# Mathematics of the Adaline learning algorithm
# Excerpts from Python Machine Learning
# Second Edition
# By Sebastian Raschka and Vahid Mirjalili[1]
# and other sources

Stepan Oskin

July 11, 2019

**Abstract**

## 1 Introduction

### 1.1 Classification

Classifying data is a common task in machine learning. Suppose some given data points each belong to one of two classes, and the goal is to decide which class a new data point will be in.

In the case of the Adaline learning algorithm, a data point is viewed as a $p$-dimensional vector (a list of $p$ numbers), and we want to know whether we can separate such points with a $(p-1)$-dimensional hyperplane. This is called a linear classifier. More formally, we can put the idea behind artificial neurons into the context of a binary classification task where we refer to our two classes as 1 (positive class) and -1 (negative class) for simplicity. There are many hyperplanes that might classify the data. We can then define a decision function ($\phi(z)$) that takes a linear combination of certain input values $x$ and a corresponding weight vector $w$, where $z$ is the so-called net input.

### 1.2 Adaline learning algorithm for classification

Adaline learning algorithm for classification presents another type of single-layer neural network. **ADAptive LInear NEuron (Adaline)** was published by *Bernard Widrow* and his doctoral student *Tedd Hoff* [2], only a few years after Frank Rosenblatt's perceptron algorithm[3], and can be considered as an improvement on the latter.

### 1.3 Minimizing continuous cost functions

The Adaline algorithm is particularly interesting because it illustrates the key concepts of defining and minimizing **continuous cost functions**. This lays the groundwork for understanding more advanced machine learning algorithms for classification, such as logistic regression, support vector machines, and regression models.

The **key difference** between the Adaline rule (also known as the **Widrow-Hoff** rule) and Rosenblatt's 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 $\phi(z)$ is simply the **identity function** of the net input, so that:

$$\phi(\boldsymbol{w}^T\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} \tag{1}$$

While the linear activation function is used for learning the weights, we still use a threshold function to make the final prediction, which is similar to the unit step function that was described for the perceptron. The main differences between the perceptron and Adaline algorithm are highlighted in figure 1.

Figure 1 shows that the Adaline algorithm compares the true class labels with the linear activation function's **continuous valued output** to compute the model error and update the weights. In contrast, the perceptron compares the true class labels to the predicted class labels.

## 2 Minimizing cost functions with gradient descent

### 2.1 Continuous cost function in Adaline

One of the key ingredients of supervised machine learning algorithms is a defined **objective function** that is to be optimized during the learning process. This objective function is often a cost function that we want to minimize. In the case of Adaline, we can define the **cost function** $J$ to

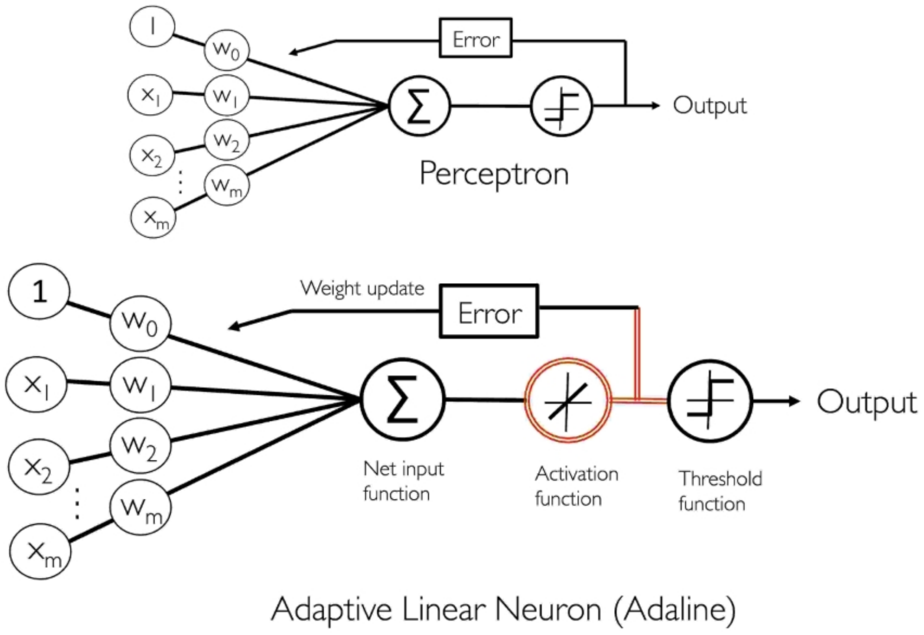


Figure 1: Adaline algorithm compares the true class labels with the linear activation function's continuous valued output to compute the model error and update the weights. In contrast, the perceptron compares the true class labels to the predicted class labels.

