*

Tarea 5:
Redes neuronales convolucionales para clasificar edad

Fecha de entrega: Domingo 28 de noviembre, 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 5 del curso EL7008-1 del semestre de primavera de 2021. Este informe comprende la implementación computacional de mecanismos que permitan leer, cargar y aumentar un conjunto de imágenes, así como también entrenar una red neuronal convolucional con *backbone* `InceptionResnetV1` con distintos hiper-parámetros.

El principal objetivo de este informe consiste en construir implementar bajo la librería `pytorch` de `Python` dos clases, una que se encargue de leer las imágenes entregadas por el cuerpo docente y aplicarles *data augmentation*, y otra que se encargue de todo lo relacionado al entrenamiento, resultados y analíticas de la red neuronal convolucional. La red `InceptionResnetV1` se importa desde otro proyecto y hay que aplicarle mínimos cambios

La **Sección 2** contiene un resumen del contexto necesario para entender las implementaciones realizadas en la tarea, esto incluye el funcionamiento de las redes neuronales convolucionales y las técnicas de *data augmentation*. En las **Sección 3** se detallan las clases implementadas en Python junto a sus respectivos métodos, todo adjunto en el notebook Tarea5.ipynb, necesarias para el desarrollo de la tarea. 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. Redes Neuronales Convolucionales

Las redes neuronales convolucionales son un tipo de red neuronal que realizan al mismo tiempo el trabajo de la extracción de características y de clasificación, como se muestra en la **Figura 1**

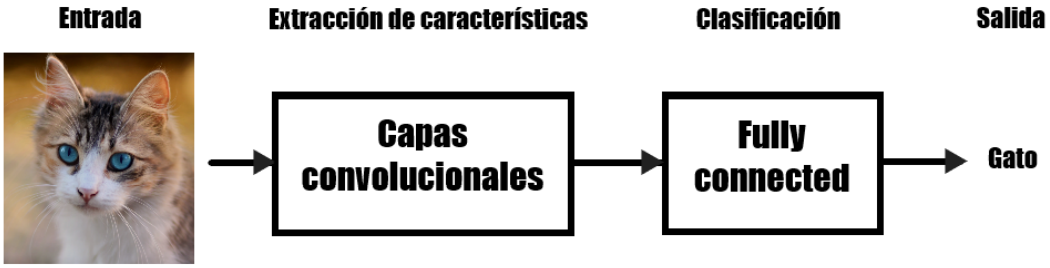

Figura 1: Red convolucional neuronal

Esto lo logran puesto que las capas convolucionales se encargan, mediante muchas convoluciones en paralelo, de extraer los *feature maps*, los cuales son entregados a las capas *fully connected* que son redes neuronales tradicionales. En esta tarea se utiliza la red neuronal convolucional "Inception-resnet-v1", pre-entrenada con el conjunto de datos "vggface2".

2.2. Aumento de datos

El aumento de datos, o *data augmentation* en inglés, corresponde a una técnica que aumenta la cantidad de datos disponibles aplicando pequeñas transformaciones a las imágenes originales. En esta tarea se utilizan 4 tipos de transformaciones:

1. Color Jitter, aplica leves cambios aleatorios de brillo, contraste, saturación y matiz a la imagen de entrada
2. Crop, recorta la imagen a un tamaño en particular, el recorte puede ser en una zona completamente aleatoria (*Random Crop*) o puede aplicarse desde el centro de la imagen de entrada (*Center Crop*).
3. Flip, a algunas imágenes, dependiendo de una probabilidad, se genera una versión girada de la imagen de entrada. En esta tarea se utiliza un giro horizontal (*Random Horizontal Flip*).
4. Normalize, aplica una normalización a las imágenes, basada en el promedio y la desviación aleatoria de las imágenes del conjunto de entrenamiento, para que sea más fácil el entrenamiento de la red.

Si bien la misión principal del *data augmentation* consiste en aumentar la cantidad de datos disponibles para entrenar, y ajustar la red, en esta tarea se opta por aplicar en cadena las transformaciones y reemplazar las imágenes originales. Es decir, no se aumenta la cantidad de datos, pero sí se aumenta la variabilidad entre ellos, pues la red tiene que enfrentarse a variaciones en iluminación, orientación, tamaño, etc.

3. Estructura base

3.1. Cargar los datos

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.

Sin embargo, a diferencia de en las tareas anteriores, en esta tarea la carga de datos se hace mediante una clase especial para leer los datos y por ende la separación en conjuntos de entrenamiento, validación y prueba también es manejada por dicha clase, la cual se presenta en la **Sección 3.2**.

3.2. Lector de la base de datos

Para tratar los datos ordenada y eficientemente se crea una clase `Data` que hereda de la clase `Dataset`. Esta es una práctica común cuando se trabaja con `pytorch`, cuando se crean estas clases hijas de `Dataset` basta con reemplazar tres métodos:

1. `__init__(self, ind0, ind1)`: Este método se encarga de inicializar el objeto `Data`, leyendo todas las imágenes que se encuentren entre los índices `[ind0, ind1]`, para los tres rangos de edad, al mismo tiempo.
2. `__len__(self)`: Este método se encarga de retornar la cantidad de datos que almacena el objeto, valor que en esta implementación se almacena en la variable interna `size`.
3. `__getitem__(self, idx)`: Este método se encarga de entregar un dato y su etiqueta según el índice de entrada `idx`. Dado que los datos se almacenan en la lista interna `data` y las etiquetas en la lista interna `labels`, basta con retornar la indexación de cada lista.

Cabe destacar que la separación entre conjuntos de entrenamiento, validación y prueba se realiza instanciando tres objetos de la clase `Data`, escogiendo adecuadamente los índices. Dado que las imágenes cargadas vienen en un orden aleatorio no hay que preocuparse de si una clase quedará sub-representada en algún conjunto.

Para recoger el 60 % de un total de 200 imágenes en el conjunto de entrenamiento se deben escoger los índices:

$$[0, 0{,}6 \cdot 200] = [0, 119]$$

Para recoger el siguiente 20 % de los datos en el conjunto de validación:

$$[120, (0{,}6 + 0{,}2) \cdot 200] = [120, 159]$$

Y el 20 % restante va al conjunto de prueba en los índices restantes [160, 199].

