

Artificial Neuron

Ibrahim A. Hameed

ML IE500618

In this lecture

- Perceptron
- PLR
- Adaline
- Feature scaling
- examples

Perceptron

- Basic unit of artificial neuron.
- Can classify a task into two classes: 1 (positive class) and -1 (negative class)
- Decision function can be defined as a linear combination of certain input values x and a corresponding weight values w .

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \quad w = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} \quad \begin{aligned} net &= w_1 x_1 + \dots + w_n x_n + b = w^T x + b \\ f(net) &= \begin{cases} 1 & \text{if } net \geq 0 \\ -1 & \text{otherwise} \end{cases} \end{aligned}$$

Notes:

- Weights cause rotation of the decision line
- Bias cause translation of the decision line

The Perceptron learning Rule PLR

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

$$\Delta w_j = \eta (t^{(i)} - y^{(i)}) x_j^{(i)}$$

$$w_j = w_j + \Delta w_j$$

$$\Delta b = \eta (t^{(i)} - y^{(i)})$$

$$b = b + \Delta b$$

Implementation

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

class Perceptron(object):

    def __init__(self, no_of_inputs, epochs=100, learning_rate=0.01):
        self.epochs = epochs
        self.learning_rate = learning_rate
        self.weights = np.zeros(no_of_inputs + 1)
        self.training_error = []

    def predict(self, inputs):
        net = np.dot(inputs, self.weights[1:])+self.weights[0]
        # step activation function
        if net >= 0:
            activation = 1
        else:
            activation = -1
        return activation

    def train(self, training_inputs, training_labels):
        for _ in range(self.epochs):
            error = 0
            for inputs, label in zip(training_inputs, training_labels):
                prediction = self.predict(inputs)
                self.weights[1:] += self.learning_rate * (label - prediction) * inputs
                self.weights[0] += self.learning_rate * (label - prediction)
                error += label - prediction
            self.training_error.append(error.mean())
```

```
# test perceptron with IRIS data
```

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

```

```

# extract first 100 class labels (50 iris-setosa and 50 iris-versicolor)
x = df.iloc[0:100, [0, 2]].values
y = df.iloc[0:100, 4].values
y = np.where(y=='Iris-setosa',1, -1)
print(y)

```

```

# use a peceptron to find a decsion boundary to separate the two classe
p = Perceptron(no_of_inputs=2, epochs=10, learning_rate=0.1)
p.train(x, y)

```

```

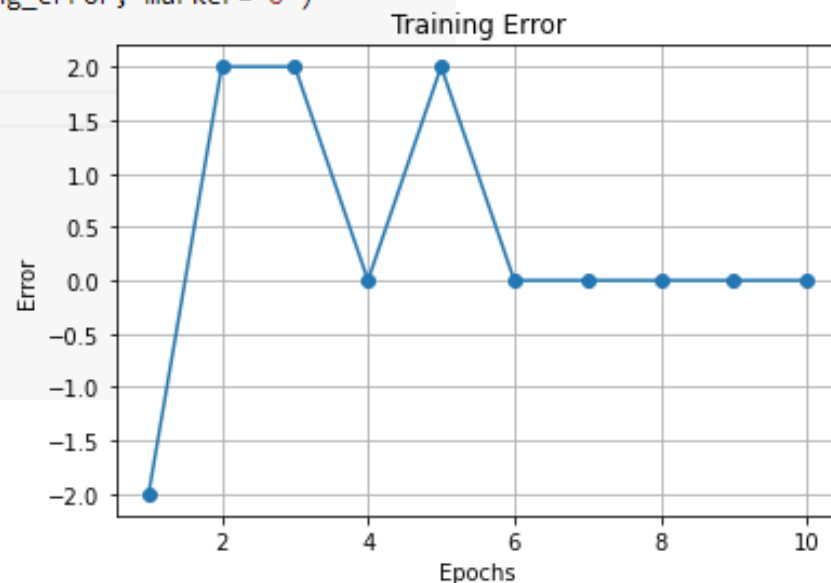
plt.plot(range(1,len(p.training_error)+1), p.training_error, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Error')
plt.title('Training Error')
plt.grid()
plt.show()

```

```

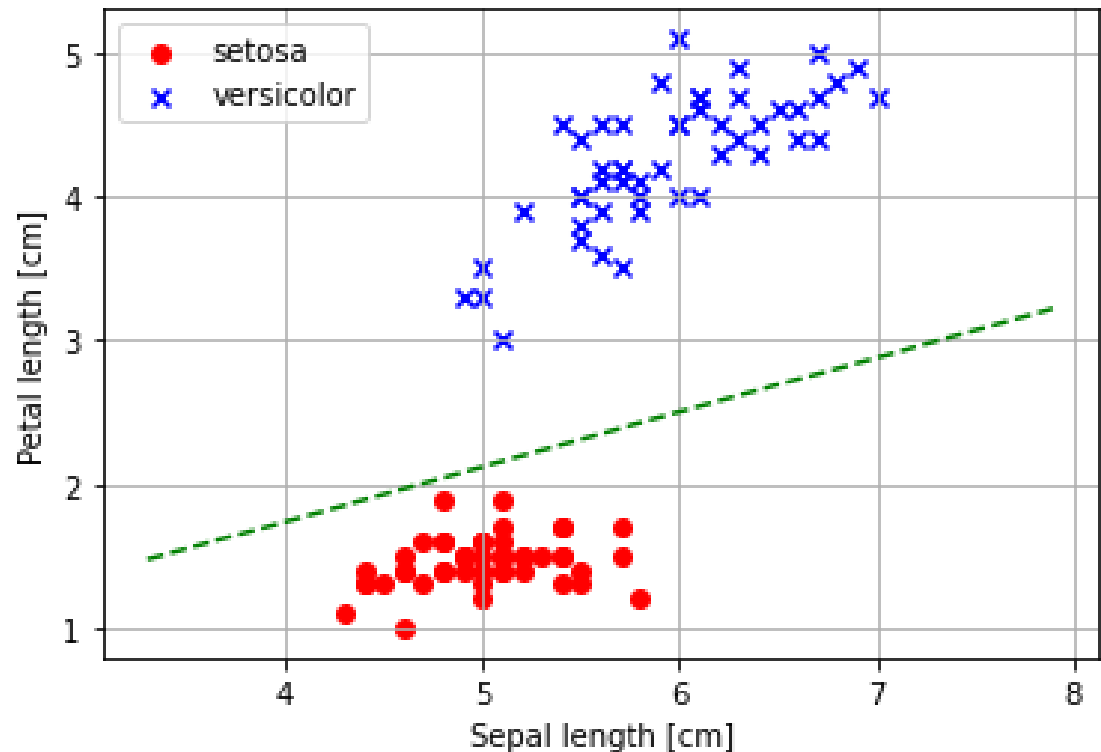
print(p.weights)
pred = []
for t in x:
    pred.append(p.predict(t))

```



```
# plot decision line
x1_min, x1_max = x[:,0].min()-1, x[:,0].max()+1
xx1 = np.arange(x1_min, x1_max, 0.1)
xx2 = -p.weights[1]/p.weights[2] * xx1 - p.weights[0]/p.weights[2]
```

```
# scatter plot
plt.scatter(x[0:50,0], x[0:50,1], color='red', marker='o', label='setosa')
plt.scatter(x[50:100,0], x[50:100,1], color='blue', marker='x', label='versicolor')
plt.plot(xx1, xx2, 'g--')
plt.xlabel('Sepal length [cm]')
plt.ylabel('Petal length [cm]')
plt.legend(loc='upper left')
plt.grid()
plt.show()
```



```
error = sum(y - pred)/len(y)
print(error)
```

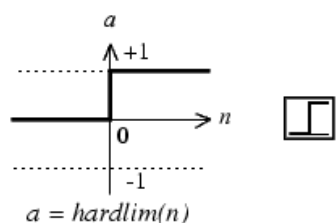
notes

- The perceptron learned a decision boundary that can classify iris training subset perfectly.
- The PLR converges iff (if and only if) the two classes can be separated perfectly by a linear hyperplane.
- If the classes cannot be separated perfectly by such a linear decision boundary, the weights will never stop updating unless we set a maximum number of epochs (i.e., iterations).

