Training Simple Machine Learning Algorithms for Classification

Hsi-Pin Ma 馬席彬

http://lms.nthu.edu.tw/course/40724

Department of Electrical Engineering

National Tsing Hua University

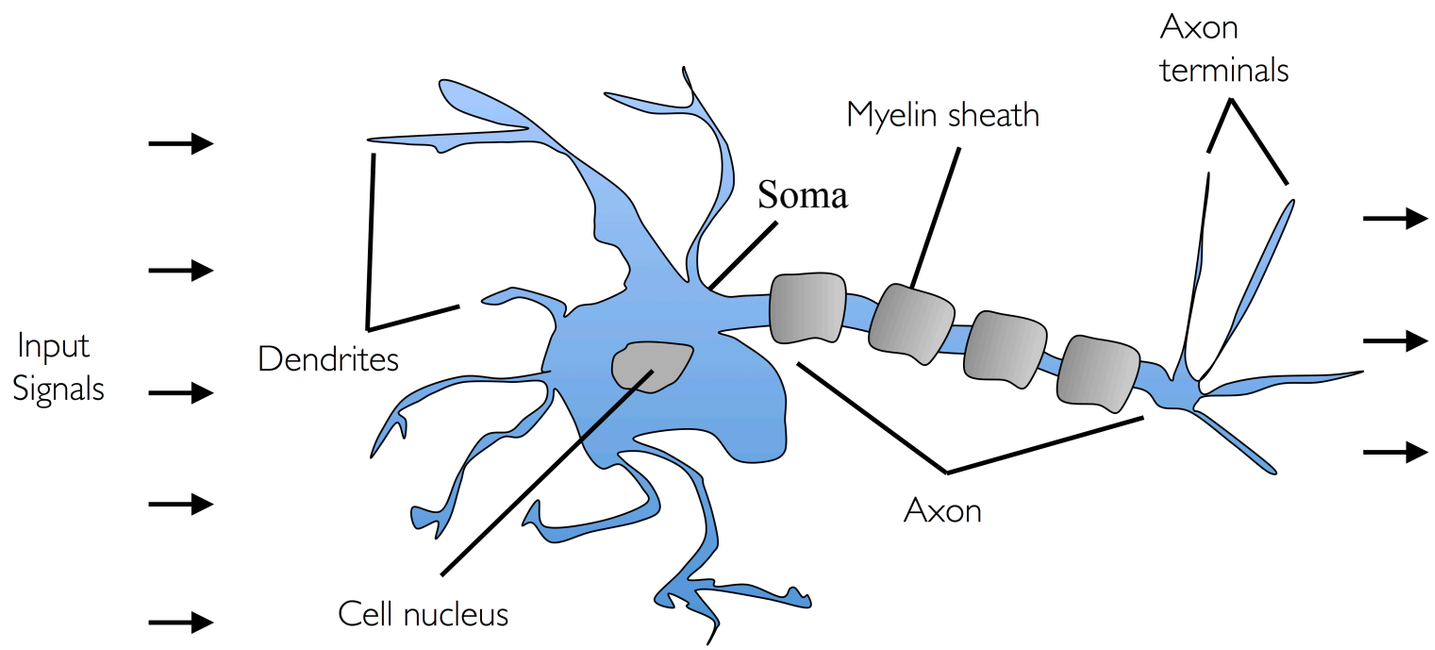
Outline

- Artificial Neurons and the Perceptron Algorithm
- Implementing a Perceptron Learning Algorithm in Python
- Adaptive Linear Neurons
- Implementing Adaline in Python
- Gradient Descent Improvements

Artificial Neurons and the Perceptron Algorithm

Artificial Neurons

- McCullock and Pitts (MCP) Neuron (1943)
  - Simple logic gate with binary output
  - Multiple signals arrive at the dendrites (multiple inputs) through synapses
  - Integrated into cell body (Soma)
  - If the accumulated signal exceeds a certain threshold, generate output, and pass on by the axon



Biological NN	Artificial NN
soma	neuron
axon, dendrite	connection
synapse	weight
potential	weighted sum
threshold	bias weight
signal	output

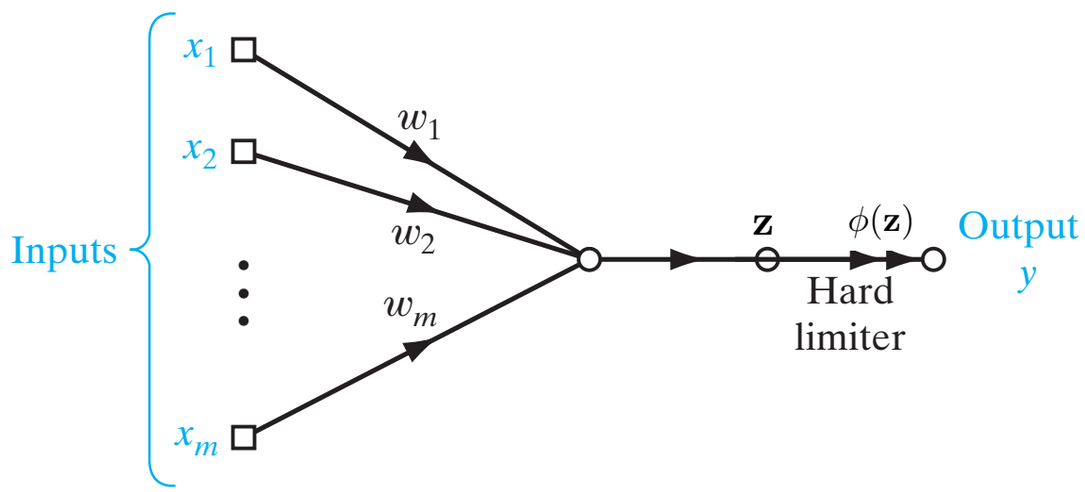
Formal Definition of an Artificial Neuron

- Binary classification task [Rosenblatt 1957]
  - Output: Positive class (1) vs. negative class (-1)
  - Input vector $\mathbf{x}$, synaptic weight vector $\mathbf{w}$, and integration (net input) $\mathbf{z}$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \quad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \quad \mathbf{z} = \mathbf{w}^{\mathbf{T}}\mathbf{x} = w_1x_1 + w_2x_2 + \ldots + w_mx_m$$

  - Define a decision (activation) function $\phi(\mathbf{z})$ and threshold

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

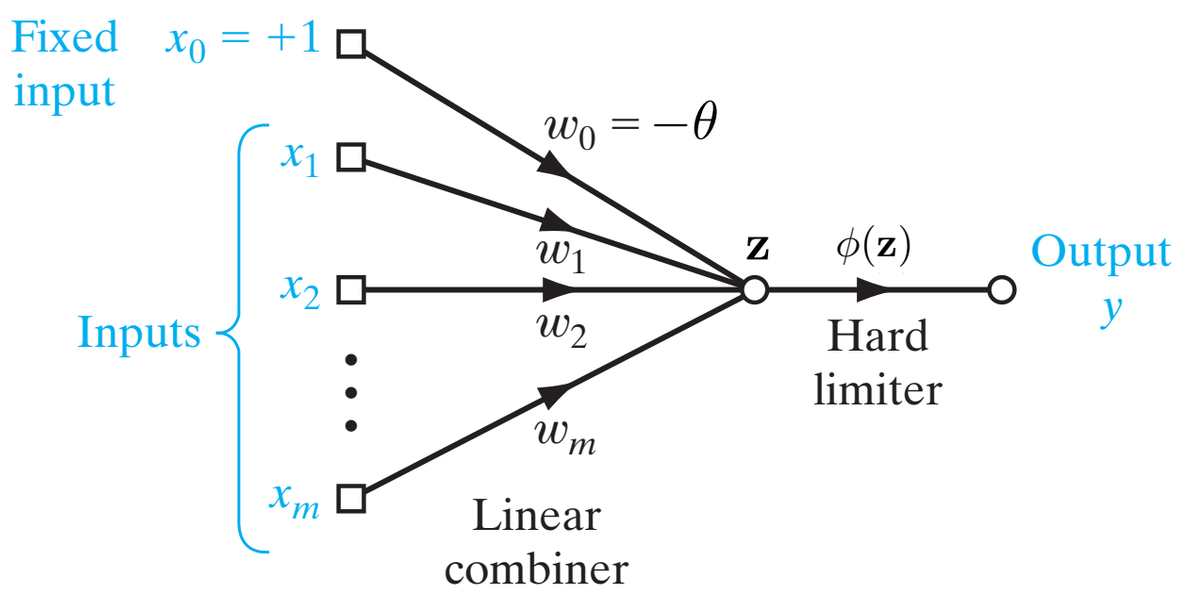


Formal Definition of an Artificial Neuron

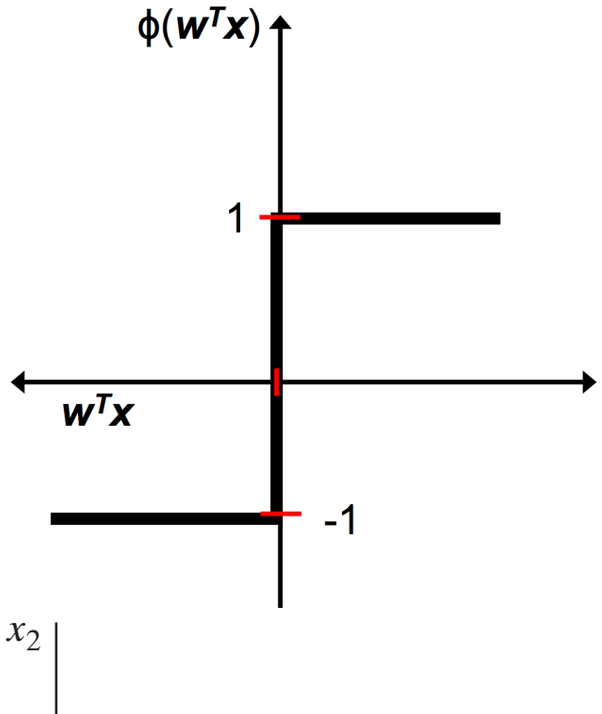
- Simplification
  - Bring the threshold to the left side of the equation and add $w_0 = -\theta$ (bias unit) and $x_0 = 1$

