Implementación de una Red Neuronal Artificial Single Layer perceptrón

Vargas Arqque Jeremyk Rufino

155183@unsaac.edu.pe

Huillca Mozo Brayan

160329@unsaac.edu.pe

Universidad Nacional de San Antonio Abad del Cusco, Facultad de Ingeniería Eléctrica, Electrónica, Informática y Mecánica

Resumen

Las Redes Neuronales Artificiales son las que en la actualidad causan un mayor impacto, debido a sus diferentes aplicaciones, recientemente esta tecnología ha captado la atención de los profesionales dedicados a la estadística y al análisis de datos, los cuales comienzan a incorporar las redes neuronales al conjunto de herramientas estadísticas orientadas a la clasificación de patrones y la estimación de variables continuas.

El presente trabajo describe una de los diferentes algoritmos para Redes Neuronales Artificiales como es el Single Layer Perceptrón, debido a su gran aplicabilidad de esta red neuronal artificial implementamos un algoritmo utilizando la herramienta Py-Spark donde podremos apreciar la formulación del algoritmo, preprocesamiento de los datos, entrenamiento y predicción.

A diferencia de muchas otras investigaciones sobre este tema, la presente considera el perceptrón de una sola capa como un proceso en el que los pesos del perceptrón van aumentando. Durante el entrenamiento de propagación y retro-propagación, el límite de decisión de perceptrón de una sola capa se vuelve idéntico o cercano al de clasificadores estadísticos.

Palabras clave: perceptrón, entrenamiento, red neuronal

Abstract

Artificial Neural Networks are the ones that currently cause the greatest impact, due to their different applications, recently this technology has caught the attention of professionals dedicated to statistics and data analysis, who are beginning to incorporate neural networks into set of statistical tools oriented to the classification of patterns and the estimation of continuous variables.

This report describes one of the different algorithms for Artificial Neural Networks such as the Single Layer Perceptron, due to its great applicability of this artificial neural network, we implement an algorithm using the Py-Spark tool where we can appreciate the formulation of the algorithm, preprocessing of the data, training and prediction.

Unlike many other investigations on this topic, this one considers the single layer perceptron as a process in which the weights of the perceptron are increasing. During propagation and backpropagation training, the single-layer perceptron decision boundary becomes identical or close to that of statistical classifiers.

1. Introducción

Una de las principales características de las redes neuronales vienes a ser su capacidad para aprender a partir de alguna fuente de información interactuando con su entorno. El psicólogo Frank Rosenblat desarrolló un modelo simple de neurona basado en el modelo de McCulloch y Pitts y en una regla de aprendizaje basado en la corrección del error. A este modelo le llamo Perceptrón en 1958. Una de las características que más interés despertó de este modelo fue la capacidad de aprender a reconocer patrones. El perceptrón está constituido por conjunto de sensores de entrada que reciben los patrones de entrada a reconocer o clasificar y una neurona de salida que se ocupa de clasificar a los patrones de entrada en dos clases, según que la salida de la misma es binaria.

El perceptrón de una sola capa es la red neuronal más simple. Contiene una capa de entrada y una capa de salida, y la capa de entrada y la capa de salida están conectadas directamente. A diferencia del modelo MP propuesto más temprano, los pesos de las sinapsis neuronales son variables, por lo que se pueden aprender a través de ciertas reglas. El problema linealmente separable se puede resolver de forma rápida y fiable.

2. Herramientas y Metodología

2.1. Single Layer Perceptrón

2.1.1.Definición

El Perceptrón simple fue introducido en 1958 por Frank Rosenblantt. Se trata del modelo más sencillo de redes neuronales artificiales, ya que consta de una sola capa de neuronas con una única salida y.

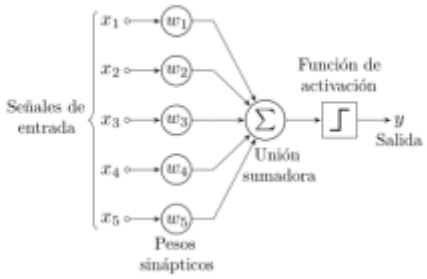

Figura 1. Ejemplo de Perceptrón simple con n = 5 entradas.

