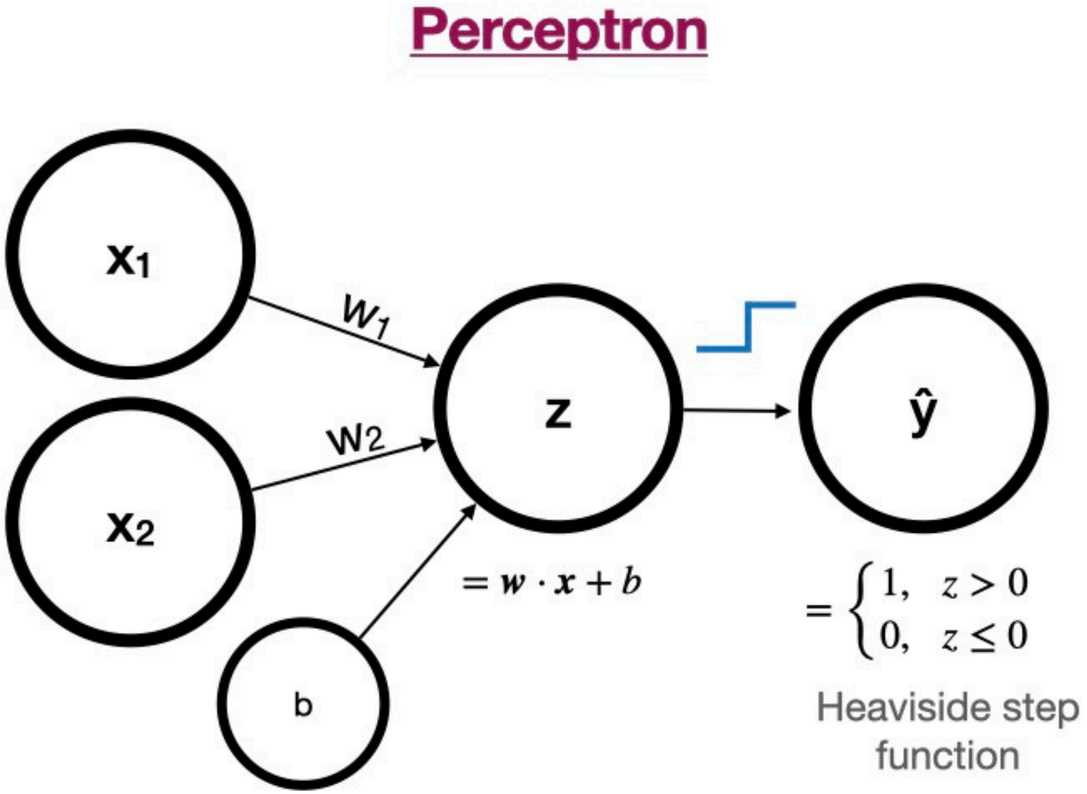
Neural Networks

Perceptron

- A perceptron is a generalized neuron where the activation function is not restricted to a sigmoid.

