lecture1

December 14, 2022

1 Lecture 1 - Basics of Machine Learning Classifiers

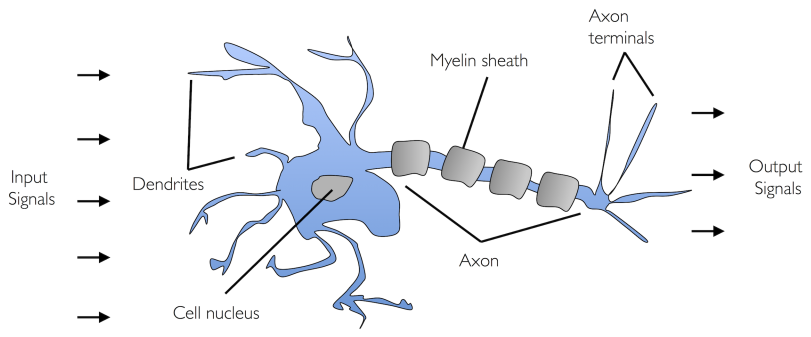
1.0.1 Overview

[1]:
```python
from IPython.display import Image
```

2 Artificial neurons

[2]:
```python
Image(filename='./figures/02_01.png', width=500)
```

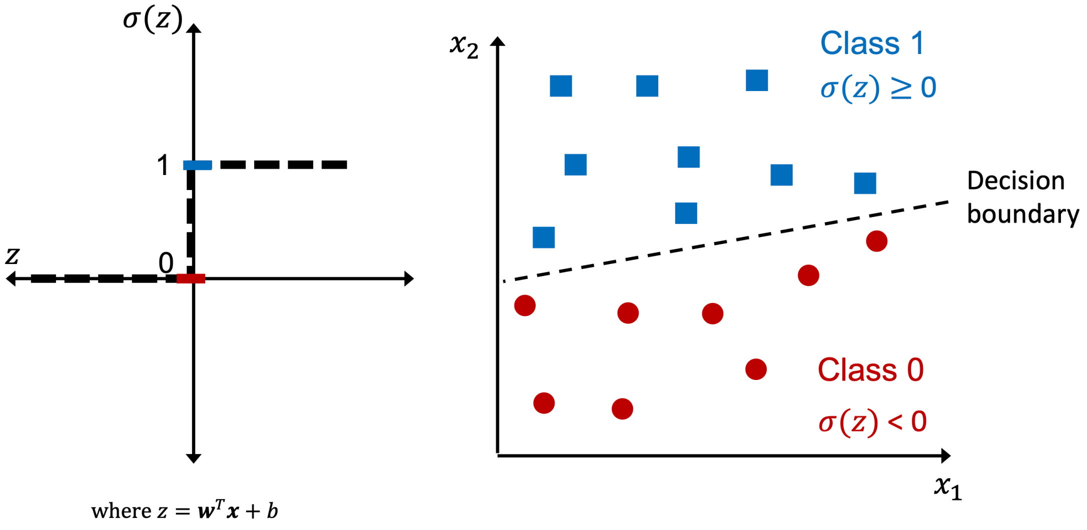
[2]:



3 The perceptron learning rule

```
[3]: Image(filename='./figures/02_02.png', width=500)
```

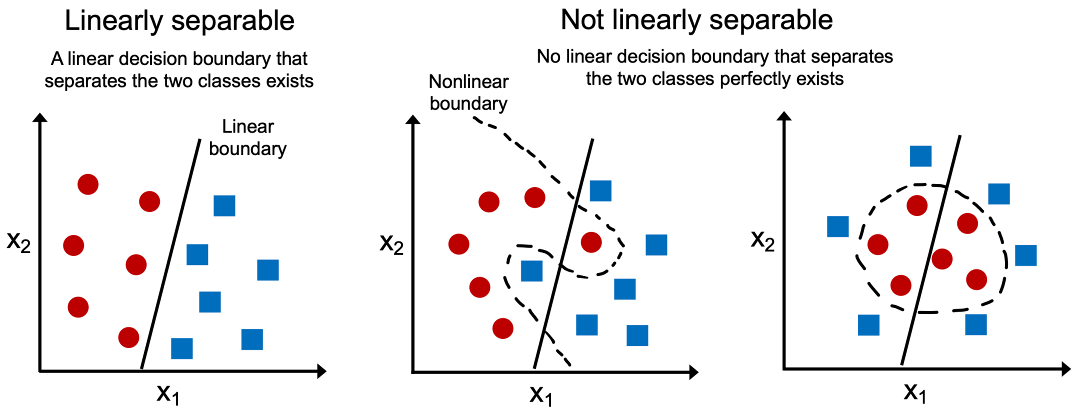
[3]:



3.1 When can we use the perceptron learning rule?

```
[4]: Image(filename='./figures/02_03.png', width=600)
```

[4]:


Linearly separable
A linear decision boundary that separates the two classes exists
Linear boundary
x_2
x_1
Not linearly separable
No linear decision boundary that separates the two classes perfectly exists
Nonlinear boundary
x_2
x_1
x_2
x_1

3.2 Learning the perceptron learning rule from data

The perceptron rule aims to separate the data using a linear decision boundary. Thus, it is the simplest possible way of classifying a dataset. We look at the training examples $\mathbf{x}$, and find weights

$\mathbf{w}$ and bias unit b such that $\sigma(z)$ with $z = \mathbf{w}^\top \mathbf{x} + b$ provides us the correct output label. The procedure is summarised below.

