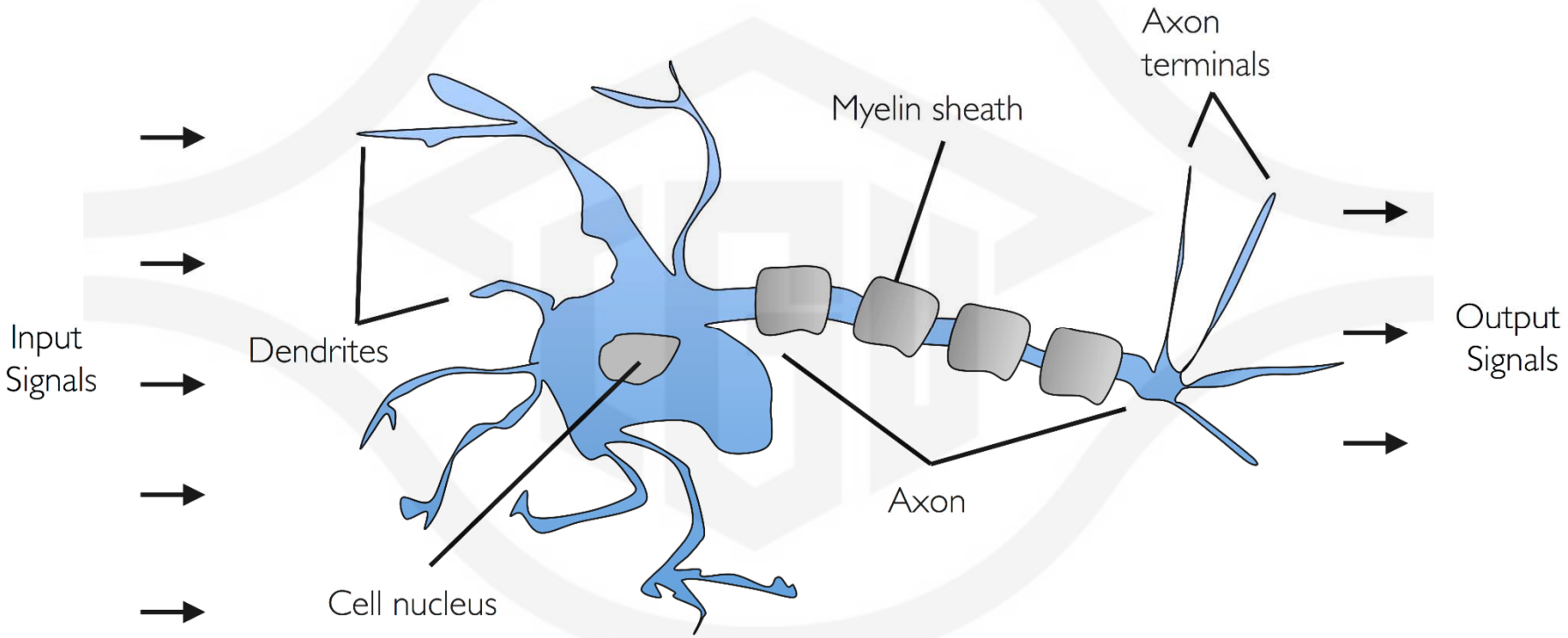
PATTERN RECOGNITION USING PYTHON

Perceptron

Wen-Yen Hsu
Dept Electrical Engineering
Chang Gung University, Taiwan

2019-Spring

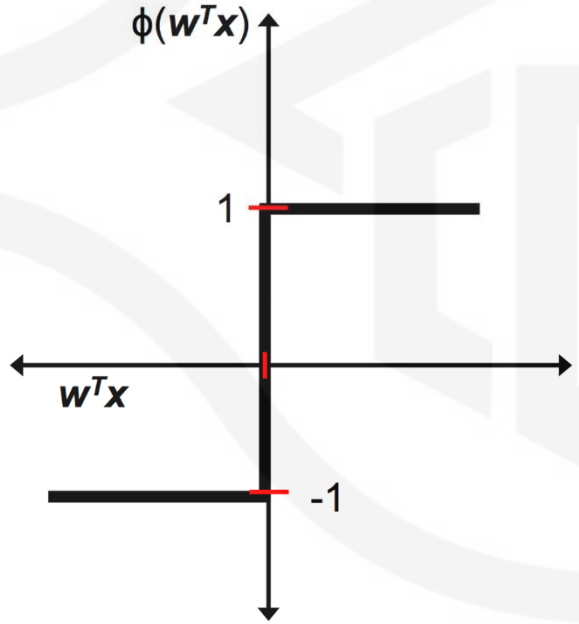
Biological Neuron

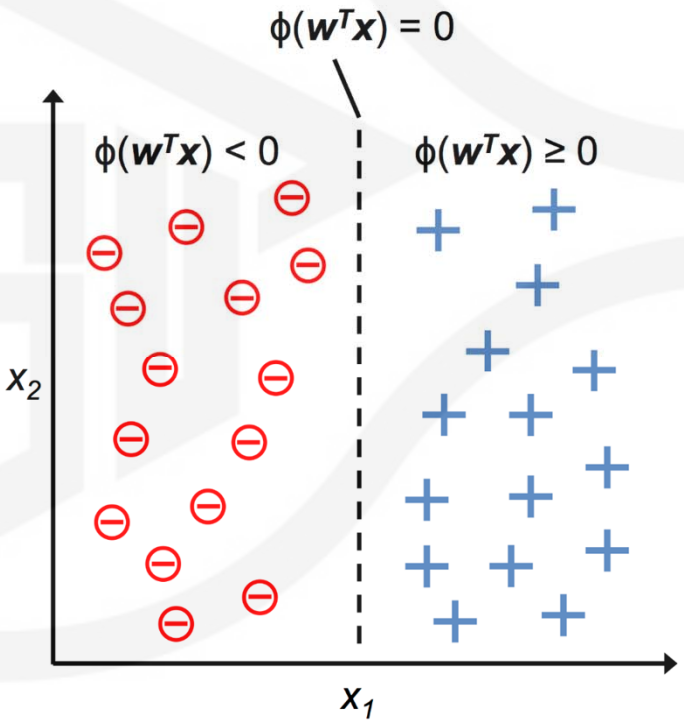


Artificial Neuron

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