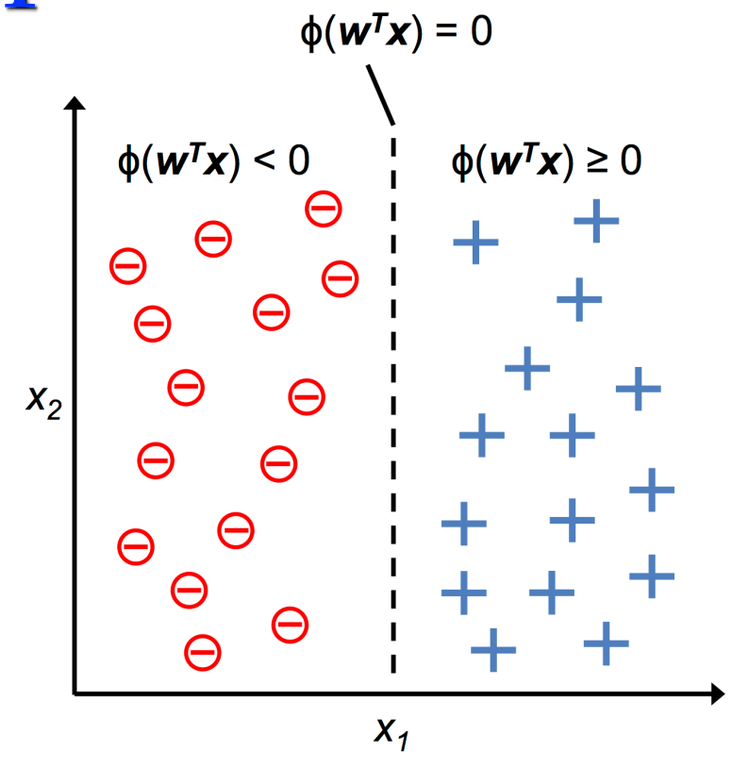
$$\mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \qquad \mathbf{w} = \begin{bmatrix} -\theta \\ w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$

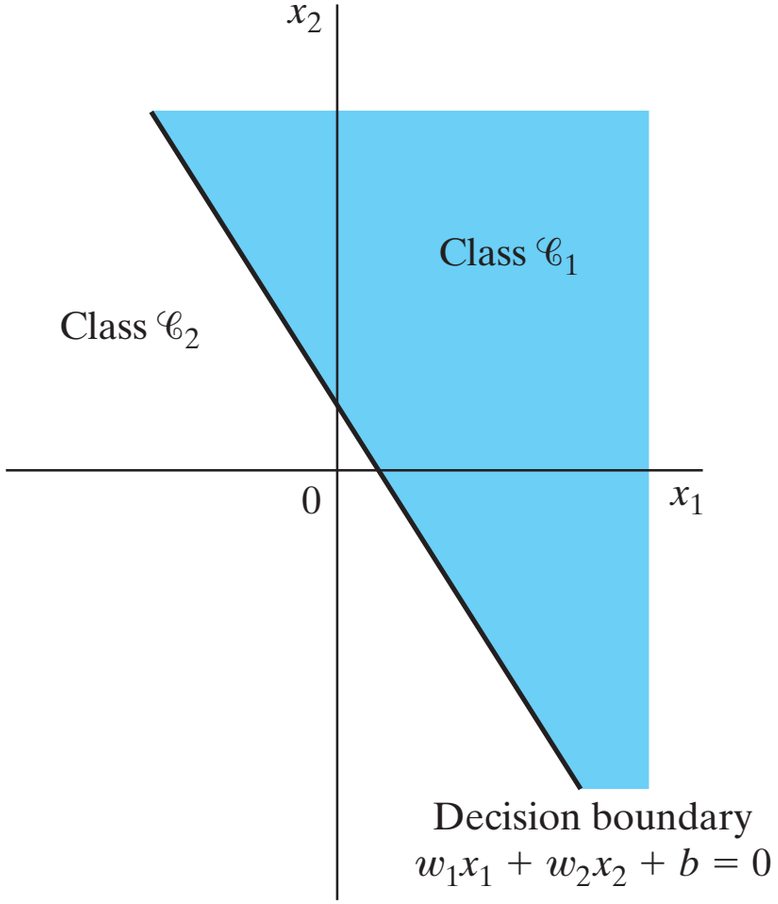
$$\phi(\mathbf{z}) = \mathrm{sgn}(\mathbf{z}) = \begin{cases} 1 & \text{if } \mathbf{z} \geq 0 \\ -1 & \text{if otherwise} \end{cases}$$



A Linear Separable Case







A decision boundary for
a 2D, 2-class pattern classification problem

Rosenblatt Perceptron Algorithm (1/2)

- The algorithm learns a linear hypothesis, i.e., a separating hyperplane, by processing training points one at a time.
- The algorithm maintains a weight vector $\mathbf{w}^{(i)} \in \mathbb{R}^M$ defining the hyperplane learned, starting with an arbitrary initial vector $\mathbf{w}^{(0)}$
- For each training sample $\mathbf{x}^{(i)}$, perform the following steps
  - Compute the prediction value $\hat{y}^{(i)}$ with current vector $\mathbf{w}^{(i)}$
  - Update the weights

$y^{(i)}$ **True class label**

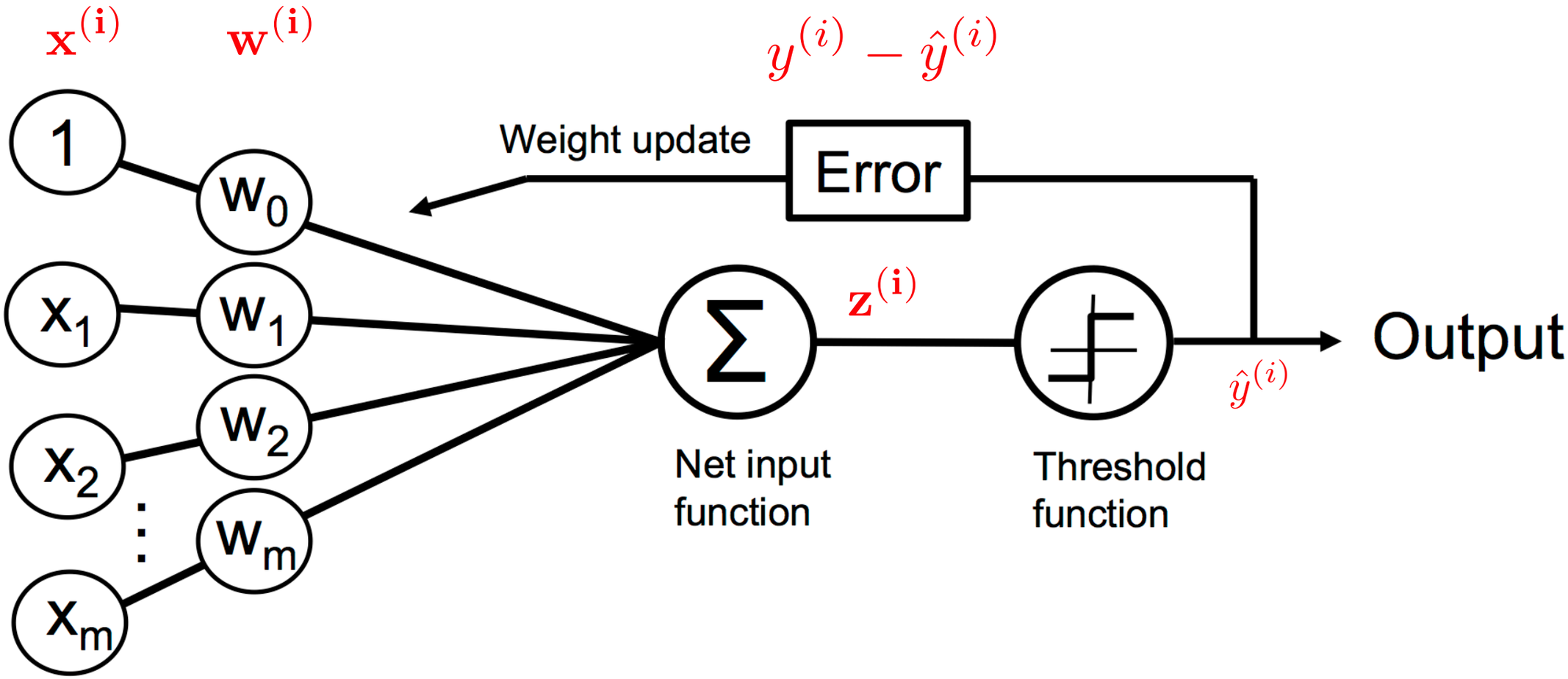
- Weight update rule

$$\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} + \Delta\mathbf{w}^{(i)}$$

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

Learning rate (0,1]

Rosenblatt Perceptron Algorithm (2/2)