2.1.2.Regla de Aprendizaje

Para la determinación de los pesos sinápticos y del umbral vamos a seguir un proceso adaptativo que consiste en comenzar con unos valores iniciales aleatorios e ir modificándolos iterativamente cuando la salida de la unidad y no coincida con la salida deseada z. Vamos a tener en cuenta que el umbral se puede considerar como el peso correspondiente a un nuevo sensor de entrada que tiene siempre una entrada igual a xn+1 = −1. La regla que vamos a seguir para modificar los pesos sinápticos se conoce con el nombre de regla de aprendizaje del Perceptrón simple [7] y viene dada por la expresión:

$$\begin{cases} w_j(k+1) = w_j(k) + \Delta w_j(k), \quad k = 1, 2, \ldots \\ \text{siendo } \Delta w_j(k) = \eta(k)[\, z(k) - y(k)]x_j(k) \end{cases}$$

Es decir,

$$\boxed{w_j(k+1) = w_j(k) + \eta(k)\ [z(k) - y(k)]x_j(k)}$$

Figura 2. Regla de aprendizaje del Perceptrón simple

Nos indica que la variación del peso wj es proporcional al producto del error z(k) − y(k) por la componente j-ésima del patrón de entrada que hemos introducido en la iteración k, es decir, xj(k). La constante de proporcionalidad η(k) es un parámetro positivo que recibe el nombre de tasa de aprendizaje, ya que cuanto mayor es, más se modifica el peso sináptico y viceversa. Es decir, es el parámetro 5 6 Capítulo 2. El Perceptrón simple que controla el proceso de aprendizaje. Cuando es muy pequeño la red aprende poco a poco. Si se toma constante en todas las iteraciones, η(k) = η > 0 tendremos la regla de aprendizaje con incremento fijo. Cuando usamos la función signo como función de transferencia, la regla de aprendizaje se puede escribir de la siguiente forma:

$$w_j(k+1) = \begin{cases} w_j(k) + 2\eta(k)x_j(k) & \text{si} \quad y(k) = -1 \text{ y } z(k) = 1 \\ w_j(k) & \text{si} \quad y(k) = z(k) \\ w_j(k) - 2\eta(k)x_j(k) & \text{si} \quad y(k) = 1 \text{ y } z(k) = -1 \end{cases}$$

Figura 3. Función signo para la Regla de Aprendizaje

2.2. Py-Spark

2.2.1.Descripción

Apache Spark es otro proyecto bajo el amparo de la fundación Apache. Spark se basa en Hadoop, y lo extiende para soportar flujos de ejecución más complejos, solucionando la mayor limitación de Hadoop. Asimismo, la gestión del sistema de computación sobre el que se ejecuta es extremadamente sencilla. Adicionalmente, requiere codificar relativamente poco en el lado del usuario, lo cual potencia la presencia de la lógica del algoritmo en el código.

Apache Spark provee una abstracción de datos más sencilla que Hadoop, conocida como Resilient Distributed Datasets, o RDD. Esta abstracción es muy similar a una lista, sin embargo, los elementos no están ordenados, por lo que no es posible acceder a ellos directamente. Los RDD son colecciones perezosas, particionadas, replicadas e inmutables de objetos. Al igual que Hadoop, al usar replicación, son tolerantes a fallos, sin embargo, a diferencia de él, proveen muchas más operaciones de alto nivel, además de map y reduce, algunas de las cuales se detallan a continuación:

- Map Al igual que Hadoop, esta función permite transformar los elementos contenidos en el RDD.
- Filter Esta función permite filtrar elementos del RDD en función de un predicado.
- Distinct Elimina elementos duplicados del RDD.
- Reduce Análoga a la etapa de mismo nombre de Hadoop, aplica una función (que, al igual que en Hadoop, debe ser asociativa para que el resultado sea robusto) incrementalmente a los elementos del RDD.
- Count Obtiene el recuento de los elementos del RDD.