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





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

Perceptron Learning Rule

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

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

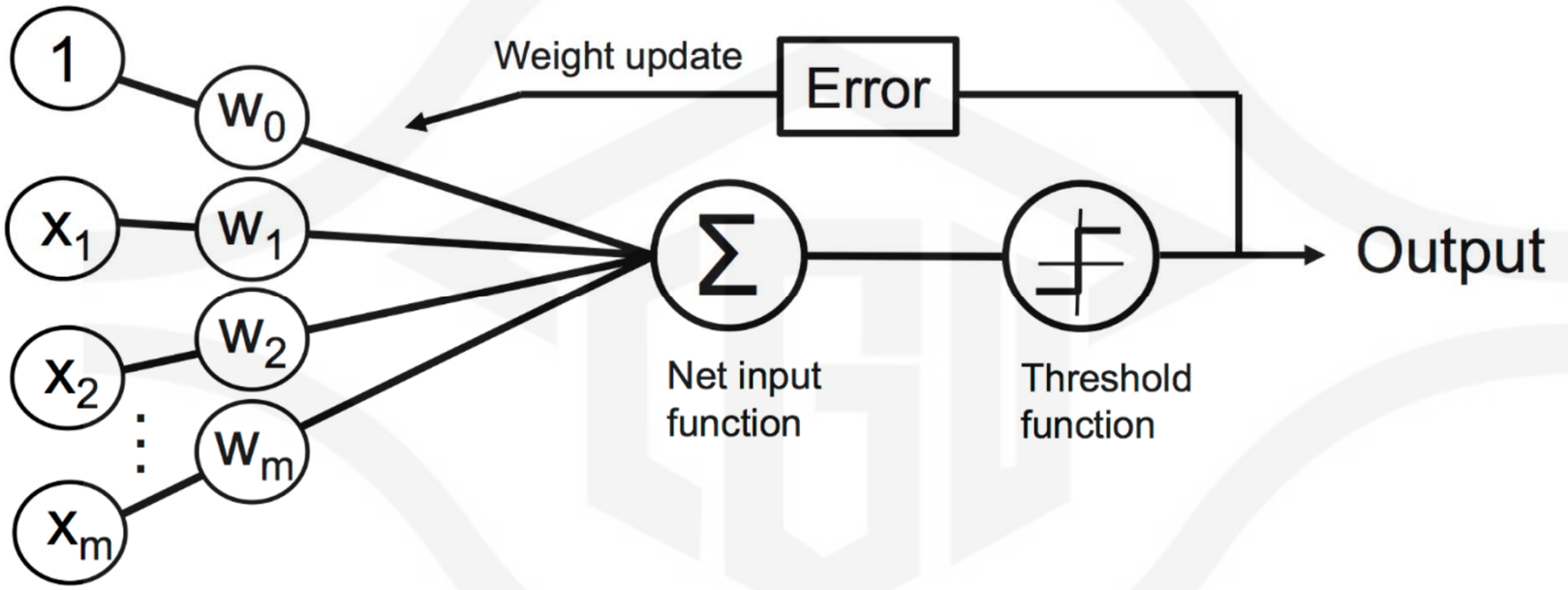
$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$

