Metodi di Apprendimento Automatico per la Fisica

Lesson 2

Dr. Autilia Vitiello

Department of Physics "Ettore Pancini"
University of Naples Federico II

2021/2022

Table of contents

Fundamentals of Supervised Learning

Artificial Neurons and Perceptrons

Introduction

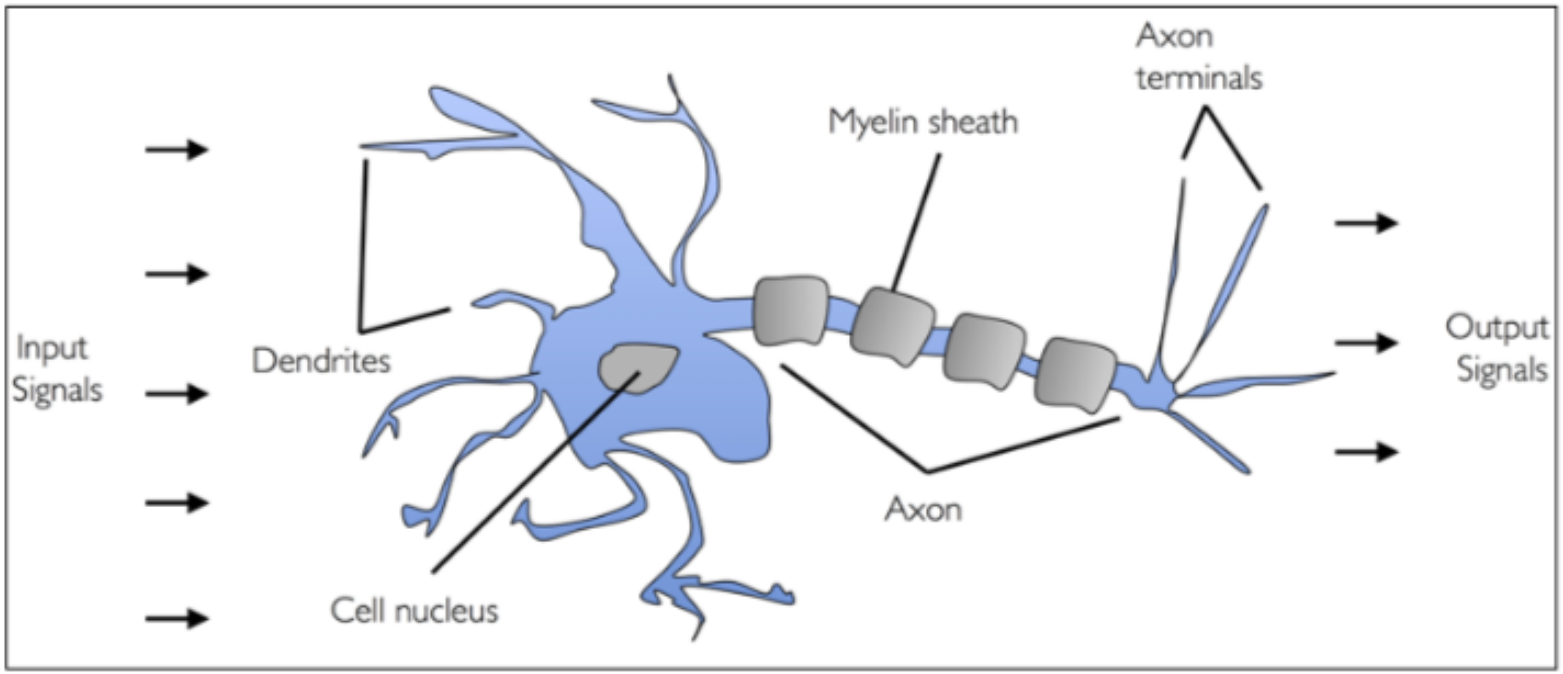
- In this lesson we will introduce the first two algorithms for supervised learning.
- The *Perceptron*;
- The *Adaptive Linear Neurons*;
- We will program and test them by using Python and a established dataset as the Iris dataset;

McCulloch-Pitts Neuron

Artificial Neurons and Perceptrons

McCulloch-Pitts Neuron

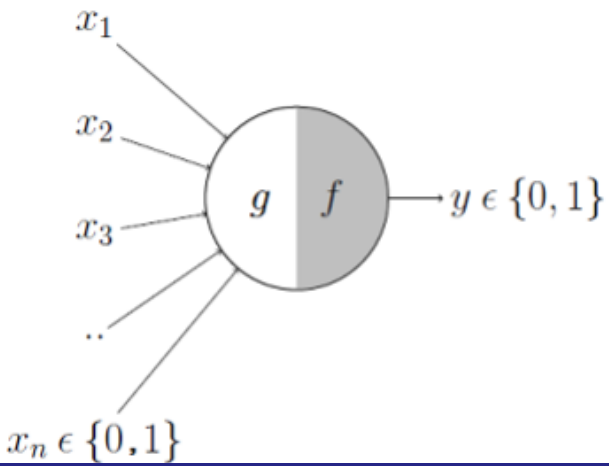
- In order to show how artificial neurons works, it is necessary to figure out how **natural neurons** works.



Artificial Neurons and Perceptrons

McCulloch-Pitts Neuron

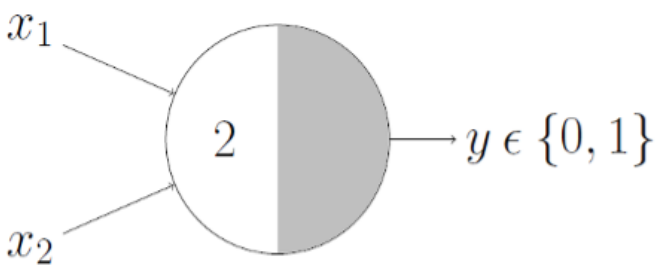
- The first artificial representation of a brain cell is the so-called **McCulloch-Pitts (MCP) neuron** introduced in 1943.
- McCulloch and Pitts described a brain cell as a logic gate with **binary outputs**;
- This gate collects and **integrates** the set of signals arriving at cell body;
- If the value of the integrated signals exceeds **a given threshold**, an output signal is generated and passed to other artificial cells by means of axons.



Artificial Neurons and Perceptrons

McCulloch-Pitts Neuron

- It is possible to consider it as a different view of conventional logic gates AND, OR, NOT, used in digital electronics;



AND function

$$x_1 + x_2 = \sum_{i=1}^{2} x_i \geq 2$$

Artificial Neurons and Perceptrons

McCulloch-Pitts Neuron: Limitations

- What about non-boolean (say, real) inputs?
- Are all inputs equal? What if we want to assign more importance to some inputs?
- Do we always need to hand code the threshold?
- Overcoming the limitations of the MCP neuron, Frank Rosenblatt, an American psychologist, proposed the classical **perception model**, the mighty artificial neuron, in 1958. It is more generalized computational model than the McCulloch-Pitts neuron where **weights and thresholds can be learnt over time**.

Rosenblatt's Perceptron is the first example of **binary classifier** in the context of **supervised learning**. Formally,

- It implements a decision function $\phi(z)$;
- z is a linear combination of input values and $\mathbf{x}$ and a weight vector $\mathbf{w}$: $z = w_1 x_1 + \ldots + w_m x_m$.

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

If the net input of a particular sample $x^{(i)}$ is greater than a defined threshold θ, the neuron predict class 1, and class -1 otherwise.
Formally:

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

θ can be moved to the left side of the equation and define a weight-zero as $w_0 = -\theta$ and $x_0 = 1$ in order to write z in a more compact way:

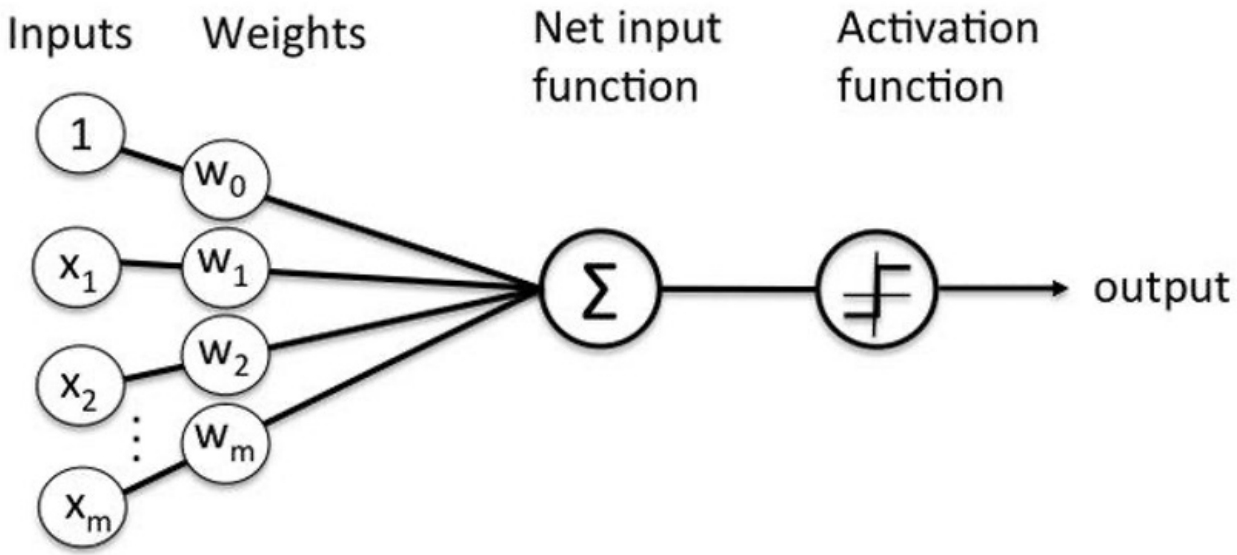
$$z = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \mathbf{w^T x}$$

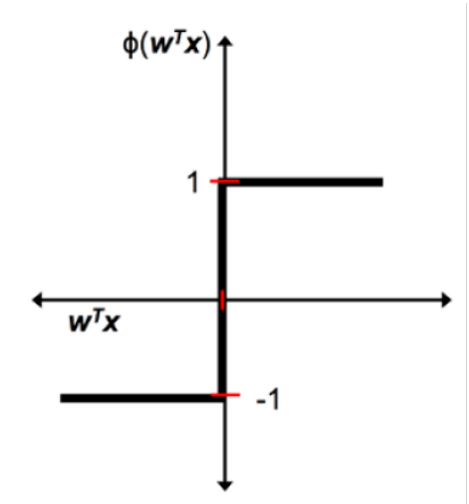
$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

In machine learning the value $w_0 = -\theta$ is called **bias unit**.

