

Neurona lineal

By: Jared Isaías Monje Flores - 217594478



14 de febrero de 2024

Centro universitario de ciencias exactas e ingenierias

UdeG

***Neurona Lineal***

Una neurona lineal es un tipo de neurona artificial en la que la salida es una función lineal de la suma ponderada de las entradas. En el contexto del aprendizaje automático, las neuronas lineales se utilizan a menudo como capas de entrada o capas de entrada o capas de salida en redes neuronales. También se pueden usar como capas ocultas, aunque en general, las capas no lineales son más efectivas para capturar relaciones complejas entre las entradas y salidas.

Es importante tener en cuenta que la neurona lineal es una de las funciones de activación más simples, y si salida no está limitada a un rango específico, lo que puede dar lugar a problemas de saturación si los procesos son demasiado grandes.

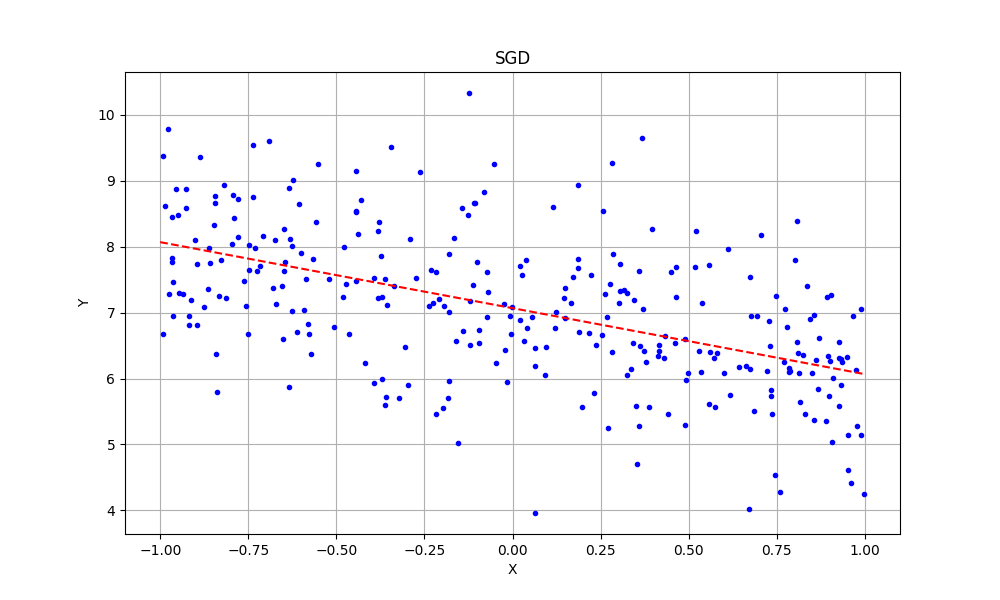
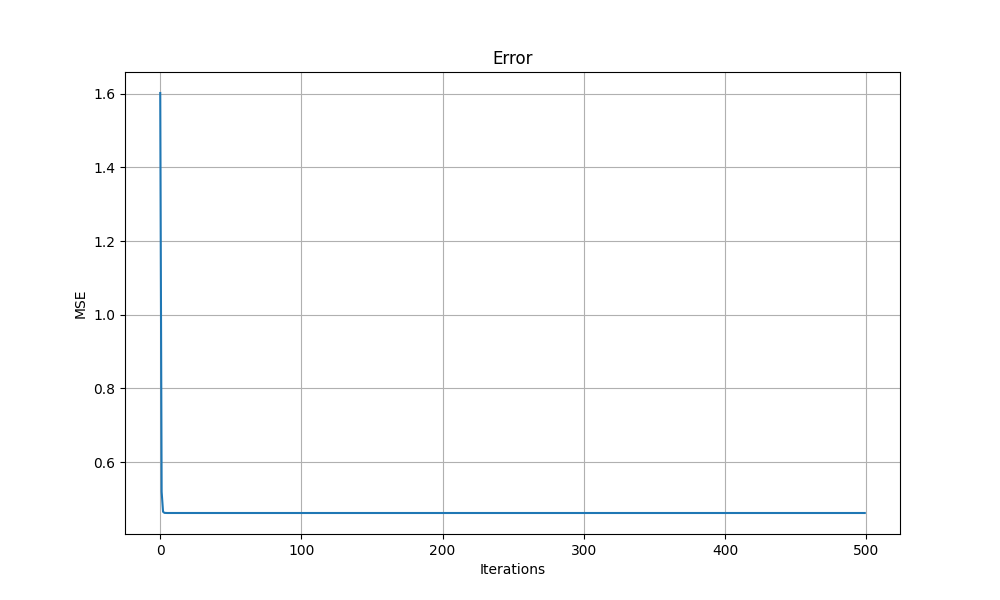
**Función objetivo**

