Área Académica de Ingeniería Mecatrónica

MT 8008 – Inteligencia Artificial

Proyecto final

Optimización evolutiva de hiperparámetros para una aplicación neuronal

Elaborado por:

Diana Cerdas Vargas

2020023718

Jeaustin Calderón Quesada

2020027413

Juan Luis Crespo Mariño, Dr. Ing

Noviembre 2023

Tabla de contenidos

Introducción

En la actualidad es cada vez más común el uso de la inteligencia artificial (IA) en una amplia variedad de áreas que van desde la industria hasta la salud y educación. Este fenómeno está principalmente relacionado con su alta capacidad para solucionar problemas complejos, o bien, para optimizar soluciones en entornos desafiantes. Es aquí donde aparecen las redes neuronales y los algoritmos evolutivos, los cuales son partes fundamentales para crear la inteligencia artificial y la mayoría de sus aplicaciones.

Para comprender qué es una red neuronal, primero se debe conocer la definición de neurona artificial. Una neurona artificial emula la estructura y funcionamiento de una neurona biológica. Cada neurona artificial recibe y procesa señales del entorno o de otras neuronas artificiales y a partir del estímulo recibido emite una señal que puede ir a otras neuronas o puede ser la respuesta del problema [1]. Por su parte, una red neuronal artificial es una estructura compuesta de neuronas artificiales interconectadas de cierta manera. Generalmente, estas redes están formadas por una capa de entrada, capas ocultas y una capa de salida [1].

Existen una amplia variedad de redes neuronales, en este caso, se van a hablar de las redes neuronales de tipo densas. Este tipo de red se caracteriza porque todas las neuronas en cualquier capa de la red están conectadas a todas las neuronas de la capa anterior [2]. Además, cada una de estas conexiones o sinapsis tienen un peso, que junto con la función de activación de una neurona hacen que esta se active o no.

Para que una red neuronal funcione de manera óptima, se debe hacer una selección correcta de los hiperparámetros de la red, como lo son: el número de capas ocultas en la red, el número de neuronas por capa oculta, el tipo de optimizador, la tasa de aprendizaje, entre otras. Estos valores son de vital importancia, ya que con la combinación de todos ellos se determina la complejidad y capacidad que tiene la red para resolver un problema. Generalmente, este tipo de redes neuronales son ampliamente utilizadas para solucionar problemas de regresión y clasificación [3].

Por otro lado, dentro de la inteligencia artificial también se tienen los algoritmos evolutivos, que se definen como una clase de algoritmos de optimización inspirados en la evolución natural que se utilizan para generar soluciones a problemas complejos en diversos campos. Estos algoritmos imitan el proceso de selección natural, reproducción y mutación observado en la evolución biológica para buscar soluciones óptimas o cercanas a óptimas en un espacio de búsqueda [4].

Presentación del problema

En este proyecto se busca diseñar un algoritmo evolutivo que optimice los hiperparámetros de una red neuronal de tipo perceptrón multicapa de tipo densa. En este caso, la red neuronal generada debe ser capaz de clasificar el tipo de movimiento de un robot en cuatro categorías: “Move-Forward” (0), “Slight-Right-Turn” (1), “Sharp-Right-Turn” (2) y “Slight-Left-Turn” (3), según los valores de tensión de 24 sensores ultrasónicos.

Los hiperparámetros a optimizar de la red neuronal se muestran en la tabla 1.

Tabla 1. Hiperparámetros para la red neuronal junto con su posible rango de valores.

Hiperparámetro	**Rango**		
Número de capas ocultas	1 a 6		
Número de neuronas por capa oculta	1 a 14		
Tasa de aprendizaje	0.00001 a 0.5		
Tipo de optimizador	1. ADAM	2. SDG	3. RMSprop
Uso de momento	Binario (sí o no)		

Entonces, el principal problema que se tiene en este proyecto asegurarse de que la red diseñada por el algoritmo evolutivo es la que mejor “mapea” la tendencia descrita por el conjunto de datos. De la mano con esto se tiene la selección de los hiperparámetros del algoritmo evolutivo que permiten optimizar la calidad de los hiperparámetros de la red neuronal.

Análisis del problema

Al abordar el problema lo primero que se hizo fue analizar el tipo de red neuronal que se debía generar. En concreto, se inició con la revisión del set de datos que se iba a utilizar para entrenar la red neuronal. Estos son clasificaciones del tipo de movimiento de un robot de acuerdo con los valores de tensión provenientes de 24 sensores ultrasónicos alrededor del robot. Como se mencionó anteriormente, las categorías son: "Move-Forward" (1), "Slight-Right-Turn" (2), "Sharp-Right-Turn" (3) y "Slight-Left-Turn" (4). Es esencial comprender inicialmente el contexto del problema real por solucionar antes de comenzar con la implementación de la solución.

