

Memoria Práctica 3

Neurocomputación - Ingeniería Informática

9 de mayo de 2021

Alejandro Santorum Varela
Sergio Galán Martín



Ciudad Universitaria de Cantoblanco
28049 Madrid
Tlf.: +34 91 497 50 00
Fax: +34 91 497 50 00
informacion.general@uam.es
<http://www.uam.es>

Índice

1 Introducción y estructura de la carpeta de la entrega	1
2 Experimentos considerados	3
2.1 Experimentos iniciales	3
2.2 Experimentos variando el optimizador	4
2.3 Experimentos variando el número de capas y neuronas	6
2.4 Búsqueda de hiperparámetros con Tuner	10
2.5 Experimentos usando Dropout	13
2.6 Experimentos utilizando regularización	17
2.7 Experimento final: generación sintética de ejemplos de la clase minoritaria usando SMOTE	18
3 Conclusiones	21

1. Introducción y estructura de la carpeta de la entrega

Llegamos a la **última práctica de Neurocomputación 2021**. En ella nos centraremos en practicar con la librería de Tensorflow **Keras** para el manejo cómodo, rápida y sencillo de redes neuronales. De esta forma podremos experimentar con varias estructuras de redes neuronales para un problema dado, en este caso el *dataset Phoneme* [1].

La **estructura de la carpeta de la entrega**, Practica3-Pareja03-Galan-Santorum, en este caso es muy sencilla, consta de un notebook de python P3-Galan-Santorum.ipynb donde hemos implementado todo el código y los experimentos de esta práctica; y esta memoria Memoria_Pr3.pdf, donde exponemos de manera sintética los experimentos realizados, los resultados y las conclusiones.

2. Experimentos considerados

2.1. Experimentos iniciales

Durante la práctica hemos podido experimentar y entrenar con la librería de manejo de redes neuronales Keras de Tensorflow. El *dataset* utilizado contiene datos sobre fonemas, descargable en [1].

Para empezar, hemos probado a usar un **número pequeño de neuronas en una sola capa oculta**, igual que el número de neuronas de la capa de entrada, es decir, la capa oculta tendrá tantas neuronas como atributos tenga el dataset (más el *bias*). Hemos usado el optimizador que venía especificado por defecto, el **sgd**. Adicionalmente, hemos vuelto a ejecutar un entrenamiento con una capa oculta más grande, con 128 neuronas. Los resultados se pueden ver en la figura 2.1.

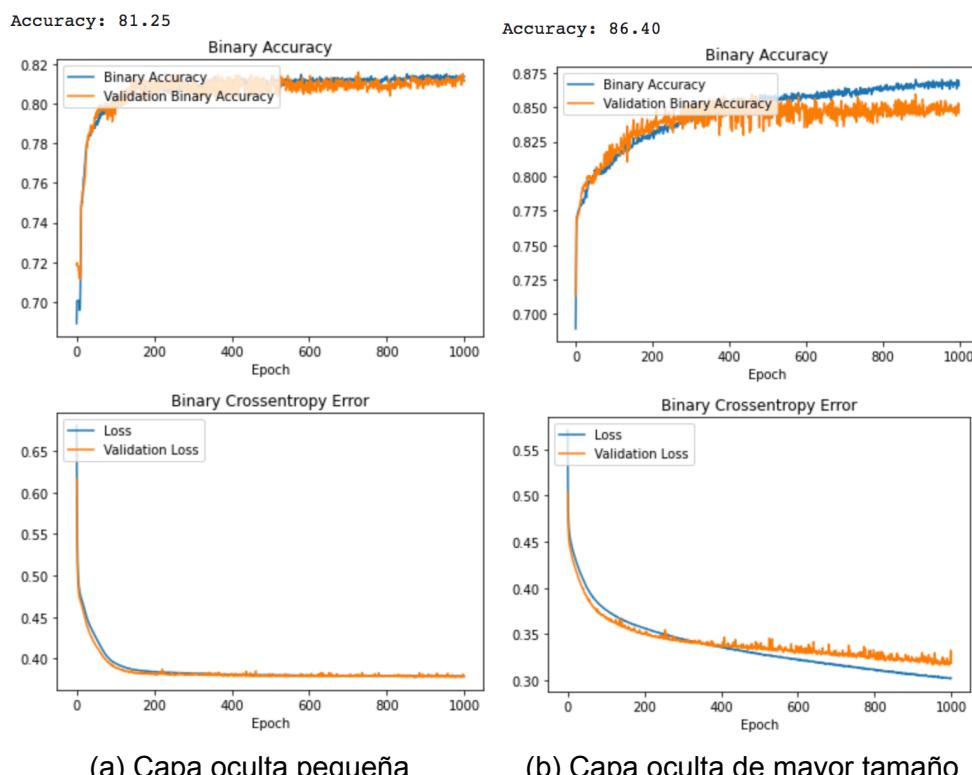


Figura 2.1: Probando diferentes tamaños capa oculta

Vemos que los resultados con un número bajo de neuronas son pobres, entorno al 81 % de precisión. Mejoran considerablemente cuando aumentamos el tamaño de la capa oculta, llegando al 86 %. El entrenamiento ha sido de 1000 épocas, las cuales son muchas para la red neuronal pequeña (el error apenas desciende en las últimas 600 épocas), pero podrían haber sido más para la red de mayor tamaño (aún no se había estabilizado el error de entropía binaria en la imagen 2.1b).

2.2. Experimentos variando el optimizador

Antes de nada, es importante destacar que las clases del *dataset* están **desbalanceadas, con un 70.6 % de una clase, y el resto de la otra**. Esto provoca que la métrica *accuracy* no sea la mejor para medir el rendimiento sobre este conjunto. Por esta razón, añadimos otras métricas: el *recall*, que intenta medir la capacidad del clasificador de encontrar los ejemplos positivos; y la *precision*, que es la capacidad del clasificador de encontrar los ejemplos negativos.

Ahora, y en los experimentos futuros, **separaremos el conjunto total de datos en tres subconjuntos**: *train set* (64 % del total), *validation set* (16 %) y *test set* (20 %). Así podremos comprobar el rendimiento de los modelos por época con el *validation set* y, al final del entrenamiento, medir los valores finales de las métricas en el *test set*.

En este apartado experimentaremos con los diferentes optimizadores utilizando la mejor estructura de red neuronal probada en la anterior sección 2.1. Empezamos probando el **Adam**. En la figura 2.2 podemos ver los resultados del entrenamiento.