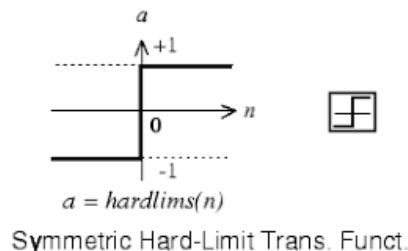
Adaline algorithm

- Stands for **adaptive linear** neuron.
- In PLR weights are updated based on a unit step function.
- Adaline introduces the concept of minimizing a continuous (convex) cost function – the sum of squared errors (SSE) between the calculated outcome and the true class label:.

$$J(w) = \frac{1}{2} \sum_i (t^{(i)} - y^{(i)})^2$$

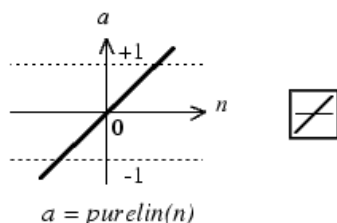
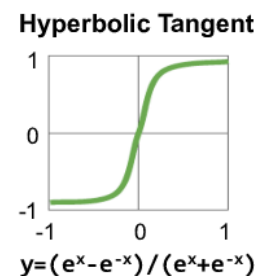
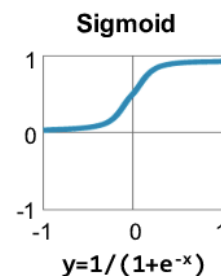


Hard-Limit Transfer Function



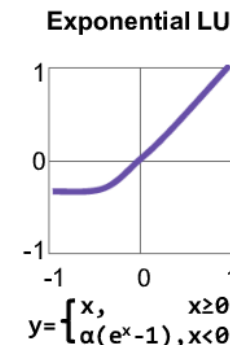
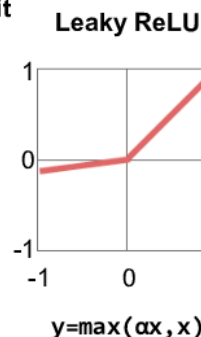
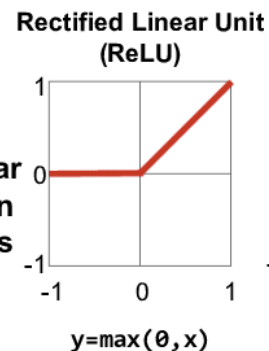
Symmetric Hard-Limit Trans. Funct.

Traditional Non-Linear Activation Functions



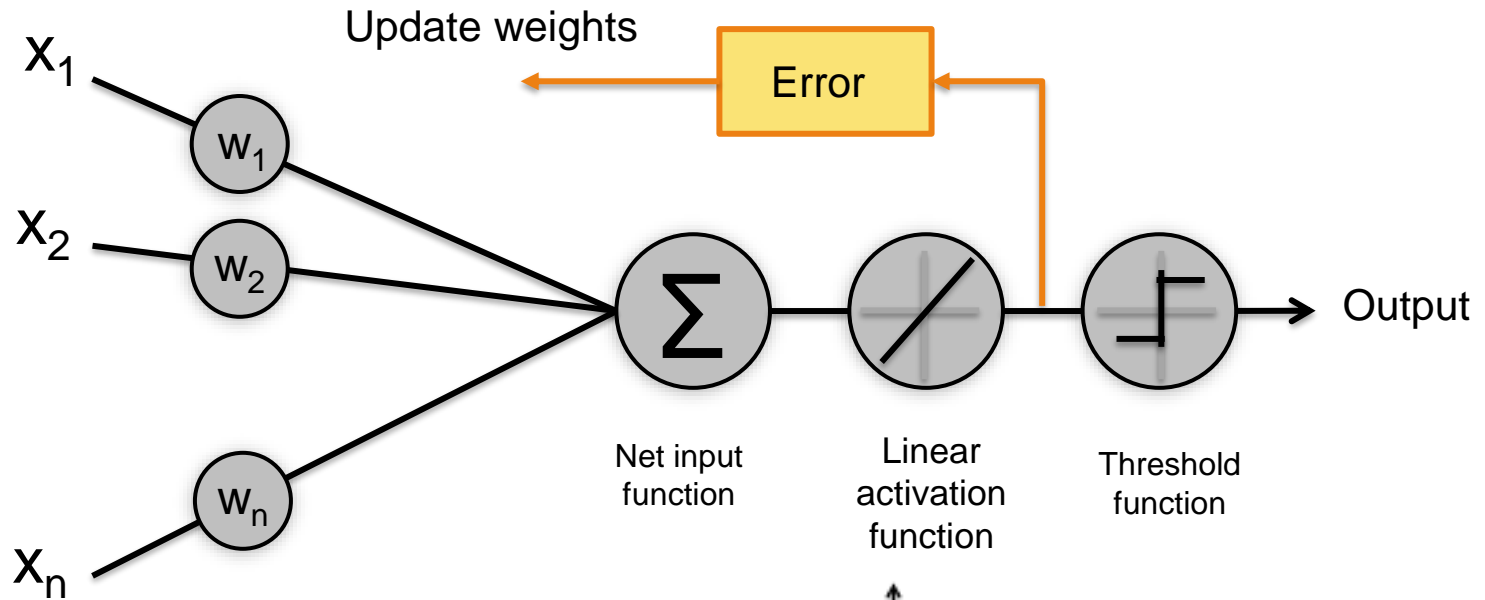
Linear Transfer Function

Modern Non-Linear Activation Functions



$\alpha = \text{small const. (e.g. 0.1)}$

Adaptive Linear Neuron



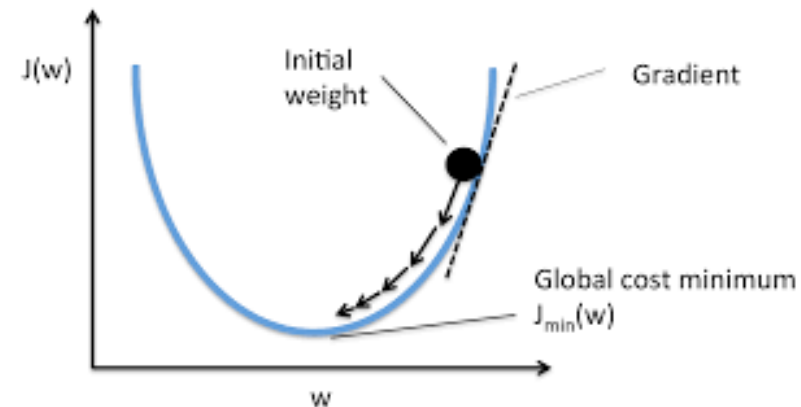
$$J(w) = \frac{1}{2} \sum_i (t^{(i)} - y^{(i)})^2$$

$$w = w + \Delta w$$

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

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = -\eta \frac{\partial J}{\partial y} \frac{\partial y}{\partial w_j}$$

$$= -\eta \left(-\sum_i (t^{(i)} - y^{(i)}) \right) x_j^{(i)} = \eta \left(\sum_i (t^{(i)} - y^{(i)}) \right) x_j^{(i)}$$



Notes: instead of updating the weights after each individual training sample, as in the perceptron, we calculate the gradient based on the whole training dataset (epoch).

