Descenso de gradiente

Tinoco Sergio.
Instituto Politécnico Nacional, Escuela Superior de Cómputo.
Redes neuronales y aprendizaje profundo

09 de marzo de 2023

En el presente documento se va a presentar el desarrollo del descenso por gradiente implementado en Google Colab así como los resultados obtenidos.

Gradiente: el gradiente es un vector que indica la dirección de la pendiente correspondiente a la función que describe el error residual de la red neuronal el cual depende de los pesos asignados a dicha red neuronal (figura 1).

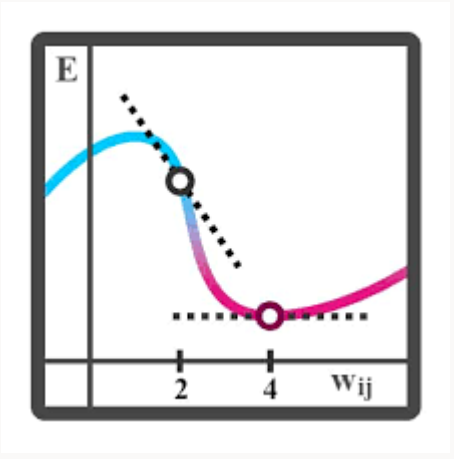


Figura 1.

Primeramente se definen las funciones con las cuales se va a generar el algoritmo de la red neuronal.

```python
# función de activación
def sigmoid(x):
    return 1/(1+np.exp(-x))

# Derivada de f
def sigmoid_prime(x):
    return sigmoid(x) * (1 - sigmoid(x))

# función h lineal
def function_h(X, W, b):
    return np.dot(W, X) + b

# Salida de la RN
def output_y(X,W,b):
    return sigmoid(function_h(X,W,b))
```

Posteriormente se calcula el término de error aplicando la ecuación 1.

$$\delta = (y - \hat{y}) f'(h) = (y - \hat{y}) f'(\sum_i w_i x_i)$$

Ecuación 1.

```python
def error_term(y,W,X,b):
    y_est=output_y(X,W,b)
    return (y-y_est)*(sigmoid_prime(y_est))
```

Se calcula el incremento de cada uno de los pesos aplicando la ecuación 2.

$$\Delta w_i = \eta \delta x_i$$

Ecuación 2.

```
def increment(W, X, b, eta, i, y):
    return eta*error_term(y,W,X,b)*X[i]
```

Se inicializan los valores de los pesos, los valores de entrada, la tasa de aprendizaje y el valor de salida esperado.

```
# valores de ejemplo
learning_rate = 0.1
x = np.array([1,1])
y = 5

# Initial weights
w = np.array([0.1,0.2])
b = 0
```

Se muestran los valores de salida de la red junto con el error.

```
# Calcular la salida de la red
y_est=output_y(x,w,b)
print('Salida:', y_est)

# Calcula el error residual de la red

print('Error:', y-y_est)

# Calcula el termino de error

print('Termino de error:', error_term(y,w,x,b))
```

Posteriormente a la primera iteración (época) se calculan los nuevos pesos.

```
# Calcula el incremento de los pesos

dw0=increment(w, x, b, learning_rate, 0, y)
dw1=increment(w, x, b, learning_rate, 1, y)
print('Incremento w1:', dw0)
print('Incremento w2:', dw1)

# Calcula el nuevo valor del los pesos
w[0]+=dw0
w[1]+=dw1
print('Nuevos pesos:',w)

# Calcula el nuevo error
y_est=output_y(x,w,b)
print('Nuevo error:',y-y_est)
```

Resultados.

```
Salida: 0.574442516811659
Error: 4.425557483188341
Termino de error: 1.019911404726894
Incremento w1: 0.10199114047268941
Incremento w2: 0.10199114047268941
Nuevos pesos: [0.20199114 0.30199114]
Nuevo error: 4.376605275390046
```

Como se pudo observar el incremento de ambos pesos resultó ser el mismo ya que al multiplicar el término de error y la tasa de aprendizaje por los valores de entrada los cuales resultaron ser 1, se obtuvo un incremento homogéneo.