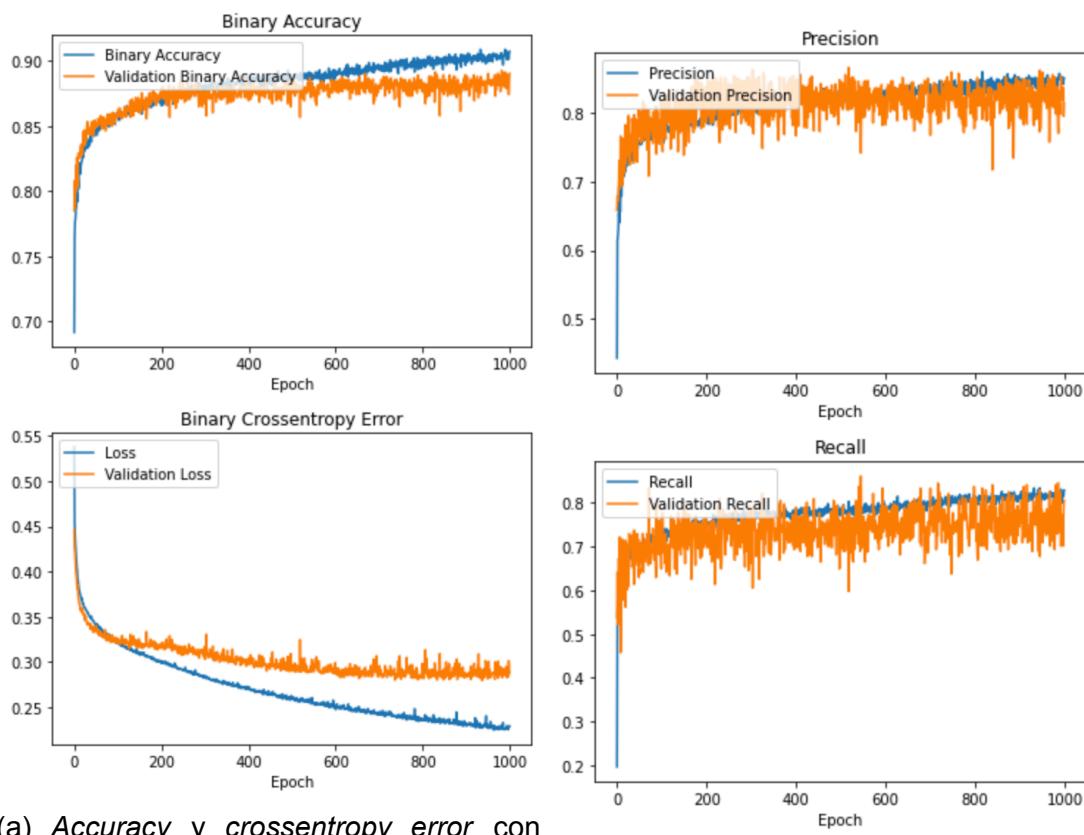


Figura 2.2: Probando el optimizador Adam tras 1000 épocas de entrenamiento

La gráfica 2.2a muestra que la *accuracy* ha ido creciendo paulatinamente época tras época. Podríamos aumentar el número de épocas para ver si aumenta la precisión final, pero la gráfica de evolución del coste indica que puede estar empezando a ocurrir

overfitting, al separarse ligeramente el coste de *train* y el de *validation*. En la figura 2.2b podemos ver que la *precision* y el *recall* siguen la misma tendencia que la *accuracy*. Los resultados, visibles en la figura 2.3, muestran una *accuracy* de **clasificación en el testset del 88.7 %, con una precision del 83.2 % y un recall del 77.5 %.**

```
136/136 [=====]
Train Binary Accuracy: 90.40
Train Precision: 83.46
Train Recall: 83.79
34/34 [=====
Test Binary Accuracy: 88.71
Test Precision: 83.28
Test Recall: 77.57
```

Figura 2.3: Resultados métricas Adam

Analizamos ahora el rendimiento usando el optimizador **RMSProp**. La figura 2.4 muestra las 4 gráficas equivalentes a las vistas en el caso del optimizador Adam. Como vemos, siguen todas la misma tendencia. No obstante, el rendimiento es ligeramente inferior en todas las métricas utilizadas. Esto se puede ver en la figura 2.5, con un *accuracy* del 87 %, una *precision* del 78 % y un *recall* del 76 %.

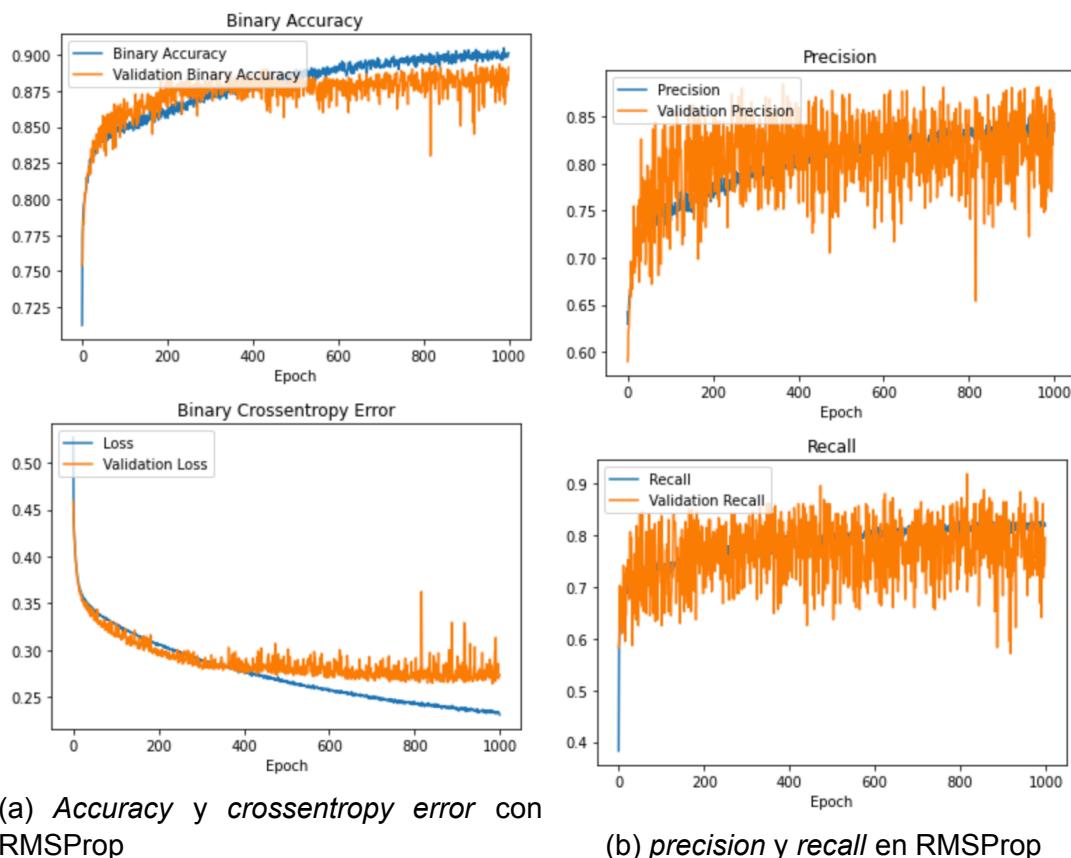


Figura 2.4: Probando el optimizador RMSProp tras 1000 épocas de entrenamiento

```
136/136 [=====]
Train Binary Accuracy: 90.17
Train Precision: 85.55
Train Recall: 80.41
34/34 [=====] -
Test Binary Accuracy: 87.42
Test Precision: 78.26
Test Recall: 76.72
```

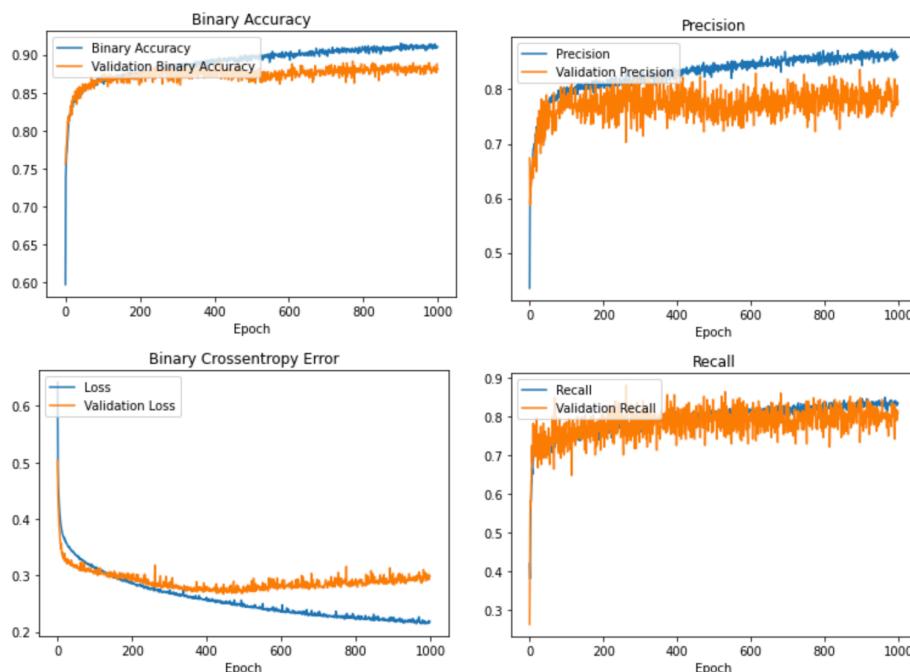
Figura 2.5: Resultados métricas RMSProp

En definitiva, **escogemos el optimizador Adam para su uso durante el resto de experimentos de la práctica**, siendo el mejor de los tres probados (adam, sgd, rmsprop).