$y^{(i)}$ is the **true class label**

$\hat{y}^{(i)}$ is the **predicted class label**

η is the **learning rate**

Concept of the Perceptron



Implemention Perceptron

- Perceptron.py
- AdalineGD.py
- AdalineSGD.py
- Iris.csv

Perceptron

- Perceptron API (application programming interface) by object

```
class Perceptron(object):
    def __init__(self, eta=0.01, n_iter=1, random_state=1):
    def fit(self, X, y):
    def net_input(self, X):
    def predict(self, X):
```

- Plotting region picture by function

```
def plot_decision_regions(X, y, classifier, resolution=0.01):
```

Training method for Perceptron

- Implement fit

```
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.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
```

net_input and predict

- Implement net_input and predict

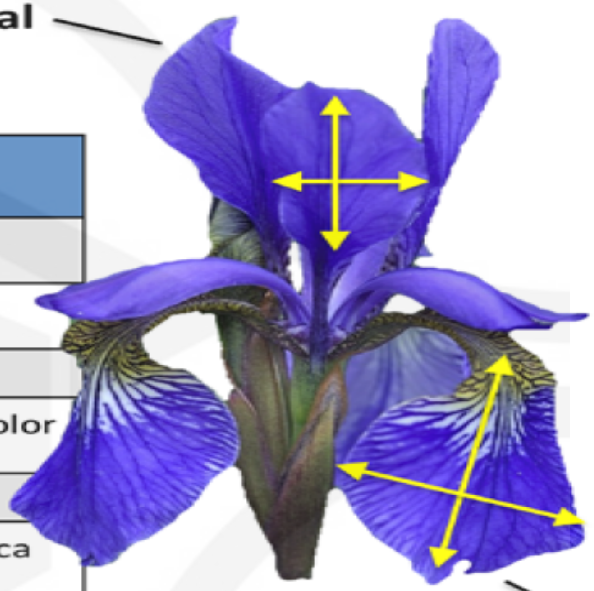
```
def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]
```

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

Iris Dataset

Samples
(instances, observations)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica



Features
(attributes, measurements, dimensions)

Class labels
(targets)

Iris.csv

- sepal_length
- petal_length
- species

	A	B	C	D	E
1	sepal_length	sepal_width	petal_length	petal_width	species
2	5.1	3.5	1.4	0.2	setosa
3	4.9	3	1.4	0.2	setosa
4	4.7	3.2	1.3	0.2	setosa
5	4.6	3.1	1.5	0.2	setosa
6	5	3.6	1.4	0.2	setosa
7	5.4	3.9	1.7	0.4	setosa
8	4.6	3.4	1.4	0.3	setosa
9	5	3.4	1.5	0.2	setosa
10	4.4	2.9	1.4	0.2	setosa
48	5.1	3.8	1.6	0.2	setosa
49	4.6	3.2	1.4	0.2	setosa
50	5.3	3.7	1.5	0.2	setosa
51	5	3.3	1.4	0.2	setosa
52	7	3.2	4.7	1.4	versicolor
53	6.4	3.2	4.5	1.5	versicolor
54	6.9	3.1	4.9	1.5	versicolor
55	5.5	2.3	4	1.3	versicolor
56	6.5	2.8	4.6	1.5	versicolor

