Universidad de Chile

FACULTAD DE ARTES

Clasificación de Kick y Snare con modelo de un Perceptron

Nehemías Rivera

Introducción al Machine-Line, con aplicaciones al audio

Prof. Rodolfo Lobos

11/12/2023

En este informe se presenta el proceso de diseño y los resultados de un modelo de perceptrón diseñado con torch, cuya finalidad es la de clasificar entre audios de Kick y Snare. Se comentarán los resultados y se compararán con un modelo de red neuronal clasificador perceptrón multicapa visto en clases.

Instrucciones iniciales y modo de empleo del modelo:

Según la instrucción, se debe diseñar un modelo de perceptrón regido por la regla antigua de actualización de pesos.

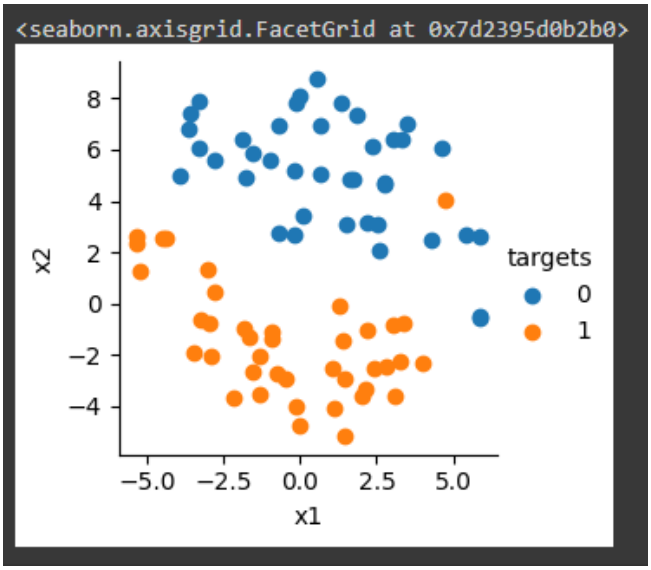
$$\theta^{k+1} = \theta_k + \Delta\theta_k$$
$$\theta_0^{k+1} = \theta_0^k + \Delta\theta_0^k$$
$$\Delta\theta = \alpha \cdot (y_i - \hat{y}_i) \cdot x_i$$
$$\Delta\theta_0 = \alpha \cdot (y_i - \hat{y}_i)$$

El código diseñado cuenta con tres etapas, una de extracción de características y de reducción de dimensionalidad, otra para el modelo perceptrón, que contiene la clase, y el entrenamiento, y una última que contiene el modelo de red neuronal multicapa.

Le etapa de extracción de características se basa en el modelo dado por el Prof. Rodolfo Lobos, el cual es un código que utiliza el modelo "wav2vec2-base" para la extracción de características, y el algoritmo T-SNE para reducir la dimensionalidad de los datos extraídos a dos dimensiones. De esta forma queda como datos de entrada para el modelo la siguiente tabla y el siguiente gráfico:

	x1	x2	targets
0	-0.739233	-2.731863	1
1	-1.672245	-1.318835	1
2	-1.321675	-3.504952	1
3	-0.903052	-1.380378	1
4	2.788563	-2.423316	1
...	...	...	...
75	3.472444	6.994761	0
76	3.293786	6.370447	0
77	3.049869	6.415592	0
78	1.850437	7.335338	0
79	1.318837	7.841558	0

80 rows × 3 columns



La etapa del modelo del perceptrón se basa en la siguiente estructura:

```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):
```

