Análisis de redes neuronales profundas

Una aplicación en detección de severidad de Parkinson

Por: Jéssica Vera Bermúdez

Fecha: 2021-11-24

El siguiente análisis empleará redes nuronales profundas para predecir la severidad de la enfermedad de Parkinson, a través de mediciones biomédicas de voz.

Los datos contienen 5,875 grabaciones de voz realizadas entre 42 pacientes diagnosticados con la enfermedad de Parkinson, a los que se les realizaron 16 mediciones biomédicas de su voz por un período de 6 meses, a través de un dispositivo de telemonitorización para el monitoreo remoto de la progresión de los síntomas. Para estas mediciones, los pacientes reclutados se encontraban en etapa temprana al momento del estudio.

La severidad de los casos dependerá de si la puntuación clínica total supera los 25 puntos. En este caso, el ejemplo sería severo; caso contrario no lo sería.

Más detalles sobre el diseño de esta investigación puede obtenerse en el sitio web oficial.

Para iniciar, cargaremos los paquetes que emplearemos en el análisis

In [1]:
```
# Paquetes a importar
from sklearn import datasets
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc, roc_auc_score
from sklearn.preprocessing import MinMaxScaler
from mlxtend.plotting import plot_decision_regions
from sklearn.metrics import confusion_matrix

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

Importación de los datos

Así también, realizaremos el acceso al directorio de Google Drive para poder importar nuestros datos que se alojan en dicha unidad. Esto lo lograremos mediante la función `drive` de `google.colab`

In [2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Una vez realizada la conexión con Google Drive, cargaremos los datos `parkinsons_updrs.data`

con la función `read_csv` de `pandas`. Para este grupo de datos, el separador es la coma ",", por lo que, especificaremos este parámetro.

In [3]:
```
# Cargamos el archivo de datos
datparkinson =  pd.read_csv('/content/drive/My Drive/Colab Notebooks/parkinsons_updrs.d

