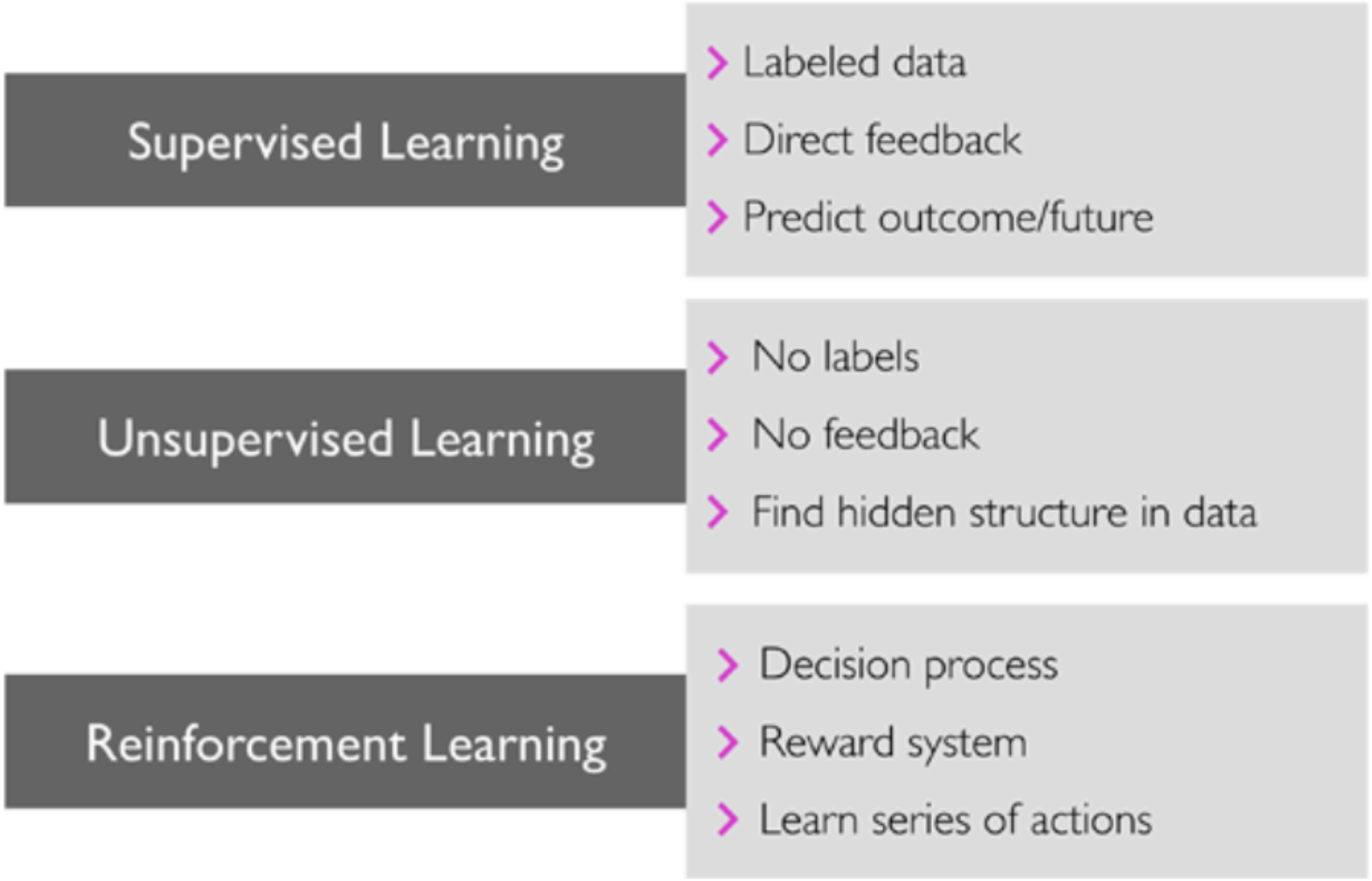
Lecture 1
Introduction

Wonjun Lee

Contents

- Giving Computers the Ability to Learn from Data
- Artificial Neurons
- Implementing a Perceptron Learning Algorithm in Python
- Adaptive Linear Neurons and the Convergence of Learning

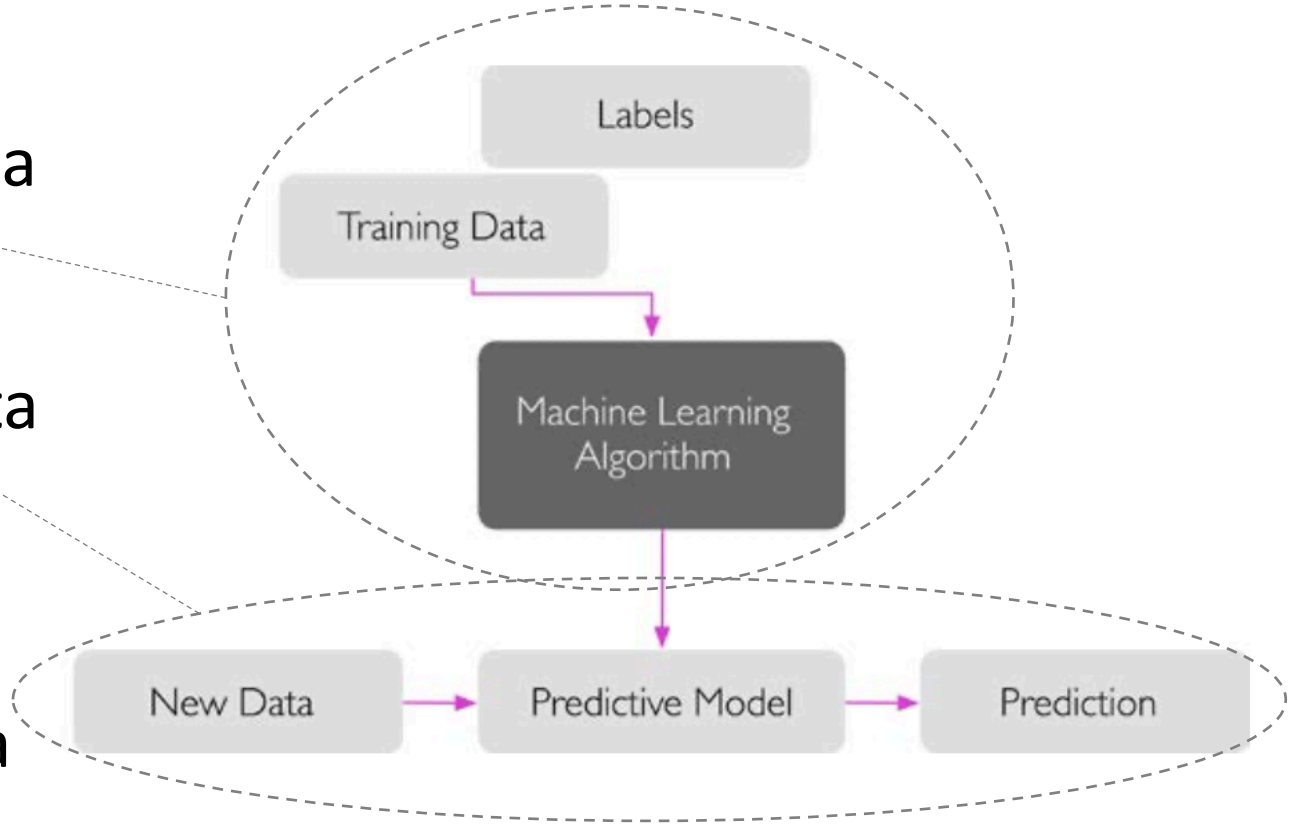
Three Types of Machine Learning (ML)



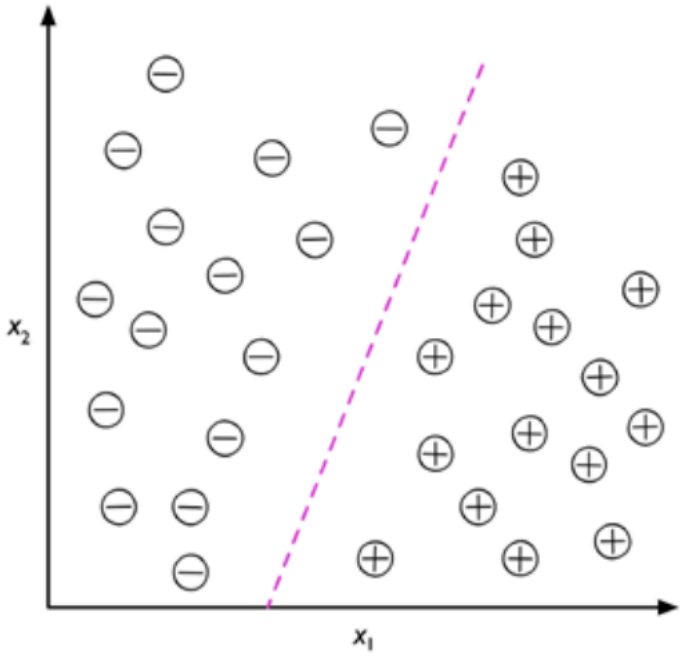
Supervised Learning

- Supervised – a set of training examples
  - ✓Output signals (labels) are known
- Labeled training data is passed to a ML algorithm
- Predictive model makes a predictions on new, unlabeled data inputs

1) Classification – discrete class labels
2) Regression – outcome signal is a continuous value