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

The update of w_j in the weight vector $\mathbf{w}$ can be written as

$$w_j := w_j + \Delta w_j, \qquad b := b + \Delta b. \tag{1}$$

The updates are computed as

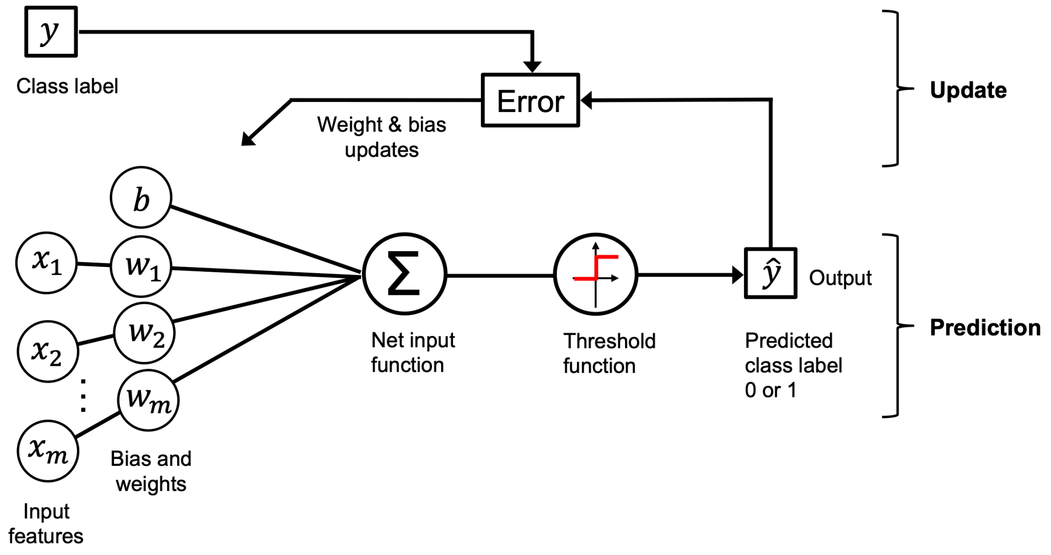
$$\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}, \qquad \Delta b = \eta(y^{(i)} - \hat{y}^{(i)}). \tag{2}$$

- η is called the learning rate
- $y^{(i)}$ is the true class lable
- $\hat{y}^{(i)}$ is the predicted class label

3.2.1 Summary of the learning procedure

[5]:
```
Image(filename='./figures/02_04.png', width=600)
```

[5]:



4 Implementing a perceptron learning rule in Python

4.1 An object-oriented approach

[6]:
```python
import numpy as np

class Perceptron:
    """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
    -----------
    Append an underscore (_) to attributes that
    are not created upon the initialization of the object
    w_ : 1d-array
      Weights after fitting.
    b_ : Scalar
      Bias unit 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

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

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

        Returns
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
        self.b_ = np.float_(0.)

        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_ += update * xi
                self.b_ += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, 0)
```

4.2 Training a perceptron model

4.2.1 Reading the data

We use a dataset where we know that the linear boundary classification rule works. We use the Iris dataset from UCI archives which records four features – sepal length, sepal width, petal lenght, petal width – for three different classes – Iris Setosa, Iris Versicolour, Iris Virginica.

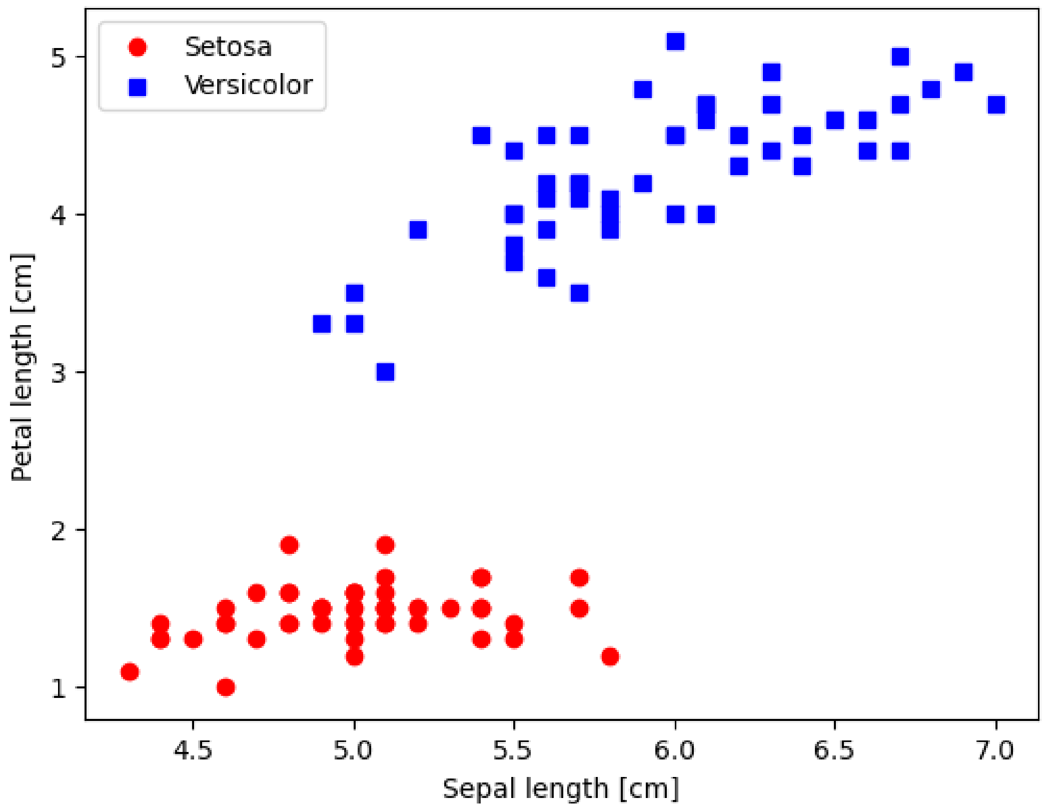
[7]:
```
import os
import pandas as pd

try:
    s = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.
 ↪data'
    print('From URL:', s)
    df = pd.read_csv(s, header=None, encoding='utf-8')

except HTTPError:
    s = 'iris.data'
    print('From local Iris path:', s)
    df = pd.read_csv(s, header=None, encoding='utf-8')

df.tail()
```

```
From URL: https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data
```

