2020

Reconocimiento de imágenes de histología colorrectal

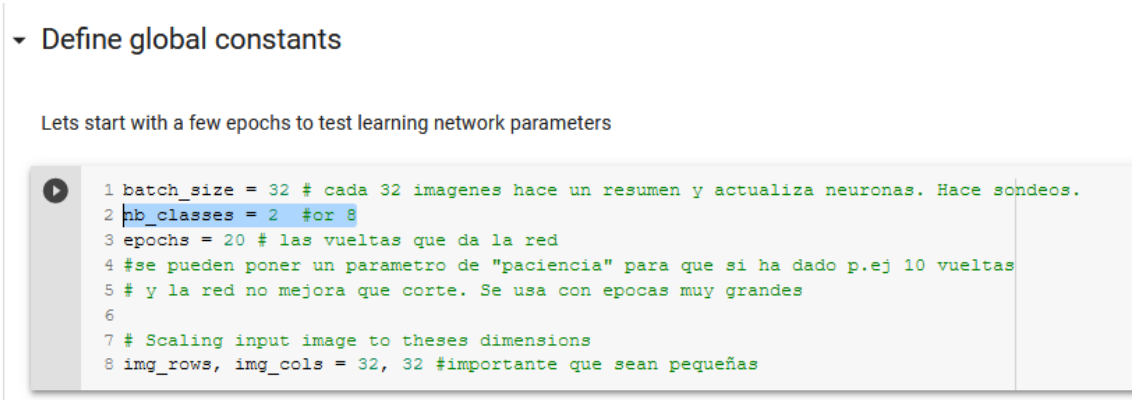
DESAFIOS DE PROGRAMACION

ANTONIO BRI PÉREZ

Contenido

1. Notas y consideraciones

La memoria ha ido redactándose conforme se ha ido realizando la práctica. Esto quiere decir que el código mostrado en los primeros apartados puede ser que ya no se encuentre en el fichero del código fuente **colorrectal_cnn.py** .No obstante, para probar la clasificación binaria es tan sencillo como actualizar la variable nb_classes = 2 en el apartado Define global constants:



Algunas de las capturas que se muestran, por ejemplo, puede ser que se correspondan con clasificación multiclase o clasificación binaria. O que incluso, algunos de los resultados se hayan realizado con modelos de redes neuronales peores que los que están en el fichero fuente. Esto se ha hecho para que en caso de querer ejecutar el fichero fuente no se demore demasiado la ejecución total.

Si se quiere volver a la clasificación multiclase será tan sencillo como actualizar nb_classes al valor 8.

2. Nivel Básico

2.1. Clasificación binaria

Si queremos realizar una clasificación binaria de nuestro data set (tiene tumor = 0, o no tiene tumor = 1) tendremos que dejar el collab tal cual se nos da. Es decir, al principio del todo tendremos que dejar la variable nb_classes a 2:

```
batch_size = 32
nb_classes = 2  #or 8
epochs = 25

# Scaling input image to theses dimensions
img_rows, img_cols = 32, 32
```

Ilustración 1

Esta variable marcará la cantidad de total de clasificaciones diferentes que nuestra red tendrá que distinguir.

A continuación, después de cargar los datos de la bbdd tendremos que añadir una condición de que si esa variable que acabamos de inicializar a 2 es igual a 2 entonces para todas las posibles clases mayores que 0 (no es tumor) tendrá que tomar el valor de 1:

Only for binary classification. All number of classes greater than 0 will be set to 1.

```
if nb_classes==2:
  y[y>0] = 1
```

Ilustración 2

De esa forma obtendremos 2 veredictos: tiene tumor o no tiene tumor. Para comprobar que hasta ahora no nos hemos equivocado para realizar la clasificación binaria podemos realizar un conteo del número de ejemplos que tenemos por clase:

```
[12]  collections.Counter(y)

Counter({0: 625, 1: 4375})
```

Ilustración 3

En este caso, al tratarse de una clasificación binaria, los 4375 ejemplos de la clase 1 corresponden a la suma de todos los ejemplos que no son clase 0. Veremos cómo estos datos cambiarán cuando hagamos la clasificación completa.

2.2. Métrica AUC

En este caso tenemos un desequilibrio alto ya que estamos ante una clasificación binaria del problema. Por ello estudiáremos la métrica AUC solo. La métrica AUC significa área bajo la curva roc y es esta curva la que nos indica el rendimiento de un modelo de clasificación en todos los umbrales de clasificación.

La métrica AUC entonces mide toda el área bidimensional por debajo de la curva ROC completa. Vendría a ser como un cálculo integral. Una forma de visualizar AUC es como la probabilidad de que el modelo clasifique un ejemplo positivo aleatorio más alto que un ejemplo negativo aleatorio.

```
y_pred = model.predict(X_test)


# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()


for i in range(classes):
    fpr[i], tpr[i], _ = metrics.roc_curve(y_test_nn[:,i], y_pred[:, i])
    roc_auc[i] = metrics.auc(fpr[i], tpr[i])
    print('AUC class {} {:.4f} '.format(i,roc_auc[i]))

print('AUC mean {:.4f} '.format(np.array(list(roc_auc.values())).mean()))
```

```
AUC class 0 0.8771
AUC class 1 0.8771
AUC mean 0.8771
```

Ilustración 4

2.3. Validación cruzada

Para la validación cruzada hemos hecho uso de la función **StratifiedKFold**

```
X, y, input_shape = load_data() #cargamos la bbdd


cvscores1 = []
skf = StratifiedKFold(n_splits=10,shuffle=True, random_state=42)
for train_index, test_index in skf.split(X, y):
  X_train, X_test = X[train_index], X[test_index]
  y_train, y_test = y[train_index], y[test_index]
  y_train_nn = keras.utils.to_categorical(y_train, nb_classes)
  y_test_nn = keras.utils.to_categorical(y_test, nb_classes)

  model = cnn_model1(X_train.shape[1:])
  history = model.fit(X_train, y_train_nn, batch_size=batch_size, epochs=epochs, validation_split=0.1, verbose=2)
  scores = model.evaluate(X_test, y_test_nn, verbose=1)

  #Resultados del primer clasificador

  print("%s: %.2f%%" % (model.metrics_names[1], scores[1] * 100))
  cvscores1.append(scores[1] * 100)

print('Mostramos las curvas de aprendizaje')
plot_learning_curves(history)

print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores1), np.std(cvscores1)))

ResultadosPrimerClasificador = cvscores1