Artificial Neurons and Perceptrons

Rosenblatt's Perceptron workflow

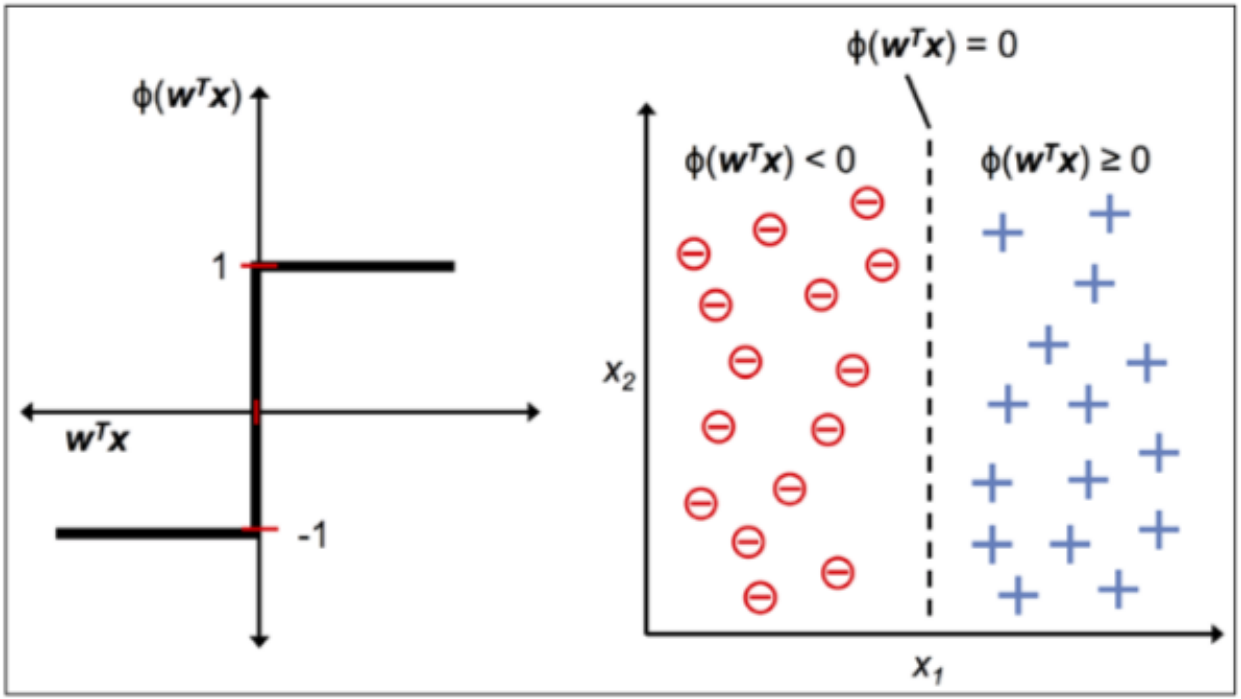




Artificial Neurons and Perceptrons

Rosenblatt's Perceptron as Binary Classifier

A classifier corresponds to a decision boundary, or a hyperplane such that the positive examples lie on one side, and negative examples lie on the other side.



The Perceptron Learning Rule

How to train a Perceptron:

1. Initialise the weights to 0 or small random numbers.
2. For each training sample $x^{(i)}$:
   1. Compute the output value $\hat{y}$.
   2. Update weights,

$$w_j = w_j + \Delta w_j$$

How to compute Δw_j:

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

- η is named learning rate (a constant between 0.0 and 1.0);
- $y^{(i)}$ is the true class label of the *i*-th training sample;
- $\hat{y}^{(i)}$ is the predict class label.

Example for a two-dimensional dataset:

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

In the case that the perceptron predicts the class label correctly:

$$\Delta w_j = \eta(-1 - (-1))x_j^{(i)} = 0$$
$$\Delta w_j = \eta(1 - 1)x_j^{(i)} = 0$$

In the case of a wrong prediction:

$$\Delta w_j = \eta(1 - (-1))x_j^{(i)} = \eta(2)x_j^{(i)}$$
$$\Delta w_j = \eta(-1 - 1)x_j^{(i)} = \eta(-2)x_j^{(i)}$$

Let us consider $x_j^{(i)} = 0.5$, $\eta = 1$ and we misclassify the target sample as -1:

$$\Delta w_j = (1 - (-1))0.5^{(i)} = (2)0.5^{(i)} = 1$$

Let us consider $x_j^{(i)} = 2$, $\eta = 1$ and we misclassify the target sample as -1:

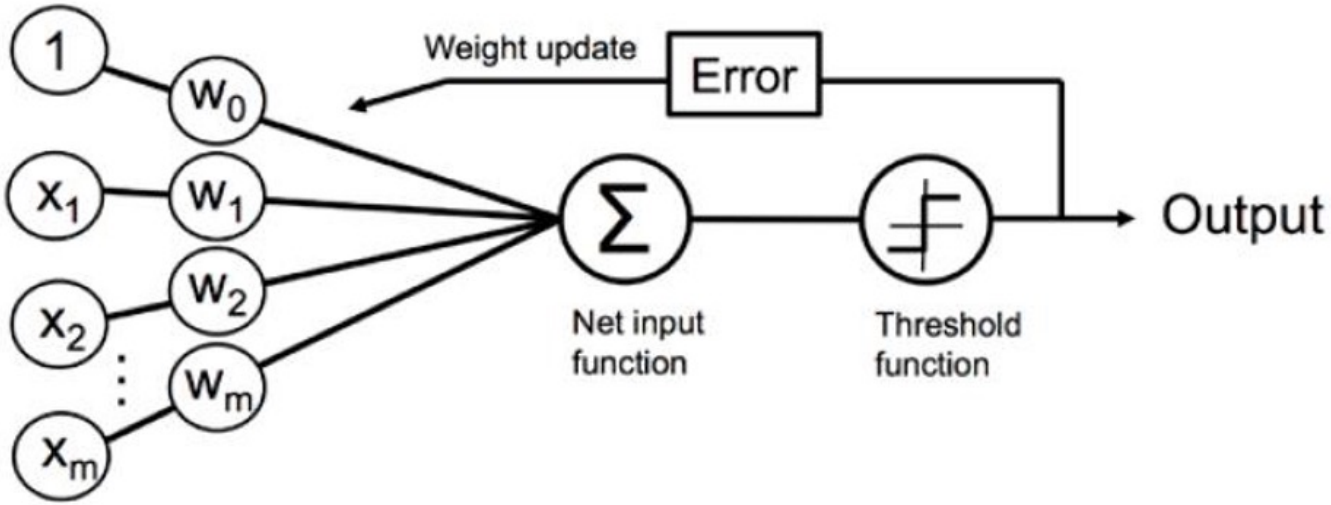
$$\Delta w_j = (1 - (-1))2^{(i)} = (2)2^{(i)} = 4$$

The weight update is proportional to the value of $x_j^{(i)}$.

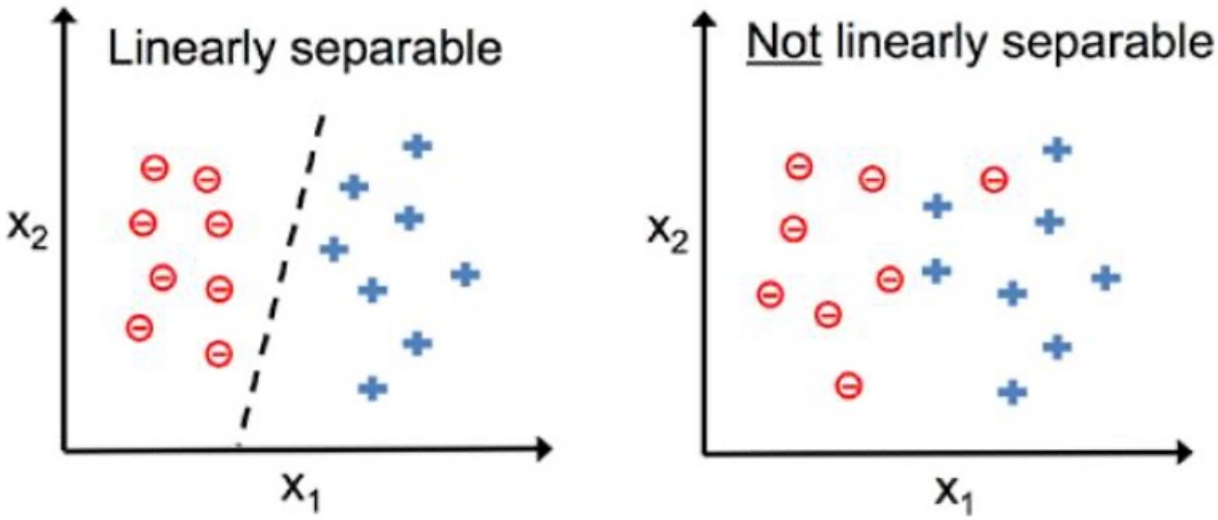
Artificial Neurons and Perceptrons

Rosenblatt's Perceptron: final scheme

The learning algorithm passes over the training dataset until all the input vectors are classified correctly (until it achieves convergence).



The convergence of a neuron is only guaranteed if the two classes are linearly separable



If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (**epochs**) and/or **a threshold for the number of tolerated misclassifications** — the perceptron would never stop updating the weights otherwise.