learn the weights as the **Sum of Squared Errors (SSE)** between the calculated outcome and the true class label:

$$J(\boldsymbol{w}) = \frac{1}{2} \sum_{i} \left( y^{(i)} - \phi\left(z^{(i)}\right) \right)^2 \tag{2}$$

The term $\frac{1}{2}$ is just added for convenience, which will make it easier to derive the gradient, as we will see in the following paragraphs. The main advantage of this continuous linear activation function, in contrast to the unit step function, is that the **cost function becomes differentiable**. Another nice property of this cost function is that it is convex; thus, we can use a simple yet powerful optimization algorithm called **gradient descent** to find the weights that minimize our cost function to classify the samples in the Iris dataset.

### 2.2 Optimizing convex and continuous functions using Gradient Descent

As illustrated in figure 2, we can describe the main idea behind gradient descent as climbing down a hill until a local or global cost minimum is reached. In each iteration, we take a step in the opposite direction of the gradient where the step size is determined by the value of the learning rate, as well as the slope of the gradient:

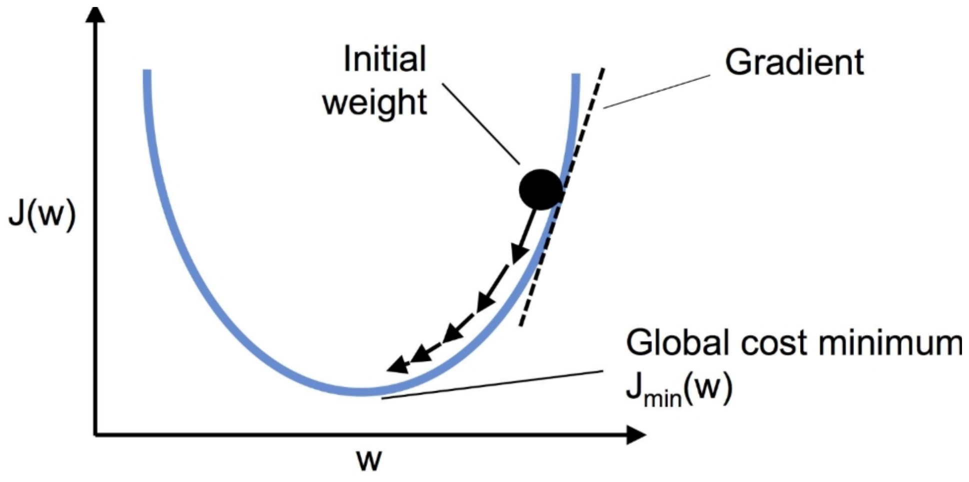


Figure 2: Main idea behind gradient descent optimization algorithm can be described as climbing down a hill until a local or global cost minimum is reached. In each iteration, we take a step in the opposite direction of the gradient where the step size is determined by the value of the learning rate, as well as the slope of the gradient.

Using gradient descent, we can now update the weights by taking a step in the opposite direction of the gradient $\nabla J(w)$ of our cost function:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w} \tag{3}$$

Where the weight change $\Delta\boldsymbol{w}$ is defined as the negative gradient multiplied by the learning rate $\eta$:

$$\Delta\boldsymbol{w} = -\eta\nabla J(\boldsymbol{w}) \tag{4}$$

### 2.3 Computing partial derivatives of the cost function

To compute the gradient of the cost function, we need to compute the partial derivative of the cost function with respect to each weight $w_j$:

$$\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)} \tag{5}$$

The full derivation can be obtained using the chain rule of differentiation:

$$\begin{aligned}
\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2 = \\
&= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2 = \\
&= \frac{1}{2} \sum_i 2 \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) = \\
&= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sum_i \left(w_j^{(i)} x_j^{(i)}\right)\right) = \\
&= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \left(-x_j^{(i)}\right) = \\
&= -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}
\end{aligned} \tag{6}$$

## 3 An object-oriented Adaline API using NumPy

### 3.1 Adaline learning rule

After the partial derivatives of the cost function with respect to weights have been computed, we can re-write the update of weight $w_j$ as:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)} \tag{7}$$

Since we update all weights simultaneously, our Adaline learning rule becomes:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w} \tag{8}$$

Although the Adaline learning rule looks identical to the perceptron rule, we should note that the $\phi\left(z^{(i)}\right)$ with is a real number and not an integer class label. Furthermore, the weight update is calculated based on all samples in the training set (instead of updating the weights incrementally after each sample), which is why this approach is also referred to as **batch gradient descent**.