# en caso de que este notebook se trabaje desde el escritorio y el archivo de datos est
# datparkinson =  pd.read_csv('parkinsons_updrs.data', sep=",")
```

Verificamos que la carga se haya realizado correctamente, al observar los primeros registros del dataset, mediante la instrucción `head`.

In [4]:
```
datparkinson.head(5)
```

Out[4]:

	subject#	age	sex	test_time	motor_UPDRS	total_UPDRS	Jitter(%)	Jitter(Abs)	Jitter:RAP	Jitter:PF
0	1	72	0	5.6431	28.199	34.398	0.00662	0.000034	0.00401	0.00
1	1	72	0	12.6660	28.447	34.894	0.00300	0.000017	0.00132	0.00
2	1	72	0	19.6810	28.695	35.389	0.00481	0.000025	0.00205	0.00
3	1	72	0	25.6470	28.905	35.810	0.00528	0.000027	0.00191	0.00
4	1	72	0	33.6420	29.187	36.375	0.00335	0.000020	0.00093	0.00

Podemos confirmar que el el tipo de objeto cargado es `DataFrame`.

In [5]:
```
type(datparkinson)
```

Out[5]:
```
pandas.core.frame.DataFrame
```

El dataset cuenta con 22 columnas y 5,875 filas que corresponden a las grabaciones de voz de los pacientes.

In [6]:
```
datparkinson.shape
```

Out[6]:
```
(5875, 22)
```

Para este análisis usaremos únicamente las 16 mediciones de voz y la puntuación total UPDRS que nos ayudará a determinar la severidad. Veamos los nombres de las columnas para seleccionar solo las que nos interesan.

In [7]:
```
datparkinson.columns
```

Out[7]:
```
Index(['subject#', 'age', 'sex', 'test_time', 'motor_UPDRS', 'total_UPDRS',
       'Jitter(%)', 'Jitter(Abs)', 'Jitter:RAP', 'Jitter:PPQ5', 'Jitter:DDP',
       'Shimmer', 'Shimmer(dB)', 'Shimmer:APQ3', 'Shimmer:APQ5',
       'Shimmer:APQ11', 'Shimmer:DDA', 'NHR', 'HNR', 'RPDE', 'DFA', 'PPE'],
      dtype='object')
```

In [8]:
```
# seleccion de las columnas que usaremos
datospark = datparkinson.loc[:, 'total_UPDRS':'PPE']
```

Estadísticas descriptivas

El resumen del conjunto de datos en análisis, nos permite observar que no existen datos faltantes en columna alguna. Así también, observamos que la proporción de grabaciones de voz que sugerirían que no son casos severos es menor a la proporción de los casos severos, ya que la mediana reportada para la puntuación total de las mediciones `total_UPDRS` es 27.5. El grupo de grabaciones de voz en estudio tiene un valor promedio de 29 y una desviación estándar de 10.7.

Respecto a las mediciones biomédicas de la voz, podemos observar que en su mayoría no superan el valor de 1, no obstante, existen algunas de ellas que presentan una alta variabilidad con valores por encima de 2, como es el caso de `Shimmer(dB)` y `HNR` .

In [9]:
```
datospark.describe(include='all')
```

Out[9]:

	total_UPDRS	Jitter(%)	Jitter(Abs)	Jitter:RAP	Jitter:PPQ5	Jitter:DDP	Shimmer	Shi
count	5875.000000	5875.000000	5875.000000	5875.000000	5875.000000	5875.000000	5875.000000	58
mean	29.018942	0.006154	0.000044	0.002987	0.003277	0.008962	0.034035	
std	10.700283	0.005624	0.000036	0.003124	0.003732	0.009371	0.025835	
min	7.000000	0.000830	0.000002	0.000330	0.000430	0.000980	0.003060	
25%	21.371000	0.003580	0.000022	0.001580	0.001820	0.004730	0.019120	
50%	27.576000	0.004900	0.000035	0.002250	0.002490	0.006750	0.027510	
75%	36.399000	0.006800	0.000053	0.003290	0.003460	0.009870	0.039750	
max	54.992000	0.099990	0.000446	0.057540	0.069560	0.172630	0.268630	

Normalización

Dado que nuestras variables predictoras presentan alta variabilidad en sus escalas, realizaremos un escalamiento, mediante normalización de mínimos y máximos. Esto es posible mediante la función `MinMaxScaler()` .

In [10]:
```
# Seleccionamos el método de normalización minmax
normalizador = MinMaxScaler()
# ajustamos al grupo de variables predictoras
normalizador.fit(datospark[datospark.columns[1:datospark.columns.size]])
# realizamos la transformación
prednorm = normalizador.transform(datospark[datospark.columns[1:datospark.columns.size]
# convertir el array en DataFrame
predictores = pd.DataFrame(prednorm, columns=['Jitter(%)', 'Jitter(Abs)', 'Jitter:RAP',
       'Shimmer', 'Shimmer(dB)', 'Shimmer:APQ3', 'Shimmer:APQ5',
       'Shimmer:APQ11', 'Shimmer:DDA', 'NHR', 'HNR', 'RPDE', 'DFA', 'PPE'])
```

De este modo, podemos observar que tras la transformación, todas nuestras variables predictoras se encuentran en el rango de 0 a 1.

In [11]:
```
# Obteniendo el mínimo y máximo de cada variable
predictores.describe(include='all').loc[['min', 'max']]
```

Out[11]:

	Jitter(%)	Jitter(Abs)	Jitter:RAP	Jitter:PPQ5	Jitter:DDP	Shimmer	Shimmer(dB)	Shimmer:APQ3
min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
max	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Creación de la variable target

Nuestro objetivo es conocer la severidad de los casos, dependiendo de la puntuación total obtenida de las mediciones biomédicas de voz. Para ello, los investigadores sugieren que una puntuación total de las mediciones de voz superior a 25, indicaría que la condición del paciente es severa. Por tanto, crearemos una variable binaria resultado de analizar si la variable `total_UPDRS` es mayor o no a 25. En caso de serlo, asignaremos el valor de 1, caso contrario, 0.

In [12]:
```
# binarización de la variable target
target = np.where(datospark['total_UPDRS']>25,1,0)
```

Verificamos que la nueva variable `target` únicamente toma los valores 0 y 1, mediante la función `unique`. Tal y como lo habíamos mencionado previamente, hay un mayor número de casos severos en el conjunto de datos analizados.

In [13]:
```
# Contando los valores únicos de target
(unique, counts) = np.unique(target, return_counts=True)
frequencies = np.asarray((unique, counts)).T
#print(frequencies)
frequencies
```

Out[13]:
```
array([[   0, 2188],
       [   1, 3687]])
```

División de la muestra

Para proceder a construir el modelo de la red neuronal, dividiremos la muestra en *entrenamiento* (67%) y *prueba* (33%). De este modo, en la muestra de entrenamiento se construirá el modelo y en la de prueba será evaluado. Para ello usaremos la función `train_test_split` y definiremos una semilla para asegurar reproducibilidad, mediante la fijación del parámetro `random_state` a cero.

In [14]:
```
# Dividir tanto las variables predictoras como la de target en train y test
xtrain, xtest, ytrain, ytest = train_test_split(predictores, target, test_size=0.33, ra
xtrain = xtrain.to_numpy()
xtest = xtest.to_numpy()
```

Comprobando que las dimensiones de las muestras divididas cumplen con lo requerido, observamos que el número de grabaciones en la muestra de entrenamiento es de 3,936 y en la de prueba es 1,939.

In [15]:
```
# Conjunto de predictoras en muestra de entrenamiento
print("xtrain: " + str(xtrain.shape))
# Variable target en la muestra de entrenamiento
print("ytrain: " + str(ytrain.shape))
# Conjunto de predictoras en muestra de entrenamiento
print("xtest: " + str(xtest.shape))
# Variable target en la muestra de entrenamiento
print("ytest: " + str(ytest.shape))
```

```
xtrain: (3936, 16)
ytrain: (3936,)
xtest: (1939, 16)
ytest: (1939,)
```

Ahora verificamos que las clases (severidad o no) mantienen similar proporción entre ambas muestras.

In [16]:
```
# Contando los valores únicos de target en muestra de entrenamiento
(uniquetrain, countstrain) = np.unique(ytrain, return_counts=True)
frequenciestrain = np.asarray((uniquetrain, countstrain)).T

# Contando los valores únicos de target en muestra de entrenamiento
(uniquetest, countstest) = np.unique(ytest, return_counts=True)
frequenciestest = np.asarray((uniquetest, countstest)).T

print("Train: " + str(frequenciestrain[:,1]/3936))
print("Test: " + str(frequenciestest[:,1]/1939))
```