3.3. Aumento de datos

Como se explicó en la **Sección 2.2** se utilizan 5 tipos de transformaciones, todas ellas del paquete `torchvision.transforms` e implementadas en una clase llamada `Transformer`:

1. `ColorJitter(brightness, contrast, saturation)`: Modifica aleatoriamente el brillo, el contraste y la saturación de la imagen de entrada. Los probabilidades de cada cambio se escogen como `brightness` = `contrast` = `saturation` = 0.4.
2. `RandomCrop(size)`: Recorta la imagen en una zona aleatoria de tamaño `size` = 140.
3. `CenterCrop(size)`: Recorta la imagen en el centro, con un tamaño `size` = 140.
4. `RandomHorizontalFlip(p)`: Tiene una probabilidad `p` de girar horizontalmente la imagen de entrada. Se escoge la probabilidad predeterminada de `p` = 0.5
5. `Normalize(mean, std)`: Aplica una normalización a la imagen de entrada. Esta normalización se calcula en base a las imágenes originales del conjunto de entrenamiento, donde `mean` contiene el promedio de cada canal y `std` la desviación estándar de cada canal.

En esta tarea se utilizan las transformaciones 1, 2, 4 y 5 para el conjunto de entrenamiento, y las transformaciones 3 y 5 para los conjuntos de prueba y validación. Para concatenar estas transformaciones conviene utilizar la función `Compose` de `torchivision.transforms` que aplica transformaciones en orden secuencial:

```
self.trainTF = Compose([
                ColorJitter(brightness = 0.4, contrast = 0.4, saturation = 0.4),
                RandomCrop(140),
                RandomHorizontalFlip(),
                Normalize(self.train_mean, self.train_std)
                ])
self.vtTF = Compose([
                CenterCrop(140),
                Normalize(self.train_mean, self.train_std)
                ])
```

Los valores `self.mean` y `self.std` se calculan mediante el método `computeMeanStd(self, training_data)` (**Listing 5**), que recorre todas las imágenes en el conjunto de entrenamiento y calcula su promedio con su desviación estándar.

Para aplicar las transformaciones a las imágenes de un objeto de la clase `Data` se crea el método `augment(self, data, train)` (**Listing 6**) que recibe el objeto `data` y un flag booleano `train` para decidir qué `Compose` de transformaciones se utiliza.

Por último cabe destacar que se deben re-dimensionar las imágenes transformadas a (160, 160) para que así puedan ser aceptadas por la red. Esto se realiza mediante el método `resize(self, image)` (**Listing 7**) que separa y re-dimensiona los tres canales de la imagen de entrada `image`, los combina en un solo array y lo retorna en forma de tensor.

Según lo presentado en las **Secciones 3.2 y 3.3** la carga y separación de datos se realiza mediante:

```
train = Data(0, 119)
val = Data(120,159)
test = Data(160,199)

transformer = Transformer(train)
transformer.augment(train, True)
transformer.augment(val, False)
transformer.augment(test, False)
```

3.4. Modelo

Para realizar el entrenamiento del modelo se reutiliza la misma estructura utilizada en la Tarea 4, con la única diferencia siendo que al crear la red se instancia un objeto de la clase `InceptionResnetV1`:

```
def makeNN(pretrained, classify, num_classes):
  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
  return InceptionResnetV1(pretrained, classify, num_classes).to(device), device
```

Dado que esta estructura ya fue explicada en extensión en el informe de la Tarea 4, no se explicará en detalle en el presente informe, pero sí se entrega un breve resumen del propósito de cada función:

1. `makeNN`: Inicializa la `InceptionResnetV1` y escoge el `device` que se utilizará (cpu o gpu).
2. `makeLoaders`: Crea los loaders para cada conjunto de datos, con un cierto `batch_size` y reordenando aleatoriamente las muestras.
3. `trainNN`: Se encarga de realizar el entrenamiento de la red, monitoreando la pérdida de entrenamiento y validación en cada época y guardando *checkpoints* de cada iteración del modelo.
4. `getBestModel`: Recibe una lista de diccionarios que contienen los parámetros de un modelo en una época junto con sus pérdidas de entrenamiento y validación, retorna el mejor modelo según la pérdida de validación.
5. `drawLossCurves`: Se encarga de graficar las curvas de pérdida de entrenamiento y validación del entrenamiento del modeo, también dibuja una línea vertical verde en la época donde se encuentra el mejor modelo.
6. `makePredictions`: Obtiene las predicciones de un modelo sobre un conjunto de datos y les da formato de lista.
7. `drawConfusionMatrix`: Recibe listas de etiquetas reales y predichas, genera la matriz de confusión asociada.
8. `stats`: Dibuja la matriz de confusión para los conjuntos de entrenamiento, validación y prueba.

4. Experimentos

Para analizar el desempeño de la red entrenada bajo distintos hiper-parámetros se realizan 7 experimentos, los cuales serán descritos a continuación:

1. Entrenar la red neuronal congelando las capas convolucionales anteriores a `block8(x)`, con `batch_size` = 32 y usando *data augmentation*.
2. Repetir el experimento (1) pero con `batch_size` = 8.
3. Repetir el experimento (1) pero sin *data augmentation*.
4. Repetir el experimento (1) pero sin congelar capas.
5. Repetir el experimento (1) pero utilizando el optimizador Adam con una tasa de aprendizaje distinta.
6. Repetir el experimento (1) pero utilizando un optimizador SGD.
7. Repetir el experimento (1) pero utilizando un optimizador SGD con una tasa de aprendizaje distinta.

Se debe analizar el desempeño de cada una de las configuraciones descritas y luego comparar el desempeño de las CNN vs los modelos en base a HOG y LBP creados en las Tareas 3 y 4.

5. Resultados

En esta sección se presentan los resultados de los experimentos mencionados en la **Sección 4**:

5.1. Experimento 1

Las curvas de pérdida de entrenamiento y validación del experimento 1 se presentan en la **Figura 2**:

Figura 2: Curvas de pérdida conjuntos de entrenamiento y validación experimento 1

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento, validación y prueba se obtienen las matrices de confusión presentadas en las **Figuras 3, 4 y 5**:

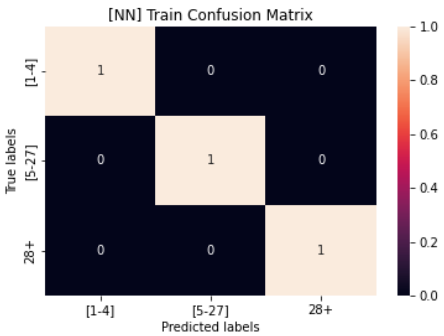

Figura 3: MC train modelo 1

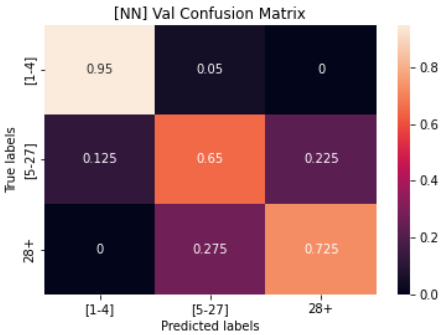

Figura 4: MC val modelo 1

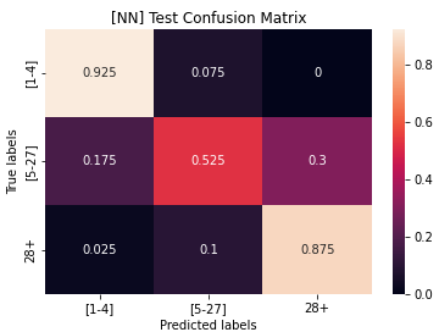

Figura 5: MC test modelo 1

5.2. Experimento 2

Las curvas de pérdida de entrenamiento y validación del experimento 2 se presentan en la **Figura 6**:

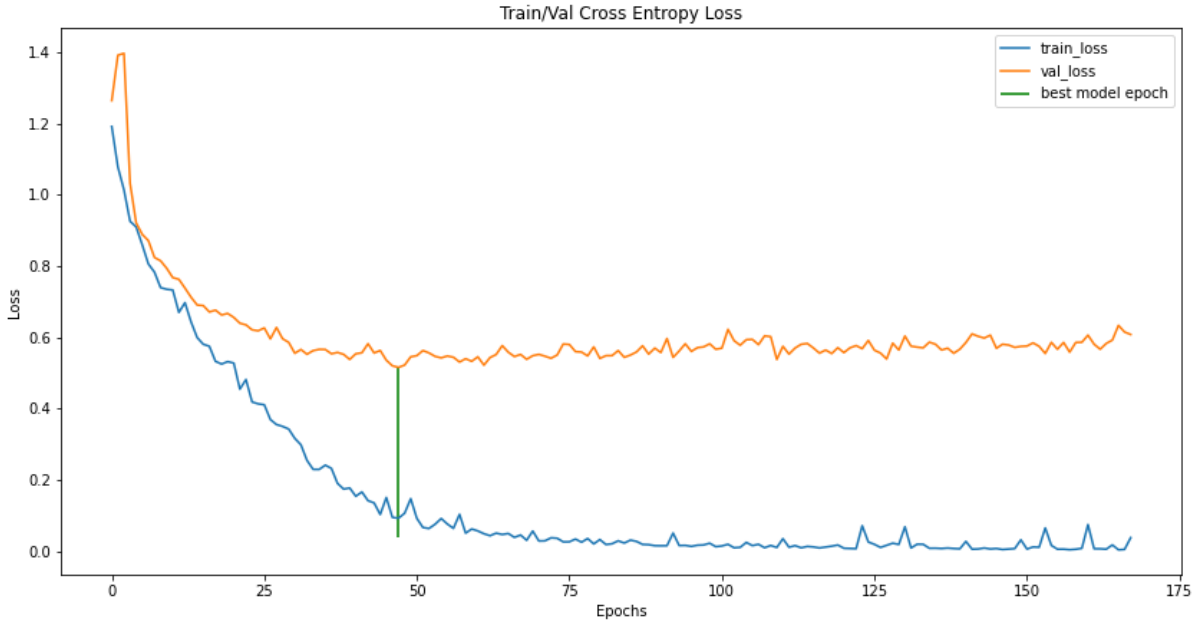

Figura 6: Curvas de pérdida conjuntos de entrenamiento y validación experimento 2

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento, validación y prueba se obtienen las matrices de confusión presentadas en las **Figuras 7, 8 y 9**:

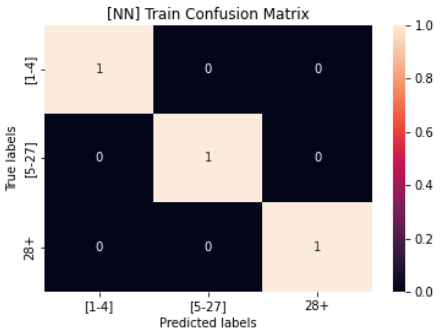

Figura 7: MC train modelo 2

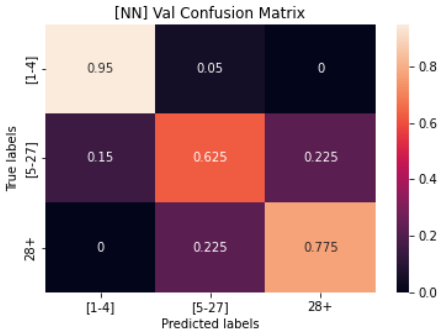

Figura 8: MC val modelo 2

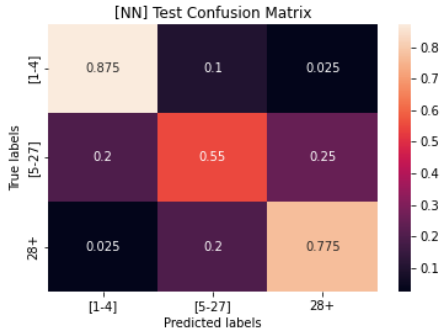

Figura 9: MC test modelo 2

5.3. Experimento 3

Las curvas de pérdida de entrenamiento y validación del experimento 3 se presentan en la **Figura 10**:

Figura 10: Curvas de pérdida conjuntos de entrenamiento y validación experimento 3

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento, validación y prueba se obtienen las matrices de confusión presentadas en las **Figuras 11, 12 y 13**:

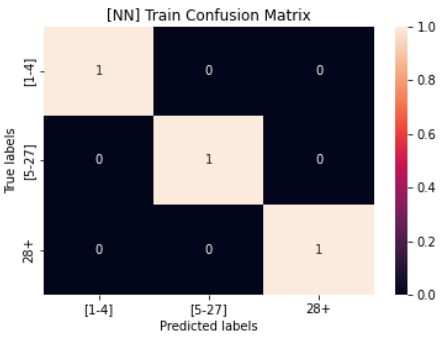

Figura 11: MC train modelo 3

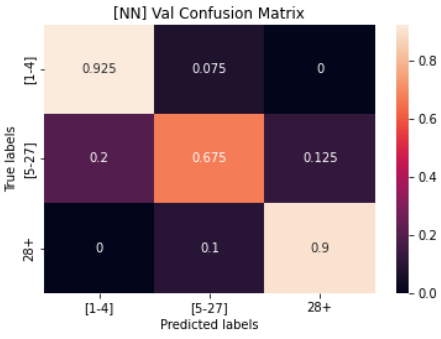

Figura 12: MC val modelo 3

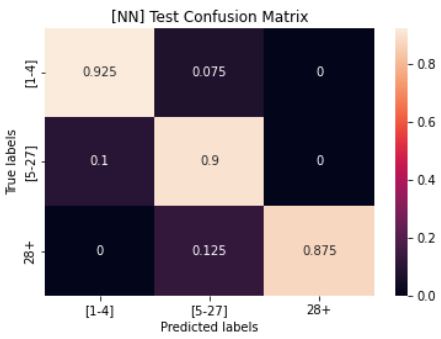

Figura 13: MC test modelo 3

5.4. Experimento 4

Las curvas de pérdida de entrenamiento y validación del experimento 4 se presentan en la **Figura 14**:

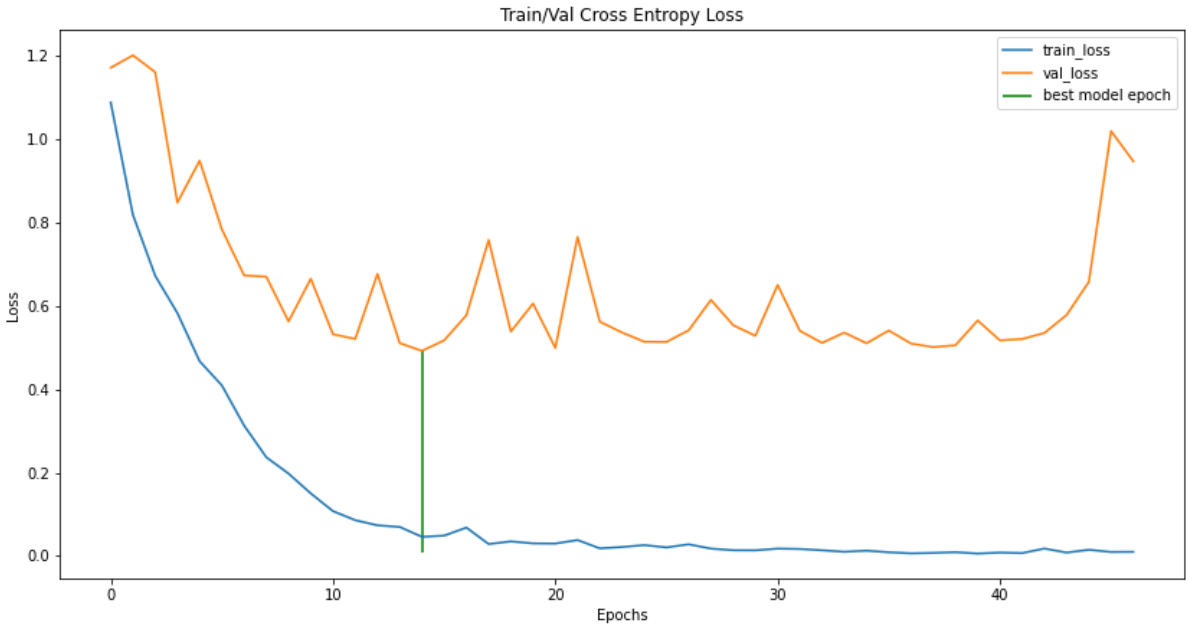

Figura 14: Curvas de pérdida conjuntos de entrenamiento y validación experimento 4

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento, validación y prueba se obtienen las matrices de confusión presentadas en las **Figuras 15, 16 y 17**:

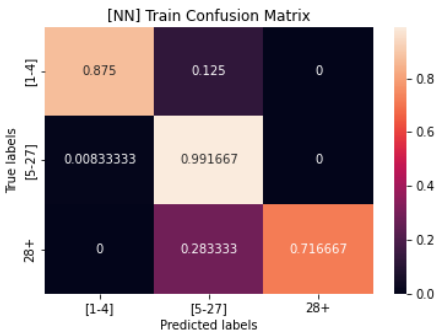

Figura 15: MC train modelo 4

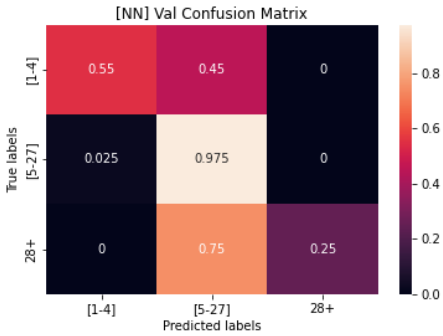

Figura 16: MC val modelo 4

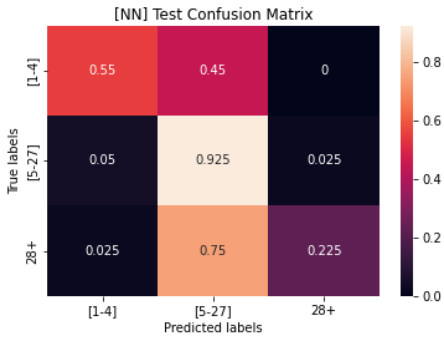

Figura 17: MC test modelo 4

5.5. Experimento 5

Las curvas de pérdida de entrenamiento y validación del experimento 5 se presentan en la **Figura 18**:

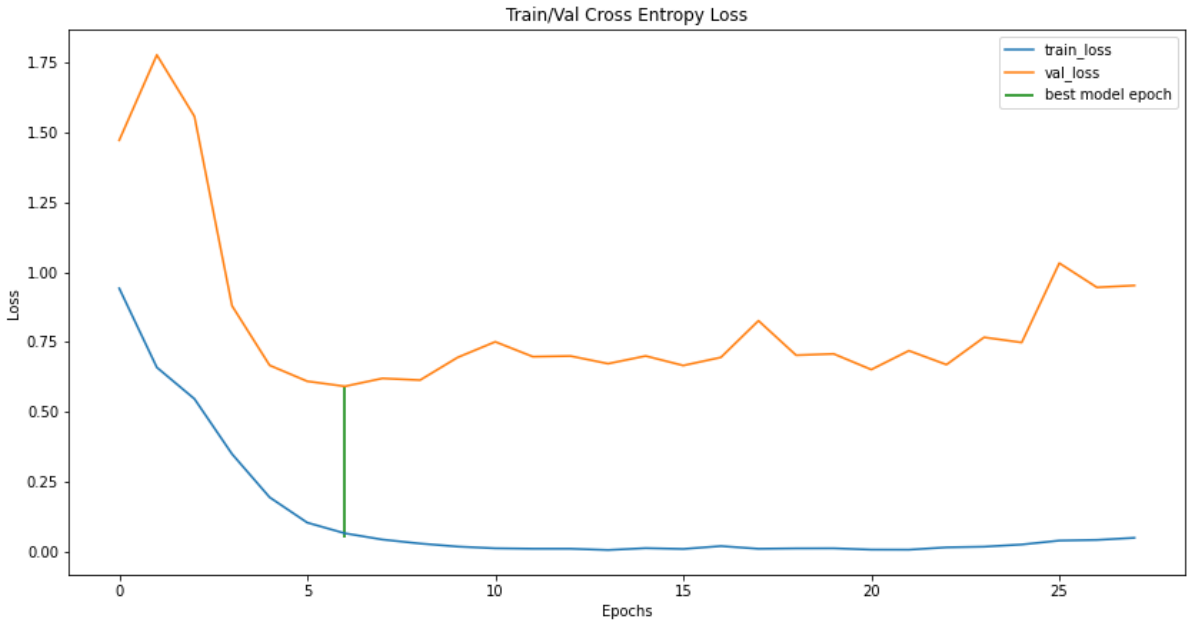

Figura 18: Curvas de pérdida conjuntos de entrenamiento y validación experimento 5

