FACULTAD DE ARTES
UNIVERSIDAD DE CHILE

Informe Desafío Perceptron

Daniel Valdés González
CEE Machine Learning
Profesor Rodolfo Lobo
Diciembre 2023

Construye tu propia clase `Perceptron` utilizando la regla antigua de actualización de pesos dada por:

$$\begin{aligned}\theta^{k+1} &= \theta_k + \Delta\theta_k \\ \theta_0^{k+1} &= \theta_0^k + \Delta\theta_0^k \\ \Delta\theta &= \alpha \cdot (y_i - \hat{y}_i) \cdot x_i \\ \Delta\theta_0 &= \alpha \cdot (y_i - \hat{y}_i)\end{aligned}$$

$\alpha \in [0, 1]$ es el learning rate, θ_0 se actualiza aparte del resto de los pesos (también conocido por bias). Tu clase debe contener al menos los siguientes métodos:

```python
class Perceptron():
    def __init__(self, num_features):
        pass
    def forward(self, x):
        pass
    def backward(self, x, y):
        pass
    def train(self, x, y, epochs):
        pass
    def evaluate(self, x, y):
```

Luego, entrena el modelo con datos de baja dimensión, la estrategia será convertir audios a una representación bi-dimensional. En el archivo `t-SNE_audio.ipynb` se encuentra la estrategia para convertir audios a vectores en dos dimensiones, esta parte será la **caja negra** del proceso, es decir, asumiremos que convierte el audio a una representación de 2 dimensiones. Puedes utilizar la misma estrategia vista en el notebook de la clase.

1. Entrena el modelo.
2. Evalúa la performance tanto en el conjunto de test como de entrenamiento.
3. Elije alguno de los modelos vistos en clases, entrénalo con los mismos datos y compara la performance con el perceptrón.

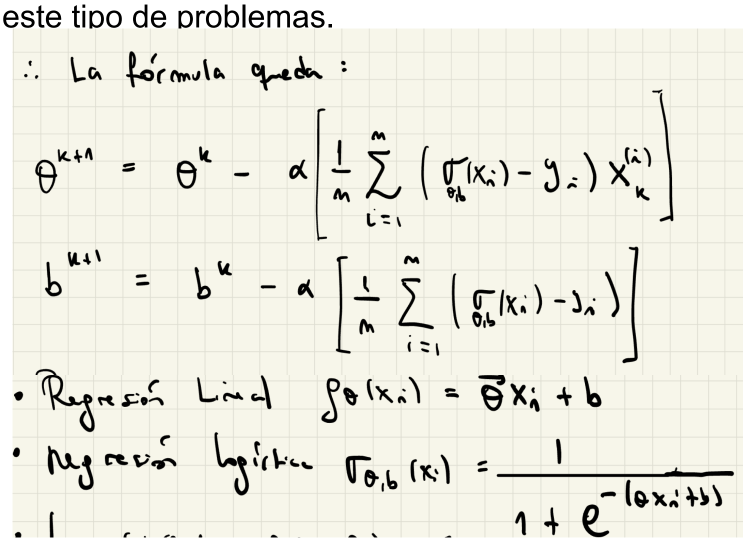
Fig. 1- Desafío

Si observamos la ecuación que se nos muestra en el desafío:

$$\begin{aligned}\theta^{k+1} &= \theta_k + \Delta\theta_k \\ \theta_0^{k+1} &= \theta_0^k + \Delta\theta_0^k \\ \Delta\theta &= \alpha \cdot (y_i - \hat{y}_i) \cdot x_i \\ \Delta\theta_0 &= \alpha \cdot (y_i - \hat{y}_i)\end{aligned}$$

Fig. 1.2- Zoom in Desafío

Podemos relacionar esta fórmula con la vista durante la clase 5, sobre regresión logística, donde se implementa una función sigmoide, que entrega valores entre 0 y 1. Esto resulta útil al enfrentarnos ante problemas de clasificación binarios, es decir de 2 categorías (0 y 1, por ejemplo); por lo que la regresión logística es el modelo más apropiado para este tipo de problemas.



