Trabajo Fin de Master

Master Universitario en Ingeniería Industrial

Redes neuronales de aprendizaje profundo aplicadas al análisis de resultados de trasplantes de riñón

Autor: César Rodríguez Ben

Tutor: Jorge Calvillo Arbizu

Co-tutor: Alejandro Talaminos Barroso

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Proyecto Fin de Master
Master Universitario en Ingeniería Industrial

Redes neuronales de aprendizaje profundo aplicadas al análisis de resultados de trasplantes de riñón

Autor:
César Rodríguez Ben

Tutor:
Jorge Calvillo Arbizu
Profesor Ayudante Doctor

Co-tutor:
Alejandro Talaminos Barroso

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Proyecto Fin de Master: Redes neuronales de aprendizaje profundo aplicadas al análisis de resultados de trasplantes de riñón

Autor: César Rodríguez Ben

Tutor: Jorge Calvillo Arbizu

Co-tutor Alejandro Talaminos Barroso

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

Agradezco a mi familia y amigos por apoyarme y ayudarme en los malos momentos y por hacerme reír y ser feliz en los buenos. Porque han sido tiempos tempestuosos y nunca se valora tanto a la gente que te acompaña como cuando la necesitas de verdad.

Agradezco también a mis tutores por la libertad, confianza y paciencia con la que siempre me guiaron, la cual ha sido clave para disfrutar de la realización de este proyecto.

Y agradezco profundamente a la plataforma Coursera por hacer la educación más abierta y accesible, y en especial a Andrew Ng, co-fundador y educador en la plataforma. Sin su dedicación y capacidad en la enseñanza hubiera sido mucho más difícil aprender lo necesario para desarrollar este proyecto.

César Rodríguez Ben

Alumno de la Escuela Técnica Superior de Ingeniería de Sevilla

Sevilla, 2021

Resumen

El trasplante de riñon es el procedimiento de trasplante más demandado en España y la supervivencia del órgano trasplantado depende de muchos factores. En este proyecto se analizarán las variables registradas de los procedimientos de trasplante de riñón que se han llevado a cabo en Andalucía en las últimas dos décadas.

Para ello, se crearán redes neuronales artificiales que utilizarán estructuras de aprendizaje profundo para encontrar patrones en estos datos. El objetivo es mejorar las predicciones que se realizan actualmente con métodos estadísticos tradicionales y así poder proveer al personal sanitario de la mayor información posible para la toma de decisiones en estos procesos.

En el desarrollo del proyecto, se construirán diferentes modelos de redes neuronales multicapa que se entrenarán mediante aprendizaje profundo con los datos cedidos por la Coordinación Autonómica de Trasplantes de Andalucía. Posteriormente se analizarán los resultados obtenidos, el éxito o fracaso de los modelos y se plantearán nuevas rutas para mejoras futuras.

Abstract

Kidney transplantation is the most demanded transplantation procedure in Spain and the survival of the transplanted organ depends on many factors. This project will analyse the variables which have been recorded in kidney transplantation procedures carried out in Andalucía in the last two decades.

For this purpose, artificial neural networks will be created using deep learning structures to find patterns in these data. The aim is to improve the predictions that are currently made using traditional statistical methods and providing healthcare personnel with as much information as possible for the decision making.

During the development of the project, different multilayer neural network models will be built and trained with deep learning methods with the data provided by the Coordinación Autonómica de Trasplantes de Andalucía. Subsequently, the results obtained and the success or failure of the models will be analysed and new routes for future improvements will be proposed.

Índice

Agradecimientos **ix**
Resumen **xi**
Abstract **xiii**
Índice **xv**
Índice de Tablas **xvii**
Índice de Figuras **xxi**
Notación **xxv**
1 Introducción y motivación **1**
2 Materiales y Métodos **5**
2.1 Objetivo *5*
2.2 Metodología *5*
2.3 Hardware *6*
2.4 Software *6*
2.4.1 Lenguaje de programación 6
2.4.2 Entorno de programación 7
2.5 Modelo de red neuronal *7*
2.5.1 Elementos y funcionamiento 8
2.5.2 Inicialización de pesos 11
2.5.3 Propagación hacia delante o Forward propagation 14
2.5.4 Función de coste 16
2.5.5 Propagación hacia atrás o Backward propagation 18
2.5.6 Actualización de los pesos 20
2.6 Normalización *21*
2.7 Regularización *23*
2.7.1 L2 regularization 24
2.7.2 Dropout regularization 25
2.7.3 Early stopping 27
2.8 Optimización *27*
2.8.1 Gradient descent with momentum 28
2.8.2 RMSprop – Root mean square 30
2.8.3 Adam 30

3 Resultados **33**
3.1 Estableciendo el objetivo *33*
3.2 Tratamiento de los datos *34*
3.2.1 Contenido 34
3.2.2 Transformación a DataFrame 37
3.2.3 Missing values 39
3.2.4 Transformación en matrices 51
3.3 Primer modelo *52*
3.4 Segundo modelo *58*
3.5 Tercer modelo *60*
3.6 Elementos de medición de rendimiento *65*
3.7 Cuarto modelo *70*
3.8 Quinto modelo *78*
3.9 Sexto modelo *89*
3.10 Séptimo modelo *95*
3.11 Octavo modelo *103*
3.12 Noveno modelo *106*
4 Conclusiones y líneas futuras **111**
4.1 Análisis de los resultados y conclusiones *111*
4.2 Próximos pasos y líneas futuras *113*
Anexos **115**
Anexo A - Modelo 1 **117**
Anexo B - Modelo 3 **121**
Anexo C - Modelo 4 **125**
Anexo D - Modelo 5 **129**
Anexo E - Modelo 6 **133**
Anexo F - Modelo 7 **137**
Anexo G - Modelo 8 **143**
Anexo H - Modelo 9 **147**
Anexo I - Elementos de medición **153**
Anexo J - Tratamiento *missing values* **155**
Referencias **161**

Índice de Tablas

Tabla 2-1. Características técnicas del ordenador 6
Tabla 3-1. Estatus del paciente 34
Tabla 3-2. Campos relativos al donante 35
Tabla 3-3. Campos relativos al receptor 35
Tabla 3-4. Campos relativos al trasplante 36
Tabla 3-5. Variables críticas seleccionadas 37
Tabla 3-6. Modificación de valores 38
Tabla 3-7. Campos con missing values y sus porcentajes 41
Tabla 3-8. Mediana de *Peso_don* según *Sexo_don* 42
Tabla 3-9 Mediana de *Talla_don* según *Sexo_don* 44
Tabla 3-10. Mediana de *Peso_pre_tx* según *Sexo_rx* 45
Tabla 3-11. Mediana de *Peso_pre_tx* según *Sexo_rx* 46
Tabla 3-12. Mediana de *Creatinina* según *Sexo_rx* y *Tipo_don* 48
Tabla 3-13. Propiedades del conjunto de datos de entrada *X_model* 52
Tabla 3-14. Propiedades del conjunto de datos de salida *Y_final* 52
Tabla 3-15. Coste modelo 1 $\alpha = 0.001$ 56
Tabla 3-16. Coste modelo 1 $\alpha = 0.01$ 57
Tabla 3-17. Coste modelo 1 $\alpha = 0.05$ 57
Tabla 3-18. Coste modelo 1 $\alpha = 0.1$ 58
Tabla 3-19. Coste modelo 2 con estructura [17, 28, 8, 6] 58
Tabla 3-20. Coste modelo 2 con estructura [17, 25, 14, 6] 59
Tabla 3-21. Coste modelo 2 con estructura [17, 20, 20, 6] 59
Tabla 3-22. Coste modelo 3 con estructura [17, 20, 20, 6, 6] 63
Tabla 3-23. Coste modelo 3 con estructura [17, 20, 20, 10, 6] 63

Tabla 3-24. Coste modelo 3 con estructura [17, 25, 25, 16, 6] 64
Tabla 3-25. Resultado modelo 3 con estructura [17, 20, 20, 6, 6] 65
Tabla 3-26. Resultado modelo 3 con estructura [17, 20, 20, 10, 6] 66
Tabla 3-27. Resultado modelo 3 con estructura [17, 25, 25, 16, 6] 66
Tabla 3-28. Coste modelo 3 con estructura [17, 25, 25, 16, 6] sobre C.D. 1 67
Tabla 3-29. Resultado modelo 3 con estructura [17, 25, 25, 16, 6] sobre C.D. 1 67
Tabla 3-30. Coste modelo 3 con estructura [17, 25, 25, 16, 6] sobre C.D. 2 67
Tabla 3-31. Resultado modelo 3 con estructura [17, 25, 25, 16, 6] sobre C.D. 2 68
Tabla 3-32. Coste modelo 3 con estructura [17, 25, 25, 16, 6] sobre C.D. 3 68
Tabla 3-33. Resultado modelo 3 con estructura [17, 25, 25, 16, 6] sobre C.D. 3 69
Tabla 3-34. Coste modelo 3 con estructura [17, 25, 25, 16, 6] y E.S. sobre C.D. 4 69
Tabla 3-35. Resultado modelo 3 con estructura [17, 25, 25, 16, 6] y E.S. sobre C.D. 4 69
Tabla 3-36. Coste modelo 3 con estructura [17, 25, 25, 16, 6] y E.S. sobre C.D. 5 70
Tabla 3-37. Resultado modelo 3 con estructura [17, 25, 25, 16, 6] y E.S. sobre C.D. 5 70
Tabla 3-38. Coste modelo 4 con *lambda* = 0.5 72
Tabla 3-39. Resultado modelo 4 con *lambda* = 0.5 73
Tabla 3-40. Coste modelo 4 con *lambda* = 0.7 73
Tabla 3-41. Resultado modelo 4 con *lambda* = 0.7 74
Tabla 3-42. Coste modelo 4 con *lambda* = 0.7 y 25.000 iteraciones 74
Tabla 3-43. Resultado modelo 4 con *lambda* = 0.7 y 25.000 iteraciones 74
Tabla 3-44. Modelo 4 con *lambda* = 0.5 y 20.000 iteraciones 75
Tabla 3-45. Modelo 4 con *lambda* = 0.5 y 25.000 iteraciones 75
Tabla 3-46. Modelo 4 con *lambda* = 0.7 y 20.000 iteraciones 75
Tabla 3-47. Modelo 4 con *lambda* = 0.7 y 25.000 iteraciones 75
Tabla 3-48. Modelo 4 con *lambda* = 0.7 y 15.000 iteraciones 76
Tabla 3-49. Modelo 4 con *lambda* = 0.7 y 20.000 iteraciones 76
Tabla 3-50. Modelo 4 con *lambda* = 0.9 y 15.000 iteraciones 76
Tabla 3-51. Modelo 4 con *lambda* = 0.9 y 20.000 iteraciones 76
Tabla 3-52. Modelo 4 con *lambda* = 0.9 y 25.000 iteraciones 76
Tabla 3-53. Modelo 4 con *lambda* = 1.2 y 20.000 iteraciones 77
Tabla 3-54. Modelo 4 con *lambda* = 1.2 y 25.000 iteraciones 77
Tabla 3-55. Coste modelo 5 con *keep_prob* = 0.5 81
Tabla 3-56. Resultado modelo 5 con *keep_prob* = 0.5 81
Tabla 3-57. Modelo 5 con *keep_prob* = 0.5 y 20.000 iteraciones 82
Tabla 3-58. Modelo 5 con *keep_prob* = 0.5 y 25.000 iteraciones 82
Tabla 3-59. Modelo 4 con *keep_prob* = 0.5 y 30.000 iteraciones 82
Tabla 3-60. Modelo 5 con *keep_prob* = 0.65 y 20.000 iteraciones 82
Tabla 3-61. Modelo 5 con *keep_prob* = 0.65 y 25.000 iteraciones 82
Tabla 3-62. Modelo 4 con *keep_prob* = 0.65 y 30.000 iteraciones 82

Tabla 3-63. Modelo 5 con *keep_prob* = 0.8 y 20.000 iteraciones 83
Tabla 3-64. Modelo 5 con *keep_prob* = 0.8 y 25.000 iteraciones 83
Tabla 3-65. Modelo 4 con *keep_prob* = 0.8 y 30.000 iteraciones 83
Tabla 3-66. Modelo 5 con *keep_prob* = 0.9 y 20.000 iteraciones 84
Tabla 3-67. Modelo 5 con *keep_prob* = 0.9 y 25.000 iteraciones 84
Tabla 3-68. Modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones 84
Tabla 3-69. Modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones sobre C.D. 1 85
Tabla 3-70. Modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones sobre C.D. 2 85
Tabla 3-71. Modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones sobre C.D. 3 85
Tabla 3-72. Media de los resultados modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones 86
Tabla 3-73. Modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 1 86
Tabla 3-74. Modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 2 86
Tabla 3-75. Modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 3 86
Tabla 3-76. Modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 4 86
Tabla 3-77. Media de los resultados modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones 87
Tabla 3-78. Coste modelo 5 con *keep_prob* = 0.8 y 100.000 iteraciones 87
Tabla 3-79. Resultado modelo 5 con *keep_prob* = 0.8 y 100.000 iteraciones 88
Tabla 3-80. Coste modelo 5 con *keep_prob* = 0.85 y 100.000 iteraciones 88
Tabla 3-81. Resultado modelo 5 con *keep_prob* = 0.85 y 100.000 iteraciones 89
Tabla 3-82. Coste modelo 6 con *keep_prob* = 0.87, β = 0.9 y 100.000 iteraciones 91
Tabla 3-83. Resultado modelo 6 con *keep_prob* = 0.87, β = 0.9 y 100.000 iteraciones 92
Tabla 3-84. Modelo 6 con *keep_prob* = 0.87, β = 0.9 y 80.000 iteraciones 92
Tabla 3-85. Modelo 6 con *keep_prob* = 0.87, β = 0.85 y 80.000 iteraciones 92
Tabla 3-86. Modelo 6 con *keep_prob* = 0.87, β = 0.8 y 80.000 iteraciones 92
Tabla 3-87. Modelo 6 con *keep_prob* = 0.85, β = 0.88 y 100.000 iteraciones 92
Tabla 3-88. Modelo 6 con *keep_prob* = 0.88, β = 0.92 y 100.000 iteraciones 92
Tabla 3-89. Modelo 6 con *keep_prob* = 0.86, β = 0.83 y 100.000 iteraciones 92
Tabla 3-90. Modelo 6 con α = 0.1 y estructura [17, 64, 32, 16, 6] 93
Tabla 3-91. Modelo 6 con α = 0.05 y estructura [17, 64, 32, 16, 6] 93
Tabla 3-92. Modelo 6 con α = 0.01 y estructura [17, 64, 32, 16, 6] 93
Tabla 3-93. Modelo 6 con α = 0.1 y estructura [17, 60, 40, 20, 6] 94
Tabla 3-94. Modelo 6 con α = 0.05 y estructura [17, 60, 40, 20, 6] 94
Tabla 3-95. Modelo 6 con α = 0.01 y estructura [17, 60, 40, 20, 6] 94
Tabla 3-96. Modelo 7 con α = 0.001, $\beta 1$ = 0.9 y 20.000 iteraciones 97
Tabla 3-97. Modelo 7 con α = 0.001, $\beta 1$ = 0.9 y 40.000 iteraciones 97
Tabla 3-98. Modelo 7 con α = 0.001, $\beta 1$ = 0.8 y 20.000 iteraciones 98
Tabla 3-99. Modelo 7 con α = 0.001, $\beta 1$ = 0.8 y 40.000 iteraciones 98
Tabla 3-100. Modelo 7 con α = 0.005, $\beta 1$ = 0.8 y 20.000 iteraciones 98
Tabla 3-101. Modelo 7 con α = 0.005, $\beta 1$ = 0.8 y 40.000 iteraciones 98

Tabla 3-102. Modelo 7 con α = 0.001, $\beta 1$ = 0.9 y 20.000 iteraciones sobre C.D. 1 99
Tabla 3-103. Modelo 7 con α = 0.001, $\beta 1$ = 0.9 y 20.000 iteraciones sobre C.D. 2 99
Tabla 3-104. Modelo 7 con α = 0.001, $\beta 1$ = 0.9 y 20.000 iteraciones sobre C.D. 3 99
Tabla 3-105. Modelo 7 con α = 0.001, $\beta 1$ = 0.9 y 20.000 iteraciones sobre C.D. 4 99
Tabla 3-106. Modelo 7 con α = 0.001, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 1 100
Tabla 3-107. Modelo 7 con α = 0.001, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 2 100
Tabla 3-108. Modelo 7 con α = 0.001, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 3 100
Tabla 3-109. Modelo 7 con α = 0.001, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 4 100
Tabla 3-110. Modelo 7 con α = 0.005, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 1 101
Tabla 3-111. Modelo 7 con α = 0.005, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 2 101
Tabla 3-112. Modelo 7 con α = 0.005, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 3 101
Tabla 3-113. Modelo 7 con α = 0.005, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 4 101
Tabla 3-114. Media modelo 7 con α = 0.001, $\beta 1$ = 0.9 y 20.000 iteraciones 102
Tabla 3-115. Media modelo 7 con α = 0.001, $\beta 1$ = 0.8 y 40.000 iteraciones 102
Tabla 3-116. Media modelo 7 con α = 0.005, $\beta 1$ = 0.8 y 40.000 iteraciones 102
Tabla 3-117. Resultados modelo 8 con α = 0.1, *keep_prob* = 0.5 103
Tabla 3-118. Resultados modelo 8 con α = 0.1, *keep_prob* = 0.75 104
Tabla 3-119. Resultados modelo 8 con α = 0.1, *keep_prob* = 0.87 105
Tabla 3-120. Resultados modelo 9 con estructura [17, 32, 32, 25, 25, 16, 6], α = 0.01 y *keep_prob* = 0.87 106
Tabla 3-121. Resultados modelo 9 con estructura [17, 32, 25, 25, 20, 16, 6], α = 0.01 y *keep_prob* = 0.87 107
Tabla 3-122. Resultados modelo 9 con estructura [17, 20, 35, 25, 25, 16, 6], α = 0.01 y *keep_prob* = 0.87 107
Tabla 3-123. Resultados modelo 9 con estructura [17, 25, 25, 25, 25, 16, 6], α = 0.01 y *keep_prob* = 0.87 108
Tabla 4-1. Resultados e hiperparámetros de los mejores modelos predictivos 113

Índice de Figuras

Figura 2-1. Estructura red neuronal multicapa 8
Figura 2-2. Procesamiento de una neurona 10
Figura 2-3. Distribución inicial de datos 22
Figura 2-4. Distribución de datos normalizados 22
Figura 2-5. Distribución de datos normalizados con eje a escala 22
Figura 2-6. Camino de optimización datos no normalizados. 23
Figura 2-7. Camino de optimización datos normalizados. 23
Figura 2-8. Red neuronal de tres capas 26
Figura 2-9. Red neuronal aplicando *dropout* 26
Figura 2-10. *Gradient descent with momentum* 28
Figura 3-1. *Missing values* 40
Figura 3-2.Conjunto de datos segregado según sexo donante 41
Figura 3-3. Conjunto de datos segregado según edad donante 41
Figura 3-4. Peso del donante según su sexo 42
Figura 3-5. *Missing values* tras la 1ª imputación 43
Figura 3-6. Altura del donante según su sexo 44
Figura 3-7. *Missing values* tras la 2ª imputación 44
Figura 3-8. Peso del paciente según su sexo 45
Figura 3-9. *Missing values* tras la 3ª imputación 45
Figura 3-10. Altura del paciente según su sexo 46
Figura 3-11. *Missing values* tras la 4ª imputación 46
Figura 3-12. Creatinina del donante según su sexo 47
Figura 3-13. Creatinina del donante según su edad 47

Figura 3-14. Creatinina del donante según su causa de la muerte 48
Figura 3-15. Creatinina del donante según su sexo y tipo de donante 48
Figura 3-16. *Missing values* tras la 5ª imputación 49
Figura 3-17. Número de incompatibilidades HLA-DR 49
Figura 3-18. Número de incompatibilidades HLA-DR según sexo donante 50
Figura 3-19. Número de incompatibilidades HLA-DR según sexo paciente 50
Figura 3-20. Número de incompatibilidades HLA-DR según causa de la muerte 50
Figura 3-21. *Missing values* tras la 6ª imputación 51
Figura 3-22. Evolución del coste modelo 1 con $\alpha = 0.001$ 56
Figura 3-23. Evolución del coste modelo 1 con $\alpha = 0.01$ 57
Figura 3-24. Evolución del coste modelo 1 con $\alpha = 0.05$ 57
Figura 3-25. Evolución del coste modelo 1 con $\alpha = 0.1$ 58
Figura 3-26. Evolución del coste modelo 1 con estructura [17, 28, 8, 6] 59
Figura 3-27. Evolución del coste modelo 1 con estructura [17, 25, 14, 6] 59
Figura 3-28. Evolución del coste modelo 1 con estructura [17, 20, 20, 6] 60
Figura 3-29. Evolución del coste modelo 3 estructura [17, 20, 20, 6, 6] 63
Figura 3-30. Evolución del coste modelo 3 estructura [17, 20, 20, 10, 6] 64
Figura 3-31. Evolución del coste modelo 3 estructura [17, 25, 25, 16, 6] 64
Figura 3-32. Evolución del coste modelo 3 estructura [17, 25, 25, 16, 6] sobre C.D. 1 67
Figura 3-33. Evolución del coste modelo 3 estructura [17, 25, 25, 16, 6] sobre C.D. 2 68
Figura 3-34. Evolución del coste modelo 3 estructura [17, 25, 25, 16, 6] sobre C.D. 3 68
Figura 3-35. Evolución del coste modelo 3 estructura [17, 25, 25, 16, 6] sobre C.D. 4 69
Figura 3-36. Evolución del coste modelo 3 estructura [17, 25, 25, 16, 6] sobre C.D. 5 70
Figura 3-37. Evolución del coste modelo 4 con *lambda* = 0.5 73
Figura 3-38. Evolución del coste modelo 4 con *lambda* = 0.7 73
Figura 3-39. Evolución del coste modelo 4 con *lambda* = 0.7 y 25.000 iteraciones 74
Figura 3-40. Evolución del coste modelo 4 con *lambda* = 0.5 y 20.000 iteraciones 75
Figura 3-41. Evolución del coste modelo 4 con *lambda* = 0.5 y 25.000 iteraciones 75
Figura 3-42. Evolución del coste modelo 4 con *lambda* = 0.7 y 20.000 iteraciones 75
Figura 3-43. Evolución del coste modelo 4 con *lambda* = 0.7 y 25.000 iteraciones 75
Figura 3-44. Evolución del coste modelo 4 con *lambda* = 0.7 y 15.000 iteraciones 76
Figura 3-45. Evolución del coste modelo 4 con *lambda* = 0.7 y 20.000 iteraciones 76
Figura 3-46. Evolución del coste modelo 4 con *lambda* = 0.9 y 15.000 iteraciones 77
Figura 3-47. Evolución del coste modelo 4 con *lambda* = 0.9 y 20.000 iteraciones 77
Figura 3-48. Evolución del coste modelo 4 con *lambda* = 0.9 y 25.000 iteraciones 77
Figura 3-49. Evolución del coste modelo 4 con *lambda* = 1.2 y 20.000 iteraciones 78
Figura 3-50. Evolución del coste modelo 4 con *lambda* = 1.2 y 25.000 iteraciones 78
Figura 3-51. Evolución del coste modelo 5 con *keep_prob* = 0.5 81
Figura 3-52. Evolución del coste modelo 5 con *keep_prob* = 0.5 82

Figura 3-53. Evolución del coste modelo 5 con *keep_prob* = 0.65 83
Figura 3-54. Evolución del coste modelo 5 con *keep_prob* = 0.8 83
Figura 3-55. Evolución del coste modelo 5 con *keep_prob* = 0.9 84
Figura 3-56. Evolución del coste modelo4 con *keep_prob* = 0.9 y 30.000 iteraciones sobre C.D. 1 85
Figura 3-57. Evolución del coste modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones sobre C.D. 2 85
Figura 3-58. Evolución del coste modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones sobre C.D. 3 85
Figura 3-59. Evolución del coste modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 1 86
Figura 3-60. Evolución del coste modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 2 86
Figura 3-61. Evolución del coste modelo4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 3 87
Figura 3-62. Evolución del coste modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 4 87
Figura 3-63. Evolución del coste modelo 5 con *keep_prob* = 0.8 y 100.000 iteraciones 88
Figura 3-64. Evolución del coste modelo 5 con *keep_prob* = 0.85 y 100.000 iteraciones 89
Figura 3-65. Evolución del coste modelo 6 con *keep_prob* = 0.87, β = 0.9 y 100.000 iteraciones 91
Figura 3-66. Evolución del coste modelo 6 con α = 0.1 y estructura [17, 64, 32, 16, 6] 93
Figura 3-67. Evolución del coste modelo 6 con α = 0.05 y estructura [17, 64, 32, 16, 6] 93
Figura 3-68. Evolución del coste modelo 6 con α = 0.01 y estructura [17, 64, 32, 16, 6] 93
Figura 3-69. Evolución del coste modelo 6 con α = 0.1 y estructura [17, 60, 40, 20, 6] 94
Figura 3-70. Evolución del coste modelo 6 con α = 0.05 y estructura [17, 60, 40, 20, 6] 94
Figura 3-71. Evolución del coste modelo 6 con α = 0.01 y estructura [17, 60, 40, 20, 6] 94
Figura 3-72. Evolución del coste modelo 7 con α = 0.001, $\beta 1$ = 0.9 y 20.000 iteraciones 98
Figura 3-73. Evolución del coste modelo 7 con α = 0.001, $\beta 1$ = 0.9 y 40.000 iteraciones 98
Figura 3-74. Evolución del coste modelo 7 con α = 0.001, $\beta 1$ = 0.8 y 20.000 iteraciones 98
Figura 3-75. Evolución del coste modelo 7 con α = 0.001, $\beta 1$ = 0.8 y 40.000 iteraciones 98
Figura 3-76. Evolución del coste modelo 7 con α = 0.005, $\beta 1$ = 0.8 y 20.000 iteraciones 99
Figura 3-77. Evolución del coste modelo 7 con α = 0.005, $\beta 1$ = 0.8 y 40.000 iteraciones 99
Figura 3-78. Evolución del coste modelo 7 con α = 0.001, $\beta 1$ = 0.9 y 20.000 iteraciones sobre C.D. 1 100
Figura 3-79. Evolución del coste modelo 7 con α = 0.001, $\beta 1$ = 0.9 y 20.000 iteraciones sobre C.D. 2 100
Figura 3-80. Evolución del coste modelo 7 con α = 0.001, $\beta 1$ = 0.9 y 20.000 iteraciones sobre C.D. 3 100
Figura 3-81. Evolución del coste modelo 7 con α = 0.001, $\beta 1$ = 0.9 y 20.000 iteraciones sobre C.D. 4 100
Figura 3-82. Evolución del coste modelo 7 con α = 0.001, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 1 101
Figura 3-83. Evolución del coste modelo 7 con α = 0.001, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 2 101
Figura 3-84. Evolución del coste modelo 7 con α = 0.001, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 3 101
Figura 3-85. Evolución del coste modelo 7 con α = 0.001, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 4 101
Figura 3-86. Evolución del coste modelo 7 con α = 0.005, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 1 102
Figura 3-87. Evolución del coste modelo 7 con α = 0.005, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 2 102
Figura 3-88. Evolución del coste modelo 7 con α = 0.005, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 3 102
Figura 3-89. Evolución del coste modelo 7 con α = 0.005, $\beta 1$ = 0.8 y 40.000 iteraciones sobre C.D. 4 102
Figura 3-90. Evolución del coste modelo 8 con α = 0.1, *keep_prob* = 0.5 104
Figura 3-91. Evolución del coste modelo 8 con α = 0.1, *keep_prob* = 0.75 104

Figura 3-92. Evolución del coste modelo 8 con α = 0.1, *keep_prob* = 0.87 105

Figura 3-93. Evolución del coste modelo 9 con estructura [17, 32, 32, 25, 25, 16, 6], α = 0.01 y *keep_prob* = 0.87 106

Figura 3-94. Evolución del coste modelo 9 con estructura [17, 32, 25, 25, 20, 16, 6], α = 0.01 y *keep_prob* = 0.87 107

Figura 3-95. Evolución del coste modelo 9 con estructura [17, 20, 35, 25, 25, 16, 6], α = 0.01 y *keep_prob* = 0.87 108

Figura 3-96. Evolución del coste modelo 9 con estructura [17, 20, 35, 25, 25, 16, 6], α = 0.01 y *keep_prob* = 0.87 109

Notación

$a^{[l]}$	Elemento *a* de la capa *l*
$a_n^{[l]}$	Elemento *a* de la neurona *n* de la capa *l*
W^T	Transpuesta de la matriz *W*
$x^{(i)}$	Elemento *i* de un conjunto *x*
$g(x)$	Función *g* de *x*
$g'(x)$	Derivada de la función *g* de *x*
a_{ij}	Elemento de una matriz correspondiente a la posición *i, j*
$\hat{y}$	Resultado de la última capa de una red neuronal
$\max(x, y)$	Función que devuelve el valor mayor entre *x* e *y*
e	Constante *e* o número de Euler
$\log(x)$	Logaritmo natural o neperiano de *x*
$P(y\|x)$	Probabilidad de *y* respecto a *x*
$\sum_{i=0}^{n} x$	Sumatorio de *i* igual a 0 a *n* del elemento *x*
$\prod_{k=1}^{n} x$	Producto de *k* igual a 1 a *n* del elemento *x*
$\frac{\partial y}{\partial x}$	Derivada parcial de *y* sobre *x*
$*$	Producto Hadamard o multiplicación elemento a elemento
$\cdot$	Producto matricial o producto de matrices
$\|A\|_F$	Normalizador de Frobenius aplicado a la matriz *A*
$<$	Menor que
$>$	Mayor que
$\leq$	Menor o igual que
$\geq$	Mayor o igual que

1 INTRODUCCIÓN Y MOTIVACIÓN

Soy súper optimista sobre las perspectivas a corto plazo de la IA porque cada vez que hay una disrupción tecnológica, nos da la oportunidad de hacer el mundo un poco diferente.

- Andrew Ng -

Mucho se ha escuchado hablar en los últimos años de la inteligencia artificial (IA): algoritmos que imitan funcionamientos cerebrales, máquinas desarrollando lenguajes propios, robots programados para imitar personas, etc. Y es que, entre titulares cargados de misticismo, con los que muchas veces se tiende a dar bombo a artículos o descubrimientos científicos, y películas futuristas, se ha creado un sentimiento que deriva, entre expectación y miedo, en diferentes conversaciones que, siendo importantes e interesantes desde puntos de vista filosóficos, poco o nada tienen que ver con la realidad de la inteligencia artificial tal y como la conocemos hoy en día.

Dicho esto, nos encontramos con una de las grandes revoluciones tecnológicas, si no la más grande, de toda la historia de la humanidad, ya que en los próximos años se verá realmente el impacto que tendrá esta inteligencia artificial y cómo cambiará las relaciones hombre-máquina. El cómo seremos capaces de integrarnos con esta definirá nuestra capacidad para atajar o resolver los problemas más grandes a los que se enfrentará la humanidad en las próximas décadas.

Para quien no esté familiarizado todavía con el campo de la inteligencia artificial, el principal hito que esta tecnología cumple, y por el cual se le aplica la palabra inteligencia a su nombre, es su capacidad de aprender. Aprender en el sentido más estricto de la palabra, entendiendo por ello el hecho de realizar un proceso, fallar y ser capaz de realizarlo mejor la siguiente vez, e iteración tras iteración perfeccionar su habilidad y aumentar el rendimiento de su trabajo. En esta explicación he utilizado estos conceptos deliberadamente para entender, que parte de la importancia que le damos a este hito y por el cual la nombramos “inteligencia” artificial es porque es ineludible encontrar semejanzas en los procesos de aprendizaje que nosotros como humanos realizamos para poder aprender cualquier habilidad que podamos considerar propia. Desde leer, escribir o caminar, todos nuestros procesos de aprendizaje pasan por repetir tantas veces como sea necesario hasta conseguir dominar una habilidad al nivel de no tener prácticamente errores al realizarla; aunque igualmente seguimos equivocándonos al leer, cometiendo alguna falta de ortografía al escribir y tropezándonos cuando menos lo esperamos.

Y si a este proceso de ensayo y error, lo llamamos inteligencia cuando termina dando sus frutos en un aprendizaje, nos sentimos apelados a llamar inteligencia artificial a algo que nosotros hemos creado y en la que reconocemos ciertos mecanismos propios que utilizamos para la búsqueda del conocimiento.

Es esta continua búsqueda del conocimiento la que hace que las posibilidades de la inteligencia artificial sean infinitas. Aunque el término infinito en ámbitos científicos no sería el correcto y no pretendo sentar cátedra al respecto, por lo que mejor diré que tiene posibilidades imposibles de imaginar en la actualidad, de la misma

manera que el ser humano en su búsqueda del conocimiento ha llegado a lugares que no se imaginaban posibles.

Es esta capacidad de aprendizaje la que abre el camino de la búsqueda del conocimiento y es esta la que crea tantas expectativas e ilusiones al respecto de esta nueva tecnología que definitivamente revolucionará cada sector de la sociedad hacia un nuevo paradigma en el que ya no nos valgamos únicamente de nuestra propia inteligencia si no que habremos desarrollado una nueva inteligencia que nos ayude en el camino.

Sin embargo, esto que escribo no se corresponde con la actualidad. En estos momentos, esta tecnología está naciendo y el mayor desafío es ser capaz de entender cómo funciona y desentrañar cuál será el siguiente paso en su evolución hacia una inteligencia artificial como la contemplamos en nuestro imaginario común.

Para hacer una distinción real de estos dos estados, los científicos suelen utilizar los términos: inteligencia artificial fuerte e inteligencia artificial débil.

Es consensuado que actualmente nos encontramos en los primeros pasos de esta inteligencia artificial débil y que queda aún mucho camino para poder realmente atravesar la barrera hacia una tecnología que se pueda equiparar a las que podemos observar en las obras literarias o audiovisuales más futuristas.

Y aún con todo esto, los primeros pasos de esta inteligencia artificial débil ya han tenido grandes resultados en diversos campos, quizá por esto resulta tan prometedora. Algunos de estos primeros logros son:

- DeeperBlue.
  Después de que en 1996 el ordenador DeepBlue perdiera contra el maestro ajedrecista Garri Kaspárov, en 1997 un nuevo modelo mejorado llamado DeeperBlue conseguiría por primera vez ganar una partida de ajedrez contra un humano.
- AlphaGo - DeepMind
  No fue hasta 2005, 8 años después de DeeperBlue y la primera victoria de la IA, que una inteligencia artificial lograra derrotar a un humano en una partida de Go, el que se dice que es el juego de mesa más complicado del mundo.
- WATSON
  Así se llamaba el proyecto de IBM que participó en el programa estadounidense “Jeopardy!”, un juego de preguntas y respuestas en el que las preguntas necesitan una interpretación humana para poder resolverlas, y donde triunfó frente a oponentes humanos
- Coches autónomos o semiautónomos
  Proyectos de coches autónomos como el que está desarrollando Google, o semiautónomos como el de la compañía Tesla, que prevén un futuro en el que el transporte esté completamente automatizado.
- Asistentes de voz
  En los últimos tiempos se han popularizado los asistentes de voz que faciliten la comunicación e interacción de los seres humanos con las máquinas o el entorno digital. Los más conocidos son proyectos como Siri, de Apple, Cortana, de Google, o Alexa, de Amazon.

El éxito de estos y otros ejemplos en sus campos de aplicación han ocurrido inequívocamente gracias a su capacidad de aprendizaje, pero este no ha sido el único pilar sobre el que se ha erigido este éxito.

Otro punto clave han sido los datos. La gran cantidad de datos almacenada durante décadas por los seres humanos desde que crearon los sistemas informáticos, los comercializaron y decidieron que deberían almacenar todos los datos generados con su uso. En las últimas décadas, conforme avanzaba la tecnología, los objetos donde se almacenaban datos cada vez se hacían más eficientes y permitían almacenar una cantidad de datos que para un ser humano eran prácticamente imposibles de analizar. A estos datos se les aplicaron programas informáticos que permitían su uso, pero no aportaban nada más de lo que el programador que lo había diseñado podía imaginar. Ya que esta es la limitación de los programas tradicionales, hacen exactamente lo que se les diga que hagan, con mucha eficiencia, es cierto, pero absolutamente nada más.

Y aquí es donde ha sobresalido, por encima de cualquier otro método analítico, la inteligencia artificial. Es capaz de procesar una cantidad inmensa de datos, interpretarlos y encontrar patrones entre ellos. Y a base de repetir y repetir, el proceso de aprendizaje se vuelve mucho más eficiente encontrando estos patrones y dando predicciones acerca de otros datos con las mismas propiedades que los que ya ha trabajado durante el aprendizaje.

Y es en este tipo de aplicaciones de inteligencia artificial donde queda recogido este proyecto. A partir de un conjunto de datos se estudiará la relación y la importancia de estos mediante modelos de inteligencia artificial de aprendizaje profundo para observar si, mediante el proceso de aprendizaje, se logra obtener una predicción de resultados equiparable a la de otros sistemas estadísticos utilizados en la actualidad.

El concepto de aprendizaje profundo (o “Deep learning”) tiene que ver con la estructura que adquieren las redes neuronales y se tratará en el siguiente capítulo de manera que se entienda la diferencia entre estos sistemas y otros que también utilizan los procesos de aprendizaje de la inteligencia artificial pero no se consideran aprendizaje profundo.

En esta línea de proyectos, el trabajo realizado se centrará en el análisis de un conjunto de datos recogidos en las últimas dos décadas de procedimiento de trasplante de riñón. Estos son datos reales cedidos por la Coordinación Autonómica de Trasplantes de Andalucía, que recopila los datos de los trasplantes realizados en la Comunidad Autónoma desde hace más de 20 años.

Los trasplantes de riñón son procedimientos quirúrgicos que se prescriben a pacientes que sufren de insuficiencia renal, la cuál es una condición que impide el correcto funcionamiento de los riñones. Estos se dedican al filtrado de desechos, minerales y otros líquidos que se hayan en la sangre con el objetivo de que no se acumulen en el cuerpo y poder expulsarlos. Al fallar este proceso de filtrado, elementos nocivos se acumulan en el interior del cuerpo y cuando la capacidad de filtrado cae por debajo del 10% se diagnostica como enfermedad renal terminal. En este punto, el paciente solamente puede eliminar estos desechos de su torrente sanguíneo por medio de la diálisis, el cual es un proceso mediante el que toda la sangre del paciente se pasa a través de una máquina que se encarga de limpiarla. Este proceso se debe repetir de manera regular con el fin de mantener al paciente sano. La única solución posible para evitar los procesos de diálisis es someterse a un trasplante de riñón.

El trasplante de riñón no se considera una cura si no un tratamiento para la insuficiencia renal. Como resultado, el paciente deberá seguir una medicación diaria para evitar que su sistema inmune provoque el rechazo del órgano implantado mientras se mantiene bajo supervisión médica. Además, el tiempo de vida del órgano implantado es incierto y depende en gran medida de las condiciones del donante, del receptor y las características propias del proceso de trasplante. Por otro lado, pese a la superioridad del trasplante sobre la diálisis, no todos los pacientes en lista de espera pueden acceder a un órgano compatible porque la demanda es mucho mayor que la oferta.

Todas estas limitaciones animan a investigar en las variables que influyen en la supervivencia del órgano tras el trasplante con el ánimo de poder predecir en el instante pre-implante cuál va a ser el resultado, medido en tiempo, del procedimiento. Con ese conocimiento sería posible emparejar a los donantes y receptores más afines para optimizar la asignación de órganos y aumentar el éxito a largo plazo de los trasplantes.

El proyecto se propone utilizar los algoritmos de la inteligencia artificial para ahondar en todas estas variables, con el fin de hallar los patrones que las relacionan con el éxito o el fracaso de los procedimientos quirúrgicos. Teniendo los datos previos al trasplante y los resultados posteriores, se pretende diseñar un modelo predictivo que pueda reducir el número de rechazos o muertes en los pacientes que se someten a un trasplante de riñón.

Previamente a este proyecto, se han hecho diversos trabajos que tratan de aumentar las posibilidades de supervivencia de un paciente ante cierta enfermedad o tratamiento. Los resultados que han logrado son muy optimistas ya que incluso con un número de datos relativamente pequeño se obtenían porcentajes muy buenos. Este proyecto se centrará en aprender los últimos y más eficientes algoritmos utilizados, desarrollar un modelo de red neuronal y encontrar la mejor manera de aplicarlo a los datos cedidos relativos a los trasplantes de riñón en Andalucía. El modelo se desarrollará en Python 3 y durante el proyecto se irá combinando las fórmulas de lo algoritmos matemáticos descritos junto con su programación correspondiente en este lenguaje.

2 MATERIALES Y MÉTODOS

El aprendizaje profundo transformará todos los sectores. La sanidad y el transporte se verán transformados por el aprendizaje profundo. Quiero vivir en una sociedad impulsada por la IA. Cuando alguien vaya a ver a un médico, quiero que la IA le ayude a proporcionar un servicio médico de mayor calidad y menor coste.

- Andrew Ng -

En este capítulo se recopilarán las bases teóricas sobre las que se basa el desarrollo del proyecto y se planteará el objetivo. Se describirá el acercamiento teórico, la metodología utilizada y las bases de conocimiento sobre las que se construye el proyecto. Se describirán los elementos básicos de una red neuronal de aprendizaje profundo y su importancia en el funcionamiento general del modelo. Se aportarán los métodos utilizados, el objetivo de emplearlo en de la red neuronal, la importancia de su aplicación y se adjuntarán sus algoritmos. Así mismo, la implementación en código de todos los elementos descritos se podrá encontrar en el Anexo A, en orden de aparición, para su consulta.

2.1 Objetivo

Este proyecto cuenta con dos objetivos interdependientes en su consecución. El primer objetivo, y por el que se lleva a cabo este proyecto, es el de analizar los datos proporcionados por la Coordinación Autonómica de Trasplantes de Andalucía mediante técnicas de redes neuronales de aprendizaje profundo. Este objetivo busca encontrar nuevos métodos para que los expertos sanitarios cuenten con más información de la que tienen en la actualidad a la hora de la toma de decisiones en un procedimiento de trasplante. En la actualidad se cuenta con métodos estadísticos y, sobre todo, la experiencia clínica de los médicos para llevar a cabo el análisis pormenorizado de los casos.

El segundo objetivo es el de aprender y comprender los métodos matemáticos y técnicas empleadas en la construcción de modelos de inteligencia artificial mediante redes neuronales de aprendizaje profundo. Además, ser capaz de llevar ese conocimiento a la práctica en forma de programació de algoritmos mediante código.

2.2 Metodología

En primer lugar, se pasará a describir cuál ha sido el acercamiento al tratamiento del problema y cómo se ha ido desarrollando el proyecto en base a estas decisiones.

El objetivo principal del proyecto no ha sido únicamente obtener resultados y comparar estos con los resultados obtenidos por otras maneras más clásicas o estadísticas, si no también comprender cómo funcionan

los algoritmos utilizados en la inteligencia artificial.

En la actualidad hay muchos programas o extensiones de lenguajes de programación orientadas a proyectos de inteligencia artificial. Algunos de estos muy conocidos son: TensorFlow [1], PyTorch [2] o Keras [3]. Algunos son bibliotecas de algoritmos, como Keras, y otros son directamente nuevos lenguajes basados en lenguajes ya existentes, como TensorFlow o PyTorch. Tienen algunas ventajas claves que los hacen muy atractivos para su empleo, como traer ya los algoritmos previamente desarrollados e implementados, necesitando solamente llamar a la función adecuada para su aplicación. Otra ventaja que tienen algunos es que corren el programa sobre la GPU del ordenador, si esta es compatible con el programa, haciendo mucho más rápido el procesamiento de los datos y, como consecuencia, reduciendo los tiempos de ejecución del modelo.

Sin embargo, se ha decidido no utilizar ninguno de estos ya que pierden de vista uno de los objetivos de este proyecto que es entender qué algoritmos se aplican en los modelos de red neuronal y cómo procesan los datos, ganando así el entendimiento de qué ocurre realmente con estos métodos y por qué funcionan.

Teniendo esto en cuenta, se ha decidido desarrollar todo el proyecto en Python 3 [4]. Se utilizarán diferentes librerías para el tratamiento previo de los datos y la creación de los algoritmos utilizados en la creación del modelo.

Los datos recibidos están en bruto y cuentan con una cantidad muy amplia de información. Habrá que analizar con qué variables se cuenta, entender su significado y reducir su número a las variables más significativas. Posteriormente, habrá que realizar un tratamiento de estos datos, ya que habrá algunas variables que sufran de tener datos perdidos. Una vez realizado este proceso se contará con un conjunto de datos preparados para ser utilizados en el entrenamiento del modelo.

Para el modelo se ha elegido la estructura de red neuronal multicapa, también conocida como perceptrón multicapa [5] [6]. Las redes neuronales de aprendizaje profundo [7] [8] resultan especialmente útiles a la hora de intentar predecir resultados en base a unos determinados datos de entrada mientras la distribución de estos datos sea igual o muy similar a la distribución de los datos con los que se haya entrenado, característica que se cumple en este caso.

La tipología de aprendizaje será de aprendizaje supervisado. Esto indica que se entrenará la red neuronal proporcionando al modelo tanto los datos de entrada como el resultado final esperado, midiendo el error resultante y recalculando los valores de las variables de la red en función de este.

2.3 Hardware

Para el desarrollo de este proyecto se ha utilizado el ordenador portátil Lenovo Ideapad 700-15isk 80ru i5 6300HQ. En la siguiente tabla se describen las características de esta máquina.

Tabla 2-1. Características técnicas del ordenador

Elemento	Característica técnica
Procesador	Intel® Core™ i5-6300HQ CPU @ 2.30GHz 2.30 GHz
RAM	8,00 GB (7,80 GB usable) DDR4
Disco duro	SSD 500 GB
SO	Windows 10 Home x64 bits
Tarjeta gráfica	Nvidia GeForce GTX 950M 4 GB GDDR3

2.4 Software

2.4.1 Lenguaje de programación

Como se ha descrito previamente, el lenguaje de programación utilizado será Python 3. Las principales

ventajas de este lenguaje por las que ha sido el elegido para basar este proyecto son varias. La primera es que es un lenguaje de código abierto, por lo que además de estar alineado con las preferencias propias como autor del trabajo también tiene la ventaja de que hay muchos sitios web y foros con un número muy elevado de participantes compartiendo información de sus experiencias, problemas y soluciones aplicadas. Esta característica de código abierto también permite que existan muchas bibliotecas para diferentes aplicaciones y estas estén constantemente actualizadas y mantenidas. Además, es un lenguaje orientado a objetos y su manera de construcción permite mucha libertad y flexibilidad a la hora de escribir el código.

Las bibliotecas utilizadas durante el proyecto son:

- **NumPy** [7]

  La biblioteca por excelencia utilizada por en los campos científicos donde se programa con Python. Permite la utilización clara y elegante de complejos procesamientos matemáticos aportando a este lenguaje la potencia computacional que se consigue en otros como C o Fortran. En este proyecto será la base de los tratamientos de matrices que son los elementos principales con los que trabajan los algoritmos de las redes neuronales.

- **pandas** [8]

  La principal funcionalidad para tratamiento de datos. Es la biblioteca utilizada por los científicos del campo del Big Data para tratar, interpretar y transformar los bloques de datos convirtiéndolos en objetos computables y procesables por un lenguaje de programación.

- **Math** [9]

  Módulo que aporta acceso a todo tipo de funciones matemáticas definidas en el estándar C. Muchas veces en este proyecto quedará relegado a un segundo plano en beneficio de NumPy debido a que prácticamente la totalidad de los cálculos matemáticos implican trabajo con matrices.

- **Matplotlib** [10]

  Biblioteca cuya funcionalidad se basa en crear gráficos a partir de los datos aportados en Python 3. La principal utilización de esta en el proyecto será el recrear gráficamente funciones de coste cuyo significado veremos más adelante.

- **Time** [11]

  Como su propio nombre indica, es una biblioteca dedicada al tratamiento del tiempo. Utilizada en su mayor parte para llevar la cuenta del tiempo de procesamiento y poder comparar diferentes modelos.

- **Seaborn** [12]

  Es una biblioteca basada en Matplotlib (arriba definida) que permite la visualización de los datos de manera analítica y añade funcionalidades estadísticas de gran valor.

2.4.2 Entorno de programación

El proyecto se llevará a cabo utilizando la aplicación Anaconda como distribución [13]. Esta lleva consigo los mismos valores de código abierto y facilidad de uso que se han seguido para la elección de Python 3 como lenguaje de programación.

Permite una sencilla instalación y tratamiento de los paquetes necesarios para el desarrollo de aplicaciones que trabajan con gran cantidad de datos.

Dentro de Anaconda se utilizará Jupyter Notebook [14], la cual es una aplicación de formato libreta donde se escribirá el código y sobre la que se puede ejecutar y visualizar los resultados directamente.

2.5 Modelo de red neuronal

En este apartado se describirán los elementos principales y el funcionamiento de una red neuronal multicapa, así como las funciones o algoritmos utilizados a lo largo del proyecto.

2.5.1 Elementos y funcionamiento

La estructura de los modelos de red neuronal multicapa, o perceptrón multicapa, se asemeja a la imagen más convencional que se tiene acerca del esquema de una red neuronal. A continuación, aparece una imagen ilustrativa de esta estructura.