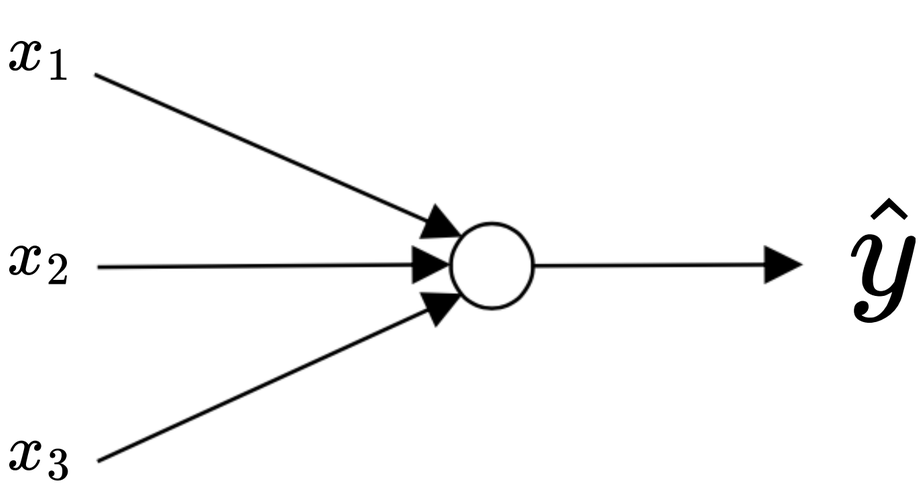
$$z = w^T x + b$$

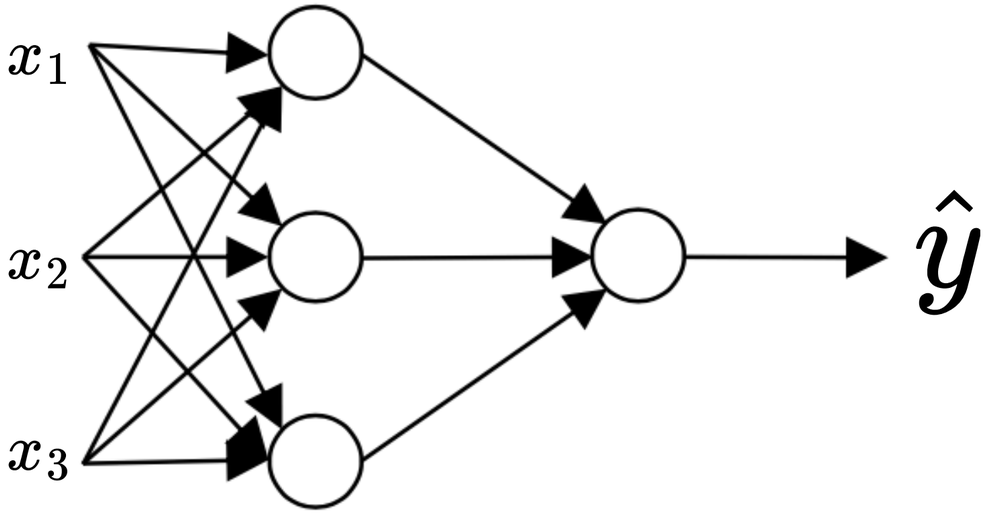
$$y = f(z)$$



Neural Networks

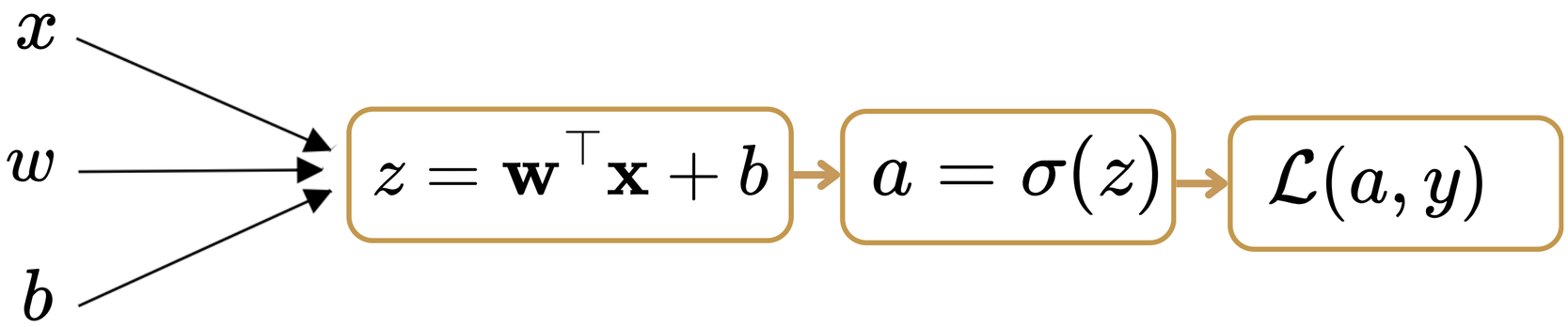
- Perceptron

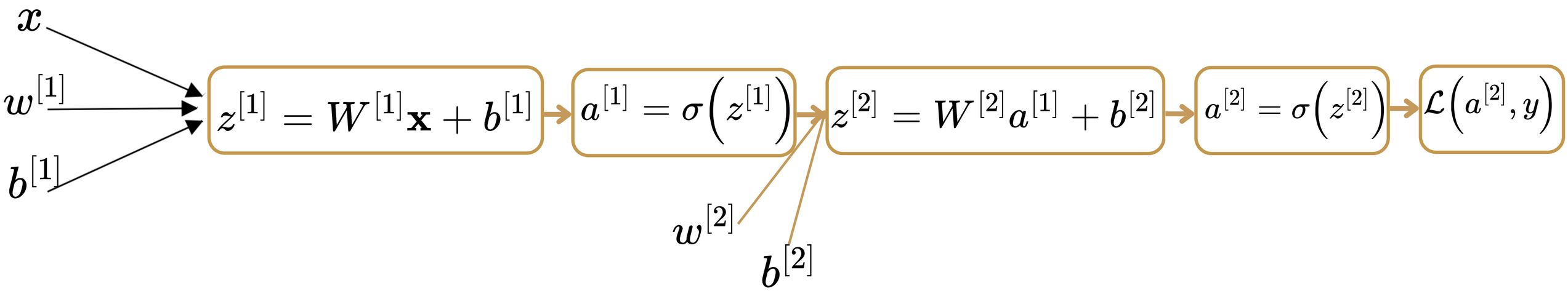




One neuron $\longrightarrow$ Many neurons

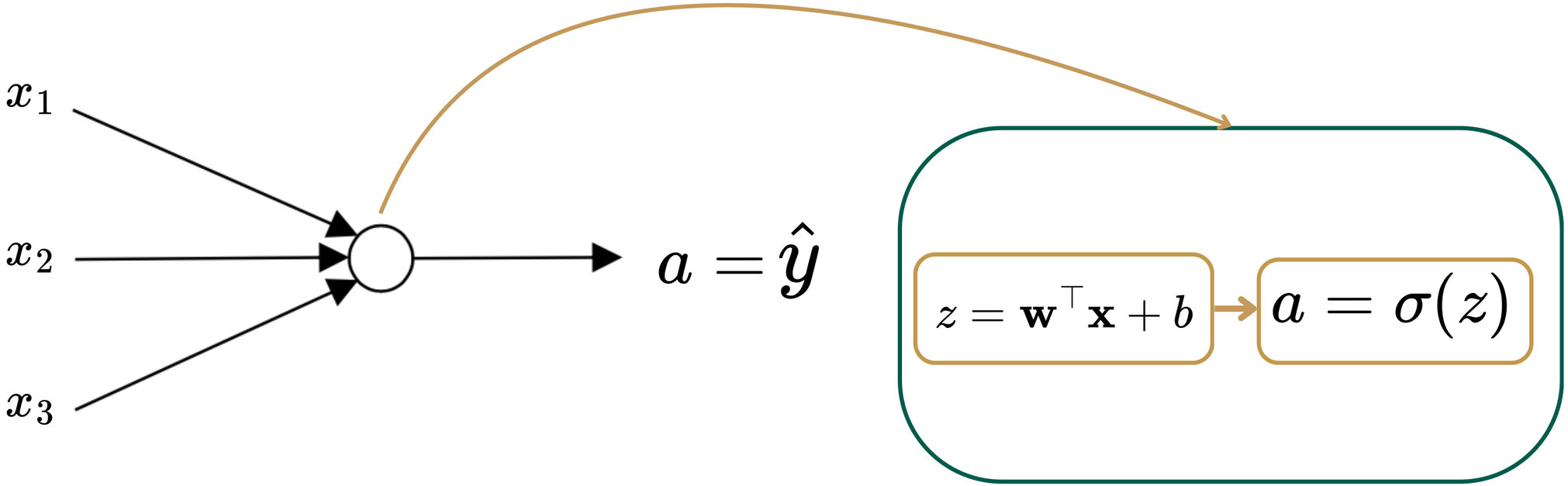
Neural Networks





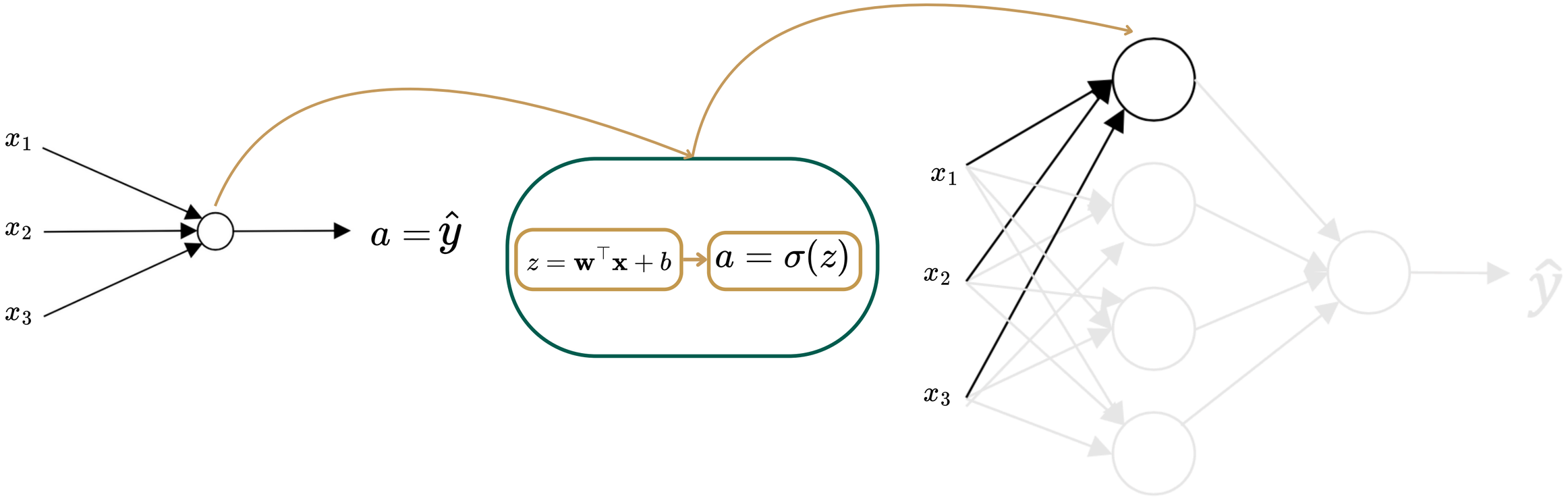
Neural Networks

Neuron Computation



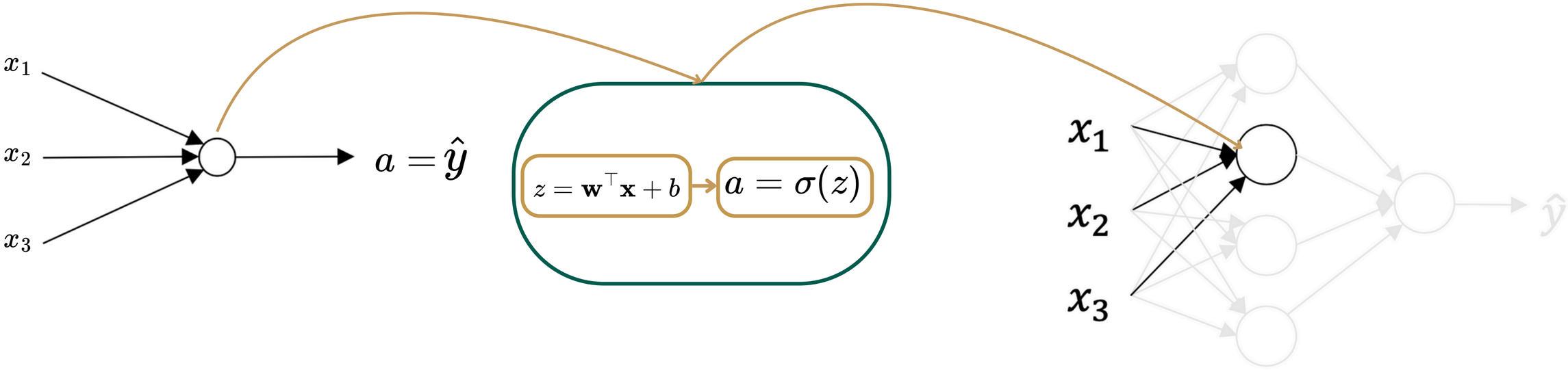
Neural Networks

Neuron Computation



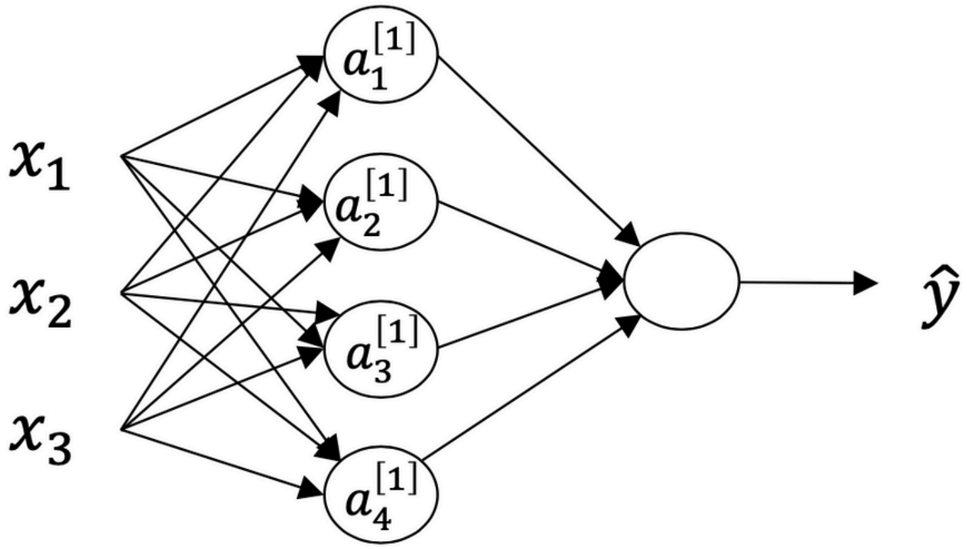
Neural Networks

Neuron Computation



Neural Networks

Neuron Computation



$$z_1^{[1]} = \mathbf{w}_1^{[1]T}\mathbf{x} + b_1^{[1]}, \qquad a_1^{[1]} = \sigma\left(z_1^{[1]}\right)$$

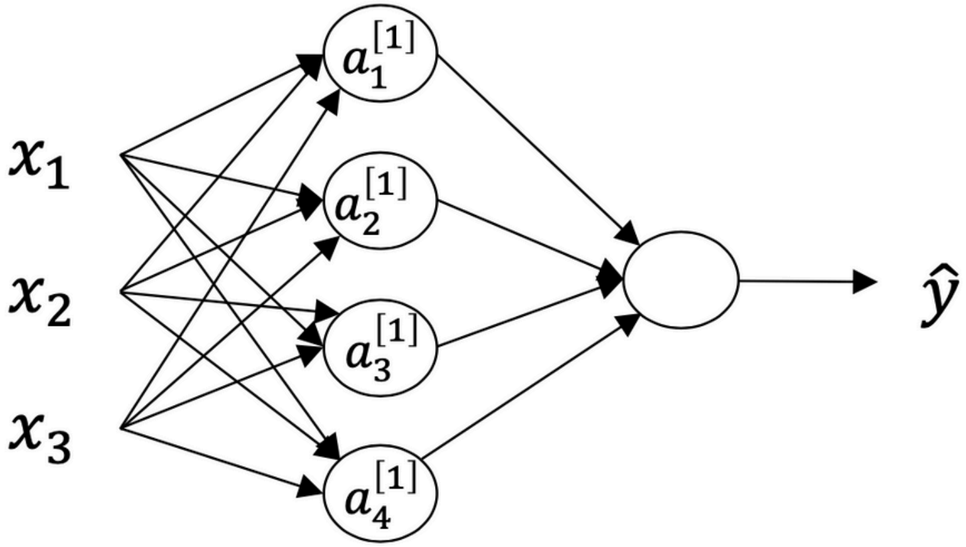
$$z_2^{[1]} = \mathbf{w}_2^{[1]T}\mathbf{x} + b_2^{[1]}, \qquad a_2^{[1]} = \sigma\left(z_2^{[1]}\right)$$

$$z_3^{[1]} = \mathbf{w}_3^{[1]T}\mathbf{x} + b_3^{[1]}, \qquad a_3^{[1]} = \sigma\left(z_3^{[1]}\right)$$

$$z_4^{[1]} = \mathbf{w}_4^{[1]T}\mathbf{x} + b_4^{[1]}, \qquad a_4^{[1]} = \sigma\left(z_4^{[1]}\right)$$

Neural Networks

Neuron Computation



Given input $\mathbf{x}$:

$$z^{[1]} = W^{[1]}\mathbf{x} + b^{[1]}$$

$$a^{[1]} = \sigma\left(z^{[1]}\right)$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = \sigma\left(z^{[2]}\right)$$

Neural Networks

Vectorization

Process many examples at once instead of looping.

Before Vectorization

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \sigma\left(z^{[1](i)}\right)$$

Vectorization

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \cdots & x^{(m)} \end{bmatrix}$$

$$A^{[1]} = \sigma\left(Z^{[1]}\right)$$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = \sigma\left(Z^{[2]}\right)$$

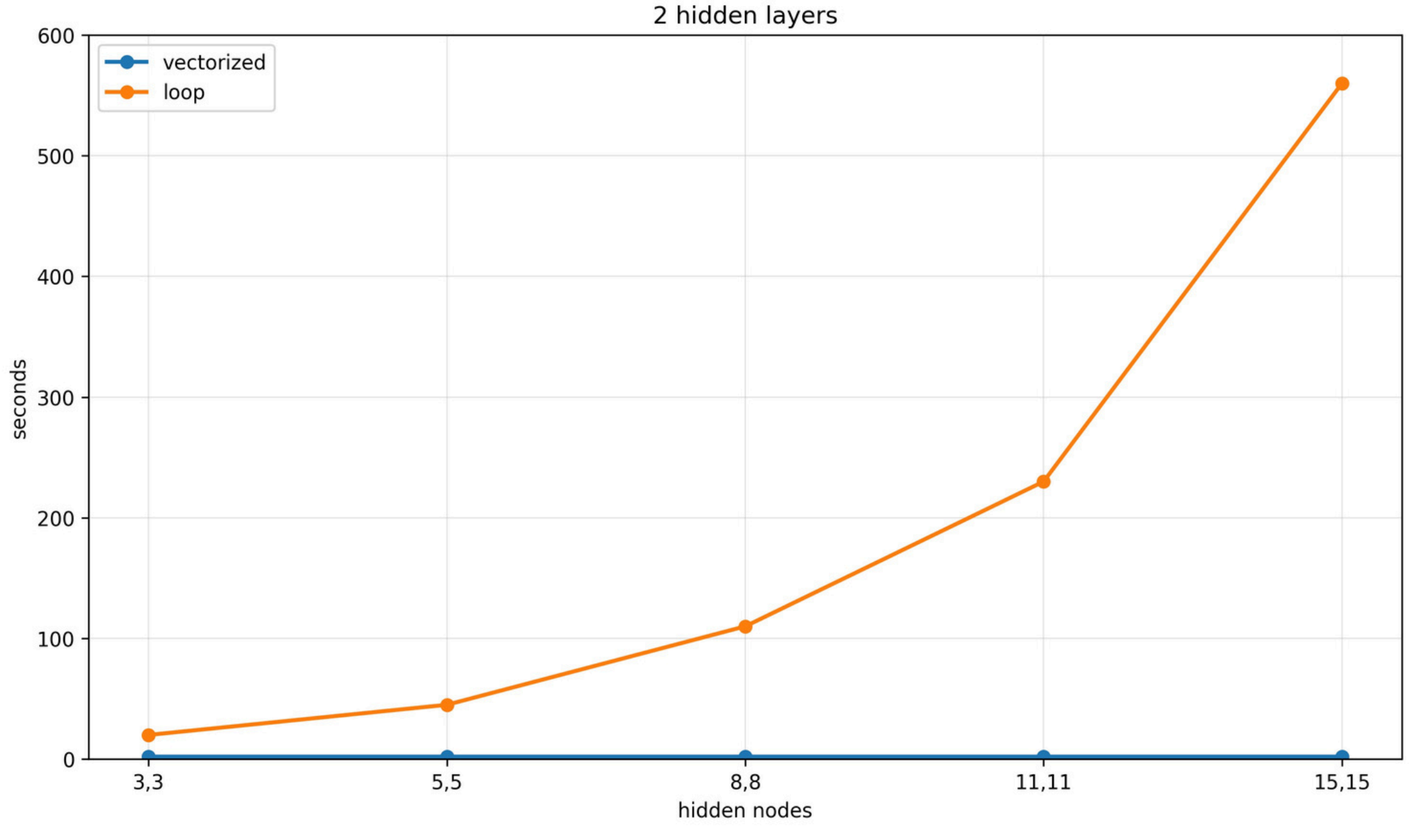
One matrix operation replaces a loop over examples.

$$X \in \mathbb{R}^{n_x \times m}, \quad Z^{[l]}, A^{[l]} \in \mathbb{R}^{n_l \times m}$$

Neural Networks

Vectorization

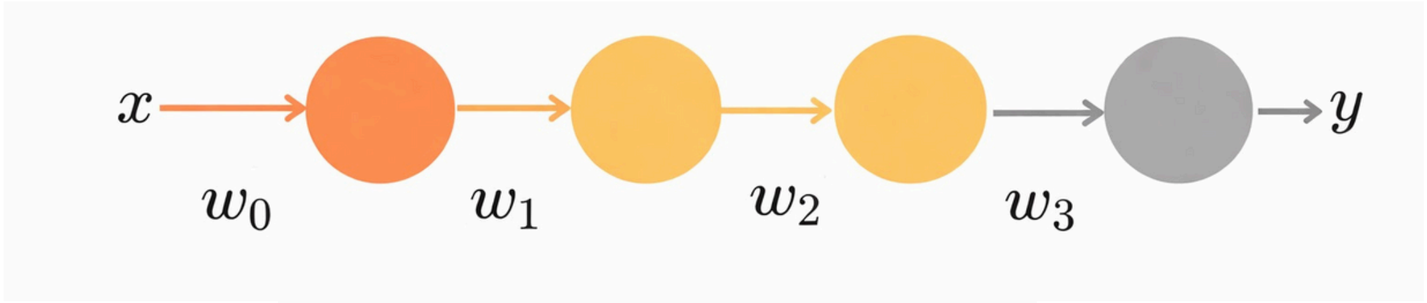
- Replacing Python loops with matrix operations dramatically improves performance

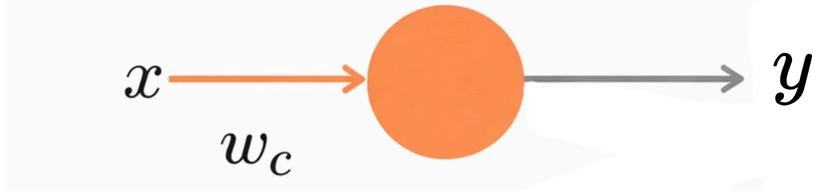


Activation Functions

Activation Functions

- Activation functions



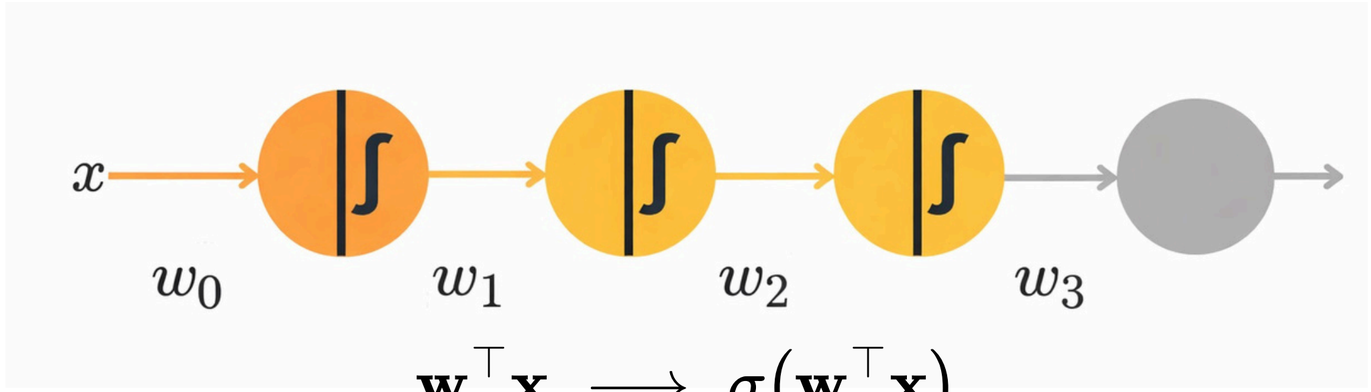


$$y = x \cdot w_0 w_1 w_2 w_3 = x \cdot w_c$$

- Without non-linear activation functions, a deep neural network reduces to a single linear model.