f (x)=−x+7∗N(0 ,1)

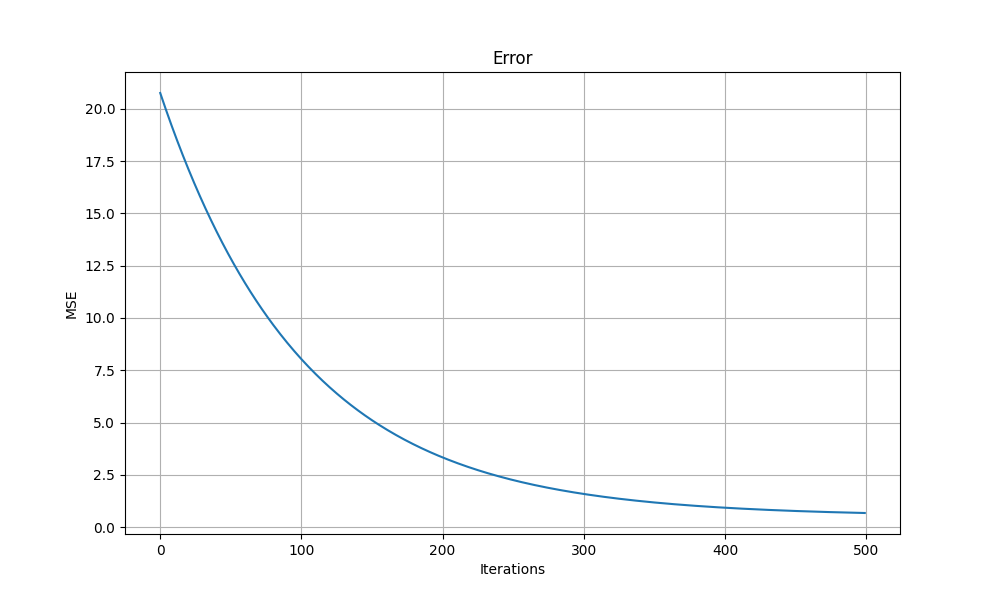
η=0.005

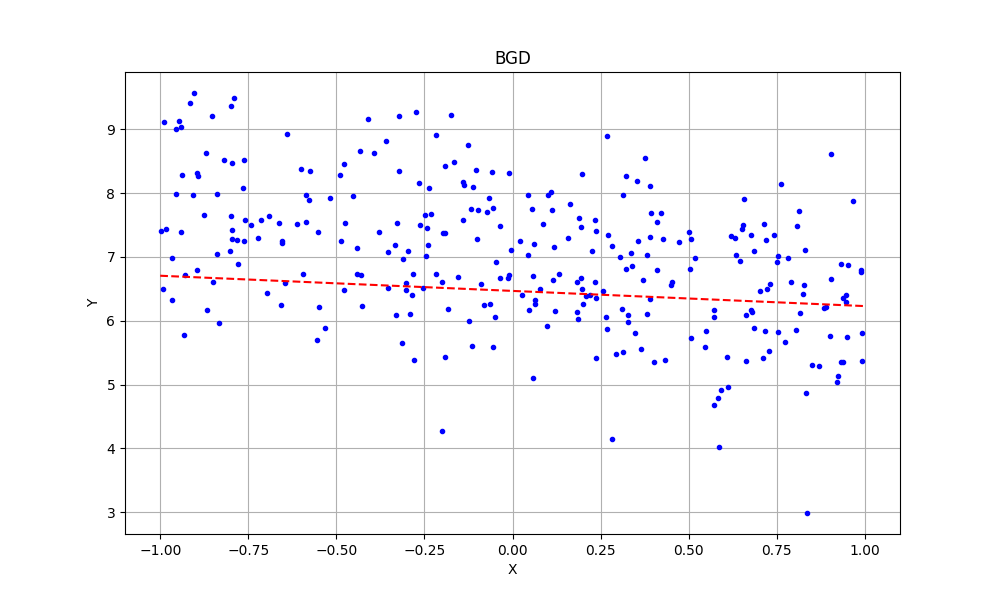
Epoch=500

**Stochastic Gradient Descent**

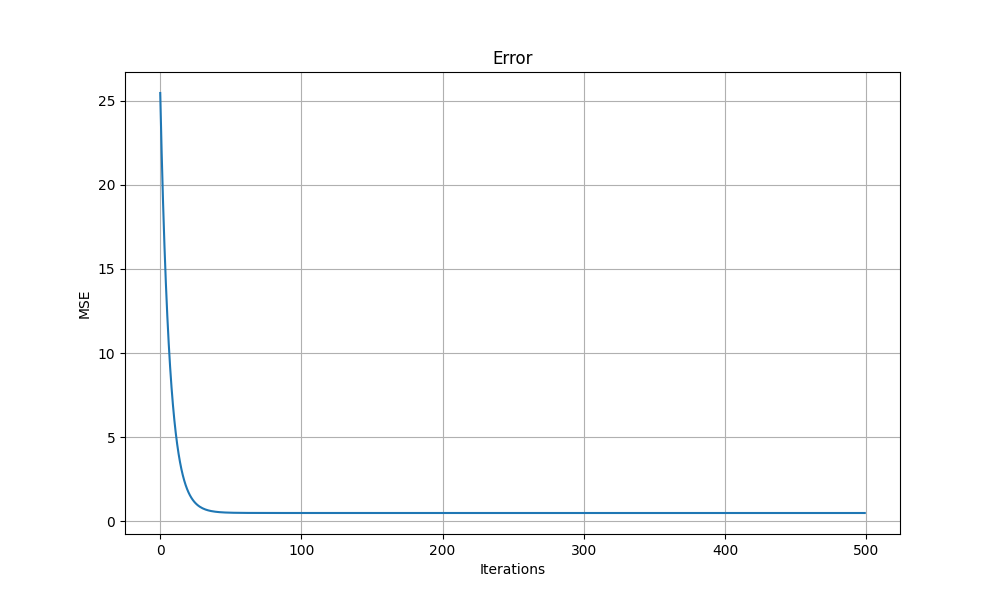


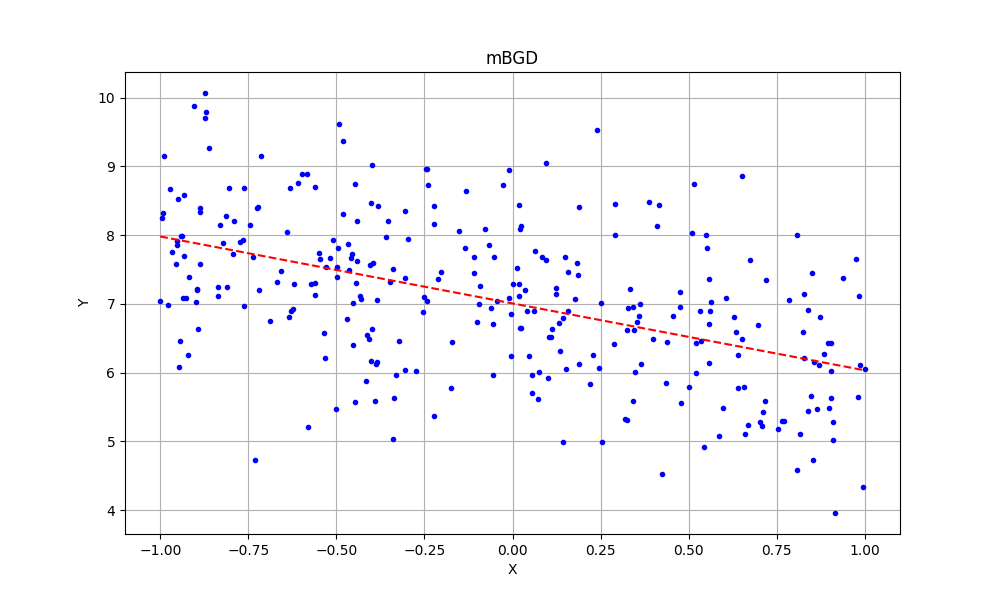
**Batch Gradient Descent**

****

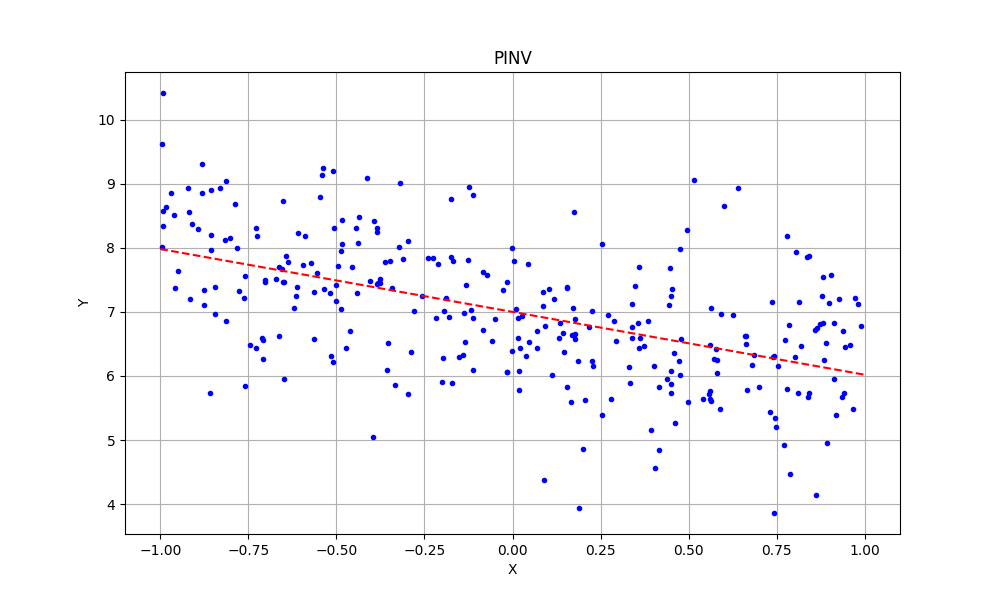
****

**Mini Batch Gradient Descent**

****

****

**Pseudo Inverse**

****

**Conclusión**

Podemos denotar, en las diferentes gráficas, como cada formato reduce su error cuadrático medio respecto a las iteraciones, con estos datos podemos concluir que no solo importa el algoritmo, sino también los parámetros de ajuste y las épocas que iteramos, siendo una correlación dependiente, con ellos veremos nuestros resultados mejorar u oscilar de ser mal ajustados dichos parámetros. Como podemos observar, la relación de tasa de aprendizaje y el algoritmo, es importante para objetivar la cantidad de épocas necesarias para una solución aceptable, ya que, como vimos en las gráficas previas, podemos tener cientos de iteraciones sin mejorar en nuestro modelo.

**Código**

*#*

*# Neurona Lineal*

*# By: Dexne*

*#*

*# para obtener mejores resultados ajusta las epocas y el learning rate*

*#*

*# Importamos librerias para graficar y realizar calculos matematicos*

import matplotlib.pyplot as plt

import numpy as np

*# Linear neuron*

class LinearNeuron:

*# Constructor*

    def \_\_init\_\_(*self*, *n\_inputs*, *learning\_rate*=0.1) -> None:

*# Inicializamos las variables*

*self*.w = -1+2\*np.random.rand( *n\_inputs* )

*self*.b = -1+2\*np.random.rand()

*self*.eta = *learning\_rate*

*# funcion prediccion*

    def predict(*self*, *X*):

