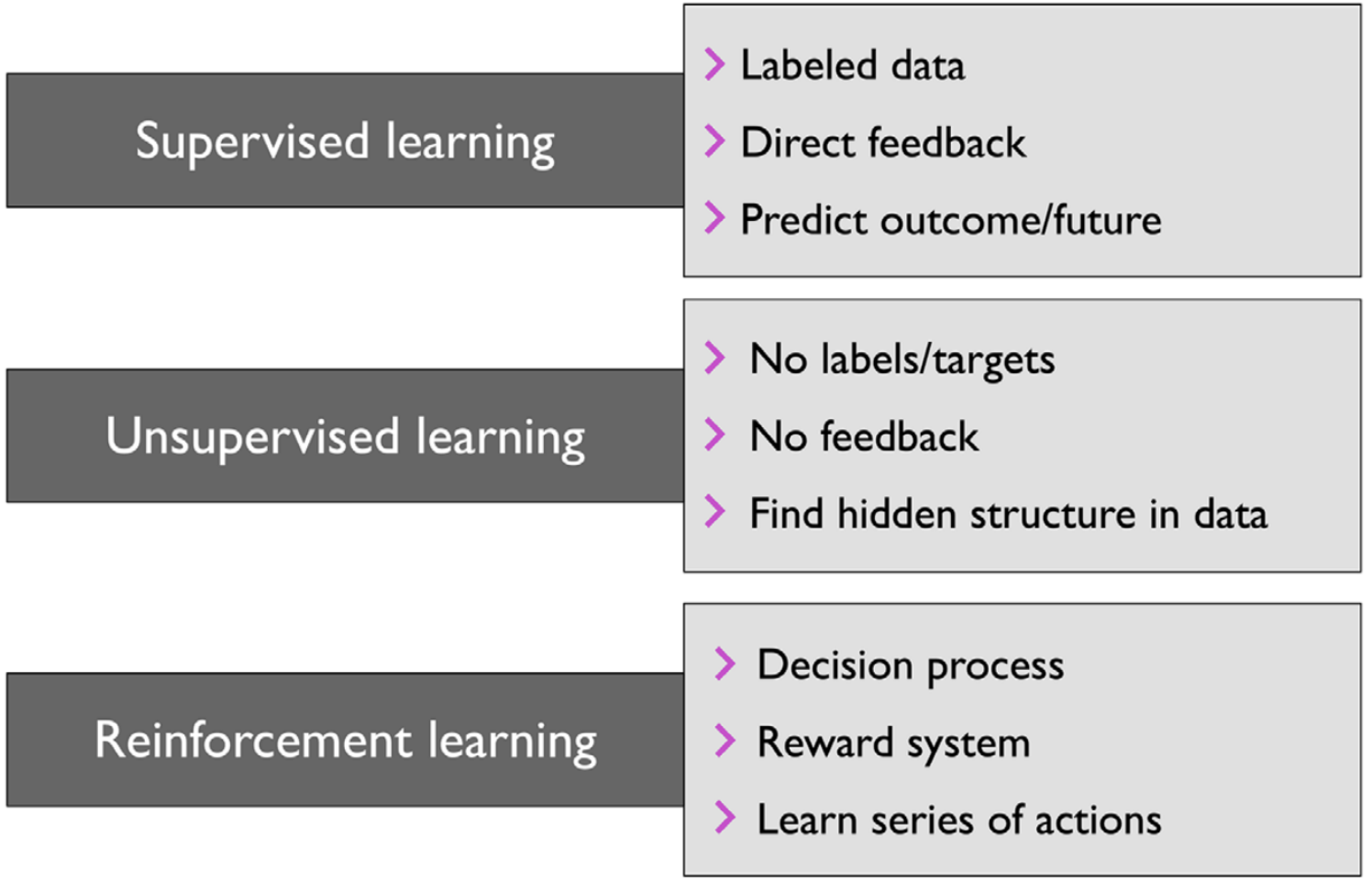
STEVENS
INSTITUTE OF TECHNOLOGY
1870

PEP 559
Machine Learning in Quantum Physics

Dr. Chunlei Qu

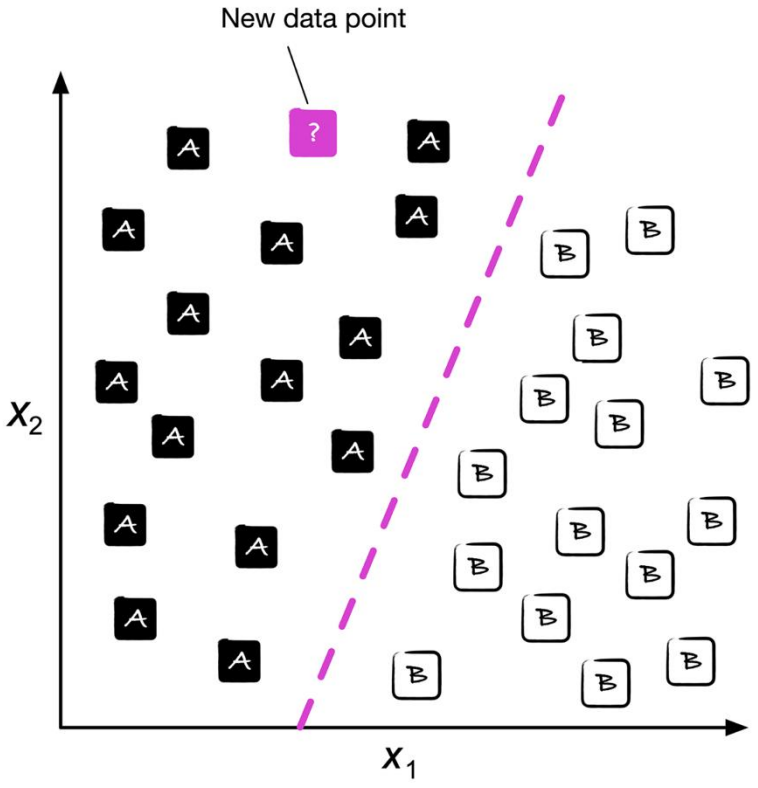
Spring 2026

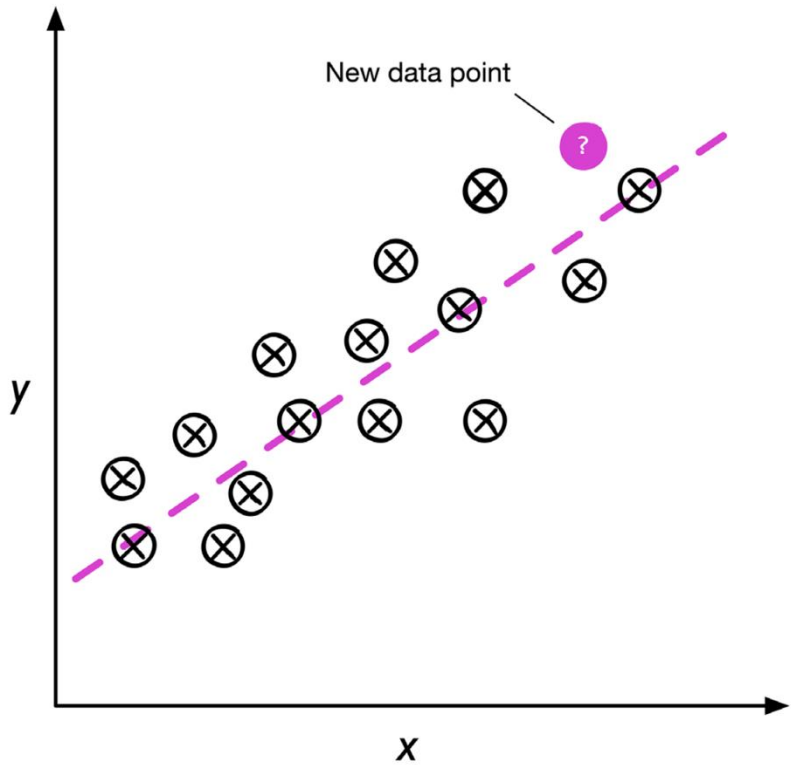
Three types of machine learning



Supervised learning

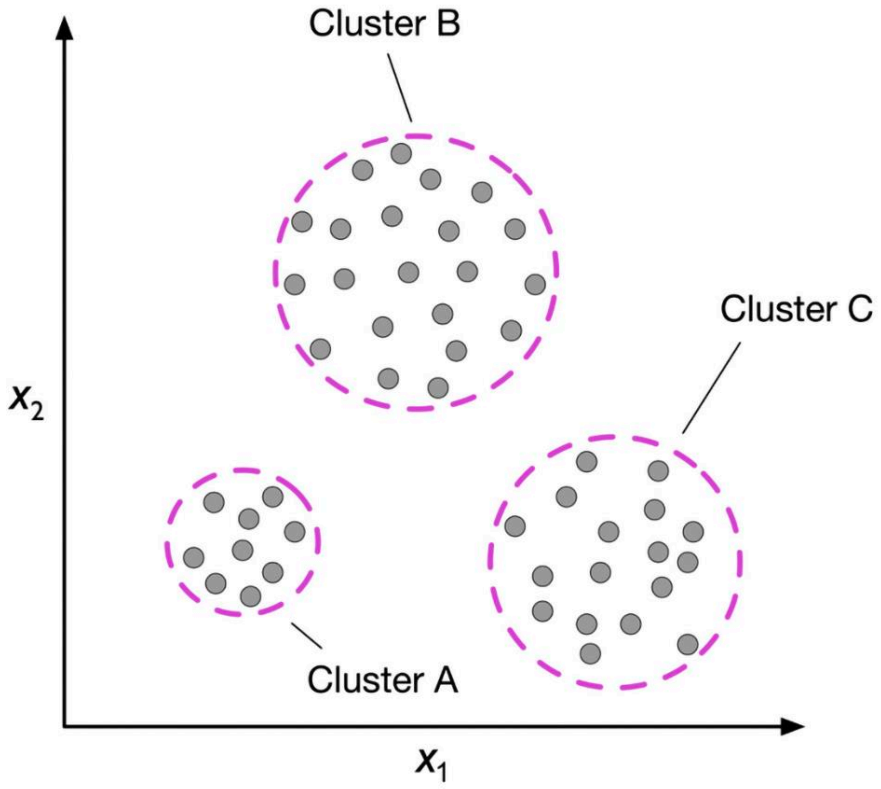
- **Classification:** labels are **discrete**, e.g., email spam detection is a binary classification task
- **Regression:** labels are **continuous**, e.g., house price vs. size