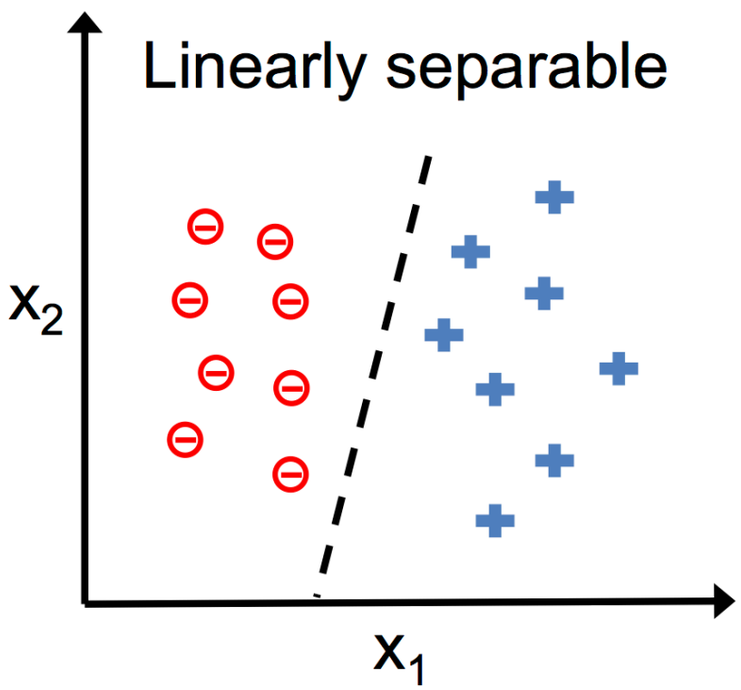
The Perceptron Algorithm

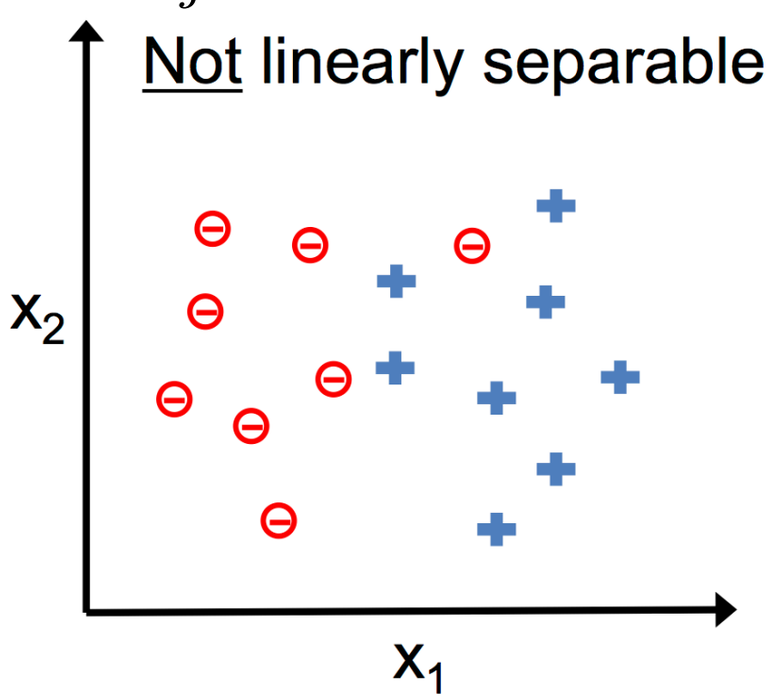
$\mathbf{w^{(1)}} \leftarrow \mathbf{w^{(0)}}$
for $\mathbf{i} \leftarrow 1$ **to** T **do**
    RECEIVE($\mathbf{x^{(i)}}$)
    $\hat{y}^{(\mathbf{i})} \leftarrow \text{sgn}(\mathbf{w^{(i)}} \cdot \mathbf{x^{(i)}})$
    RECEIVE($y^{(\mathbf{i})}$)
    if $(\hat{y}^{(\mathbf{i})} \neq y^{(\mathbf{i})})$ **then**
        $\mathbf{w^{(i+1)}} \leftarrow \mathbf{w^{(i)}} + \eta(\mathbf{y^{(i)}} - \mathbf{\hat{y}^{(i)}})\mathbf{x^{(i)}}, \eta > \mathbf{0}$
    else
        $\mathbf{w^{(i+1)}} \leftarrow \mathbf{w^{(i)}}$
return $\mathbf{w^{(T+1)}}$

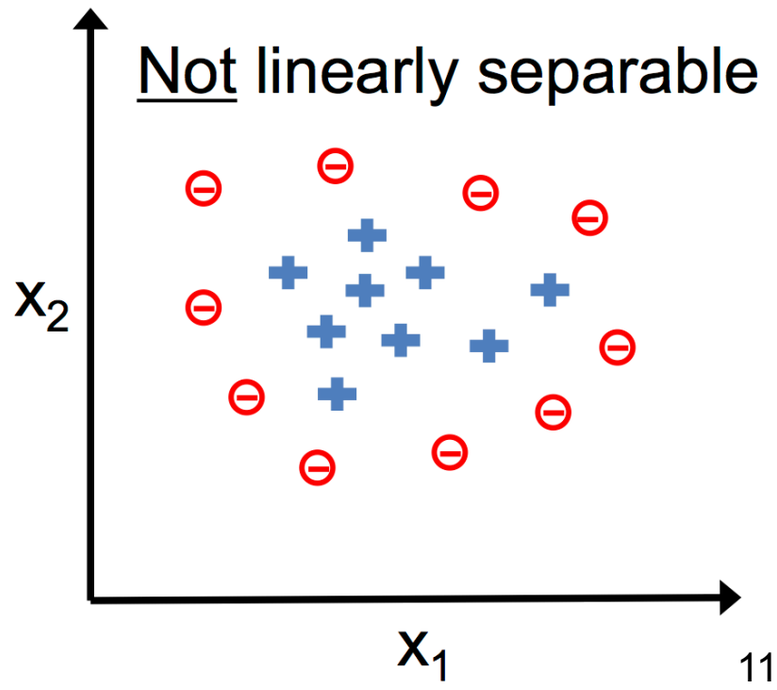
Convergence of the Perceptron Algorithm

- Guaranteed if the two classes in the training set is linear separable and the learning rate is sufficiently small
  - If two classes cannot be separated by a linear decision boundary, we can set a maximum ***number of epochs*** (passes over the training set) and/or a threshold for the number of tolerated misclassifications.
- The hyperplane as decision boundary is defined by

$$\sum_{j=1}^{m} w_j x_j - \theta = 0$$







Implementing a Perceptron Learning Algorithm in Python

An Object-Oriented Perceptron API

- Take an object-oriented approach to define the perceptron interface as a Python class
  - New *Perceptron* objects
  - Learn data from data via *fit* method
  - Make prediction via *predict* method
  - Append an underscore (_) to attributes that are not being created upon the initialization of the object
- Install the libraries first
  - NumPy, pandas, Matplotlib
- Next, see the ***jupyter notebook***

Training a Perceptron Model on the Iris Dataset

- Reading-in the Iris data
- Plotting the Iris data
- Training the perceptron model
- A function for plotting decision regions

- Also see the ***Jupiter notebook***

Perceptron API (1/3)

```
import numpy as np


class Perceptron(object):
    """Perceptron 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.
    errors_ : list
      Number of misclassifications (updates) 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
```

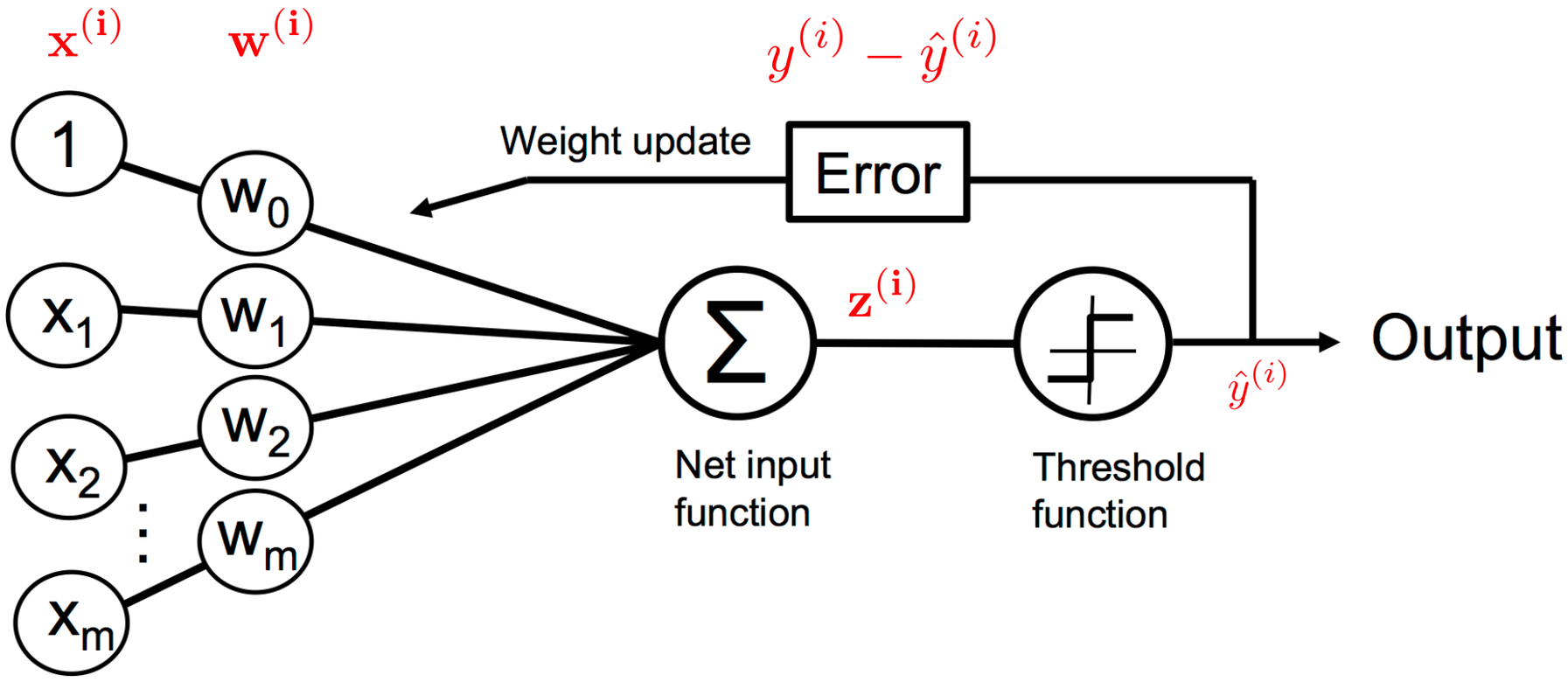
Perceptron API (2/3)

```
    def fit(self, X, y):
        """Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
          Training vectors, where n_samples is the number of samples and
          n_features is the number of features.
        y : array-like, shape = [n_samples]
          Target values.

        Returns
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self
```

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

Perceptron API (3/3)

```
    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```



Training a Perceptron Model on Iris Dataset

Reading-in the Data

