*

Tarea 4:
Clasificación de edad usando LBP y redes neuronales

Fecha de entrega: Domingo 31 de octubre, 23:59 hrs.

Estudiante: Francisco Molina L.
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla Z.
Semestre: Primavera 2021

Índice

1. Introducción

En el presente informe se desarrollan las actividades correspondientes a la Tarea 4 del curso EL7008-1 del semestre de primavera de 2021. Este informe comprende la implementación computacional de la extracción de características LBP, y del entrenamiento y análisis de redes neuronal que clasifican rostros de personas según su edad.

El principal objetivo de este informe es lograr implementar mediante programación el cálculo de las características LBP, lo que implica computar los patrones binarios locales y calcular histogramas en distintos cuadrantes de la imagen. Los clasificadores no se programan a bajo nivel, sino que simplemente se importan desde la librería `pytorch` y se analiza su desempeño cuando se entrenan con las características LBP programadas.

La **Sección 2** contiene un resumen del contexto necesario para entender las implementaciones realizadas en la tarea, esto incluye el cálculo de los LBP, cómo se obtiene el vector de características LBP y una explicación detrás del funcionamiento de las Redes Neuronales según [2]. En las **Secciones 3 y 4** se detallan las funciones implementadas en Python, adjuntas en el notebook Tarea4.ipynb, necesarias para calcular las características LBP, realizar el entrenamiento de los clasificadores y realizar las predicciones sobre los conjuntos de datos. En la **Sección 5** se presentan los resultados obtenidos de la implementación en Python, en la **Sección 6** se realiza un análisis de dichos resultados, destacando los casos interesantes, y en la **Sección 7** se presentan las conclusiones del informe.

2. Marco Teórico

2.1. Características LBP

El primer paso para realizar reconocimiento de clases de objetos es extraer las características de los objetos. Las características escogidas para esta tarea corresponden a los Patrones Binarios Locales (LBP), cuyo cálculo se describe a continuación.

En primer lugar, un patrón binario local es un descriptor visual que codifica la vecindad de cada pixel mediante comparaciones en la intensidad de los pixeles contra algún umbral. Si los pixeles de la vecindad son mayores o iguales al umbral se anota un 1 en su posición y si no se anota un 0, finalmente se "desenrolla" la cadena de valores en sentido horario para así formar un número binario, cuyo valor en base 10 se anota en la posición del pixel central, y se repite el procedimiento con el siguiente pixel.

En el caso más simple se considera una vecindad de tamaño (3x3), donde el umbral de comparación es el mismo pixel central, como se muestra en la **Figura 1**:

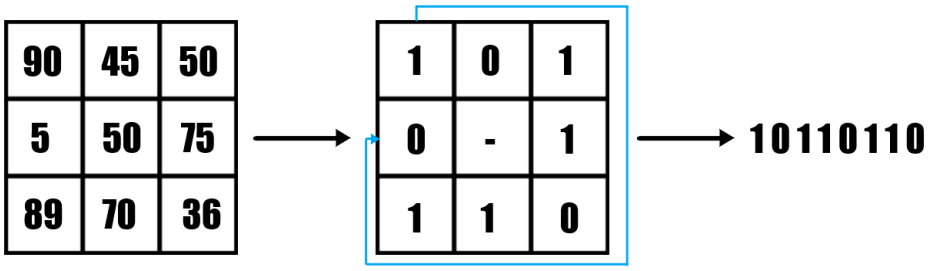

Figura 1: Ejemplo de LBP vecindad de 3x3

Existen variaciones del LBP, con distintos umbrales, tamaños de la vecindad e incluso formas. En particular en esta tarea se implementa un LBP circular, donde se toman `P` muestras de un círculo de radio `R` alrededor del pixel central.

Para calcular las coordenadas de los pixeles que corresponden a la vecindad se utilizan coordenadas polares:

$$\begin{aligned} x &= R \cdot cos(n \cdot \theta_{step}) \\ y &= R \cdot sin(n \cdot \theta_{step}) \end{aligned} \tag{1}$$

Donde $n \in [0, P]$ y:

$$\theta_{step} = \frac{360^\circ}{P} \tag{2}$$

En la práctica, para comenzar la secuencia binaria en el pixel de la esquina superior izquierda y recorrer el resto en sentido horario, se recorren los posibles valores de θ_{step} desde 180° a -180°, cada $-\theta_{step}$ grados.

En esta tarea se calculan dos características LBP circulares, LBP(8,1) y LBP(12,2), para las cuales los puntos de sus vecindades son aquellos mostrados en la **Figura 2**:

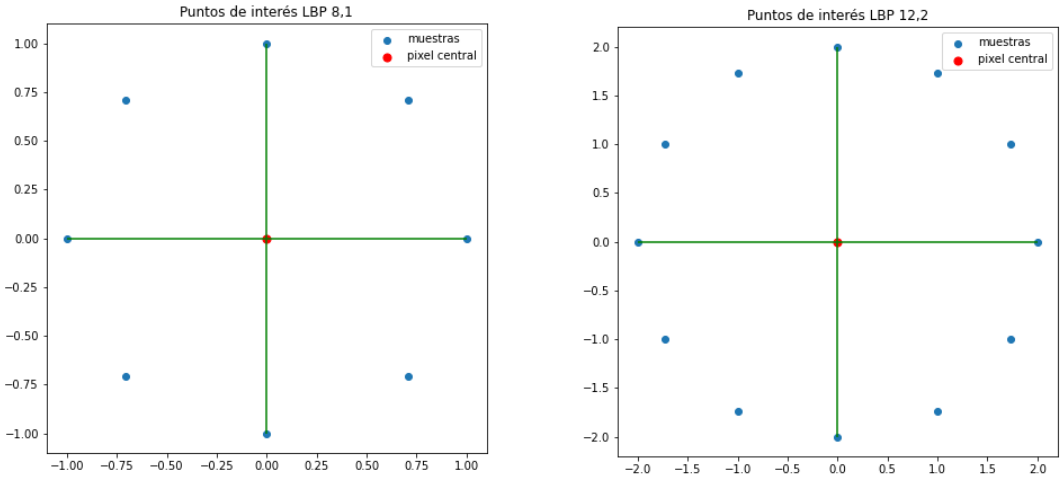

Figura 2: Puntos vecindades LBP(8,1) y LBP(12,2)

Por lo general se debe aplicar interpolación bilineal a las coordenadas obtenidas mediante las ecuaciones (1), puesto que no siempre es claro a cuál pixel deberían corresponder. Sin embargo, para el caso de LBP(8,1) y LBP(12,2) basta con redondear los valores de x e y, pues no hay mucha ambigüedad.

Una vez identificados los pixeles de la vecindad se realiza el procedimiento descrito en la Figura 1 para todos los pixeles de la imagen, colocando el valor del número binario obtenido en la posición correspondiente a cada pixel central. Al realizar este procedimiento para todos los pixeles de la imagen se obtiene su LBP, como se muestra en la **Figura 3**:

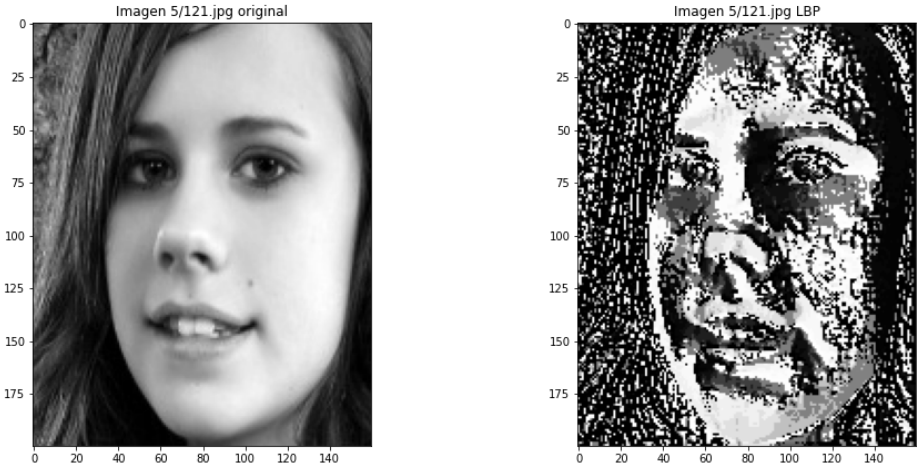

Figura 3: Ejemplo del cálculo de LBP en una imagen

Una vez obtenido el LBP de una imagen se puede calcular su vector de características mediante los siguientes pasos:

1. Calcular la representación LBP de una imagen
2. Dividir la representación LBP en cuatro bloques
3. Calcular un histograma de los valores LBP en cada bloque
4. Concatenar, de manera consistente, los histogramas de cada bloque

Para esta tarea se decide realizar los histogramas considerando sólo los valores uniformes, los cuales corresponden a los valores cuya secuencia binaria tiene a lo más dos cambios de 1 a 0 o viceversa, es decir, 00011100 es uniforme, pero 11011100 no, y agrupando todos los valores no uniformes en un solo bin.

Esto reduce la cantidad de bines del histograma de 2^P a la cantidad de valores uniformes en dicho rango (59 para LBP(8,1) y 135 para LBP(12,2)). De modo que el vector de características para el caso LBP(8,1) tiene $59 \cdot 4 = 236$ elementos y para el caso LBP(12,2) tiene $135 \cdot 4 = 540$.

2.2. Redes Neuronales

Las redes neuronales artificiales son redes distribuidas y paralelas de procesadores elementales simples (neuronas), en otras palabras, son un modelo muy aproximado de la estructura del cerebro humano.