2.3. Anaconda Navigator

Anaconda Navigator es una interfaz gráfica de escritorio incluida en el paquete de distribución de Anaconda, que se puede usar para iniciar aplicaciones fácilmente y administrar paquetes, entornos y canales de conda sin usar comandos de línea de comandos. Navigator puede buscar paquetes en Anaconda Cloud o en los almacenes locales de Anaconda. Proporcione las versiones Windwos, maxOS y Linux.

Después de instalar Anaconda, las siguientes aplicaciones se incluyen de forma predeterminada:

- Jupyter Notebook
- Orange App
- QTConsole
- Glueviz
- Spyder
- RStudio

2.3.1.Jupyter Notebook

Jupyter Notebook proporciona un entorno pensado para satisfacer necesidades concretas y ajustarse al flujo de trabajo de la ciencia de datos y la simulación numérica. En una sola interfaz, los usuarios pueden escribir, documentar y ejecutar código, visualizar datos, realizar cálculos y ver los resultados. Concretamente, la fase de prototipado incluye la ventaja de que el código se organiza en celdas independientes, es decir, es posible probar bloques concretos de código de forma individual. Gracias a que existen muchos kernels o núcleos adicionales, Jupyter no se limita al lenguaje de programación Python, lo que aporta muchísima flexibilidad a la hora de crear código y de hacer análisis.

Algunos de los principales usos que se da a Jupyter Notebook:

- Depuración de datos: distinguir entre los datos que son importantes y los que no lo son al ejecutar un análisis de big data.
- Modelización estadística: método matemático para estimar la probabilidad de distribución de una característica concreta.
- Creación y entrenamiento de modelos de aprendizaje automático: diseño, programación y entrenamiento de modelos basados en aprendizaje automático
- Visualización de datos: representación gráfica de datos para visualizar con claridad patrones, tendencias, interdependencias, etc.

2.4. Implementación del Algoritmo

2.4.1. Inicialización del entorno spark en jupyter notebook

```
In [1]: import findspark
        findspark.init()
        findspark.find()
Out[1]: 'C:\\spark-3.2.0-bin-hadoop3.2'

In [2]: import pyspark

In [3]: from pyspark import SparkContext
        sc=SparkContext("local","Test-app")
```

Figura 4. Inicialización de py-spark

2.4.2. Carga y Preparación de los datos

Los datos utilizados para la predicción utilizando el algoritmo Single Layer perceptrón es el conjunto de datos de Flor de Iris, es un conjunto de datos multivariante introducido por Ronald Fisher en su artículo de 1936, *The use of multiple measurements in taxonomic problems* (*El uso de medidas múltiples en problemas taxonómicos*)

2.4.2.1. Carga de la data

Para este ejemplo se utilizaron solo dos especies de flor de iris como Iris-setosa e Iris-versicolor, además se tomaron 39 datos de la data completa, 14 datos de entrenamiento y 25 datos de test.

CARGAR DATA

```
In [30]: spark = SparkSession.builder.appName('pyspark-ml').getOrCreate()

In [31]: iris= spark.read.csv('Iris.csv', header=True, inferSchema=True)
         iris.show(40)
+---+-------------+------------+-------------+------------+--------------+
| Id|SepalLengthCm|SepalWidthCm|PetalLengthCm|PetalWidthCm|       Species|
+---+-------------+------------+-------------+------------+--------------+
|  1|          5.1|         3.5|          1.4|         0.2|   Iris-setosa|
|  2|          4.9|         3.0|          1.4|         0.2|   Iris-setosa|
|  3|          4.7|         3.2|          1.3|         0.2|   Iris-setosa|
|  4|          4.6|         3.1|          1.5|         0.2|   Iris-setosa|
|  5|          5.0|         3.6|          1.4|         0.2|   Iris-setosa|
|  6|          5.4|         3.9|          1.7|         0.4|   Iris-setosa|
|  7|          4.6|         3.4|          1.4|         0.3|   Iris-setosa|
```

Figura 5. Datos de Flor de Iris

2.4.2.2. Tipos de datos

Los datos de entrada son de tipo double y los de salida de tipo string.