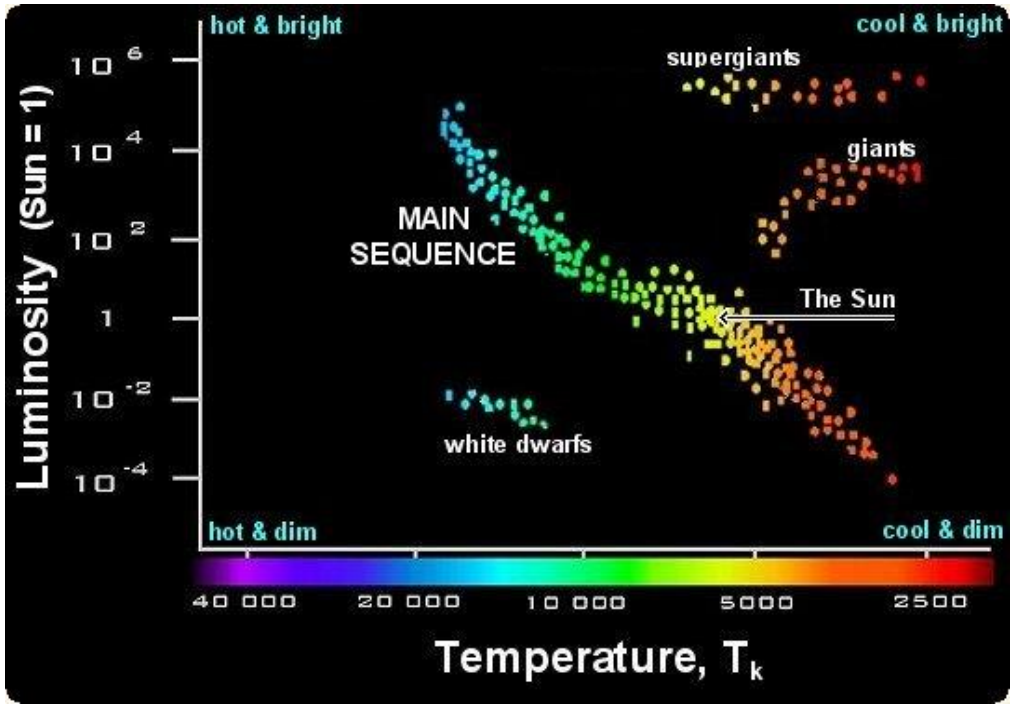




Unsupervised learning

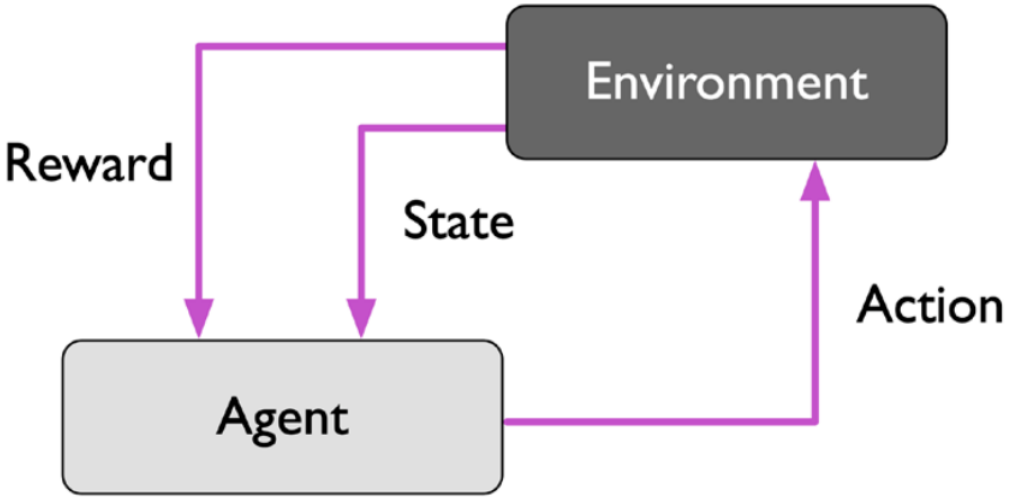
- **Clustering**: Discovering hidden structure of **unlabeled** data
- For example, the **Hertzsprung-Russell diagram** groups stars by temperature and luminosity

