Redes neuronales profundas

Fátima Illana Guerra
Cristina Fernández Gómez

11 de abril de 2024

Índice

1. Introducción

El objetivo de este trabajo es llevar a cabo el entrenamiento de una red neuronal profunda. Para ello, le introduciremos un dataset de reservas de hotel, el cual contiene en un principio 36275 instancias y 19 variables. Iremos probando diferentes combinaciones de parámetros e hiperparámetros para tratar de hacer la red lo más precisa posible en base a los resultados que vayamos logrando con cada modelo, probando con distintos métodos de inicialización de pesos, regularización y estrategias de optimización para lograr la red neuronal más óptima posible.

El dataset contiene información sobre reservas hechas en un hotel, como pueden ser el número de noches que se alojarán, número de adultos, número de niños... Entre otros. A partir de estos datos, pretendemos hallar una red neuronal que sea capaz de predecir si un cliente cancelará, o no su reserva en base a su perfil: si es un cliente que repite, si ya canceló otras veces... además de los datos mencionados previamente.

Dividiremos los datos de forma que, usaremos el 80 % para el conjunto de entrenamiento, un 10 % para el de desarrollo, y el último 10 % para el de test. Dado que nos enfrentamos a un problema de clasificación binaria, si nuestro cliente cancelará o no la reserva, aplicaremos la función sigmoide en la capa de salida de nuestra red neuronal, permitiéndonos así interpretar la cancelación de una reserva.

2. Proceso de diseño

Comenzamos usando la arquitectura del cuaderno que implementa la red neuronal profunda con Keras para estimar el valor medio de una casa. En esta primera iteración, establecemos los siguientes parámetros:

Cuadro 1: Configuración del modelo incial

Configuración
Épocas: 1000
Tasa de Aprendizaje: 0.1
Capas Ocultas: [500, 250, 75, 25]
Mini-Batch: 512
Función de Activación: ReLU
Optimizador: SGD(LR)
Iniciador: Nada
Regularizador: Nada

Con lo que obtenemos los siguientes resultados:

Cuadro 2: Resultados del modelo inicial

	Train Error	**Test Error**	**Bias**	**Varianza**	**Tiempo**
Modelo 1	2.4 %	12.5 %	-7.6 %	10.1 %	323s

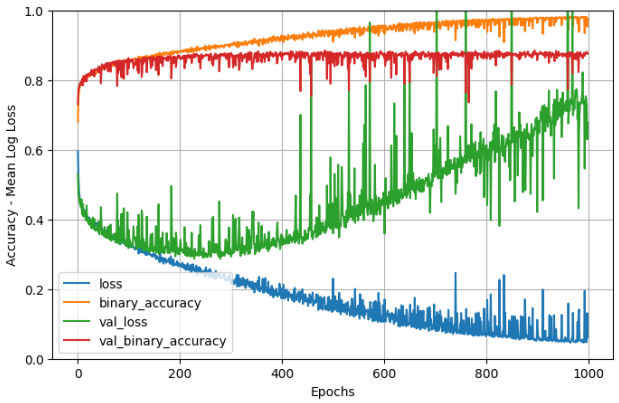


Figura 1: Enter Caption

Podemos observar que el modelo está claramente sobreajustado, porque el error de entrenamiento es muy bajo comparado con el de test, haciendo que la varianza sea elevada. Además, vemos que hay una diferencia considerable entre el error de entrenamiento y el error de Bayes, provocando que el bias sea negativo. El primer cambio que vamos a llevar a cabo es la función de activación de ReLU a ELU, ya que es continua y derivable. Además, tiene un rango de salida tanto para valores positivos como negativos, solucionando el problema de las "neuronas muertas.[a]l que tiene que hacer frente ReLU. Todas estas ventajas deberían mejorar el proceso de entrenamiento.

