Lecture slides for this course
have been prepared by Dr. Le Minh Huy,
EEE, Phenikaa University

PHENIKAA UNIVERSITY

Deep Learning

Chapter 2 Building Neural Network from Scratch

Dr. Van-Toi NGUYEN

EEE, Phenikaa University

Chapter 2: Building Neural Network from Scratch

1. Shallow neural network
2. Deep neural network
3. Building neural network: step-by-step (modulation)
4. Regularization
5. Dropout
6. Batch Normalization
7. Optimizers
8. Hyper-parameters
9. Practice

Previous Lecture Overview

Chapter 1: Course Infor & Programming review - week 1

1. Course introduction and grades
2. History of Deep learning
3. Deep learning applications

Chapter 2: Building Neural Network from Scratch – week 2-7

1. Shallow neural network - week 2
2. Deep neural network - week 3
3. Building neural network: step-by-step (modulation) - week 3
4. Regularization - week 4
5. Dropout - week 4
6. Batch Normalization - week 5
7. Optimizers - week 6
8. Hyper-parameters - week 7
9. Practice- week

Midterm

Chapter 3: Convolutional Neural Network - week 8-10

1. Convolutional operator
2. History of CNN
3. Deep Convolutional Models
4. Layers in CNN
5. Applications of CNN
6. Practice

Midterm summary

Chapter 4: TensorFlow Library- week 11-13

1. Introduction to TensorFlow
2. Building a deep neural network with TensorFlow
3. Applications
4. Practice

Chapter 5: Recurrent Neural Network week 14-15

1. Unfolding Computational Graphs
2. Building a Recurrent Neural Networks
3. Long Short-Term Memory
4. Vision with Language Processing
5. Application of RNN
6. Practice

Previous Lecture Overview

45 hours at Classes:

Theory + Coding practice

90 hours shelf-study at home:

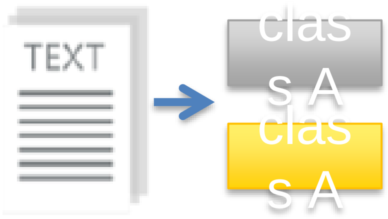
Theory + Coding practice

Previous Lecture Overview

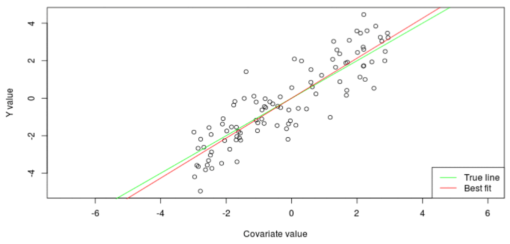
Supervised: Learning with a **labeled training** set of data
Example: learn the *classification* of images based on image labels (dogs/cats, day time, numbers, etc.)

Unsupervised: Discover **patterns** in **unlabeled** data
Example: *cluster* similar documents based on text

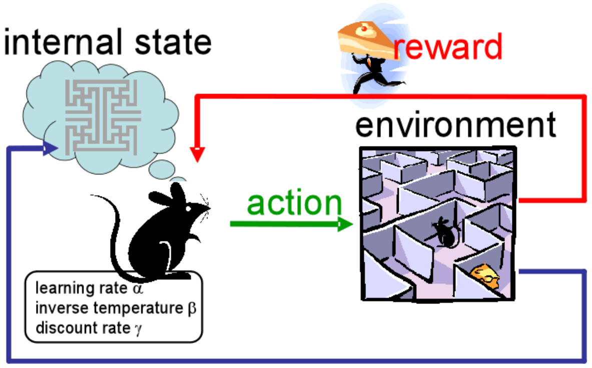
Reinforcement learning: learn to **act** based on **feedback/reward**
Example: learn to play Go, reward: *win or lose*

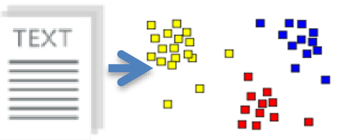


Classification



Regression



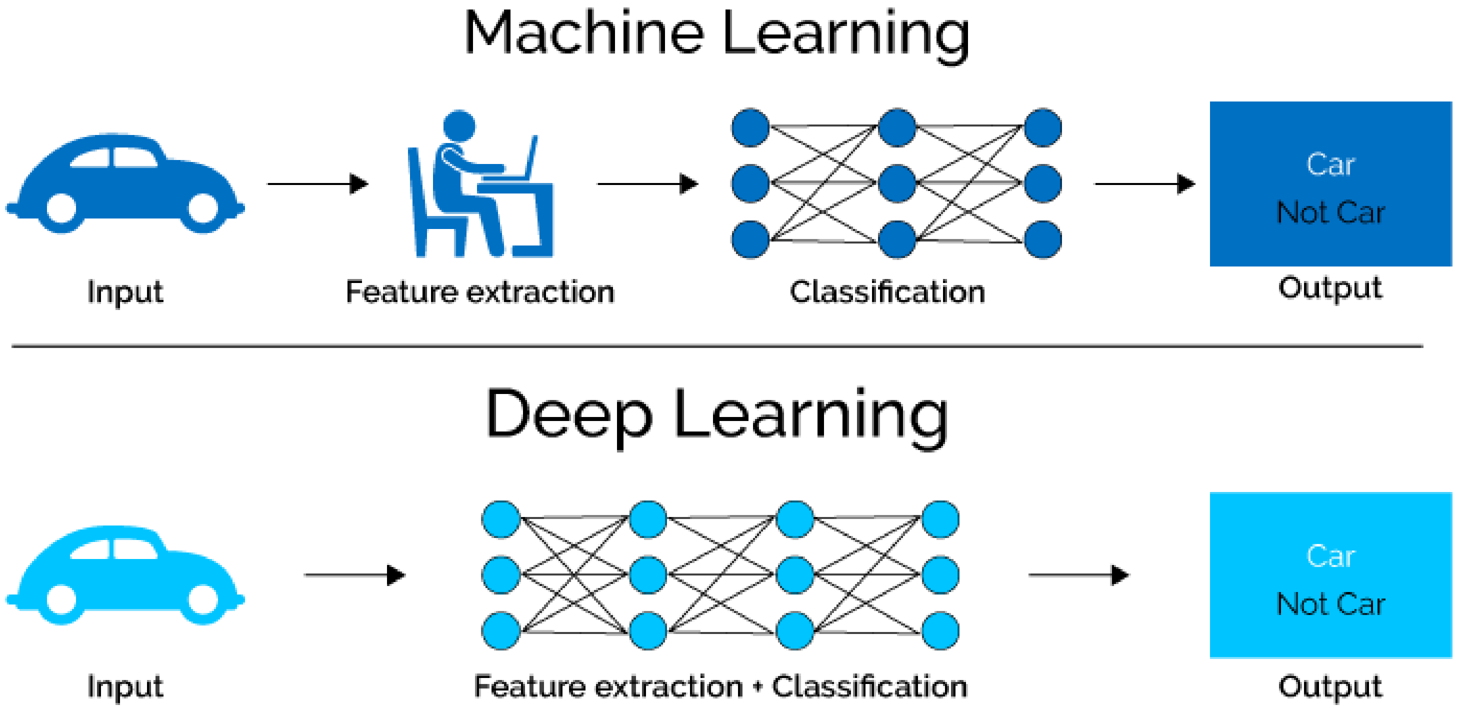


Clustering

Sources: *http://mbjoseph.github.io/2013/11/27/measure.html*
https://becominghuman.ai/the-very-basics-of-reinforcement-learning-154f28a79071

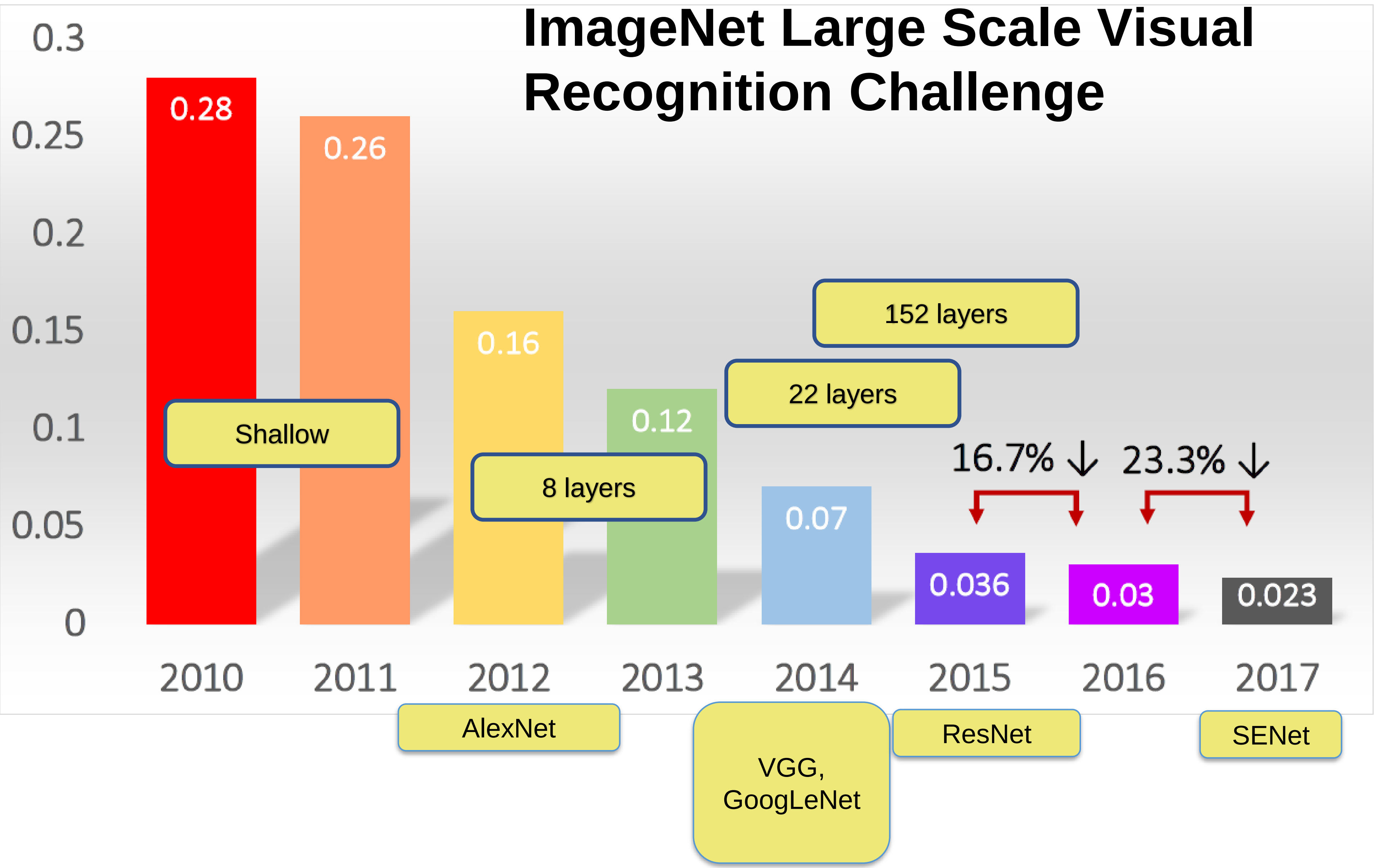
Previous Lecture Overview

- A sub-field of machine learning for learning **representations** of data.
- Exceptionally effective at **learning patterns**.
- Deep learning algorithms attempt to learn (multiple levels of) representation by using a **hierarchy of multiple layers**
- If you provide the system **tons of information**, it begins to understand it and respond in useful ways.



https://www.xenonstack.com/blog/static/public/uploads/media/machine-learning-vs-deep-learning.png