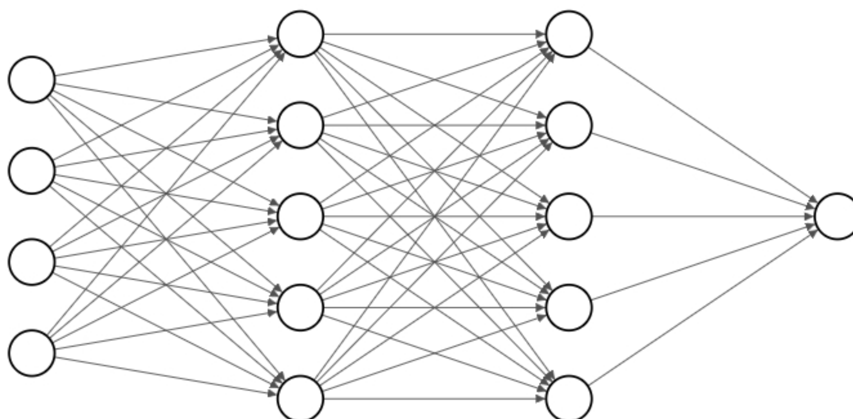

Figura 2-1. Estructura red neuronal multicapa

Los principales elementos que se encuentran a primera vista son las neuronas, o nodos. Las neuronas están agrupadas en capas y cada capa está conectada con la siguiente. La red puede clasificarse como totalmente conectada, como en la red de la Figura 2-1, o parcialmente conectada si algunas de las conexiones (flechas) hubieran sido eliminadas.

Esta estructura se llama multicapa, porque además de la capa de entrada, situada en el extremo izquierdo, y la capa de salida, situada en el extremo derecho, se puede encontrar capas intermedias, las cuales son llamadas capas ocultas. Además, la red neuronal que se observa en la Figura 2-1 sería definida como una red neuronal de 3 capas, dado que la capa de entrada no se contabiliza a la hora de describirla.

A través de la capa de entrada introducimos los datos de nuestro ejemplo, y estos se propagan hacia delante a todas las neuronas de la capa siguiente. En esta nueva capa, las neuronas aplican transformaciones algebraicas a los datos que han recibido de la capa anterior y propagan este resultado a todas las neuronas de la capa siguiente. Debe ser resaltado que todas las neuronas reciben todos los resultados de la capa anterior previamente a hacer cualquier tipo de cálculo que esté definido en la propia neurona. De nuevo estos resultados se propagan hacia delante y terminan en la capa de salida la cuál es la encargada de interpretar los datos obtenidos de la capa anterior y dar un resultado concreto. En último punto, se evaluará el rendimiento de esta red comparando el resultado obtenido con el resultado real que se tiene previamente. Esta diferencia entre resultado y realidad se le llama coste y el cálculo de este se denomina función de coste. Cuanto más alto sea nuestro coste menos precisión tendrá nuestro modelo en sus predicciones, pero, como se verá más adelante, el fin de este proceso no es lograr tener coste cero ya que esto puede tener otras implicaciones también desfavorables para el éxito del modelo a la hora de aplicarlo.

Sin embargo, minimizar este coste será el objetivo principal para lograr el aprendizaje del modelo y lograr mejores predicciones [17]. Para ello se realizará una propagación hacia atrás, que está definida por el cálculo de los gradientes de las variables implicadas en la propagación hacia delante. Se comienza calculando el valor del gradiente de la capa de salida y se avanza por las capas siguiendo el orden contrario al de la propagación hacia delante hasta llegar a la capa de entrada. Estos gradientes se utilizarán junto con el *learning rate*, o ratio de aprendizaje [15], para actualizar los pesos de la red neuronal. El *learning rate* es el primer hiperparámetro descrito en este proyecto y el más importante a tener en cuenta. Esta variable definirá la velocidad de aprendizaje del algoritmo de manera directa, siendo un factor multiplicador de la cantidad que variarán los pesos de la red en cada iteración.

Estos pesos son los parámetros de la red, son los encargados de propagar hacia delante los valores de entrada y son los valores que con cada cálculo e iteración iremos modificando y actualizando, en pos de obtener resultados más acertados, y, en última instancia, son la red misma. Estos pesos son los que definen el comportamiento de la red neuronal y dan forma al modelo. Si se pudiera hacer un símil con la construcción, la

estructura de la red neuronal sería un plano de un edificio y los pesos serían todos los materiales que componen el edificio una vez construido. La distribución y composición de estos son los que permiten que el plano, tal y como estaba definido, tenga como resultado un edificio funcional.

Estos pesos vendrán definidos por matrices y se llamarán W y b. A su vez, el tamaño de estas matrices vendrá definido por la estructura de la red. Por ejemplo, en la red de la Figura 2-1 se tiene una capa de entrada, dos capas ocultas y una capa de salida cuyos tamaños están definidos tal que:

$$\begin{gathered} n^{[0]} = 4 \\ n^{[1]} = 5 \\ n^{[2]} = 5 \\ n^{[3]} = 1 \end{gathered} \tag{2-1}$$

Por lo que el tamaño de los pesos quedaría definido de manera:

$$\begin{aligned} W^{[1]} &= \left(n^{[1]}, n^{[0]}\right) = (5,4) \\ W^{[2]} &= \left(n^{[2]}, n^{[1]}\right) = (5,5) \\ W^{[3]} &= \left(n^{[3]}, n^{[2]}\right) = (1,5) \\ b^{[1]} &= \left(n^{[1]}, 1\right) = (5,1) \\ b^{[2]} &= \left(n^{[2]}, 1\right) = (5,1) \\ b^{[3]} &= \left(n^{[3]}, 1\right) = (1,1) \end{aligned} \tag{2-2}$$

Es importante remarcar que aquí se están definiendo los tamaños de las matrices, no los valores que esta alberga, que nada tienen que ver. Estos valores deberán inicializarse al comienzo de la etapa de aprendizaje y serán los actualizados en cada propagación hacia atrás que se haga en la red. Mientras que el tamaño siempre se mantendrá igual.

De forma general, siendo *l* el número de la capa, se puede describir el tamaño de los pesos tal que:

$$\begin{aligned} W^{[l]} &= \left(n^{[l]}, n^{[l-1]}\right) \\ b^{[l]} &= \left(n^{[l]}, 1\right) \end{aligned} \tag{2-3}$$

Otra variable de gran importancia que se ha mencionado con anterioridad es el *learning rate*. Es una de las variables que se consideran hiperparámetros. Esta debe ser definida por el autor del modelo y deberá afinarse según los resultados lo pidan. Este es el hiperparámetro más importante a tener en cuenta, pero no es el único, se verán otros incluidos en el modelo en el apartado correspondiente.

Ahora que se conocen las variables principales implicadas en el funcionamiento general de la red se pretende hacer un acercamiento al funcionamiento concreto de cada neurona.

Si centramos el estudio en una sola neurona podemos diferenciar dos procesos que toman parte entre la entrada de los datos y la salida de los resultados. El primer proceso es el que viene definido por una función lineal que implica a los datos de entrada y los pesos de la red que vendrá definida por la función:

$$z = w^T x + b \tag{2-4}$$

En el seguno proceso, se aplicará lo que se conoce como función de activación al resultado z de la función lineal del primer proceso. Esta función de activación no es lineal y es clave en el funcionamiento característico de la red neuronal. Esta función vendrá definida por:

$$a = g(z) \tag{2-5}$$

Siendo *g()* una función de activación que será definida por el autor del modelo. Normalmente, y prácticamente en la totalidad de los modelos de redes neuronales, se utiliza la misma función de activación para todas las neuronas de la misma capa. Sin embargo, se pueden utilizar funciones de activación diferentes para capas diferentes. Probar diferentes funciones de activación es otra tarea del autor en busca de obtener un resultado óptimo.

Con lo explicado, se suele definir de manera gráfica el funcionamiento de una neurona como la siguiente imagen:

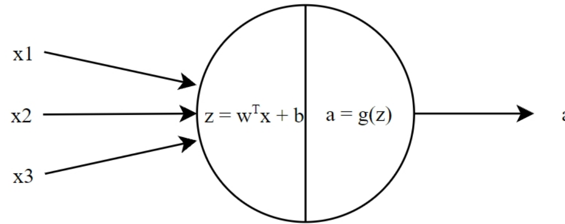

Figura 2-2. Procesamiento de una neurona

La notación para referenciar estos cálculos utiliza el superíndice para referirse a la capa a la que pertenece la variable y el subíndice para referirse al nodo dentro de esa capa. A pesar de que poco o nada va a ser referenciado por nodos en el desarrollo del proyecto se contempla necesario realizar la siguiente descripción para facilitar la comprensión del funcionamiento de la red neuronal. Para los cálculos realizados en la capa 1 de la red neuronal de la Figura 2-1 se resolvería tal que:

$$\begin{aligned} n_1^{[1]} &\rightarrow z_1^{[1]} = w_1^{[1]^T} x + b_1^{[1]} \ ; \ a_1^{[1]} = g\left(z_1^{[1]}\right) \\ n_2^{[1]} &\rightarrow z_2^{[1]} = w_2^{[1]^T} x + b_2^{[1]} \ ; \ a_1^{[1]} = g\left(z_2^{[1]}\right) \\ n_3^{[1]} &\rightarrow z_3^{[1]} = w_3^{[1]^T} x + b_3^{[1]} \ ; \ a_1^{[1]} = g\left(z_3^{[1]}\right) \\ n_4^{[1]} &\rightarrow z_4^{[1]} = w_4^{[1]^T} x + b_4^{[1]} \ ; \ a_1^{[1]} = g\left(z_3^{[1]}\right) \end{aligned} \tag{2-6}$$

Gracias al uso de la técnica de vectorización, se elimina la necesidad de realizar estos cálculos de manera iterativa, neurona por neurona, con cada capa del modelo, o expresado de forma práctica, se eliminan bucles 'for' en el código, haciéndolo mucho más eficiente y ahorrando mucho tiempo de computación. La vectorización permite realizar todos los cálculos vectoriales de cada neurona de una sola vez utilizando matrices formadas por estos vectores. La vectorización es una de las características más importantes que aporta la biblioteca de NumPy, y se basa en realizar todas esas operaciones utilizando bibliotecas compiladas en C, que ofrecen un mayor rendimiento. Con esta técnica el procesamiento se mejora varios órdenes de magnitud, haciendo el código unas 100-300 veces más rápido.

La vectorización permite escribir estas operaciones de forma matricial y se expresaría tal que:

$$Z^{[1]} = W^{[1]^T} X + b^{[1]} \tag{2-7}$$

$$A^{[1]} = g(Z^{[1]}) \tag{2-8}$$

Otra técnica que se utiliza en la ecuación (2-7) es broadcasting. Esta técnica permite al autor del código mucha flexibilidad a la hora de escribir las ecuaciones, permitiendo, en este caso en particular, sumar una matriz de tamaño (5,4) con una matriz de tamaño (5,1) y obtener un resultado coherente sin errores o warnings. El broadcasting funciona de forma que si uno de los ejes coincide en tamaño y el otro eje tiene valor 1 replica la matriz (5,1) hasta tener una (5,4). De forma general, el broadcasting funciona tal que:

$$\begin{aligned} (m,n) + (m,1) &\rightarrow (m,n) + (m,n) \\ (m,n) + (1,n) &\rightarrow (m,n) + (m,n) \end{aligned} \tag{2-9}$$

$$(m, 1) + (1,1) \rightarrow (m, 1) + (m, 1)$$
$$(1, n) + (1,1) \rightarrow (1, n) + (1, n)$$

Pudiendo sustituir la suma por la resta, la multiplicación o la división obteniendo los mismos resultados.

La técnica de broadcasting es muy útil y aporta mucha flexibilidad, pero también tiene su lado peligroso ya que puede llevar a errores difíciles de encontrar debido a que siempre se obtendrá un resultado, aunque no sea el correcto si se utiliza de forma equivocada.

Además de entender las dimensiones de los pesos es importante entender las dimensiones de los datos de entrada. Una vez ha quedado claro que tanto la red como los datos de entrada serán matrices hay que saber a qué hace referencia cada dimensión de la matriz de los datos de entrada. Esta matriz viene definida por:

$$X = (n_x, m) \tag{2-10}$$

Donde m será el número de ejemplos individuales del conjunto de datos y n_x el número de datos diferentes que posea cada ejemplo. Así, un conjunto de 1000 imágenes de 16 bits tendrá 1000 ejemplos individuales y cada ejemplo tendrá 16 datos. Esta n_x será la que defina el tamaño de la capa de entrada.

Ahora que se tiene un conocimiento general de cómo funciona la red neuronal y qué variables están implicadas en el procesamiento de los datos, se procede a explicar en detalle cada parte del modelo y cuál es su función concreta en este.

2.5.2 Inicialización de pesos

Como se comenta en anteriores apartados, los pesos son lo que dan vida a la red y su transformación y corrección permiten el aprendizaje del modelo. Por este motivo, la inicialización de los pesos no puede ser un proceso trivial y debe prestársele mucha atención a la manera escogida para realizar este primer paso.

La inicialización de los pesos será lo primero que se calculará cuando se comience a entrenar un modelo, para ello solamente son necesarias dos variables, el número de capas y el número de neuronas de cada capa. Así estos quedarán definidos normalmente en una sola variable, que será una lista de valores numéricos. La longitud de la lista hará referencia al número de capas y el valor de cada elemento de la lista al número de neuronas de cada capa, correspondiente a la posición de este valor en la lista.

Así, como todas las enumeraciones que se hacen en Python, la lista comienza en el lugar 0. Siempre todos los elementos de Python, así como de otros lenguajes de programación, asignan el valor 0 al valor situado en la primera posición del elemento. Esto se corresponde muy bien con la forma de numerar las capas en una red neuronal, ya que la primera capa, la capa de entrada, se define como capa 0, y en la lista la posición 0 la ocupará el tamaño de la capa de entrada. Como ejemplo, para ilustrar la explicación, la lista correspondiente a las capas de la red neuronal de la Figura 2-1 sería:

Definición de las capas de la red neuronal de la Figura 2-1

```
Layers_dimensions = [X.shape[0], 5, 5, 1]
```

Como aclaración, dado que es el primer elemento que se introduce en código en el desarrollo del proyecto, se indicará que prácticamente la totalidad de las variables que se encuentren en el código están definidas en inglés. Dado que, como autor, realizo todo el estudio de programación en este idioma, se ha decidido continuar con él ya que el idioma utilizado para declarar variables no influye en su funcionamiento.

Como se puede observar, se definen numéricamente tanto las capas ocultas como la capa de salida. La capa de salida se definirá según sea la necesidad del resultado que queremos obtener de la red. Si es un resultado binario, se definirá con una sola neurona. Si, en cambio, se trata de un clasificador se definirán tantas neuronas como clases a diferenciar. La definición numérica de las capas ocultas queda a selección del autor, teniendo que modificarlas y ajustarlas según se obtengan mejores resultados. Tanto el número de capas, como el número de neuronas de cada capa son definidos como hiperparámetros y será objetivo del autor ajustarlos y afinarlos según convenga.

Se puede observar que en la capa 0 no se utiliza un número sino una función del módulo NumPy para definir su tamaño. Para ser completamente correcto, en este caso se utiliza la propiedad shape que posee la matriz X, la cual es equivalente a utilizar la función shape() sobre la matriz X. El uso de la propiedad en lugar de la función añade ligereza y soltura a la hora de escribir el código. Añadimos entre corchetes el eje del que queremos obtener el valor. Por definición del propio lenguaje, el eje 0 será el que correspondiente al número de filas y el eje 1, al de las columnas, de nuevo haciendo referencia a que el primer valor de un conjunto se numere comenzando en el 0.

Ahora que ya se conocen las dimensiones de nuestra red, se procede a inicializar los pesos. Se inicializan de dos formas diferentes según se refiera la variable W o a la variable b. Para la primera, se crea una matriz de números aleatorios siguiendo una distribución normal estándar. Para la segunda, se crea una matriz de ceros.

$$W = \begin{bmatrix} a_{11} & \dots & a_{1j} \\ \dots & \dots & \dots \\ a_{i1} & \dots & a_{ij} \end{bmatrix}$$
$$b = \begin{bmatrix} 0 \\ \dots \\ 0 \end{bmatrix} \quad (2\text{-}11)$$

Siendo *a* un número aleatorio e *i*, *j* las dimensiones de la matriz W.

Como se deben crear varios pesos correspondientes a cada capa, se crea una función encargada de este proceso que alberga un bucle que facilita la creación de estas diferentes variables. Esta función queda definida en el código siguiente:

Inicialización de los pesos

```
def init_params(Layers_dimensions):

    np.random.seed()
    parameters = {}
    L = len(layers_dims) - 1

    for l in range(L):

        parameters["W" + str(l+1)] =
            np.random.randn(layers_dims[l+1], layers_dims[l]) * 0.001

        parameters["b" + str(l+1)] = np.zeros((layers_dims[l+1], 1))

    return parameters
```

En este código se pueden apreciar diferentes partes. La primera es la definición de una función que se hace con el parámetro *def*, continuado por el nombre de la función y entre paréntesis las variables sobre las que se aplica la función.

El primer elemento de la función es *np.random.seed()* el cual selecciona de manera aleatoria una semilla que será la utilizada para generar números aleatorios. Esto tiene que ver con el problema de generación de números realmente aleatorios en lenguajes de programación, un tema demasiado extenso que no se verá en el desarrollo de este proyecto. Para resumir, es una forma de utilizar diferentes aleatoriedades cada vez que se ejecuta, aumentando la aleatoriedad programada hacia una aleatoriedad más real.

Seguidamente se crea el diccionario *parameters*. Los diccionarios en Python, son conjuntos de elementos dobles que se componen de un nombre y de un valor, así, entre otras cosas, puede hacerse referencia a una variable en concreto dentro de un grupo de datos sin necesidad de saber su posición dentro del objeto.

La variable *L* se define como el número de capas del modelo y se resta *1* correspondiente a la capa de entrada,

que como se indicaba anteriormente no corresponde a una capa de cálculo.

A continuación, se abre un bucle *for* en el que se itera sobre el número de capas, y se definen en el diccionario *parameters*, los nombres de las variables con *["W" + str(l+1)]* creando así, en la primera iteración, la variable *W1*, que hace referencia a la variable $W^{[1]}$. Se asigna a esta variable una matriz de números aleatorios de dimensiones $(n^{[l+1]}, n^{[l]})$, tal y como se definía en la ecuación 2-3. Y, de la misma forma, se crea una variable $b^{[1]}$ y se asigna una matriz de ceros con las dimensiones $(n^{[l+1]}, 1)$, tal y como se definía en la ecuación 2-3. Se añade también un multiplicador de 0.001 a la matriz W para que los números generados sean pequeños y permitan un procesamiento más ágil y una convergencia más rápida hacia un resultado correcto en los pasos siguientes.

Este factor de multiplicación viene a tratar un problema que enfrentan las redes neuronales profundas que se llama *Vanishing/Exploding Gradients*, o traducido Gradientes que Desaparecen/Explotan. Este problema trata acerca de lo que ocurre, en redes neuronales muy profundas, cuando los valores de los pesos están por encima de 1 provocan cálculos exponenciales dando resultados que tienden al infinito, mientras que, si los pesos son menores, entonces, tienden a 0.

$$si\ W^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} \rightarrow \hat{y} = W^{L} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} X \approx 1.5^{L}\, X$$
$$si\ W^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \rightarrow \hat{y} = W^{L} \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}^{L-1} X \approx 0.5^{L}\, X \qquad (2\text{-}12)$$

Esto muestra el problema de entrenar redes muy profundas viendo cómo en el caso de $w > 1$ aumentará exponencialmente mientras que en $w < 1$ decrecerá de la misma manera. De forma análoga ocurre con los valores negativos.

Hay una solución parcial a este problema que pasa por la manera en cómo se inicializan los pesos. Para ello se han diseñado nuevas funciones que permiten que los pesos de una red neuronal profunda se ajusten al tamaño de las neuronas de la capa a la que pertenecen con el fin de optimizar los recursos y obtener resultados mejores. Este tema se trata en los trabajos de Yoshua Bengio y Xavier Glorot [16] con gran extensión y muchos investigadores han aportado diferentes algoritmos que mejoran este proceso. Un artículo muy interesante que recoge todos estos métodos y analiza el porqué de su funcionamiento es el realizado por S. K. Kumar [17].

La solución aplicada varía según la función de activación utilizada en las capas de la red neuronal. En el caso de este proyecto se ha empleado la He Initialization [18]. Esta inicialización se aplica a la función de activación ReLU que se explicará en el siguiente apartado.

La He Initialization se basa en aplicar un factor de multiplicación tal que:

$$\sqrt{\frac{2}{n^{[l-1]}}} \qquad (2\text{-}13)$$

Esto provoca que la inicialización de los pesos de forma aleatoria tenga una varianza:

$$v^2 = \frac{2}{N} \qquad (2\text{-}14)$$

De forma que la función de inicialización de los pesos quedará programada tal que:

Inicalización de los pesos mediante He Initialization

```
def init_params_he(layers_dims):

    np.random.seed()
    parameters = {}
    L = len(layers_dims) - 1
```

```
for l in range(L):

    parameters["W" + str(l+1)] =
                np.random.randn(layers_dims[l+1], layers_dims[l])
                                    * np.sqrt(2 / layers_dims[l])

    parameters["b" + str(l+1)] = np.zeros((layers_dims[l+1], 1))

return parameters
```

2.5.3 Propagación hacia delante o Forward propagation

Una vez se tienen inicializados los pesos ya se puede iniciar el entrenamiento de la red. Como se comentaba en anteriores apartados el primer paso es propagar hacia delante los datos utilizando para ello la estructura de la red y los pesos que la forman.

Cada neurona será la encargada de llevar a cabo dos operaciones: una función lineal y una función de activación.

En la primera capa se utilizarán los datos de los ejemplos utilizados para entrenar la red neuronal y se le aplicará una función lineal tal que:

$$Z^{[1]} = {W^{[1]}}^T X + b^{[1]} \tag{2-15}$$

La operación ${W^{[1]}}^T X$ es un producto de matrices, por lo que se transpone la matriz correspondiente al peso para lograr las dimensiones correctas. A su vez, esto provoca que la matriz resultante tenga las dimensiones necesarias para la siguiente capa.

A continuación, a la variable $Z^{[1]}$ se le aplica una función de activación tal que:

$$A^{[1]} = g\left(Z^{[1]}\right) \tag{2-16}$$

Se le llama función de activación a la función no lineal aplicada al resultado de la ecuación (2-14) y el nombre viene del símil biológico con las neuronas del cerebro humano, que se dice que cuando reciben información y envían una señal a la siguiente neurona esa neurona se ha activado.

Esta función de activación es clave en el funcionamiento de las redes neuronales y es lo que las caracteriza ya que, si en vez de tener estas funciones simplemente se aplicasen las funciones lineales, un resultado de la capa *l* no sería más que una función lineal del resultado de las capas anteriores, como se muestra a continuación.

Si definimos la función de activación tal que:

$$g\left(Z^{[l]}\right) = Z^{[l]} \tag{2-17}$$

Implicaría que:

$$\begin{aligned} A^{[1]} &= Z^{[1]} = {W^{[1]}}^T X + b^{[1]} \\ A^{[2]} &= Z^{[2]} = {W^{[2]}}^T A^{[1]} + b^{[2]} \end{aligned} \tag{2-18}$$

Y quedaría algo tal que:

$$\begin{aligned} A^{[2]} &= W^{[2]}\left(W^{[1]} X + b^{[1]}\right) + b^{[2]} \\ A^{[2]} &= \left(W^{[2]} W^{[1]}\right) X + \left(W^{[2]} b^{[1]} + b^{[2]}\right) \\ A^{[2]} &= W' X + b' \end{aligned} \tag{2-19}$$

El resultado es una función lineal, por lo que pierde el sentido tener capas ocultas.

En cuanto a las funciones de activación utilizadas en redes neuronales hay diversas y con diferentes propósitos. Por ejemplo, la más utilizada en un principio para clasificación binaria era la función sigmoide y posteriormente también se ha utilizado la tangente hiperbólica [19].

En este proyecto las funciones utilizadas para las capas ocultas y la capa de salida serán diferentes por la necesidad propia del modelo. La función utilizada en las capas ocultas será la ReLU, o Rectified Linear Unit, y para la capa de salida utilizaremos la Softmax classifier.

2.5.3.1 Función de activación ReLU

Esta función se ha vuelto la más popular y la más utilizada en modelos de inteligencia artificial de redes neuronales. Esto se debe a sus buenos resultados y su simpleza, tanto al aplicarla como en su coste computacional. Se puede valorar su utilidad en los trabajos de A. F. Agarap [20].

La función ReLU quedaría definida tal que:

$$g(z) = \max(0, z) \tag{2-20}$$

Básicamente, coge el resultado de la primera operación de la neurona, z, y aplica una comparación, si $z > 0$ devuelve el valor z y si $z < 0$ devuelve un 0.

Pese a su simpleza se ha descubierto como la función de activación más efectiva en términos de resultados y rapidez de entrenamiento. Y esta será la que se aplique a las capas ocultas del proyecto.

Su implementación en código sería:

ReLU

```
def relu(x):

    s = np.maximum(0, x)

    return s
```

2.5.3.2 Función de activación Softmax classifier

Para la capa de salida se ha decidido utilizar la Softmax classifier [21] dado que es una función que permite realizar una clasificación del resultado en varios estados. El número de estados del clasificador viene dado por el número de neuronas que tenga la última capa.

El resultado de esta función de activación es una matriz con una columna y un número de filas correspondiente al número de estados. Como el proyecto se realiza con aprendizaje supervisado, es el programador el que proporciona el resultado objetivo para su comparación con el resultado obtenido del modelo. Los resultados que se utilizan para entrenar esta red neuronal deberán ser traducidos a matrices one-hot, que son matrices que tienen un elemento de valor 1 y el resto de valores en 0.

Por ejemplo, en un modelo que tuviese 3 estados, se deben traducir estos estados tal que:

$$\begin{gathered} Estado = 1 \rightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \\ Estado = 2 \rightarrow \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \\ Estado = 3 \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \end{gathered} \tag{2-21}$$

Quedando, una vez traducidos los resultados, una matriz de dimensiones (3, m), siendo *m* el número de ejemplos.

El objetivo de este clasificador es tener un 1 en el estado objetivo y un 0 en el resto de estados. Para ello se toma el valor obtenido en Z en la última capa y se ponderan los resultados de forma que la suma de los elementos sea 1. Una vez ponderados, se toma el elemento mayor, el más cercano a 1, como el resultado proporcionado por la red. Este proceso se realiza de la siguiente manera:

$$t_i = e^{z_i^{[L]}}$$
$$a_i^{[L]} = \frac{t_i}{\sum_i t_i} \quad (2\text{-}22)$$

Este proceso, escrito como matriz queda:

$$A^{[L]} = \frac{e^{Z^{[L]}}}{\sum_i e^{Z^{[L]}}} \quad (2\text{-}23)$$

De esta forma obtenemos una red que permite clasificar en diferentes estados, lo cual será completamente necesario para poder desarrollar redes neuronales que distingan más allá de clasificaciones binarias blanco/negro, arriba/abajo, presencia/no presencia.

Su implementación en código se realizaría de la siguiente manera:

Softmax Classifier

```
def softmax(x):

    expo = np.exp(x)
    exp_sum = np.sum(expo, axis=0)
    s = expo / exp_sum

    return s
```

2.5.4 Función de coste

La función de coste será el cálculo por el cuál mediremos cuánto ha errado el modelo con respecto a los datos reales proporcionados en el entrenamiento y utilizar esta medición para optimizar los parámetros pertenecientes a la red, los pesos.

Esta función vendrá expresada a partir del cálculo de las pérdidas. En un clasificador binario, se definirá como la probabilidad de que y=1 dado un ejemplo x. Lo que se define como:

$$\hat{y} = P(y = 1|x) \quad (2\text{-}24)$$

Si se plantea:

$$si\ y = 1 : P(y|x) = \hat{y}$$
$$si\ y = 1 : P(y|x) = 1 - \hat{y} \quad (2\text{-}25)$$

Quedará tal que:

$$P(y|x) = \hat{y}^y (1 - \hat{y})^{1-y} \quad (2\text{-}26)$$

Por lo que:

$$\log P(y|x) = log\hat{y}^{y}(1-\hat{y})^{1-y}$$
$$\log P(y|x) = y\ log\hat{y} + (1-y)\log(1-\hat{y}) \quad (2\text{-}27)$$
$$\log P(y|x) = -\mathcal{L}(\hat{y}, y)$$

El símbolo negativo es debido a que cuando se está entrenando un algoritmo se quiere que las probabilidades sean grandes por lo que se necesita minimizar las pérdidas. Y minimizar las pérdidas implica maximizar el logaritmo de la probabilidad.

Desde aquí se calcula la función de coste, de manera que, tomando que los ejemplos de entrenamiento están distribuidos idéntica e independientemente, las probabilidades de todo el set será la multiplicación de las probabilidades individuales.

$$p(estados\ en\ el\ set\ entrenamiento) = \prod_{i=1}^{m} p\left(y^{(i)} \middle| x^{(i)}\right)$$
$$\log p(estados\ en\ el\ set\ entrenamiento) = \log \prod_{i=1}^{m} p\left(y^{(i)} \middle| x^{(i)}\right) \quad (2\text{-}28)$$
$$\log p(estados\ en\ el\ set\ entrenamiento) = \sum_{i=1}^{m} \log p\left(y^{(i)} \middle| x^{(i)}\right) = \sum_{i=1}^{m} -\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

En estadística existe un principio llamado Principio de máxima verosimilitud que consiste en escoger lo parámetros que maximizan el resultado. En este caso:

$$\log\ \mathrm{p}(\ldots) \quad (2\text{-}29)$$

O, lo que es lo mismo, que maximizan:

$$-\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) \quad (2\text{-}30)$$

Lo que justifica la función de coste del modelo. Como queremos minimizar el coste en vez de maximizarlo cambiamos el signo de la función:

$$\mathcal{J}(w,b) = \sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) \quad (2\text{-}31)$$

Y, por conveniencia, para que las cantidades estén en la escala correcta se añade 1/m como factor de escala:

$$\mathcal{J}(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) \quad (2\text{-}32)$$

A esta función de coste se la denomina Cross-entropy loss function [25].

Sin embargo, el clasificador utilizado en este proyecto es multi clase por lo que habrá que realizar los siguientes ajustes al proceso de calcular el coste para implementarlo en el modelo [26].

La función de pérdida quedaría definida tal que:

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{n} y_j \log \hat{y}_j \tag{2-33}$$

Siendo *n* el número de estados, o clases, posibles del clasificador.

La función de coste permanece igual definida ya que lo que se modifica es el cálculo de las pérdidas en cada ejemplo.

Su implementación en código será tal que:

Función de coste de Softmax classifier

```
def compute_cost(A4, Y, eps=0.0000000001):

    m = Y.shape[1]

    loss = - np.sum(np.multiply(Y, np.log(A4+eps)), axis=0)

    cost = 1/m * np.sum(loss)

    return cost
```

Los argumentos de entrada son la salida de la red neuronal y los valores reales con los que se contrasta. Además, se añade una épsilon para evitar que el cálculo del logaritmo tienda a infinito si el valor de A4 es 0.

2.5.5 Propagación hacia atrás o Backward propagation

La propagación hacia atrás es la segunda parte del proceso de cálculo del modelo y es donde se implementa el entrenamiento y se actualizan los valores para obtener un resultado mejor en la siguiente iteración. Este proceso se basa en realizar derivadas de los valores y propagarlos hacia atrás siguiendo la estructura de la red neuronal.

La técnica base empleada para esta propagación hacia atrás se denomina *Gradient descent*, o descenso del gradiente, dado que los cálculos implicados serán gradientes de la función de coste. Estos gradientes se expresan tal que:

$$\frac{\partial \mathcal{J}(w,b)}{\partial var} \tag{2-34}$$

Siendo *var* la variable sobre la que se quiere calcular el gradiente.

La propagación se ve en la forma de obtener estos gradientes, ya que, poniendo de ejemplo la última capa de una red neuronal, sería tal que:

$$\begin{gathered}
\frac{\partial \mathcal{J}(w,b)}{\partial A^{[L]}} = \frac{\partial \mathcal{J}(w,b)}{\partial A^{[L]}} \\
\frac{\partial \mathcal{J}(w,b)}{\partial Z^{[L]}} = \frac{\partial \mathcal{J}(w,b)}{\partial A^{[L]}} * \frac{\partial A^{[L]}}{\partial Z^{[L]}} \\
\frac{\partial \mathcal{J}(w,b)}{\partial W^{[L]}} = \frac{1}{m} * \frac{\partial \mathcal{J}(w,b)}{\partial Z^{[L]}} * \frac{\partial Z^{[L]}}{\partial W^{[L]}} \\
\frac{\partial \mathcal{J}(w,b)}{\partial b^{[L]}} = \frac{1}{m} * \frac{\partial \mathcal{J}(w,b)}{\partial Z^{[L]}} * \frac{\partial Z^{[L]}}{\partial b^{[L]}}
\end{gathered} \tag{2-35}$$

Por convención, a lo largo del código siempre se encontrarán estos gradientes referenciados según la variable

sobre la que se hace la derivada y esto ha trascendido también para denominar así a estas variables en todos los demás ámbitos, de cara a hacer más legible tanto el código como el texto. Por lo que, de ahora en adelante, si escribiera las variables que aparecen arriba (2-35) se verían tal que:

$$\begin{gathered}\frac{\partial \mathcal{J}(w,b)}{\partial A^{[L]}} \rightarrow dA^{[L]} \\ \frac{\partial \mathcal{J}(w,b)}{\partial Z^{[L]}} \rightarrow dZ^{[L]} \\ \frac{\partial \mathcal{J}(w,b)}{\partial W^{[L]}} \rightarrow dW^{[L]} \\ \frac{\partial \mathcal{J}(w,b)}{\partial b^{[L]}} \rightarrow db^{[L]}\end{gathered} \tag{2-36}$$

El primer elemento a derivar, por lo tanto, es la última capa y esta derivada depende de la función de coste empleada. Dado que se está utilizando la *cross-entropy loss function* (2-32) para calcular el coste del modelo el valor de la primera derivada será:

$$dA^{[L]} = \frac{y \log a + (1-y) \log (1-a)}{da} = -\frac{y}{a} + \frac{1-y}{1-a} \tag{2-37}$$

Aun así, esta primera derivada no se suele incluir en el código ya que por sí misma tiene poco valor y solamente se utiliza para calcular la siguiente derivada $dZ^{[L]}$. Por lo que se suele escribir directamente esta derivada siguiendo las reglas de (2-35) y para ello habrá que saber cómo se derivan las funciones de activación utilizadas.

Para dar una vista general del bloque de propagación hacia atrás, antes de introducir las derivadas de estas funciones, se muestra el siguiente ejemplo de las derivadas asociadas a una capa *l* de una red neuronal:

$$\begin{gathered}dA^{[l]} = W^{[l+1]^T} \cdot dZ^{[l+1]} \\ dZ^{[l]} = dA^{[l]} * g'\left(Z^{[l]}\right) \\ dW^{[l]} = \frac{1}{m} * dZ^{[l]} \cdot A^{[l-1]^T} \\ db^{[l]} = \frac{1}{m} * \sum_{j=1}^{n} dZ_j^{[l]}\end{gathered} \tag{2-38}$$

En el elemento $dZ^{[l]}$ es donde se encuentra la derivada de la función de activación que variará según la función escogida en cada capa. El resto de elementos se mantendrán definidos igual que aquí independientemente de la capa en la que se encuentren, a excepción de $dA^{[L]}$ que está definido en (2-37). Por lo que se pasará a definir las derivadas de las funciones de activación mencionadas previamente.

2.5.5.1 Derivada de la función de activación ReLU

Así como la definición de esta función su derivada también es muy sencilla.

$$g'(z) = \begin{cases} 0 & si\ z < 0 \\ 1 & si\ z > 0 \\ indef & si\ z = 0 \end{cases} \tag{2-39}$$

Este último no es computable por lo que en el código deberá sustituirse y dejarse definido en uno de los otros valores, tal que:

$$g'(z) = \begin{cases} 0 & si\ z \leq 0 \\ 1 & si\ z > 0 \end{cases} \tag{2-40}$$

En código es posible implementarlo definiendo una función que realice esta comparativa y obtener una matriz resultante. Sin embargo, se considera más práctico y efectivo aprovechar la simpleza de escritura para incluirlo directamente en el cálculo de $dZ^{[l]}$. Esto se realiza de la siguiente forma:

Como implementar derivada de ReLU

```
dZ3 = np.multiply(dA3, np.int64(A3 > 0))
```

Lo que se realiza es una multiplicación entre $dA^{[l]}$y una matriz que se crea mediante una comparativa de si el valor $A^{[l]}$ es mayor que cero. Si esta condición se cumple introduce un *True*, mientras que si no se cumple introduce un *False*. Cuando se realizan cálculos matemáticos con este tipo de matrices, Python las traduce automáticamente en 1 y 0 correspondientemente, facilitando así el cálculo de la derivada de esta función de activación.

2.5.5.2 Derivada de la función de activación Softmax classifier

Para no entrar en mucha definición matemática acerca de la derivación de la fusión Softmax classifier, la cual se puede encontrar en otros artículos [22] [23], se indicará directamente el cálculo de la derivada $dZ^{[L]}$como se mencionaba anteriormente.

Esta derivada tendrá el valor de:

$$dZ^{[L]} = dA^{[L]} * g'\left(Z^{[L]}\right) = \hat{y} - y \quad (2\text{-}41)$$

Por lo que, a la hora de implementarlo en código, será tan simple como escribir:

Como implementar derivada de ReLU

```
dZ4 = A4 - Y
```

Como se puede observar, tanto la derivada de la ReLU como esta derivada se han podido implementar directamente sobre el bloque de código de la propagación hacia atrás, haciendo este proceso más eficiente computacionalmente.

2.5.6 Actualización de los pesos

Una vez que se ha realizado la inicialización de los pesos, la propagación hacia delante, el cálculo del coste y la propagación hacia atrás, solamente queda el último paso para hacer efectivo el aprendizaje de la red neuronal: actualizar los valores de los pesos.

Es un proceso simple en el que se ven implicados los pesos, los gradientes calculados en la propagación hacia atrás y el *learning rate*, definido con la variable α. Como se mencionó previamente, este *learning rate* es un hiperparámetro que deberá ajustarse empíricamente y es en este paso donde se ve su importancia.

Los pesos se verán modificados tal que:

$$\begin{gathered} W^{[l]} = W^{[l]} - \alpha * dW^{[l]} \\ b^{[l]} = b^{[l]} - \alpha * db^{[l]} \end{gathered} \quad (2\text{-}42)$$

Cuanto más grande sea el *learning rate* más cambiarán los valores de los pesos de la red neuronal. Y si bien esto puede ser muy bueno al principio, cuando los parámetros aún no están entrenados, puede ser muy perjudicial a la hora de encontrar un óptimo por la gran variabilidad y poca estabilidad que les da a estos parámetros. Por el otro lado, escoger un *learning rate* muy pequeño puede llevar a un tiempo de aprendizaje excesivamente largo.

Otro de los peligros de escoger un *learning rate* pequeño son los falsos óptimos. Si el modelo llega a un falso optimo y su *learning rate* no es lo suficientemente grande como para salir de él habrá finalizado el aprendizaje sin haber optimizado sus parámetros. Otra posibilidad es que llegue a un punto de silla, donde el gradiente es cero en ambos parámetros por lo que el aprendizaje es nulo.

Todas estas opciones deben ser contempladas cuando se está diseñando el modelo y es necesario realizar diferentes pruebas con diferentes *learning rates* para lograr afinar todo lo posible el valor de este. Y cada vez que se modifique el modelo será necesario volver a afinar este hiperparámetro.

Con este último paso se habría finalizado la construcción de una red neuronal de aprendizaje profundo básica, un programa construido de esta manera sería capaz de aprender y lograr realizar predicciones bastante acertadas. Sin embargo, el objetivo de este proyecto no solo es construir una red neuronal si no también intentar desarrollar un modelo que iguale o mejore los resultados obtenidos por los métodos estadísticos tradicionales. Y para ello se implementarán una serie de métodos de optimización que harán que el modelo desarrollado sea más efectivo, rápido y fiable.

Estos métodos varían desde tratamiento de los datos de entrada hasta modificaciones completas de partes del propio modelo descritas en la sección anterior. Todas han tenido éxito mejorando el funcionamiento del programa, aunque aquí se limitará su explicación a su funcionamiento teórico y su implementación en código, mientras que sus resultados podrán ser apreciados en el siguiente punto, correspondiente al desarrollo práctico.

Su orden en este proyecto es el orden en el que se han ido implementando según se desarrollaba el modelo y se descubrían las necesidades a cubrir.

Estos métodos serán los que se verán a continuación.

2.6 Normalización

El primero de estos métodos no es exclusivo de las redes neuronales sino un método ampliamente utilizado en estadística. Se trata de normalizar los datos para hacer que la distribución de los valores de cada variable sea mucho más compacta [29].

Este método permite que el aprendizaje sea más rápido y el modelo pueda hallar un camino más directo hacia el punto óptimo en el proceso del descenso del gradiente.

El método se aplica a los datos previamente a introducirlos en el modelo para su aprendizaje y se basa en dos sencillas operaciones: restar la media del conjunto de los valores y dividirlo por su varianza.

Si se define la media y la varianza tal que:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$$
$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x_i^2 \tag{2-43}$$

Se aplicaría:

$$x_{norm} = \frac{x - \mu}{\sigma} \tag{2-44}$$

El objetivo de esta operación es distribuir los valores alrededor de cero respetando su proporcionalidad para que sigan teniendo el mismo significado dentro del conjunto. Por supuesto, esto debe aplicarse individualmente a cada variable.

Este proceso se puede apreciar mejor en el siguiente gráfico.

Si tenemos una distribución de valores tal que:

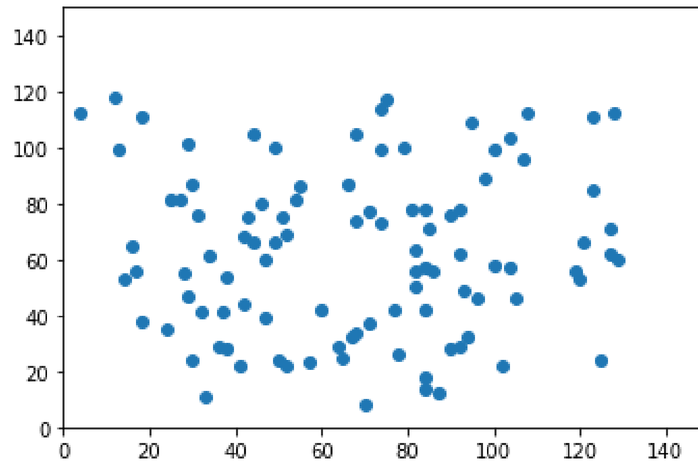

Figura 2-3. Distribución inicial de datos

Una vez aplicada la normalización a estos datos, quedaría tal que:

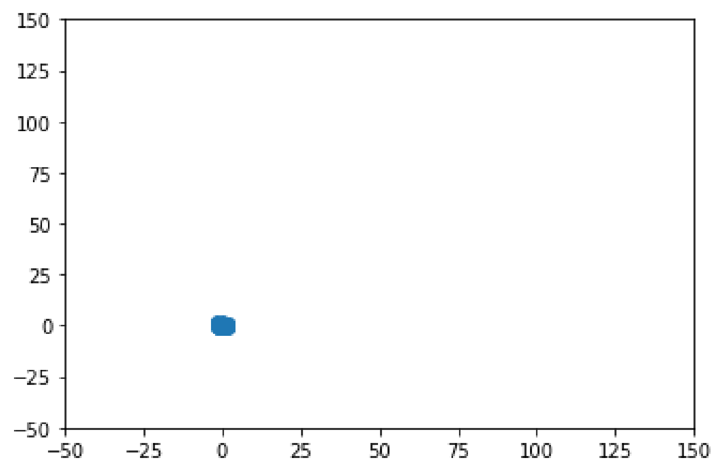

Figura 2-4. Distribución de datos normalizados

Se puede comprobar que todos los datos que había previamente distribuidos a lo largo de ambos ejes están ahora concentrados en el origen.

Si rehacemos la escala del gráfico se puede comprobar que la proporción se ha mantenido:

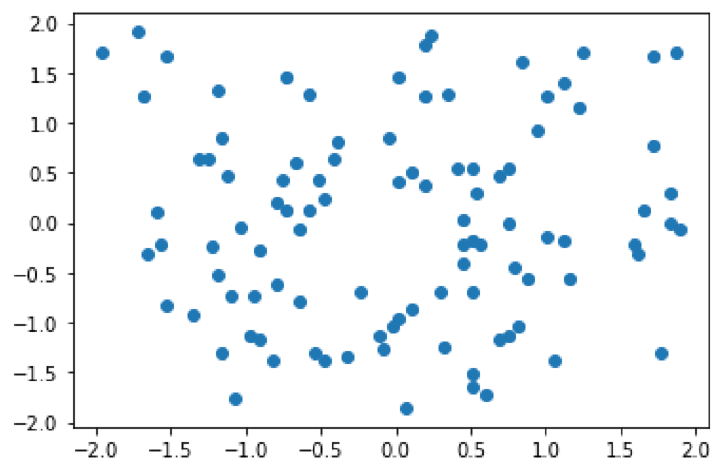

Figura 2-5. Distribución de datos normalizados con eje a escala

Esta normalización de los datos permite una convergencia mucho más rápida hacia el óptimo y disminuye el coste computacional del proceso. Entendiendo el proceso de optimización y de descenso del gradiente como

un camino a lo largo del gráfico se podría observar la diferencia en los siguientes gráficos.

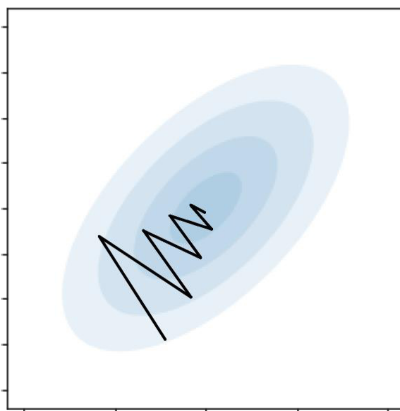

Figura 2-6. Camino de optimización datos no normalizados.

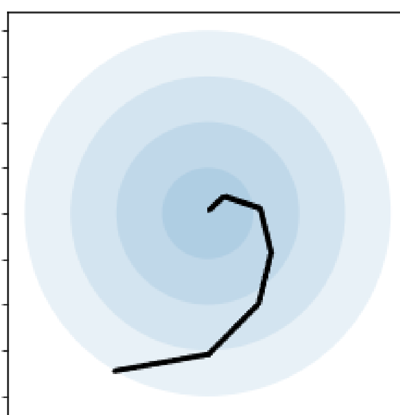

Figura 2-7. Camino de optimización datos normalizados.

Este cambio en el camino a seguir durante el proceso de aprendizaje permite establecer un *learning rate* más alto y así reducir el tiempo de aprendizaje del algoritmo.

2.7 Regularización

Tras haber preparado los datos e iniciado el proceso de entrenamiento siempre se deben separar los datos en dos o tres grupos, según la cantidad de datos y el objetivo del proyecto. En este caso se ha optado por separarlos en dos grupos dado que la cantidad de datos era limitada y conseguir así maximizar la eficacia de estos.

Estos dos conjuntos en los que se dividirán los datos son: el conjunto de entrenamiento y el conjunto de prueba. Como sus propios nombres indican, el primero se utilizará para entrenar la red, durante decenas o cientos de miles de iteraciones, mientras que el otro se utilizará para comprobar su eficacia en datos nuevos sobre los que no ha sido entrenado. Se debe cumplir que ambos conjuntos deben tener la misma distribución o, en el peor caso, una distribución muy similar, esto significa que deben estar recogidos de la misma población, en temporalidades similares y en condiciones lo más iguales posibles.

Tras haber realizado el proceso de entrenamiento a un modelo y pasar al momento de ponerlo a prueba, pueden medirse 2 variables para evaluar el resultado: el error y la varianza.

El error se define como la diferencia entre el porcentaje de acierto objetivo y el porcentaje de acierto conseguido sobre el conjunto de datos de entrenamiento. Si este error es elevado se dice que el modelo sufre de *underfitting* o subajuste, y necesita un periodo de entrenamiento más largo o desarrollar una red neuronal más grande.

La varianza se define como la diferencia entre la precisión del modelo sobre el conjunto de datos de entrenamiento y su precisión sobre el conjunto de datos de prueba. Si esta diferencia es elevada se dice que el

modelo sufre de *overfitting* o sobreajuste. Esto es debido a que el modelo ha mejorado mucho su eficacia para predecir el resultado de los ejemplos del conjunto de entrenamiento, pero ha perdido objetividad con el resto de casos, volviéndose inútil para su aplicación práctica. Cuando esto ocurre puede ser solucionado añadiendo más cantidad de datos al conjunto de entrenamiento o aplicando métodos de regularización.

A continuación, se verán los diferentes métodos de regularización que se han aplicado durante el desarrollo del modelo [30] [31].

2.7.1 L2 regularization

Este método de regularización [32] se basa en la aplicación del normalizador de Frobenius, definido como la raíz cuadrada de la suma de los cuadrados absolutos de los elementos de la matriz sobre la que se aplica. Esto quedaría reflejado de la siguiente manera.

Siendo A una matriz de dimensiones (m, n):

$$\|A\|_F = \sqrt{\sum_i^m \sum_j^n |a_{ij}|} \tag{2-45}$$

En este caso se aplica a las matrices de los pesos, aunque usualmente se suele omitir su aplicación sobre el parámetro *b* y aquí tampoco se incluirá. Y puede definirse tal que:

$$\|W\|_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} {W_{ij}^{[l]}}^2 \tag{2-46}$$

Los índices de los sumatorios atienden a las dimensiones de W tal y como se recoge en (2-3).