[7]:
```
       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
```

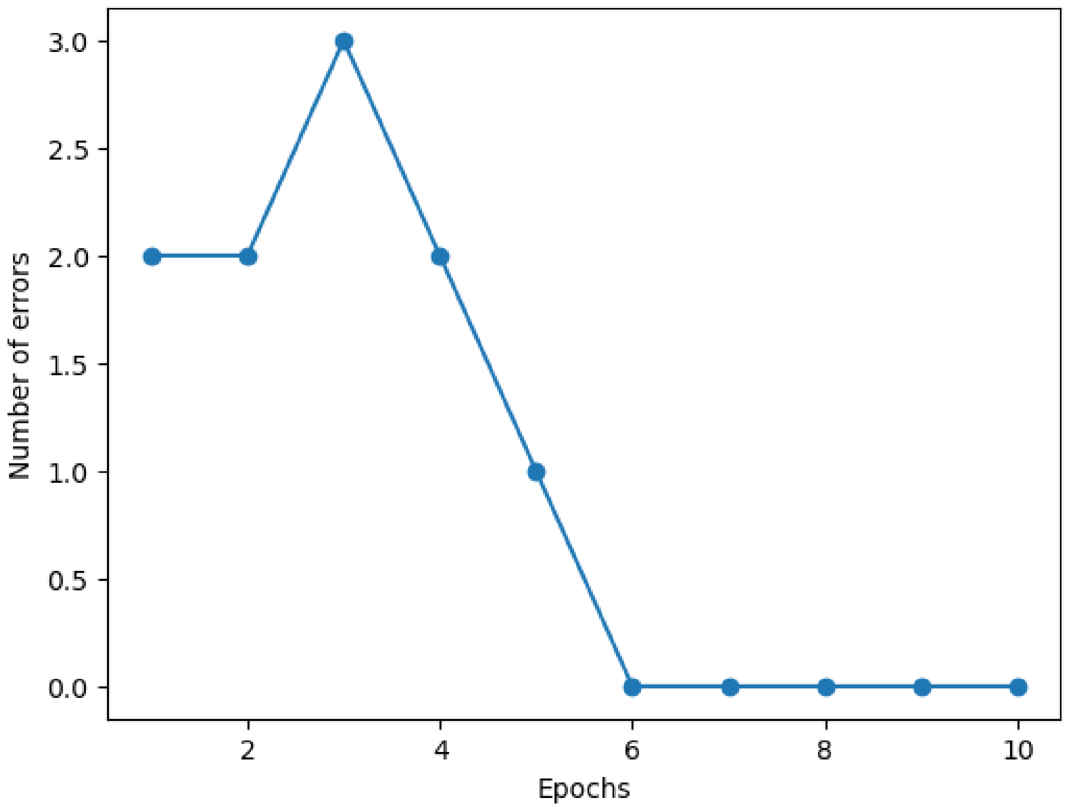
4.2.2 Plotting the data

[8]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

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

# 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='s', 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()
```



4.2.3 Training the perceptron model

[9]:
```
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 errors')

