STEVENS
INSTITUTE OF TECHNOLOGY
1870

PEP 559
Machine Learning in Quantum Physics

Dr. Chunlei Qu

Spring 2025

Four Modules

- Module A: Machine Learning
- Module B: Deep Learning
- Module C: Quantum Information
- Module D: Machine Learning for Quantum Physics

Three types of machine learning

Supervised learning

- Labeled data
- Direct feedback
- Predict outcome/future

Unsupervised learning

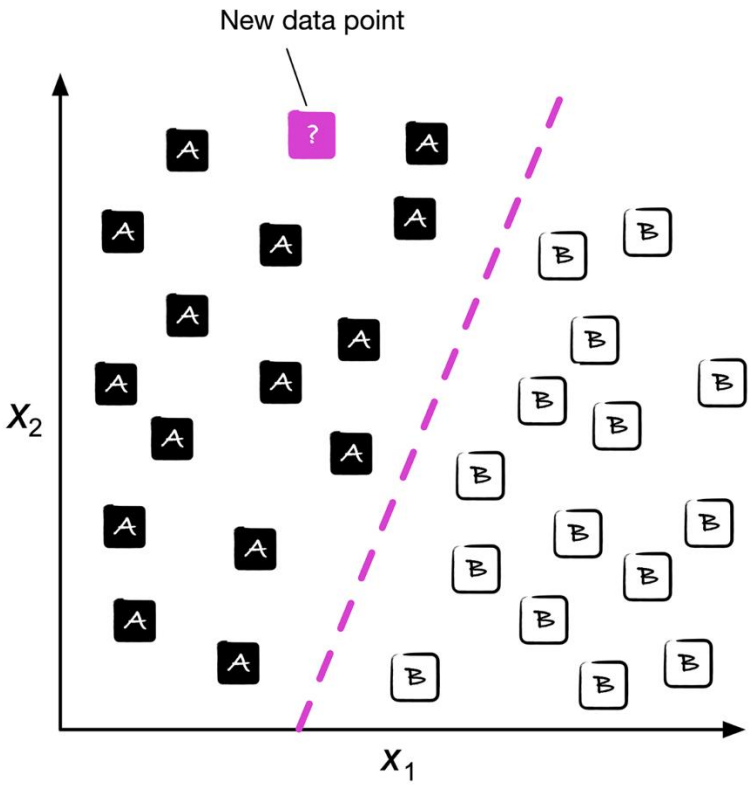
- No labels/targets
- No feedback
- Find hidden structure in data

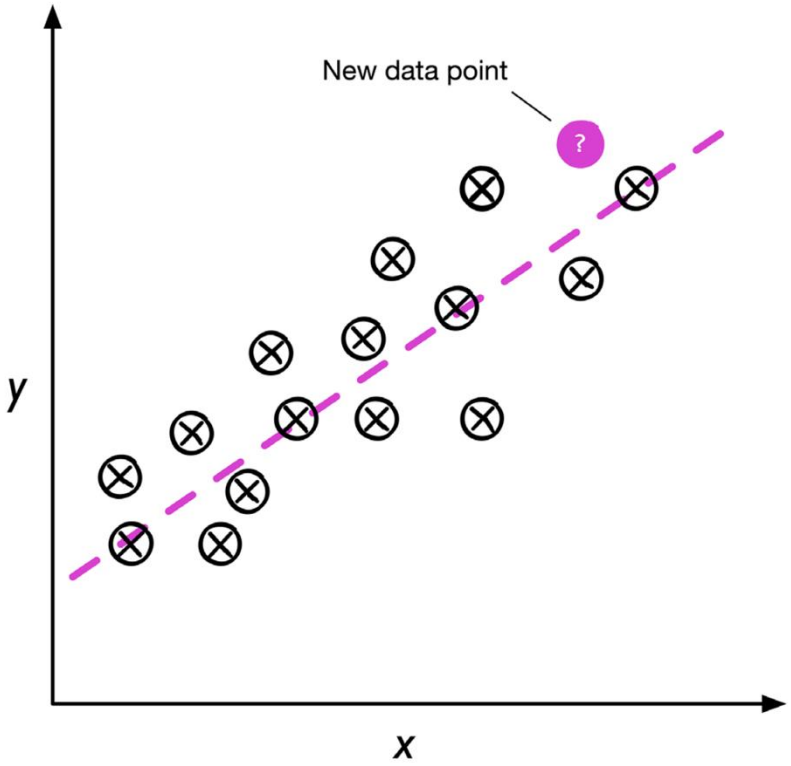
Reinforcement learning

- Decision process
- Reward system
- Learn series of actions

Supervised learning

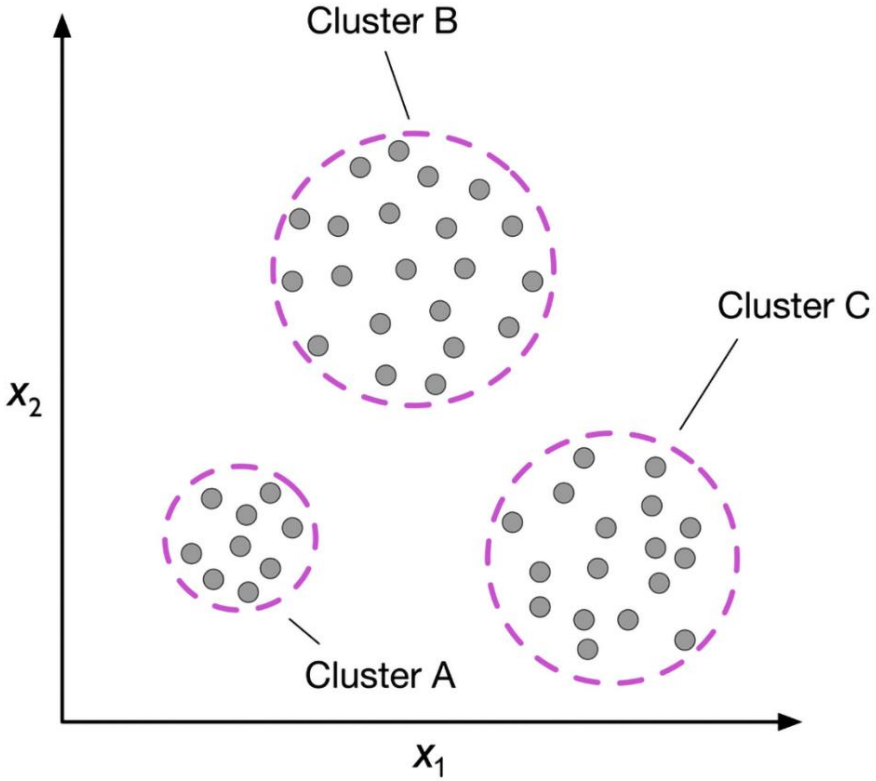
- **Classification:** labels are **discrete**, e.g., email spam detection is a binary classification task
- **Regression:** labels are **continuous**, e.g., house price vs. size