```
import pandas as pd


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

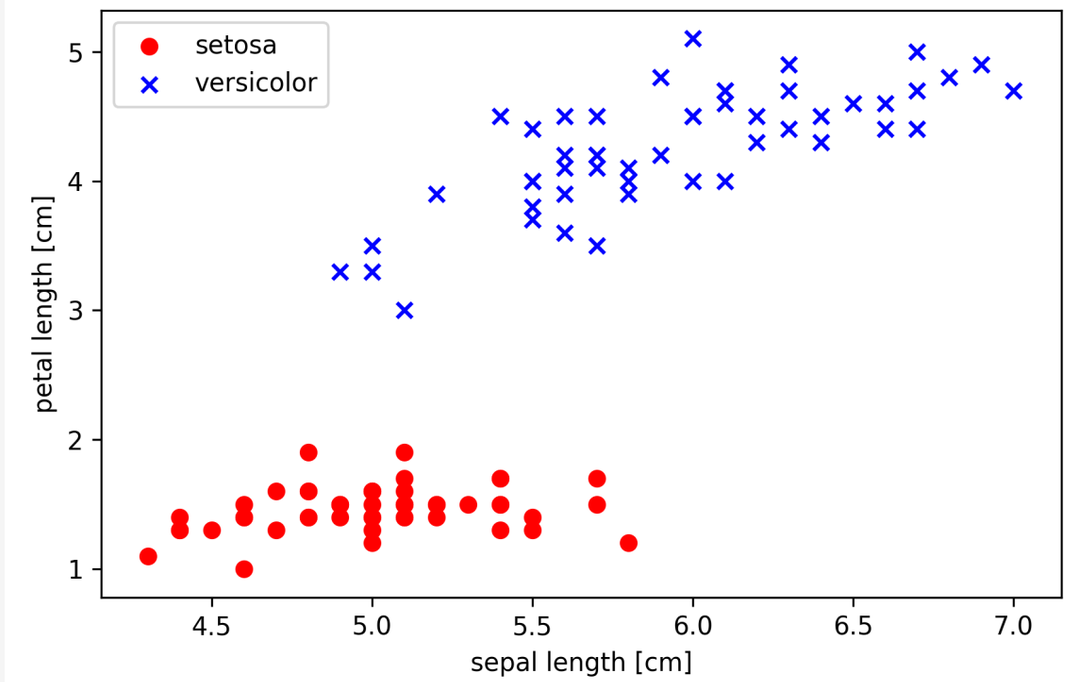
	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

```
df = pd.read_csv('your/local/path/to/iris.data', header=None)
```

Training a Perceptron Model on Iris Dataset

Plot the Data

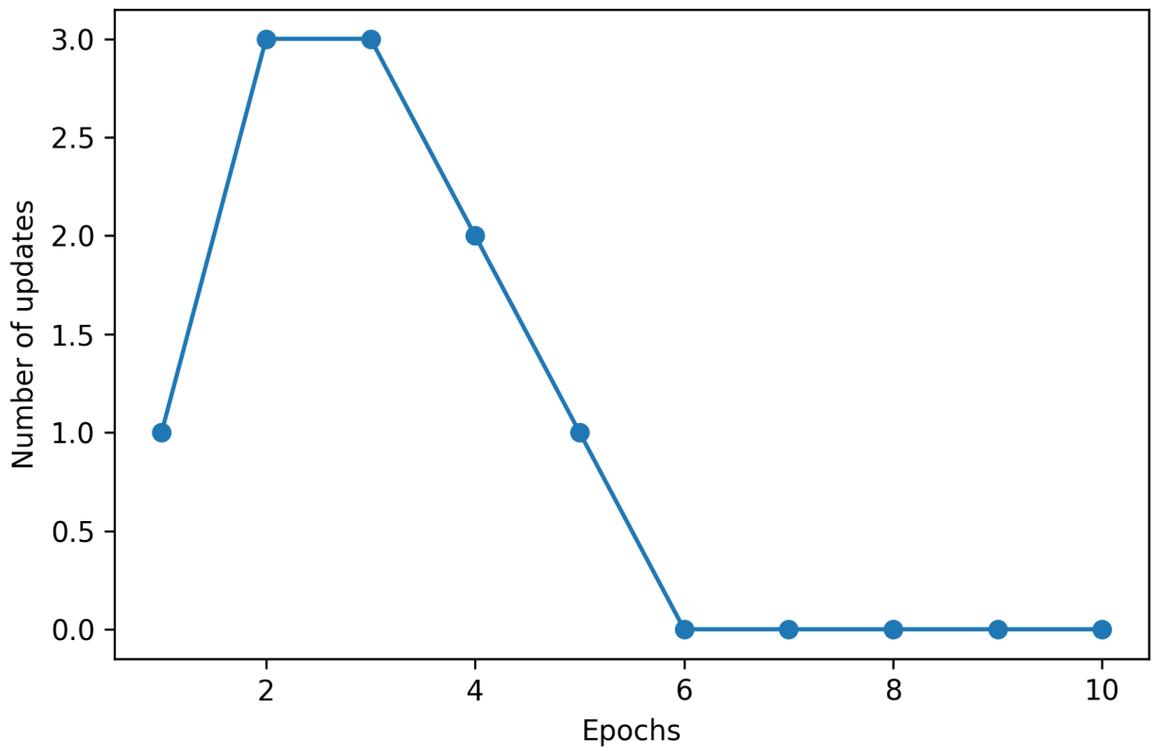
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

# 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 Iris Dataset

Training the Perceptron Model

```
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()
```

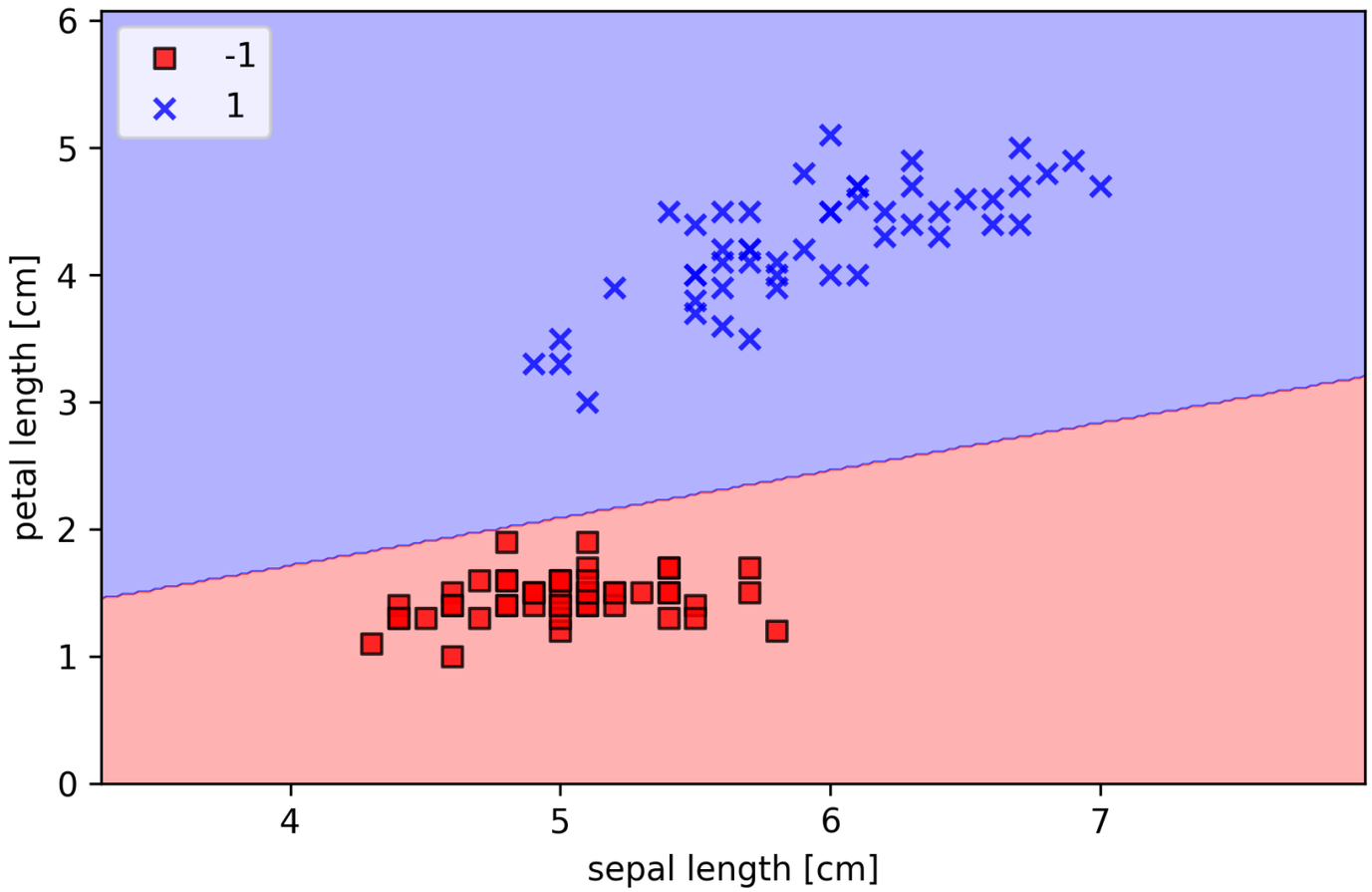


Training a Perceptron Model on Iris Dataset
Function to Plot the Decision Regions (1/2)

```
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 samples
    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')
```

Training a Perceptron Model on Iris Dataset
Function to Plot the Decision Regions (2/2)

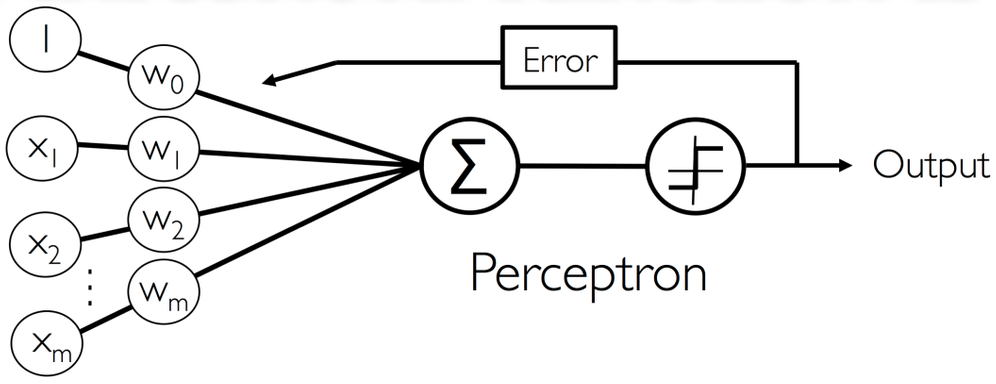
```
plot_decision_regions(X, y, classifier=ppn)
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')



# plt.savefig('images/02_08.png', dpi=300)
plt.show()
```