```
Train: [0.37245935 0.62754065]
Test: [0.37235688 0.62764312]
```

Redes neuronales

A continuación construiremos dos redes neuronales profundas densas (*fully connected*) para predecir la severidad de las grabaciones de voz y evaluaremos el desempeño de ambas. Ambas harán uso del tipo de modelo secuencial, ya que nos interesa modelar la red de modo que cada capa anterior sea entrada de la capa posterior. Para este propósito usaremos la herramienta `keras`, por lo que procederemos a importar las dependencias necesarias.

In [17]:
```
# Dependencias a usar
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
from tensorflow.keras.optimizers import SGD
```

Red neuronal de dos capas ocultas

A continuación crearemos una red neuronal profunda densa de dos capas ocultas, cada una con 10 neuronas.

Construcción

Iniciaremos definiendo una semilla para asegurar la reproducibilidad de la red.

Para definir esta red requeriremos definir

- el número de entradas, que en este caso son las 16 mediciones biomédicas de voz;
- el número de neuronas de cada una de las dos capas ocultas, en este caso 10 para ambas; y
- el tamaño de la salida, que en este caso es una neurona que nos indicará si la grabación representa condición severa o no.

La función de activación que usaremos entre las capas ocultas será `ReLu`, ya que según el estado del arte, es más rápida para entrenar las capas ocultas, y para la capa de salida será `sigmoid`, ya que según la literatura, es adecuada para el objetivo que buscamos: clasificar en una de las dos clases excluyentes referentes a la severidad.

In [18]:
```
# Semilla
np.random.seed(123)
```

In [19]:
```
# Tipo de modelo secuencial
model1 = Sequential()

# Primera capa oculta: 10 neuronas, 16 nodos de entrada, función de activación ReLu
model1.add(Dense(10, input_dim=16, activation='relu'))
# Segunda capa oculta: 10 neuronas, función de activación ReLu
model1.add(Dense(10, activation='relu'))
# Capa de salida: 1 neurona, función de activación sigmoide
model1.add(Dense(1, activation='sigmoid'))
```

Ahora definiremos la función de costo, el optimizador, las métricas y la tasa de aprendizaje. Para este caso, emplearemos como optimizador al algoritmo de gradiente descendiente Vanila `SGD` con una tasa de aprendizaje `lr = 0.1`. La función de costo que usaremos será el error cuadrático medio `mean_squared_error` y la métrica para juzgar sobre el desempeño de la red neuronal será la precisión `accuracy`.

In [20]:
```
model1.compile(optimizer=SGD(lr=0.1),
               loss='mean_squared_error',
               metrics=['accuracy'])
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/gradient_descent.py:102: UserW
arning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(SGD, self).__init__(name, **kwargs)
```

Entrenamiento

Ahora que hemos culminado con la construcción de la red neuronal, procederemos a entrenar la red, haciendo uso de la muestra de entrenamiento: `xtrain` e `ytrain`. Para este efecto,

definiremos el número de épocas en 300 y una división de validación de 0.20. Esto con el fin de observar si existe sobreajuste.

In [21]:
```
history = model1.fit(xtrain, ytrain, validation_split=0.2, epochs=300, verbose=None)
```

In [22]:
```
# Summary of last epoch
print('Accuracy train: ' + str(history.history['accuracy'][-1]))
print('Accuracy validation: ' + str(history.history['val_accuracy'][-1]))
print('Loss train: ' + str(history.history['loss'][-1]))
print('Loss validation: ' + str(history.history['val_loss'][-1]))
```

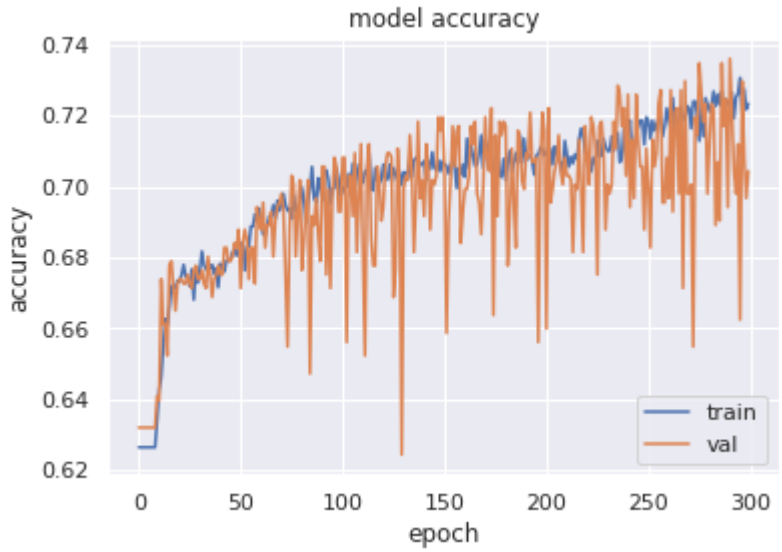
```
Accuracy train: 0.7233163714408875
Accuracy validation: 0.7043147087097168
Loss train: 0.18462778627872467
Loss validation: 0.1931856870651245
```

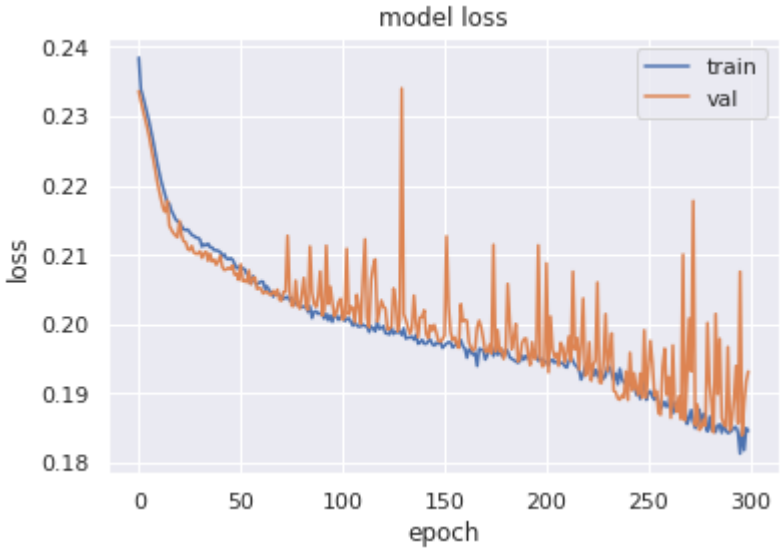
Para resumir, graficaremos cómo se han dado la precisión `accuracy` y el costo `loss` de entrenamiento y de validación.

Los gráficos muestran que la data de validación presenta alta variabilidad, a diferencia de la data de entrenamiento, tanto en el error cuadrático medio como en la precisión. De acuerdo con esta información, la precisión de ambos grupos de datos es similar, al igual que el error cuadrático medio.

In [23]:
```
# Accuracy
# Graficar el accuracy de entrenamiento
plt.plot(history.history['accuracy'])
# Graficar el accuracy de la validación
plt.plot(history.history['val_accuracy'])
# título del gráfico
plt.title('model accuracy')
# etiqueta de los ejes
plt.ylabel('accuracy')
plt.xlabel('epoch')
# leyenda
plt.legend(['train', 'val'], loc='lower right')
plt.show()