Este elemento regulador se implementaría en nuestro modelo en dos diferentes apartados: en el cálculo del coste y en la propagación hacia atrás.

En lo que respecta al cálculo de la función de coste se añadiría al final multiplicado por *lambda*, que será un parámetro de regularización y un nuevo hiperparámetro a afinar por el autor, y un factor de escala, el mismo que se utiliza en la función de pérdida.

$$\mathcal{J}(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2}\frac{1}{m}\sum_{l=1}^{L} \|W^{[l]}\|_F^2 \tag{2-47}$$

Esto, implementado en código, quedaría de la siguiente manera:

Función de coste con regularización L2

```
def compute_cost_with_L2reg(A4, Y, parameters, lambd):

    m = Y.shape[1]

    W1 = parameters["W1"]
    W2 = parameters["W2"]
    W3 = parameters["W3"]
    W4 = parameters["W4"]

    cross_entr_cost = compute_cost(A4, Y)

    L2reg_cost = (1/m) * (lambd/2) * (np.sum(np.square(W1)) +
np.sum(np.square(W2)) + np.sum(np.square(W3)) + np.sum(np.square(W4)))
```

```
cost = cross_entr_cost + L2reg_cost

return cost
```

El segundo elemento donde se ve reflejada esta regularización es, como se mencionaba, en la propagación hacia atrás, dado que se deriva la propia función de coste donde se implementa este factor regulador. Como es función de W solamente el cálculo de la derivada de este parámetro incluirá la regularización.

El cálculo del gradiente de este elemento quedará tal que:

$$\frac{\partial \mathcal{J}(w,b)}{\partial W^{[l]}} = dW^{[l]} = \frac{1}{m} * dZ^{[l]} \cdot A^{[l-1]^T} + \frac{\lambda}{m} W^{[l]} \tag{2-48}$$

Esto afecta directamente al proceso de actualización de los pesos que resulta en:

$$\begin{gathered} W^{[l]} = W^{[l]} - \alpha * dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha * \left(\frac{1}{m} * dZ^{[l]} \cdot A^{[l-1]^T} + \frac{\lambda}{m} W^{[l]}\right) \\ W^{[l]} = W^{[l]} - \frac{\alpha\lambda}{m} W^{[l]} - \alpha\left(\frac{1}{m} * dZ^{[l]} \cdot A^{[l-1]^T}\right) \\ W^{[l]} = \left(1 - \frac{\alpha\lambda}{m}\right) W^{[l]} - \alpha\left(\frac{1}{m} * dZ^{[l]} \cdot A^{[l-1]^T}\right) \end{gathered} \tag{2-49}$$

A este factor que multiplica $W^{[l]}$ se le denomina *Weight decay*, o decaimiento del peso [33], causado por la regularización.

Como se extrae de las ecuaciones (2-47) y (2-48), si se escogiese una *lamba* igual a cero se anularía la normalización. Mientras que si se escogiese una *lambda* muy alta los pesos tenderían a cero, como se puede ver en la ecuación (2-49).

2.7.2 Dropout regularization

Las redes neuronales de aprendizaje profundo tienden siempre a sobre ajustar cuando se entrenan con una cantidad de datos limitada o sesgada. Para evitar este comportamiento se han utilizado conjuntos de redes neuronales entrenados por separado que una vez unidos mejoran sus predicciones. Este buen resultado tiene la contraposición que entrenar varios modelos es muy costoso computacionalmente y conlleva mucho tiempo.

El método *dropout* [34] [35] viene a sustituir esta solución aportando lo mejor de tener varias estructuras neuronales diferentes y entrenar un solo modelo a la vez. Se basa en una mecánica muy simple: cada neurona tiene una probabilidad, definida por el autor, de apagarse en cada iteración. Esto provoca que cada iteración de entrenamiento de la red neuronal se realiza sobre una estructura diferente a la anterior.

Para visualizar el método, si se tuviera una red neuronal tal que:

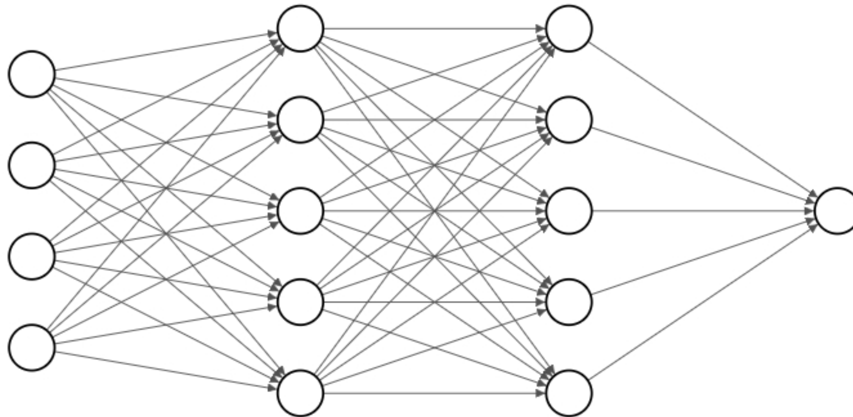

Figura 2-8. Red neuronal de tres capas

Y se aplicara *dropout* a esta red neuronal con una probabilidad de apagado del 20% a la primera capa y un 40% a la segunda capa, una posibilidad de resultado sería:

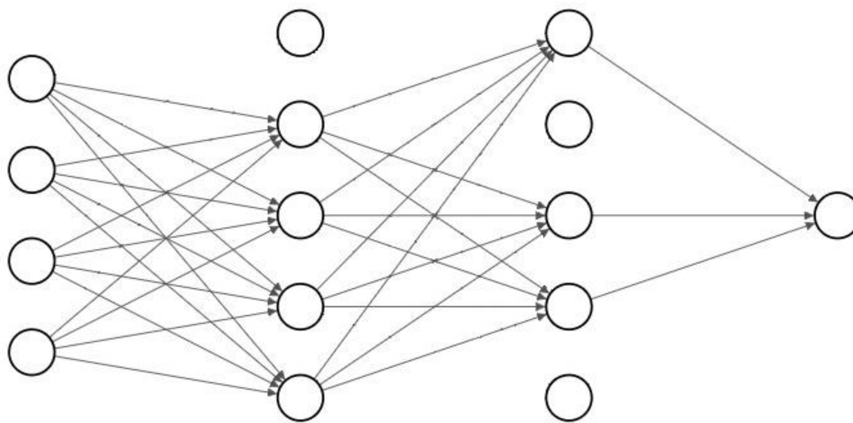

Figura 2-9. Red neuronal aplicando *dropout*

Sobre esta nueva estructura se realizaría la propagación hacia delante, el cálculo de la función de coste y la propagación hacia atrás y por último la actualización de los pesos. Y cuando se iniciase de nuevo el ciclo esta estructura sería diferente, variando así con cada iteración.

Como es lógico, este método solamente se aplica a las capas ocultas, ya que no se puede aplicar a la capa de entrada ya que se estaría sesgando la información ni a la capa de salida ya que se estaría modificando el resultado de la red.

Este método modifica dos partes del proceso que son la propagación hacia delante y la propagación hacia detrás. Además, habrá que aplicar el *dropout* invertido para que la función de coste sea real, esto también permite que a la hora de utilizar el modelo sobre el conjunto de prueba no haya que utilizar *dropout* y se pueda utilizar la red completa.

La aplicación de *dropout* es puramente práctica por lo que se describirá su aplicación directamente en código. En la propagación hacia delante se implementará a la salida de la capa. A continuación, se muestra un ejemplo de cómo se implementaría en una capa individual.

Propagación hacia delante con *dropout*

```
Z1 = np.dot(W1, X) + b1
A1 = relu(Z1)

D1 = np.random.rand(A1.shape[0], A1.shape[1])
D1 = (D1 < keep_prob).astype(int)
```

```
A1 = np.multiply(A1, D1)
A1 = A1 / keep_prob
```

Se puede observar que se crea una matriz D de valores aleatorios y esta se convierte en una matriz de unos y ceros haciendo una comparación con la variable *keep_prob* que es la probabilidad de que la neurona esté encendida. Posteriormente, se aplica esta máscara multiplicando esta variable por la matriz A para apagar los resultados de las neuronas correspondiente de la matriz D. En último lugar, se aplica el *dropout* invertido dividiendo los valores de la matriz A por la probabilidad establecida.

De la misma forma, se muestra cómo se aplicaría a una capa durante la propagación hacia atrás.

Propagación hacia atrás con *dropout*

```
dA1 = np.dot(W2.T, dZ2)
dA1 = np.multiply(dA1, D1)
dA1 = dA1 / keep_prob
dZ1 = np.multiply(dA1, np.int64(A1 > 0))
```

Se puede observar, que primero se calcula el gradiente respecto a dA, posteriormente se aplica la máscara correspondiente a la matriz D para apagar las neuronas que no entraron en juego en la propagación hacia delante para mantener su consistencia. Se vuelve a aplicar *dropout* invertido y se continúa con la propagación hacia atrás hacia la siguiente capa.

El motivo del funcionamiento de este método es simple: como la neurona encargada de realizar el cálculo que le corresponde depende de la información que le llega y no sabe qué neurona se va a "apagar" no puede confiar en una sola de ellas y se ve obligada a repartir el peso equitativamente entre todas ellas. Esto produce que los pesos se hagan más pequeños en un fenómeno que se llama *Shrink weights*. Tiene un efecto similar al L2 regularization.

Se puede aplicar diferentes probabilidades a cada capa según el tamaño de esta en la red. Si se aplica una probabilidad igual a 1 no se aplica regularización. Esto solamente se utiliza en matrices pequeñas donde interesa conservar todos los datos.

Dropout se utiliza mucho en visión por ordenador dado que se tienen muchos datos de cada ejemplo y se generan matrices de dimensiones muy grandes.

Dropout es el algoritmo de regularización más efectivo y con mejores resultados en su aplicación.

2.7.3 Early stopping

Sin ser un método en sí mismo, el *early stopping* [36], o paro temprano, es una manera de evitar que la red neuronal produzca *overfiting*. Se basa en detener el aprendizaje antes de que se produzca el sobreajuste al conjunto de entrenamiento.

El problema de este método tiene que ver con el principio de ortogonalización. Esto viene a ser, realizar una tarea por cada vez, y es que el uso de *early stopping* mezcla el objetivo de optimizar la función de coste y el objetivo de evitar el *overfitting* y no consigue ningún resultado óptimo en ninguno de los dos.

Aun así, sin ser un método que se vaya a aplicar específicamente en el proyecto es una directriz con la que se realizan todos los ciclos de entrenamiento de cada modelo y se podrá observar como a lo largo del desarrollo de este se utiliza esta técnica para ajustar los resultados obtenidos hacia el objetivo buscado.

2.8 Optimización

En la sección 2.4.5 y 2.4.6 se empleaba la técnica del *gradient descent*, o descenso del gradiente, para implementar el aprendizaje minimizando la función de coste y, posteriormente, actualizar los parámetros de la red. La eficacia de este algoritmo será la que determine el tiempo necesario de entrenamiento para que la red

neuronal sea capaz de aprender y realizar predicciones acertadas.

En esta sección se verán otras técnicas de aprendizaje que optimizarán el tiempo requerido por el modelo para conseguir buenos resultados, pudiendo ser una diferencia entre tardar un par de días a nada más que un par de horas [37].

Pese a que, en este proyecto, los tiempos de entrenamiento no se dilatarán tanto por modelo, dado que no se tiene un conjunto de datos tan extenso, es completamente crucial su empleo dado que se estarán entrenando diferentes modelos a la vez por lo que el tiempo que requiere en conjunto sí es significativo.

2.8.1 Gradient descent with momentum

Además de la mejora de tiempo, la técnica de *gradient descent* sufre de otro problema particular que es su completa dependencia del gradiente medido en cada punto individual en el que se encuentra.

Dado que el aprendizaje se realiza según el gradiente medido en el punto del espacio en cada iteración que se realiza sobre el conjunto de datos, esto puede llevar a oscilar repetidas veces sobre espacios que presenten grandes cantidades de curvatura o que sus gradientes sean ruidosos. También puede llevar a detenerse sobre puntos planos en el espacio en el que el gradiente sea cero e impida así el aprendizaje.

Por este motivo, se realiza una modificación de esta técnica para imbuir de cierta inercia a este proceso de optimización, reduciendo así el impacto individual de cada punto frente a la corriente general de los pasos anteriores [24]. Justamente, el nombre de esta técnica viene de la analogía física de la inercia de un cuerpo a la hora de agregar nuevas aceleraciones a su movimiento.

Para ello se utilizará la media ponderada exponencialmente de los gradientes previos y se implementará en el cálculo del gradiente de cada variable. Esto se implementará creando dos nuevas variables, una para cada peso, sobre la que se realizará esta ponderación tal que:

$$v_{dW^{[l]}} = \beta * v_{dW^{[l]}} + (1 - \beta) * dW^{[l]}$$
$$v_{db^{[l]}} = \beta * v_{db^{[l]}} + (1 - \beta) * db^{[l]} \quad (2\text{-}50)$$

Donde β es un nuevo hiperparámetro a afinar por el autor.

El valor de β determinará cuanta importancia se da a los valores previos del gradiente sobre el valor actual del gradiente individual. Si $\beta = 0$ se implementará *gradient descent sin momentum*.

Este nuevo algoritmo modificará el camino de aprendizaje suavizando sus oscilaciones y redireccionando el movimiento hacia el punto óptimo, alcanzando este de manera más rápida. En la Figura 2-10 se puede observar la diferencia de camino negro, que sería el *gradient descent*, con respecto al camino naranja, que sería el *gradient descent with momentum*.

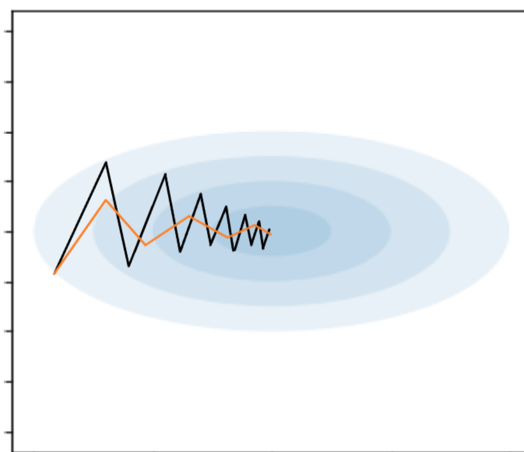

Figura 2-10. *Gradient descent with momentum*

Una vez implementadas estas nuevas dos variables se modificaría la actualización de los pesos de la red de manera que:

$$W^{[l]} = W^{[l]} - \alpha * v_{dW^{[l]}}$$
$$b^{[l]} = b^{[l]} - \alpha * v_{db^{[l]}} \quad (2\text{-}51)$$

Como se indicaba anteriormente, β es un nuevo hiperparámetro que habrá que afinar, aunque por norma general se suele aplicar un valor de 0.9, dado que produce buenos resultados de forma general. Aun así, en el desarrollo de este proyecto se podrá observar cómo finalmente sí se decide afinar, probando diferentes valores de cara a optimizar el aprendizaje.

Para su aplicación en código deberá crearse una nueva función para inicializar los valores de las nuevas variables y modificar la función de actualización de pesos. Ambas se pueden ver a continuación.

Inicialización variables *GD with momentum*

```
def init_momentum_var(parameters):

    L = len(parameters) // 2
    v = {}
    for l in range(L):

        v["dW" + str(l+1)] = np.zeros((parameters["W" +
                                                  str(l+1)].shape))

        v["db" + str(l+1)] = np.zeros((parameters["b" +
                                                  str(l+1)].shape))

    return v
```

Actualización de pesos con *GD with momentum*

```
def upd_params_with_momentum(parameters, gradients, v, beta,
                                                   learning_rate):

    L = len(parameters) // 2

    for l in range(L):

        v["dW" + str(l+1)] = beta * v["dW" + str(l+1)] + (1-beta) *
                                      gradients["dW" + str(l+1)]

        v["db" + str(l+1)] = beta * v["db" + str(l+1)] + (1-beta) *
                                      gradients["db" + str(l+1)]

        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                                 learning_rate * v["dW" + str(l+1)]

        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
                                 learning_rate * v["db" + str(l+1)]

    return parameters, v
```

2.8.2 RMSprop – Root mean square

Otro algoritmo utilizado en la optimización del aprendizaje es el *RMSprop*, o media cuadrática. Como su propio nombre indica, su implementación se basa en la aplicación de la técnica estadística para obtener la raíz cuadrada del valor medio de los cuadrados de las medidas de una magnitud.

Si bien esta técnica no se implementará directamente en el desarrollo del proyecto su comprensión es clave para desarrollar el siguiente método de optimización por lo que se considera indispensable incluirlo.

Para su implementación también es necesario crear dos nuevas variables que se definirán como:

$$s_{dW^{[l]}} = \beta_2 * s_{dW^{[l]}} + (1 - \beta_2) * {dW^{[l]}}^2$$
$$s_{db^{[l]}} = \beta_2 * s_{db^{[l]}} + (1 - \beta_2) * {db^{[l]}}^2 \quad (2\text{-}52)$$

Siendo β_2 un nuevo hiperparámetro, que se define con el subíndice 2 para diferenciarlo de la β utilizada en el cálculo del *gradient descent with momentum*.

Los cuadrados de las variables $dW^{[l]}$ y $db^{[l]}$ son multiplicaciones elemento a elemento, no multiplicación de matrices.

Estas nuevas variables se implementan en la actualización de los pesos tal que:

$$W^{[l]} = W^{[l]} - \alpha * \frac{dW^{[l]}}{\sqrt{s_{dW^{[l]}}} + \varepsilon}$$
$$b^{[l]} = b^{[l]} - \alpha * \frac{db^{[l]}}{\sqrt{s_{db^{[l]}}} + \varepsilon} \quad (2\text{-}53)$$

Siendo $\varepsilon = 10^{-8}$.

Se añade una variable ε con el objetivo que no se realicen divisiones entre cero.

Con esta técnica se realizaría una optimización similar a la observada en la Figura 2-10, aumentando la eficacia del aprendizaje y reduciendo el tiempo de entrenamiento.

2.8.3 Adam

Este algoritmo de optimización funciona unificando los dos algoritmos anteriores, *momentum* y *RMSprop*, y aplicándolos a la vez.

Es un caso raro dentro del aprendizaje profundo ya que es una técnica que ha mostrado muy buenos resultados en un rango muy amplio de campos de aplicación. Su nombre, Adam, viene derivado de *adaptative moment estimation*, y fue propuesto por Diederik Kingma y Jimmy Ba en 2015 [39].

Este método requiere de declarar las variables (2-50) y (2-52) y, a su vez, declarar cuatro variables nuevas que quedan definidas tal que:

$$v_{dW^{[l]}}^{corrected} = \frac{v_{dW^{[l]}}}{1 - \beta_1^t}$$
$$v_{db^{[l]}}^{corrected} = \frac{v_{db^{[l]}}}{1 - \beta_1^t}$$
$$s_{dW^{[l]}}^{corrected} = \frac{s_{dW^{[l]}}}{1 - \beta_2^t}$$
$$s_{db^{[l]}}^{corrected} = \frac{s_{db^{[l]}}}{1 - \beta_2^t} \quad (2\text{-}54)$$

Siendo *t* el número de la iteración en el proceso de aprendizaje.

Siendo β_1 y β_2 hiperparámetros a afinar por el autor.

Por convención se suelen utilizar los siguientes valores para estos hiperparámetros:

$$\beta_1 = 0.9$$
$$\beta_2 = 0.999 \quad (2\text{-}55)$$

Estas nuevas variables se aplicarán a la actualización de los pesos tal que:

$$W^{[l]} = W^{[l]} - \alpha * \frac{v_{dW^{[l]}}^{corrected}}{\sqrt{s_{dW^{[l]}}^{corrected}} + \varepsilon}$$
$$b^{[l]} = b^{[l]} - \alpha * \frac{v_{db^{[l]}}^{corrected}}{\sqrt{s_{db^{[l]}}^{corrected}} + \varepsilon} \quad (2\text{-}56)$$

Siendo $\varepsilon = 10^{-8}$.

Este método de optimización se aplicará en código de la siguiente manera:

Inicialización variables Adam

```
def init_adam(parameters):

    L = len(parameters) // 2

    v = {}
    s = {}

    for l in range(L):

        v["dW" + str(l+1)] = np.zeros((parameters["W" +
                                                       str(l+1)].shape))
        v["db" + str(l+1)] = np.zeros((parameters["b" +
                                                       str(l+1)].shape))

        s["dW" + str(l+1)] = np.zeros((parameters["W" +
                                                       str(l+1)].shape))
        s["db" + str(l+1)] = np.zeros((parameters["b" +
                                                       str(l+1)].shape))

    return v, s
```

Actualización de pesos con Adam

```
def upd_params_with_adam(parameters, gradients, v, s, t,
                         learning_rate, beta1, beta2, epsilon = 1e-8):

    L = len(parameters) // 2

    v_corr = {}
    s_corr = {}

    for l in range(L):
```

```
        v["dW" + str(l+1)] = beta1 * v["dW" + str(l+1)] + (1-beta1) *
                                                   gradients["dW" + str(l+1)]

        v["db" + str(l+1)] = beta1 * v["db" + str(l+1)] + (1-beta1) *
                                                   gradients["db" + str(l+1)]


        s["dW" + str(l+1)] = beta2 * s["dW" + str(l+1)] + (1-beta2) *
  np.multiply(gradients["dW" + str(l+1)], gradients["dW" + str(l+1)])

        s["db" + str(l+1)] = beta2 * s["db" + str(l+1)] + (1-beta2) *
  np.multiply(gradients["db" + str(l+1)], gradients["db" + str(l+1)])


        v_corr["dW" + str(l+1)] = v["dW" + str(l+1)] / (1 -
                                                                (beta1**t))

v_corr["db" + str(l+1)] = v["db" + str(l+1)] / (1 -
                                                                (beta1**t))

        s_corr["dW" + str(l+1)] = s["dW" + str(l+1)] / (1 -
                                                                (beta2**t))
        s_corr["db" + str(l+1)] = s["db" + str(l+1)] / (1 -
                                                                (beta2**t))


        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                             learning_rate * (v_corr["dW" + str(l+1)] /
                          (np.sqrt(s_corr["dW" + str(l+1)] + epsilon)))

        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
                             learning_rate * (v_corr["db" + str(l+1)] /
                          (np.sqrt(s_corr["db" + str(l+1)] + epsilon)))


    return parameters, v, s
```

3 RESULTADOS

Construir una IA avanzada es como lanzar un cohete. El primer reto es maximizar la aceleración, pero una vez que empieza a coger velocidad, también hay que centrarse en la dirección.

- Jaan Tallinn -

En este capítulo se llevará a cabo el desarrollo del modelo empleando los algoritmos, métodos, técnicas y funciones vistos en el capítulo anterior. Todo esto se desarrollará en orden cronológico según se ha ido avanzando en su construcción.

En un primer punto se verá el tratamiento de los datos originales, qué incluyen, en qué formato vienen y cómo los transformaremos para llegar a tenerlos en el formato necesario para su utilización.

Una vez se tengan los datos de forma apropiada se procederá al desarrollo de los modelos. Se mostrarán cronológicamente los modelos creados y los resultados obtenidos de cada uno de ellos, así como las mejoras que presenta respecto al modelo anterior. Junto con la presentación de cada modelo se incluirán las observaciones del autor y un comentario de cara a facilitar el entendimiento de los datos mostrados.

3.1 Estableciendo el objetivo

La red neuronal de aprendizaje profundo que se desarrollará se entrenará mediante aprendizaje supervisado. Esto quiere decir que además de proporcionarle los datos sobre los que realizará el entrenamiento también se debe proporcionar el resultado esperado de estos datos.

Para ello habrá que definir cuál será el campo objetivo que se quiere predecir con el modelo. Después de revisar los datos se decide que el campo objetivo que se quiere predecir es el campo Estatus del paciente.

Este campo puede tomar 6 valores diferentes como se puede ver en la Tabla 3-1.

La red neuronal que se desarrollará en este proyecto intentará inferir patrones en los datos proporcionados para predecir correctamente el estado futuro de un receptor que se somete a un trasplante en base a los datos previos a este procedimiento quirúrgico.

Todo proyecto de redes neuronales debe apuntar a conseguir una tasa de error igual al *Bayes error rate* [28], que es el error mínimo posible para un clasificador estadístico y es análoga al error irreducible. Dada la dificultad de determinar este error, se suele comenzar por hacer objetivo el error que conseguiría una persona experta en el campo de aplicación del proyecto, siempre con el objetivo de terminar superándolo. Por ello en este proyecto se definirá como error objetivo el que tiene un médico ordenando un trasplante de un órgano determinado sobre un paciente determinado. Este error, sin embargo, tiene dos posibles medidas dado que el ambiente en el que se trabaja es el de la salud y hay muchas variables implicadas en la determinación del éxito o el fracaso de un procedimiento.

Si se analizan los datos de manera directa, de los 5078 casos, terminan fallando 1196, es decir un 23.5% dando así una tasa de acierto del 76.5%.

No obstante, médicamente se considera un éxito si el órgano trasplantado funciona durante 2 años, aunque falle posteriormente a esta fecha. Leyendo de esta manera los datos tenemos que de 5078 casos fallan 484, lo que es un error del 9.53%, o sea una tasa de acierto del 90.47%.

Dado que no se tendrán en cuenta las temporalidades a la hora de procesar los datos, se establecerá el error objetivo de 23.5%.

Así mismo, se tienen los métodos estadísticos tradicionales, con los que se comparará el modelo desarrollado, que tienen un acierto entre un 60% y un 70%. Siendo muy pocos los que realmente se acercan a esta tasa del 70%.

Por lo que, habiendo establecido ya el objetivo al que se apunta en el proceso de aprendizaje de la red neuronal de aprendizaje profundo que se desarrollará en este proyecto, se pasa a relatar los pasos, avances y resultados que se han obtenido.

Tabla 3-1. Estatus del paciente

Valor	Significado
1	Trasplante funcionante en seguimiento
2	Perdido para seguimiento con trasplante funcionante
3	Fracaso y vuelta a diálisis (paciente vivo)
4	Vuelta a diálisis y muerte posterior a 90 días de inicio de la técnica
5	Vuelta a diálisis y muerte en los primeros 90 días
6	Muerte con trasplante funcionante

3.2 Tratamiento de los datos

Dado que la red neuronal profunda que se construirá trabaja con lenguaje Python deberán tratarse estos datos para que su formato sea el apropiado a la hora de introducirlos en la red. Sabiendo cuál es el objetivo que se desea alcanzar se deberá ver cuál es el punto de partida.

Los datos obtenidos, cedidos por la Coordinación Autonómica de Trasplantes de Andalucía, están almacenados en un fichero Excel. Estos datos, por su carácter privado y para respetar el acuerdo alcanzado para su cesión y tratamiento no se publicarán en este proyecto.

Se puede saber, sin embargo, las características de estos datos, así como los diferentes campos de los que se guarda registro en este proceso de trasplante de riñón.

3.2.1 Contenido

En las siguientes tablas se visualizan todos los campos de los que se tiene registro en el archivo Excel. Se incluirán tres columnas donde se dejará registro del nombre del campo, el significado de este campo y el rango de valores que toma en el conjunto de datos.

Se dividirán estos campos en tres tablas diferentes: relativas al donante, relativas al receptor y relativas al proceso de trasplante de riñón.

Tabla 3-2. Campos relativos al donante

Nombre del campo	Significado del campo
Donante	Código identificador del donante
Edad	Edad del donante
Sexo	Sexo del donante
Tipo de donante	Estado del donante en el momento de la donación del órgano
Grupo sanguíneo	Grupo sanguíneo del donante
COD	Causa de la muerte
Peso	Peso del donante
Talla	Altura del donante
BMI	Índice de masa corporal del donante
Fx_defunción	Fecha defunción del donante
Creatinina	Nivel de creatinina del donante
HTA	Presencia hipertensión arterial en el donante
Diabetes	Presencia diabetes en el donante
Hepatitis	Presencia hepatitis en el donante
Compat. A1	Compatibilidad HLA-A1 del donante
Compat. B1	Compatibilidad HLA-B1 del donante
Compat. DR1	Compatibilidad HLA-DR1 del donante
Compat. A2	Compatibilidad HLA-A2 del donante
Compat. B2	Compatibilidad HLA-B2 del donante
Compat. DR2	Compatibilidad HLA-DR2 del donante
VHC	Presencia virus de la hepatitis C en el donante
VHB	Presencia virus de la hepatitis B en el donante
CMV	Presencia citomegalovirus en el donante
KDRI	Índice de riesgo del donante de riñón
KDPI	Índice del perfil del donante de riñón

Tabla 3-3. Campos relativos al receptor

Nombre del campo	Significado del campo
Receptor	Código identificador del receptor
Edad	Edad del receptor
Sexo	Sexo del receptor
Fx_nacimiento	Fecha de nacimiento del receptor
Enf Renal Primaria	Enfermedad renal primaria del receptor
Diabetes	Presencia diabetes en el receptor
Hepatitis	Presencia hepatitis B en el receptor
Hepatitis C	Presencia hepatitis C en el receptor
VIH	Presencia VIH en el receptor
Cod. Tipo CMV	Código de presencia de virus similar a citomegalovirus
Cod. 1º TRS	Código primer trasplante
Fx inicio 1º TRS	Fecha de inicio del primer trasplante
Cod. TRS anterior Tx	Hepatitis en el donante

Fx inicio TRS anterior Tx	Compatibilidad HLA-A1
Num. Tx renal	Número de trasplante renal
Peso pre-tx	Peso del receptor previo al trasplante
Talla pre-tx	Altura del receptor previa al trasplante
Grupo sanguíneo	Grupo sanguíneo del receptor
Compat. A1	Compatibilidad HLA-A1 del receptor
Compat. B1	Compatibilidad HLA-B1 del receptor
Compat. DR1	Compatibilidad HLA-DR1 del receptor
Compat. A2	Compatibilidad HLA-A2 del receptor
Compat. B2	Compatibilidad HLA-B2 del receptor
Compat. DR2	Compatibilidad HLA-DR2 del receptor
Num. Transfusiones	Número de transfusiones realizadas
Anticuerpos citotóxicos pre-tx	Porcentaje de anticuerpos citotóxicos previo al trasplante
Otro Tx no renal (0)	Otro trasplante no renal
Fx otro Tx no renal (0)	Fecha de otro trasplante que no sea de tipo renal
Otro Tx no renal (1)	Otro trasplante que no sea de tipo renal
Fx otro Tx no renal (1)	Fecha de otro trasplante que no sea de tipo renal
Otro Tx no renal (2)	Otro trasplante que no sea de tipo renal
Fx otro Tx no renal (2)	Fecha de otro trasplante que no sea de tipo renal
Tx Previo Órgano Solido	Otro trasplante del receptor de órgano sólido
Años en diálisis	Años en diálisis del paciente
EPTS	Índice de supervivencia estimada después del trasplante ponderada sobre el conjunto de datos

Tabla 3-4. Campos relativos al trasplante

Nombre del campo	Significado del campo
Fx_Trasplante	Fecha en la que se realiza el trasplante
Num. Incomp. HLA-A	Número incompatibilidades HLA-A entre donante y receptor
Num. Incomp. HLA-B	Número incompatibilidades HLA-B entre donante y receptor
Num. Incomp. HLA-DR	Número incompatibilidades HLA-DR entre donante y receptor
Valor prueba cruzada lifotox.	Resultado prueba cruzada linfocitaria por citotoxicidad
Tiempo isquemia fría	Tiempo entre la muerte del donante y el procedimiento de trasplante
Riñón	Riñón trasplantado
¿Rechazo agudo 1º año?	Receptor sufre rechazo agudo durante el primer año
Disfunción inicial	Receptor sufre disfunción inicial
Fx fracaso Tx	Fecha fracaso del trasplante
Fx muerte Paciente	Fecha muerte del paciente (receptor)
Causa muerte	Causa de la muerte
Fx pérdida seguimiento	Fecha pérdida del seguimiento sobre el paciente
Fx siguiente Re-Tx al Tx tratado	Fecha de retrasplante (si aplica) al trasplante tratado
Fx estudio	Fecha en la que se realiza el estudio
Estatus del paciente	Estado actual del paciente del trasplante
SV injerto	Supervivencia en días del órgano funcionante tras el trasplante
Sv Paciente	Supervivencia en días del paciente tras el trasplante

3.2.2 Transformación a DataFrame

Dado que todos estos datos están almacenados en un fichero Excel la mejor manera de tratarlos es con la biblioteca Pandas [26]. Por lo que se comienza importando la biblioteca en nuestro archivo:

Importar la biblioteca Pandas

```
import pandas as pd
```

Se cargarán todos los datos en un objeto denominado DataFrame, que está dedicado a almacenar tablas de datos. Este primer paso se realizará tal que:

Carga de fichero

```
file = pd.read_excel('C:/Users/Equipo/Documents/TFM/Proyecto/Data/
                                             BD_TrasplanteRenal.xlsx')
```

Ahora que se tiene todo el archivo cargado en la variable *file* ya podemos utilizarlo.

Lo siguiente es decidir cuáles son los valores críticos representativos para reducir la cantidad de variables con las que trabajará el modelo y optimizar el uso de recursos. Estos campos han sido seleccionados por los profesionales sanitarios implicados en el proyecto y para ello se han basado tanto su experiencia y conocimientos como en estudios sobre la literatura clínica.

Finalmente, han sido seleccionadas 17 variables que se reflejan en la Tabla 3-5 diferenciadas según corresponda a Donante, Receptor o Trasplante.

Tabla 3-5. Variables críticas seleccionadas

Donante	Receptor	Trasplante
Edad	Edad	Num. Incomp. HLA-DR
Sexo	Sexo	Tiempo isquemia fría
Tipo de donante	Num. Tx renal	
Causa de la muerte	Peso pre-tx	
Peso	Talla pre-tx	
Talla	Años en diálisis	
Creatinina	EPTS	
KDRI		

Una vez seleccionados estos campos se definen en variables y se realiza una selección de estas columnas y se crea un nuevo DataFrame.

Creación DataFrame con campos seleccionados

```
list_col_don = ["Donante", "Edad_don", "Sexo_don", "Tipo_don", "COD",
"Peso_don", "Talla_don", "Creatinina_don", "KDRI"]

list_col_rx = ["Edad_rx", "Sexo_rx", "Num_Tx_renal", "Peso_pre_tx",
"Talla_pre_tx", "Años_dialisis", "EPTS"]

list_col_tx = ["Num_Incomp_HLA_DR", "Tiempo_isquemia_fria"]
list_col = list_col_don + list_col_rx + list_col_tx
```

```
tabla_trab = file.loc[:, list_col]
```

A continuación, se define la tabla de resultados sobre la que se va a entrenar la red neuronal de aprendizaje profundo. El campo sobre el que se evaluarán los resultados y para la que se entrenará el modelo predictivo es el campo *Estatus del paciente*, como se definió en la sección anterior. Por lo que seleccionamos este campo y creamos una tabla nueva.

Creación DataFrame con campo resultado

```
tabla_res = file.loc[:,["Donante", "Estatus_paciente"]]
```

En ambas tablas se incluye el campo Donante como código para mantener la referencia entre ellas y verificar que las transformaciones que sufren son idénticas y no se pierde la vinculación datos-resultado.

Ahora ya se tienen almacenados los datos de entrenamiento y los resultados en dos DataFrame separados.

Como se decía en el apartado 2.4.3.2, donde se hablaba de la función *Softmax classifier*, se establecía que el formato de los datos que requería esta función de activación era matrices *one-hot*. Por lo que se transformarán estos valores 1-6 a matrices de dimensiones (6, 1) con un 1 en el estado correspondiente y un 0 en el resto.

Para ello, primero se define las equivalencias y se almacenan en la variable *results*. Luego se realiza el mapeo de los datos y se guardan en una nueva columna denominada *EDP_Mat*.

Conversión de los valores de la tabla resultado a matrices

```
results = {
    1 : np.array([1, 0, 0, 0, 0, 0]),
    2 : np.array([0, 1, 0, 0, 0, 0]),
    3 : np.array([0, 0, 1, 0, 0, 0]),
    4 : np.array([0, 0, 0, 1, 0, 0]),
    5 : np.array([0, 0, 0, 0, 1, 0]),
    6 : np.array([0, 0, 0, 0, 0, 1])}

tabla_res["EDP_Mat"] = tabla_res["Estatus_paciente"].map(results)
```

A continuación, se vuelven a unir todos los datos, los datos de entrada y los datos de salida, en un nuevo DataFrame denominado *tabla_trabajo_XY*. Sobre este se procederá a realizar la transformación de los datos que tengan valores no legibles por el modelo o que tienen valores que deben ser reasignados. Estos son los valores desconocidos, que estén guardados en la tabla con diferentes códigos, o valores binarios que estén definidos con códigos y deberán ser transformados en 1 y 0. La siguiente tabla muestra las transformaciones que fueron necesarias y a continuación se presenta el código encargado de hacer la modificación.

Los valores desconocidos serán sustituidos por NaN, que significa *Not a Number*, que es el tipo de valor con el que se definen valores desconocidos en Python en la biblioteca NumPy.

Tabla 3-6. Modificación de valores

Valor original	Valor modificado
-1	NaN
-1.0	NaN
Desconocido	NaN
ME / DA	0 / 1

Modificación de valores

```
t_t_XY1 = tabla_trabajo_XY.replace(-1, np.nan)
t_t_XY2 = t_t_XY1.replace(-1.0, np.nan)
t_t_XY3 = t_t_XY2.replace("Desconocido", np.nan)
t_t_XY4 = t_t_XY3.replace(["ME", "DA"], [0, 1])
```

Por último, se crea un DataFrame a partir de esta última tabla *t_t_XY4*. Para establecer un original independiente a partir del que seguir trabajando

Creación DataFrame de punto de partida

```
T_T_XY = DataFrame(t_t_XY4)
```

Hasta aquí se tenía una tabla donde las filas eran cada caso individual y las columnas eran los campos de variables. A partir de aquí se separan definitivamente el conjunto de datos de entrada *X* y el conjunto de datos de salida, o el resultado, *Y* y se realiza la transposición de ambas para tener la disposición con la que se trabajará desde aquí.

Separación, transposición de datos

```
X = (T_T_XY.loc[:, list_col]).T
Y = (T_T_XY.loc[:, ["Donante", "EDP_Mat"]]).T
```

Se realiza una permutación de los datos para alterar su orden de manera que sea aleatorio. Una vez se comprueba que las permutaciones respetan el orden en ambos conjuntos de datos, por lo que se mantiene la relación entrada-resultado, se elimina definitivamente el campo *Donante* de ambos conjuntos dado que no es una variable significativa.

Permutación y eliminación de campo *Donante*

```
m = X.shape[1]
permut = list(np.random.permutation(m))

X_perm = X.loc[:, permut]
Y_perm = Y.loc[:, permut]

X_perm = X_perm.drop(["Donante"])
Y_perm = Y_perm.drop("Donante", axis=0)
```

Quedan definidas como los conjuntos de trabajo las variables *X_perm* y *Y_perm*. A continuación, habrá que definir cómo tratar con los *missing values*.

3.2.3 Missing values

Se denomina *missing values*, o valores faltantes, a esos valores que no están registrados dentro de una tabla de datos. Esto se debe a que en muchas ocasiones estos valores se borran, no se registran o se pierden en transcripciones.

Las redes neuronales son muy robustas en su funcionamiento cuando se enfrentan a *missing values* y no se ven afectadas en gran medida por su presencia. Sin embargo, estos valores han de cubrirse de alguna manera ya que, aun siendo robustas, necesitan recibir un valor numérico en esos campos y no puede introducirse los

valores NaN que se tienen en este punto del desarrollo.

Para ello se ha de realizar un estudio estadístico desde diferentes puntos para encontrar cuál es el valor correcto por el que se deben sustituir estos *missing values*. Se buscarán correlaciones con otros valores presentes en el conjunto de datos y se inferirá el valor más adecuado con respecto a ellos.

Para esta parte se utilizará la biblioteca Seaborn, que permitirá realizar todo tipo de gráficos permitiendo así un estudio más visual de este problema.

Lo primero que se hace es sacar un gráfico de calor que muestra la cantidad de *missing values* por campo y, a continuación, se incluye una tabla con los porcentajes.

Importación de biblioteca y creación de gráfico de calor inicial

```
import seaborn as sns

X_graf = X_perm.T
sns.heatmap(X_graf.isnull(), yticklabels=False, cbar=False,
                                                   cmap='viridis')
```

Esto devuelve el siguiente gráfico.

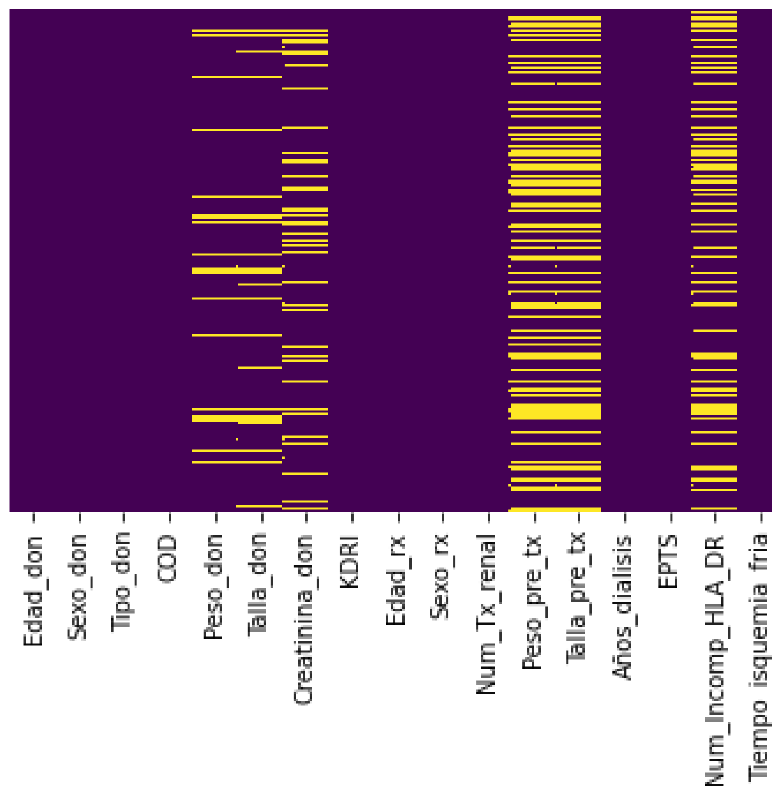

Figura 3-1. *Missing values*

Las líneas amarillas representan cada valor faltante. Solamente 6 campos de 17 presentan *missing values*. Pese a su apariencia, ningún campo sobrepasa el 32% de missing values. Los porcentajes de valores desconocidos se pueden apreciar en la Tabla 3-6.

Tabla 3-7. Campos con missing values y sus porcentajes

Campo	Porcentaje missing values
Peso_don	10,7813 %
Talla_don	12,7117 %
Creatinina_don	18,3573 %
Peso_pre_tx	31,0326 %
Talla_pre_tx	31,0326 %
Num_Incomp_HLA_DR	28,7561 %

Se tratarán estos campos en orden, estudiándolos para buscar los valores más adecuados a imputar en cada caso.

3.2.3.1 Peso del donante

El primer *missing value* con el que se tratará es el Peso del donante. En un primer intento de ahondar más en los datos se grafica el conjunto de datos separándolos por su sexo y por su edad, como se puede ver en las figuras siguientes.

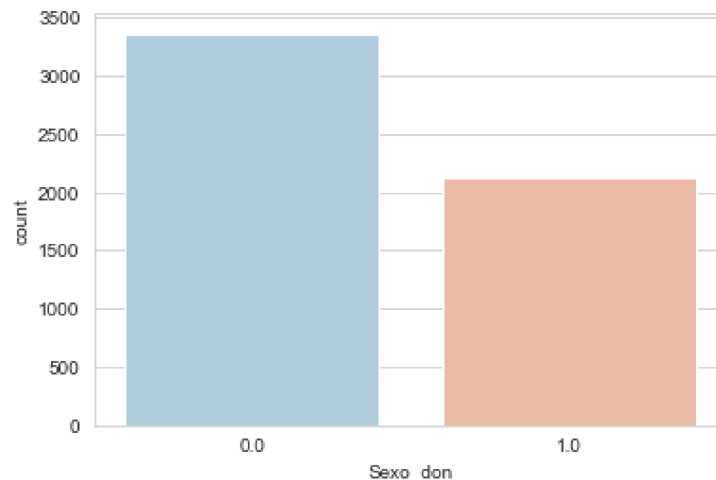

Figura 3-2.Conjunto de datos segregado según sexo donante

Siendo 0 los hombres y 1 las mujeres, se observa que hay mayor cantidad de órganos donados provenientes de hombres en el conjunto de datos por lo que puede ser un campo relevante a la hora de buscar correlaciones.

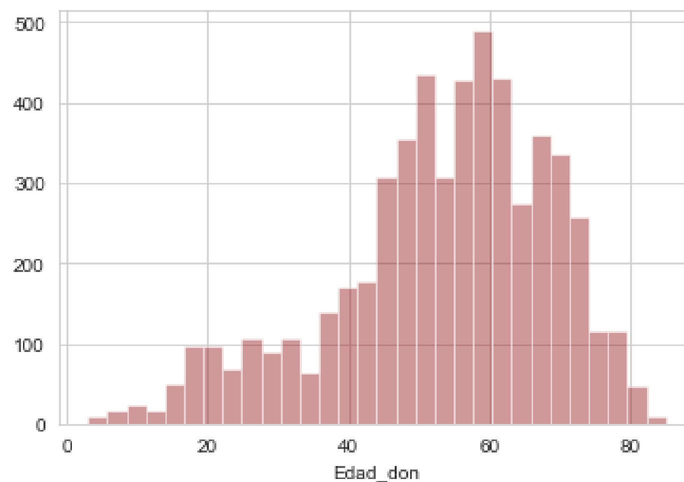

Figura 3-3. Conjunto de datos segregado según edad donante

La edad está muy distribuida y podría dar origen a mucho ruido en los datos por lo que se reserva para uso en

caso de necesidad.

Se decide realizar un gráfico que represente el peso del donante según su sexo y lo muestre como diagrama de cajas para poder apreciar los cuartiles en los que está repartido este campo.

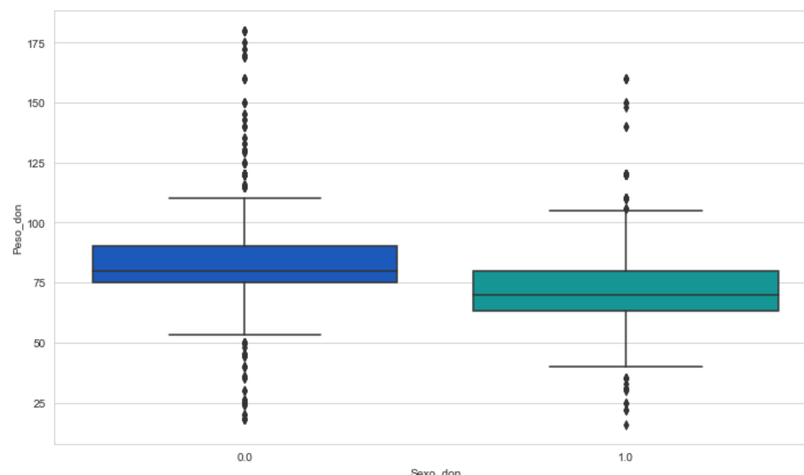

Figura 3-4. Peso del donante según su sexo

Como se puede observar, el rango entre los cuartiles 25 y 75 es bastante pequeño e, incluso teniendo valores atípicos en la serie, los valores máximo y mínimo están bastante acotados. Por ello se decide utilizar la mediana, cuartil 50, como valor de sustitución en este campo.

Para ello se calcula el valor exacto de cada sexo.

Cálculo de la mediana de *Peso_don* según *Sexo_don*

```
X_graf_h = X_graf[X_graf["Sexo_don"]==0.0]
X_graf_m = X_graf[X_graf["Sexo_don"]==1.0]

percentil_50_h = np.percentile(X_graf_h["Peso_don"].dropna(), 50,
                      axis=0, interpolation='linear', keepdims=False)

percentil_50_m = np.percentile(X_graf_m["Peso_don"].dropna(), 50,
                      axis=0, interpolation='linear', keepdims=False)
```

Este cálculo arroja los siguientes valores:

Tabla 3-8. Mediana de *Peso_don* según *Sexo_don*

Sexo_don	Peso_don
0	80
1	70

Se procede a imputar estos valores en el DataFrame y sustituir así los *missing values* del campo *Peso_don.*

Imputar los valores en el campo *Peso_don*

```
def imputar_peso_don(cols):
    Peso_don = cols[0]
    Sexo_don = cols[1]

    if pd.isnull(Peso_don):
```

```
        if Sexo_don == 0.0:
            return 80.0

        else:
            return 70.0
    else:
        return Peso_don

X_graf["Peso_don"] = X_graf[["Peso_don", "Sexo_don"]].apply(
                                             imputar_peso_don, axis=1)
```

Se comprueba que se han imputado correctamente los valores y ya no existen *missing values* en la columna Peso_don.