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

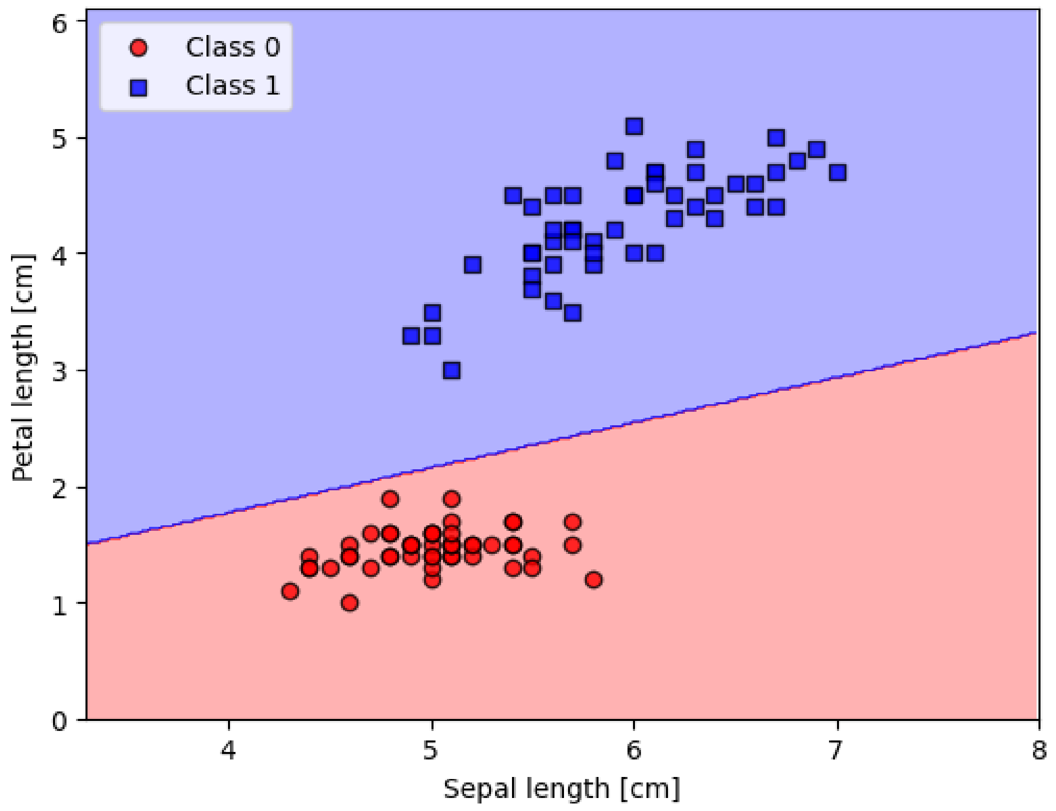


4.2.4 A function for plotting decision regions

[10]:
```
from matplotlib.colors import ListedColormap


def plot_decision_regions(X, y, classifier, resolution=0.02):

    # setup marker generator and color map
    markers = ('o', 's', '^', '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))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(x1_min, x1_max)
    plt.ylim(x2_min, x2_max)

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

4.2.5 Plotting the decision regions

[11]:
```
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()
```



5 Adaptive linear neurons and the convergence of learning

5.1 Minimizing cost functions with gradient descent

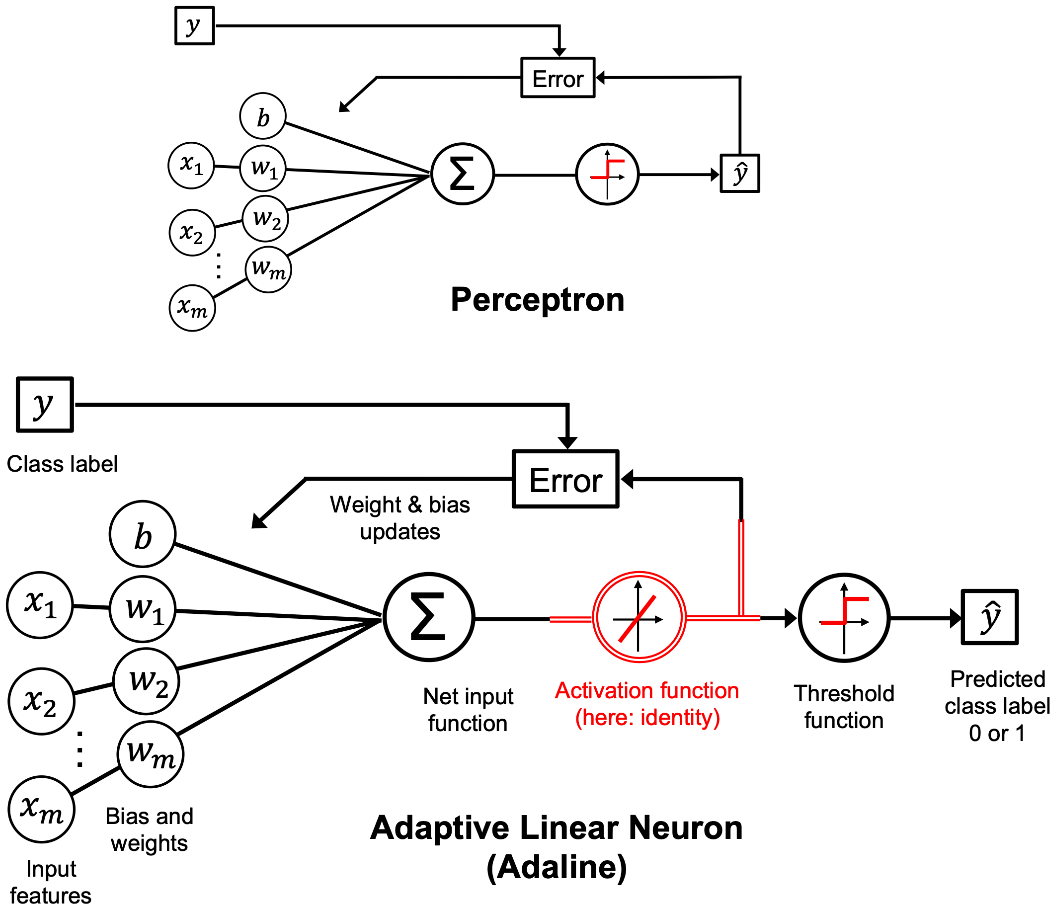
The Adaline algorithm illustrates the key concepts of defining and minimizing continuous loss functions needed for more advanced machine learning algorithms.

The key difference between the Adaline rule (also known as the Widrow-Hoff rule) and perceptron is that the weights are updated based on a linear activation function rather than a unit step function like in the perceptron. In Adaline, this linear activation function, $\sigma(z)$, is simply the identity function of the net input, so that $\sigma(z) = z$.

While the linear activation function is used for learning the weights, a threshold function is still used to make the final prediction, which is similar to the unit step function.

[12]:
```
Image(filename='./figures/02_09.png', width=600)
```

[12]:



5.1.1 Loss function

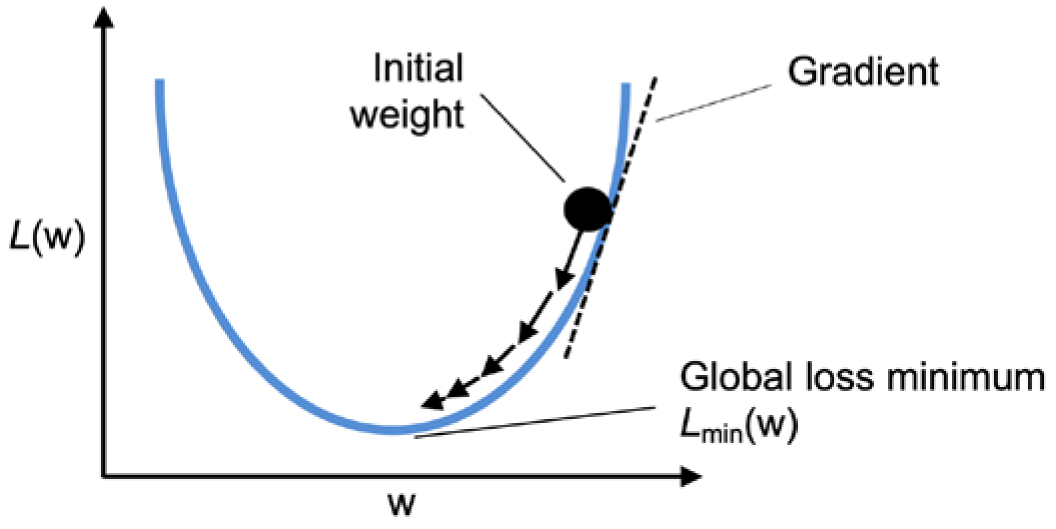
In the case of Adaline, we define the loss function, L, to learn the model parameters as the mean squared error (MSE) between the calculated outcome and the true class label:

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \sigma(z^{(i)})\right)^2 \tag{3}$$

- the activation function σ used in Adaline is linear, instead of a step function, which makes L differentiable
- L as defined is convex and thus **gradient descent** can be used to find the minimizer