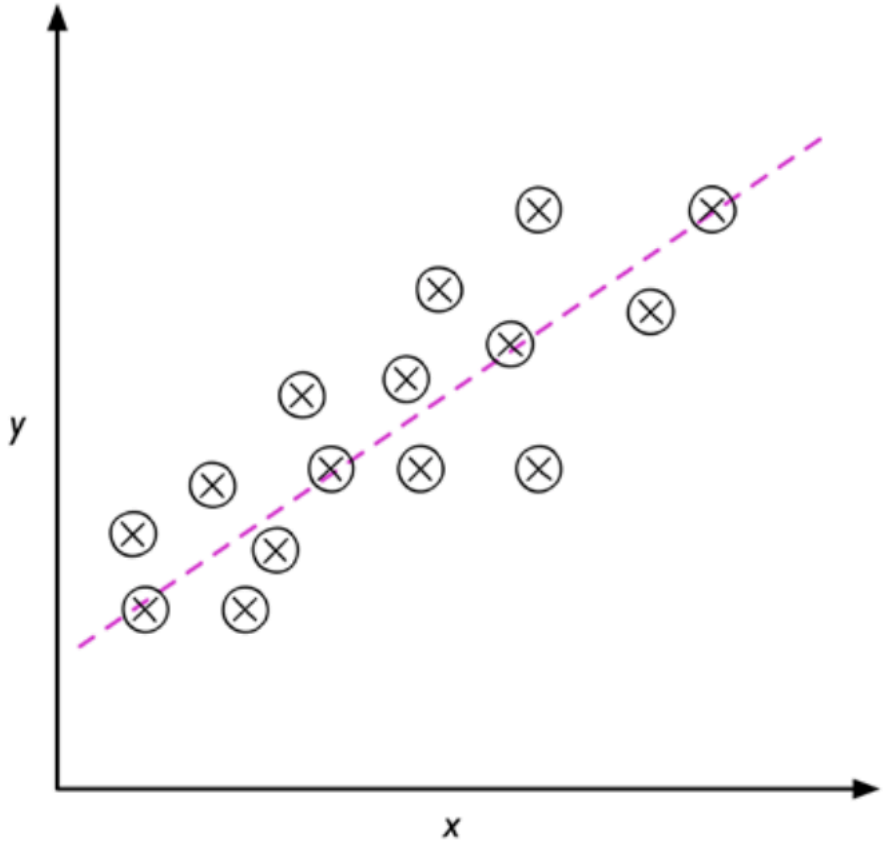
1) Classification for Predicting Class Labels



A binary classification task given 30 training examples

- Predict categorical class labels of new instance
  - ✓Based on past observations
  - ✓Class labels – discrete, unordered values
- Ex) Spam filter – distinguishes between spam and non-spam emails
- 30 training examples – 15 negative class (-), 15 positive class (+) → Two dimensional : x_1 and x_2
- Dashed line – decision boundary
- Multiclass classification – ex) handwritten character recognition

2) Regression for Predicting Continuous Outcomes

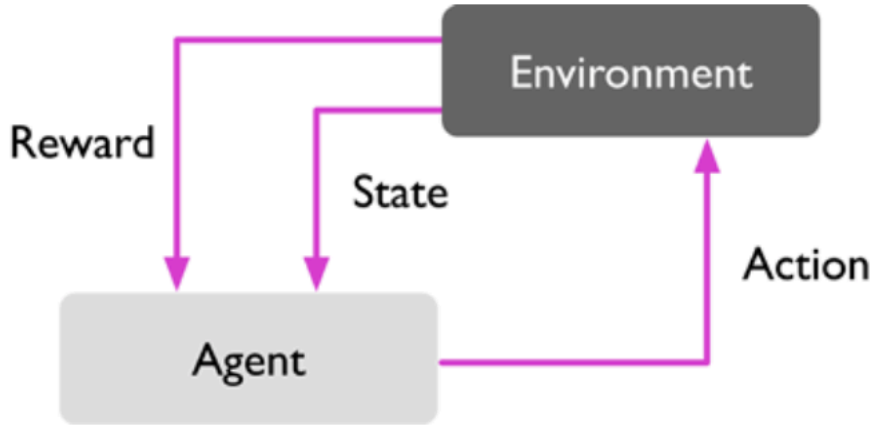


x – a feature variable
y – target variable
Minimize distance between data and line

- Regression analysis – prediction of continuous outcomes
  - ✓Predictor variables – explanatory → **features**
  - ✓Response variables – outcome → **target variables**
  - ✓Goal – Find relationship between predictor and response variables
- Ex) SAT score prediction : study time → score

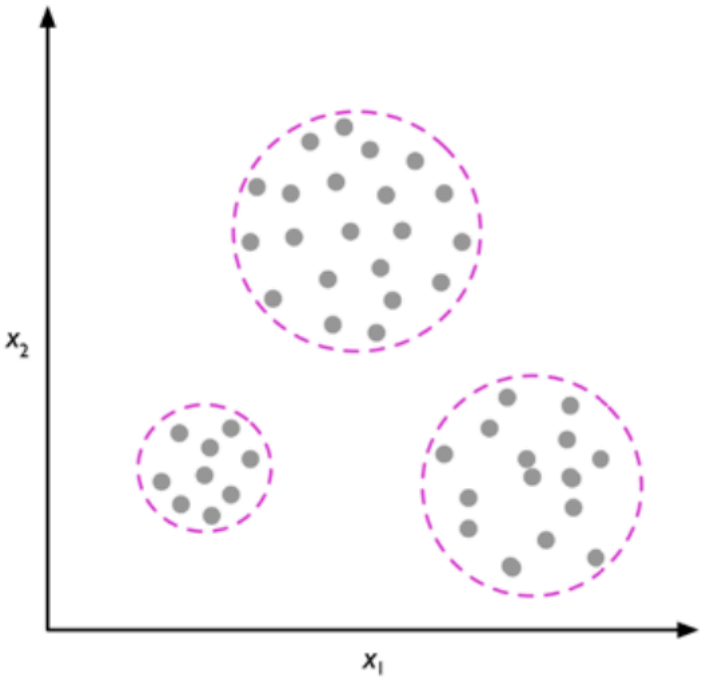
Solving Interactive Problems with Reinforcement Learning

- Reinforcement Learning
  - ✓Goal – develop a system (agent) that improves its performance interacting with environment
  - ✓Reward signal – not the correct ground truth
  - ✓Through interaction with environment – an agent learns a series of action that maximizes the reward via trial-and-error approach
- Ex) Chess engine
  - ✓Agent decides moves depending on state of environment and
  - ✓Rewards can be defined as **win** or **lose** at the end



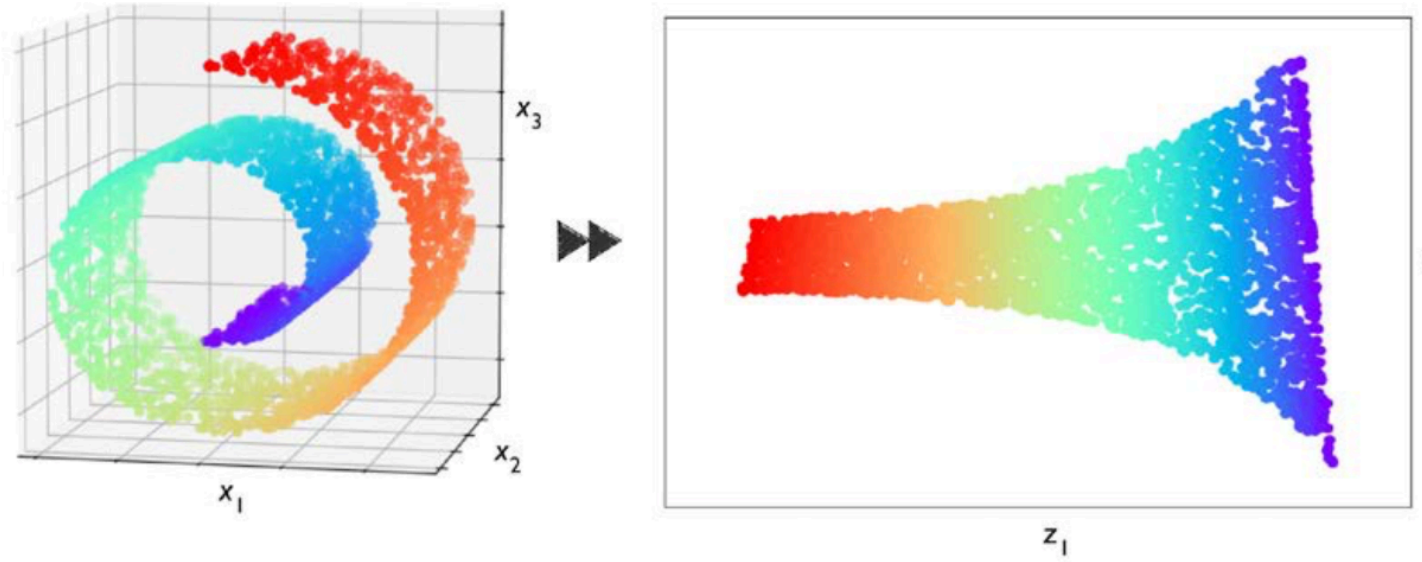
Discovering Hidden Structures with Unsupervised Learning

- Extract meaningful information without guidance of a known outcome variable or reward function
- Finding subgroups with clustering
  - ✓Clustering – organize a pile of information into meaningful subgroups without prior knowledge
  - ✓Each cluster defines a group of objects that share a certain degree of similarity but are more dissimilar to objects in other clusters → unsupervised classification



organizing unlabeled data into three distinct groups based on the similarity of their features, x_1 and x_2