2.3. Experimentos variando el número de capas y neuronas

Una vez fijado el optimizador a usar, vamos a intentar discernir la mejor **estructura de la red neuronal posible** para el *dataset*. Para ello, intentaremos **variar el número de capas ocultas**, así como el **número de neuronas por cada una de ellas**.

Tomamos como referencia los experimentos iniciales, vistos en el apartado 2.1, donde se han probado configuraciones con una sola capa oculta. Empezamos probando una red neuronal de **dos capas ocultas**, con **16 y 32 neuronas** respectivamente.



(a) *Accuracy* y *crossentropy error* (b) *precision* y *recall* con dos capas ocultas de 16 y 32 neuronas

Figura 2.6: Probando red neuronas de tres capas ocultas de 16 y 32 neuronas

En la figura 2.6 podemos ver las gráficas resultantes del entrenamiento de esta red neuronal, que muestran que las tres métricas han ido creciendo rápidamente en las 100 primeras épocas y después han seguido creciendo paulatinamente. La gráfica del error (métrica que se intenta minimizar en el entrenamiento), empieza a desvelar un poco de *overfitting* en las últimas épocas.

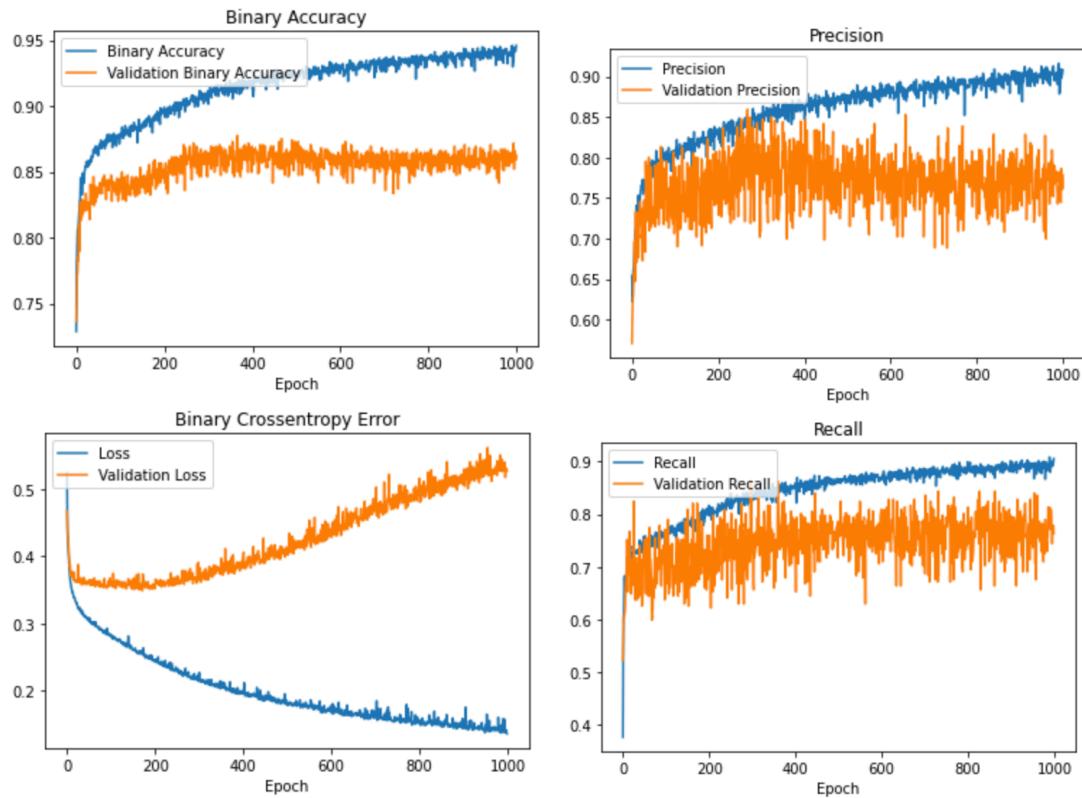
Podemos ver las métricas en el *test set* en la imagen 2.7 que en general el rendimiento es peor que el experimento de la sección 2.2 cuando probamos el optimizador Adam con una capa oculta y 128 neuronas. Solo hay una pequeña mejora en el *recall* del 1 %, pero se empeora más en las otras dos métricas.

```
136/136 [=====]
Train Binary Accuracy: 90.77
Train Precision: 84.55
Train Recall: 84.09
34/34 [=====] -
Test Binary Accuracy: 86.86
Test Precision: 76.25
Test Recall: 78.71
```

Figura 2.7: Resultados métricas con dos capas ocultas de 16 y 32 neuronas

Viendo que dos capas ocultas de 16 y 32 neuronas no es la mejor, probamos a aumentar el número de neuronas en las capas. En concreto, probamos con **32 y 64 neuronas** respectivamente. En la figura 2.8 podemos ver las gráficas que describen el proceso de entrenamiento. En líneas generales, las gráficas de las tres métricas muestran un claro estancamiento, siendo innecesario un entrenamiento de 1000 épocas. Por otro lado, la gráfica del error binario señala un claro *overfitting*, al mejorar en el *train set* pero al empeorar drásticamente en el *validation set*.

No obstante, los resultados en el *test set* son prometedores, consiguiendo un **88 % de precisión, un 80 % de precision y un 78 % de recall** (imagen 2.9). A pesar de sobre-entrenar el modelo, este parece que ha obtenido mejores resultados en el *test set*. En un futuro experimento intentaremos regularizar el modelo para ver si conseguimos mejorar estos resultados .



(a) *Accuracy* y *crossentropy error* con dos capas ocultas de 32 y 64 neuronas (b) *precision* y *recall* con dos capas ocultas de 32 y 64 neuronas

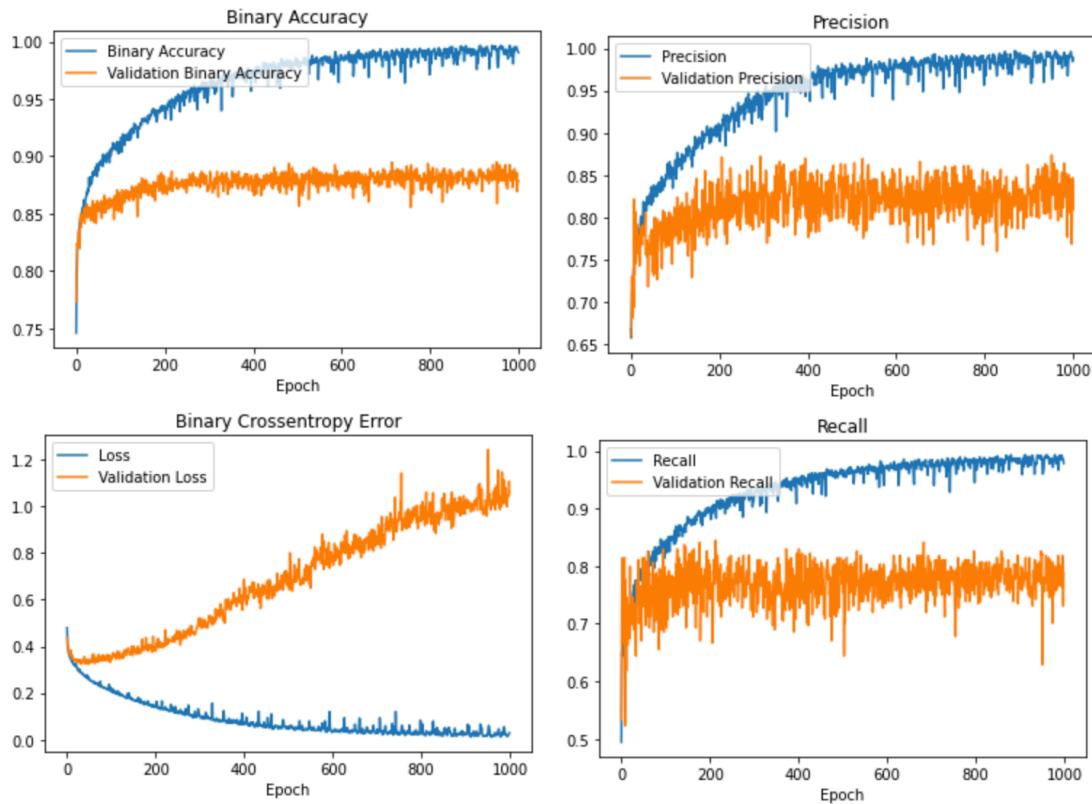
Figura 2.8: Probando red neuronas de tres capas ocultas de 32 y 64 neuronas

```
136/136 [=====]
Train Binary Accuracy: 93.25
Train Precision: 88.89
Train Recall: 87.91
34/34 [=====] -
Test Binary Accuracy: 88.07
Test Precision: 80.77
Test Recall: 78.50
```