# Loss
# Graficar el costo de entrenamiento
plt.plot(history.history['loss'])
# Graficar el costo de la validacion
plt.plot(history.history['val_loss'])
# título del gráfico
plt.title('model loss')
# etiqueta de los ejes
plt.ylabel('loss')
plt.xlabel('epoch')
# leyenda
plt.legend(['train', 'val'], loc='upper right')
plt.show()
```





Test

Al evaluar la red neuronal que entrenamos en la data de prueba, se obtiene una precisión de 69% y el error cuadrático medio fue 0.19.

In [24]:
```
# evaluate the model
test_results = model1.evaluate(xtest, ytest, verbose=1)
print(f'Test results - Loss: {test_results[0]} - Accuracy: {test_results[1]*100}%')
```

```
61/61 [==============================] - 0s 1ms/step - loss: 0.1933 - accuracy: 0.6911
Test results - Loss: 0.19327816367149353 - Accuracy: 69.10778880119324%
```

Si ahora analizamos cómo se comportan la precisión y el costo, para ambas muestras, observamos que ambas medidas presentan similar comportamiento entre las muestras. Esto indica que la red ajustada no presenta problemas de sobreajuste.

Adicionalmente, podemos observar que en la data de validación, existe un mínimo de la precisión y un máximo del error cuadrático medio entre las épocas 60 y 80.

In [25]:
```
historytest = model1.fit(xtrain, ytrain, validation_data=(xtest,ytest), epochs=100, ver
```

In [26]:

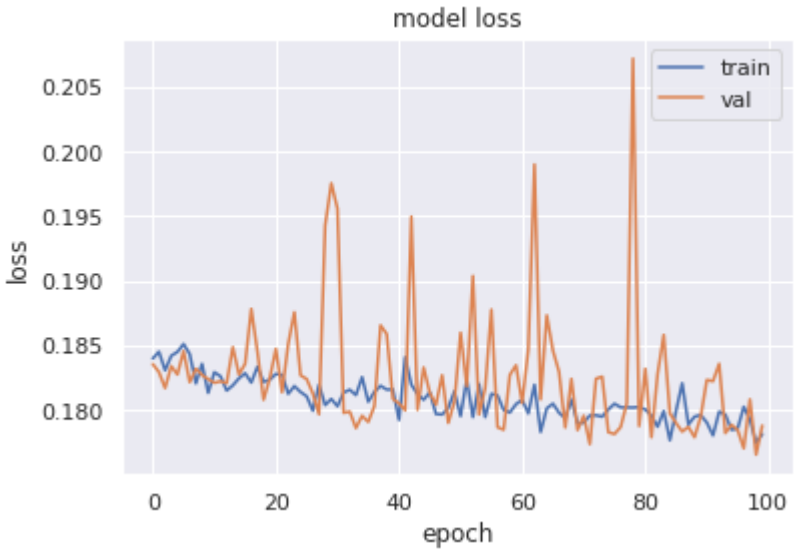
```
# Summary of last epoch
print('Accuracy train: ' + str(historytest.history['accuracy'][-1]))
print('Accuracy test: ' + str(historytest.history['val_accuracy'][-1]))
print('Loss train: ' + str(historytest.history['loss'][-1]))
print('Loss test: ' + str(historytest.history['val_loss'][-1]))
```

```
Accuracy train: 0.730182945728302
Accuracy test: 0.7287261486053467
Loss train: 0.17813873291015625
Loss test: 0.17879840731620789
```

In [27]:

```
# Accuracy
# Graficar el accuracy de entrenamiento
plt.plot(historytest.history['accuracy'])
# Graficar el accuracy de la validación
plt.plot(historytest.history['val_accuracy'])
# título del gráfico
plt.title('model accuracy')
# etiqueta de los ejes
plt.ylabel('accuracy')
plt.xlabel('epoch')
# leyenda
plt.legend(['train', 'val'], loc='lower right')
plt.show()

# Loss
# Graficar el costo de entrenamiento
plt.plot(historytest.history['loss'])
# Graficar el costo de la validacion
plt.plot(historytest.history['val_loss'])
# título del gráfico
plt.title('model loss')
# etiqueta de los ejes
plt.ylabel('loss')
plt.xlabel('epoch')
# leyenda
plt.legend(['train', 'val'], loc='upper right')
plt.show()
```



Red neuronal de tres capas ocultas

A continuación crearemos una red neuronal profunda densa de tres capas ocultas, de 10, 20 y 10 neuronas, respectivamente.

Construcción

Iniciaremos definiendo una semilla para asegurar la reproducibilidad de la red y definiendo que el modelo será secuencial.

Para definir esta red requeriremos definir

- el número de entradas, que en este caso son las 16 mediciones biomédicas de voz;
- el número de neuronas de cada una de las tres capas ocultas, en este caso 10, 20 y 10, respectivamente; y
- el tamaño de la salida, que en este caso es una neurona que nos indicará si la grabación representa condición severa o no.

La función de activación que usaremos entre las capas ocultas, al igual que en la red anterior, será `ReLu` y para la capa de salida será `sigmoid`, por las razones explicadas en la red neuronal previa.

In [28]:

```
# Semilla
np.random.seed(123)

# Tipo de modelo secuencial
model2 = Sequential()

# Primera capa oculta: 10 neuronas, 16 nodos de entrada, función de activación ReLu
model2.add(Dense(10, input_dim=16, activation='relu'))
# Segunda capa oculta: 20 neuronas, función de activación ReLu
model2.add(Dense(20, activation='relu'))
# Tercera capa oculta: 10 neuronas, función de activación ReLu
model2.add(Dense(10, activation='relu'))
# Capa de salida: 1 neurona, función de activación sigmoide
model2.add(Dense(1, activation='sigmoid'))
```

Ahora definiremos la función de costo, el optimizador, las métricas y la tasa de aprendizaje. Para este caso, emplearemos los mismos parámetros que para la red neuronal anterior: como optimizador, algoritmo de gradiente descendiente Vanila `SGD` con una tasa de aprendizaje `lr = 0.1` ; como función de costo, el error cuadrático medio `mean_squared_error` ; y, como métrica, la precisión `accuracy` .

In [29]:
```
model2.compile(optimizer=SGD(lr=0.1),
               loss='mean_squared_error',
               metrics=['accuracy'])
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/gradient_descent.py:102: UserW
arning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(SGD, self).__init__(name, **kwargs)
```

Entrenamiento

Ahora que hemos culminado con la construcción de la red neuronal, procederemos a entrenar la red, haciendo uso de la muestra de entrenamiento: `xtrain` e `ytrain` . Para este efecto, definiremos el número de épocas en 300 y una división de validación de 0.20. Esto con el fin de observar si existe sobreajuste.

In [30]:
```
history2 = model2.fit(xtrain, ytrain, validation_split=0.2, epochs=300, verbose=None)
```

Para resumir, graficaremos cómo se han dado la precisión `accuracy` y el costo `loss` de entrenamiento y de validación.

Los gráficos muestran que la data de validación presenta alta variabilidad, a diferencia de la data de entrenamiento, tanto en el error cuadrático medio como en la precisión. De acuerdo con esta información, tanto la precisión como el error cuadrático medio presentan valores similares entre las muestras de entrenamiento y validación.

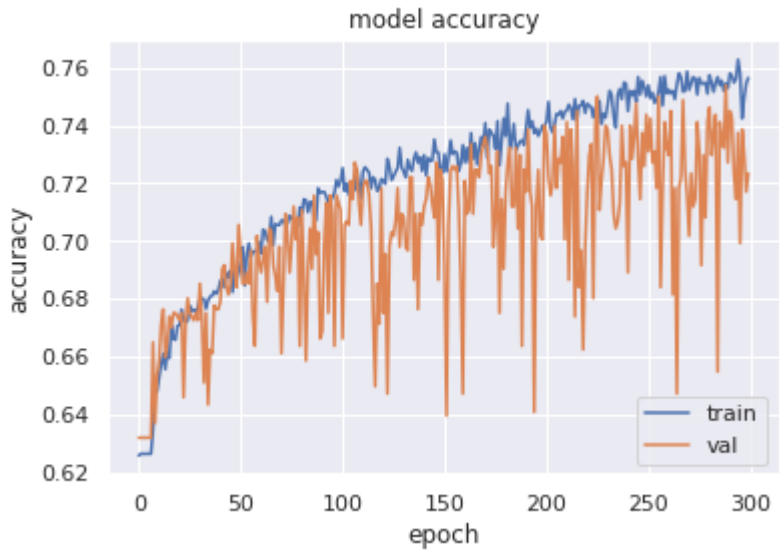
In [31]:
```
# Summary of last epoch
print('Accuracy train: ' + str(history2.history['accuracy'][-1]))
print('Accuracy validation: ' + str(history2.history['val_accuracy'][-1]))
print('Loss train: ' + str(history2.history['loss'][-1]))
print('Loss validation: ' + str(history2.history['val_loss'][-1]))
```

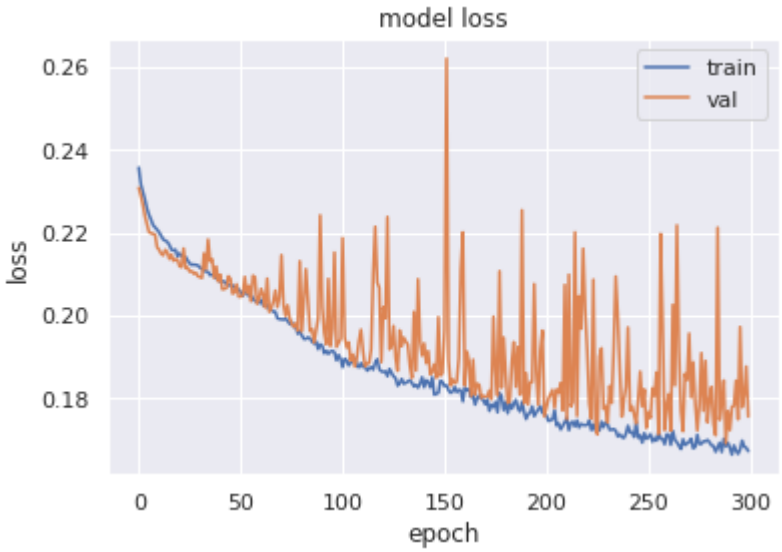
```
Accuracy train: 0.7563532590866089
Accuracy validation: 0.7233502268791199
Loss train: 0.16724491119384766
Loss validation: 0.1756085753440857
```

In [32]:
```
# Accuracy
# Graficar el accuracy de entrenamiento
plt.plot(history2.history['accuracy'])
# Graficar el accuracy de la validación
plt.plot(history2.history['val_accuracy'])
# título del gráfico
plt.title('model accuracy')
# etiqueta de los ejes
plt.ylabel('accuracy')
```