La estructura fundamental de las redes neuronales artificiales son las neuronas. Estas unidades procesadoras de información cuentan con tres elementos básicos:

1. Pesos sinápticos que ponderan las entradas
2. Un sumador o combinador lineal
3. Una función de activación no lineal que limita la amplitud de la salida

Con esto en mente, el modelo de una neurona es el mostrado en la **Figura 4**:

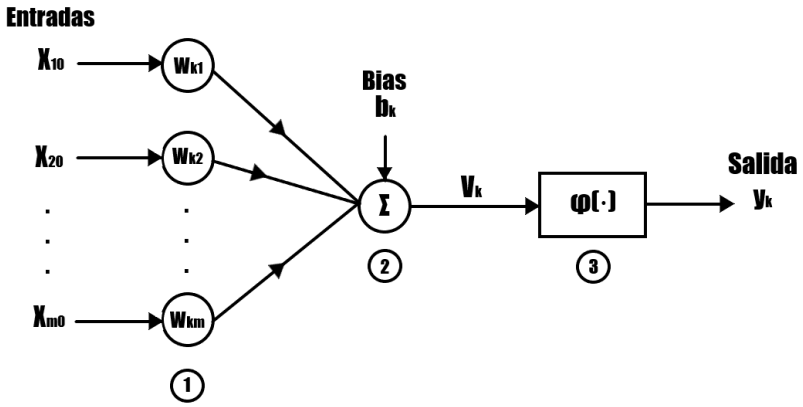

Figura 4: Modelo de una neurona artificial

Las dos ecuaciones que gobiernan el comportamiento de una neurona son:

$$\begin{aligned} u_k &= \sum_{j=1}^{n} w_{kj} x_j \\ y_k &= \psi(u_k + b_k) \end{aligned} \tag{3}$$

La constante b_k (el bias) tiene el efecto de aplicar una transformación afín a la salida u_k del combinador lineal:

$$v_k = u_k + b_k \tag{4}$$

Se suele añadir el bias a la entrada del combinador lineal expandiendo en 1 la dimensión de la entrada y asignándole a dicha entrada un peso igual al bias, de modo que las ecuaciones de la neurona se reformulan como:

$$\begin{aligned} s_k &= \sum_{j=0}^{n} w_{kj} x_j \\ y_k &= \psi(s_k) \end{aligned} \tag{5}$$

Existen muchas funciones de activación que se pueden elegir para $\psi(\cdot)$, pero para esta tarea se escoge la función ReLU (*Rectified Linear Unit*), la cual corresponde a:

$$\psi(s_k) = max(0, x) \tag{6}$$

Al combinar múltiples neuronas se obtienen las redes neuronales artificiales. En particular para esta tarea se utiliza una red neuronal multicapa con una única capa oculta, como se muestra en la **Figura 5**:

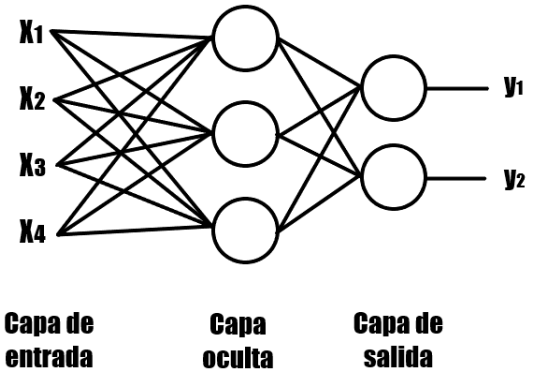

Figura 5: Ejemplo de red neuronal multicapa 4-3-2

Se le dice capa oculta puesto que para el usuario esta no es visible, a diferencia de las capas de entrada y salida.

El entrenamiento de las redes neuronales se realiza mediante aprendizaje supervisado, lo que consiste en mostrarle ejemplos etiquetados a la red y actualizar sus pesos w_{ij} en base al error. Esto es lo mismo que decir que se tiene una función objetivo $f(w)$ para la cual se quiere minimizar el valor esperado de esta función $\mathbb{E}[f(w)]$ con respecto a sus parámetros. Para esta tarea se utiliza el algoritmo ADAM que mediante estimaciones de momentos de primer y segundo orden computa una actualización de los pesos.

Esta actualización de pesos se lleva a cabo mediante el algoritmo de *Backpropagation*, el cual consiste en dos pasos:

1. Hacia adelante: Con los pesos w_{ij} fijos (e inicializados al azar) se calcula la respuesta de la red, propagando la entrada a través de las unidades ocultas y la capa de salida, con lo cual se determina el error.

2. Hacia atrás: La señal del error es propagada hacia atrás usando los mismos pesos de la red. Luego, se ajustan los pesos de toda la red al mismo tiempo utilizando el algoritmo de optimización escogido.

El error calculado en esta tarea está dado por la función de pérdida de *Cross Entropy*. La entropía cruzada de una distribución $\hat{y}$ relativa a una distribución y está dada por:

$$H(y, \hat{y}) = -\mathbb{E}_y[log(\hat{y})] \tag{7}$$

Donde $\mathbb{E}_y$ corresponde al valor esperado de y.

3. Extracción de características LBP

3.1. Preparar conjuntos de entrenamiento, prueba y validación

Se repite el mismo procedimiento realizado para la tarea 3, se deben separar las 600 imágenes entregadas por el equipo docente, las cuales incluyen 200 imágenes de personas entre 1 a 4 años, 200 imágenes de personas entre 5 a 27 años y 200 imágenes de personas mayores a 28 años, en los conjuntos de entrenamiento, prueba y validación.

La proporción indicada para la separación es del 60 % de las imágenes en el conjunto de entrenamiento, un 20 % en el conjunto de validación y el otro 20 % en el conjunto de prueba.

Para esto se decide almacenar las rutas de las imágenes (de la forma `carpeta/número.jpg`) en una lista `X`, y las clases de las imágenes en una lista `y`. Donde las clases de las imágenes corresponden a las carpetas: 0 para las personas entre 1 a 4 años, 1 para las personas entre 5 a 27 años, y 2 para las personas mayores a 28.

Una vez obtenidas las listas `X` e `y`, estas se dividen en los conjuntos indicados mediante la función `train_test_split` de `sklearn`. Para lograr las proporciones deseadas basta con aplicar dos veces la función, la primera dejando el 20 % de los datos para el conjunto de prueba, y la segunda repartiendo el resto, como se muestra en el **Listing 1**:

```
# dividir train 80% y test 20%
X_train, X_test, y_train, y_test  = train_test_split(X, y, test_size=0.2,
    random_state=1)

# dividir train 60% (0.75*0.8) y val 20% (0.25*0.8)
X_train, X_val, y_train, y_val  = train_test_split(X_train, y_train, test_size
    =0.25, random_state=1)
```

Listing 1: repartir datos

3.2. Identificar secuencias uniformes

Como se explicó en la **Sección 2.1**, los histogramas que forman el vector de características se construyen sólo considerando las secuencias binarias uniformes. Por lo que se requiere una forma de identificarlas.

La función encargada de esta tarea es `isUniform(seq)` que recibe como entrada una secuencia binaria en formato de string `seq`. Se utilizan dos punteros, uno que recorre la secuencia y otro que se queda atrás revisando los valores. Al principio ambos punteros comienzan en el primer caracter de la secuencia, a medida que `recorre` avanza por la secuencia, se comparan los valores que `recorre` encuentra con el valor al que `revisa` está apuntando, cada vez que estos valores no coinciden se suma uno al contador de cambios y se actualiza la posición de revisa a la posición donde se encontró el último cambio.

Dado que sólo se quiere saber si se encontraron más de dos cambios de 1 a 0 o viceversa en una secuencia binaria, se añade una condición de que si el contador de cambios es mayor a 2 la función retorna inmediatamente `False`, mientras que si se recorre toda la secuencia sin encontrar cambios se retorna `True`, como se muestra en el **Listing 2**:

```
def isUniform(seq):
  recorre = revisa = cambios = 0
  while recorre < len(seq):
    if seq[revisa] == seq[recorre]:
      recorre += 1
    else:
      revisa = recorre
      recorre += 1
      cambios += 1
    if cambios > 2:
      return False
  return True
```

Listing 2: Versión simplificada de isUniform() (Listing 10)

3.3. Transformar secuencia binaria a número entero

Como se explicó en la **Sección 2.1**, al calcular el patrón binario local de un pixel, se debe anotar el valor en base 10 de la secuencia binaria en la posición del pixel central. Para esto se requiere una función que tome una secuencia binaria y calcule su valor en base 10. Es sabido que la representación en base 10 de una secuencia binaria de n dígitos está dada por:

$$sec_{10} = \sum_{i=0}^{n} 2^i \quad \forall i \in [0, n] \mid sec_2[i] == 1 \tag{1}$$

La función encargada de esto es `binaryToInt(sec)`, que recibe como entrada una secuencia binaria en formato de string `seq`. Dado que el MSB (bit más significativo) se encuentra en la posición 0 de la secuencia y el LSB (bit menos significativo) se encuentra en la última posición de la secuencia, el orden en el que se recorren los elementos de la secuencia y los valores del exponente de 2 que les corresponden se encuentran en un orden invertido, como se muestra en la **Figura 6**:

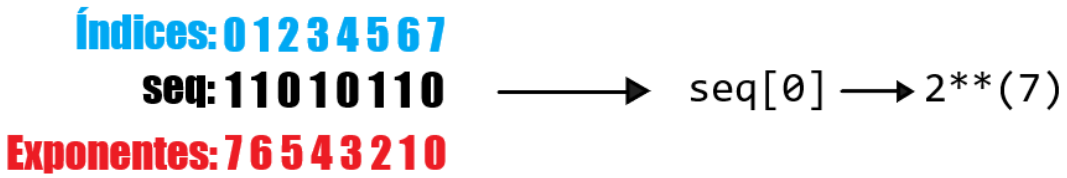

Figura 6: Ejemplo orden de índices y exponentes en secuencia binaria de 214

De modo que en la función se implementa un `for loop` invertido, que comienza en el largo de la secuencia y termina en 0 (orden de los exponentes), para revisar los valores de la secuencia se le resta el índice al largo de la secuencia (orden de los índices), y si el valor es 1 se suma al total 2^i, como se muestra en el **Listing 3**:

```
def binaryToInt(seq):
  total = 0
  for i in range(len(seq) - 1, -1, -1):
    if int(seq[(len(seq)  - 1) - i]) == 1:
      total += 2**(i)
  return total
```

Listing 3: Versión simplificada de binaryToInt() (Listing 11)

3.4. Cálculo de patrones binarios locales

Como se explicó en la **Sección 2.1**, para calcular los patrones binarios locales se debe recorrer la imagen pixel a pixel, identificando los pixeles de su vecindad, realizando las comparaciones binarias para crear la secuencia binaria y anotar su valor en base 10 en la posición del pixel.

Los puntos que corresponden a la vecindad deseada se guardan en una lista, para esto se recorre un `for loop` de -180 a 180, avanzando cada θ_{step} grados y se guardan tuplas que contienen la aproximación de los resultados de (1). Para comenzar la secuencia en el pixel de la vecindad que está más cerca de la esquina superior izquierda se mueve el primer elemento de la lista al final.

```
points = []
for theta in range(180,-180,-step):
  points.append( (round(r*np.cos(np.deg2rad(theta))),
                  round(r*np.sin(np.deg2rad(theta)))) )
points.append(points.pop(0))
```

Esta lista de tuplas contiene las coordenadas de los puntos de la vecindad con respecto a la posición (0,0), por lo que para identificar los pixeles de la vecindad de un pixel [y,x] basta con sumar [y,x] a las tuplas contenidas en la lista.

Entonces, se recorre la imagen pixel a pixel, para cada pixel se recorre la lista de puntos de la vecindad y se obtienen las coordenadas de los pixeles de la vecindad sumando [y,x] a los elementos de la lista. Para cada pixel en la vecindad se tienen dos casos, si su valor es mayor o igual al del pixel central se anota un 1 en la salida, y si su valor es menor se anota un 0. En ocasiones las coordenadas de la vecindad se salen de la imagen (coordenadas negativas o mayores al tamaño de la imagen), en cuyos casos simplemente se anota un 0 en la salida.

Finalmente, una vez obtenida la secuencia binaria que le corresponde al pixel [y,x] se revisa si es uniforme o no. Si es uniforme, se anota el valor en base 10 de la secuencia en la posición [y,x] de la salida, y si no lo es, se anota un 5. El valor 5, escogido arbitrariamente puesto que es el primer valor no uniforme en los rangos $[0, 2^8]$ o $[0, 2^{12}]$, se anota para identificar todas las secuencias no uniformes y poder agruparlas en el histograma que se construirá más adelante.