## 3.2 Description of the implementation

Since the perceptron rule and Adaline are very similar, *Raschka and Mirjalili*[1] took the perceptron implementation that was defined earlier and changed the `fit` method so that the weights are updated by minimizing the cost function via gradient descent.

Instead of updating the weights after evaluating each individual training sample, as in the perceptron, the gradient is calculated based on the whole training dataset via `self.eta * errors.sum()` for the bias unit (zero-weight) and via `self.eta * X.T.dot(errors)` for the weights 1 to m where `X.T.dot(errors)` is a matrix-vector multiplication between our feature matrix and the error vector.

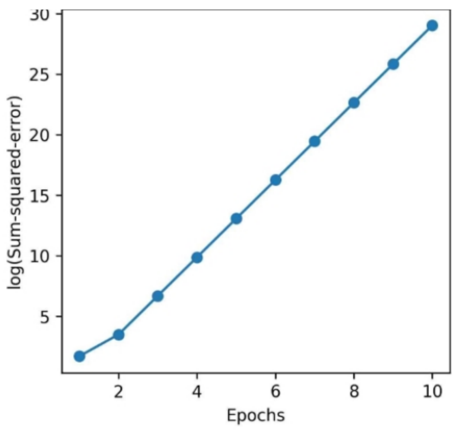
Note that the `activation` method has no effect in the code since it is simply an identity function. The `activation` function (computed via the activation method) was added to illustrate how information flows through a single layer neural network: features from the input data, net input, activation, and output. Other classifiers, such as logistic regression, use a non-identity, nonlinear activation function. **Logistic regression** model is closely related to Adaline with the only difference being its activation and cost function.

Now, similar to the previous perceptron implementation, cost values are collected in a `self.cost_` list to check whether the algorithm converged after training.

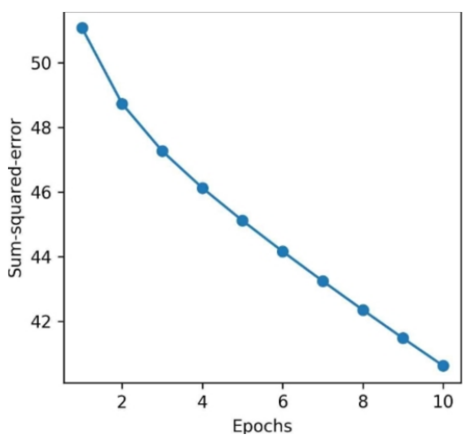
## 3.3 Selecting the learning rate

In practice, it often requires some experimentation to find a good learning rate $\eta$ for optimal convergence. On figure 3, two different learning rates, $\eta = 0.1$ and $\eta = 0.0001$, were chosen and the cost functions versus the number of epochs were plotted to see how well the Adaline implementation learns from the training data.

The learning rate $\eta$ (`eta`), as well as the number of epochs (`n_iter`), are the so-called hyperparameters of the perceptron and Adaline learning algorithms.



(a) $\eta = 0.01$



(b) $\eta = 0.0001$

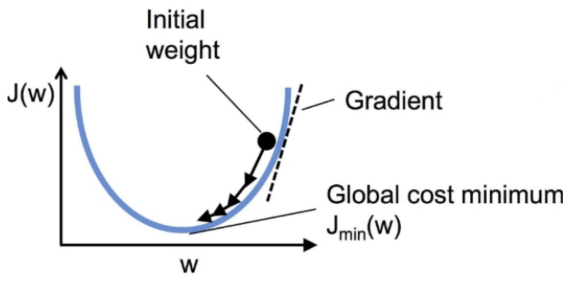
Figure 3: It often requires some experimentation to find a good learning rate $\eta$ for optimal convergence. Two different learning rates, $\eta = 0.1$ (3a) and $\eta = 0.0001$(3b), were chosen and the cost functions versus the number of epochs were plotted to see how well the Adaline implementation learns from the training data.