Una vez clara la problemática, a partir de la información analizada se determinó que la red neuronal debe tener 24 neuronas de entrada (una para cada sensor) y, además, va a tener cuatro neuronas de salida, una para cada tipo de movimiento. También del mismo set de datos se obtuvo que el total de datos que se tienen para el desarrollo de la red neuronal es de 5456, de los cuales se toman un 70% para el entrenamiento, 20% para la validación y un 10% para probar la red ya entrenada. Adicionalmente, en la figura 1 se muestra el número de datos en cada categoría de manera gráfica.

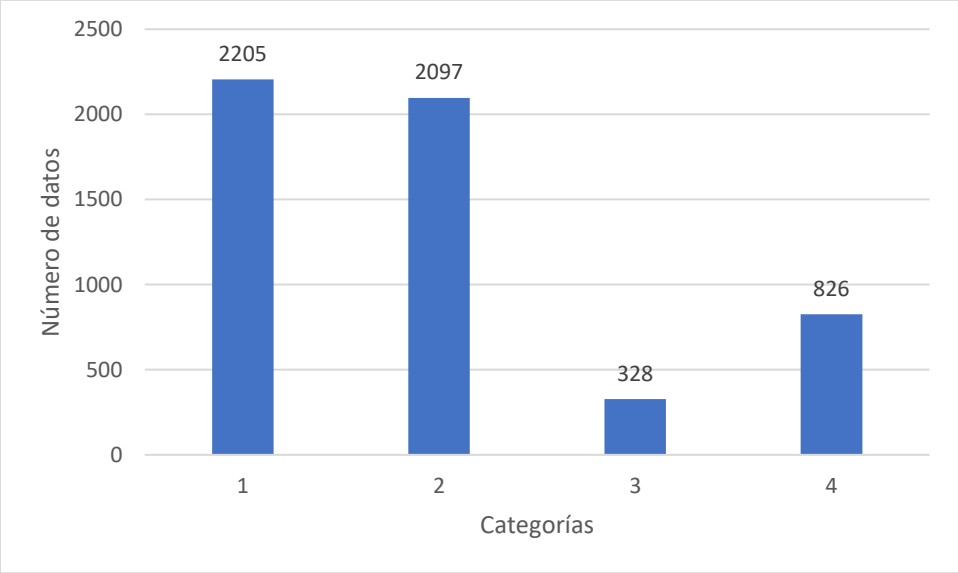

Figura 1. Cantidad de datos en cada una de las categorías en el set de datos.

Cabe destacar que a los datos originales se les tuvo que hacer unos cambios para poder crear la red neuronal. En primer lugar, el archivo se abrió desde Excel y se cargó en formato de tabla. Por lo que solamente se eliminaron unas casillas innecesarias y se le colocaron etiquetas para cada sensor. También, se tuvieron que modificar los valores de las categorías para poder generar la red neuronal, es decir, pasar de nombres a valores numéricos, además para utilizar la función de Categorical Crossentropy de Keras, las etiquetas deben comenzar en 0. Por ende, las categorías finales son: "Move-Forward" (0), "Slight-Right-Turn" (1), "Sharp-Right-Turn" (2) y "Slight-Left-Turn" (3).

Entonces, luego de esto, se procedió a generar el código en Python para la red neuronal, dejando como variables los hiperparámetros que se deben optimizar. Una vez que se tuvo la red, se le agregó la función de "Early Stopping", para hacer el entrenamiento más efectivo y evitar el overfitting ya que esta función se encarga de detener el entrenamiento de una red neuronal cuando el valor de pérdida de entrenamiento no cambia de manera positiva en cierto número de iteraciones. Finalmente se hicieron un par de pruebas para asegurar que la red funcionaba correctamente.

Seguidamente, se optó por dividir el código de la red hecha previamente para que funcionara como una función que tenga como entrada los valores de los hiperparámetros y devuelve el valor de precisión, la pérdida de entrenamiento y la pérdida de validación. Así mismo, para que esta funcione correctamente se subdividió el código de la red en tres funciones adicionales. La primera, se encarga de leer los datos del documento de Excel y devuelve el dataframe. La segunda recibe los valores de los hiperparámetros y genera el modelo de acuerdo con los valores recibidos y la tercera se encarga de realizar la prueba final de la red y generar el reporte.