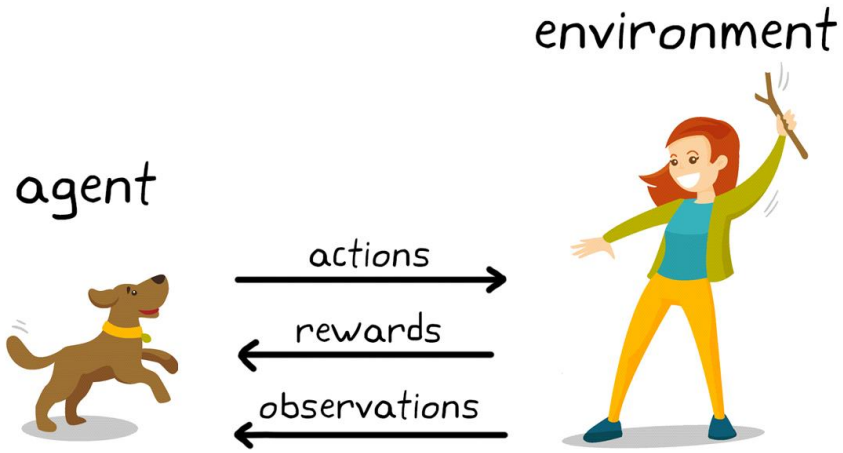




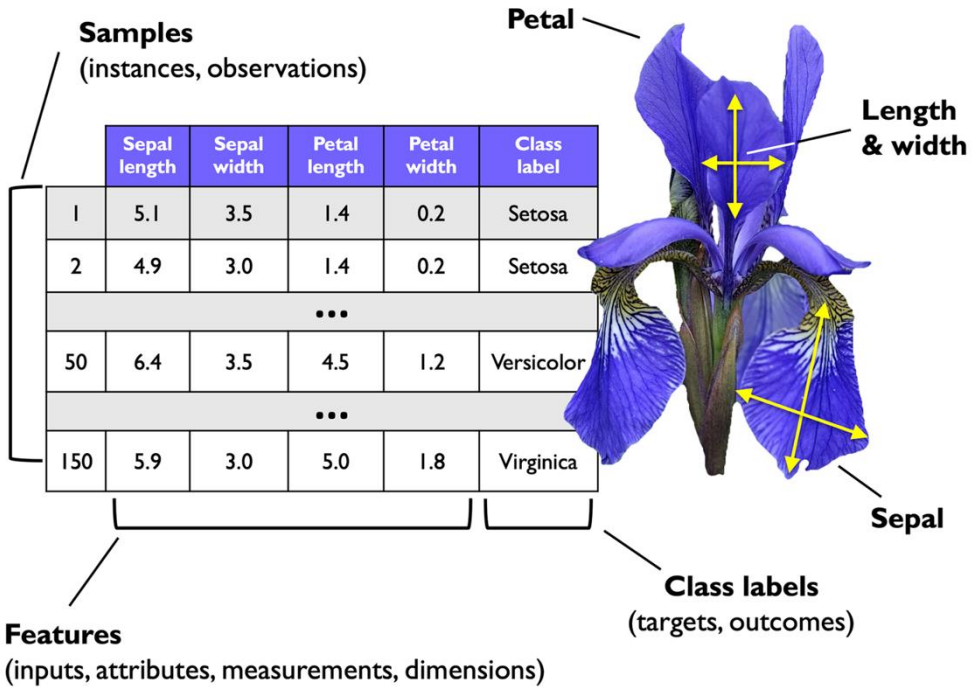
Reinforcement learning

- 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

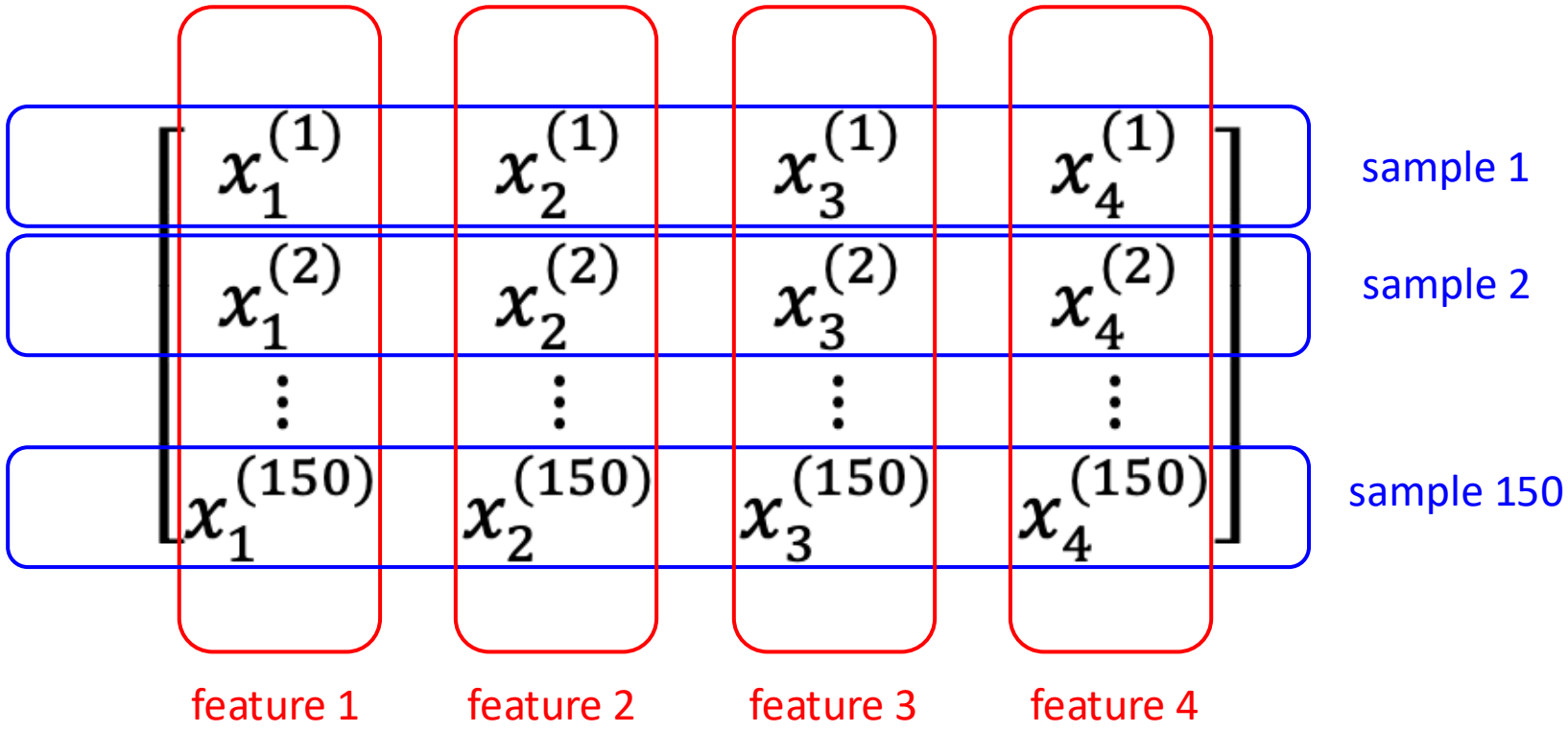
The Iris DataSet

- **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

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



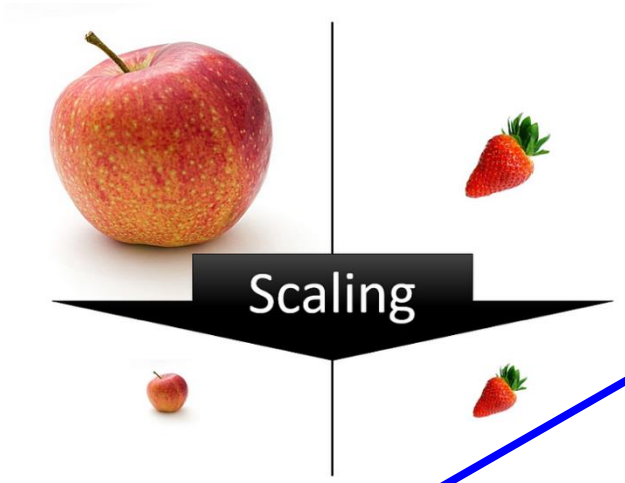
Terminology

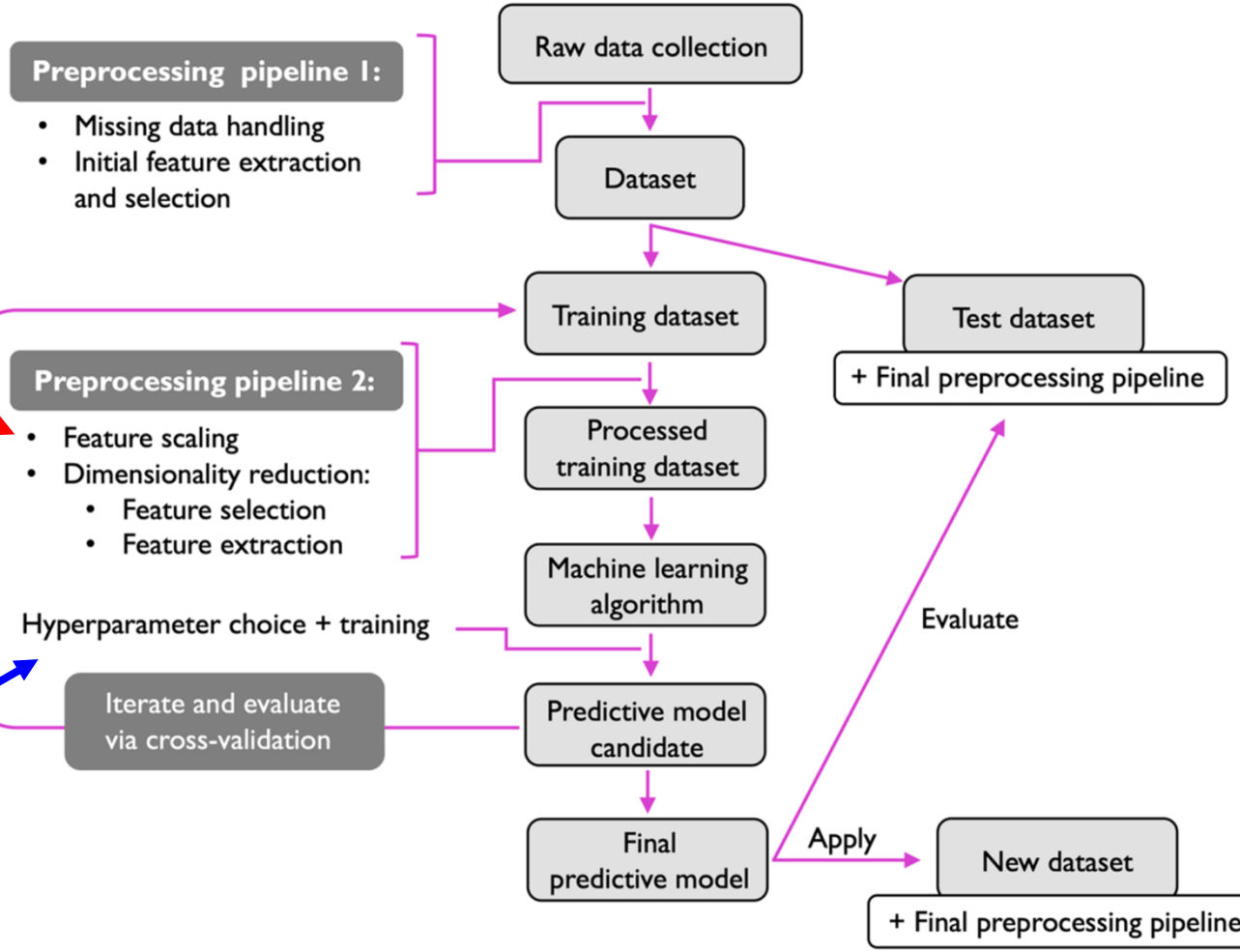
- **Training example**: a row in the data matrix, also known as an observation, record, instance, or sample
- **Feature**: a column in the data matrix, also known as predictor, variable, input, attribute
- **Target**: also known as class label, ground truth, outcome, output, etc.
- **Loss function**: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

Machine learning 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.