Comprobar imputación mediante gráfico de calor

```
sns.heatmap(X_graf.isnull(), yticklabels=False, cbar=False,
                                                      cmap='viridis')
```

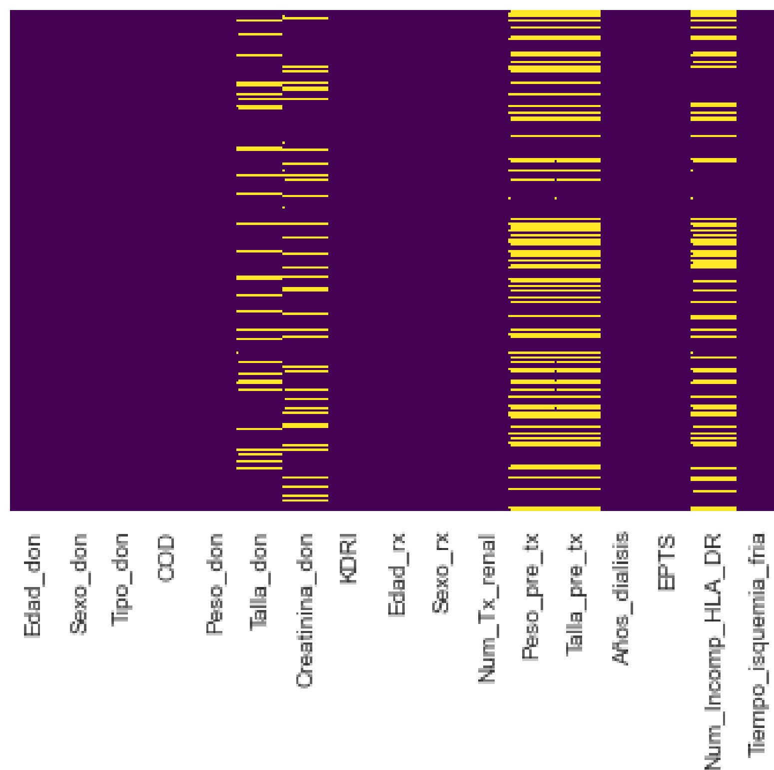

Figura 3-5. *Missing values* tras la 1ª imputación

A continuación, se realizará el mismo proceso de estudio e imputación de valores para los otros 5 campos que presentan *missing values*. Se presentará la información gráfica y se omitirá la misma explicación realizada en este punto a excepción de que haya algún punto reseñable a tratar. De igual forma, se omitirá el código relacionado con este proceso. El código completo relacionado con este campo y con los 5 campos siguientes puede ser consultado en el Anexo J.

3.2.3.2 Talla del donante

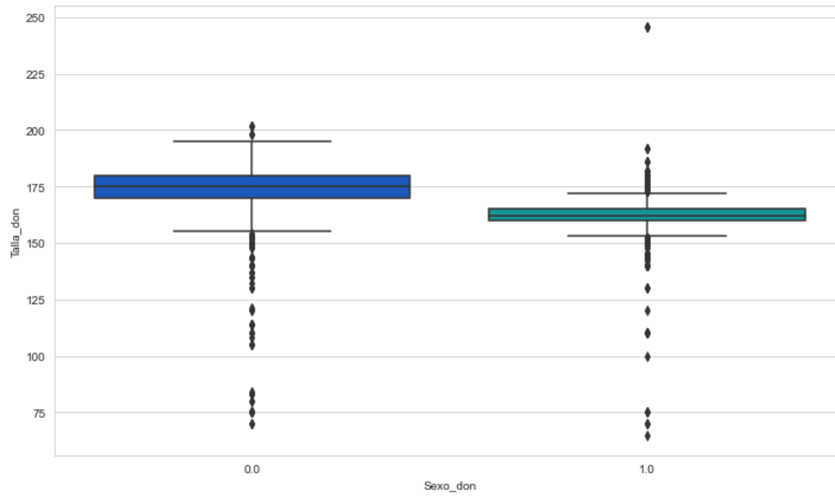

Figura 3-6. Altura del donante según su sexo

Tabla 3-9 Mediana de *Talla_don* según *Sexo_don*

Sexo_don	Talla_don
0	175
1	162

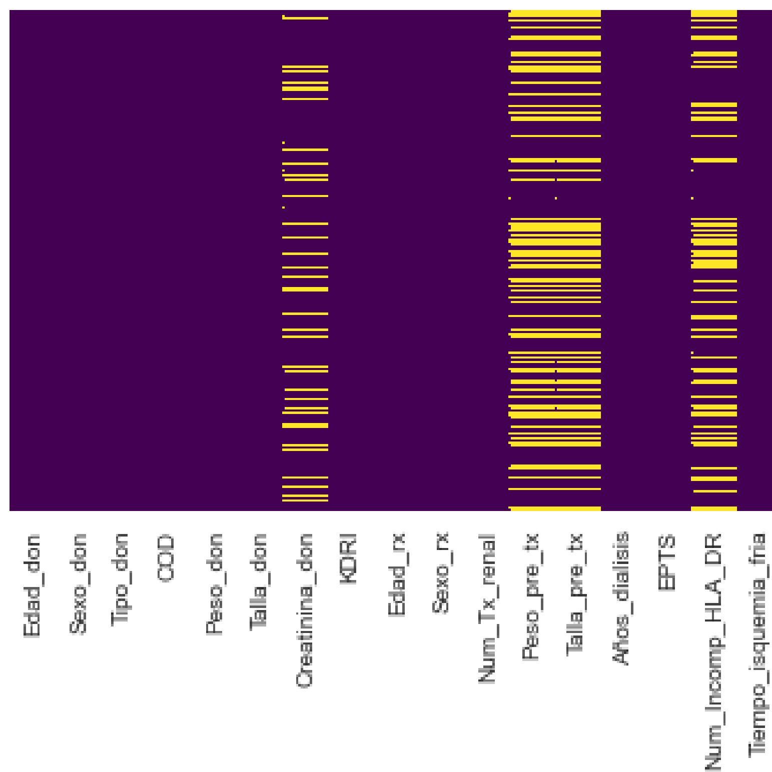

Figura 3-7. *Missing values* tras la 2ª imputación

3.2.3.3 Peso del receptor previo al trasplante

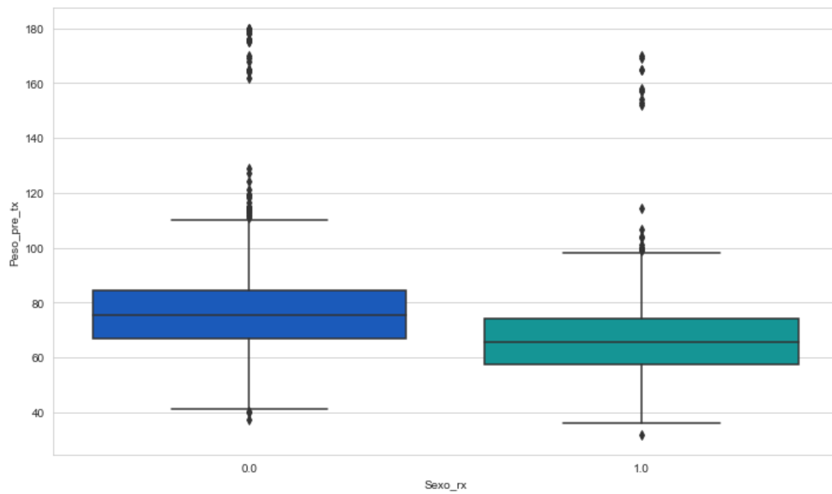

Figura 3-8. Peso del paciente según su sexo

Tabla 3-10. Mediana de *Peso_pre_tx* según *Sexo_rx*

Sexo_rx	Peso_pre_tx
0	75.5
1	65.45

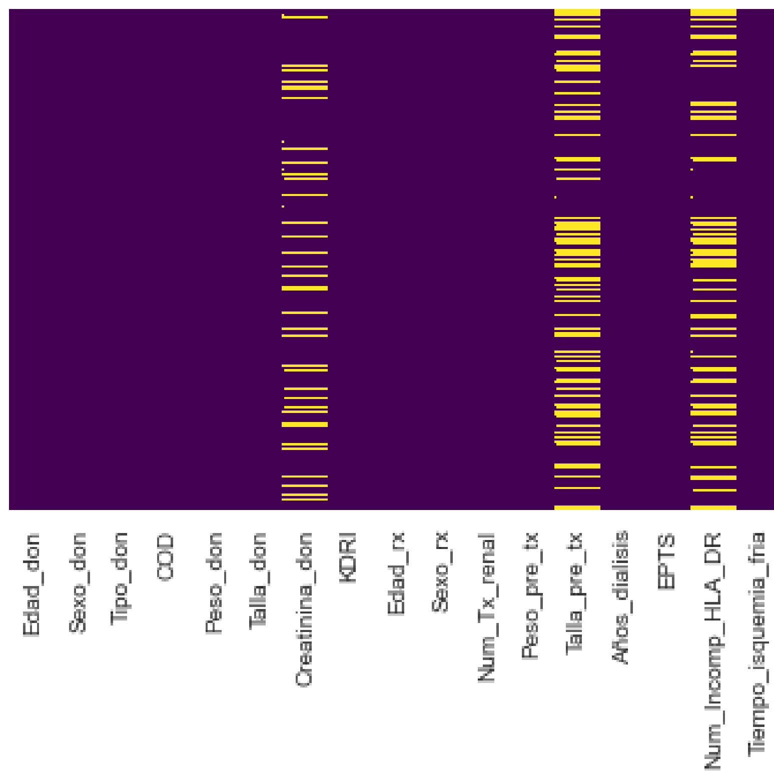

Figura 3-9. *Missing values* tras la 3ª imputación

3.2.3.4 Talla del receptor previa al trasplante

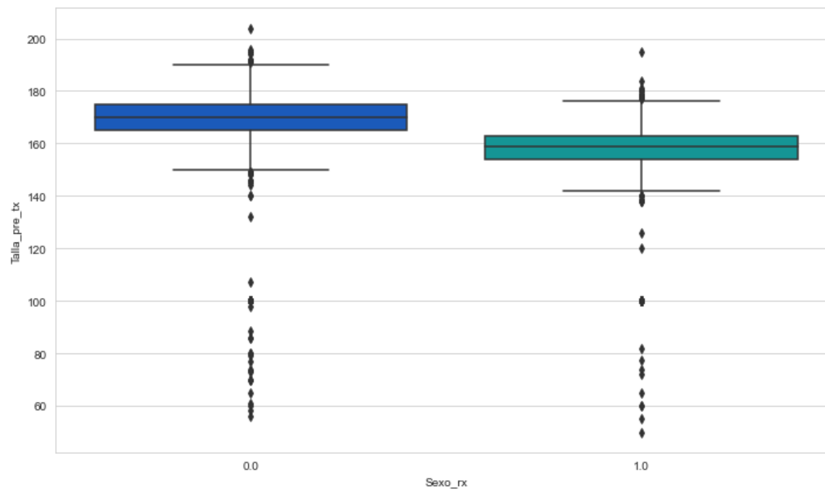

Figura 3-10. Altura del paciente según su sexo

Tabla 3-11. Mediana de *Peso_pre_tx* según *Sexo_rx*

Sexo_rx	Talla_pre_tx
0	170
1	159

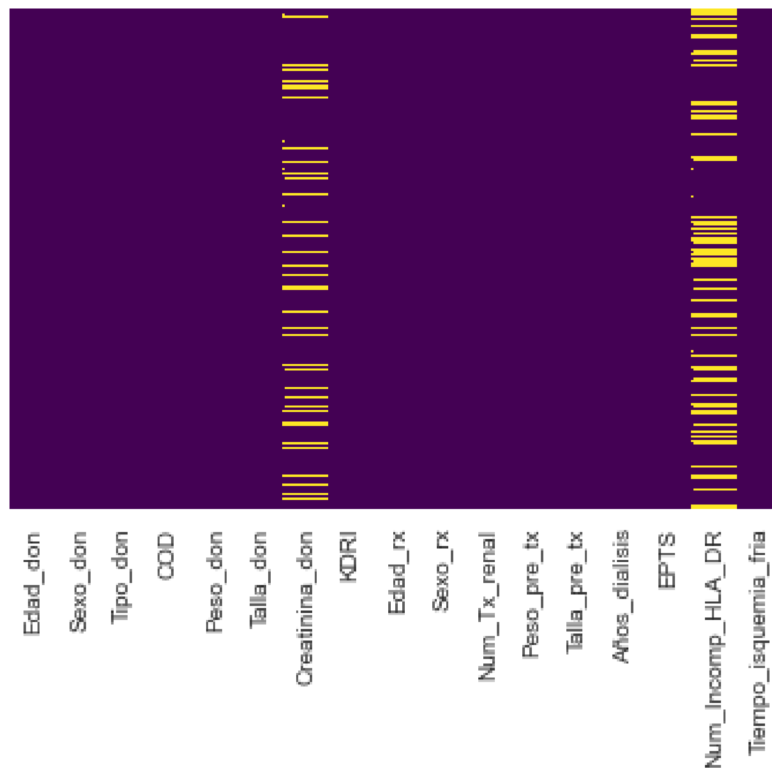

Figura 3-11. *Missing values* tras la 4ª imputación

3.2.3.5 Creatinina del donante

Dado que no se tiene claro en función de qué varía el valor de la creatinina se realizarán varias pruebas para valorar cuál es la correlación más fiable.

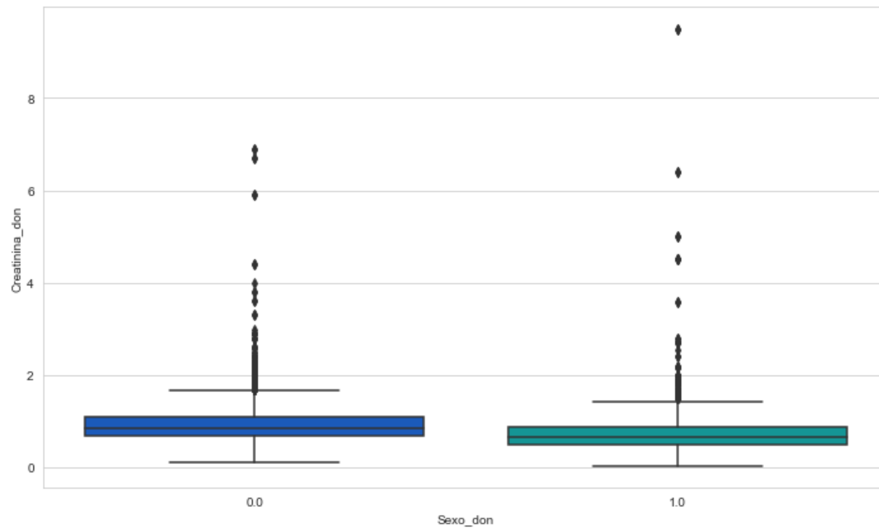

Figura 3-12. Creatinina del donante según su sexo

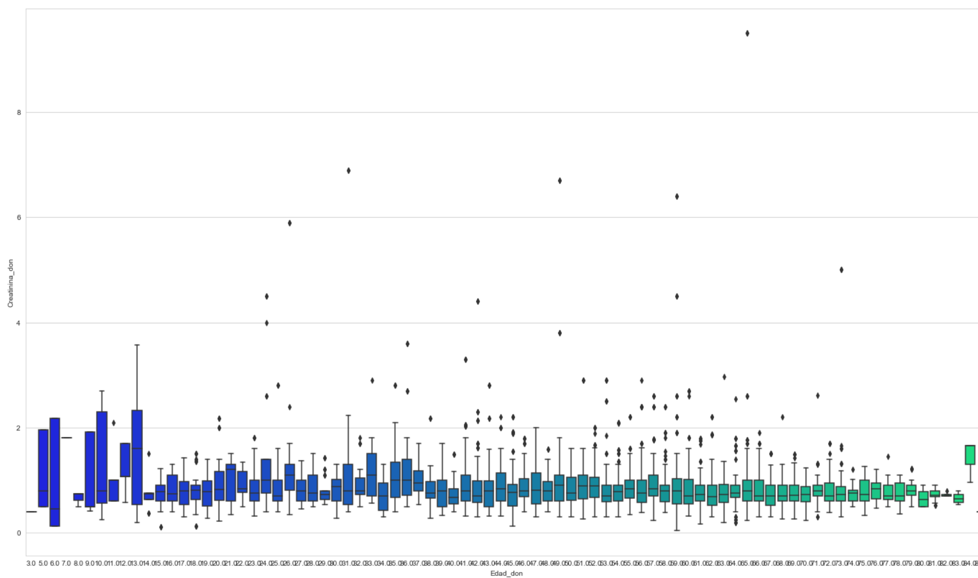

Figura 3-13. Creatinina del donante según su edad

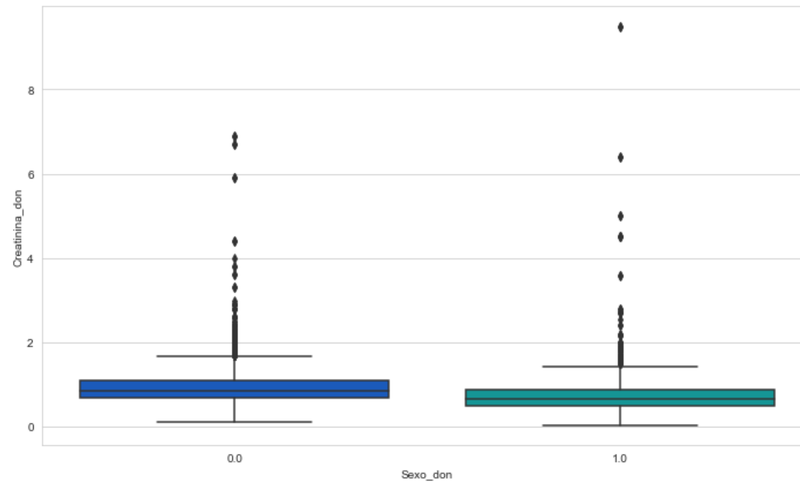

Figura 3-14. Creatinina del donante según su causa de la muerte

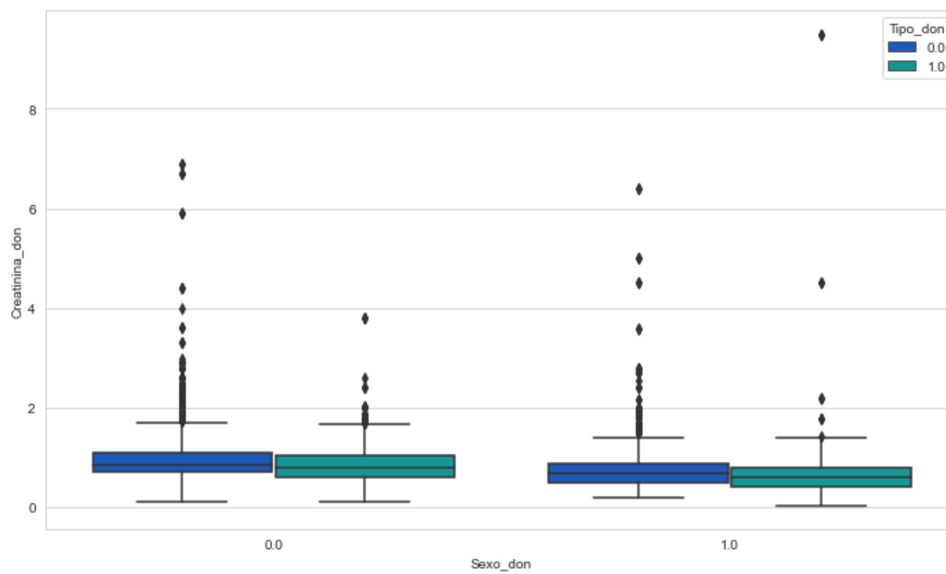

Figura 3-15. Creatinina del donante según su sexo y tipo de donante

Finalmente se decide utilizar una doble correlación entre el valor de la creatinina y el sexo del donante junto con el tipo de donante.

Tabla 3-12. Mediana de *Creatinina* según *Sexo_rx* y *Tipo_don*

Sexo_don	Tipo_don	Creatinina
0	0	0.8
0	1	0.72
1	0	0.8
1	1	0.7

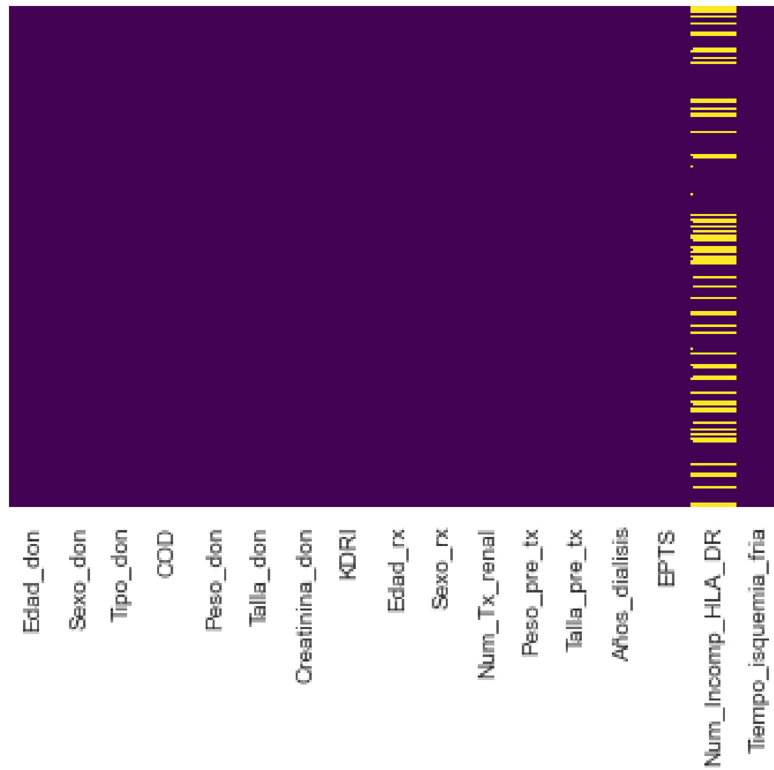

Figura 3-16. *Missing values* tras la 5ª imputación

3.2.3.6 Número de incompatibilidades HLA-DR

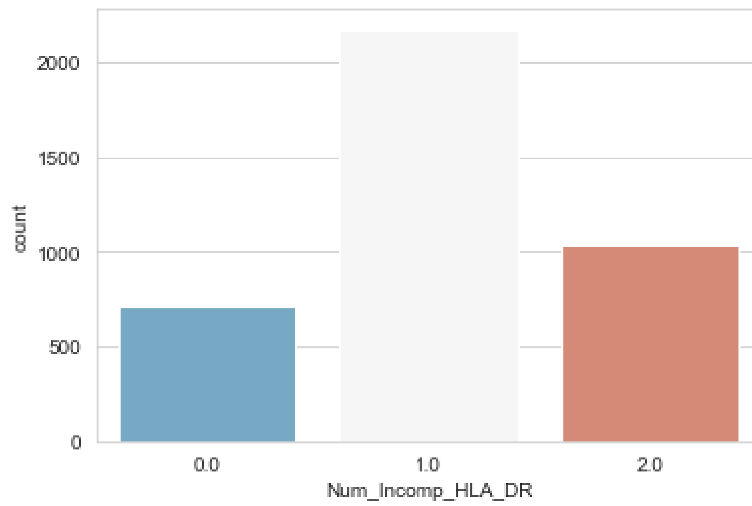

Figura 3-17. Número de incompatibilidades HLA-DR

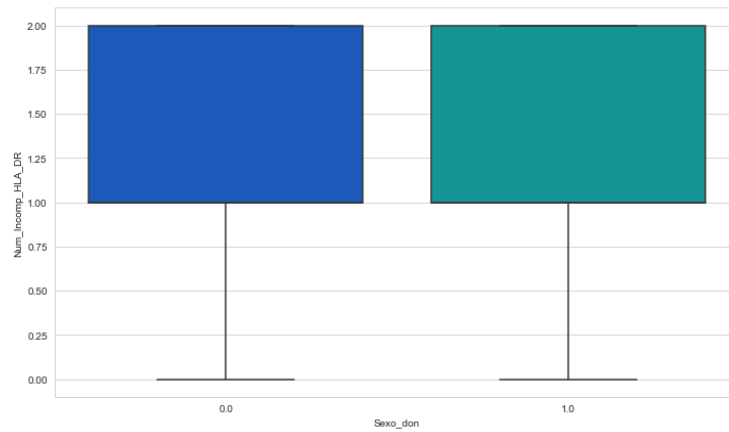

Figura 3-18. Número de incompatibilidades HLA-DR según sexo donante

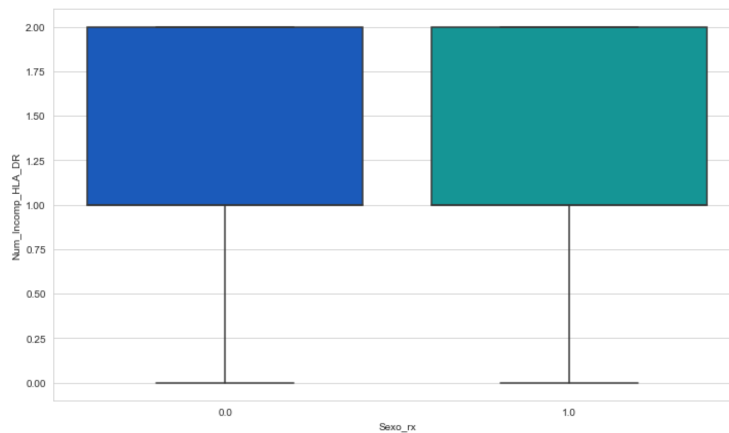

Figura 3-19. Número de incompatibilidades HLA-DR según sexo paciente

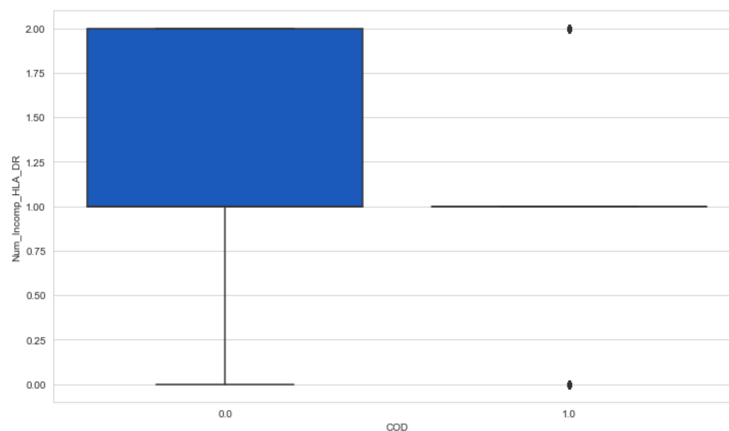

Figura 3-20. Número de incompatibilidades HLA-DR según causa de la muerte

Se puede ver muy poca trazabilidad de las diferencias del número de incompatibilidades HLA-DR con respecto a los otros campos. De esta manera se decide suplir los *missing values* en este campo por el valor 1 dado que, como se observa en la Figura 3-17, este valor está presente en más del 50% de los casos.

Con lo que, tras esta última imputación ya se tiene todo el conjunto de datos completo y los *missing values* han sido tratados de manera que los nuevos valores estén dentro del rango normal correspondiente para el campo en cuestión.

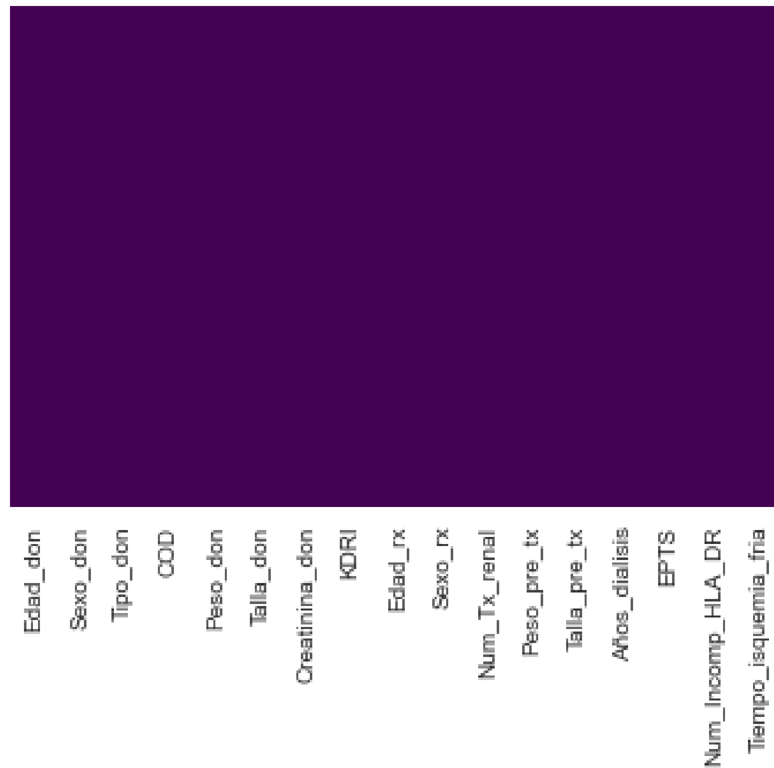

Figura 3-21. *Missing values* tras la 6ª imputación

3.2.4 Transformación en matrices

Ahora ya se tienen ambos conjuntos de datos preparados para su utilización en el entrenamiento de las redes neuronales de aprendizaje profundo. Pero siguen estando almacenados en un DataFrame y para que el modelo pueda leerlos y utilizarlos correctamente deben estar almacenados en matrices.

Los objetos que utilizaremos para ello serán las numpy.arrays, las matrices de la biblioteca NumPy. Para ello se parte de los DataFrame construidos en las secciones anteriores, el conjunto de datos de entrada está almacenado en la variable *X_graf* y el conjunto de datos de salida está almacenado en la variable *Y*.

Primero, para utilizarse, deberá importarse la biblioteca.

Importación de biblioteca NumPy

```
import numpy as np
```

La transformación del conjunto de datos de entrada será sencilla. Ejecutando los siguientes comandos se tendrán los datos almacenados en una matriz con las dimensiones (n, m), siendo *n* el número de variables y *m* el número de ejemplos, en este caso de trasplantes de riñón.

Transformar conjunto de datos de entrada en una matriz

```
X_model = np.array(X_graf.T, dtype=float)
```

Si comprobamos las propiedades de esta matriz nos devuelve los valores que se observan en la Tabla 3-12.

Tabla 3-13. Propiedades del conjunto de datos de entrada *X_model*

Propiedad	Valor
shape	(17, 5491)
dtype	float64

Sin embargo, si se ejecutase la misma operación sobre el conjunto de datos de salida Y, no obtendríamos el resultado esperado, dado que la matriz de dimensiones (6, 1) que alberga el valor del estado del paciente está almacenada en una sola columna. Por lo que, al realizar esta operación, nos devolvería una matriz de dimensiones (1, 5491).

Para evitar esto y obtener la matriz con las dimensiones correctas (6, 5491) se debe realizar un ajuste a este proceso.

Transformar conjunto de datos de salida en una matriz

```
Y_model = np.array(Y)
Y_final = np.empty((6,5491), dtype=float)

for i in range(5491):

    res = Y_model[0][i]
    Y_final[:, i] = res
```

Si ahora se comprueban las propiedades de esta matriz se confirmará que los valores son los correctos. Estos pueden observarse en la Tabla 3-13.

Tabla 3-14. Propiedades del conjunto de datos de salida *Y_final*

Propiedad	Talla_pre_tx
shape	(6, 5491)
dtype	float64

Ahora ya se tienen ambos conjuntos preparados y transformados para poder ser utilizados en el entrenamiento de un modelo de aprendizaje profundo. Por lo que el siguiente paso es crear la estructura de la red neuronal y definir su funcionamiento.

3.3 Primer modelo

El primer modelo que se crea es un modelo simple. Tendrá los elementos básicos necesarios, enumerados en la sección 2.4, y además incluirá la normalización del conjunto de datos de entrada, dado que la función softmax que trabaja con exponenciales es muy sensible a los valores y puede producir errores si no se realiza este ajuste. Por lo que se procederá a definir las funciones que integran el modelo y posteriormente se definirá el modelo en sí mismo.

El primer elemento es la inicialización de pesos. En este primer modelo la inicialización se hará aleatoria sin ningún ajuste.

Inicialización de pesos

```
def init_params(layers_dims):

    np.random.seed(7)
    parameters = {}
```

```
    L = len(layers_dims) - 1

    for l in range(L):

        parameters["W" + str(l+1)] = np.random.randn(
                                    layers_dims[l+1], layers_dims[l])

        parameters["b" + str(l+1)] = np.zeros((layers_dims[l+1], 1))

    return parameters
```

El siguiente elemento que programar será la propagación hacia delante. Este primer modelo será una red neuronal de 3 capas, por lo que se configurarán las dos capas ocultas con la función de activación ReLU y la capa de salida con la función de activación *softmax classifier*.

Propagación hacia delante

```
def forward_prop(X, parameters):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]

    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)
    Z3 = np.dot(W3, A2) + b3
    A3 = softmax(Z3)

    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3)

    return A3, cache
```

ReLU

```
def relu(x):

    s = np.maximum(0, x)

    return s
```

Sotmax Classifier

```
def softmax(x):

    expo = np.exp(x)
    exp_sum = np.sum(expo, axis=0)
    s = expo / exp_sum
```

```
    return s
```

Una vez, el modelo ha realizado el cálculo de la propagación hacia delante y producido una predicción sobre los datos de entrada se obtendrá una medida de su acierto mediante la función de coste.

Función de coste de Softmax classifier

```
def compute_cost(A4, Y, eps=0.00000000001):

    m = Y.shape[1]

    loss = - np.sum(np.multiply(Y, np.log(A4+eps)), axis=0)

    cost = 1/m * np.sum(loss)

    return cost
```

Y, a continuación, se calcula el gradiente de esta función de coste con respecto de todas las variables mediane el proceso de propagación hacia atrás.

Propagación hacia atrás

```
def back_prop(X, Y, cache):

    Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3 = cache

    m = X.shape[1]

    dZ3 = A3 - Y
    dW3 = 1/m * np.dot(dZ3, A2.T)
    db3 = 1/m * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1/m * np.dot(dZ2, A1.T)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1/m * np.dot(dZ1, X.T)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1" : dW1, "db1" : db1, "dZ1" : dZ1, "dA1" : dA1,
             "dW2" : dW2, "db2" : db2, "dZ2" : dZ2, "dA2" : dA2,
             "dW3" : dW3, "db3" : db3, "dZ3" : dZ3}

    return grads
```

Por último, se actualizan los valores de los pesos de la red neuronal.

Actualización de pesos

```
def update_parameters(parameters, gradients, learning_rate):
```

```
    L = len(parameters) // 2

    for l in range(L):

        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                             learning_rate * gradients["dW" + str(l+1)]

        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
                             learning_rate * gradients["db" + str(l+1)]

    return parameters
```

Ya se tienen todos los elementos definidos por lo que se da paso a la creación del modelo. Este primer modelo tendrá una primera capa oculta de 20 neuronas y una segunda capa oculta de 10. Tanto la capa de entrada como la capa de salida tienen la restricción de tener 17 y 6 neuronas respectivamente.

Se establece un *learning rate* por defecto de 0.01, pero se incluye este valor en los argumentos de la función por lo que podrá ser modificado cada vez que se ejecute determinando el nombre de la variable y el valor objetivo.

El mismo caso se tiene para el número de iteraciones que se marca con el valor por defecto de 30.010. Este 10 extra se incluye para la impresión del coste de la que se habla en el siguiente párrafo.

Modelo 1

```
def model_1(X, Y, learning_rate = 0.01, num_iter = 30010,
                                                    print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layer_dims = [X.shape[0], 20, 10, 6]

    parameters = init_params_he(layers_dims)

    for i in range(num_iter):

        A3, cache = forward_prop(X, parameters)

        cost = compute_cost(A3, Y)

        gradients = back_prop(X, Y, cache)

        parameters = update_parameters(parameters, gradients,
                                                       learning_rate)

        if print_cost and i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
        if print_cost and i % 1000 == 0:
            costs.append(cost)


    plt.plot(costs)
    plt.ylabel('Cost')
```

```
        plt.xlabel('Iterations (x1,000)')
        plt.title("Learning rate =" + str(learning_rate))
        plt.show()

        return parameters
```

Como se puede observar, además de todo el proceso iterativo de entrenamiento se añaden elementos que registran el coste. Se tiene uno para imprimir el coste en pantalla cada 10.000 iteraciones y se tiene otro para guardarlo en una lista cada 1.000 iteraciones. Este último elemento es el que se utiliza al final para hacer un gráfico de la evolución de este valor, así puede visualizarse el comportamiento del modelo.

Ya creado este primer modelo se ejecuta su lanzamiento y se prueba su funcionamiento.

El resultado de cada modelo de entrenamiento se presentará en una tabla con la evolución de los costes y en el gráfico resultante correspondiente. Para cada modelo se ejecutarán diferentes entrenamientos variando los hiperparámetros y comparando los resultados para conseguir un buen ajuste de estos.

Para este modelo se probarán diferentes valores para el learning rate mientras que el número de iteraciones se mantendrá igual en todos.

- α=0.001

Tabla 3-15. Coste modelo 1 $\alpha = 0.001$

Iteración	Coste
0	17.514369992768298
10.000	0.9366959440352572
20.000	0.8714879711866625
30.000	0.8491992644508328

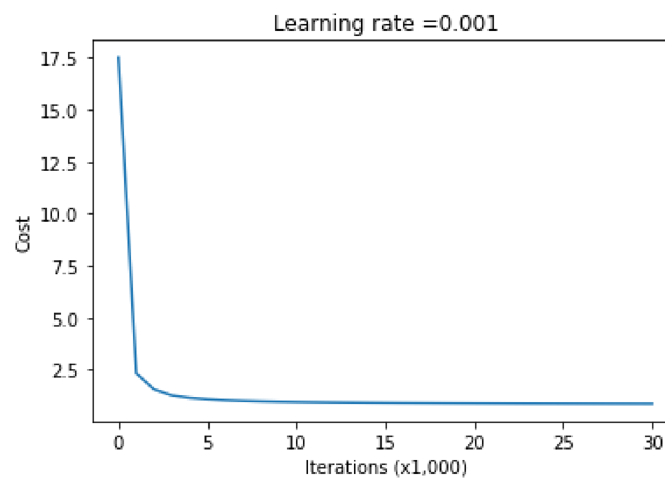

Figura 3-22. Evolución del coste modelo 1 con $\alpha = 0.001$

- α=0.01

Tabla 3-16. Coste modelo 1 $\alpha = 0.01$

Iteración	Coste
0	17.514369992768298
10.000	0.8105096059040159
20.000	0.795931158502194
30.000	0.7880331729123541

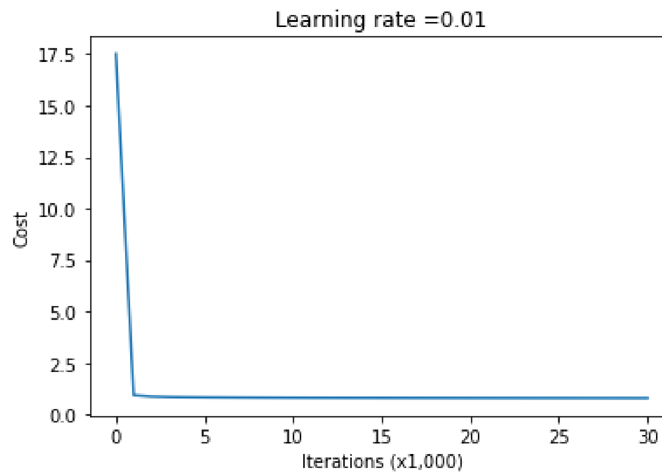

Figura 3-23. Evolución del coste modelo 1 con $\alpha = 0.01$

- α=0.05

Tabla 3-17. Coste modelo 1 $\alpha = 0.05$

Iteración	Coste
0	17.514369992768298
10.000	0.7786033328600044
20.000	0.7614019635096139
30.000	0.749529264322663

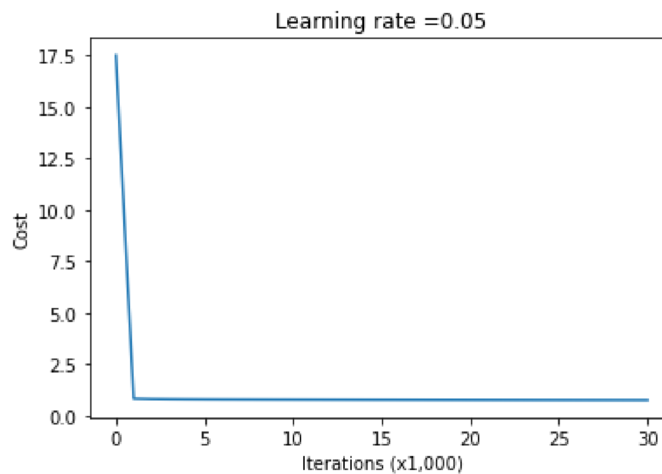

Figura 3-24. Evolución del coste modelo 1 con $\alpha = 0.05$

- α=0.1

Tabla 3-18. Coste modelo 1 α = 0.1

Iteración	Coste
0	17.514369992768298
10.000	0.7619983750843095
20.000	0.7402036444773602
30.000	0.7269441508152209

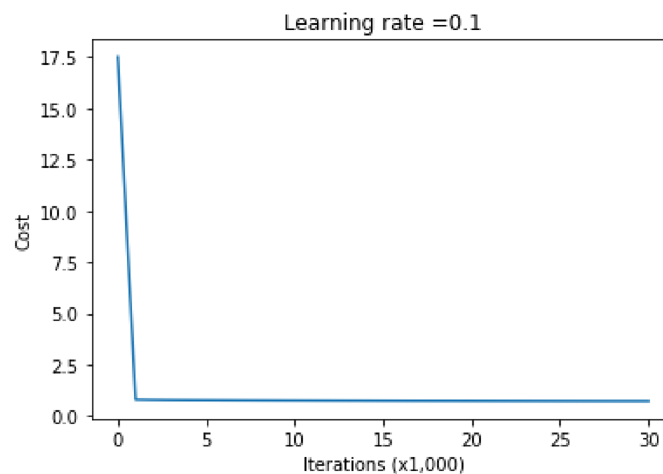

Figura 3-25. Evolución del coste modelo 1 con α = 0.1

Como se puede observar, cuanto más alto es el valor del *learning rate*, más rápido desciende el valor de la función de coste. Sin embargo, también se aprecia que después del primer descenso inicial, las siguientes iteraciones no son muy eficaces a la hora de reducir el coste, y seguramente se esté produciendo el fenómeno de *overfitting*, o sobreajuste. Esto es debido a que este modelo no está optimizado. A lo largo de los siguientes modelos se observará como los diferentes cambios aplicados van mejorando los resultados obtenidos.

3.4 Segundo modelo

En este segundo modelo se sigue la estructura definida en el primero, pero se jugará con las neuronas de las capas ocultas para ver cómo afectan estas al resultado. Además, como particularidad en este modelo, se realiza una pequeña modificación a la hora de obtener los gráficos para omitir las primeras iteraciones y poder observar más claramente la curva de descenso.

Por lo tanto, sin definir ningún código nuevo, se pasa a presentar los resultados obtenidos con las diferentes dimensiones de la red neuronal.

El hiperparámetro del *learning rate* se mantiene constante en 0.1 para todas las variaciones del modelo 2, así como el número de iteraciones se establece en 30.000 para todas por igual.

- layers_dims = [17, 28, 8, 6]

Tabla 3-19. Coste modelo 2 con estructura [17, 28, 8, 6]

Iteración	Coste
10.000	0.7480631012677226
20.000	0.7235482247370316
30.000	0.7097168160162771

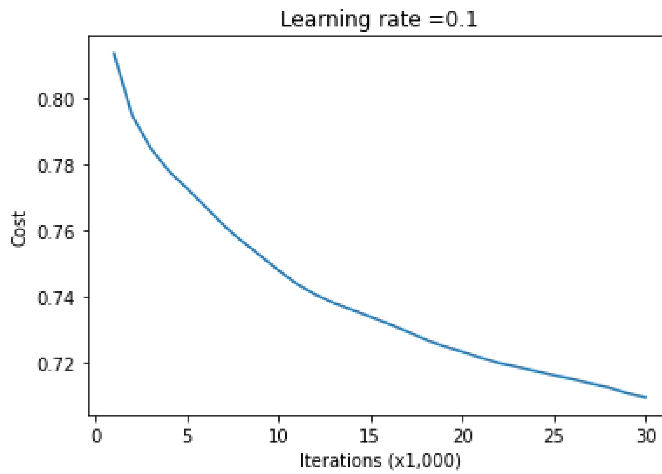

Figura 3-26. Evolución del coste modelo 1 con estructura [17, 28, 8, 6]

- layers_dims = [17, 25, 14, 6]

Tabla 3-20. Coste modelo 2 con estructura [17, 25, 14, 6]

Iteración	Coste
10.000	0.7542845179417105
20.000	0.7297041924371445
30.000	0.7130392884292414

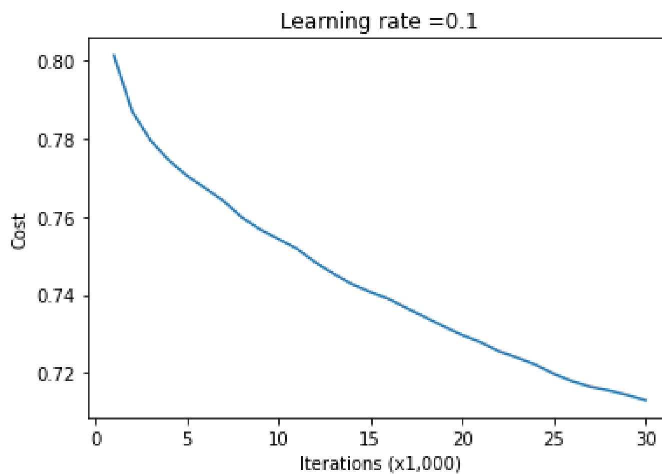

Figura 3-27. Evolución del coste modelo 1 con estructura [17, 25, 14, 6]

- layers_dims = [17, 20, 20, 6]

Tabla 3-21. Coste modelo 2 con estructura [17, 20, 20, 6]

Iteración	Coste
10.000	0.7141150830105295
20.000	0.6782178183591386
30.000	0.661722056813843

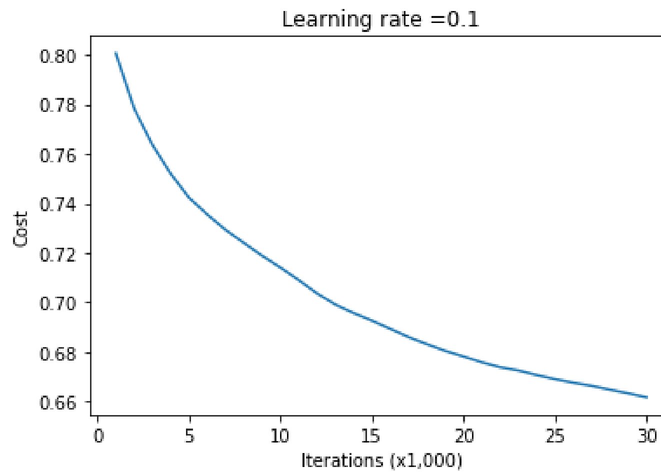

Figura 3-28. Evolución del coste modelo 1 con estructura [17, 20, 20, 6]

Esta tercera variación del segundo modelo es la que mejores resultados ha dado con bastante diferencia, siendo la única que baja de 0.7 en el valor de la función de coste. Además, lo consigue con bastante amplitud, ya que lo supera antes de la mitad del entrenamiento.

De este modelo se sacan dos nociones muy importantes.

La primera es que se confirma la máxima de que no por ser más grande necesariamente es mejor. Las redes neuronales de más tamaño han dado peores resultados y es que no por tener más neuronas en cada capa se obtiene un resultado mejor. Por eso es muy importante probar diferentes estructuras cuando se está comenzando un nuevo proyecto de redes neuronales.

La segunda es una noción práctica que se irá repitiendo en los siguientes modelos y es que, por alguna razón, las redes neuronales producen mejores resultados cuando tienen capas contiguas del mismo tamaño. Esta noción puede ser completamente circunstancial, pero, como se observará en el desarrollo, será de gran utilidad.

3.5 Tercer modelo

En este tercer modelo se realizan modificaciones importantes, tanto en su estructura como en las funciones que lo componen.

El más relevante, o el que a la larga tendrá mayor impacto en los resultados, es la modificación de la función de inicialización de los pesos. Esta deja de ser aleatoria y se comienza a utilizar la inicialización de He [18], como se comenta en el apartado 2.4.2. Esta será la inicialización utilizada para todos los modelos a partir de este punto.

Inicialización de He

```
def init_params_he(layers_dims):

    np.random.seed()
    parameters = {}
    L = len(layers_dims) - 1

    for l in range(L):

        parameters["W" + str(l+1)] = np.random.randn(
                                  layers_dims[l+1], layers_dims[l]) *
                                          np.sqrt(2 / layers_dims[l])
        parameters["b" + str(l+1)] = np.zeros((layers_dims[l+1], 1))
```

```
    return parameters
```

La segunda modificación que se realiza en este modelo respecto de los anteriores es que se establece un modelo de red neuronal de 4 capas. Se añade así una nueva capa oculta y se realizan las modificaciones pertinentes en las funciones de propagación hacia delante y propagación hacia atrás.