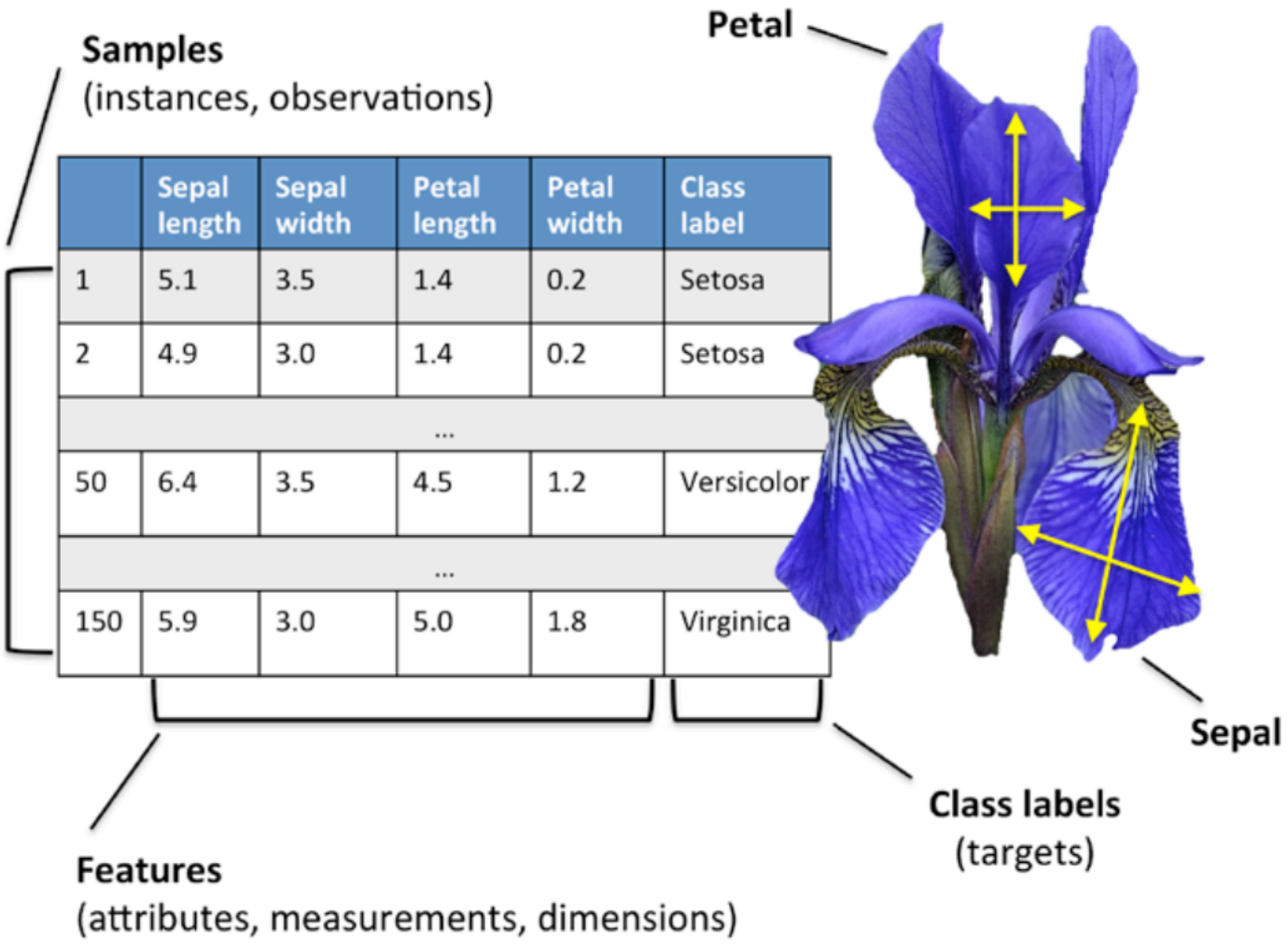
Dimensionality Reduction for Data Compression



nonlinear dimensionality reduction was applied to compress a 3D Swiss Roll onto a new 2D feature subspace

- High dimensionality – challenge for limited storage space + computational performance of ML algorithms
- Unsupervised dimensionality reduction – commonly used approach in feature preprocessing to remove noise from data and compress data onto a smaller dimensional subspace while retaining most of relevant information
- Useful for visualizing data – high Dimension → 2D or 3D

Notation and Conventions



	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

Notation and Conventions

Iris dataset, consisting of 150 examples and four features, can then be written as a 150 × 4 matrix, $\boldsymbol{X} \in \mathbb{R}^{150\times4}$:

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

Each row represents one flower instance and can be written as a four-dimensional row vector, $\boldsymbol{X} \in \mathbb{R}^{1\times4}$:

$$\boldsymbol{x}^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix}$$

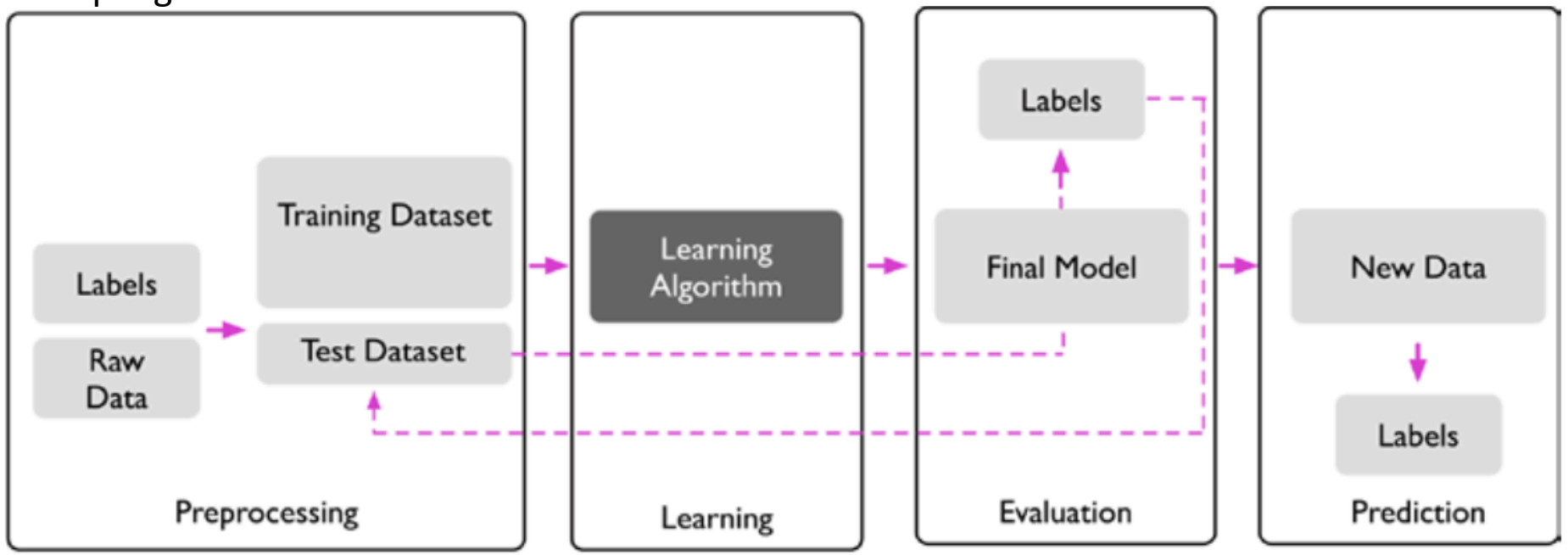
Each feature dimension is a 150-dimensional column vector $\boldsymbol{X} \in \mathbb{R}^{150\times1}$, Ex) $\boldsymbol{x_j} = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \dots \\ x_j^{(150)} \end{bmatrix}$

Target variables (here, class labels) as a 150-dimensional column vector :

$$\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(150)} \end{bmatrix} (\text{y} \in \{\text{Setosa, Versicolor, Virginica}\})$$

Roadmap for Building ML Systems

Feature Extraction and Scaling
Feature Selection
Dimensionality Reduction
Sampling



Model Selection
Cross-Validation
Performance Metrics
Hyperparameter Optimization

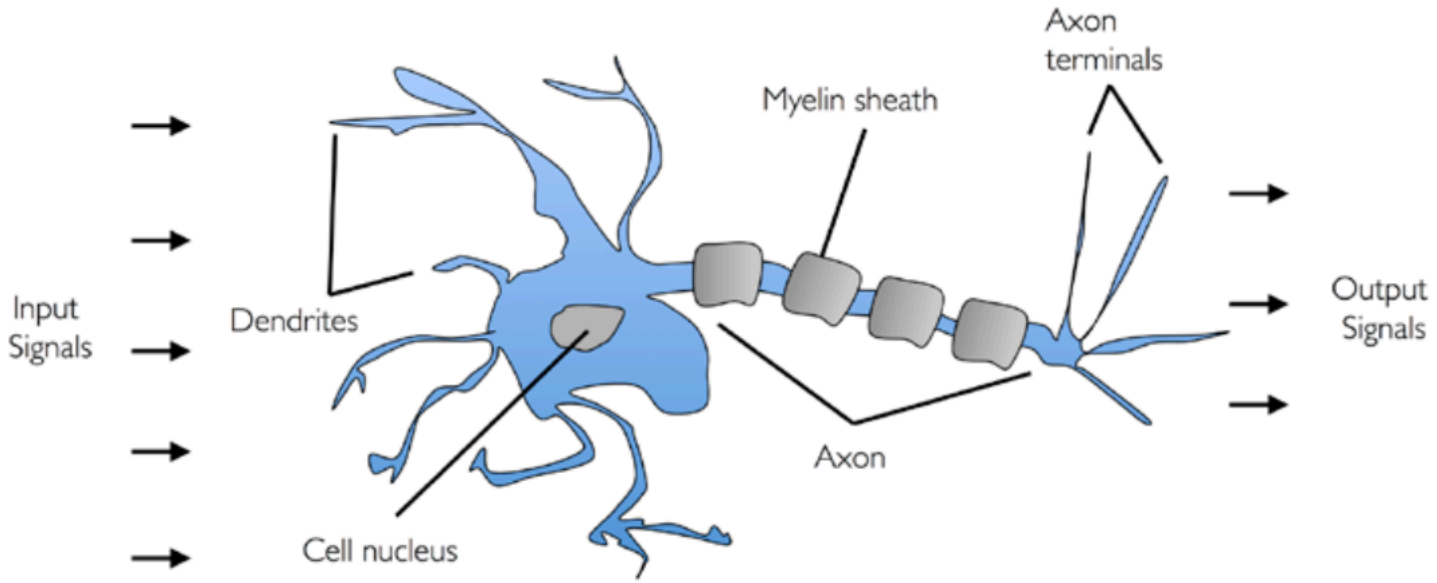
Training and Selecting a Predictive Model

- Each classification algorithm has its inherent biases, and no single classification model enjoys superiority
- Before we can compare different models, we first have to decide upon a metric to measure performance
  - ✓One commonly used metric is classification accuracy, which is defined as the proportion of correctly classified instances