```
plt.xlabel('epoch')
# Leyenda
plt.legend(['train', 'val'], loc='lower right')
plt.show()

# Loss
# Graficar el costo de entrenamiento
plt.plot(history2.history['loss'])
# Graficar el costo de la validacion
plt.plot(history2.history['val_loss'])
# título del gráfico
plt.title('model loss')
# etiqueta de los ejes
plt.ylabel('loss')
plt.xlabel('epoch')
# Leyenda
plt.legend(['train', 'val'], loc='upper right')
plt.show()
```





Test

Al evaluar la red neuronal que entrenamos en la data de prueba, obtenemos una precisión 73% y el error cuadrático medio fue aproximadamente 0.18.

In [33]:
```
# evaluate the model
test_results = model2.evaluate(xtest, ytest, verbose=1)
print(f'Test results - Loss: {test_results[0]} - Accuracy: {test_results[1]*100}%')
```

```
61/61 [==============================] - 0s 1ms/step - loss: 0.1797 - accuracy: 0.7334
Test results - Loss: 0.17970150709152222 - Accuracy: 73.33677411079407%
```

Si ahora analizamos cómo se comportan la precisión y el costo, para ambas muestras, observamos que ambas medidas presentan similar comportamiento entre las muestras. La precisión reportada en ambas muestras fue similar, así como, el valor del costo. Esto indica que la red ajustada no presenta problemas de sobreajuste.

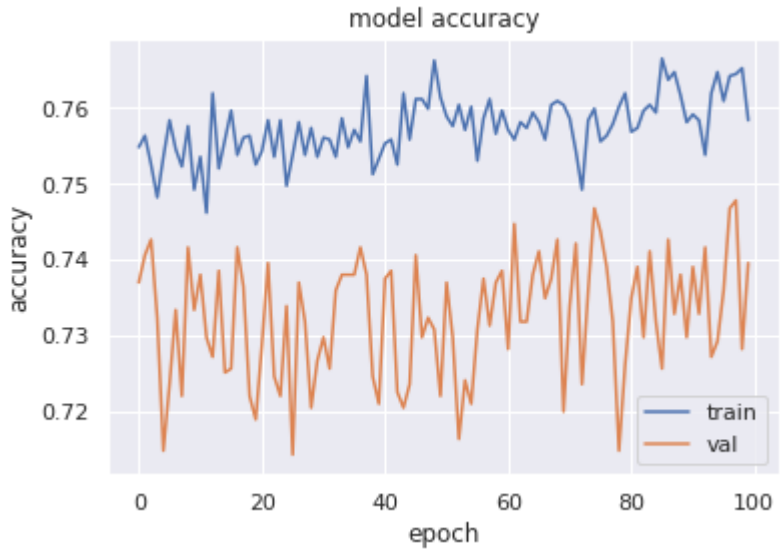
In [34]:
```
history2test = model2.fit(xtrain, ytrain, validation_data=(xtest,ytest), epochs=100, ve
```

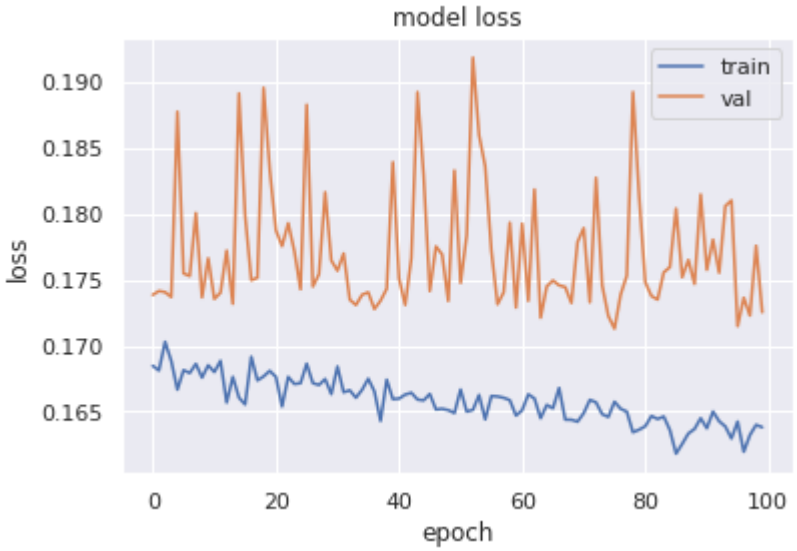
In [35]:
```
# Summary of last epoch
print('Accuracy train: ' + str(history2test.history['accuracy'][-1]))
print('Accuracy test: ' + str(history2test.history['val_accuracy'][-1]))
print('Loss train: ' + str(history2test.history['loss'][-1]))
print('Loss test: ' + str(history2test.history['val_loss'][-1]))
```