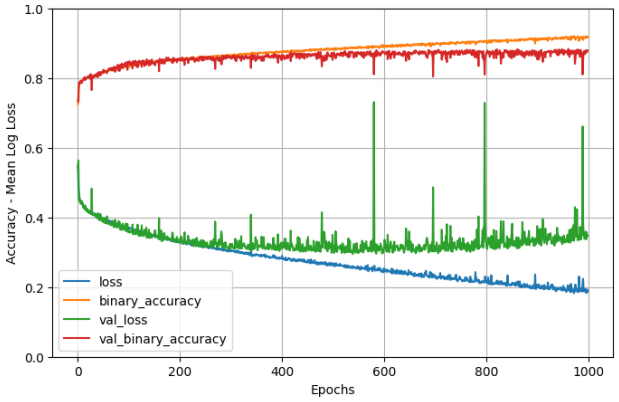
Cuadro 3: Configuración del modelo incial ELU

Configuración
Épocas: 1000
Tasa de Aprendizaje: 0.1
Capas Ocultas: [500, 250, 75, 25]
Mini-Batch: 512
Función de Activación: eLU
Optimizador: SGD(LR)
Iniciador: Nada
Regularizador: Nada

Con lo que obtenemos los siguientes resultados:

Cuadro 4: Resultados del modelo inicial ELU

	Train Error	**Test Error**	**Bias**	**Varianza**	**Tiempo**
Modelo 1	8.1 %	12.1 %	-1.9 %	4.0 %	323s



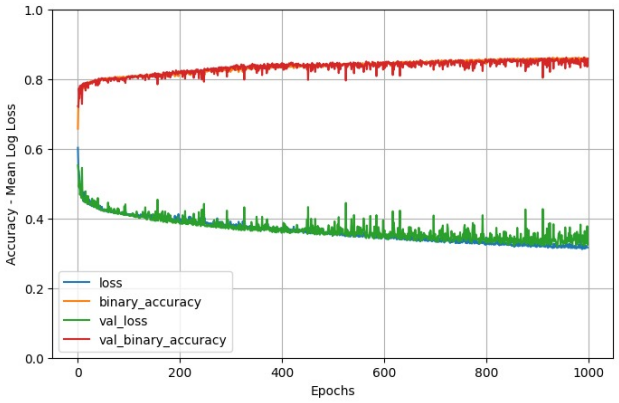
Vemos que los resultados obtenidos son mejores que con el algoritmo ReLU, los errores se han equilibrado y hemos conseguido reducir el bias y la varianza. Sin embargo, tenemos un tiempo de ejecución bastante elevado, probamos aumentando el tamaño del batch con varios valores, con lo que también obtendremos una mayor precisión en el gradiente al estar basado en más datos.

Cuadro 5: Resultados del modelo 1

	Train Error	**Test Error**	**Bias**	**Varianza**	**Tiempo**
Batch 1000	11.4 %	20.1 %	1.4 %	8.7 %	140s
Batch 2048	14.1 %	14.2 %	4.1 %	0.1 %	143s
Batch 4096	2.7 %	12.9 %	-7.3 %	10.2 %	563s

Cuadro 6: Configuración del modelo 1

Configuración
Épocas: 1000
Tasa de Aprendizaje: 0.1
Capas Ocultas: [500, 250, 75, 25]
Mini-Batch: 2048
Función de Activación: eLU
Optimizador: SGD(LR)
Iniciador: Nada
Regularizador: Nada



Podemos observar que nos quedan mejores resultados con un tamaño de batch de 2048, por lo que es el que dejamos establecido. Hemos conseguido reducir el tiempo de ejecución, sin embargo, podemos observar que tenemos un bias bastante elevado. Una de las opciones para poder reducirlo es aplicar un algoritmo de optimización. Comenzaremos usando el algoritmo `ADAM`, que suele converger más rápido que los demás y su tasa de aprendizaje requiere menos adaptación.

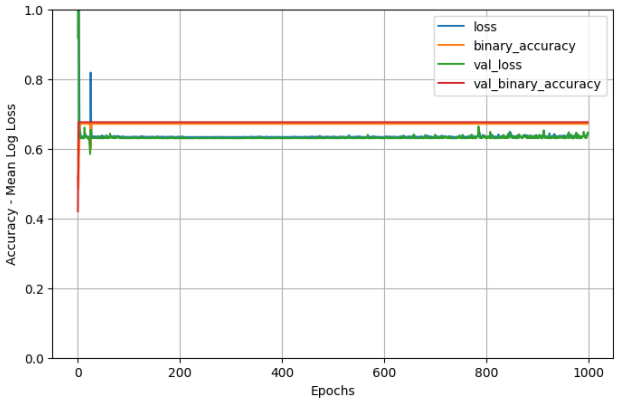
Cuadro 7: Configuración del modelo 2

Configuración
Épocas: 1000
Tasa de Aprendizaje: 0.1
Capas Ocultas: [500, 250, 75, 25]
Mini-Batch: 2048
Función de Activación: eLU
Optimizador: ADAM
Iniciador: Nada
Regularizador: Nada