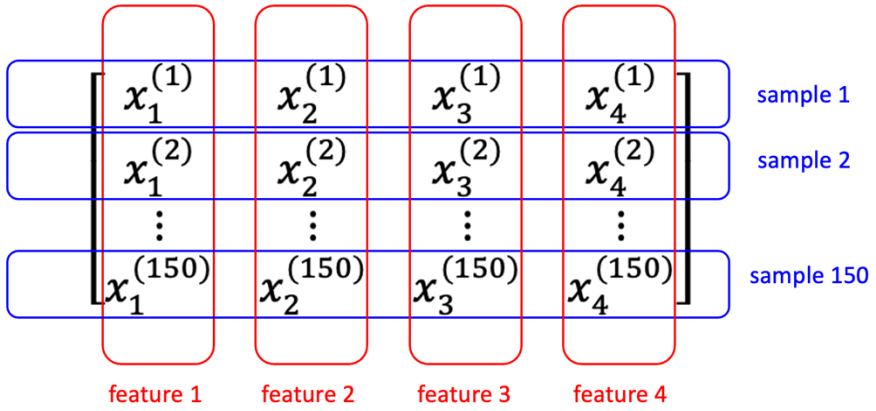


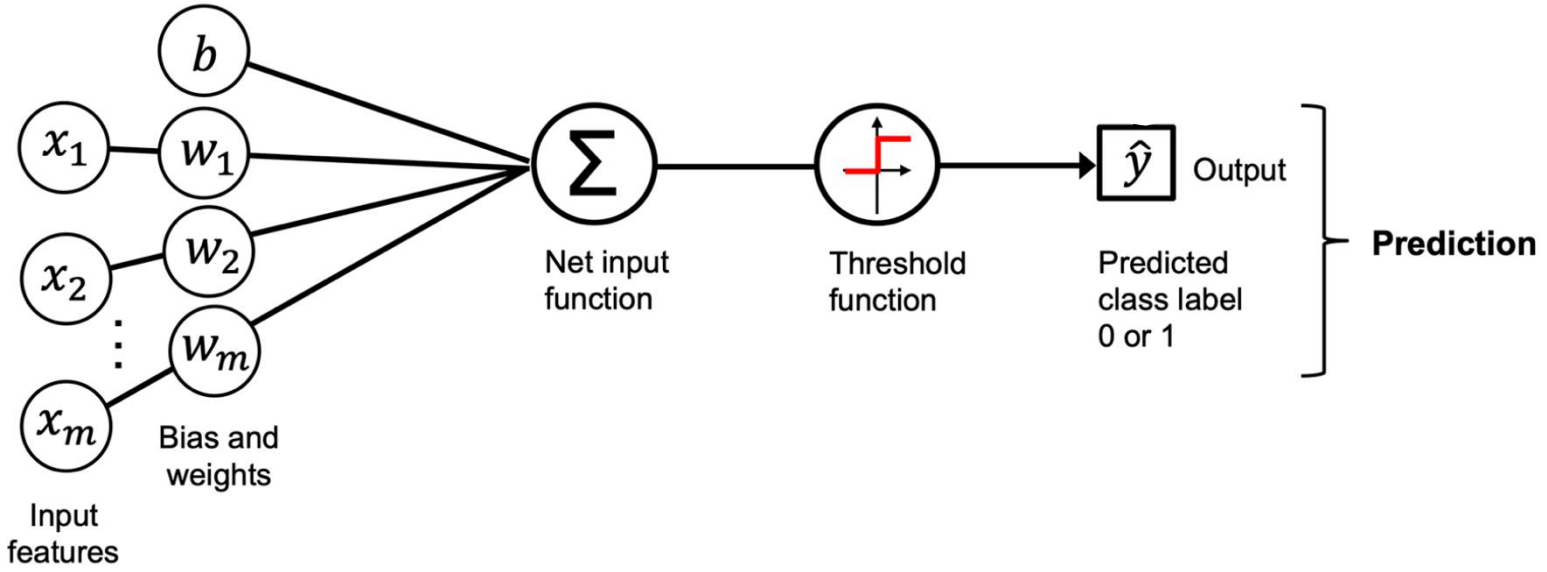


Hyperparameters: learning rate, batch size, number of neurons per layer, dropout, etc.

McCulloch-Pitts (MCP) neuron model

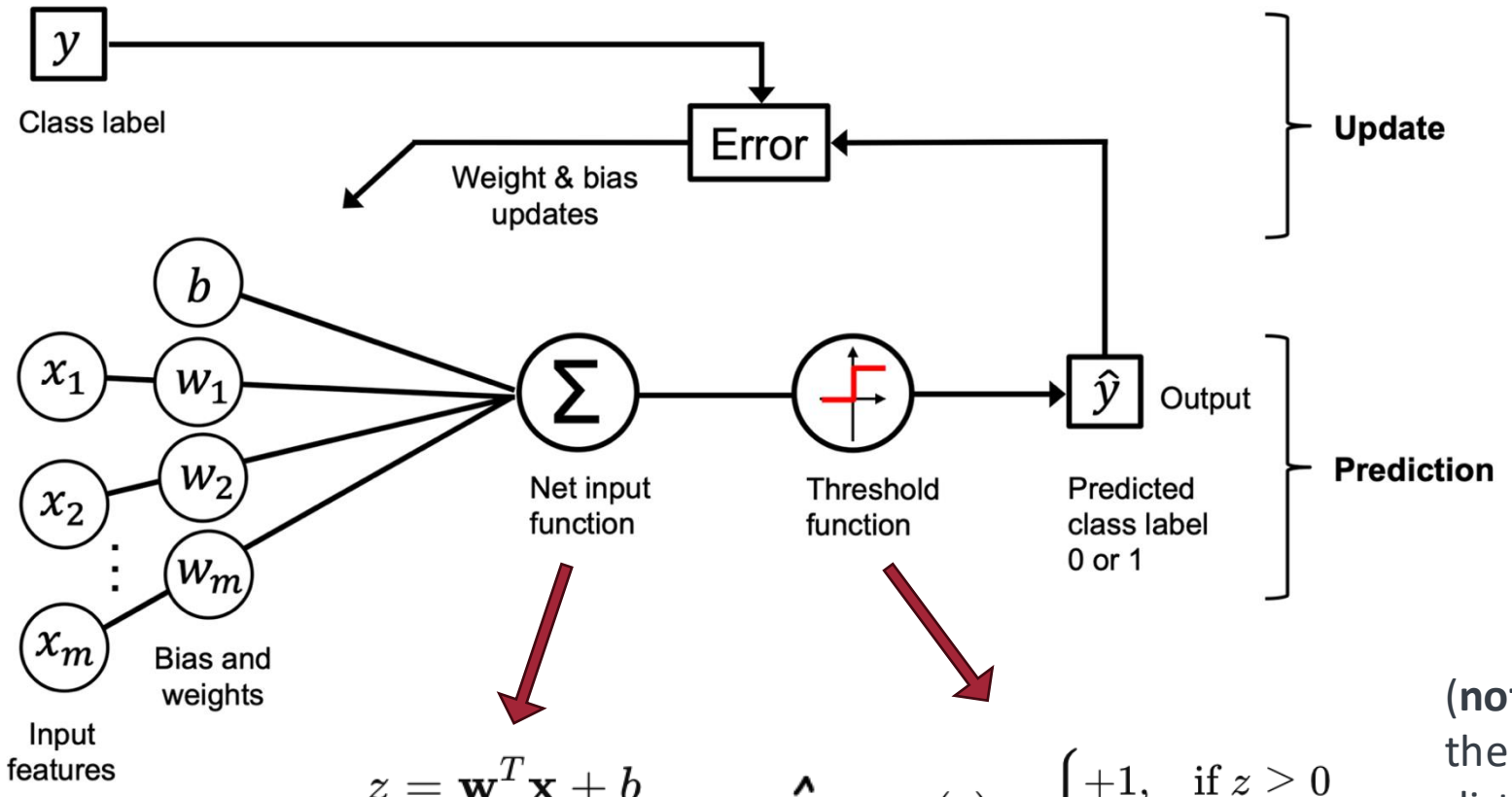
- **Pre-determined weights, no learning capability**





Rosenblatt's perceptron model

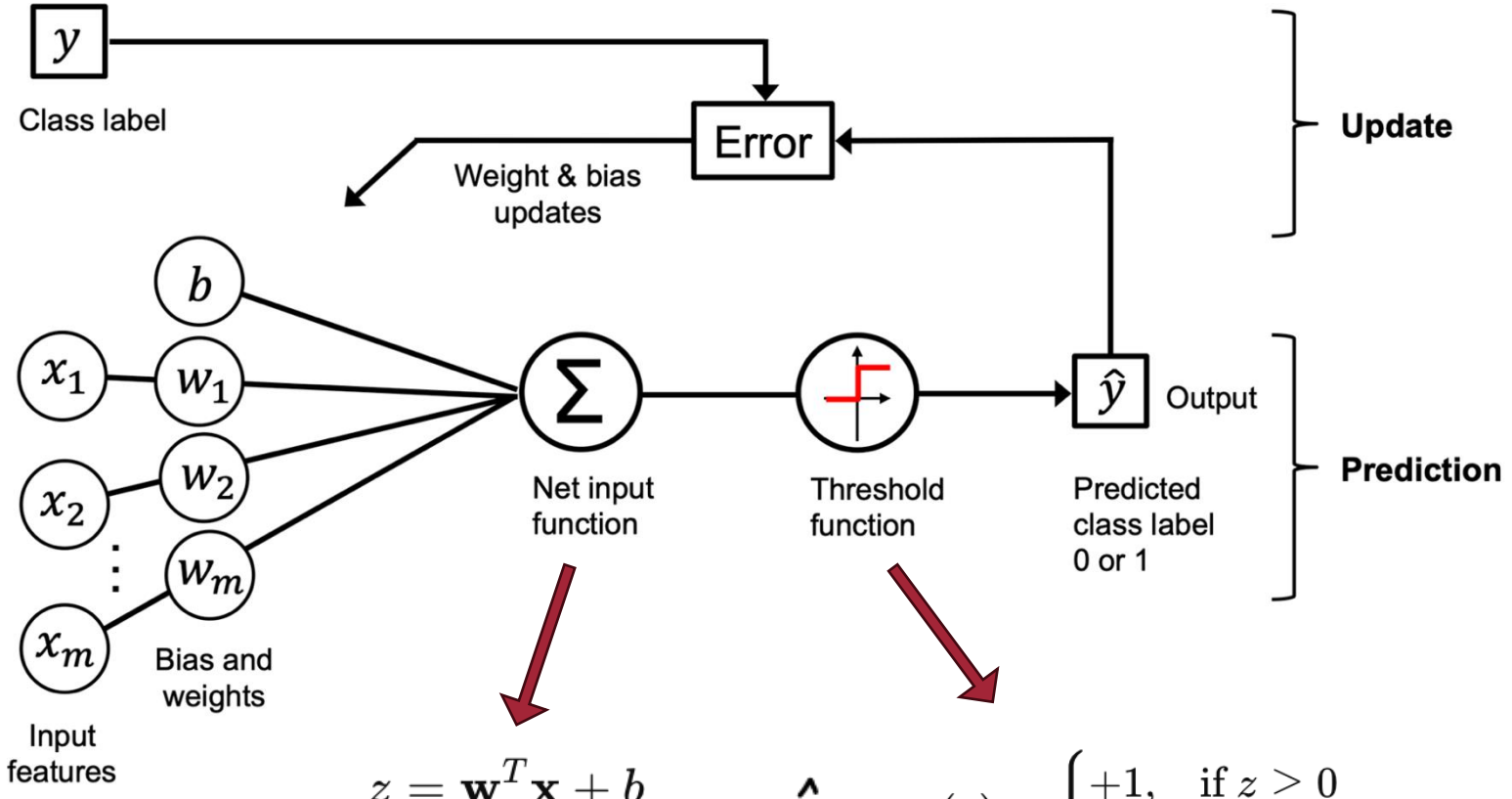
- Proposed an algorithm that would automatically **learn the optimal weight** coefficients



(**notation**: add a "hat" for the predicted value $\hat{y}$ to distinguish from actual label y)

Key idea to adjust the weight (and bias)

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



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:

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

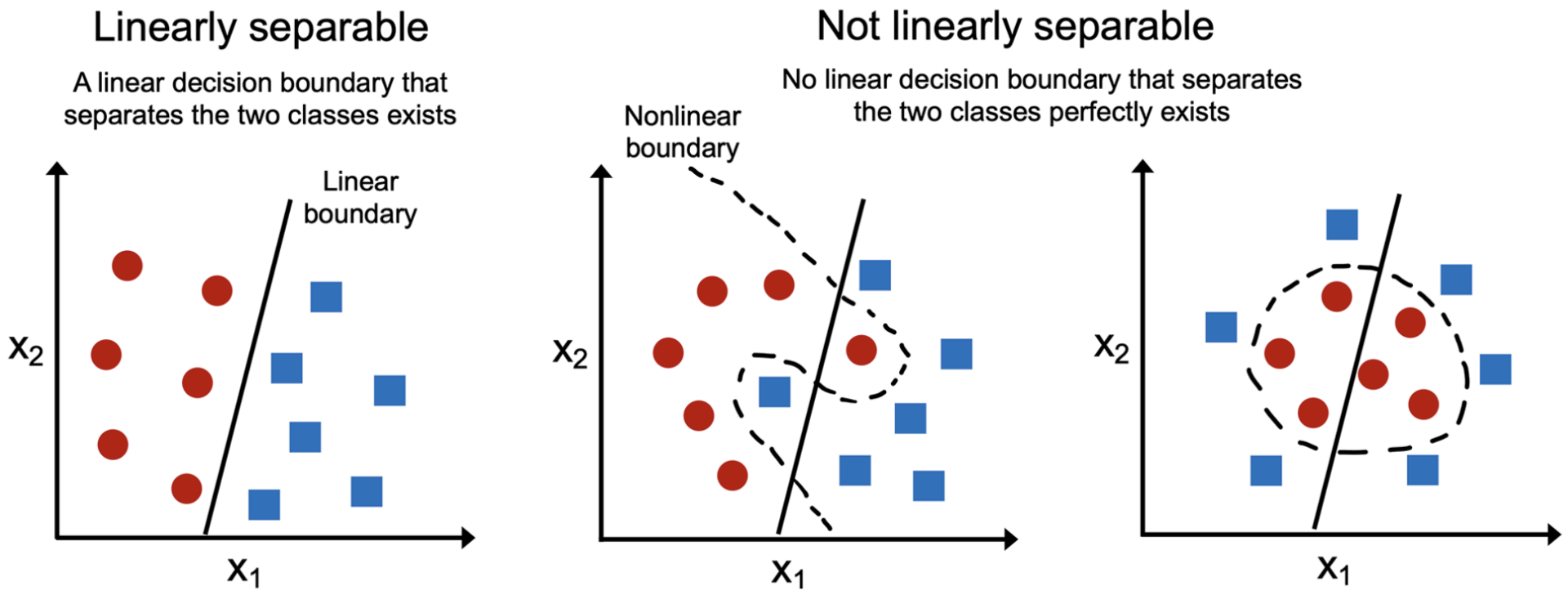
learning rate

actual class label

predicted class label

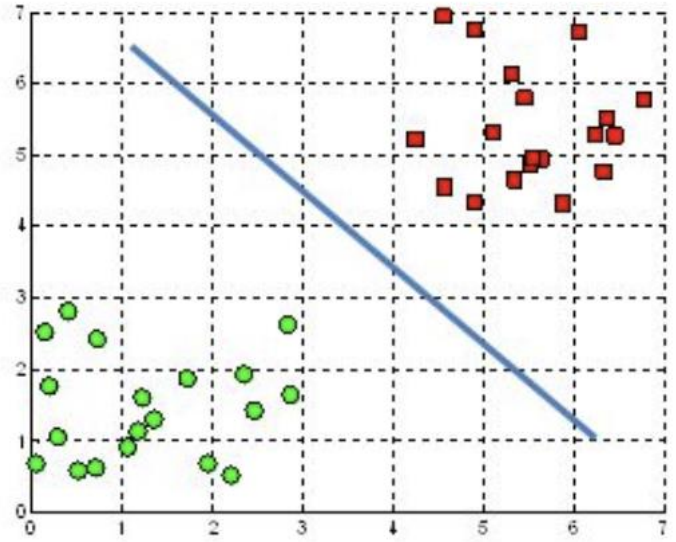
Applicable to linearly separable data only

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

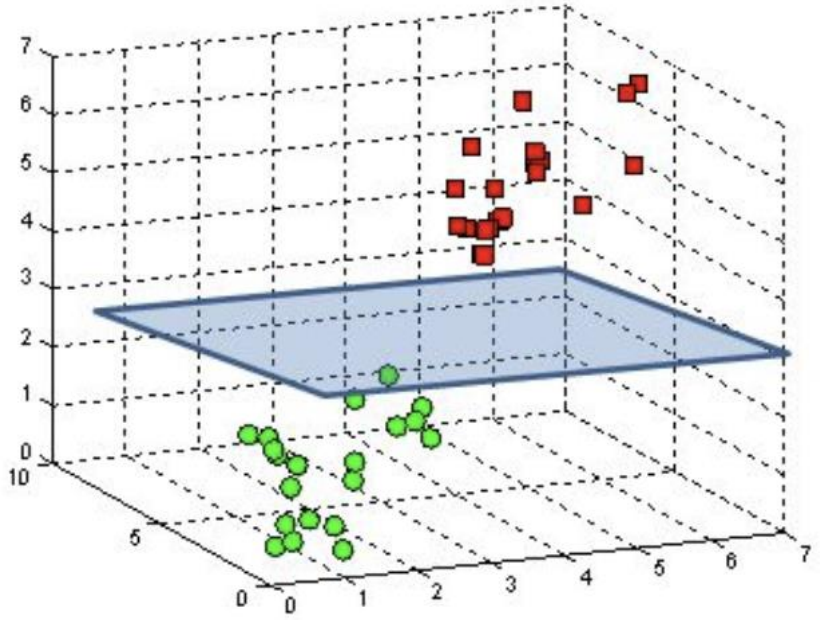


Applicable to linearly separable data only

A hyperplane in $\mathbb{R}^2$ is a line



A hyperplane in $\mathbb{R}^3$ is a plane

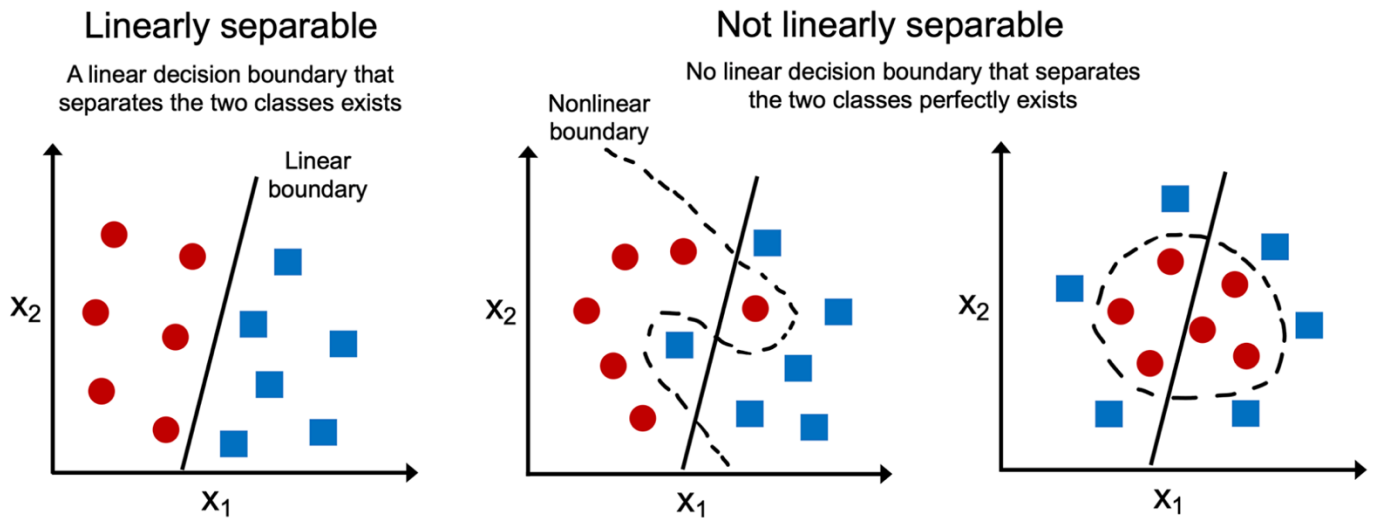


With more than two features, the boundary will be "**hyperplane**"

Perceptron convergence theorem

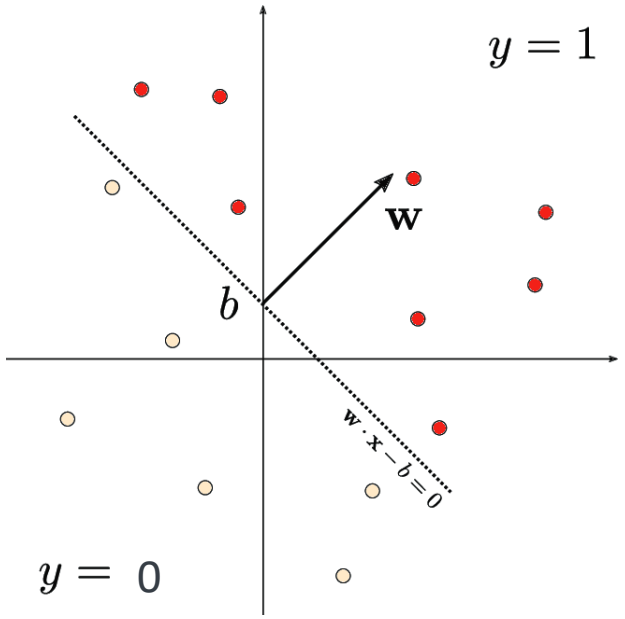
- Rosenblatt proved mathematically that the perceptron learning rule **converges** if the two classes can be **separated by a linear hyperplane.**

- 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.