```

Ilustración 5

Con esta función de sklearn podemos obtener los índices de prueba y entrenamiento necesarios para realizar la división en test y entrenamiento. Es una variación del KFold. En el caso de shuffle tenemos que dejarlo a True para que mezcle las muestras de cada clase antes de dividirlo en lotes y en el caso de random state lo fijamos a un valor constante para así obtener siempre la misma aleatoriedad para cada pliegue de cada clase.

En concreto, obtenemos los siguientes resultados (ambos con clasificación binaria) para validación normal vs validación cruzada:

Validación cruzada:

```
[12]  loss, acc = model.evaluate(X_test, y_test_nn, batch_size=batch_size)
      print(f'loss: {loss:.2f} acc: {acc:.2f}')

    16/16 [==============================] - 0s 3ms/step - loss: 0.5478 - accuracy: 0.8180
    loss: 0.55 acc: 0.82
```

Ilustración 6

Validación normal:

```
loss, acc = model.evaluate(X_test, y_test_nn, batch_size=batch_size)
print(f'loss: {loss:.2f} acc: {acc:.2f}')

40/40 [==============================] - 0s 4ms/step - loss: 0.3747 - accuracy: 0.8760
loss: 0.37 acc: 0.88
```

Ilustración 7

2.4. Estudio comparativo (Wilcoxon)

El estudio comparativo de Wilcoxon nos ofrece un valor que nos dice que posibilidades tiene un modelo de ser mejor que otro. En este caso, hemos hecho dos modelos muy parecidos para demostrar así que nos sale un valor cercano al 0.5. En contraposición, en el desafío 2 hemos hecho un modelo peor a cosa hecha para ver como ese valor de Wilcoxon se acerca más a 0 o a 1 si hemos elegido *greater* o *less* en los parámetros.

Para poder realizar el estudio comparativo Wilcoxon necesitamos crear otro modelo de red neuronal para poder ser comparado con el primero.

▾ Modelo 2

```python
def cnn_model2(input_shape):
  model2_input = keras.Input(shape=(input_shape), name='img')
  x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(model2_input)
  x = layers.MaxPooling2D(2, 2)(x)

  x = layers.Conv2D(32, (3, 3), padding='same', activation='relu')(x)
  x = layers.MaxPooling2D(2,2)(x)
  x = layers.Dropout(0.2)(x)

  x = layers.Conv2D(32, (3, 3), padding='same', activation='relu')(x)
  x = layers.MaxPooling2D(2,2)(x)
  x = layers.Dropout(0.2)(x)

  x = layers.Flatten()(x)
  x = layers.Dropout(0.2)(x)
  outputs = layers.Dense(nb_classes, activation='softmax')(x)

  model2 = keras.Model(model2_input, outputs, name='model2')


  model2.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'] )


  return model2
```

Ilustración 8

Hemos elegido un modelo de red convolucional ya que funcionan muy bien para la clasificación de imágenes.

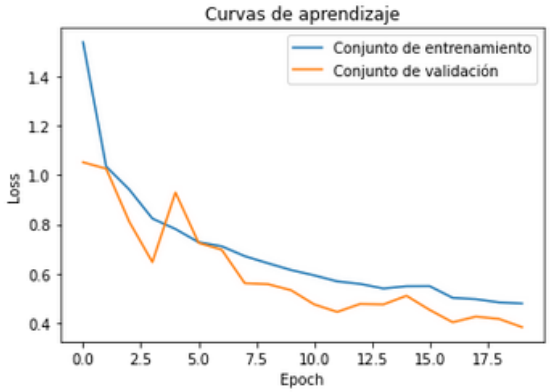
Se trata de un modelo con una capa convolucional con 32 filtros de 3x3 seguida de un MaxPooling de 2x2. A continuación, añadimos otra capa convolucional de 32 filtros también de tamaño 3x3 seguida de una maxpooling de 2x2. Seguidamente, añadimos, nuevamente otra capa convolucional igual que las anteriores seguida de otro maxPooling con un dropout. Finalmente, añadimos una capa fully connected y una capa densa con el mismo número de clases.

Entrenamos el modelo 2:

```
#Resultados del segundo clasificador
cvscores2 = []
skf = StratifiedKFold(n_splits=10,shuffle=True, random_state=42)
for train_index, test_index in skf.split(X, y):
  X_train, X_test = X[train_index], X[test_index]
  y_train, y_test = y[train_index], y[test_index]
  y_train_nn = keras.utils.to_categorical(y_train, nb_classes)
  y_test_nn = keras.utils.to_categorical(y_test, nb_classes)

  model2 = cnn_model2(X_train.shape[1:])
  history = model2.fit(X_train, y_train_nn, batch_size=batch_size, epochs=epochs, validation_split=0.1, verbose=2)
  scores2 = model2.evaluate(X_test, y_test_nn, verbose=1)



  print("%s: %.2f%%" % (model2.metrics_names[1], scores2[1] * 100))
  cvscores2.append(scores2[1] * 100)

print('Mostramos las curvas de aprendizaje')
plot_learning_curves(history)

print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores2), np.std(cvscores2)))
#Resultados del segundo clasificador

ResultadosSegundoClasificador = cvscores2
```

Y obtenemos un accuracy del 81.18%

```
accuracy: 84.00%
Mostramos las curvas de aprendizaje
```




```
81.18% (+/-3.33%)
```

WILCOXON RANKED TEST

```
[59] 1 from scipy.stats import wilcoxon
     2 import warnings
     3 warnings.filterwarnings('ignore')
     4
     5 wilcox_V, p_value =  wilcoxon(ResultadosPrimerClasificador, ResultadosSegundoClasificador, alternative='greater', zero_method='wilcox', correction=False)
     6
     7 print('Resultado completo del test de Wilcoxon')
     8 print(f'Wilcox V: {wilcox_V}, p-value: {p_value:.2f}')
     9

    Resultado completo del test de Wilcoxon
    Wilcox V: 29.0, p-value: 0.44
```

Ilustración 9

El valor de p-value es 0.44 por lo tanto podemos afirmar que podemos rechazar la hipótesis nula con una confianza del 0.44% de que los resultados del primer clasificador son peores que los del segundo clasificador.

