Colegio de Ciencias e Ingeniería

Inteligencia Artificial

NRC: 4004

Tarea 3

Integrantes:

Gabriel Oña– 320597

26 de noviembre del 2023

Deber 3

1. **Implementación de Algoritmo de Aprendizaje en Secuencia:**
    - **Implemente el algoritmo de aprendizaje secuencial del perceptrón y corra el mismo por 100 épocas. Inicializando los pesos de forma aleatoria para los tres datasets. En todos los casos enfóquese en medir el accuracy (es decir, el número de aciertos de clasificación versus el número total de eventos).**
    - **Para los tres datasets recopile el accuracy por época y genere gráficos (eje equis el número de época y en el eje ye el valor de accuracy para esa época).**
    - **Para todos los datasets dibuje el decision boundary (es decir, el hiperplano que divide los datos entre categoría +1 y -1)**

El algoritmo secuencial de aprendizaje para un perceptrón procesa cada conjunto de datos provenientes de los datasets para calcular la predicción, el error con respecto a la clasificación de elementos y actualiza los pesos del perceptrón en el mismo periodo de iteración.

A continuación, se presenta el pseudocódigo del algoritmo secuencial:

For each epoch:

For each event in X:

Y' = perceptron(x,w)

Wnew = Wold + η(Yi-Y')x

De esta forma se implementó la clase perceptrón de tipo secuencial que se indica en la figura 1.

In [44]:

```
class perceptron(object):
    def __init__(self,size,learning_rate, function):
        self.weights = np.random.randn(size+1)
        self.learning_rate = learning_rate
        self.accuracy = []
        self.activation = function

    def getWeights(self):
        return self.weights

    def getAccuracy(self,X,Y):
        predicciones = []
        for xi in X:
            predicciones.append(self.predict(xi))
        temp = 0
        for i in range(len(predicciones)):
            if predicciones[i] == Y[i]:
                temp += 1
        return temp/len(predicciones)

    def predict(self, data):
        V = np.dot(data,self.weights[1:]) + self.weights[0]
        return self.activation(V)

    def updates(self,xi,yi):
        y_predict = self.predict(xi)

        self.weights[1:] += self.learning_rate*(yi-y_predict)*xi
        self.weights[0] += self.learning_rate*(yi-y_predict)

    def training(self,X,Y,epochs):
        for i in range(epochs):
            for xi,yi in zip(X,Y):
                self.updates(xi,yi)
            self.accuracy.append(self.getAccuracy(X,Y))
```

Figure 1. Definición clase Perceptron secuencial

Como se puede apreciar, en la clase training se encuentra la actualización tipo secuencial indicada en pseudocódigo. Adicionalmente, se implementa código para el cálculo de la precisión del modelo el cual se va a guardar en cada iteración de épocas. Esto indicará el comportamiento temporal que tiene este algoritmo en el entrenamiento del perceptrón.

Adicionalmente, se implementa las funciones de activación que se ajustan a cada tipo de dataset. A continuación, se presenta:

In [45]:

```
def sgn_function(V):
    if isinstance(V,np.number):
        if V >= 0:
            return 1
        else:
            return 0
    else:
        respuesta = []
        for vi in V:
            if vi >= 0:
                respuesta.append(1)
            else:
                respuesta.append(0)
        return np.array(respuesta)
def sgn_negativo(V):
    if isinstance(V,np.number):
        if V >= 0:
            return 1
        else:
            return -1
    else:
        respuesta = []
        for vi in V:
            if vi >= 0:
                respuesta.append(1)
            else:
                respuesta.append(-1)
        return np.array(respuesta)

def xor_function(V):
    if V >= 1:
        return 1
    else:
        return 0
```

Figura 2. Funciones de Activación

Se implementó por lo tanto esta clase para tres datasets, el primero con distribución lineal, el segundo con distribución tipo xor y el último con distribución no lineal. A continuación, se presenta los resultados.

- **Lineal**

```
P1 = perceptron(2,0.1,sgn_function)
P1.training(X_linear,Y_linear,100)
print("Accuracy: " + str(P1.accuracy[99]*100) + "%")

plt.scatter(blue_linear[:,0],blue_linear[:,1])
plt.scatter(red_linear[:,0],red_linear[:,1])

plt.xlim(0, 5)
plt.ylim(0, 5)
```

```
Accuracy: 100.0%
```

Out[66]: (0.0, 5.0)