De igual forma se pudo observar que el nuevo error generado a partir de los pesos actualizados se redujo en 0.049 lo que implica que el algoritmo está funcionando.

Posteriormente se replicó el algoritmo pero ahora con una tasa de aprendizaje de 0.3, lo que nos dio un nuevo error de 4.2866, que es aún más bajo que en la primera prueba.

```
# valores de ejemplo
learning_rate = 0.3
x = np.array([1,1])
y = 5
```

```
Salida: 0.574442516811659
Error: 4.425557483188341
Termino de error: 1.019911404726894
Incremento w1: 0.3059734214180682
Incremento w2: 0.3059734214180682
Nuevos pesos: [0.40597342 0.50597342]
Nuevo error: 4.28660161824467
```

Descenso por gradiente completo

Para programar el algoritmo de descenso por gradiente de forma completa se van a definir una serie de parámetros similares a el descenso por un paso agregando un parámetro que nos va a indicar el número de veces que se va a ejecutar dicho algoritmo, este parámetro se conoce como el número de “épocas”.

Para este ejemplo se va a trabajar con un dataset de valores numéricos (variables cuantitativas).

Primeramente se cargan los paquetes correspondientes

```
# importamos los paquete necesarios
import numpy as np

# cargamos datos de ejemplo
from data_prep import features, targets, features_test, targets_test
```

Se define la función de activación sigmoide.

```
def sigmoid(x):

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

Se inicializan los pesos de forma aleatoria con una distribución normal para evitar “neuronas muertas”.

```
weights = np.random.normal(scale=1 / n_features**.5, size=n_features)
```

Se hace una prueba de la precisión de la red antes de realizar el entrenamiento utilizando los pesos inicializados previamente.

```
tes_out = sigmoid(np.dot(features_test, weights))
predictions = tes_out > 0.5
accuracy = np.mean(predictions == targets_test)
print("Prediction accuracy: {:.3f}".format(accuracy))
```

Resultado.

```
Prediction accuracy: 0.475
```

Como se puede observar la precisión del modelo resultó ser mala.

Posteriormente se define los hiper paramétricos de la red, que como se mencionó anteriormente corresponden con el número de épocas y la tasa de aprendizaje.

```
# número de épocas
epochs = 1000
# tasa de aprendizaje
learnrate = 0.5
```

Se calculan los valores de salida por cada época y se obtienen los parámetros del error para realizar la actualización de los pesos.

```
output = sigmoid(np.dot(x,weights))

#calcula el error
error = y-output

#termino de error
t_error=error*output*(1-output)

#calcula el incremento
del_w+=learnrate*t_error*x
```

Se actualizan los pesos con los parámetros obtenidos previamente

```
#Actualiza los pesos
weights += del_w * (1/len(features.values))
```

Para verificar que el algoritmo funcione correctamente, se imprimen los valores del error cuadrático medio del modelo entrenado.