Activation Functions

- Activation functions

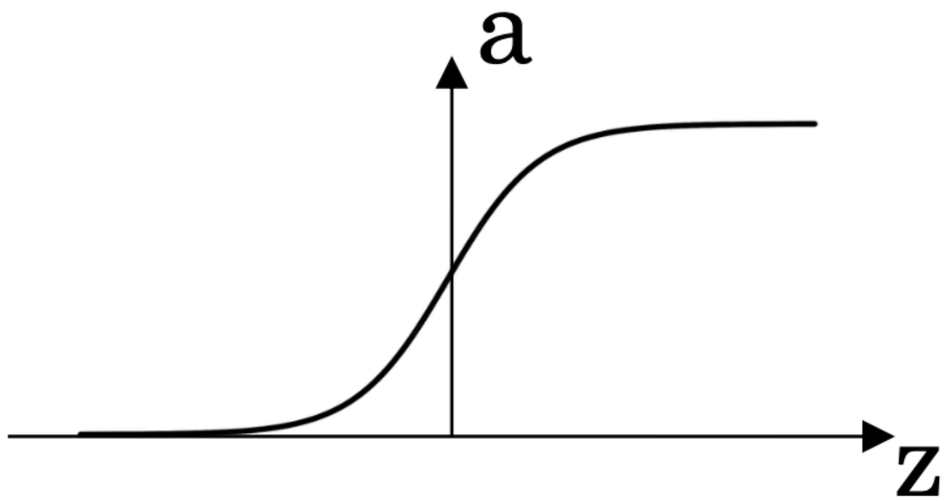


- Non-linear activations prevent layer collapse.

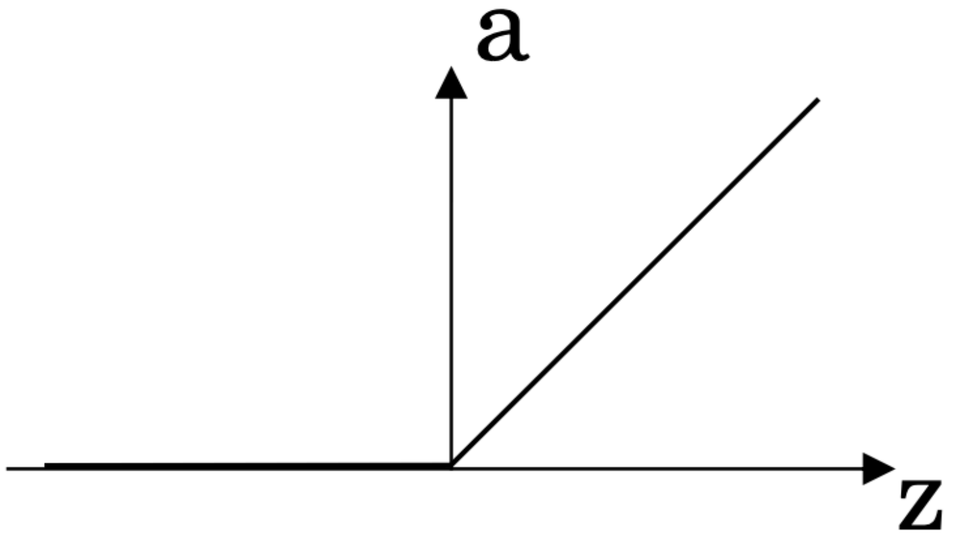
$$y = \sigma\Big(w_3\,\sigma\big(w_2\,\sigma\big(w_1\,\sigma(w_0 x)\big)\big)\Big)$$

Activation Functions

- Activation functions



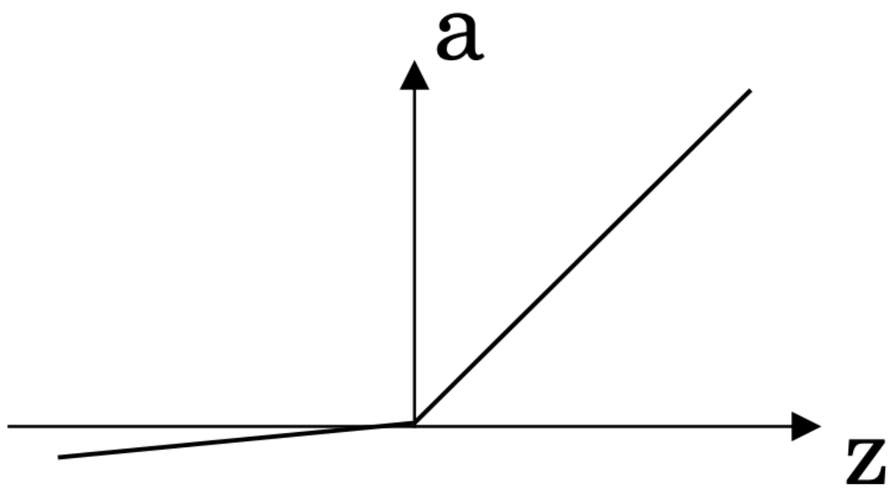
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

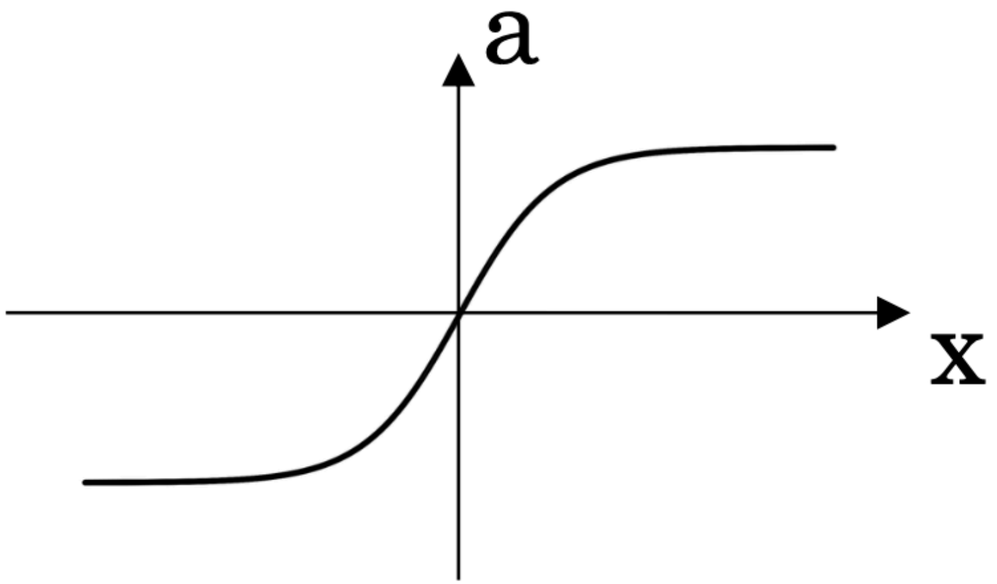


$$\mathrm{ReLU}(z) = \max(0, z)$$

Activation Functions

- Activation functions





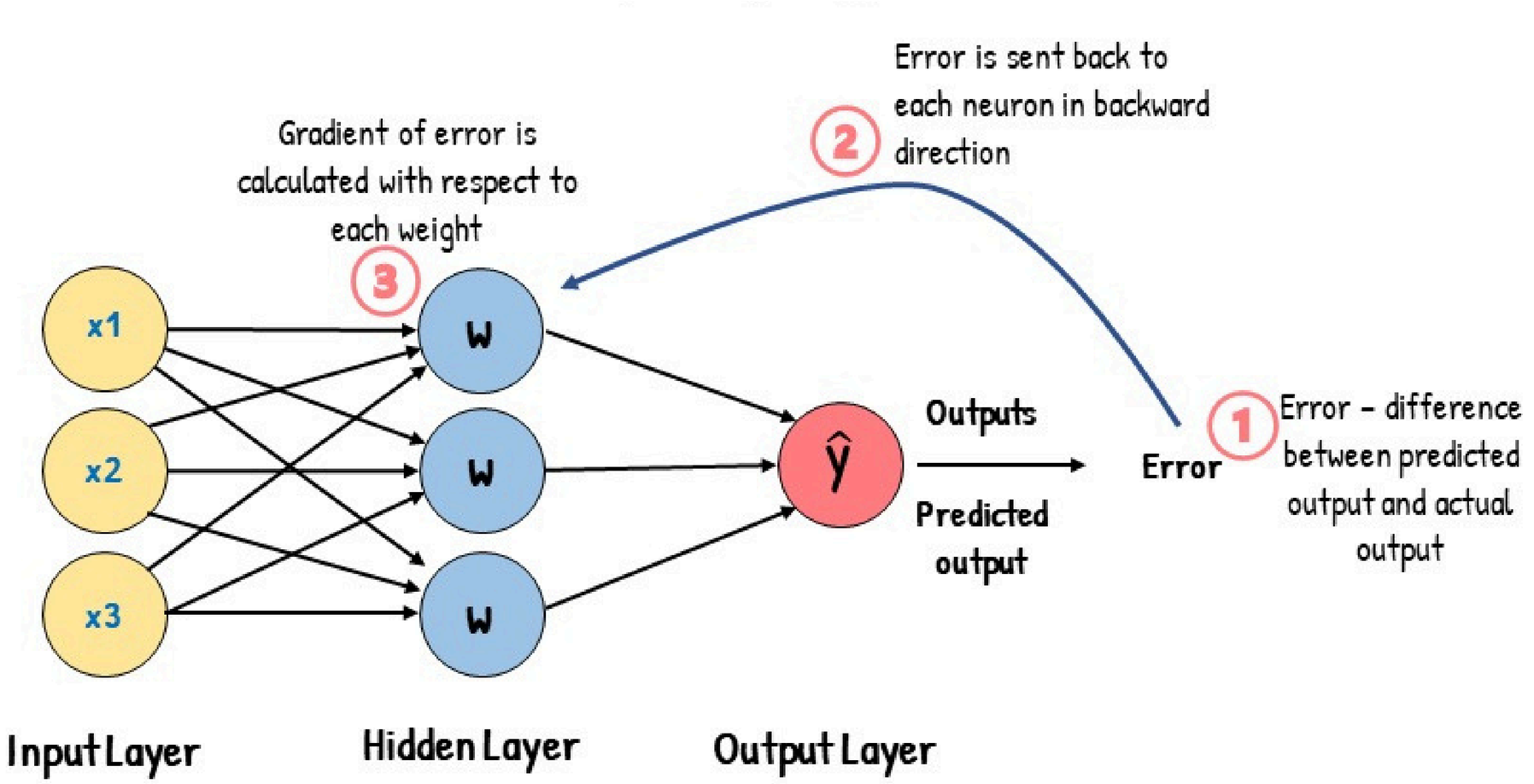
$$\text{LeakyReLU}(z) = \max(\alpha z, z)$$

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

Backpropagation

Activation Functions

- Backpropagation



Backpropagation

- The error is first calculated at the output layer
- Then, it propagated backward through the network to update weights and biases in each layer.

$$dz^{[2]} = a^{[2]} - y$$

$$dW^{[2]} = dz^{[2]}\, {a^{[1]}}^{T}$$

$$db^{[2]} = dz^{[2]}$$

$$dz^{[1]} = {W^{[2]}}^{T} dz^{[2]} \odot {g^{[1]}}'\left(z^{[1]}\right)$$

$$dW^{[1]} = dz^{[1]}\, x^{T}$$

$$db^{[1]} = dz^{[1]}$$

Backpropagation

- The vectorized version of backpropagation allows gradients to be computed efficiently over all training examples at once by averaging across the batch.

$$dZ^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[2]}_{:,i}$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} \odot g^{[1]'}(Z^{[1]})$$