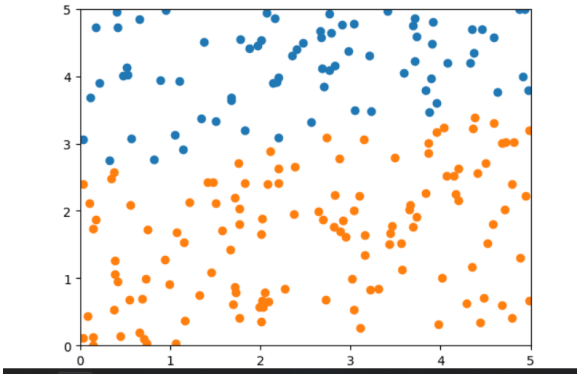

Figura 3. Training and plot dataset 1

Se entrena por lo tanto al primer perceptrón con dos datos de entrada y un dato de salida. El ratio de entrenamiento se lo coloca en 0.1 y la función de activación es de tipo escalón unitario que pasa de 0 a 1 en la transición de datos negativos a positivos. Se aprecia que el porcentaje de precisión del modelo es del 100%. Esto indica que el modelo encontró una manera de dividir ambas clasificaciones. A continuación, se presenta la gráfica de precisión vs época.

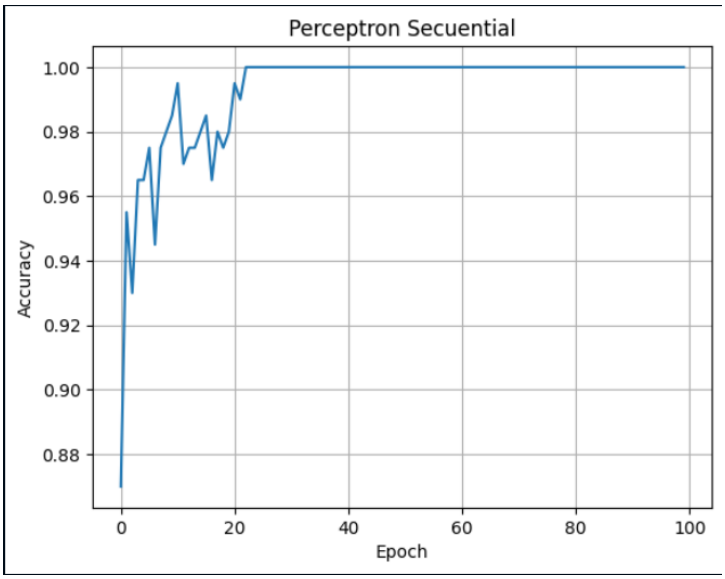

Figure 4. Precisión durante el entrenamiento

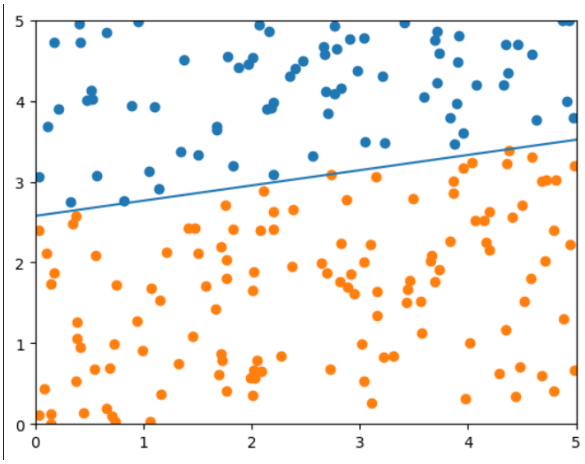

Figure 5. Frontera de clasificación

Vemos que el perceptrón logra ajustar mediante la aproximación lineal de forma adecuada, por lo que, ante cualquier entrada, el perceptrón sabrá clasificar entre estos dos tipos de datos.

- XOR

Para los tipos de datos XOR se busca igualmente usar solo un perceptrón lo cual requiere que manipulemos los datos para intentar que su distribución se pueda clasificar de manera lineal. Se detalla el proceso que se llevó a cabo.

$$|(x > 0.5) - (y > 0.5)| < a$$

$$|(x < 0.5) - (y < 0.5)| < a$$

$$|(x > 0.5) - (y < 0.5)| > a$$

$$|(x < 0.5) - (y > 0.5)| > a$$

$$\exists a \in \mathbb{R} \rightarrow cumple\ con\ las\ condiciones\ anteriores$$

Por lo tanto, se procesa los datos con el siguiente código:

```
Xxor = np.loadtxt("XXOR.csv", delimiter=",",dtype = float)
Yxor = np.loadtxt("YXOR.csv")
red_xor = []
blue_xor = []
x_xor_processed = abs(Xxor[:,0] - Xxor[:,1])
```