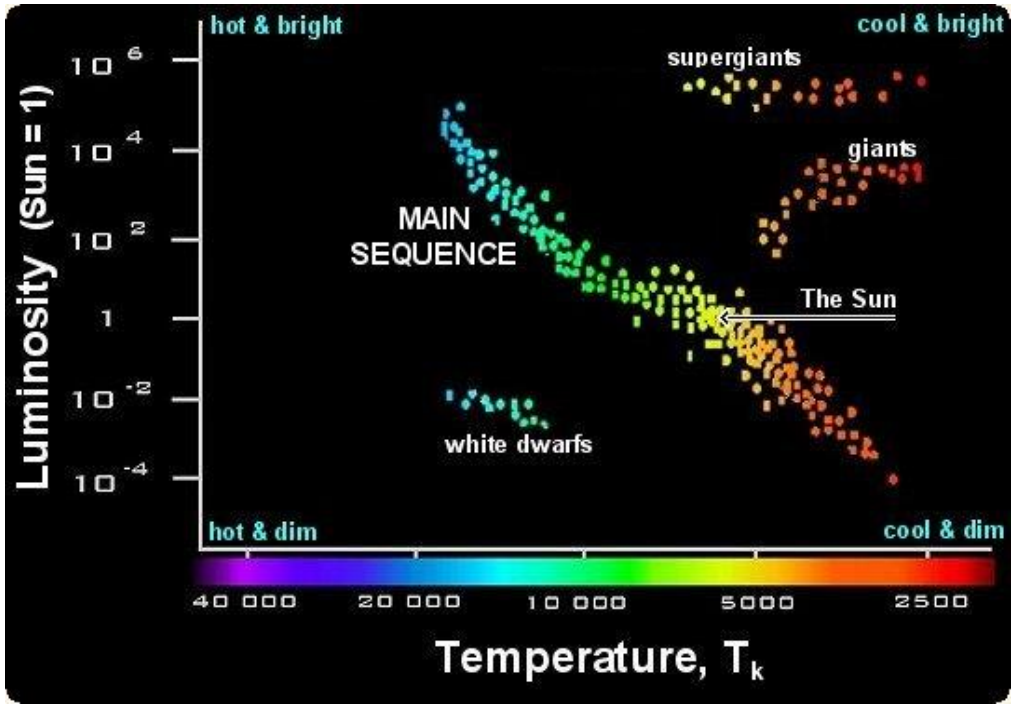




Unsupervised learning

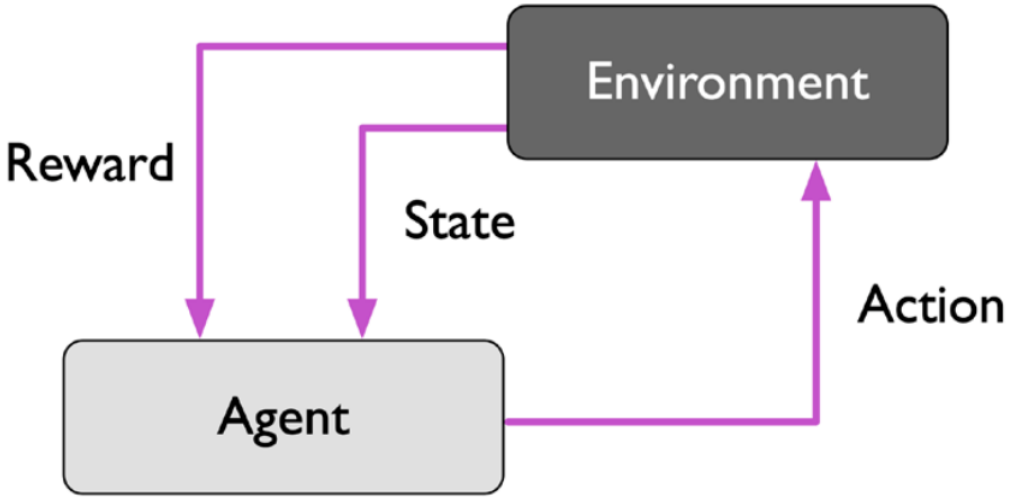
- **Clustering**: Discovering hidden structure of **unlabeled** data
- For example, the **Hertzsprung-Russell diagram** groups stars by temperature and luminosity

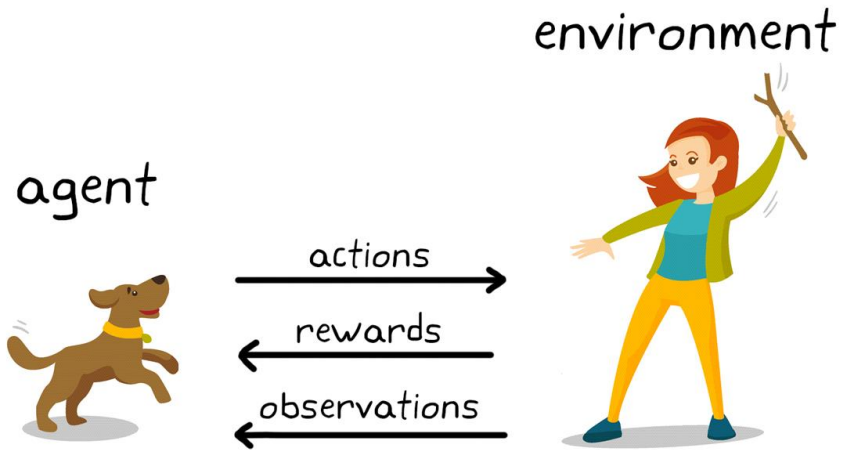




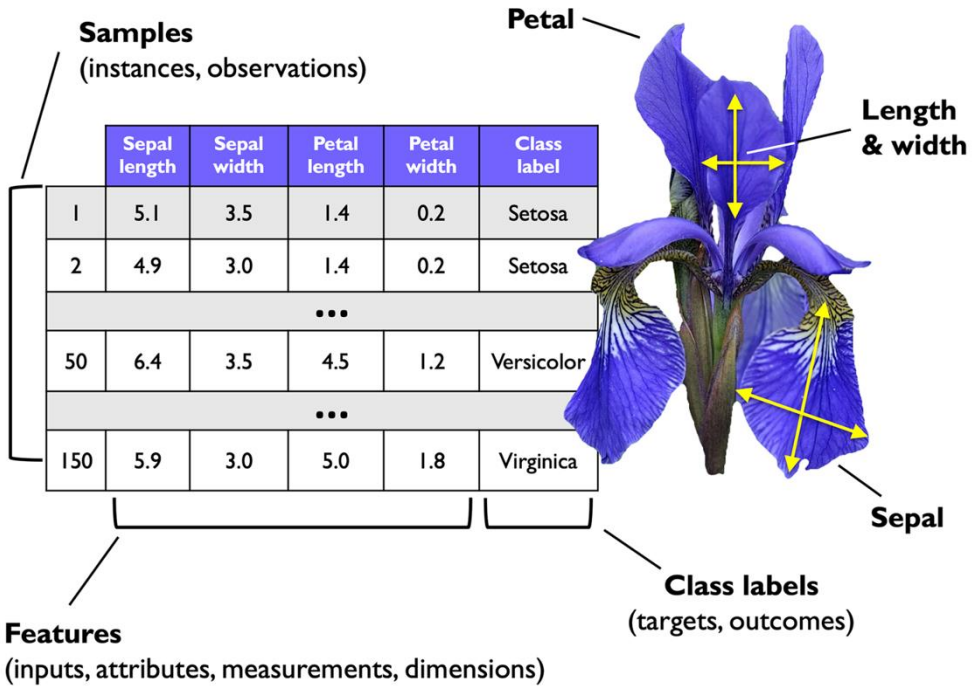
Reinforcement learning

- To develop a system (**agent**) that improves its performance based on interactions with the environment





Notation and Terminology



	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

The Iris DataSet

- **4 Features**: Sepal length, Sepal width, Petal length, Petal width
- **150 Samples** or instances or observations, etc.
- **Class labels**: Setosa, Versicolor, Virginica.