Load Iris data

- Read data by pandas

```
import pandas as pd
df = pd.read_csv('iris.csv', header=None)
```

- Organize data

```
y = df.iloc[1:101, 4].values
y = np.where(y == 'setosa', -1, 1)
X = df.iloc[1:101, [0, 2]].values
X = X.astype(np.float)
```

Training Phase

- Initialize the object and training

```
max_inter = 7
ppn = Perceptron(eta=0.1, n_iter = max_inter)
ppn.fit(X, y)
```

- Test the effect of training iteration

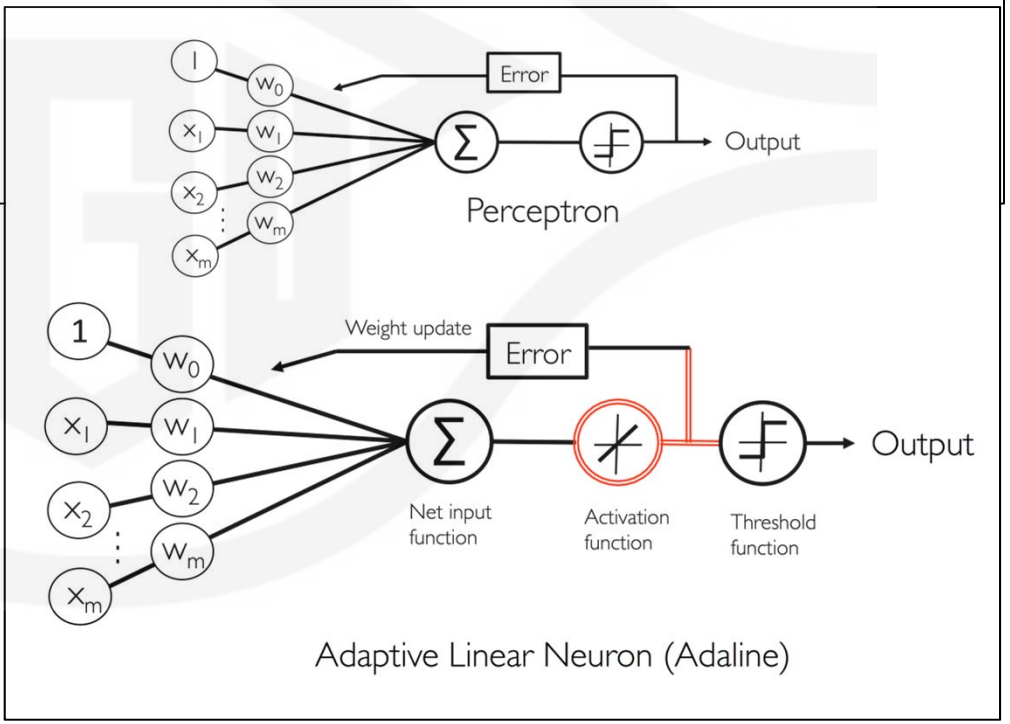
```
for i in range(1, 10, 1):
    ppn = Perceptron(eta=0.01, n_iter=i)
    ppn.fit(X, y)
```

- n_iter : Epochs

GD Adaline

- AdalineGD API

```
class AdalineGD(object):
    def __init__(self, eta=0.01, n_iter=5, random_state=1):
    def fit(self, X, y):
    def net_input(self, X):
    def activation(self, X):
    def predict(self, X):
```