La función que se encarga de implementar el procedimiento recién descrito es `LBP(img, p, r)` que recibe como entradas la imagen a la cual se le calcula el patrón LBP `img`, la cantidad de puntos en la vecindad `p` y el radio de la vecindad `r`. Una versión simplificada de esta función se muestra en el **Listing 4**:

```
def LBP(img, p, r):
    out = np.zeros((img.shape[0],img.shape[1]))
    step = int(360/p)

    points = []
    for theta in range(180,-180,-step):
      points.append( (round(r*np.cos(np.deg2rad(theta))),
                      round(r*np.sin(np.deg2rad(theta)))) )
    points.append(points.pop(0))

    for y in range(img.shape[0]):
      for x in range(img.shape[1]):
        word = ""
        for point in points:
          py, px = sumTuples(point,(y,x))
          if (min(py,px) < 0) or ((py >= img.shape[0]) or (px >= img.shape[1])):
            word += "0"
            continue
          if img[py, px] >= img[y,x]:
            word += "1"
          else:
            word += "0"
        if isUniform(word):
          out[y,x] = binaryToInt(word)
        else:
          out[y,x] = float(5)
    return out
```

Listing 4: Versión simplificada de LBP() (Listing 13)

La suma de [y,x] a las tuplas de la lista `points` se realiza mediante una pequeña función auxiliar `sumTuples(tup1, tup2)` que recibe como entrada dos tuplas `tup1` y `tup2`. Esta función simplemente suma las tuplas elemento a elemento, como se muestra en el **Listing 5**:

```
def sumTuples(tup1, tup2):
  return (tup1[0] + tup2[0], tup1[1] + tup2[1])
```

Listing 5: Versión simplificada de sumTuples() (Listing 12)

3.5. Histograma de LBP

Como también se explicó en la **Sección 2.1**, después de calcular la representación LBP de una imagen, esta se debe dividir en cuatro bloques, en cada uno de ellos calcular un histograma, y concatenar los 4 histogramas para así obtener el vector de características. Dividir la entrada en 4 bloques es simple, sólo se necesitan dos `for loops` de 0 a 2 cada uno y tomar slices de la entrada en intervalos correspondientes a la mitad de las distancias:

```
vStep = int(lbp.shape[0]/2)
hStep = int(lbp.shape[1]/2)
for y in range(2):
    for x in range(2):
        sec = lbp[vStep*y:vStep*(y+1), hStep*x:hStep*(x+1)]
```

Para construir el histograma, primero se identifican todos los posibles valores en los rangos $[0, 2^8]$ o $[0, 2^{12}]$ dependiendo del caso. Se crea un vector de ceros de largo igual a la cantidad de posibles valores uniformes más uno, se recorre dicho vector, se busca la cantidad de veces que sus elementos aparecen en la subsección de la entrada (uno de los cuatro bloques) y se anota la cantidad en el vector de salida, en la posición correspondiente al valor uniforme.

En otras palabras, a cada uno de los valores uniformes posibles se le asigna un bin del histograma, simplemente en orden de aparición, y a todos los valores no uniformes se les asigna un único bin, el último. Esto se realiza para las cuatro secciones de la entrada y los histogramas se concatenan mediante la función `np.concatenate`. La función encargada de este procedimiento es `LBPHistogram(lbp, points)`, la cual recibe como entradas la versión LBP de una imagen `lbp` y la cantidad de pixeles de la vecindad considerada `points`. Se presenta una versión simplificada de dicha función en el **Listing 6**:

```
def LBPHistogram(lbp, points):
  unique = makeUniformList(points)
  out = np.array([])
  vStep = int(lbp.shape[0]/2)
  hStep = int(lbp.shape[1]/2)

  for y in range(2):
    for x in range(2):
      sec = lbp[vStep*y:vStep*(y+1), hStep*x:hStep*(x+1)]
      secHist = np.zeros(len(unique))
      for i in range(len(unique)):
        secHist[i] = (sec == unique[i]).sum()
      out = np.concatenate((out, secHist))
  return out
```

Listing 6: Versión simplificada de LBPHistogram() (Listing 15)

Para obtener la lista de valores uniformes simplemente se recorren todos los valores posibles en los rangos $[0, 2^8]$ o $[0, 2^{12}]$ dependiendo del caso y se utiliza la función `isUniform` para decidir si se añaden a la salida o no.

La función encargada de esto es `makeUniformList(points)`, la cual recibe la cantidad de puntos escogidos para la vecindad `points`. Para obtener la representación binaria de los números se utiliza la función `bin()`, se le quitan los primeros dos elemenots (0b) y se añaden ceros a la izquierda para que tenga tantos dígitos como puntos hay en la vecindad mediante la función `zfill()`, como se muestra en el **Listing 7**

```
def makeUniformList(points):
  uniformList = []
  for i in range(2**points):
    if isUniform(bin(i)[2:].zfill(points)):
      uniformList.append(i)
  uniformList.append(5) # primer numero no uniforme 00000101
  return np.array(uniformList, np.double)
```

Listing 7: Versión simplificada de makeUniformList() (Listing 14)

Se añade el valor 5 al final de la lista, para que así el último bin del histograma creado en `LBPHistogram` cuente las ocurrencias de valores no uniformes.

3.6. Extraer características de los conjuntos

Ya teniendo las funciones necesarias para calcular el vector de características LBP se deben calcular las características de todas las imágenes entregadas por el cuerpo docente, con el fin de entrenar los clasificadores.

Para esto basta con, para cada carpeta de imágenes, iterar sobre las imágenes que contiene, calcular su representación LBP y computar su histograma:

```
histLen = len(makeUniformList(points)) # 59 para LBP_8, 135 para LBP_12
# conjunto de entrenamiento
LBP_X_train = np.zeros((X_train.shape[0],histLen*4))
for i in range(len(X_train)):
  img = cv2.imread(f"{PATH}/{X_train[i]}", 0).astype(np.double)
  LBPchar = LBP(img,points,r)
  LBPhist = LBPHistogram(LBPchar, points)
  LBP_X_train[i,:] = LBPhist

# conjunto de validacion
LBP_X_val = np.zeros((X_val.shape[0],histLen*4))
for i in range(len(X_val)):
  img = cv2.imread(f"{PATH}/{X_val[i]}", 0).astype(np.double)
  LBPchar = LBP(img,points,r)
  LBPhist = LBPHistogram(LBPchar, points)
  LBP_X_val[i,:] = LBPhist

# conjunto de prueba
LBP_X_test = np.zeros((X_test.shape[0],histLen*4))
for i in range(len(X_test)):
  img = cv2.imread(f"{PATH}/{X_test[i]}", 0).astype(np.double)
  LBPchar = LBP(img,points,r)
  LBPhist = LBPHistogram(LBPchar, points)
  LBP_X_test[i,:] = LBPhist
```

Listing 8: Versión simple de Listing **??**

Cabe destacar que se normalizan los histogramas mediante la función `StandardScaler()` de `sklearn`. Se entrena el scaler sobre el conjunto de entrenamiento y se aplica la normalización a los 3 conjuntos:

```
scaler = StandardScaler()
scaler.fit(LBP_X_train)
LBP_X_train = scaler.transform(LBP_X_train)
LBP_X_val = scaler.transform(LBP_X_val)
LBP_X_test = scaler.transform(LBP_X_test)
```

Por último, para después crear los `DataLoaders` que necesita `pytorch` se crean listas que contienen las tuplas (vector de características, etiqueta):

```
data_train = []
for i in range(len(LBP_X_train)):
  data_train.append([LBP_X_train[i], y_train[i]])

data_val = []
for i in range(len(LBP_X_val)):
  data_val.append([LBP_X_val[i], y_val[i]])

data_test = []
for i in range(len(LBP_X_test)):
  data_test.append([LBP_X_test[i], y_test[i]])
```

4. Entrenar clasificadores

Para realizar las predicciones se trabaja únicamente con redes neuronales, en particular se utiliza la biblioteca `pytorch` para construir y entrenar dichas redes.

Se entrenan 4 redes, todas con una única capa oculta, con función de activación ReLU, una capa de salida con 3 neuronas, con función de pérdida `CrossEntropyLoss` y optimizador ADAM. Las diferencias entre las 4 redes radican en variar la cantidad de neuronas en la capa oculta y la tasa de aprendizaje.

Antes de construir las redes por separado se debe constuir la clase `Network`, que permite crear las redes neuronales con los parámetros deseados:

```
class Network(nn.Module):

    def __init__(self, input_size, num_classes, hidden_neurons):
        super().__init__()

        self.fc1 = nn.Linear(input_size, hidden_neurons)
        self.fc2 = nn.Linear(hidden_neurons, num_classes)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Esta clase tiene dos métodos, `__init__`, el constructor donde se inicializa la red en función de los parámetros que se le entregan, y `forward` que define cómo se realiza la propagación de la señal desde la entrada hasta la salida.

Una vez definida la estructura de la red se definen algunas funciones auxiliares que facilitan la creación de las 4 redes.

En primer lugar se necesita una función para crear una red y asignarle un dispositivo (cpu o gpu), para lo cual se crea la función `makeNN(input_size, num_classes, hidden_neurons)` la cual recibe el tamaño de la entrada de la red `input_size`, la cantidad de neuronas que debe haber a la salida `num_classes` y la cantidad de neuronas en la capa oculta `hidden_neurons`.

Esta función crea una red neuronal con los parámetros deseados y le asigna una gpu si esta se encuentra disponible o si no una cpu, como se muestra en el **Listing 17**:

```
def makeNN(input_size, num_classes, hidden_neurons):
  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
  return Network(input_size, num_classes, hidden_neurons).to(device), device
```

Listing 9: makeNN

Para entrenar las redes estas deben recibir los datos en formato de `DataLoader`, los cuales corresponden a tensores que contienen las tuplas (vector de características, etiqueta) particionados en `batches`. La función encargada de crear los `DataLoaders` es `makeLoaders(data_train, data_val, data_test, batch_size)` que recibe los conjuntos de entrenamiento, validación y prueba (`data_train, data_val y data_test`) y el tamaño deseado para los `batches`, `batch_size`:

```
def makeLoaders(data_train, data_val, data_test, batch_size):
  train_loader = DataLoader(dataset = data_train, batch_size = batch_size,
                            shuffle = True)
  val_loader = DataLoader(dataset = data_val, batch_size = batch_size,
                            shuffle = True)
  test_loader = DataLoader(dataset = data_test, batch_size = batch_size,
                            shuffle = True)
  return train_loader, val_loader, test_loader
```

Una vez creada la red y los loaders de los datos, ya se puede entrenar. Este procedimiento contempla varios pasos. De manera general, la red se entrena con el conjunto de entrenamiento y se prueba su rendimiento en el conjunto de validación, a medida que avanza el entrenamiento se van almacenando los errores en el conjunto de entrenamiento (`train_loss`), los errores en el conjunto de validación (`val_loss`) y los modelos que corresponden a dichos errores (`models`).

La función encargada del entrenamiento es `trainNN(model, learning_rate, train_loader, val_loader, num_epochs, device)`, la cual recibe como entrada el modelo que se quiere entrenar (`model`), la tasa de aprendizaje (`learning_rate`), los loaders de entrenamiento y validación (`train_loader, val_loader`), la cantidad de iteraciones en las que se entrena el modelo (`num_epochs`), el dispositivo en el cual se realiza el entrenamiento (`device`) y la cantidad de rondas en las que `val_loss` tiene que amuentar para detener el entrenamiento (`es_rounds`).

Para optimizar el rendimiento de la red es necesario detener el entrenamiento de esta cuando se detecte overfitting, es decir, hay que detener el entrenamiento cuando `val_loss` aumente y `train_loss` siga disminuyendo. Para lograr esto se deben almacenar ambas pérdidas, los modelos obtenidos en cada iteración y se necesita un contador para las épocas en las que `val_loss` aumenta:

```
def trainNN(model, learning_rate, train_loader, val_loader, num_epochs, device,
    es_rounds):
 # Checkpoint de loss y modelos
  train_losses = []
  val_losses = []
  models = []

  # Variables early stopping
  down_counter = 0
```

También es necesario inicializar la función de pérdida y el optimizador que, como se mencionó en la **Sección 2.2**, corresponden a `CrossEntropyLoss` y ADAM:

```
# criterio de loss y optimizador
  criterion = nn.CrossEntropyLoss()
  optm = Adam(model.parameters(), lr = learning_rate)
```

Para realizar el entrenamiento se crea un loop que repita el entrenamiento la cantidad de veces especificada por `num_epoch`, se selecciona el modo de entrenamiento para la red (indicarle a `pytorch` que debe modificar los pesos de la red) y se inicializa el contador del error de entrenamiento para dicha época:

```
# Entrenamiento
  for epoch in range(num_epochs):
    model.train()
    train_loss = 0.0
```

Se le muestran todos los batches del conjunto de entrenamiento a la red, se envian tanto los datos como las etiquetas al dispositivo indicado por `device`, se limpian los gradientes de la iteración anterior, se propaga la señal hacia adelante y se computa el error, y se ajustan los pesos hacia atrás guardando el error de la época:

```
    for index, (d_train, l_train) in enumerate(train_loader): # para cada batch
      d_train = d_train.to(device) # train data
      l_train = l_train.to(device) # train labels

      # limpiar gradientes
      optm.zero_grad()

      # forwards
      train_scores = model(d_train.float())
      loss = criterion(train_scores, l_train)

      # backwards
      loss.backward()
      optm.step()
      train_loss += loss.item()
```

Luego se realiza un procedimiento análogo con el conjunto de validación, se selecciona el modo de evaluación para la red (que `pytorch` no modifique los gradientes), se inicializa el contador del error de validación, se indica que no se deben calcular gradientes en las siguientes propagaciones, y para todos los batches del conjunto de validación se computa el error:

```
    # Monitoreo validacion
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
      for index, (d_val, l_val) in enumerate(val_loader): # para cada batch de
   validacion
        d_val = d_val.to(device) # validation data
        l_val = l_val.to(device) # validation labels

        val_scores = model(d_val.float())
        loss = criterion(val_scores, l_val)
        val_loss += loss.item()
```

El error de entrenamiento y el de validación fueron calculados de manera acumulada para todo el batch, por lo cual es necesario dividirlos por la cantidad de datos en cada conjunto para obtener el error real de la época:

```
    # Loss de entrenamiento y validacion de la epoca
    train_loss = train_loss/len(train_loader)
    val_loss = val_loss/len(val_loader)
```

Una vez obtenidos el error de entrenamiento y el de validación de la época se puede revisar si es necesario detener el entrenamiento. Como se mencionó esto, el entrenamiento se detiene cuando `train_loss` disminuye y `val_loss` aumenta durante `es_rounds` épocas.

Para medir si estos errores aumentan o disminuyen, se compara su valor actual con el promedio de todos los errores anteriores, para que así este mecanismo sea resistente al ruido presente en el entrenamiento. Si se cumple que `train_loss` es menor al promedio de `train_losses` y que `val_loss` es mayor al promedio de `val_losses` entonces el contador `down_counter` aumenta en 1, en caso contrario el contador se reinicia a 0. Luego, si `down_counter` es mayor a `es_rounds` se detiene el entrenamiento:

```
    # Early Stopping
    with warnings.catch_warnings():
      warnings.simplefilter("ignore", category=RuntimeWarning)
      if (val_loss > np.mean(val_losses)) and (train_loss < np.mean(train_losses)):
        down_counter += 1
      else:
        down_counter = 0
      if down_counter > es_rounds:
        print(f'Validation loss rising, stopping training!')
        break
```

Cabe destacar que se suprimen las advertencias obtenidas por calcular el promedio de una lista vacía puesto que no conlleva ningún error y sólo entorpece la visualización del entrenamiento.

Finalmente, si el entrenamiento no se detuvo, se almacenan los errores y el modelo de la época, y al finalizar el loop se retornan todos los errores y modelos almacenados:

```
    # Guardar losses y modelo de la epoca
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    models.append({'epoch': epoch, 'train_loss': train_loss, 'val_loss': val_loss,
                   'model': model.state_dict()})
  return train_losses, val_losses, models
```

La función completa se adjunta en el anexo, en el **Listing 19**.

Una vez obtenidos los errores y los modelos del entrenamiento, se debe escoger al modelo que haya obtenido el menor error de validación. Para lo cual se crea la función `getBestModel(models)` que recibe la lista de modelos obtenidos durante el entrenamiento `models`, la cual corresponde a una lista de diccionarios que contienen la época, el error de entrenamiento, el error de validación y el modelo.

La función simplemente recorre al lista de diccionarios y busca el menor error de validación, luego retorna dicho error, el modelo y la época asociada:

```
def getBestModel(models):
  best_loss = np.inf
  best_model = models[0]
  last_epoch = 0
  for i in range(len(models)):
    if models[i]['val_loss'] < best_loss:
      best_loss = models[i]['val_loss']
      best_model = models[i]['model']
      last_epoch = models[i]['epoch']
  return best_loss, best_model, last_epoch
```

El siguiente paso después de decidir cuál es el mejor modelo es hacer predicciones con este, para lo cual se crea la función `makePredictions(model, data)`, la cual recibe el modelo que realiza las predicciones (`model`) y los datos sobre los cuales se quiere predecir (`data`).

La función simplemente recorre los datos uno a uno y va guardando las predicciones realizadas por el modelo en una lista. Para asegurar que no haya errores de compatibilidad se deben enviar tanto el modelo como los datos a la cpu y se define que ambos son del tipo `torch.DoubleTensor`. Las predicciones realizadas por el modelo son los valores de las 3 neuronas de la capa de salida, por lo que hay que obtener el índice de la neurona con el mayor valor:

```
def makePredictions(model, data):
  predictions = []
  model.type(torch.DoubleTensor)
  model.to('cpu')
  model.eval()
  for i in range(len(data)):
    input = torch.tensor(np.double(data[i])).type(torch.DoubleTensor)
    input.to('cpu')
    pred = model(input)
    predictions.append((pred == torch.max(pred)).nonzero(as_tuple=True)[0].item())
  return predictions
```

Hay un par de funciones auxiliares más que no serán explicadas en detalle, como `drawLossCurves`, `drawConfusionMatrix` y `stats`, puesto que son simplemente las que se encargan de presentar los gráficos, pero sí se incluyen en el Anexo.

5. Resultados

Se construyen 4 redes neuronales, especificando 700 rondas de entrenamiento, un batch size de 18 y 5 rondas como criterio de detención, a continuación se presentan los resultados de entrenar estas redes:

5.1. LBP 8,1

A continuación se presentan los resultados obtenidos con LBP 8,1:

5.1.1. NN 0: 64 neuronas, 0.0001 lr

Al entrenar la red neuronal con 64 neuronas en la capa oculta y una tasa de aprendizaje de 0.0001 se obtienen las curvas de pérdida mostradas en la **Figura 7**:

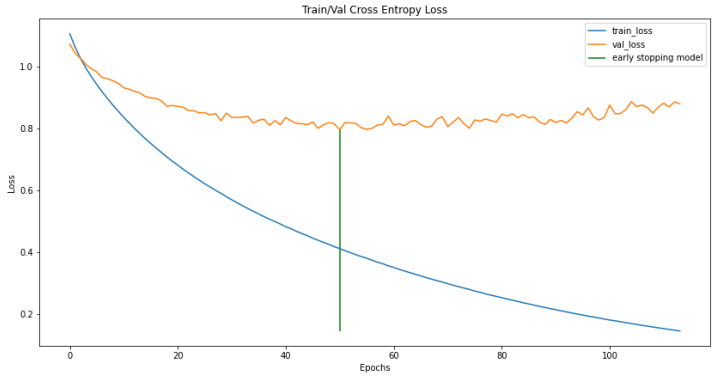

Figura 7: Curvas de Cross Entropy Loss modelo 0 en conjuntos de entrenamiento y validación

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento y validación se obtienen las matrices de confusión presentadas en la **Figura 8**:

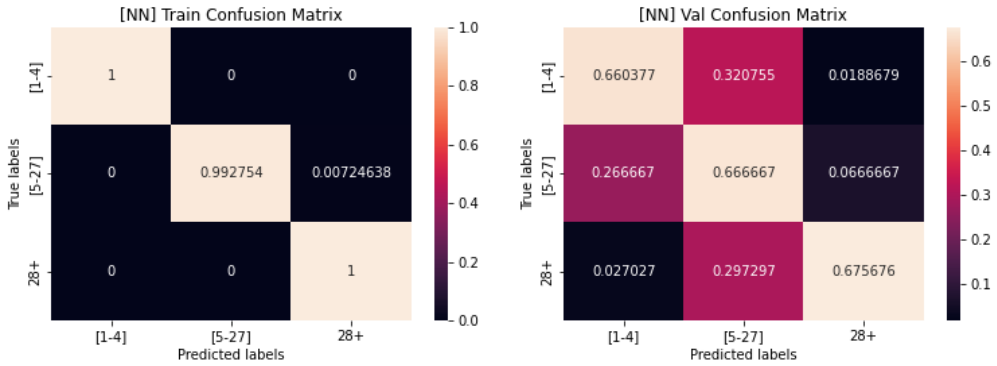

Figura 8: Matriz de confusión modelo 0 sobre entrenamiento y validación

Este modelo obtiene `test_accuracy` = 0.997, `val_accuracy` = 0.667.

5.1.2. NN 1: 64 neuronas, 0.00001 lr

Al entrenar la red neuronal con 64 neuronas en la capa oculta y una tasa de aprendizaje de 0.00001 se obtienen las curvas de pérdida mostradas en la **Figura 9**:

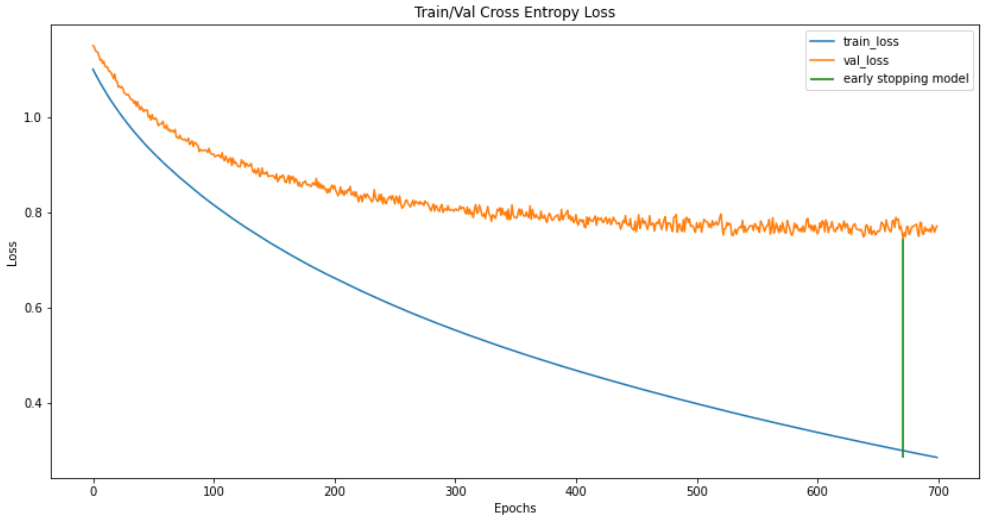

Figura 9: Curvas de Cross Entropy Loss modelo 1 en conjuntos de entrenamiento y validación

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento y validación se obtienen las matrices de confusión presentadas en la **Figura 10**:

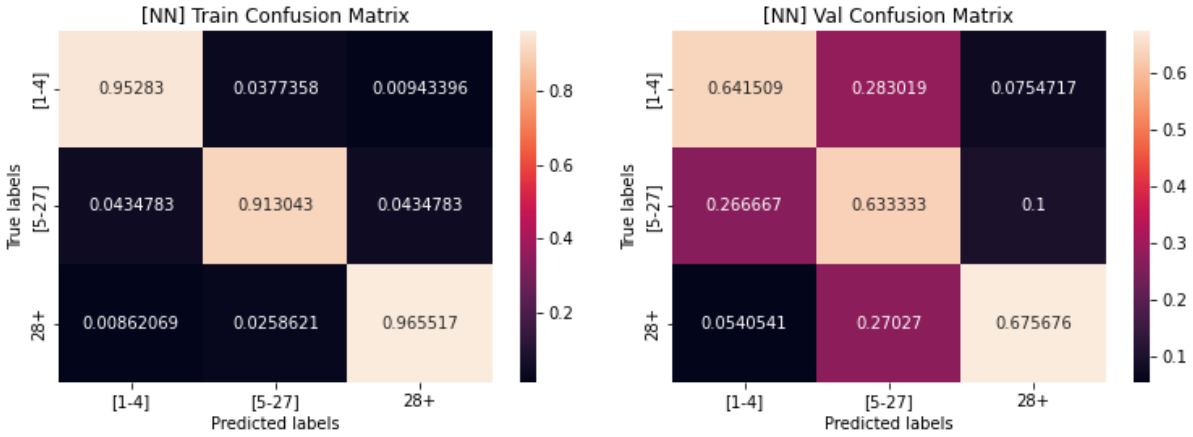

Figura 10: Matriz de confusión modelo 1 sobre entrenamiento y validación

Este modelo obtiene `test_accuracy` = 0.941, `val_accuracy` = 0.650.

5.1.3. NN 2: 128 neuronas, 0.0001 lr

Al entrenar la red neuronal con 128 neuronas en la capa oculta y una tasa de aprendizaje de 0.0001 se obtienen las curvas de pérdida mostradas en la **Figura 11**:

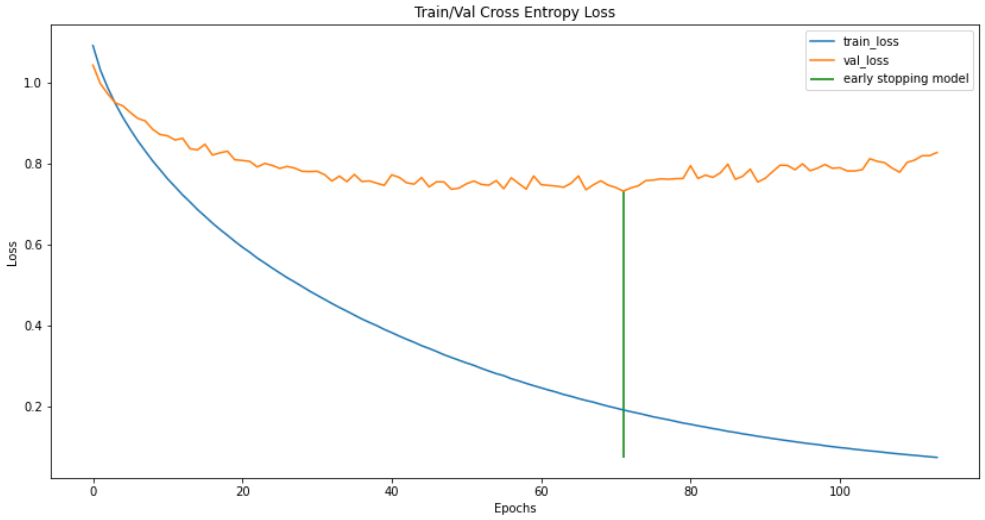

Figura 11: Curvas de Cross Entropy Loss modelo 2 en conjuntos de entrenamiento y validación

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento y validación se obtienen las matrices de confusión presentadas en la **Figura 12**:

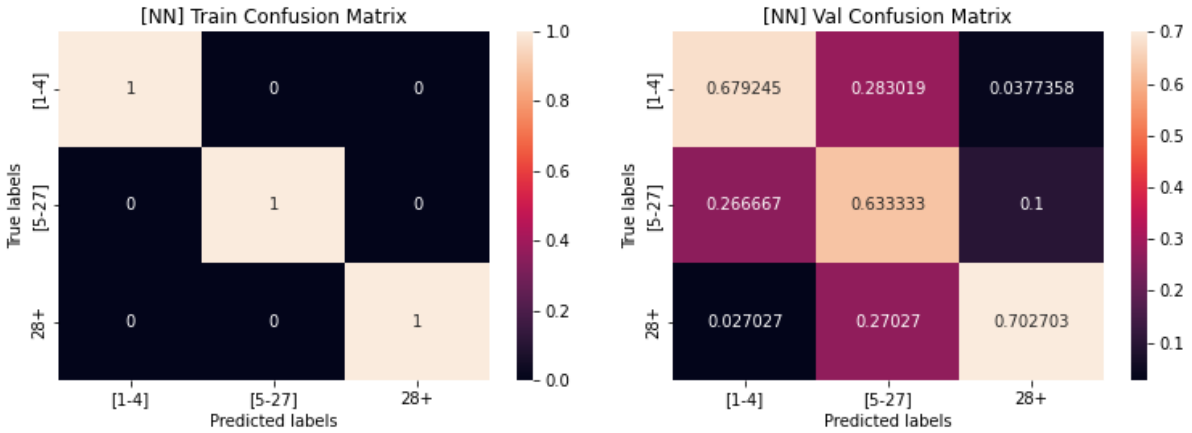

Figura 12: Matriz de confusión modelo 2 sobre entrenamiento y validación

Este modelo obtiene `test_accuracy` = 1.00, `val_accuracy` = 0.675.

5.1.4. NN 3 : 128 neuronas, 0.00001 lr

Al entrenar la red neuronal con 128 neuronas en la capa oculta y una tasa de aprendizaje de 0.00001 se obtienen las curvas de pérdida mostradas en la **Figura 13**:

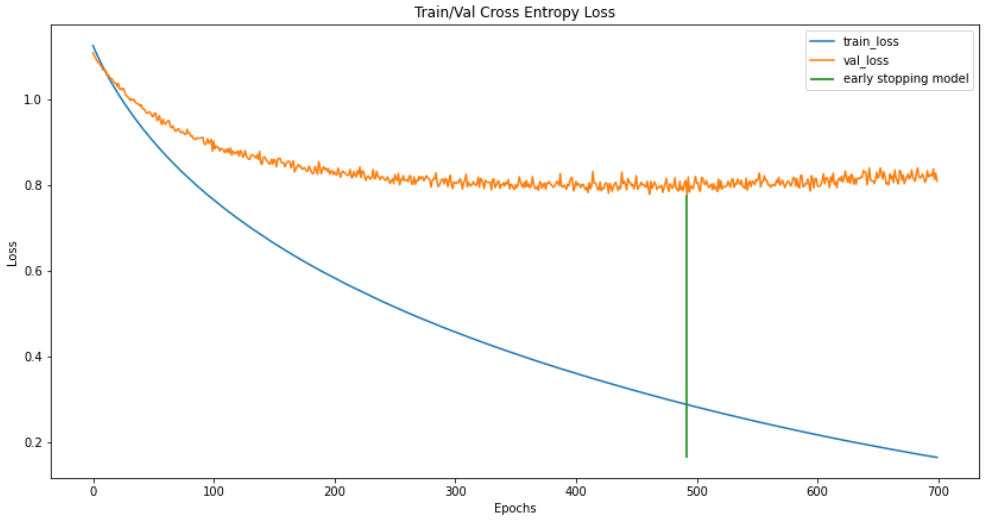

Figura 13: Curvas de Cross Entropy Loss modelo 3 en conjuntos de entrenamiento y validación

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento y validación se obtienen las matrices de confusión presentadas en la **Figura 14**:

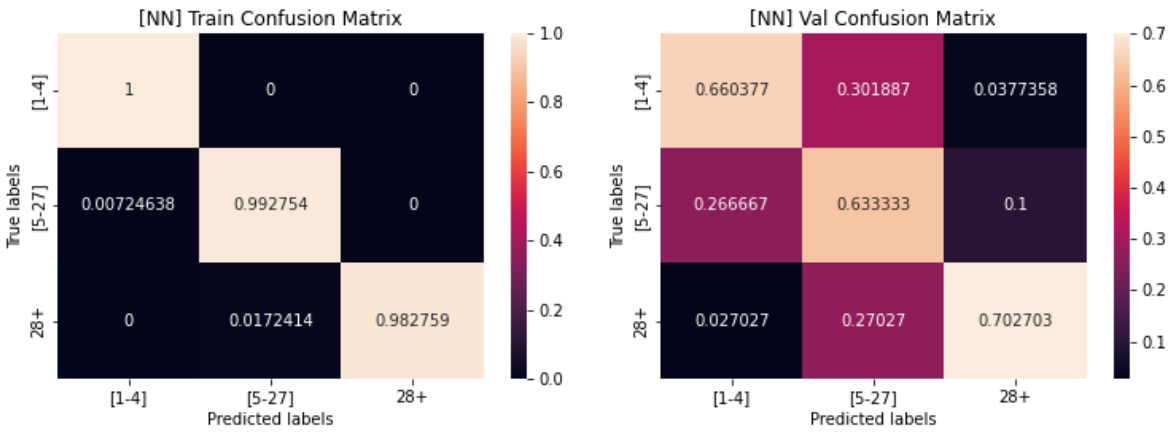

Figura 14: Matriz de confusión modelo 3 sobre entrenamiento y validación

Este modelo obtiene `test_accuracy` = 0.991, `val_accuracy` = 0.667.

El mejor de estos 4 modelos según la `accuracy` en el conjunto de validación es el modelo 2 (5.2.3), con una `accuracy` de validación de 0.675, por lo que se elige este conjunto para realizar la predicción sobre el conjunto de prueba, cuyos resultados se presentan en la **Figura 15**:

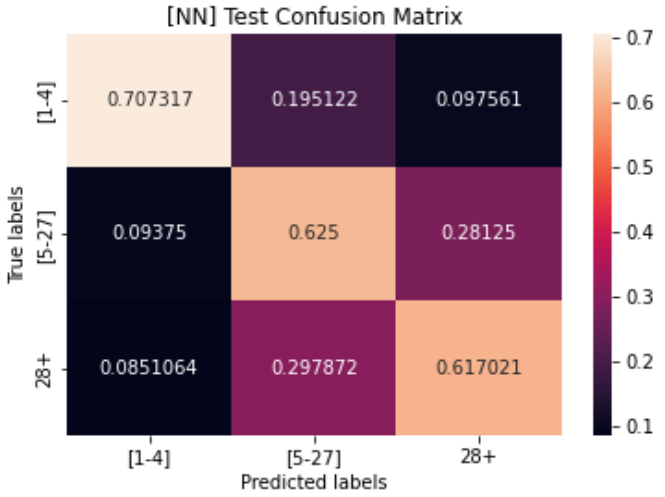

Figura 15: Matriz de confusión modelo 2 sobre conjunto de prueba

Este modelo alcanza una `accuracy` de 0.65 en el conjunto de prueba.

5.2. LBP 12.2

A continuación se presentan los resultados obtenidos con LBP 12,2:

5.2.1. NN 0: 64 neuronas, 0.0001 lr