Este resultado no es sorprendente ya que podemos ver como los resultados del primer clasificador son ligeramente superiores a los del modelo 2. Es por eso que Wilcoxon nos dice que tiene un 44% de posibilidades de ser peor que el modelo 2 o mejor dicho, tiene un 56% de posibilidades de ser mejor. De nuevo, ambos modelos son muy parecidos en resultados, eso explica este resultado.

2.5. Aumentado de datos

A continuación, vamos a realizar un aumentado de datos para compensar el desequilibrio existente entre ambas clases. Si ejecutamos el collab conforme se nos da podremos observar que la métrica AUC arroja un valor de 0.5

- Rotaciones en el rango 0 – 20
- Traslaciones horizontales y verticales de 0.2
- Flips horizontales

```
1  """
2  Aumentado de datos
3  """
4  from keras.preprocessing.image import ImageDataGenerator
5
6
7
8  datagen2 = ImageDataGenerator(rotation_range=20,
9                                width_shift_range=0.2,
10                               height_shift_range=0.2,
11                               horizontal_flip=True)
12
13 datagen2.fit(X_train)
```

Ilustración 10

Esto lo realizamos sobre el conjunto X_train solo y entrenaremos nuestro modelo con el conjunto X_train. No debemos aplicarlo sobre el X_test también porque de esta forma nos aseguramos de que nuestra red no ha visto imágenes preprocesadas antes de la validación.

Una vez hemos utilizado esta función, tendremos que entrenar de nuevo nuestro modelo para las nuevas transformaciones.

```
16
17 history = model.fit_generator(datagen2.flow(X_train, y_train_nn, batch_size=32),
18                                steps_per_epoch=len(X_train) / 32,
19                                epochs=25,
20                                validation_data=datagen2.flow(X_test, y_test_nn))
21
22 print('Mostramos las curvas de aprendizaje')
23 plot_learning_curves(history)
24
25
26 # Evaluamos usando el test set
27 score = model.evaluate(X_test, y_test_nn, verbose=0)
28
29 print('Resultado en el test set:')
30 print('Test loss: {:0.4f}'.format(score[0]))
31 print('Test accuracy: {:0.2f}%'.format(score[1] * 100))
```

Ilustración 11

Por tanto, compilamos nuestro modelo con el optimizador adam y lo entrenamos de forma que cargamos en el datagen el X_train y el y_train_nn. De esa forma nos aseguramos de que estamos entrenando a nuestro modelo solo con las imágenes preprocesadas.

Testing AUC result for two and multiple classes

```
1 y_scores = model.predict(X_test) # Confidence prediction per class
2 y_pred = y_scores.argmax(axis=1) # Select classes with most confidence prediction
3
4 if nb_classes ==  2:
5   print(f'AUC {metrics.roc_auc_score(y_test, np.round(y_scores[:,1],2)):.4f} ')
6 else:
7   print(f'AUC {metrics.roc_auc_score(y_test, y_scores, multi_class = "ovr"):.4f} ')
```

```
AUC 0.8532
```

Finalmente, podemos ver como la métrica AUC pasa a arrojar un valor de 0.8532. Una forma de interpretar la métrica AUC es que un modelo que arroja un valor AUC de 0% quiere decir que el 100% de sus predicciones son erróneas. En este caso podemos decir que alrededor de un 15% de las predicciones son erróneas.

3. Nivel Medio

3.1. Preprocesado de imágenes

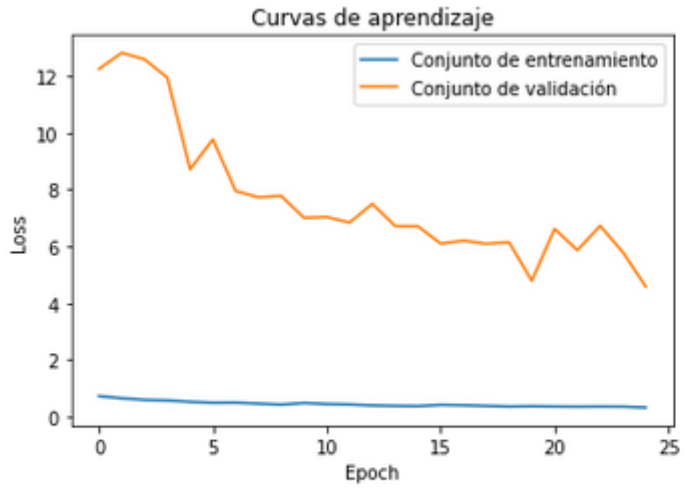
Ahora, vamos a probar a añadir algunas modificaciones al primer modelo que hemos creado. En concreto, le hemos añadido unas funciones para modificar el contraste de la imagen y otra para controlar la ecualización del histograma