Artificial Neurons and Perceptrons

Implementing a Perceptron in Python

An object-oriented approach is used to design and develop a perceptron interface as a Python class. In this way, it is possible to initialize new `Perceptron` objects able to learn from data via a `fit` method, and make prediction via a predict method.

```
1  import numpy as np
2      class Perceptron(object):
3          """Perceptron classifier.
4          Parameters
5          ------------
6          eta : float
7              Learning rate (between 0.0 and 1.0)
8          n_iter : int
9              Passes over the training dataset.
10         random_state : int
11             Random number generator seed for random weight initialization.
12
13         Attributes
14         ------------
15         w_ : 1d-array
16             Weights after fitting.
17         errors_ : list
18             Number of misclassifications (updates) in each epoch.
19         """
20
21         def __init__(self, eta=0.01, n_iter=50, random_state=1):
22             self.eta = eta
23             self.n_iter = n_iter
24             self.random_state = random_state
```

Artificial Neurons and Perceptrons

Implementing a Perceptron in Python

An object-oriented approach is used to design and develop a perceptron interface as a Python class. In this way, it is possible to initialize new `Perceptron` objects able to learn from data via a `fit` method, and make prediction via a predict method.

```
1  import numpy as np
2      class Perceptron(object):
3          """Perceptron classifier.
4          Parameters
5          ------------
6          eta : float
7              Learning rate (between 0.0 and 1.0)
8          n_iter : int
9              Passes over the training dataset.
10         random_state : int
11             Random number generator seed for random weight initialization.
12
13         Attributes
14         ------------
15         w_ : 1d-array
16             Weights after fitting.
17         errors_ : list
18             Number of misclassifications (updates) in each epoch.
19         """
20
21         def __init__(self, eta=0.01, n_iter=50, random_state=1):
22             self.eta = eta
23             self.n_iter = n_iter
24             self.random_state = random_state
```

Artificial Neurons and Perceptrons

Implementing a Perceptron in Python

An object-oriented approach is used to design and develop a perceptron interface as a Python class. In this way, it is possible to initialize new `Perceptron` objects able to learn from data via a `fit` method, and make prediction via a predict method.

```
1  import numpy as np
2      class Perceptron(object):
3          """Perceptron classifier.
4          Parameters
5          ------------
6          eta : float
7              Learning rate (between 0.0 and 1.0)
8          n_iter : int
9              Passes over the training dataset.
10         random_state : int
11             Random number generator seed for random weight initialization.
12
13         Attributes
14         ------------
15         w_ : 1d-array
16             Weights after fitting.
17         errors_ : list
18             Number of misclassifications (updates) in each epoch.
19         """
20
21         def __init__(self, eta=0.01, n_iter=50, random_state=1):
22             self.eta = eta
23             self.n_iter = n_iter
24             self.random_state = random_state
```

Artificial Neurons and Perceptrons

Implementing a Perceptron in Python

An object-oriented approach is used to design and develop a perceptron interface as a Python class. In this way, it is possible to initialize new `Perceptron` objects able to learn from data via a `fit` method, and make prediction via a predict method.

```
1  import numpy as np
2      class Perceptron(object):
3          """Perceptron classifier.
4          Parameters
5          ------------
6          eta : float
7              Learning rate (between 0.0 and 1.0)
8          n_iter : int
9              Passes over the training dataset.
10         random_state : int
11             Random number generator seed for random weight initialization.
12
13         Attributes
14         ------------
15         w_ : 1d-array
16             Weights after fitting.
17         errors_ : list
18             Number of misclassifications (updates) in each epoch.
19         """
20
21         def __init__(self, eta=0.01, n_iter=50, random_state=1):
22             self.eta = eta
23             self.n_iter = n_iter
24             self.random_state = random_state
```

Artificial Neurons and Perceptrons

Implementing a Perceptron in Python

An object-oriented approach is used to design and develop a perceptron interface as a Python class. In this way, it is possible to initialize new `Perceptron` objects able to learn from data via a `fit` method, and make prediction via a predict method.

```
25      def fit(self, X, y):
26          """Fit training data.
27          Parameters
28          ----------
29          X : {array-like}, shape = [n_samples, n_features]
30              Training vectors, where n_samples is the number of
31              samples and n_features is the number of features.
32          y : array-like, shape = [n_samples]
33              Target values.
34
35          Returns
36          -------
37          self : object
38          """
39          rgen = np.random.RandomState(self.random_state)
40          self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
41          self.errors_ = []
42
43          for _ in range(self.n_iter):
44              errors = 0
45              for xi, target in zip(X, y):
46                  update = self.eta * (target - self.predict(xi))
47                  self.w_[1:] += update * xi
48                  self.w_[0] += update
49                  errors += int(update != 0.0)
50              self.errors_.append(errors)
```

Artificial Neurons and Perceptrons

Implementing a Perceptron in Python

An object-oriented approach is used to design and develop a perceptron interface as a Python class. In this way, it is possible to initialize new `Perceptron` objects able to learn from data via a `fit` method, and make prediction via a predict method.

```
25      def fit(self, X, y):
26          """Fit training data.
27          Parameters
28          ----------
29          X : {array-like}, shape = [n_samples, n_features]
30              Training vectors, where n_samples is the number of
31              samples and n_features is the number of features.
32          y : array-like, shape = [n_samples]
33              Target values.
34
35          Returns
36          -------
37          self : object
38          """
39          rgen = np.random.RandomState(self.random_state)
40          self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
41          self.errors_ = []
42
43          for _ in range(self.n_iter):
44              errors = 0
45              for xi, target in zip(X, y):
46                  update = self.eta * (target - self.predict(xi))
47                  self.w_[1:] += update * xi
48                  self.w_[0] += update
49                  errors += int(update != 0.0)
50              self.errors_.append(errors)
```

Artificial Neurons and Perceptrons

Implementing a Perceptron in Python

An object-oriented approach is used to design and develop a perceptron interface as a Python class. In this way, it is possible to initialize new `Perceptron` objects able to learn from data via a `fit` method, and make prediction via a predict method.

```
25      def fit(self, X, y):
26          """Fit training data.
27          Parameters
28          ----------
29          X : {array-like}, shape = [n_samples, n_features]
30              Training vectors, where n_samples is the number of
31              samples and n_features is the number of features.
32          y : array-like, shape = [n_samples]
33              Target values.
34
35          Returns
36          -------
37          self : object
38          """
39          rgen = np.random.RandomState(self.random_state)
40          self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
41          self.errors_ = []
42
43          for _ in range(self.n_iter):
44              errors = 0
45              for xi, target in zip(X, y):
46                  update = self.eta * (target - self.predict(xi))
47                  self.w_[1:] += update * xi
48                  self.w_[0] += update
49                  errors += int(update != 0.0)
50              self.errors_.append(errors)
```

Artificial Neurons and Perceptrons

Implementing a Perceptron in Python

An object-oriented approach is used to design and develop a perceptron interface as a Python class. In this way, it is possible to initialize new `Perceptron` objects able to learn from data via a `fit` method, and make prediction via a predict method.

```
51              return  self
52
53          def  net_input ( self ,  X):
54              """Calculate  net  input"""
55              return  np.dot(X,  self .w_[1:])  + self .w_[0]
56
57          def  predict ( self ,  X):
58              """Return class  label  after  unit  step"""
59              return  np.where( self . net_input (X) >= 0.0, 1, -1)
```

Artificial Neurons and Perceptrons

Implementing a Perceptron in Python

An object-oriented approach is used to design and develop a perceptron interface as a Python class. In this way, it is possible to initialize new `Perceptron` objects able to learn from data via a `fit` method, and make prediction via a predict method.

```
51              return self
52
53          def net_input(self, X):
54              """Calculate net input"""
55              return np.dot(X, self.w_[1:]) + self.w_[0]
56
57          def predict(self, X):
58              """Return class label after unit step"""
59              return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Artificial Neurons and Perceptrons

Implementing a Perceptron in Python

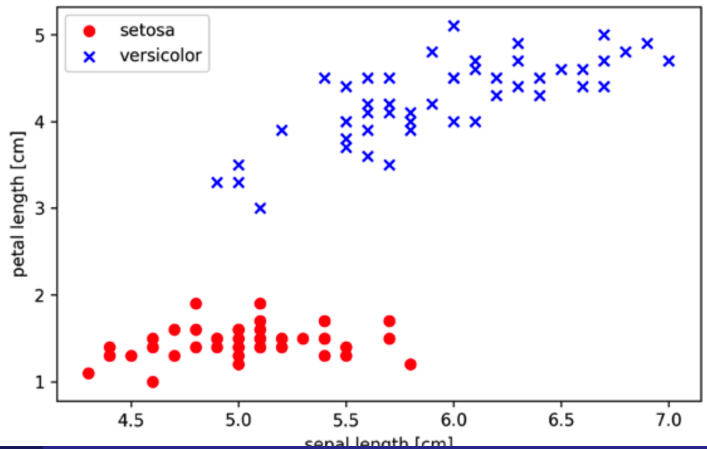
```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'
...                  'machine-learning-database/iris/iris.data',
...                  'header=None')
>>> df.tail()
```

	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

Artificial Neurons and Perceptrons

Implementing a Perceptron in Python

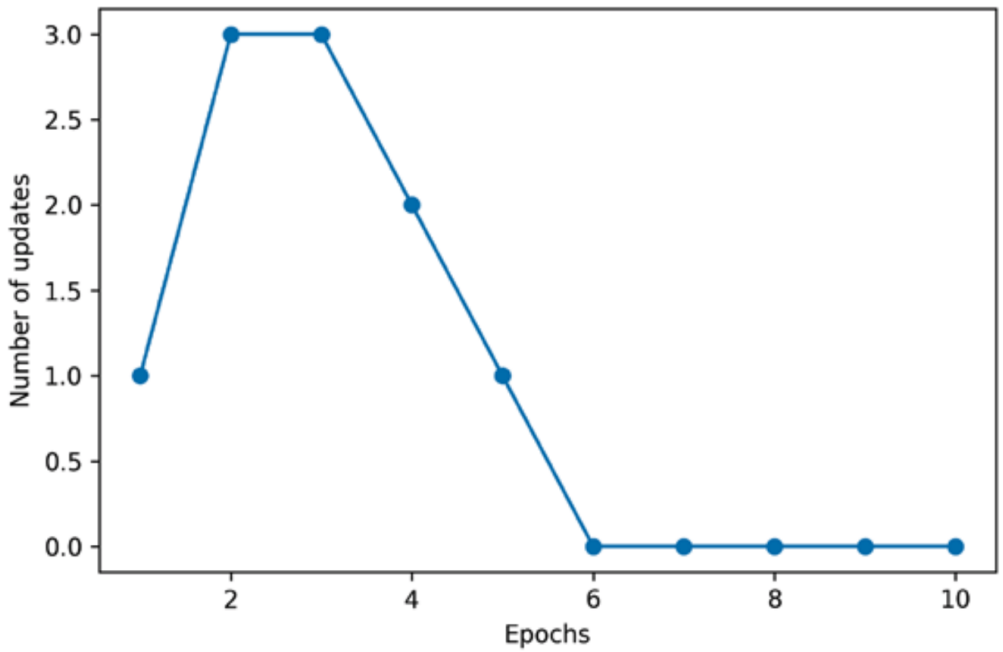
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> # select setosa and versicolor
>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', -1, 1)
>>> # extract sepal length and petal length
>>> X = df.iloc[0:100, [0, 2]]. values
>>> # plot data
>>> plt.scatter(X[:50, 0], 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.show()
```