Previous Lecture Overview

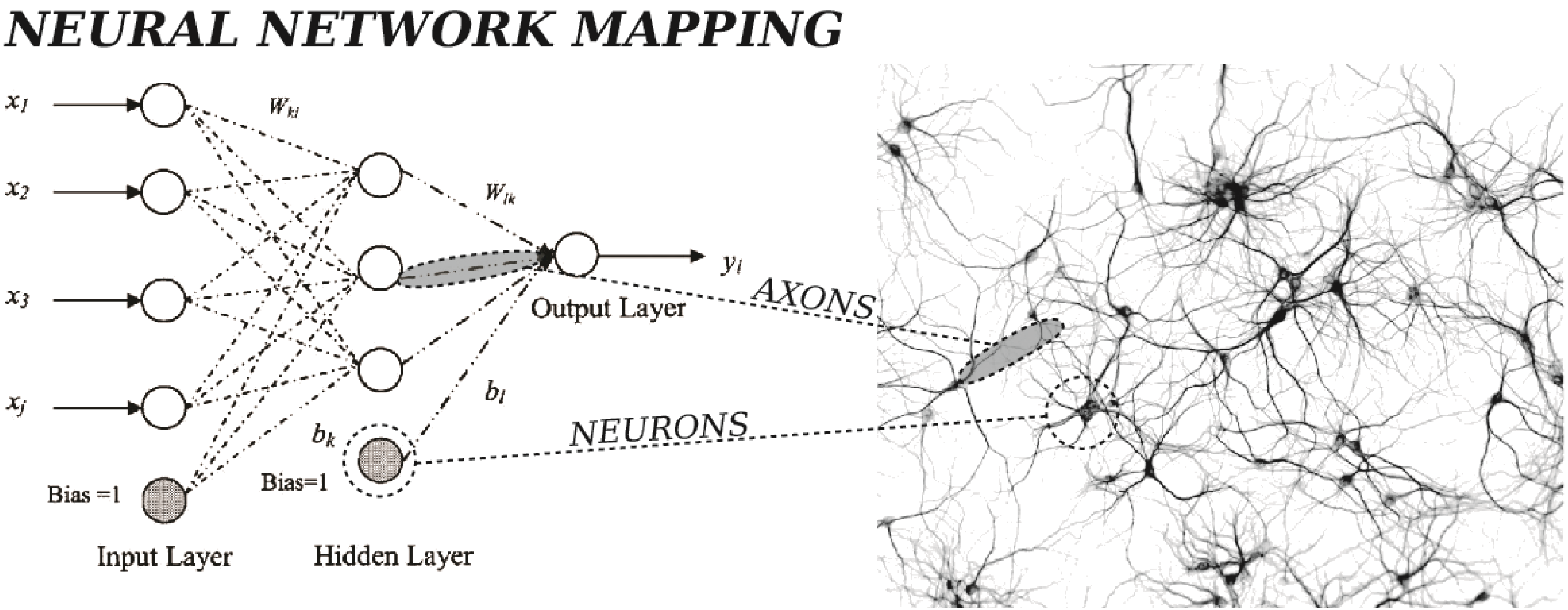


Basic of Neural Network

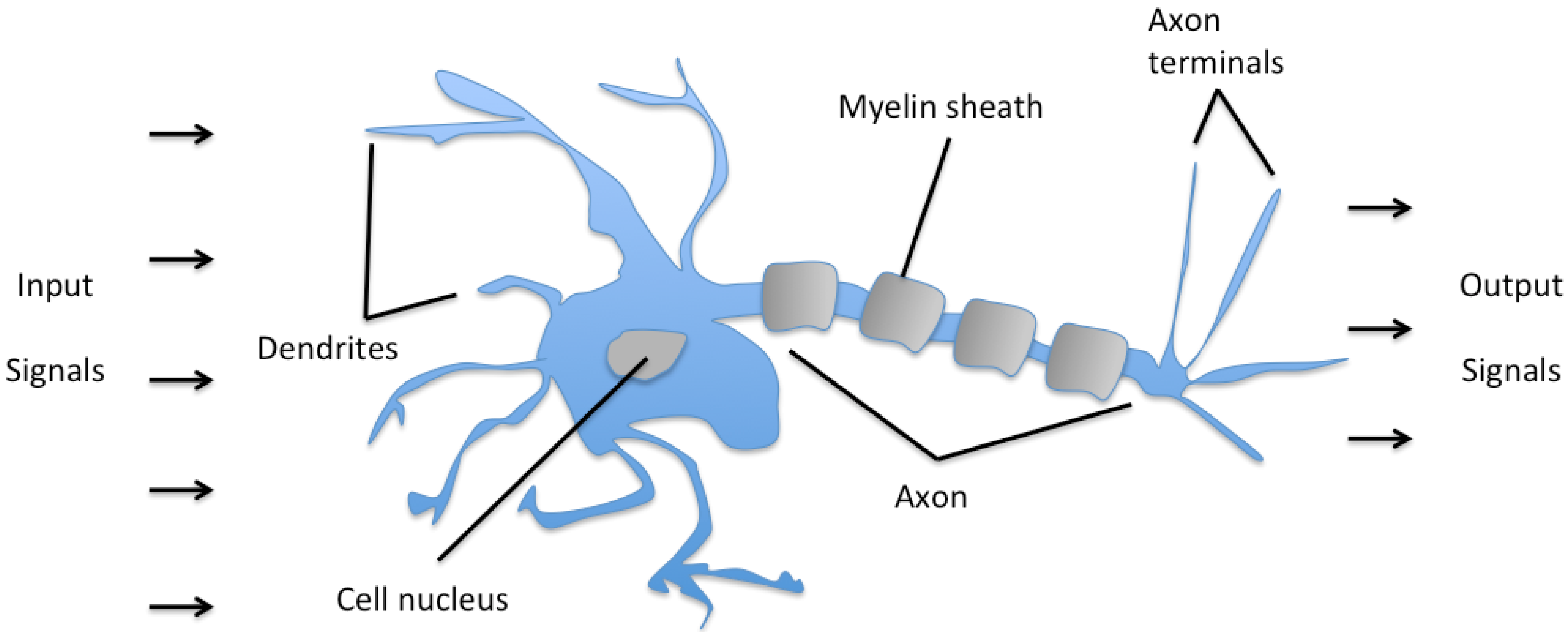
- The Perceptron and its Learning Rule (Frank Rosenblatt, 1957)
- Adaptive Linear Neuron and Delta Rule (Widrow & Hoff, 1960)
- Logistic Regression and Gradient Descent

1. Shallow Neural Network

Biologically inspired (akin to the neurons in a brain)



Artificial Neurons and the McCulloch-Pitts Model (1943)



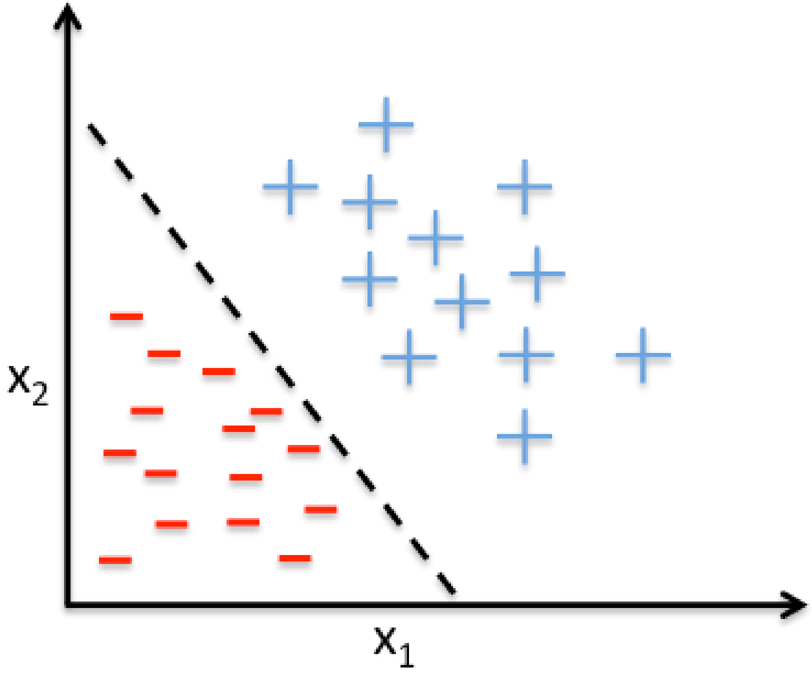
Schematic of a biological neuron.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4):115–133, 1943.

1. Shallow Neural Network

Frank Rosenblatt's Perceptron (1957)

- Supervised learning
- Single-layer
- Binary linear classifier
- To predict to which of 2 possible categories, a certain data point belongs on a set of input variables



Example of a linear decision boundary for binary classification.

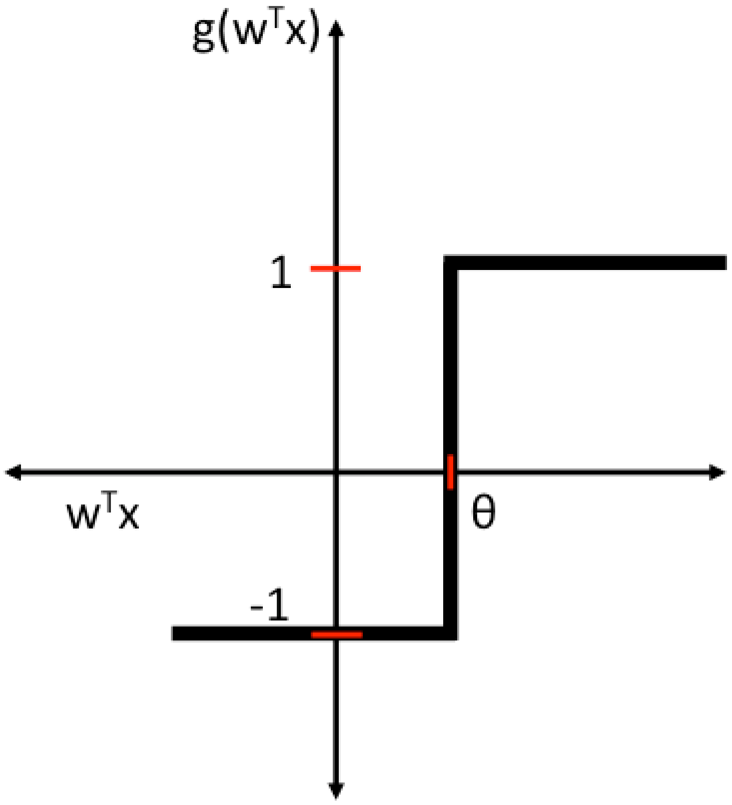
F. Rosenblatt. The perceptron, a perceiving and recognizing automaton Project Para. Cornell Aeronautical Laboratory, 1957.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4):115–133, 1943.

1. Shallow Neural Network

Frank Rosenblatt's Perceptron (1957)

- Positive class: +1
- Negative class: −1
- Activation function: g(z) = 1 if z ≥ θ; −1 o/w
  - where z is a linear combination of input values x and weights w, that is, $z = w_1x_1 + w_2x_2 + \ldots + w_mx_m = \sum_{j=1}^{m} x_j w_j = \mathbf{w}^T\mathbf{x}$
  - $\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$ is the weight vector
  - $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ is an m-dimensional sample from the training data set



Unit step function.