A continuación, se comenzó con el diseño del algoritmo evolutivo, para ello en primer lugar se definió la codificación que va a tener cada individuo y de acuerdo con el rango de los valores de los hiperparámetros de la red (ver tabla 1) se definió el espacio de alelos. Posteriormente, se definieron las combinaciones de hiperparámetros para el algoritmo evolutivo y además se decidieron los valores que iban a tener los hiperparámetros fijos (posteriormente se va a profundizar y justificar las decisiones al respecto). Una vez esto defiindo, se creó el algoritmo genético en una función y se hicieron unas pequeñas pruebas para verificar que la integración de ambos algoritmos funcionara. Cuando ya se tenía certeza que la programación completa funcionaba, se comenzó con el estudio de hiperparámetros.

Una vez se tenían todas las iteraciones, se probaron las redes neuronales de acuerdo con los genes del mejor individuo de cada generación y con base a estos resultados se escoge cuál es la combinación de hiperparámetros que generó la mejor red neuronal. Para finalizar, para poder validar que la red seleccionada como mejor tiene un error y un comportamiento aceptable, se hicieron una serie de pruebas para observar su comportamiento y las métricas de 5 iteraciones y también se hizo un estudio de sensibilidad de entradas, así como distintas pruebas que permiten argumentar el grado de operación de la solución seleccionada.

En la figura 2 se muestra un diagrama de flujo que resume la metodología planteada para la resolución del problema.

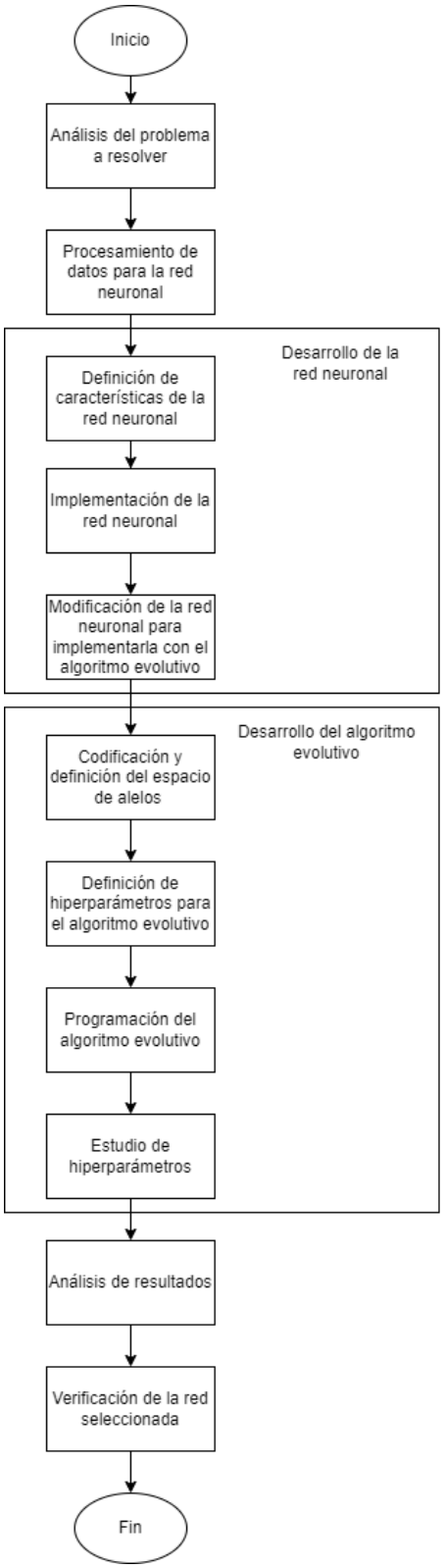

Figura 2. Diagrama de flujo de la metodología seguida para la solución del problema planteado (Elaboración propia).

Descripción de la solución

A continuación, se van a discutir en subsecciones las decisiones de diseño más importantes que se tuvieron que tomar para la implementación de esta solución. Estas soluciones constituyen temas relacionados con el algoritmo evolutivo, sin embargo, como es de esperar, las decisiones también implican una afectación en la red neuronal artificial y esta situación se tomó en cuenta.

Codificación

La codificación se realizó de manera directa para algunos de los genes, en la que cada uno de ellos representa los posibles hiperparámetros, mientras que otros si deben ser interpretados por la función de calidad. En la tabla 2 se muestra un ejemplo de la codificación realizada.