Obteniendo los siguientes resultados:

Cuadro 8: Resultados del modelo 2

	Train Error	**Test Error**	**Bias**	**Varianza**	**Tiempo**
Modelo 2	32.9 %	32.4 %	22.9 %	0.5 %	114s



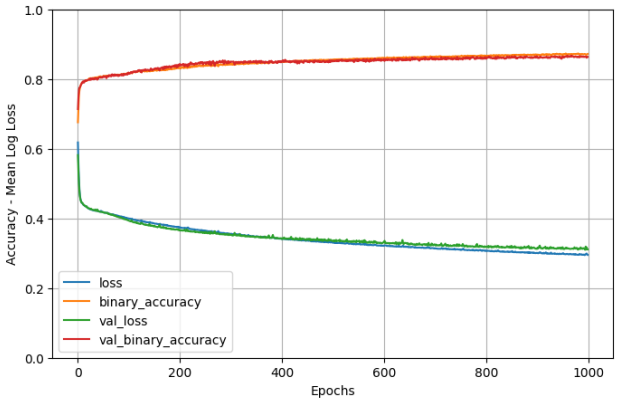
Ambos errores han aumentado considerablemente, al igual que el bias, por lo que cambiamos la tasa de aprendizaje para tratar de conseguir mayor precisión y estabilidad en el modelo.

Cuadro 9: Resultados del modelo 3

	Train Error	**Test Error**	**Bias**	**Varianza**	**Tiempo**
Tasa 0.01	8.5 %	13.6 %	-1.5 %	5.1 %	323s
Tasa 0.001	1.4 %	14.4 %	-8.6 %	13.0 %	292s
Tasa 0.0001	12.8 %	13.6 %	2.8 %	0.8 %	143s

Cuadro 10: Configuración del modelo 3

Configuración
Épocas: 1000
Tasa de Aprendizaje: 0.0001
Capas Ocultas: [500, 250, 75, 25]
Mini-Batch: 2048
Función de Activación: eLU
Optimizador: ADAM
Iniciador: Nada
Regularizador: Nada



Tanto los errores como el bias han descendido de manera considerable con la tasa de 0.0001, pero trataremos de disminuir aún más el bias para reducir el error de entrenamiento. Por ello, utilizaremos un inicializador de pesos para que distribuya los pesos de manera más efectiva, evitando que sean demasiado elevados y pueda provocar que el gradiente tienda a 0, y por tanto los pesos no se actualicen de manera adecuada (Vanishing Gradient Problem)

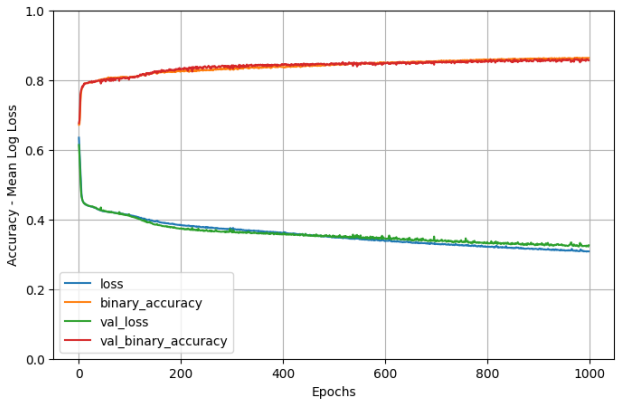
Aplicaremos distintos rangos de la uniforme, además de la He normal y uniforme y la normal básica.