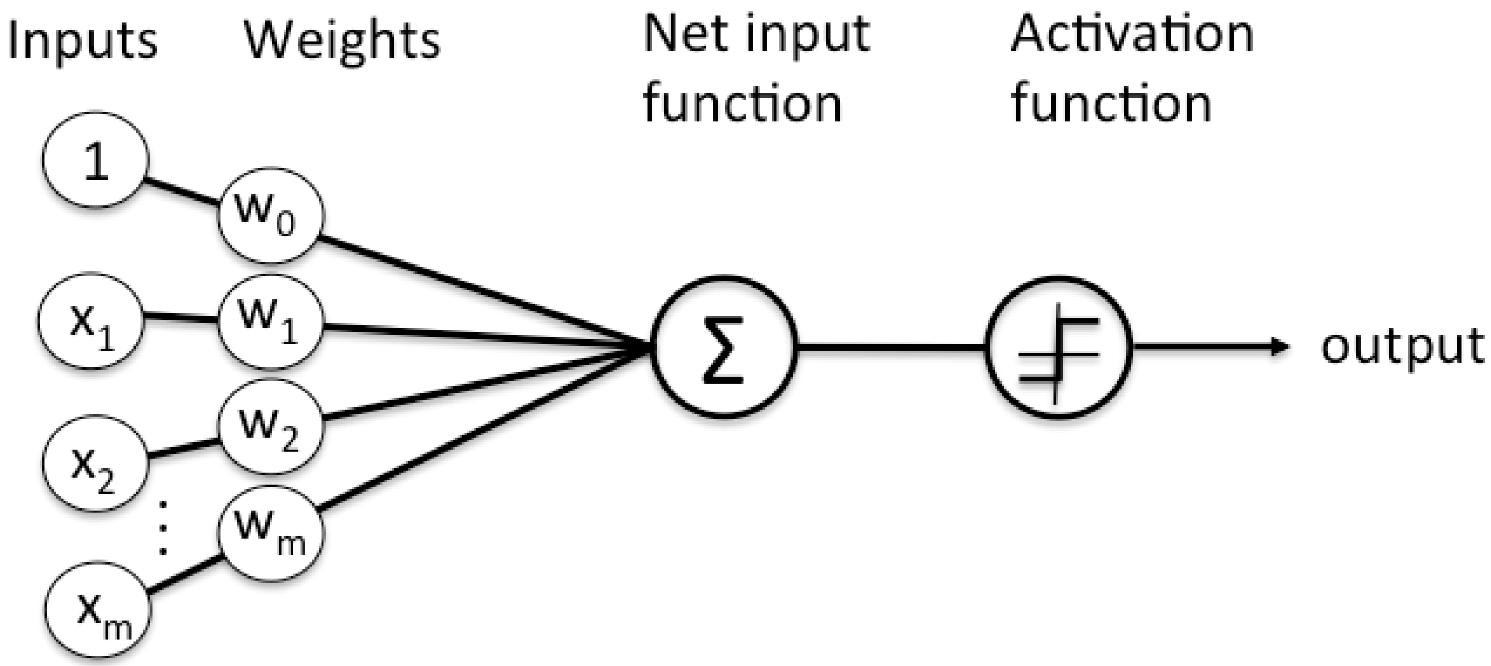
1. Shallow Neural Network

Frank Rosenblatt's Perceptron (1957)

- To simplify calculations, move θ to the origin such that the activation function becomes
- $g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$

1. Shallow Neural Network

Frank Rosenblatt's Perceptron (1957)



Schematic of Rosenblatt's perceptron.

1. Shallow Neural Network

Frank Rosenblatt's Perceptron (1957)

- Initialize the weights to 0 or small random numbers.
- For each training sample $x^{(i)}$:
  - Calculate the output value $y^{(i)} = g(z^{(i)})$
  - Update the weights as follows:

$$\mathbf{w_j := w_j + \eta \, (y'^{(i)} - y^{(i)})}$$

where η is the learning rate, $0.0 < \eta < 1$,
$y'^{(i)}$ is is the actual true class label, and
$y^{(i)}$ is the predicted class label.

1. Shallow Neural Network

Frank Rosenblatt’s Perceptron (1957)

- Classify the flowers in the **Iris** dataset using the perceptron rule
- Iris dataset from [UCI Machine Learning Repository]

More complete version:
https://github.com/rasbt/mlxtend/blob/master/mlxtend/classifier/perceptron.py

```
import numpy as np
class Perceptron(object):
    def __init__(self, eta=0.01, epochs=50):
        self.eta = eta
        self.epochs = epochs

    def train(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.errors_ = []
        for _ in range(self.epochs):
            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

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

    def predict(self, X):
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

1. Shallow Neural Network

Frank Rosenblatt’s Perceptron (1957)

Classify 2 flower species: Setosa and Versicolor using sepal length and petal length

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

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

- # sepal length and petal length
- X = df.iloc[0:100, [0,2]].values

1. Shallow Neural Network

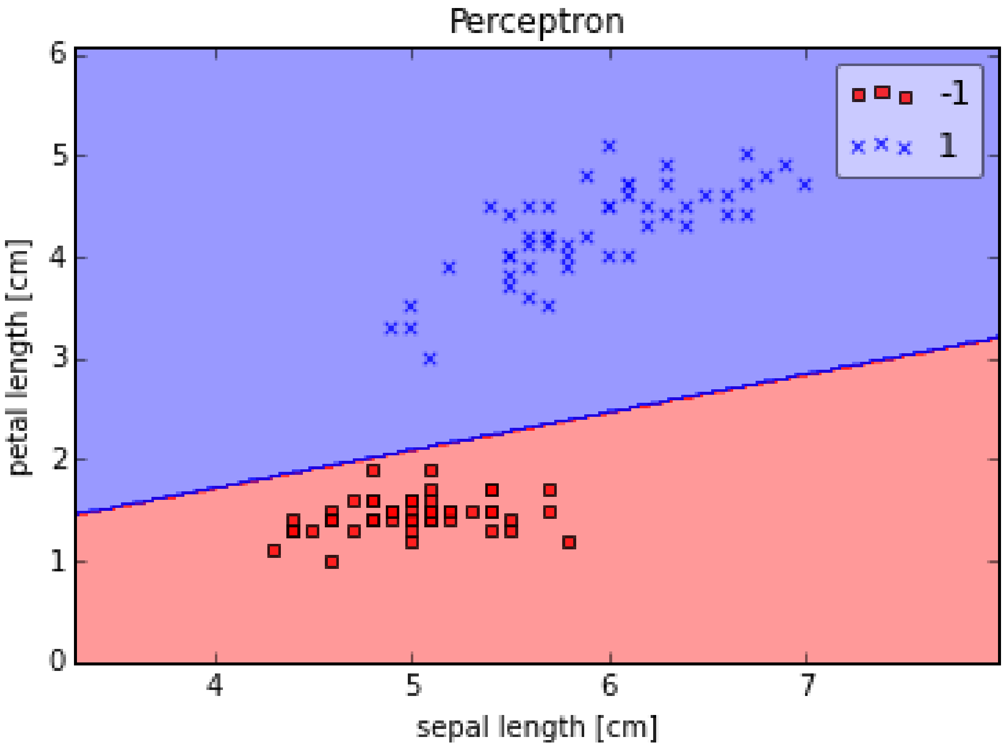
Frank Rosenblatt’s Perceptron (1957)

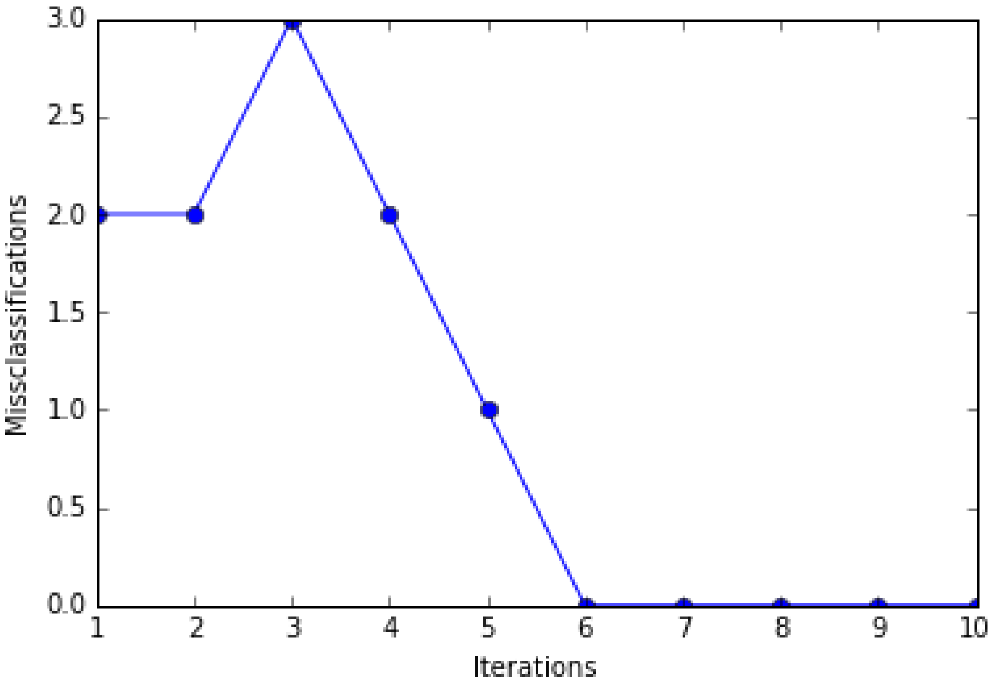
```
%matplotlib inline
import matplotlib.pyplot as plt
from mlxtend.plotting import plot_decision_regions