Figura 2.9: Resultados métricas con dos capas ocultas de 32 y 64 neuronas

Nuestra curiosidad nos ha obligado a probar **tres capas ocultas**. Quizá sea innecesario, ya que vemos que el modelo está sobreentrenándose, pero no perdemos nada por probar.

En la figura 2.10 podemos ver los resultados gráficos del entrenamiento de una red neuronal de tres capas ocultas, de **32, 64 y 128 neuronas** respectivamente. Igual que en el caso anterior, las tres métricas consideradas han sufrido un estancamiento en el *validation set* a lo largo del entrenamiento, a partir de la época 100 aproximadamente. De igual forma, la gráfica del coste indica que sigue habiendo *overfitting*.



(a) *Accuracy* y *crossentropy error* con tres capas ocultas de 32, 64 y 128 neuronas (b) *precision* y *recall* con tres capas ocultas de 32, 64 y 128 neuronas

Figura 2.10: Probando red neuronal de tres capas ocultas de 32, 64 y 128 neuronas

Pero de nuevo las métricas en el *test set* nos sorprenden. Obtenemos un **91 % de accuracy, un 84 % de precision y un 83 % de recall**, visible en la imagen 2.11. Esto nos lleva a sospechar que el sobre-entrenamiento se está produciendo sobre el coste binario de error (que es la función que el modelo minimiza en el entrenamiento), pero no sobre las otras tres métricas de interés. Seguiremos experimentando, con estas configuraciones de red neuronal, así como añadiendo parámetros de **regularización**, para intentar mejorar estos resultados.

```
136/136 [=====]
Train Binary Accuracy: 97.25
Train Precision: 95.39
Train Recall: 95.32
34/34 [=====] -
Test Binary Accuracy: 90.75
Test Precision: 84.23
Test Recall: 82.57
```

Figura 2.11: Resultados métricas con dos capas ocultas de 32, 64 y 128 neuronas

2.4. Búsqueda de hiperparámetros con Tuner

En este apartado vamos a utilizar el afinador **Keras Tuner** para ratificar los resultados de la sección anterior. Podemos utilizar Tuner para realizar una **búsqueda de hiperparámetros en mallado o Grid Search**. Nos vamos a centrar en el número de neuronas por capa y en la constante de aprendizaje como hiperparámetros objetivo.

En primer lugar, fijamos el número de capas ocultas a **dos**, y decidimos probar (en cada una de ellas) **32, 48, 64, 80, 96, 112 y 128 neuronas**. Además, Analizamos qué **constante de aprendizaje** es la mejor entre las siguientes: **0.01, 0.001 y 0.0001**. Esto lo podemos encontrar en la imagen 2.12.

```
def build_model(hp):
    nn = Sequential()
    nn.add(Input(shape=(x_size,)))
    nn.add(Dense(hp.Int('units_1', min_value=32, max_value=128, step=16), activation="relu"))
    nn.add(Dense(hp.Int('units_2', min_value=32, max_value=128, step=16), activation="relu"))
    nn.add(Dense(1, activation="sigmoid"))
    nn.compile(optimizer=keras.optimizers.Adam(hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])),)
    return nn
```

Figura 2.12: Búsqueda de hiperparámetros con Tuner (dos capas ocultas)

Las mejores configuraciones encontradas utilizando Tuner son las de la imagen 2.13.

```
Results summary
Results in ./test_2
Showing 10 best trials
Objective(name='val_binary_accuracy', direction='max')
Trial summary
Hyperparameters:
units_1: 48
units_2: 96
learning_rate: 0.01
Score: 0.893308679262797
Trial summary
Hyperparameters:
units_1: 96
units_2: 64
learning_rate: 0.01
Score: 0.8923836151758829
Trial summary
Hyperparameters:
units_1: 112
units_2: 80
learning_rate: 0.01
Score: 0.890533447265625
```

Figura 2.13: Mejores configuraciones obtenidas con Tuner (dos capas ocultas)

Adicionalmente, vamos a fijar el número de capas ocultas a **tres**, y decidimos probamos los mismos hiperparámetros que en el experimento anterior. Esto lo podemos encontrar en la imagen 2.14.

```
def build_model_3_layer(hp):
    nn = Sequential()
    nn.add(Input(shape=(x_size,)))
    nn.add(Dense(hp.Int('units_1', min_value=32, max_value=128, step=16), activation="relu"))
    nn.add(Dense(hp.Int('units_2', min_value=32, max_value=128, step=16), activation="relu"))
    nn.add(Dense(hp.Int('units_3', min_value=32, max_value=128, step=16), activation="relu"))
    nn.add(Dense(1, activation="sigmoid"))
    nn.compile(optimizer=keras.optimizers.Adam(hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])))
    return nn
```

Figura 2.14: Búsqueda de hiperparámetros con Tuner (tres capas ocultas)

De nuevo, las mejores configuraciones encontradas utilizando Tuner son las de la imagen 2.15.

```
Results summary
Results in ./test_4
Showing 10 best trials
Objective(name='val_binary_accuracy', direction='max')
Trial summary
Hyperparameters:
units_1: 128
units_2: 80
units_3: 112
learning_rate: 0.01
Score: 0.892691950003306
Trial summary
Hyperparameters:
units_1: 64
units_2: 80
units_3: 64
learning_rate: 0.001
Score: 0.89053346713384
Trial summary
Hyperparameters:
units_1: 112
units_2: 64
units_3: 96
learning_rate: 0.0001
Score: 0.8606229027112325
```

Figura 2.15: Mejores configuraciones obtenidas con Tuner (tres capas ocultas)

Seleccionamos la mejor configuración utilizando dos capas ocultas: **48 neuronas en la primera capa oculta y 96 neuronas en la segunda. Constante de aprendizaje igual a 0.01**. Utilizando el optimizador **Adam** y tras **1000 épocas** de entrenamiento podemos ver las gráficas resultantes de este modelo en la figura 2.16.

Los resultados son parecidos a los que obteníamos en el apartado 2.3: las métricas de interés mejoran muy lentamente a partir de la época 100-200 y la gráfica de error binario indica que se está produciendo **sobreajuste**. Esto lo intentaremos tratar en las siguientes secciones, pero por ahora vamos a centrarnos en corroborar nuestros resultados con el *Tuner*.

En la imagen 2.17 encontramos los resultados de las tres métricas utilizadas para este experimento. Como vemos son muy parecidos a los obtenidos en la sección 2.3, con un 87 % de *accuracy*, un 81 % de *precision* y un 74 % de *recall*.