Cuadro 11: Resultados del modelo 4

	Train Error	**Test Error**	**Bias**	**Varianza**	**Tiempo**
Uniforme [-0.1, 0.1]	13.7 %	14.3 %	3.7 %	0.6 %	149s
Uniforme [-0.3,0.3]	1.3 %	13.1 %	-8.7 %	11.8 %	364s
RandomNormal	2.0 %	13.2 %	-8.0 %	11.2 %	384s
He Normal	1.6 %	13.9 %	-8.4 %	12.3 %	361s
He Uniforme	1.3 %	14.1 %	-8.7 %	12.8 %	383s

Cuadro 12: Configuración del modelo 4

Configuración
Épocas: 1000
Tasa de Aprendizaje: 0.0001
Capas Ocultas: [500, 250, 75, 25]
Mini-Batch: 2048
Función de Activación: eLU
Optimizador: ADAM
Iniciador: Uniforme [-0.1, 0.1]
Regularizador: Nada



Observando los resultados, podemos ver que la que nos da mejor rendimiento es la Uniforme en el intervalo [-0.1,0.1] Sin embargo, apenas hubo diferencia en la disminución del sesgo, por lo que probamos con otra de las alternativas para disminuirlo: añadir una capa oculta. En nuestro caso añadimos una con 500 neuronas. De esta manera, el modelo podrá hallar patrones más complejos y posiblemente, reducir el sesgo:

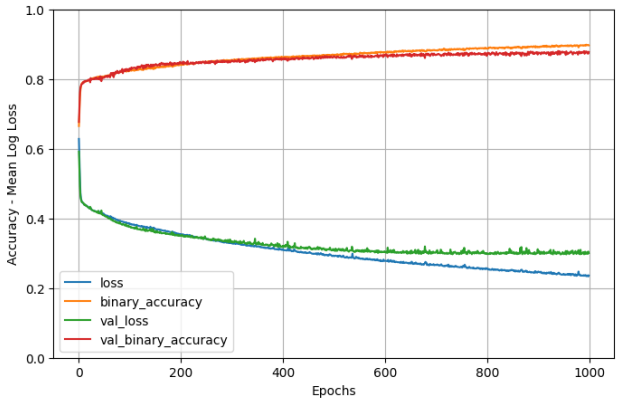
Cuadro 13: Configuración del modelo 5

Configuración
Épocas: 1000
Tasa de Aprendizaje: 0.0001
Capas Ocultas: [500, 500, 250, 75, 25]
Mini-Batch: 2048
Función de Activación: eLU
Optimizador: ADAM
Iniciador: Uniforme [-0.1, 0.1]
Regularizador: Nada

Obteniendo los siguientes resultados:

Cuadro 14: Resultados del modelo 5

	Train Error	**Test Error**	**Bias**	**Varianza**	**Tiempo**
Modelo 5	10.3 %	12.2 %	0.3 %	1.9 %	152s



Dado que en la primera prueba hemos obtenido buenos resultados, no probamos a cambiar el número de neuronas de la capa. Hemos conseguido reducir considerablemnte el sesgo, además de los errores de test y entrenamiento. Sin embargo, ahora la varianza está algo más elevada, lo que produce que la red no sea capaz de adaptarse a datos que no ha visto en la fase de entrenamiento y se sobreajuste. Para tratar de solucionar este problema, aplicamos regularización, que se encarga especificamente de resolver el sobreajuste. Empezamos usando regularizador droput con tasa 0.1, con lo que desactiva un porcentaje de neuronas (un 10 % en nuestro caso) por capa en cada iteración del entrenamiento, evitando que la clasificación recaiga en unas pocas neuronas, promoviendo así una mejor generalización.

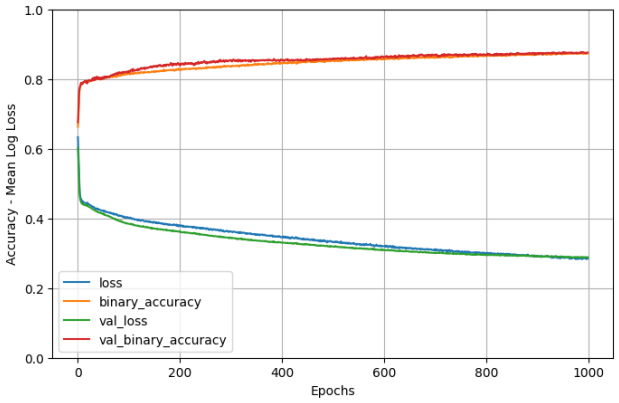
Cuadro 15: Configuración del modelo 6

Configuración
Épocas: 1000
Tasa de Aprendizaje: 0.0001
Capas Ocultas: [500, 500, 250, 75, 25]
Mini-Batch: 2048
Función de Activación: eLU
Optimizador: ADAM
Iniciador: Uniforme [-0.1, 0.1]
Regularizador: Dropout(0.1)