ppn = Perceptron(epochs=10, eta=0.1)
ppn.train(X, y)
print('Weights: %s' % ppn.w_)
plot_decision_regions(X, y, clf=ppn)
plt.title('Perceptron')
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.show()
plt.plot(range(1, len(ppn.errors_)+1), ppn.errors_, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Misclassifications')
plt.show()
```

1. Shallow Neural Network

Frank Rosenblatt's Perceptron (1957)

- Perceptron converges after 6^{th} iteration
- Weights: [-0.4 -0.68 1.82]

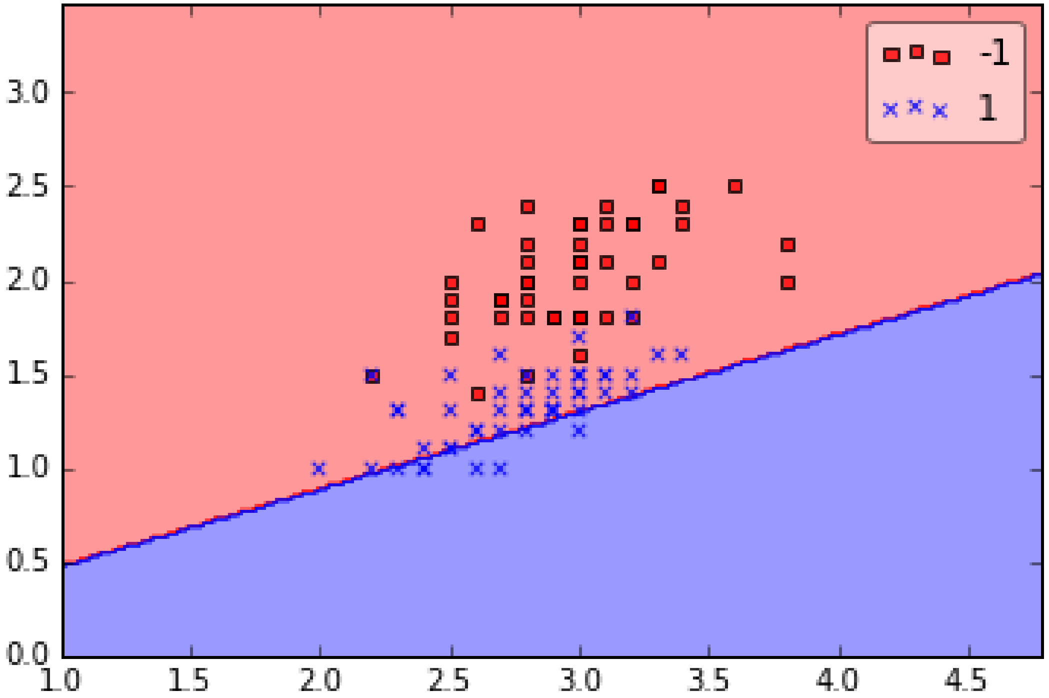


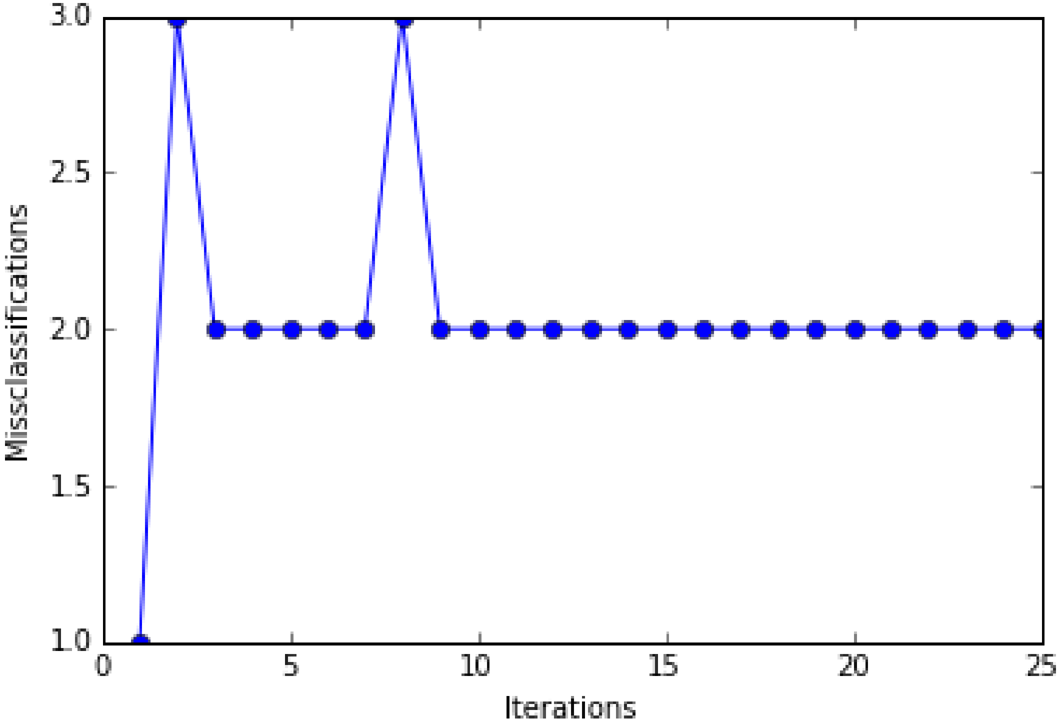


1. Shallow Neural Network

Frank Rosenblatt's Perceptron (1957)

- The 2 classes must be separable by a linear hyperplane
- If not, then the perceptron algorithm does NOT converge!





http://sebastianraschka.com/Articles/2015_singlelayer_neurons.html#problems-with-perceptrons

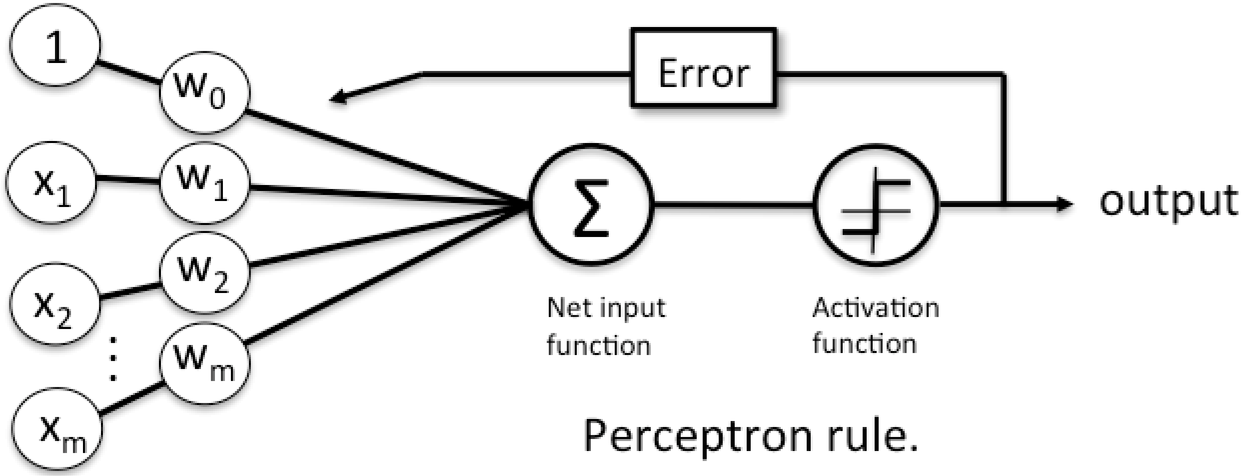
Adaptive Linear Neurons and the Delta Rule (1960)

- Bernard Widrow and Tedd Hoff proposed Adaptive Linear Neurons (Adaline)
- Linear activation function: $g(z) = z$.
- It is differentiable, so we can define a cost function and minimize it!

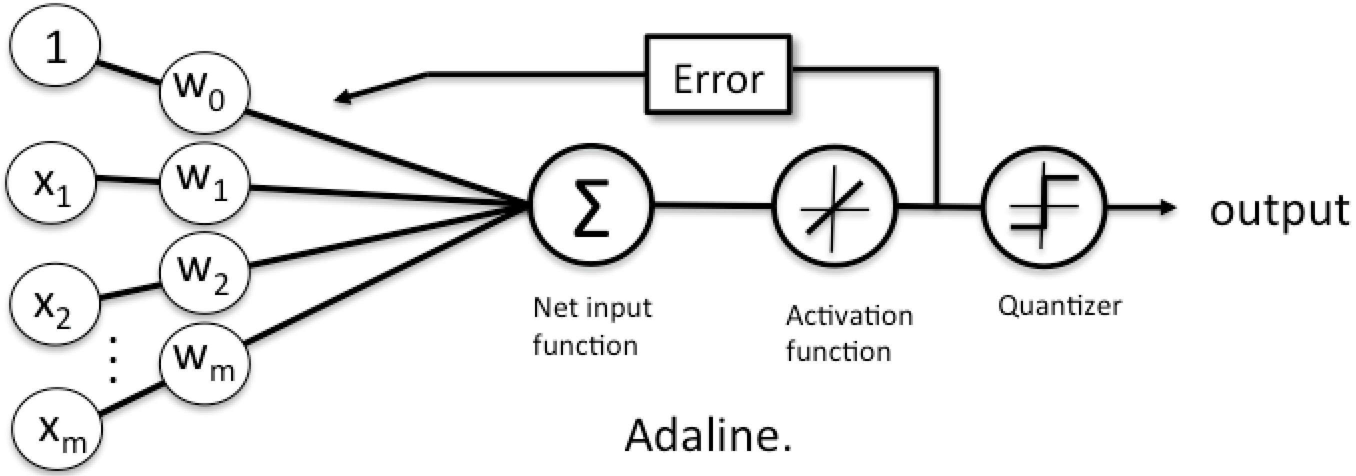
B. Widrow et al. Adaptive ”Adaline” neuron using chemical ”memistors”. Number Technical Report 1553-2. Stanford Electron. Labs., Stanford, CA, October 1960.

1. Shallow Neural Network

Adaptive Linear Neurons and the Delta Rule (1960)



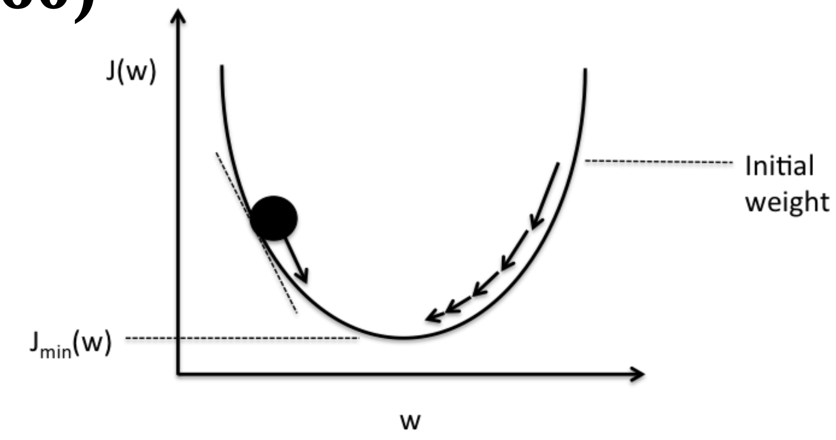
Perceptron rule.



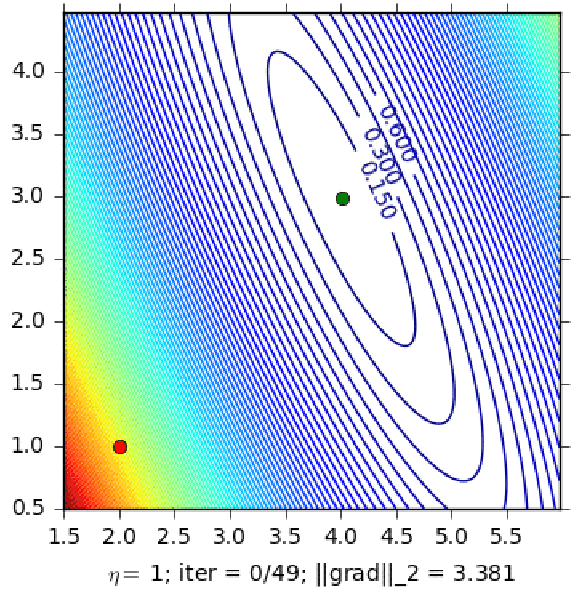
Adaline.

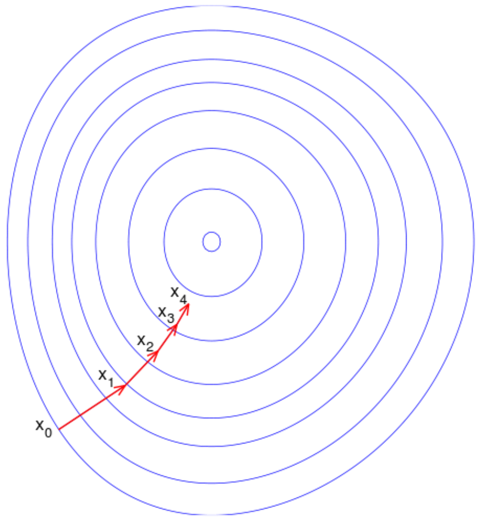
Adaptive Linear Neurons and the Delta Rule (1960)

- **Gradient Descent**
  - A first-order iterative optimization algorithm for finding the minimum of a function
  - Take steps proportional to the **negative of the gradient** of the function at the current point



Schematic of gradient descent.





Adaptive Linear Neurons and the Delta Rule (1960)

- Cost function: sum of squared errors (SSE)
  - $J(w) = \frac{1}{2}\sum_i (y'^{(i)} - y^{(i)})^2$
- To minimize SSE, we can use "gradient descent"
- A step in the opposite direction of gradient

$$\Delta w = -\alpha \nabla J(w)$$

where α is the learning rate, $0 < \alpha < 1$

- Thus, we need to compute the partial derivative of the cost function for each weight in the weight vector,

$$\Delta w_j = -\alpha \frac{\partial J}{\partial w_j}$$

Adaptive Linear Neurons and the Delta Rule (1960)

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

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

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

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

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

Adaptive Linear Neurons and the Delta Rule (1960)

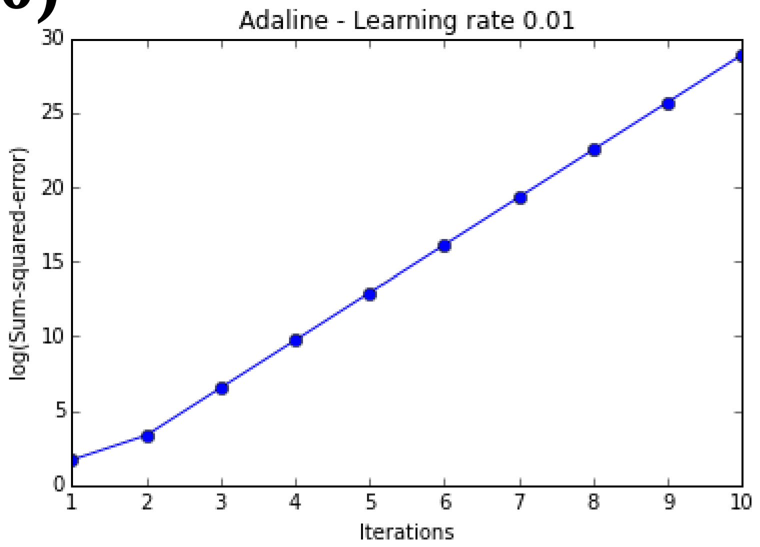
- A step in gradient descent:
  - $\Delta \mathrm{w_j} = -\alpha \frac{\partial J}{\partial w_j} = -\alpha \sum_i \left(y'^{(i)} - y^{(i)}\right)\left(-x_j^{(i)}\right) = \alpha \sum_i (y'^{(i)} - y^{(i)}) x_j^{(i)}$
- Update weight vector:
  - $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}$
- Differences with the perceptron rule
  - The output $y^{(i)}$ is a real number, not a class label as in perceptron learning rule.
  - Weight update is based on "all samples in the training set" (Batch GD)

Adaptive Linear Neurons and the Delta Rule (1960)

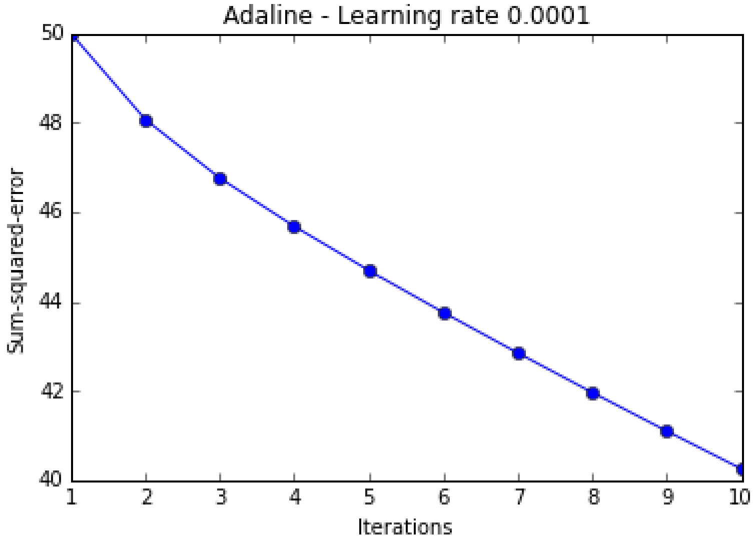
```
import numpy as np