El modo en que llamamos los métodos de la clase es el siguiente:

```python
perceptron = Perceptron()

# Entrenamos el perceptrón
alpha = 0.01 #Learning rate
epochs = 15
repeat= 1    #Cada cuanto veces graficamos
temporal_learn = [] #Lista donde almacenamos la presicion del modelo en distintas epocas

for i in range(epochs):

  epoch = int(i+1)

  #Entrenamos
  perceptron.train(X_train, y_train, epoch, alpha)

  #Evaluamos el perceptrón
  test_accuracy = perceptron.evaluate(X_test, y_test)

  temporal_learn.append(test_accuracy[0])

  if epoch % repeat == 0:

      print(str(epoch)+ " " + "epochs done!") # visualizamos los resultados

      #Graficamos
      import matplotlib.pyplot as plt

      plt.scatter(X[:,0], X[:,1], c=y)

      #Graficamos la frontera de decisión
      x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
      x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1

      xx1, xx2 = torch.meshgrid(torch.arange(x1_min, x1_max, 0.1), torch.arange(x2_min, x2_max, 0.1))

      xx = torch.transpose(torch.stack([xx1.flatten(), xx2.flatten()]), 0, 1)

      Z = np.array([perceptron.forward(xi) for xi in xx])

      Z = Z.reshape(xx1.shape)

      plt.contourf(xx1, xx2, Z, alpha=0.3, cmap='Set2')

      plt.title(f"Frontera de desición, epoca: {epoch}")

      plt.show()
```

Inicialmente instanciamos nuestra clase *Perceptron*. Luego definimos los parámetros del modelo, como la tasa de aprendizaje y el número de iteraciones que realizará el entrenamiento. Se define una lista llamada *temporal_learn_rate* que almacenará los valores de *accuracy* según la época de iteración. Posteriormente se usan los métodos de entrenamiento y evaluación dentro de un ciclo *for*, donde en cada iteración se realiza un entrenamiento del modelo, y se almacena el *accuracy* en una lista. Luego se grafican los datos de entrada junto a la frontera de decisión, de tal forma que cada tantas veces (*repeat*) se visualiza el gráfico. De esta forma entrenamos al modelo, evaluamos su comportamiento, y vemos cómo evoluciona la representación gráfica de la recta del hiperplano del modelo según la época de iteración. En general, para la revisión de resultados de este modelo se utilizará una tasa de aprendizaje de 0,01.

Diseño de perceptrón:

La clase del perceptrón comienza de la siguiente forma.

```
class Perceptron(torch.nn.Module):

    def __init__(self):
        super().__init__()

        #Definimos parametros para perceptron
        self.weights = torch.zeros((2))
        self.weights_1 = torch.rand(())
        self.weights_0 = torch.rand(())
        self.bias = torch.zeros(())
```

Aquí inicializamos los parámetros de peso y sesgo del modelo. Los pesos de inicializan por separado para actualizarlos de forma independientes. Luego en el cálculo de la predicción se actualiza la matriz *self.weights* con los pesos entrenados tal como se muestra a continuación.

```
def forward(self, x):

    #Calculamos modelo de perceptron (W*x)

    self.weights[0] = self.weights_0
    self.weights[1] = self.weights_1

    z = torch.add(torch.dot(self.weights, x),self.bias) #w_0*x_1 + w_1*x_2 + b

    #Función de activación
    if z > 0:
            y_pred = torch.ones(())
    if z <= 0:
            y_pred = torch.zeros(())

    return y_pred
```

En este método se hace el cálculo de la predicción del modelo. Se actualizan los valores de la matriz de pesos, y se realiza el calculo de la predicción. z es el hiperplano $\omega_0 * x_1 + \omega_1 *$

$x_2 + \beta$, y la función de activación es la *función signo*, donde si z es mayor que 0, la predicción será 1, y si es menor, la predicción será 0. Posteriormente, la clase sigue de la siguiente forma.

```
def backward(self, x, y, alpha):

    #Actualización de parametros
    self.errors = torch.sub(y,self.forward(x)) #y - y_pred

    self.weights_1 = torch.add(self.weights_1, torch.mul(alpha, torch.mul(self.errors,x[1]))) # w + alpha*error*x_1
    self.weights_0 = torch.add(self.weights_0, torch.mul(alpha, torch.mul(self.errors,x[0]))) # w + alpha*error*x_0
    self.bias = torch.add(self.bias, torch.mul(alpha, self.errors)) #Bias

def train(self, X, Y, epochs, alpha):
    self.epochs = epochs
    #Entrenamiento
    for epoch in range(epochs):
      for x, y in zip(X, Y):
        self.backward(x, y, alpha)
```

Aquí se definen dos métodos, *backward* y *train*. El método *train* utiliza el método se usa en el método *backward* para calcular el error del modelo, y así actualizar los parámetros de pesos y sesgo con la regla dada inicialmente. El método *train* aplica cada dato del dataset entregado al método *backward* para actualizar los parámetros. Esto ocurre itera veces que sea especificado en la variable epochs, lo cual indica los ciclos de entrenamiento. La clase termina con los siguientes métodos.