        Y\_est = np.dot(*self*.w, *X*) + *self*.b

        return Y\_est

*# funcion generadora para el batch*

    def batcher(*self*, *X*, *Y*, *batch\_size*):

        p = *X*.shape[1]

        li, lu = 0, *batch\_size*

        while True:

            if li < p:

                yield *X*[:, li:lu], *Y*[:,li:lu] *# Limites*

                li, lu = li + *batch\_size*, lu + *batch\_size*

            else:

                return None

*# MSE ( Error Cuadratico Medio )*

    def MSE( *self*, *X*, *Y* ):

        p = *X*.shape[1]

        Y\_est = *self*.predict( *X* )

        return (1/(2\*p))\*np.sum((*Y*-Y\_est)\*\*2)

*# Funcion de entrenamiento*

    def fit( *self*, *X*, *Y*, *solver*='BGD', *epochs*=500, *batch\_size*=20):

        error\_history = [] *# almacenamos para poder calcular el error*

        p = *X*.shape[1]

*# SGC ( Gradiente Descendiente Estocastico )*

        if *solver* == 'SGD':

            for \_ in range( *epochs* ):

                for i in range( p ):

                    y\_est = *self*.predict( *X*[:,i] )

*self*.w += *self*.eta \* ( *Y*[:,i] - y\_est ) \* *X*[:,i]

*self*.b += *self*.eta \* ( *Y*[:,i] - y\_est )

*# Al terminar calculamos el error*

                error\_history.append( *self*.MSE(*X*,*Y*) )

*# BGD ( Gradiente Descendente por Batch )*

        elif *solver* == 'BGD':

            for \_ in range( *epochs* ):

                Y\_est = *self*.predict( *X* )

*# @ = broadcasting*

*self*.w += ( *self*.eta/p ) \* ((*Y* - Y\_est) @ *X*.T).ravel()

*self*.b += ( *self*.eta/p ) \* np.sum(*Y* - Y\_est )

                error\_history.append( *self*.MSE(*X*,*Y*) )

*# Mini Batcher Gradiente Descendente ( mBGD )*

        elif *solver* == 'mBGD':

            for \_ in range( *epochs* ):

                mini\_batch = *self*.batcher(*X*,*Y*, *batch\_size*)

                for mX, mY in mini\_batch:

                    p = mX.shape[1]

                    Y\_est = *self*.predict( mX )

*# @ broadcasting*

*self*.w += ( *self*.eta/p ) \* ((mY - Y\_est) @ mX.T).ravel()

*self*.b += ( *self*.eta/p ) \* np.sum(mY - Y\_est)

                error\_history.append( *self*.MSE(*X*,*Y*) )

*# Pseudo-inverso ( Metodo Directo )*

        elif *solver* == 'PINV':

            X\_hat = np.concatenate((np.ones((1,p)), *X*), *axis*=0)

            w\_hat = np.dot(*Y*, np.linalg.pinv(X\_hat))

*self*.w = w\_hat[0,1:]

*self*.b = w\_hat[0,0]

*# Independientemente de cual metodo elegimos retornamos el error\_history*

        return error\_history

if \_\_name\_\_ == '\_\_main\_\_':

    p = 300

    x = -1 + 2\*np.random.rand( p ).reshape(1,-1)

    y = ( -x + 19 -12 ) + np.random.randn(1,p)

    solver = 'SGD'

    neuron = LinearNeuron( 1, 0.005 )

    print( neuron.w )

    error = neuron.fit( x,y, *solver*=solver )

    print( neuron.w )

    plt.figure( *figsize*=( 10,6 ))

    plt.title(solver)

    plt.xlabel('X')

    plt.ylabel('Y')

    plt.grid( True )

    plt.plot(x,y, '.b')

    xn = np.array([[-1,1]])

    plt.plot(xn.ravel(), neuron.predict(xn), '--r')

*# MSE ( Error Cuadratico Medio )*

    plt.figure( *figsize*=(10,6) )

    plt.title("Error")

    plt.ylabel("MSE")

    plt.xlabel("Iterations")

    plt.grid( True )

    plt.plot( error )

    plt.show()