Figure 6. Procesamiento data Xor

A continuación, se presenta los resultados:

```
In [73]: P2 = perceptron(1,0.1,xor_function)
         P2.training(x_xor_processed,Yxor,100)
         print("Accuracy: " + str(P2.accuracy[99]*100) + "%")

         plt.scatter(blue_xor[:,0],blue_xor[:,1])
         plt.scatter(red_xor[:,0],red_xor[:,1])

         plt.xlim(-0.5, 1.5)
         plt.ylim(-0.5, 1.5)

Accuracy: 100.0%
Out[73]: (-0.5, 1.5)
```

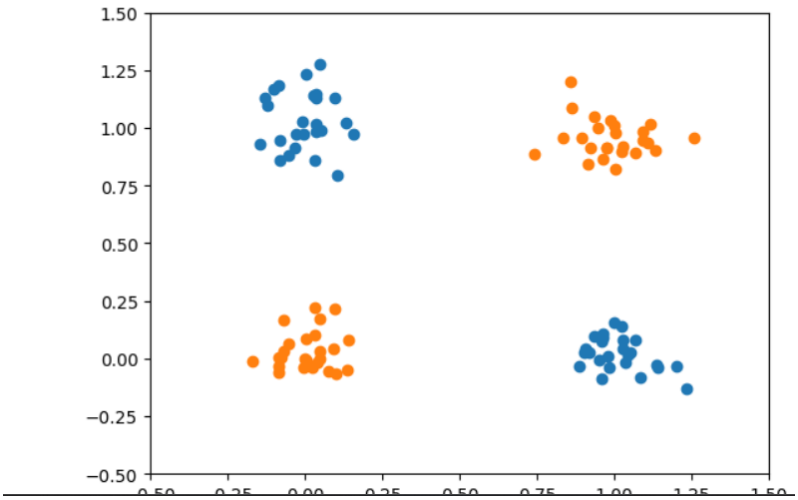

Figura 7. Xor data

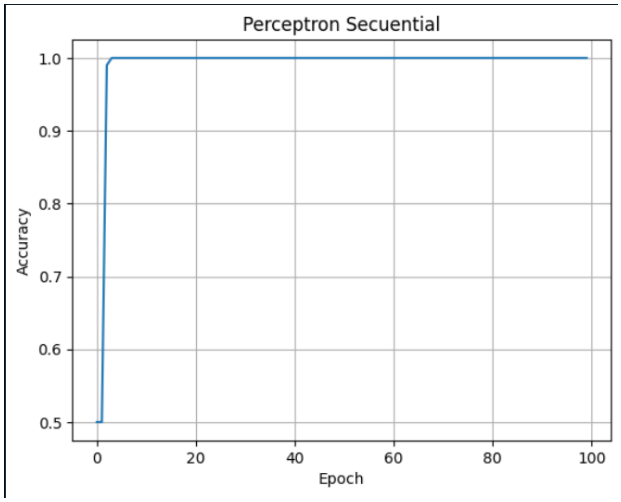

Figura 8. Precisión Xor perceptrón

Vemos que, para esta implementación, la velocidad de convergencia es sumamente alta ya que solo se maneja una entrada que se busca la variable ‘a’ que divida ambas clasificaciones. Se vuelve un problema trivial para el perceptrón.

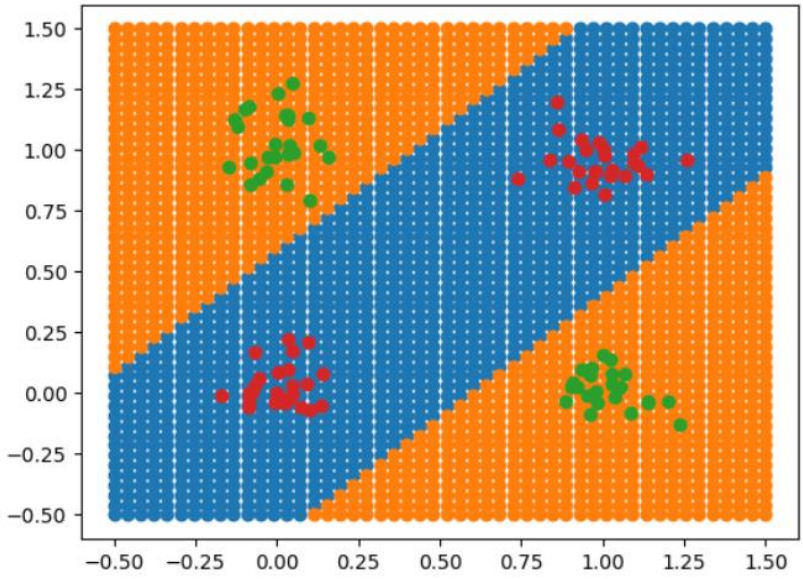

Figura 9. Frontera de decisión Xor

Finalmente, se implementa el algoritmo secuencial para el set de datos no lineales. Igualmente se busca procesar las coordenadas de los datos de entrada para que la clasificación pueda hacerse de forma lineal y así hacer posible que un único perceptrón pueda lograr este objetivo. Para esto se utilizó la siguiente ecuación:

$$r^2 = x^2 + y^2$$

De esta manera, el perceptrón solo debe encontrar una frontera entre los datos del radio con en el que se encuentran con respecto al centro del diagrama. A continuación, se presenta los resultados.

```
In [79]: P3 = perceptron(1,0.1,sgn_negativo)
         P3.training(x_nonlinear_processed,Ynonlinear,100)
         print("Accuracy: " + str(P3.accuracy[99]*100) + "%")
         plt.scatter(blue_nolinear[:,0],blue_nolinear[:,1])
         plt.scatter(red_nolinear[:,0],red_nolinear[:,1])


         plt.xlim(-3.5, 3.5)
         plt.ylim(-3.5, 3.5)

Accuracy: 100.0%
Out[79]: (-3.5, 3.5)
```

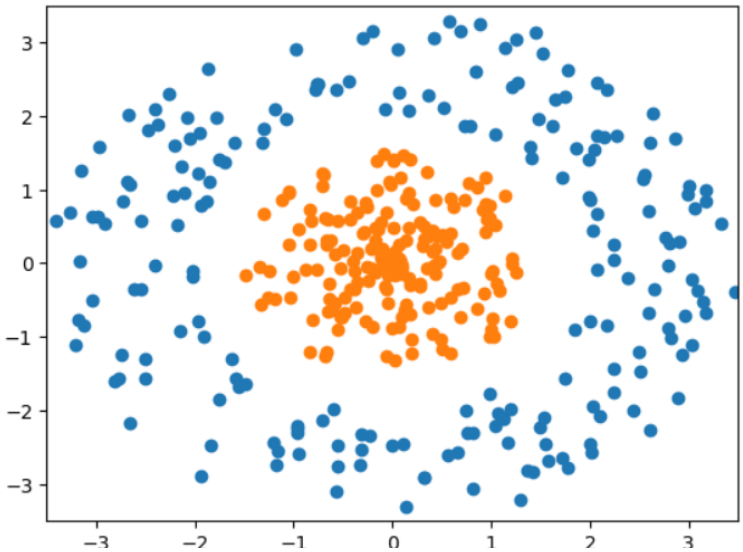

Figure 10. Data set no lineal

Se logra un ajuste de datos del 100% que asimismo como en el dataset anterior, ya que se vuelve un problema trivial que el perceptrón resuelve de forma inmediata.