Al entrenar la red neuronal con 64 neuronas en la capa oculta y una tasa de aprendizaje de 0.0001 se obtienen las curvas de pérdida mostradas en la **Figura 16**:

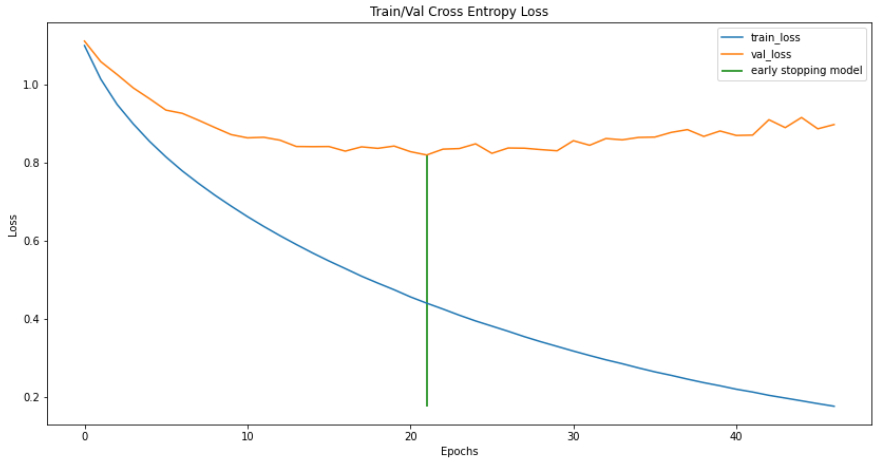

Figura 16: Curvas de Cross Entropy Loss modelo 0 en conjuntos de entrenamiento y validación

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento y validación se obtienen las matrices de confusión presentadas en la **Figura 17**:

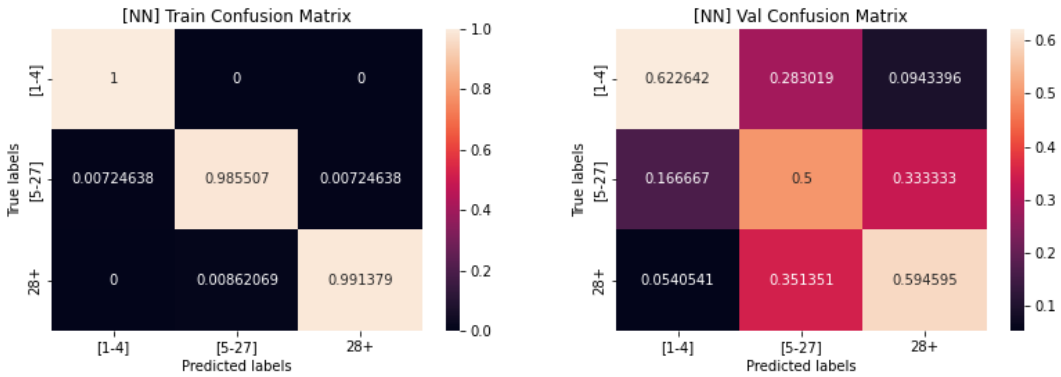

Figura 17: Matriz de confusión modelo 0 sobre entrenamiento y validación

Este modelo obtiene `test_accuracy` = 0.99, `val_accuracy` = 0.58.

5.2.2. NN 1: 64 neuronas, 0.00001 lr

Al entrenar la red neuronal con 64 neuronas en la capa oculta y una tasa de aprendizaje de 0.00001 se obtienen las curvas de pérdida mostradas en la **Figura 18**:

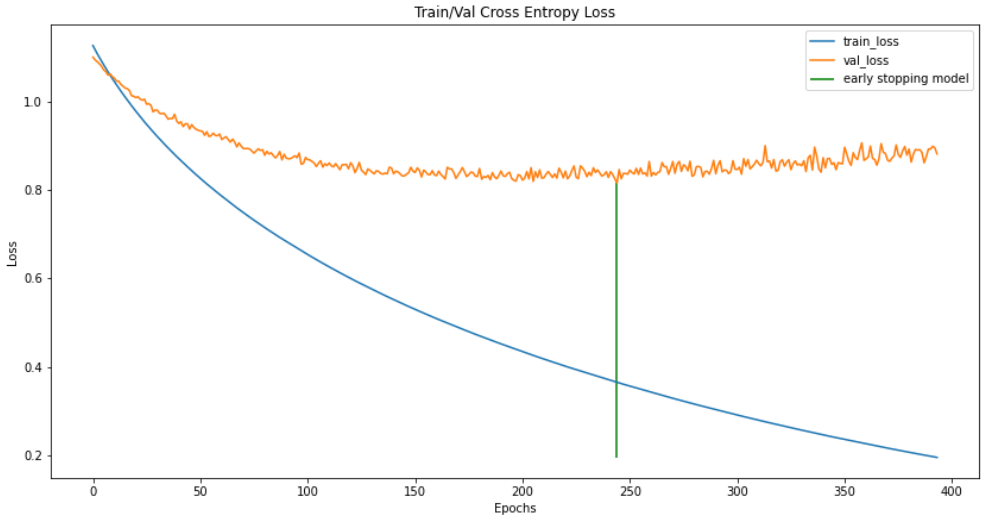

Figura 18: Curvas de Cross Entropy Loss modelo 1 en conjuntos de entrenamiento y validación

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento y validación se obtienen las matrices de confusión presentadas en la **Figura 19**:

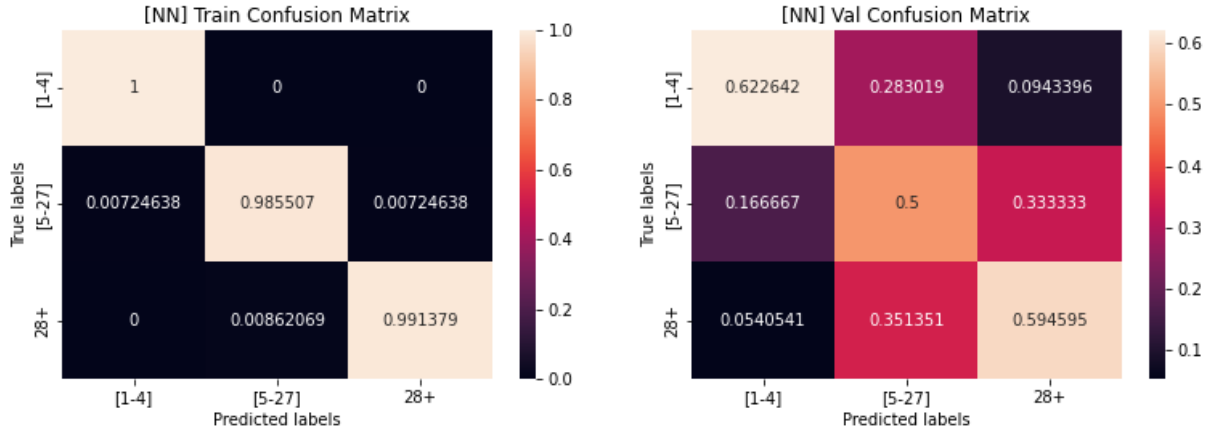

Figura 19: Matriz de confusión modelo 1 sobre entrenamiento y validación

Este modelo obtiene `test_accuracy` = 0.99, `val_accuracy` = 0.58.

5.2.3. NN 2: 128 neuronas, 0.0001 lr

Al entrenar la red neuronal con 128 neuronas en la capa oculta y una tasa de aprendizaje de 0.0001 se obtienen las curvas de pérdida mostradas en la **Figura 20**:

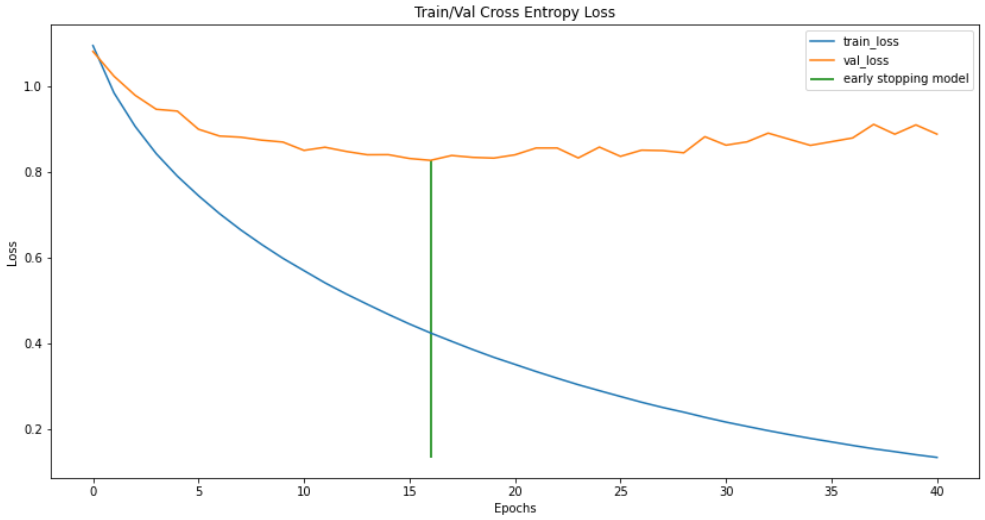

Figura 20: Curvas de Cross Entropy Loss modelo 2 en conjuntos de entrenamiento y validación

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento y validación se obtienen las matrices de confusión presentadas en la **Figura 21**:

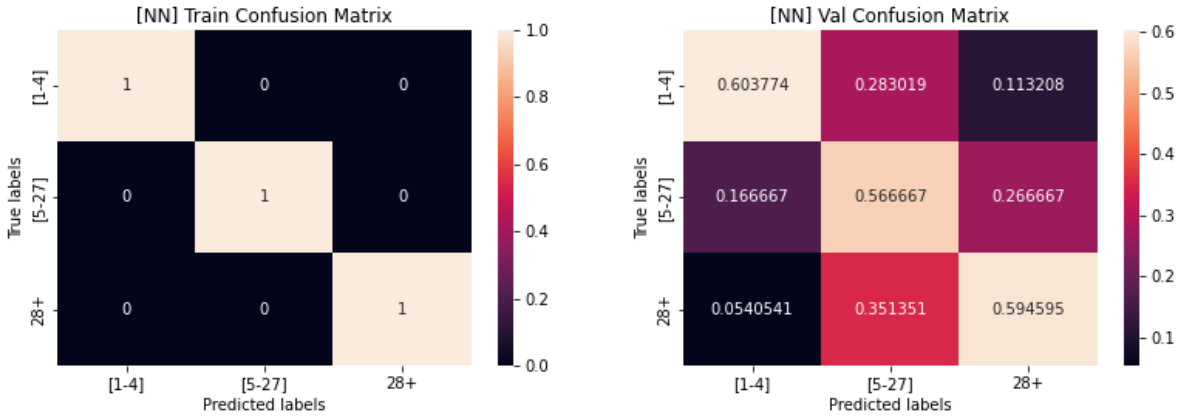

Figura 21: Matriz de confusión modelo 2 sobre entrenamiento y validación

Este modelo obtiene `test_accuracy` = 1.00, `val_accuracy` = 0.59.

5.2.4. NN 3 : 128 neuronas, 0.00001 lr

Al entrenar la red neuronal con 128 neuronas en la capa oculta y una tasa de aprendizaje de 0.00001 se obtienen las curvas de pérdida mostradas en la **Figura 22**:

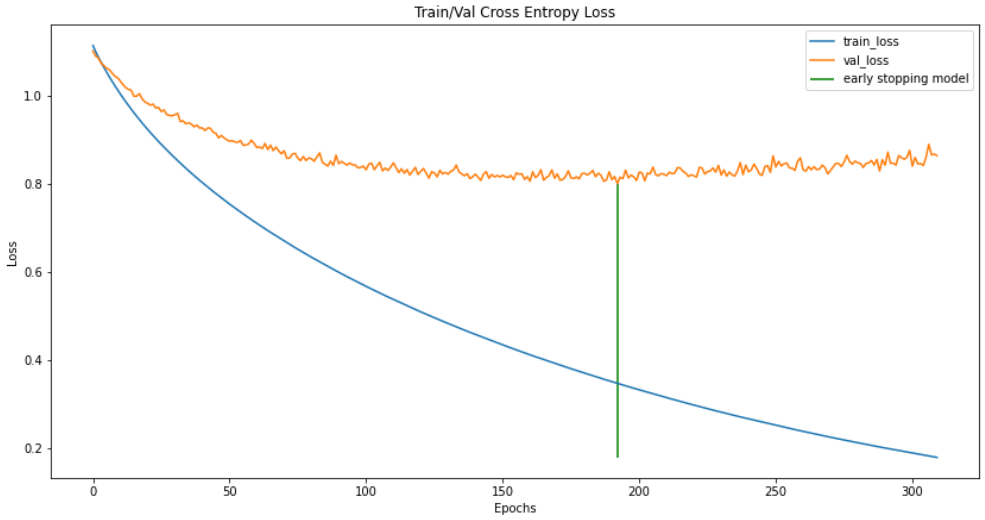

Figura 22: Curvas de Cross Entropy Loss modelo 3 en conjuntos de entrenamiento y validación

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento y validación se obtienen las matrices de confusión presentadas en la **Figura 23**:

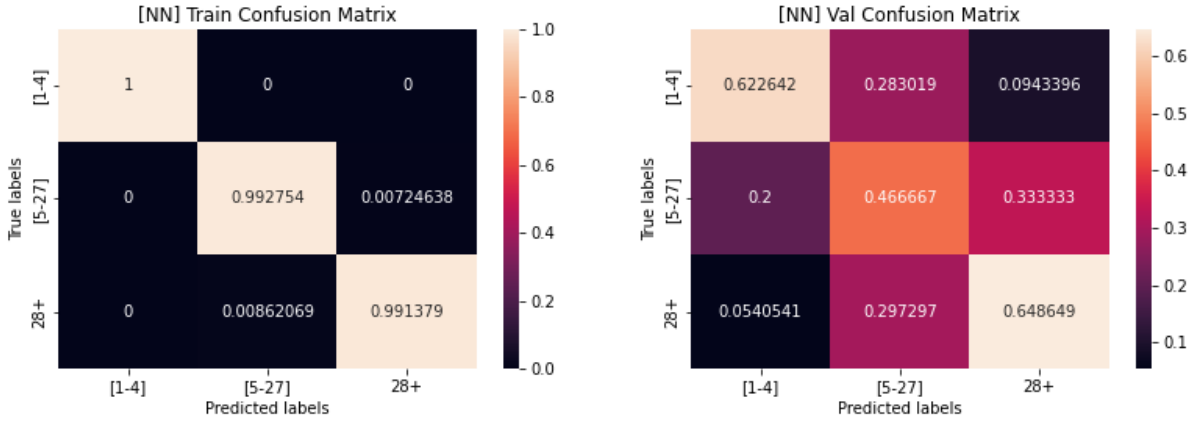

Figura 23: Matriz de confusión modelo 3 sobre entrenamiento y validación

Este modelo obtiene `test_accuracy` = 0.99, `val_accuracy` = 0.59.

El mejor de estos 4 modelos según la `accuracy` en el conjunto de validación es el modelo 2 (5.2.3), con una `accuracy` de 0.6, por lo que se elige este conjunto para realizar la predicción sobre el conjunto de prueba, cuyos resultados se presentan en la **Figura 24**:

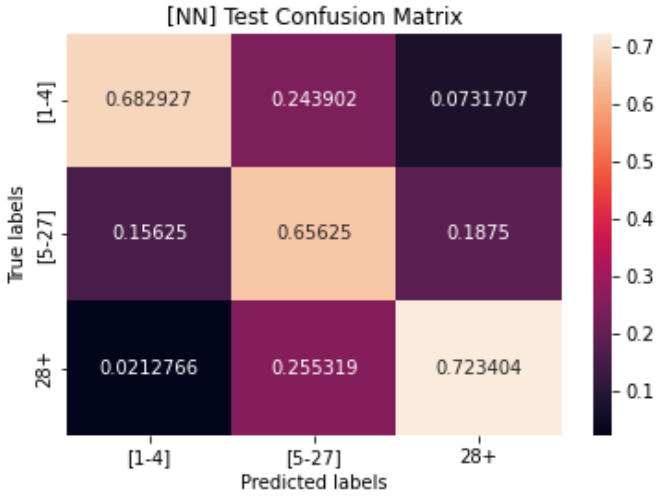

Figura 24: Matriz de confusión modelo 2 sobre conjunto de prueba

Este modelo alcanza una `accuracy` de 0.7 en el conjunto de prueba.

6. Análisis de resultados

En primer lugar se puede apreciar que el mecanismo de `early_stopping` funciona como se esperaba, en las **Figuras 7, 9, 11, 13, 16, 18, 20 y 22** se puede apreciar que, pese a que se indicaron 700 rondas de entrenamiento, el entrenamiento se detuvo antes, cuando la curva de validación comienza a aumentar. También se puede apreciar que los modelos habrían sufrido `over-fitting` si simplemente se usase el último modelo producto de entrenar una cantidad fija de iteraciones, pues en todas las figuras las curvas del error de validación y de entrenamiento divergen rápidamente.

En dichas figuras la linea verde vertical representa al modelo que obtuvo el mejor error en el conjunto de validación, los cuales en todos los casos se encontraron unas cuantas épocas antes de que se detuviera el entrenamiento, esto se debe a que el ruido en el entrenamiento volvió difícil que el error de validación aumentase durante 5 épocas consecutivas. Esto se podría remediar disminuyendo la cantidad de rondas necesarias para detener el entrenamiento.