class AdalineGD(object):
    def __init__(self, alpha=0.01, epochs=50):
        self.alpha = alpha
        self.epochs = epochs
    def train(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []
        for i in range(self.epochs):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.alpha * X.T.dot(errors)
            self.w_[0] += self.alpha * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self
def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]
    def activation(self, X):
        return self.net_input(X)
    def predict(self, X):
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

Adaptive Linear Neurons and the Delta Rule (1960)

```
ada = AdalineGD(epochs=10, alpha=0.01).train(X, y)
plt.plot(range(1, len(ada.cost_)+1), np.log10(ada.cost_), marker='o')
plt.xlabel('Iterations')
plt.ylabel('log(Sum-squared-error)')
plt.title('Adaline - Learning rate 0.01')
plt.show()
```

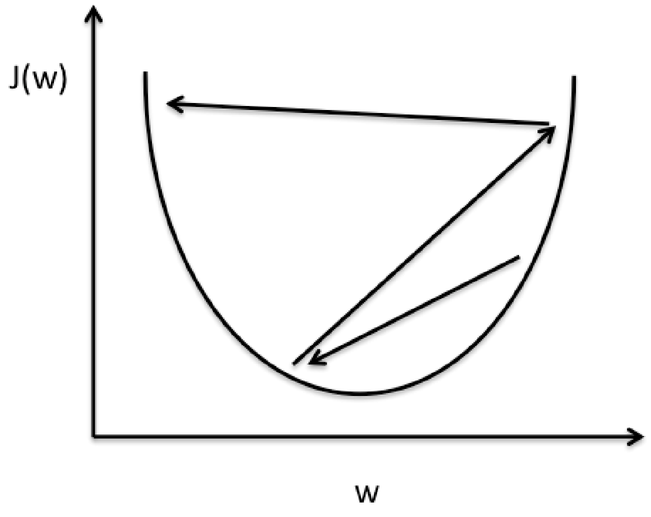



```
ada = AdalineGD(epochs=10, alpha=0.0001).train(X, y)
plt.plot(range(1, len(ada.cost_)+1), ada.cost_, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Sum-squared-error')
plt.title('Adaline - Learning rate 0.0001')
plt.show()
```

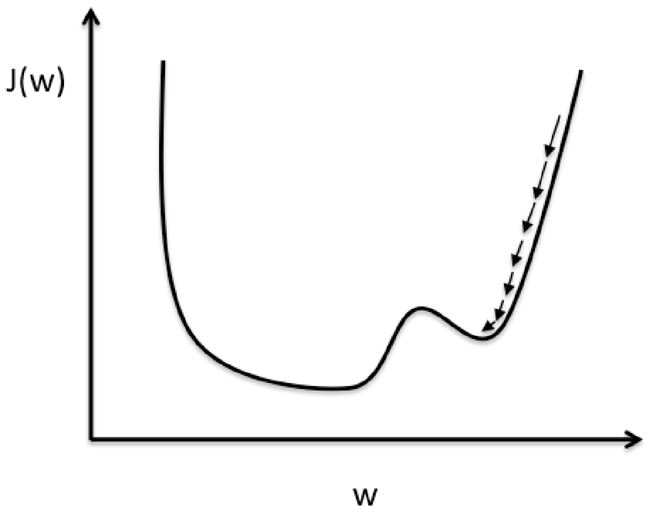


Adaptive Linear Neurons and the Delta Rule (1960)

- If the learning rate is TOO LARGE, gradient descent will overshoot the minima and diverge.
- If the learning rate is too small, gradient descent will require too many epochs to converge and can become trapped in local minima more easily.



Large learning rate: Overshooting.



Small learning rate: Many iterations until convergence and trapping in local minima.

Adaptive Linear Neurons and the Delta Rule (1960)

- If features are scaled on the same scale, gradient descent converges faster and prevents weights from becoming too small (weight decay).

- Common way for feature scaling

$$x_{j,std} = \frac{x_j - \mu_j}{\sigma_j}$$

where μ_j is the sample mean of the feature x_j and σ_j the standard deviation.

- After standardization, the features will have unit variance and centered around mean zero.

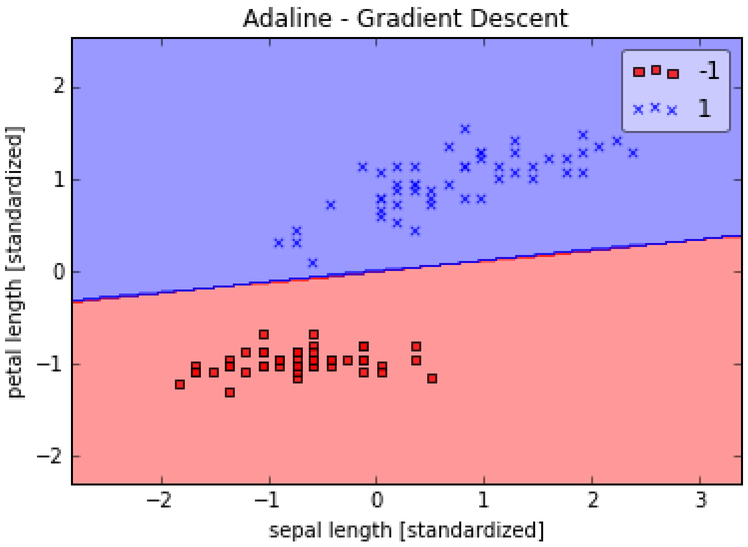
Adaptive Linear Neurons and the Delta Rule (1960)

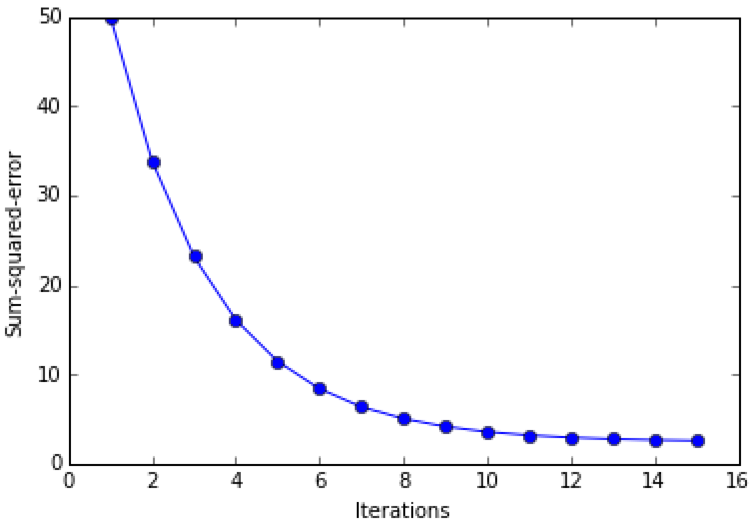
```
# standardize features
X_std = np.copy(X)
X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

1. Shallow Neural Network

Adaptive Linear Neurons and the Delta Rule (1960)

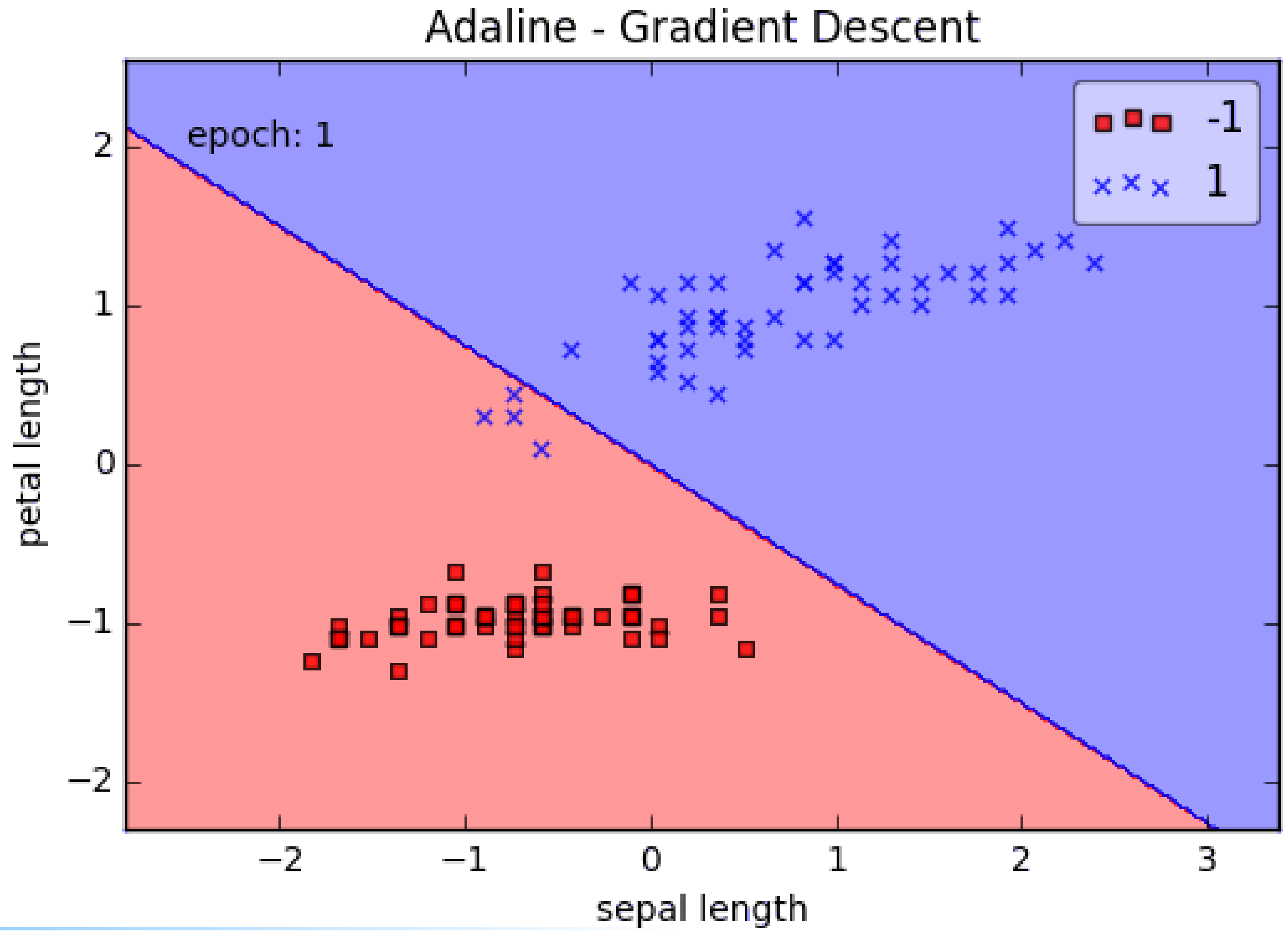
```
%matplotlib inline
import matplotlib.pyplot as plt
from mlxtend.plotting import plot_decision_regions

ada = AdalineGD(epochs=15, eta=0.01)

ada.train(X_std, y)
plot_decision_regions(X_std, y, clf=ada)
plt.title('Adaline - Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.show()

plt.plot(range(1, len( ada.cost_)+1), ada.cost_, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Sum-squared-error')
plt.show()
```





1. Shallow Neural Network

Adaptive Linear Neurons and the Delta Rule (1960)



PHENIKAA UNIVERSITY

Adaptive Linear Neurons and the Delta Rule (1960)