```
[13]: Image(filename='./figures/grad_des.png', width=500)
```

[13]:



Using gradient descent, we update the model parameters by taking a step in the opposite direction of the gradient, $\nabla L(\mathbf{w}, b)$ of the loss function $L(\mathbf{w}, b)$ (**why opposite direction**?):

$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}, \qquad b := b + \Delta b. \tag{4}$$

The changes are given as

$$\Delta\mathbf{w} = -\eta\nabla_{\mathbf{w}}L(\mathbf{w}, b), \qquad \Delta b = -\eta\nabla_{b}L(\mathbf{w}, b). \tag{5}$$

In other words,

$$\Delta w_j = -\eta\frac{\partial L}{\partial w_j}, \qquad \Delta b = -\eta\frac{\partial L}{\partial b}. \tag{6}$$

From basic calculus, we get

$$\frac{\partial L}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \sigma(z^{(i)})\right)^2 \tag{7}$$

$$= \frac{\partial}{\partial w_j}\frac{1}{n}\sum_{i=1}^{n}\Big(y^{(i)} - (\sum_j w_j x_j^{(i)} + b)\Big)^2 \tag{8}$$

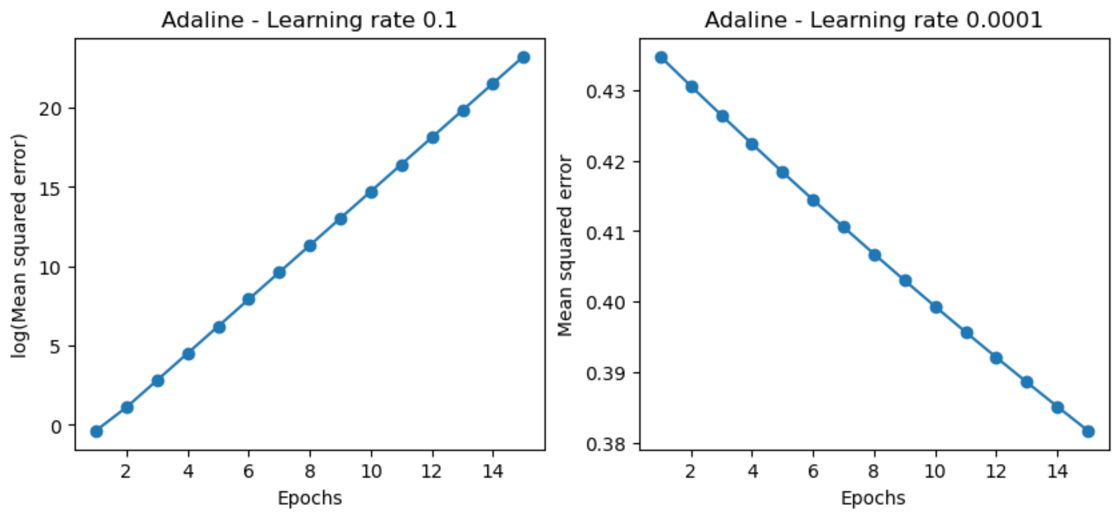
$$= \frac{2}{n}\sum_{i=1}^{n}\left(y^{(i)} - \sigma(z^{(i)})\right)\Big(-x_j^{(i)}\Big). \tag{9}$$

- Note that the weight update is calculated based on all examples in the training dataset (instead of updating the parameters incrementally after each training example), which is why this approach is also referred to as **full batch gradient descent**.

5.2 Implementing an Adaptive Linear Neuron in Python

[14]:
```python
class AdalineGD:
    """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.
    b_ : Scalar
      Bias unit after fitting.
    losses_ : list
      Mean squared eror loss function values 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

    def fit(self, X, y):
        """ Fit training data.
        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
          Training vectors, where n_examples is the number of examples and
          n_features is the number of features.
        y : array-like, shape = [n_examples]
          Target values.

        Returns
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
        self.b_ = np.float_(0.)
        self.losses_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X)
            # Please note that the "activation" method has no effect
            # in the code since it is simply an identity function. We
            # could write `output = self.net_input(X)` directly instead.
            # The purpose of the activation is more conceptual, i.e.,
            # in the case of logistic regression (as we will see later),
            # we could change it to
            # a sigmoid function to implement a logistic regression classifier.
            output = self.activation(net_input)
            errors = (y - output)

            #for w_j in range(self.w_.shape[0]):
            #    self.w_[w_j] += self.eta * (2.0 * (X[:, w_j]*errors)).mean()

            self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
            self.b_ += self.eta * 2.0 * errors.mean()
            loss = (errors**2).mean()
            self.losses_.append(loss)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    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)) >= 0.5, 1, 0)
```

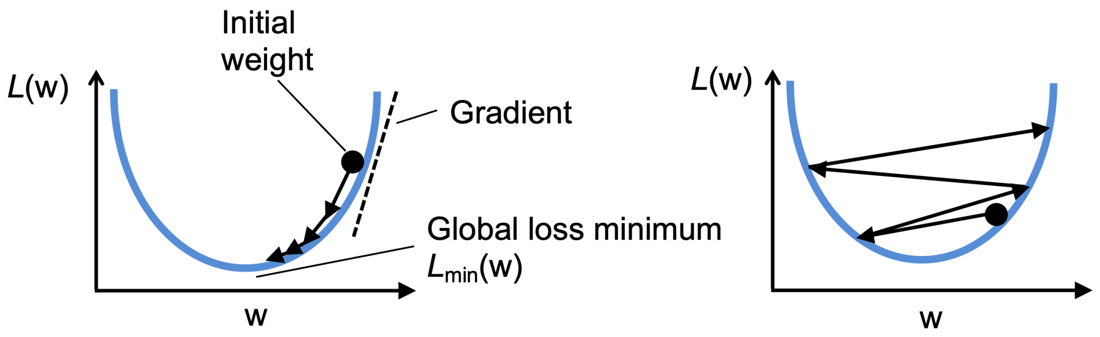
[15]:
```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))

ada1 = AdalineGD(n_iter=15, eta=0.1).fit(X, y)
ax[0].plot(range(1, len(ada1.losses_) + 1), np.log10(ada1.losses_), marker='o')
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('log(Mean squared error)')
ax[0].set_title('Adaline - Learning rate 0.1')

ada2 = AdalineGD(n_iter=15, eta=0.0001).fit(X, y)
ax[1].plot(range(1, len(ada2.losses_) + 1), ada2.losses_, marker='o')
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('Mean squared error')
ax[1].set_title('Adaline - Learning rate 0.0001')

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



From the above plots, we can see the effect of choosing appropriate learning rate. If the learning rate is chosen to be too large, then we do not obtain convergence.