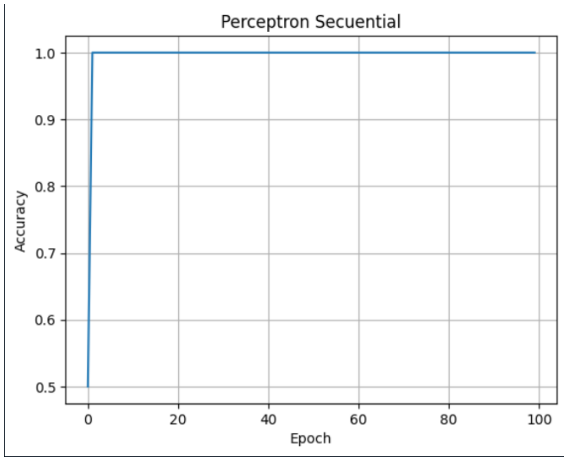

Figure 11. Precisión de entrenamiento no lineal

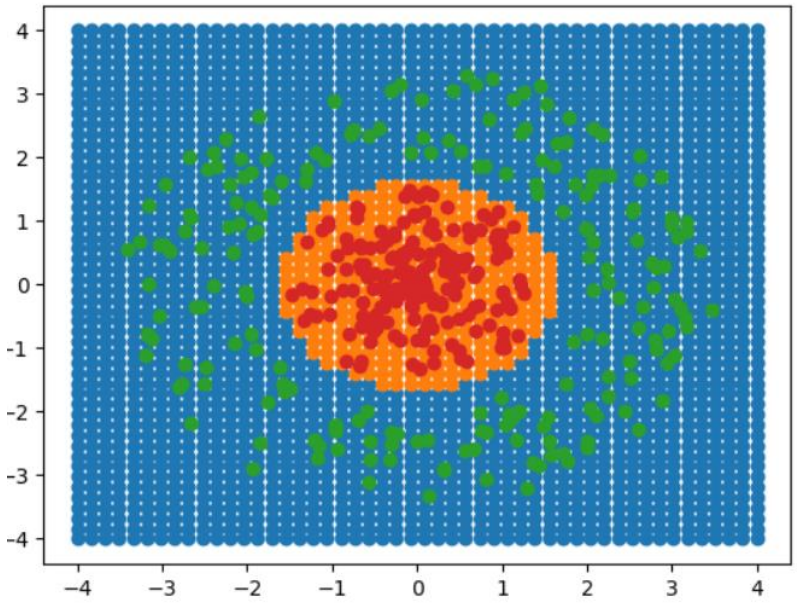

Figure 12. Frontera no lineal

2. **Implementación Algoritmo Batch.**
    - **Implemente el algoritmo de aprendizaje batch del perceptrón y corra el mismo por 100 épocas. Inicializando los pesos de forma aleatoria para los tres datasets. En todos los casos enfóquese en medir el accuracy.**
    - **Para los tres datasets recopile el accuracy por época y genere gráficos de accuracy versus época.**
    - **Para todos los datasets dibuje el decision boundary (es decir, el hiperplano que divide los datos entre categoría +1 y -1.)**

A continuación, se presenta el pseudocódigo del algoritmo tipo Batch.

For each epoch:

    For each event in X:

        Yi = perceptron(xi,w)

    For each y in Y:

$$Wnew = Wold + \eta(Yi-Y')x$$

La ventaja de este algoritmo es que se puede hacer la predicción de todo el dataset que se desea separada de la actualización de los pesos. Por lo tanto, se separan estos dos procesos que lo hace fácil de implementar para sets de datos sumamente grandes. A continuación, se presenta la clase creada.

In [41]:

```
class perceptron_batch(object):
    def __init__(self,size,learning_rate, function):
        self.weights = np.random.randn(size+1)
        self.learning_rate = learning_rate
        self.accuracy = []
        self.activation = function

    def getWeights(self):
        return self.weights

    def getAccuracy(self,X,Y):
        predicciones = []
        for xi in X:
            predicciones.append(self.predict(xi))
        temp = 0
        for i in range(len(predicciones)):
            if predicciones[i] == Y[i]:
                temp += 1
        return temp/len(predicciones)

    def predict(self, data):
        V = np.dot(data,self.weights[1:]) + self.weights[0]
        return self.activation(V)

    def updates_weights(self,error,xi):
        self.weights[1:] += self.learning_rate*error*xi
        self.weights[0] += self.learning_rate*error

    def training(self,X,Y,epochs):
        for i in range(epochs):
            errors = []
            for xi,yi in zip(X,Y):
                errors.append(yi - self.predict(xi))
            for error,xi in zip(errors,X):
                self.updates_weights(error,xi)
            self.accuracy.append(self.getAccuracy(X,Y))
```

Figure 13. Clase perceptrón batch

Se obtiene los siguientes resultados.

- Lineal

```
P1_batch = perceptron_batch(2,0.1,sgn_function)
P1_batch.training(X_linear,Y_linear,100)
print("Accuracy: " + str(P1_batch.accuracy[99]*100) + "%")

plt.scatter(blue_linear[:,0],blue_linear[:,1])
plt.scatter(red_linear[:,0],red_linear[:,1])

plt.xlim(0, 5)
plt.ylim(0, 5)
```

```
ccuracy: 99.5%
(0.0, 5.0)
```

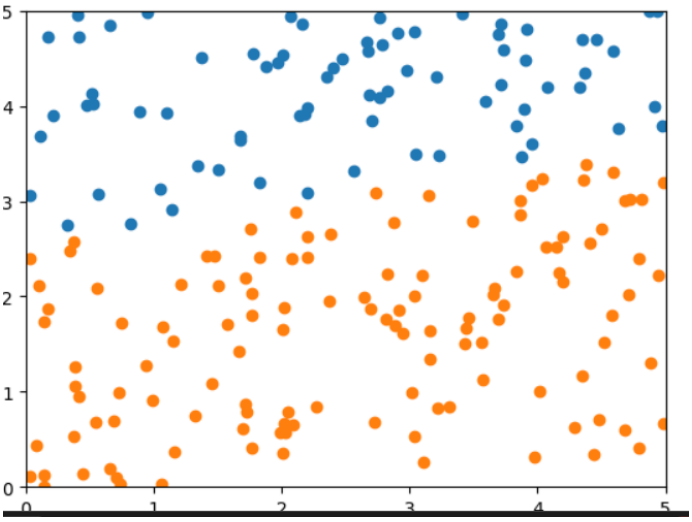

Figura 14. Data set lineal batch.

Se puede ver que el modelo no logra durante las épocas una precisión del 100%.