Training Simple Machine Learning Algorithm for Classification

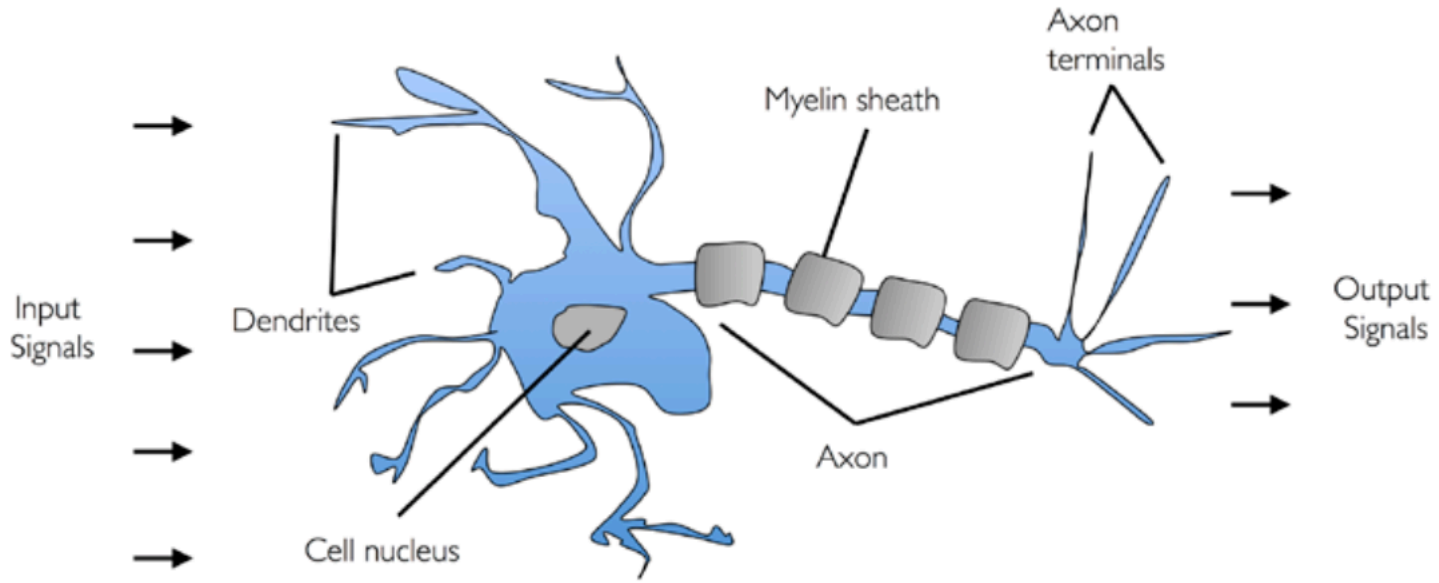
Artificial Neurons

- Biological neurons are interconnected nerve cells in the brain
- Nerve cells are involved in the processing and transmitting of chemical and electrical signals



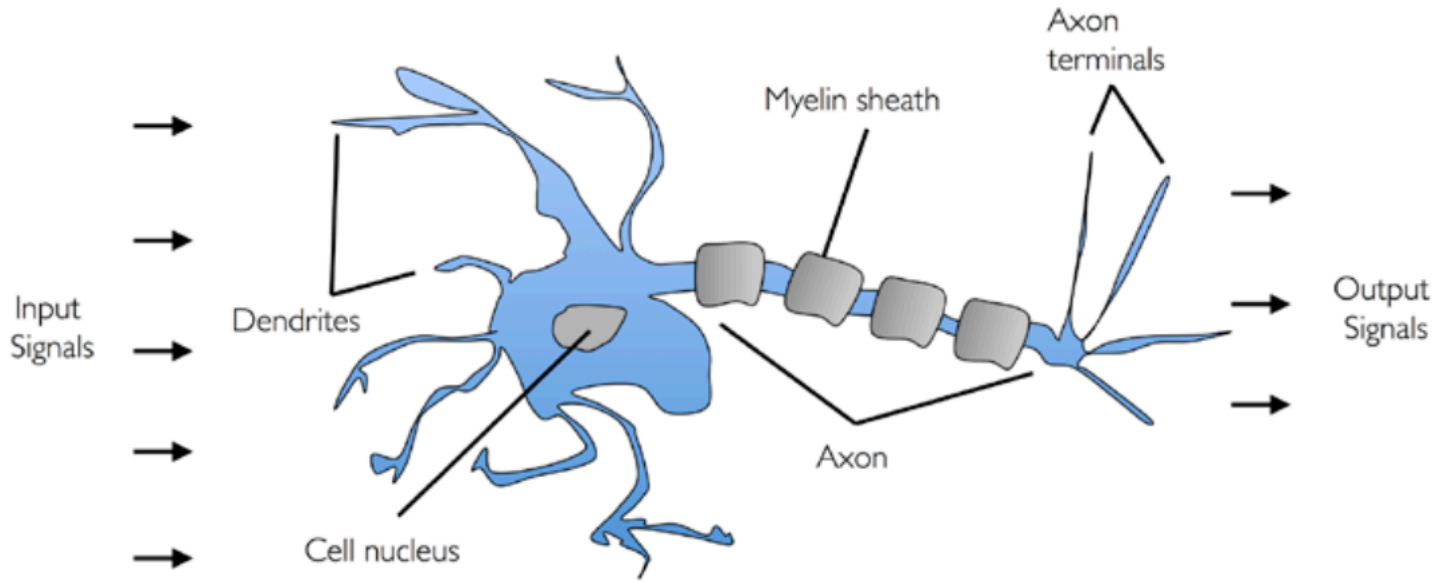
Artificial Neurons

- A simple logic gate with binary outputs
  - ✓Multiple signals arrive at the dendrites → they are then integrated into the cell body, and if the accumulated signal exceed a certain threshold, an output signal is generated that will be passed on by the axon



Artificial Neurons

- Rosenblatt proposed an algorithm
  - ✓Automatically learn the optimal weight coefficients that would then be multiplied with the input features in order to make the decision of whether a neuron transmits a signal or not



Formal Definition of an Artificial Neuron

- $\phi(z)$ – decision function, takes input (net input) $z = w_1x_1 + w_2x_2 + \cdots + w_mx_m$:

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

- If net input > threshold θ → 1, -1 otherwise
- Decision function is a variance of step function:

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta, \\ -1 & \text{otherwise.} \end{cases}$$

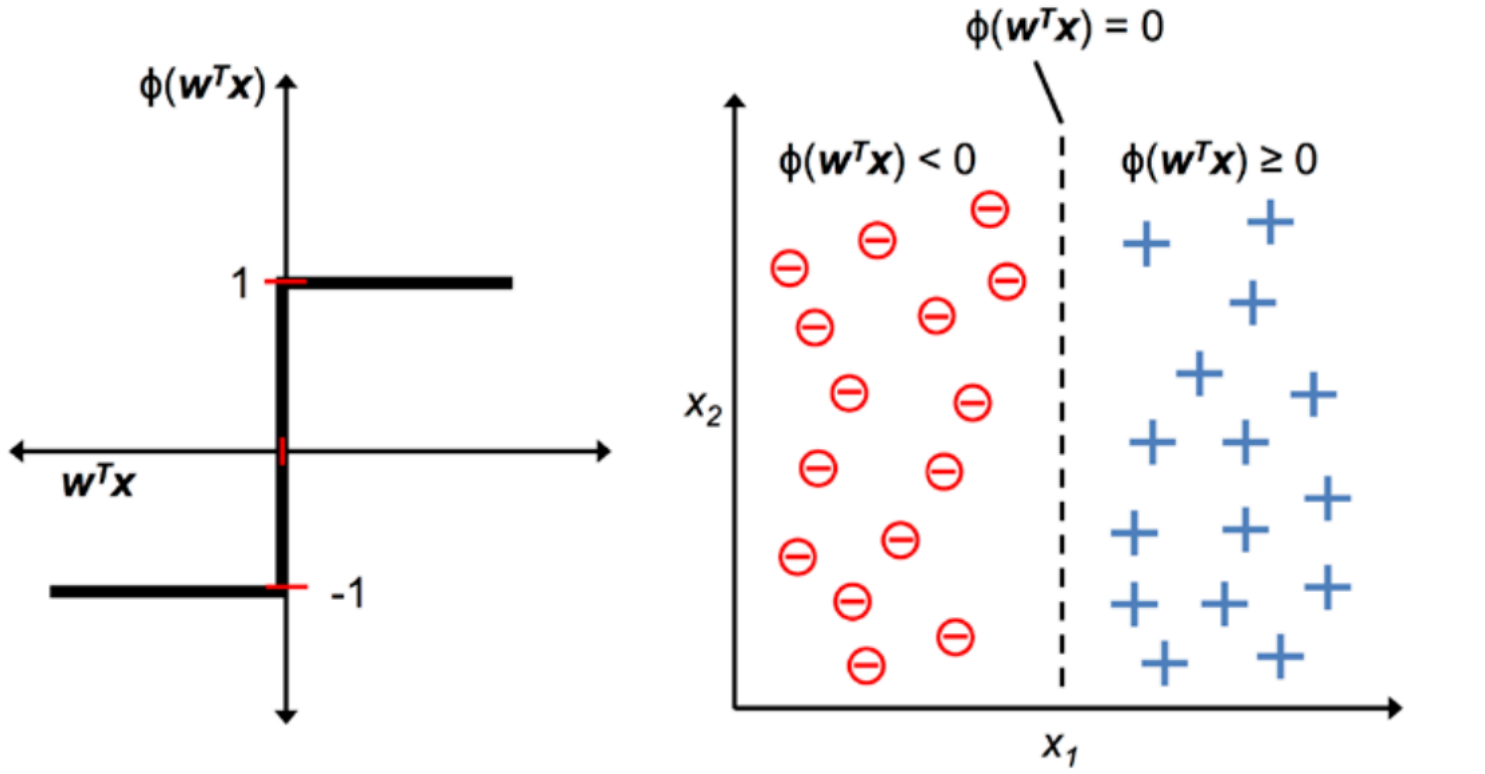
- Define a weight-zero as $w_0 = -\theta$ and $x_0 = 1$ →

$$z = w_0x_0 + w_1x_1 + \cdots + w_mx_m = \boldsymbol{w}^T\boldsymbol{x} \qquad \phi(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise.} \end{cases}$$

- Negative threshold, or $w_0 = -\theta$ called **bias** unit

Formal Definition of an Artificial Neuron

- Net input $z = \boldsymbol{w}^T\boldsymbol{x}$ is squashed into a binary output (–1 or 1) by the decision function of the perceptron



Perceptron Learning Rule

- Perceptron algorithm
  1. Initialize weights to small random numbers
  2. For each training example, $x^{(i)}$:
     a. Compute output value, $\hat{y}$
     b. Update the weights

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

- Update value = Δw_j is calculated by the perceptron learning rule:

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

  - ✓ η = learning rate (0 ~ 1)
  - ✓ $y^{(i)}$ = true class label (of *i*th training example)
  - ✓ $\hat{y}^{(i)}$ = predicted class label

- Weights in the weight vector are being updated simultaneously

Perceptron Learning Rule

$$\Delta w_0 = \eta\left(y^{(i)} - output^{(i)}\right)$$
$$\Delta w_1 = \eta\left(y^{(i)} - output^{(i)}\right)x_1^{(i)}$$
$$\Delta w_2 = \eta\left(y^{(i)} - output^{(i)}\right)x_2^{(i)}$$

- Correct prediction - weights remain unchanged, since update values are 0:

(1) $y^{(i)} = -1, \quad \hat{y}^{(i)} = -1, \quad \Delta w_j = \eta\left(-1 - (-1)\right)x_j^{(i)} = 0$

(2) $y^{(i)} = 1, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta(1 - 1)x_j^{(i)} = 0$

- Wrong prediction - weights are being pushed toward the direction of pos or neg class:

(3) $y^{(i)} = 1, \quad \hat{y}^{(i)} = -1, \quad \Delta w_j = \eta\left(1 - (-1)\right)x_j^{(i)} = \eta(2)x_j^{(i)}$ To the positive direction

(4) $y^{(i)} = -1, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta(-1 - 1)x_j^{(i)} = \eta(-2)x_j^{(i)}$ To the negative direction

Perceptron Learning Rule

- Ex 1) Assume $x_j^{(i)} = 0.5$ $\hat{y}^{(i)} = -1,$ $y^{(i)} = +1,$ $\eta = 1$

$w_j := w_j + \Delta w_j$

→ increase corresponding weight by 1 ($\Delta w_j = 1(1 - (-1))0.5 = (2)0.5 = 1$)

→ $x_j^{(i)} \times w_j$ would be more positive the next time

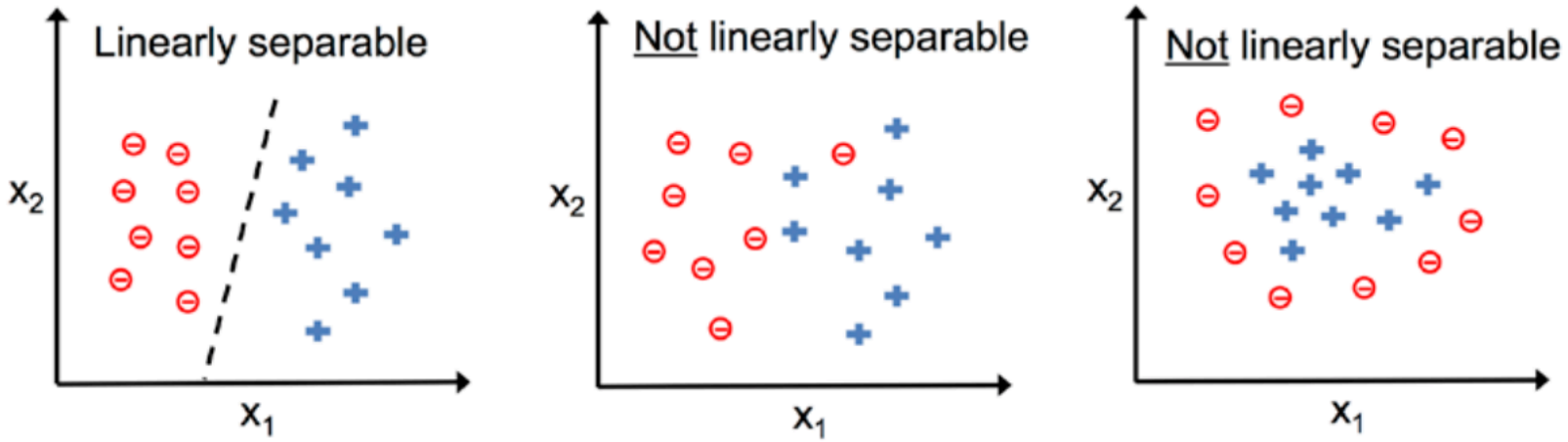
→ more likely to be above the threshold

- Ex 2) Assume $x_j^{(i)} = 4$

→ push decision boundary by an even larger extent ($\Delta w_j = 1(1 - (-1))2 = (2)2 = 4$) to classify correctly

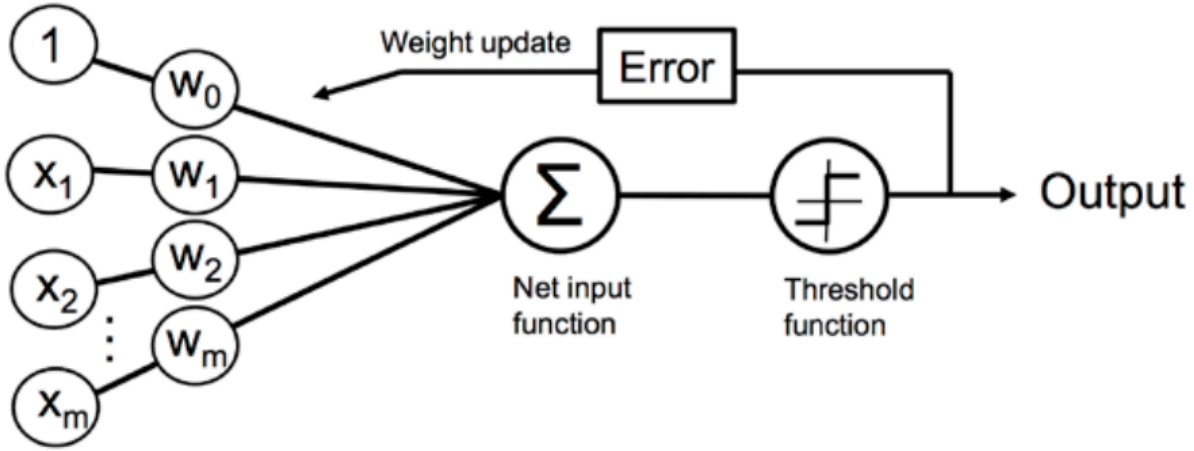
Perceptron Learning Rule

- Convergence of the perceptron is only guaranteed if the two classes are linearly separable and the learning rate is sufficiently small and/or a threshold for the # of tolerated misclassifications
  - ✓The perceptron would never stop updating the weight otherwise



Perceptron Learning Rule

- Illustrates how the perceptron receives the inputs of an example, **x**, and combines them with the weights, **w**, to compute the net input
- The net input is then passed on to the threshold function, which generates a binary output of –1 or +1—the predicted class label of the example
- During the learning phase, this output is used to calculate the error of the prediction and update the weights.



Perceptron Learning Rule – Code (Docker)

- Install Docker (Install Docker Engine)
  - ✓Mac : Supported platforms → Desktop → Docker Desktop for Mac (macOS) → Choose ”Mac with Intel chip” or “Mac with Apple chip” (It will ask you to login first when you download the .dmg file)
  - ✓Windows: Supported platforms->Desktop->Docker Desktop for Windows → Click on “Docker Desktop for Windows” (It will ask you to login first when you download the .dmg file)
- Docker Documentation → Click here

Perceptron Learning Rule – Code (Docker)

- Download code (Download zip)

https://github.com/rasbt/python-machine-learning-book-3rd-edition

- Unzip in your local computer, for example, **/Users/wonjunlee/myJupyter**

Perceptron Learning Rule

wonjunlee@OSXADMs-MacBook-Air myJupyter % **pwd**

/Users/wonjunlee/myJupyter

wonjunlee@OSXADMs-MacBook-Air myJupyter % **docker run -p 8888:8888 -v /Users/wonjunlee/myJupyter:/home/jovyan/work wjlee96/machine_learning1**

source code under your local drive (/Users/wonjunlee/myJupyter) is mounted to "/home/jovyan/work" in the container

```
wonjunlee@OSXADMs-MacBook-Air myJupyter % docker run -p 8888:8888 -v /Users/wonjunlee/myJupyter:/home/jovyan/work machine_le
arning1
WARN: Jupyter Notebook deprecation notice https://github.com/jupyter/docker-stacks#jupyter-notebook-deprecation-notice.
Executing the command: jupyter notebook
[W 2021-06-02 18:53:52.573 LabApp] 'ip' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp. Be
 sure to update your config before our next release.
[W 2021-06-02 18:53:52.573 LabApp] 'port' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp.
Be sure to update your config before our next release.
[W 2021-06-02 18:53:52.573 LabApp] 'port' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp.
Be sure to update your config before our next release.
[W 2021-06-02 18:53:52.575 LabApp] 'port' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp.
Be sure to update your config before our next release.
[I 2021-06-02 18:53:52.594 LabApp] JupyterLab extension loaded from /opt/conda/lib/python3.9/site-packages/jupyterlab
[I 2021-06-02 18:53:52.594 LabApp] JupyterLab application directory is /opt/conda/share/jupyter/lab
[I 18:53:52.626 NotebookApp] Serving notebooks from local directory: /home/jovyan
[I 18:53:52.627 NotebookApp] Jupyter Notebook 6.3.0 is running at:
[I 18:53:52.627 NotebookApp] http://da67ec584a98:8888/?token=4c204f0eb0afcd2630d3c75e75d7fafcf37495d98b4a3383
[I 18:53:52.627 NotebookApp]  or http://127.0.0.1:8888/?token=4c204f0eb0afcd2630d3c75e75d7fafcf37495d98b4a3383
[I 18:53:52.627 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 18:53:52.635 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///home/jovyan/.local/share/jupyter/runtime/nbserver-9-open.html
    Or copy and paste one of these URLs:
        http://da67ec584a98:8888/?token=4c204f0eb0afcd2630d3c75e75d7fafcf37495d98b4a3383
     or http://127.0.0.1:8888/?token=4c204f0eb0afcd2630d3c75e75d7fafcf37495d98b4a3383
```