Adaptive Linear Neurons

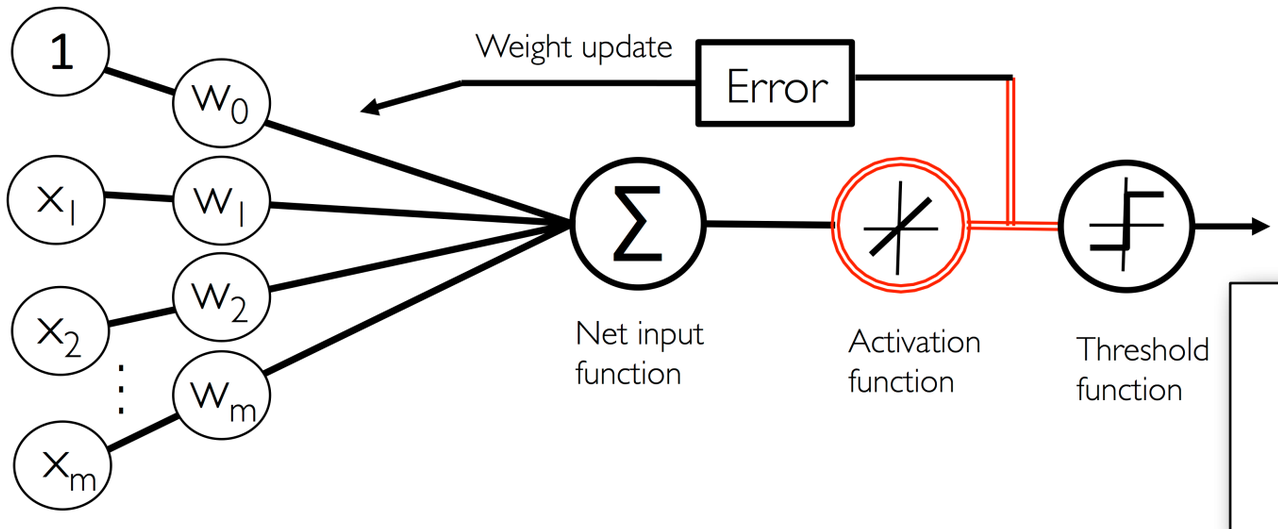
Widrow-Hoff (Adaline) Learning Rule (1960)

- Weights updated based on a linear activation function $\phi(\mathbf{w}^{\mathbf{T}}\mathbf{x}) = \mathbf{w}^{\mathbf{T}}\mathbf{x}$
  - In perceptron, weights updated by a sign function
- A threshold function is then used for prediction



Perceptron

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



Adaptive Linear Neuron (Adaline)

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

Cost Functions

- ML algorithms often define an objective function to be optimized during learning
- It is often a cost function we want to minimize
- Adaline uses a cost function as

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

  - Sum of Squared Errors (SSE)

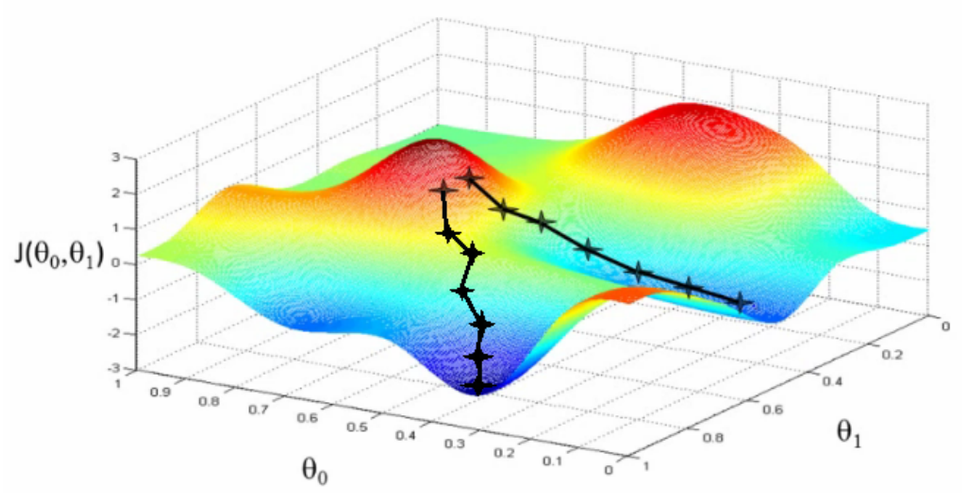
Advantages of Adaline Cost Function

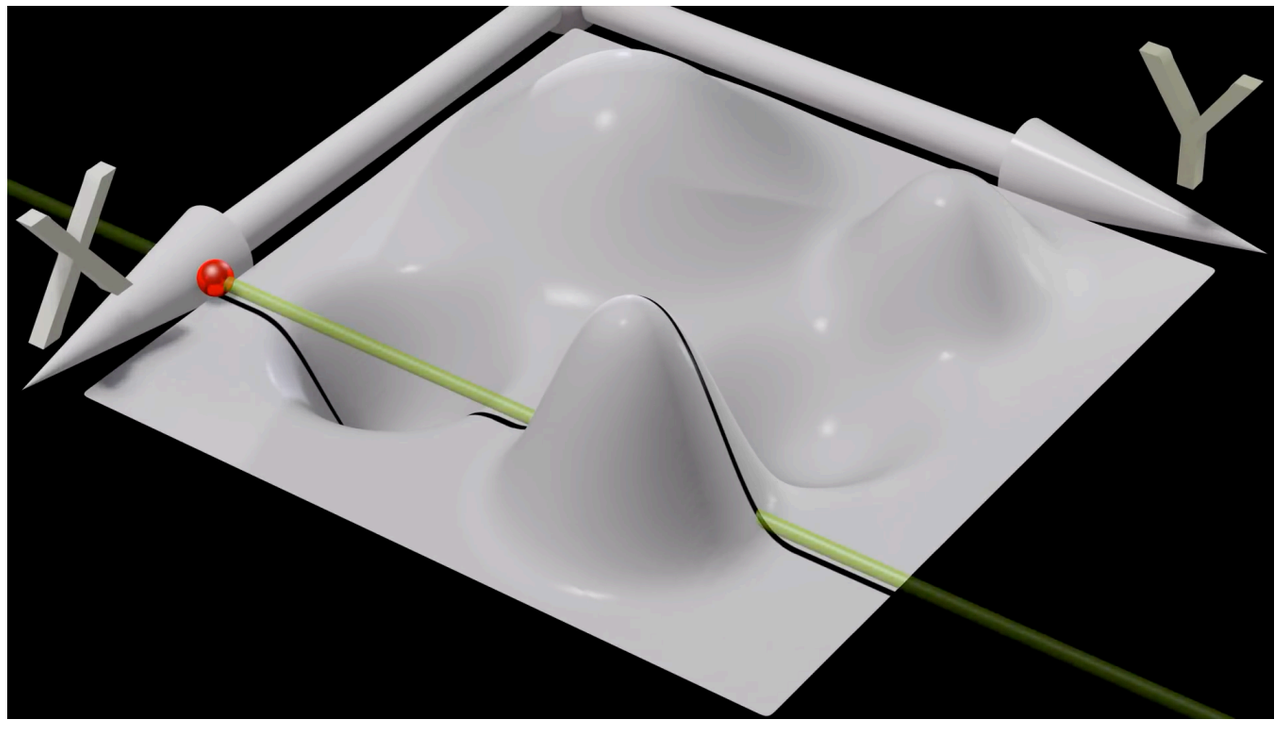
- The linear activation function is ***differentiable convex***
  - Unlike the unit step function in Perceptron
- Can use ***gradient descent*** to learn weights

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

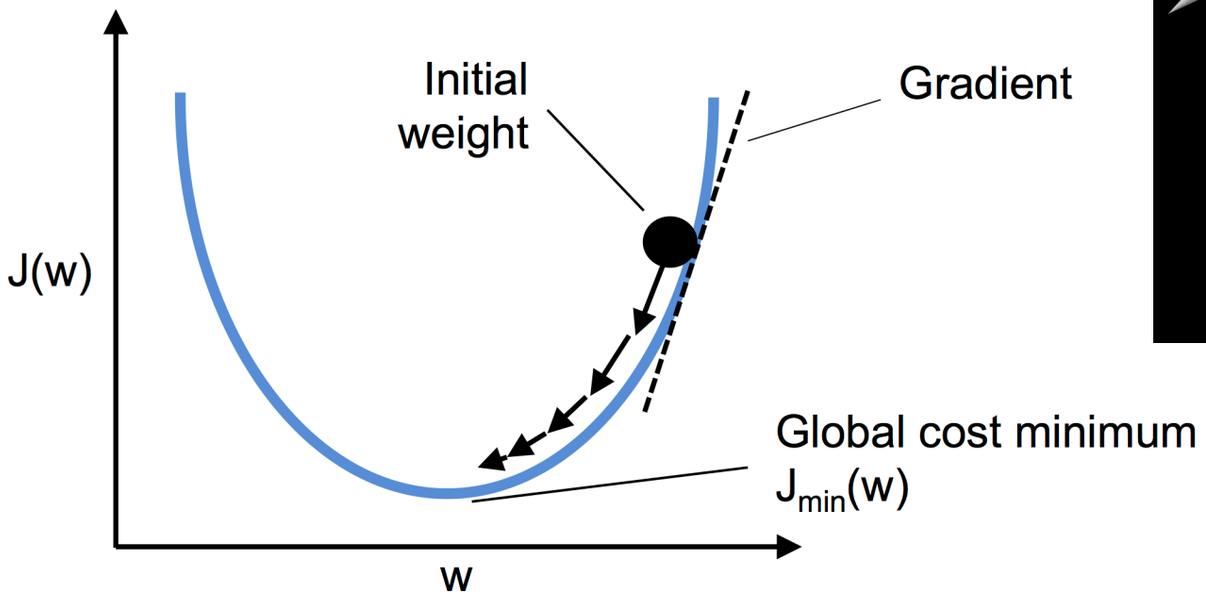
Gradient Descent

- Climbing down a hill until a local or global cost minimum is reached





[Eugene Khutoryansky]



Gradient Descent

- Weights updated by taking small steps
- Step size determined by learning rate
- Take a step away from the gradient $\nabla J(\mathbf{w})$ of the cost function

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

- The weight change is defined as

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

Gradient Computation

- To compute the gradient, first, to compute the partial derivative with respect to each weight w_j

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

- Weight updated of weight w_j

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

- We ***update*** all weights ***simultaneously in a single epoch***. based on all samples in the training set. So Adaline rule becomes

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

Adaline Rule vs. Perceptron Rule

- Looks (almost) identical
- Activation function in Adaline is a real number, while perceptron is an integer class label.
- In Adaline, weight updates is done based on all samples in a single epoch, while in perceptron the weights updated incrementally after each sample.
- This approach is known as ***batch*** gradient descent.