TIPOS DE DATOS

```
In [33]: iris.printSchema()

root
 |-- Id: integer (nullable = true)
 |-- SepalLengthCm: double (nullable = true)
 |-- SepalWidthCm: double (nullable = true)
 |-- PetalLengthCm: double (nullable = true)
 |-- PetalWidthCm: double (nullable = true)
 |-- Species: string (nullable = true)
```

Figura 6. Tipos de datos

2.4.2.3. Normalización Binaria

Normalizamos los datos de salida, en este caso a la especie Iris-Setosa reemplzamos por 1 y a Iris-Versicolor por 0, colocamos estos datos en una columna vectorial llamada Train_Test.

NORMALIZACION BINARIA

```
In [34]: #Si la columna de entrada es string, la convertimos en numero. Los índices están en [0, numLabels).
indexer = StringIndexer(inputCol = 'Species', outputCol = 'Train_Test')
data_iris= indexer.fit(iris).transform(iris)
data_iris.show(40)
```

```
+---+-------------+------------+-------------+------------+---------------+----------+
| Id|SepalLengthCm|SepalWidthCm|PetalLengthCm|PetalWidthCm|        Species|Train_Test|
+---+-------------+------------+-------------+------------+---------------+----------+
|  1|          5.1|         3.5|          1.4|         0.2|    Iris-setosa|       1.0|
|  2|          4.9|         3.0|          1.4|         0.2|    Iris-setosa|       1.0|
|  3|          4.7|         3.2|          1.3|         0.2|    Iris-setosa|       1.0|
|  4|          4.6|         3.1|          1.5|         0.2|    Iris-setosa|       1.0|
|  5|          5.0|         3.6|          1.4|         0.2|    Iris-setosa|       1.0|
|  6|          5.4|         3.9|          1.7|         0.4|    Iris-setosa|       1.0|
|  7|          4.6|         3.4|          1.4|         0.3|    Iris-setosa|       1.0|
|  8|          5.0|         3.4|          1.5|         0.2|    Iris-setosa|       1.0|
|  9|          4.4|         2.9|          1.4|         0.2|    Iris-setosa|       1.0|
| 51|          7.0|         3.2|          4.7|         1.4|Iris-versicolor|       0.0|
| 52|          6.4|         3.2|          4.5|         1.5|Iris-versicolor|       0.0|
| 53|          6.9|         3.1|          4.9|         1.5|Iris-versicolor|       0.0|
| 54|          5.5|         2.3|          4.0|         1.3|Iris-versicolor|       0.0|
| 55|          6.5|         2.8|          4.6|         1.5|Iris-versicolor|       0.0|
| 56|          5.7|         2.8|          4.5|         1.3|Iris-versicolor|       0.0|
| 57|          6.3|         3.3|          4.7|         1.6|Iris-versicolor|       0.0|
| 58|          4.9|         2.4|          3.3|         1.0|Iris-versicolor|       0.0|
| 59|          6.6|         2.9|          4.6|         1.3|Iris-versicolor|       0.0|
| 60|          5.2|         2.7|          3.9|         1.4|Iris-versicolor|       0.0|
| 61|          5.0|         2.0|          3.5|         1.0|Iris-versicolor|       0.0|
| 62|          5.9|         3.0|          4.2|         1.5|Iris-versicolor|       0.0|
| 63|          6.0|         2.2|          4.0|         1.0|Iris-versicolor|       0.0|
| 64|          6.1|         2.9|          4.7|         1.4|Iris-versicolor|       0.0|
```

Figura 7. Normalización Binaria

2.4.2.4. Fusión de columnas

Utilizando la función Vector Assembler de Py-Spark creamos un transformador de características que fusiona varias columnas en una columna vectorial.