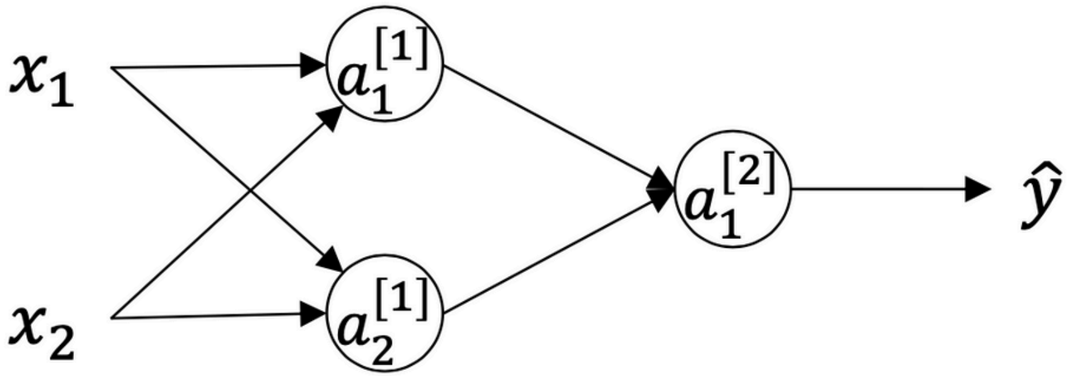
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}$$

$$db^{[1]} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[1]}_{:,i}$$

Weights Initialization

Weights Initialization

- Initialize weights to zero



$$W^{[1]} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \qquad W^{[2]} = (0 \quad 0)$$

$$b^{[2]} = 0$$

$$b^{[1]} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

All neurons start identical when weights are initialized to zero

Weights Initialization

- Initialize weights to zero

$$W^{[1]} = 0,\ b^{[1]} = 0$$

$$z^{[1]} = W^{[1]}x + b^{[1]} = 0$$

$$a^{[1]} = g(z^{[1]}) = g(0)$$

$$dz^{[1]} = W^{[2]T}dz^{[2]} \odot g'(z^{[1]}) = W^{[2]T}dz^{[2]} \odot g'(0)$$

$$dW^{[1]} = dz^{[1]}x^T$$

$$dW_1^{[1]} = dW_2^{[1]} = \cdots$$

$$W_1^{[1]} = W_2^{[1]} = \cdots \quad \forall \text{ iterations}$$

Weights Initialization

- Initialize weights to zero

$$W^{[1]} = 0,\ b^{[1]} = 0$$

$$z^{[1]} = W^{[1]}x + b^{[1]} = 0$$

$$a^{[1]} = g(z^{[1]}) = g(0)$$

$$dz^{[1]} = W^{[2]T}dz^{[2]} \odot g'(z^{[1]}) = W^{[2]T}dz^{[2]} \odot g'(0)$$

$$dW^{[1]} = dz^{[1]}x^T$$

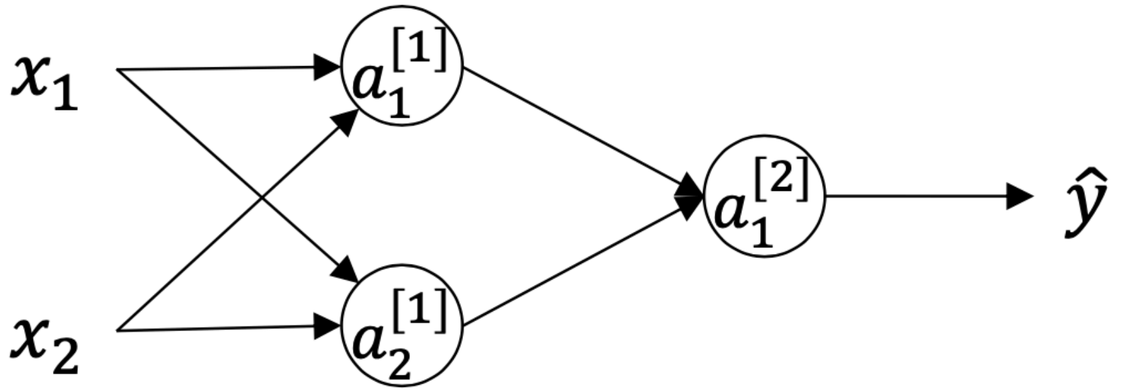
$$dW_1^{[1]} = dW_2^{[1]} = \cdots$$

$$W_1^{[1]} = W_2^{[1]} = \cdots \quad \forall \text{ iterations}$$

Zero initialization $\Rightarrow$ no symmetry breaking

Weights Initialization

- Initialize weights randomly



$$W^{[1]} = \begin{pmatrix} \epsilon_1 & \epsilon_2 \\ \epsilon_3 & \epsilon_4 \end{pmatrix} \qquad W^{[2]} = \begin{pmatrix} \epsilon_5 & \epsilon_6 \end{pmatrix}$$

$$b^{[1]} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \qquad b^{[2]} = 0$$

ϵ are small random numbers

Weights Initialization

- Initialize weights randomly

$$W^{[1]} \sim \mathcal{N}(0, \epsilon), \quad b^{[1]} = 0$$

$$z^{[1]} = W^{[1]}x + b^{[1]} \neq 0$$

$$a^{[1]} = g(z^{[1]}) \quad \text{(different for each neuron)}$$

Weights Initialization

- Initialize weights randomly

Backpropagation effect

$$dz^{[1]} = W^{[2]T} dz^{[2]} \odot g'(z^{[1]})$$

$$dW^{[1]} = dz^{[1]} x^T$$

$$dW_1^{[1]} \neq dW_2^{[1]} \neq \cdots$$

$$W_1^{[1]} \neq W_2^{[1]} \neq \cdots \quad \forall \text{ iterations}$$

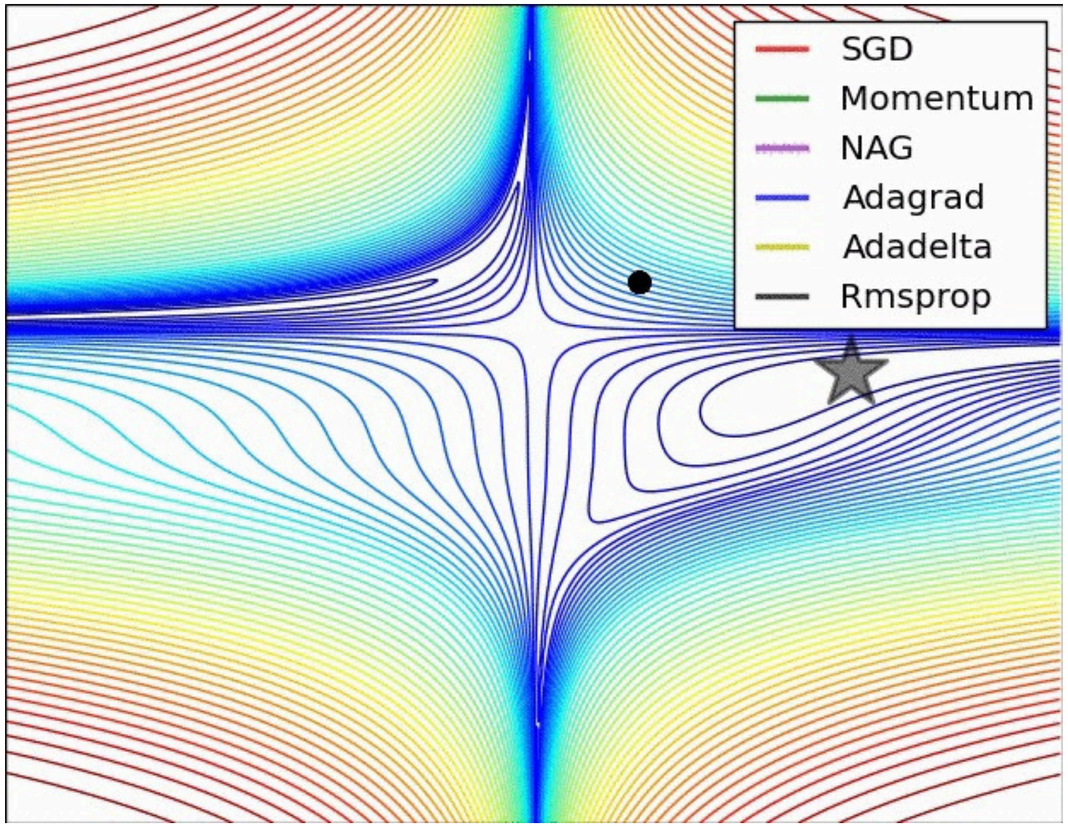
Random initialization $\Rightarrow$ symmetry breaking

Optimization

Optimization

- Optimizers are algorithms that update a model's parameters in order to minimize the loss function during training.

  - Optimizers are a core component of deep learning models.
  - Different optimizers lead to different convergence behaviors and performance.

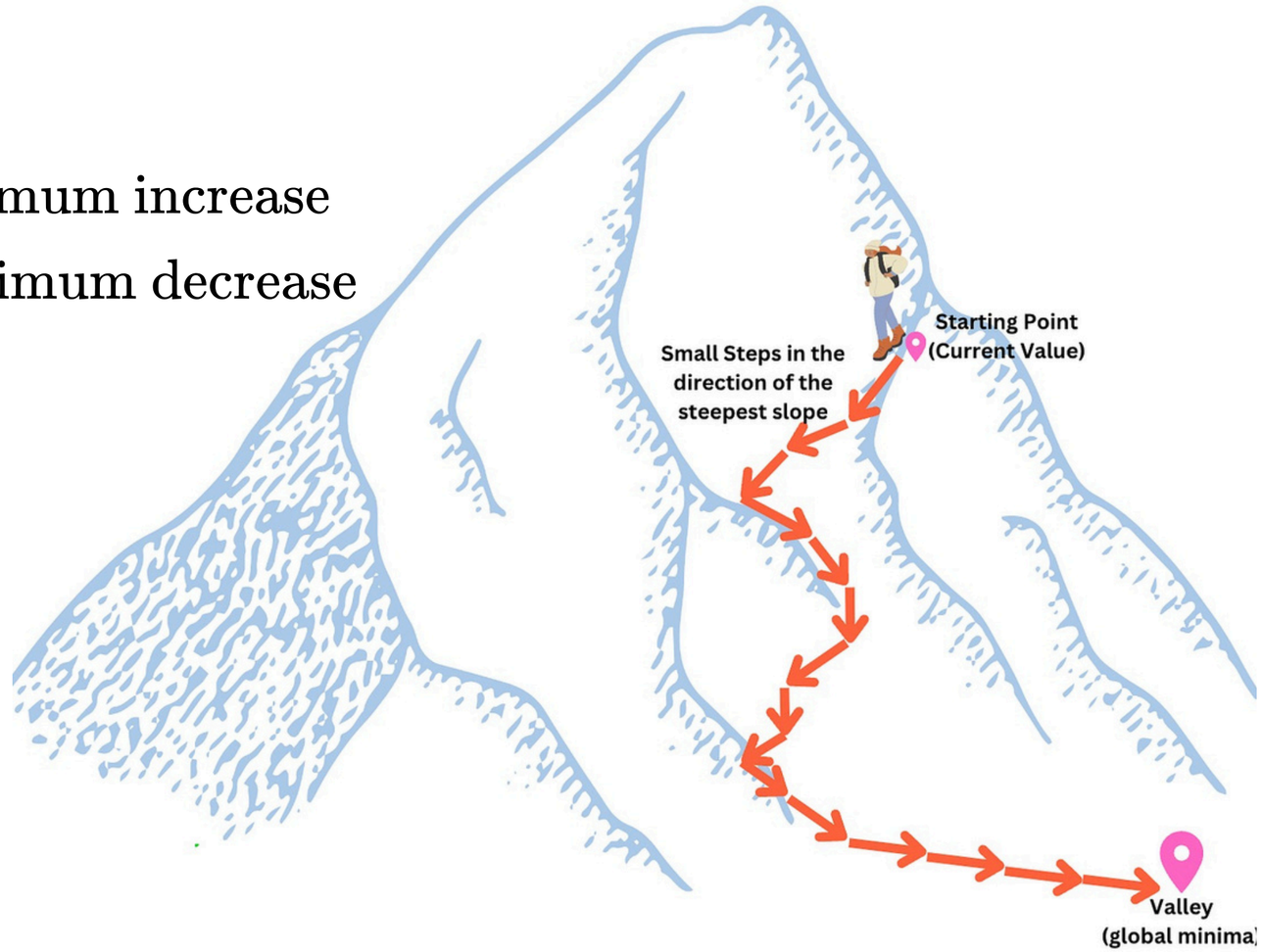


Optimization

- Direction of Maximum Increase and Decrease

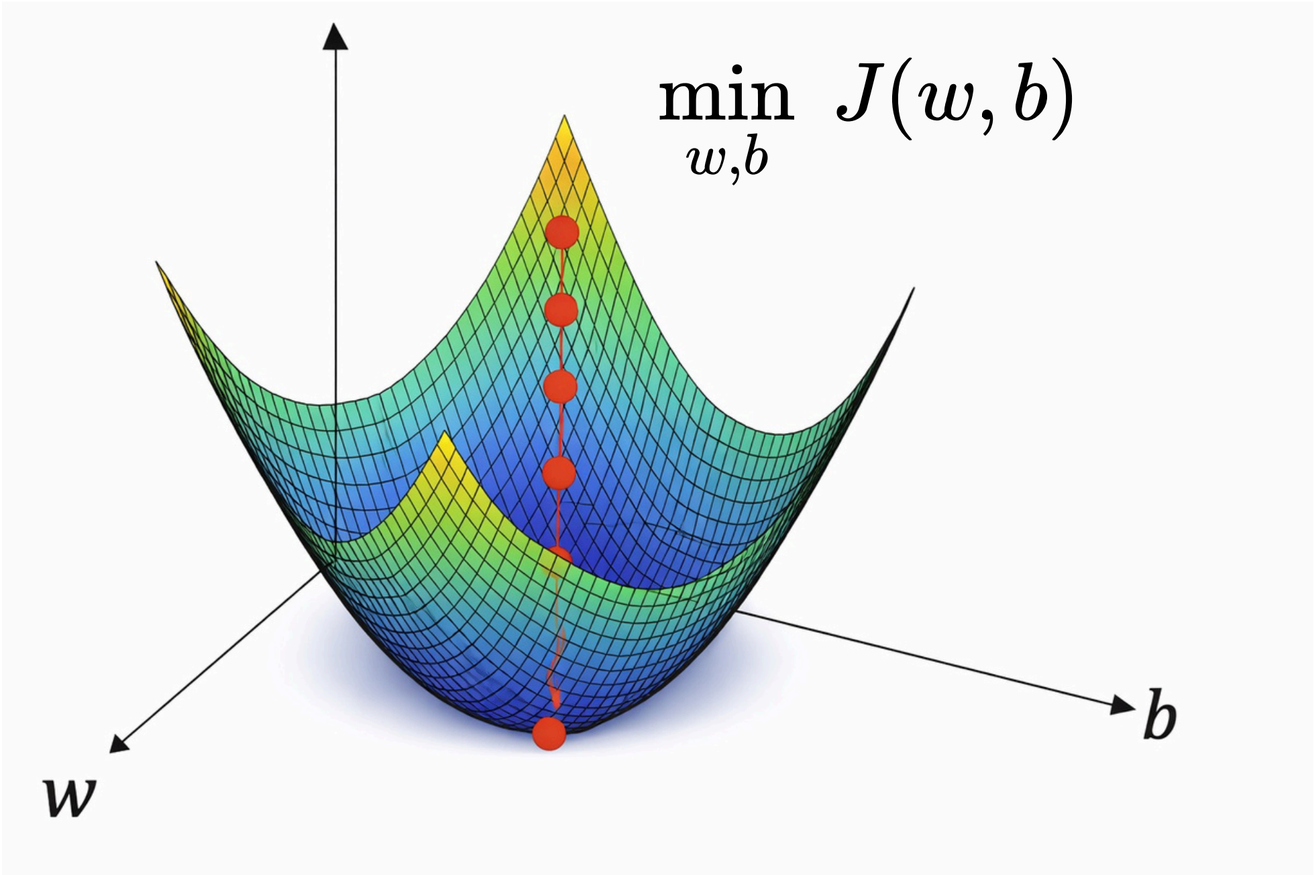
$\nabla f(x)$: direction of maximum increase