Artificial Neurons and Perceptrons

Implementing a Perceptron in Python

```
>>> 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.show()
```



Artificial Neurons and Perceptrons

Let's implement a small convenience function to visualize the decision boundaries for two-dimensional datasets:

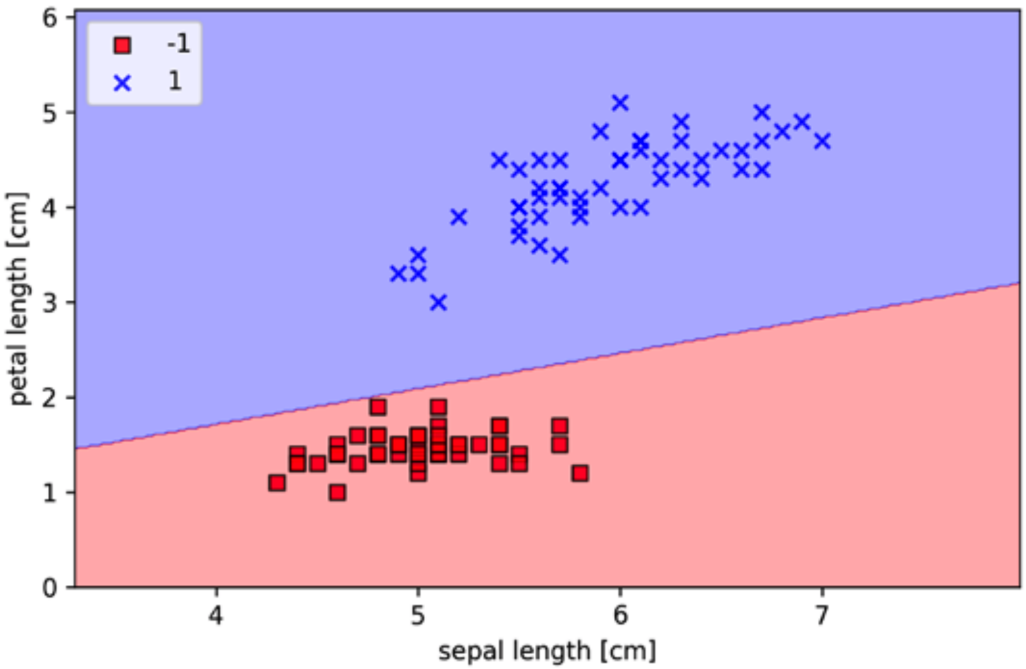
```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, resolution=0.02):
    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class examples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1], alpha=0.8, c=colors[idx],
            marker=markers[idx],
            label=cl,
            edgecolor='black')
```

Artificial Neurons and Perceptrons

After executing the preceding code example, we should now see a plot of the decision regions, as shown in the following figure:



As we can see in the plot, the perceptron learned a decision boundary that is able to classify all flower examples in the Iris training subset perfectly.

Adaptive Linear Neurons: Adaline

Adaptive Linear Neurons: Adaline

- *ADAptive Linear NEuron (Adaline)* is another type of single-layer neural network;
- Published by Bernard Widrow and Tedd Hoff few years after Frank Rosenblatt's perceptron algorithm (1960);
- It can be considered as an improvement of the perceptron:
  - Adaline uses **continuous predicted values** to learn the model weights, which is more "powerful" since it tells us by "how much" we were right or wrong.
  - Adaline uses the **gradient descent** to find the most suitable weights that minimize the error to classify the training data samples.

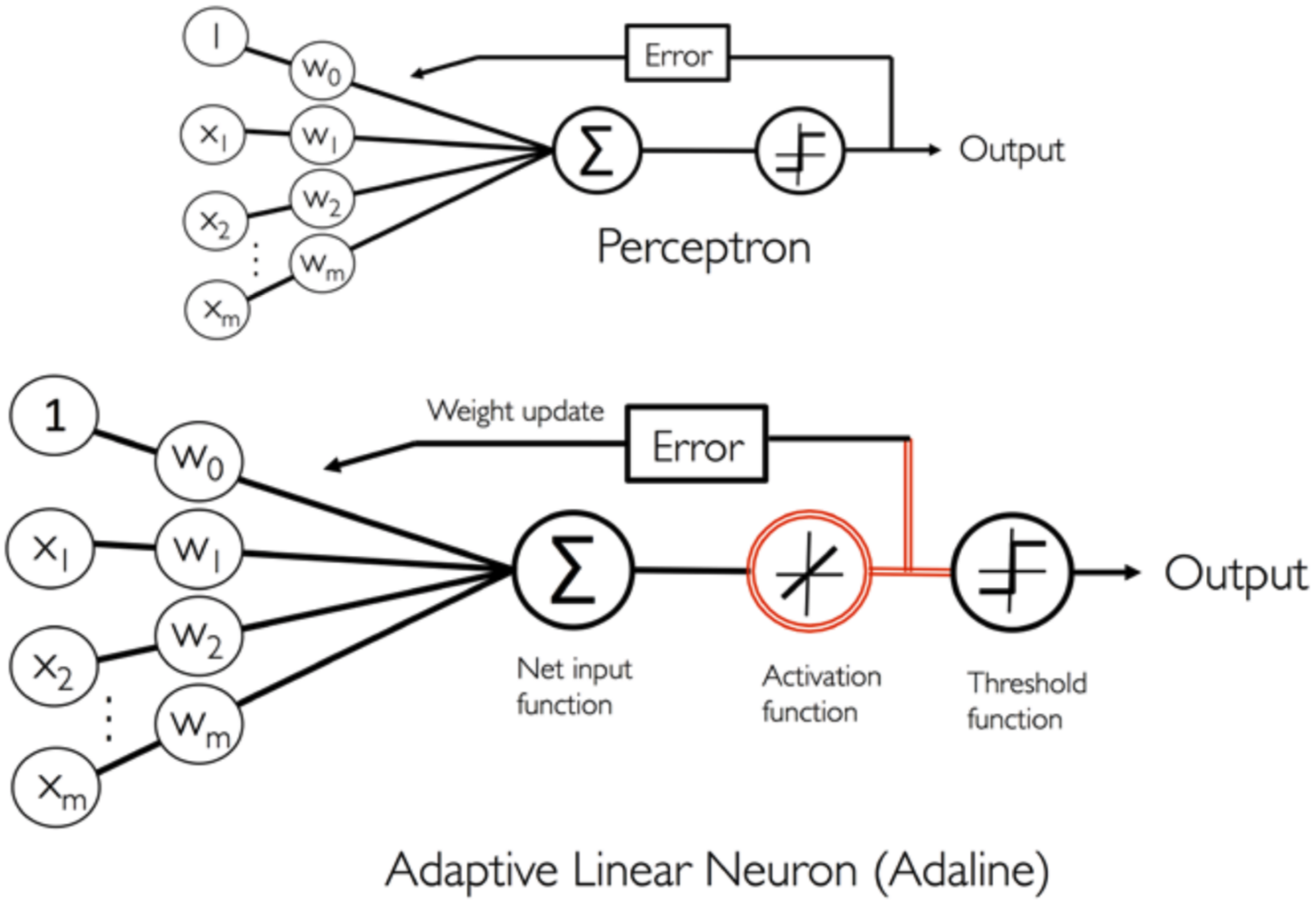
Adaline: activation function

- The key difference between Adaline learning rule (aka *Widrow-Hoff rule*) and Rosenblatt's perceptron is that the weights are updated by a **linear activation function** rather than a step function:

$$\phi(\mathbf{w}^T\mathbf{x}) = \mathbf{w}^T\mathbf{x}$$

- The activation function, used for learning the weights, is the **identity function** for Adaline
- However, a threshold function is still used to make a final decision.