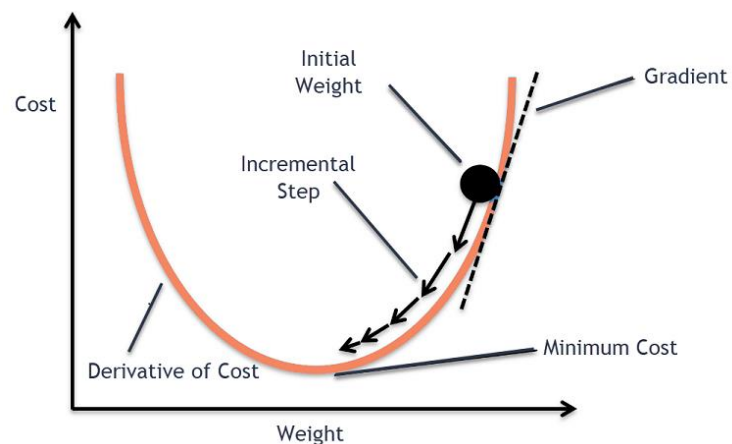
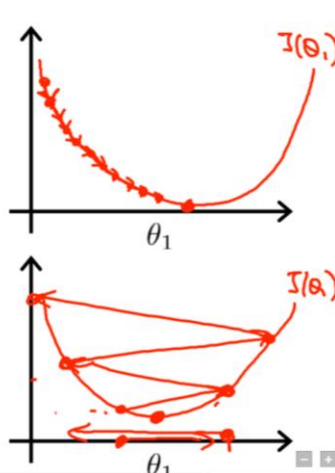
Training mode

- Batch gradient
- Mini Batch gradient
- Stochastic Gradient Descent (SGD) – online learning
- Large learning rate may overshoot the global minimum and fail to converge or even diverge

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

If α is too small, gradient descent can be slow.

If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.



Implementation

<https://github.com/ibribr/ML/blob/master/Adaline.ipynb>
<https://github.com/ibribr/ML/blob/master/Adaline.ipynb>

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from numpy import random
```

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

```
# extract first 100 class labels (50 iris-setosa and 50 iris-versicolor)
x = df.iloc[0:100, [0, 2]].values
y = df.iloc[0:100,4].values
y = np.where(y=='Iris-setosa',1, -1)
print(y)
print(len(x))
```

```
class adaline(object):
    def __init__(self, epochs=100, eta=0.1):
        self.epochs = epochs
        self.eta = eta

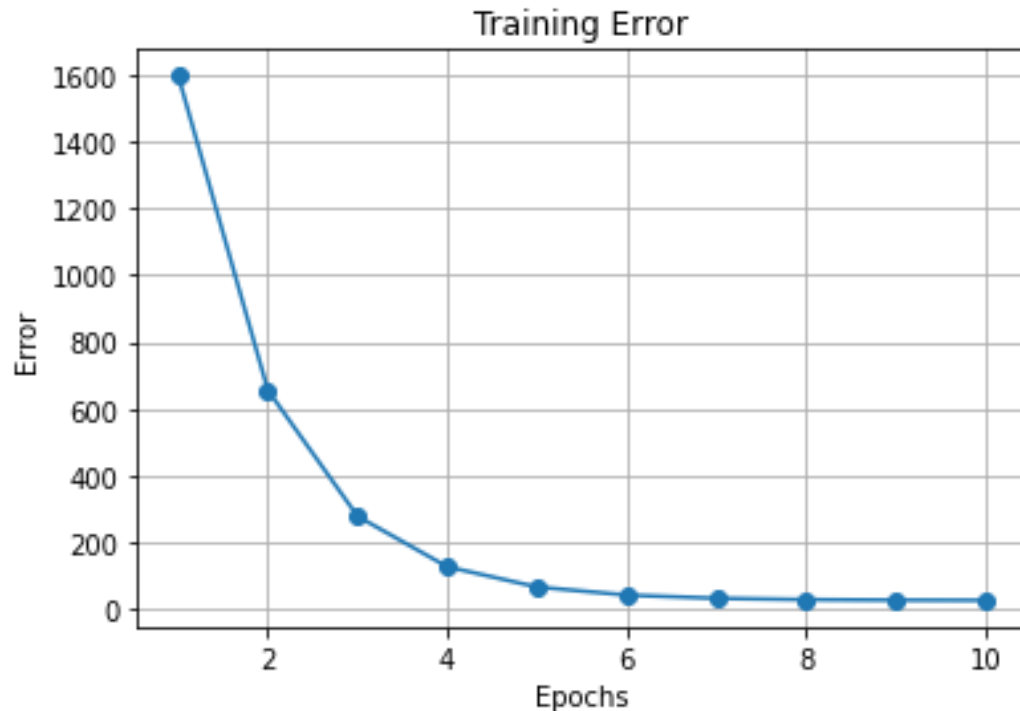
    def train(self, training_inputs, training_labels):
        x = training_inputs
        t = training_labels
        self.cost = [] # to plot cost function over epochs
        self.w = random.rand(training_inputs.ndim+1)
        for i in range(self.epochs):
            net = np.dot(x, self.w[1:])+self.w[0]
            y = net #linear activation function
            error = (t - y) # this is vector
            #update weights using sum of gradients
            self.w[1:] += self.eta * (np.dot(error, x)).mean()
            self.w[0] += self.eta * error.mean()
            cost = 0.5 * (error**2).sum()
            self.cost.append(cost)
        return self

    def predict(self, inputs):
        net=np.dot(inputs, self.w[1:])+self.w[0]
        return(np.where(net>=0, 1, -1))
```

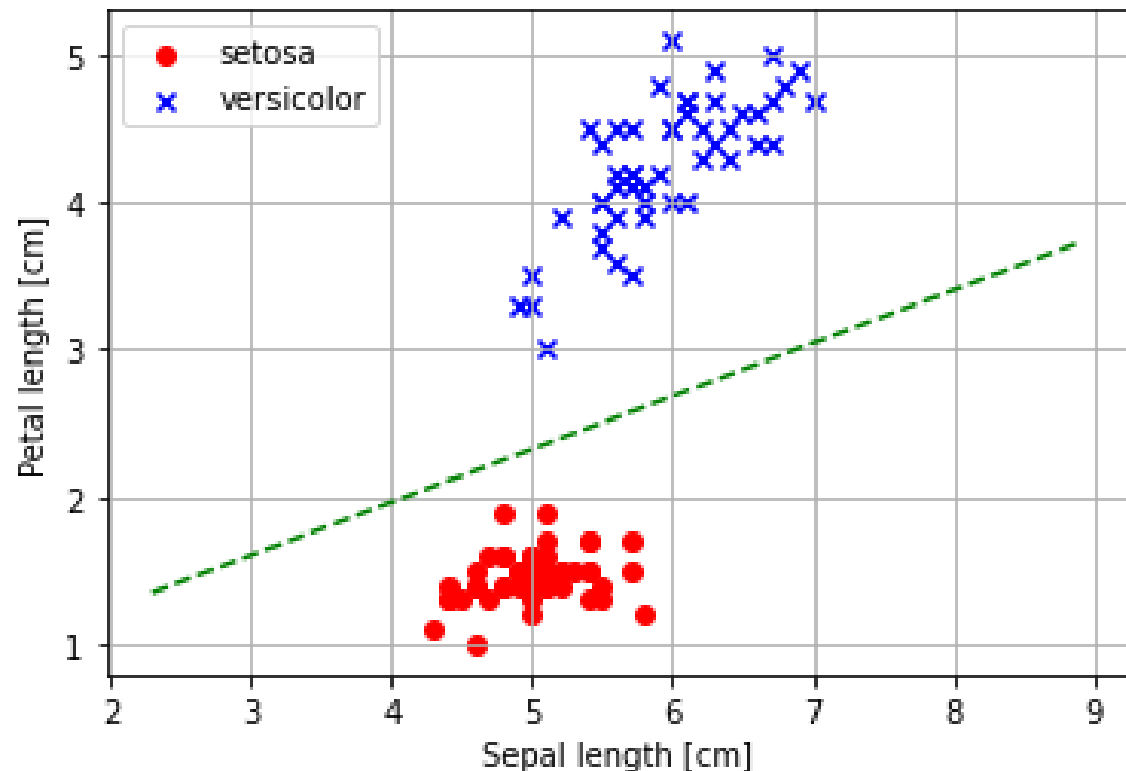
```
model=adaline(epochs=10, eta=0.0001)
model.train(x, y)
predictions = model.predict(x)