PREPROCESADO DE IMAGENES

```
from keras.preprocessing.image import ImageDataGenerator
from skimage import data, img_as_float
from skimage import exposure


for i in range(0, X_train.shape[0]):
  X_train[i] = exposure.equalize_adapthist(X_train[i], clip_limit=0.03)

for i in range(0, X_train.shape[0]):
  X_train[i] = exposure.equalize_hist(X_train[i], mask=None)




datagen2 = ImageDataGenerator(brightness_range=(1.0, 1.0))
datagen2.fit(X_train)


history = model.fit(datagen2.flow(X_train, y_train_nn, batch_size=32),
                              steps_per_epoch=len(X_train) / 32,
                              epochs=25,
                              validation_data=(X_test, y_test_nn))

print('Mostramos las curvas de aprendizaje')
plot_learning_curves(history)


# Evaluamos usando el test set
score = model.evaluate(X_test, y_test_nn, verbose=0)

print('Resultado en el test set:')
print('Test loss: {:0.4f}'.format(score[0]))
print('Test accuracy: {:0.2f}%'.format(score[1] * 100))
```

Ilustración 12

Este parámetro recibe dos argumentos siendo cada uno de ellos los límites entre los cuales la imagen va a tomar un cierto brillo. Un valor de 1 significa que la imagen no cambia, menos de 1 que se oscurece y más de 1 que aumenta de brillo. Si probamos con un rango de 1.0 – 1.5 obtenemos lo siguiente:




```
Resultado en el test set:
Test loss: 4.5796
Test accuracy: 23.80%
```

Ilustración 13

Básicamente la red no aprende nada si aumentamos el brillo de las imágenes. Es más, cada vez ofrece peores resultados. No se muy bien porque ocurre esto.

3.2. Clasificación con todas las categorías

En este punto tengo que matizar que al realizar la clasificación con todas las categorías el trabajo realizado hasta ahora se ve afectado en la medida que por ejemplo, ahora el apartado de las métricas AUC realizada en la sección básica ahora se ve influenciada por la presencia de 8 clases así que el collab se ha adaptado a ello.

Ahora lo que queremos es que nuestra red reciba una imagen y nos diga, de entre todas las categorías con las que ha sido entrenada a que categoría pertenece esa imagen, un clasificador multiclase.

Lo primero de todo es indicarle a nuestro modelo que ahora podrá recibir hasta 8 clases. Para ello, nos vamos a la zona donde definimos las constantes globales y actualizamos el valor de nb_classes a 8:

```
batch_size = 32 # cada 32 imagenes hace un resumen y actualiza neuronas. Hace sondeos.
nb_classes = 8  #or 8
epochs = 25 # las vueltas que da la red
#se pueden poner un parametro de "paciencia" para que si ha dado p.ej 10 vueltas
# y la red no mejora que corte. Se usa con epocas muy grandes

# Scaling input image to theses dimensions
img_rows, img_cols = 32, 32 #importante que sean pequeñas
```

Ilustración 14

El siguiente paso será eliminar la celda de código que comprobaba si el valor de esa variable era 2 porque ahora no nos interesa que aquellas imágenes que no sean tumor nos las agrupe en la misma clase, sino que las distinga de forma individual. Por tanto, **esta celda debemos eliminarla:**

```
[10]  1 if nb_classes==2:
      2   y[y>0] = 1
```

Ilustración 15

Asimismo, podemos comprobar que estamos realizando los ajustes correctos realizando un conteo de las muestras que tenemos por cada clase de nuestro set de datos:

```
[142]  1 import collections
       2
       3 collections.Counter(y)
```

```
Counter({0: 625, 1: 625, 2: 625, 3: 625, 4: 625, 5: 625, 6: 625, 7: 625})
```

Ilustración 16

Ahora si multiplicamos 625 por 7 debe darnos 4375 que es el número total de muestras que obteníamos en la clasificación binaria cuando no eran tumores (eran de la clase 0). Deducimos así que hemos hecho los ajustes correctos.

Otra forma de comprobar que el modelo trabaja con las 8 clases es mostrar por pantalla el vector de etiquetas que usará nuestro modelo para realizar la validación:

```
#X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
#X, y, input_shape = load_data() #cargamos la bbdd

skf = StratifiedKFold(n_splits=10,shuffle=True, random_state=42)
for train_index, test_index in skf.split(X, y):
  X_train, X_test = X[train_index], X[test_index]
  y_train, y_test = y[train_index], y[test_index]

print(y_test)
# Convert integers to one-hot vector
y_train_nn = keras.utils.to_categorical(y_train, nb_classes)
y_test_nn = keras.utils.to_categorical(y_test, nb_classes)

print(f'x_train {X_train.shape} X_test {X_test.shape}')
print(f'y_train {y_train.shape} y_test {y_test.shape}')
print(f'y_train_nn {y_train_nn.shape} y_test_nn {y_test_nn.shape}') #vectores para las redes nr
#si tuvieramos 8 imagenes para clasificar seria (3750, 8) y (1250, 8)
```

```
[5 6 5 7 3 5 1 1 0 7 5 5 4 7 5 0 7 4 0 4 7 7 4 1 1 3 1 3 4 3 3 0 6 0 5 1 4
 2 6 2 1 7 1 5 1 1 4 5 2 3 2 0 4 1 0 0 4 1 5 5 0 2 2 2 2 2 3 4 7 2 0 1 6 0
 0 4 6 1 2 4 3 6 2 3 7 1 0 5 2 3 2 1 1 5 7 2 3 7 7 6 3 0 2 2 4 0 4 3 0 1 3
 6 7 4 5 4 0 5 5 5 5 0 2 2 0 0 4 1 2 2 3 5 0 2 5 3 7 3 2 6 5 6 6 5 6 6 4 7
 7 0 5 4 1 6 0 0 7 4 6 7 0 3 4 2 3 0 3 1 0 0 3 3 6 4 3 7 4 0 6 1 7 6 1 6 1
 7 6 4 7 7 0 3 6 1 6 5 5 5 5 4 3 3 4 2 5 0 2 5 3 6 4 6 3 5 0 3 4 4 7 7 6 3
 1 2 7 1 0 3 7 6 2 1 3 5 1 5 3 5 0 2 2 6 4 5 2 3 6 1 1 6 4 6 7 5 5 3 3 5 6
 0 5 4 2 0 0 7 7 4 7 6 3 4 4 1 7 6 4 0 1 2 3 2 3 2 1 1 4 6 1 1 1 4 0 7 5 1
 5 7 5 6 6 6 3 5 5 2 4 4 6 6 2 1 4 5 3 3 5 2 2 6 0 3 6 1 0 7 7 1 1 7 0 1 5
 6 3 0 2 0 1 6 0 7 4 5 7 5 7 0 4 6 0 2 6 3 4 3 6 6 1 1 1 5 0 7 5 1 3 3 4 3
 5 5 3 5 6 4 4 4 3 2 5 6 2 5 2 0 7 1 1 5 4 2 2 6 4 5 0 7 7 6 0 2 2 4 3 1 1
 7 7 7 7 1 2 6 0 5 6 6 6 4 4 6 2 0 7 2 3 2 0 4 6 0 1 2 3 3 7 5 6 3 2 5 7 6
 2 7 2 3 4 5 7 2 3 0 0 0 2 1 6 0 6 4 2 1 1 4 7 7 7 7 0 6 7 1 4 1 3 5 4 4 7
 4 0 5 1 2 7 3 3 1 3 7 0 7 6 2 2 3 4 2]
x_train (4500, 32, 32, 3) X_test (500, 32, 32, 3)
y_train (4500,) y_test (500,)
y_train_nn (4500, 8) y_test_nn (500, 8)
```