Data Matrix

- Superscript = **sample** index = row index
- Subscript = **feature** index = column index

$$\boldsymbol{X} \in \mathbb{R}^{150\times 4}$$

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

sample vector

feature vector

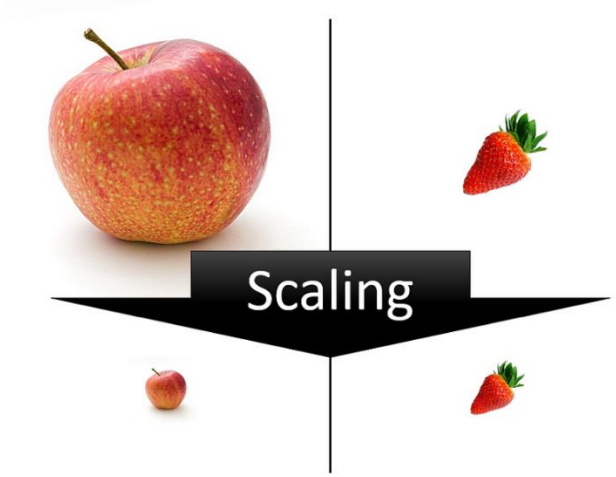
Terminology

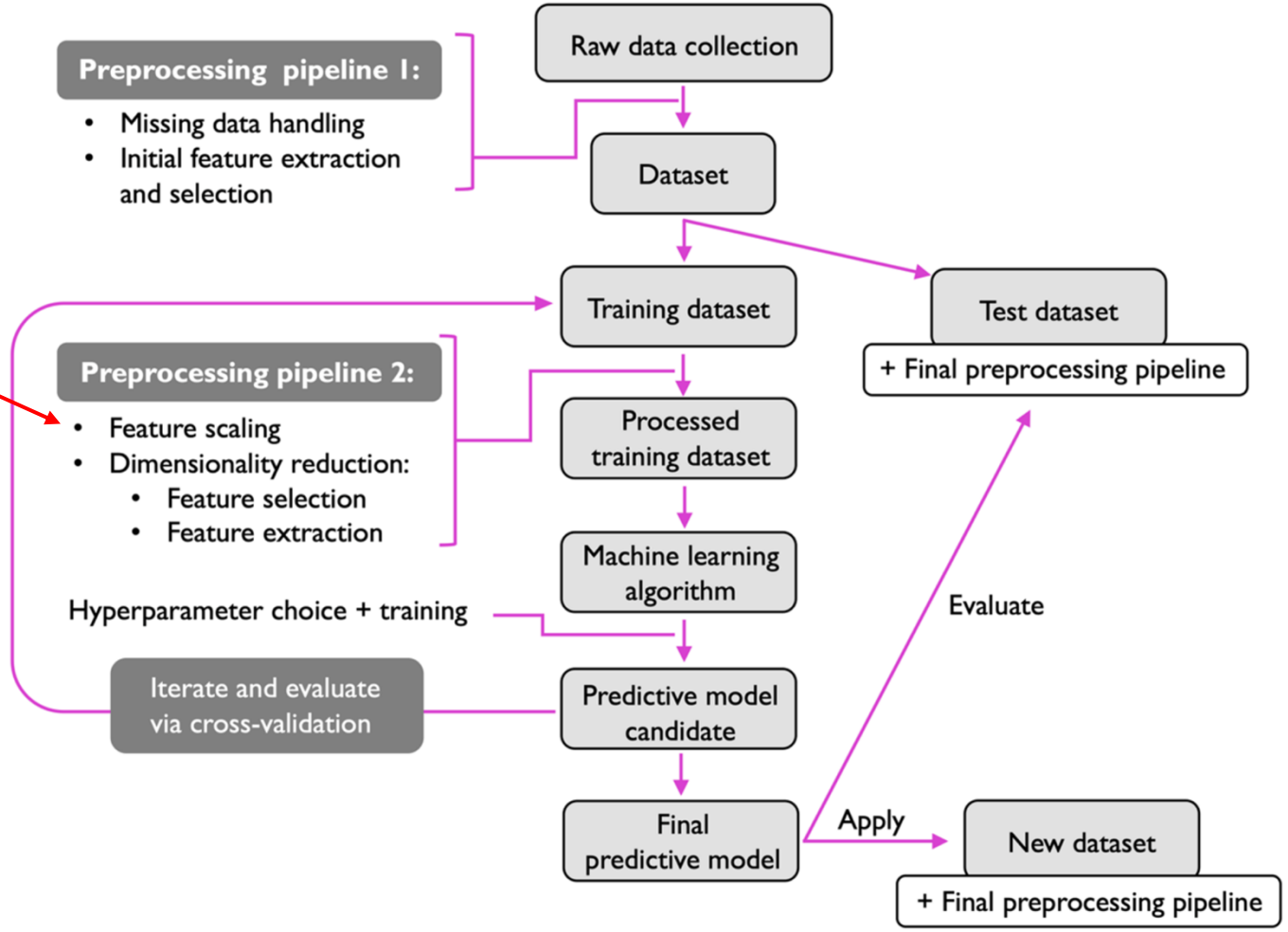
- **Training example**: a row in the data matrix, also known as an observation, record, instance, or sample
- **Feature**: a column in the data matrix, also known as predictor, variable, input, attribute
- **Target**: also known as class label, ground truth, outcome, output, etc.
- **Loss function**: also known as cost function or error function

ML typical workflow

- **Feature scaling**

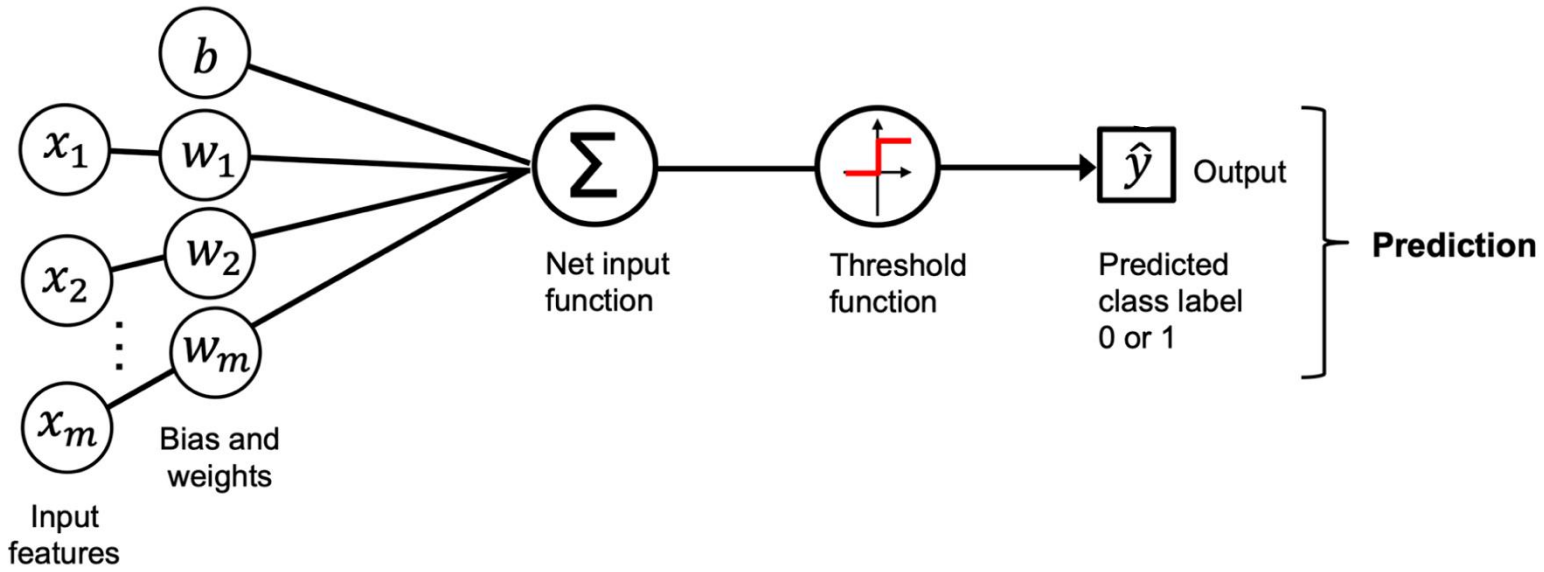
The features should be on the same scale for optimal performance. Normally, we transform it to a standard distribution with zero mean and unit variance.