Tabla 2. Ejemplo de codificación para un individuo del algoritmo evolutivo.

Cantidad de capas ocultas	Cantidad de neuronas por capa oculta	Taza de aprendizaje	Optimizador	Momento (sí o no)
1.0	14.0	0.03531385958194733	1.0	1.0

Los primeros tres rubros de la tabla corresponden a los genes que se encuentran ya representados de tal manera que se pueden ingresar directamente como hiperparámetros de la red neuronal artificial, mientras que el optimizador y el momento se encuentran definidos de manera tal que deben ser interpretadas. El optimizador se encuentra relacionado tal y como lo muestra la tabla 1, mientras que el momento un uno representa el uso del momento y el cero representa el no uso del momento. Cabe resaltar que el valor preciso de momento para los optimizadores SDG y RMSprop se definió basándose en la literatura y en los valores recomendados mayormente usados [5].

Mutación

La mutación es esencial en la solución implementada, esto porque vela por mantener una variabilidad en el reservorio genético. Sin embargo, el algoritmo de mutación está altamente condicionado por la codificación de la solución y esto se debe a que es necesario mantener cierta congruencia entre los valores resultantes de la mutación y el problema a resolver, ya que no es deseado que se exploren combinaciones que se encuentra fuera del espacio de alelos. Además, la codificación presentó un reto a la hora de utilizar alguna de las funciones previamente definidas por la librería DEAP, ya que ninguno de los algoritmos permite obtener posibles variaciones en espacios de alelos distintos para cada gen, en especial para este caso en el que se tienen genes representados por valores enteros y otros por valores decimales. Para solucionar esto se implementó una función de mutación personalizada denominada *customMutation* que se puede observar en el anexo 1. Esta función fue basada en el algoritmo de mutación denominado *uniform* [6], en este se itera por cada uno de los genes del individuo y se modifica con probabilidad definida en *indp* dentro de un rango de enteros definido por el usuario. Al personalizar esta función, se mantuvo intacta la lógica de mutación, con la diferencia esencial de que el valor por el cual se sustituiría el gen

corresponde a un valor entero o decimal, según corresponda, definido en un rango específico para cada uno de los genes.

Optimizadores

Una de las decisiones puntuales que se tomaron corresponde a los optimizadores por utilizar, esto porque se solicita utilizar tres optimizadores distintos, sin embargo, no se especifica cuáles son los algoritmos exactos que deben implementarse. Una de las razones por las cuales se delimitaron los posibles algoritmos de optimización, fue la existencia previa en la librería recomendad para la implementación de la red neuronal artificial. Keras provee una variedad limitada de optimizadores y únicamente en dos de estas opciones es posible configurar si se desea utilizar el momento o no. Por esta razón, se escogieron los dos algoritmos que permitían la no selección del momento y se escogió ADAM, puesto que en la literatura se menciona que es el algoritmo más robusto en la actualidad [3]. A raíz de que ADAM no permite desactivar el momento, se presenta la posibilidad de escoger una combinación de hiperparámetros no válida, esto se va a profundizar más adelante cuando se discuta la función de calidad.

Reproducción

Con la posibilidad restante en consideración, se concluyó en la decisión de implementar el método de cruce en un único punto. Dentro de las razones por las que se eligió este método se pueden mencionar su eficiencia computacional, esencial en la implementación de este algoritmo puesto que era necesario realizar un estudio de hiperparámetros que implica una repetida ejecución, así como la variación, este algoritmo a pesar de no procurar la variación del reservorio genético como su prioridad, permite tener una diversidad considerable ya que el punto de cruce es aleatorio cada vez que se cruza una pareja de individuos.

Función de calidad

La función de calidad se planteó con el fin de encontrar la mejor configuración de la red, es decir, la red que mejor resuelve el problema. Por esta razón, se dejan de lado parámetros que tomen en cuenta la complejidad computacional o cantidad de recursos computacionales necesarios para encontrar la solución, y se enfoca únicamente en parámetros que evalúen la capacidad de clasificación de la red neuronal.

Los valores de pérdida son parte de la función de calidad puesto que estos representan la diferencia entre la predicción de la red y los valores reales, se decidió tomar en cuenta la pérdida de entrenamiento y la de validación puesto que existe la posibilidad de que se dé un *overfitting* en el que la pérdida de entrenamiento sea baja, pero la pérdida de validación alta. Por esta razón, se incluyen ambos, se plantean como una suma que desea ser minimizada.