Ilustración 17

Aquí se puede apreciar que se está trabajando con 8 clases.

Finalmente, cuando ejecutamos todas las celdas de código podemos visualizar los datos que nos arroja nuestro modelo entrenado para realizar hasta 8 clasificaciones:

```
Predictions
Counter({7: 72, 2: 68, 0: 65, 1: 64, 3: 61, 5: 61, 6: 56, 4: 53})
Confusion matrix
[[55  0  1  1  1  4  0  0]
 [ 0 43 11  0  8  0  0  0]
 [ 2  6 51  2  1  1  0  0]
 [ 4  0  2 56  0  1  0  0]
 [ 0 15  3  0 40  1  3  0]
 [ 4  0  0  2  3 54  0  0]
 [ 0  0  0  0  0  0 52 11]
 [ 0  0  0  0  0  0  1 61]]
              precision    recall  f1-score   support

       TUMOR       0.85      0.89      0.87        62
      STROMA       0.67      0.69      0.68        62
     COMPLEX       0.75      0.81      0.78        63
      LYMPHO       0.92      0.89      0.90        63
      DEBRIS       0.75      0.65      0.70        62
      MUCOSA       0.89      0.86      0.87        63
     ADIPOSE       0.93      0.83      0.87        63
       EMPTY       0.85      0.98      0.91        62

    accuracy                           0.82       500
   macro avg       0.83      0.82      0.82       500
weighted avg       0.83      0.82      0.82       500
```

Ilustración 18

Como era previsible, la precisión del modelo ha bajado ya que ahora la variedad de clasificaciones que debe hacer ha aumentado. Las mejoras sobre este modelo las realizaremos más adelante.

```
[145]  1 loss, acc = model.evaluate(X_test, y_test_nn, batch_size=batch_size)
       2 print(f'loss: {loss:.2f} acc: {acc:.2f}')

       16/16 [==============================] - 0s 3ms/step - loss: 0.8995 - accuracy: 0.6140
       loss: 0.90 acc: 0.61
```

Ilustración 19

Estamos hablando de una pérdida (loss = 0.9) frente a una pérdida de 0.38 que nos arrojaba la clasificación binaria y un accuracy de 0.61 que es muy próximo a 0.5. Esto último sería similar a preguntar a alguien que no conoce del tema y acertase de casualidad. Son resultados inaceptables, y más en temas de salud.

3.3. Métricas AUC y accuracy

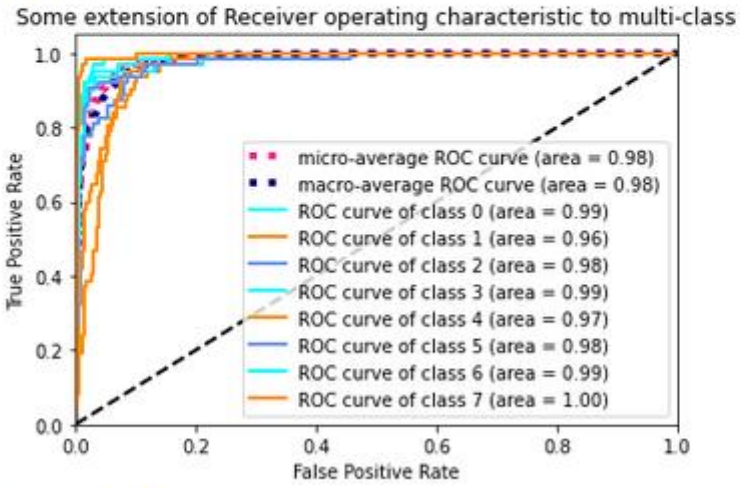
En este caso vamos a estudiar las métricas AUC y accuracy sobre el modelo 1

```
y_pred = model.predict(X_test)


# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()

classes = 0
if y_test_nn.shape[1] == 2:
  classes = nb_classes
else:
  classes = nb_classes

for i in range(classes):
    fpr[i], tpr[i], _ = metrics.roc_curve(y_test_nn[:,i], y_pred[:, i])
    roc_auc[i] = metrics.auc(fpr[i], tpr[i])
    print('AUC class {} {:.4f} '.format(i,roc_auc[i]))

print('AUC mean {:.4f} '.format(np.array(list(roc_auc.values())).mean()))

fpr["micro"], tpr["micro"], _ = roc_curve(y_test_nn.ravel(), y_pred.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