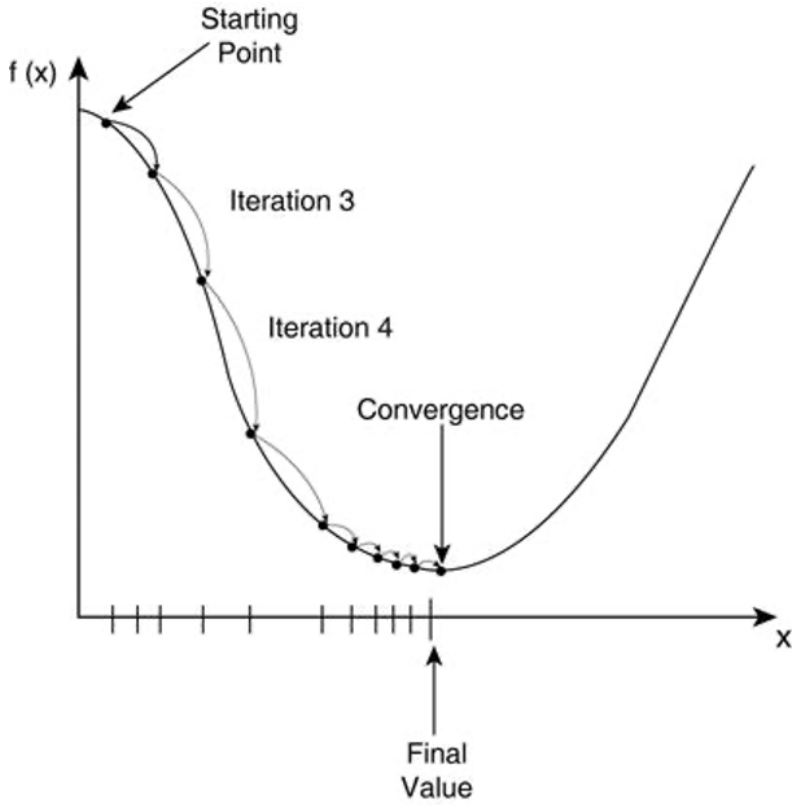
Adaline workflow



Adaline learning procedure: Gradient Descent

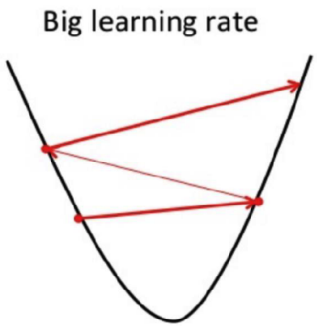
Gradient descent procedure:

- To minimize the function $f(x)$, the method starts at a point x_0 and, as many times as needed, moves from x_i to x_{i+1} by minimizing along the line extending from x_i in the direction of $-\nabla f(x_i)$, the local downhill gradient.
- It follows that, if $x_{i+1} = x_i - \gamma \nabla f(x_i)$ for $\gamma \in \mathcal{R}_+$ small enough, then $f(x_i) \geq f((x_{i+1}))$
- By considering the sequence $x_0, x_1, x_2, \ldots$ we obtain a sequence $f(x_0) \geq f(x_1) \geq f(x_2) \geq \ldots$ until one achieves the global optimum.

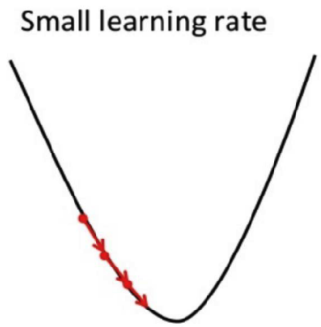


Gradient Descent: learning rate

- We have discussed $x_{i+1} = x_i - \gamma \nabla f(x_i)$
- γ is the learning rate constant that determines the size of the steps.



With a **high learning rate** we can cover more ground each step (so it **learns faster**), **but we risk overshooting** the lowest point since the slope of the hill is constantly changing.



With a very **low learning rate**, we can **confidently move in the direction of the negative gradient** since we are recalculating it frequently, but calculating the gradient is time-consuming, so it will take us a very **long time** to get to the bottom

Adaptive Linear Neurons: Adaline

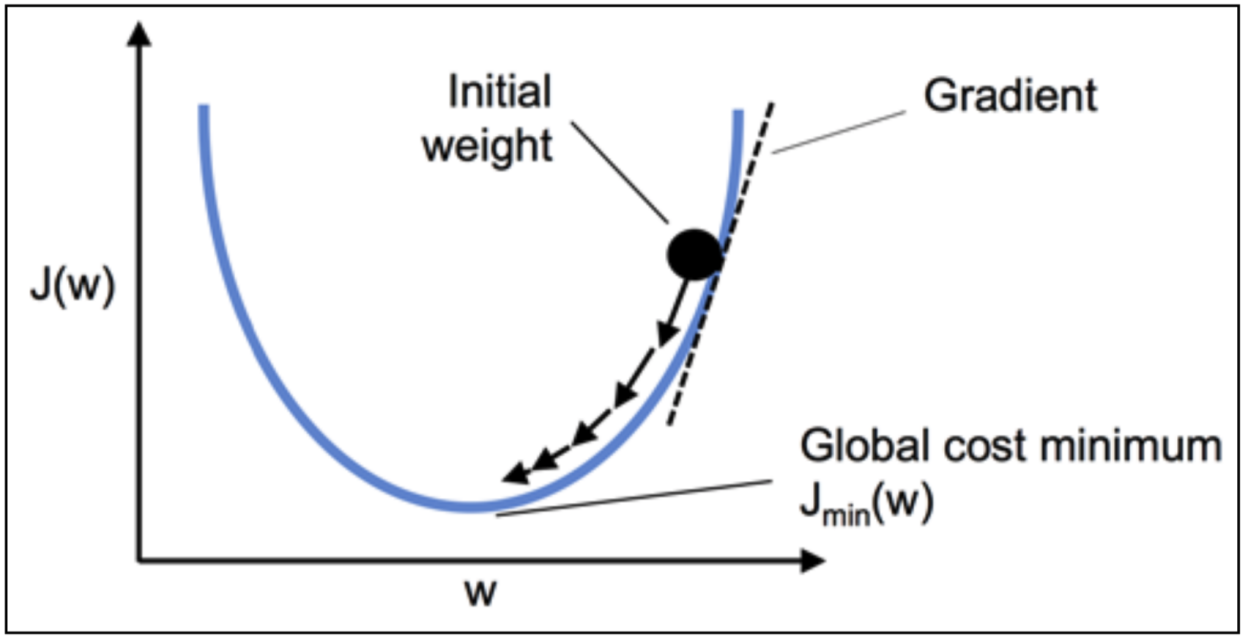
- Learning procedure consists of minimizing the cost function with gradient descent.
- For Adaline, the cost function J defined to learn the weights as the **Sum of Squared Errors(SSE)** between the calculated outcome and the true class label:

$$J(\mathbf{w}) = \frac{1}{2} \sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

- The term $\frac{1}{2}$ is added for a future convenience in the computation of the gradient;
- The function J is differentiable and convex;

Adaptive Linear Neurons: Adaline

- The **gradient descent** works as *climbing down hill* until a local or global cost minimum is reached.



Adaptive Linear Neurons: Adaline

To compute the gradient of the cost function SSE, it is necessary to compute the partial derivative of the cost function with respect the weight w_j:

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{w_j}\frac{1}{2}\sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

$$= \frac{1}{2}\frac{\partial}{\partial w_j}\sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

$$= \frac{1}{2}\sum_i 2(y^{(i)} - \phi(z^{(i)}))\frac{\partial}{\partial w_j}(y^{(i)} - \phi(z^{(i)}))$$

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

$$= \sum_i (y^{(i)} - \phi(z^{(i)}))(-x_j^{(i)})$$

$$= -\sum_i (y^{(i)} - \phi(z^{(i)})x_j^{(i)}$$

Adaline: updating the weights

- All the weights are updated simultaneously by the Adaline learning rule:

$$\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$$

- The updating of the weights is obtained by taking a step in the opposite direction of the gradient $\nabla J(\mathbf{w})$ of the cost function $J(\mathbf{w})$ since the value $\Delta\mathbf{w}$ is updated by using the negative gradient multiplied by the learning rate:

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

- As a consequence, the the update the weight w_j is:

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

Adaptive Linear Neurons: Adaline

```
class AdalineGD(object):
    """ADAptive LInear NEuron classifier.
    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.
    random_state : int
        Random number generator seed for random weight initialization.

    Attributes
    ------------
    w_ : 1d-array
        Weights after fitting.
    cost_ : list
        Sum-of-squares cost function value in each epoch.
"""

def __init__(self, eta=0.01, n_iter=50, random_state=1):
    self.eta = eta
    self.n_iter = n_iter
    self.random_state = random_state
```

Adaptive Linear Neurons: Adaline

```
1  class AdalineGD(object):
2      """ADAptive LInear NEuron classifier.
3      Parameters
4      ------------
5      eta : float
6          Learning rate (between 0.0 and 1.0)
7      n_iter : int
8          Passes over the training dataset.
9      random_state : int
10         Random number generator seed for random weight initialization.
11
12     Attributes
13     ------------
14     w_ : 1d-array
15         Weights after fitting.
16     cost_ : list
17         Sum-of-squares cost function value in each epoch.
18 """
19
20 def __init__(self, eta=0.01, n_iter=50, random_state=1):
21     self.eta = eta
22     self.n_iter = n_iter
23     self.random_state = random_state
```

Adaptive Linear Neurons: Adaline

```
1  class AdalineGD(object):
2      """ADAptive LInear NEuron classifier.
3      Parameters
4      ------------
5      eta : float
6          Learning rate (between 0.0 and 1.0)
7      n_iter : int
8          Passes over the training dataset.
9      random_state : int
10         Random number generator seed for random weight initialization.
11
12     Attributes
13     ------------
14     w_ : 1d-array
15         Weights after fitting.
16     cost_ : list
17         Sum-of-squares cost function value in each epoch.
18 """
19
20 def __init__(self, eta=0.01, n_iter=50, random_state=1):
21     self.eta = eta
22     self.n_iter = n_iter
23     self.random_state = random_state
```

Adaptive Linear Neurons: Adaline

```
class AdalineGD(object):
    """ADAptive LInear NEuron classifier.
    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.
    random_state : int
        Random number generator seed for random weight initialization.

    Attributes
    ------------
    w_ : 1d-array
        Weights after fitting.
    cost_ : list
        Sum-of-squares cost function value in each epoch.
"""

def __init__(self, eta=0.01, n_iter=50, random_state=1):
    self.eta = eta
    self.n_iter = n_iter
    self.random_state = random_state
```