$-\nabla f(x)$: direction of maximum decrease



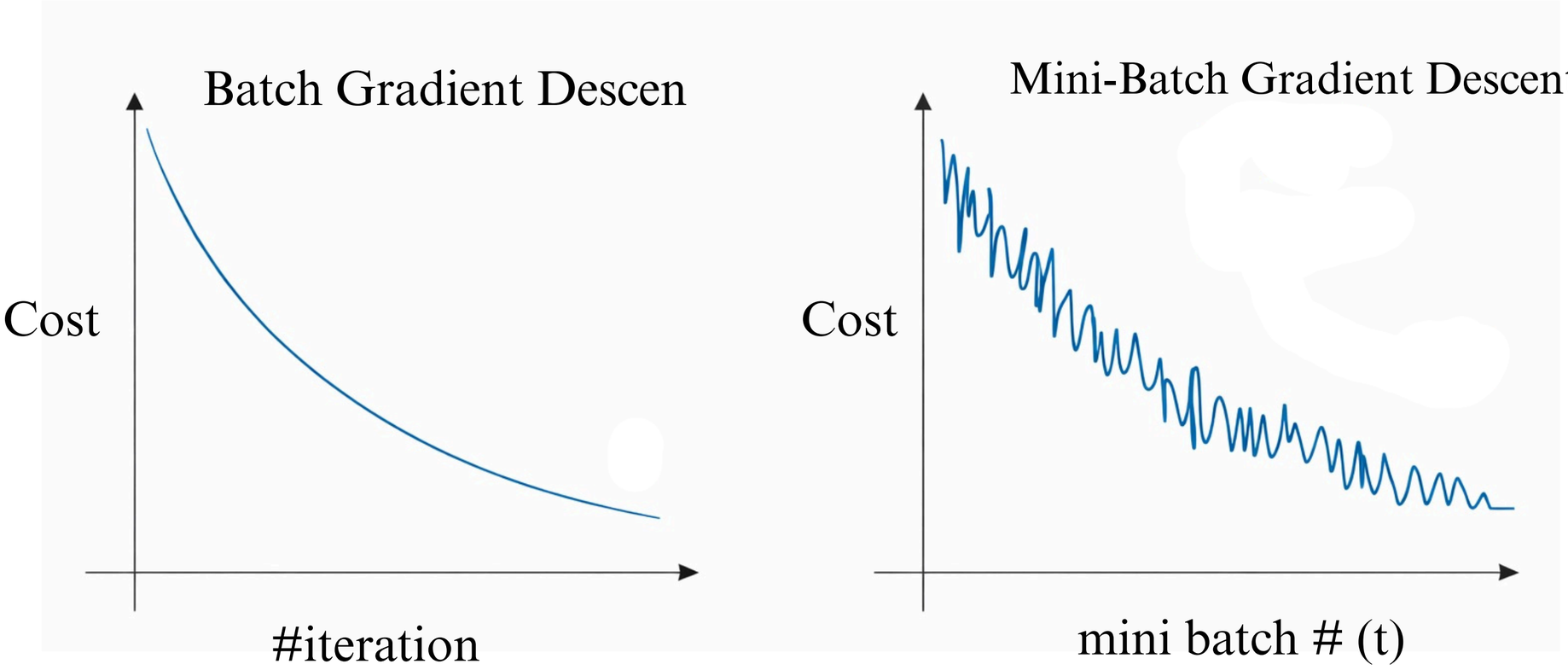
Optimization

- Gradient Descent
  - Moving in the direction of steepest loss decrease



Optimization

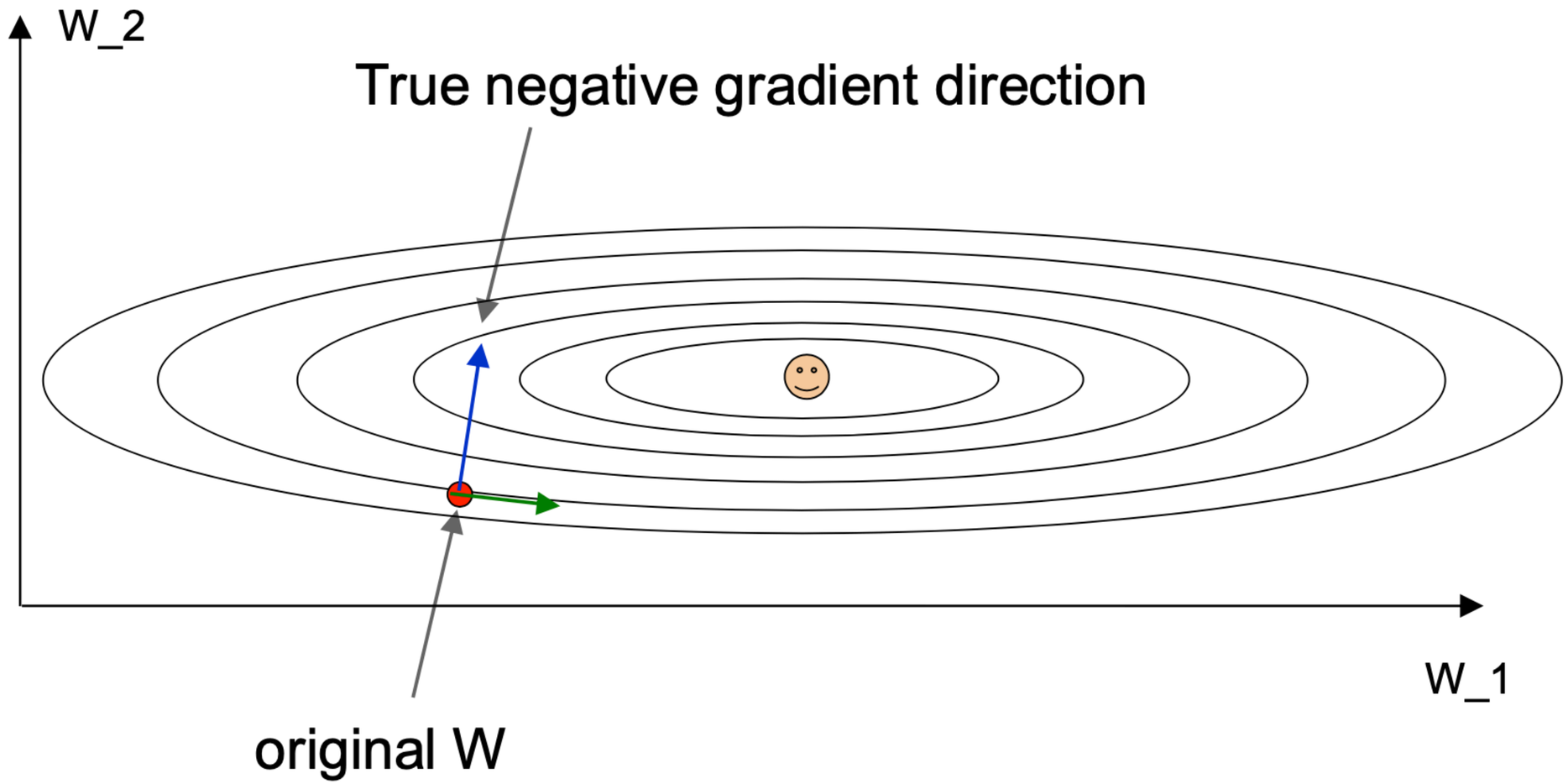
- Mini-Batch Gradient Descent
  - Instead of using all training examples or just one, we update parameters using a small batch of samples.