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento, validación y prueba se obtienen las matrices de confusión presentadas en las **Figuras 19, 20 y 21**:

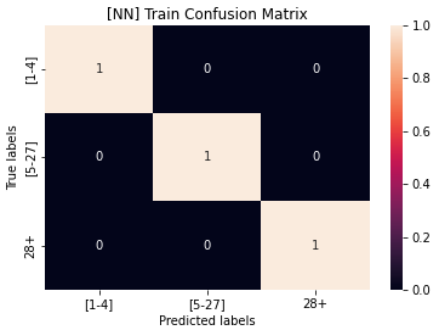

Figura 19: MC train modelo 5

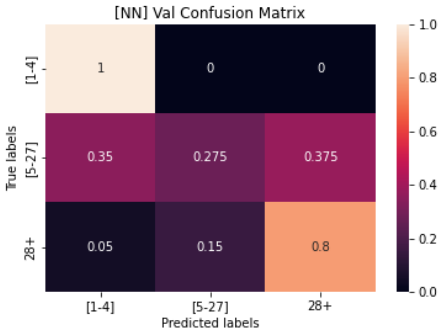

Figura 20: MC val modelo 5

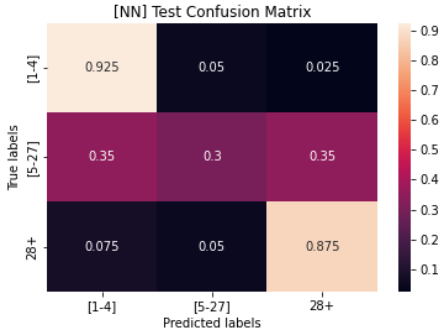

Figura 21: MC test modelo 5

5.6. Experimento 6

Las curvas de pérdida de entrenamiento y validación del experimento 6 se presentan en la **Figura 22**:

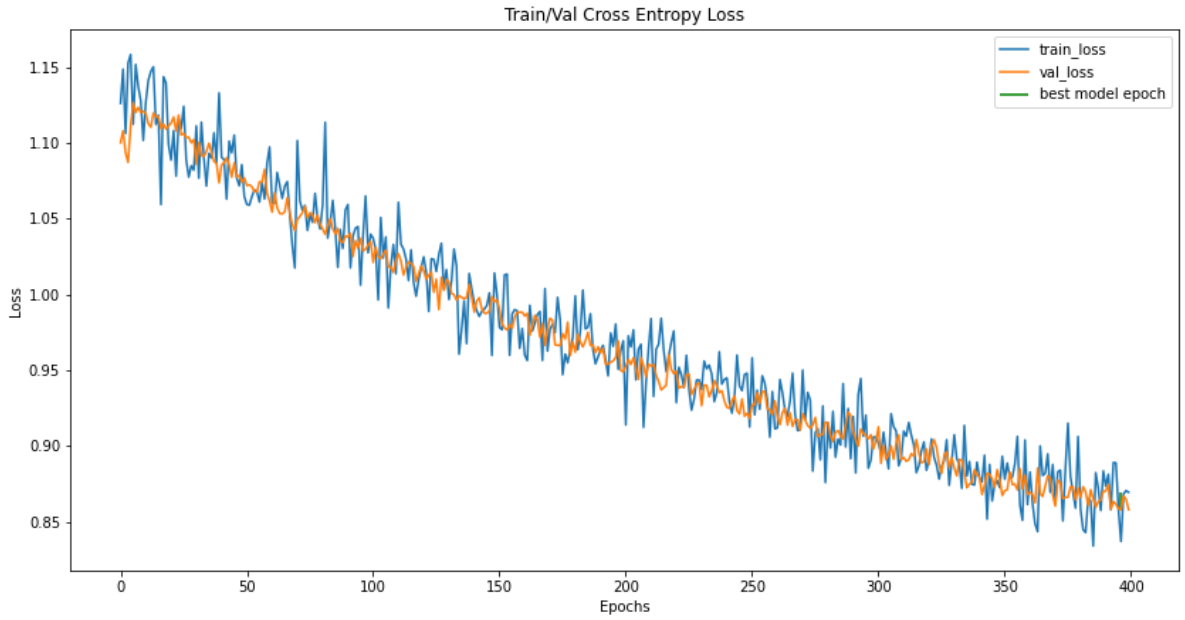

Figura 22: Curvas de pérdida conjuntos de entrenamiento y validación experimento 6

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento, validación y prueba se obtienen las matrices de confusión presentadas en las **Figuras 23, 24 y 25**:

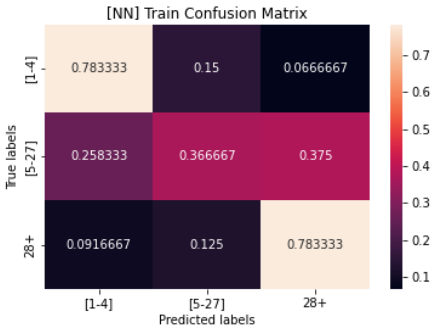

Figura 23: MC train modelo 6

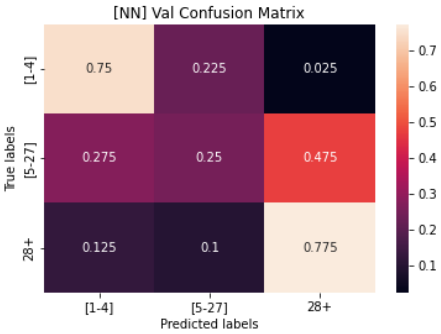

Figura 24: MC val modelo 6

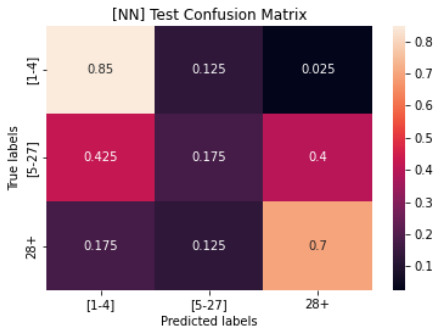

Figura 25: MC test modelo 6

5.7. Experimento 7

Las curvas de pérdida de entrenamiento y validación del experimento 7 se presentan en la **Figura 26**:

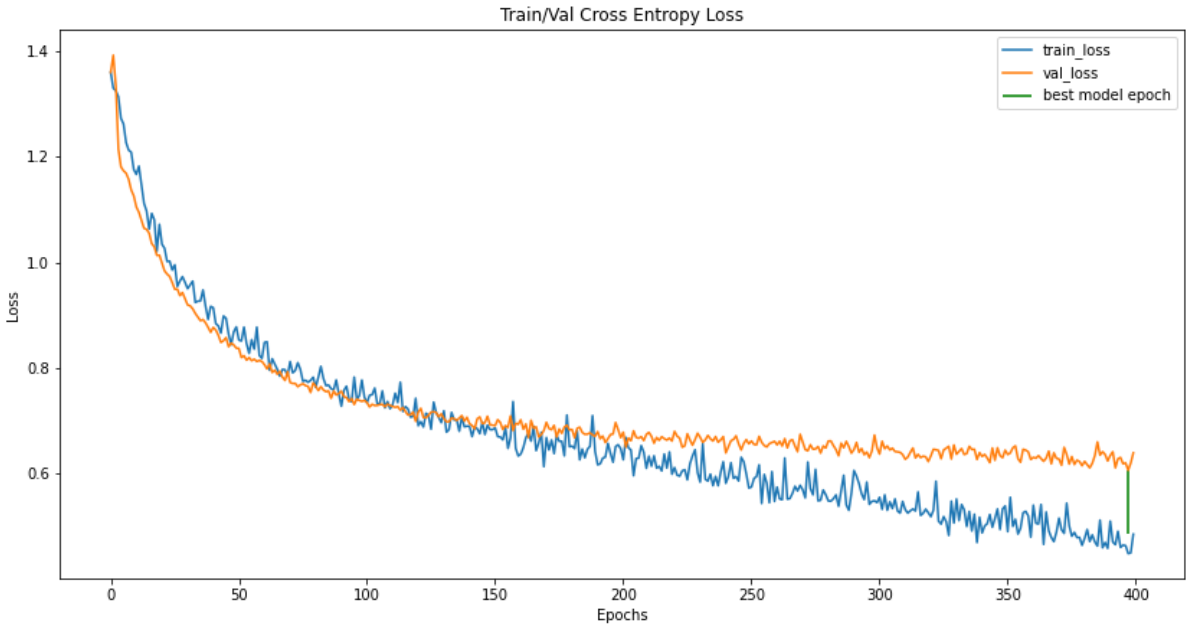

Figura 26: Curvas de pérdida conjuntos de entrenamiento y validación experimento 7

Al realizar las predicciones con este modelo sobre los conjuntos de entrenamiento, validación y prueba se obtienen las matrices de confusión presentadas en las **Figuras 27, 28 y 29**:

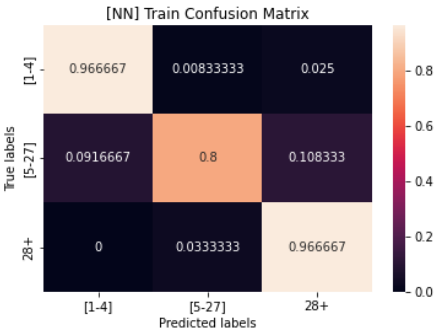

Figura 27: MC train modelo 7

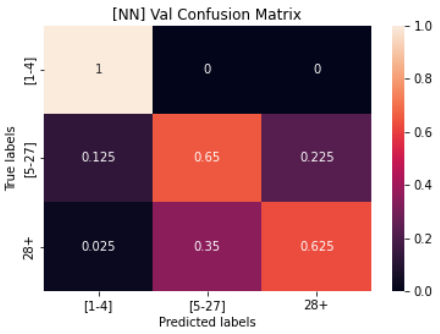

Figura 28: MC val modelo 7

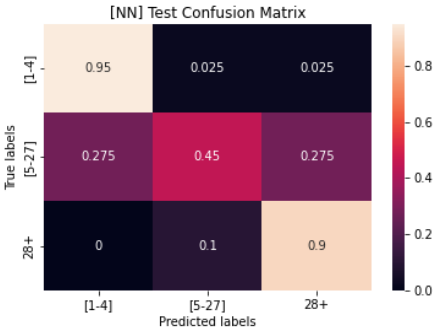

Figura 29: MC test modelo 7

6. Análisis de resultados

6.1. Análisis Experimento 1

Este experimento se establece como el *baseline* contra el cual se comparan el resto de los experimentos. Se puede apreciar en la **Figura 2** que en tan sólo 40 épocas se encuentra el mejor modelo con una pérdida de validación cercana a 0.6. Este modelo tiene un gran desempeño en las clases más extremas, es decir, puede distinguir muy bien personas entre (1 - 4) y personas mayores a 28, pero le cuesta más diferenciar las edades intermedias (5 - 27) como se aprecia en las **Figuras 4 y 5**.

Pese a la dificultad recién descrita para este modelo, y considerando que se entrena con apenas 360 imágenes, los resultados son aceptables llegando a (0.73 de *accuracy* en validación y 0.82 en prueba).

6.2. Análisis Experimento 2

Disminuyendo el `batch_size` a 8. Se puede apreciar en la **Figura 6** que el mejor modelo también se encuentra bastante rápido, en tan sólo 50 épocas y con una pérdida de validación cercana a 0.5. Se repite el mismo patrón (**Figuras 8 y 9**) descrito en la **Sección 6.1** de que para el modelo es más fácil distinguir las edades más extremas, lo cual tiene sentido puesto que sólo se varía la cantidad de datos que se le entregan en una sola pasada a la red.

Sin embargo este modelo obtiene ligeramente mejores resultados que el anterior (0.8 de *accuracy* en validación y 0.84 en prueba), por lo que, por lo menos en este problema en particular, un `batch_size` más pequeño mejora la capacidad de generalización del modelo.