Obtenemos los siguientes resultados:

Cuadro 16: Resultados del modelo 6

	Train Error	**Test Error**	**Bias**	**Varianza**	**Tiempo**
Modelo 6	12.7 %	12.4 %	2.7 %	-0.3 %	134s



Como hemos conseguido reducir la varianza, mantenemos la configuración con una tasa de dropout del 0.1. Sin embargo, como el bias aumentó, tratamos de disminuirlo añadiendo más neuronas a la primera capa.

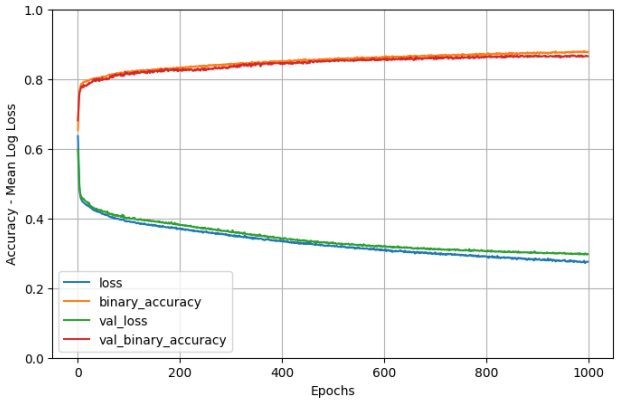
Cuadro 17: Configuración del modelo 7

Configuración
Épocas: 1000
Tasa de Aprendizaje: 0.0001
Capas Ocultas: [700, 500, 250, 75, 25]
Mini-Batch: 2048
Función de Activación: eLU
Optimizador: ADAM
Iniciador: Uniforme [-0.1, 0.1]
Regularizador: Dropout(0.1)

Obteniendo los siguientes resultados:

Cuadro 18: Resultados del modelo 7

	Train Error	**Test Error**	**Bias**	**Varianza**	**Tiempo**
Modelo 7	12.4 %	12.4 %	2.4 %	0 %	143s



El bias se redujo pero muy ligeramente, tratamos de reducirlo algo más volviendo a aumentar todavía más el número de neuronas de la primera capa y añadiendo una nueva capa oculta.

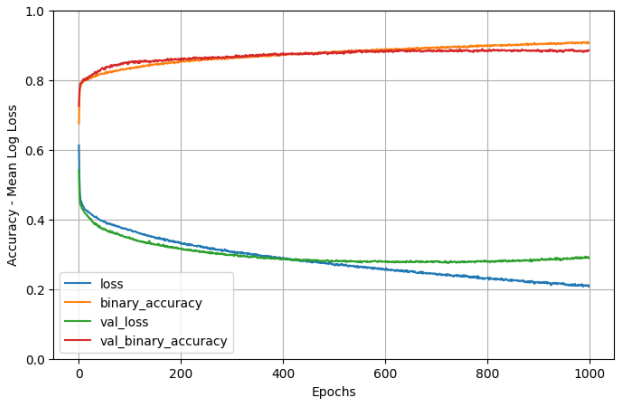
Cuadro 19: Configuración del modelo 8

Configuración
Épocas: 1000
Tasa de Aprendizaje: 0.0001
Capas Ocultas: [1000, 750, 500, 250, 75, 25]
Mini-Batch: 2048
Función de Activación: eLU
Optimizador: ADAM
Iniciador: Uniforme [-0.1, 0.1]
Regularizador: Dropout(0.1)

Obteniendo así:

Cuadro 20: Resultados del modelo 8

	Train Error	**Test Error**	**Bias**	**Varianza**	**Tiempo**
Modelo 8	9.3 %	11.5 %	-0.7 %	2.2 %	175s



Aunque hemos reducido el bias, consiguiendo también reducir el error de test y de entrenamiento, la varianza aumentó ligeramente, por lo que cambiamos la tasa del droput a 0.2 para incrementar el efecto de la regularización.