[16]:
```
Image(filename='./figures/02_12.png', width=700)
```

[16]:



5.3 Improving gradient descent through feature scaling

Every machine learning algorithm requires some sort of feature scaling for optimal performance.

Gradient descent is one of the many algorithms that benefit from feature scaling.

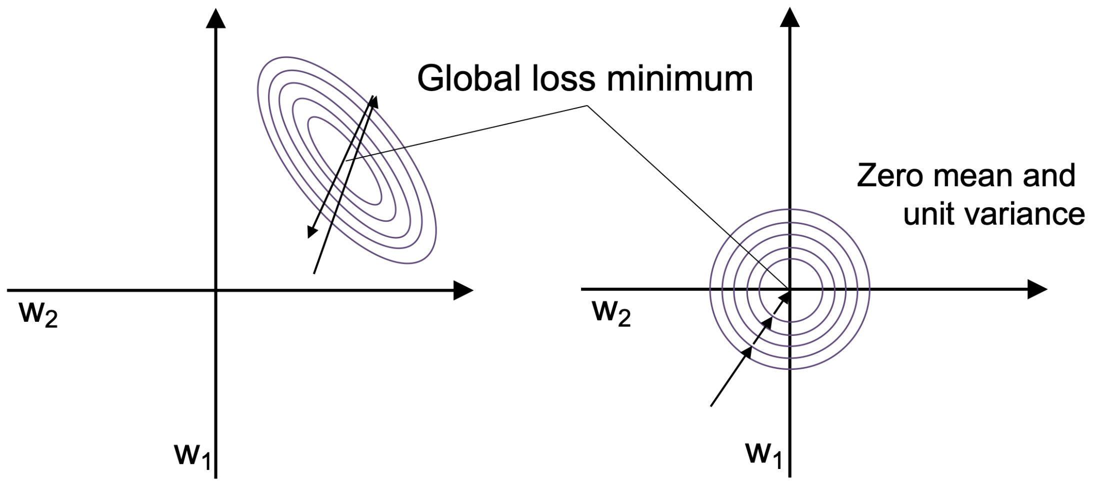
Standardisation shifts the mean of each feature so that it is centered at zero and each feature has a standard deviation of 1 (unit variance).

If the features are on vastly different scales, a learning rate that works well for updating one weight might be too large or too small to update the other weight equally well.

Using standardised features can stabilise the training such that the optimizer has to go through fewer steps to find a good or optimal solution.

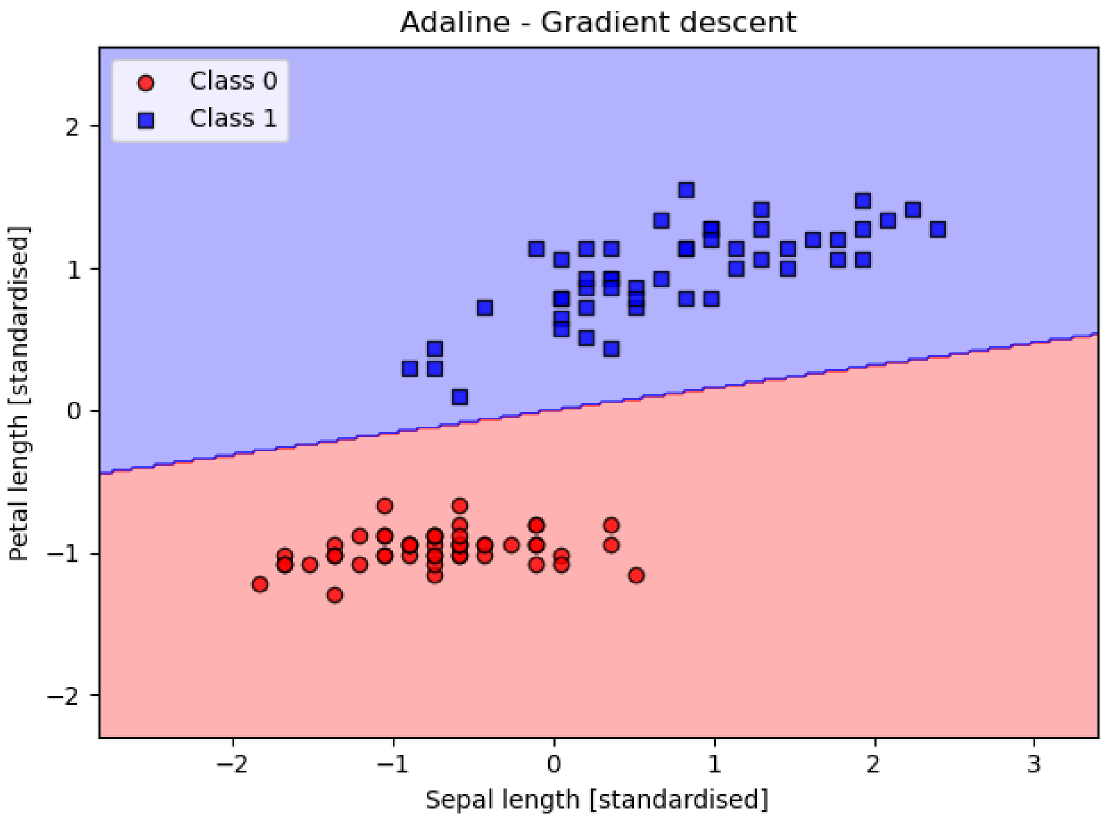
```
[17]: Image(filename='./figures/02_13.png', width=700)
```