∴ La fórmula queda :

$$\theta^{k+1} = \theta^{k} - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\left(\sigma_{\theta,b}(x_i) - y_i\right)x_k^{(i)}\right]$$

$$b^{k+1} = b^{k} - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\left(\sigma_{\theta,b}(x_i) - y_i\right)\right]$$

- Regresión Lineal $f_\theta(x_i) = \vec{\theta} x_i + b$
- Regresión logística $\sigma_{\theta,b}(x_i) = \dfrac{1}{1 + e^{-(\theta x_i + b)}}$

Fig. 3. Apuntes Profesor Clase 5

El objetivo de este modelo es encontrar los parámetros que minimizan la función de costo, para esto ocupamos gradiente descendiente.

El presente trabajo incluye una revisión de los módulos principales que fui implementando en las 3 clases de perceptrón que desarrollé; a partir de referencias disponibles en el repositorio de github del curso, consultas realizadas a chatgpt e ideas que se me fueron ocurriendo.

Como se ve en la siguiente imágen, los primeros módulos son en gran parte el archivo adjunto que se indica en el desafío para realizar la clase:

Parte 1 (de t-SNE.ipynb)

1. convertir audio a vector
2. reducir dimensiones

Esta parte del código esta tal cual el archivo según se indica en el trabajo. Puede que haya añadido algunas líneas de código para yo mismo entender lo que estaba pasando y asegurar un correcto funcionamiento

```
[1]  !pip install transformers         #instalamos biblioteca transformers

[18] import IPython.display as ipd
     from IPython.display import Audio
     import librosa
     import librosa.display
     import numpy as np
     import matplotlib.pyplot as plt
     import pathlib
     from pathlib import Path
     import sklearn
     from transformers import AutoFeatureExtractor #generador de características
     import os
     import seaborn as sns

[3]  !ls #revisamos listado del directorio actual

[4]  os.chdir('drive/MyDrive/Machine Learning/perceptron_dataset') #cambiamos el directorio

[5]  MyPath = pathlib.Path("/content/drive/MyDrive/Machine Learning/perceptron_dataset")
     if MyPath.exists() and MyPath.is_dir():
         print("La ruta es válida y el directorio existe.")
     else:
         print("La ruta no es válida o el directorio no existe.")
```

Fig. 4.

```
[6]   sr = 16000   #frecuencia de muestreo
```

Cargando audios

```
[7]  kick_signals = [                      #cargamos los archivos de audio y los discretizamos(?)
         librosa.load(p, sr = sr)[0] for p in MyPath.glob('kick/Copia de Bass Sample *.wav')
     ]

     snare_signals = [
         librosa.load(p, sr = sr)[0] for p in MyPath.glob('snare/Copia de Snare Sample *.wav')
     ]

[8]  kick_signals[0].shape                      #comprobando

[9]  len(kick_signals) #chekeando que se hayan exportado los 40 audios

[10] audio = Audio(data = kick_signals[0],rate = sr) #[0-39] para poder escuchar los archivos
     display(audio)
```

caja negra

```
[11] feature_extractor = AutoFeatureExtractor.from_pretrained("facebook/wav2vec2-base")

[12] snare_inputs = feature_extractor(snare_signals, sampling_rate=sr, return_tensors="pt")

[14] snare_inputs["input_values"].shape            #verificamos que los audios se hayan cargado correctamente

[15] kick_inputs = feature_extractor(kick_signals, sampling_rate=sr, return_tensors="pt")
```

convierte todos los audios en una matriz que representa el conjunto de audios. la fila representa la cantidad de muestras (40 kick, 40 snare) y la cantidad de columnas tiene relación con la duración (d, en segundos) de cada audio; es decir (sr*d)

Fig. 5.

```
[21] import torch

[22] X = torch.cat((kick_embeddings, snare_embeddings), 0) #se concatenan ambos vectores para formar el conjunto de entrada

[23] X.shape      #corroboramos las dimensiones
```

2. APLICAMOS t-SNE

2. para reducir dimensiones

```
[24] from sklearn import manifold

     tsne = manifold.TSNE(n_components = 2, random_state = 42)

[26] transformed_data = tsne.fit_transform(X)

[27] transformed_data.shape[1]

[28] X.shape

[29] import pandas as pd

[30] df_tsne = pd.DataFrame(transformed_data)
     df_tsne['targets'] = 40*[1] + 40*[0]

[31] df_tsne['targets']

[32] df_tsne.columns = ['x1','x2','targets']

[33] df_tsne
```

Fig. 6.