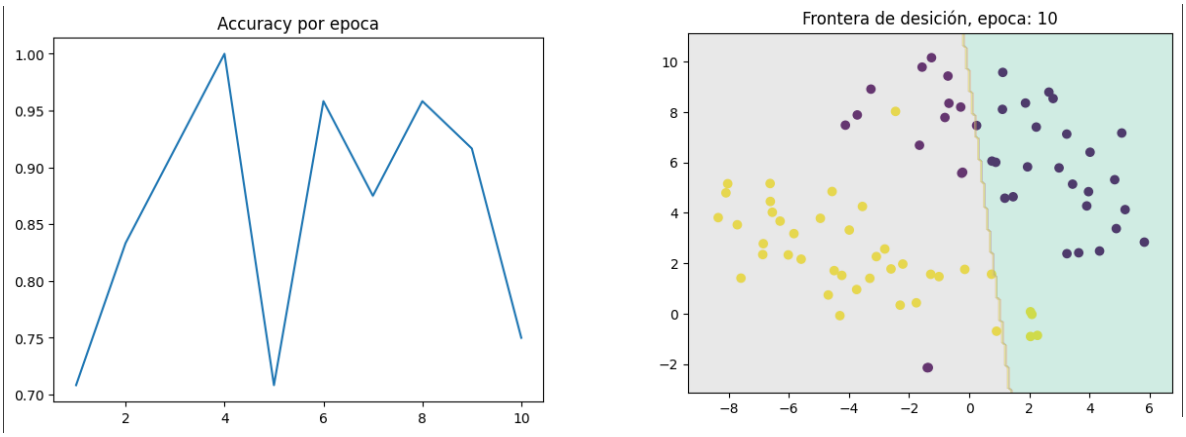
```
def evaluate(self, x, y):
    #Evaluamos modelo
    correct = 0
    total = 0
    for i in range(len(x)):
        output = self.forward(x[i])

        if output == y[i]:
            correct += 1
        total += 1

    return [correct / total, correct, total]
```

El método *evaluate* evalúa el *accuracy* del modelo entrenado comparando el valor de salida del modelo con los valores “target” de los datos de entrada. Y por último el método *print_* simplemente imprime datos de interés.

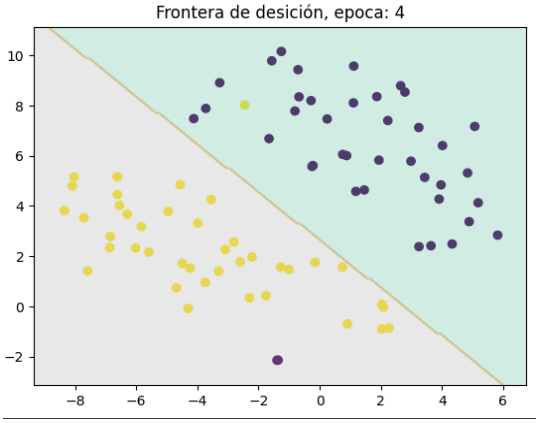
El modelo de red neuronal multicapa utiliza el método MLPClassifier de la librería sklearn, el cual es un modelo de perceptrón multicapa para clasificar datos. Al final de este modelo se grafica la matriz de confusión que muestra los resultados de este.