Training method for Adaline Perceptron

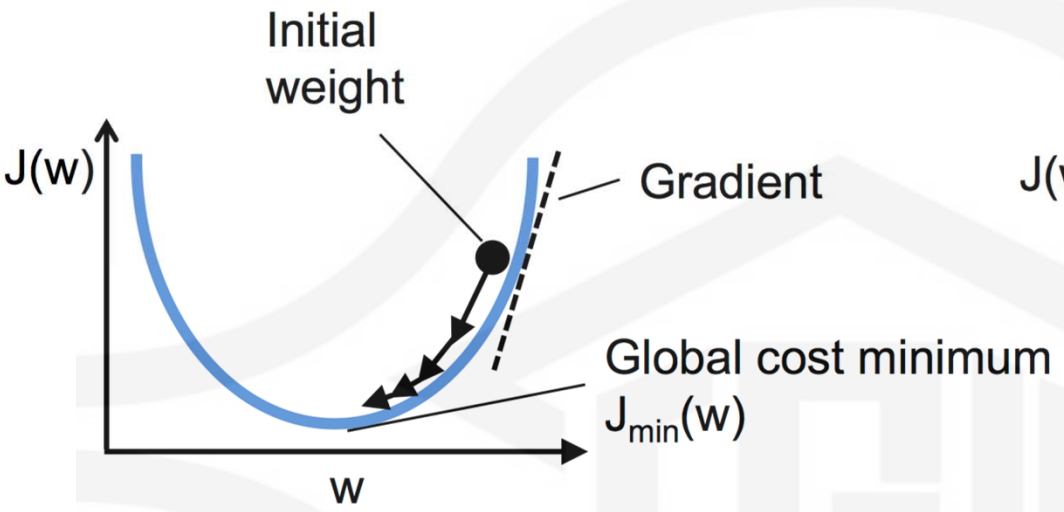
- Implement fit

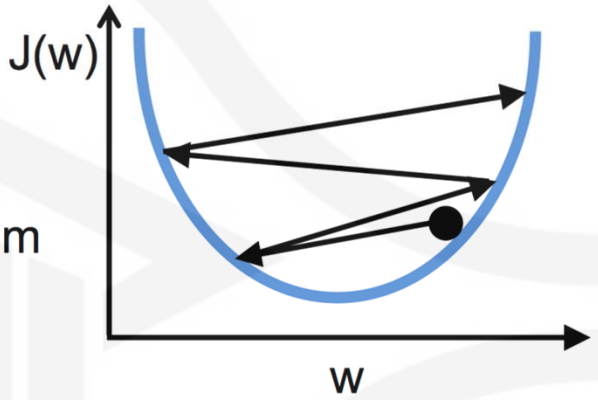
```
def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            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
```