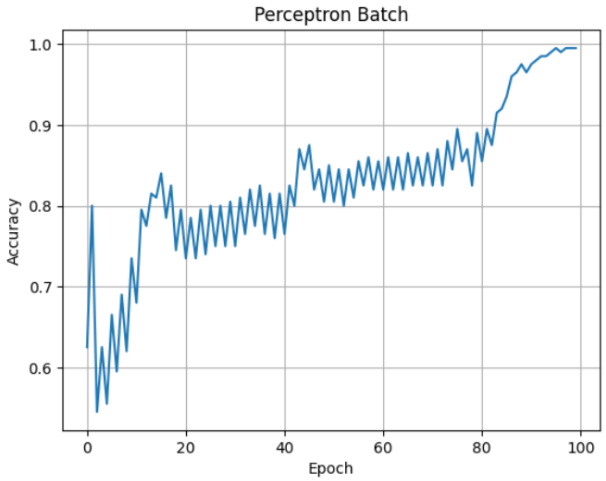

Figura 15. Entrenamiento batch lineal

Vemos que el ajuste de precisión es errático hasta llegar a un punto en el que el modelo encuentra el camino por donde va a lograr su convergencia total. Se aprecia que pasada la época 80, el algoritmo deja de ser errático y va a converger al 100% de precisión.

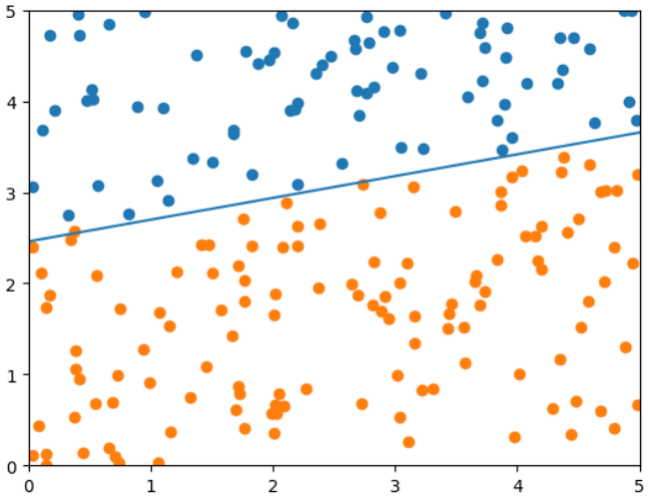

Figura 16. Frontera lineal tipo batch

Se aprecia que los datos azules están separados correctamente, sin embargo, un dato naranja se encuentra en la otra frontera. Por ello la precisión no es del 100%.

- Xor

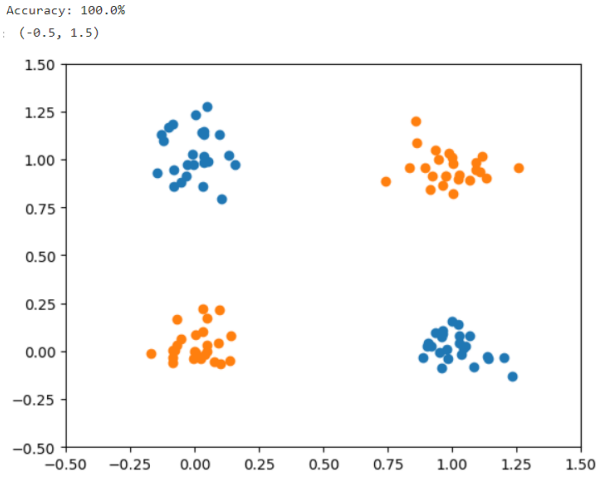

Figura 17. Xor Batch

La precisión del modelo es del 100%.

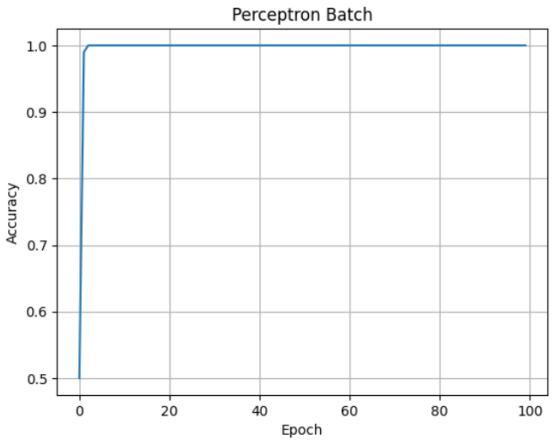

Figura 18. Convergencia Xor batch

Vemos que el set de datos xor con la manipulación de los datos se vuelve un problema trivial.