Por otro lado, también es importante tomar en cuenta el nivel de atino durante la etapa de prueba, para eso se incorporó el valor de precisión de cada una de las categorías en la función y se sumaron, como se sabe que en el caso ideal en que todas las predicciones sean correctas la suma debería ser igual a 4, entonces lo que se suma a la función de calidad corresponde a

la diferencia entre el 4 y la suma realizada. Por lo tanto, se tiene entonces una función de calidad modelada por la expresión que se muestra a continuación.

$$y = loss + loss_{val} + (4 - \Sigma P_i)$$

Finalmente, fue indispensable tomar en consideración el caso mencionado anteriormente en el que se tiene una combinación de hiperparámetros no implementable, es decir, una combinación en la que se propone usar ADAM como optimizador, pero a la vez se pide no usar momento. Esto no se puede realizar por la naturaleza del optimizidaor ADAM, por lo que se tomó en cuenta como una excepción en la función de calidad y se le asigna un valor de 10 como calidad. Este valor es considerablemente alto, más de 10 veces mayor a los valores del rango esperado, con esta penalización se procura que estas soluciones sean descartadas eventualmente en el proceso de selección.

Resultados

Estudio de hiperparámetros del algoritmo evolutivo

Para poder definir la mejor combinación de hiperparámetros para el algoritmo evolutivo se realizaron 3 iteraciones, las cuales se muestran a continuación:

Iteración 1

En la tabla 3 se puede observar la combinación de hiperparámetros ejecutada en este caso.

Tabla 3. Parámetros del algoritmo evolutivo para el caso 1.

Método de selección	Población inicial	Número de generaciones	Probabilidad de mutación
Torneo	70	12	0.15

En la figura 3 se puede observar la gráfica que describe la evolución de la calidad y la desviación estándar en términos de la generación para la primera iteración.

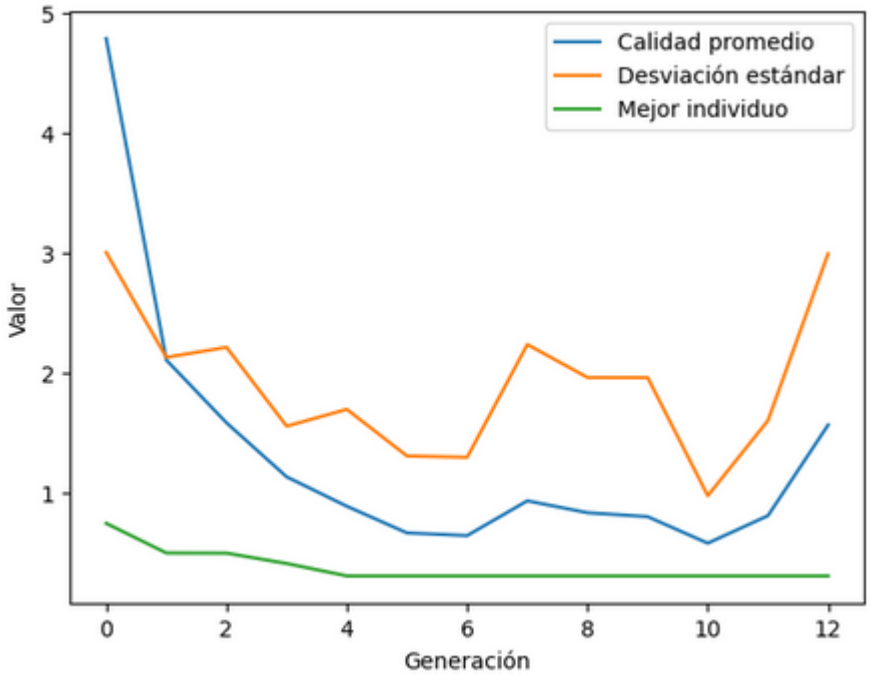

Figura 3. Gráfica de calidad promedio, desviación estándar y calidad del mejor individuo con respecto a generación para la iteración 1.

La mejor calidad de esta combinación corresponde a 0.30700. Mientras que la desviación estándar de la última iteración fue 2.99202. Asimismo, en la tabla 4 se pueden observar los hiperparámetros de la red para el mejor individuo.

Tabla 4. Resultado del mejor individuo.

Número de capas ocultas	Número de neuronas por capa oculta	Tasa de aprendizaje	Optimizador	Momentum
1	9	0.01916	1 (ADAM)	1

La tabla 5 muestra los resultados de la red con los hiperparámetros del mejor individuo de la iteración 1.