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

The bias b is absorbed in the dot product notation

$$\mathbf{x}_i \text{ becomes } \begin{bmatrix} \mathbf{x}_i \\ 1 \end{bmatrix}$$

$$\mathbf{w} \text{ becomes } \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix}$$

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

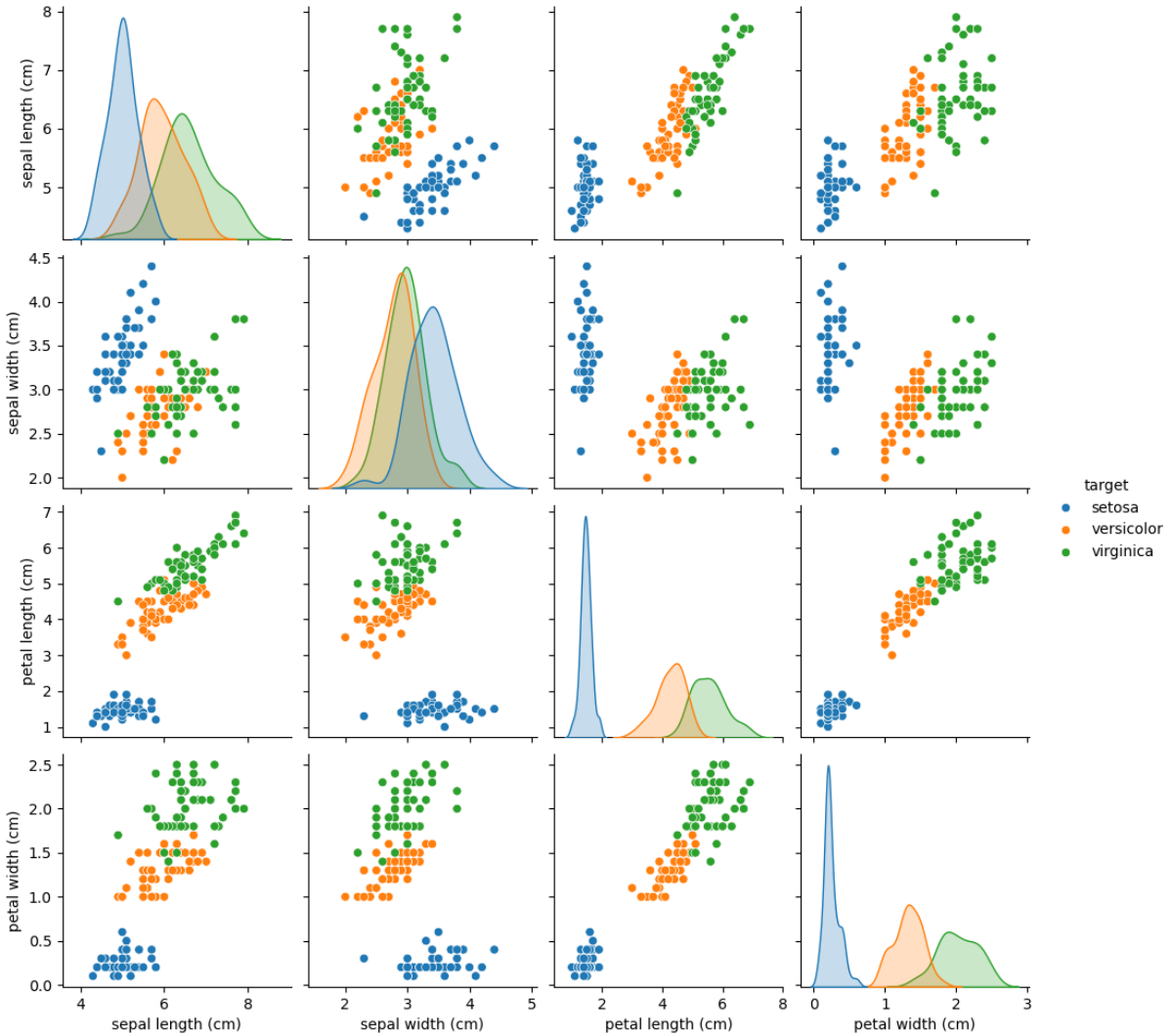
So this needs to be 0 at the boundary, and it is zero at 90°

When x corresponds to a data in one class, the angle shall be smaller than 90 degrees
When x corresponds to a data in the other class, the angle shall be larger than 90 degrees

Demo: Iris flowers classification

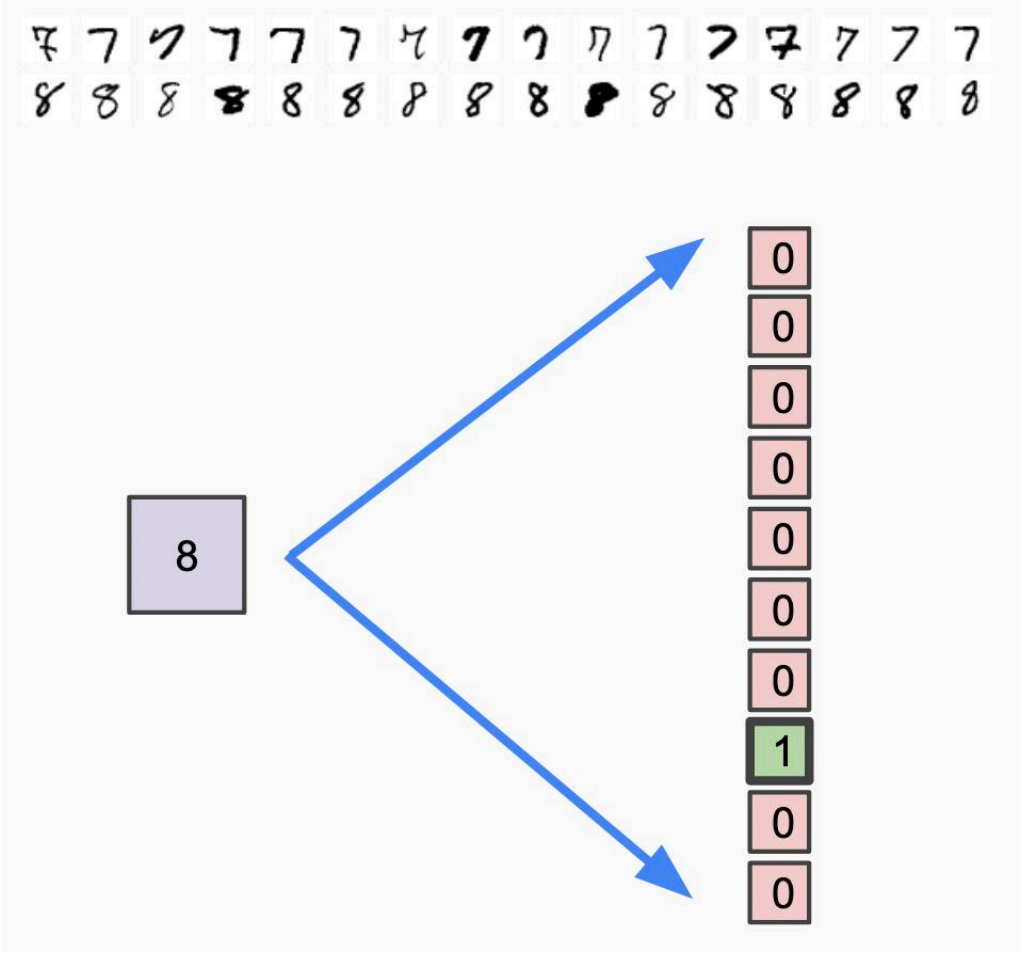
- See jupyter notebook: **demo_Iris.ipynb**

- 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 the 3rd class** before we apply Rosenblatt Perceptron model



One-hot encoding

- The Iris flower dataset has 3 labels: Setosa, Versicolor, Virginica.
- We can encode them with 3 neurons in the output layer: [1, 0, 0], [0, 1, 0], [0, 0, 1]
- **One-hot encoding**: only one neuron will be "on" and all the others will be "off".



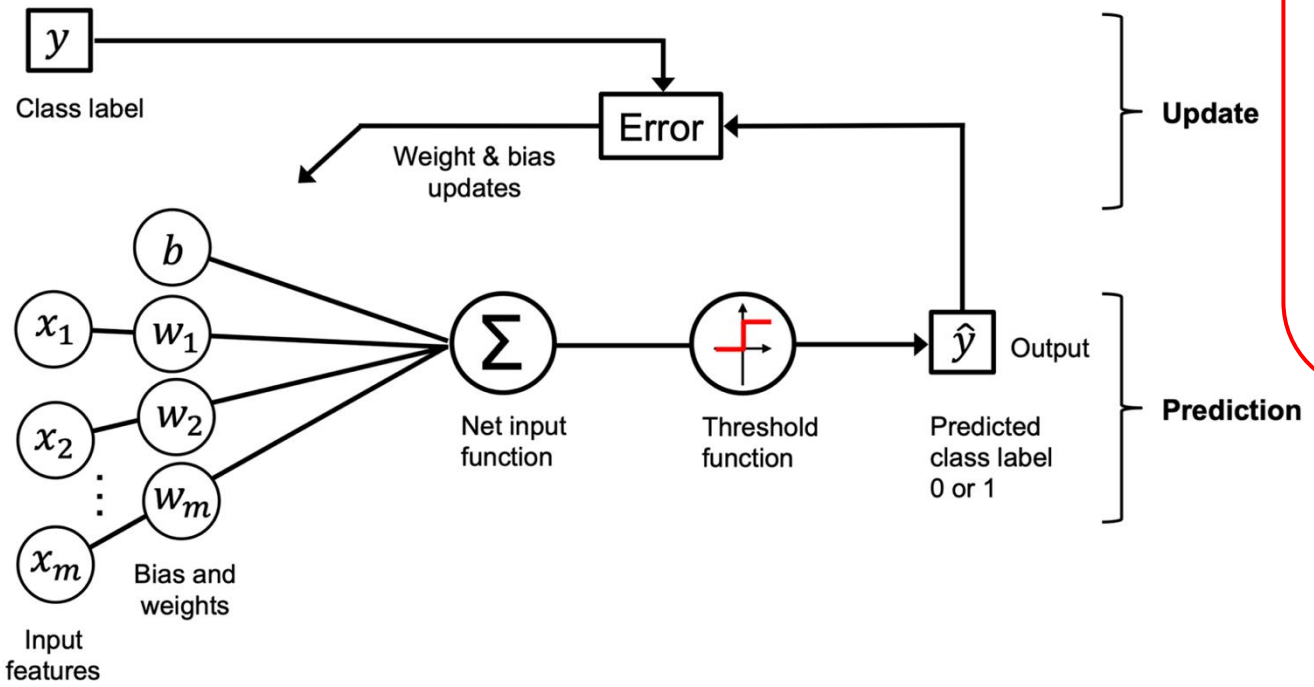
For **MNIST handwritten digits**, we have 10 possible labels (0 through 9).