Implementing Adaline in Python

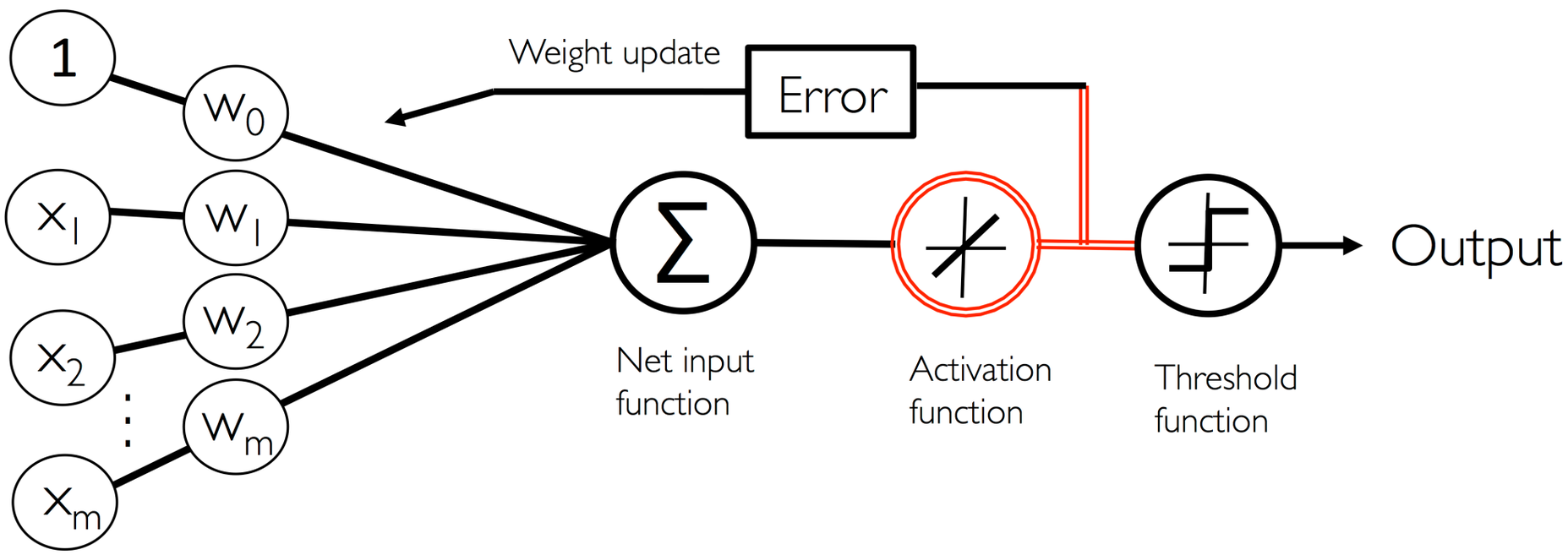
Adaline API (1/2)

```
class AdalineGD(object):
    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
    def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self
```

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

Adaline API (1/2)

```
def net_input(self, X):
    """Calculate net input"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def activation(self, X):
    """Compute linear activation"""
    return X

def predict(self, X):
    """Return class label after unit step"""
    return np.where(self.activation(self.net_input(X)) >= 0.0, 1, -1)
```

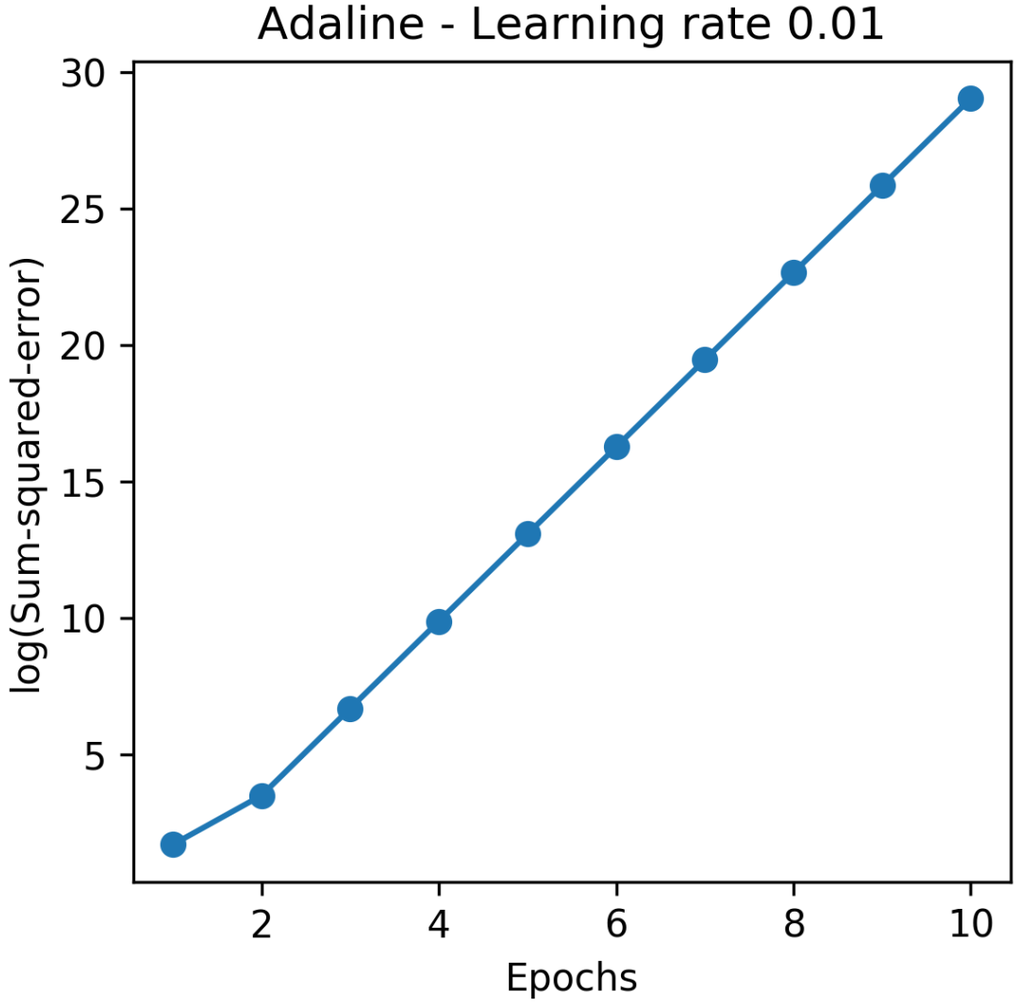


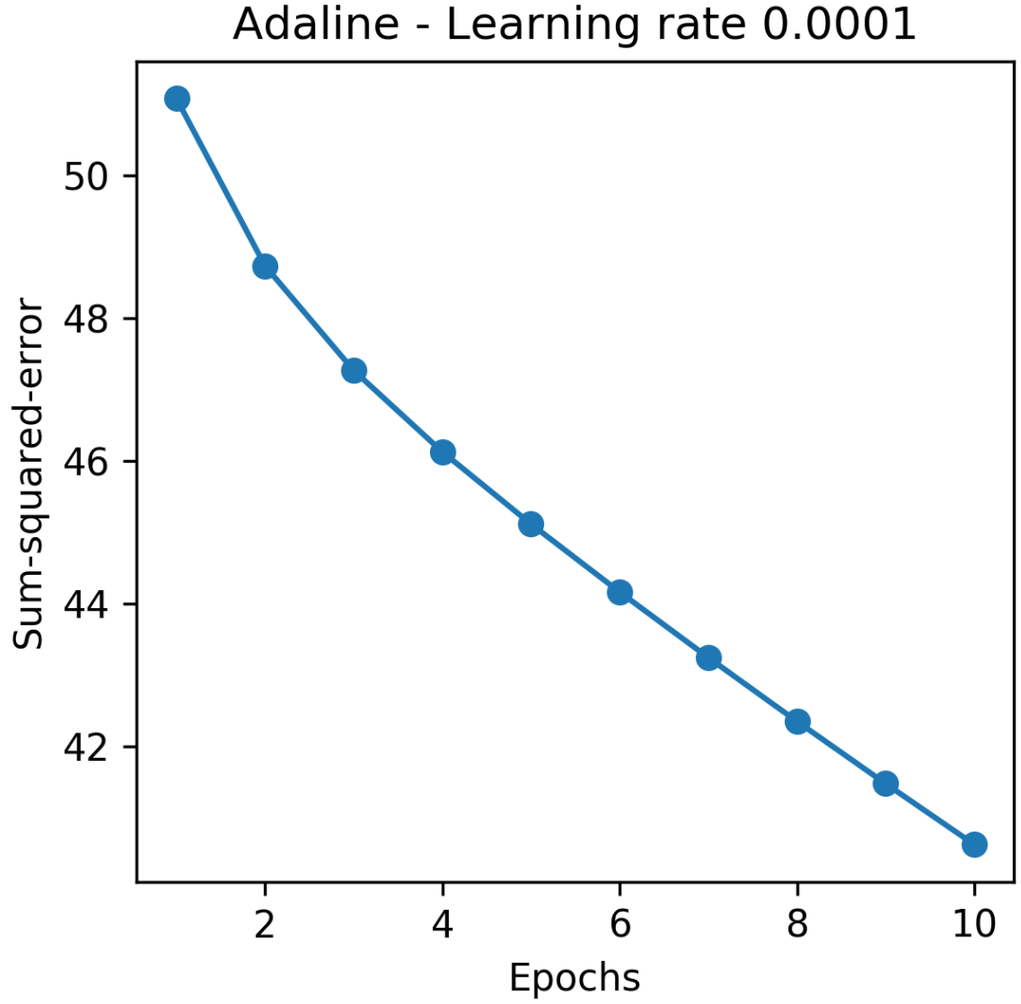
Different Learning Rate (1/3)

```
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.savefig('images/02_11.png', dpi=300)
plt.show()
```