Resultados:

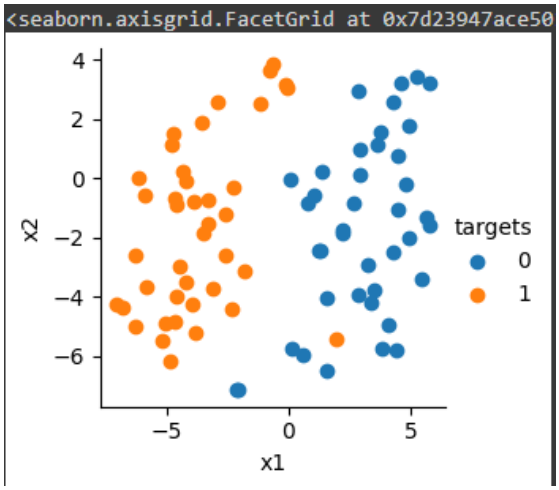
El modelo entrenado con un dataset de entrenamiento de 56 datos ha mostrado los siguientes resultados para un total de 10 épocas de iteración.



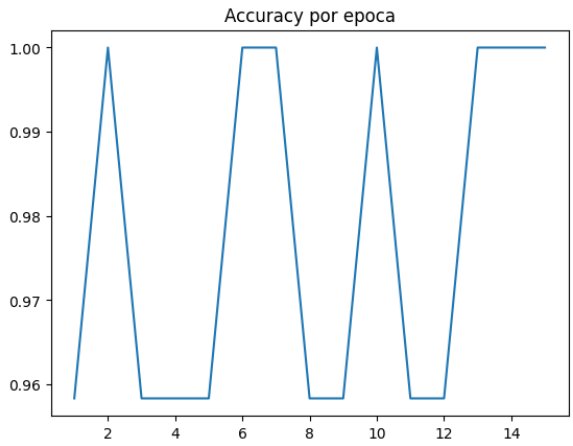
Se puede observar que el modelo funciona mejor cuando el ciclo de entrenamiento es de 4 veces, teniendo en ese punto una precisión de 1.

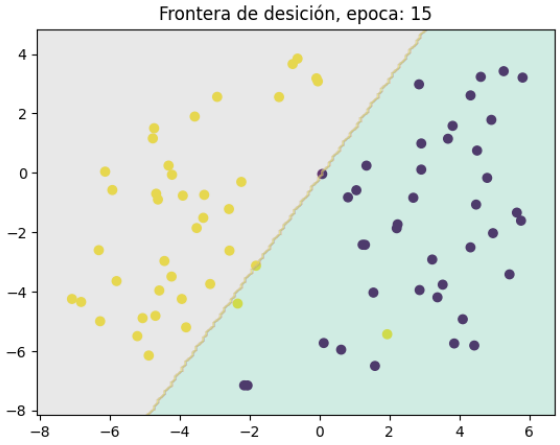


Podemos probar con otra forma de distribución de los datos como la siguiente y ver su comportamiento. Veamos el siguiente gráfico.



Se puede apreciar que esta distribución tiene una mayor pendiente. Veamos el comportamiento del modelo con 15 iteraciones.





Vemos que, para esta ocasión, que si bien los resultados de *accuracy* son muy aproximados a 1, 4 ciclos de entrenamiento no brindad el mejor resultado.

Comparación con modelo de red neuronal:

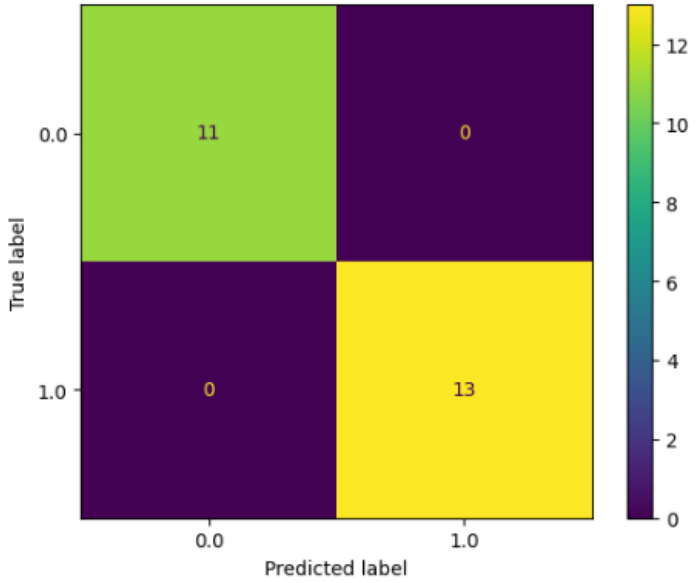
El modelo de clasificación binaria utilizado para comparar nuestro perceptrón es un modelo de red neuronal, que se basa en un perceptrón multicapa entregado por el Prof. Rodolfo Lobos. Este modelo tiene 5000 capas y 10 neuronas, y utiliza la técnica de entrenamiento de gradiente descendente estocástico.