6.3. Análisis Experimento 3

No utilizar *data augmentation*. En este caso las imágenes son más uniformes, tienen menos variabilidad debido a que no hay cambios inesperados en brillo, saturación, contraste ni tamaño de las caras. Esta menor variabilidad debería implicar que para la red será más fácil aprender el dataset, lo que se evidencia en la **Figura 10**, donde se puede ver que en tan solo 25 épocas se encuentra el mejor modelo con una pérdida cercana a 0.4.

Sorprendentemente, este modelo casi no tiene el problema de los modelos 1 y 2 de confundirse con las edades intermedias, como se puede apreciar en las **Figuras 12 y 13** donde en validación la red acierta a casi el 70 % de las imágenes y en prueba acierta al 90 %! Se puede afirmar entonces que este modelo definitivamente obtiene mejores resultados que los anteriores (*accuracy* de 0.82 en validación y de 0.85 en prueba).

Este buen rendimiento puede atribuirse a la poca cantidad de datos y a su poca variabilidad. A primera vista el modelo tiene una generalización fantástica, pero lo más probable es que falle introduciendo variaciones de iluminación y tamaño.

6.4. Análisis Experimento 4

Descongelar las capas del backbone. No hace falta ver los resultados presentados en la **Sección 5.4** para darse cuenta de que esta es una mala idea, una red neuronal con una gran cantidad de parámetros no puede ser completamente entrenada con unas míseras 300 imágenes. Como se puede apreciar en la **Figura 14** el modelo tiende rápidamente a *overfittearse*, las curvas de pérdida de entrenamiento y validación se separan rápidamente.

Si no se hubiese implementado el mecanismo de *early stopping* el modelo habría sufrido un *over-fitting* incluso peor, como se puede apreciar en la **Figura 15** la matriz de confusión de entrenamiento no es perfecta, esto se debe a que el entrenamiento se detiene en apenas 15 épocas. De todas formas, el modelo no tiene capacidad de generalización, como se puede apreciar en las **Figuras 16 y 17** que el modelo tiende a siempre predecir la etiqueta (5 - 27), por lo cual acierta en el 90 % de los casos de esa categoría tanto en validación como en prueba, pero tiene grandes fallas en las otras dos etiquetas.

6.5. Análisis Experimento 5

Adam conn una distinta tasa de aprendizaje. En este experimento se opto por aumentar la tasa de aprendizaje de 0.0001 a 0.001, por lo que se obtiene una curva de pérdida de validación más inestable como se puede ver en la **Figura 18**. Asimismo, el modelo sufre también de *over-fitting* puesto que no tiene la chance de realmente aprender del conjunto de entrenamiento cuando ya se está disparando el error de validación, el entrenamiento se detiene en tan sólo 15 épocas.

Dado que el modelo también tiene *over-fitting* se puede apreciar el mismo comportamiento descrito en la **Sección 6.4**, en las **Figuras 15, 20 y 21** se ve que el modelo casi siempre predice la etiqueta (5-27) porque no logra aprender las diferencias entre las clases.

6.6. Análisis Experimento 6

Utilizar un optimizador SGD. Se puede apreciar en la **Figura 22** que las curvas de pérdida tienen una forma casi lineal a diferencia del resto de los experimentos que tienen una caída más exponencial, esto impidió que el mecanismo de *early stopping* funcionara y el entrenamiento se detuvo cuando alcanzó el máximo de 400 épocas. También se ve que el entrenamiento fue muy inestable, por lo que las curvas son muy ruidosas, y además de todo esto las curvas van casi a la par.

Este extraño comportamiento probablemente se debe a que el optimizador SDG necesitaba una tasa de entrenamiento mayor y más iteraciones para converger. Así es difícil evaluar si el modelo sufre de *over-fitting* o no, pero se puede apreciar que sigue el mismo patrón (**Figuras 24 y 25**) de diferenciar bien las clases extremas, pero a este le va especialmente mal diferenciando las edades intermedias.

6.7. Análisis Experimento 7

Utilizar un optimizador SGD con una tasa de aprendizaje distinta, en este experimento se decidió subir la tasa de aprendizaje de 0.00001 a 0.0001. Como se mencionó en la **Sección 6.6** una tasa de aprendizaje mayor efectivamente ayudó al optimizador SGD a obtener mejores resultados, como se puede apreciar en la **Figura 26**, las curvas efectivamente se separan llegando a las 400 épocas, cerca del error de 0.6.

Dado que las curvas siguen siendo ruidosas y no se alcanza a apreciar un crecimiento en el promedio de la curva de validación, probablemente se obtengan incluso mejores resultados aumentando un poco más la tasa de aprendizaje y/o aumentando la cantidad de posibles épocas de entrenamiento.

Dada la poca separación entre las curvas, el modelo generaliza bastante bien. Como se puede ver en las **Figuras 28 y 29** se repite el patrón de que al modelo le cuesta menos distinguir las clases extremas (haciendo ese trabajo bastante bien) que las intermedias.

6.8. Balance Tareas 5, 3 y 4

Antes de comparar el desmpeño de las CNN contra los modelos basados en HOG y LBP se debe establecer cuál fue el mejor modelo CNN obtenido en esta tarea, pues se está comparando contra los mejores modelos HOG y LBP obtenidos en sus respectivas tareas.

Como se mencionó en la **Sección 6.3** el modelo que no utiliza *data augmentation* obtiene los mejores resultados, llegando a *accuracy* de 0.8 en validación y 0.85 en prueba. Sin embargo, como también se discutió en aquella sección, este modelo probablemente tendría un peor desempeño con otros datos debido a la poca variabilidad en su conjunto de entrenamiento. Por ende, se descarta como contendor del título de mejor modelo CNN, al menos hasta que se demuestre que tiene un buen desempeño en imágenes con cambios en la iluminación y tamaño, pero eso se escapa del alcance de esta tarea. Entonces, meramente por los números el mejor modelo sería el modelo 2. También hay que considerar que si bien el modelo del experimento 7 obtiene casi tan buenos resultados como el modelo 2, este se demora 350 épocas más en dejar de entrenar y tiene curvas más erráticas.

Recapitulando, los mejores modelos de cada tarea son:

1. Tarea 3: HOG con SVM → *accuracy* de 0.84
2. Tarea 4: LBP con NN → *accuracy* de 0.6
3. Tarea 5: CNN → *accuracy* de 0.84

Se puede ver que la CNN obtiene resultados muy similares al modelo HOG. Es mucho más simple implementar HOG desde cero que una CNN desde cero, y aún así obtienen resultados iguales. También es más fácil entrenar una SVM con pocos datos que los millones de parámetros que hay que entrenar de una CNN, aunque esto se compensa utilizando un *backbone* pre-entrenado como se hace en la presente tarea. Pese a todas estas ventajas de HOG, se considera que es preferible optar por una CNN cuando se pueda, puesto que la etapa de extracción de características se realiza de manera automática y por ende permite enfocar más atención a las otras aristas del problema.