```
out = sigmoid(np.dot(features, weights))
loss = np.mean((out - targets) ** 2)
if last_loss and last_loss < loss:
    print("Train loss: ", loss, "  WARNING - Loss Increasing")
else:
    print("Train loss: ", loss)
last_loss = loss
```

Finalmente se calcula nuevamente la precisión del modelo utilizando los nuevos pesos.

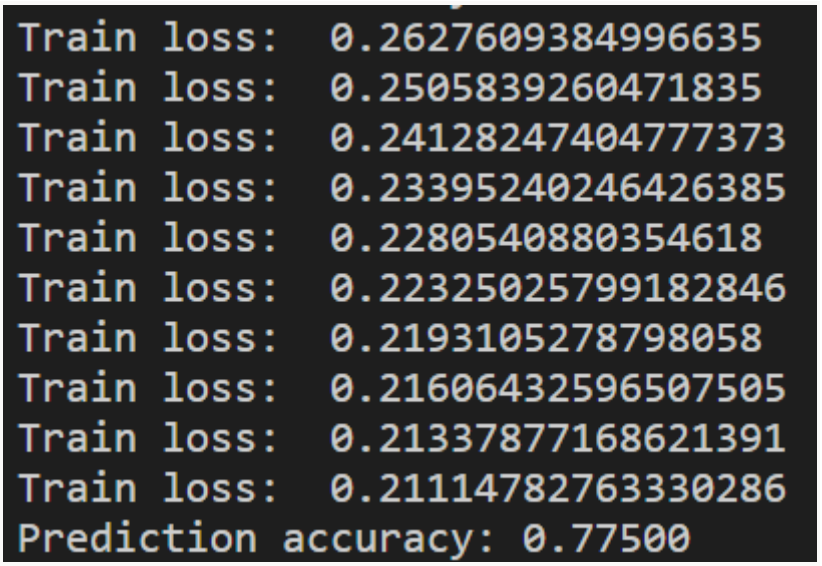
```
tes_out = sigmoid(np.dot(features_test, weights))
predictions = tes_out > 0.5
accuracy = np.mean(predictions == targets_test)
print("Prediction accuracy: {:.5f}".format(accuracy))
```

Resultados.

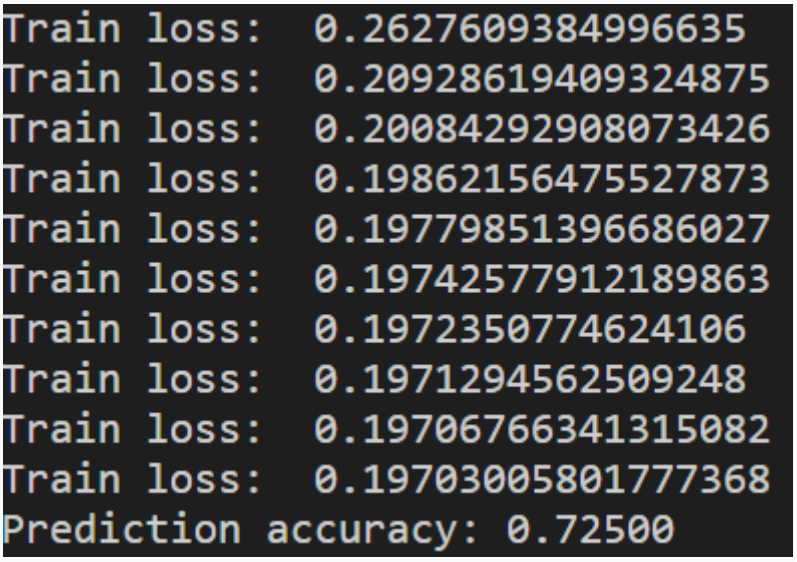
Número de épocas: 10

```
Prediction accuracy: 0.475
Train loss:  0.2627609384996635
Train loss:  0.261377597556950 34
Train loss:  0.260035123660686 65
Train loss:  0.25873190537061 82
Train loss:  0.257466377967583 37
Train loss:  0.256237028247114 86
Train loss:  0.255042397929535 65
Train loss:  0.253881085892184 17
Train loss:  0.252751749410841 94
Train loss:  0.251653104578100 25
Prediction accuracy: 0.50000
```

Número de épocas: 100



Número de épocas: 1000



Como se pudo observar en los resultados, el número de épocas influye en la precisión del modelo, ya que al tener un mayor número de iteraciones, el modelo queda mejor ajustado. Sin embargo el número de épocas puede influir de forma negativa en la precisión del modelo ya que como se pudo observar, cuando se realizó un entrenamiento con 1000 épocas, se obtuvo una precisión menor a cuando se tienen 100 épocas, lo que implica que el modelo tuvo una mayor pérdida, o bien mayor error.