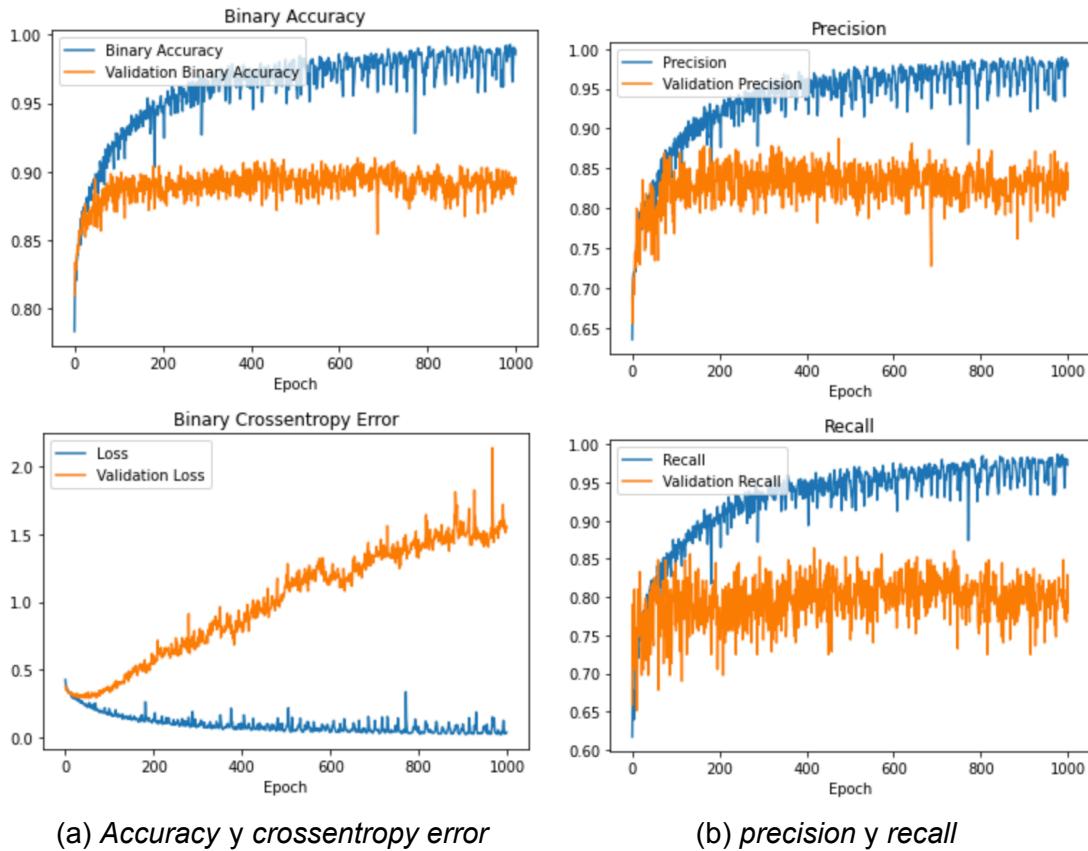


Figura 2.16: Gráficas de la mejor configuración obtenida con Tuner de 2 capas ocultas

```

136/136 [=====]
Train Binary Accuracy: 96.58
Train Precision: 95.85
Train Recall: 92.40
34/34 [=====] -
Test Binary Accuracy: 87.60
Test Precision: 80.99
Test Recall: 74.19

```

Figura 2.17: Resultados métricas de la mejor configuración obtenida con Tuner de 2 capas ocultas

Pasamos a probar la mejor configuración utilizando **tres capas ocultas**: 128 neuronas en la primera capa oculta, 80 neuronas en la segunda y 112 en la tercera. La constante de aprendizaje escogida es igual a 0.01. Utilizando el optimizador Adam y tras 1000 épocas de entrenamiento podemos ver las gráficas resultantes de este modelo en la figura 2.18.

No hay nada nuevo, las gráficas siguen la misma tendencia que con las configuraciones anteriores. No obstante, los resultados son diferentes: hemos obtenido de nuevo un **90 % de accuracy, y hemos aumentado considerablemente la precision, con un 82 %, y el recall, con un 82 %**. Esto se muestra en la imagen 2.19.

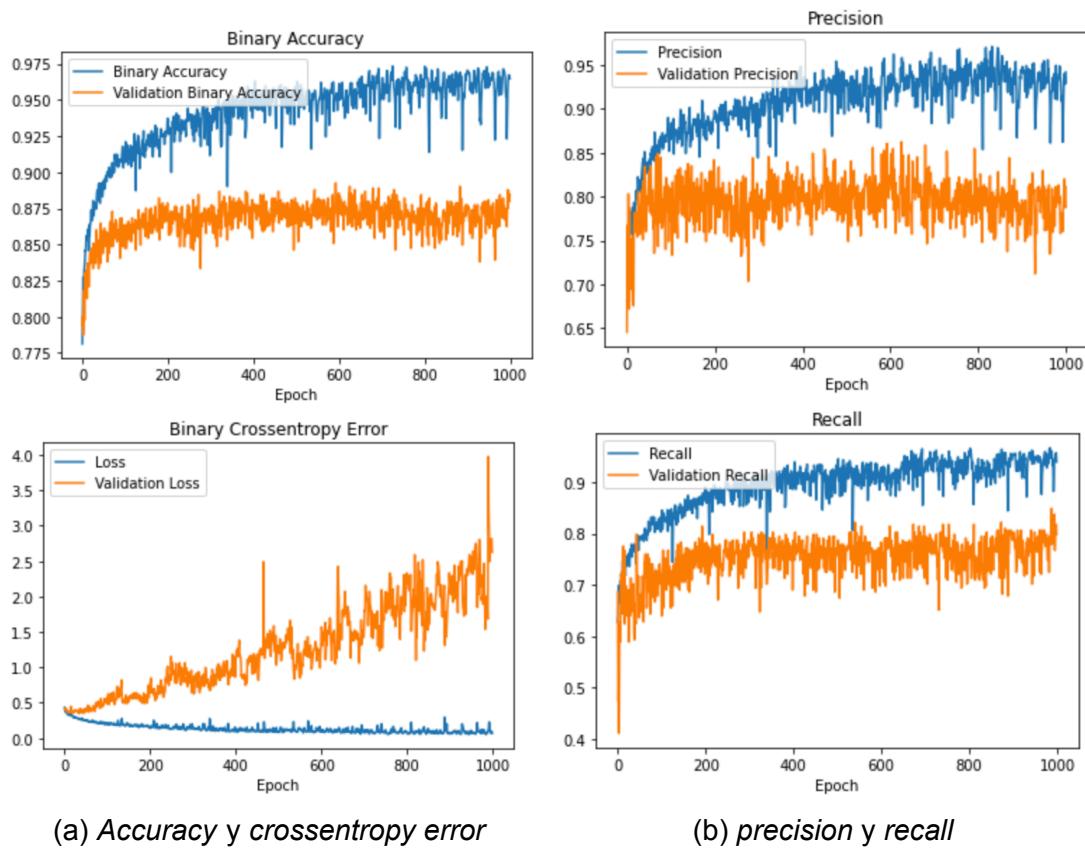


Figura 2.18: Gráficas de la mejor configuración obtenida con Tuner de 3 capas ocultas

```
136/136 [=====]
Train Binary Accuracy: 95.54
Train Precision: 91.74
Train Recall: 93.39
34/34 [=====] -
Test Binary Accuracy: 90.01
Test Precision: 81.85
Test Recall: 82.39
```

Figura 2.19: Resultados métricas de la mejor configuración obtenida con Tuner de 3 capas ocultas