Common mini-batch sizes are 32/64/128 examples

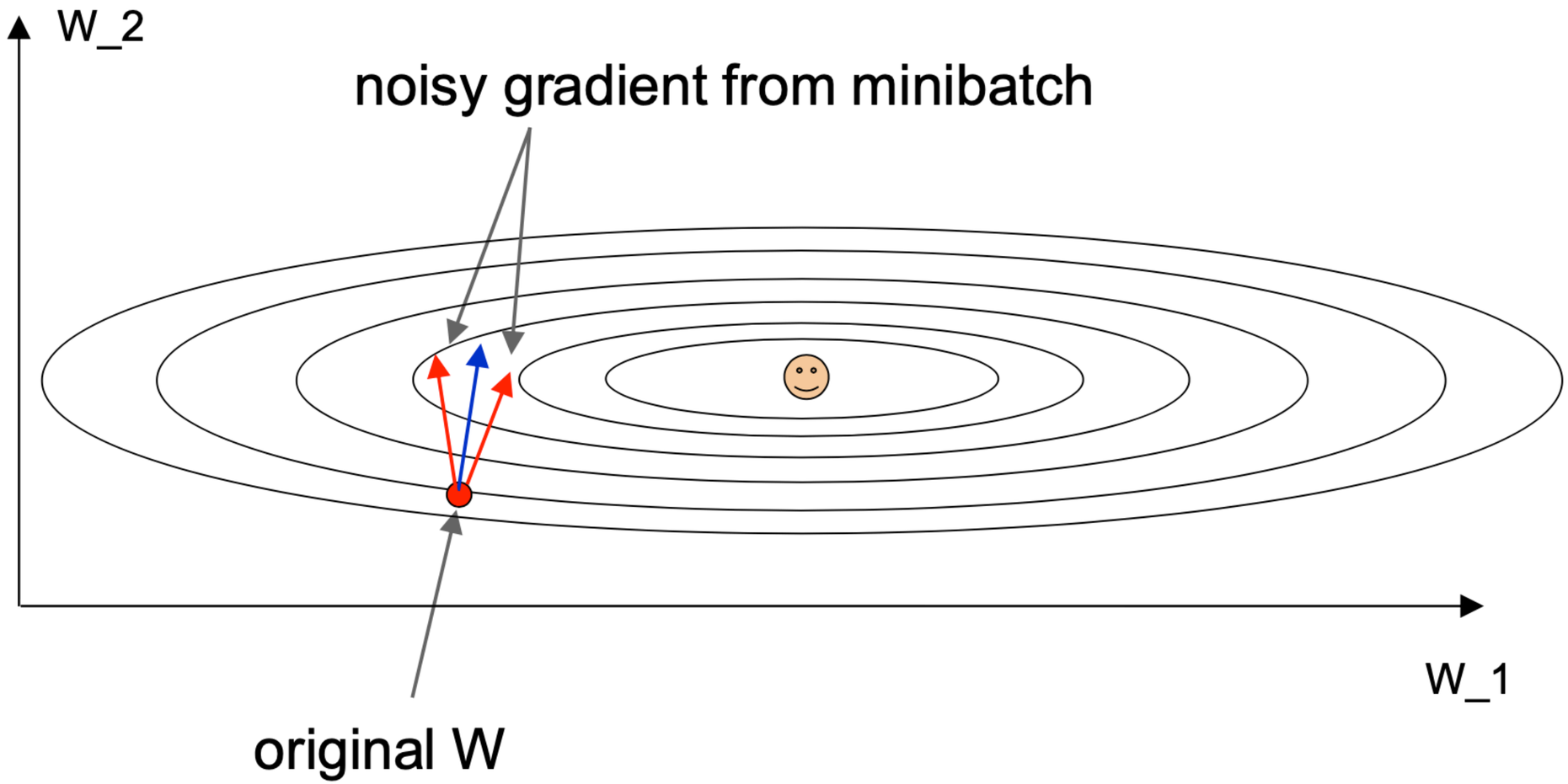
Optimization

- Minibatch updates



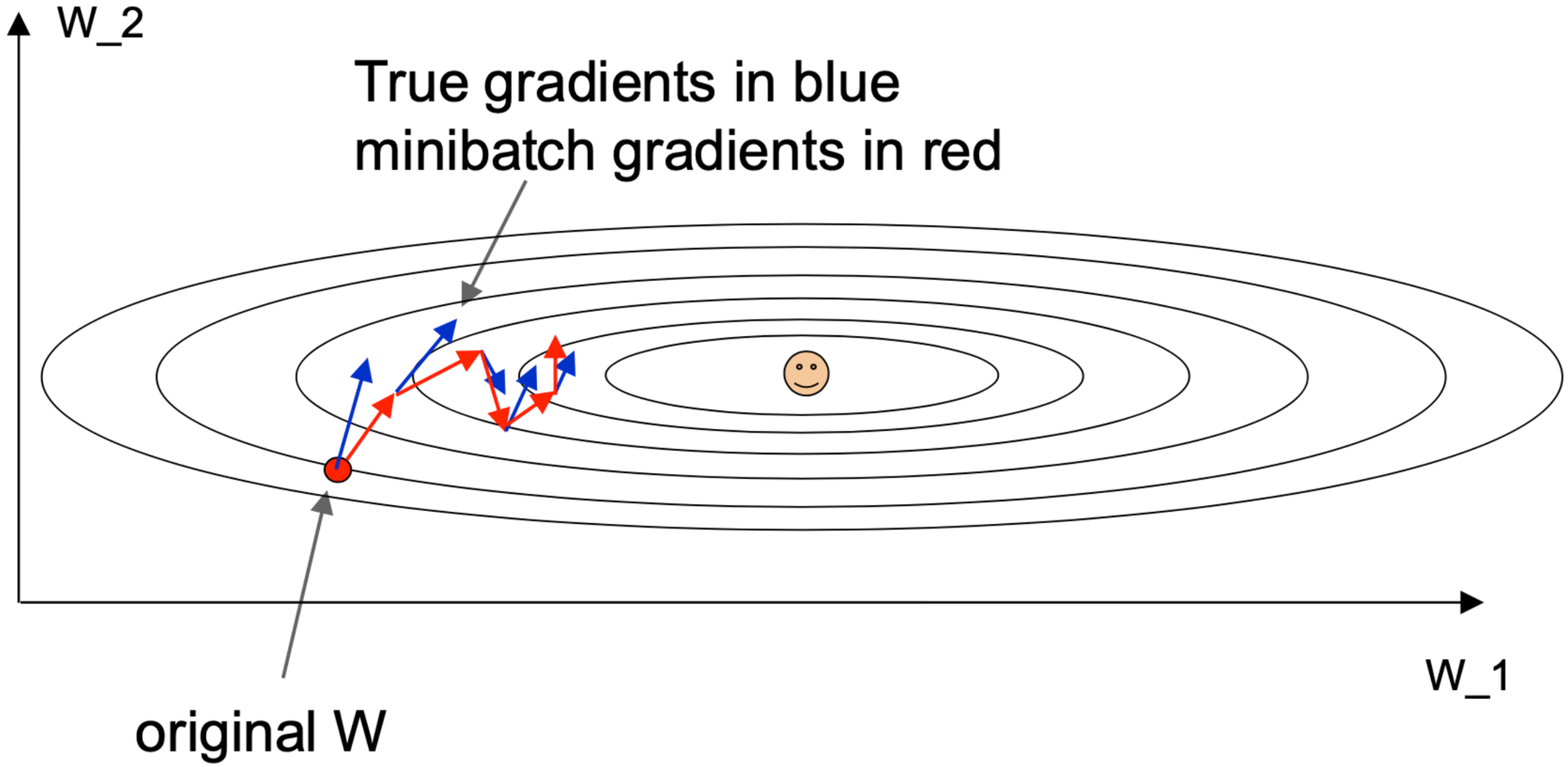
Optimization

- Minibatch updates



Optimization

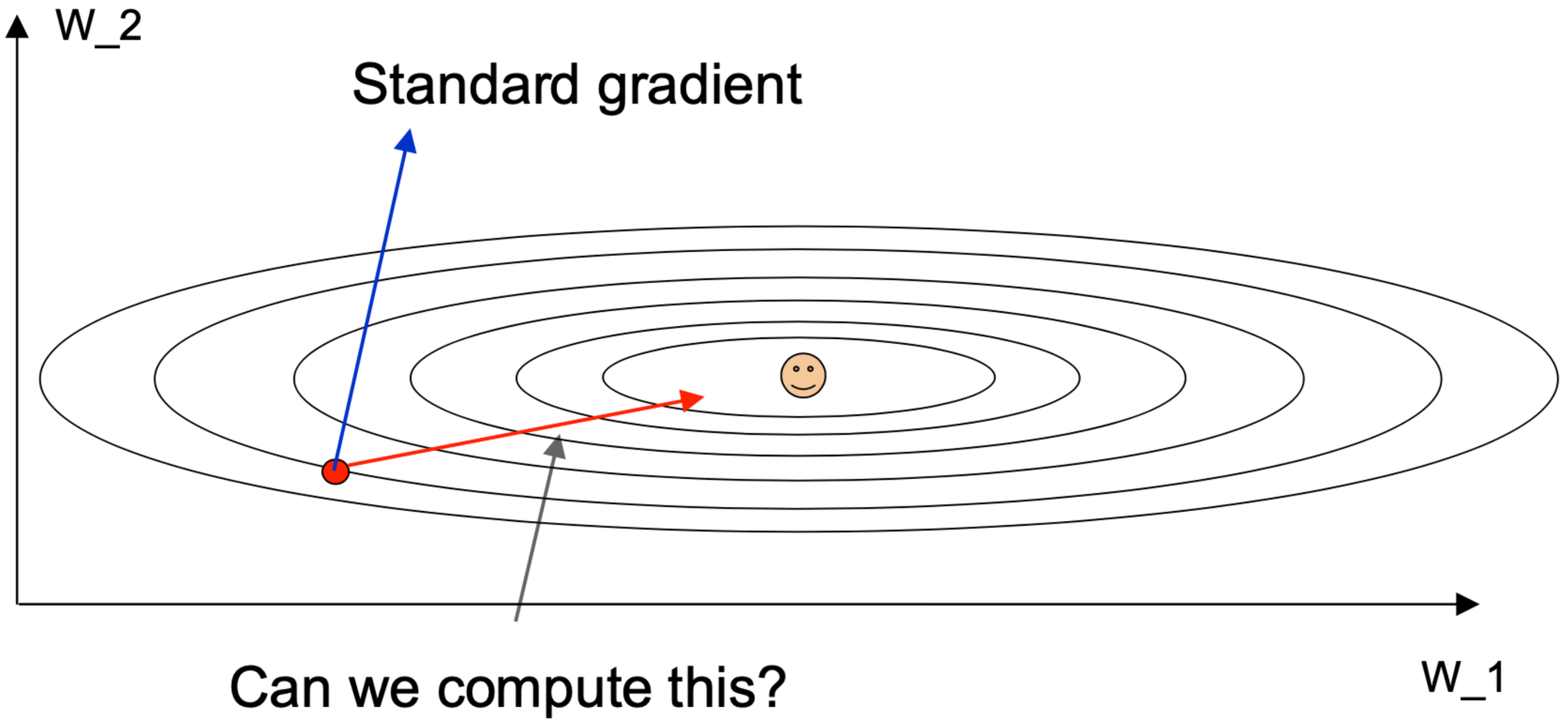
- Minibatch updates



Gradients are noisy but still make good progress on average

Optimization

- Minibatch updates



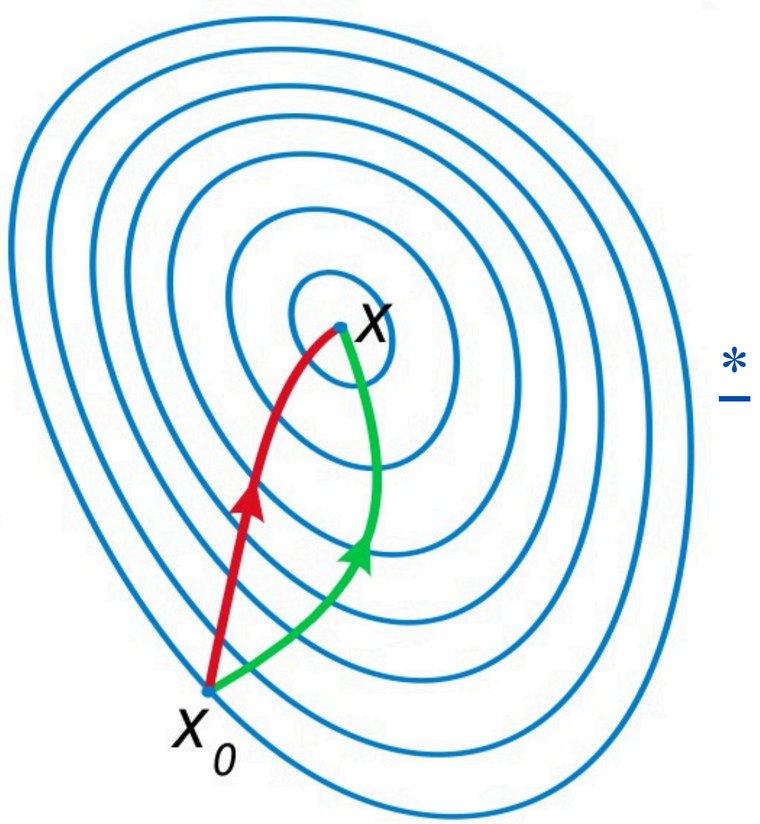
Optimization

- Newton's Method for Optimization

$$x_{n+1} = x_n - H_f(x_n)^{-1} \nabla f(x_n)$$

$\nabla f(x_n)$: gradient of f at x_n

$H_f(x_n) = \nabla^2 f(x_n)$: Hessian matrix (second-order derivatives)

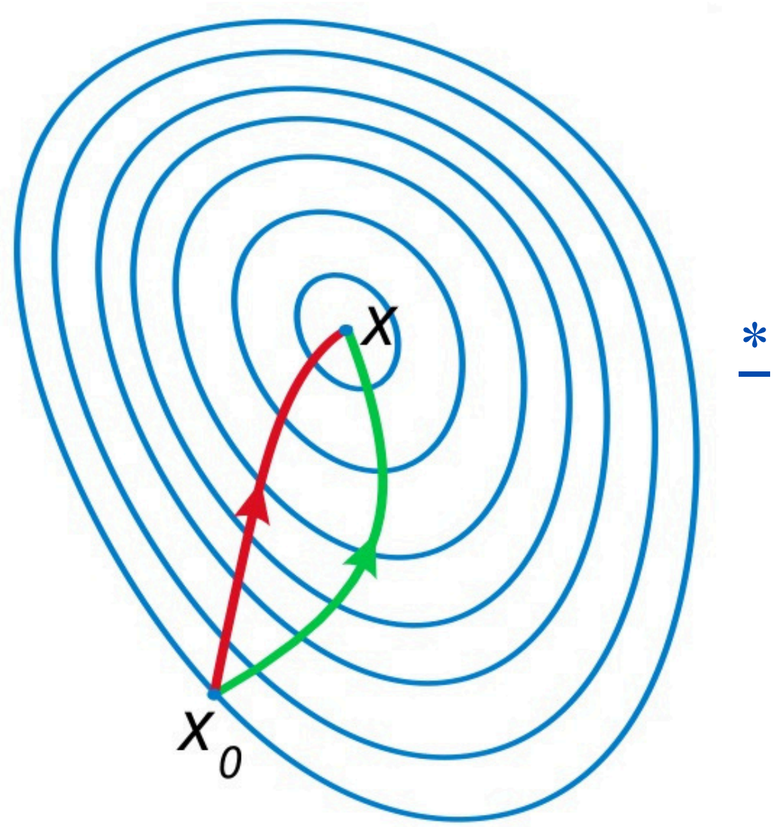


A comparison of gradient descent(green) and Newton's method (red) for minimizing a function

Optimization

Newton's Method for Optimization

- Uses second-order curvature information
- Quadratic convergence near the optimum
- No learning rate required



Why Rarely Used in Deep Learning?

Optimization

- Newton's Method for Optimization

Why Rarely Used in Deep Learning?

- Hessian is $O(d^2)$ to store
- Matrix inversion is $O(d^3)$
- Unstable for non-convex loss landscapes
- Not scalable to millions of parameters

Deep Learning trades fast convergence for scalability.

Optimization

- Momentum Gradient Descent

$$v_{t+1} = \beta v_t + \nabla J(\theta_t)$$

Momentum Update

$$v_{t+1} = \beta v_t + \nabla J(\theta_t)$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

α : learning rate

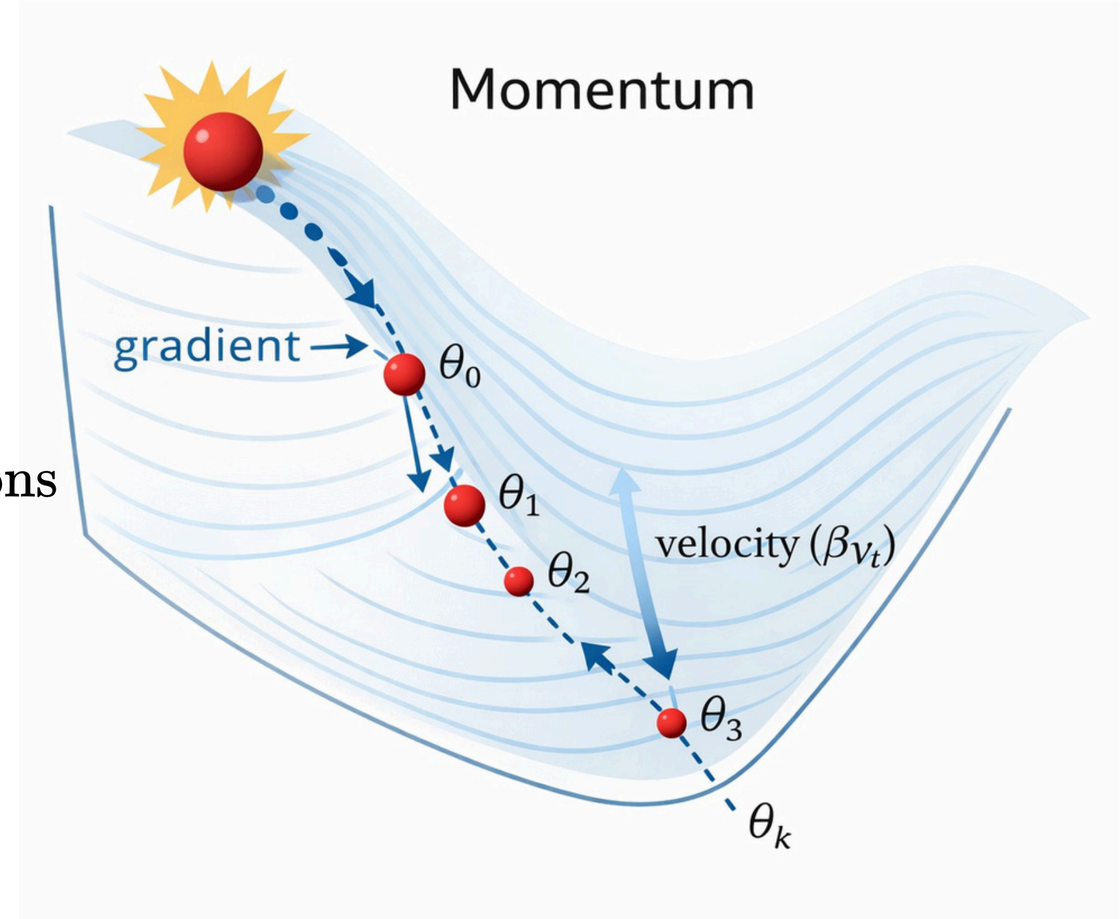
$\beta \in [0, 1)$: momentum coefficient (typically 0.9)

v_t : velocity (exponentially averaged gradient)

Optimization

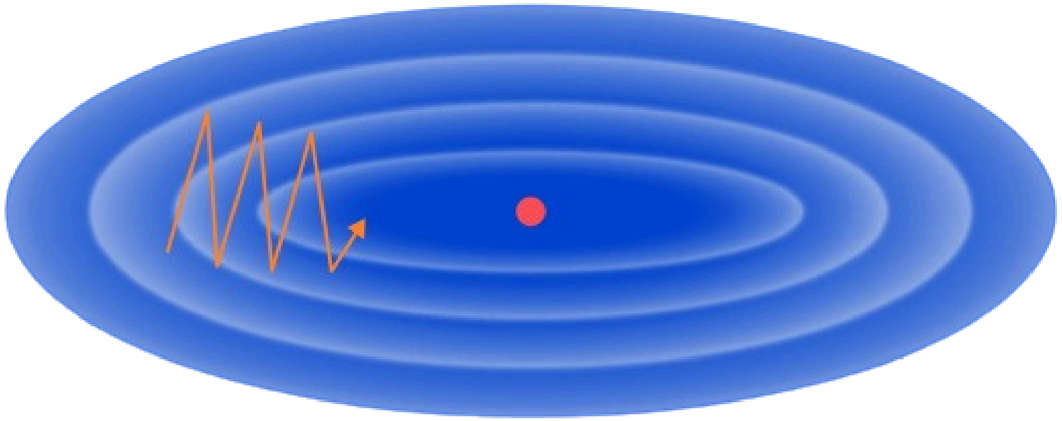
Momentum Gradient Descent

- Accumulates gradients over time
- Dampens oscillations in steep directions
- Accelerates convergence along consistent directions