Different Learning Rate (2/3)

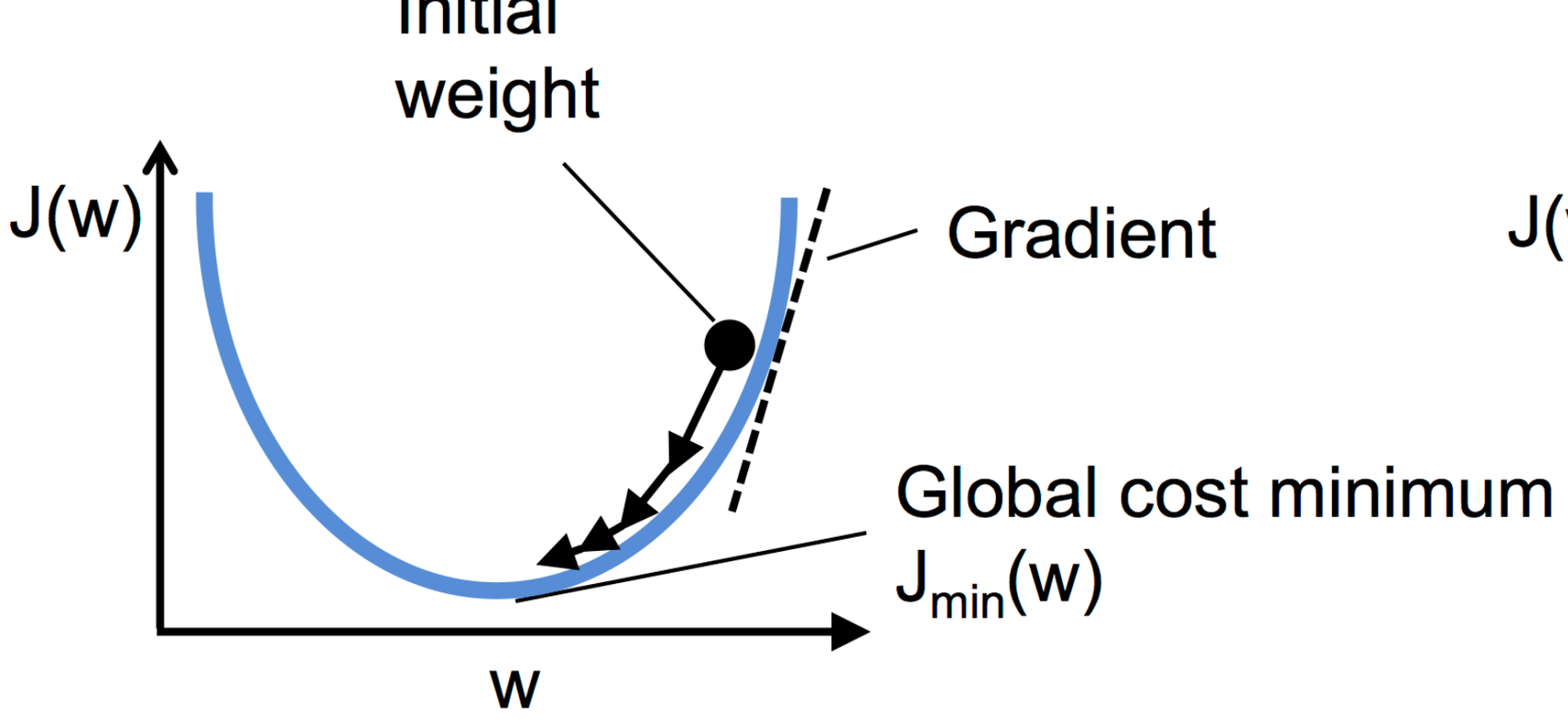
- Too large learning rate -> overshoot the global minimum
- Too small learning rate -> large number of epochs to converge to the global minimum



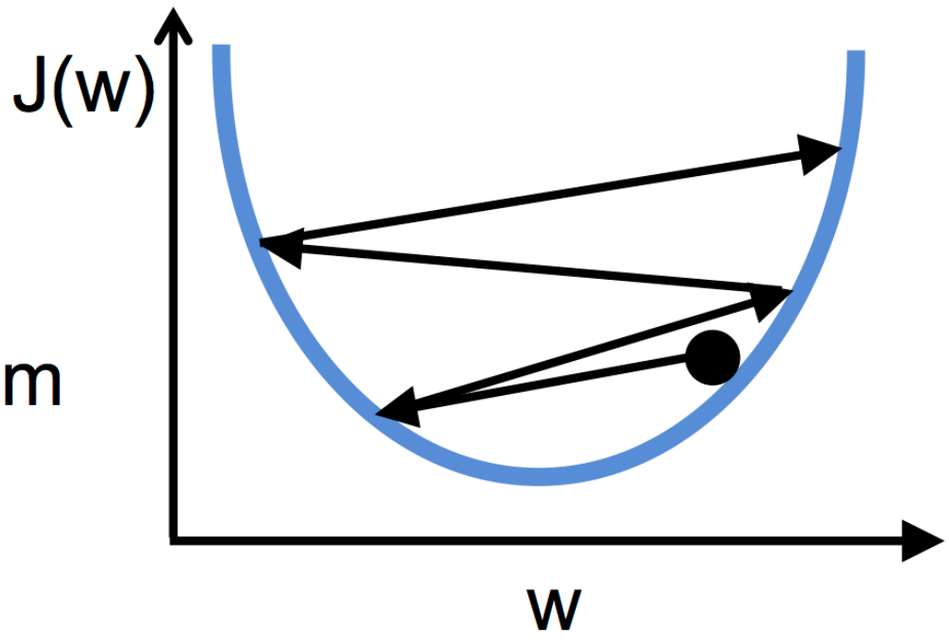


Different Learning Rate (3/3)

- The magnitude of learning rate η determines the convergence behavior of Adaline
- The ***learning rate*** and the ***number of epochs*** are the ***hyperparameters*** of Perceptron and Adaline



Well-chosen learning rate



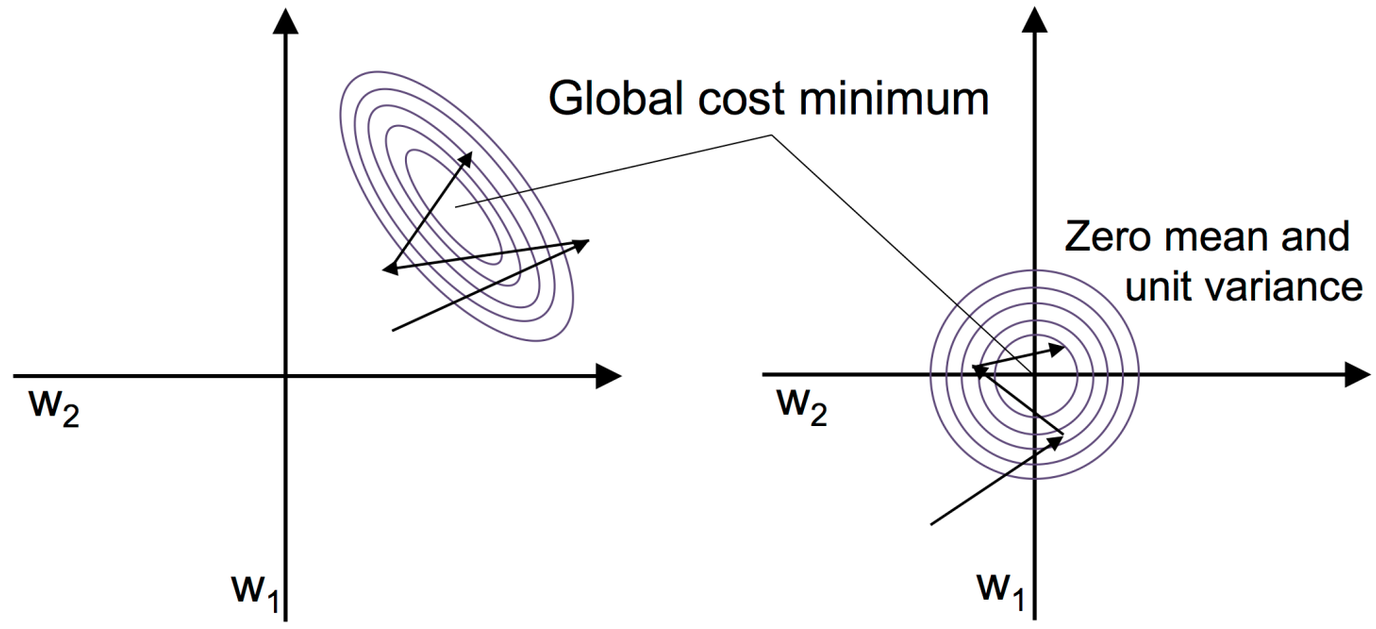
Large learning rate

Gradient Descent Improvements

Standardization: A Feature Scaling Method

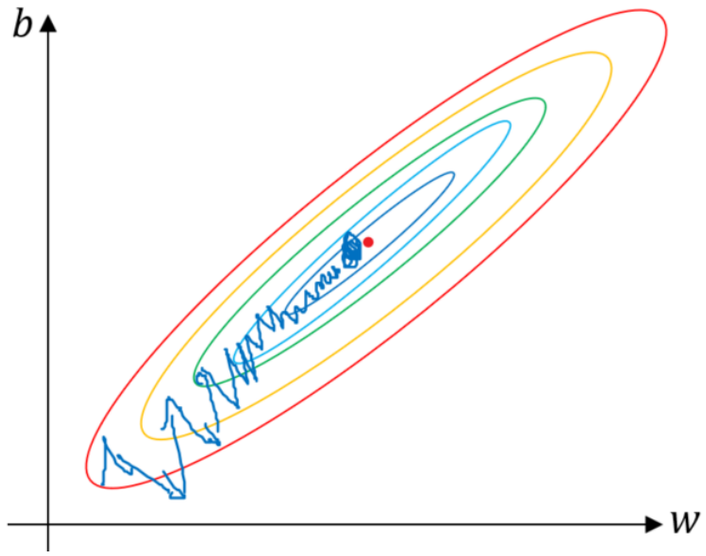
- Standardization makes the feature shift mean to 0 with standard deviation of 1 (for *j*th feature x_j)
- The standardized feature x'_j