We will have **10** neurons in the output layer.

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 150



$$w_j := w_j + \Delta w_j$$

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

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

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

learning rate

predicted class label

actual class label

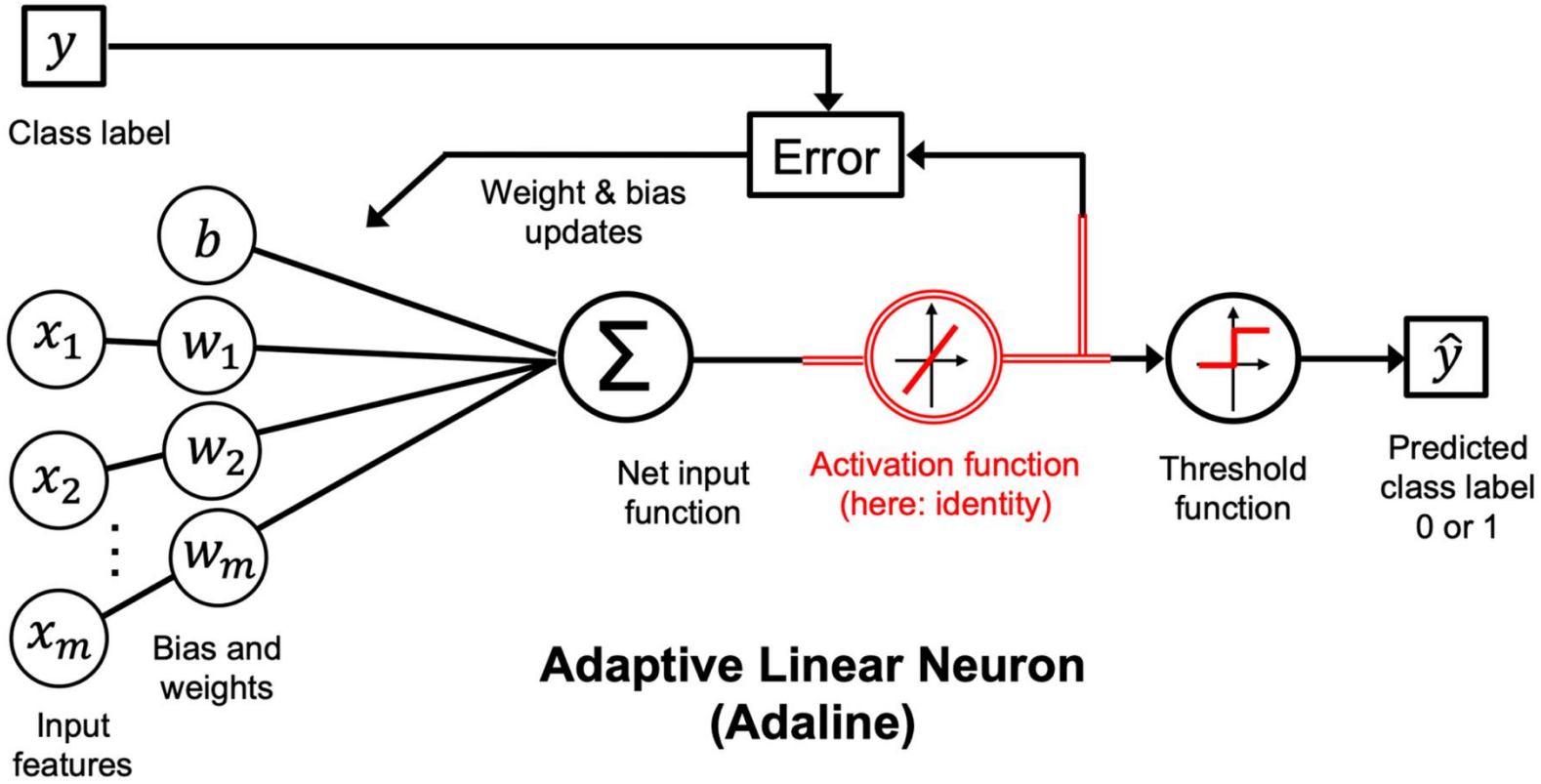
j: feature index

i: sample index

m=4 (features)

Adaptive linear neuron (Adaline)

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



Key improvement from the perceptron model

Learning

- In the Adaline rule, the activation function is just the identity $\sigma(z) = z$
- We compute the **error** between predicted label (weighted sum, may not be 0 or 1) and the true label
- The weights are updated based on the total error for **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 with full-batch gradient descent

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

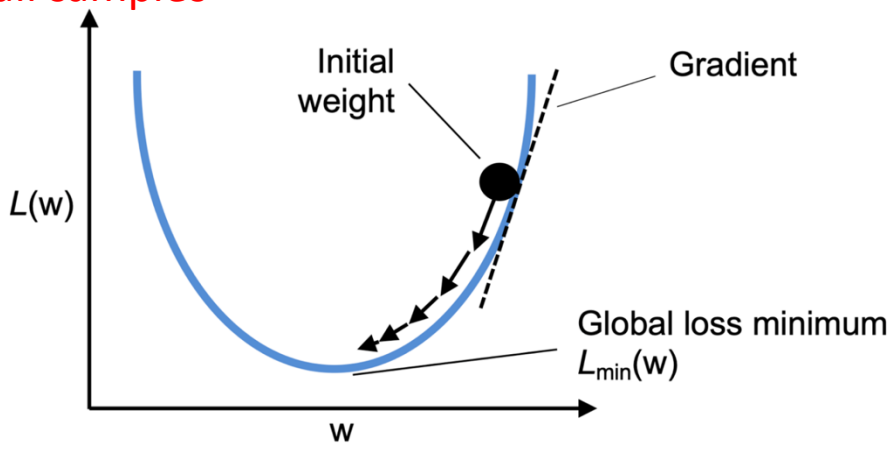
- **Mean squared errors (MSE)**

weighted sum of i-th sample

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

Sum over all samples

actual class label



- **Adaline learning or updating rule**

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

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

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

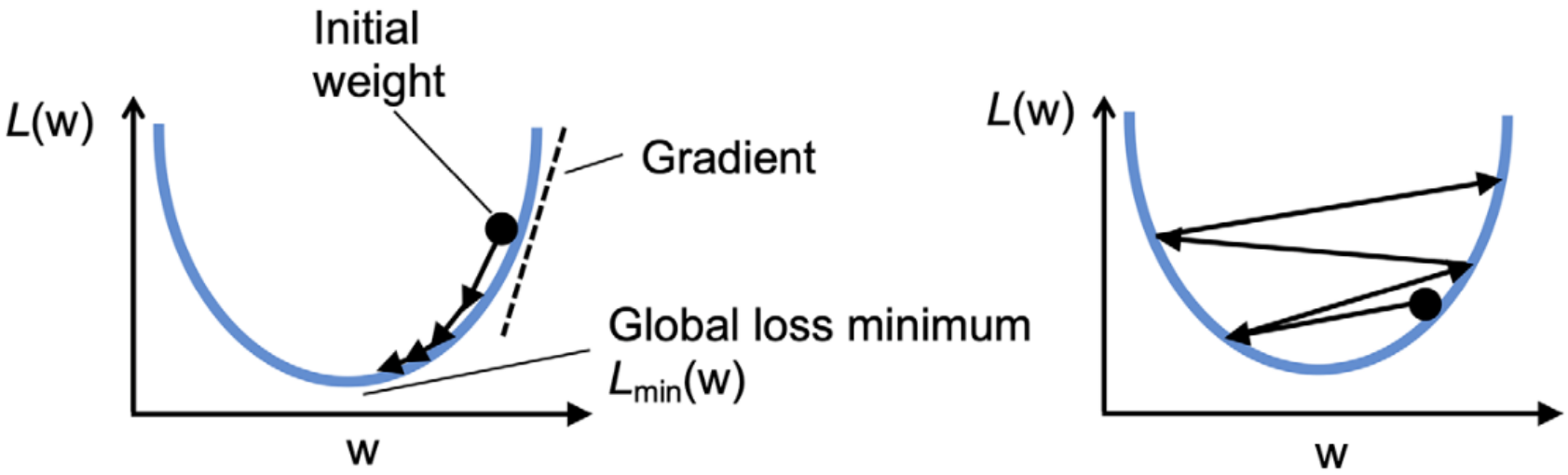
$$\frac{\partial L}{\partial b} = -\frac{2}{n}\sum_{i}\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)$$