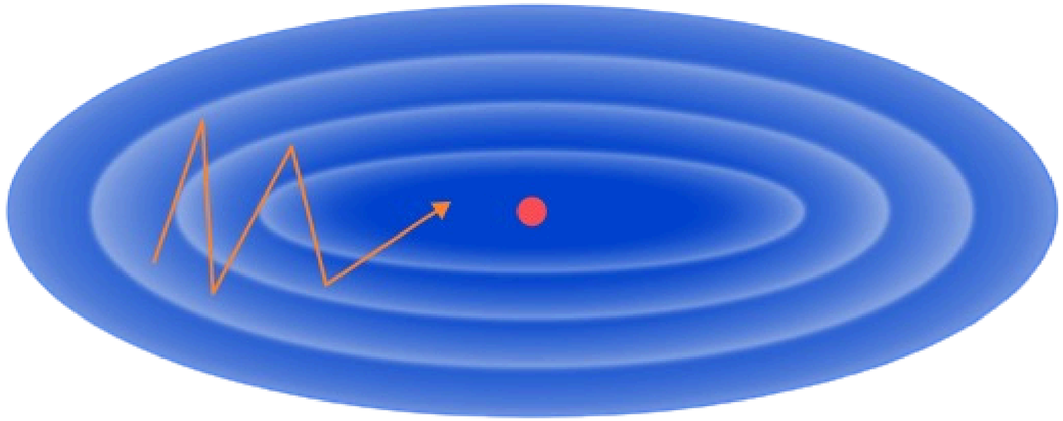


Optimization

- SGD without Momentum



- SGD with Momentum



Optimization

- AdaGrad update

$$g_t = \nabla_\theta J(\theta_t)$$

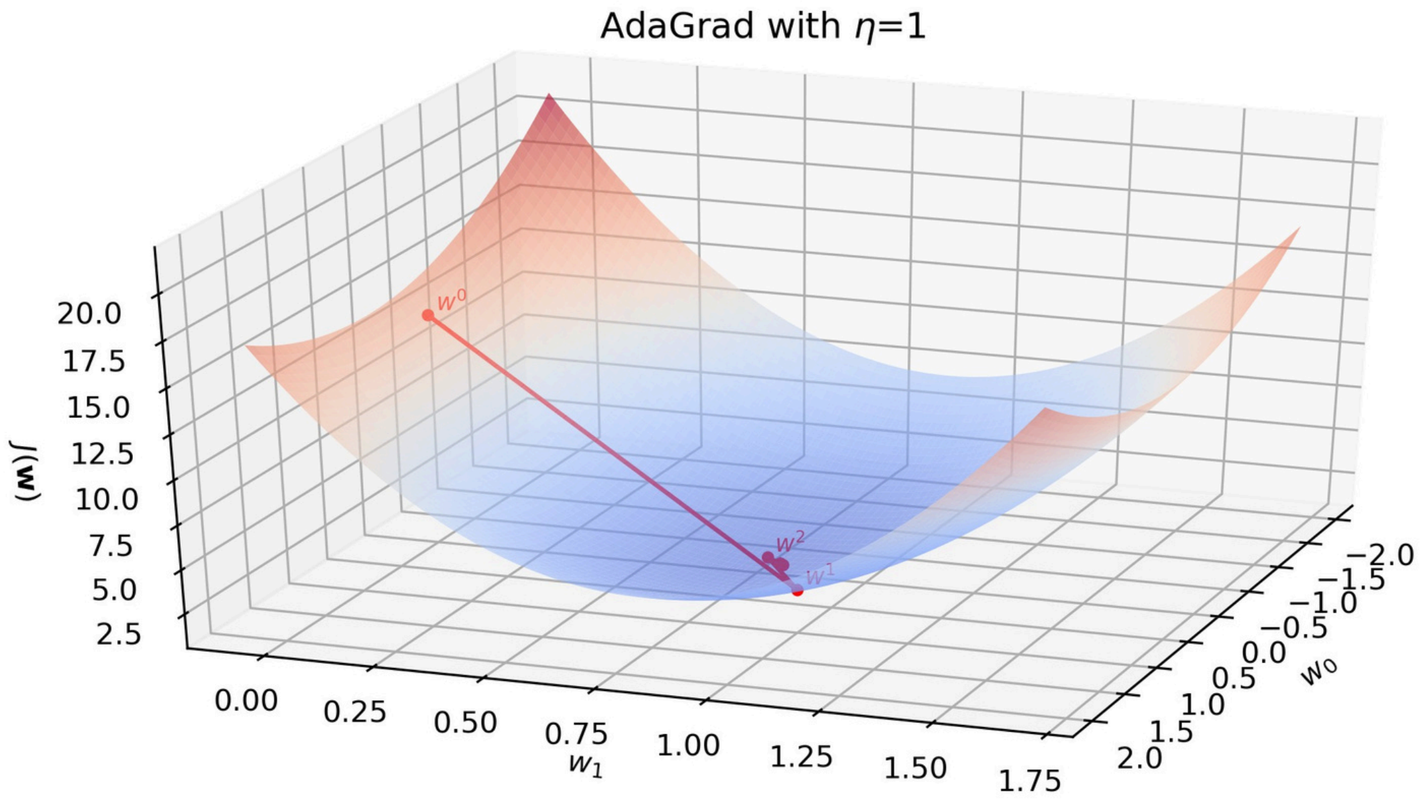
$$G_t = G_{t-1} + g_t \odot g_t$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t} + \varepsilon} \odot g_t$$

Scalar (per-parameter i) form:

$$G_{t,i} = G_{t-1,i} + g_{t,i}^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,i}} + \varepsilon} g_{t,i}$$



Optimization

RMSProp

$$g_t = \nabla_\theta J(\theta_t)$$

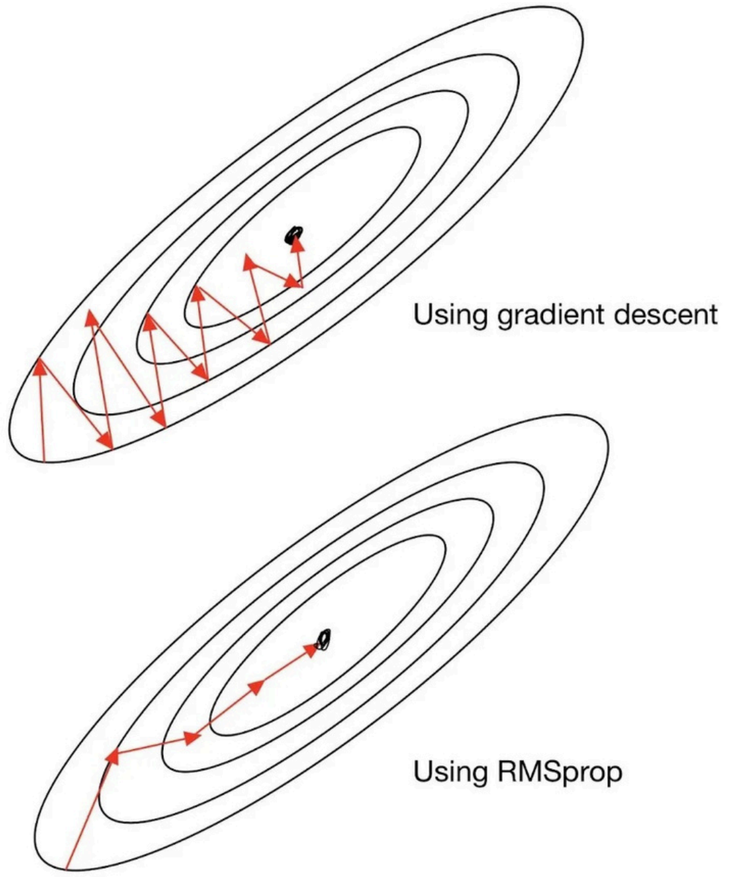
$$E[g^2]_t = \rho\, E[g^2]_{t-1} + (1-\rho)\, g_t \odot g_t$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{E[g^2]_t + \varepsilon}} \odot g_t$$

α : learning rate

$\rho \in [0, 1)$: decay rate (typically 0.9)

ε : small constant for numerical stability



RMSProp uses an exponentially decaying average of squared gradients.

Optimization

Adam

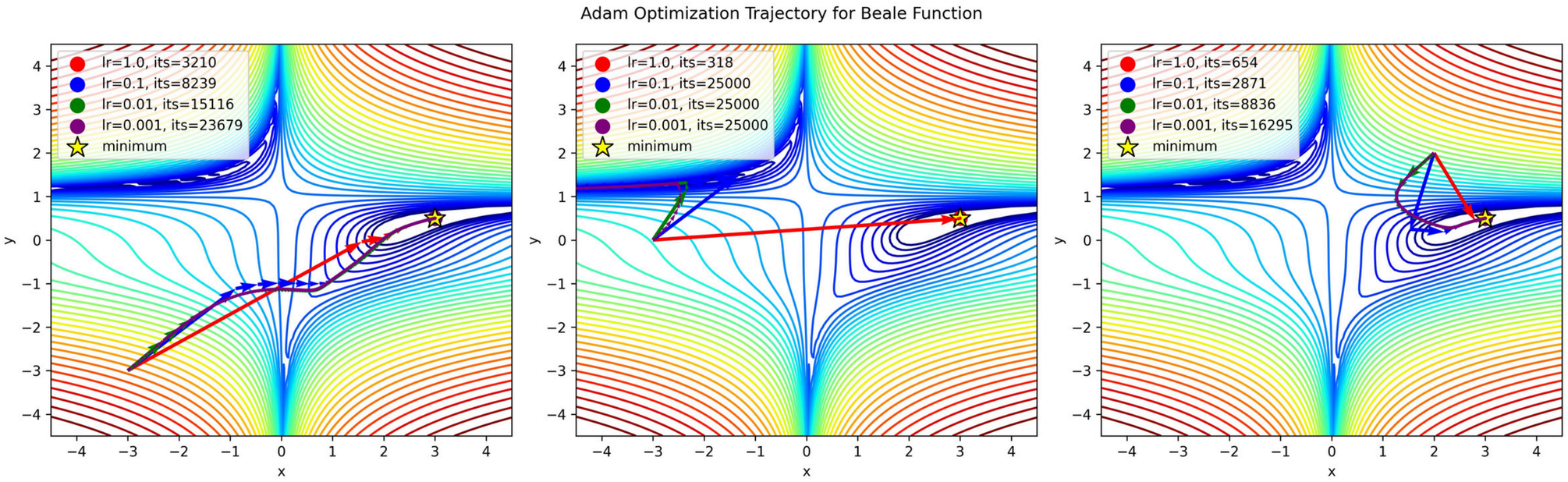
$$g_t = \nabla_\theta J(\theta_t)$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

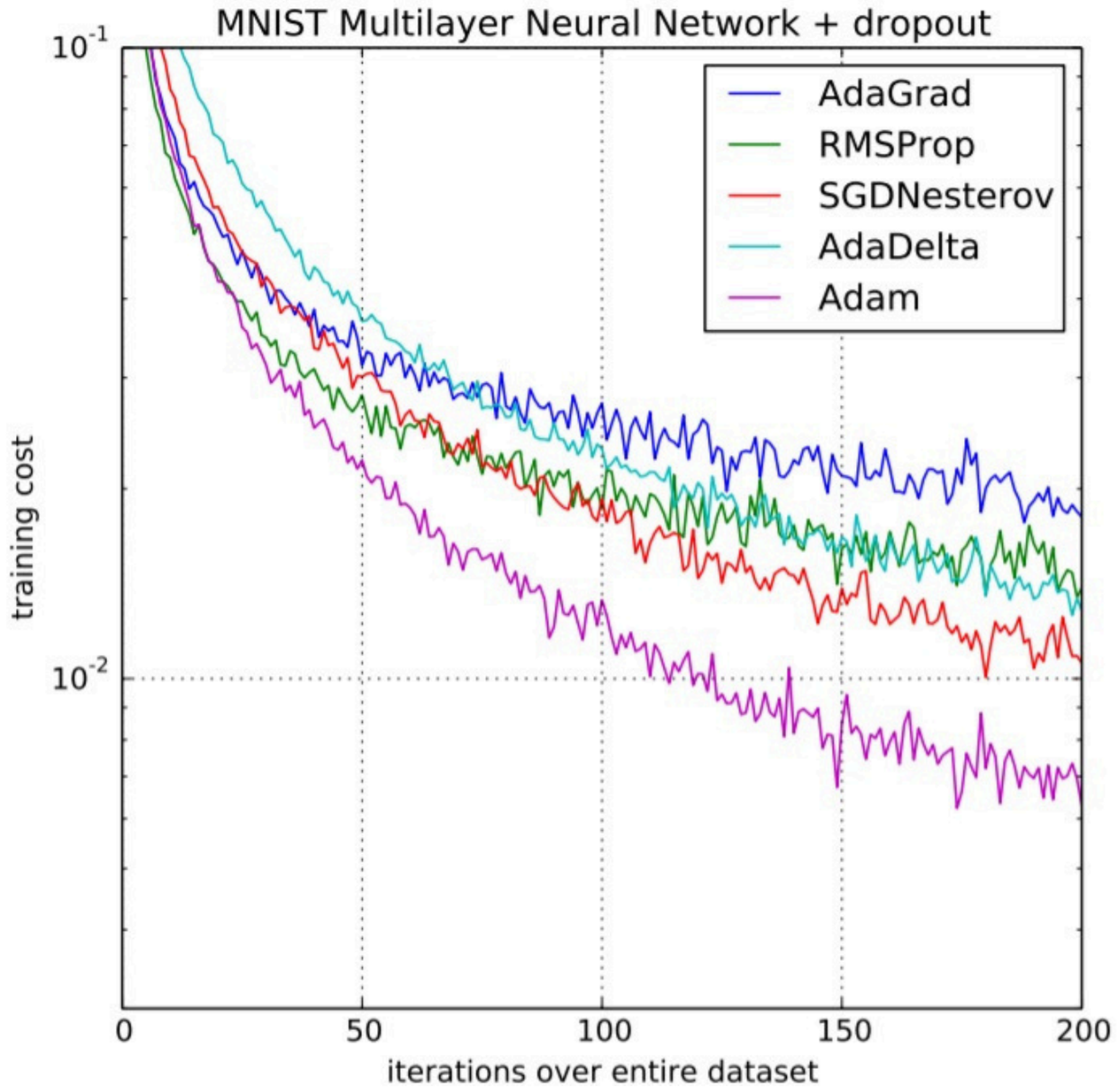
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\text{Adam} = \text{Momentum} + \text{RMSProp}$$