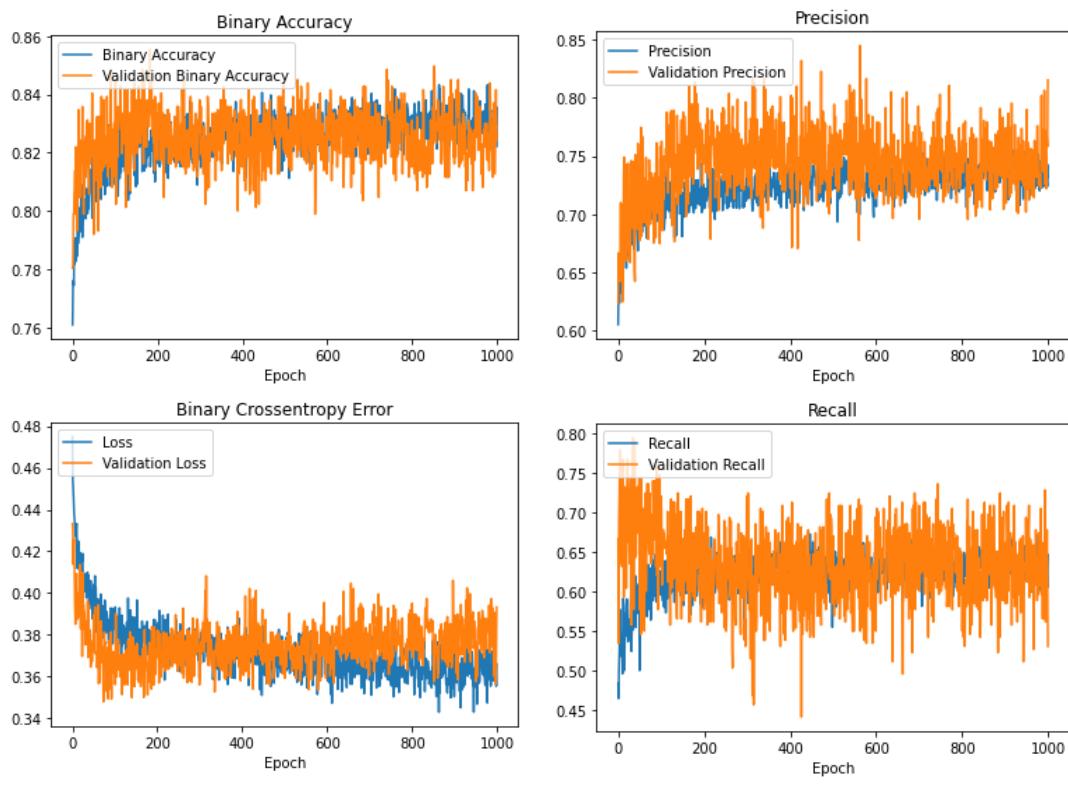
En las siguientes secciones nuestra atención se centrará en intentar mejorar el **proceso de entrenamiento para no sobreajustar los datos**. Para ello probaremos diferentes técnicas, como *dropout*, regularización y/o reducir el número de épocas de entrenamiento.

2.5. Experimentos usando Dropout

Ya que apreciamos cierto sobreajuste en los anteriores experimento, vamos a probar a aplicar una de las técnicas para reducir este fenómeno, el **Dropout**. Esta técnica consiste en **desactivar** cierto **porcentaje** de las neuronas de una capa a cada época.

ca, intentando suavizar la especificidad de la red. Keras nos permite implementar esta técnica de manera muy sencilla añadiendo una capa Dropout justo antes de la capa donde queremos aplicarla. Por ello, vamos a partir de la red de dos capas de 48 y 96 neuronas encontrada por el Tuner (ya que, pese a no ser la que mejores resultados tenía, se le acercaba y también parecía tener bastante sobreajuste).

Hemos realizado 3 ejecuciones, una aplicando un Dropout de **0.2 a ambas capas ocultas**, otra aplicando el **Dropout de 0.2 a la primera capa oculta** y otra aplicando el **Dropout de 0.2 a la segunda capa oculta**. Procedemos a exponer los resultados.



(a) Accuracy y crossentropy error

(b) precision y recall

Figura 2.20: Gráficas al aplicar Dropout de 0.2 a ambas capas ocultas

```

136/136 [=====]
Train Binary Accuracy: 82.77
Train Precision: 80.74
Train Recall: 52.61
34/34 [=====] -
Test Binary Accuracy: 79.93
Test Precision: 80.29
Test Recall: 48.69

```

Figura 2.21: Resultados al aplicar Dropout de 0.2 a ambas capas ocultas

Las figuras 2.20 y 2.21 nos indican que el *overfitting* ha sido **reducido considerablemente**. Esto lo podemos ver fácilmente en la gráfica del error binario, donde el error en el *train set* **no** es muy diferente del de *validation set*. No obstante, los resultados

son los peores hasta el momento, con un 79 % de *accuracy*, 80 % de *precision* y un pésimo *recall* del 49 %. Vamos por tanto a ligeroar la cantidad de neuronas apagadas mediante *dropout* para intentar mejorar el rendimiento.

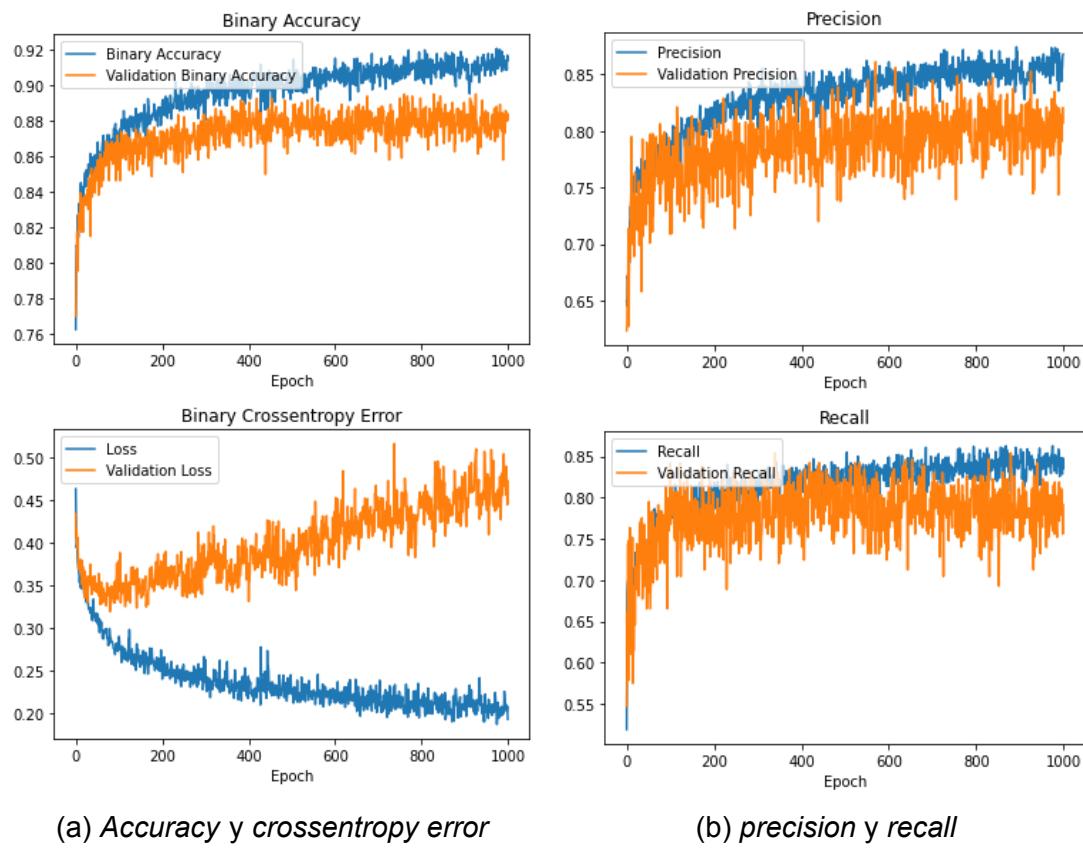


Figura 2.22: Gráficas al aplicar Dropout de 0.2 a la primera capa oculta

```

136/136 [=====]
Train Binary Accuracy: 93.01
Train Precision: 88.95
Train Recall: 87.14
34/34 [=====] -
Test Binary Accuracy: 88.34
Test Precision: 80.94
Test Recall: 77.81