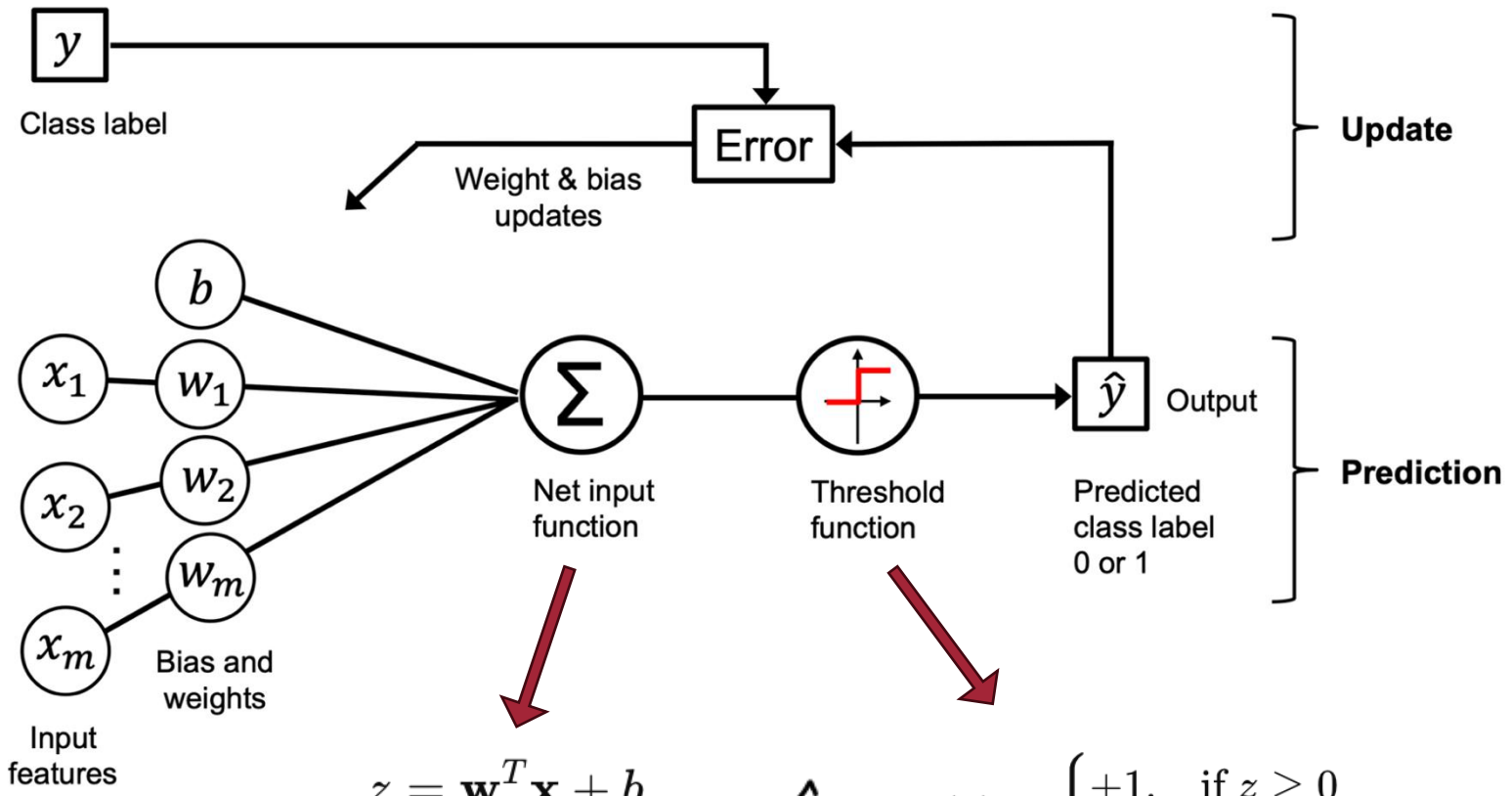
McCulloch-Pitts (MCP) neuron model

- **Pre-determined weights, no learning capability**



Rosenblatt's perceptron model

- Proposed an algorithm that would automatically **learn the optimal weight** coefficients

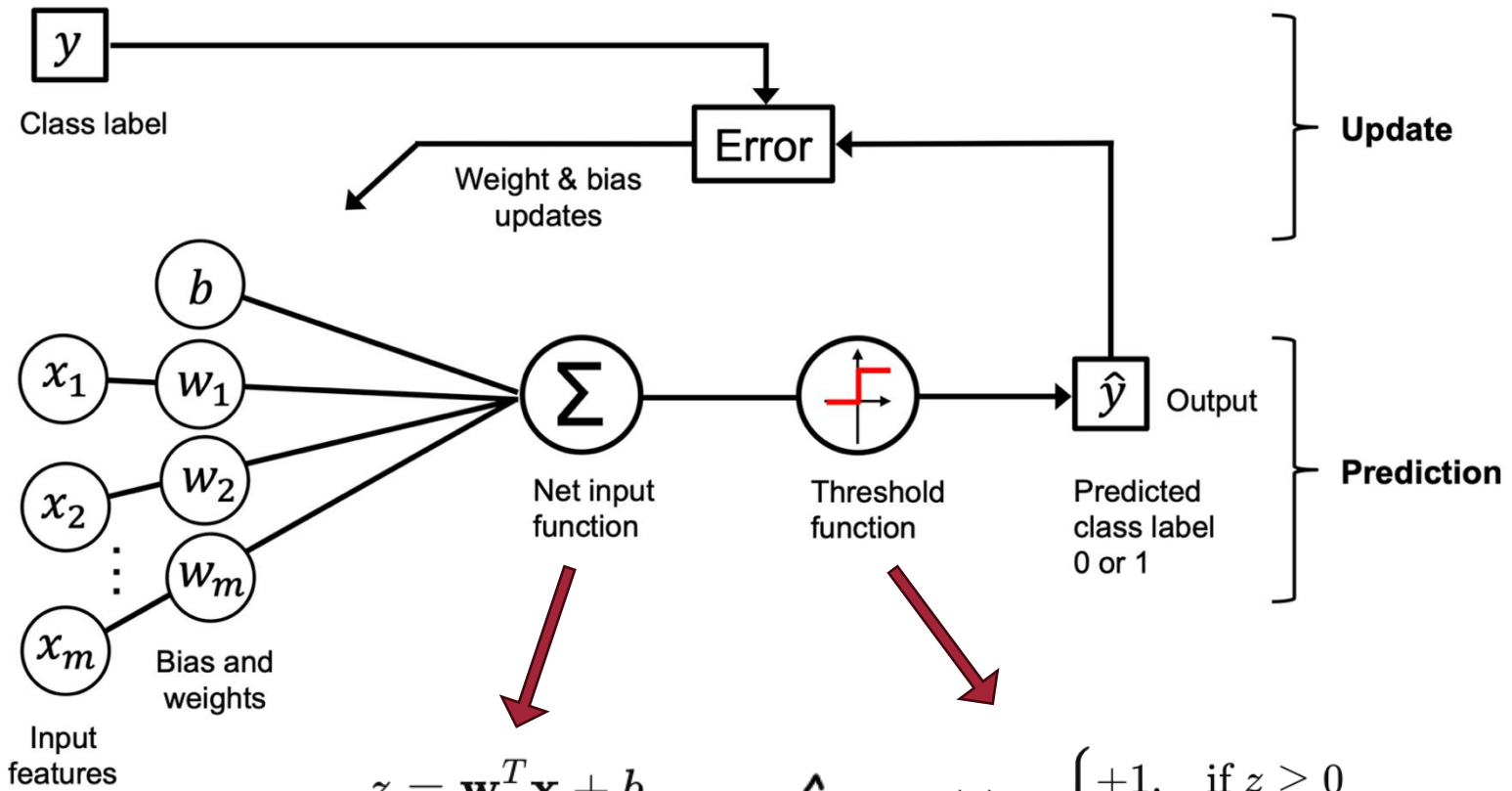


$$z = \mathbf{w}^T\mathbf{x} + b$$

$$\hat{y} = \text{sgn}(z) = \begin{cases} +1, & \text{if } z \geq 0 \\ -1, & \text{if } z < 0 \end{cases}$$