En este caso juntamos las 4 primeras columnas correspondientes a longitud de sépalo, ancho de sépalo, longitud de pétalo, ancho de pétalo y el valor de salida normalizado y en una sola columna vectorial llamada features.

```
In [36]: #Un transformador de características que fusiona varias columnas en una columna vectorial.
vectorAssembler = VectorAssembler(inputCols = ['SepalLengthCm','SepalWidthCm','PetalLengthCm','PetalWidthCm','Train_Test'],
norm_iris = vectorAssembler.transform(data_iris)
norm_iris.show(40)
```

```
+---+-------------+------------+-------------+------------+---------------+----------+--------------------+
| Id|SepalLengthCm|SepalWidthCm|PetalLengthCm|PetalWidthCm|        Species|Train_Test|            features|
+---+-------------+------------+-------------+------------+---------------+----------+--------------------+
|  1|          5.1|         3.5|          1.4|         0.2|    Iris-setosa|       1.0|[5.1,3.5,1.4,0.2,...|
|  2|          4.9|         3.0|          1.4|         0.2|    Iris-setosa|       1.0|[4.9,3.0,1.4,0.2,...|
|  3|          4.7|         3.2|          1.3|         0.2|    Iris-setosa|       1.0|[4.7,3.2,1.3,0.2,...|
|  4|          4.6|         3.1|          1.5|         0.2|    Iris-setosa|       1.0|[4.6,3.1,1.5,0.2,...|
|  5|          5.0|         3.6|          1.4|         0.2|    Iris-setosa|       1.0|[5.0,3.6,1.4,0.2,...|
|  6|          5.4|         3.9|          1.7|         0.4|    Iris-setosa|       1.0|[5.4,3.9,1.7,0.4,...|
|  7|          4.6|         3.4|          1.4|         0.3|    Iris-setosa|       1.0|[4.6,3.4,1.4,0.3,...|
|  8|          5.0|         3.4|          1.5|         0.2|    Iris-setosa|       1.0|[5.0,3.4,1.5,0.2,...|
|  9|          4.4|         2.9|          1.4|         0.2|    Iris-setosa|       1.0|[4.4,2.9,1.4,0.2,...|
| 10|          4.9|         3.1|          1.5|         0.1|    Iris-setosa|       1.0|[4.9,3.1,1.5,0.1,...|
| 11|          5.4|         3.7|          1.5|         0.2|    Iris-setosa|       1.0|[5.4,3.7,1.5,0.2,...|
| 12|          4.8|         3.4|          1.6|         0.2|    Iris-setosa|       1.0|[4.8,3.4,1.6,0.2,...|
| 13|          4.8|         3.0|          1.4|         0.1|    Iris-setosa|       1.0|[4.8,3.0,1.4,0.1,...|
| 14|          4.3|         3.0|          1.1|         0.1|    Iris-setosa|       1.0|[4.3,3.0,1.1,0.1,...|
```

Figura 8. Fusión de Columnas

2.4.3. Código del Algoritmo

2.4.3.1. Función Sigmoide

Módulo de la función sigmoide.

```
#Su propósito es escalar la salida en el rango (0, 1) o rango (-1,+1)
def sigmoide(x):
    return (1/(1+math.e**(-x)))
```

Figura 9

2.4.3.2. Función Propagación

Este es el módulo de la función de propagación donde se realiza la operación producto punto.

```
def propagacion(dato,w1,w2,w3,w4):
    xw=dato.flatMap(lambda x: [(x[0]*w1+x[1]*w2+x[2]*w3+x[3]*w4)])
    xw=xw.collect()
    fa=sigmoide(xw[0])
    return fa
```

Figura 10

2.4.3.3. Función de Retro propagación

La función de retro propagación calcula los nuevos pesos para la predicción.

```
#Metodo de Retropropagacion eficiente
def retropropagacion(dato,coef_apren,prop,w1,w2,w3,w4):
    w=[w1,w2,w3,w4]
    wr=dato.flatMap(lambda x:[(i+coef_apren*(x[4]-prop)*x[w.index(i)]) for i in w])
    wr_aux=wr.collect()
    return wr_aux[0],wr_aux[1],wr_aux[2],wr_aux[3]
```