As we can see in the resulting cost-function plots, we encountered two different types of problem. Figure 3a shows what could happen if we choose a learning rate that is too large. Instead of minimizing the cost function, the error becomes larger in every epoch, because we overshoot the global minimum. On the other hand, we can see that the cost decreases on figure 3b, but the chosen learning rate $\eta = 0.0001$ is so small that the algorithm would require a very large number of epochs to converge to the global cost minimum.

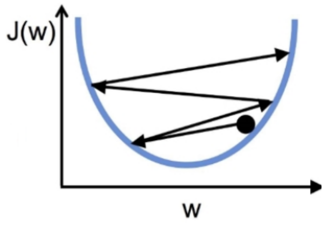
Figure 4 illustrates what might happen if we change the value of a particular weight parameter to minimize the cost function $J$. The left subfigure illustrates the case of a well-chosen learning rate, where the cost decreases gradually, moving in the direction of the global minimum. The subfigure on the right, however, illustrates what happens if we choose a learning rate that is too large —we overshoot the global minimum:

## 3.4 Improving gradient descent through feature scaling

Many machine learning algorithms require some sort of feature scaling for optimal performance. Gradient descent is one of the many algorithms that benefit from feature scaling. One of the feature scaling methods that can be



(a) Well-selected learning rate



(b) Learning rate too large

Figure 4: If the selected learning rate $\eta$ is too large, instead of minimizing the cost function, the error becomes larger in every epoch, because we overshoot the global minimum.

used is called standardization. Standardization gives our data the property of a standard normal distribution, which helps gradient descent learning to converge more quickly. Standardization shifts the mean of each feature so that it is centered at zero and each feature has a standard deviation of 1. For instance, to standardize the $j^{th}$ feature, we can simply subtract the sample mean $\mu_j$ from every training sample and divide it by its standard deviation $\sigma_j$:

$$\boldsymbol{x'}_j = \frac{\boldsymbol{x}_j - \mu_j}{\sigma_j} \tag{9}$$

Here, $\boldsymbol{x}_j$ is a vector consisting of the $^{jth}$ feature values of all training samples n, and this standardization technique is applied to each feature j in our dataset.

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 figure 5, where the subfigures represent the cost surface as a function of two model weights in a two-dimensional classification problem.

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

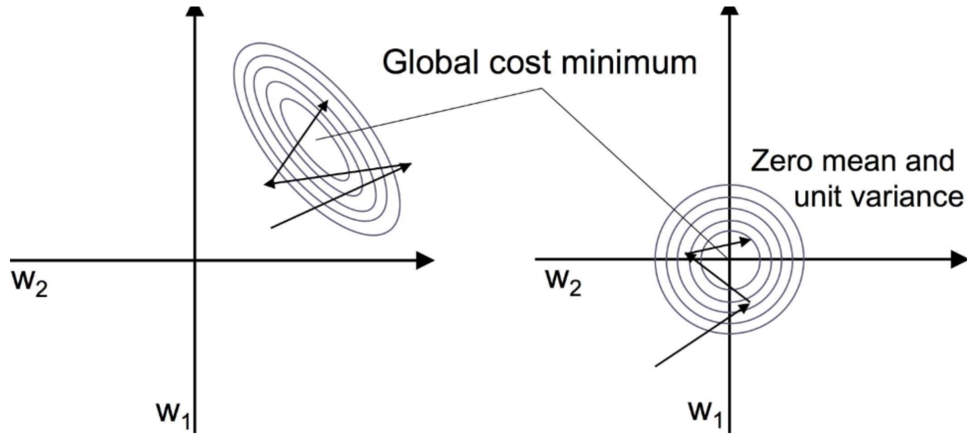


Figure 5: Cost surface as a function of two model weights in a two-dimensional classification problem. Standardization helps with gradient descent learning because the optimizer has to go through fewer steps to find a good or optimal solution (the global cost minimum)