```

Figura 2.23: Resultados al aplicar Dropout de 0.2 a la primera capa oculta

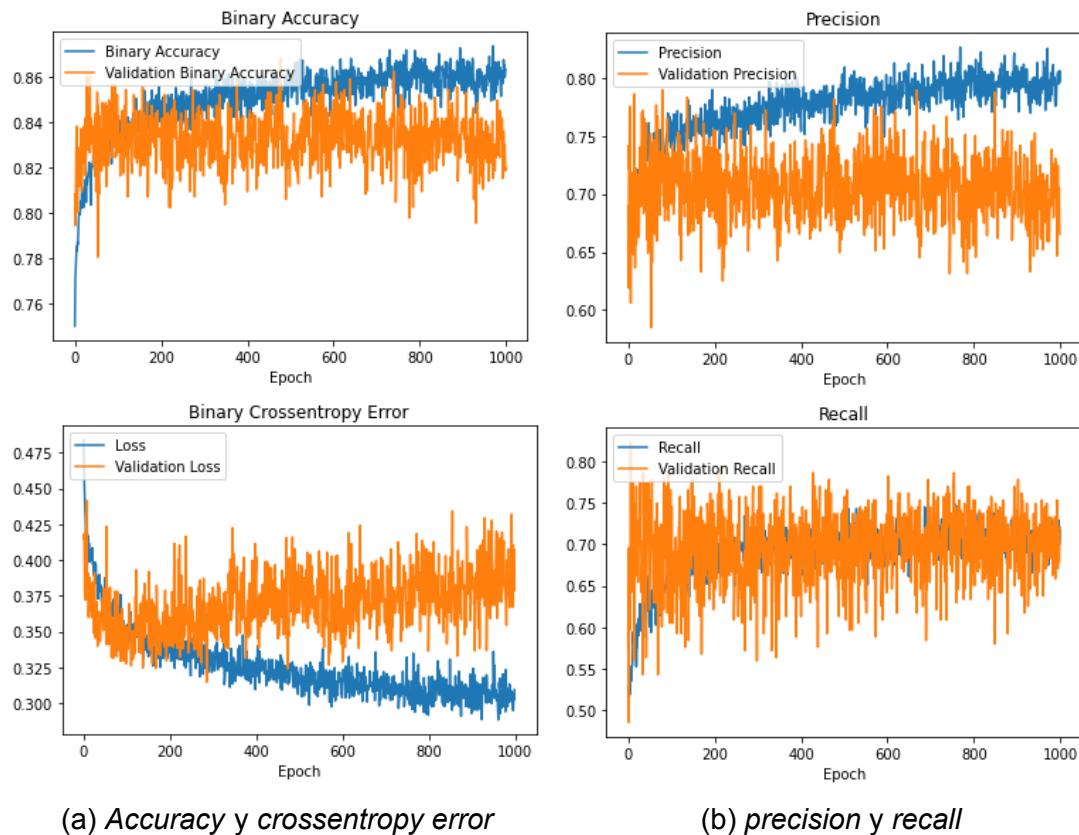


Figura 2.24: Gráficas al aplicar Dropout de 0.2 a la segunda capa oculta

```

136/136 [=====]
Train Binary Accuracy: 93.01
Train Precision: 88.95
Train Recall: 87.14
34/34 [=====] -
Test Binary Accuracy: 88.34
Test Precision: 80.94
Test Recall: 77.81

```

Figura 2.25: Resultados al aplicar Dropout de 0.2 a la segunda capa oculta

A la vista de los resultados, podemos apreciar que, en este caso, la **técnica de Dropout no nos está aportando mejoras**, sino que está empeorando los resultados, viendo además que en el segundo caso se sigue apreciando el **sobreajuste** con peores resultados. Por tanto, **descartamos** esta técnica para nuestros datos.

2.6. Experimentos utilizando regularización

A partir de ahora vamos a continuar con la **arquitectura de red de tres capas ocultas que probamos al principio de los experimentos (32, 64 y 128 neuronas)**, ya que nos aporta resultados muy **similares** a la encontrada por el Tuner con un **menor** número de neuronas y una precisión ligeramente mayor.

Viendo que el Dropout no nos está aportando una mejora a nuestros modelos vamos a intentar aplicar otra técnica para reducir el sobreajuste, la regularización. Keras nos ofrece tres tipos de regularización, L1, L2 y la combinación de ambas. La regularización consiste en modificar la función de coste utilizada para realizar la actualización de pesos, usando lo que en matemáticas se conoce como la norma L1 (para la regularización L1 Lasso) y la norma L2 (para la regularización L2 Ridge).

Ya que Keras nos ofrece la posibilidad de aplicar ambas regularizaciones, nuestro experimento va a consistir en aplicar esa **regularización L1 y L2 con constantes 0.0001 y 0.001 respectivamente** para ver si nos ayuda a obtener mejores resultados.

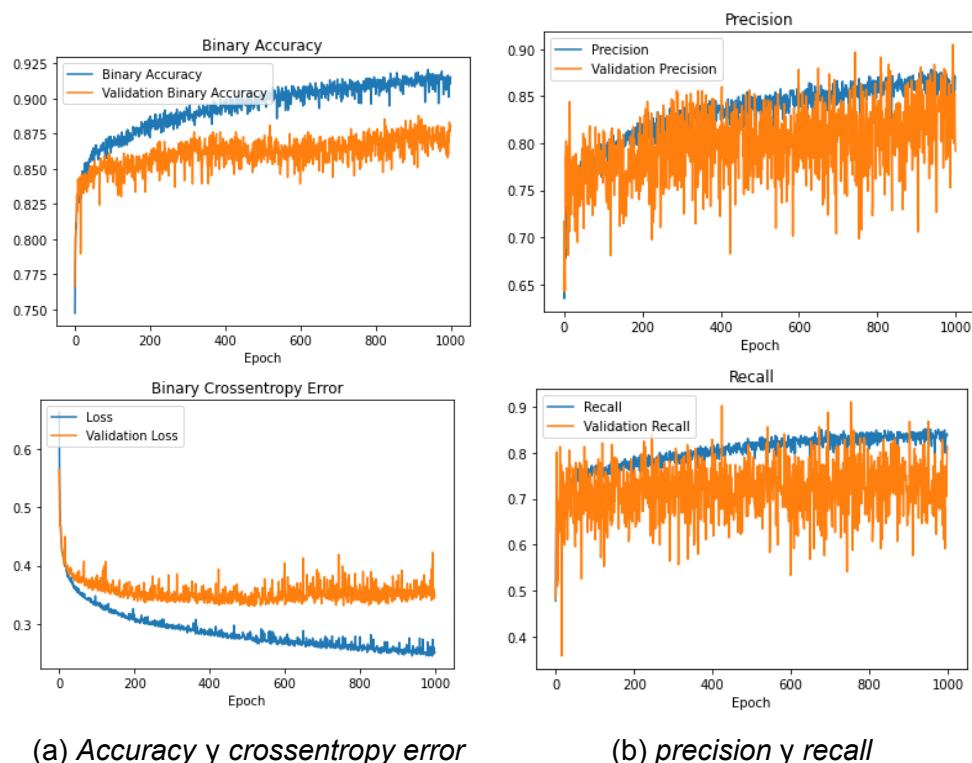


Figura 2.26: Gráficas al aplicar regularización L1 y L2

Como podemos ver en la imagen 2.27, tenemos unos resultados **ligeramente inferiores** a los obtenidos sin la regularización, **pero el sobreajuste se ha reducido bastante**, como podemos apreciar en las gráficas, por lo que consideramos que esta técnica **sí nos reporta una mejora en nuestros resultados**.

```
136/136 [=====]
Train Binary Accuracy: 91.19
Train Precision: 82.91
Train Recall: 88.42
34/34 [=====] -
Test Binary Accuracy: 88.90
Test Precision: 79.19
Test Recall: 82.79
```

Figura 2.27: Resultados al aplicar regularización L1 y L2

2.7. Experimento final: generación sintética de ejemplos de la clase minoritaria usando SMOTE

Como último experimento vamos a utilizar una **técnica de generación de datos sintéticos** para crear nuevos ejemplos de la clase **minoritaria** y así atacar el problema del **desbalanceo** de datos.