Propagación hacia delante

```
def forward_prop(X, parameters):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]
    W4 = parameters["W4"]
    b4 = parameters["b4"]

    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)
    Z3 = np.dot(W3, A2) + b3
    A3 = relu(Z3)
    Z4 = np.dot(W4, A3) + b4
    A4 = softmax(Z4)

    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3, Z4, A4,
                                                              W4, b4)

    return A4, cache
```

Propagación hacia atrás

```
def back_prop(X, Y, cache):

    Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3, Z4, A4, W4, b4 =
                                                                cache
    m = X.shape[1]

    dZ4 = A4 - Y
    dW4 = 1/m * np.dot(dZ4, A3.T)
    db4 = 1/m * np.sum(dZ4, axis=1, keepdims=True)

    dA3 = np.dot(W4.T, dZ4)
    dZ3 = np.multiply(dA3, np.int64(A3 > 0))
    dW3 = 1/m * np.dot(dZ3, A2.T)
    db3 = 1/m * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1/m * np.dot(dZ2, A1.T)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)
```

```
    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1/m * np.dot(dZ1, X.T)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1" : dW1, "db1" : db1, "dZ1" : dZ1, "dA1" : dA1,
             "dW2" : dW2, "db2" : db2, "dZ2" : dZ2, "dA2" : dA2,
             "dW3" : dW3, "db3" : db3, "dZ3" : dZ3, "dA3" : dA3,
             "dW4" : dW4, "db4" : db4, "dZ4" : dZ4}

    return grads
```

Y, por último, se realizan los ajustes en el modelo.

Modelo 3

```
def model_3(X, Y, learning_rate = 0.1, num_iter = 30010,
                                                print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layers_dims = [X.shape[0], 25, 25, 16, 6]

    parameters = init_params_he(layers_dims)

    for i in range(num_iter):

        A4, cache = forward_prop(X, parameters)

        cost = compute_cost(A4, Y)

        gradients = back_prop(X, Y, cache)

        parameters = update_parameters(parameters, gradients,
                                                  learning_rate)

        if print_cost and i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
        if print_cost and i % 1000 == 0:
            costs.append(cost)


    plt.plot(costs)
    plt.ylabel('Cost')
    plt.xlabel('Iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters, costs
```

Siguiendo los buenos resultados de la tercera variación del modelo 2 se decide realizar una estructura similar cuando se definen las dimensiones de la red neuronal. Para poner a prueba la efectividad del nuevo modelo se

definirán diferentes estructuras de red y se observarán los resultados obtenidos.

De nuevo se decide mantener fijos los hiperparámetros del learning rate y el número de iteraciones, siendo 0.1 y 30.000 respectivamente.

Además, se comienza a medir el tiempo de ejecución, para así tener un nuevo elemento con el que comparar los modelos. Esto va a permitir evaluar el consumo de recursos que necesitan diferentes estructuras y en caso de desempeños similares se podrá escoger el modelo más eficiente.

- layers_dims = [17, 20, 20, 6, 6]

Tabla 3-22. Coste modelo 3 con estructura [17, 20, 20, 6, 6]

Iteración	Coste
0	2.070564353180656
10.000	0.6816243015833864
20.000	0.620566949185511
30.000	0.6153598430870492
Tiempo de ejecución	165.18095755577087 s

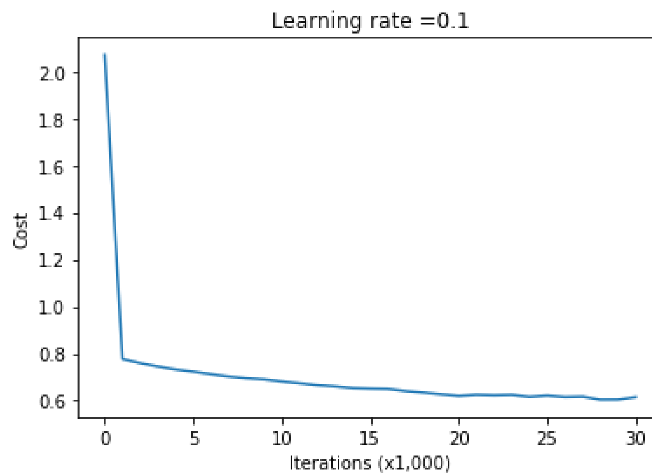

Figura 3-29. Evolución del coste modelo 3 estructura [17, 20, 20, 6, 6]

- layers_dims = [17, 20, 20, 10, 6]

Tabla 3-23. Coste modelo 3 con estructura [17, 20, 20, 10, 6]

Iteración	Coste
0	2.080451683070203
10.000	0.6469534000310354
20.000	0.5999623319614014
30.000	0.5845774058888861
Tiempo de ejecución	167.17461824417114 s

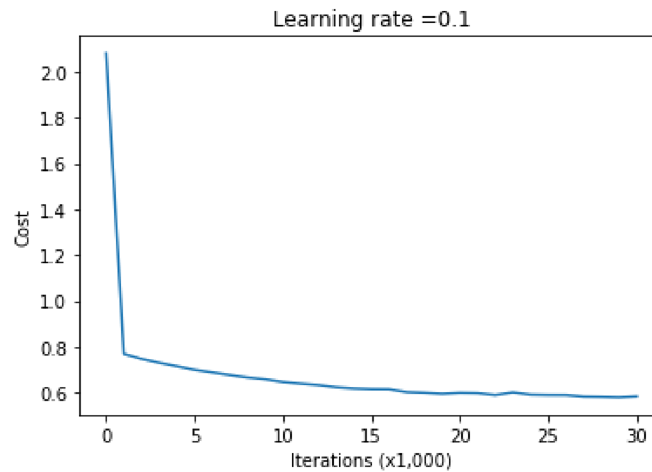

Figura 3-30. Evolución del coste modelo 3 estructura [17, 20, 20, 10, 6]

- layers_dims = [17, 25, 25, 16, 6]

Tabla 3-24. Coste modelo 3 con estructura [17, 25, 25, 16, 6]

Iteración	Coste
0	2.7512739521731455
10.000	0.6019351253376685
20.000	0.5389973908532752
30.000	0.5093929313149543
Tiempo de ejecución	362.2532072067261 s

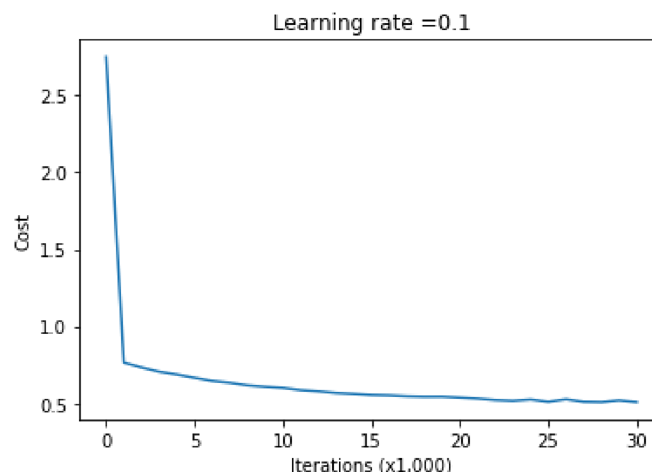

Figura 3-31. Evolución del coste modelo 3 estructura [17, 25, 25, 16, 6]

El primer cambio que se observa es que el coste de la primera iteración ha descendido enormemente. Incluso aumentando el tamaño de la red el valor se ha reducido en un factor de 8, aproximadamente. Esto confirma que la inicialización de los pesos no es un proceso trivial y demuestra el gran valor de los algoritmos de inicialización desarrollados por los autores.

En segundo lugar, se verifica que todas las redes neuronales entrenadas en este proceso han mejorado los resultados de las anteriores. En parte este resultado viene porque la inicialización específica permite que se comience el proceso de minimización de coste desde un punto inicial más próximo al óptimo. Por otro lado, viene por el aumento de tamaño de la estructura de la red neuronal. Mientras que antes solamente un modelo descendía de un valor de la función de coste de 0.7, ahora todos descienden e incluso este último roza el 0.5.

Si bien es cierto que esta tercera variación del modelo mejora enormemente a las otras dos, también se comprueba que el coste computacional es bastante más grande, debido a que las dimensiones de la red son mayores. Sin embargo, el coste computacional apenas pasa de los 6 minutos por lo que se decidirá continuar con esta estructura y realizar nuevos cambios y modificaciones a partir de esta.

3.6 Elementos de medición de rendimiento

En este punto se considera necesario comenzar a medir directamente si el modelo que se ha creado está consiguiendo buenos resultados o no, en otras palabras, medir el porcentaje de aciertos.

Para ello, primero, se necesita convertir la matriz de resultados, que tiene valores entre 0 y 1, en matrices *one-hot*, que tengan un 1 en su valor máximo y un 0 en el resto. Esto se realizará de la siguiente manera:

Convertir resultado en matriz *one-hot*

```
def pred_to_ones(Y):

    Max = np.max(Y, axis=0)

    one_hot = np.int64(Y==Max)

    return one_hot
```

Seguidamente se crea una función que calculará el porcentaje de aciertos de la predicción. Para ello ejecutará una propagación hacia delante, convertirá el resultado en una matriz *one-hot* y la comparará con el resultado objetivo que se le proporciona al modelo.

Calcular porcentaje de acierto

```
def predict(X, Y, parametros, learning_rate = 0.1):

    m = X.shape[1]

    A4, cache = forward_prop(X, parametros)

    prediction = pred_to_ones(A4)

    result = np.all(prediction==Y, axis=0)

    rights = np.sum(result)

    percent = rights / m

    return percent
```

Si ejecutamos este cálculo del porcentaje de aciertos sobre las variaciones del modelo 3 que se mostraban anteriormente en esta sección, con resultados reflejados en las Tablas 3-22, 3-23 y 3-24 se obtienen los siguientes resultados.

Tabla 3-25. Resultado modelo 3 con estructura [17, 20, 20, 6, 6]

Porcentaje de acierto	79.22054270624659 %

Tabla 3-26. Resultado modelo 3 con estructura [17, 20, 20, 10, 6]

Porcentaje de acierto	80.58641413221636 %

Tabla 3-27. Resultado modelo 3 con estructura [17, 25, 25, 16, 6]

Porcentaje de acierto	81.5152067018758 %

Cabe destacar que este porcentaje de acierto lo están obteniendo sobre el mismo conjunto de datos sobre los que se ha entrenado el modelo, por lo que todavía no es un valor determinante de su éxito. Pero si se puede confirmar que el modelo está aprendiendo y ajustando los parámetros de la red neuronal para predecir correctamente el resultado.

También se puede comprobar que el modelo con más porcentaje de acierto es el modelo con el menor coste final en el proceso de la minimización de la función de coste. Con lo que también se confirma esta relación entre ambos parámetros.

Ahora ya se tiene un modelo funcional y un método para medir su desempeño en forma de porcentaje de aciertos. Lo siguiente que habrá que realizar es la separación del conjunto de datos en dos bloques: entrenamiento y prueba.

La proporción que se sigue para esta repartición es de 80/20. El 80% de los datos se utilizarán para entrenar el modelo y el otro 20% se utilizará para poner a prueba este modelo sobre datos nuevos con los que no ha trabajado. Teniendo 5491 casos en total se dividirán en 4394 casos para el conjunto de entrenamiento y 1097 para el conjunto de prueba.

En este punto, se creará una función que permute aleatoriamente los datos para crear así conjuntos de entrenamiento y pruebas distintos cada vez que se quiera. Esta función se implementa tal que:

Permutación aleatoria del conjunto de datos

```
def randomize_data(X, Y):

    m = X.shape[1]

    permut = np.array(np.random.permutation(m))

    shuffled_X = np.copy(X[:, permut])
    shuffled_Y = np.copy(Y[:, permut])

    return shuffled_X, shuffled_Y
```

Tras cada permutación se crearán los conjuntos de datos tal que:

Creación conjuntos de entrenamiento y pruebas

```
X_shuffled, Y_shuffled = randomize_data(X_normal, Y_final)
X_train = np.copy(X_shuffled[:,:4393])
Y_train = np.copy(Y_shuffled[:,:4393])
X_test = np.copy(X_shuffled[:,4394:])
Y_test = np.copy(Y_shuffled[:,4394:])
```

De esta manera podemos entrenar el mismo modelo sobre diferentes conjuntos de datos y tener una medida del resultado más cercana a la realidad haciendo la media de los porcentajes de acierto.

Esto se pone a prueba sobre el modelo 3 con estructura [17, 25, 25, 16, 6] que es la variación que mejores resultados había tenido tanto en minimización de la función de coste como en porcentaje de aciertos sobre el conjunto de entrenamiento. Se establece un *learning rate* de 0.1 y se entrena durante 30.000 iteraciones sobre 3 diferentes conjuntos de datos diferentes.

Los resultados son:

- Conjunto de datos 1

Tabla 3-28. Coste modelo 3 con estructura [17, 25, 25, 16, 6] sobre C.D. 1

Iteración	Coste
0	2.4720808588480883
10.000	0.4839365896931866
20.000	0.4650219842811899
30.000	0.35886526654958456

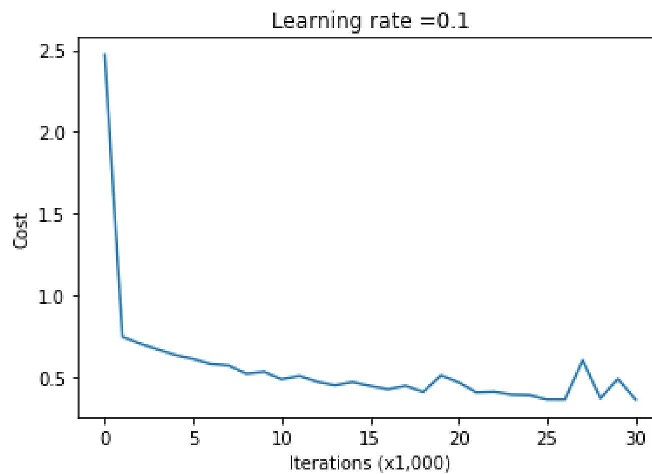

Figura 3-32. Evolución del coste modelo 3 estructura [17, 25, 25, 16, 6] sobre C.D. 1

Tabla 3-29. Resultado modelo 3 con estructura [17, 25, 25, 16, 6] sobre C.D. 1

Conjunto de datos	Porcentaje de acierto
Entrenamiento	86.95652173913043 %
Pruebas	64.53965360072926 %

- Conjunto de datos 2

Tabla 3-30. Coste modelo 3 con estructura [17, 25, 25, 16, 6] sobre C.D. 2

Iteración	Coste
0	1.8750598746263776
10.000	0.5435221528243483
20.000	0.42698553024833424
30.000	0.40011813989428596

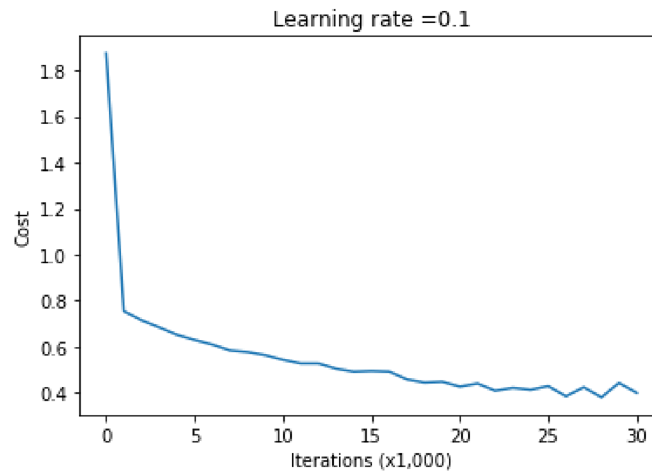

Figura 3-33. Evolución del coste modelo 3 estructura [17, 25, 25, 16, 6] sobre C.D. 2

Tabla 3-31. Resultado modelo 3 con estructura [17, 25, 25, 16, 6] sobre C.D. 2

Conjunto de datos	Porcentaje de acierto
Entrenamiento	85.59071249715456 %
Pruebas	60.34639927073838 %

- Conjunto de datos 3

Tabla 3-32. Coste modelo 3 con estructura [17, 25, 25, 16, 6] sobre C.D. 3

Iteración	Coste
0	2.2838264458839697
10.000	0.5328361080286885
20.000	0.4542018540135375
30.000	0.4285121742432865

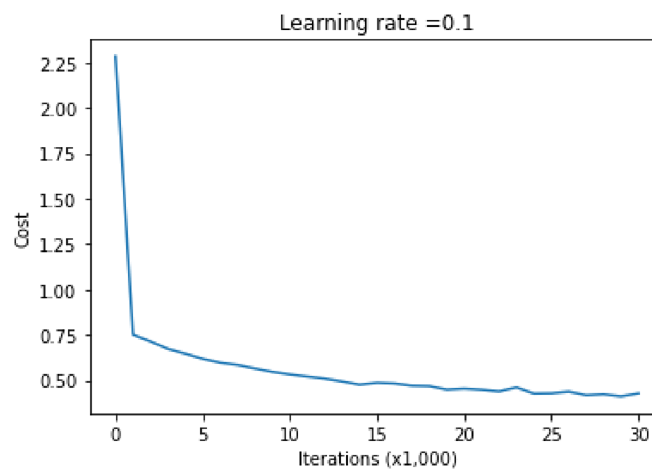

Figura 3-34. Evolución del coste modelo 3 estructura [17, 25, 25, 16, 6] sobre C.D. 3

Tabla 3-33. Resultado modelo 3 con estructura [17, 25, 25, 16, 6] sobre C.D. 3

Conjunto de datos	Porcentaje de acierto
Entrenamiento	85.8183473708172 %
Pruebas	59.8906107566089 %

Tras observar los datos se pueden apreciar varios datos muy relevantes. El primero que, independientemente del conjunto de datos, el porcentaje de acierto sobre el conjunto de entrenamiento es muy estable variando entre 85.5% y 86.9%. En todos los casos el porcentaje de acierto ha superado con creces el objetivo de 76.5% que se estableció para el modelo.

Sin embargo, el rendimiento sobre el conjunto de pruebas está lejos de este valor, oscilando entre un 64.5% y un 59.9%. Esta oscilanción nos da una media de acierto de 61.59%.

Esto nos devuelve que hay una gran varianza entre los resultados del conjunto de entrenamiento y el conjunto de pruebas, lo que signigica que el modelo está cayendo en el *overfitting* o sobreajuste. Como se mencionaba en la sección 2.6 habrá que utilizar métodos de regularización para solventar esto y lograr un mayor porcentaje de aciertos con el conjunto de pruebas.

En una rápida comprobación se utiliza la técnica del *early stopping* para comprobar cómo variará este modelo solamente reduciendo el número de iteraciones de entrenamiento. Fijamos este hiperparámetro en 20.000 iteraciones y realizamos un entrenamiento sobre nuevos conjuntos de datos.

- *Early stopping* en conjunto de datos 4

Tabla 3-34. Coste modelo 3 con estructura [17, 25, 25, 16, 6] y E.S. sobre C.D. 4

Iteración	Coste
0	1.8630301162650862
10.000	0.5424047077830803
20.000	0.44076475702964485

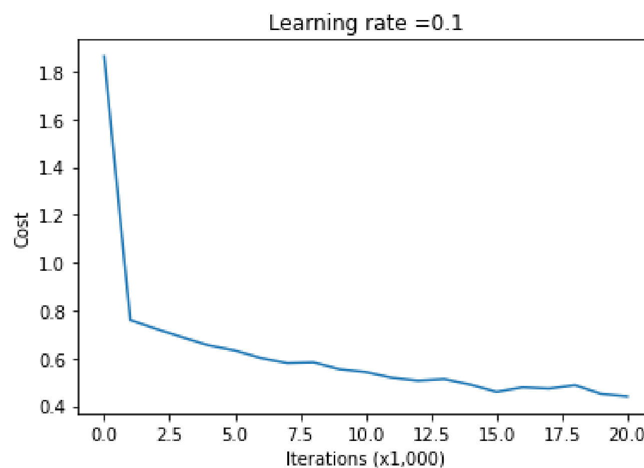

Figura 3-35. Evolución del coste modelo 3 estructura [17, 25, 25, 16, 6] sobre C.D. 4

Tabla 3-35. Resultado modelo 3 con estructura [17, 25, 25, 16, 6] y E.S. sobre C.D. 4

Conjunto de datos	Porcentaje de acierto
Entrenamiento	82.60869565217391 %
Pruebas	63.81039197812215 %

- *Early stopping* en conjunto de datos 5

Tabla 3-36. Coste modelo 3 con estructura [17, 25, 25, 16, 6] y E.S. sobre C.D. 5

Iteración	Coste
0	2.658816575806685
10.000	0.5543439360175924
20.000	0.4275941036716107

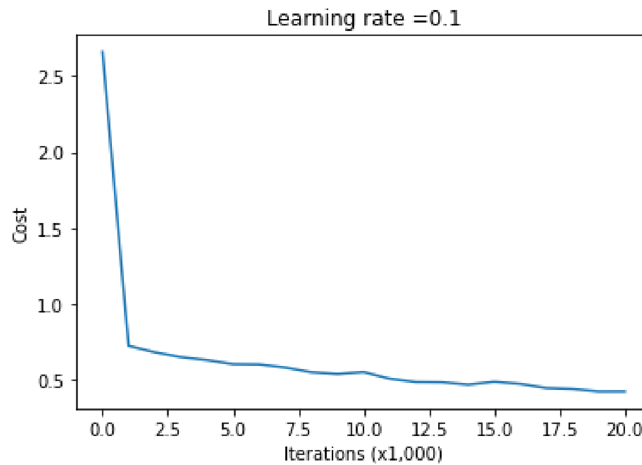

Figura 3-36. Evolución del coste modelo 3 estructura [17, 25, 25, 16, 6] sobre C.D. 5

Tabla 3-37. Resultado modelo 3 con estructura [17, 25, 25, 16, 6] y E.S. sobre C.D. 5

Conjunto de datos	Porcentaje de acierto
Entrenamiento	82.7452765763715 %
Pruebas	68.18596171376481 %

Como se puede observar, ambos modelos se han beneficiado del uso del *early stopping*. Pese a que se reduce el porcentaje de acierto en el conjunto de entrenamiento sus resultados en el conjunto de pruebas han mejorado notablemente hasta alcanzar un 68.2% de aciertos. Esto es una mejora considerable pero como se comentaba en la sección 2.6.3, el *early stopping* no es un método regulador fiable. A continuación, se realizará la implementación de métodos de regularización más efectivos.

3.7 Cuarto modelo

Tanto los resultados del modelo tres sobre los conjuntos de datos 1-3 como las pruebas con el *early stopping* dejan en evidencia la necesidad de incorporar métodos de regularización al modelo. En este cuarto modelo se implementará el método *L2 Regularization* que se introducía en la sección 2.6.1.

Este método implica modificar dos elementos del modelo construido: la función de coste y la propagación hacia atrás.

Cálculo de la función de coste con *L2 Regularization*

```
def compute_cost_with_L2reg(A4, Y, parameters, lambd):

    m = Y.shape[1]
```

```
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    W3 = parameters["W3"]
    W4 = parameters["W4"]

    cross_entr_cost = compute_cost(A4, Y)

    L2reg_cost = (1/m) * (lambd/2) * (np.sum(np.square(W1)) +
                  np.sum(np.square(W2)) + np.sum(np.square(W3)) +
                                          np.sum(np.square(W4)))

    cost = cross_entr_cost + L2reg_cost

    return cost
```

Propagación hacia atrás con *L2 Regularization*

```
def back_prop_with_L2reg(X, Y, cache, lambd):

    Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3, Z4, A4, W4, b4 =
                                                            cache

    m = X.shape[1]

    dZ4 = A4 - Y
    dW4 = 1/m * np.dot(dZ4, A3.T) + ((lambd/m) * W4)
    db4 = 1/m * np.sum(dZ4, axis=1, keepdims=True)

    dA3 = np.dot(W4.T, dZ4)
    dZ3 = np.multiply(dA3, np.int64(A3 > 0))
    dW3 = 1/m * np.dot(dZ3, A2.T) + ((lambd/m) * W3)
    db3 = 1/m * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1/m * np.dot(dZ2, A1.T) + ((lambd/m) * W2)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1/m * np.dot(dZ1, X.T) + ((lambd/m) * W1)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1" : dW1, "db1" : db1, "dZ1" : dZ1, "dA1" : dA1,
             "dW2" : dW2, "db2" : db2, "dZ2" : dZ2, "dA2" : dA2,
             "dW3" : dW3, "db3" : db3, "dZ3" : dZ3, "dA3" : dA3,
             "dW4" : dW4, "db4" : db4, "dZ4" : dZ4}

    return grads
```

Una vez aplicados los cambios en las funciones se modifica el modelo de forma correspondiente para incluir estos nuevos elementos.

Modelo 4

```
def model_4(X, Y, learning_rate = 0.1, num_iter = 25010, lambd = 0,
                                                            print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layers_dims = [X.shape[0], 25, 25, 16, 6]

    parameters = init_params_he(layers_dims)

    for i in range(num_iter):

        A4, cache = forward_prop(X, parameters)

        cost = compute_cost_with_L2reg(A4, Y, parameters, lambd)

        gradients = back_prop_with_L2reg(X, Y, cache, lambd)

        parameters = update_parameters(parameters, gradients,
                                                               learning_rate)

        if print_cost and i % 5000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
        if print_cost and i % 500 == 0:
            costs.append(cost)


    plt.plot(costs)
    plt.ylabel('Cost')
    plt.xlabel('Iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters, costs
```

Una vez definido el modelo lo primero que se hará será probarlo y ver los resultados obtenidos con diferentes valores de *lambda*, para así intentar ajustar este nuevo hiperparámetro a la red neuronal que se ha utilizado hasta ahora. La estructura no varía, se sigue utilizando unas capas con dimensiones [17, 25, 25, 16, 6].

Los entrenamientos comienzan configurando un *learning rate* de 0.1 y un entrenamiento durante 30.000 iteraciones.

- *lambda* = 0.5

Tabla 3-38. Coste modelo 4 con *lambda* = 0.5

Iteración	Coste
0	2.339111283913261
10.000	0.5568973295806796
20.000	0.4637999396810877
30.000	0.3944432765761647

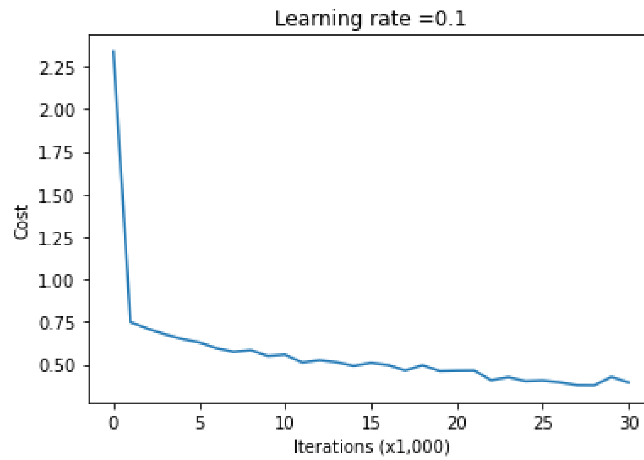

Figura 3-37. Evolución del coste modelo 4 con *lambda* = 0.5

Tabla 3-39. Resultado modelo 4 con *lambda* = 0.5

Conjunto de datos	Porcentaje de acierto
Entrenamiento	88.7093102663328 %
Pruebas	62.8988149498632 %

- *lambda* = 0.7

Tabla 3-40. Coste modelo 4 con *lambda* = 0.7

Iteración	Coste
0	2.507223646744474
10.000	0.569250825423103
20.000	0.4512391567707476
30.000	0.5143694256078543

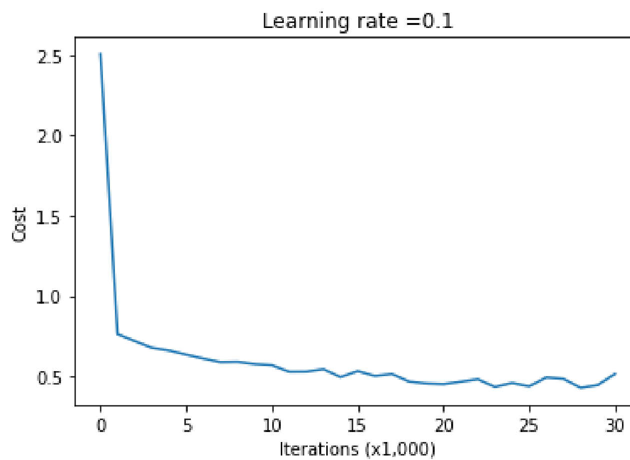

Figura 3-38. Evolución del coste modelo 4 con *lambda* = 0.7

Tabla 3-41. Resultado modelo 4 con *lambda* = 0.7

Conjunto de datos	Porcentaje de acierto
Entrenamiento	86.84270430229911 %
Pruebas	60.1640838650866 %

Se ven mejoras en el comportamiento, pero en este último entrenamiento del modelo 4 con una *lambda* de 0.7 se observa que las últimas 10.000 iteraciones han hecho aumentar el valor de la función de coste. Analizando el gráfico se decide reducir el número de iteraciones de entrenamiento a 25.000 y comprobar qué resultado se obtiene.

- lambda = 0.7 y 25.000 iteraciones

Tabla 3-42. Coste modelo 4 con *lambda* = 0.7 y 25.000 iteraciones

Iteración	Coste
0	2.0382740117729385
5.000	0.6283066578805838
10.000	0.5678489982950663
15.000	0.531653966141389
20.000	0.4847170606961958
25.000	0.4692188145905164

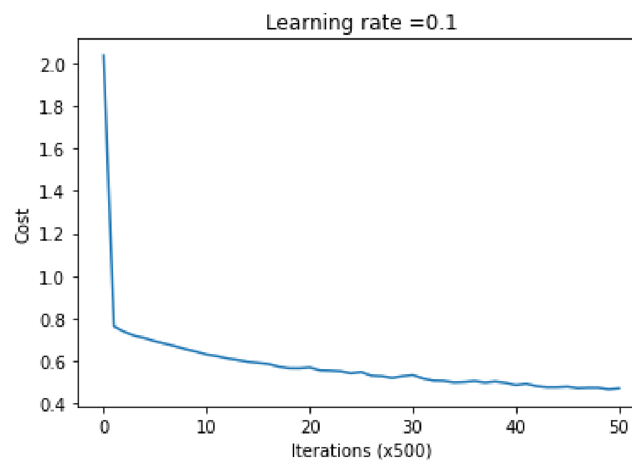

Figura 3-39. Evolución del coste modelo 4 con *lambda* = 0.7 y 25.000 iteraciones

Tabla 3-43. Resultado modelo 4 con *lambda* = 0.7 y 25.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	81.37946733439563 %
Pruebas	66.54512306289881 %

Se comprueba su gran mejoría. Analizando los resultados de los 3 entrenamientos se puede verificar que la regularización ha conseguido aumentar el porcentaje de acierto del conjunto de entrenamiento. Sin embargo, se deben hacer diferentes pruebas para ajustar correctamente el nuevo hiperparámetro *lambda* y también incluir en estas pruebas diferentes variaciones del número de intervenciones.

Se obtienen un conjunto de datos de entrenamiento y otro de pruebas y se realiza el entrenamiento de 4 modelos diferentes. Al entrenar los modelos sobre los mismos datos se puede hacer una buena estimación de qué modelo es más prometedor.

A continuación, se aportarán diferentes tablas con los resultados de diferentes configuraciones del modelo 4. Para su comodidad a la hora de leerlo se ha decidido incluir únicamente las tablas de resultados.

Tabla 3-44. Modelo 4 con *lambda* = 0.5 y 20.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	83.56476212155702 %
Pruebas	67.91248860528715 %

Tabla 3-45. Modelo 4 con *lambda* = 0.5 y 25.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	85.63623947188709 %
Pruebas	64.35733819507748 %

Tabla 3-46. Modelo 4 con *lambda* = 0.7 y 20.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	83.26883678579559 %
Pruebas	68.82406563354604 %

Tabla 3-47. Modelo 4 con *lambda* = 0.7 y 25.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	85.36307762349192 %
Pruebas	65.17775752051048 %

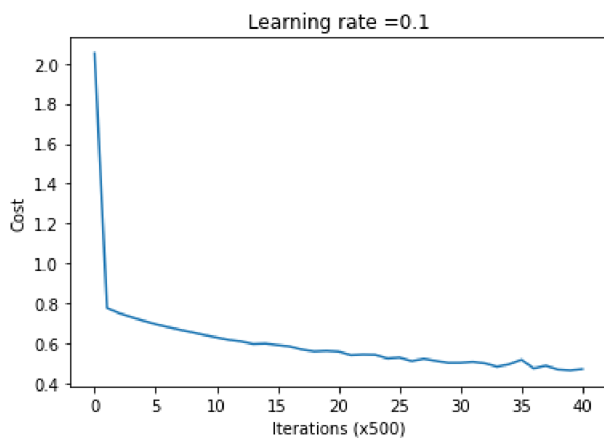

Figura 3-40. Evolución del coste modelo 4 con *lambda* = 0.5 y 20.000 iteraciones

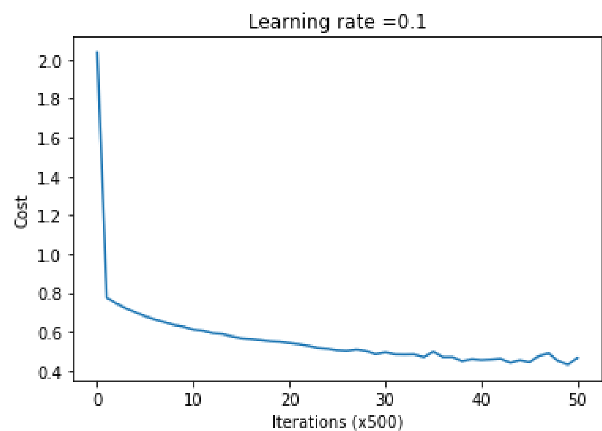

Figura 3-41. Evolución del coste modelo 4 con *lambda* = 0.5 y 25.000 iteraciones

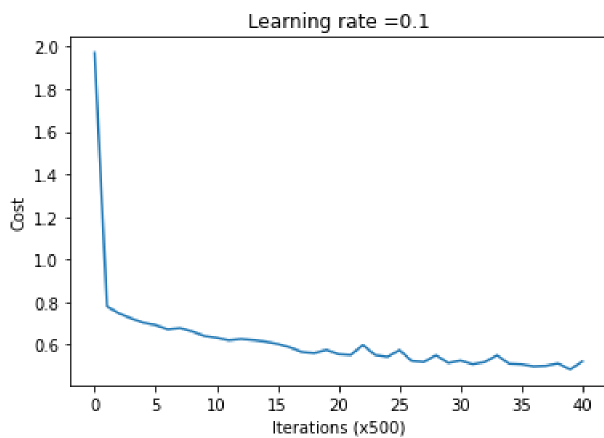

Figura 3-42. Evolución del coste modelo 4 con *lambda* = 0.7 y 20.000 iteraciones

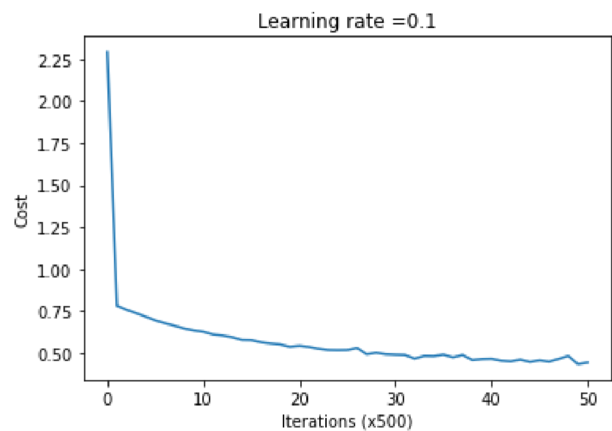

Figura 3-43. Evolución del coste modelo 4 con *lambda* = 0.7 y 25.000 iteraciones

Se puede observar claramente los mejores resultados obtenidos con un número de iteraciones menor. Se volverán a obtener nuevos conjuntos de datos de entrenamiento y pruebas y se realizará una nueva tanda de entrenamientos.

Tabla 3-48. Modelo 4 con *lambda* = 0.7 y 15.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	80.28681994081494 %
Pruebas	63.71923427529627 %

Tabla 3-49. Modelo 4 con *lambda* = 0.7 y 20.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	84.38424766674254 %
Pruebas	67.27438468550593 %

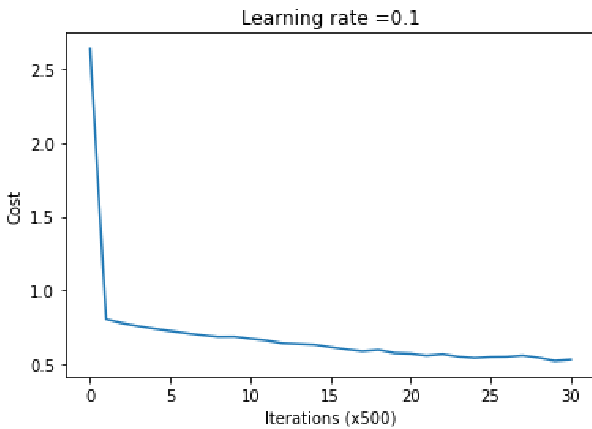

Figura 3-44. Evolución del coste modelo 4 con *lambda* = 0.7 y 15.000 iteraciones

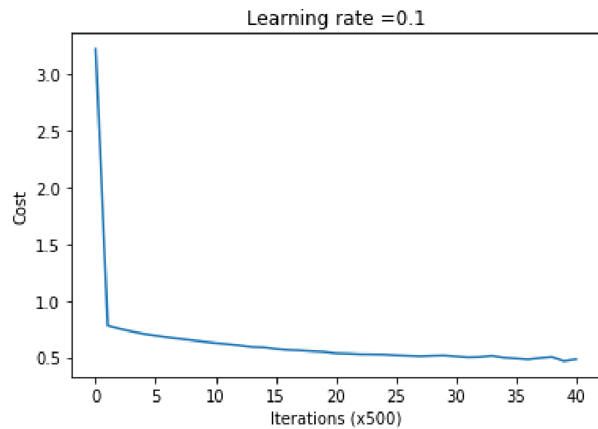

Figura 3-45. Evolución del coste modelo 4 con *lambda* = 0.7 y 20.000 iteraciones

Tabla 3-50. Modelo 4 con *lambda* = 0.9 y 15.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	81.17459594809925 %
Pruebas	60.25524156791249 %

Tabla 3-51. Modelo 4 con *lambda* = 0.9 y 20.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	83.7696335078534 %
Pruebas	68.3682771194165 %

Tabla 3-52. Modelo 4 con *lambda* = 0.9 y 25.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	84.77122695196904 %
Pruebas	63.44576116681859 %

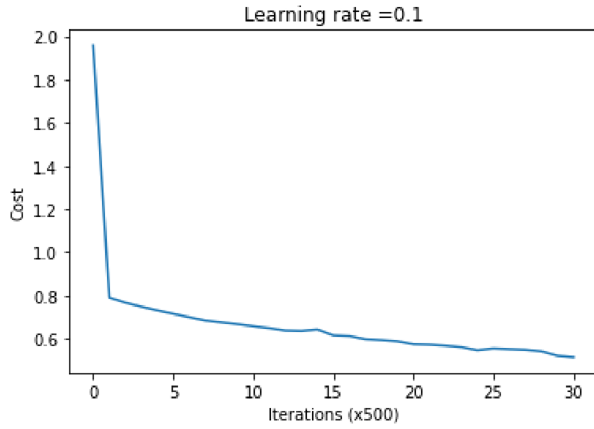

Figura 3-46. Evolución del coste modelo 4 con *lambda* = 0.9 y 15.000 iteraciones

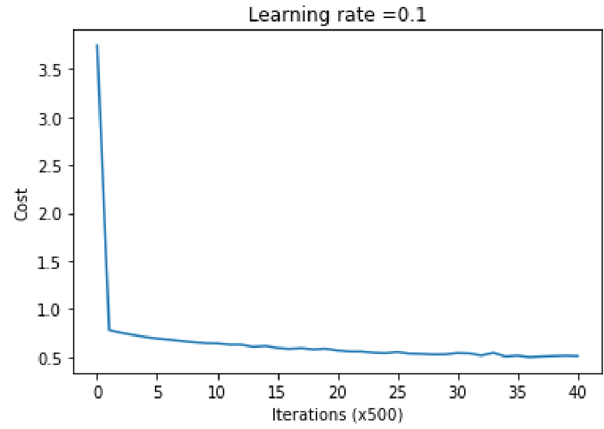

Figura 3-47. Evolución del coste modelo 4 con *lambda* = 0.9 y 20.000 iteraciones

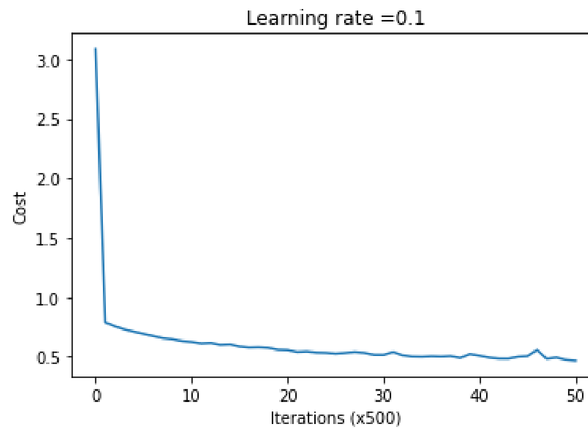

Figura 3-48. Evolución del coste modelo 4 con *lambda* = 0.9 y 25.000 iteraciones

Tabla 3-53. Modelo 4 con *lambda* = 1.2 y 20.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	83.95174140678352 %
Pruebas	67.27438468550593 %

Tabla 3-54. Modelo 4 con *lambda* = 1.2 y 25.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	85.02162531299795 %
Pruebas	61.62260711030082 %

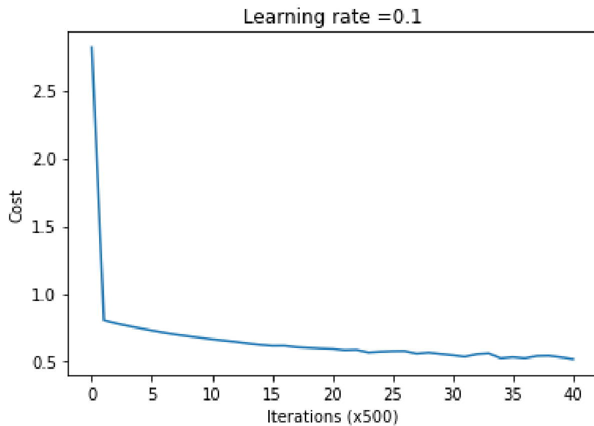

Figura 3-49. Evolución del coste modelo 4 con *lambda* = 1.2 y 20.000 iteraciones

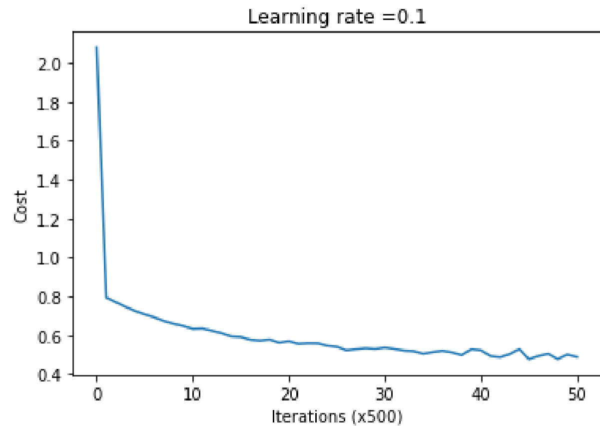

Figura 3-50. Evolución del coste modelo 4 con *lambda* = 1.2 y 25.000 iteraciones

Se puede comprobar que en las 20.000 iteraciones ocurre la mayor mejora en los resultados de ambos sets, mientras que a las 25.000 se observa una gran caída en el porcentaje de aciertos del conjunto de pruebas. La aplicación del método *L2 Regularization* claramente ha aportado una mejora la efectividad del modelo, ya todos los resultados superan el 60% y hay algunas configuraciones que alcanzan el 68% de acierto. Aun así, el principal objetivo, reducir la varianza, no se ha conseguido por lo que se requerirá el empleo de otro método de regularización.

3.8 Quinto modelo

En el quinto modelo se mantiene el objetivo que se tenía en el cuarto: reducir la diferencia de acierto entre ambos conjuntos de datos, la varianza. Pero en este se abordará utilizando el método *dropout*, descrito en la sección 2.6.2.

Este método se basa en la modificación de la estructura de la red neuronal aleatoriamente en cada iteración, y para su implementación habrá que modificar los elementos del modelo implicados en el desplazamiento de los datos sobre esta: la propagación hacia delante y la propagación hacia atrás.

Propagación hacia delante con *dropout*

```
def forward_prop_with_dropout(X, parameters, keep_prob):

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]
    W4 = parameters["W4"]
    b4 = parameters["b4"]

    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)

    D1 = np.random.rand(A1.shape[0], A1.shape[1])
    D1 = (D1 < keep_prob).astype(int)
    A1 = np.multiply(A1, D1)
    A1 = A1 / keep_prob
```

```
    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)

    D2 = np.random.rand(A2.shape[0], A2.shape[1])
    D2 = (D2 < keep_prob).astype(int)
    A2 = np.multiply(A2, D2)
    A2 = A2 / keep_prob

    Z3 = np.dot(W3, A2) + b3
    A3 = relu(Z3)

    D3 = np.random.rand(A3.shape[0], A3.shape[1])
    D3 = (D3 < keep_prob).astype(int)
    A3 = np.multiply(A3, D3)
    A3 = A3 / keep_prob

    Z4 = np.dot(W4, A3) + b4
    A4 = softmax(Z4)

    cache = (Z1, A1, D1, W1, b1, Z2, A2, D2, W2, b2, Z3, A3, D3, W3,
                                                  b3, Z4, A4, W4, b4)

    return A4, cache
```

Propagación hacia atrás con *dropout*

```
def back_prop_with_dropout(X, Y, cache, keep_prob):

    Z1, A1, D1, W1, b1, Z2, A2, D2, W2, b2, Z3, A3, D3, W3, b3, Z4,
                                                   A4, W4, b4 = cache
    m = X.shape[1]

    dZ4 = A4 - Y
    dW4 = 1/m * np.dot(dZ4, A3.T)
    db4 = 1/m * np.sum(dZ4, axis=1, keepdims=True)

    dA3 = np.dot(W4.T, dZ4)
    dA3 = np.multiply(dA3, D3)
    dA3 = dA3 / keep_prob
    dZ3 = np.multiply(dA3, np.int64(A3 > 0))
    dW3 = 1/m * np.dot(dZ3, A2.T)
    db3 = 1/m * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dA2 = np.multiply(dA2, D2)
    dA2 = dA2 / keep_prob
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1/m * np.dot(dZ2, A1.T)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dA1 = np.multiply(dA1, D1)
    dA1 = dA1 / keep_prob
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
```

```
    dW1 = 1/m * np.dot(dZ1, X.T)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1" : dW1, "db1" : db1, "dZ1" : dZ1, "dA1" : dA1,
             "dW2" : dW2, "db2" : db2, "dZ2" : dZ2, "dA2" : dA2,
             "dW3" : dW3, "db3" : db3, "dZ3" : dZ3, "dA3" : dA3,
             "dW4" : dW4, "db4" : db4, "dZ4" : dZ4}

    return grads
```

Estos cambios en los elementos se verán reflejados en un nuevo modelo descrito a continuación.

Modelo 5

```
def model_5(X, Y, learning_rate = 0.1, num_iter = 20010,
                                   keep_prob = 0.5, print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layers_dims = [X.shape[0], 25, 25, 16, 6]

    parameters = init_params_he(layers_dims)

    for i in range(num_iter):

        A4, cache = forward_prop_with_dropout(X, parameters,
                                                         keep_prob)

        cost = compute_cost(A4, Y)

        gradients = back_prop_with_dropout(X, Y, cache, keep_prob)

        parameters = update_parameters(parameters, gradients,
                                                    learning_rate)

        if print_cost and i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
        if print_cost and i % 1000 == 0:
            costs.append(cost)


    plt.plot(costs)
    plt.ylabel('Cost')
    plt.xlabel('Iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters, costs
```

Queda así creado el nuevo modelo con *dropout*. Este trae un nuevo hiperparámetro que se define *keep_prob* que indica la probabilidad de una neurona de mantenerse encendida. Se establece este valor por defecto en 0.5

y antes de intentar ajustar se prueba el nuevo modelo para observar los resultados que ofrece. Se realizará un entrenamiento durante 30.000 iteraciones

Tabla 3-55. Coste modelo 5 con *keep_prob* = 0.5

Iteración	Coste
0	4.576361387450361
5.000	0.8241192923329835
20.000	0.8154211265517065
25.000	0.8033411627747206

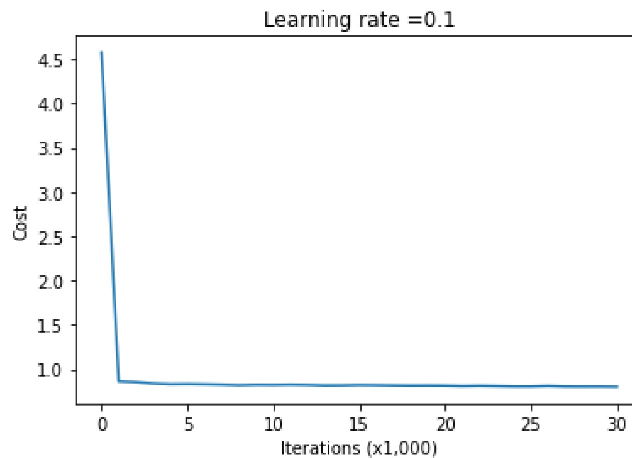

Figura 3-51. Evolución del coste modelo 5 con *keep_prob* = 0.5

Tabla 3-56. Resultado modelo 5 con *keep_prob* = 0.5

Conjunto de datos	Porcentaje de acierto
Entrenamiento	73.54882768040064 %
Pruebas	77.30173199635370 %

Como se puede observar este nuevo modelo trae muchos cambios en el comportamiento de los datos. Lo primero es que vemos un aumento muy grande en el porcentaje de acierto del conjunto de pruebas, un aumento de casi 10 puntos porcentuales con respecto al mejor resultado del modelo anterior. Sin embargo, ha habido una caída aún mayor en el resultado del conjunto de entrenamiento. De hecho, se da un caso curioso donde el modelo trabaja mejor con datos nuevos que con datos con los que ha entrenado. Esta es una característica que puede ocurrir en dropout debido a que su funcionamiento provoca que los pesos se repartan y se reduzcan. Esto mejora enormemente la objetividad frente a nuevos datos, pero da lugar a estos fenómenos.