Key idea to adjust the weight (and bias)

- If predicted label is 1, but the actual label is 0, we want to reduce the weight
- If predicted label is 0, but the actual label is 1, we want to enhance the weight



$$z = \mathbf{w}^T\mathbf{x} + b$$

$$\hat{y} = \text{sgn}(z) = \begin{cases} +1, & \text{if } z \geq 0 \\ -1, & \text{if } z < 0 \end{cases}$$

The perceptron learning rule

1. Initialize the weights and bias unit to 0 or small random numbers
2. For each training example, $\boldsymbol{x}^{(i)}$:
   a. Compute the output value, $\hat{y}^{(i)}$
   b. Update the weights and bias unit

$$w_j := w_j + \Delta w_j$$
$$\text{and} \quad b := b + \Delta b$$

The update values ("deltas") are computed as follows:

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$
$$\text{and} \quad \Delta b = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)$$

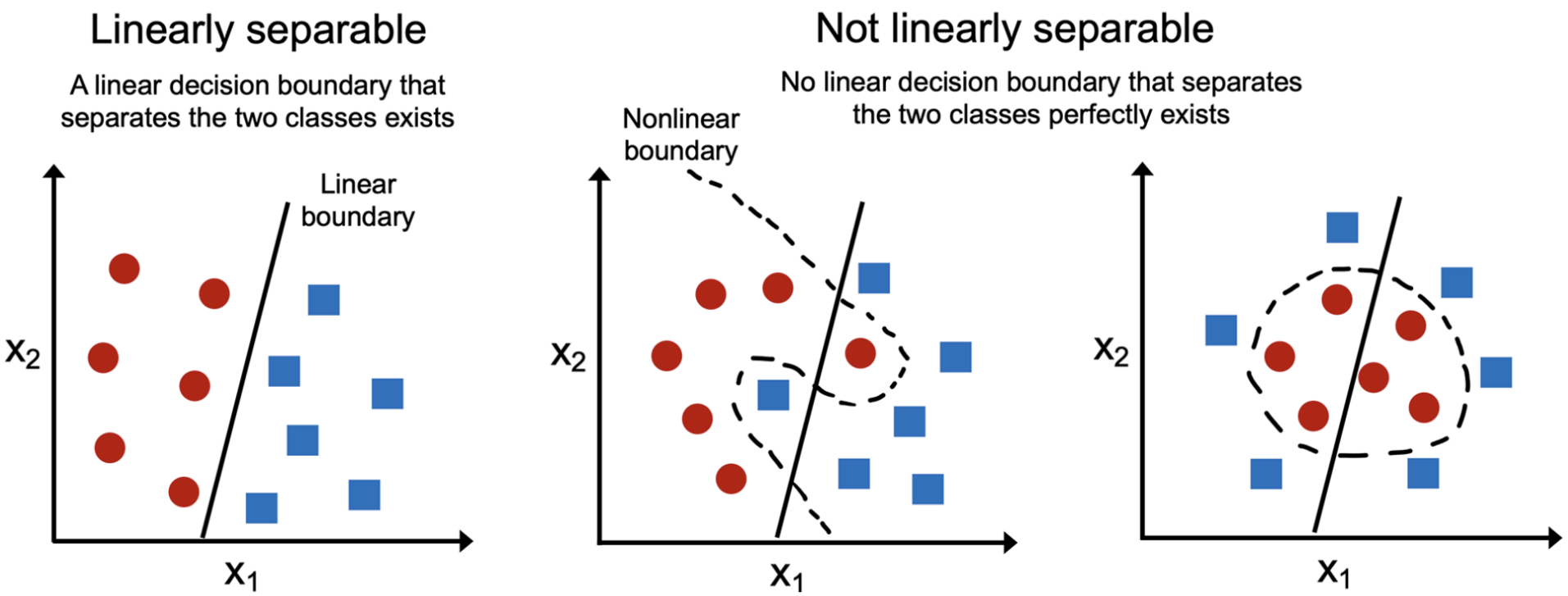
predicted class label

learning rate

actual class label

Applicable to linearly separable data only

- The algorithm finds the linear decision boundary after certain number of iterations (**epochs**)



Python basics

- See Jupyter Notebook: **Python_CheatSheet.ipyb**

- **Virtual environment** with conda
- **Jupyter Notebook**
- Essential packages: **numpy, matplotlib, pandas, seaborn, scipy**