print(model.cost)
print(model.w)
print(predictions)

plt.plot(range(1,len(model.cost)+1), model.cost, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Error')
plt.title('Training Error')
plt.grid()
plt.show()
```

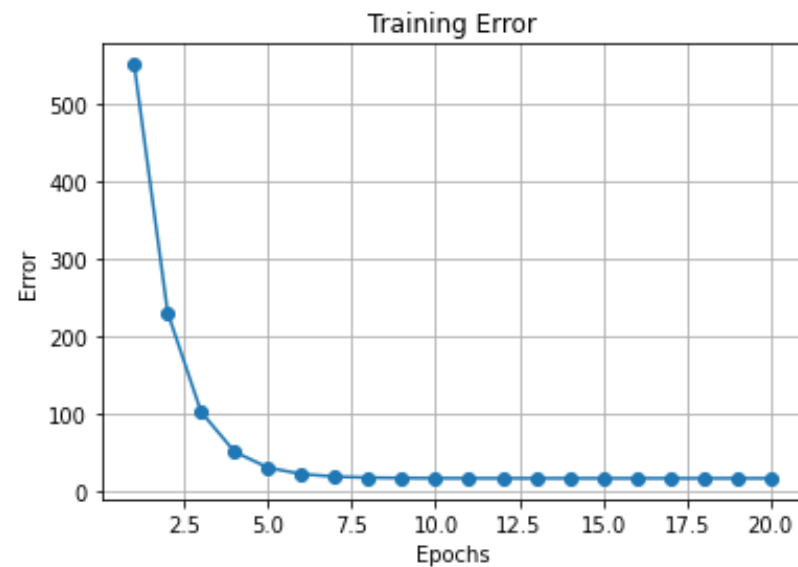
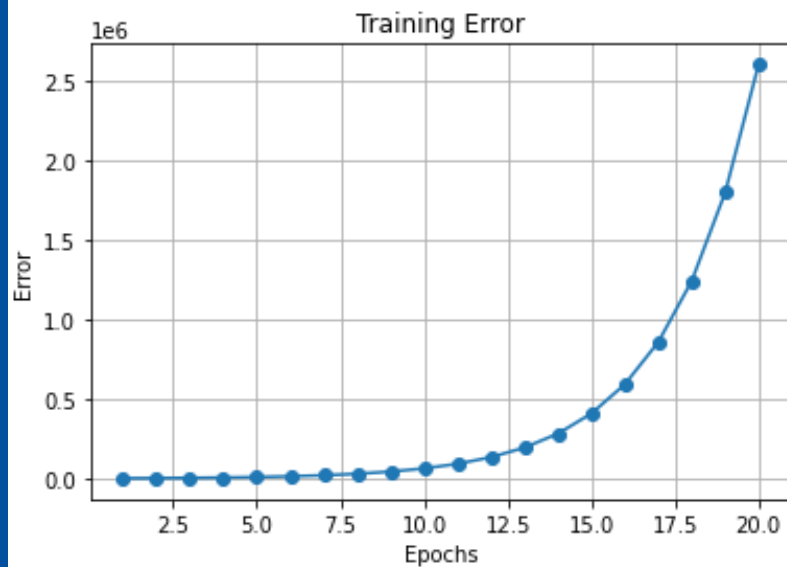


```
# scatter plot of predictions
plt.scatter(x[predictions==1,0], x[predictions==1,1], color='red', marker='o', label='setosa')
plt.scatter(x[predictions==-1,0], x[predictions==-1,1], color='blue', marker='x', label='versicolor')
# decision line
xx1 = np.arange(x[:,0].min()-2, x[:,0].max()+2, 0.1)
xx2 = -model.w[1]/model.w[2] * xx1 - model.w[0]/model.w[2]
plt.plot(xx1, xx2, 'g--')
plt.xlabel('Sepal length [cm]')
plt.ylabel('Petal length [cm]')
plt.legend(loc='upper left')
plt.grid()
plt.show()
```



Learning rate

- 0.0006 vs 0.0001



How to improve gradient descent through feature scaling?

- Standardization shifts the mean and variance of each feature so that it is centered at zero and has a feature standard deviation of 1 (i.e., standard normal distribution).

$$x_j = \frac{x_j - \mu_j}{\sigma_j}$$

```
# visualization distribution of the feature inputs
```

```
xn = (x-x.mean(axis=0))/x.std(axis=0)
```

```
print(x.mean(axis=0))
```

```
print(x.std(axis=0))
```

```
#print([x, xn])
```

```
fig, axs = plt.subplots(2, 2)
```

```
sns.distplot(x[:,0],ax=axs[0,0]);
```

```
axs[0,0].grid()
```

```
sns.distplot(xn[:,0],ax=axs[0,1]);
```

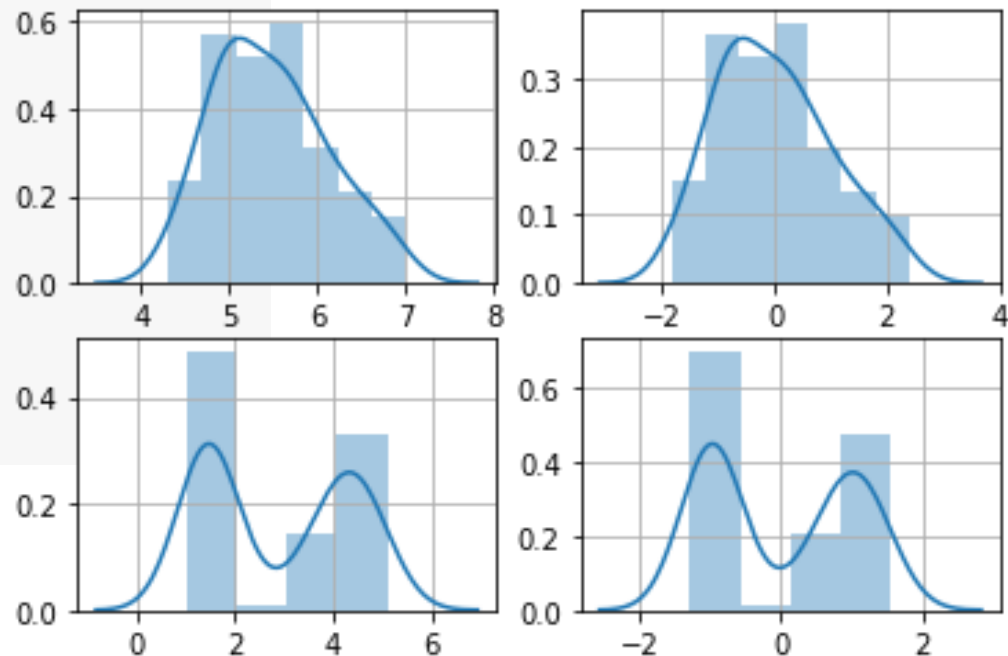
```
axs[0,1].grid()
```