```
Accuracy train: 0.7583841681480408
Accuracy test: 0.7395564913749695
Loss train: 0.16382244229316711
Loss test: 0.17254121601581573
```

In [36]:
```
# Accuracy
# Graficar el accuracy de entrenamiento
plt.plot(history2test.history['accuracy'])
# Graficar el accuracy de la validación
plt.plot(history2test.history['val_accuracy'])
# título del gráfico
plt.title('model accuracy')
# etiqueta de los ejes
plt.ylabel('accuracy')
plt.xlabel('epoch')
# leyenda
plt.legend(['train', 'val'], loc='lower right')
plt.show()

# Loss
# Graficar el costo de entrenamiento
plt.plot(history2test.history['loss'])
# Graficar el costo de la validacion
plt.plot(history2test.history['val_loss'])
# título del gráfico
plt.title('model loss')
# etiqueta de los ejes
plt.ylabel('loss')
plt.xlabel('epoch')
# leyenda
plt.legend(['train', 'val'], loc='upper right')
plt.show()
```





Curvas ROC y AUC

En esta sección evaluaremos el desempeño de ambos modelos de redes neuronales mediante las curvas ROC y el indicador de AUC. Para ello es necesario obtener la probabilidad de que una grabación de voz pertenezca a un paciente con condición de Parkinson severo. Esto es posible mediante los valores predichos que devuelve cada modelo, ya que empleamos a la función sigmoide como de activación para la capa de salida y este valor representaría la probabilidad de que un caso sea severo.

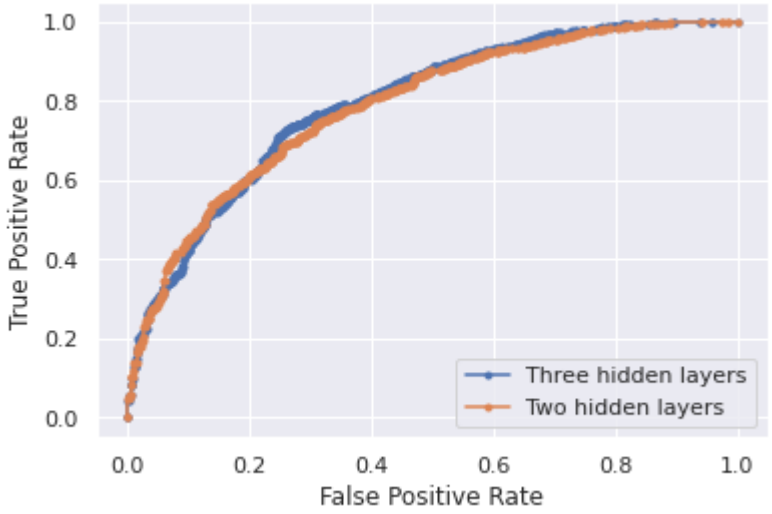
Las curvas y el área bajo las curvas de cada modelo, nos muestran que ambos modelos presentan similar desempeño.

In [37]:

```
# valores predichos - probabilidades de que un caso sea severo
pred1 = model1.predict(xtest) # modelo de dos capas ocultas
pred2 = model2.predict(xtest) # modelo de tres capas ocultas

# calcular curvas roc
fpr1, tpr1, _ = roc_curve(ytest, pred1)
fpr2, tpr2, _ = roc_curve(ytest, pred2)
```

```
# gráfico de las curvas ROC
plt.plot(fpr2, tpr2, marker='.', label='Three hidden layers')
plt.plot(fpr1, tpr1, marker='.', label='Two hidden layers')
# axis labels
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
# show the legend
plt.legend()
# show the plot
plt.show()
```



In [38]:

```
# Calcular AUC
auc1 = roc_auc_score(ytest, pred1) # modelo de dos capas ocultas
auc2 = roc_auc_score(ytest, pred2) # modelo de dos capas ocultas

print('AUC')
print('Modelo 2 capas ocultas: %.3f' % auc1)
print('Modelo 3 capas ocultas: %.3f' % auc2)
```

```
AUC
Modelo 2 capas ocultas: 0.790
Modelo 3 capas ocultas: 0.796
```

Matrices de confusión

A continuación, discretizaremos el valor de la predicción de ambos modelos, de modo que si supera el valor de 0.5, consideraremos que es un caso severo; en caso contrario, no lo es. Con esto calcularemos algunas de las medidas de desempeño de la predicción.

In [39]:

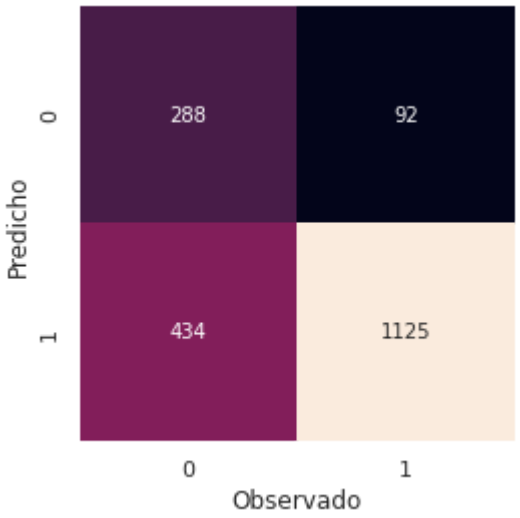
```
# binarización de las predicciones del modelo
ypred1 = np.where(pred1>0.5,1,0)
ypred2 = np.where(pred2>0.5,1,0)
```

Según los resultados de las matrices de confusión, el modelo de 2 capas ocultas tiene mayor número de aciertos en los casos severos, pero al mismo tiempo, menos aciertos en los casos no

severos, conllevando a una mayor tasa de falsos positivos, aunque la tasa de falsos negativos es menor.

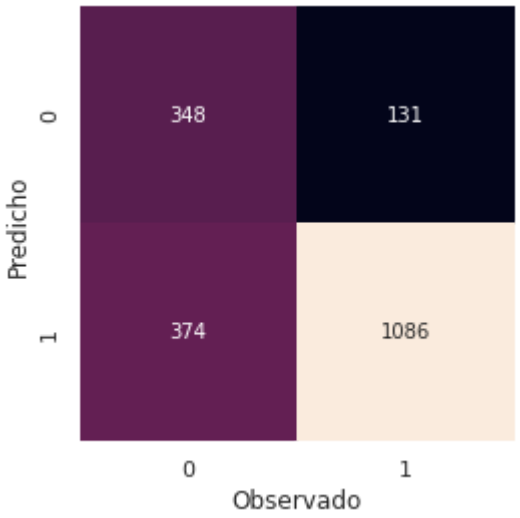
In [40]:
```
# Matriz de confusión para el modelo de 2 capas ocultas
mat1 = confusion_matrix(ypred1, ytest)
names1 = np.unique(ypred1)
sns.heatmap(mat1, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=names1, yticklabels=names1)
plt.xlabel('Observado')
plt.ylabel('Predicho')
```

Out[40]:
```
Text(89.18, 0.5, 'Predicho')
```



In [41]:
```
# Matriz de confusión para el modelo de 3 capas ocultas
mat2 = confusion_matrix(ypred2, ytest)
names2 = np.unique(ypred2)
sns.heatmap(mat2, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=names2, yticklabels=names2)
plt.xlabel('Observado')
plt.ylabel('Predicho')
```

Out[41]:
```
Text(89.18, 0.5, 'Predicho')
```



In [ ]:
```
# Medidas de desempeño de la matriz de confusión
n = len(ytest) # total datos en test