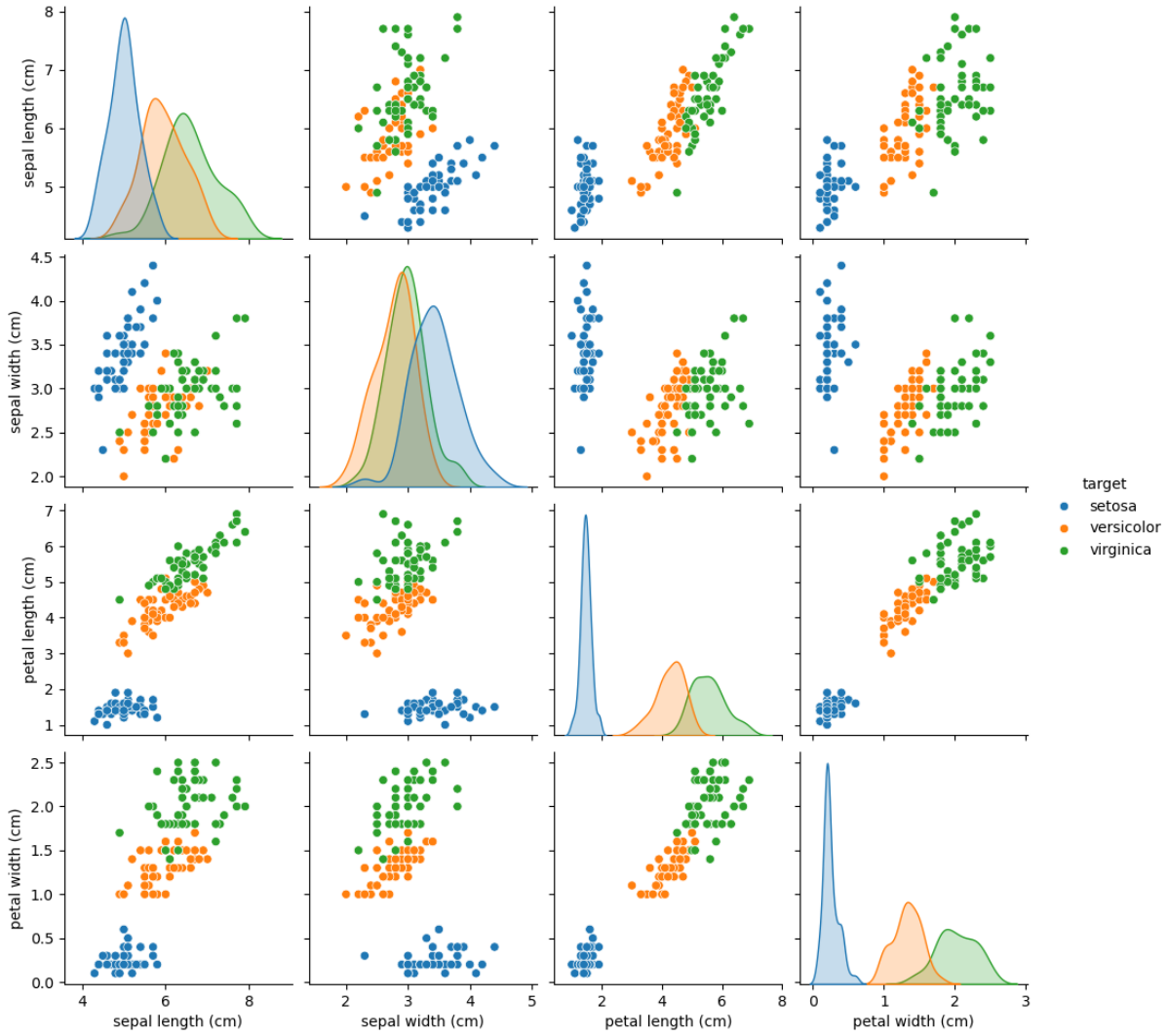
Demo: Iris flowers classification

- See jupyter notebook: **demo_Iris.ipynb**

- Iris Dataset: **4 feature variables, 3 classes, 150 samples**

- Rosenblatt's model is specifically designed for **binary classification** tasks

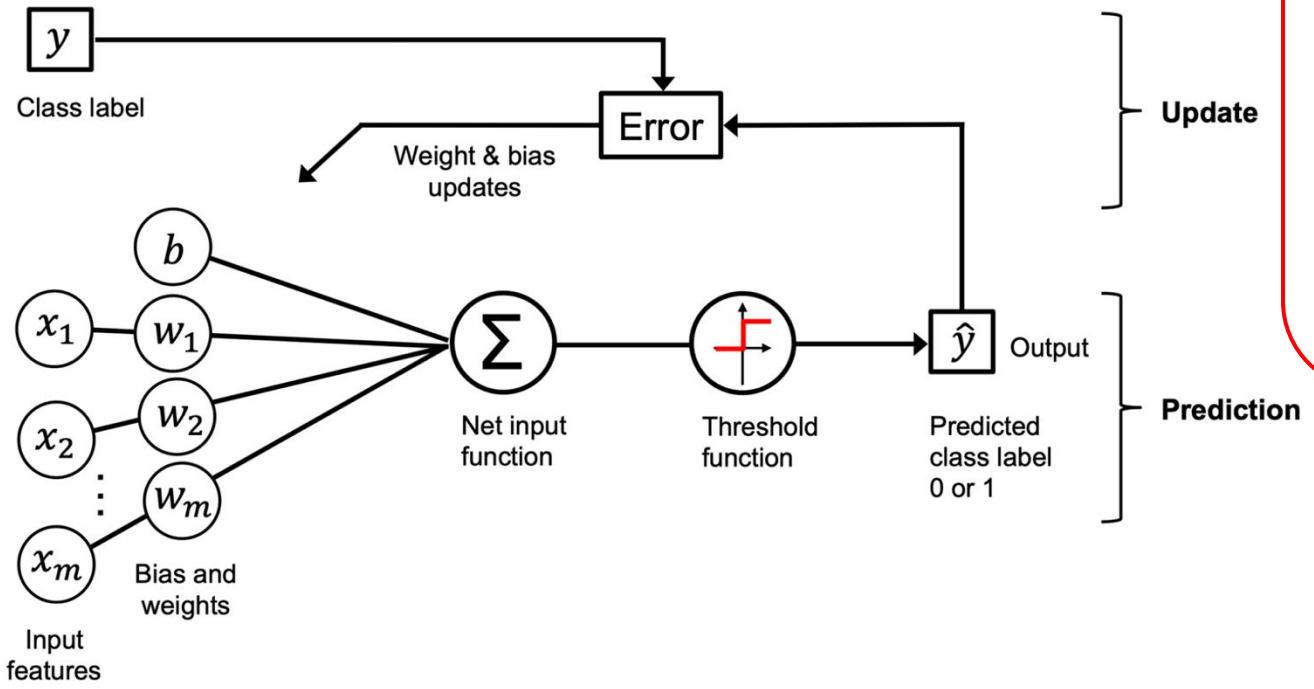
- Need to **remove the data for one class** before we apply Rosenblatt Perceptron model



Rosenblatt perceptron

- **Single-layer** NN
- The weights are updated based on a **step function**
- The weight update is calculated incrementally after **EACH** training example

- Sample 1
- Sample 2
- Sample 3
- …
- Sample 100



$$w_j := w_j + \Delta w_j$$
$$\text{and} \quad b := b + \Delta b$$

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$
$$\text{and} \quad \Delta b = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)$$

learning rate

actual class label

predicted class label

j: feature index

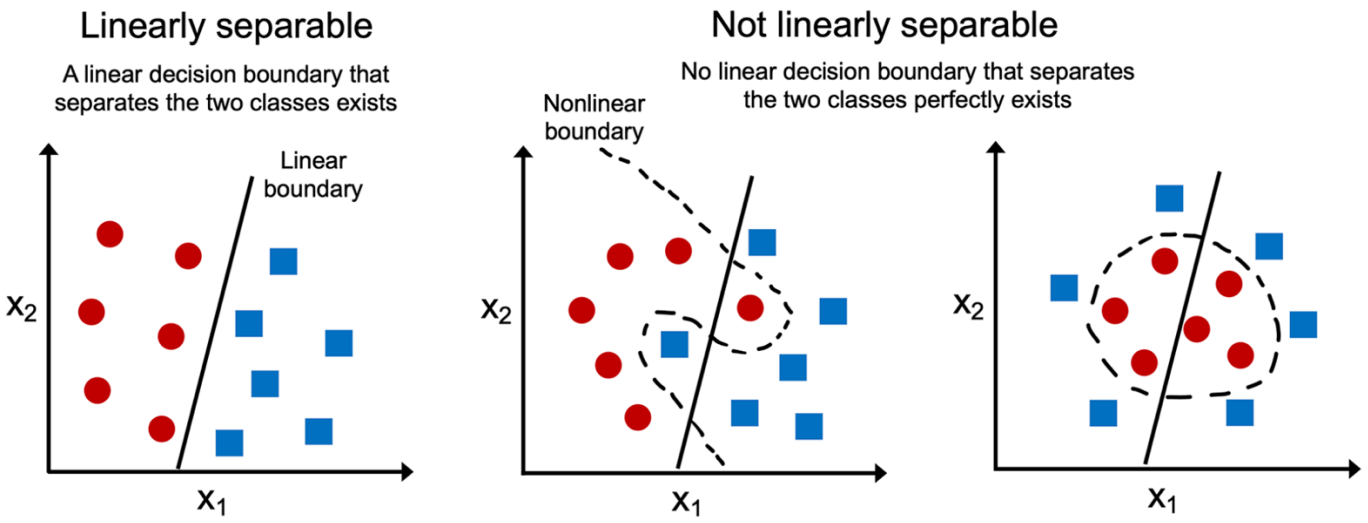
i: sample index

m=4 (features)