```
sns.distplot(x[:,1],ax=axs[1,0]);
```

```
axs[1,0].grid()
```

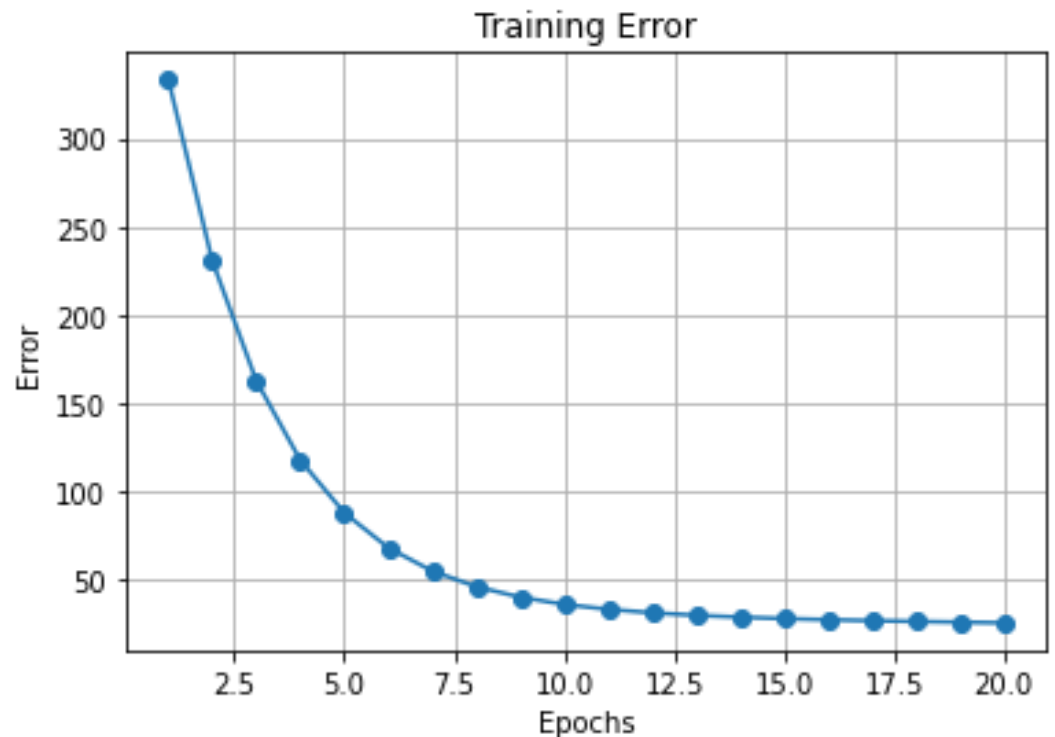
```
sns.distplot(xn[:,1],ax=axs[1,1]);
```

```
axs[1,1].grid()
```



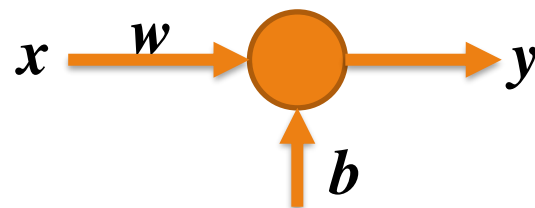
Train Adaline with scaled features