### 3.5 Training Adaline model on two classes from the Iris dataset

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

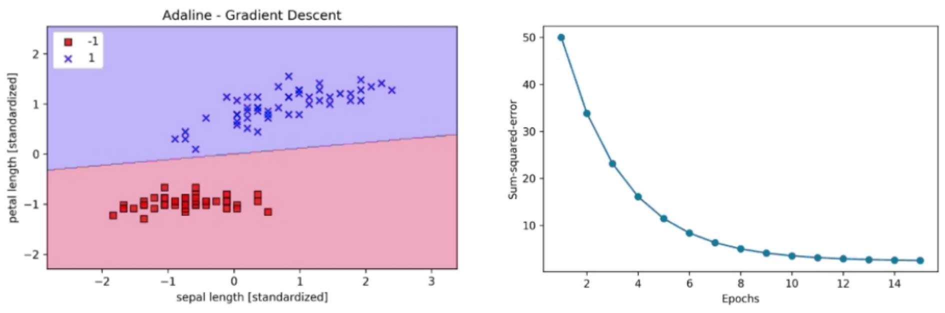
To test the Adaline implementation, Iris flower dataset for classification was used (presented in table 1), from which two flower classes Setosa and Versicolor were loaded. Although the Adaline rule is not restricted to two dimensions, only two features were considered (`sepal length` and `petal length` for visualization purposes. Though only the two flower classes Setosa and Versicolor were chosen for practical reasons, the Adaline algorithm can be extended to multi-class classification —for example, the **One-versus-All (OvA)** technique.

First, the first 100 class labels that correspond to the 50 `Iris-setosa` and 50 `Iris-versicolor` flowers were extracted, and the class labels were converted into the two integer class labels `1` (`versicolor`) and `-1` (`setosa`) that we assign to a vector `y`, where the values method of a `pandas DataFrame` yields the corresponding `NumPy` representation. In this two-dimensional feature subspace, a linear decision boundary should be sufficient to separate Setosa from Versicolor flowers. Thus, a linear classifier such as Adaline should be able to classify the flowers in this dataset perfectly.

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 5.1 | 3.5 | 1.4 | 0.2 | Iris-setosa |
| 1 | 4.9 | 3.0 | 1.4 | 0.2 | Iris-setosa |
| 2 | 4.7 | 3.2 | 1.3 | 0.2 | Iris-setosa |
| 3 | 4.6 | 3.1 | 1.5 | 0.2 | Iris-setosa |
| 4 | 5.0 | 3.6 | 1.4 | 0.2 | Iris-setosa |

Table 1: The first 5 rows from the Iris flower dataset for classification. The four features include `petal_width`, `petal_length`, `sepal_width`, and `sepal_length`. The three class labels are `Iris-setosa`, `Iris-versicolor`, and `Iris-virginica`.

### 3.6 Performance of Adaline on two linearly separable classes



(a) Decision boundary (b) Cost per epoch

Figure 6: Decision boundary learned by Adaline and the cost for each epoch. Adaline has converged after training on the standardized features using a learning rate $\eta = 0.01$. However, SSE remains non-zero even though all samples were classified correctly.

Figure 6 shows the decision boundary learned by Adaline and the cost for each epoch and can be used to check whether the algorithm converged and found a decision boundary that separates the two Iris flower classes.

As we can see in the plots, Adaline has now converged after training on the standardized features using a learning rate $\eta = 0.01$. However, note that the SSE remains non-zero even though all samples were classified correctly.

## 4 Large-scale machine learning and stochastic gradient descent

In the previous section, a cost function was minimized by taking a step in the opposite direction of a cost gradient that is calculated from the whole training set; this is why this approach is sometimes also referred to as **batch gradient descent**. In many machine learning applications, it is not uncommon to have a very large dataset with millions of data points. In this case, running 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. Alternatively, weights can be updated on each sample, as is done in stochastic gradient descent, or per mini-batches of samples, as is done in mini-batch gradient descent.

### 4.1 Updating weights for each training sample

A popular alternative to the batch gradient descent algorithm is **stochastic gradient descent**, sometimes also called **iterative** or **online gradient descent**. Instead of updating the weights based on the sum of the accumulated errors over all samples $\boldsymbol{x}^{(i)}$:

$$\Delta \boldsymbol{w} = \eta \sum_{i} \left( y^{(i)} - \phi\left(z^{(i)}\right)\right) \boldsymbol{x}^{(i)} \tag{10}$$

We update the weights incrementally for each training sample:

$$\eta \left( y^{(i)} - \phi\left(z^{(i)}\right)\right) \boldsymbol{x}^{(i)} \tag{11}$$

Although stochastic gradient descent can be considered as an approximation of gradient descent, 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 stochastic gradient descent can escape shallow local minima more readily if we are working with nonlinear cost functions. To obtain satisfying results via stochastic gradient descent, it is **important to present it training data in a random order**; also, we want to **shuffle the training set for every epoch** to prevent cycles.