	x1	x2	targets
0	6.085392	1.840192	1
1	5.218927	1.306673	1
2	-0.585032	-3.000730	1
3	3.585593	-1.677265	1
4	5.100288	1.647039	1
...	...	...	...
75	-3.322230	1.382697	0
76	-4.137670	-0.354485	0
77	-2.077505	0.394700	0
78	-2.602193	1.723334	0
79	-1.699998	2.015516	0

80 rows × 3 columns

Fig. 7.

```
grid = sns.FacetGrid(df_tsne,hue = "targets")
grid.map(plt.scatter,"x1","x2").add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x7d5fbacc8b80>

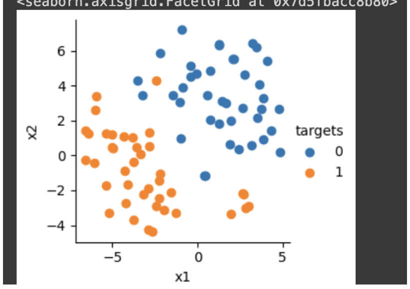


Fig. 8.

Implementación de las clases

Intenté entrenar el modelo implementando la función sigmoide en la predicción, sin embargo, tenía problemas graficando la frontera de percepción (posteriormente lo resolví). En reemplazo utilice una función de paso simple (que entrega 1 si x mayor a 0, y entrega 0 en cualquier otro caso). Esta función la implementé de dos maneras distintas. Finalmente, las tres clases de Perceptrón que incluí son similares en su definición e implementación; las principales diferencias se encuentran en las siguientes líneas de código:

```
def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

def predict(self, X):
    return self.sigmoid(np.dot(X, self.weights) + self.bias)
```

Fig. 9. Función sigmoide implementada en el primer Perceptrón

```
def _step_function(self, x):
    # Función de paso simple: devuelve 1 si x es mayor o igual a cero, 0 de lo contrario
    return 1 if x >= 0 else 0

def _predict(self, X):
    # Predice la clase para cada entrada en X usando la función de paso
    return np.array([self._step_function(np.dot(self.weights, x) + self.bias) for x in X])
```

Fig. 10. Función de paso implementada en el segundo Perceptrón