```
import numpy as np

class AdalineSGD(object):
    def __init__(self, alpha=0.01, epochs=50):
        self.alpha = alpha
        self.epochs = epochs
    def train(self, X, y, reinitialize_weights=True):
        if reinitialize_weights:
            self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []
        for i in range(self.epochs):
            for xi, target in zip(X, y):
                output = self.net_input(xi)
                error = (target - output)
                self.w_[1:] += self.alpha * xi.dot(error)
                self.w_[0] += self.alpha * error
            cost = ((y - self.activation(X))**2).sum() /
2.0
            self.cost_.append(cost)
        return self

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

    def activation(self, X):
        return self.net_input(X)

    def predict(self, X):
        return np.where(self.activation(X) >= 0.0,
1, -1)
```

Adaptive Linear Neurons and the Delta Rule (1960)

- Batch Gradient Descent (BGD)
  - Cost function is minimized based on the complete training dataset (all samples)
- Stochastic Gradient Descent (SGD)
  - Weights are incrementally updated after each individual training sample
  - Converges faster than BGD since weights are updated immediately after each training sample
  - Computationally more efficient, especially for large datasets
- Mini-batch Gradient Descent (MGD)
  - Compromise between BGD and SGD, dataset is divided into mini-batches
  - Smoother convergence than SGD

Adaptive Linear Neurons and the Delta Rule (1960)

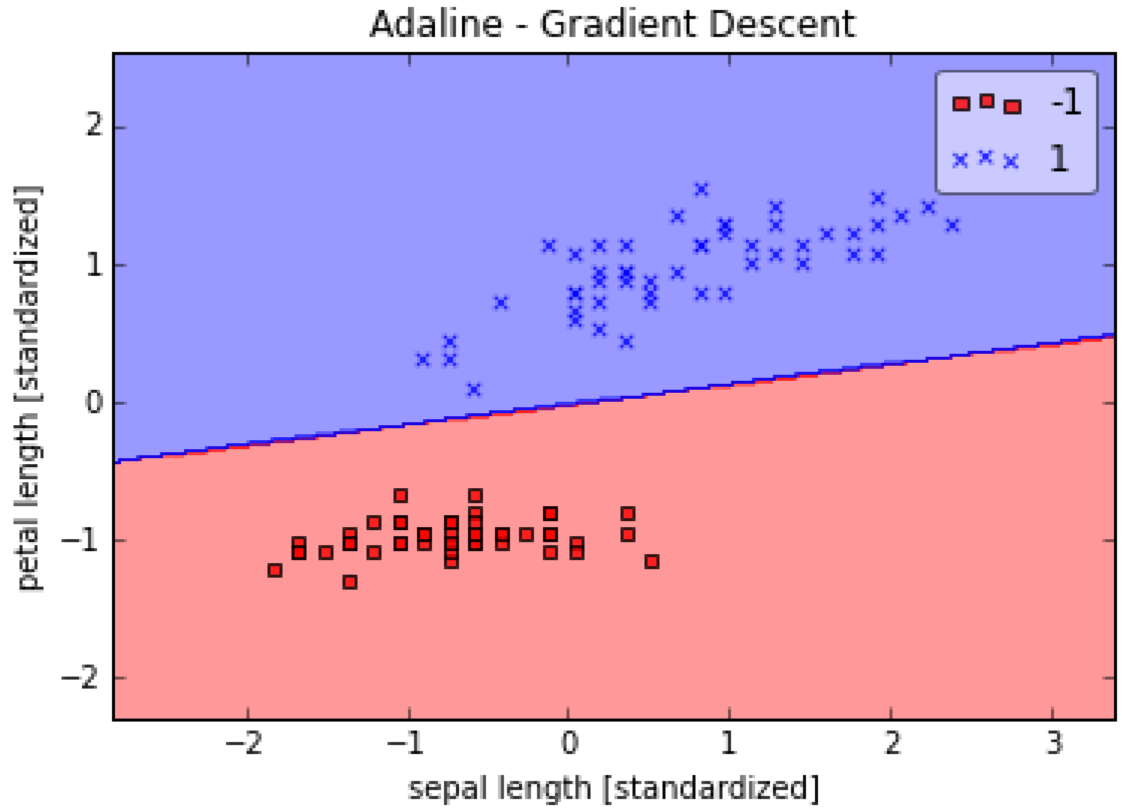
Adaline with SGD

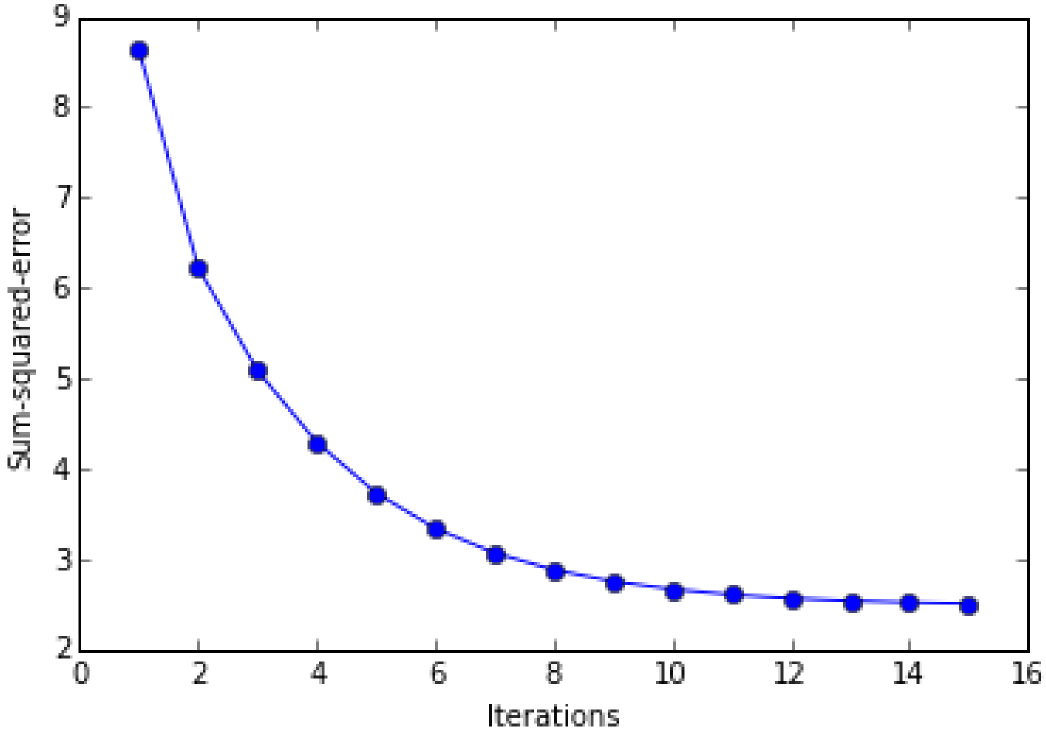
```
import numpy as np

class AdalineSGD(object):
    def __init__(self, alpha=0.01, epochs=50):
        self.alpha = alpha
        self.epochs = epochs
    def train(self, X, y, reinitialize_weights=True):
        if reinitialize_weights:
            self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []
        for i in range(self.epochs):
            for xi, target in zip(X, y):
                output = self.net_input(xi)
                error = (target - output)
                self.w_[1:] += self.alpha * xi.dot(error)
                self.w_[0] += self.alpha * error
            cost = ((y - self.activation(X))**2).sum() / 2.0
            self.cost_.append(cost)
        return self

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

    def activation(self, X):
        return self.net_input(X)

    def predict(self, X):
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

1. Shallow Neural Network

Adaptive Linear Neurons and the Delta Rule (1960)