Otra diferencia que se puede apreciar es la forma que toma el gráfico, el cual muestra un descenso vertiginoso en las primeras 1000 iteraciones, pero después se estabiliza y apenas cambia. Esto contrasta frente a los modelos anteriores que, aunque también tenían descensos acusados al principio, sí conservaban una línea descendente durante el proceso de minimización de la función de coste.

A continuación, se entrenará este modelo con diferentes configuraciones para poder entender estos resultados y ver cómo varían los resultados según los hiperparámetros *keep_prob* y el número de iteraciones.

Todas las variaciones del modelo van a entrenarse sobre el mismo conjunto de datos y, en este caso, se incluirán los gráficos para cada valor de *keep_prob* diferente y el mayor número de intervenciones sobre el que se entrene ya que se considera de interés para la comprensión del funcionamiento.

Tabla 3-57. Modelo 5 con *keep_prob* = 0.5 y 20.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	73.54882768040064 %
Pruebas	77.30173199635370 %

Tabla 3-58. Modelo 5 con *keep_prob* = 0.5 y 25.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	73.54882768040064 %
Pruebas	77.30173199635370 %

Tabla 3-59. Modelo 4 con *keep_prob* = 0.5 y 30.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	73.54882768040064 %
Pruebas	77.30173199635370 %

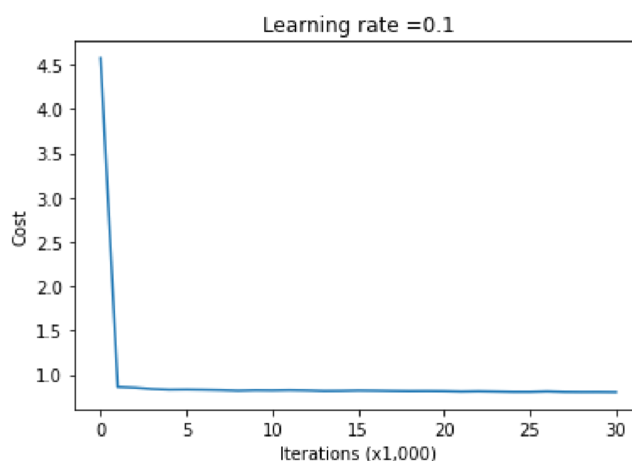

Figura 3-52. Evolución del coste modelo 5 con *keep_prob* = 0.5

Tabla 3-60. Modelo 5 con *keep_prob* = 0.65 y 20.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	73.54882768040064 %
Pruebas	77.30173199635370 %

Tabla 3-61. Modelo 5 con *keep_prob* = 0.65 y 25.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	73.5715911677669 %
Pruebas	77.2105742935278 %

Tabla 3-62. Modelo 4 con *keep_prob* = 0.65 y 30.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	73.75369906669702 %
Pruebas	77.11941659070192 %

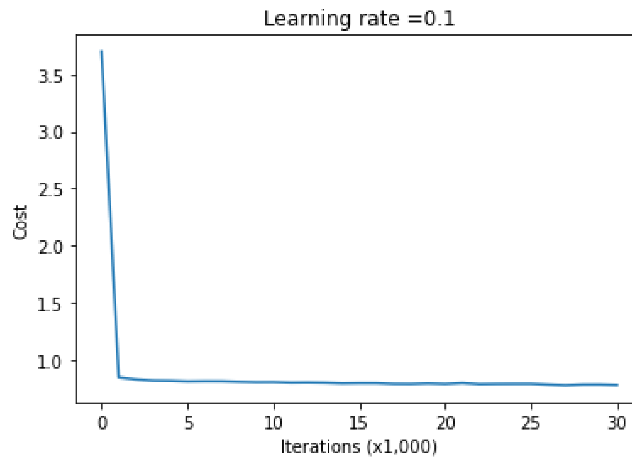

Figura 3-53. Evolución del coste modelo 5 con *keep_prob* = 0.65

Tabla 3-63. Modelo 5 con *keep_prob* = 0.8 y 20.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	73.79922604142954 %
Pruebas	76.93710118505014 %

Tabla 3-64. Modelo 5 con *keep_prob* = 0.8 y 25.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	74.36831322558616 %
Pruebas	76.57247037374658 %

Tabla 3-65. Modelo 4 con *keep_prob* = 0.8 y 30.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	74.98292738447531 %
Pruebas	77.02825888787602 %

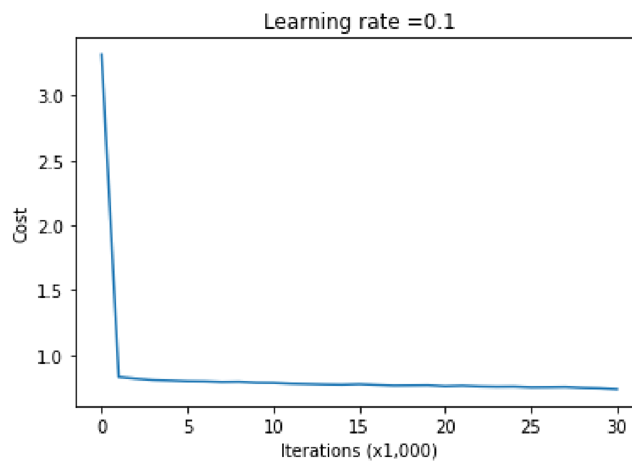

Figura 3-54. Evolución del coste modelo 5 con *keep_prob* = 0.8

Tabla 3-66. Modelo 5 con *keep_prob* = 0.9 y 20.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	75.36990666970180 %
Pruebas	76.20783956244302 %

Tabla 3-67. Modelo 5 con *keep_prob* = 0.9 y 25.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	75.11950830867289 %
Pruebas	76.11668185961714 %

Tabla 3-68. Modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	76.32597313908491 %
Pruebas	76.20783956244302 %

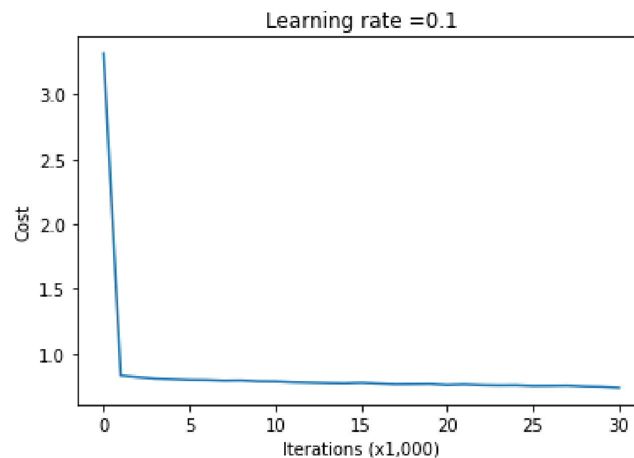

Figura 3-55. Evolución del coste modelo 5 con *keep_prob* = 0.9

Comparando estos resultados se puede comprobar que prácticamente todas las variaciones del modelo tienen el mismo comportamiento en lo que respecta al gráfico de la evolución del coste. Si se observan los modelos con *keep_prob* = 0.5 se halla algo bastante extraño y es que los 3 modelos tienen los mismos resultados. Son 3 modelos diferentes, con inicializaciones de pesos diferentes y entrenados durante un número de iteraciones diferentes, pero dan exactamente la misma predicción de los datos. Este comportamiento parece como si la utilización del *dropout* provocase un tipo de saturación en el rendimiento de los modelos que no puede ser mejorado.

En cambio, si se observan los siguientes modelos, con valores de *keep_prob* mayores, se comprueba que esta saturación desaparece cuanto mayor es este valor. Sin embargo, solo desaparece en términos absolutos pero la línea general de comportamiento se conserva y los resultados no llegan a variar más de 1 punto porcentual en el conjunto de pruebas. Por otro lado, el conjunto de entrenamiento sí llega a mejorar algo más de 2 puntos porcentuales.

Y con esto se llega al último modelo, entrenado durante 30.000 iteraciones con un *keep_prob* = 0.9, dando un resultado en el conjunto de entrenamiento del 76.33% y en el conjunto de prueba del 76.20%. Con este modelo se puede decir que se ha conseguido el principal objetivo que era alcanzar el error obtenido por un experto en el campo el cuál se había establecido en 23.5%, dejando un acierto del 76.50%. Este modelo cumple con las medidas ya que presenta un error del 0.17% frente a este objetivo y una varianza del 0.13%.

Para comprobar el éxito del modelo se realizan varios entrenamientos sobre diferentes conjuntos de datos, de esta manera se puede tener una visión más objetiva de los resultados obtenidos. Tras realizar el entrenamiento y comprobación sobre 3 conjuntos de datos diferentes estos son los resultados que muestran.

Tabla 3-69. Modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones sobre C.D. 1

Conjunto de datos	Porcentaje de acierto
Entrenamiento	78.96653767357159 %
Pruebas	72.83500455788514 %

Tabla 3-70. Modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones sobre C.D. 2

Conjunto de datos	Porcentaje de acierto
Entrenamiento	76.32597313908491 %
Pruebas	74.56700091157703 %

Tabla 3-71. Modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones sobre C.D. 3

Conjunto de datos	Porcentaje de acierto
Entrenamiento	77.23651263373549 %
Pruebas	72.47037374658158 %

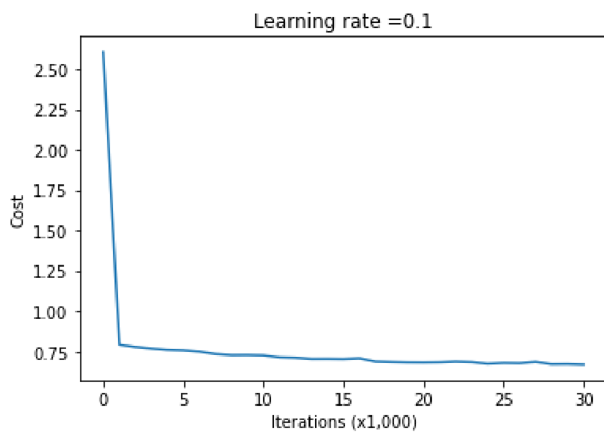

Figura 3-56. Evolución del coste modelo4 con *keep_prob* = 0.9 y 30.000 iteraciones sobre C.D. 1

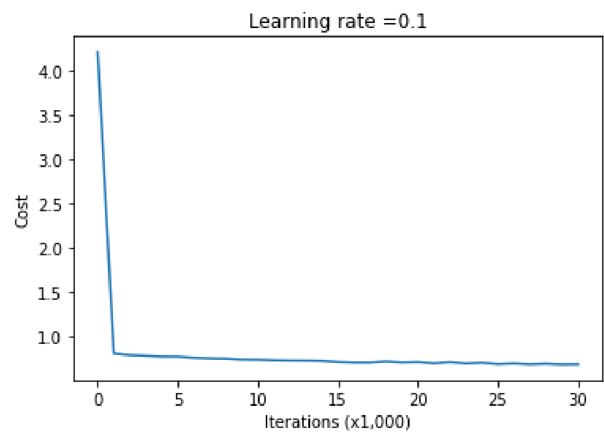

Figura 3-57. Evolución del coste modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones sobre C.D. 2

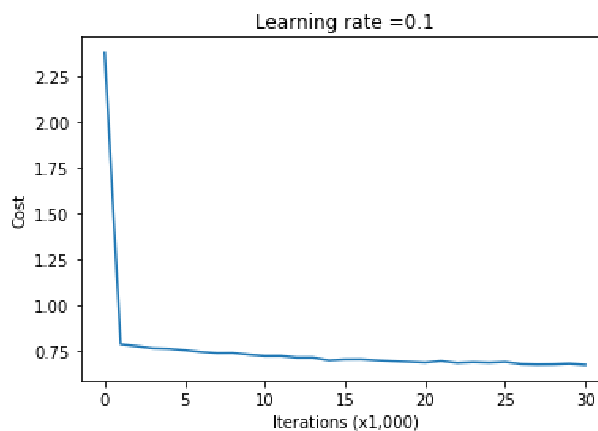

Figura 3-58. Evolución del coste modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones sobre C.D. 3

Haciendo la media de los 4 entrenamientos, el primero más estos tres, se tiene un resultado menor del que se había tenido en un primero momento. Igualmente, el resultado es muy bueno y está por encima de cualquier resultado obtenido hasta ahora, como se puede ver en la siguiente tabla.

Tabla 3-72. Media de los resultados modelo 4 con *keep_prob* = 0.9 y 30.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	77.21374914636923 %
Pruebas	74.02005469462170 %

Buscando mejorar los resultados, se modifican ligeramente las condiciones iniciales estableciendo un *keep_prob* con valor de 0.87.

Tabla 3-73. Modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 1

Conjunto de datos	Porcentaje de acierto
Entrenamiento	75.87070339175962 %
Pruebas	73.38195077484048 %

Tabla 3-74. Modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 2

Conjunto de datos	Porcentaje de acierto
Entrenamiento	76.50808103801502 %
Pruebas	74.0200546946217 %

Tabla 3-75. Modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 3

Conjunto de datos	Porcentaje de acierto
Entrenamiento	77.12269519690417 %
Pruebas	73.92889699179581 %

Tabla 3-76. Modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 4

Conjunto de datos	Porcentaje de acierto
Entrenamiento	74.73252902344639 %
Pruebas	76.84594348222424 %

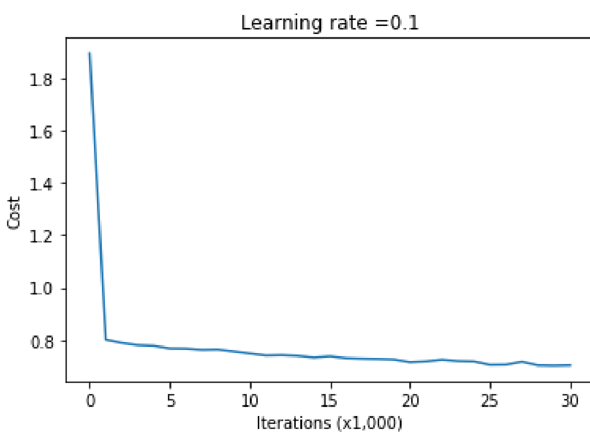

Figura 3-59. Evolución del coste modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 1

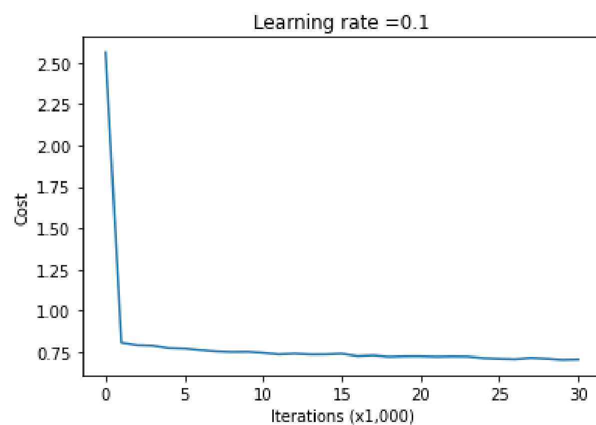

Figura 3-60. Evolución del coste modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 2

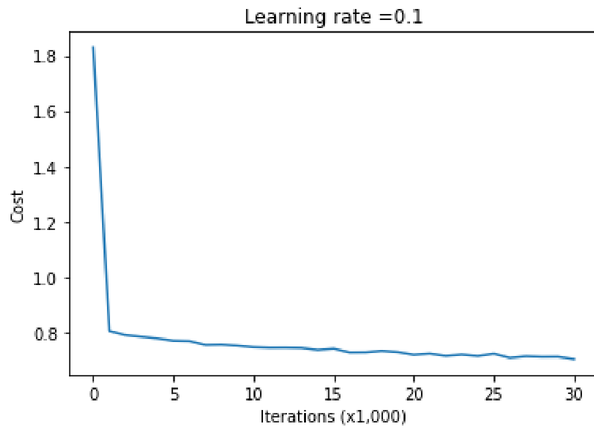

Figura 3-61. Evolución del coste modelo4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 3

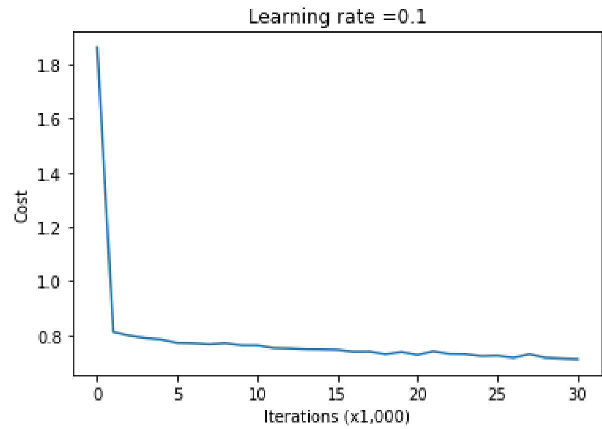

Figura 3-62. Evolución del coste modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones sobre C.D. 4

Este nuevo modelo aumenta ligeramente el porcentaje de acierto en el conjunto de pruebas llegando al 74.54%, superando así la anterior configuración, pero desciende su acierto en el conjunto de entrenamiento, 76.05%. Los resultados pueden tener dos lecturas en este sentido pero sin duda es un modelo que consigue unos resultados muy superiores a los anteriores modelos desarrollados hasta este punto.

Tabla 3-77. Media de los resultados modelo 4 con *keep_prob* = 0.87 y 30.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	76.05850216253130 %
Pruebas	74.54421148587056 %

Estos resultados están muy cerca del objetivo marcado del 76.50%. Sin embargo, también se ha hablado de que este error debería ser superado o, al menos, debería intentar superarse, dada la gran capacidad de cálculo que ofrecen las redes neuronales de aprendizaje profundo. Por lo que se diseñarán nuevos modelos que intenten implementar nuevos métodos que aumenten estos resultados.

Antes de eso, se quiere poner a prueba un poco más este modelo, dado que la cantidad de entrenamiento hasta ahora no ha producido *overfitting*. Tanto el modelo de *keep_prob* = 0.9 como el de valor *keep_prob*= 0.8 tienden al alza si se analizan sus datos. Como el modelo con *keep_prob* = 0.9 ya llegó al puto de ser mayor el rendimiento en el conjunto de entrenamiento, se decide utilizar un nuevo valor para el modelo y se establece un *keep_prob* = 0.85. Se realiza para estos dos modelos un entrenamiento durante 100.000 iteraciones para revisar cómo evolucionan ante estas nuevas condiciones.

- *keep_prob* = 0.8 y 100.000 iteraciones

Tabla 3-78. Coste modelo 5 con *keep_prob* = 0.8 y 100.000 iteraciones

Iteración	Coste
0	4.220516587753388
10.000	0.7879451221176813
20.000	0.760614217636909
30.000	0.7527524178708573
40.000	0.7417325356118082
50.000	0.7342251268992233
60.000	0.7300249418147665
70.000	0.724321855648457
80.000	0.7230635762456924
90.000	0.722823081204253

100.000	0.7145092720894943

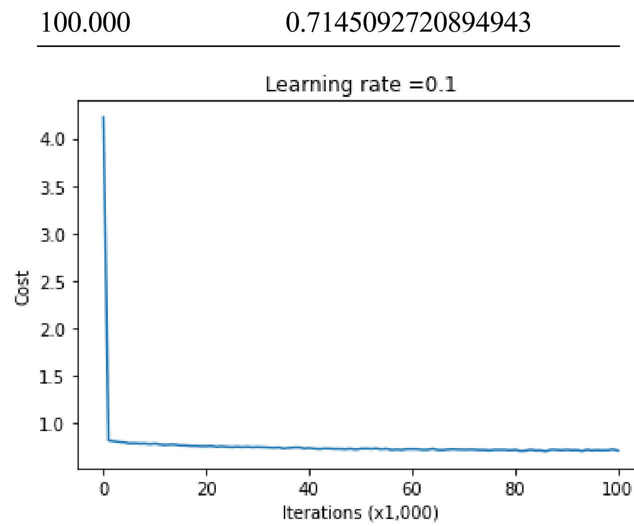

Figura 3-63. Evolución del coste modelo 5 con *keep_prob* = 0.8 y 100.000 iteraciones

Tabla 3-79. Resultado modelo 5 con *keep_prob* = 0.8 y 100.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	74.07238788982472 %
Pruebas	76.93710118505014 %

- *keep_prob* = 0.85 y 100.000 iteraciones

Tabla 3-80. Coste modelo 5 con *keep_prob* = 0.85 y 100.000 iteraciones

Iteración	Coste
0	2.1768734557262026
10.000	0.7681268125431406
20.000	0.7501877703508307
30.000	0.7302616238724877
40.000	0.7253562481533302
50.000	0.7083251937171967
60.000	0.7115675997239183
70.000	0.693345859602339
80.000	0.6959127430117654
90.000	0.6909588399626102
100.000	0.6977315667211936

Figura 3-64. Evolución del coste modelo 5 con *keep_prob* = 0.85 y 100.000 iteraciones

Tabla 3-81. Resultado modelo 5 con *keep_prob* = 0.85 y 100.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	76.28044616435238 %
Pruebas	77.11941659070192 %

Se observa que el modelo con *keep_prob* = 0.85 presenta buenos resultados pese a en las últimas 40.000 iteraciones su coste oscila sin llegar a descender. Esta oscilación no ocurre en el modelo con *keep_prob* = 0.8 pero sin embargo presenta resultados peores que cuando se entrenó hasta las 30.000 iteraciones. Esto muestra una dificultad a la hora de realizar la minimización del coste y ajustar los parámetros del modelo al conjunto de datos de entrenamiento, algo muy característico del método *dropout*. Por lo que se decide implementar algoritmos de optimización para acelerar este proceso y conseguir sobreponer los impedimentos que conlleva el *dropout*.

3.9 Sexto modelo

En este sexto modelo se buscará lograr un mejor resultado en el conjunto de entrenamiento aumentando la capacidad de minimizar la función de coste mediante el algoritmo de optimización *gradient descent with momentum*. Como se comentó en la sección 2.7.1, este algoritmo consiste en añadir un momento de inercia al proceso de minimización de la función de coste.

Para utilizar este algoritmo habrá que crear una nueva función para iniciar las variables necesarias y modificar el elemento relativo a la actualización de los parámetros de la red neuronal.

Inicialización variables *GD with momentum*

```
def init_momentum_var(parameters):

    L = len(parameters) // 2
    v = {}
    for l in range(L):

        v["dW" + str(l+1)] = np.zeros((parameters["W" +
                                              str(l+1)].shape))

        v["db" + str(l+1)] = np.zeros((parameters["b" +
                                              str(l+1)].shape))
```

```
    return v
```

Actualización de pesos con *GD with momentum*

```
def upd_params_with_momentum(parameters, gradients, v, beta,
                                                learning_rate):

    L = len(parameters) // 2

    for l in range(L):

        v["dW" + str(l+1)] = beta * v["dW" + str(l+1)] + (1-beta) *
                                        gradients["dW" + str(l+1)]

        v["db" + str(l+1)] = beta * v["db" + str(l+1)] + (1-beta) *
                                        gradients["db" + str(l+1)]

        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                                    learning_rate * v["dW" + str(l+1)]

        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
                                    learning_rate * v["db" + str(l+1)]

    return parameters, v
```

Estos cambios se verán reflejados en el modelo de la siguiente manera:

Modelo 6

```
def model_6(X, Y, learning_rate = 0.1, num_iter = 100010, keep_prob =
                                0.5, beta = 0.9, print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layers_dims = [X.shape[0], 25, 25, 16, 6]

    parameters = init_params_he(layers_dims)

    v = init_momentum_var(parameters)

    for i in range(num_iter):

        A4, cache = forward_prop_with_dropout(X, parameters,
                                                    keep_prob)

        cost = compute_cost(A4, Y)

        gradients = back_prop_with_dropout(X, Y, cache, keep_prob)

        parameters, v = upd_params_with_momentum(parameters,
                                gradients, v, beta, learning_rate)
```

```
        if print_cost and i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
        if print_cost and i % 1000 == 0:
            costs.append(cost)


    plt.plot(costs)
    plt.ylabel('Cost')
    plt.xlabel('Iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters, costs
```

Una vez creado el modelo se procede a ponerlo en práctica y ver sus resultados. Se realiza el entrenamiento del modelo definiendo un *keep_prob* = 0.87 y durante 100.000 iteraciones. Además, este algoritmo introduce un nuevo hiperparámetro, β, que también habrá que ajustar. En este primer entrenamiento se decide establecer un valor de 0.9 para este hiperparámetro.

Tabla 3-82. Coste modelo 6 con *keep_prob* = 0.87, β = 0.9 y 100.000 iteraciones

Iteración	Coste
0	3.6682932944727282
10.000	0.7348760592094877
20.000	0.6947067737650212
30.000	0.6710045057594355
40.000	0.6796963578221235
50.000	0.6661576428670876
60.000	0.6612193805729207
70.000	0.6547580649351932
80.000	0.650825748524587
90.000	0.6587300725276016
100.000	0.6454662819348047

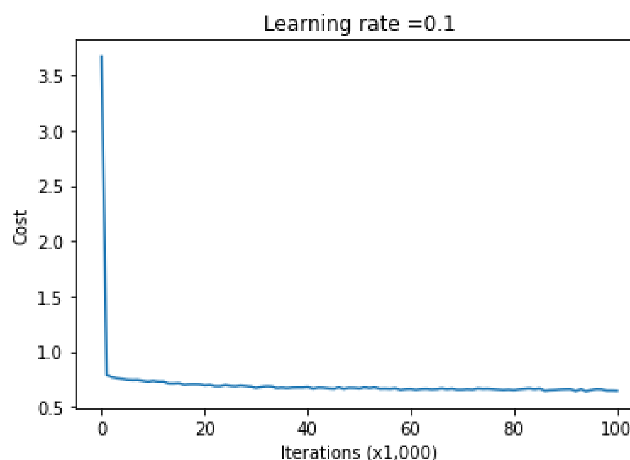

Figura 3-65. Evolución del coste modelo 6 con *keep_prob* = 0.87, β = 0.9 y 100.000 iteraciones

Tabla 3-83. Resultado modelo 6 con *keep_prob* = 0.87, β = 0.9 y 100.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	78.39745048941498 %
Pruebas	72.01458523245214 %

Se comprueba que el *gradient descent with momentum* ha mejorado la capacidad del modelo de minimizar el coste de una manera constante hasta llegar a un coste inferior al 0.65, pero esto también ha afectado a los resultados obtenidos. Se procede a realizar pruebas de diferentes configuraciones del modelo para verificar estos datos.

Tabla 3-84. Modelo 6 con *keep_prob* = 0.87, β = 0.9 y 80.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	77.66901889369452 %
Pruebas	70.10027347310848 %

Tabla 3-85. Modelo 6 con *keep_prob* = 0.87, β = 0.85 y 80.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	78.42021397678124 %
Pruebas	70.55606198723792 %

Tabla 3-86. Modelo 6 con *keep_prob* = 0.87, β = 0.8 y 80.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	78.60232187571136 %
Pruebas	71.28532360984503 %

Tabla 3-87. Modelo 6 con *keep_prob* = 0.85, β = 0.88 y 100.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	77.44138402003187 %
Pruebas	71.10300820419325 %

Tabla 3-88. Modelo 6 con *keep_prob* = 0.88, β = 0.92 y 100.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	79.21693603460050 %
Pruebas	70.37374658158614 %

Tabla 3-89. Modelo 6 con *keep_prob* = 0.86, β = 0.83 y 100.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	79.21693603460050 %
Pruebas	70.37374658158614 %

El entrenamiento de estos modelos ha durado 92 minutos en total. Tras diferentes ajustes de los hiperparámetros se comprueba que este modelo no produce una mejora consistente de los resultados obtenidos con el modelo 5. Mientras sí consigue aumentar ligeramente los resultados en el conjunto de entrenamiento, lo cuál es la función principal de los algoritmos de optimización, la gran caída que tienen los resultados en el conjunto de pruebas no compensa el cambio de modelo.

Por lo tanto, se decide modificar la estructura de la red neuronal. Se crean dos modelos nuevos con estructuras [17, 64, 32, 16, 6] y [17, 60, 40, 20, 6]. Ambos modelos siguen siendo redes neuronales de 4 capas pero en este caso el número de neuronas en cada capa oculta ha aumentado considerablemente. Se realizan diferentes configuraciones de los hiperparámetros y se entrenan estos nuevos modelos esperando mejoras en sus resultados.

En estos entrenamientos se mantendrá constante el hiperparámetro keep_prob con un valor de 0.87 y se entrenarán los modelos durante 100.000 iteraciones. El hiperparámetro que variará será el *learning rate*.

Tabla 3-90. Modelo 6 con α = 0.1 y estructura [17, 64, 32, 16, 6]

Conjunto de datos	Porcentaje de acierto
Entrenamiento	89.98406555884362 %
Pruebas	68.36827711941659 %

Tabla 3-91. Modelo 6 con α = 0.05 y estructura [17, 64, 32, 16, 6]

Conjunto de datos	Porcentaje de acierto
Entrenamiento	87.66218984748464 %
Pruebas	68.64175022789426 %

Tabla 3-92. Modelo 6 con α = 0.01 y estructura [17, 64, 32, 16, 6]

Conjunto de datos	Porcentaje de acierto
Entrenamiento	76.73571591167767 %
Pruebas	71.01185050136737 %

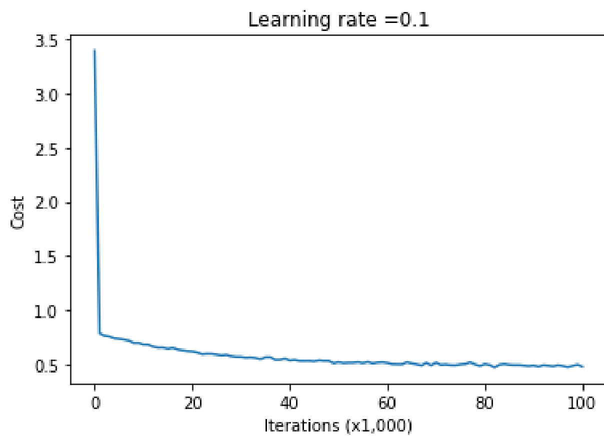

Figura 3-66. Evolución del coste modelo 6 con α = 0.1 y estructura [17, 64, 32, 16, 6]

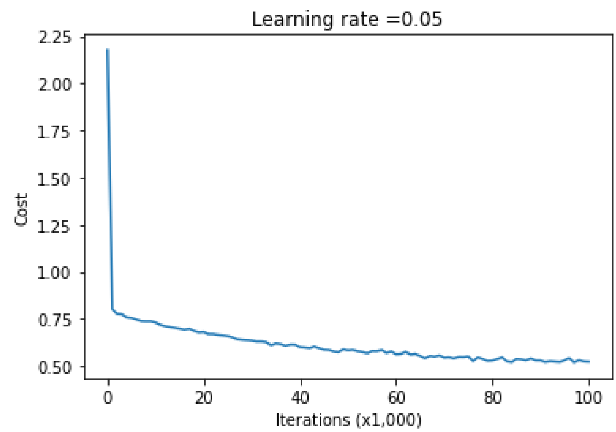

Figura 3-67. Evolución del coste modelo 6 con α = 0.05 y estructura [17, 64, 32, 16, 6]

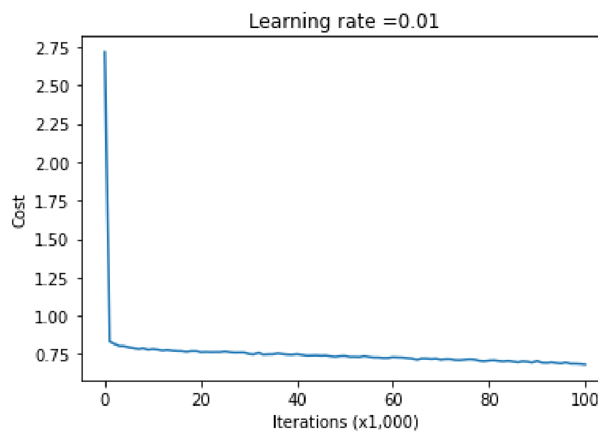

Figura 3-68. Evolución del coste modelo 6 con α = 0.01 y estructura [17, 64, 32, 16, 6]

Tabla 3-93. Modelo 6 con α = 0.1 y estructura [17, 60, 40, 20, 6]

Conjunto de datos	Porcentaje de acierto
Entrenamiento	91.09947643979057 %
Pruebas	67.73017319963537 %

Tabla 3-94. Modelo 6 con α = 0.05 y estructura [17, 60, 40, 20, 6]

Conjunto de datos	Porcentaje de acierto
Entrenamiento	88.43614841793763 %
Pruebas	68.55059252506837 %

Tabla 3-95. Modelo 6 con α = 0.01 y estructura [17, 60, 40, 20, 6]

Conjunto de datos	Porcentaje de acierto
Entrenamiento	77.16822217163669 %
Pruebas	71.83226982680037 %

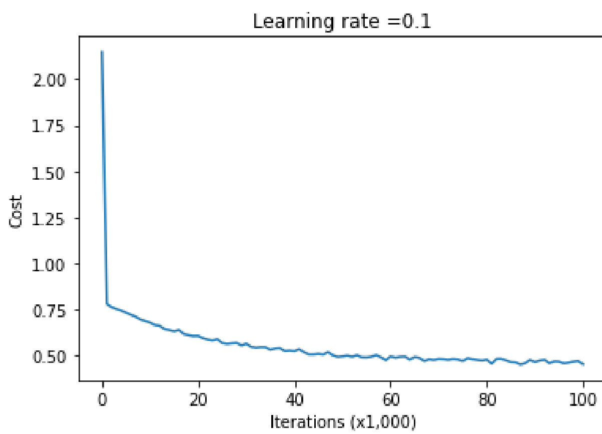

Figura 3-69. Evolución del coste modelo 6 con α = 0.1 y estructura [17, 60, 40, 20, 6]

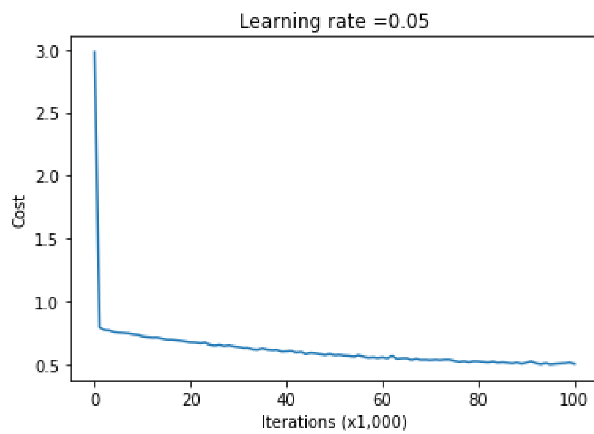

Figura 3-70. Evolución del coste modelo 6 con α = 0.05 y estructura [17, 60, 40, 20, 6]

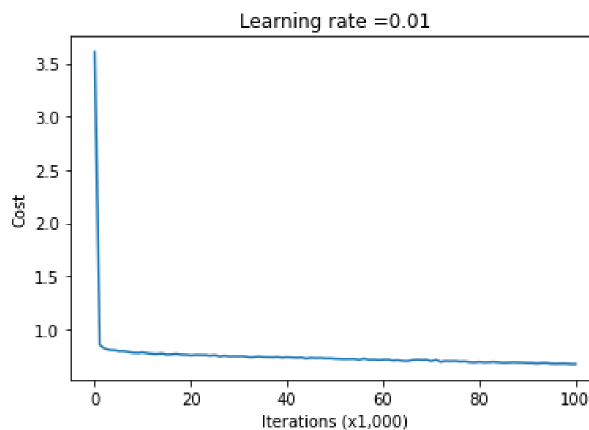

Figura 3-71. Evolución del coste modelo 6 con α = 0.01 y estructura [17, 60, 40, 20, 6]

Se puede observar la gran mejora en los resultados sobre el conjunto de entrenamiento, en algunos casos superior al 90%. Por otro lado, la caída en el porcentaje de aciertos sobre el conjunto de pruebas es aún más agravada, dejando un *overtfitting* muy elevado, cuando ya se están utilizando algoritmos de regularización. En estos modelos se puede ver el gran impacto que tiene el *learning rate* en la optimización de una red neuronal y

cómo debe ser muy tenido en cuenta el proceso de ajustar este hiperparámetro. Además, el entrenamiento de estos 6 modelos duró 316 minutos. Esto hace muy costoso el entrenamiento si lo comparamos con los 92 minutos que fueron necesarios para entrenar las 6 primeras versiones de este modelo obteniendo resultados similares.

No se logran resultados que mejoren lo anterior en el ámbito global por lo que se decide abandonar esta vía de investigación y tratar de lograr mejor suerte utilizando otro algoritmo de optimización.

3.10 Séptimo modelo

Tras no haber logrado los resultados esperados con el modelo anterior se decide utilizar un algoritmo de optimización más potente para tratar de conseguir un cambio en el rendimiento del modelo sobre los conjuntos de datos. El algoritmo que se utilizará será el Adam, *adaptative moment estimation*, visto en la sección 2.7.3. Unifica el *gradient descent with momentum* utilizado en el anterior modelo con el *RMSprop*, explicado en la sección 2.7.2, obteniendo una capacidad de optimización superior a la de estos algoritmos por separado.

La implementación de este algoritmo implica la creación de una nueva función que inicialice las variables utilizadas por el Adam y la modificación de la función dedicada a actualizar los pesos de la red neuronal.

Inicializar variables Adam

```
def init_adam(parameters):

    L = len(parameters) // 2

    v = {}
    s = {}

    for l in range(L):

        v["dW" + str(l+1)] = np.zeros((parameters["W" +
                                                str(l+1)].shape))

        v["db" + str(l+1)] = np.zeros((parameters["b" +
                                                str(l+1)].shape))

        s["dW" + str(l+1)] = np.zeros((parameters["W" +
                                                str(l+1)].shape))

        s["db" + str(l+1)] = np.zeros((parameters["b" +
                                                str(l+1)].shape))

    return v, s
```

Actualización de pesos con Adam

```
def upd_params_with_adam(parameters, gradients, v, s, t,
                         learning_rate, beta1, beta2, epsilon = 1e-8):

    L = len(parameters) // 2

    v_corr = {}
    s_corr = {}

    for l in range(L):
```

```
        v["dW" + str(l+1)] = beta1 * v["dW" + str(l+1)] + (1-beta1) *
                                          gradients["dW" + str(l+1)]

        v["db" + str(l+1)] = beta1 * v["db" + str(l+1)] + (1-beta1) *
                                          gradients["db" + str(l+1)]


        s["dW" + str(l+1)] = beta2 * s["dW" + str(l+1)] + (1-beta2) *
                              np.multiply(gradients["dW" + str(l+1)],
                                         gradients["dW" + str(l+1)])

        s["db" + str(l+1)] = beta2 * s["db" + str(l+1)] + (1-beta2) *
                              np.multiply(gradients["db" + str(l+1)],
                                         gradients["db" + str(l+1)])


        v_corr["dW" + str(l+1)] = v["dW" + str(l+1)] / (1 -
                                                       (beta1**t))

        v_corr["db" + str(l+1)] = v["db" + str(l+1)] / (1 -
                                                       (beta1**t))


        s_corr["dW" + str(l+1)] = s["dW" + str(l+1)] / (1 -
                                                       (beta2**t))

        s_corr["db" + str(l+1)] = s["db" + str(l+1)] / (1 -
                                                       (beta2**t))


        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                           learning_rate * (v_corr["dW" + str(l+1)] /
                         (np.sqrt(s_corr["dW" + str(l+1)] + epsilon)))

        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
                           learning_rate * (v_corr["db" + str(l+1)] /
                         (np.sqrt(s_corr["db" + str(l+1)] + epsilon)))

    return parameters, v, s
```

Una vez creadas las funciones correspondientes de desarrolla el nuevo modelo.

Modelo 7

```
def model_7(X, Y, learning_rate = 0.1, num_iter = 50010, beta1 = 0.9,
                                 beta2 = 0.999, print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layers_dims = [X.shape[0], 25, 25, 16, 6]
```

```
    parameters = init_params_he(layers_dims)

    v, s = init_adam(parameters)

    t = 0

    for i in range(num_iter):

        A4, cache = forward_prop_with_dropout(X, parameters,
                                                          keep_prob)

        cost = compute_cost(A4, Y)

        gradients = back_prop_with_dropout(X, Y, cache, keep_prob)

        t = t + 1

        parameters, v, s = upd_params_with_adam(parameters,
                                     gradients, v, s, t, learning_rate,
                                          beta1, beta2, epsilon = 1e-8)

        if print_cost and i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
        if print_cost and i % 1000 == 0:
            costs.append(cost)


    plt.plot(costs)
    plt.ylabel('Cost')
    plt.xlabel('Iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters, costs
```

Este algoritmo de optimización es mucho más potente con lo que serán necesarias muchas menos iteraciones sobre el conjunto de datos de entrenamiento para llegar a unos resultados objetivo. Esto se puede ver a continuación en los diferentes modelos que se han puesto a prueba para ajustar los hiperparámetros de la red neuronal. Los nuevos hiperparámetros a definir son β_1 y β_2, además de los ya utilizados previamente *keep_prob*, *learning rate* y el número de iteraciones. Mientras que el hiperparámetro β_2 se mantendrá en un valor constante de 0.999 y el *keep_prob* se mantendrá en 0.87, se realizarán modificaciones sobre los otros 3 para encontrar la mejor configuración.

La estructura se utilizará la misma que se viene estableciendo en los últimos modelos [17, 25, 25, 16, 6] dado que es la que mejores resultados ha dado con el menor coste computacional.

Tabla 3-96. Modelo 7 con $\alpha = 0.001$, $\beta_1 = 0.9$ y 20.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	77.03164124743911 %
Pruebas	72.92616226071102 %

Tabla 3-97. Modelo 7 con $\alpha = 0.001$, $\beta_1 = 0.9$ y 40.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	77.82836330525836 %
Pruebas	71.83226982680037 %

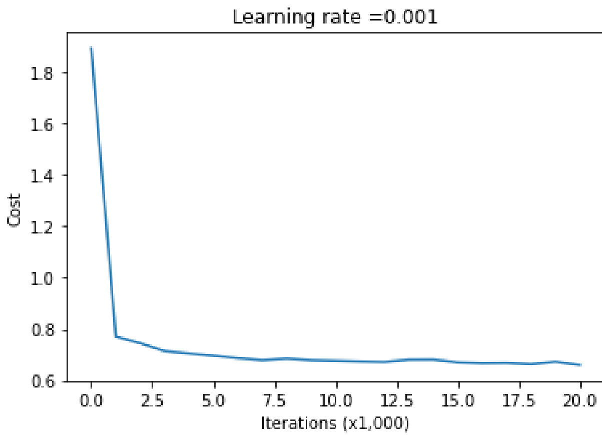

Figura 3-72. Evolución del coste modelo 7 con α = 0.001, β_1 = 0.9 y 20.000 iteraciones

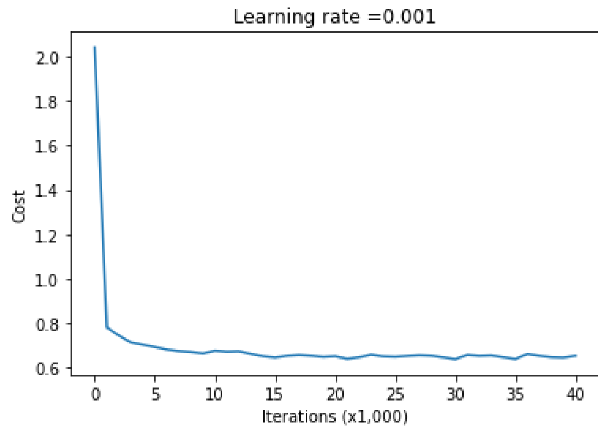

Figura 3-73. Evolución del coste modelo 7 con α = 0.001, β_1 = 0.9 y 40.000 iteraciones

Tabla 3-98. Modelo 7 con α = 0.001, β_1 = 0.8 y 20.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	78.37468700204872 %
Pruebas	72.10574293527803 %

Tabla 3-99. Modelo 7 con α = 0.001, β_1 = 0.8 y 40.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	79.2169360346005 %
Pruebas	72.74384685505926 %

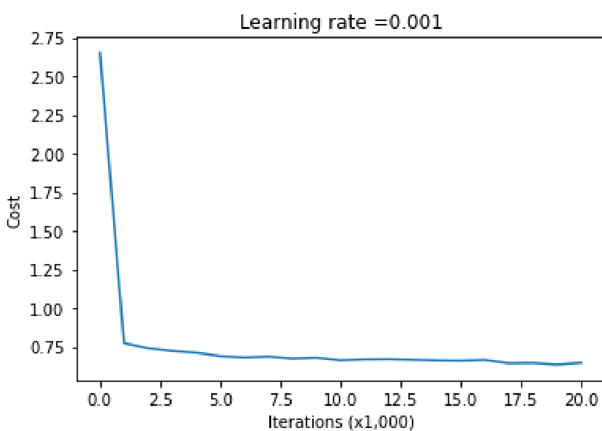

Figura 3-74. Evolución del coste modelo 7 con α = 0.001, β_1 = 0.8 y 20.000 iteraciones

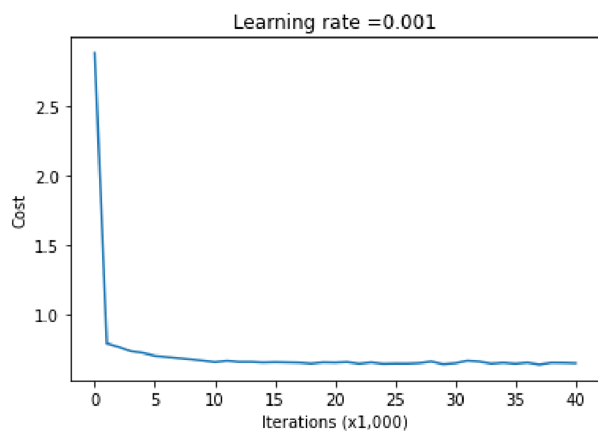

Figura 3-75. Evolución del coste modelo 7 con α = 0.001, β_1 = 0.8 y 40.000 iteraciones

Tabla 3-100. Modelo 7 con α = 0.005, β_1 = 0.8 y 20.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	79.62667880719326 %
Pruebas	70.7383773928897 %

Tabla 3-101. Modelo 7 con α = 0.005, β_1 = 0.8 y 40.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	80.26405645344867 %
Pruebas	72.28805834092981 %

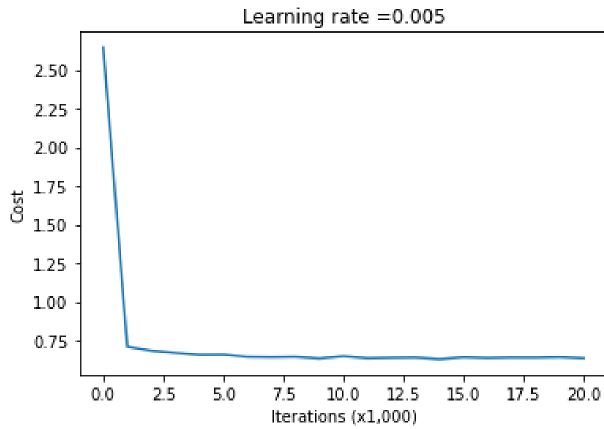

Figura 3-76. Evolución del coste modelo 7 con α = 0.005, β_1 = 0.8 y 20.000 iteraciones

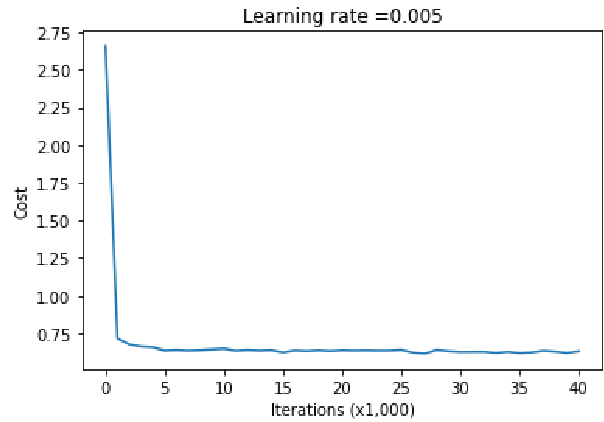

Figura 3-77. Evolución del coste modelo 7 con α = 0.005, β_1 = 0.8 y 40.000 iteraciones

Después de analizar los resultados de los diferentes modelos entrenados se decide continuar las pruebas con la primera, la cuarta y la sexta variación. Los tres obtienen los mejores resultados en el conjunto de pruebas mientras mantienen un alto resultado en el conjunto de entrenamiento. Lo que ha mejorado enormemente es la eficiencia, para entrenar estos 6 modelos solamente se han necesitado 29 minutos obteniendo resultados similares o superiores al anterior modelo, que necesitaba 92 minutos para ello.

A continuación, se realizará el entrenamiento de estos tres modelos sobre cuatro conjuntos de datos diferentes para realizar una media entre sus resultados y tener una medición algo más objetiva de su rendimiento ante nuevos conjuntos de datos.