En cuanto a las predicciones realizadas por los modelos, se puede apreciar que, tanto para LBP 8_1 como para LBP12_2, en los cuatro casos (**Figuras 8, 10, 12, 14, 17, 19, 21 y 23**) los modelos prácticamente no cometieron errores en el conjunto de entrenamiento y en particular el mejor modelo, el 2, no cometió ningún error en dicho conjunto. Esto es de esperarse puesto que es sabido que una red neuronal tarde o temprano va a aprender todos los ejemplos del conjunto de entrenamiento, ya sea porque reconoce los patrones o porque simplemente memoriza.

Pero esto no es muy relevante, puesto que lo que se desea es que los modelos sean capaces de generalizar, así que hay que prestar mayor atención a los resultados en el conjunto de validación. Los resultados que obtuvieron los modelos LBP 8_1 son aceptables, llegando a alcanzar una `accuracy` de valdiación de 0.675 en el mejor modelo, mientras que los modelos LBP 12_2 son un tanto decepcionantes puesto que ninguno logra una `accuracy` mayor a 0.6.

Se puede apreciar que todos los modelos tuvieron mayores errores al confundir el grupo del medio con los extremos, es decir, confundir [5-27] con [1-4] o +28. Esto tiene sentido si se piensa en los casos bordes, caras de personas entre 5 a 10 años se parecen mucho más a personas entre 1 y 4 que a personas entre 11 a 27. Lo mismo pasa para el otro lado, personas mayores a 20 se parecen mucho más a mayores de 28 que a personas entre 5 a 14 años. Esta teoría también se sustenta observando que casi no hay errores de confundir [1-4] con 28+. Esto se podría solucionar separando más las clases de los datos, para que haya más consistencia entre personas del mismo conjunto.

En ambos casos, LBP 8_1 y LBP 12_2, el mejor modelo es el 2 (128 neuronas y 0.0001 lr). La `accuracy` de validación del modelo 2 es mayor en LBP 8_1 que en LBP 12_2 (0.675 vs 0.59), sin embargo, la `accuracy` en validación es mayor para 12_2 que para 8_1 (0.65 vs 0.7). Dado que se deben tratar los datos de prueba como si no se tuviesen, es decir, que no deben influenciar las decisiones que se toman en cuanto a los modelos, y que de todas formas ambos obtienen `accuracies` bastante similares, se puede afirmar que el mejor modelo corresponde al modelo 2 de LBP 8_1. Cabe destacar que dicho modelo sigue sin superar los resultados obtenidos en la Tarea 3 con las características HOG (que llegaron a 0.84 de `accuracy` en el conjunto de prueba.)

En cuanto a los hiperparámetros de los modelos, se puede apreciar que aumentar la cantidad de neuronas de la capa oculta vuelve más poderosos a los modelos, pues los modelos 2 y 3 son consistentemente mejores que los modelos 0 y 1. Por otra parte, una tasa de aprendizaje muy pequeña implica un entrenamiento más largo, estos modelos tuvieron entrenamientos de 300 o 500 rondas versus las 50 rondas de los otros modelos, también dificulta el mecanismo de `early_stopping` puesto que con una tasa de aprendizaje muy pequela hay mayor ruido en en el entrenamiento y por ende es más difícil que el error de validación aumente consistentemente.

Por lo general se obtienen mejores y más rápidos resultados con la tasa de aprendizaje mayor, pero se presume que tasas de aprendizaje muy grandes podrían traer resultados aún peores.

7. Conclusión

Se logra implementar en Python todas las funciones necesarias tanto para el cálculo de las características LBP, como para el entrenamiento y predicción de las redes neuronales. Se concluye que las características LBP no son las mejores para la tarea de detectar rostros y que una opción preferible son las HOG.

Se conlcuye que las características LBP 8_1 son mejores, por lo menos en la tarea de clasificar rostros, que las características LBP 12_2. También se intuye que se obtendrían mejores resultados separando de mejor forma las clases de los datos y sofisticando la red neuronal, es decir, añadiendo más neuronas/capas y manteniendo una tasa de aprendizaje no muy pequeña. También se podrían probar otros tamaños para el batch size, y otra cantidad de rondas para el criterio de detención del entrenamiento.

Se logró utilizar de manera satisfactoria la librería pytorch para todo lo que implicaba el entrenamiento de la red neuronal y también se logró implementar de manera satisfactoria un mecanismo de early_stopping.

8. Anexo

8.1. Extracción de características LBP