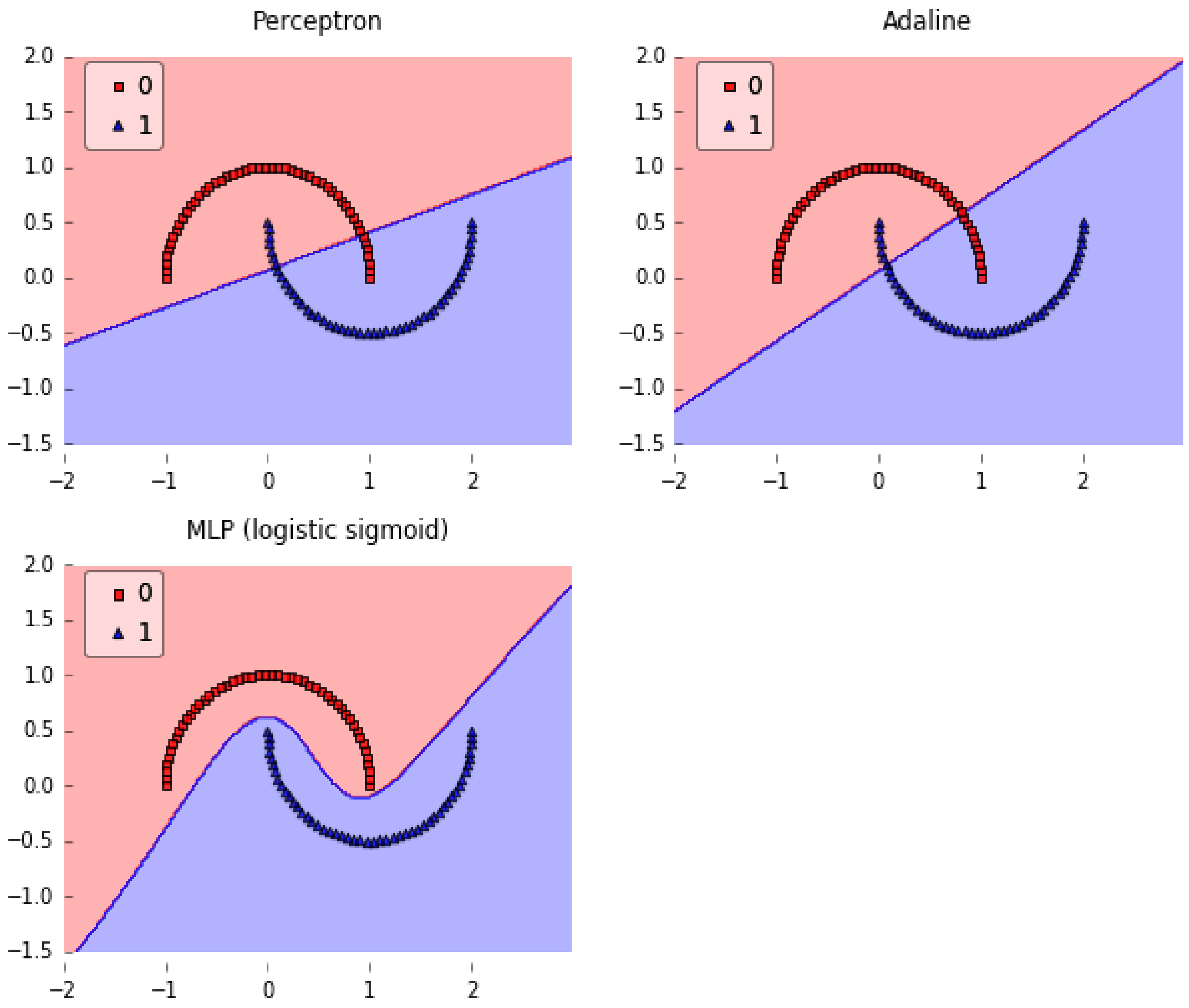
Adaline with SGD





Logistic Regression

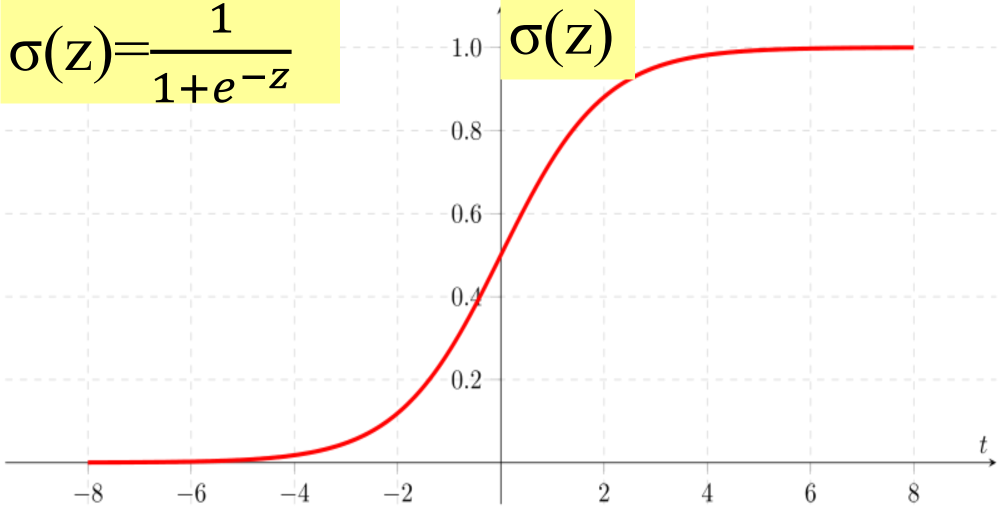
Perceptron vs. Adaline vs. Multi-Layer Perceptrons (Logistic Regression)



Logistic Regression

- Definition:
  - Given input $x \in \mathcal{R}^{n_x}$, calculate the probability $\hat{y} = P(y = 1|x), 0 \leq \hat{y} \leq 1$.
- Parameters:
  - Weights: $w \in \mathcal{R}^{n_x}$
  - Bias: $b \in \mathcal{R}$
- Output:
  - $\hat{y} = \sigma(z) = \sigma(w^T x + b)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the **sigmoid activation function**

$\sigma(z)=\frac{1}{1+e^{-z}}$

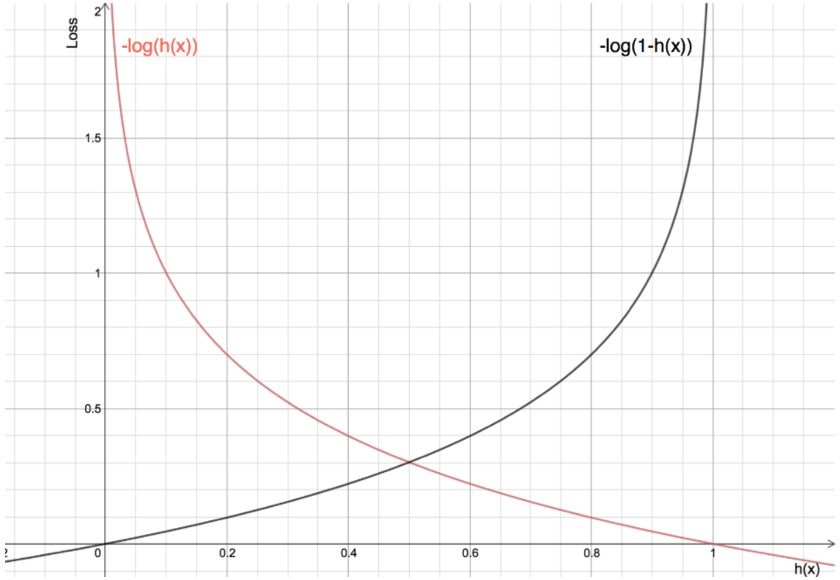


If z is large positive number, σ(z) → 1

If z is small negative number, σ(z) → 0

Logistic Regression

- For i^{th} input $x^{(i)}$, $\hat{y}^{(i)} = \sigma\left(z^{(i)}\right) = \sigma\left(w^T x^{(i)} + b\right)$ where $\sigma\left(z^{(i)}\right) = \frac{1}{1+e^{-z^{(i)}}}$
- For each labeled data $(x^{(i)}, y^{(i)})$, we could like $\hat{y}^{(i)} \approx y^{(i)}$, where $\hat{y}^{(i)}$is the predicted output and $y^{(i)}$is the actual expected ground truth value.
- Loss (error)function for each input is defined using Cross-Entropy or Log Loss
$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$
- Intuition
  - If y=1, $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$
    - Need large $\hat{y}$
  - If y=0, $\mathcal{L}(\hat{y}, y) = -\log(1 - \hat{y})$
    - Need small $\hat{y}$



Note: we do not use sum of squared errors because it will be not convex in logistic regression

PHENIKAA UNIVERSITY

Logistic Regression

- Cost function is the average of all cross-entropy losses

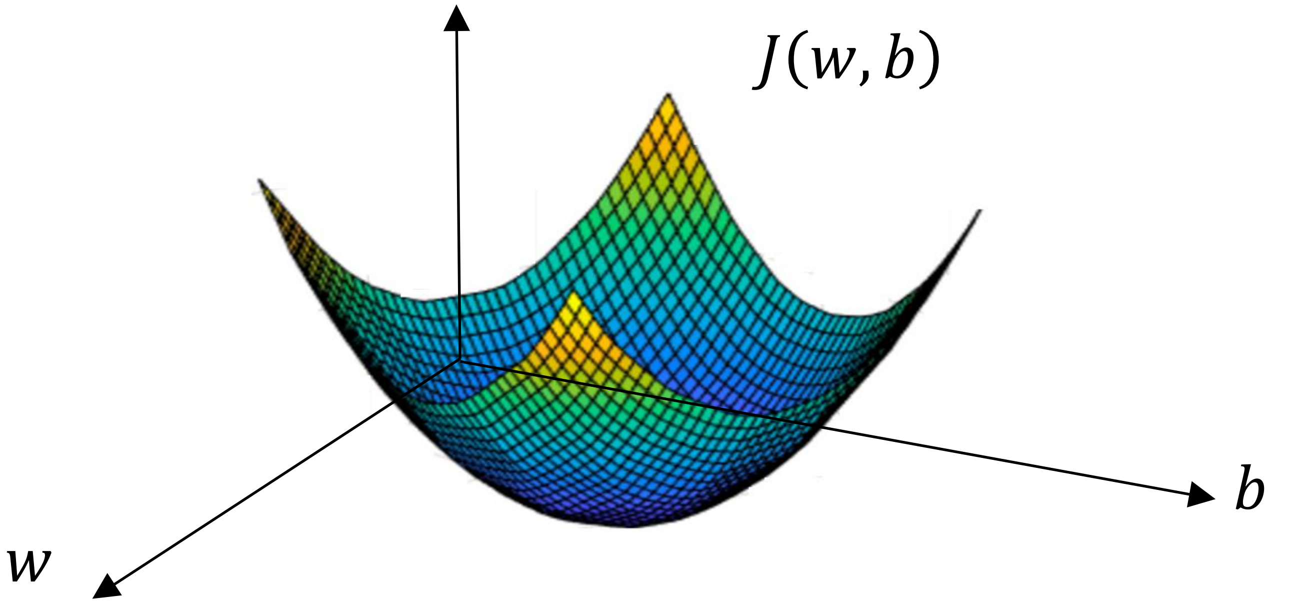
$$\mathcal{J}(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\hat{y}^{(i)} + \left(1-y^{(i)}\right)\log\left(1-\hat{y}^{(i)}\right)\right]$$

- Goal:
  - Find vectors **w** and **b** that minimize the cost function (total loss)
- Logistic regression can be viewed as a small neural network!

Logistic Regression

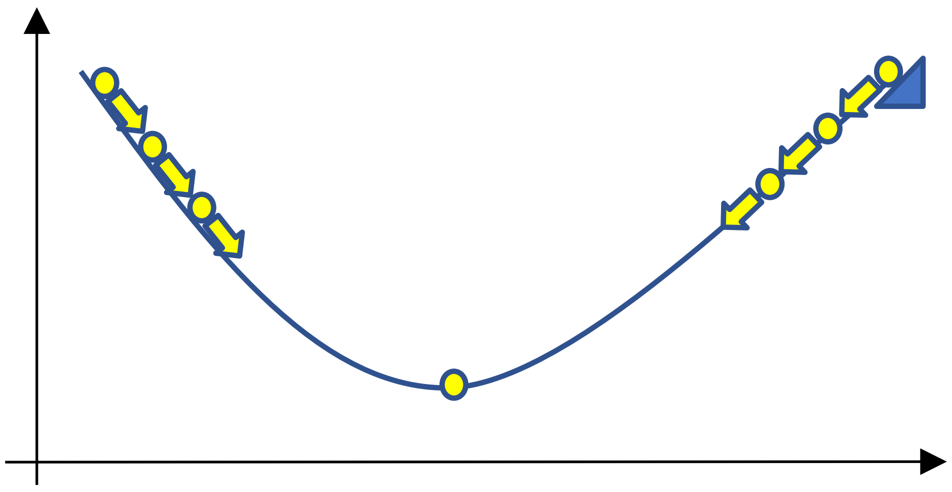
Convergence

- $\hat{y}^{(i)} = \sigma\left(z^{(i)}\right) = \sigma\left(w^T x^{(i)} + b\right)$, where $\sigma\left(z^{(i)}\right) = \frac{1}{1+e^{-z^{(i)}}}$
- $\mathcal{J}(w, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log \hat{y}^{(i)} + \left(1 - y^{(i)}\right) \log\left(1 - \hat{y}^{(i)}\right)\right]$
- Find w, b that minimize $\mathcal{J}(w, b)$