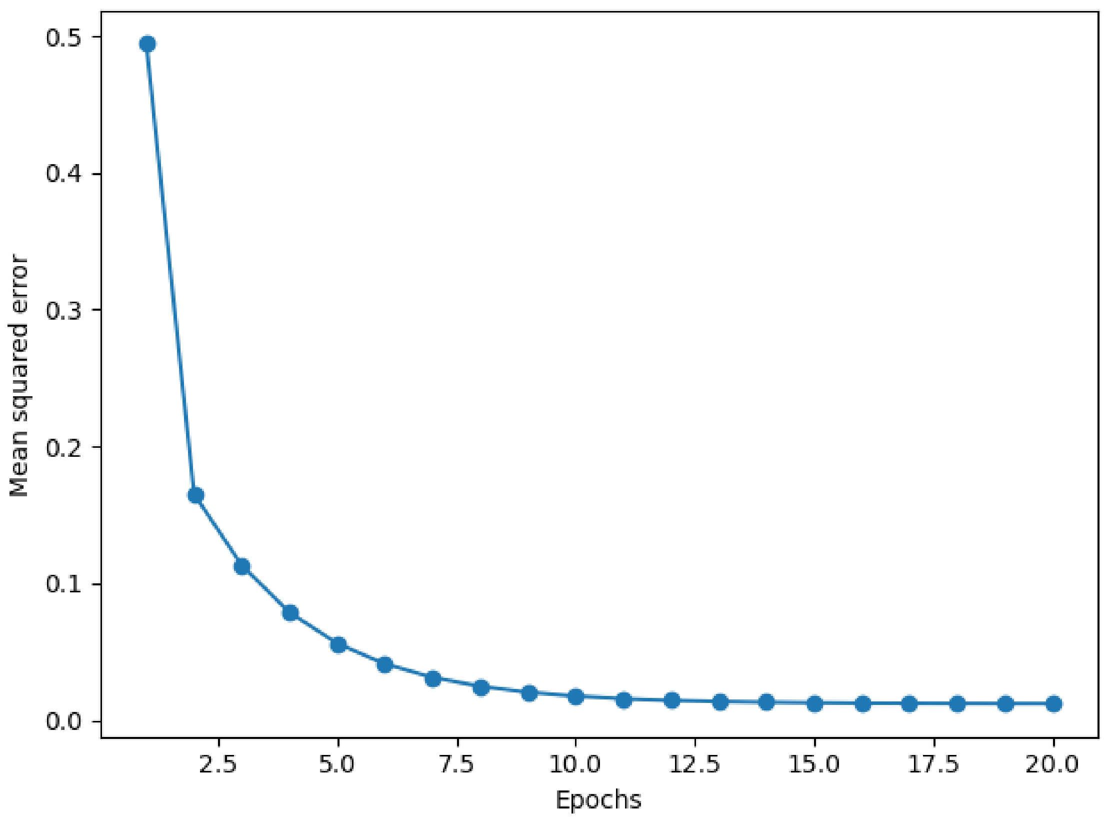
[17]:



[18]:
```
# standardise 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()
```