Tabla 5. Resultados de pérdida de entrenamiento y validación y precisión de la red con los hiperparámetros del mejor individuo de la iteración 1.

Prueba	Pérdida de entrenamiento	Pérdida de validación	Precisión
1	0.1632	0.2371	0.9174
2	0.1304	0.2123	0.9332
3	0.1581	0.2470	0.9458

Iteración 2

En la tabla 6 se puede observar la combinación de hiperparámetros ejecutada en este caso.

Tabla 6. Parámetros del algoritmo evolutivo para la iteración 2.

Método de selección	Población inicial	Número de generaciones	Probabilidad de mutación
Torneo	70	12	0.05

En la figura 4 se puede observar la gráfica que describe la evolución de la calidad y la desviación estándar en términos de la generación para la segunda iteración.

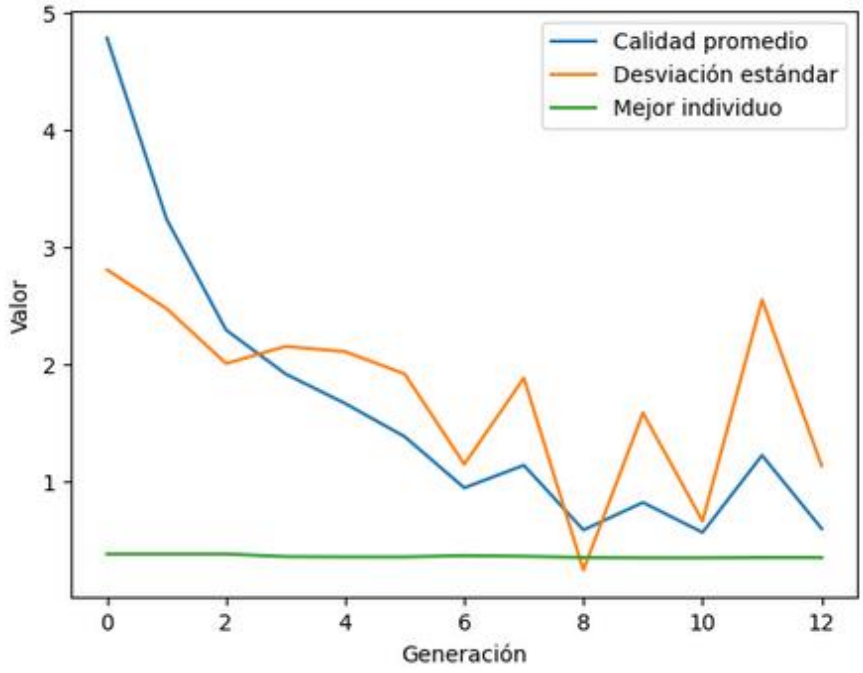

Figura 4. Gráfica de calidad promedio, desviación estándar y calidad del mejor individuo con respecto a generación para la iteración 2.

La mejor calidad de esta combinación corresponde a 0.34988. Mientras que la desviación estándar de la última iteración fue 1.13475. Asimismo, en la tabla 7 se pueden observar los hiperparámetros de la red para el mejor individuo.

Tabla 7. Resultado del mejor individuo.

Número de capas ocultas	Número de neuronas por capa oculta	Tasa de aprendizaje	Optimizador	Momentum
1	13	0.01140	1 (ADAM)	1

La tabla 8 muestra los resultados de la red con los hiperparámetros del mejor individuo de la iteración 2.

Tabla 8. Resultados de pérdida de entrenamiento y validación y precisión de la red con los hiperparámetros del mejor individuo de la iteración 2.

Prueba	Pérdida de entrenamiento	Pérdida de validación	Precisión
1	0.1323	0.2455	0.9287
2	0.1089	0.1813	0.9479
3	0.1016	0.1824	0.9478

Iteración 3

En la tabla 9 se puede observar la combinación de hiperparámetros ejecutada en este caso.

Tabla 9. Parámetros del algoritmo evolutivo para la iteración 3.

Método de selección	Población inicial	Número de generaciones	Probabilidad de mutación
Torneo	85	15	0.05

En la figura 5 se puede observar la gráfica que describe la evolución de la calidad y la desviación estándar en términos de la generación para esta iteración.