```

Ilustración 20

A continuación, vamos a imprimir las curvas ROC de todas las clases. Hemos omitido el código aquí pero se encuentra dentro del notebook de Python.

```
AUC class 0 0.9874
AUC class 1 0.9600
AUC class 2 0.9758
AUC class 3 0.9890
AUC class 4 0.9694
AUC class 5 0.9794
AUC class 6 0.9923
AUC class 7 0.9953
AUC mean 0.9811
```




```
Acc: 0.8240
```

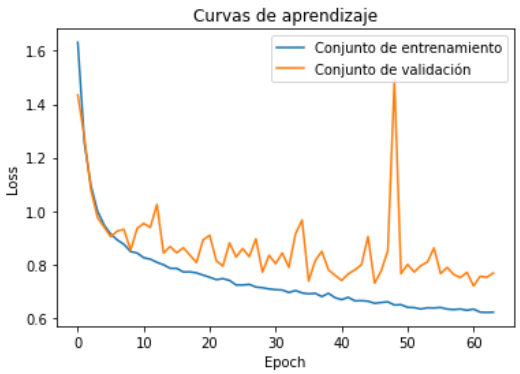
Ilustración 21

Como sabemos, una forma de interpretar el AUC es que el modelo clasifique un ejemplo positivo aleatorio más alto que un ejemplo negativo aleatorio. Además, un modelo cuyas predicciones son un 100% incorrectas (es decir, que nunca acierta) tiene un AUC de 0.0 mientras que un modelo cuyas predicciones son 100% correctas tiene un AUC de 1.0. En nuestro caso vemos como las AUC son bastante elevadas para todas y cada una de las clases lo cual es un buen indicativo de que tenemos un modelo sólido

3.4. Ajuste del algoritmo de optimización (mejores resultados con menos épocas)

En primer lugar vamos a probar el optimizador RMSprop para el segundo modelo que hemos propuesto. Este optimizador es un modelo adaptativo de aprendizaje propuesto por Geoff Hinton. Este modelo también divide la tasa de aprendizaje por un promedio de gradientes cuadrados que disminuye exponencialmente. Este algoritmo nos arroja, para 64 épocas un accuracy de:

```
16/16 [==============================] - 0s 2ms/step - loss: 0.6967 - accuracy: 0.7240
Mostramos las curvas de aprendizaje
```

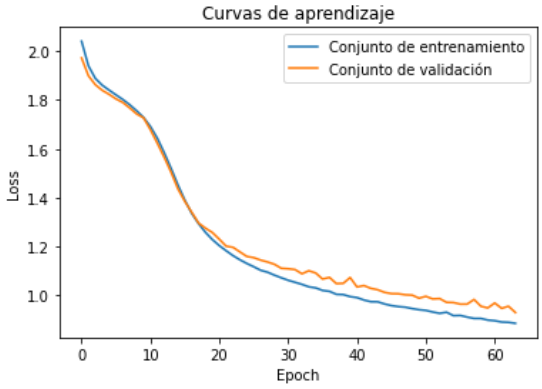



```
accuracy: 72.40%
72.40% (+/-0.00%)
```

Ilustración 22

Vamos a probar ahora con el optimizador adamax. Adamax implementa el algoritmo adamax que es una variante del algoritmo Adam basado en la norma del infinito. Este optimizador nos arroja los siguientes resultados para 64 épocas:

```
16/16 [==============================] - 0s 2ms/step - loss: 0.8818 - accuracy: 0.6480
Mostramos las curvas de aprendizaje
```



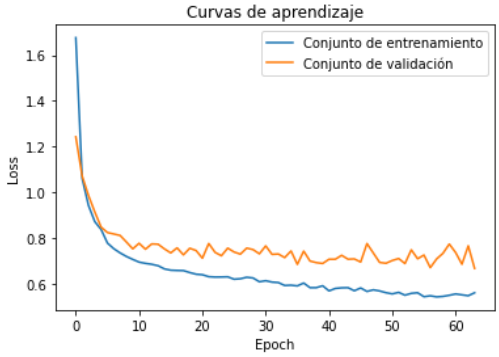

```
accuracy: 64.80%
64.80% (+/-0.00%)
```

Ilustración 23

Como vemos, es peor que el algoritmo RMSprop

Vamos ahora con el algoritmo Adam. Este es un método de descenso por gradiente estocástico basado en la estimación adaptativa de momentos de primer y segundo orden. Obtenemos los siguientes resultados:

```
127/127 - 0s - loss: 0.5616 - accuracy: 0.7886 - val_loss: 0.6681 - val_accuracy: 0.7400
16/16 [==============================] - 0s 2ms/step - loss: 0.6115 - accuracy: 0.7480
Mostramos las curvas de aprendizaje
```

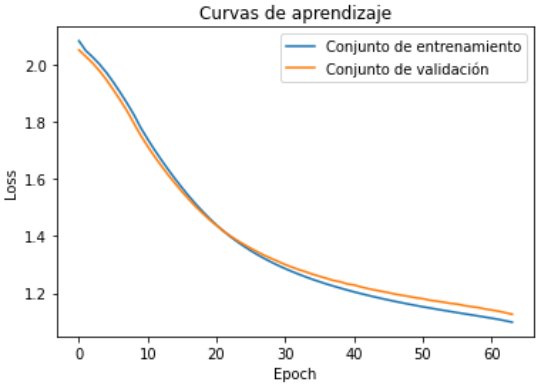



```
accuracy: 74.80%
74.80% (+/-0.00%)
```

Ilustración 24

Hasta el momento, este es el mejor optimizador que hemos encontrado. Vamos a probar finalmente con adagrad. Este optimizador trabaja con tasas de aprendizaje específicas de parámetros que se adaptan en relación con la frecuencia con la que se actualiza un parámetro durante el entrenamiento. Cuantas más actualizaciones reciba un parámetro, menos veces se actualizará el optimizador.

```
127/127 - 0s - loss: 1.0981 - accuracy: 0.5726 - val_loss: 1.1260 - val_accuracy: 0.5489
16/16 [==============================] - 0s 2ms/step - loss: 1.0682 - accuracy: 0.5720
Mostramos las curvas de aprendizaje
```




```
accuracy: 57.20%
57.20% (+/-0.00%)
```