```
cpdef np.npy_bool isUniform(str string):
  cdef int recorre = 0
  cdef int revisa = 0
  cdef int cambios = 0
  cdef int stop = len(string)
  while recorre < stop:
    if string[revisa] == string[recorre]:
      recorre += 1
    else:
      revisa = recorre
      recorre += 1
      cambios += 1
    if cambios > 2:
      return False
  return True
```

Listing 10: isUniform

```
cpdef np.double_t binaryToInt(str seq):
  cdef int i
  cdef int stop = len(seq)
  cdef float total = 0
  for i in range(stop - 1, -1, -1): # for de n a 0 para el exponente de 2 pero
   elementos de la seq van al reves
    if int(seq[(stop  - 1) - i]) == 1:
      total += 2**(i)
  return total
```

Listing 11: binaryToInt

```
cdef tuple sumTuples(tuple tup1, tuple tup2):
  return (tup1[0] + tup2[0], tup1[1] + tup2[1])
```

Listing 12: sumTuples

```
cpdef np.ndarray[np.double_t, ndim=2] LBP(np.ndarray[np.double_t, ndim=2] img, int
    p, float r):
     cdef int imgCols = img.shape[0]
     cdef int imgRows = img.shape[1]
     cdef np.ndarray[np.double_t, ndim=2] out = np.zeros((imgCols,imgRows))
     cdef int step = int(360/p)
     cdef list points = []
     cdef int y, x, theta, py, px
     cdef tuple point
     cdef str word

     for theta in range(180,-180,-step):
       points.append( (round(r*np.cos(np.deg2rad(theta))), round(r*np.sin(np.deg2rad
    (theta)))) )
     points.append(points.pop(0))

     for y in range(imgCols):
       for x in range(imgRows):
         word = ""
         for point in points:
           py, px = sumTuples(point,(y,x))
           if (min(py,px) < 0) or ((py >= imgCols) or (px >= imgRows)):
             word += "0"
             continue
           if img[py, px] >= img[y,x]:
             word += "1"
           else:
             word += "0"
         if isUniform(word):
           out[y,x] = binaryToInt(word)
         else:
           out[y,x] = float(5)
     return out
```

Listing 13: LBP

```
cpdef np.ndarray[np.double_t, ndim = 1] makeUniformList(int binLen):
  cdef list uniformList = []
  cdef int num = 2**binLen
  cdef int i
  for i in range(num):
    if isUniform(bin(i)[2:].zfill(binLen)):
      uniformList.append(i)
  uniformList.append(5) # primer numero no uniforme 00000101
  return np.array(uniformList, np.double)
```

Listing 14: makeUniformList

```
cpdef np.ndarray[np.double_t, ndim=1] LBPHistogram(np.ndarray[np.double_t, ndim=2]
    lbp, int binLen):
  cdef np.ndarray[np.double_t, ndim=1] unique = makeUniformList(binLen)
  cdef np.ndarray[np.double_t, ndim=2] sec
  cdef np.ndarray[np.double_t, ndim=1] secHist
  cdef np.ndarray[np.double_t, ndim=1] out = np.array([])
  cdef int lbpCols = lbp.shape[0]
  cdef int lbpRows = lbp.shape[1]
  cdef int vStep = int(lbpCols/2)
  cdef int hStep = int(lbpRows/2)
  cdef int y,x

  for y in range(2):
    for x in range(2):
      sec = lbp[vStep*y:vStep*(y+1), hStep*x:hStep*(x+1)]
      secHist = np.zeros(len(unique))
      for i in range(len(unique)):
        secHist[i] = (sec == unique[i]).sum()
      out = np.concatenate((out, secHist))
  return out
```

Listing 15: LBPHistogram

8.2. Entrenar clasificadores

```
class Network(nn.Module):

    def __init__(self, input_size, num_classes, hidden_neurons):
        super().__init__()

        self.fc1 = nn.Linear(input_size, hidden_neurons)
        self.fc2 = nn.Linear(hidden_neurons, num_classes)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Listing 16: Clase Network

```
def makeNN(input_size, num_classes, hidden_neurons):
  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
  return Network(input_size, num_classes, hidden_neurons).to(device), device
```

Listing 17: makeNN

```
def makeLoaders(data_train, data_val, data_test, batch_size):
  train_loader = DataLoader(dataset = data_train, batch_size = batch_size, shuffle
    = True)
  val_loader = DataLoader(dataset = data_val, batch_size = batch_size, shuffle =
    True)
  test_loader = DataLoader(dataset = data_test, batch_size = batch_size, shuffle =
    True)
  return train_loader, val_loader, test_loader
```

Listing 18: makeLoaders

```
def trainNN(model, learning_rate, train_loader, val_loader, num_epochs, device,
    es_rounds):
  # Checkpoint de loss y modelos
  train_losses = []
  val_losses = []
  models = []

  # Variables early stopping
  down_counter = 0

  # criterio de loss y optimizador
  criterion = nn.CrossEntropyLoss()
  optm = Adam(model.parameters(), lr = learning_rate)

  # Entrenamiento
  for epoch in range(num_epochs):
    model.train()
    train_loss = 0.0
    for index, (d_train, l_train) in enumerate(train_loader): # para cada batch de
    train
      d_train = d_train.to(device) # train data
      l_train = l_train.to(device) # train labels

      # limpiar gradientes
      optm.zero_grad()

      # forwards
      train_scores = model(d_train.float())
      loss = criterion(train_scores, l_train)

      # backwards
      loss.backward()
      optm.step()
      train_loss += loss.item()

    # Monitoreo validacion
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
      for index, (d_val, l_val) in enumerate(val_loader): # para cada batch de
    validacion
        d_val = d_val.to(device) # validation data
        l_val = l_val.to(device) # validation labels

```

```
        val_scores = model(d_val.float())
        loss = criterion(val_scores, l_val)
        val_loss += loss.item()

    # Loss de entrenamiento y validacion de la epoca
    train_loss = train_loss/len(train_loader)
    val_loss = val_loss/len(val_loader)

    # Comunicar estado actual del modelo
    print(f'Epoch {epoch+1} \t Training Loss: {train_loss} \t Validation Loss: {val_loss}')

    # Early Stopping
    with warnings.catch_warnings():
      warnings.simplefilter("ignore", category=RuntimeWarning)
      if (val_loss > np.mean(val_losses)) and (train_loss < np.mean(train_losses)):
        down_counter += 1
      else:
        down_counter = 0
      if down_counter > es_rounds:
        print(f'Validation loss rising, stopping training!')
        break
    # Guardar losses y modelo de la epoca
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    models.append({'epoch': epoch, 'train_loss': train_loss, 'val_loss': val_loss, 'model': model.state_dict()})
  return train_losses, val_losses, models
```

Listing 19: trainNN

```
def getBestModel(models):
  best_loss = np.inf
  best_model = models[0]
  last_epoch = 0
  for i in range(len(models)):
    if models[i]['val_loss'] < best_loss:
      best_loss = models[i]['val_loss']
      best_model = models[i]['model']
      last_epoch = models[i]['epoch']
  return best_loss, best_model, last_epoch
```

Listing 20: getBestModel

```python
def drawLossCurves(models, best_loss, train_losses, val_losses, last_epoch, index):
  fig = plt.figure(figsize = (14,7))
  plt.title("Train/Val Cross Entropy Loss")
  plt.xlabel("Epochs")
  plt.ylabel("Loss")
  plt.plot(range(len(models)), train_losses, label = 'train_loss')
  plt.plot(range(len(models)), val_losses, label = 'val_loss')
  plt.vlines(x = last_epoch, ymin= models[-1]['train_loss'], ymax= best_loss, color
    ='g', label = 'early stopping model')
  plt.legend()
  plt.savefig(f"TrainVal_CEL_model_{index}.png")
  plt.show()
```

Listing 21: drawLossCurves

```python
def makePredictions(model, data):
  predictions = []
  model.type(torch.DoubleTensor)
  model.to('cpu')
  model.eval()
  for i in range(len(data)):
    input = torch.tensor(np.double(data[i])).type(torch.DoubleTensor)
    input.to('cpu')
    pred = model(input)
    predictions.append((pred == torch.max(pred)).nonzero(as_tuple=True)[0].item())
  return predictions
```

Listing 22: makePredictions

```python
def drawConfusionMatrix(y_true, y_pred, title):
  cm = confusion_matrix(y_true, y_pred)
  ax= plt.subplot()
  sns.heatmap(cm, annot=True, fmt='g', ax=ax)
  ax.set_xlabel('Predicted labels')
  ax.set_ylabel('True labels')
  ax.set_title(f'{title} Confusion Matrix')
  ax.xaxis.set_ticklabels(['[1-4]', '[5-27]', '28+'])
  ax.yaxis.set_ticklabels(['[1-4]', '[5-27]', '28+'])
  plt.savefig(f'{title}_CM.png')
```

Listing 23: drawConfusionMatrix

```python
def stats(model, index):
  train_pred = makePredictions(model, LBP_X_train)
  val_pred = makePredictions(model, LBP_X_val)

  train_acc = drawConfusionMatrix(y_train, train_pred, "[NN] Train", index)
  val_acc = drawConfusionMatrix(y_val, val_pred, "[NN] Val", index)
  return val_acc
```

Listing 24: stats

Referencias

[1] Apunte de las clases de Procesamiento Avanzado de Imágenes - EL7008 - 1, Primavera 2021, Javier Ruiz del Solar.

[2] Apunte de las clases de Inteligencia Computacional - EL4106-1, Primavera 2020, Pablo Estevez V.

[3] Tarea 4 - EL7008 (Primavera 2021): Clasificación de edad usando LBP y redes neuronales, Javier Ruiz del Solar, Patricio Loncomilla.