Figura 11

2.4.3.4. Función de Entrenamiento

Este es la función de Entrenamiento donde generamos los pesos necesarios para la predicción, en este caso se realizará un bucle utilizando Py-Spark con un numero de Épocas igual a 10

```
#Este es el modulo de entrenamiento mejorado
def entrenamiento_mejorado(dato,Epoc,coef,w1,w2,w3,w4):
    Sa=dato.map(lambda x:[retropropagacion(x,coef,propagacion(x,w1,w2,w3,w4),w1,w2,w3,w4) for i in Epoc])
    return Sa
```

Figura 12

2.4.3.5. Main()

```
#Datos de Entrenamiento
Train =[[[5.1,3.5,1.4,0.2,0]],[[5.8,4.0,1.2,0.2,0]],[[7.0,3.2,4.7,1.4,1]],[[5.2,2.7,3.9,1.4,1]]]
datos=sc.parallelize(Train)
#Definimos nuestras pesos iniciales
w1=-0.1
w2=0.1
w3=-0.2
w4=0.25
umbral=0.8
#Diez epocas para un mejor entrenamiento
Epocas_aux=[1,1,1,1,1,1,1,1,1,1]
coeficiente_aprendizaje=0.25
Salida=entrenamiento2(datos.collect(),Epocas_aux,coeficiente_aprendizaje,w1,w2,w3,w4,Train)
print(Salida.collect())
```

Figura 13

3. Resultados y Discusión

TEST		Prediccion	Species- Salida
[[5.1, 3.5, 1.4, 0.2, 1.0]]	=>	0.040327943008868584	Iris-setosa
[[4.9, 3.0, 1.4, 0.2, 1.0]]	=>	0.07946544738312641	Iris-setosa
[[4.7, 3.2, 1.3, 0.2, 1.0]]	=>	0.05469113975431278	Iris-setosa
[[4.6, 3.1, 1.5, 0.2, 1.0]]	=>	0.09214484890670102	Iris-setosa
[[5.0, 3.6, 1.4, 0.2, 1.0]]	=>	0.03662203500213437	Iris-setosa
[[5.4, 3.9, 1.7, 0.4, 1.0]]	=>	0.0492656936798413	Iris-setosa
[[4.6, 3.4, 1.4, 0.3, 1.0]]	=>	0.05914577172856062	Iris-setosa
[[5.0, 3.4, 1.5, 0.2, 1.0]]	=>	0.05693953729237911	Iris-setosa
[[4.4, 2.9, 1.4, 0.2, 1.0]]	=>	0.10321971645299766	Iris-setosa
[[4.9, 3.1, 1.5, 0.1, 1.0]]	=>	0.07610236109067407	Iris-setosa
[[5.4, 3.7, 1.5, 0.2, 1.0]]	=>	0.03467125254168261	Iris-setosa
[[4.8, 3.4, 1.6, 0.2, 1.0]]	=>	0.0727642849074713	Iris-setosa
[[4.8, 3.0, 1.4, 0.1, 1.0]]	=>	0.07354444558766846	Iris-setosa
[[4.3, 3.0, 1.1, 0.1, 1.0]]	=>	0.04850703484467984	Iris-setosa
[[5.8, 4.0, 1.2, 0.2, 1.0]]	=>	0.011600757592067917	Iris-setosa
[[5.7, 4.4, 1.5, 0.4, 1.0]]	=>	0.016134633836746376	Iris-setosa
[[5.4, 3.9, 1.3, 0.4, 1.0]]	=>	0.022782242376323907	Iris-setosa
[[5.1, 3.5, 1.4, 0.3, 1.0]]	=>	0.04502512580149212	Iris-setosa
[[5.7, 3.8, 1.7, 0.3, 1.0]]	=>	0.04576815365253706	Iris-setosa
[[7.0, 3.2, 4.7, 1.4, 0.0]]	=>	0.990066143719718	Iris-Versicolor
[[6.4, 3.2, 4.5, 1.5, 0.0]]	=>	0.9890670123980453	Iris-Versicolor
[[6.9, 3.1, 4.9, 1.5, 0.0]]	=>	0.9949282899191606	Iris-Versicolor
[[5.5, 2.3, 4.0, 1.3, 0.0]]	=>	0.9913451784450206	Iris-Versicolor
[[6.5, 2.8, 4.6, 1.5, 0.0]]	=>	0.9945102566489331	Iris-Versicolor
[[5.7, 2.8, 4.5, 1.3, 0.0]]	=>	0.9934351116720196	Iris-Versicolor
[[6.3, 3.3, 4.7, 1.6, 0.0]]	=>	0.9927478028624472	Iris-Versicolor
[[4.9, 2.4, 3.3, 1.0, 0.0]]	=>	0.9549497740313839	Iris-Versicolor
[[6.6, 2.9, 4.6, 1.3, 0.0]]	=>	0.9918897024392125	Iris-Versicolor
[[5.2, 2.7, 3.9, 1.4, 0.0]]	=>	0.9855729037057518	Iris-Versicolor
[[5.0, 2.0, 3.5, 1.0, 0.0]]	=>	0.9810710527042072	Iris-Versicolor
[[5.9, 3.0, 4.2, 1.5, 0.0]]	=>	0.9869406439059207	Iris-Versicolor
[[6.0, 2.2, 4.0, 1.0, 0.0]]	=>	0.9875190447641659	Iris-Versicolor
[[6.1, 2.9, 4.7, 1.4, 0.0]]	=>	0.9949204199752292	Iris-Versicolor
[[5.6, 2.9, 3.6, 1.3, 0.0]]	=>	0.9577936342337198	Iris-Versicolor
[[6.7, 3.1, 4.4, 1.4, 0.0]]	=>	0.9856278371516598	Iris-Versicolor