Perceptron Learning Rule

wonjunlee@OSXADMs-MacBook-Air ~ % **docker container ls**

CONTAINER
ID IMAGE COMMAND CREATED STATUS PORTS NAMES

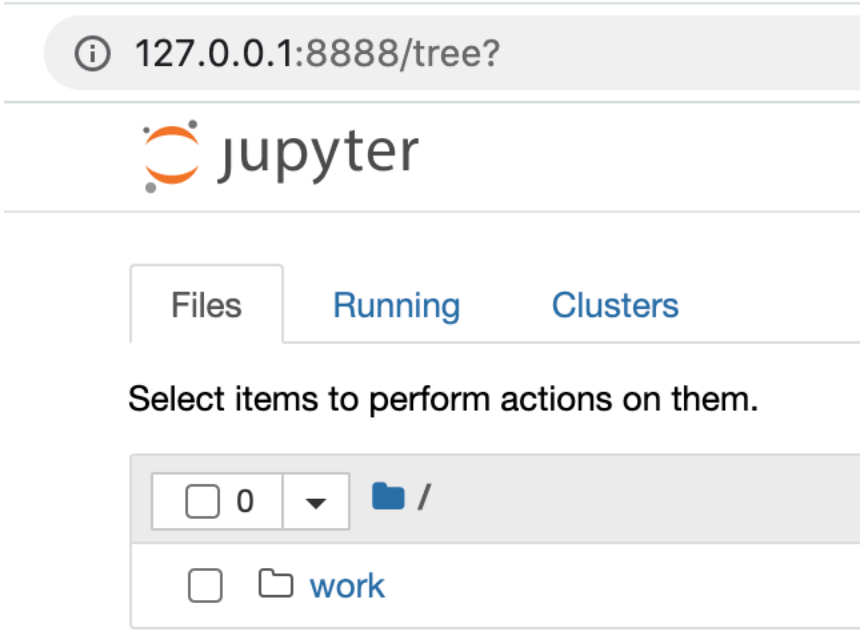
80cb2b8abdce wjlee96/machine_learning1 "tini -g -- start-no…" 4 minutes ago Up 4 minutes 0.0.0.0:8888->8888/tcp sleepy_bouman

wonjunlee@OSXADMs-MacBook-Air ~ % **docker exec -it 80cb /bin/bash**

(base) **jovyan@80cb2b8abdce**:**~**$

Perceptron Learning Rule

- Open a web browser and connect to 127.0.0.1:8888



Perceptron Learning Rule

- **An object-oriented perceptron API :**

http://127.0.0.1:8888/notebooks/work/python-machine-learning-book-3rd-edition-master/ch02/ch02.ipynb

An object-oriented perceptron API

- ***fit*** method
  - ✓Initialize weights in self.w_
  - ✓self.w_[0] : bias
  - ✓self.w contains random numbers from a normal distribution with std dev 0.01
  - ✓Keep initial weights to non-zero – if all the weights are initialized to zero → eta affects only the scale of the weight vector, not the direction (ref.)
  - ✓Loops over all individual examples in the training dataset and updates weights
  - ✓Calls predict method to predict class labels for weight update
  - ✓Collect # of misclassifications during each epoch (self.errors_)
- ***predict*** method
  - ✓Called in fit method
  - ✓Can also be used to predict the class labels of new data after training
- ***net_input*** method
  - ✓np.dot – calculates vector dot product, $w^T x$

Training a Perceptron Model on the Iris Dataset

- Restrict to two feature variables (dimensions)
- Use *pandas* library – load Iris dataset from UCI ML repository into a *DataFrame* object

```
import os
import pandas as pd


s = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'
print('URL:', s)

df = pd.read_csv(s,
                 header=None,
                 encoding='utf-8')

df.tail()
```

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Training a Perceptron Model on the Iris Dataset

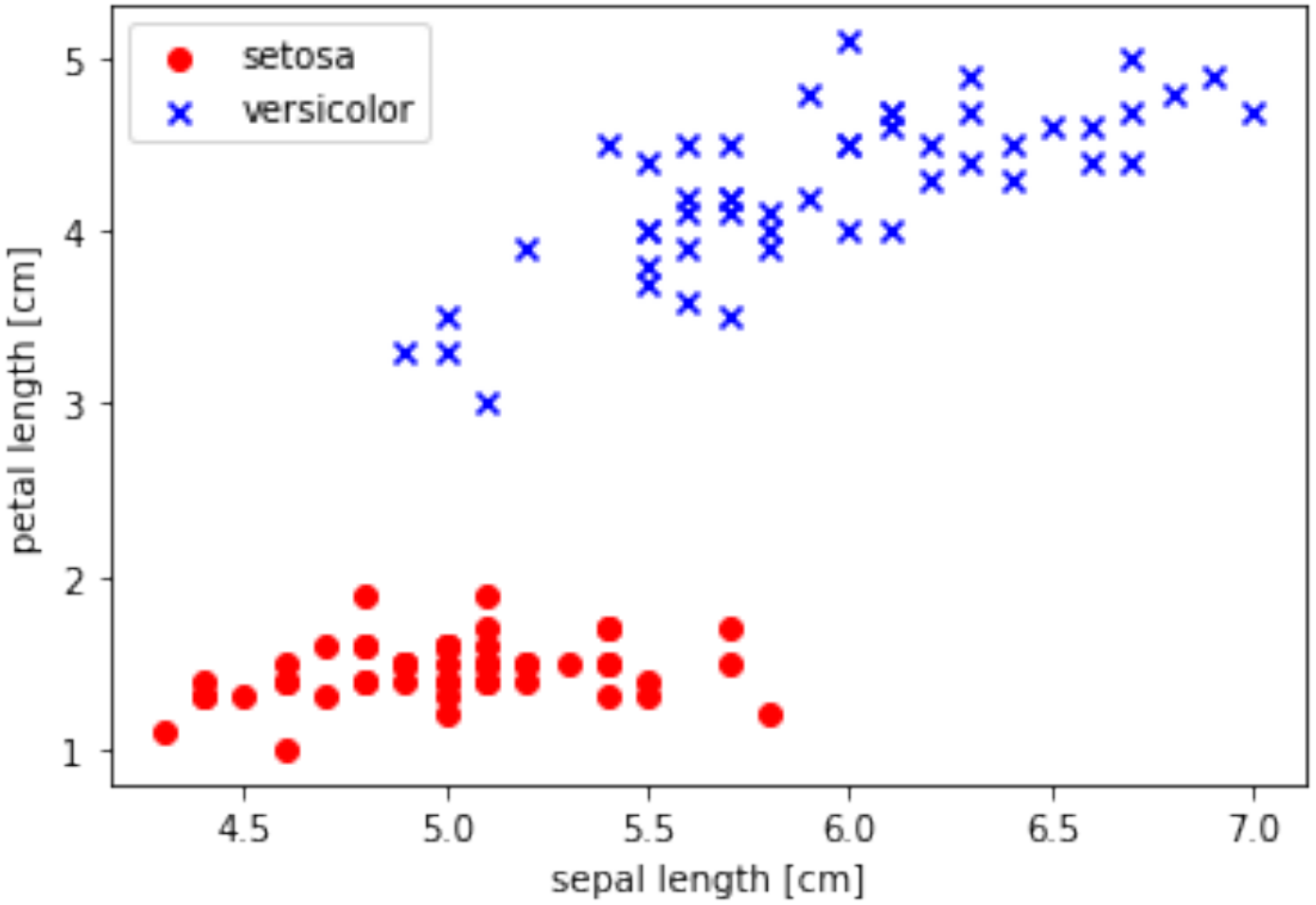
- Extract the first 100 class labels (50 Iris-setosa and 50 Iris-Versicolor)

  → 1 (versicolor) and -1 (setosa) : y

- Extract the 1st feature column (sepal length) and 3rd column (petal length) of 100 training examples : x

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

# select setosa and versicolor
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)

# extract sepal length and petal length
X = df.iloc[0:100, [0, 2]].values

# plot data
plt.scatter(X[:50, 0], X[:50, 1],
            color='red', marker='o', label='setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],
            color='blue', marker='x', label='versicolor')

plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')

# plt.savefig('images/02_06.png', dpi=300)
plt.show()
```

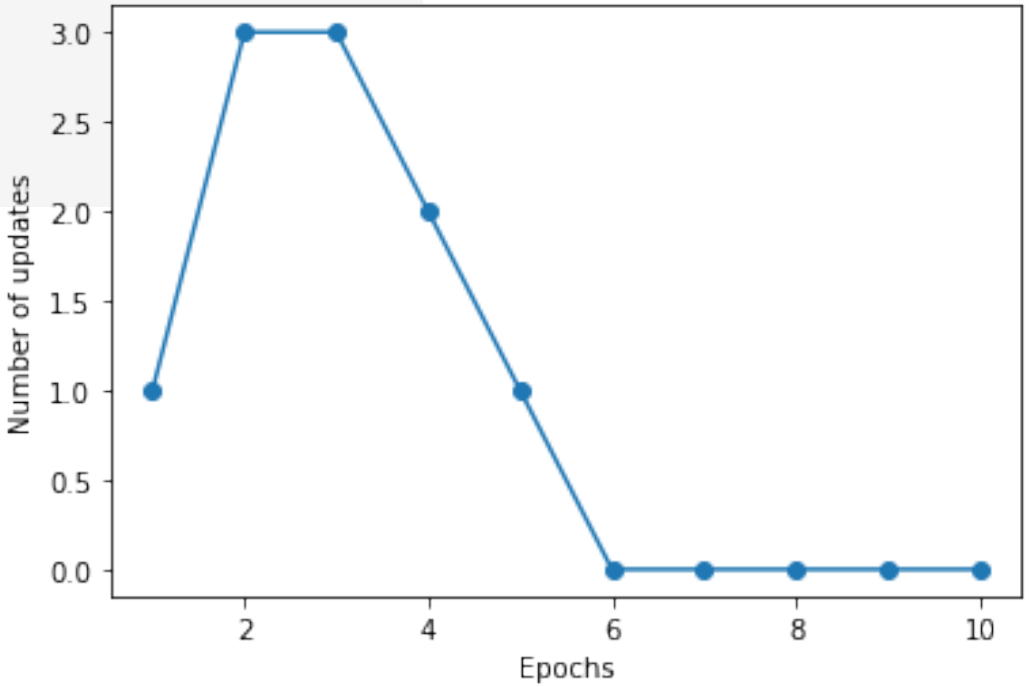
Training a Perceptron Model on the Iris Dataset



Training a Perceptron Model on the Iris Dataset

- Plot of the misclassification errors versus the number of epochs

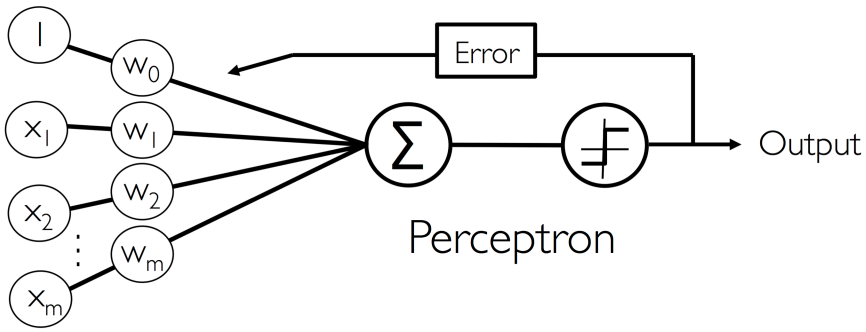
```
ppn = Perceptron(eta=0.1, n_iter=10)
ppn.fit(X, y)

plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of updates')