Advantages of this MSE loss function

i) Differentiable

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

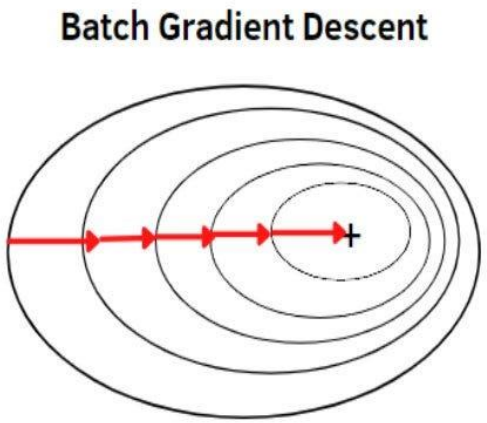
Learning rate

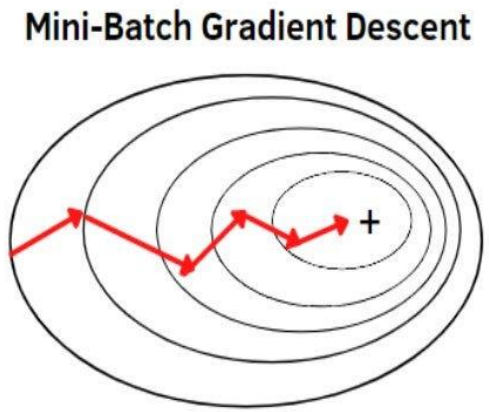


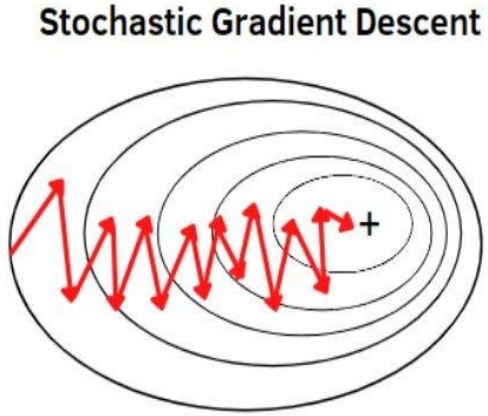
- If we choose a learning rate that is too large --- we **overshoot** the global minimum
- If it is too small --- training will be slow and might get stuck in local minima (for other complex loss functions). However, MSE loss function is convex and there are no local minima.

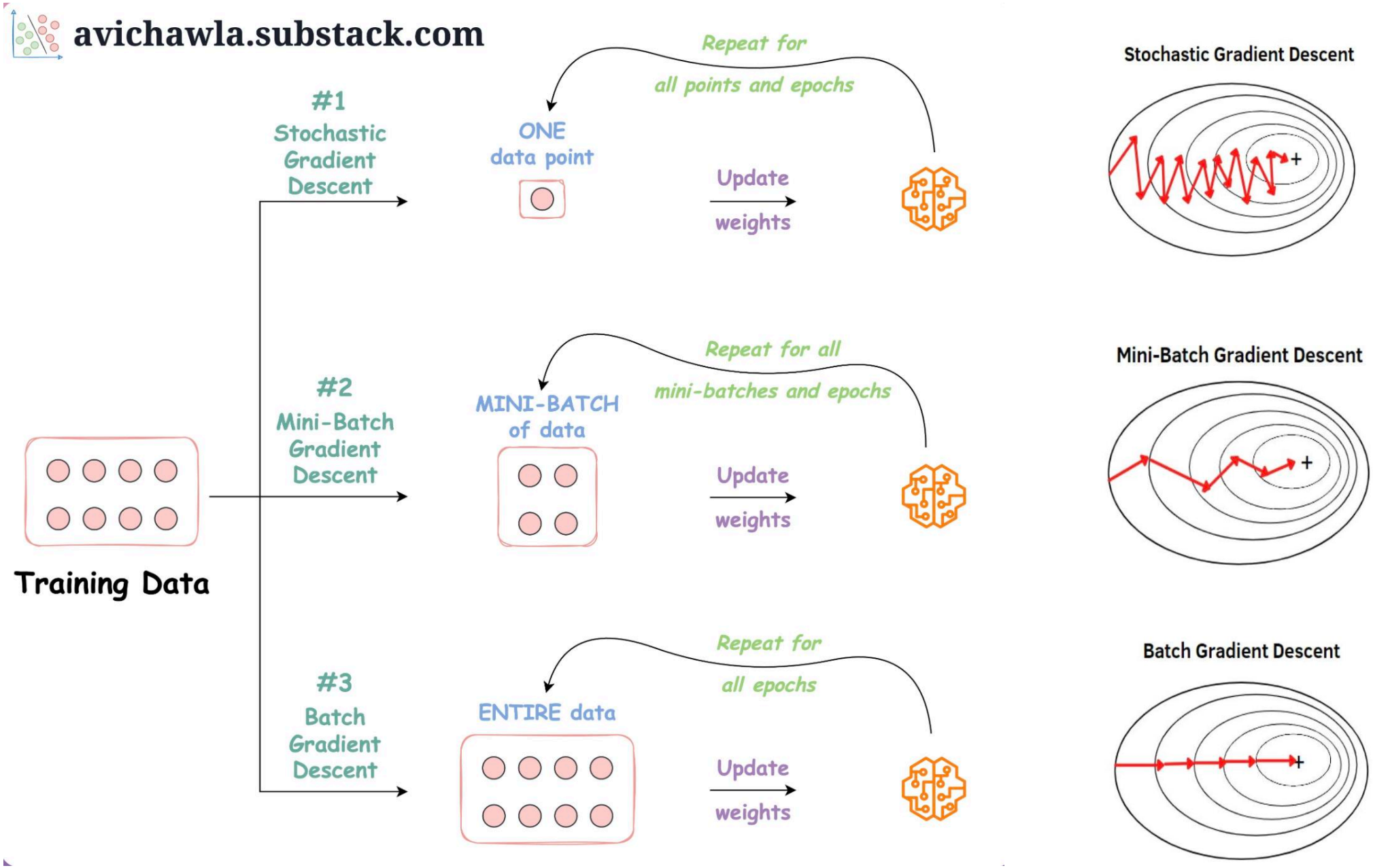
Gradient Descent

- **Batch gradient descent**: use the entire dataset to compute the gradient before taking a step to toward the minimum
- **Mini-batch gradient descent**: use a small subset to approximate the gradient. So each step is nosier, but more frequent, offering a **good balance between stability and speed**
- **Stochastic gradient descent**: update after each individual sample. It introduces more noise but can help escape local minima








avichawla.substack.com
#1
Stochastic Gradient Descent
Repeat for all points and epochs
ONE data point
Update weights
Stochastic Gradient Descent
#2
Mini-Batch Gradient Descent
Repeat for all mini-batches and epochs
MINI-BATCH of data
Update weights
Mini-Batch Gradient Descent
Training Data
#3
Batch Gradient Descent
Repeat for all epochs
ENTIRE data
Update weights
Batch Gradient Descent

More on 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.

Feature scaling

- Many ML algorithms require feature scaling for optimal performance
- Gradient descent is one of the them that benefit from feature scaling.

- **Standardization**

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

- **Normalization**

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

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

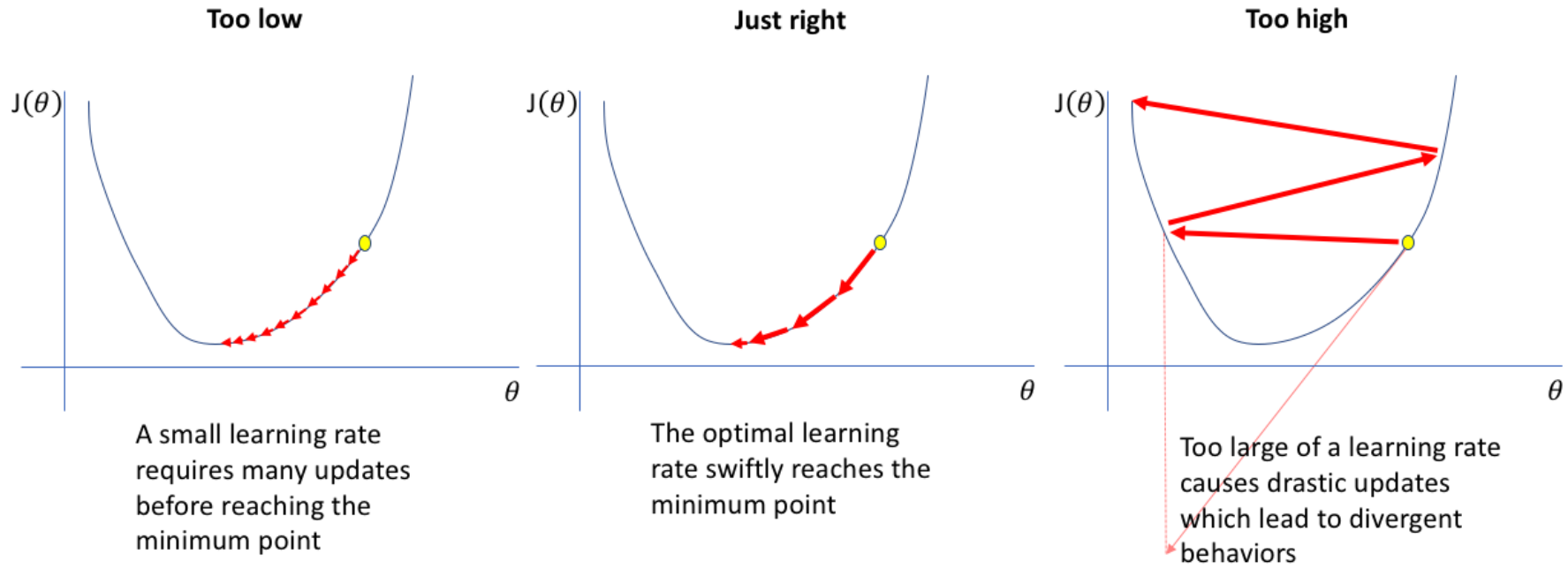
- After feature scaling, it is **easier to find a learning rate that works well for all weights** (and bias).

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.



Python demos

- See Jupyter notebook

Assignments

- **In-class assignment:**

- **After-class assignment:** will be posted on Github