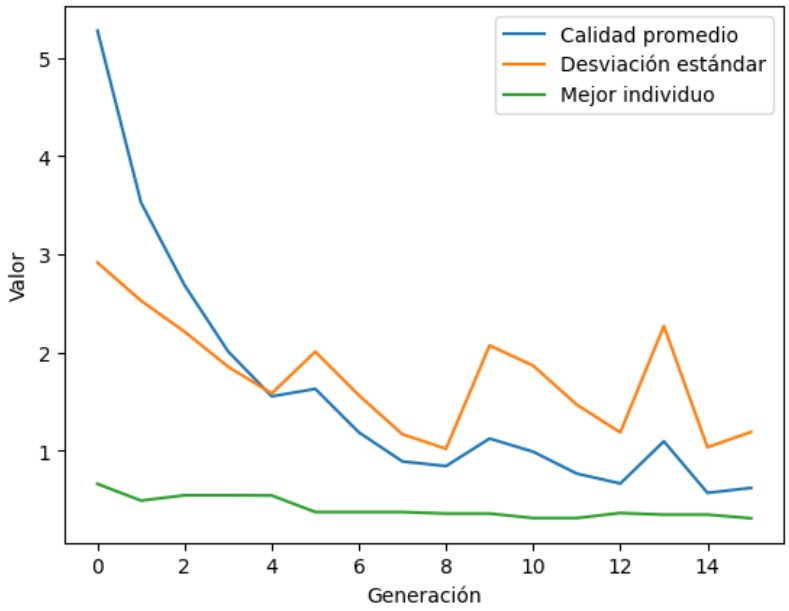

Figura 5. Gráfica de calidad promedio, desviación estándar y calidad del mejor individuo con respecto a generación para la iteración 3.

La mejor calidad de esta combinación corresponde a 0.30836. Mientras que la desviación estándar de la última iteración fue 1.18791. Asimismo, en la tabla 10 se pueden observar los hiperparámetros de la red para el mejor individuo.

Tabla 10. Resultado del mejor individuo.

Número de capas ocultas	Número de neuronas por capa oculta	Tasa de aprendizaje	Optimizador	Momentum
1	14	0.02002	1 (ADAM)	1

La tabla 11 muestra los resultados de la red con los hiperparámetros del mejor individuo de la iteración 3.

Tabla 11. Resultados de pérdida de entrenamiento y validación y precisión de la red con los hiperparámetros del mejor individuo de la iteración 3.

Prueba	Pérdida de entrenamiento	Pérdida de validación	Precisión
1	0.0800	0.1886	0.9780
2	0.0641	0.2074	0.9674
3	0.0899	0.1748	0.9651

Iteración 4

En la tabla 12 se puede observar la combinación de hiperparámetros ejecutada en este caso.

Tabla 12. Parámetros del algoritmo evolutivo para la iteración 4.

Método de selección	Población inicial	Número de generaciones	Probabilidad de mutación
Torneo	100	15	0.01

En la figura 6 se puede observar la gráfica que describe la evolución de la calidad y la desviación estándar en términos de la generación para esta iteración.

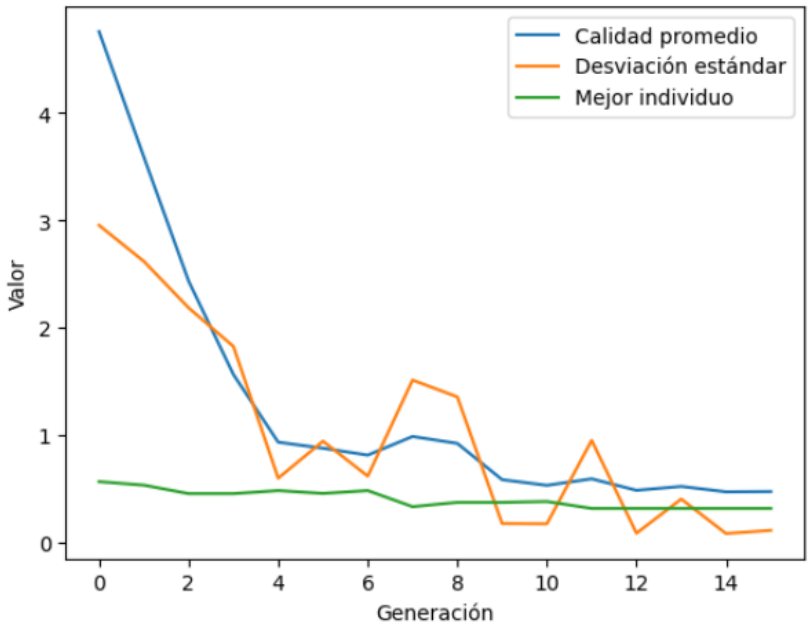

Figura 6. Gráfica de calidad promedio, desviación estándar y calidad del mejor individuo con respecto a generación para la iteración 4.