# plt.savefig('images/02_07.png', dpi=300)
plt.show()
```

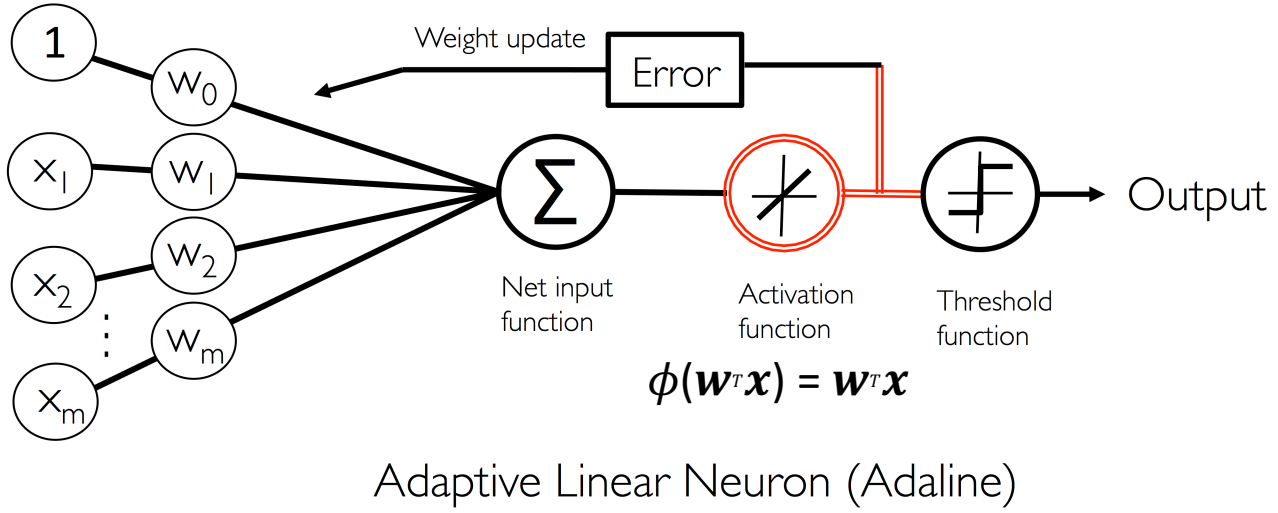


- Perceptron converged after the sixth epoch and should now be able to classify the training examples perfectly

Adaptive Linear Neurons and Convergence of Learning

- **Adaptive Linear Neurons (Adaline)** – Single-layer neural network (NN)
  - ✓Defining and minimizing continuous cost functions
  - ✓Weights are updated based on a linear activation function rather than a unit step function
  - ✓compares the true class labels with the linear activation function's continuous valued output to compute the model error and update the weights





Minimizing Cost Functions with Gradient Descent

- Objective function – a cost function (*J*) that we want to minimize
  - ✓Consider an input vector `x`, a matrix `W`, a target `y`, and a loss function `loss`
  - ✓You can use W to compute a target candidate `y_pred`, and compute the loss between the target candidate `y_pred` and the target `y`:

```
y_pred = dot(W, x)
loss_value = loss(y_pred, y)
```

  - ✓If the data inputs `x` and `y` are frozen, then this can be interpreted as a function mapping values of `W` to loss values:

```
loss_value = J(W)
```

Minimizing Cost Functions with Gradient Descent

- Learns weights as the sum of squared errors (SSE) : $J(\boldsymbol{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$
- With a function $J(\mathbb{W})$, you can reduce $J(\mathbb{W})$ by moving W in the **opposite** direction from the gradient:
  - ✓for example, ***W1 = W0 – step × ∇(J)(W0)*** (where step is a small scaling factor) : means going against the curvature, which intuitively should put you lower on the curve

Minimizing Cost Functions with Gradient Descent

$$w = w + \Delta w$$

$$\text{where } \Delta w = -\eta \nabla J(w)$$

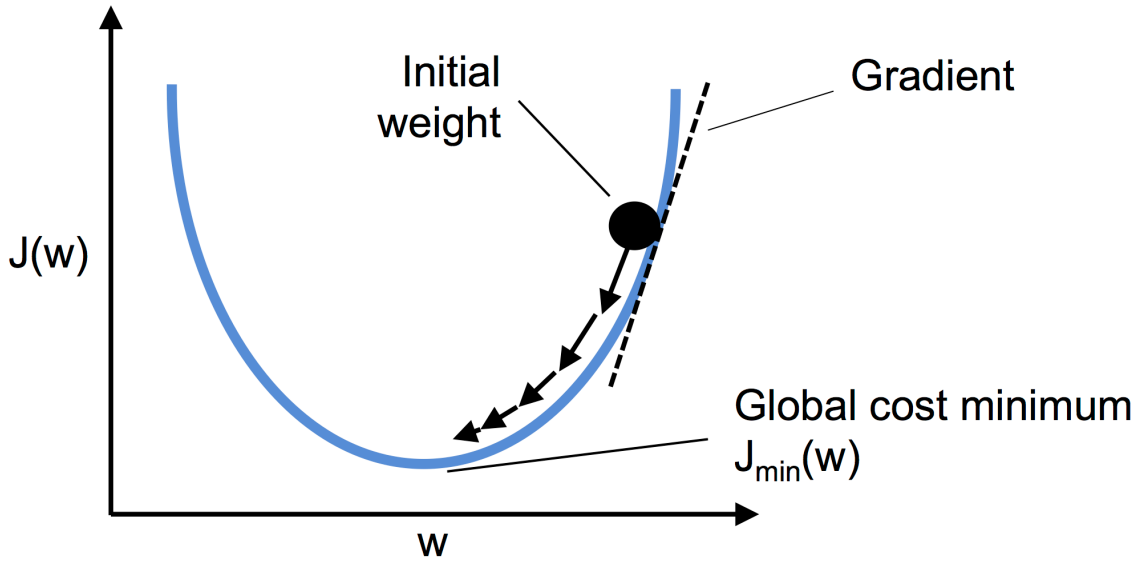
$$\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

- The weight update is calculated based on all examples in the training dataset (instead of updating the weights incrementally after each training example) → **batch gradient descent**

Minimizing Cost Functions with Gradient Descent

- Advantage of cost function
  - ✓Differential
  - ✓Convex – can use a very simple yet powerful optimization algorithm (gradient descent)
- Climbing down a hill until a local and global cost minimum is reached



Implementing Adaline in Python

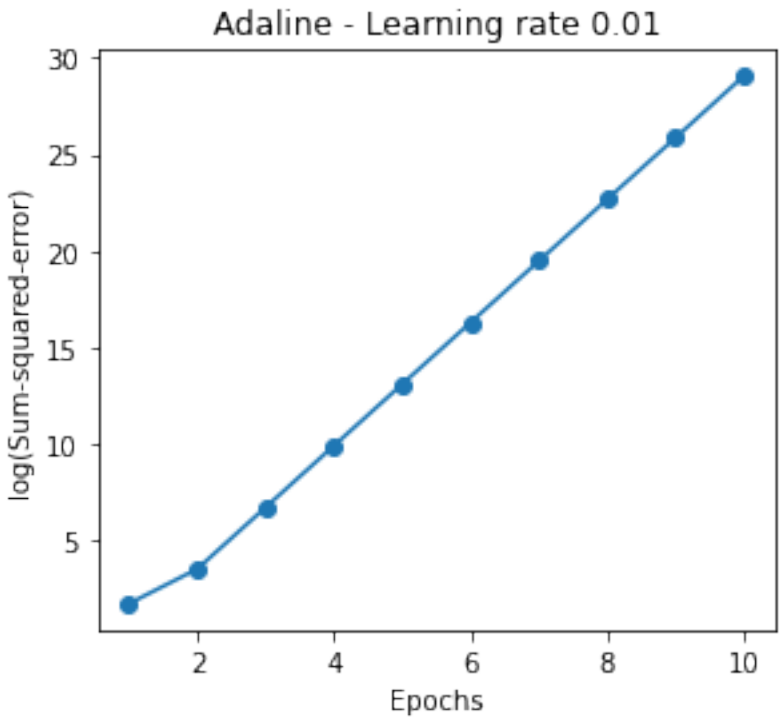
- http://127.0.0.1:8888/notebooks/work/python-machine-learning-book-3rd-edition-master/ch02/ch02.ipynb#Implementing-an-adaptive-linear-neuron-in-Python

Implementing Adaline in Python

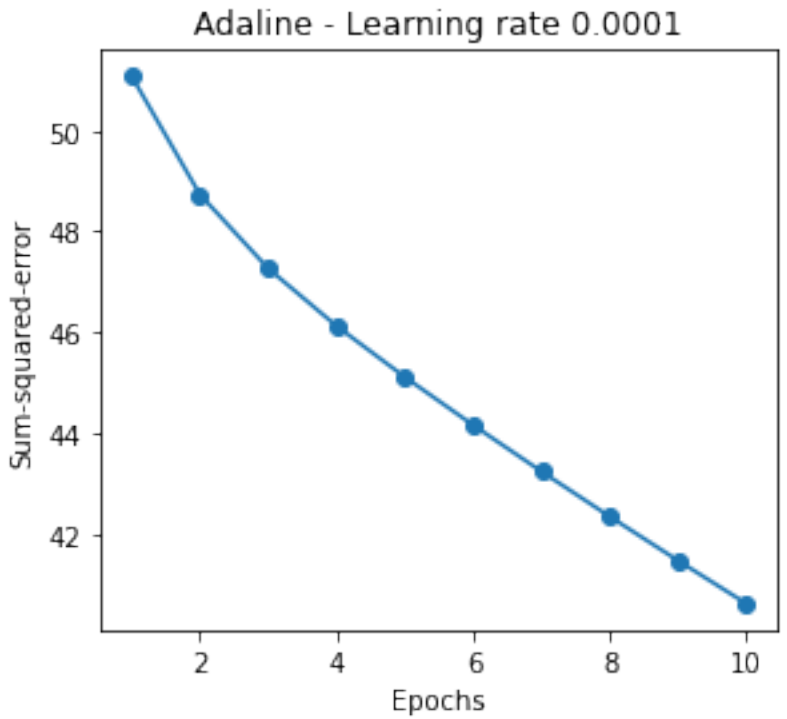
- calculate the gradient based on the whole training dataset via *self.eta * errors.sum()* for the bias unit (zero-weight), and via *self.eta * X.T.dot(errors)* for the weights 1 to *m*,
  - ✓*X.T.dot(errors)* is a matrix-vector multiplication between our feature matrix and the error vector
- In practice, it often requires some experimentation to find a good learning rate, η, for optimal convergence
- learning rate, η, epoch – hyperparameters

Implementing Adaline in Python

eta=0.01

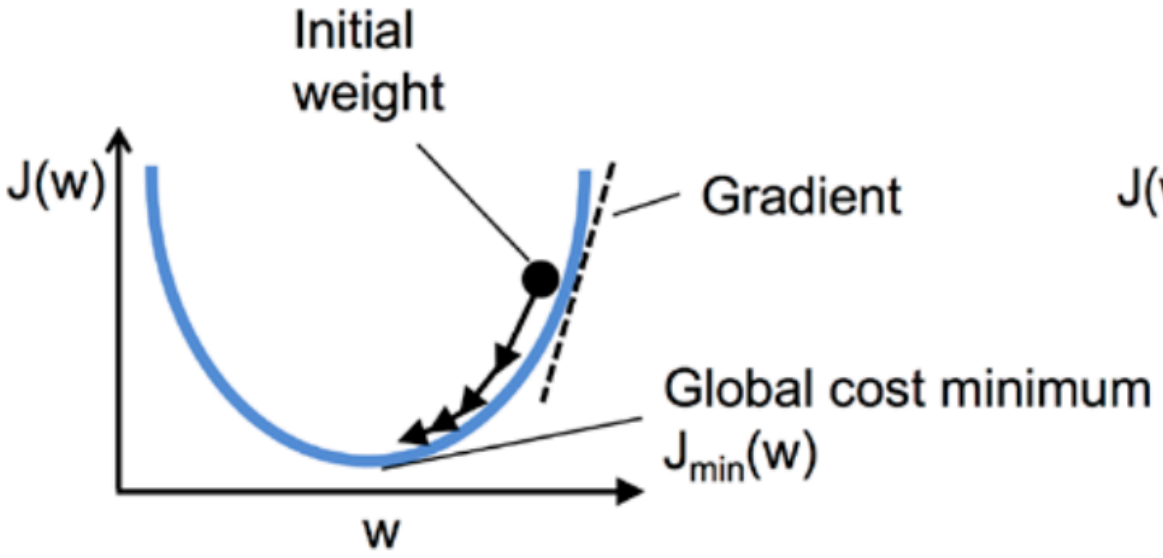


eta=0.0001

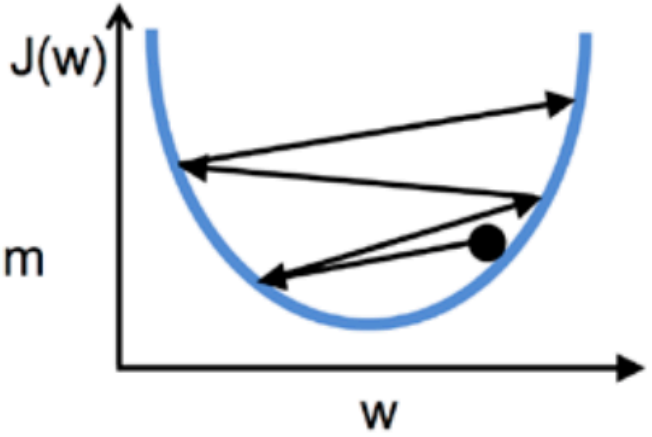


Implementing Adaline in Python

well-chosen learning rate



A too large learning rate

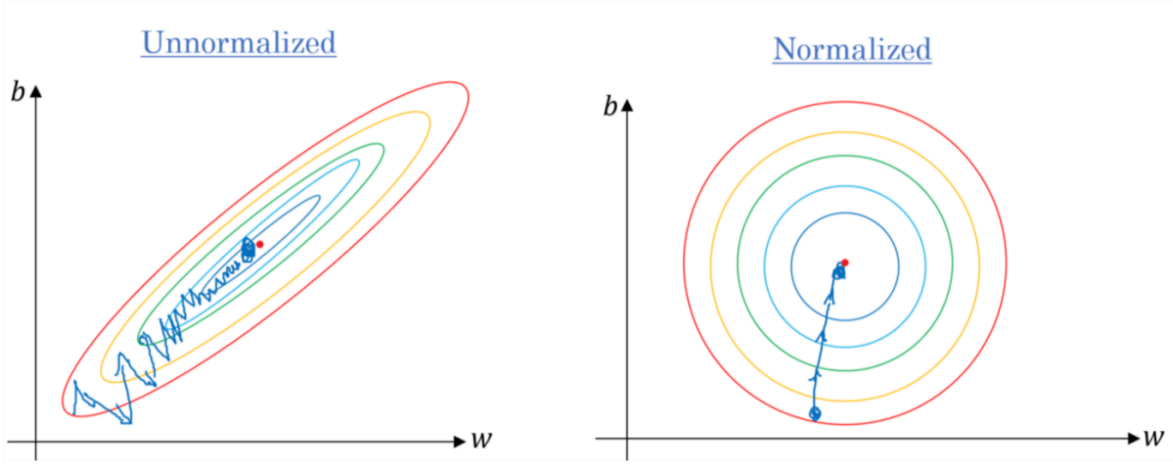


Improving Gradient Descent Through Feature Scaling

- Standardization – Standard normal distribution: zero-mean and unit variance → gradient descent learning to converge more quickly
- *j*th feature: x_j, sample mean: μ_j from every training example, std dev.: σ_j :

$$x_j' = \frac{x_j - \mu_j}{\sigma_j}$$

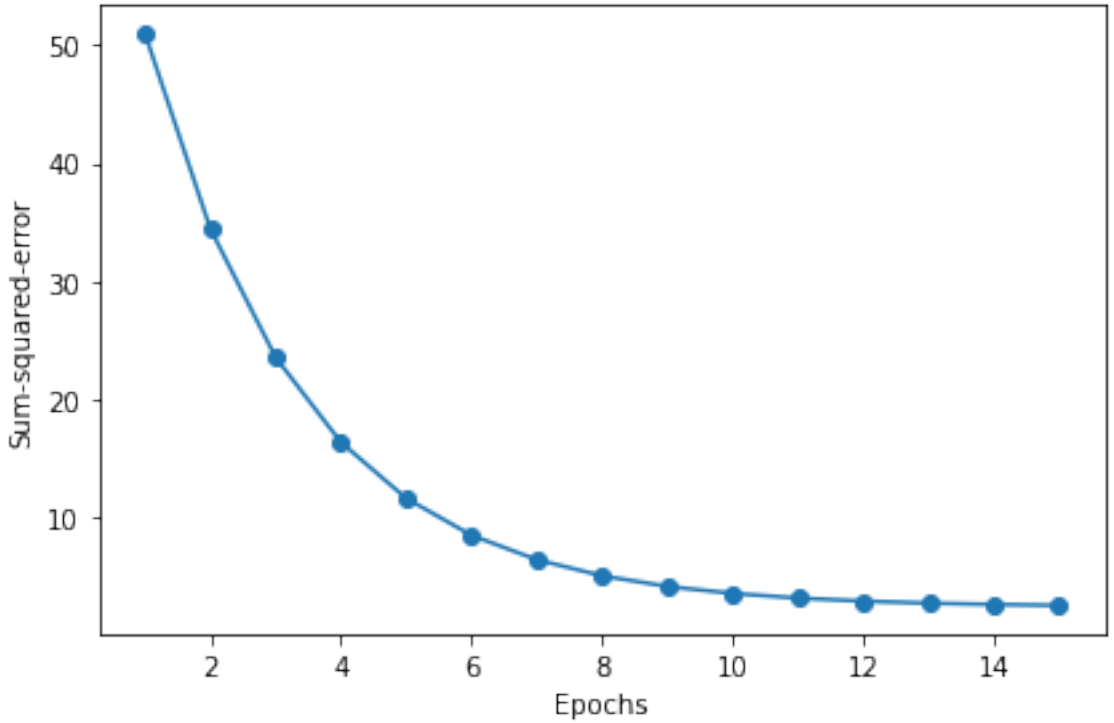
```
X_std = np.copy(X)
X_std[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()
X_std[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()
```



Improving Gradient Descent Through Feature Scaling

```
ada_gd = AdalineGD(n_iter=15, eta=0.01)
ada_gd.fit(X_std, y)

plot_decision_regions(X_std, y, classifier=ada_gd)
plt.title('Adaline - Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
# plt.savefig('images/02_14_1.png', dpi=300)
plt.show()

plt.plot(range(1, len(ada_gd.cost_) + 1), ada_gd.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Sum-squared-error')

plt.tight_layout()
# plt.savefig('images/02_14_2.png', dpi=300)
plt.show()
```

Improving Gradient Descent Through Feature Scaling



Large-scale machine learning and stochastic gradient descent

- **Batch Gradient Descent** – update the weights based on the sum of the accumulated errors over all training examples, $x^{(i)}$:

$$\Delta \boldsymbol{w} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) \boldsymbol{x}^{(i)}$$

- **Stochastic gradient descent (SGD)** – update the weights incrementally for each training example:

$$\eta \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) \boldsymbol{x}^{(i)}$$

- **Mini-batch Gradient Descent** – learning can be understood as applying batch gradient descent to smaller subsets of the training data, for example, 32 training examples at a time