Figura 14

El entrenamiento se realizó a 14 datos obtenidos de la data set Flor de Iris, los resultados obtenidos de la predicción son 99% correctos, para poder obtener estos resultados se incrementó el número de Épocas que nos ayudaron a que nuestros pesos tengan una mejor predicción, podríamos indicar que a mayor número de Épocas tendremos una tasa mayor para realizar la predicción que es la idea fundamental para crear una Red Neuronal, el tener un mayor porcentaje para la predicción.

4. Trabajo a futuro

Los trabajos a futuro a considerar sobre este trabajo son mejorar la funcionabilidad del algoritmo, para que evite hacer mas de lo que se necesita, para obtener una mayor rapidez y mejor predicción, además de la aplicabilidad a mayores datos de salidas, aumentando neuronas para su mejor predicción, en resumen, un Multilayer Perceptrón, ya que la mayoría de los data sets que podemos encontrar tienen 2 o más salidas y un Multilayer Perceptrón que sería mucho más general.

5. Conclusiones

- El Single Layer Perceptrón es un clasificador binario, que nos ayuda a separar los datos de entrada en dos categorías que deben ser linealmente separables, es decir que los datos deben ser separados por una recta.

- Los proyectos de redes neuronales modernas suelen trabajar desde unos miles a unos pocos millones de unidades neuronales, a pesar de ser muchas neuronas, siguen siendo de una magnitud menos compleja que la del cerebro humano.

- Las Redes Neuronales Artificiales Single Layer Perceptrón son una forma muy interesante de adentrarse al análisis de datos y Big Data, entender el comportamiento de Redes Neuronales muy complejas; ya que, al trabajar con una sola neurona, podemos entender cómo es que realizarían las tareas diversas neuronas

6. Referencias

Viedma, D. D. T., & López, D. S. G. Apache Spark.

Morera Munt, A., & Alcalá Nalvaiz, J. T. (2018). Introducción a los modelos de redes neuronales artificiales El Perceptrón simple y multicapa (Doctoral dissertation, Tesis de pregrado, Universidad Saragoza]. Saguan Repositorio Institucional de Documentos https://cutt. ly/lEVVf7v).

Yan, Y., & Yan, J. (2018). Hands-on data science with Anaconda: utilize the right mix of tools to create high-performance data science applications. Packt Publishing Ltd.