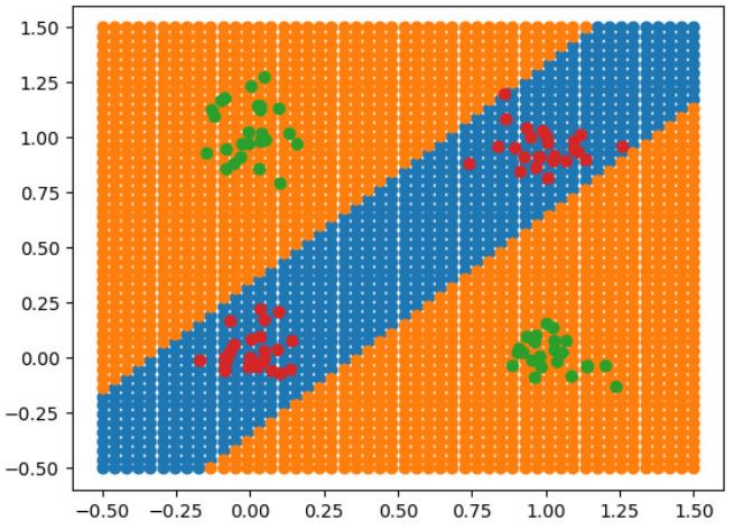

Figure 19. Frontera de Xor batch

Se puede apreciar que la frontera de decisión logra separar ambas clasificaciones de manera correcta.

- No lineal

```
In [89]: P3_batch = perceptron_batch(1,0.1,sgn_negativo)
         P3_batch.training(x_nonlinear_processed,Ynonlinear,100)
         print("Accuracy: " + str(P3_batch.accuracy[99]*100) + "%")
         plt.scatter(blue_nolinear[:,0],blue_nolinear[:,1])
         plt.scatter(red_nolinear[:,0],red_nolinear[:,1])


         plt.xlim(-3.5, 3.5)
         plt.ylim(-3.5, 3.5)

Accuracy: 100.0%
Out[89]: (-3.5, 3.5)
```

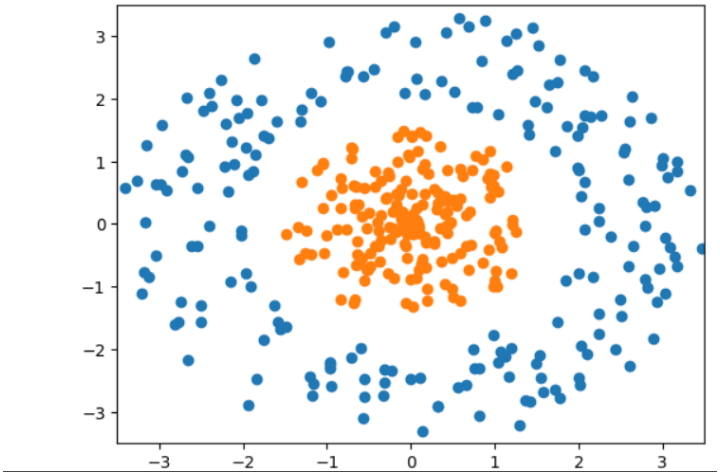

Figure 20. Entrenamiento batch no lineal

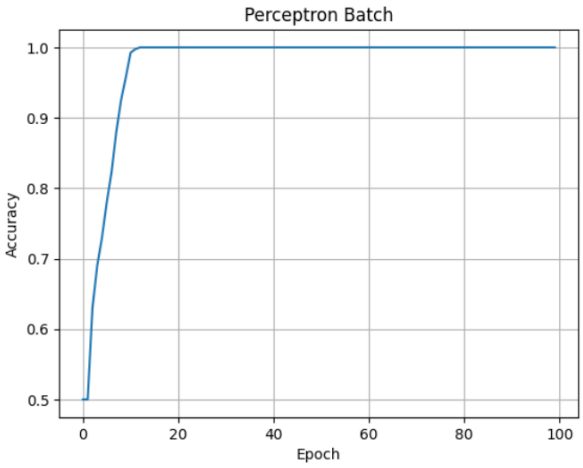

Figure 21. Precisión de entrenamiento no lineal batch

Igualmente, vemos que el algoritmo converge al 100% de precisión.

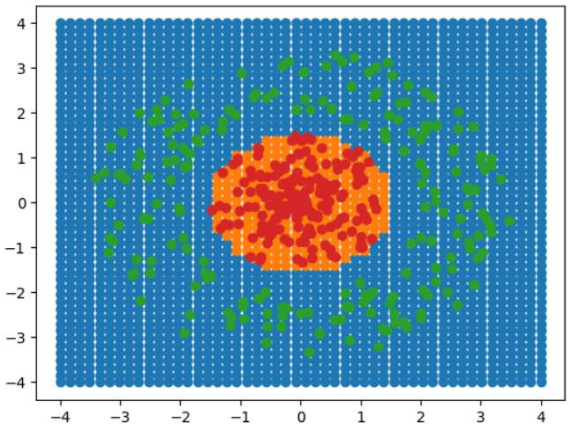

Figure 22. Frontera de decisión no lineal batch

3. **Comparar las soluciones:**

- **Luego de la implementación de las secciones 1 y 2, compare los resultados obtenidos y presente sus conclusiones.**

A continuación, se compara el proceso de entrenamiento con respecto a las épocas de iteración que presenta el perceptrón tipo batch y tipo secuencial para las 3 tablas de datos.