Perceptron convergence theorem

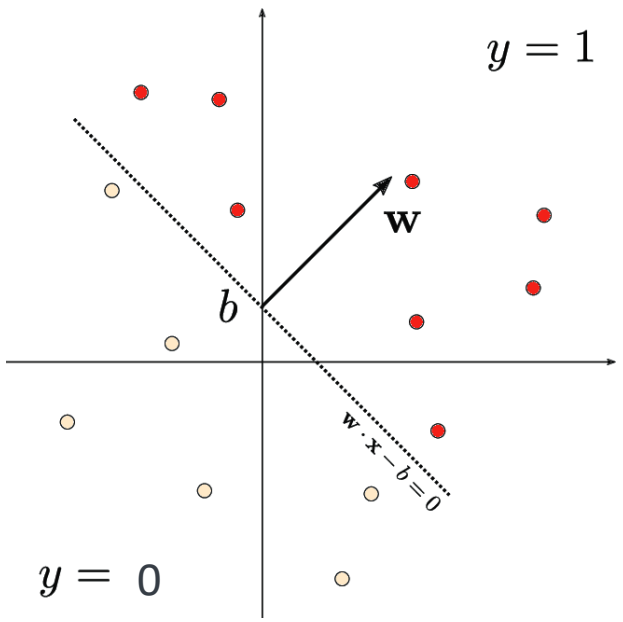
- Rosenblatt proved mathematically that the perceptron learning rule **converges** if the two classes can be **separated by a linear hyperplane.**

- If two classes cannot be separated by a linear hyperplane, the weights will never stop updating unless we set a maximum number of iterations (or epochs)



Geometric intuition

The weight vector is perpendicular to the decision boundary.



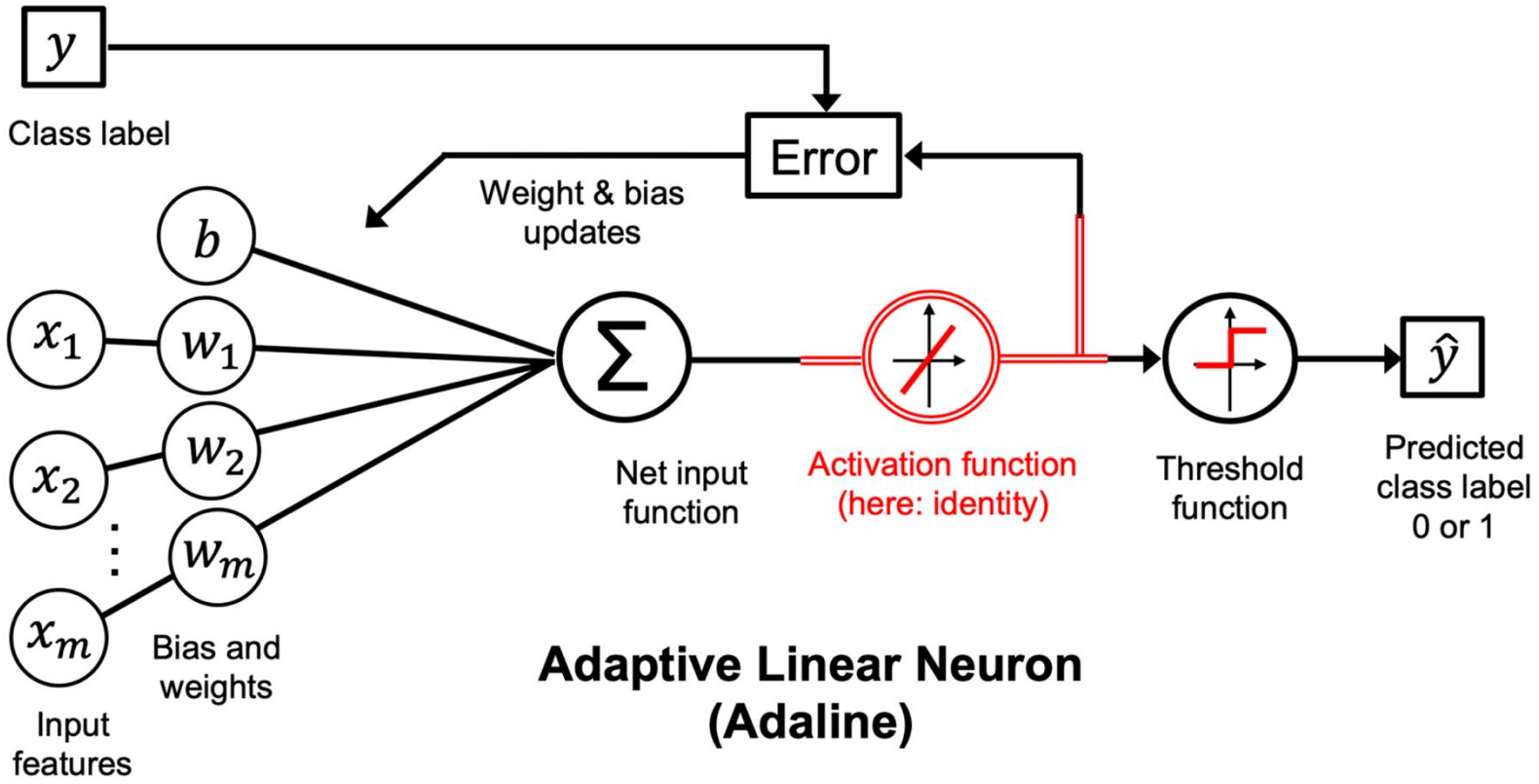
$$\hat{y} = \begin{cases} 0, & \mathbf{w}^T\mathbf{x} \leq 0 \\ 1, & \mathbf{w}^T\mathbf{x} > 0 \end{cases}$$

$$\mathbf{w}^T\mathbf{x} = ||\mathbf{w}|| \cdot ||\mathbf{x}|| \cdot \underbrace{\cos(\theta)}$$

So this needs to be 0 at the boundary, and it is zero at 90°

Adaptive linear neuron (Adaline)

- A generalized Rosenblatt's neuron model by Bernard Widrow and Tedd Hoff (**1960**)



Key difference

Learning

$$\sigma(z) = z$$

- In the Adaline rule, the weights are updated based on a **linear activation function** rather than a step function
- The weight update is calculated based on **all samples** in the training dataset (instead of updating the parameters incrementally after each training sample)
- It is referred to as **full batch gradient descent**

Prediction

- While the linear activation function is used for learning the weights, we still use a **step function** to make the final prediction

Adaline learning: Gradient descent

- To minimize the **objective function**, or loss or cost function

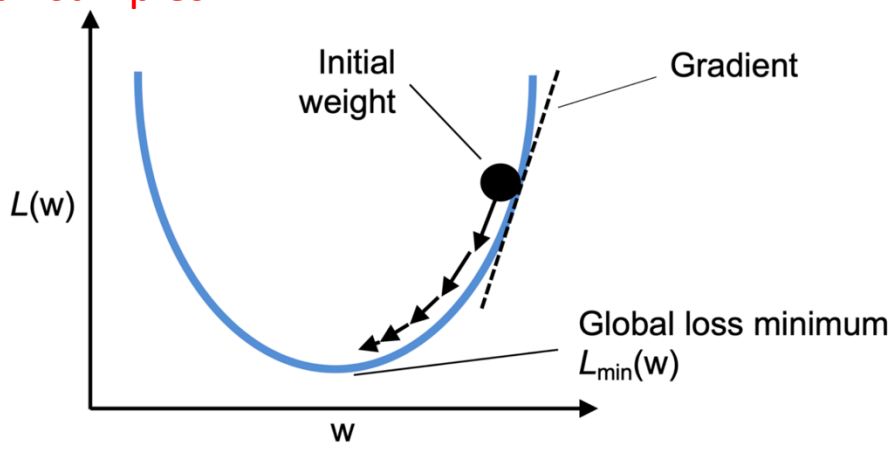
- **Mean squared errors (MSE)**

weighted sum of i-th feature vector

$$L(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)^2$$

actual class label

Sum over all samples



- **Adaline learning or updating rule**

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}, \quad b := b + \Delta b$$

$$\Delta w_j = -\eta\frac{\partial L}{\partial w_j} \text{ and } \Delta b = -\eta\frac{\partial L}{\partial b}$$

$$\frac{\partial L}{\partial w_j} = -\frac{2}{n}\sum_i\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)x_j^{(i)}$$