```
# test training with scaled features
model1=adaline(epochs=20, eta=0.01)
model1.train(xn, y)
plt.plot(range(1,len(model1.cost)+1), model1.cost, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Error')
plt.title('Training Error')
plt.grid()
plt.show()
```

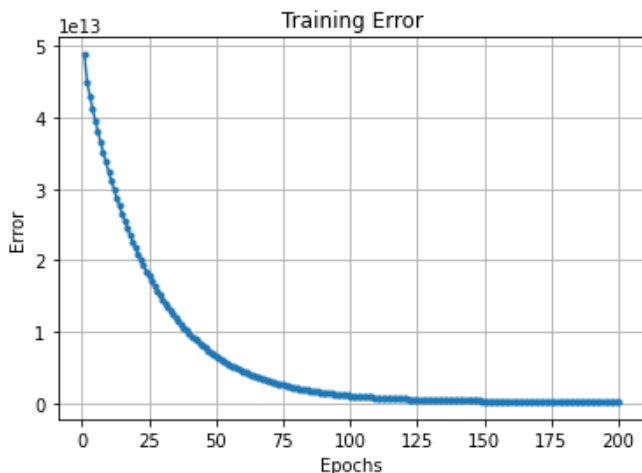


Use Adaline to predict annual salary in terms of number of years of experience

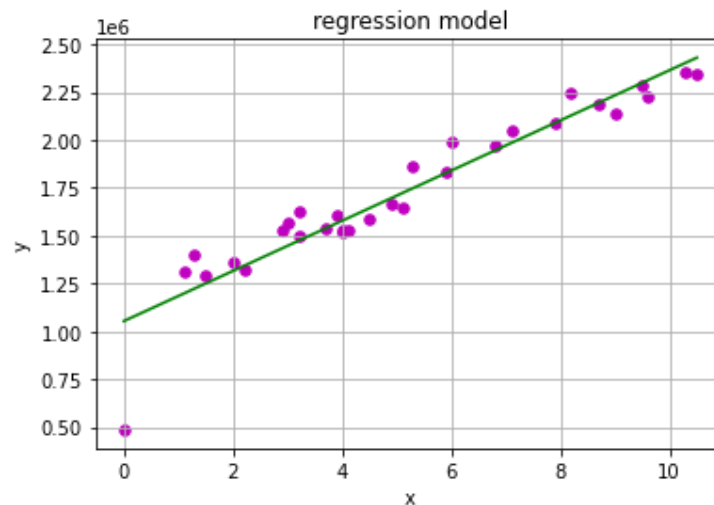
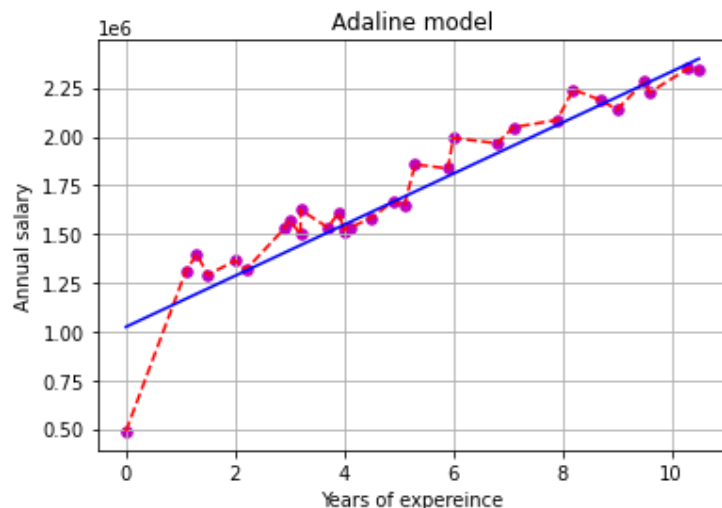
- Annual salary dataset



$$y = wx + b$$



	w	b
Adaline	1697073.26	380044.39
regression	131008.22	1053819.82

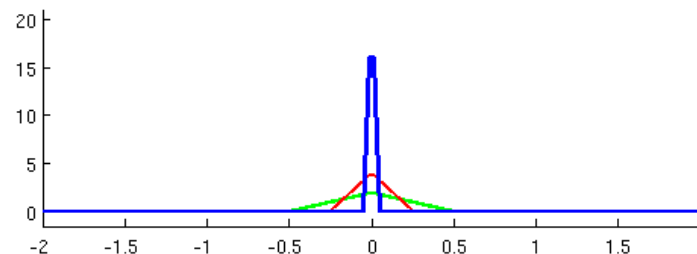
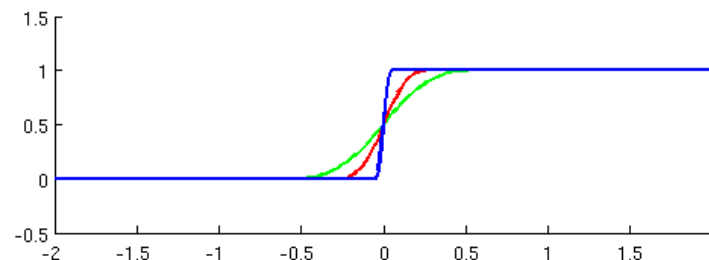


Activation functions

- The Activation Functions can be basically divided into 2 types:
 1. Linear Activation Function
 2. Non-linear Activation Functions

Unit step (hardlim) activation function

- Used for binary classification
- Derivative of step is delta (impulse) function



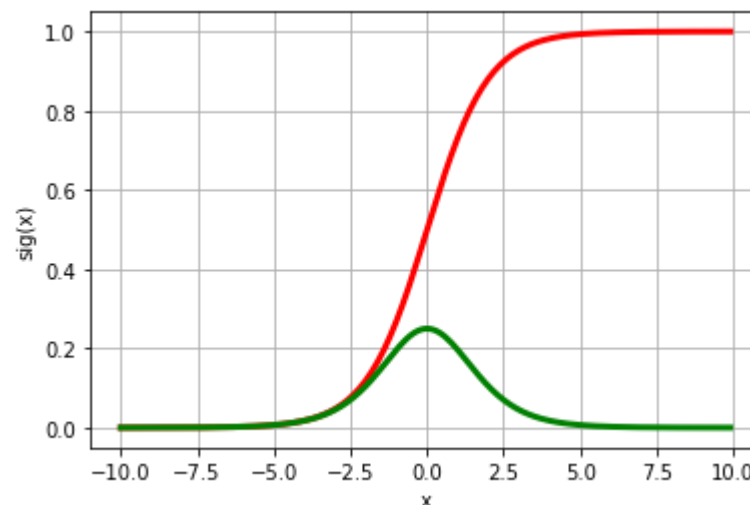
$$h(t) = \begin{cases} 1 & t \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\delta(t) = \frac{dh(t)}{dt} \quad \text{or} \quad \int_{-\infty}^{\infty} \delta(t) dt = h(t) \Big|_{-\infty}^{\infty} = h(\infty) - h(-\infty) = 1 - 0 = 1 \quad (\text{or } 2)$$

Sigmoid

- It is used a squashing function that limits the output to a range between 0 and 1.
- It predicts probability of a binary classification problem.

$$\begin{aligned} s(x) &= \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} \\ \frac{ds(x)}{dx} &= \frac{d}{dx} (1+e^{-x})^{-2} \\ &= -(1+e^{-x})^{-2} \times \frac{d}{dx} (1+e^{-x}) \\ &= \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1+e^{-x}-1}{(1+e^{-x})^2} \\ &= \frac{1}{1+e^{-x}} - \frac{1}{(1+e^{-x})^2} \\ &= s(x) - s^2(x) = s(x)(1-s(x)) \end{aligned}$$



Softmax

- Softmax is known as normalized exponential function
- Softmax function calculates the probabilities distribution of the event over 'n' different events (i.e., it calculate the probabilities of each target class over all possible target classes)
- The range will 0 to 1, and the sum of all the probabilities will be equal to one.
- If the softmax function used for multi-classification model it returns the probabilities of each class and the target class will have the high probability.
- Softmax is a multivariable function, generally. You wouldn't take a softmax of a single variable just like you wouldn't take a maximum of a single variable.

$$\text{soft}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^k e^{x_j}} \quad \forall i = 1, 2, \dots, k$$

```
#softmax: normalized exponential function
# it Calculate the softmax for the give inputs (array)
def softmax(x):
    return(np.exp(x)/np.exp(x).sum())

x = [2,3,5,6]
print(softmax(x))
print(softmax(x).sum())
```

```
[0.01275478 0.03467109 0.25618664 0.69638749]
1.0
```

Hyperbolic tangent function: $\tanh(x)$

- The range of the \tanh function is from (-1 to 1).
- \tanh is also sigmoidal (s - shaped).

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

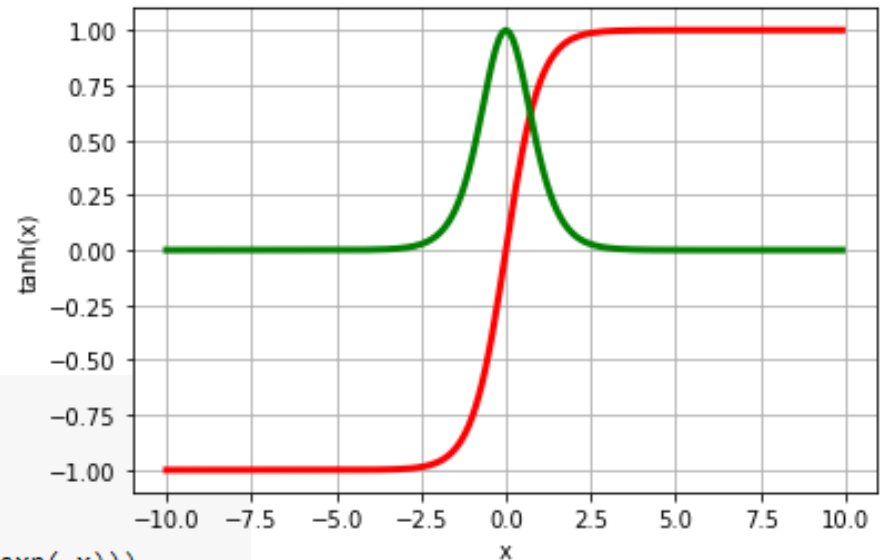
$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x)$$

```
# tanh(x)
x = np.arange(-10,10,0.1)

def tanh(x):
    return((np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x)))

def tanh_derivative(x):
    return(1-tanh(x)**2)

plt.plot(x,tanh(x),'r-', linewidth=3.0)
plt.plot(x,tanh_derivative(x),'g-', linewidth=3.0)
plt.xlabel('x')
plt.ylabel('tanh(x)')
plt.grid()
plt.show()
```



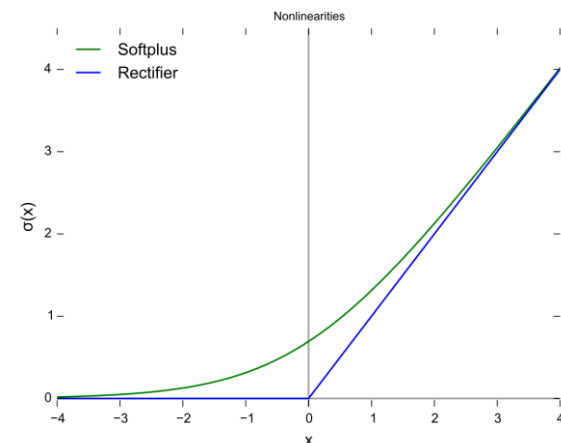
Rectified linear unit: ReLU

- The ReLU is the most used activation function in the world right now in almost all the convolutional neural networks or deep learning.
- Has a range from 0 to infinity
- ReLU activation function turns the negative input values into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately
- Softplus is a smooth approximation to the rectifier.

$$R(x) = \max(0, x)$$

$$sp(x) = \ln(1 + e^x) \text{ or } \frac{\ln(1 + e^{kx})}{k}$$

$$sp'(x) = \frac{1}{1 + e^{-kx}}$$



How to implement ReLU and Softplus activation functions?