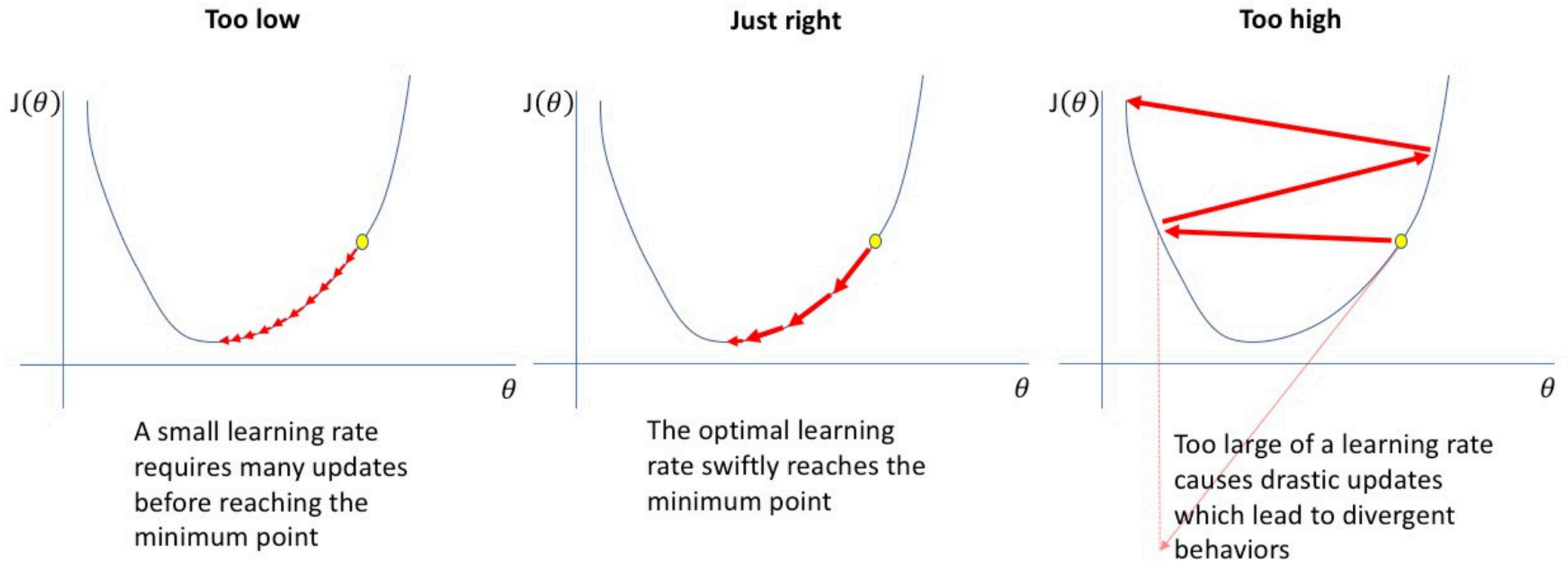
Optimization



Learning Rate

Learning Rate

- SGD, SGD with Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter.



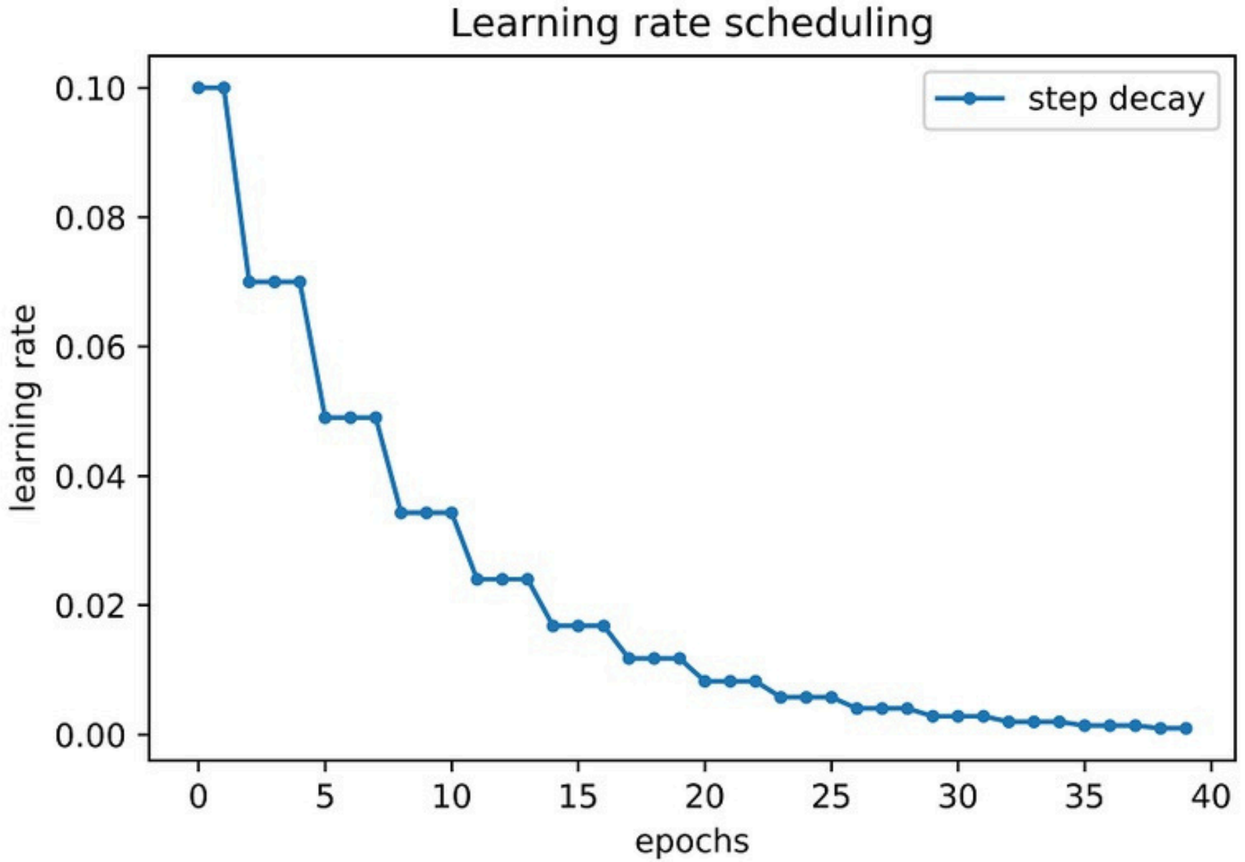
Learning Rate

Learning Rate Schedulers

Step Decay

$$\alpha_t = \alpha_0 \cdot \gamma^{\lfloor t/k \rfloor}$$

Drop learning rate every k epochs.



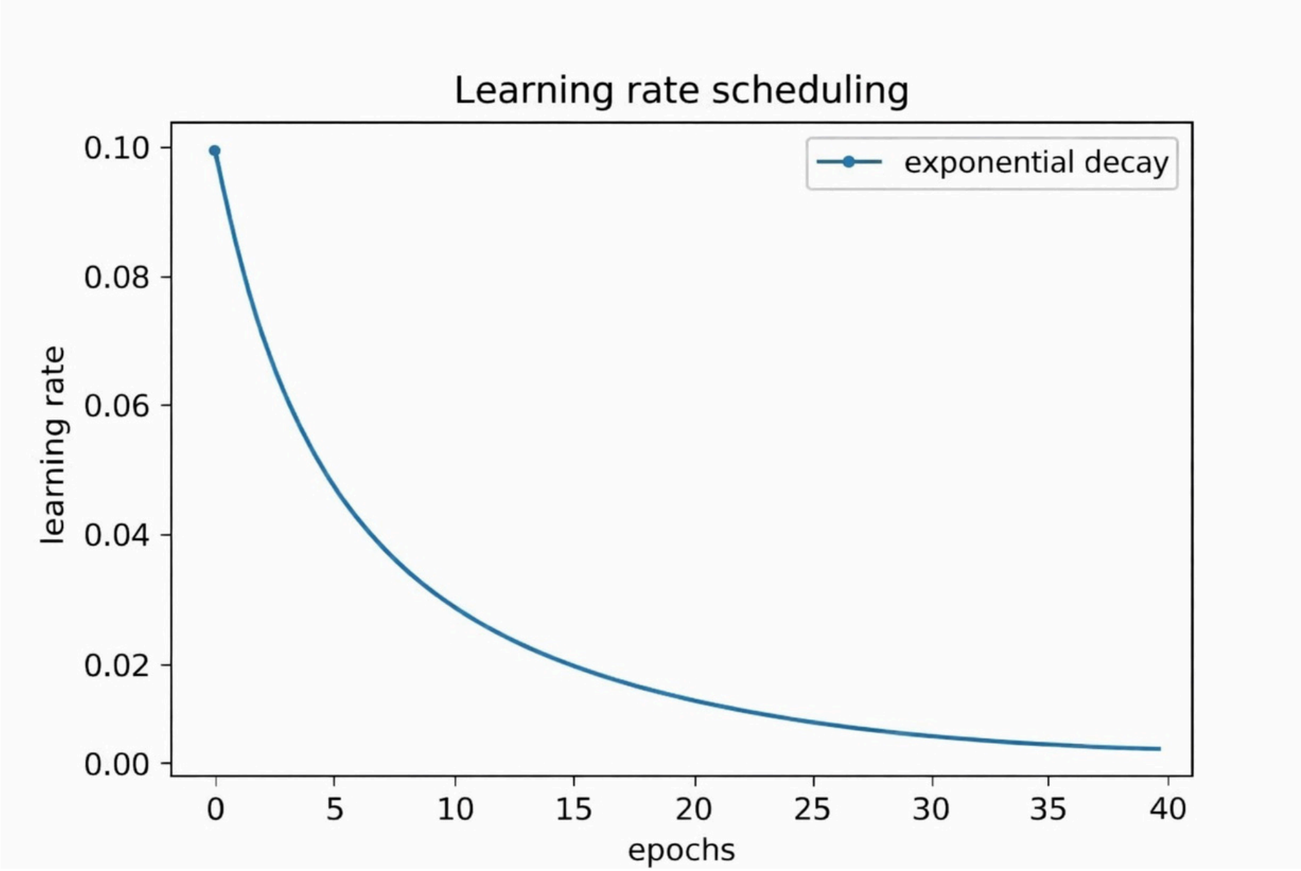
Learning Rate

Learning Rate Schedulers

Exponential Decay

$$\alpha_t = \alpha_0 e^{-kt}$$

Smooth continuous decay.

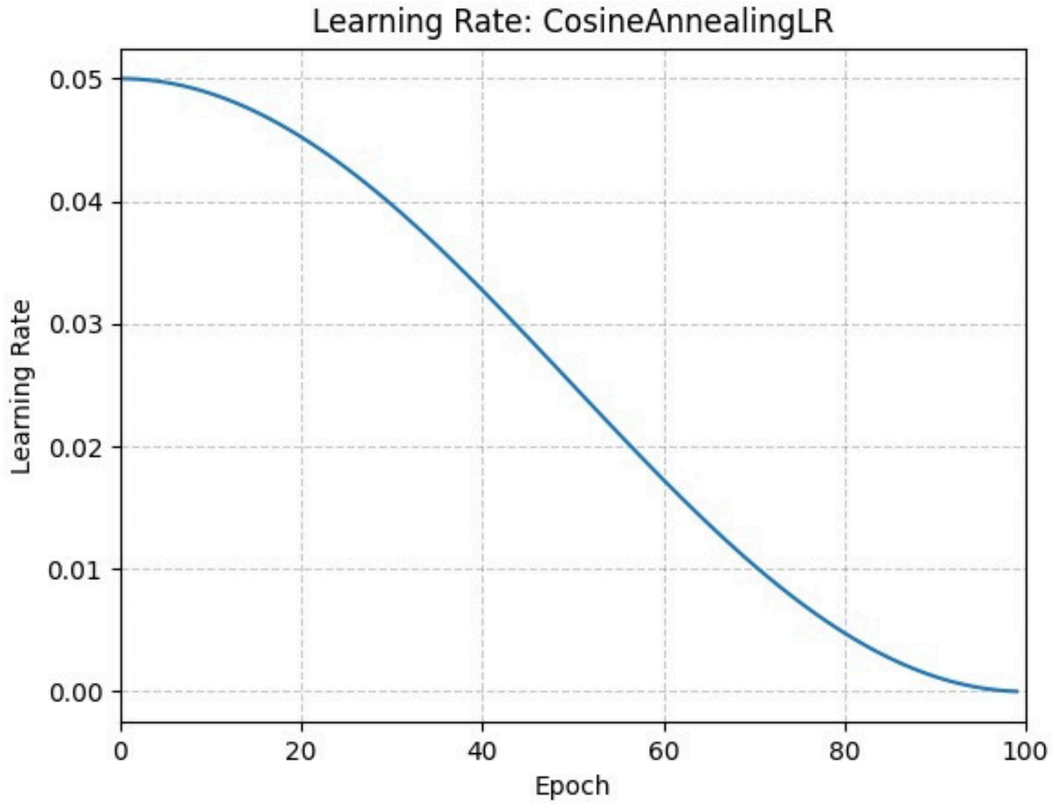


Learning Rate Schedulers

Cosine Annealing

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})\left(1 + \cos\left(\frac{T_{\text{cur}}}{T_{\max}}\pi\right)\right)$$

Most popular in modern deep learning.




Q
A

Thank You