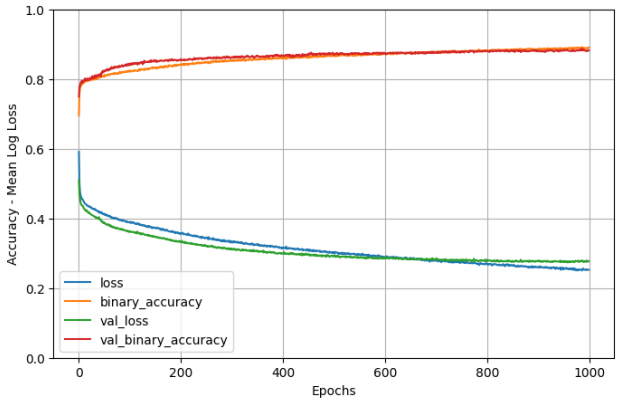
Cuadro 21: Configuración del modelo 9

Configuración
Épocas: 1000
Tasa de Aprendizaje: 0.0001
Capas Ocultas: [1000, 750, 500, 250, 75, 25]
Mini-Batch: 2048
Función de Activación: eLU
Optimizador: ADAM
Iniciador: Uniforme [-0.1, 0.1]
Regularizador: Dropout(0.2)

Obteniendo los siguientes resultados:

Cuadro 22: Resultados del modelo 9

	Train Error	**Test Error**	**Bias**	**Varianza**	**Tiempo**
Modelo 9	11.0 %	11.8 %	1.0 %	0.8 %	157s



Llegados a estos resultados, podemos observar que los errores de entrenamiento y test son muy bajos y están balanceados, además de que el bias y la varianza también tienen valores pequeños, por lo que nos detacantamos por este modelo como nuestro modelo final.

3. Resultados

Nuestro modelo final está compuesto por los siguientes hiperparámetros:

Cuadro 23: Configuración del modelo 9

Configuración
Épocas: 1000
Tasa de Aprendizaje: 0.0001
Capas Ocultas: [1000, 750, 500, 250, 75, 25]
Mini-Batch: 2048
Función de Activación: eLU
Optimizador: ADAM
Iniciador: Uniforme [-0.1, 0.1]
Regularizador: Dropout(0.2)

Como podemos ver, el modelo final es bastante profundo, al haber 6 capas ocultas con un número considerable de neuronas. Además, el proceso de aprendizaje de la red también es complejo, al repetirse durante 1000 épocas con una tasa de aprendizaje bastante cercana a 0. No obstante, para compensar la complejidad de la estructura, establecemos un tamaño de Mini-Batch alto y un regularizador con una tasa también considerable (0.2). El empleo de un iniciador y de un algoritmo de optimización también mejora su aprendizaje.

De forma que, aplicando dicho modelo neuronal al conjunto de test final, obtenemos una precisión del 87.2 %, al haber predicho correctamente 3178 instancias y, por ende, habiendo clasificado erróneamente 449 de las 3627 instancias que conforman el conjunto de test.

Esto lo podemos observar en la matriz de confusión resultante:

Cuadro 24: Matriz de Confusión

	Cancelada	**No Cancelada**	**Suma**
Predicción Cancelada	928	201	1129
Predicción No Cancelada	248	2250	2498

Como podemos observar, 928 reservas canceladas de las 1176 que había en el conjunto de test fueron correctamente clasificadas, confundiendo las otras 248 con reservas no canceladas. Por otra parte, 2250 reservas no canceladas de las 2451 que teníamos fueron correctamente clasificadas, confundiendo las otras 201 con reservas canceladas.