```
# ReLU: Rectifier
x = np.arange(-10,10,0.1)

def relu_(x):
    return(np.maximum(x,0))

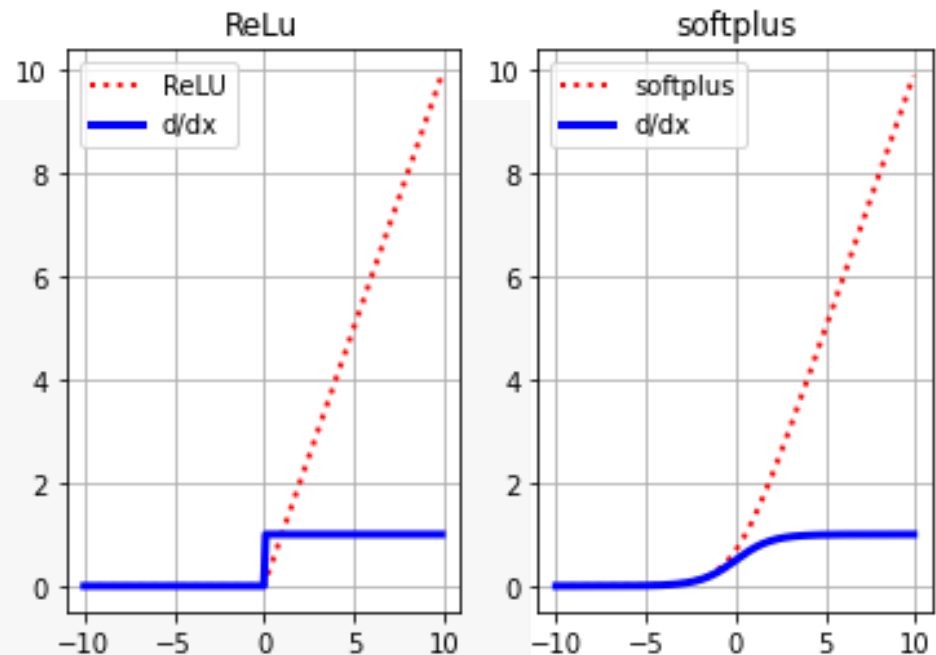
def softplus_(x):
    return(np.log((1+np.exp(x))))

def relu_derivative(x):
    return(np.where(x>=0,1,0))

def softplus_derivative(x):
    return(1/(1+np.exp(-x)))
```

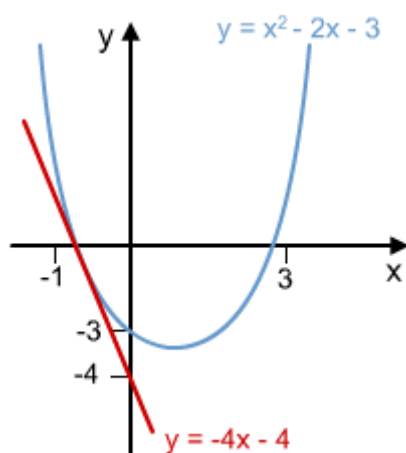
```
plt.subplot(1, 2, 1)
plt.plot(x, relu_(x), 'r:', linewidth=2.0, label='ReLU')
plt.plot(x, relu_derivative(x), 'b-', linewidth = 3.0, label='d/dx')
plt.legend(loc='upper left')
plt.grid()
plt.title('ReLu')