Ilustración 25

En vista a estos resultados podemos ordenar los 4 clasificadores probados en este orden:

POSICION	ALGORITMO
1ª Posición	Adam: 74,80 %
2ª Posición:	RMSprop: 72,4%
3ª Posición	Adamax: 64,8%
4ª Posición	Adagrad: 57,20%

No es casualidad que Adam sea el mejor d ellos 4 optimizadores ya que Adam es un algoritmo muy popular en el campo del Deep Learning. Además, combina los beneficios tanto del AdaGrad como el de RMSprop. Adam además, es computacionalmente eficiente, ocupa poca memoria y es apropiado para objetivos no estacionarios así como apropiado para problemas con mucho ruido o gradientes muy dispersos

3.5. Aumentado de datos

Ahora al tratarse de la clasificación con todas las clases tenemos un equilibrio entre las mismas. Vamos a realizar entonces el aumentado de datos a nuestro conjunto de entrenamiento sobre el modelo 2:

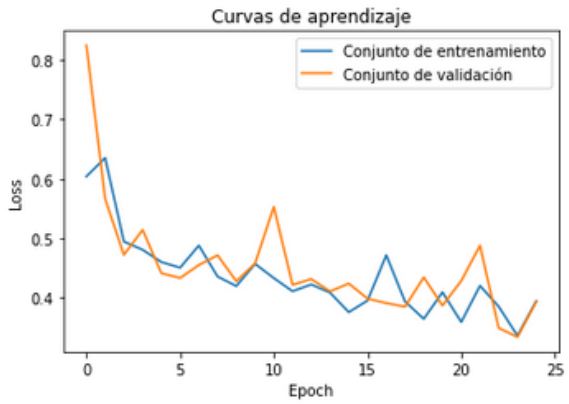
AUMENTADO DE DATOS

```
[20]
"""
Aumentado de datos
"""
from keras.preprocessing.image import ImageDataGenerator



datagen2 = ImageDataGenerator(rotation_range=20,
                              width_shift_range=0.2,
                              height_shift_range=0.2,
                              horizontal_flip=True)



history = model2.fit_generator(datagen2.flow(X_train, y_train_nn, batch_size=32),
                               steps_per_epoch=len(X_train) / 32,
                               epochs=25,
                               validation_data=(X_test, y_test_nn))

print('Mostramos las curvas de aprendizaje')
plot_learning_curves(history)


# Evaluamos usando el test set
score = model2.evaluate(X_test, y_test_nn, verbose=0)

print('Resultado en el test set:')
print('Test loss: {:0.4f}'.format(score[0]))
print('Test accuracy: {:0.2f}%'.format(score[1] * 100))
```

Ilustración 26

En concreto, cambiamos el rango de rotación al rango 0-20, realizamos traslaciones horizontales y verticales un 20% del tamaño y Flips horizontales. Finalmente, entrenamos nuestro modelo 2:

Mostramos las curvas de aprendizaje




```
Resultado en el test set:
Test loss: 0.5982
Test accuracy: 80.80%
```

Ilustración 27

Como vemos, gracias al aumentado de datos hemos pasado un accuracy de 78,80% en el modelo 2 a un accuracy del 85,80% lo cual no está nada mal.

3.6. Notación funcional de keras

Con la API funcional de keras podemos crear modelos más dinámicos que con Sequential. Gracias a esta API podemos manejar modelos con topología no lineal, modelos con capas compartidas y modelos con múltiples entradas o salidas. Se basa en la idea de que un modelo de aprendizaje profundo suele ser un gráfico acíclico dirigido de capas. La API funcional es un conjunto de herramientas para construir gráficos de capas.

Al mismo tiempo que hemos aprovechado para adaptar las redes neuronales a la notación funcional de keras hemos aprovechado para mejorar el modelo que venía con el notebook así como mejorar el modelo 2 sobre el que hemos trabajado en la parte básica del desafío.

Primer modelo:

```
[7]  1 #X_train.shape[1:]
     2
     3 def cnn_model1(input_shape):
     4   model = keras.Input(shape=(input_shape), name='img')
     5   x = layers.Conv2D(32, (5, 5), activation='relu', padding='same')(model)
     6   x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
     7   x = layers.MaxPooling2D(pool_size=(2,2))(x)
     8   x = layers.Dropout(0.2)(x)
     9
    10   x = layers.Conv2D(64, (5, 5), activation='relu', padding='same')(x)
    11   x = layers.Conv2D(64, (3, 3), activation='relu')(x)
    12   x = layers.MaxPooling2D(pool_size=(2,2))(x)
    13   x = layers.Dropout(0.2)(x)
    14   x = layers.Flatten()(x)
    15   x = layers.Dense(512, activation='relu')(x)
    16   x = layers.Dropout(0.2)(x)
    17   outputs = layers.Dense(nb_classes, activation='softmax')(x)
    18
    19   model = keras.Model(model, outputs, name='model')
    20
    21
    22   model.compile(loss='categorical_crossentropy',optimizer='adam', metrics=['accuracy'])
    23
    24   return model
```

Ilustración 28

La imagen de arriba muestra el primer modelo ya mejorado con respecto al modelo original del notebook. Recordemos que el modelo original era este:

```
def cnn_model():
    #
    # Neural Network Structure
    #

    model = Sequential()

    model.add(layers.Conv2D(6, (5, 5)))
    model.add(layers.Activation("sigmoid"))
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))

    model.add(layers.Conv2D(16, (5, 5)))
    model.add(layers.Activation("sigmoid"))
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))

    model.add(layers.Flatten())

    model.add(layers.Dense(120))
    model.add(layers.Activation("sigmoid"))

    model.add(layers.Dense(84))
    model.add(layers.Activation("sigmoid"))

    model.add(layers.Dense(nb_classes))
    model.add(layers.Activation('softmax'))

    return model
```

Ilustración 29

La diferencia principal entre el modelo mejorado y el modelo original es que el modelo mejorado presenta más capas de convolución al principio de la red así como alguna capa de dropout para evitar el sobre entrenamiento. Además, también incorpora una capa densa con más neurona que el modelo original. Todo ello hace que obtengamos mejores resultados.