[19]:
```
ada_gd = AdalineGD(n_iter=20, eta=0.5)
ada_gd.fit(X_std, y)

plot_decision_regions(X_std, y, classifier=ada_gd)
plt.title('Adaline - Gradient descent')
plt.xlabel('Sepal length [standardised]')
plt.ylabel('Petal length [standardised]')
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.losses_) + 1), ada_gd.losses_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Mean squared error')

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


Adaline - Gradient descent
Class 0
Class 1
Petal length [standardised]
Sepal length [standardised]
2
1
0
−1
−2
−2
−1
0
1
2
3



Thus, by using standardized features, we reach convergence much faster.

5.4 Large scale machine learning and stochastic gradient descent

If we have a very large dataset, for example, with millions of data points, which is not uncommon in many machine learning applications, running a **full batch gradient descent** can be computationally quite costly in such scenarios, since we need to reevaluate the whole training dataset each time we take one step towards the global minimum.

A popular alternative to the full batch gradient descent algorithm is **stochastic gradient descent** (SGD), which is sometimes also called iterative or online gradient descent.

Instead of updating the weights based on the sum of the accumulated errors over all training examples, $\mathbf{x}^{(i)}$:

$$\Delta w_j = \frac{2\eta}{n} \sum_{i=1}^{n} (y^{(i)} - \sigma(z^{(i)})) \left(x_j^{(i)} \right), \tag{10}$$

we update the parameters incrementally for each training example, for instance:

$$\Delta w_j = 2\eta(y^{(i)} - \sigma(z^{(i)}))x_j^{(i)}, \qquad \Delta b = 2\eta(y^{(i)} - \sigma(z^{(i)})). \tag{11}$$

- SGD can be considered as an approximation of gradient descent but it typically reaches convergence much faster because of the more frequent weight updates.

- Since each gradient is calculated based on a single training example, the error surface is noisier than in gradient descent, which can also have the advantage that SGD can escape shallow local minima more readily if we are working with nonlinear loss functions (relevant for multilayer artificial neural networks).
- To obtain satisfying results via SGD, it is important to present training data in a random order; also, we want to shuffle the training dataset for every epoch to prevent cycles.

SGD can also be used for **online learning**. In online learning, the model is trained on the fly as new training data arrives. This is especially useful if we are accumulating large amounts of data, for example, customer data in web applications. Using online learning, the system can immediately adapt to changes, and the training data can be discarded after updating the model if storage space is an issue.

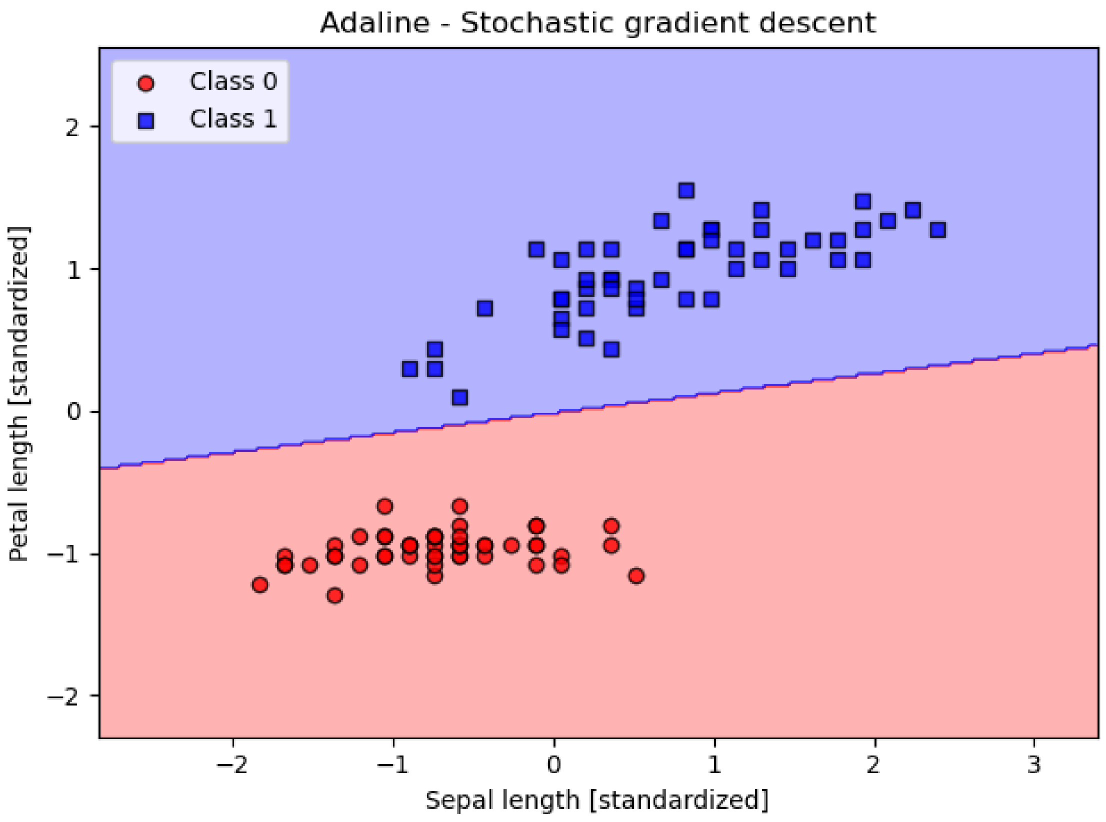
Mini-batch gradient descent

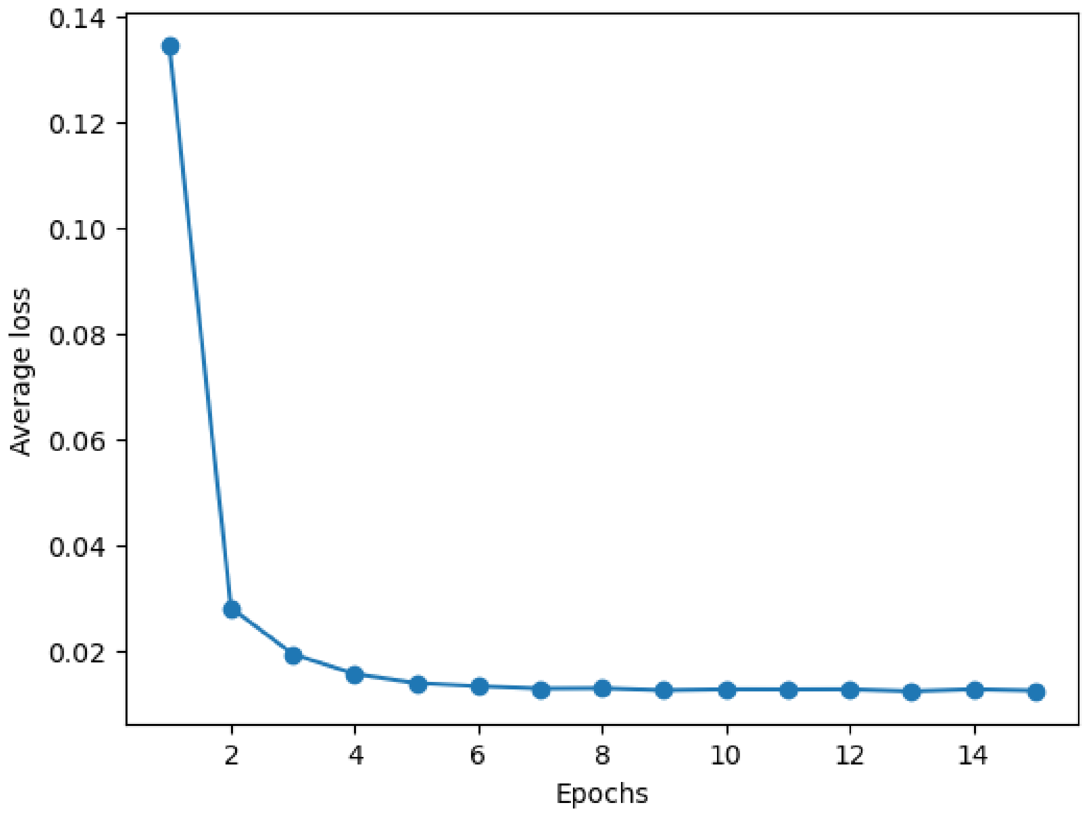
- Mini-batch gradient descent can be understood as applying full batch gradient descent to smaller subsets of the training data, for example, 32 training examples at a time.
- Advantage over full batch gradient descent is that convergence is reached faster via mini-batches because of the more frequent weight updates.
- Mini-batch learning allows us to replace the for loop over the training examples in SGD with vectorized operations leveraging concepts from linear algebra (for example, implementing a weighted sum via a dot product), which can further improve the computational efficiency of our learning algorithm.

[20]:
```
class AdalineSGD:
    """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.
    b_ : Scalar
        Bias unit after fitting.
    losses_ : list
      Mean squared error loss 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

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

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

        Returns
        -------
        self : object

        """
        self._initialize_weights(X.shape[1])
        self.losses_ = []
        for _ in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            losses = []
            for xi, target in zip(X, y):
                losses.append(self._update_weights(xi, target))
            avg_loss = np.mean(losses)
            self.losses_.append(avg_loss)
        return self

    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) ## single pass no loop
        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=m)
        self.b_ = np.float_(0.)
        self.w_initialized = True

    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_ += self.eta * 2.0 * xi * (error)
        self.b_ += self.eta * 2.0 * error
        loss = error**2
        return loss

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    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)) >= 0.5, 1, 0)
```

[21]:
```
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.savefig('figures/02_15_1.png', dpi=300)
plt.show()

plt.plot(range(1, len(ada_sgd.losses_) + 1), ada_sgd.losses_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Average loss')

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




0.14
0.12
0.10
0.08
0.06
0.04
0.02
Average loss
2
4
6
8
10
12
14
Epochs