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

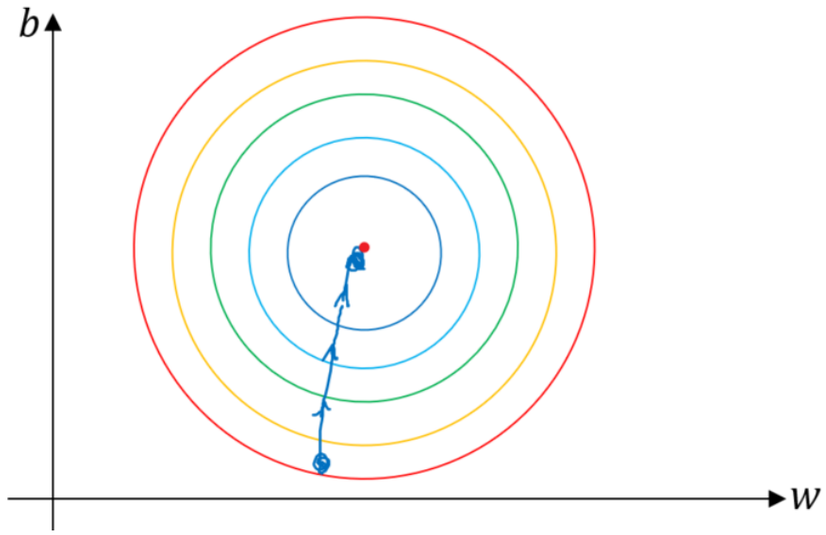



```
# standardize features
X_std = np.copy(X)
X_std[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()
X_std[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()
```

Unnormalized



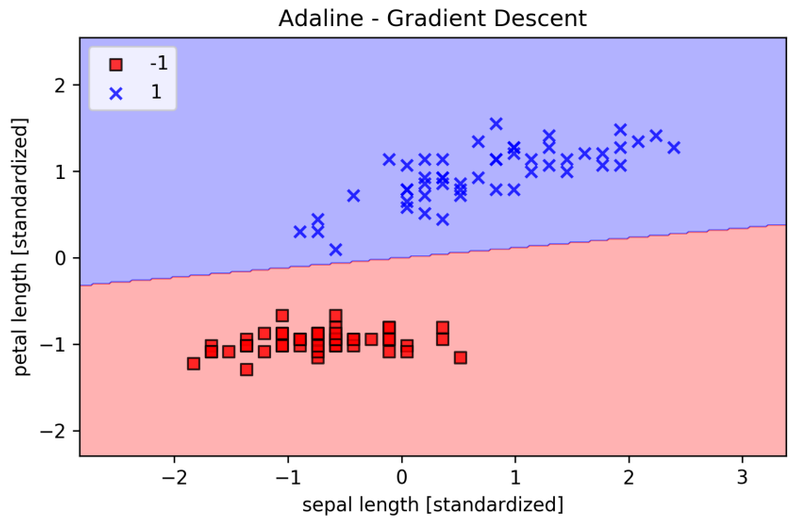
Normalized



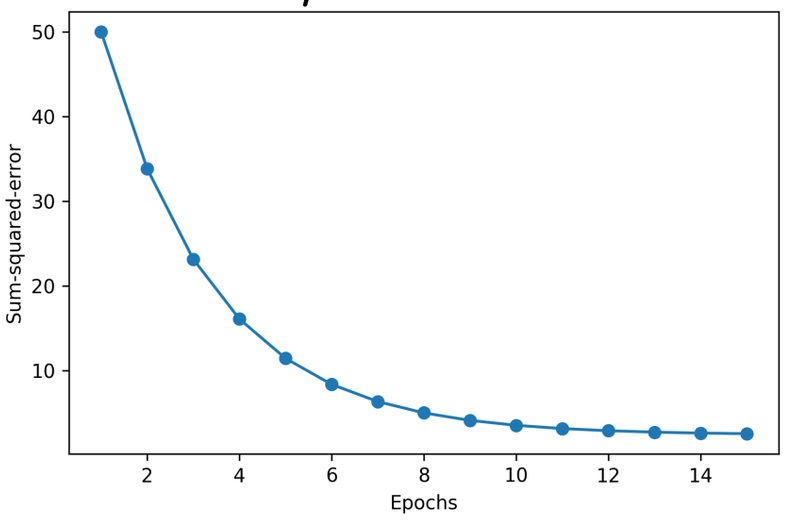
Standardization: A Feature Scaling Method

- Standardization can easily be achieved using the built-in NumPy methods **mean** and **std**.

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

plot_decision_regions(X_std, y, classifier=ada)
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.cost_) + 1), ada.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()
```



$\eta = 0.01$



Online (Stochastic) Gradient Descent

- Running batch gradient descent on a very large training dataset can be computationally costly

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

- The OGD/SGD technique processes a sample point in the training set in each epoch

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

- It is important to present the training data to SGD in a random order.
- Typically, SGD reaches global minimum much faster than the batch gradient descent
  - There are more frequent weight updates
  - Nosier updates help escaping shallow local minimum

Adaline SGD (1/3)

```
class AdalineSGD(object):
    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

    def fit(self, X, y):
        self._initialize_weights(X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            cost = []
            for xi, target in zip(X, y):
                cost.append(self._update_weights(xi, target))
            avg_cost = sum(cost) / len(y)
            self.cost_.append(avg_cost)
        return self
```

Adaline SGD (2/3)

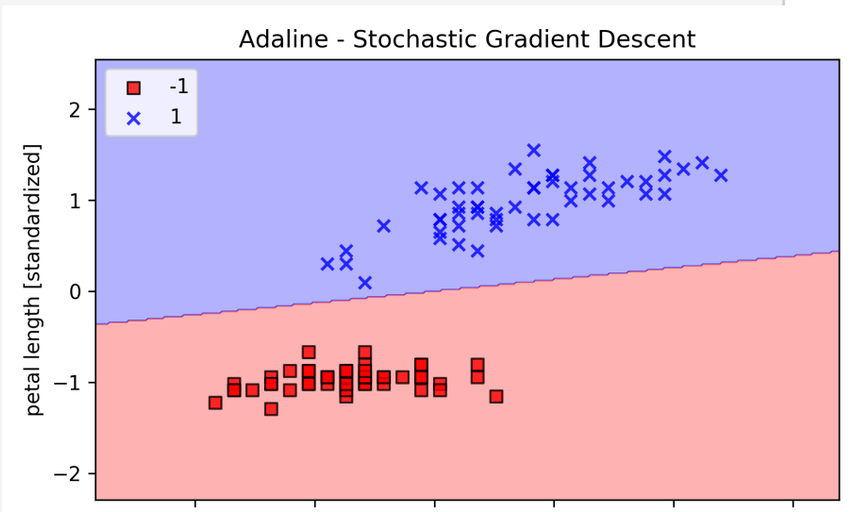
```
    def partial_fit(self, X, y):
        """Fit training data without reinitializing the weights"""
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
        """Shuffle training data"""
        r = self.rgen.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):
        """Initialize weights to small random numbers"""
        self.rgen = np.random.RandomState(self.random_state)
        self.w_ = self.rgen.normal(loc=0.0, scale=0.01, size=1 + m)
        self.w_initialized = True
```

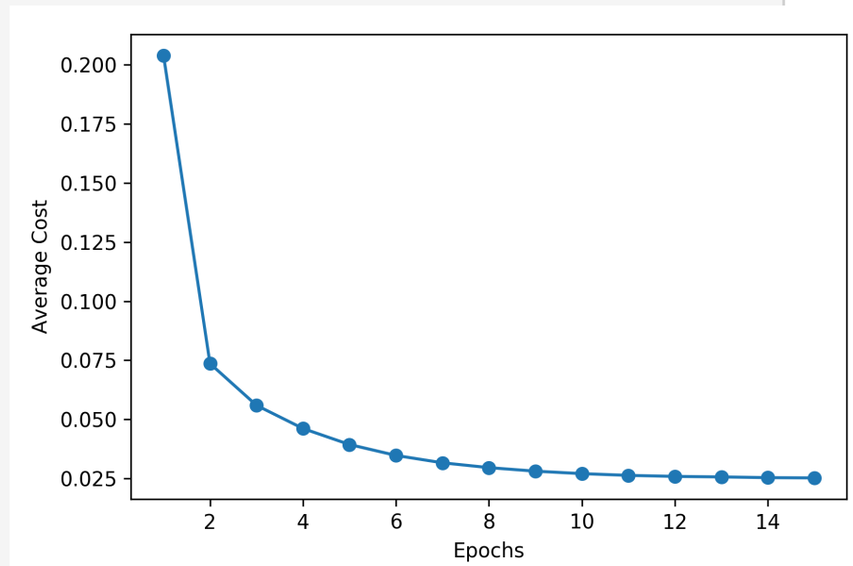
Adaline SGD (3/3)

```
    def _update_weights(self, xi, target):
        """Apply Adaline learning rule to update the weights"""
        output = self.activation(self.net_input(xi))
        error = (target - output)
        self.w_[1:] += self.eta * xi.dot(error)
        self.w_[0] += self.eta * error
        cost = 0.5 * error**2
        return cost

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Compute linear activation"""
        return X

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(self.net_input(X)) >= 0.0, 1, -1)
```

Adaline SGD (3/3)

```
ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
ada.fit(X_std, y)

plot_decision_regions(X_std, y, classifier=ada)
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.savefig('images/02_15_1.png', dpi=300)
plt.show()

plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Average Cost')

plt.tight_layout()
# plt.savefig('images/02_15_2.png', dpi=300)
plt.show()
```





Online Learning with Streaming Data

- The model is trained on the fly as new training data arrived
  - The system can immediately adapt new changes and the training data can be discarded after model updating if storage space is and issue
  - Call the **`partial_fit`** method on individual samples
    - Does not reinitialize the weights