Adaptive Linear Neurons: Adaline

```
25  def fit(self, X, y):
26      """ Fit training data.
27      Parameters
28      ----------
29      X : {array-like}, shape = [n_examples, n_features]
30          Training vectors, where n_examples
31          is the number of examples and
32          n_features is the number of features.
33      y : array-like, shape = [n_examples]
34          Target values.
35
36      Returns
37      ----------
38      self : object
39
40      """
41      rgen = np.random.RandomState(self.random_state)
42      self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
43      self.cost_ = []
44
45      for i in range(self.n_iter):
46          net_input = self.net_input(X)
47          output = self.activation(net_input) errors = (y - output)
48          self.w_[1:] += self.eta * X.T.dot(errors)
49          self.w_[0] += self.eta * errors.sum()
50          cost = (errors**2).sum() / 2.0
```

Adaptive Linear Neurons: Adaline

```
25  def fit(self, X, y):
26      """ Fit training data.
27      Parameters
28      ----------
29      X : {array-like}, shape = [n_examples, n_features]
30          Training vectors, where n_examples
31          is the number of examples and
32          n_features is the number of features.
33      y : array-like, shape = [n_examples]
34          Target values.
35
36      Returns
37      ----------
38      self : object
39
40      """
41      rgen = np.random.RandomState(self.random_state)
42      self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
43      self.cost_ = []
44
45      for i in range(self.n_iter):
46          net_input = self.net_input(X)
47          output = self.activation(net_input) errors = (y - output)
48          self.w_[1:] += self.eta * X.T.dot(errors)
49          self.w_[0] += self.eta * errors.sum()
50          cost = (errors**2).sum() / 2.0
```

Adaptive Linear Neurons: Adaline

```
25  def fit(self, X, y):
26      """ Fit training data.
27      Parameters
28      ----------
29      X : {array-like}, shape = [n_examples, n_features]
30          Training vectors, where n_examples
31          is the number of examples and
32          n_features is the number of features.
33      y : array-like, shape = [n_examples]
34          Target values.
35
36      Returns
37      ----------
38      self : object
39
40      """
41      rgen = np.random.RandomState(self.random_state)
42      self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
43      self.cost_ = []
44
45      for i in range(self.n_iter):
46          net_input = self.net_input(X)
47          output = self.activation(net_input) errors = (y - output)
48          self.w_[1:] += self.eta * X.T.dot(errors)
49          self.w_[0] += self.eta * errors.sum()
50          cost = (errors**2).sum() / 2.0
```

Adaptive Linear Neurons: Adaline

```
51              self.cost_.append(cost)
52          return self
53
54      def net_input(self, X):
55          """Calculate net input"""
56          return np.dot(X, self.w_[1:]) + self.w_[0]
57
58      def activation(self, X):
59          """Compute linear activation"""
60          return X
61
62      def predict(self, X):
63          """Return class label after unit step"""
64          return np.where(self.activation(self.net_input(X)) >= 0.0, 1, -1)
```

Adaptive Linear Neurons: Adaline

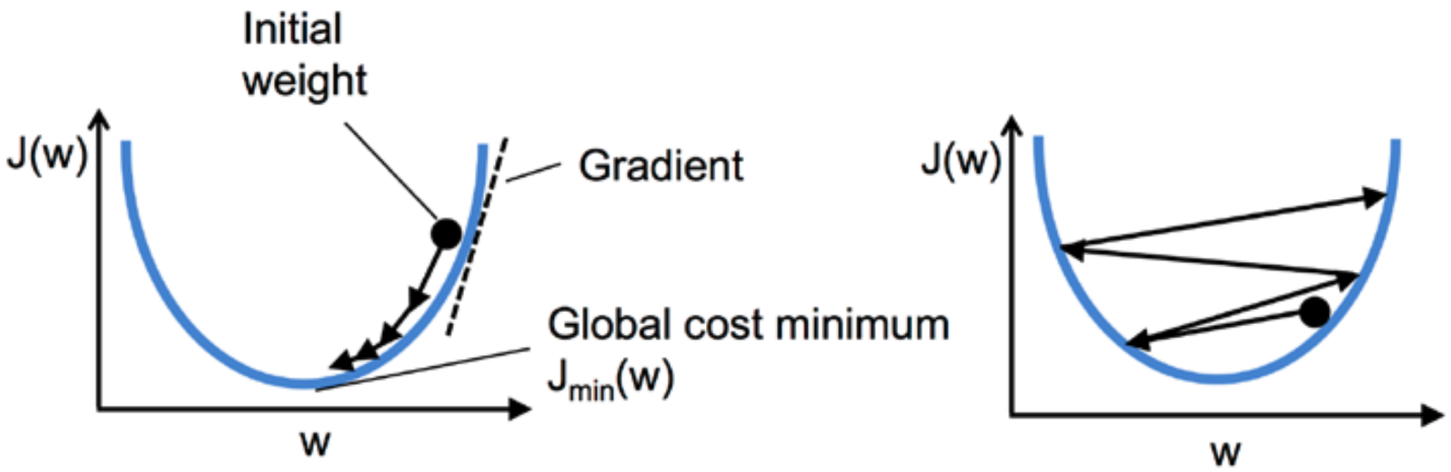
```
51              self.cost_.append(cost)
52          return self
53
54      def net_input(self, X):
55          """Calculate net input"""
56          return np.dot(X, self.w_[1:]) + self.w_[0]
57
58      def activation(self, X):
59          """Compute linear activation"""
60          return X
61
62      def predict(self, X):
63          """Return class label after unit step"""
64          return np.where(self.activation(self.net_input(X)) >= 0.0, 1, -1)
```

Adaptive Linear Neurons: Adaline

```
51              self.cost_.append(cost)
52          return self
53
54      def net_input(self, X):
55          """Calculate net input"""
56          return np.dot(X, self.w_[1:]) + self.w_[0]
57
58      def activation(self, X):
59          """Compute linear activation"""
60          return X
61
62      def predict(self, X):
63          """Return class label after unit step"""
64          return np.where(self.activation(self.net_input(X)) >= 0.0, 1, -1)
```

Adaptive Linear Neurons: Adaline

The following figure illustrates what might happen if we change the value of a particular weight parameter to minimize the cost function, J. The left subfigure illustrates the case of a well-chosen learning rate, where the cost decreases gradually, moving in the direction of the global minimum. The subfigure on the right, however, illustrates what happens if we choose a learning rate that is too large—we overshoot the global minimum:



Adaptive Linear Neurons: Adaline

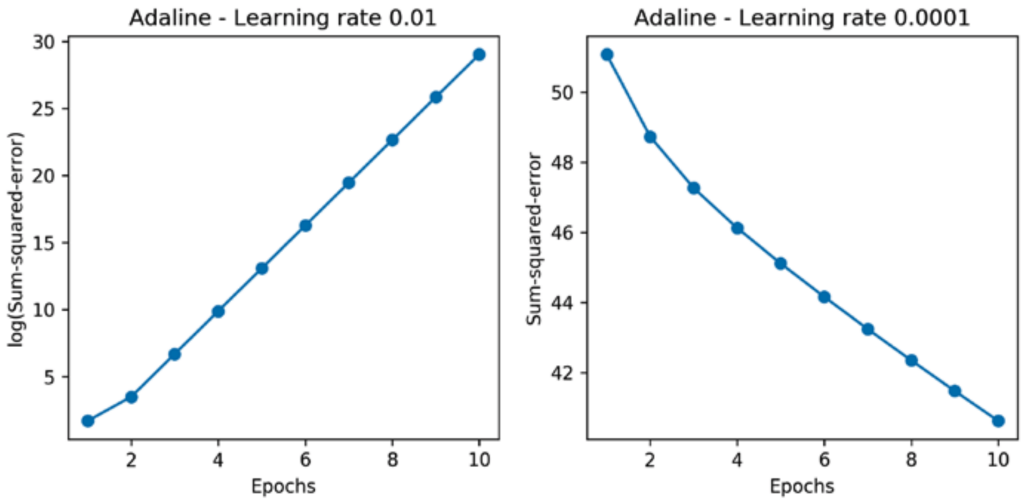
- In practice, it often requires some experimentation to find a good learning rate, η, for optimal convergence. So, let's choose two different learning rates, $\eta = 0.1$ and $\eta = 0.0001$, to start with and plot the cost functions versus the number of epochs to see how well the Adaline implementation learns from the training data.
- In the next lectures, methods for hyper-parameters tuning will be introduced.

Let's now plot the cost against the number of epochs for the two different learning rates.

Adaptive Linear Neurons: Adaline

```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
>>> ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
>>> ax[0].plot(range(1, len(ada1.cost_) + 1), np.log10(ada1.cost_), marker='o')
>>> ax[0].set_xlabel('Epochs')
>>> ax[0].set_ylabel('log(Sum-squared-error)')
>>> ax[0].set_title('Adaline - Learning rate 0.01')

>>> ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
>>> ax[1].plot(range(1, len(ada2.cost_) + 1), ada2.cost_, marker='o')
>>> ax[1].set_xlabel('Epochs')
>>> ax[1].set_ylabel('Sum-squared-error')
>>> ax[1].set_title('Adaline - Learning rate 0.0001')
>>> plt.show()
```



Batch Gradient Descent

- In the first variant of Adaline, the gradient descent works by computing the opposite direction of a cost gradient that is calculated from the whole training dataset. For this reasons, this approach is also referred to as **batch gradient descent**.
- If we have a very large dataset with millions of data points, which is not uncommon in many machine learning applications, **running batch gradient descent can be computationally quite costly** (evaluating the sum-gradient may require expensive evaluations of the gradients from all summing functions).
- Two other variants:
  - Stochastic gradient descent
  - Mini-batch gradient descent

Stochastic Gradient Descent

- Instead of updating the weights based on the sum of the accumulated errors over all training examples:

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

  the weights are incrementally updated for each training example as follows:

$$\eta(y^{(i)} - \phi(z^{(i)}))\mathbf{x}^{(i)}$$

- It is necessary to **shuffle** training examples at each epoch in order to avoid cycle.
- Often, the learning rate η is computed in an adaptive way:

$$\eta = \frac{c_1}{[\textit{number of iterations}] + c_2}$$

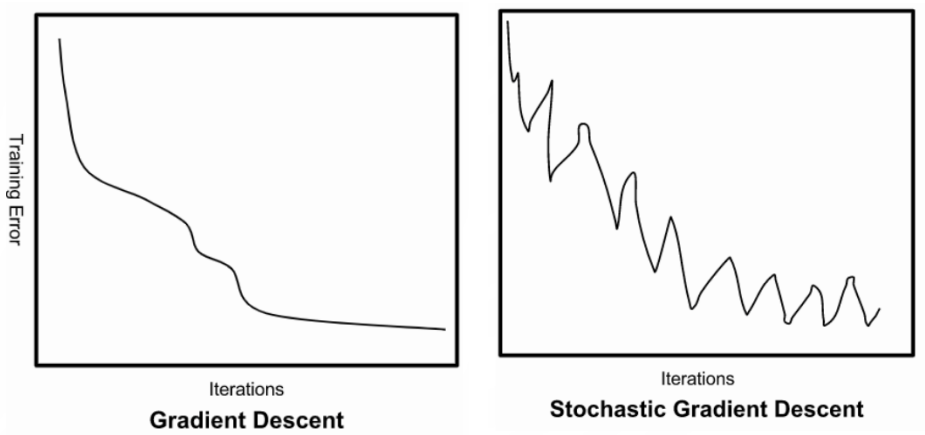
Stochastic Gradient Descent: pros and cons

Prons:

- By using a single example for each update, the computation is simpler, as a consequence, SGD is usually **much faster**.
- Stochastic gradient descent is used to implement the **online learning**: updating our model online, i.e. with new examples on-the-fly.