# Para el modelo 1: 2 capas ocultas
tp1 = mat1[1,1] # true positive
tn1 = mat1[0,0] # true negative
fp1 = mat1[1,0] # false positive
fn1 = mat1[0,1] # false negative
# totales
actualp1 = mat1.sum(axis=0)[1] # total positivos observados
actualn1 = mat1.sum(axis=0)[0] # total negativos observados
predictp1 = mat1.sum(axis=1)[1] # total positivos predichos
predictn1 = mat1.sum(axis=1)[0] # total negativos predichos
# algunos indicadores
accu1 = round((tn1+tp1)/n,3) # accuracy
errr1 = round(1-accu1,3) # error rate
sens1 = round(tp1/actualp1,3) # sensitivity
spec1 = round(tn1/actualn1,3) # specificity
prec1 = round(tp1/predictp1,3) # precision

# Para el modelo 1: 3 capas ocultas
tp2 = mat2[1,1] # true positive
tn2 = mat2[0,0] # true negative
fp2 = mat2[1,0] # false positive
fn2 = mat2[0,1] # false negative
# totales
actualp2 = mat2.sum(axis=0)[1] # total positivos observados
actualn2 = mat2.sum(axis=0)[0] # total negativos observados
predictp2 = mat2.sum(axis=1)[1] # total positivos predichos
predictn2 = mat2.sum(axis=1)[0] # total negativos predichos
# algunos indicadores
accu2 = round((tn2+tp2)/n,3) # accuracy
errr2 = round(1-accu2,3) # error rate
sens2 = round(tp2/actualp2,3) # sensitivity
spec2 = round(tn2/actualn2,3) # specificity
prec2 = round(tp2/predictp2,3) # precision

# creando lista de resultados de ambos modelos
performance = {'DosCapas':[accu1, errr1, sens1, spec1, prec1],
        'TresCapas':[accu2, errr2, sens2, spec2, prec2]}

# Crear DataFrame.
dfperf = pd.DataFrame(performance, index =['Accuracy', 'ErrorRate', 'Sensitivity', 'Spe
```

Ambos modelos presentan similares indicadores de desempeño, a diferencia de la especificidad, ya que como mencionamos previamente, el modelo de dos capas presenta menor desempeño en la predicción de casos no severos.

In [67]:
```
print("Medidas de desempeño de la capacidad predictiva de los modelos:")
print(dfperf)
```

```
Medidas de desempeño de la capacidad predictiva de los modelos:
             DosCapas  TresCapas
Accuracy        0.729      0.740
ErrorRate       0.271      0.260
Sensitivity     0.924      0.892
Specificity     0.399      0.482
Precision       0.722      0.744
```

Conclusiones

En este estudio se tuvo como objetivo predecir, mediante redes neuronales profundas densas, la severidad de la condición de pacientes con Parkinson (`0 = no severo` , `1 = severo`) al evaluar 16 mediciones biomédicas de su voz. Por lo que se entrenó dos redes neuronales profundas densas secuenciales: una de dos capas (10, 10 neuronas respectivamente) y otra de tres capas (10, 20, 10 neuronas respectivamente). Para las capas ocultas se empleó como función de activación a la función `ReLu` , mientras que para la capa de salida se empleó la función de activación `sigmoid` . El optimizador empleado fue el gradiente descendiente Vanila `SGD` con una tasa de aprendizaje de 0.1. El error cuadrático medio se definió como función de costo, y la precisión como métrica.

Los resultados indicaron que ambas redes neuronales entrenadas en este estudio presentaron similar desempeño en la predicción de la severidad de los pacientes considerando las medidas biomédicas de su voz. El modelo de dos capas ocultas presentó una ligera mayor precisión, sobre todo en la detección de casos de severidad. Mientras que el modelo de tres capas presentó una menor tasa de falsos positivos. No obstante, este desempeño descrito se intercambia entre una ejecución y otra, de modo, que en ocasiones el modelo de tres capas presenta mayor precisión en la detección de casos positivos, pero un menor desempeño en los casos que no son severos. Esto tiene que ver con la similitud en el desempeño de los modelos.