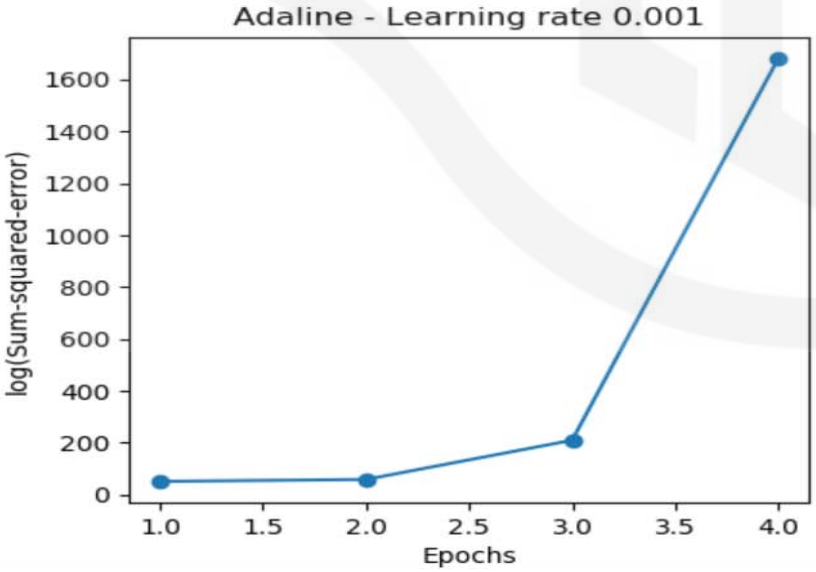
- Implement activation function

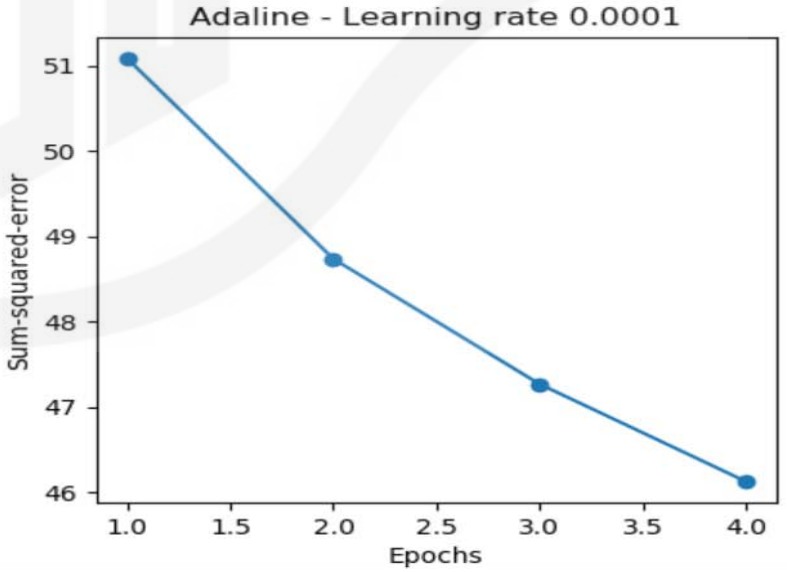
```
def activation(self, X):
        """Compute linear activation"""
        return X
```

