CI-0148 Aprendizaje Máquina
Prof. Pablo Sauma Chacón

Laboratorio 4

Objetivo: Familiarizar al estudiante con la creación y el uso de redes neuronales desde una perspectiva de modelo matemático..

Enunciado: Lea las instrucciones del documento y resuelva el enunciado en Python. Al inicio de su documento adjunte su nombre y carnet como un comentario. Este trabajo es de carácter individual, tampoco se aceptarán códigos que no hayan sido desarrollados por su persona.

Para este laboratorio se trabajará con el set de datos: titanic.csv. Implemente las siguientes funciones de activación (y sus derivadas asociadas) que serán utilizadas para la construcción de redes neuronales.

1. Función `sigmoid(x)` que recibe un número `x` y retorna el resultado de la función sigmoide para dicho `x`: `1/(1 + e^(-x))`

2. Función `d_sigmoid(y)` que recibe un número `y` (tal que `y = sigmoid(x)`) y retorna el resultado de la derivada de la función sigmoide para el `y` dado: `y*(1-y)`

3. Función `tanh(x)` que recibe un número `x` y retorna el resultado de la función tangente hiperbólico para dicho `x`: `(e^x - e^(-x))/(e^x + e^(-x))`

4. Función `d_tanh(y)` que recibe un número `y` (tal que `y = tanh(x)`) y retorna el resultado de la derivada de la función tangente hiperbólico para el `y` dado: `1 - y^2`

5. Función `relu(x)` que recibe un número `x` y retorna el resultado de la función lineal rectificada para dicho `x`: `(x if x>0 else 0)`

6. Función `d_relu(y)` que recibe un número `y` (tal que `y = relu(x)`) y retorna el resultado de la derivada de la función lineal rectificada para el `y` dado: `1 if y>0 else 0`

7. Función `lrelu(x)` que recibe un número `x` y retorna el resultado de la función lineal rectificada con fuga para dicho `x`: `(x if x>0 else 0.01x)`

8. Función `d_lrelu(y)` que recibe un número `y` (tal que `y = lrelu(x)`) y retorna el resultado de la derivada de la función lineal rectificada con fuga para el `y` dado: `1 if y>0 else 0.01`

También deberá implementar la clase **`DenseNN`** que corresponde a una red neuronal densa con los siguientes métodos:

9. Método constructor `__init__(self, layers, activation, seed=0)` que recibe un arreglo de números enteros `layers` que especifica la cantidad de neuronas en cada capa de la red neuronal (tome en cuenta que el primer valor corresponde a la cantidad de datos de entrada y el último valor la cantidad de valores de salida) y un arreglo de caracteres/hileras `activation` que especifica la función de activación a utilizar en cada una de las capas de la red (con excepción de la capa de entrada). `'s'` corresponde a la función sigmoide, `'t'` a la función tangente hiperbólico, `'r'` a la función relu y `'l'` a leaky-relu. Por último, recibe una semilla `seed` que se utiliza para sembrar la aleatoriedad con la que se generan los valores de los pesos iniciales. Utilice la inicialización de Xavier para iniciar los pesos de la red.
    a. Recuerde que al crear la matriz de pesos, debe considerar los pesos del "bias" adicional a las neuronas de la capa adicional.
    b. Puede además inicializar todas las estructuras necesarias para el funcionamiento de la red neuronal: estructura para almacenar los valores netos, valores de activación y errores δ de cada capa.

10. Método `predict(self, x)` que recibe una matriz de datos `x` de tamaño `nxm`, donde `m` coincide con el primer valor de `layers` al construir la red neuronal. Este método ejecuta el *forward propagation* aplicando la multiplicación con la matrices de pesos y las funciones de activación de cada capa. Finalmente, retorna una matriz de datos `nxp`, donde `p` coincide con el último de `layers` y corresponde a los valores predichos por la red neuronal para los casos `x`.

11. Método `train(self, lr=0.05, momentum=0, decay=0)` que recibe los parámetros de entrenamiento e inicializa las matrices Δw para acumular los cambios a aplicar para cada una de las matrices de pesos. Inicia el contador de *epoch* en 0.

12. Método `backpropagation(self, x, y)` que recibe una matriz de datos `nxm x`, una matriz de valores esperados `nxp y`, y ejecuta *forward propagation* y posteriormente *back propagation*. Asuma que la función de error a utilizar será MSE independiente del problema/función de activación de la capa final. Debe retornar el total del error.
    a. Tome en cuenta lo visto en clase a la hora de calcular los valores de error δ.
    b. Cuide las dimensiones de las matrices, recuerde que cada matriz debe considerar sus pesos del *bias*, sin embargo estos valores no son considerados al transmitir el error hacia capas anteriores (dado que no forman parte de una neurona).
    c. Considere que `x` y `y` pueden poseer múltiples datos a procesar en una sola iteración; sin embargo si lo desea puede implementar la evaluación de los casos de manera iterativa (no es óptimo, pero es válido).
    d. Los valores Δw obtenidos se deben acumular, pero no aplicar, esto se hará en el método `step`.

13. Método `step(self)` que ejecuta la actualización de los pesos de la matriz, aplica la actualización de la tasa de aprendizaje en caso de decaimiento y aplica el momentum. Además avanza el contador de época en 1.

14. Utilice la clase implementada con los datos de titanic, vaya imprimiendo los valores del error producto de cada iteración de backpropagation, seguido de un llamado a step para actualizar los pesos. ¿Se va reduciendo el error? Intente diferentes combinaciones de parámetros, arquitecturas y funciones de activación para lograr que su algoritmo converja. ¿Cuál combinación le produjo el mejor resultado? ¿Qué aprendió de estos experimentos? (Se vale no haber aprendido nada, venimos a experimentar)