La técnica en cuestión es denominada **SMOTE (Synthetic Minority Oversampling Technique)** [2], y se basa en generar **nuevos ejemplos a partir de los ya existentes**, creando pequeñas perturbaciones. SMOTE se puede usar a través de la librería imblearn, instalable con el comando `!pip install -U imblearn`, y se importa ejecutando `from imblearn.over_sampling import SMOTE`.

Adicionalmente, vamos a utilizar los mismos parámetros de regularización de la sección 2.6, ya que se vió que reducía ligeramente el *overfitting* sin perjudicar en exceso las métricas objetivo.

En la imagen podemos ver como utilizar la librería `imblearn` para ejecutar la técnica SMOTE. Básicamente le decimos que genere tantos ejemplos de la clase minoritaria para que la proporción de ejemplos de la clase minoritaria entre los de la mayoritaria sea igual a 0.65 (por cada 100 ejemplos de la mayoritaria habrá 65 de la minoritaria).

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

oversample = SMOTE(sampling_strategy=0.65)
x_train, y_train = oversample.fit_resample(x_train, y_train)

nn = Sequential()

nn.add(Input(shape=(x_size,)))
nn.add(Dense(32, activation='relu', kernel_regularizer=regularizers.l1_l2(l1 = 1e-4, l2 = 1e-3)))
nn.add(Dense(64, activation='relu', kernel_regularizer=regularizers.l1_l2(l1 = 1e-4, l2 = 1e-3)))
nn.add(Dense(128, activation='relu', kernel_regularizer=regularizers.l1_l2(l1 = 1e-4, l2 = 1e-3)))
nn.add(Dense(1, activation='sigmoid'))

nn.compile(optimizer='adam', loss='binary_crossentropy', metrics=metrics)

history = nn.fit(x_train, y_train, epochs=700, verbose=0, validation_split=0.2)
```

Figura 2.28: Uso de la técnica SMOTE de la librería imblearn

En la figura 2.29 podemos ver el resultado del entrenamiento tras **700 épocas de una red neuronal de 3 capas ocultas, de 32, 64 y 128 neuronas** respectivamente, con **regularización L1** con constante 0.0001 y regularización **L2** con constante 0.001.

La gráfica del error muestra que **no se ha sobreentrenado tanto** como estábamos viendo en la sección 2.3. Además, las gráficas de *accuracy*, *precision* y *recall* indican que estas métricas han ido mejorando con una tendencia altista en general.

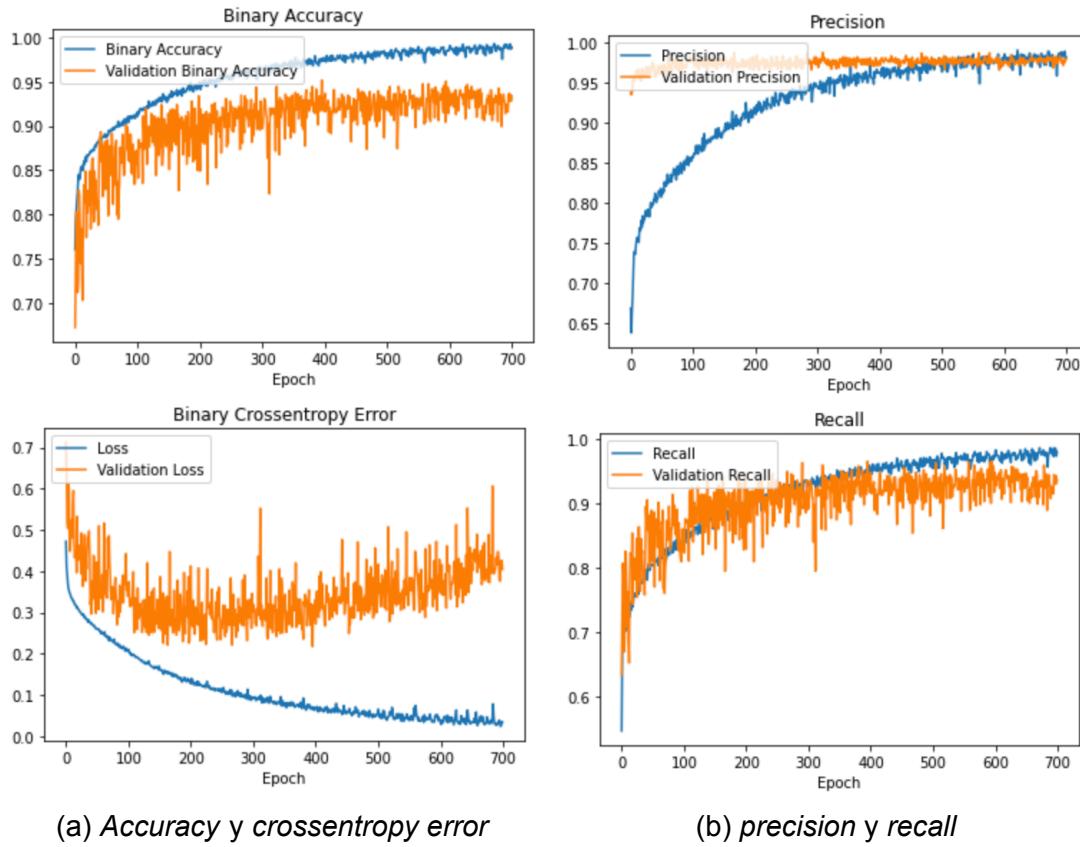


Figura 2.29: Gráficas al aplicar SMOTE y regularización tras 700 épocas de entrenamiento

En la imagen 2.30 vemos los resultados finales. Sorprendentemente hemos obtenido un **91.2 % de accuracy**, nuestra **mejor marca hasta ahora**. Además, **la precision y el recall también alcanzan sus valores más altos** en toda la práctica, con un **86 % en ambas métricas**. Parece que la **generación de nuevos ejemplos de la clase minoritaria ha ayudado satisfactoriamente a mejorar el rendimiento de la red neuronal propuesta**.

```
159/159 [=====]
Train Binary Accuracy: 98.32
Train Precision: 98.72
Train Recall: 96.99
34/34 [=====] -
Test Binary Accuracy: 91.21
Test Precision: 85.76
Test Recall: 85.50
```

Figura 2.30: Resultados tras el uso de la técnica SMOTE

3. Conclusiones

Como hemos podido ver en esta práctica, además de la **multitud de arquitecturas** de redes neuronales, existen numerosas **técnicas y métodos** que nos permiten **ajustar y pulir** los resultados de nuestros modelos. Por ello, es importante conocer bien estas técnicas para poder **discernir cuales aplicar** cuando nos encontramos con un nuevo conjunto de datos. Por ejemplo, si las clases están **desbalanceadas**, **SMOTE** es una muy buena opción para mejorar los resultados, o si detectamos un **gran sobreaprendizaje** podemos aplicar **regularización o Dropout**.

En resumen, la experimentación con **varios parámetros** y la aplicación de **diferentes técnicas** son clave para extraer **información y buenos resultados** de nuestros conjuntos de datos, **no hay una receta que funcione para todos**.

Bibliografía

- [1] Phoneme Dataset (2019). DataHub. <https://datahub.io/machine-learning/phoneme>
- [2] Jason Brownlee. SMOTE for Imbalanced Classification with Python (2020). Machine Learning Mastery. <https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/>
- [3] Jason Brownlee. Predictive Model for the Phoneme Imbalanced Classification Dataset (2021). Machine Learning Mastery. <https://machinelearningmastery.com/predictive-model-for-the-phoneme-imbalanced-classification-dataset/>