Segundo modelo:

```
model2_input = keras.Input(shape=(X_train.shape[1:]), name='img')

x = layers.Conv2D(16, (28, 28), padding='same', activation='relu', input_shape=input_shape)(model2_input)
x = layers.MaxPooling2D(2, 2)(x)

x = layers.Conv2D(32, (28, 28), padding='same', activation='relu')(x)
x = layers.MaxPooling2D(2,2)(x)
x = layers.Conv2D(64, (5, 5), padding='same', activation='relu')(x)

x = layers.Flatten()(x)
x = layers.Dense(120, activation='relu')(x)
x = layers.Dense(64, activation='relu')(x)
x = layers.Dense(12, activation='relu')(x)
outputs = layers.Dense(nb_classes, activation='softmax')(x)

model2 = keras.Model(model2_input, outputs, name='model2')


model2.compile(loss='categorical_crossentropy', optimizer='adagrad', metrics=['accuracy'] )

# Iniciamos el entrenamiento durante
history = model2.fit(X_train, y_train_nn, batch_size=batch_size, epochs=64, validation_split=0.1, verbose=2)
scores2 = model2.evaluate(X_test, y_test_nn, verbose=1)

print('Mostramos las curvas de aprendizaje')
plot_learning_curves(history)

cvscores2 = []
print("%s: %.2f%%" % (model2.metrics_names[1], scores2[1] * 100))
cvscores2.append(scores2[1] * 100)
print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores2), np.std(cvscores2)))
```

Ilustración 30

En cuanto al segundo modelo con notación funcional quedaría de la siguiente manera:

Modelo 2

+ Código

```
def cnn_model2(input_shape):
  model2_input = keras.Input(shape=(input_shape), name='img')
  x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(model2_input)
  x = layers.MaxPooling2D(2, 2)(x)

  x = layers.Conv2D(32, (3, 3), padding='same', activation='relu')(x)
  x = layers.MaxPooling2D(2,2)(x)
  x = layers.Dropout(0.2)(x)

  x = layers.Conv2D(32, (3, 3), padding='same', activation='relu')(x)
  x = layers.MaxPooling2D(2,2)(x)
  x = layers.Dropout(0.2)(x)

  x = layers.Flatten()(x)
  x = layers.Dropout(0.2)(x)
  outputs = layers.Dense(nb_classes, activation='softmax')(x)

  model2 = keras.Model(model2_input, outputs, name='model2')


  model2.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'] )


  return model2
```

Ilustración 31

3.7. En que se fija el modelo (SHAPLEY)

Para ver en que valores se centra el modelo podemos utilizar la librería SHAPLEY que no es más que un enfoque teórico para explicar el resultado de cualquier modelo de aprendizaje. Esta librería conecta la asignación óptima de créditos con explicaciones locales utilizando valores de teoría de juegos. Para utilizar shapley primero tendremos que instalarlo en la máquina local, si no, no podremos importarlo:

How install extra packages

Google Colab installs a series of basic packages if we need any additional package just install it.

```
[1]  !pip install -q keras sklearn
     !pip install shap
```

Ilustración 32

SHAP

```
1
2 import shap
3
4 background = X_train[np.random.choice(X_train.shape[0], 100, replace=False)]
5
6 e = shap.DeepExplainer(model, background)
7
8 shap_values = e.shap_values(X_test[1:6])
9
10 shap.image_plot(shap_values, -X_test[1:6])
```

Ilustración 33

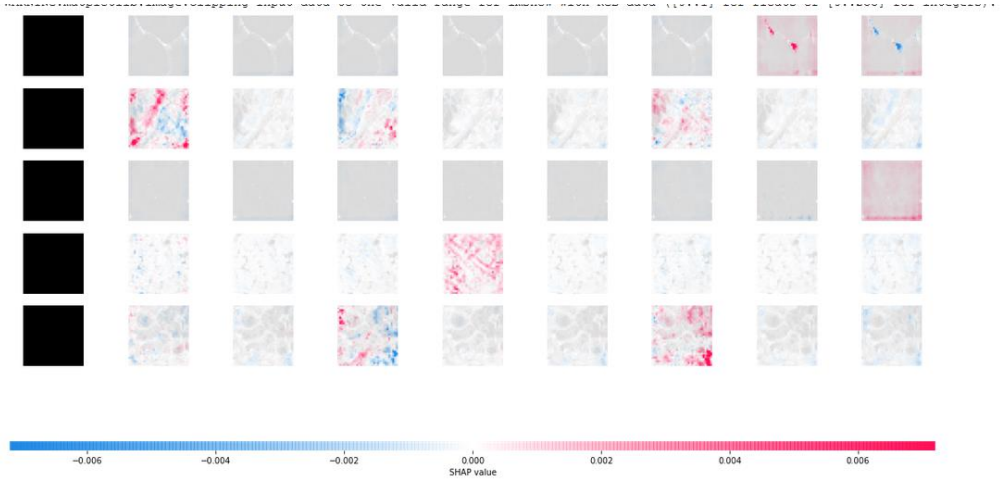


Ilustración 34

Finalmente, podemos ver en la captura superior en que zonas de las imágenes se fija el modelo para emitir un veredicto. Cada una de las columnas (sin contar la de la izquierda) es una de las clases de nuestro conjunto de datos.