plt.subplot(1, 2, 2)
plt.plot(x, softplus_(x), 'r:', linewidth=2.0, label='softplus')
plt.plot(x, softplus_derivative(x), 'b-', linewidth = 3.0, label='d/dx')
plt.legend(loc='upper left')
plt.grid()
plt.show()
```



Why differentiation/derivative?

- The main use for differentiation is to find the **gradient** of a function at any point on the graph.
- When updating the curve, to know in which direction and how much to change or update the curve depending upon the slope.



$$y = x^2 - 2x - 3$$

$$\frac{dy}{dx} = 2x - 2$$

$$\text{For } x = -1 \rightarrow \frac{dy}{dx} = -2 - 2 = -4 \text{ (slope)}$$

-ve gradient means moving in the direction of increasing x

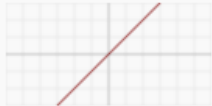

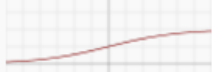






$$\rightarrow \text{and } y = (-1)^2 - 2(-1) - 3 = 0$$

Equation of the tangent:

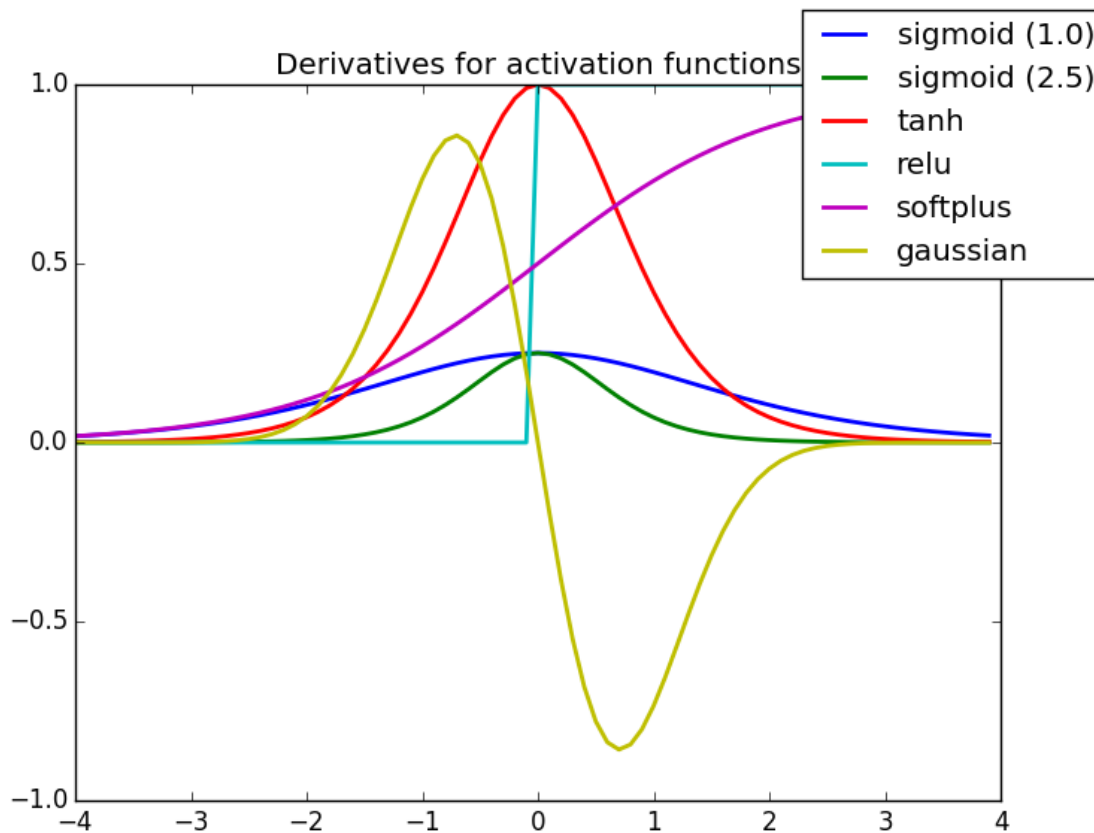
$$y - y_1 = m(x - x_1) \Rightarrow y = -4x - 4$$

$$y - 0 = -4(x - (-1))$$

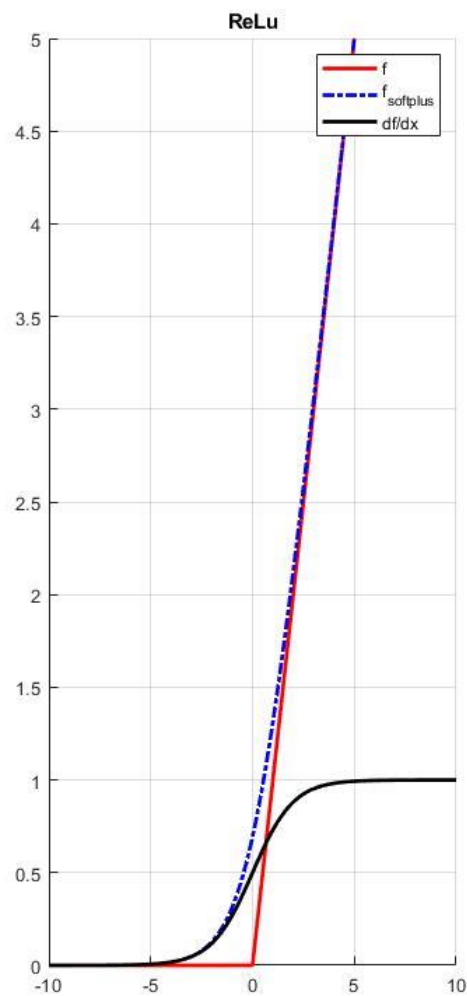
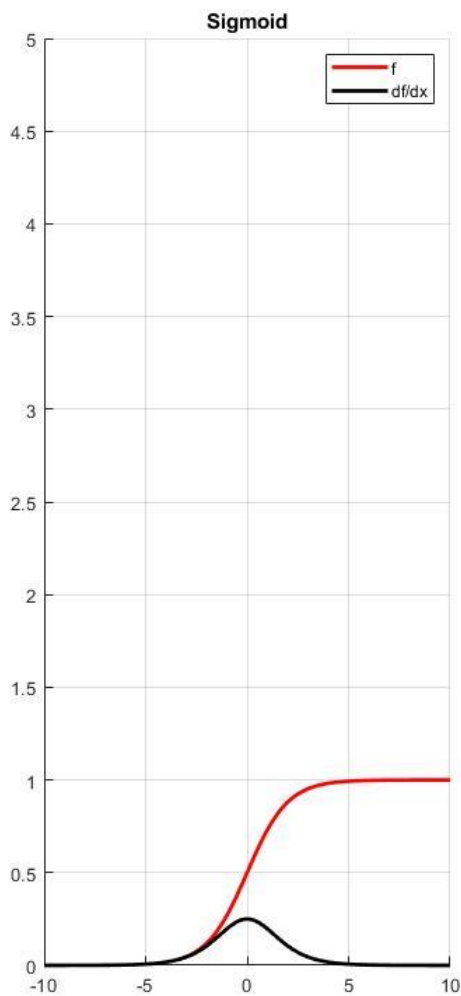
Derivatives & differentiations

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Derivatives of activation functions



ReLU & vanishing gradient problem



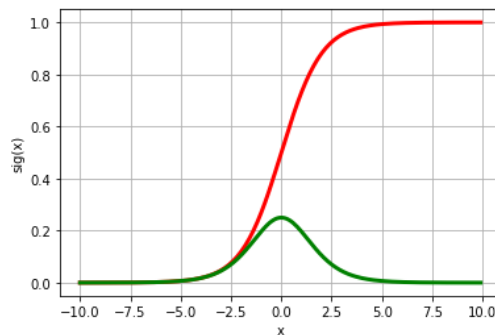
Vanishing gradient (VG) problem

- VG problem is encountered when training artificial neural networks with gradient-based learning methods and backpropagation.
- It describes the situation where a deep multilayer feedforward neural network or a recurrent neural network is unable to propagate useful gradient information from the output end of the model back to the layers near the input to the model and hence the inability of the model with many layers to learn on a given dataset or to prematurely converge to a poor solution.
- In gradient-based learning, each of the neural network's weights receive an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training.

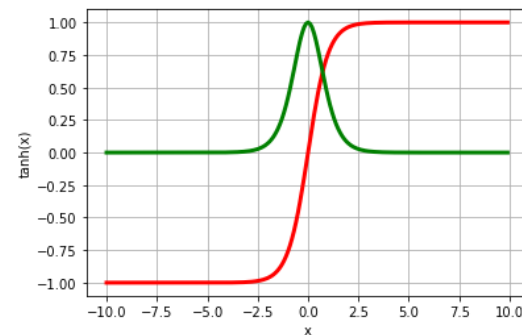
$$\Delta w = -\eta \frac{\partial E}{\partial w}$$

Vanishing gradient (VG) problem

- In some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training.
- As one example of the problem cause, traditional activation functions such as the hyperbolic tangent function have gradients in the range (0, 1), and backpropagation computes gradients by the chain rule. This has the effect of multiplying n of these small numbers to compute gradients of the "front" layers in an n-layer network, meaning that the gradient (error signal) decreases exponentially with n while the front layers train very slowly or not train at all.
- ReLu is used to improve the flow of gradients through the model.



Sigmoid



Hyperbolic tangent

Training/learning modes

- **Batch gradient descent:**
 - Minimize the cost function by taking a step in the opposite direction of the cost gradient from the whole training set.
 - Very computationally expensive specially for large datasets with millions of data points where we need to reevaluate the whole training dataset each time we take one step towards the global minimum.
- **Stochastic gradient descent – iterative or online gradient descent:**
 - Instead of updating the weights based on the sum of the accumulated errors over all samples, we updated the weights incrementally for each training sample.
 - It reaches convergence much faster because of the more frequent updates.
 - Gradient is based on a single training example and therefore the error surface is noisier and can escape shallow local minima.
 - To obtain satisfying results via SGD, it is important to shuffle the training set for every epoch to present cycles.
 - In SGD, learning rate η is often replaced by an adaptive learning rate that decreases over time (c_1 and c_2 are constants).
 - Can be used for online learning where our model is trained on the fly as new training data arrives. In online learning, the model can immediately adapt to changes and the training data can be discarded after updating the model if storage space is an issue.

BGD

$$\Delta w = \eta \sum_i (t^{(i)} - y^{(i)}) x^{(i)}$$

SGD

$$\Delta w = \eta (t^{(i)} - y^{(i)}) x^{(i)} \quad \eta = \frac{c_1}{[\text{number of iterations}] + c_2}$$

Training/learning modes

- **Mini-batch learning:**
 - Is 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.

Home assignment

- **Go to:**
https://github.com/ibribr/ML/blob/master/Adaline_v2.ipynb
 - Batch gradient is applied
 - Repeat it using SGD and mini-batch gradient
 - Compare results in terms of number of epochs, time, memory use, etc.