Choose Learning Rate





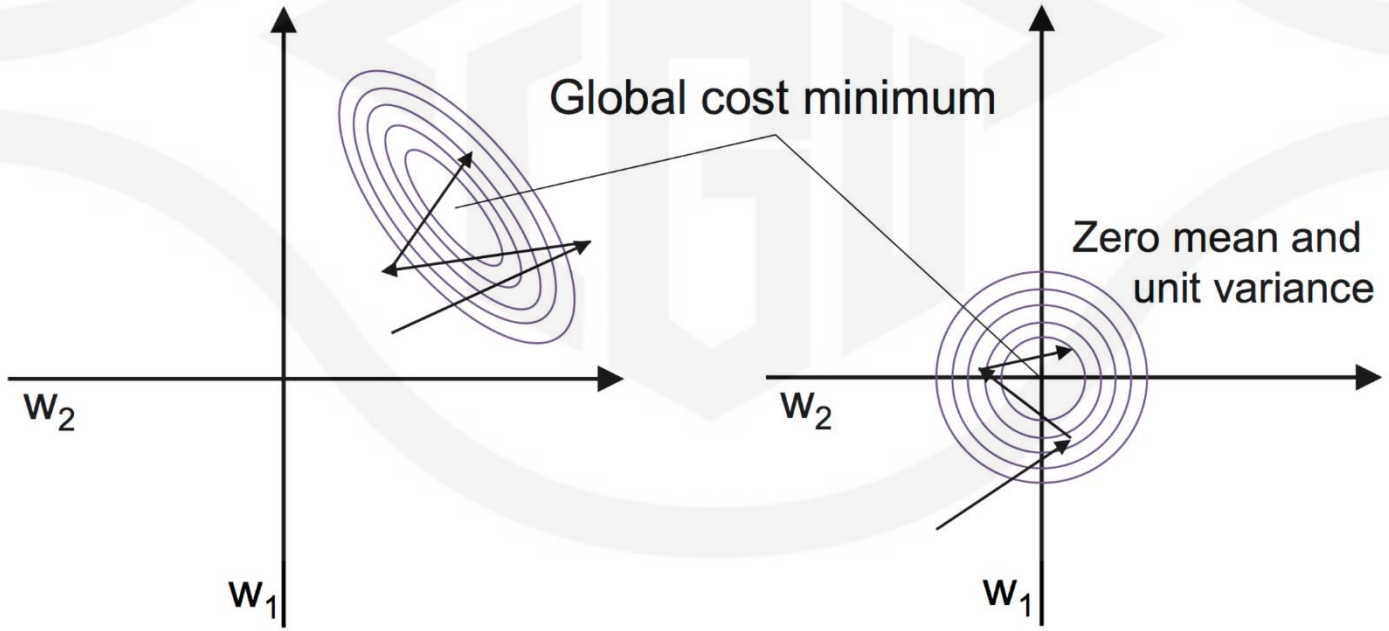




Feature Scaling

Apply standardization to each variable

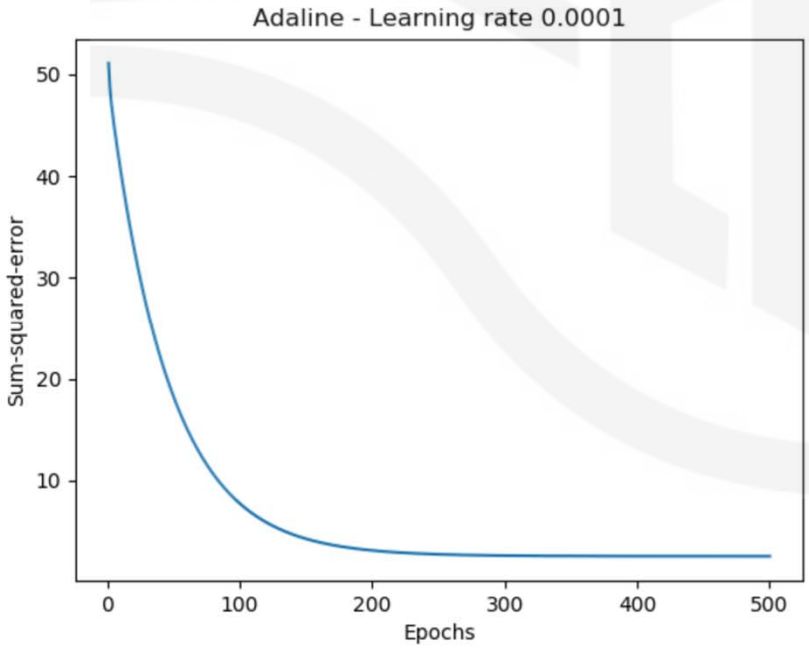
$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

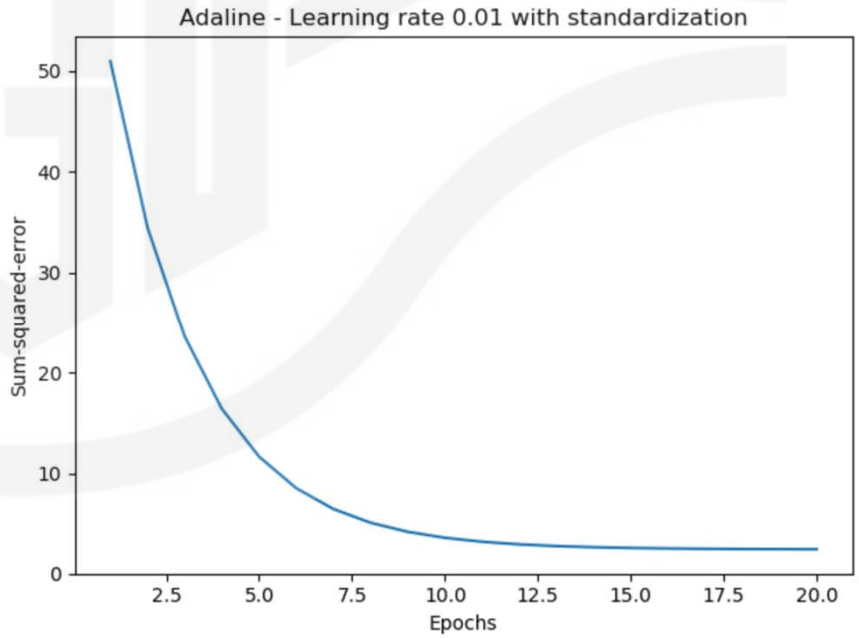


Standardization

- 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()
```





Stochastic Gradient Descent

- Stochastic gradient descent can reaches convergence much faster because of the more frequent weight updates
- Stochastic gradient descent can escape shallow local minima more readily if we are working with nonlinear cost functions

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

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

SGD Adaline (Stochastic Gradient Descent)

- AdalineSGD API

```
class AdalineSGD(object):
    def __init__(self, eta=0.01, n_iter=10, shuffle=True, random_state=None):
    def fit(self, X, y):
    def partial_fit(self, X, y):
    def _shuffle(self, X, y):
    def _initialize_weights(self, m):
    def _update_weights(self, xi, target):
    def net_input(self, X):
    def activation(self, X):
    def predict(self, X):
```

Training method for SGD Adaline Perceptron

- Implement

```
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
```

shuffle and update_weights

- Implement

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

```
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
```

Reference

- Sebastian Raschka, Vahid Mirjalili. Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow. Second Edition. Packt Publishing, 2017.