- **Data lineal**

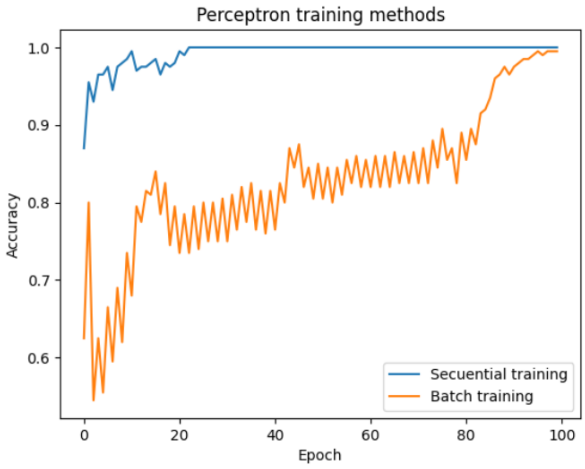

Figure 23. Comparación data lineal

Se puede concluir algunas cosas.

- Escoger una inicialización randómica de los pesos del perceptrón hace que ambos modelos tengan diferente inicio de la exactitud del modelo.
- Podemos observar que el algoritmo Batch presenta un comportamiento errático durante su entrenamiento con el paso de las épocas hasta que encuentra un camino de convergencia.
- El algoritmo Secuencial logra converger a un modelo 100% preciso de manera más rápida que el algoritmo Batch con el trade off de que si se usa gran cantidad de datos, no se puede parar el proceso de actualización. De esta forma el algoritmo Batch puede ser entrenado en una GPU.
- **Data XOR**

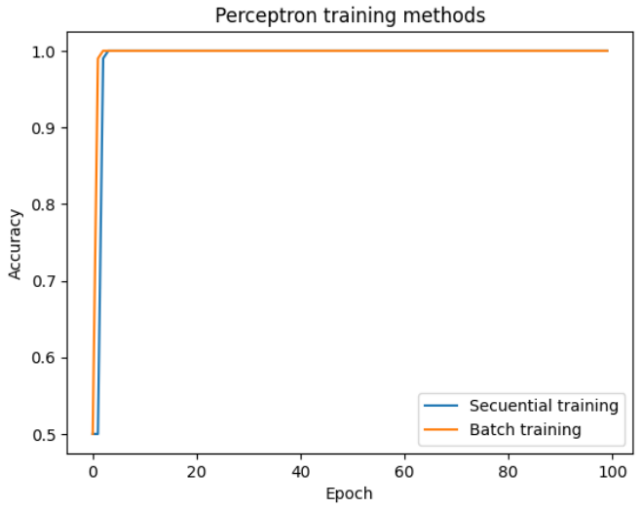

Figura 24. Comparación data Xor

En este caso se concluye que ambos tipos de entrenamiento dan una respuesta similar en tiempo. El único cambio se debe a la aleatoriedad inicial.

- **Data no lineal**

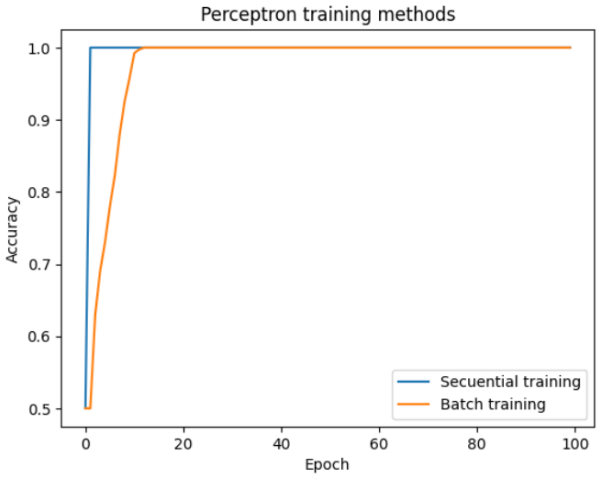

Figure 25. Comparación no lineal

Para este caso, la data no lineal afecta en el tiempo de convergencia del algoritmo batch ya que no se está actualizando en cada predicción los pesos del perceptrón. De esta forma su convergencia es más lenta.

Referencias:

Notas en clase.