Tabla 3-102. Modelo 7 con α = 0.001, β_1 = 0.9 y 20.000 iteraciones sobre C.D. 1

Conjunto de datos	Porcentaje de acierto
Entrenamiento	80.37787389027999 %
Pruebas	73.92889699179581 %

Tabla 3-103. Modelo 7 con α = 0.001, β_1 = 0.9 y 20.000 iteraciones sobre C.D. 2

Conjunto de datos	Porcentaje de acierto
Entrenamiento	79.64944229455952 %
Pruebas	72.19690063810392 %

Tabla 3-104. Modelo 7 con α = 0.001, β_1 = 0.9 y 20.000 iteraciones sobre C.D. 3

Conjunto de datos	Porcentaje de acierto
Entrenamiento	80.03642157978602 %
Pruebas	70.92069279854147 %

Tabla 3-105. Modelo 7 con α = 0.001, β_1 = 0.9 y 20.000 iteraciones sobre C.D. 4

Conjunto de datos	Porcentaje de acierto
Entrenamiento	77.80559981789210 %
Pruebas	73.01731996353692 %

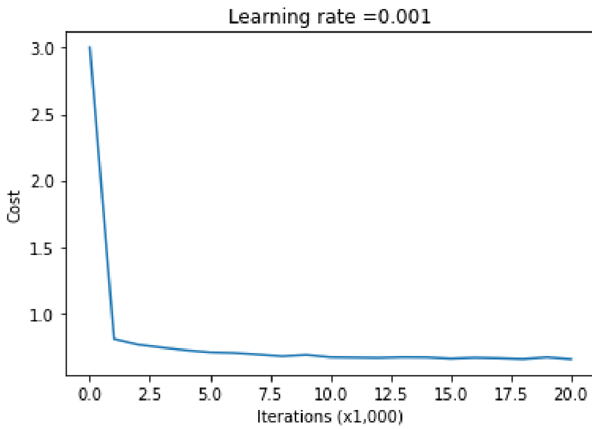

Figura 3-78. Evolución del coste modelo 7 con α = 0.001, β_1 = 0.9 y 20.000 iteraciones sobre C.D. 1

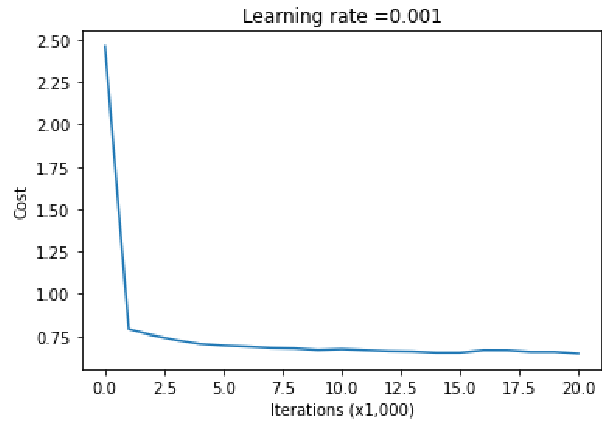

Figura 3-79. Evolución del coste modelo 7 con α = 0.001, β_1 = 0.9 y 20.000 iteraciones sobre C.D. 2

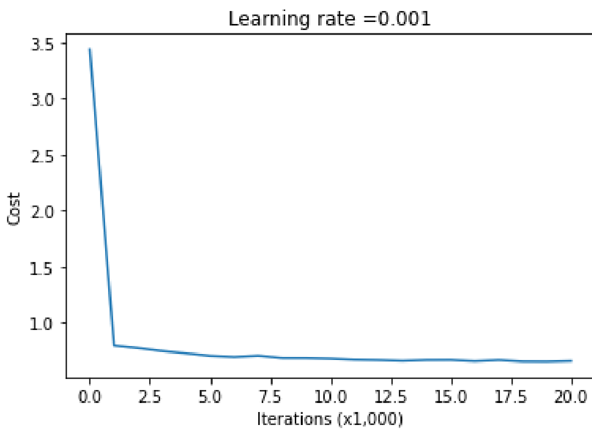

Figura 3-80. Evolución del coste modelo 7 con α = 0.001, β_1 = 0.9 y 20.000 iteraciones sobre C.D. 3

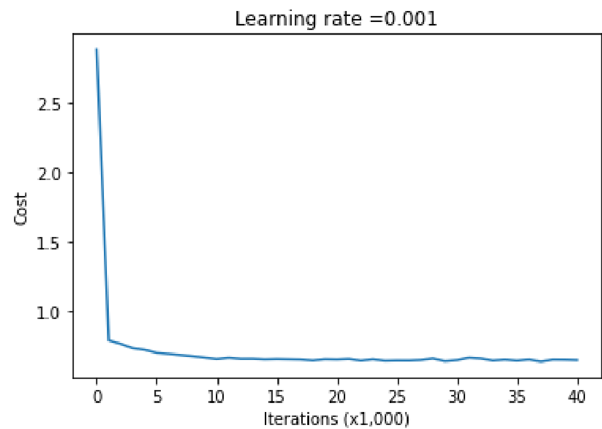

Figura 3-81. Evolución del coste modelo 7 con α = 0.001, β_1 = 0.9 y 20.000 iteraciones sobre C.D. 4

Tabla 3-106. Modelo 7 con α = 0.001, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 1

Conjunto de datos	Porcentaje de acierto
Entrenamiento	78.23810607785112 %
Pruebas	74.02005469462170 %

Tabla 3-107. Modelo 7 con α = 0.001, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 2

Conjunto de datos	Porcentaje de acierto
Entrenamiento	78.60232187571136 %
Pruebas	71.92342752962625 %

Tabla 3-108. Modelo 7 con α = 0.001, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 3

Conjunto de datos	Porcentaje de acierto
Entrenamiento	80.55998178921011 %
Pruebas	70.55606198723792 %

Tabla 3-109. Modelo 7 con α = 0.001, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 4

Conjunto de datos	Porcentaje de acierto
Entrenamiento	78.98930116093785 %
Pruebas	72.92616226071102 %

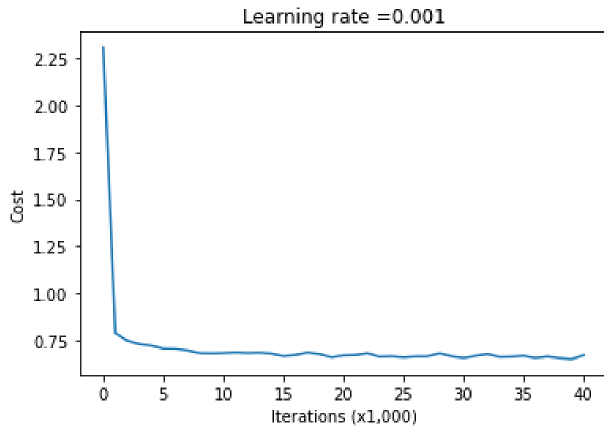

Figura 3-82. Evolución del coste modelo 7 con α = 0.001, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 1

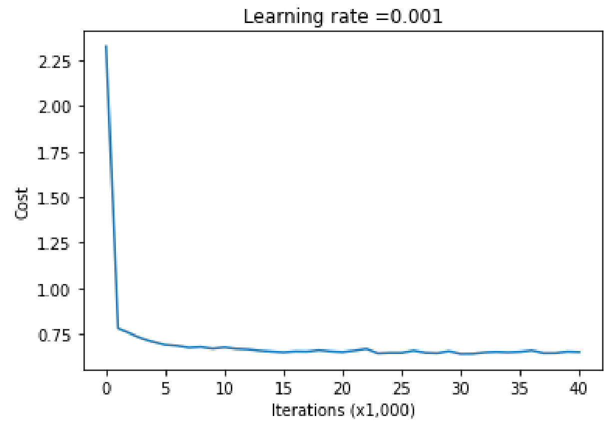

Figura 3-83. Evolución del coste modelo 7 con α = 0.001, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 2

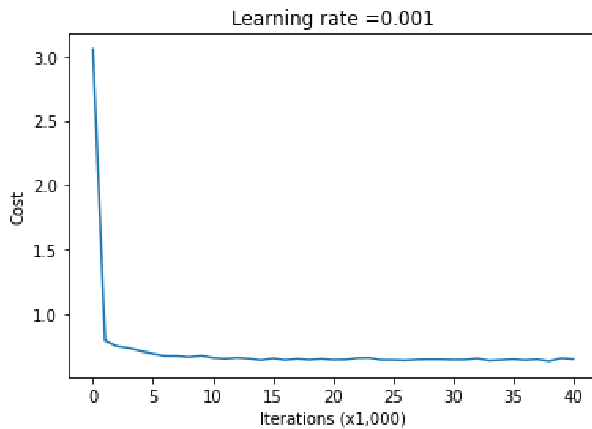

Figura 3-84. Evolución del coste modelo 7 con α = 0.001, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 3

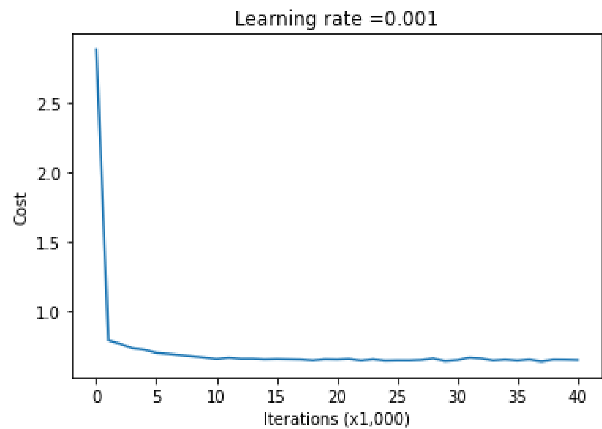

Figura 3-85. Evolución del coste modelo 7 con α = 0.001, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 4

Tabla 3-110. Modelo 7 con α = 0.005, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 1

Conjunto de datos	Porcentaje de acierto
Entrenamiento	80.83314363760528 %
Pruebas	73.01731996353692 %

Tabla 3-111. Modelo 7 con α = 0.005, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 2

Conjunto de datos	Porcentaje de acierto
Entrenamiento	80.46892783974505 %
Pruebas	71.64995442114859 %

Tabla 3-112. Modelo 7 con α = 0.005, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 3

Conjunto de datos	Porcentaje de acierto
Entrenamiento	78.48850443888004 %
Pruebas	71.83226982680037 %

Tabla 3-113. Modelo 7 con α = 0.005, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 4

Conjunto de datos	Porcentaje de acierto
Entrenamiento	80.19576599134988 %
Pruebas	72.19690063810392 %

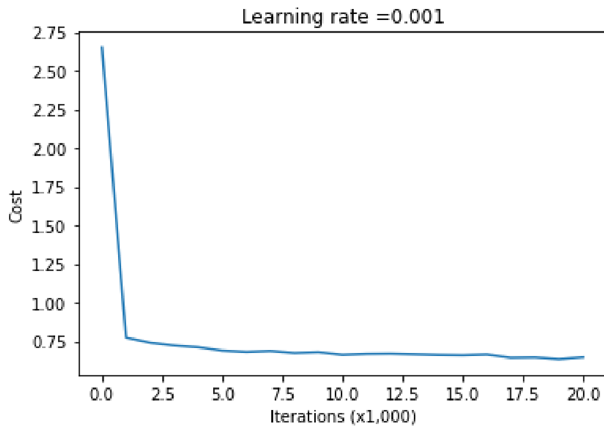

Figura 3-86. Evolución del coste modelo 7 con α = 0.005, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 1

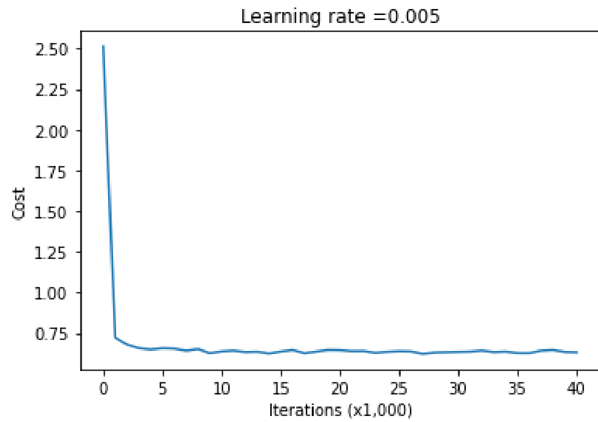

Figura 3-87. Evolución del coste modelo 7 con α = 0.005, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 2

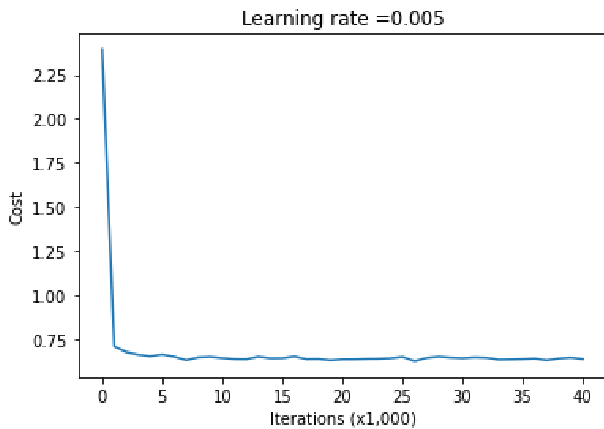

Figura 3-88. Evolución del coste modelo 7 con α = 0.005, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 3

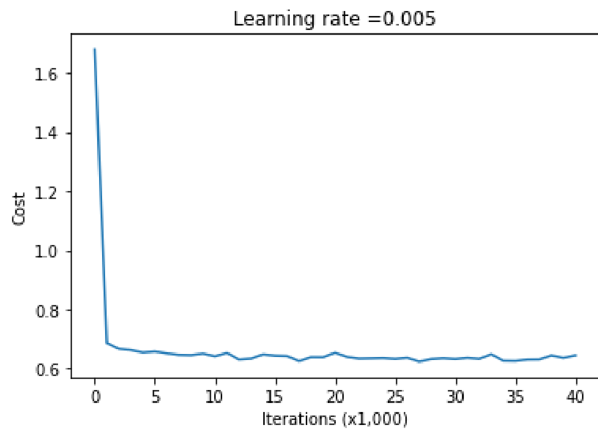

Figura 3-89. Evolución del coste modelo 7 con α = 0.005, β_1 = 0.8 y 40.000 iteraciones sobre C.D. 4

Haciendo la media de los resultados obtenidos se obtienen los siguientes resultados.

Tabla 3-114. Media modelo 7 con α = 0.001, β_1 = 0.9 y 20.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	79.46733439562941 %
Pruebas	72.51595259799454 %

Tabla 3-115. Media modelo 7 con α = 0.001, β_1 = 0.8 y 40.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	79.09742772592760 %
Pruebas	72.35642661804922 %

Tabla 3-116. Media modelo 7 con α = 0.005, β_1 = 0.8 y 40.000 iteraciones

Conjunto de datos	Porcentaje de acierto
Entrenamiento	79.99658547689505 %
Pruebas	72.17411121239745 %

El entrenamiento ha durado 70 minutos y los resultados son consistentemente superiores a los obtenidos del

modelo 6 con un tiempo menor de entrenamiento para un número superior de modelos diferentes. Esto deja en evidencia la superioridad del algoritmo Adam sobre el *gradient descent with momentum*. Sin embargo, si bien se obtienen resultados superiores al modelo 5 en el conjunto de entrenamiento, los resultados sobre el conjunto de pruebas han bajado, pese a estar utilizando *dropout* como método de regularización. Esto pone de manifiesto la dificultad de mejorar los resultados en el conjunto de entrenamiento sin producir *overfitting*.

Para intentar evitar esto se puede aumentar la cantidad de datos de entrenamiento o probar diferentes arquitecturas en la red neuronal. Dado que los datos que se poseen son limitados y no existe la posibilidad de ampliarlos se decide crear nuevas estructuras para analizar sus resultados.

3.11 Octavo modelo

Anteriormente se probó a aumentar el número de neuronas que había en cada capa. En este modelo se aumentarán el número de capas ocultas y se probarán diferentes configuraciones para intentar encontrar el modelo adecuado.

La primera modificación en la estructura será crear una red neuronal de 5 capas, siendo esta de dimensiones [17, 25, 25, 16, 16, 6]. Además, se hace una modificación en el modelo para que realice una comprobación de los resultados obtenidos con ambos conjuntos de datos cada 10.000 iteraciones y así seguir más de cerca su evolución.

Se prueban 3 diferentes modelos, en los que únicamente varía el hiperparámetro *keep_prob*. El learning rate se mantiene constante en 0.1 y los 3 modelos se entrenarán durante 60.000 iteraciones. Se utiliza el método *dropout* como regularizador. A continuación, se muestran los resultados obtenidos con estos modelos.

Tabla 3-117. Resultados modelo 8 con $\alpha = 0.1$, *keep_prob* = 0.5

Iteración	Conjunto de datos	Porcentaje de acierto
10.000	Entrenamiento	74.25449578875484 %
	Pruebas	74.56700091157703 %
20.000	Entrenamiento	74.25449578875484 %
	Pruebas	74.56700091157703 %
30.000	Entrenamiento	74.25449578875484 %
	Pruebas	74.56700091157703 %
40.000	Entrenamiento	74.25449578875484 %
	Pruebas	74.56700091157703 %
50.000	Entrenamiento	74.27725927612110 %
	Pruebas	74.56700091157703 %
60.000	Entrenamiento	74.27725927612110 %
	Pruebas	74.56700091157703 %

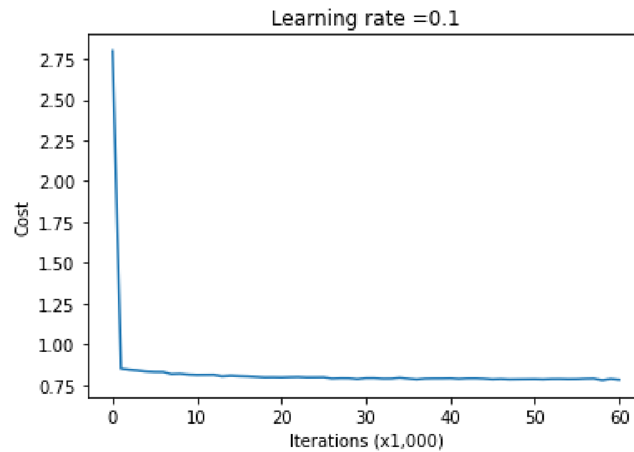

Figura 3-90. Evolución del coste modelo 8 con α = 0.1, *keep_prob* = 0.5

Tabla 3-118. Resultados modelo 8 con α = 0.1, *keep_prob* = 0.75

Iteración	Conjunto de datos	Porcentaje de acierto
10.000	Entrenamiento	74.25449578875484 %
	Pruebas	74.56700091157703 %
20.000	Entrenamiento	74.27725927612110 %
	Pruebas	74.56700091157703 %
30.000	Entrenamiento	74.41384020031869 %
	Pruebas	74.38468550592525 %
40.000	Entrenamiento	75.16503528340541 %
	Pruebas	74.47584320875114 %
50.000	Entrenamiento	75.50648759389939 %
	Pruebas	74.65815861440291 %
60.000	Entrenamiento	75.75688595492830 %
	Pruebas	74.20237010027347 %

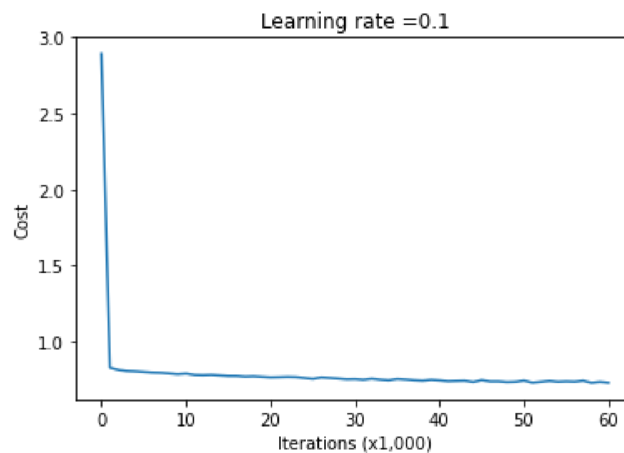

Figura 3-91. Evolución del coste modelo 8 con α = 0.1, *keep_prob* = 0.75

Tabla 3-119. Resultados modelo 8 con α = 0.1, *keep_prob* = 0.87

Iteración	Conjunto de datos	Porcentaje de acierto
10.000	Entrenamiento	74.73252902344639 %
	Pruebas	74.29352780309936 %
20.000	Entrenamiento	76.05281129068974 %
	Pruebas	74.11121239744758 %
30.000	Entrenamiento	77.09993170953790 %
	Pruebas	74.02005469462170 %
40.000	Entrenamiento	77.91941725472342 %
	Pruebas	73.01731996353692 %
50.000	Entrenamiento	79.08035511040291 %
	Pruebas	72.56153144940748 %
60.000	Entrenamiento	79.58115183246073 %
	Pruebas	72.65268915223336 %

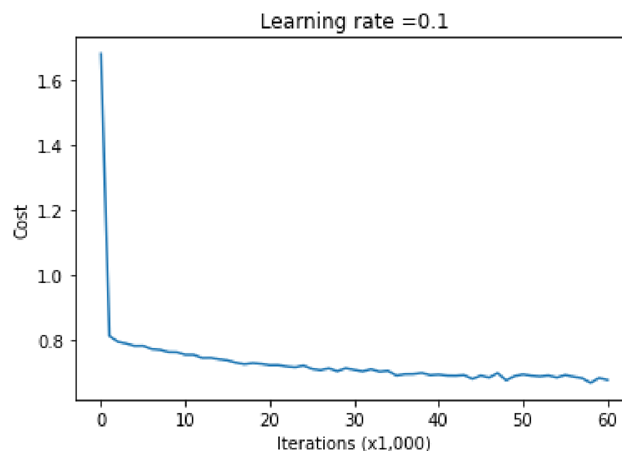

Figura 3-92. Evolución del coste modelo 8 con α = 0.1, *keep_prob* = 0.87

En estos tres modelos pueden observarse características muy determinantes de cada uno de ellos. En el primero se ve la actuación reguladora del método *dropout* que mantiene constante durante más de 40.000 iteraciones el porcentaje de acierto del conjunto de entrenamiento mientras que el conjunto de pruebas se mantiene constante con el mismo resultado desde antes de la iteración 10.000. Esto significa que llega a un óptimo antes de la iteración 10.000 y que luego el hecho de que en cada iteración se apaguen la mitad de las neuronas de cada capa le impide mejorar sus resultados.

Por otro lado, en la segunda variación del modelo se observa como la red neuronal es capaz de aprender mejorando los resultados en ambos conjuntos hasta la iteración 50.000 donde mejora solamente en el conjunto de entrenamiento empeorando en el de pruebas. Se muestra así donde comienza el sobreajuste a los datos del primer conjunto.

En la tercera variación del modelo este sobre ajuste ocurre mucho antes, a partir de las 30.000 iteraciones el modelo solamente aumenta el porcentaje de aciertos en el conjunto de entrenamiento mientras se reduce el del conjunto de pruebas.

Aunque todos los resultados mejoran lo obtenido en el modelo anterior, estos hay que compararlos con los obtenidos en el modelo 5, dado que no se utiliza algoritmo de optimización. Obtiene resultados ligeramente inferiores a este modelo, pero con un mayor coste computacional, dado que su estructura es más grande. Ante esta perspectiva se decide crear un nuevo modelo con un tamaño mayor e investigar otras nuevas estructuras y ver si se obtienen mejores resultados.

3.12 Noveno modelo

Ante la intuición de que el anterior modelo no iba a mejorar los resultados se decide crear una red neuronal de 6 capas y observar si se pueden mejorar los resultados de esta forma. Esta vez se decide utilizar el algoritmo de optimización Adam, además del método de regularización *dropout*. Para comprobar la evolución en el entrenamiento de estas redes neuronales se mantiene la comprobación cada 10.000 iteraciones.

En este caso, se probarán diferentes estructuras para ver cómo afectan estas a la capacidad predictiva del modelo. Se decide establecer un *learning rate* constante de valor 0.1 y un *keep_prob* de 0.87 para los modelos que se entrenarán. Se muestran, a continuación, los resultados de las diferentes variaciones del modelo.

- layers_dims = [17, 32, 32, 25, 25, 16, 6]

Tabla 3-120. Resultados modelo 9 con estructura [17, 32, 32, 25, 25, 16, 6], α = 0.01 y *keep_prob* = 0.87

Iteración	Conjunto de datos	Porcentaje de acierto
10.000	Entrenamiento	85.65900295925336 %
	Pruebas	71.64995442114859 %
20.000	Entrenamiento	86.41019804234009 %
	Pruebas	72.28805834092981 %
30.000	Entrenamiento	85.95492829501480 %
	Pruebas	72.10574293527803 %
40.000	Entrenamiento	86.70612337810153 %
	Pruebas	72.19690063810392 %
50.000	Entrenamiento	87.75324379694970 %
	Pruebas	70.19143117593437 %
60.000	Entrenamiento	86.06874573184612 %
	Pruebas	71.55879671832270 %

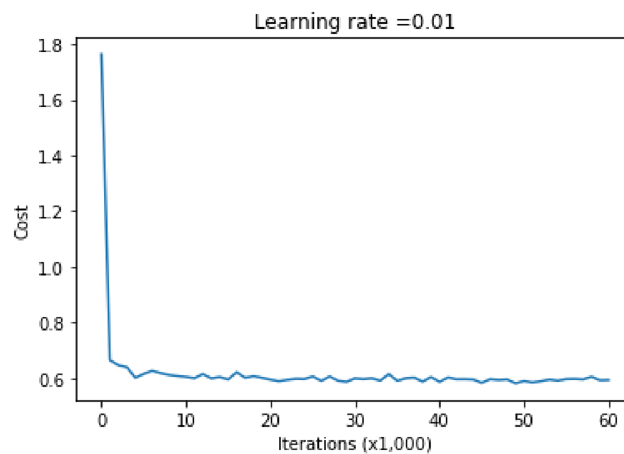

Figura 3-93. Evolución del coste modelo 9 con estructura [17, 32, 32, 25, 25, 16, 6], α = 0.01 y *keep_prob* = 0.87

- layers_dims = [17, 32, 25, 25, 20, 16, 6]

Tabla 3-121. Resultados modelo 9 con estructura [17, 32, 25, 25, 20, 16, 6], α = 0.01 y *keep_prob* = 0.87

Iteración	Conjunto de datos	Porcentaje de acierto
10.000	Entrenamiento	84.15661279307990 %
	Pruebas	69.64448495897904 %
20.000	Entrenamiento	84.77122695196904 %
	Pruebas	68.73290793072014 %
30.000	Entrenamiento	83.81516048258594 %
	Pruebas	70.28258887876025 %
40.000	Entrenamiento	85.97769178238106 %
	Pruebas	67.54785779398359 %
50.000	Entrenamiento	84.31595720464375 %
	Pruebas	68.82406563354604 %
60.000	Entrenamiento	83.79239699521966 %
	Pruebas	70.00911577028259 %

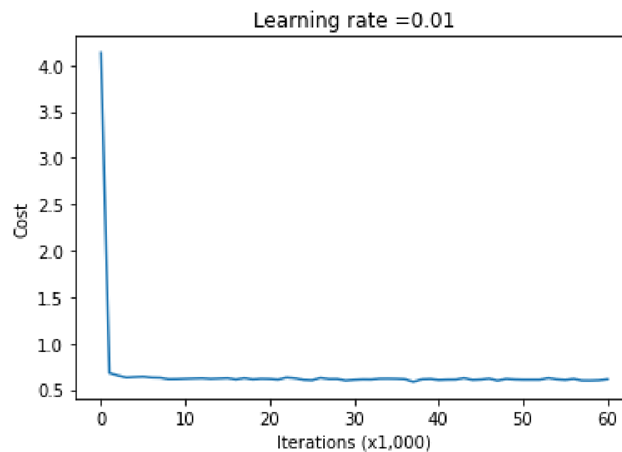

Figura 3-94. Evolución del coste modelo 9 con estructura [17, 32, 25, 25, 20, 16, 6], α = 0.01 y *keep_prob* = 0.87

- layers_dims = [17, 20, 35, 25, 25, 16, 6]

Tabla 3-122. Resultados modelo 9 con estructura [17, 20, 35, 25, 25, 16, 6], α = 0.01 y *keep_prob* = 0.87

Iteración	Conjunto de datos	Porcentaje de acierto
10.000	Entrenamiento	81.60710220805828 %
	Pruebas	71.83226982680037 %
20.000	Entrenamiento	82.03960846801730 %
	Pruebas	72.28805834092981 %
30.000	Entrenamiento	81.69815615752333 %
	Pruebas	71.64995442114859 %
40.000	Entrenamiento	81.40223082176189 %
	Pruebas	73.10847766636280 %

50.000	Entrenamiento	81.10630548600045 %
	Pruebas	73.74658158614403 %
60.000	Entrenamiento	81.22012292283178 %
	Pruebas	73.56426618049225 %

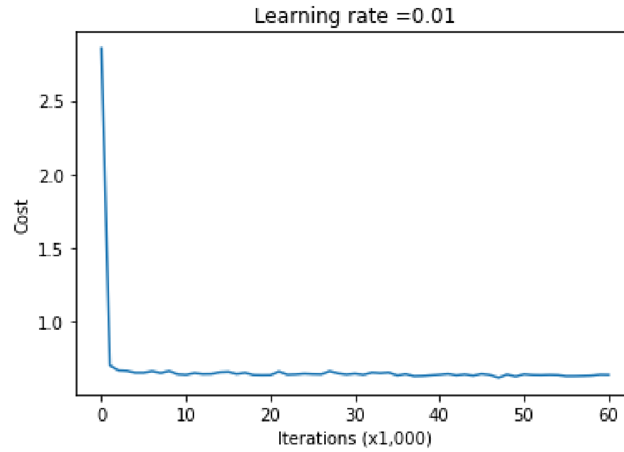

Figura 3-95. Evolución del coste modelo 9 con estructura [17, 20, 35, 25, 25, 16, 6], α = 0.01 y *keep_prob* = 0.87

- layers_dims = [17, 25, 25, 25, 25, 16, 6]

Tabla 3-123. Resultados modelo 9 con estructura [17, 25, 25, 25, 25, 16, 6], α = 0.01 y *keep_prob* = 0.87

Iteración	Conjunto de datos	Porcentaje de acierto
10.000	Entrenamiento	82.49487821534259 %
	Pruebas	70.00911577028259 %
20.000	Entrenamiento	82.67698611427271 %
	Pruebas	69.09753874202370 %
30.000	Entrenamiento	82.69974960163897 %
	Pruebas	70.00911577028259 %
40.000	Entrenamiento	82.65422262690644 %
	Pruebas	69.00638103919782 %
50.000	Entrenamiento	83.10949237423173 %
	Pruebas	70.19143117593437 %
60.000	Entrenamiento	82.99567493740041 %
	Pruebas	69.73564266180492 %

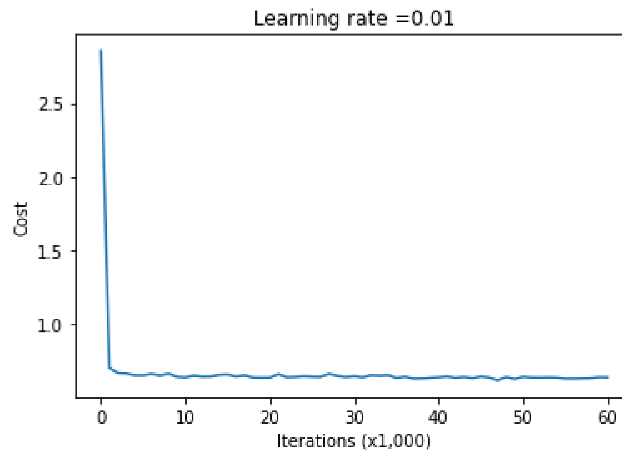

Figura 3-96. Evolución del coste modelo 9 con estructura [17, 20, 35, 25, 25, 16, 6], $\alpha = 0.01$ y *keep_prob* = 0.87

Después de diferentes configuraciones, se obtienen algunos resultados relativamente buenos ya que consigue aumentar bastante el porcentaje de acierto en el conjunto de entrenamiento mientras se mantiene el de pruebas por encima del 70%. En especial la tercera variación, que roza el 74% en el conjunto de pruebas mientras logra un 81.1% en el conjunto de entrenamiento. Con esto se confirma la eficacia de la implementación de algoritmos de optimización en la consecución de aumentar el porcentaje de aciertos en el conjunto de entrenamiento. Aunque sigue penalizando los resultados del conjunto de pruebas siendo más bajo que cuando no se utiliza algoritmo de optimización.

Con este último modelo se da por concluida la investigación en la creación de diferentes estructuras de redes neuronales para optimizar los resultados obtenidos. En el siguiente capítulo se analizarán globalmente todos los resultados y se plantearán futuros caminos a tomar.

4 CONCLUSIONES Y LÍNEAS FUTURAS

A menudo les digo a mis alumnos que no se dejen engañar por el nombre de "inteligencia artificial": no tiene nada de artificial. La IA está hecha por seres humanos, educada a comportarse por seres humanos y, en última instancia, a influir en la vida de los seres humanos y en la sociedad humana.

- Fei-Fei Li -

Tras desarrollar nueve modelos diferentes, con diversas variaciones según el modelo, llega el momento de analizar los resultados e intentar darle un sentido a todos los valores obtenidos en el capítulo anterior. Si bien es cierto que se ha ido comentando cada modelo individualmente, se ve necesario analizar qué ventajas presenta cada uno así como recopilar qué dificultades han aparecido en el camino y cuáles han podido ser resueltas y cómo afrontar las que han quedado por resolver. Para ello se dividirá este capítulo en dos secciones, una que analice lo ocurrido y otra que plantee los siguientes pasos a tomar en pos de lograr un modelo con mejor capacidad predictiva, que logre superar lo aquí expuesto.

4.1 Análisis de los resultados y conclusiones

El objetivo marcado al inicio del tercer capítulo era llegar a alcanzar el error que los datos, cedidos por la Coordinación Autonómica de Trasplantes de Andalucía, mostraban en el absoluto de los casos. Este se definió diferenciando los trasplantes que a día de hoy seguían siendo operativos de los que por unos u otros motivos habían fallado. Estos datos mostraban que el 76.50% de los trasplantes realizados habían sido un éxito.

Se comenzó analizando los datos y eligiendo, entre estos, las variables más significativas según la literatura clínica y la opinión de los expertos sanitarios. Esto arrojó 17 variables de las 77 que formaban el conjunto de datos de los procedimientos. Como en todo registro, no todos los datos se guardan mientras que otros no se llegan a medir en ningún momento, por lo que se tuvo que realizar un tratamiento y cubrir estos valores perdidos realizando correlaciones con otros valores de la tabla. Aunque esto introduce un ligero sesgo desde principio, las redes neuronales se comportan de manera muy robusta ante estas situaciones.

Una vez se tenían los datos preparados para su análisis se creó el primer modelo, un modelo simple que incluyera todos los elementos necesarios de una red neuronal de aprendizaje profundo. Una vez se inició el entrenamiento de este se consiguieron los primeros resultados y se vió como la minimización de la función de coste conseguía que el modelo aprendiese y mejorase con cada iteración. A pesar de esto, se apreciaban las carencias de este modelo cuando se analizaban los gráficos de evolución del coste y se observaba la lentitud de optimizar la función de coste cuanto más se acercaba a un cierto valor.

Para solucionar esto se realizaron diferentes variaciones en la estructura de la red neuronal y se creó un segundo modelo donde, para apreciar mejor el descenso del valor de la función de coste se obtuvieron gráficos que se generaran eliminando las iteraciones iniciales y permitiendo así aumentar la escala. Se comprobó que el

valor del coste seguía descendiendo pese a su lentitud en el proceso y se decidió crear un nuevo modelo que generase un resultado mejor.

Así se creó el tercer modelo, con una red neuronal de 4 capas, frente a las redes neuronales de 3 capas de los modelos anteriores. Probando diferentes arquitecturas se comprobó la gran mejoría que había producido esta nueva estructura y se decidió crear elementos de medición que permitieran entender los resultados de forma más directa. Así, por primera vez en el desarrollo del proyecto, se obtuvo una medida del aprendizaje que estaba realizando la red neuronal. Este situaba al modelo en un porcentaje de acierto que rondaba el 80%. Se confirmaba así que los buenos resultados minimizando la función de coste significaban el aprendizaje de la red neuronal y situaban por encima del objetivo la capacidad de aprender el modelo. Pero este valor no representaba la capacidad predictiva del modelo, ya que esto se debe probar sobre datos no utilizados para su entrenamiento. Y aquí se decide separar el conjunto de datos en un conjunto de entrenamiento y un conjunto de pruebas, con una proporción 80:20. Esto arroja el primer valor real sobre la capacidad predictiva del modelo y devuelve valores del 85% sobre el conjunto de entrenamiento y del 60% sobre el conjunto de pruebas. Esto sí confirmaba la capacidad predictiva del modelo, siendo capaz de acertar un 60% de los resultados de datos que no había visto antes. Sin embargo, estaba lejos del objetivo y se planteó la necesidad de reducir esta variación, que ronda el 25%.

Para ello se crearon los modelos 4 y 5, con estructuras similares al modelo anterior pero utilizando métodos de regularización, con el fin de reducir o eliminar la varianza. El primer método, utilizado en el modelo 4, es el de *L2 Regularization* y se logra mejorar los resultados significativamente. Ya todos los modelos superan el 60% de acierto en el conjunto de pruebas, los mejores rondan un 68% mientras se mantiene el porcentaje de acierto del conjunto de entrenamiento por encima del 80%. Este modelo demuestra la capacidad de mejora que implica los métodos de regularización pero el aquí aplicado no demuestra ser lo suficientemente potente.

Por lo que se crea un nuevo modelo, el modelo 5, utilizando el método de regularización *dropout*. Viendo los resultados no cabe duda de la diferencia de efectividad entre este nuevo método y el empleado en el modelo anterior. Tras probar diferentes configuraciones, ajustando los hiperparámetros implicados, se obtienen resultados que superan ampliamente el 70% de acierto en el conjunto de entrenamiento, si bien es cierto que desciende el porcentaje de acierto del conjunto de entrenamiento a las mismas cifras. Los modelos más prometedores se ponen a prueba sobre diferentes conjuntos de datos para obtener una medida objetiva de sus resultados, obteniendo el primero un 74.02% en el conjunto de pruebas y un 77.21% en el conjunto de entrenamiento. Mientras que el segundo obtuvo un 74.54% en el conjunto de pruebas y un 76.05% en el conjunto de entrenamiento. Estos datos, contrastados entre 4 conjuntos de datos diferentes, presentan una prueba sólida de la capacidad predictiva de estos modelos y establecen una medida muy cercana al objetivo establecido.

Pruebas posteriores de este mismo modelo con muchas más iteraciones de entrenamiento presentan porcentajes de acierto mayores en el conjunto de pruebas, llegando a sobrepasar un 77%, sin embargo muestran resultados peores sobre el conjunto de entrenamiento y no pueden ser tomados como correctos dado que la predicción debe ser concordante en este aspecto. Estos resultados pueden ser explicados por tener un conjunto de pruebas más sencillo de analizar, además de ser este menor en tamaño y, por lo tanto, más fácil de aumentar el porcentaje de aciertos.

Teniendo en cuenta todos estos resultados y, como se menciona en el desarrollo del modelo 5, una saturación cuando se intenta reducir el valor de la función de coste del modelo, se decide aplicar métodos de optimización para lograr sobrepasar este desempeño. Así se crearon los modelos 6 y 7. En el modelo 6 se utiliza la técnica *gradient descent with momentum* y, a pesar de sí lograr el principal objetivo de seguir reduciendo el coste, aumentando así el porcentaje de aciertos en el conjunto de entrenamiento, reduce inequívocamente el resultado sobre el conjunto de pruebas. Debido a la capacidad de optimización que se logra, se ve necesario, por primera vez, reducir el *learning rate* del modelo para evitar el *overfitting*. Siendo cierto que todos los resultados se mantienen por encima del 70% de aciertos en el conjunto de pruebas y se logra un 78% en el conjunto de entrenamiento, no son los resultados esperados y se decide emplear otro método de optimización buscando mejores resultados.

Este método será Adam, *adaptative moment estimation*, y se creará con él el modelo 7. Este método demuestra su capacidad de optimización desde el inicio obligando a reducir aún más el *learning rate*, a un décimo del valor del anterior modelo. Se reduce también el número de iteraciones necesarias a menos de la mitad para alcanzar valores de la función de coste similares o incluso menores. Sin embargo los resultados que se

obtienen son similares a los obtenidos en el modelo anterior, aunque ligeramente superiores. Los resultados sobre el conjunto de entrenamiento rondando el 80% y sobre el conjunto de pruebas superan el 72%. Con diversas pruebas, este modelo demuestra que su capacidad para optimizar los resultados en el conjunto de pruebas es aún mayor, pero esto provocaría *overfitting* y reducción del porcentaje de aciertos sobre el conjunto de entrenamiento. Por lo que es necesario otro método que reduzca la varianza entre los conjuntos de datos. Dado que ya se está utilizando un método regularizador quedaban dos opciones, añadir más datos o probar arquitecturas nuevas. Dado que los datos son limitados y ya se están utilizando todos los posibles se opta por la segunda opción.

Así se crean los modelos 8 y 9. El modelo 8 es una red neuronal de 5 capas, aumentando en una las capas ocultas con respecto al anterior modelo. Se realizan pruebas con diferentes configuraciones de los hiperparámetros del *dropout*, pero no se ven mejoras reales respecto al modelo 5 y tiene un coste computacional mayor, por lo que se abandona en busca de una arquitectura diferente.

El modelo 9 presenta una red neuronal de 6 capas, una más que el modelo anterior, y se aplica desde el inicio tanto *dropout* como Adam. Se prueban diferentes estructuras para encontrar una red neuronal que mejore los resultados anteriores. Se obtienen diferentes medidas y se observa que sí mejora al modelo 6 manteniendo unos resultados sobre el conjunto de pruebas en torno al 72 % pero aumentando el porcentaje de acierto del conjunto de entrenamiento hasta el 85%, superándolo en algunos casos. Sin embargo, no se obtienen mejoras en la capacidad predictiva del modelo sobre el conjunto de pruebas y el coste computacional es prácticamente el doble respecto al modelo 5.

Tras haber desarrollado 9 modelos diferentes, siguiendo las necesidades que se presentaban en cada punto del estudio, se determina que el modelo con mayor capacidad predictiva es el modelo 5 con una configuración de sus hiperparámetros como se reflejan en la Tabla 4-1.

Tabla 4-1. Resultados e hiperparámetros de los mejores modelos predictivos

Modelo	5	5
α	0.1	0.1
keep_prob	0.9	0.87
Iteraciones	30.000	30.000
Entrenamiento	77.21374914636923 %	76.05850216253130 %
Pruebas	74.02005469462170 %	74.54421148587056 %

Sin embargo, los resultados nos muestran que estos modelos difícilmente podrán mejorar en sus resultados, y el modelo más prometedor para entrenar con más datos, y poder así lograr mejores resultados, son el modelo 7 y el modelo 9. Por resultados podría decirse que el 9 es superior pero su alto costo computacional aumentará cuantos más datos necesite procesar y esto es un factor muy a tener en cuenta.

Estos resultados tienen dos visiones diferentes. Por un lado, no se ha logrado llegar al porcentaje de aciertos que un experto en el campo logra, marcado en un 76.50%. Por otro lado, los métodos estadísticos tradicionales utilizados en la actualidad tienen un porcentaje de acierto de entre el 65% y el 70%, siendo muy pocos los que logran acercarse a la segunda cifra. Esto demuestra la gran capacidad predictiva de las redes neuronales de aprendizaje profundo.

Por lo que se ha creado una herramienta más potente que las existentes hasta ahora para analizar los datos de los procedimientos de trasplante de riñon y poder proveer de nueva información al respecto, realizando predicciones más acertadas que cualquier método hasta ahora. Esto se considera un éxito del trabajo y aporta una previsión de grandes mejoras en el futuro.

4.2 Próximos pasos y líneas futuras

Tras haber trabajado con los datos, estudiado diferentes modelos y obtenido diversos resultados se plantean diferentes acciones a tomar, tanto a corto plazo como a medio-largo plazo.

La acción más evidente es la obtención de más datos, en este proyecto se ha trabajado con los resultados de los trasplantes de Andalucía, que representa el 17.8% de la población de España. Trabajando con nuevas

comunidades autónomas se podría conseguir un aumento del 500% de cantidad en los datos de trabajo, eso considerando que el 20% de trasplantes de riñón realizados en España se practica en Andalucía. Esta nueva cantidad de datos lograría, sin duda aumentar la diversidad de los conjuntos de entrenamiento y pruebas logrando un entrenamiento más profundo sin llegar a sobreajustarse a los datos.

Por otro lado, también se plantea el aumento del número de variables empleadas para analizar el conjunto de datos, dado que en este proyecto se han utilizado solamente 17, queda margen de mejora en este aspecto, aunque también requiere de un trabajo muy extenso en el proceso de análisis de estos nuevos campos.

Estas dos últimas vías pasan por otra de obligado cumplimiento en la práctica que es la aplicación de lenguajes de programación que permitan ejecutar el código sobre la GPU. Los últimos modelos estaban teniendo un coste computacional muy elevado y sus entrenamientos estaban entre los 20 minutos y 1 hora, para cada uno. Esto aumentará exponencialmente cuando se añadan nuevos datos y nuevas variables por lo que será necesario utilizar la capacidad computacional que ofrecen las tarjetas gráficas en este aspecto. Actualmente las tarjetas gráficas dedidas de Nvidia son las únicas que permiten este tipo de tratamiento, por lo que será un requerimiento si se quiere analizar muchos modelos y estudiar ampliamente las posibilidades que ofrecen las redes neuronales.

Un futuro camino, que diverge ligeramente del objetivo de este proyecto pero que sería el fin real a la hora de utilizar estos métodos en el ámbito sanitario, sería el establecer un objetivo diferente a la hora de predecir los resultados de los procedimientos de trasplante. En este proyecto se ha intentado lograr predecir el resultado de un trasplante de riñon sin tener en cuenta la temporalidad de estos datos. Un objetivo, que se acercaría a la manera de funcionar de la toma de decisiones en estos casos, sería dividir los datos por fechas, establecer el estado del órgano trasplantadoa 1 año, 2 años y más de 2 años después del procedimiento de trasplante. Diferenciando así nuevas posibilidades y proporcionando al personal sanitario una información mucho más detallada. Esto no ha sido posible llevarlo a cabo en este proyecto por extenderse demasiado del objetivo del trabajo y por la escasa cantidad de datos para realizar estas subdivisiones.

Con estos nuevos enfoques, es presumible que se puedan lograr resultados mejores que los obtenidos en el desarrollo de este proyecto y lograr así aportar una nueva herramienta a los expertos en el campo de trasplantes que les ayuden en su toma de decisiones.