Logistic Regression

Convergence



Repeat {

$$w := w - \propto \frac{\delta J(w)}{\delta w}$$

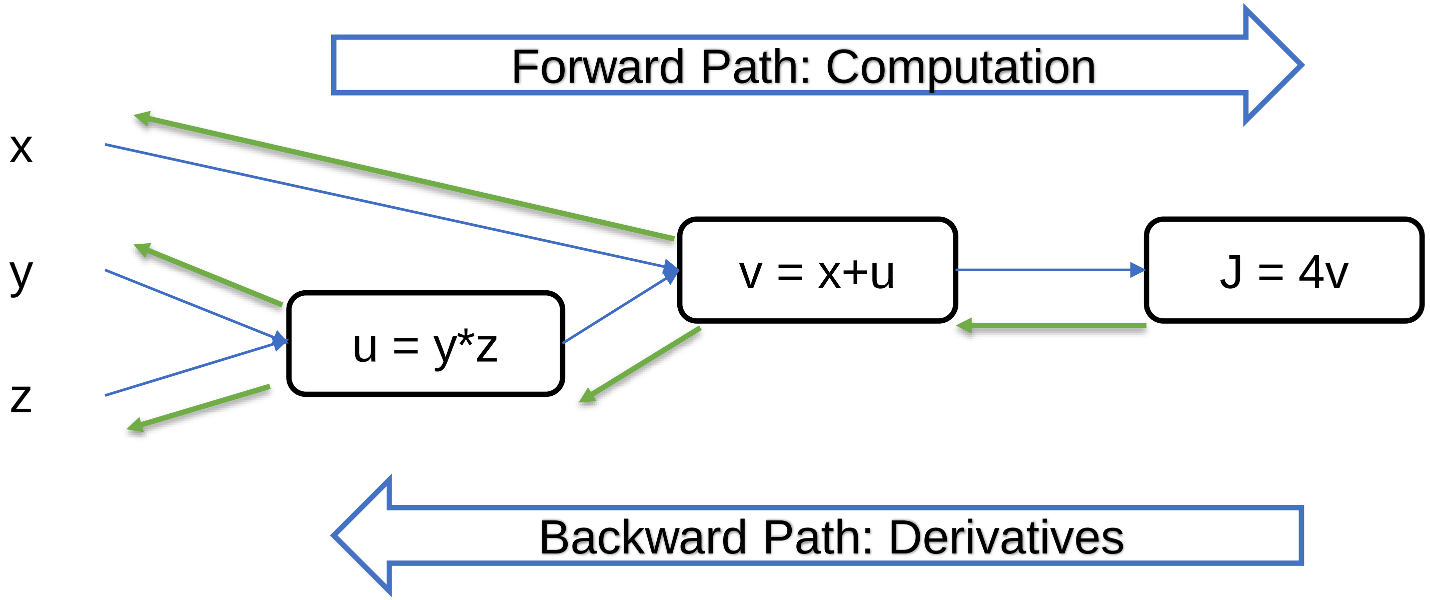
}

1. Shallow Neural Network

Logistic Regression

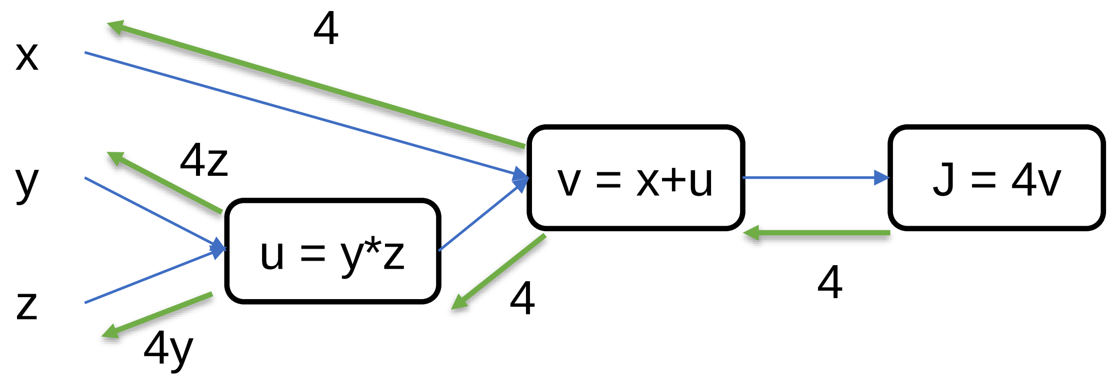
Computation Graph

- A graph that depicts all the computations required for a function in a forward path
- For example: J(x, y, z) = 4(x + yz)



Logistic Regression

Computation Graph



- $\dfrac{\delta J}{\delta v} = 4$
- $\dfrac{\delta J}{\delta x} = \dfrac{\delta J}{\delta v}\dfrac{\delta v}{\delta x} = 4 \times 1 = 4$
- $\dfrac{\delta J}{\delta u} = \dfrac{\delta J}{\delta v}\dfrac{\delta v}{\delta u} = 4 \times 1 = 4$
- $\dfrac{\delta J}{\delta y} = \dfrac{\delta J}{\delta u}\dfrac{\delta u}{\delta y} = 4 \times z = 4z$
- $\dfrac{\delta J}{\delta z} = \dfrac{\delta J}{\delta u}\dfrac{\delta u}{\delta z} = 4 \times y = 4y$

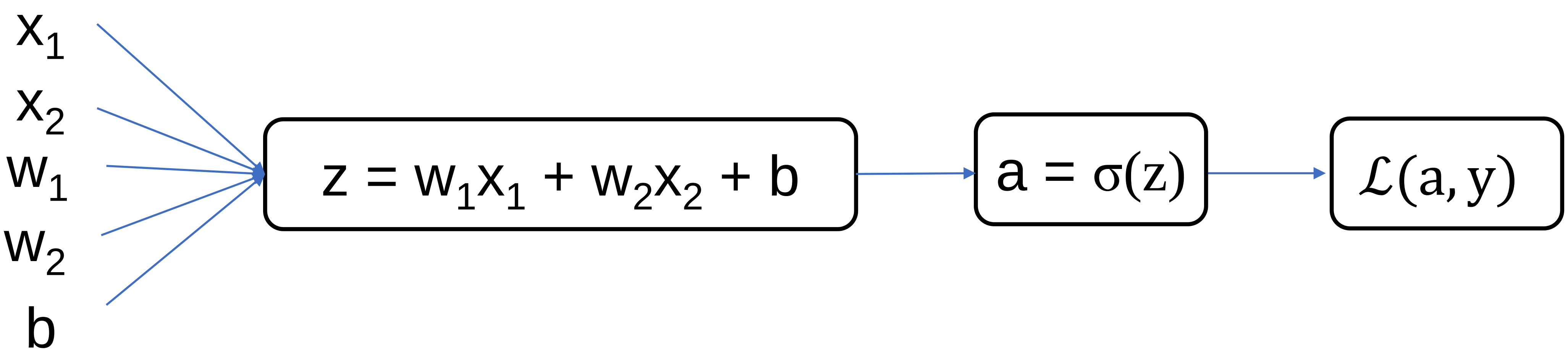
Logistic Regression

Computation Graph

$$z = w^T x + b$$

$$\hat{y} = a = \sigma(z)$$

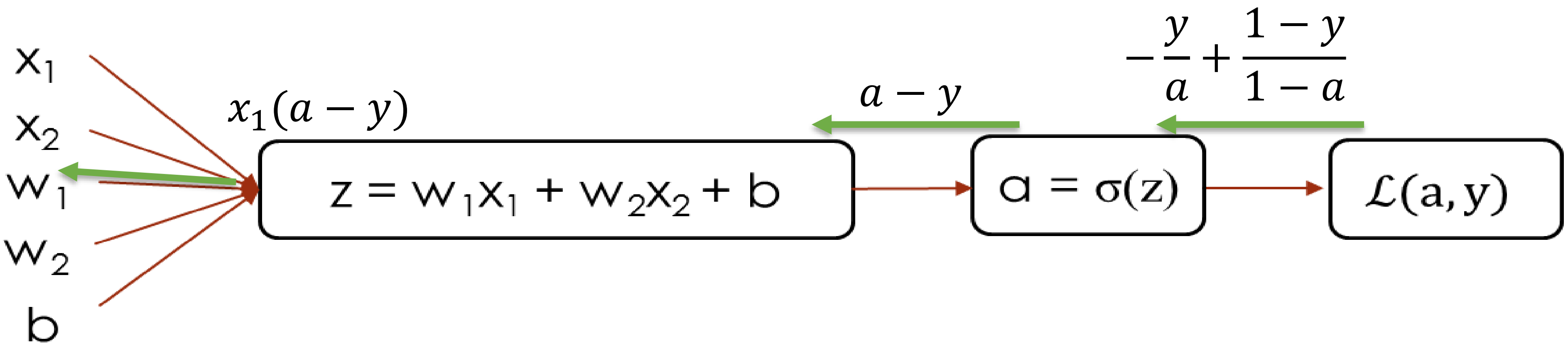
$$\mathcal{L}(a, y) = -(y \log(a) + (1 - y) \log(1 - a))$$



1. Shallow Neural Network

Logistic Regression

Computation Graph



- $\frac{\delta L(a,y)}{\delta a}=\left(-\frac{y}{a}+\frac{1-y}{1-a}\right)$
- $\frac{\delta L(a,y)}{\delta z}=\frac{\delta L(a,y)}{\delta a}\frac{\delta a}{\delta z}=\left(-\frac{y}{a}+\frac{1-y}{1-a}\right)\left(a(1-a)\right)=a-y$
- $\frac{\delta L(a,y)}{\delta w_1}=\frac{\delta L(a,y)}{\delta a}\frac{\delta a}{\delta z}\frac{\delta z}{\delta w_1}=\left(-\frac{y}{a}+\frac{1-y}{1-a}\right)a(1-a)x_1=x_1(a-y)=x_1\frac{\delta L(a,y)}{\delta z}$

1. Shallow Neural Network

Logistic Regression Code

$J = 0;\ \ dw_1 = 0;\ \ dw_2 = 0;\ \ db = 0;$

For $i = 1$ to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$

$dw_2 \mathrel{+}= x_1^{(i)} dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

$J \mathrel{/}= m;\ \ \ dw_1 \mathrel{/}= m;\ \ \ dw_2 \mathrel{/}= m;\ \ \ db \mathrel{/}= m;$

Update weights:

$w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

$b := b - \alpha\, db$

Logistic Regression

```
import numpy as np
a = np.array([1,2,3,4])
print(a)

import time
a = np.random.rand(1000000)
b = np.random.rand(1000000)
tic = time.time()
c = np.dot(a,b)
toc = time.time()

print(c)
print("Vectorized version:" + str(1000*(toc-tic) + "ms")
```

Code

```
c = 0
tic = time.time()
for i = range(1000000):
    c += a[i]*b[i]
toc = time.time()
print(c)
print("For loop:" + str(1000*(toc-tic) + "ms")
```

Logistic Regression

Vectorization

dw = np.zeros((nx, 1))

$J = 0$; $\cancel{dw_1 = 0;\ dw_2 = 0;}$ $db = 0$;

For i = 1 to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$\cancel{dw_1 \mathrel{+}= x_1^{(i)}}\, dz^{(i)}$

$\cancel{dw_2 \mathrel{+}= x_1^{(i)}}\, dz^{(i)}$ $dw \mathrel{+}= x^{(i)} dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

$J /= m$, $\cancel{dw_1 /= m,\ dw_2 /= m,}$ $db /= m$

dw /= m

One for loop is thus removed!!!

Logistic Regression

Vectorization

- $X = [x^{(1)}\ x^{(2)}\ \ldots\ x^{(m)}]$
- $Z = [z^{(1)}\ z^{(2)}\ \ldots\ z^{(m)}] = w^T X + [b\ b\ \ldots\ b] =$ np.dot(w.T, X) + b
- $A = [a^{(1)}\ a^{(2)}\ \ldots\ a^{(m)}] =$ sigmoid(Z)

1. Shallow Neural Network

Logistic Regression Vectorization

- $dz^{(1)} = a^{(1)} – y^{(1)}$, $dz^{(2)} = a^{(2)} – y^{(2)}$, … (all m examples)
- $dZ = [dz^{(1)}\ dz^{(2)}\ \dots\ dz^{(m)}]$
- $A = [a^{(1)}\ \dots\ a^{(m)}]$
- $Y = [y^{(1)}\ \dots\ y^{(m)}]$
- $dZ = A – Y = [a^{(1)}-y^{(1)}\ \dots\ a^{(m)}-y^{(m)}]$
- $db = \frac{1}{m}\sum_{i=1}^{m} dz^{(i)} = \frac{1}{m} np.sum(dZ)$
- $dw = \frac{1}{m} X dZ^T$

1. Shallow Neural Network

Logistic Regression

$J = 0$; $dw = np.zeros((nx,1))$; $db = 0$;

For $i = 1$ to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw \mathrel{+}= x^{(i)} * dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

$J \mathrel{/}= m$, $dw \mathrel{/}= m$, $db \mathrel{/}= m$

Vectorization

$Z = w^T X + b$ = np.dot(w.T, X) + b

$A = \text{sigmoid}(Z)$

$dZ = A - Y$

$dw = 1/m\ X\ dZ^T$

$db = 1/m$ np.sum(dZ)

$w := w - \alpha\, dw$

$b := b - \alpha\, db$

Vectorized Version: Without any for loop!