### 4.2 Adaptive learning rate

In stochastic gradient descent implementations, the fixed learning rate is often replaced by an adaptive learning rate that decreases over time, for example:

$$\frac{c_1}{[\text{number of iterations}] + c2} \tag{12}$$

Where $c_1$ and $c_2$ are constants. We shall note that stochastic gradient descent does not reach the global minimum, but an area very close to it. And using an adaptive learning rate, we can achieve further annealing to the cost minimum.

### 4.3 Online learning

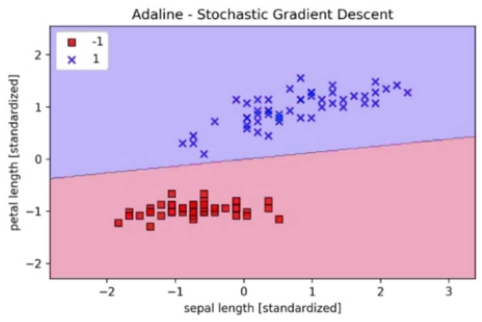
Another advantage of stochastic gradient descent is that we can use it for **online learning**. In online learning, our model is trained on the fly as new training data arrives. This is especially useful if we are accumulating large amounts of data, for example, customer data in web applications. Using 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.

### 4.4 Mini-batch learning

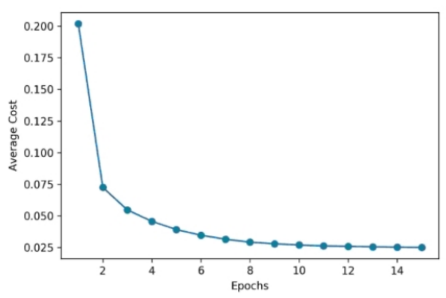
A compromise between batch gradient descent and stochastic gradient descent is so-called **mini-batch learning**. Mini-batch learning can be understood as applying batch gradient descent to smaller subsets of the training data, for example, 32 samples at a time. The advantage over batch gradient descent is that convergence is reached faster via mini-batches because of the more frequent weight updates. Furthermore, mini-batch learning allows us to replace the 'for' loop over the training samples in stochastic gradient descent with vectorized operations, which can further improve the computational efficiency of our learning algorithm.

### 4.5 Implementing stochastic gradient descent in Python

*Raschka and Mirjalili*[1] make a few adjustments to modify the Adaline learning rule using gradient descent to update the weights via stochastic gradient descent. Inside the `fit` method, weights are now updated after each training sample. Furthermore, we will implement an additional `partial_fit` method, which does not reinitialize the weights, for online learning. In order to check whether our algorithm converged after training, cost as the average cost of the training samples in each epoch is calculated. Additionally, an option to shuffle the training data before each epoch is added to avoid repetitive cycles when we are optimizing the cost function; via the `random_state` parameter, specification of a random seed is allowed for reproducibility.



(a) Decision boundary



(b) Cost per epoch

Figure 7: Decision boundary learned by AdalineSGD and the cost for each epoch. As we can see, when using stochastic gradient descent, the average cost goes down pretty quickly, and the final decision boundary after 15 epochs looks similar to the batch gradient descent Adaline.

The _shuffle method that we are now using in the `AdalineSGD` classifier works as follows: via the `permutation` function in `np.random`, a random sequence of unique numbers in the range 0 to 100 is generated. Those numbers can then be used as indices to shuffle the feature matrix and class label vector.

We can then use the `fit` method to train the `AdalineSGD` classifier and use our `plot_decision_regions` to plot our training results (shown in figure 7).

As we can see, the average cost goes down pretty quickly, and the final decision boundary after 15 epochs looks similar to the batch gradient descent Adaline. If we want to update our model, for example, in an online learning scenario with streaming data, we could simply call the `partial_fit` method on individual samples —for instance `ada.partial_fit(X_std[0, :], y[0])`.

## References

[1] S. Raschka and V. Mirjalili, *Python Machine Learning, 2nd Ed.* . Birmingham, UK: Packt Publishing, 2 ed., 2017.

[2] B. Widrow and M. E. Hoff, “An Adaptive ”Adaline” Neuron Using Chemical ”Memistors”, Technical Report No. 1553-2,” tech. rep., Stanford University, Stanford, 1960.

[3] F. Rosenblatt, “The Perceptron: a Percieving and Recognizing Automation,” tech. rep., Cornell Aeronautical Laboratory, Inc., Buffalo, NY, 1957.