ANEXOS

Anexo A - Modelo 1

Modelo 1

```
def model_1(X, Y, learning_rate = 0.01, num_iter = 30010,
                                                    print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layer_dims = [X.shape[0], 20, 10, 6]

    parameters = init_params_he(layers_dims)

    for i in range(num_iter):

        A3, cache = forward_prop(X, parameters)

        cost = compute_cost(A3, Y)

        gradients = back_prop(X, Y, cache)

        parameters = update_parameters(parameters, gradients,
                                                        learning_rate)

        if print_cost and i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
        if print_cost and i % 1000 == 0:
            costs.append(cost)


    plt.plot(costs)
    plt.ylabel('Cost')
    plt.xlabel('Iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

Inicialización de pesos

```
def init_params(layers_dims):

    np.random.seed(7)
```

```
    parameters = {}
    L = len(layers_dims) - 1

    for l in range(L):

        parameters["W" + str(l+1)] = np.random.randn(
                                    layers_dims[l+1], layers_dims[l])

        parameters["b" + str(l+1)] = np.zeros((layers_dims[l+1], 1))

    return parameters
```

Propagación hacia delante

```
def forward_prop(X, parameters):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]

    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)
    Z3 = np.dot(W3, A2) + b3
    A3 = softmax(Z3)

    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3)

    return A3, cache
```

ReLU

```
def relu(x):

    s = np.maximum(0, x)

    return s
```

Sotmax Classifier

```
def softmax(x):

    expo = np.exp(x)
    exp_sum = np.sum(expo, axis=0)
    s = expo / exp_sum

    return s
```

Función de coste de Softmax classifier

```
def compute_cost(A4, Y, eps=0.00000000001):

    m = Y.shape[1]

    loss = - np.sum(np.multiply(Y, np.log(A4+eps)), axis=0)

    cost = 1/m * np.sum(loss)

    return cost
```

Propagación hacia atrás

```
def back_prop(X, Y, cache):

    Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3 = cache

    m = X.shape[1]

    dZ3 = A3 - Y
    dW3 = 1/m * np.dot(dZ3, A2.T)
    db3 = 1/m * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1/m * np.dot(dZ2, A1.T)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1/m * np.dot(dZ1, X.T)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1" : dW1, "db1" : db1, "dZ1" : dZ1, "dA1" : dA1,
             "dW2" : dW2, "db2" : db2, "dZ2" : dZ2, "dA2" : dA2,
             "dW3" : dW3, "db3" : db3, "dZ3" : dZ3}

    return grads
```

Actualización de pesos

```
def update_parameters(parameters, gradients, learning_rate):

    L = len(parameters) // 2

    for l in range(L):

        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] –
                                  learning_rate * gradients["dW" + str(l+1)]

        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] –
                                  learning_rate * gradients["db" + str(l+1)]
```

```
    return parameters
```

Anexo B - Modelo 3

Modelo 3

```
def model_3(X, Y, learning_rate = 0.1, num_iter = 30010,
                                                    print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layers_dims = [X.shape[0], 25, 25, 16, 6]

    parameters = init_params_he(layers_dims)

    for i in range(num_iter):

        A4, cache = forward_prop(X, parameters)

        cost = compute_cost(A4, Y)

        gradients = back_prop(X, Y, cache)

        parameters = update_parameters(parameters, gradients,
                                                   learning_rate)

        if print_cost and i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
        if print_cost and i % 1000 == 0:
            costs.append(cost)


    plt.plot(costs)
    plt.ylabel('Cost')
    plt.xlabel('Iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters, costs
```

Inicialización de He

```
def init_params_he(layers_dims):

    np.random.seed()
```

```
    parameters = {}
    L = len(layers_dims) - 1

    for l in range(L):

        parameters["W" + str(l+1)] = np.random.randn(
                                  layers_dims[l+1], layers_dims[l]) *
                                          np.sqrt(2 / layers_dims[l])
        parameters["b" + str(l+1)] = np.zeros((layers_dims[l+1], 1))

    return parameters
```

Propagación hacia delante

```
def forward_prop(X, parameters):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]
    W4 = parameters["W4"]
    b4 = parameters["b4"]

    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)
    Z3 = np.dot(W3, A2) + b3
    A3 = relu(Z3)
    Z4 = np.dot(W4, A3) + b4
    A4 = softmax(Z4)

    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3, Z4, A4,
                                                               W4, b4)

    return A4, cache
```

Función de coste de Softmax classifier

```
def compute_cost(A4, Y, eps=0.00000000001):

    m = Y.shape[1]

    loss = - np.sum(np.multiply(Y, np.log(A4+eps)), axis=0)

    cost = 1/m * np.sum(loss)

    return cost
```

Propagación hacia atrás

```
def back_prop(X, Y, cache):

    Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3, Z4, A4, W4, b4 =
                                                                  cache
    m = X.shape[1]

    dZ4 = A4 - Y
    dW4 = 1/m * np.dot(dZ4, A3.T)
    db4 = 1/m * np.sum(dZ4, axis=1, keepdims=True)

    dA3 = np.dot(W4.T, dZ4)
    dZ3 = np.multiply(dA3, np.int64(A3 > 0))
    dW3 = 1/m * np.dot(dZ3, A2.T)
    db3 = 1/m * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1/m * np.dot(dZ2, A1.T)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1/m * np.dot(dZ1, X.T)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1" : dW1, "db1" : db1, "dZ1" : dZ1, "dA1" : dA1,
             "dW2" : dW2, "db2" : db2, "dZ2" : dZ2, "dA2" : dA2,
             "dW3" : dW3, "db3" : db3, "dZ3" : dZ3, "dA3" : dA3,
             "dW4" : dW4, "db4" : db4, "dZ4" : dZ4}

    return grads
```

Actualización de pesos

```
def update_parameters(parameters, gradients, learning_rate):

    L = len(parameters) // 2

    for l in range(L):

        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                             learning_rate * gradients["dW" + str(l+1)]

        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
                             learning_rate * gradients["db" + str(l+1)]

    return parameters
```

Anexo C - Modelo 4

Modelo 4

```
def model_4(X, Y, learning_rate = 0.1, num_iter = 25010, lambd = 0,
                                                    print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layers_dims = [X.shape[0], 25, 25, 16, 6]

    parameters = init_params_he(layers_dims)

    for i in range(num_iter):

        A4, cache = forward_prop(X, parameters)

        cost = compute_cost_with_L2reg(A4, Y, parameters, lambd)

        gradients = back_prop_with_L2reg(X, Y, cache, lambd)

        parameters = update_parameters(parameters, gradients,
                                                      learning_rate)

        if print_cost and i % 5000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
        if print_cost and i % 500 == 0:
            costs.append(cost)


    plt.plot(costs)
    plt.ylabel('Cost')
    plt.xlabel('Iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters, costs
```

Inicialización de He

```
def init_params_he(layers_dims):

    np.random.seed()
```

```
    parameters = {}
    L = len(layers_dims) - 1

    for l in range(L):

        parameters["W" + str(l+1)] = np.random.randn(
                                 layers_dims[l+1], layers_dims[l]) *
                                        np.sqrt(2 / layers_dims[l])
        parameters["b" + str(l+1)] = np.zeros((layers_dims[l+1], 1))

    return parameters
```

Propagación hacia delante

```
def forward_prop(X, parameters):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]
    W4 = parameters["W4"]
    b4 = parameters["b4"]

    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)
    Z3 = np.dot(W3, A2) + b3
    A3 = relu(Z3)
    Z4 = np.dot(W4, A3) + b4
    A4 = softmax(Z4)

    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3, Z4, A4,
                                                              W4, b4)

    return A4, cache
```

Cálculo de la función de coste con *L2 Regularization*

```
def compute_cost_with_L2reg(A4, Y, parameters, lambd):


    m = Y.shape[1]

    W1 = parameters["W1"]
    W2 = parameters["W2"]
    W3 = parameters["W3"]
    W4 = parameters["W4"]

    cross_entr_cost = compute_cost(A4, Y)

    L2reg_cost = (1/m) * (lambd/2) * (np.sum(np.square(W1)) +
```

```
                    np.sum(np.square(W2)) + np.sum(np.square(W3)) +
                                             np.sum(np.square(W4)))

    cost = cross_entr_cost + L2reg_cost

    return cost
```

Propagación hacia atrás con *L2 Regularization*

```
def back_prop_with_L2reg(X, Y, cache, lambd):


    Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3, Z4, A4, W4, b4 =
                                                                  cache

    m = X.shape[1]

    dZ4 = A4 - Y
    dW4 = 1/m * np.dot(dZ4, A3.T) + ((lambd/m) * W4)
    db4 = 1/m * np.sum(dZ4, axis=1, keepdims=True)

    dA3 = np.dot(W4.T, dZ4)
    dZ3 = np.multiply(dA3, np.int64(A3 > 0))
    dW3 = 1/m * np.dot(dZ3, A2.T) + ((lambd/m) * W3)
    db3 = 1/m * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1/m * np.dot(dZ2, A1.T) + ((lambd/m) * W2)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1/m * np.dot(dZ1, X.T) + ((lambd/m) * W1)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1" : dW1, "db1" : db1, "dZ1" : dZ1, "dA1" : dA1,
             "dW2" : dW2, "db2" : db2, "dZ2" : dZ2, "dA2" : dA2,
             "dW3" : dW3, "db3" : db3, "dZ3" : dZ3, "dA3" : dA3,
             "dW4" : dW4, "db4" : db4, "dZ4" : dZ4}

    return grads
```

Actualización de pesos

```
def update_parameters(parameters, gradients, learning_rate):

    L = len(parameters) // 2

    for l in range(L):

        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                                   learning_rate * gradients["dW" + str(l+1)]
```

```
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
                            learning_rate * gradients["db" + str(l+1)]

    return parameter
```

Anexo D - Modelo 5

Modelo 5

```
def model_5(X, Y, learning_rate = 0.1, num_iter = 20010,
                                   keep_prob = 0.5, print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layers_dims = [X.shape[0], 25, 25, 16, 6]

    parameters = init_params_he(layers_dims)

    for i in range(num_iter):

        A4, cache = forward_prop_with_dropout(X, parameters,
                                                        keep_prob)

        cost = compute_cost(A4, Y)

        gradients = back_prop_with_dropout(X, Y, cache, keep_prob)

        parameters = update_parameters(parameters, gradients,
                                                    learning_rate)

        if print_cost and i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
        if print_cost and i % 1000 == 0:
            costs.append(cost)


    plt.plot(costs)
    plt.ylabel('Cost')
    plt.xlabel('Iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters, costs
```

Inicialización de He

```
def init_params_he(layers_dims):

    np.random.seed()
    parameters = {}
    L = len(layers_dims) - 1

    for l in range(L):

        parameters["W" + str(l+1)] = np.random.randn(
                                  layers_dims[l+1], layers_dims[l]) *
                                          np.sqrt(2 / layers_dims[l])
        parameters["b" + str(l+1)] = np.zeros((layers_dims[l+1], 1))

    return parameters
```

Propagación hacia delante con *dropout*

```
def forward_prop_with_dropout(X, parameters, keep_prob):

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]
    W4 = parameters["W4"]
    b4 = parameters["b4"]

    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)

    D1 = np.random.rand(A1.shape[0], A1.shape[1])
    D1 = (D1 < keep_prob).astype(int)
    A1 = np.multiply(A1, D1)
    A1 = A1 / keep_prob

    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)

    D2 = np.random.rand(A2.shape[0], A2.shape[1])
    D2 = (D2 < keep_prob).astype(int)
    A2 = np.multiply(A2, D2)
    A2 = A2 / keep_prob

    Z3 = np.dot(W3, A2) + b3
    A3 = relu(Z3)

    D3 = np.random.rand(A3.shape[0], A3.shape[1])
    D3 = (D3 < keep_prob).astype(int)
    A3 = np.multiply(A3, D3)
    A3 = A3 / keep_prob
```

```
    Z4 = np.dot(W4, A3) + b4
    A4 = softmax(Z4)

    cache = (Z1, A1, D1, W1, b1, Z2, A2, D2, W2, b2, Z3, A3, D3, W3,
                                                   b3, Z4, A4, W4, b4)

    return A4, cache
```

Función de coste de Softmax classifier

```
def compute_cost(A4, Y, eps=0.00000000001):

    m = Y.shape[1]

    loss = - np.sum(np.multiply(Y, np.log(A4+eps)), axis=0)

    cost = 1/m * np.sum(loss)

    return cost
```

Propagación hacia atrás con *dropout*

```
def back_prop_with_dropout(X, Y, cache, keep_prob):

    Z1, A1, D1, W1, b1, Z2, A2, D2, W2, b2, Z3, A3, D3, W3, b3, Z4,
                                                  A4, W4, b4 = cache
    m = X.shape[1]

    dZ4 = A4 - Y
    dW4 = 1/m * np.dot(dZ4, A3.T)
    db4 = 1/m * np.sum(dZ4, axis=1, keepdims=True)

    dA3 = np.dot(W4.T, dZ4)
    dA3 = np.multiply(dA3, D3)
    dA3 = dA3 / keep_prob
    dZ3 = np.multiply(dA3, np.int64(A3 > 0))
    dW3 = 1/m * np.dot(dZ3, A2.T)
    db3 = 1/m * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dA2 = np.multiply(dA2, D2)
    dA2 = dA2 / keep_prob
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1/m * np.dot(dZ2, A1.T)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dA1 = np.multiply(dA1, D1)
    dA1 = dA1 / keep_prob
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1/m * np.dot(dZ1, X.T)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1" : dW1, "db1" : db1, "dZ1" : dZ1, "dA1" : dA1,
```

```
            "dW2" : dW2, "db2" : db2, "dZ2" : dZ2, "dA2" : dA2,
            "dW3" : dW3, "db3" : db3, "dZ3" : dZ3, "dA3" : dA3,
            "dW4" : dW4, "db4" : db4, "dZ4" : dZ4}

    return grads
```

Actualización de pesos

```
def update_parameters(parameters, gradients, learning_rate):

    L = len(parameters) // 2

    for l in range(L):

        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                             learning_rate * gradients["dW" + str(l+1)]

        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
                             learning_rate * gradients["db" + str(l+1)]

    return parameters
```

Anexo E - Modelo 6

Modelo 6

```
def model_6(X, Y, learning_rate = 0.1, num_iter = 100010, keep_prob =
                                   0.5, beta = 0.9, print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layers_dims = [X.shape[0], 25, 25, 16, 6]

    parameters = init_params_he(layers_dims)

    v = init_momentum_var(parameters)

    for i in range(num_iter):

        A4, cache = forward_prop_with_dropout(X, parameters,
                                                              keep_prob)

        cost = compute_cost(A4, Y)

        gradients = back_prop_with_dropout(X, Y, cache, keep_prob)

        parameters, v = upd_params_with_momentum(parameters,
                                   gradients, v, beta, learning_rate)

        if print_cost and i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
        if print_cost and i % 1000 == 0:
            costs.append(cost)


    plt.plot(costs)
    plt.ylabel('Cost')
    plt.xlabel('Iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters, costs
```

Inicialización de He

```
def init_params_he(layers_dims):

    np.random.seed()
    parameters = {}
    L = len(layers_dims) - 1

    for l in range(L):

        parameters["W" + str(l+1)] = np.random.randn(
                                  layers_dims[l+1], layers_dims[l]) *
                                          np.sqrt(2 / layers_dims[l])
        parameters["b" + str(l+1)] = np.zeros((layers_dims[l+1], 1))

    return parameters
```

Inicialización variables *GD with momentum*

```
def init_momentum_var(parameters):

    L = len(parameters) // 2
    v = {}
    for l in range(L):

        v["dW" + str(l+1)] = np.zeros((parameters["W" +
                                                     str(l+1)].shape))

        v["db" + str(l+1)] = np.zeros((parameters["b" +
                                                     str(l+1)].shape))

    return v
```

Propagación hacia delante con *dropout*

```
def forward_prop_with_dropout(X, parameters, keep_prob):

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]
    W4 = parameters["W4"]
    b4 = parameters["b4"]

    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)

    D1 = np.random.rand(A1.shape[0], A1.shape[1])
    D1 = (D1 < keep_prob).astype(int)
    A1 = np.multiply(A1, D1)
```

```
    A1 = A1 / keep_prob

    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)

    D2 = np.random.rand(A2.shape[0], A2.shape[1])
    D2 = (D2 < keep_prob).astype(int)
    A2 = np.multiply(A2, D2)
    A2 = A2 / keep_prob

    Z3 = np.dot(W3, A2) + b3
    A3 = relu(Z3)

    D3 = np.random.rand(A3.shape[0], A3.shape[1])
    D3 = (D3 < keep_prob).astype(int)
    A3 = np.multiply(A3, D3)
    A3 = A3 / keep_prob

    Z4 = np.dot(W4, A3) + b4
    A4 = softmax(Z4)

    cache = (Z1, A1, D1, W1, b1, Z2, A2, D2, W2, b2, Z3, A3, D3, W3,
                                                  b3, Z4, A4, W4, b4)

    return A4, cache
```

Función de coste de Softmax classifier

```
def compute_cost(A4, Y, eps=0.00000000001):

    m = Y.shape[1]

    loss = - np.sum(np.multiply(Y, np.log(A4+eps)), axis=0)

    cost = 1/m * np.sum(loss)

    return cost
```

Propagación hacia atrás con *dropout*

```
def back_prop_with_dropout(X, Y, cache, keep_prob):

    Z1, A1, D1, W1, b1, Z2, A2, D2, W2, b2, Z3, A3, D3, W3, b3, Z4,
                                                   A4, W4, b4 = cache
    m = X.shape[1]

    dZ4 = A4 - Y
    dW4 = 1/m * np.dot(dZ4, A3.T)
    db4 = 1/m * np.sum(dZ4, axis=1, keepdims=True)

    dA3 = np.dot(W4.T, dZ4)
    dA3 = np.multiply(dA3, D3)
    dA3 = dA3 / keep_prob
    dZ3 = np.multiply(dA3, np.int64(A3 > 0))
```

```
    dW3 = 1/m * np.dot(dZ3, A2.T)
    db3 = 1/m * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dA2 = np.multiply(dA2, D2)
    dA2 = dA2 / keep_prob
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1/m * np.dot(dZ2, A1.T)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dA1 = np.multiply(dA1, D1)
    dA1 = dA1 / keep_prob
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1/m * np.dot(dZ1, X.T)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1" : dW1, "db1" : db1, "dZ1" : dZ1, "dA1" : dA1,
             "dW2" : dW2, "db2" : db2, "dZ2" : dZ2, "dA2" : dA2,
             "dW3" : dW3, "db3" : db3, "dZ3" : dZ3, "dA3" : dA3,
             "dW4" : dW4, "db4" : db4, "dZ4" : dZ4}

    return grads
```

Actualización de pesos con *GD with momentum*

```
def upd_params_with_momentum(parameters, gradients, v, beta,
                                                       learning_rate):

    L = len(parameters) // 2

    for l in range(L):

        v["dW" + str(l+1)] = beta * v["dW" + str(l+1)] + (1-beta) *
                                            gradients["dW" + str(l+1)]

        v["db" + str(l+1)] = beta * v["db" + str(l+1)] + (1-beta) *
                                            gradients["db" + str(l+1)]

        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                                    learning_rate * v["dW" + str(l+1)]

        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
                                    learning_rate * v["db" + str(l+1)]

    return parameters, v
```

Anexo F - Modelo 7

Modelo 7

```
def model_7(X, Y, learning_rate = 0.1, num_iter = 50010, beta1 = 0.9,
                                     beta2 = 0.999, print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layers_dims = [X.shape[0], 25, 25, 16, 6]

    parameters = init_params_he(layers_dims)

    v, s = init_adam(parameters)

    t = 0

    for i in range(num_iter):

        A4, cache = forward_prop_with_dropout(X, parameters,
                                                               keep_prob)

        cost = compute_cost(A4, Y)

        gradients = back_prop_with_dropout(X, Y, cache, keep_prob)

        t = t + 1

        parameters, v, s = upd_params_with_adam(parameters,
                                   gradients, v, s, t, learning_rate,
                                        beta1, beta2, epsilon = 1e-8)

        if print_cost and i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
        if print_cost and i % 1000 == 0:
            costs.append(cost)


    plt.plot(costs)
    plt.ylabel('Cost')
    plt.xlabel('Iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters, costs
```

Inicialización de He

```
def init_params_he(layers_dims):

    np.random.seed()
    parameters = {}
    L = len(layers_dims) - 1

    for l in range(L):

        parameters["W" + str(l+1)] = np.random.randn(
                                  layers_dims[l+1], layers_dims[l]) *
                                          np.sqrt(2 / layers_dims[l])
        parameters["b" + str(l+1)] = np.zeros((layers_dims[l+1], 1))

    return parameters
```

Inicializar variables Adam

```
def init_adam(parameters):

    L = len(parameters) // 2

    v = {}
    s = {}

    for l in range(L):

        v["dW" + str(l+1)] = np.zeros((parameters["W" +
                                                      str(l+1)].shape))

        v["db" + str(l+1)] = np.zeros((parameters["b" +
                                                      str(l+1)].shape))

        s["dW" + str(l+1)] = np.zeros((parameters["W" +
                                                      str(l+1)].shape))

        s["db" + str(l+1)] = np.zeros((parameters["b" +
                                                      str(l+1)].shape))

    return v, s
```

Propagación hacia delante con *dropout*

```
def forward_prop_with_dropout(X, parameters, keep_prob):

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]
```

```
    W4 = parameters["W4"]
    b4 = parameters["b4"]

    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)

    D1 = np.random.rand(A1.shape[0], A1.shape[1])
    D1 = (D1 < keep_prob).astype(int)
    A1 = np.multiply(A1, D1)
    A1 = A1 / keep_prob

    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)

    D2 = np.random.rand(A2.shape[0], A2.shape[1])
    D2 = (D2 < keep_prob).astype(int)
    A2 = np.multiply(A2, D2)
    A2 = A2 / keep_prob

    Z3 = np.dot(W3, A2) + b3
    A3 = relu(Z3)

    D3 = np.random.rand(A3.shape[0], A3.shape[1])
    D3 = (D3 < keep_prob).astype(int)
    A3 = np.multiply(A3, D3)
    A3 = A3 / keep_prob

    Z4 = np.dot(W4, A3) + b4
    A4 = softmax(Z4)

    cache = (Z1, A1, D1, W1, b1, Z2, A2, D2, W2, b2, Z3, A3, D3, W3,
                                                        b3, Z4, A4, W4, b4)

    return A4, cache
```

Función de coste de Softmax classifier

```
def compute_cost(A4, Y, eps=0.00000000001):

    m = Y.shape[1]

    loss = - np.sum(np.multiply(Y, np.log(A4+eps)), axis=0)

    cost = 1/m * np.sum(loss)

    return cost
```

Propagación hacia atrás con *dropout*

```
def back_prop_with_dropout(X, Y, cache, keep_prob):

    Z1, A1, D1, W1, b1, Z2, A2, D2, W2, b2, Z3, A3, D3, W3, b3, Z4,
                                                         A4, W4, b4 = cache
    m = X.shape[1]
```

```
    dZ4 = A4 - Y
    dW4 = 1/m * np.dot(dZ4, A3.T)
    db4 = 1/m * np.sum(dZ4, axis=1, keepdims=True)

    dA3 = np.dot(W4.T, dZ4)
    dA3 = np.multiply(dA3, D3)
    dA3 = dA3 / keep_prob
    dZ3 = np.multiply(dA3, np.int64(A3 > 0))
    dW3 = 1/m * np.dot(dZ3, A2.T)
    db3 = 1/m * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dA2 = np.multiply(dA2, D2)
    dA2 = dA2 / keep_prob
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1/m * np.dot(dZ2, A1.T)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dA1 = np.multiply(dA1, D1)
    dA1 = dA1 / keep_prob
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1/m * np.dot(dZ1, X.T)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1" : dW1, "db1" : db1, "dZ1" : dZ1, "dA1" : dA1,
             "dW2" : dW2, "db2" : db2, "dZ2" : dZ2, "dA2" : dA2,
             "dW3" : dW3, "db3" : db3, "dZ3" : dZ3, "dA3" : dA3,
             "dW4" : dW4, "db4" : db4, "dZ4" : dZ4}

    return grads
```

Actualización de pesos con Adam

```
def upd_params_with_adam(parameters, gradients, v, s, t,
                         learning_rate, beta1, beta2, epsilon = 1e-8):

    L = len(parameters) // 2

    v_corr = {}
    s_corr = {}

    for l in range(L):

        v["dW" + str(l+1)] = beta1 * v["dW" + str(l+1)] + (1-beta1) *
                                              gradients["dW" + str(l+1)]

        v["db" + str(l+1)] = beta1 * v["db" + str(l+1)] + (1-beta1) *
                                              gradients["db" + str(l+1)]


        s["dW" + str(l+1)] = beta2 * s["dW" + str(l+1)] + (1-beta2) *
                                 np.multiply(gradients["dW" + str(l+1)],
                                             gradients["dW" + str(l+1)])
```

```
        s["db" + str(l+1)] = beta2 * s["db" + str(l+1)] + (1-beta2) *
                              np.multiply(gradients["db" + str(l+1)],
                                          gradients["db" + str(l+1)])


        v_corr["dW" + str(l+1)] = v["dW" + str(l+1)] / (1 -
                                                        (beta1**t))

        v_corr["db" + str(l+1)] = v["db" + str(l+1)] / (1 -
                                                        (beta1**t))


        s_corr["dW" + str(l+1)] = s["dW" + str(l+1)] / (1 -
                                                        (beta2**t))

        s_corr["db" + str(l+1)] = s["db" + str(l+1)] / (1 -
                                                        (beta2**t))


        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                           learning_rate * (v_corr["dW" + str(l+1)] /
                         (np.sqrt(s_corr["dW" + str(l+1)] + epsilon)))

        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
                           learning_rate * (v_corr["db" + str(l+1)] /
                         (np.sqrt(s_corr["db" + str(l+1)] + epsilon)))

    return parameters, v, s
```

Anexo G - Modelo 8

Modelo 8

```
def model_8(X, Y, X_test, Y_test, learning_rate = 0.1, num_iter =
                            20010, keep_prob = 0.5, print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layers_dims = [X.shape[0], 25, 25, 16, 16, 6]

    parameters = init_params_he(layers_dims)

    for i in range(num_iter):

        A5, cache = forward_prop_with_dropout_5l(X, parameters,
                                                         keep_prob)

        cost = compute_cost(A5, Y)

        gradients = back_prop_with_dropout_5l(X, Y, cache, keep_prob)

        parameters = update_parameters(parameters, gradients,
                                                     learning_rate)

        if print_cost and i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))

            print("---------------------------------------------")
            print("Con el Train set: ")
            predict(X, Y, parameters)
            print("Con el Test set: ")
            predict(X_test, Y_test, parameters)
            print("---------------------------------------------")
        if print_cost and i % 1000 == 0:
            costs.append(cost)


    plt.plot(costs)
    plt.ylabel('Cost')
    plt.xlabel('Iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters, costs
```

Inicialización de He

```
def init_params_he(layers_dims):

    np.random.seed()
    parameters = {}
    L = len(layers_dims) - 1

    for l in range(L):

        parameters["W" + str(l+1)] = np.random.randn(
                                 layers_dims[l+1], layers_dims[l]) *
                                         np.sqrt(2 / layers_dims[l])
        parameters["b" + str(l+1)] = np.zeros((layers_dims[l+1], 1))

    return parameters
```

Propagación hacia delante con *dropout*

```
def forward_prop_with_dropout_5l(X, parameters, keep_prob):

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]
    W4 = parameters["W4"]
    b4 = parameters["b4"]
    W5 = parameters["W5"]
    b5 = parameters["b5"]

    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)

    D1 = np.random.rand(A1.shape[0], A1.shape[1])
    D1 = (D1 < keep_prob).astype(int)
    A1 = np.multiply(A1, D1)
    A1 = A1 / keep_prob

    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)

    D2 = np.random.rand(A2.shape[0], A2.shape[1])
    D2 = (D2 < keep_prob).astype(int)
    A2 = np.multiply(A2, D2)
    A2 = A2 / keep_prob

    Z3 = np.dot(W3, A2) + b3
    A3 = relu(Z3)

    D3 = np.random.rand(A3.shape[0], A3.shape[1])
    D3 = (D3 < keep_prob).astype(int)
    A3 = np.multiply(A3, D3)
```

```
    A3 = A3 / keep_prob

    Z4 = np.dot(W4, A3) + b4
    A4 = relu(Z4)

    D4 = np.random.rand(A4.shape[0], A4.shape[1])
    D4 = (D4 < keep_prob).astype(int)
    A4 = np.multiply(A4, D4)
    A4 = A4 / keep_prob

    Z5 = np.dot(W5, A4) + b5
    A5 = softmax(Z5)

    cache = (Z1, A1, D1, W1, b1, Z2, A2, D2, W2, b2, Z3, A3, D3, W3,
                              b3, Z4, A4, D4, W4, b4, Z5, A5, W5, b5)

    return A5, cache
```

Función de coste de Softmax classifier

```
def compute_cost(A4, Y, eps=0.00000000001):

    m = Y.shape[1]

    loss = - np.sum(np.multiply(Y, np.log(A4+eps)), axis=0)

    cost = 1/m * np.sum(loss)

    return cost
```

Propagación hacia atrás con *dropout*

```
def back_prop_with_dropout_5l(X, Y, cache, keep_prob):


    Z1, A1, D1, W1, b1, Z2, A2, D2, W2, b2, Z3, A3, D3, W3, b3, Z4,
                               A4, D4, W4, b4, Z5, A5, W5, b5 = cache

    m = X.shape[1]

    dZ5 = A5 - Y
    dW5 = 1/m * np.dot(dZ5, A4.T)
    db5 = 1/m * np.sum(dZ5, axis=1, keepdims=True)

    dA4 = np.dot(W5.T, dZ5)
    dA4 = np.multiply(dA4, D4)
    dA4 = dA4 / keep_prob
    dZ4 = np.multiply(dA4, np.int64(A4 > 0))
    dW4 = 1/m * np.dot(dZ4, A3.T)
    db4 = 1/m * np.sum(dZ4, axis=1, keepdims=True)

    dA3 = np.dot(W4.T, dZ4)
    dA3 = np.multiply(dA3, D3)
    dA3 = dA3 / keep_prob
```

```
    dZ3 = np.multiply(dA3, np.int64(A3 > 0))
    dW3 = 1/m * np.dot(dZ3, A2.T)
    db3 = 1/m * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dA2 = np.multiply(dA2, D2)
    dA2 = dA2 / keep_prob
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1/m * np.dot(dZ2, A1.T)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dA1 = np.multiply(dA1, D1)
    dA1 = dA1 / keep_prob
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1/m * np.dot(dZ1, X.T)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1" : dW1, "db1" : db1, "dZ1" : dZ1, "dA1" : dA1,
             "dW2" : dW2, "db2" : db2, "dZ2" : dZ2, "dA2" : dA2,
             "dW3" : dW3, "db3" : db3, "dZ3" : dZ3, "dA3" : dA3,
             "dW4" : dW4, "db4" : db4, "dZ4" : dZ4, "dA4" : dA4,
             "dW5" : dW5, "db5" : db5, "dZ5" : dZ5}

    return grads
```

Actualización de pesos

```
def update_parameters(parameters, gradients, learning_rate):

    L = len(parameters) // 2

    for l in range(L):

        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                               learning_rate * gradients["dW" + str(l+1)]

        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
                               learning_rate * gradients["db" + str(l+1)]

    return parameters
```

Anexo H - Modelo 9

Modelo 9

```
def model_10(X, Y, X_test, Y_test, layers_dims, learning_rate = 0.01,
                               num_iter = 20010, keep_prob = 0.87,
                        beta1=0.8, beta2=0.999, print_cost=True):

    m = X.shape[1]

    grads = {}
    costs = []

    layers_dims = layers_dims

    parameters = init_params_he(layers_dims)

    v, s = init_adam(parameters)

    t = 0

    for i in range(num_iter):

        A6, cache = forward_prop_with_dropout_6l(X, parameters,
                                                    keep_prob)

        cost = compute_cost(A6, Y)

        gradients = back_prop_with_dropout_6l(X, Y, cache, keep_prob)

        t = t + 1

        parameters, v, s = upd_params_with_adam(parameters,
                                gradients, v, s, t, learning_rate,
                                     beta1, beta2, epsilon = 1e-8)

        if print_cost and i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))

            print("------------------------------------------")
            print("Con el Train set: ")
            predict_6l(X, Y, parameters)
            print("Con el Test set: ")
            predict_6l(X_test, Y_test, parameters)
            print("------------------------------------------")
        if print_cost and i % 1000 == 0:
            costs.append(cost)
```

```
    plt.plot(costs)
    plt.ylabel('Cost')
    plt.xlabel('Iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters, costs
```

Inicialización de He

```
def init_params_he(layers_dims):

    np.random.seed()
    parameters = {}
    L = len(layers_dims) - 1

    for l in range(L):

        parameters["W" + str(l+1)] = np.random.randn(
                                 layers_dims[l+1], layers_dims[l]) *
                                         np.sqrt(2 / layers_dims[l])
        parameters["b" + str(l+1)] = np.zeros((layers_dims[l+1], 1))

    return parameters
```

Inicializar variables Adam

```
def init_adam(parameters):

    L = len(parameters) // 2

    v = {}
    s = {}

    for l in range(L):

        v["dW" + str(l+1)] = np.zeros((parameters["W" +
                                                 str(l+1)].shape))

        v["db" + str(l+1)] = np.zeros((parameters["b" +
                                                 str(l+1)].shape))

        s["dW" + str(l+1)] = np.zeros((parameters["W" +
                                                 str(l+1)].shape))

        s["db" + str(l+1)] = np.zeros((parameters["b" +
                                                 str(l+1)].shape))

    return v, s
```

Propagación hacia delante con *dropout*

```
def forward_prop_with_dropout_6l(X, parameters, keep_prob):

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]
    W4 = parameters["W4"]
    b4 = parameters["b4"]
    W5 = parameters["W5"]
    b5 = parameters["b5"]
    W6 = parameters["W6"]
    b6 = parameters["b6"]

    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)

    D1 = np.random.rand(A1.shape[0], A1.shape[1])
    D1 = (D1 < keep_prob).astype(int)
    A1 = np.multiply(A1, D1)
    A1 = A1 / keep_prob

    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)

    D2 = np.random.rand(A2.shape[0], A2.shape[1])
    D2 = (D2 < keep_prob).astype(int)
    A2 = np.multiply(A2, D2)
    A2 = A2 / keep_prob

    Z3 = np.dot(W3, A2) + b3
    A3 = relu(Z3)

    D3 = np.random.rand(A3.shape[0], A3.shape[1])
    D3 = (D3 < keep_prob).astype(int)
    A3 = np.multiply(A3, D3)
    A3 = A3 / keep_prob

    Z4 = np.dot(W4, A3) + b4
    A4 = relu(Z4)

    D4 = np.random.rand(A4.shape[0], A4.shape[1])
    D4 = (D4 < keep_prob).astype(int)
    A4 = np.multiply(A4, D4)
    A4 = A4 / keep_prob

    Z5 = np.dot(W5, A4) + b5
    A5 = relu(Z5)

    D5 = np.random.rand(A5.shape[0], A5.shape[1])
    D5 = (D5 < keep_prob).astype(int)
    A5 = np.multiply(A5, D5)
    A5 = A5 / keep_prob
```

```
    Z6 = np.dot(W6, A5) + b6
    A6 = softmax(Z6)

    cache = (Z1, A1, D1, W1, b1, Z2, A2, D2, W2, b2, Z3, A3, D3, W3,
                                    b3, Z4, A4, D4, W4, b4, Z5, A5,
                                        D5, W5, b5, Z6, A6, W6, b6)

    return A6, cache
```

Función de coste de Softmax classifier

```
def compute_cost(A4, Y, eps=0.0000000001):

    m = Y.shape[1]

    loss = - np.sum(np.multiply(Y, np.log(A4+eps)), axis=0)

    cost = 1/m * np.sum(loss)

    return cost
```

Propagación hacia atrás con *dropout*

```
def back_prop_with_dropout_6l(X, Y, cache, keep_prob):

    Z1, A1, D1, W1, b1, Z2, A2, D2, W2, b2, Z3, A3, D3, W3, b3, Z4,
                                    A4, D4, W4, b4, Z5, A5, D5, W5,
                                         b5, Z6, A6, W6, b6 = cache

    m = X.shape[1]

    dZ6 = A6 - Y
    dW6 = 1/m * np.dot(dZ6, A5.T)
    db6 = 1/m * np.sum(dZ6, axis=1, keepdims=True)

    dA5 = np.dot(W6.T, dZ6)
    dA5 = np.multiply(dA5, D5)
    dA5 = dA5 / keep_prob
    dZ5 = np.multiply(dA5, np.int64(A5 > 0))
    dW5 = 1/m * np.dot(dZ5, A4.T)
    db5 = 1/m * np.sum(dZ5, axis=1, keepdims=True)

    dA4 = np.dot(W5.T, dZ5)
    dA4 = np.multiply(dA4, D4)
    dA4 = dA4 / keep_prob
    dZ4 = np.multiply(dA4, np.int64(A4 > 0))
    dW4 = 1/m * np.dot(dZ4, A3.T)
    db4 = 1/m * np.sum(dZ4, axis=1, keepdims=True)

    dA3 = np.dot(W4.T, dZ4)
```

```
    dA3 = np.multiply(dA3, D3)
    dA3 = dA3 / keep_prob
    dZ3 = np.multiply(dA3, np.int64(A3 > 0))
    dW3 = 1/m * np.dot(dZ3, A2.T)
    db3 = 1/m * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dA2 = np.multiply(dA2, D2)
    dA2 = dA2 / keep_prob
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1/m * np.dot(dZ2, A1.T)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dA1 = np.multiply(dA1, D1)
    dA1 = dA1 / keep_prob
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1/m * np.dot(dZ1, X.T)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1" : dW1, "db1" : db1, "dZ1" : dZ1, "dA1" : dA1,
             "dW2" : dW2, "db2" : db2, "dZ2" : dZ2, "dA2" : dA2,
             "dW3" : dW3, "db3" : db3, "dZ3" : dZ3, "dA3" : dA3,
             "dW4" : dW4, "db4" : db4, "dZ4" : dZ4, "dA4" : dA4,
             "dW5" : dW5, "db5" : db5, "dZ5" : dZ5, "dA5" : dA5,
             "dW6" : dW6, "db6" : db6, "dZ6" : dZ6}

    return grads
```

Actualización de pesos con Adam

```
def upd_params_with_adam(parameters, gradients, v, s, t,
                         learning_rate, beta1, beta2, epsilon = 1e-8):

    L = len(parameters) // 2

    v_corr = {}
    s_corr = {}

    for l in range(L):

        v["dW" + str(l+1)] = beta1 * v["dW" + str(l+1)] + (1-beta1) *
                                             gradients["dW" + str(l+1)]

        v["db" + str(l+1)] = beta1 * v["db" + str(l+1)] + (1-beta1) *
                                             gradients["db" + str(l+1)]


        s["dW" + str(l+1)] = beta2 * s["dW" + str(l+1)] + (1-beta2) *
                            np.multiply(gradients["dW" + str(l+1)],
                                        gradients["dW" + str(l+1)])

        s["db" + str(l+1)] = beta2 * s["db" + str(l+1)] + (1-beta2) *
                            np.multiply(gradients["db" + str(l+1)],
                                        gradients["db" + str(l+1)])
```

```
        v_corr["dW" + str(l+1)] = v["dW" + str(l+1)] / (1 -
                                                    (beta1**t))

        v_corr["db" + str(l+1)] = v["db" + str(l+1)] / (1 -
                                                    (beta1**t))


        s_corr["dW" + str(l+1)] = s["dW" + str(l+1)] / (1 -
                                                    (beta2**t))

        s_corr["db" + str(l+1)] = s["db" + str(l+1)] / (1 -
                                                    (beta2**t))


        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                              learning_rate * (v_corr["dW" + str(l+1)] /
                           (np.sqrt(s_corr["dW" + str(l+1)] + epsilon)))

        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
                              learning_rate * (v_corr["db" + str(l+1)] /
                           (np.sqrt(s_corr["db" + str(l+1)] + epsilon)))

    return parameters, v, s
```

Anexo I - Elementos de medición

Predicción en redes neuronales de 4 capas

```
def predict(X, Y, parametros, learning_rate = 0.1):

    m = X.shape[1]

    A4, cache = forward_prop(X, parametros)

    cost = compute_cost(A4, Y)

    prediction = pred_to_ones(A4)

    result = np.all(prediction==Y, axis=0)

    rights = np.sum(result)

    percent = rights / m

    print("Ha tenido una tasa de acierto de: ", percent)
    print("Y eso con una perdida de: ", cost)

    return percent, cost
```

Transformar predicción en matrices de 1 y 0

```
def pred_to_ones(Y):

    Max = np.max(Y, axis=0)

    one_hot = np.int64(Y==Max)

    return one_hot
```

Anexo J - Tratamiento *missing values*

Importación de biblioteca y creación de gráfico de calor inicial

```
import seaborn as sns

X_graf = X_perm.T
sns.heatmap(X_graf.isnull(), yticklabels=False, cbar=False,
                                                    cmap='viridis')
```

Cálculo de la mediana de *Peso_don* según *Sexo_don*

```
X_graf_h = X_graf[X_graf["Sexo_don"]==0.0]
X_graf_m = X_graf[X_graf["Sexo_don"]==1.0]

percentil_50_h = np.percentile(X_graf_h["Peso_don"].dropna(), 50,
                      axis=0, interpolation='linear', keepdims=False)

percentil_50_m = np.percentile(X_graf_m["Peso_don"].dropna(), 50,
                      axis=0, interpolation='linear', keepdims=False)
```

Imputar los valores en el campo *Peso_don*

```
def imputar_peso_don(cols):
    Peso_don = cols[0]
    Sexo_don = cols[1]

    if pd.isnull(Peso_don):

        if Sexo_don == 0.0:
            return 80.0

        else:
            return 70.0
    else:
        return Peso_don

X_graf["Peso_don"] = X_graf[["Peso_don", "Sexo_don"]].apply(
                                              imputar_peso_don, axis=1)
```

Comprobar imputación mediante gráfico de calor

```
sns.heatmap(X_graf.isnull(), yticklabels=False, cbar=False,
                                                    cmap='viridis')
```

Cálculo de la mediana de *Talla_don* según *Sexo_don*

```
X_graf_h = X_graf[X_graf["Sexo_don"]==0.0]
X_graf_m = X_graf[X_graf["Sexo_don"]==1.0]

percentil_50_h = np.percentile(X_graf_h["Talla_don"].dropna(), 50,
                      axis=0, interpolation='linear', keepdims=False)

percentil_50_m = np.percentile(X_graf_m["Talla_don"].dropna(), 50,
                      axis=0, interpolation='linear', keepdims=False)
```

Imputar los valores en el campo *Talla_don*

```
def imputar_talla_don(cols):
    Talla_don = cols[0]
    Sexo_don = cols[1]

    if pd.isnull(Talla_don):

        if Sexo_don == 0.0:
            return 175.0

        else:
            return 162.0
    else:
        return Talla_don

X_graf["Talla_don"] = X_graf[["Talla_don", "Sexo_don"]].apply(
                                           imputar_talla_don, axis=1)
```

Cálculo de la mediana de *Peso_pre_tx* según *Sexo_rx*

```
X_graf_h = X_graf[X_graf["Sexo_rx"]==0.0]
X_graf_m = X_graf[X_graf["Sexo_rx"]==1.0]

percentil_50_h = np.percentile(X_graf_h["Peso_pre_tx"].dropna(), 50,
                      axis=0, interpolation='linear', keepdims=False)

percentil_50_m = np.percentile(X_graf_m["Peso_pre_tx"].dropna(), 50,
                      axis=0, interpolation='linear', keepdims=False)
```

Imputar los valores en el campo *Peso_pre_tx*

```
def imputar_peso_pre_tx(cols):
    Peso_pre_tx = cols[0]
    Sexo_rx = cols[1]

    if pd.isnull(Peso_pre_tx):

        if Sexo_rx == 0.0:
            return 75.5

        else:
            return 65.45
    else:
        return Peso_pre_tx

X_graf["Peso_pre_tx"] = X_graf[["Peso_pre_tx", "Sexo_rx"]].apply(
                                         imputar_peso_pre_tx, axis=1)
```

Cálculo de la mediana de *Talla_pre_tx* según *Sexo_rx*

```
X_graf_h = X_graf[X_graf["Sexo_rx"]==0.0]
X_graf_m = X_graf[X_graf["Sexo_rx"]==1.0]

percentil_50_h = np.percentile(X_graf_h["Talla_pre_tx"].dropna(), 50,
                      axis=0, interpolation='linear', keepdims=False)

percentil_50_m = np.percentile(X_graf_m["Talla_pre_tx"].dropna(), 50,
                      axis=0, interpolation='linear', keepdims=False)
```

Imputar los valores en el campo *Talla_pre_tx*

```
def imputar_talla_pre_tx(cols):
    Talla_pre_tx = cols[0]
    Sexo_rx = cols[1]

    if pd.isnull(Talla_pre_tx):

        if Sexo_rx == 0.0:
            return 170.0

        else:
            return 159.0
    else:
        return Talla_pre_tx

X_graf["Talla_pre_tx"] = X_graf[["Talla_pre_tx", "Sexo_rx"]].apply(
                                        imputar_talla_pre_tx, axis=1)
```

Cálculo de la mediana de *Creatinina* según *Sexo_don* y *Tipo_don*

```
X_graf_h = X_graf[X_graf["Sexo_rx"]==0.0]
X_graf_h1 = X_graf_h[X_graf_h["Tipo_don"]==0.0]
X_graf_h2 = X_graf_h[X_graf_h["Tipo_don"]==1.0]

X_graf_m = X_graf[X_graf["Sexo_rx"]==1.0]
X_graf_m1 = X_graf_m[X_graf_m["Tipo_don"]==0.0]
X_graf_m2 = X_graf_m[X_graf_m["Tipo_don"]==1.0]

percentil_50_h1 = np.percentile(X_graf_h1["Creatinina_don"].dropna(),
                   50, axis=0, interpolation='linear', keepdims=False)
percentil_50_h2 = np.percentile(X_graf_h2["Creatinina_don"].dropna(),
                   50, axis=0, interpolation='linear', keepdims=False)
percentil_50_m1 = np.percentile(X_graf_m1["Creatinina_don"].dropna(),
                   50, axis=0, interpolation='linear', keepdims=False)
percentil_50_m2 = np.percentile(X_graf_m2["Creatinina_don"].dropna(),
                   50, axis=0, interpolation='linear', keepdims=False)
```

Imputar los valores en el campo *Creatinina*

```
def imputar_creatinina_don(cols):
    Creatinina_don = cols[0]
    Sexo_rx = cols[1]
    Tipo_don = cols[2]

    if pd.isnull(Creatinina_don):

        if Sexo_rx == 0.0:

            if Tipo_don == 0.0:
                return 0.8

            else:
                return 0.72
        else:

            if Tipo_don == 0.0:
                return 0.8

            else:
                return 0.7

    else:
        return Creatinina_don

X_graf["Creatinina_don"] = X_graf[["Creatinina_don", "Sexo_rx",
                                                        "Tipo_don"]].apply(
                                              imputar_creatinina_don, axis=1)
```

Imputar los valores en el campo *Num_Incomp_HLA_DR*

```
def imputar_num_incomp_hla_dr(cols):
    Num_incomp_HLA_DR = cols[0]

    if pd.isnull(Num_incomp_HLA_DR):

        return 1.0

    else:
        return Num_incomp_HLA_DR

X_graf['Num_Incomp_HLA_DR'] = X_graf[["Num_Incomp_HLA_DR"]].apply(
                                    imputar_num_incomp_hla_dr, axis=1)
```

REFERENCIAS

[1] «TensorFlow,» Google, [En línea]: https://www.tensorflow.org.

[2] «PyTorch,» FAIR, [En línea]: https://pytorch.org.

[3] «Keras: the Python deep learning API,» [En línea]: https://keras.io.

[4] «Python,» [En línea]: https://www.python.org.

[5] P. Marius, V. E. Balas, L. Perescu-Popescu y N. E. Mastorakis, Multilayer perceptron and neural networks, 2009.

[6] A. Rana, A. S. Rawat, A. Bijalwan y H. Bahuguna, Application of Multi Layer (Perceptron) Artificial Neural Network in the Diagnosis System: A Systematic Review, 2018.

[7] S. K. Sarvepalli, Deep Learning in Neural Networks: The science behind an Artificial Brain, 2015.

[8] «DeepLearning.AI,» [En línea]: https://www.deeplearning.ai/programs/.

[9] «NumPy. The fundamental package for scientific computing with Python,» 2021. [En línea]: https://numpy.org.

[10] «pandas,» The pandas development team, 2021. [En línea]: https://pandas.pydata.org.

[11] «math — Mathematical functions,» [En línea]: https://docs.python.org/3/library/math.html.

[12] «Matplotlib: Visualization with Python,» The Matplotlib Development team, 2021. [En línea]: https://matplotlib.org.

[13] «time — Time access and conversions,» [En línea]: https://docs.python.org/3/library/time.html.

[14] M. Waskom, «seaborn: statistical data visualization,» 2021. [En línea]: https://seaborn.pydata.org.

[15] «Anaconda. The World's Most Popular Dara Science Platform,» ©2021 Anaconda Inc., [En línea]: https://www.anaconda.com.

[16] «Project Jupyter,» [En línea]: https://jupyter.org.

[17] F. Emmert-Streib, Z. Yang, H. Feng, S. Tripathi y M. Dehmer, An Introductory Review of Deep Learning for Prediction Models With Big Data, 2020.

[18] Y. Wu, L. Liu, J. Bae, K.-H. Chow, A. Iyengar, C. Pu, W. Wei, L. Yu y Q. Zhang, Demystifying Learning Rate Policies for High Accuracy Training of Deep Neural Networks, 2019.

[19] Y. Bengio y X. Glorot, Understanding the difficulty of training deep feedforward neural networks, 2010.

[20] S. K. Kumar, On weight initialization in deep neural networks, 2017.

[21] K. He, X. Zhang, S. Ren y J. Sun, Deep Residual Learning for Image Recognition, 2015.

[22] C. E. Nwankpa, W. Ijomah, A. Gachagan y S. Marshall, Activation Functions: Comparison of Trends in Practice and Research for Deep Learning, 2018.

[23] A. F. Agarap, Deep Learning using Rectified Linear Units (ReLU), 2018.

[24] I. Kouretas y V. Paliouras, Simplified Hardware Implementation of the Softmax Activation Function, 2019.

[25] Z. Zhang y M. R. Sabuncu, Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels, 2018.

[26] R. Das y S. Chaudhuri, On the Separability of Classes with the Cross-Entropy Loss Function, 2019.

[27] R. A. Dunne y N. A. Campbelly, On The Pairing Of The Softmax Activation And Cross{Entropy Penalty Functions And The Derivation Of The Softmax Activation Function, 1997.

[28] B. Gao y L. Pavel, On the Properties of the Softmax Function with Application in Game Theory and Reinforcement Learning, 2017.

[29] P. J. M. Ali y R. H. Faraj, Data Normalization and Standardization: A Technical Report, 2014.

[30] N. Ismoilov y S.-B. Jang, A Comparison of Regularization Techniques in Deep Neural Networks, 2018.

[31] J. Kukačka, V. Golkov y D. Cremers, Regularization for Deep Learning: A Taxonomy, 2017.

[32] T. v. Laarhoven, L2 Regularization versus Batch and Weight Normalization, 2017.

[33] Z. Xie, I. Sato y M. Sugiyama, Understanding and Scheduling Weight Decay, 2020.

[34] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever y R. Salakhutdinov, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, 2014.

[35] S. W. P. L. Stefan Wager, Dropout Training as Adaptive Regularization, 2013.

[36] Y. Bai, E. Yang, B. Han, Y. Yang, J. Li, G. N. Yinian Mao y T. Liu, Understanding and Improving Early Stopping for Learning with Noisy Labels, 2021.

[37] S. Ruder, An overview of gradient descent optimization, 2017.

[38] G. Nakerst, J. Brennan y M. Haque, Gradient descent with momentum --- to accelerate or to super-accelerate?, 2020.

[39] D. P. Kingma y J. Ba, Adam: A Method for Stochastic Optimization, 2015.

[40] P. Izmailov, S. Vikram, M. D. Hoffman y A. G. Wilson, What Are Bayesian Neural Network Posteriors

Really Like?, 2021.

[41] W. McKinney, Python for Data Analysis, 2012.