Nos damos cuenta de que los resultados son ligeramente dispares, ya que en proporción parece que clasifica mejor las reservas no canceladas. Vamos a estudiar las métricas de clasificación para obtener más conclusiones:

Cuadro 25: Métricas de Clasificación

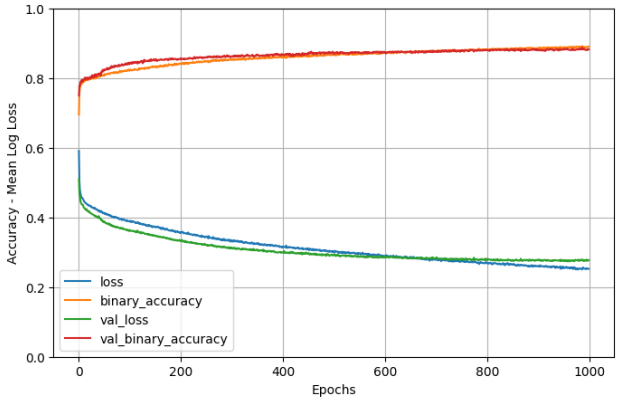
	Precisión	**Recall**	**F1-Score**	**Support**
Cancelada	0.82	0.79	0.81	1176
No Cancelada	0.90	0.92	0.91	2451
Accuracy			0.88	3627
Macro Avg	0.86	0.85	0.86	3627
Wighted Avg	0.88	0.88	0.88	3627

A partir de la métricas nos damos cuenta de que el rendimeinto es mucho mejor a la hora de clasificar reservas no canceladas. Por ejemplo, vemos que el 82 % de las reservas clasificadas como canceladas realmente son canceladas mientras que dicha precisión, en el caso de reservas no canceladas se eleva a un 90 %. Por otro lado, solo un 79 % de las reservas canceladas se han clasificado bien, frente al 92 % de reservas no canceladas clasificadas correctamente. Además, el F1-Score en el segundo caso es un 10 % mayor, lo cual significa que hay un mejor balance entre precisción y recall.

Estos resultados pueden deberse a la desproporción en instancias clasificadas como canceladas (1176) frente al número de instancias clasificadas como no canceladas (2451) que hay en el conjunto de test y, probablemente, en el conjunto de datos completo. Es decir, que hay un desbalance de clases que afecta ligeramente al aprendizaje del modelo.

Por ello, en cuanto a las métricas de rendimiento generales del modelo, nos vamos a fijar en el "Weighted Average"que proporciona una media ponderada de todas las métricas, ajustándose mejor al desbalance de los datos y, en nuestro caso, reflejando un poco mejor el rendimiento de nuestro modelo puesto que "Weighted Average"proporciona una media equilibrada entre clases. Por lo que vemos en general se ha obtenido un rendimiento del 88 %.

Además, volvemos a mostrar cómo la precisión evoluciona durante el proceso de entrenamiento, con el modelo final:



Vemos que la precisión de validación al inicio es algo mayor que la de entrenamiento, pero que rápidamente convergen a un valor muy similar. Esto se puede deber a que, al tener más datos, el conjunto de entrenamiento inicialmente tenga un rendimiento menor.

4. Conclusiones

Con este trabajo nos hemos dado cuenta de lo cuidadosos que hay que ser a la hora de crear una red neuronal. El mínimo cambio en cualquier hiperparámetro puede provocar que los resultados se disparen. En nuestro caso, nos sorprendió mucho cómo aumentaron ambos errores, tanto el de entrenamiento como el de test, cuando utilizamos el algoritmo de optimización ADAM, haciendo que subieran a un 32.9 % y 32.4 %, respectivamente. Sin embargo, conseguimos solucionarlo probando diferentes tasas de entrenamiento.

Otra dificultad que nos supuso el trabajo, es que en varias ocasiones nos sucedió que, cuando solucionábamos un problema, como el de reducir el bias, nos aparecía otro, como el aumento de la varianza. Es por ello que fue algo complejo encontrar un modelo que estabilizara ambos valores, porque es de vital importancia que los dos sean bajos para conseguir un modelo que generalice bien y que sea capaz de adaptarse a los datos de entrenamiento pero sin sobreajustarse.

Lograr un equilibrio de todos los hiperparámetros no es una tarea sencilla, es imprescindible comprender para qué sirve exactamente cada uno y cómo puede afectar a nuestra red el más mínimo cambio, este trabajo nos ayudó mucho para comprender mejor el nivel de complejidad que tiene construir una red neuronal.