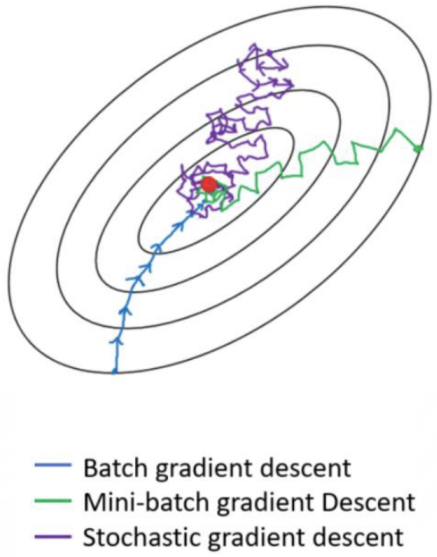
Cons:

- Due to its stochastic (random) approach, this algorithm is **less regular** than the previous one: the objective function fluctuates heavily on one example at each learning step.



Mini-batch Gradient Descent

- The **mini-batch learning** is a compromise between classical batch gradient descent and stochastic gradient descent.
- Mini-batch learning can be understood as applying batch gradient descent to smaller subsets of the training data, for example, 32 training examples at a time.
- This algorithm reduces the noise occurred in the Stochastic Gradient Descent and still is much faster than Batch Gradient Descent.



Stochastic Gradient Descent

Only few adjustments are required to modify the Adaline learning algorithm and update the weights via SGD.

- Inside the `fit` method, a modification is required to update the weights after each training example;
- An additional partial_fit method is introduced, which does not reinitialize the weights, for online learning;
- In order to check whether our algorithm converged after training, the cost is computed as the average cost of the training examples in each epoch;
- Finally, an option to shuffle the training data before each epoch is added to avoid repetitive cycles when the cost function is optimizing.

Stochastic Gradient Descent

```
1  class AdalineSGD(object):
2      """ADAptive LInear NEuron classifier.
3
4      Parameters
5      ------------
6      eta : float
7          Learning rate (between 0.0 and 1.0)
8      n_iter : int
9          Passes over the training dataset.
10     shuffle : bool (default: True)
11         Shuffles training data every epoch if True to prevent cycles.
12     random_state : int
13         Random number generator seed for random weight initialization.
14
15     Attributes
16     -----------
17     w_ : 1d-array
18         Weights after fitting.
19     cost_ : list
20         Sum-of-squares cost function value averaged over
21         all training examples in each epoch.
22     """
23
24 def __init__(self, eta=0.01, n_iter=10, shuffle=True, random_state=None):
25     self.eta = eta
26     self.n_iter = n_iter
27     self.w_initialized = False
28     self.shuffle = shuffle
29     self.random_state = random_state
```

Stochastic Gradient Descent

```
1  class AdalineSGD(object):
2      """ADAptive LInear NEuron classifier.
3
4      Parameters
5      ------------
6      eta : float
7          Learning rate (between 0.0 and 1.0)
8      n_iter : int
9          Passes over the training dataset.
10     shuffle : bool (default: True)
11         Shuffles training data every epoch if True to prevent cycles.
12     random_state : int
13         Random number generator seed for random weight initialization.
14
15     Attributes
16     ------------
17     w_ : 1d-array
18         Weights after fitting.
19     cost_ : list
20         Sum-of-squares cost function value averaged over
21         all training examples in each epoch.
22     """
23
24 def __init__(self, eta=0.01, n_iter=10, shuffle=True, random_state=None):
25     self.eta = eta
26     self.n_iter = n_iter
27     self.w_initialized = False
28     self.shuffle = shuffle
29     self.random_state = random_state
```

Stochastic Gradient Descent

```
1  class AdalineSGD(object):
2      """ADAptive LInear NEuron classifier.
3
4      Parameters
5      ------------
6      eta : float
7          Learning rate (between 0.0 and 1.0)
8      n_iter : int
9          Passes over the training dataset.
10     shuffle : bool (default: True)
11         Shuffles training data every epoch if True to prevent cycles.
12     random_state : int
13         Random number generator seed for random weight initialization.
14
15     Attributes
16     ------------
17     w_ : 1d-array
18         Weights after fitting.
19     cost_ : list
20         Sum-of-squares cost function value averaged over
21         all training examples in each epoch.
22     """
23
24 def __init__(self, eta=0.01, n_iter=10, shuffle=True, random_state=None):
25     self.eta = eta
26     self.n_iter = n_iter
27     self.w_initialized = False
28     self.shuffle = shuffle
29     self.random_state = random_state
```

Stochastic Gradient Descent

```
class AdalineSGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.
    shuffle : bool (default: True)
        Shuffles training data every epoch if True to prevent cycles.
    random_state : int
        Random number generator seed for random weight initialization.

    Attributes
    ------------
    w_ : 1d-array
        Weights after fitting.
    cost_ : list
        Sum-of-squares cost function value averaged over
        all training examples in each epoch.
    """

def __init__(self, eta=0.01, n_iter=10, shuffle=True, random_state=None):
    self.eta = eta
    self.n_iter = n_iter
    self.w_initialized = False
    self.shuffle = shuffle
    self.random_state = random_state
```

Stochastic Gradient Descent

```
31  def fit(self, X, y):
32      """ Fit training data.
33      Parameters
34      ----------
35      X : {array-like}, shape = [n_examples, n_features]
36          Training vectors, where n_examples is the number of
37          examples and n_features is the number of features.
38      y : array-like, shape = [n_examples]
39          Target values.
40
41      Returns
42      -------
43      self : object
44      """
45
46      self._initialize_weights(X.shape[1])
47      self.cost_ = []
48      for i in range(self.n_iter):
49          if self.shuffle:
50              X, y = self._shuffle(X, y)
51          cost = []
52          for xi, target in zip(X, y):
53              cost.append(self._update_weights(xi, target))
54          avg_cost = sum(cost) / len(y)
55          self.cost_.append(avg_cost)
56      return self
```

Stochastic Gradient Descent

```
58  def partial_fit(self, X, y):
59      """Fit training data without reinitializing the weights"""
60      if not self.w_initialized:
61          self._initialize_weights(X.shape[1])
62      if y.ravel().shape[0] > 1:
63          for xi, target in zip(X, y):
64              self._update_weights(xi, target)
65      else:
66          self._update_weights(X, y)
67      return self
68
69  def _shuffle(self, X, y):
70      """Shuffle training data"""
71      r = self.rgen.permutation(len(y))
72
73      return X[r], y[r]
74
75  def _initialize_weights(self, m):
76      """Initialize weights to small random numbers"""
77      self.rgen = np.random.RandomState(self.random_state)
78      self.w_ = self.rgen.normal(loc=0.0, scale=0.01, size=1 + m)
79      self.w_initialized = True
```

Stochastic Gradient Descent

```
58  def partial_fit(self, X, y):
59      """Fit training data without reinitializing the weights"""
60      if not self.w_initialized:
61          self._initialize_weights(X.shape[1])
62      if y.ravel().shape[0] > 1:
63          for xi, target in zip(X, y):
64              self._update_weights(xi, target)
65      else:
66          self._update_weights(X, y)
67      return self
68
69  def _shuffle(self, X, y):
70      """Shuffle training data"""
71      r = self.rgen.permutation(len(y))
72
73      return X[r], y[r]
74
75  def _initialize_weights(self, m):
76      """Initialize weights to small random numbers"""
77      self.rgen = np.random.RandomState(self.random_state)
78      self.w_ = self.rgen.normal(loc=0.0, scale=0.01, size=1 + m)
79      self.w_initialized = True
```

Stochastic Gradient Descent

```
58  def partial_fit(self, X, y):
59      """Fit training data without reinitializing the weights"""
60      if not self.w_initialized:
61          self._initialize_weights(X.shape[1])
62      if y.ravel().shape[0] > 1:
63          for xi, target in zip(X, y):
64              self._update_weights(xi, target)
65      else:
66          self._update_weights(X, y)
67      return self
68
69  def _shuffle(self, X, y):
70      """Shuffle training data"""
71      r = self.rgen.permutation(len(y))
72
73      return X[r], y[r]
74
75  def _initialize_weights(self, m):
76      """Initialize weights to small random numbers"""
77      self.rgen = np.random.RandomState(self.random_state)
78      self.w_ = self.rgen.normal(loc=0.0, scale=0.01, size=1 + m)
79      self.w_initialized = True
```