7. Conclusión

Se logra implementar en Python todas las clases y sus métodos necesarios tanto para la lectura y aumento de datos, como para manejar todo lo relacionado a la red neuronal convolucional. Se concluye que las CNN son muy poderosas, tienen el potencial de superar a todos los otros métodos vistos en el curso, pero requieren de una calibración muy cuidadosa para obtener buenos resultados. Probablemente estos modelos tendrían mejor desempeño si se tuviese un conjunto de datos de mayor tamaño, pero en problemas reales este no siempre es el caso.

Por otra parte, si bien la forma en la que se implementa el *data augmentation* en esta tarea añade variabilidad en el conjunto de datos, se estima que se podrían obtener incluso mejores resultados si el *data augmentation*, en vez de simplemente reemplazar los datos originales y terminar con la misma cantidad, se utilizase para aumentar el volúmen de datos, generando tres o cuatro variaciones de cada imagen original.

Se evidenció que es imposible entrenar de manera correcta una CNN con millones de parámetros con tan solo 300 imágenes en el conjunto de entrenamiento, y por ende se evidencia el poder de la técnica de congelar las primeras capas de la red y realizar *fine tuning* en las últimas capas para así ajustar la red a un problema en particular. Además, el mecanismo de *early_stopping* implementado también brinda una gran utilidad para ahorrar tiempo y evitar el *over-fitting* de los modelos

También se pudo observar que los optimizadores ADAM y SGD tienen el potencial de entregar resultados muy similares, pero que hay que realizar un trabajo adicional escogiendo los hiperparámetros adecuados para cada uno.

Por último, queda de manifiesto lo increíblemente útiles que son las librerías como `pytorch`, pues en las tareas 3 y 4 se implementaron las funciones de extracción de características en bajo nivel, pero si se intentase implementar una CNN desde cero esta tarea sería poco menos que imposible.

8. Anexo

8.1. Data

```
class Data(Dataset):
```

```
    def __init__(self, ind0, ind1):
    # class variables
    self.data = []
    self.labels = []
    self.size = 0

    # read images
    ages = [1,5,28]
    for group in range(len(ages)):
      for index in range(ind0,ind1+1):
        img = cv2.imread(f"{PATH}/{ages[group]}/{index}.jpg",1)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = torch.from_numpy(img)
        self.data.append(img)
        self.labels.append(group)
        self.size += 1
```

Listing 1: data_init

```
    def __len__(self):
    return self.size
```

Listing 2: __len__

```
    def __getitem__(self, idx):
    return self.data[idx], self.labels[idx]
```

Listing 3: __getitem__

8.2. Transformer

```
class Transformer():
```

```
    def __init__(self, training_data):
        self.tensor = ToTensor()
        self.train_mean, self.train_std = self.computeMeanStd(training_data)
        self.norm = Normalize(self.train_mean, self.train_std)
        self.trainTF = Compose([
                ColorJitter(brightness = 0.4, contrast = 0.4, saturation = 0.4),
                RandomCrop(140),
                RandomHorizontalFlip(),
                self.norm
                ])
        self.vtTF = Compose([CenterCrop(140),
                self.norm
                ])
        return
```

Listing 4: transformer_init

```
    def computeMeanStd(self, training_data):
        mean = np.zeros(3)
        std = np.zeros(3)
        all = [[],[],[]]
        for image in training_data.data:
          image = image.numpy()
          channels = cv2.split(image)
          for ch in range(len(channels)):
            all[ch].append(channels[ch])
        for ch in range(len(channels)):
          mean[ch] = np.mean(all[ch])
          std[ch] = np.std(all[ch])
        return mean, std
```

Listing 5: computeMeanStd

```
    def augment(self, data, train = True):
    if train:
      for i in range(len(data.data)):
        image = self.trainTF(torch.permute(data.data[i], (2,0,1)).float()/255)
        data.data[i] = self.resize(image)
      return
    for i in range(len(data.data)):
      image = self.vtTF(torch.permute(data.data[i], (2,0,1)).float()/255)
      data.data[i] = self.resize(image)
    return
```

Listing 6: augment

```python
def resize(self, image):
  image = torch.permute(image, (1,2,0))
  channels = [R, G, B] = cv2.split(image.numpy())
  R = cv2.resize(R,(160,160))
  G = cv2.resize(G,(160,160))
  B = cv2.resize(B,(160,160))
  return self.tensor(cv2.merge((R,G,B)))
```

Listing 7: resize

8.3. Modelo

```
def makeNN(pretrained, classify, num_classes):
  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
  return InceptionResnetV1(pretrained, classify, num_classes).to(device), device
```

Listing 8: 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 9: 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().to(device)
      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
          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)
          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
            d_val = d_val.to(device) # validation data
            l_val = l_val.to(device) # validation labels

            val_scores = model(d_val)
            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 10: 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 11: 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 = 'best model epoch')
  plt.legend()
  plt.savefig(f"TrainVal_CEL_model_{index}.png")
  plt.show()
```

Listing 12: drawLossCurves

```python
def makePredictions(model, data_loader, device):
  predictions = []
  model.to(device)
  model.eval()
  real = []
  for index, (batch, label) in enumerate(data_loader):
    batch = batch.to(device)
    label = label.to(device)
    pred = model(batch)
    for image in range(len(pred)):
      predictions.append((pred[image] == torch.max(pred[image])).
                                          nonzero(as_tuple = True)[0].item())
      real.append(label[image].item())
  return predictions, real
```

Listing 13: makePredictions

```python
def drawConfusionMatrix(y_true, y_pred, title):
  Set = title.replace("[NN] ", '')
  acc = accuracy_score(y_true, y_pred, normalize = True)
  print(f'Normalized accuracy score on {Set} set = {acc}')
  cm = confusion_matrix(y_true, y_pred, normalize = 'true')
  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')
  plt.show()
  return acc
```

Listing 14: drawConfusionMatrix

```python
def stats(model, index, train_loader, val_loader, test_loader, device):
  train_pred, train_real = makePredictions(model, train_loader, device)
  val_pred, val_real = makePredictions(model, val_loader, device)
  test_pred, test_real = makePredictions(model, test_loader, device)

  train_acc = drawConfusionMatrix(train_real, train_pred, "[NN] Train")
  val_acc = drawConfusionMatrix(val_real, val_pred, "[NN] Val")
  test_acc = drawConfusionMatrix(test_real, test_pred, "[NN] Test")
  return val_acc
```

Listing 15: 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.