```
def predict(self, X):
    return 1 if (np.dot(X, self.weights) + self.bias) > 0 else 0
```

Fig. 11. Función de paso implementada en el tercer Perceptrón

A continuación se observan la gráfica de los resultados y las fronteras de decisión obtenidas en el entrenamiento de cada clase de perceptrón, con el tag 1 = Kick (bombo) y el tag 0 = Snare(caja):

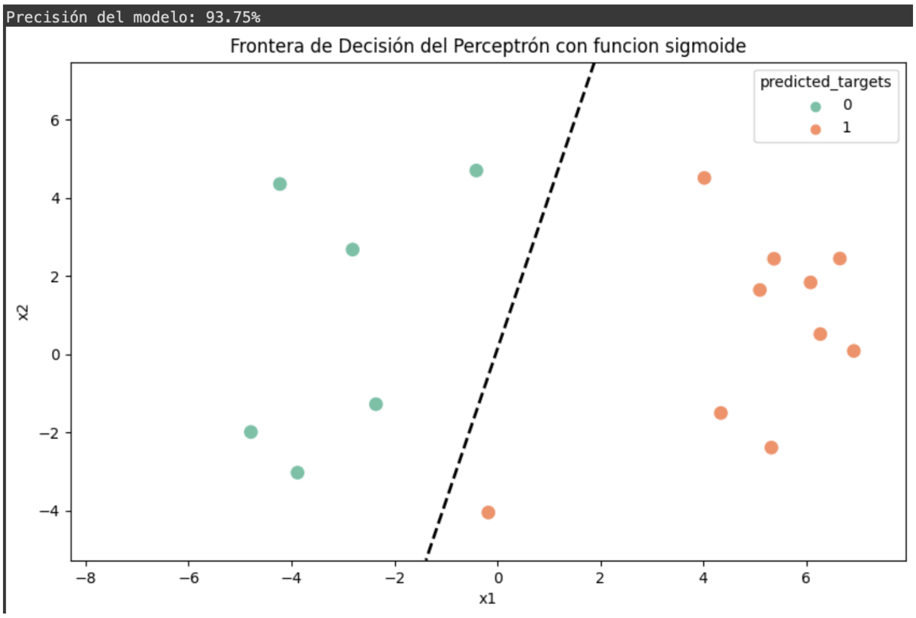

Precisión del modelo: 93.75%
Frontera de Decisión del Perceptrón con funcion sigmoide
predicted_targets
0
1
x2
x1

Fig. 12.

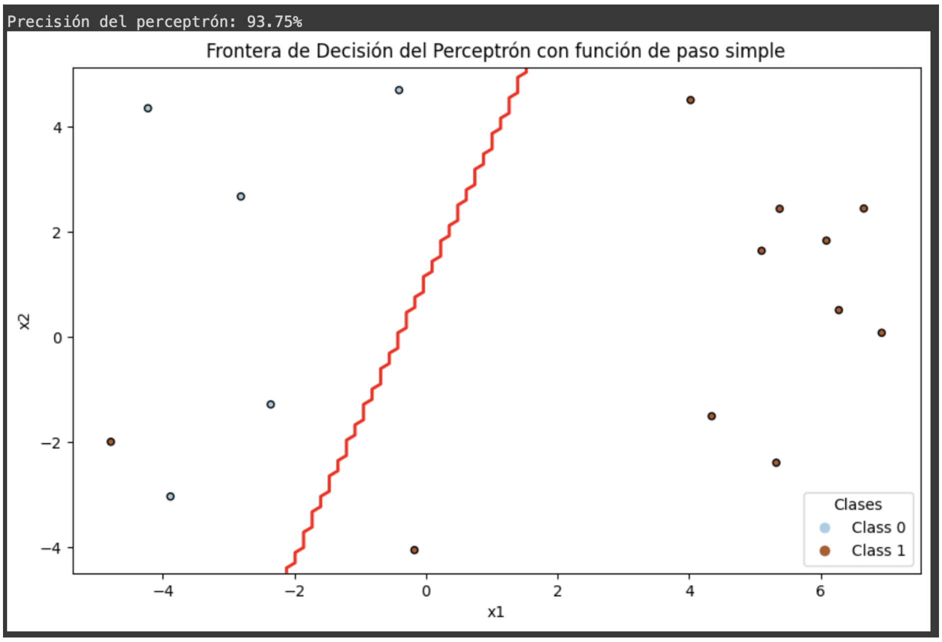

Precisión del perceptrón: 93.75%
Frontera de Decisión del Perceptrón con función de paso simple
Clases
Class 0
Class 1
x2
x1

Fig. 13.

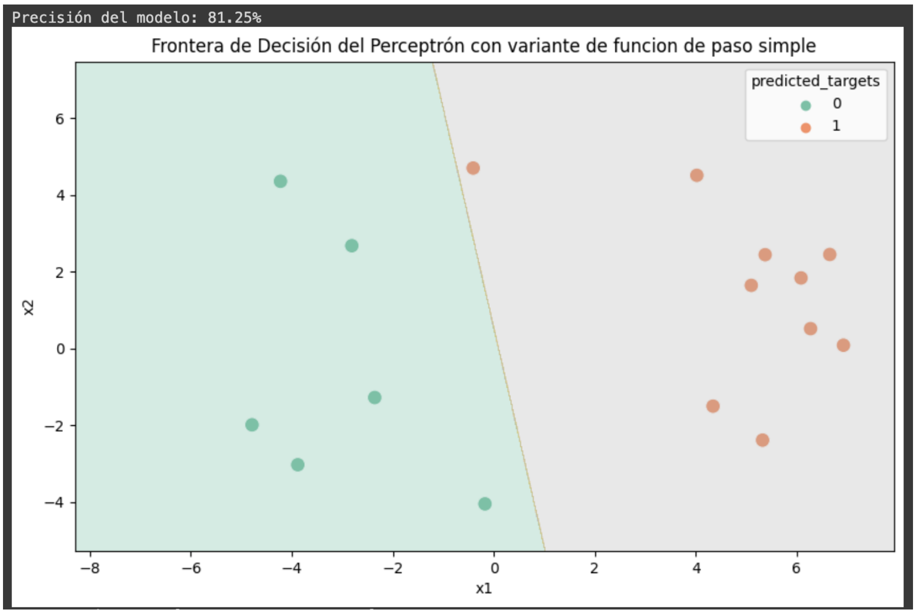

Precisión del modelo: 81.25%
Frontera de Decisión del Perceptrón con variante de funcion de paso simple
predicted_targets
0
1
x1
x2
−8
−6
−4
−2
0
2
4
6
6
4
2
0
−2
−4

Fig. 14.