La clase del modelo comienza así.

```python
class MiPrimeraRed:

    def __init__(self, hidden_layer_sizes=(5000, 10), alpha=1e-5, solver='sgd', max_iter = 1000):
        self.hidden_layer_sizes = hidden_layer_sizes  #número de capas escondidas (5000 capas y 10 neuronas)
        self.alpha = alpha #learning rate
        self.solver = solver #optimizador, por ahora utilizaremos el que conocemos Gradiente Estocástico
        self.max_iter = max_iter

        ### INICIALIZAMOS EL PERCEPTRON MULTICAPA
        self.nn = MLPClassifier(solver= self.solver,
                                alpha=self.alpha,
                                hidden_layer_sizes = self.hidden_layer_sizes,
                                max_iter = self.max_iter ,
                                random_state=1) #aleatoriedad en la generacion de los pesos y el sesgo inciales

    def train_model(self, X_train, y_train):
        try:
            ### AQUI ENTRENAMOS EL MODELO
            self.nn.fit(X_train, y_train)


            print("Modelo entrenado exitosamente!")
        except Exception as e:
            print(f"Algo ha ocurrido al momento de entrenar: {e}")

    def measuring_model_accuracy(self, X_test, y_test):

        # confirmamos que el modelo ya fue entrenado para poder medir accuracy
        if hasattr(self.nn, 'coefs_'):
            preds = self.nn.predict(X_test)
            print('Accuracy', accuracy_score(y_test, preds), '\n')
            print("Matriz de Confusión")
            print(confusion_matrix(y_test, preds))
        else:
            print('El modelo no ha sido entrenado, no es posible medir accuracy')
```

Primeramente, se inicializan los parámetros para el método MLPClassifier, que son el número de capas y neuronas, la técnica de entrenamiento, tasa de aprendizaje, el número de ciclos de entrenamiento y la aleatoriedad de los pesos y sesgos. Luego el método *train* utiliza el método *.fit* entrena el modelo con los datos de entrenamiento utilizados en el modelo de Perceptrón anterior. Posteriormente sigue el método *measuring_model_accuracy* que mide la precisión del modelo y entrega la matriz de confusión de 2x2, ya que es clasificación binaria.

Una vez entrenado el modelo, evaluamos su comportamiento. Esta evaluación entrega una precisión de 1.0, y la siguiente matriz de confusión.



Comparación:

Podemos observar que tanto el modelo de Perceptrón diseñado, y el modelo de red neuronal obtienen resultados similares en primera instancia. Los dos son capaces de alcanzar una precisión de 1.0. Esto se debe meramente a que los datos de entrada son simples, y no requieren mayor de un modelo más sofisticado. Veamos más detalladamente los resultados.

	y_pred	targets
0	1.0	1.0
1	1.0	1.0
2	1.0	1.0
3	1.0	1.0
4	1.0	1.0
5	1.0	1.0
6	1.0	1.0
7	0.0	0.0
8	1.0	1.0
9	1.0	1.0
10	0.0	0.0
11	1.0	1.0
12	0.0	0.0
13	1.0	1.0
14	0.0	0.0
15	0.0	0.0
16	0.0	0.0
17	0.0	0.0
18	0.0	0.0
19	0.0	0.0
20	1.0	1.0
21	0.0	0.0
22	1.0	1.0
23	0.0	0.0

Estos son los resultados de predicción de nuestro perceptrón. Si nos tomamos el tiempo, vemos que predice perfectamente los valores para las entradas dadas. De esto podemos concluir que nuestro modelo de perceptrón funciona muy bien para predecir si un audio es Kick o Snare.