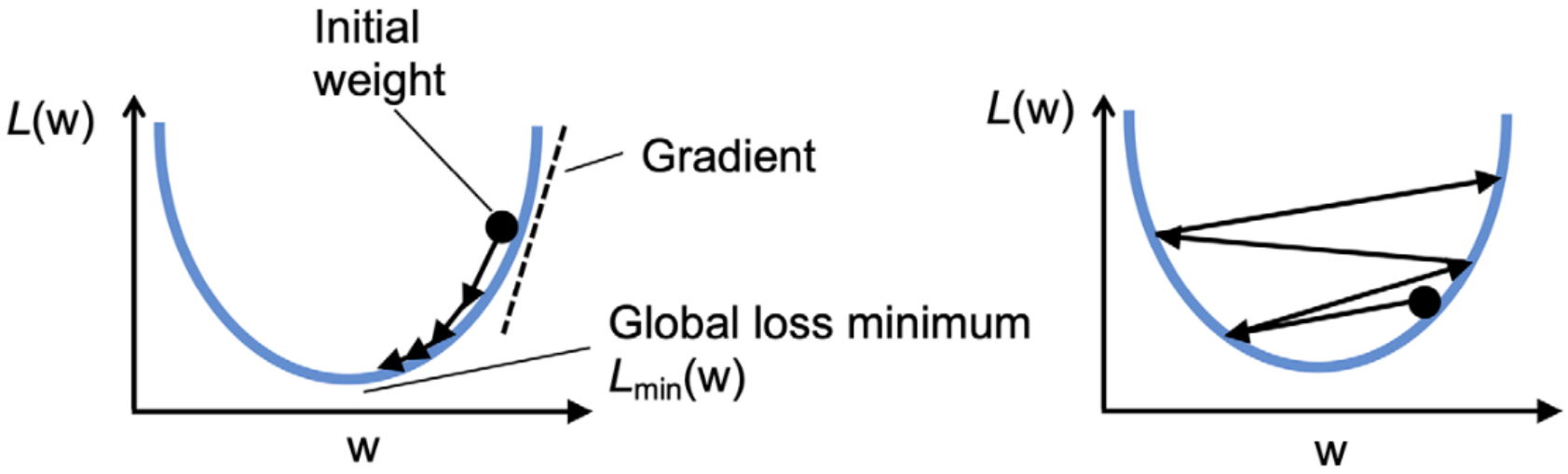
$$\frac{\partial L}{\partial b} = -\frac{2}{n}\sum_i\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)$$

Advantages of this MSE loss function

i) Differentiable

ii) It is convex; thus a local or global minimum can be reached by climbing down the hill (along the negative direction of the gradient)

Learning rate



- If we choose a learning rate that is too large --- we **overshoot** the global minimum
- If it is too small --- training will be slow and might get stuck in local minima (for complex loss function). However, MSE loss function is convex and there are no local minima.

Python implementation of Adaline

- See Jupyter notebook: **demo_iris_Adaline.ipynb**

Feature scaling

- Many ML algorithms require feature scaling for optimal performance

- Gradient descent is one of the them that benefit from feature scaling. Other algorithms, such as regularization and k-means, also strongly depend on feature scaling. While the decision trees and random forests don't need to worry about feature scaling.

- **Standardization**

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

$$x_{norm}^{(i)} = \frac{x^{(i)} - \mathbf{x}_{min}}{\mathbf{x}_{max} - \mathbf{x}_{min}}$$

standardization

min-max scaling ("normalization")

	input	standardized	normalized
0	0	-1.46385	0.0
1	1	-0.87831	0.2
2	2	-0.29277	0.4
3	3	0.29277	0.6
4	4	0.87831	0.8
5	5	1.46385	1.0

- **Normalization**

- After feature scaling, it is **easier to find a learning rate that works well for all weights** (and bias).

Stochastic gradient descent

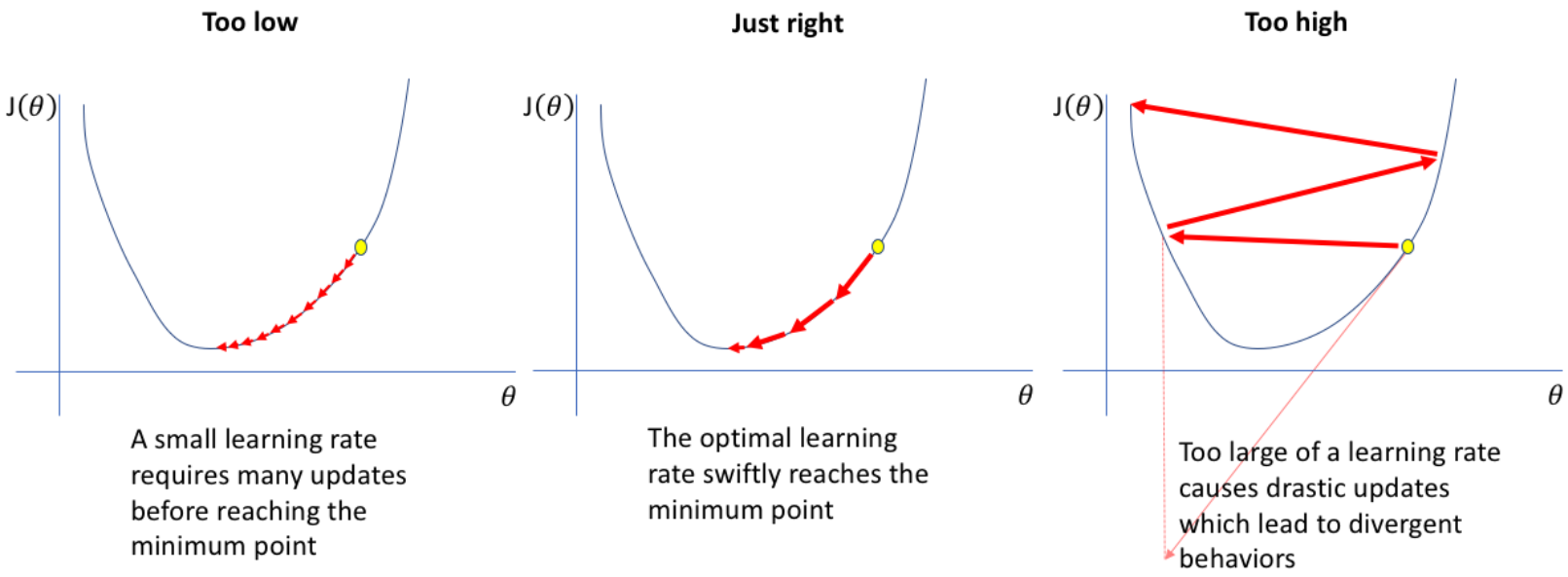
- For very large dataset with millions of data points, full batch gradient descent can be computationally expensive
- Instead of updating the weights based on the sum of the **accumulated errors over all training sample**, we update the parameters **incrementally for each training sample --- SDG**
- **Or use mini-batch gradient descent – apply full batch gradient to smaller subset of the training data.**
- Compared to SGD, we can replace the for loop over the training examples with vectorized operations, which can further improve the computational efficiency.

Adaptive learning rate

In SGD implementations, the fixed learning rate, η, is often replaced by an adaptive learning rate that decreases over time

$$\frac{c_1}{[\text{number of iterations}] + c_2}$$

where c1 and c2 are constants.
SGD does not reach the global loss minimum but an area very close to it.



Python demos

- See Jupyter notebook