Stochastic Gradient Descent

```
81   def _update_weights(self, xi, target):
82       """Apply Adaline learning rule to update the weights"""
83       output = self.activation(self.net_input(xi))
84       error = (target - output)
85       self.w_[1:] += self.eta * xi.dot(error)
86       self.w_[0] += self.eta * error
87       cost = 0.5 * error**2
88       return cost
89
90   def net_input(self, X):
91       """Calculate net input"""
92       return np.dot(X, self.w_[1:]) + self.w_[0]
93
94   def activation(self, X):
95       """Compute linear activation"""
96       return X
97
98   def predict(self, X):
99       """Return class label after unit step"""
100      return np.where(self.activation(self.net_input(X)) >= 0.0, 1, -1)
```

Stochastic Gradient Descent

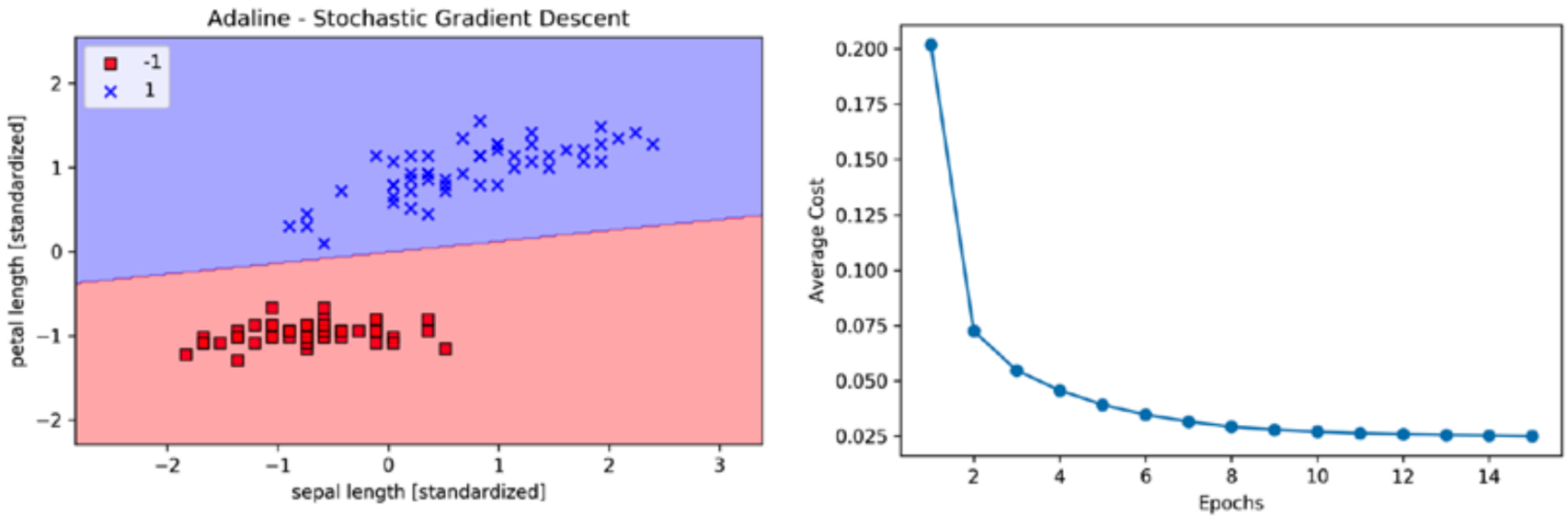
The fit method is used to train the AdalineSGD classifier and use our `plot_decision_regions` to plot our training results:

```
>>> ada_sgd = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada_sgd.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier =ada_sgd)
>>> plt.title('Adaline - Stochastic Gradient Descent')
>>> plt.xlabel('sepal length [standardized]')

>>> plt.ylabel('petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
>>> plt.plot(range(1, len(ada_sgd.cost_) + 1), ada_sgd.cost_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Average Cost')
>>> plt.tight_layout()
>>> plt.show()
```

Stochastic Gradient Descent

The two plots that we obtain from executing the preceding code example are shown in the following figure:



Feature Scaling

Feature Scaling

Feature data can have different scales and ranges. This can be a **problem for gradient descent**:

- The **weights updated is proportional to feature value**, so, with features being on different scales, certain weights may update faster than others
- **It is difficult to select the most suitable learning rate value**: if we choice the value based on the input value having the smallest range, small learning rate it takes ages for the large range to converge; on the contrary, if we choice high value for learning rate, the gradient descent might not converge for small ranges.

Feature scaling is a method used to adjust the range of features of data.

Feature Scaling: Normalization

Normalization is a scaling technique in which values are shifted and rescaled so that they end up ranging between 0 and 1. The general formula for a **min-max** of [0, 1] is given as:

$$x' = \frac{x - min(\mathbf{x})}{max(\mathbf{x}) - min(\mathbf{x})}$$

where x' is the normalized value and x is the original one. To rescale a range between an arbitrary set of values [a, b], the formula becomes:

$$x' = a + \frac{(x - min(\mathbf{x}))(b - a)}{max(\mathbf{x}) - min(\mathbf{x})}$$

Another form of normalization is called **mean-normalization**. The formula is:

$$x' = \frac{x - average(\mathbf{x})}{max(\mathbf{x}) - min(\mathbf{x})}$$

Feature Scaling: Standardization

- Feature standardization makes the values of each feature in the data have zero-mean (when subtracting the mean in the numerator) and unit-variance.
- The general method of calculation is to determine the distribution mean and standard deviation for each feature. Next we subtract the mean from each feature. Then we divide the values (mean is already subtracted) of each feature by its standard deviation.
- Formally:

$$x' = \frac{x - \mu}{\sigma}$$

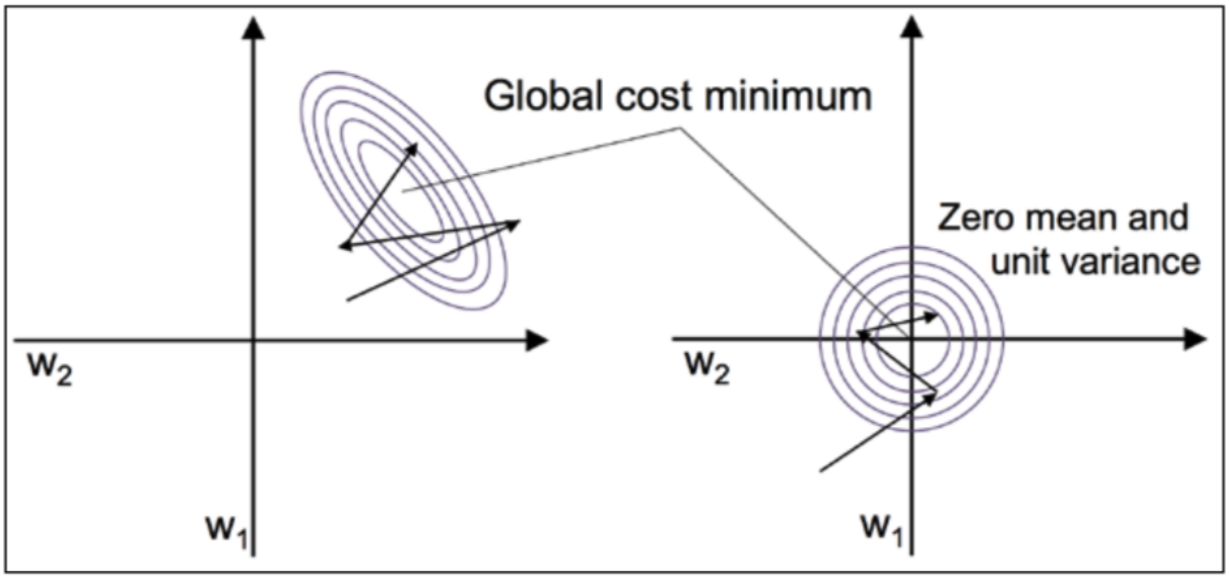
where x' is the standardized value, x is the original value, μ is the average of the values and σ is the standard deviation.

Feature Scaling: Standardization vs Normalization

- Although normalization via min-max scaling is a commonly used technique that is useful when we need values in a bounded interval, **standardization can be more practical** for many machine learning algorithms, especially for optimization algorithms such as gradient descent.
- Using standardization, we center the feature columns at mean 0 with standard deviation 1 so that the feature columns takes the form of a normal distribution, which makes it **easier to learn the weights**.
- Furthermore, standardization **maintains useful information about outliers** and makes the algorithm less sensitive to them in contrast to min-max scaling, which scales the data to a limited range of values.

Feature Scaling: Standardization

- One of the reasons why standardization helps with gradient descent learning is that the optimizer has to go through fewer steps to find a good or optimal solution (the global cost minimum), as illustrated in the following figure, where the subfigures represent the cost surface as a function of two model weights in a two-dimensional classification problem.



Feature scaling

Normalization and Standardization in Python

- You can perform the standardization and normalization manually by executing the following code examples:

```
>>> ex = np.array([0, 1, 2, 3, 4, 5])
>>> print('standardized:', (ex - ex.mean()) / ex.std())
standardized: [-1.46385011 -0.87831007 -0.29277002 0.29277002
0.87831007 1.46385011]
>>> print('normalized:', (ex - ex.min()) / (ex.max() - ex.min()))
normalized: [ 0.  0.2  0.4  0.6  0.8  1. ]
```

Feature scaling

Normalization and Standardization in Python

- You can perform the standardization and normalization manually by executing the following code examples:

```
>>> ex = np.array([0, 1, 2, 3, 4, 5])
>>> print('standardized:', (ex - ex.mean()) / ex.std())
standardized: [-1.46385011 -0.87831007 -0.29277002 0.29277002
0.87831007 1.46385011]
>>> print('normalized:', (ex - ex.min()) / (ex.max() - ex.min()))
normalized: [ 0.  0.2  0.4  0.6  0.8  1. ]
```

Feature Scaling in Python

Standardization can easily be achieved by using the built-in NumPy methods `mean` and `std`:

```
>>> 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()
```

After standardization, we will train Adaline again and we will see that it now converges after a small number of epochs using a learning rate of $\eta = 0.01$:

```
>>> 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.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.show()
```