La mejor calidad de esta combinación corresponde a 0.31455. Mientras que la desviación estándar de la última iteración fue 0.1061. Asimismo, en la tabla 13 se pueden observar los hiperparámetros de la red para el mejor individuo.

Tabla 13. Resultado del mejor individuo.

Número de capas ocultas	Número de neuronas por capa oculta	Tasa de aprendizaje	Optimizador	Momentum
1	13	0.01965	1 (ADAM)	1

La tabla 14 muestra los resultados de la red con los hiperparámetros del mejor individuo de la iteración 4.

Tabla 14. Resultados de pérdida de entrenamiento y validación y precisión de la red con los hiperparámetros del mejor individuo de la iteración 4.

Prueba	Pérdida de entrenamiento	Pérdida de validación	Precisión
1	0.0984	0.2200	0.9707
2	0.0724	0.2439	0.9761
3	0.0682	0.1529	0.9723

Iteración 5

En la tabla 15 se puede observar la combinación de hiperparámetros ejecutada en este caso.

Tabla 15. Parámetros del algoritmo evolutivo para la iteración 5.

Método de selección	Población inicial	Número de generaciones	Probabilidad de mutación
Torneo	150	15	0.025

En la figura 7 se puede observar la gráfica que describe la evolución de la calidad y la desviación estándar en términos de la generación para esta iteración.

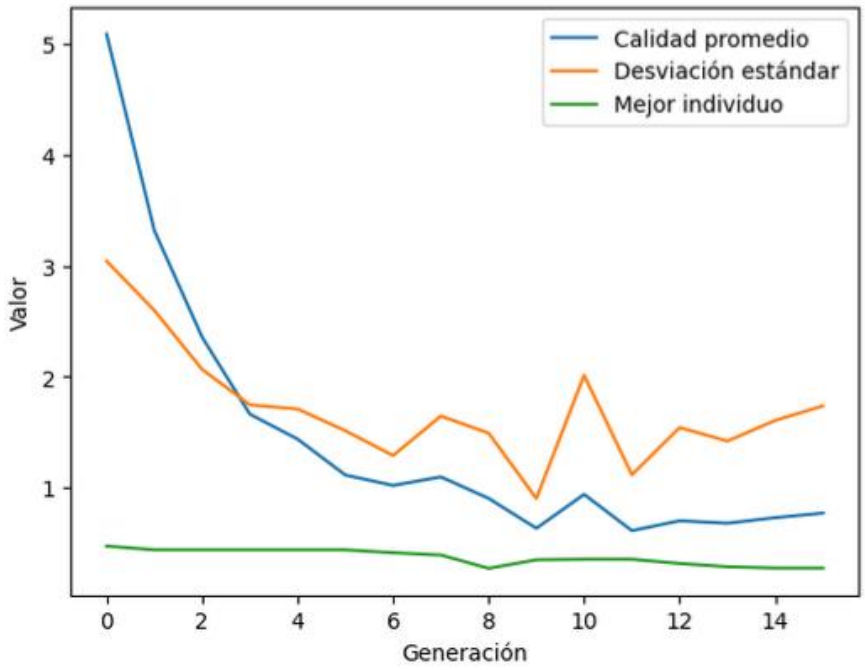

Figura 7. Gráfica de calidad promedio, desviación estándar y calidad del mejor individuo con respecto a generación para la iteración 5.

La mejor calidad de esta combinación corresponde a 0.27390. Mientras que la desviación estándar de la última iteración fue 1.73602. Asimismo, en la tabla 16 se pueden observar los hiperparámetros de la red para el mejor individuo.

Tabla 16. Resultado del mejor individuo.

Número de capas ocultas	Número de neuronas por capa oculta	Tasa de aprendizaje	Optimizador	Momentum
1	14	0.08917	1 (ADAM)	1

La tabla 17 muestra los resultados de la red con los hiperparámetros del mejor individuo de la iteración 5.

Tabla 17. Resultados de pérdida de entrenamiento y validación y precisión de la red con los hiperparámetros del mejor individuo de la iteración 5.

Prueba	Pérdida de entrenamiento	Pérdida de validación	Precisión
1	0.1305	0.2372	0.9516
2	0.1476	0.2101	0.9265
3	0.0859	0.2730	0.9303

Resultados del modelo escogido

Una vez entrenado el modelo con los hiperparámetros ganadores, se procedió a realizar distintas iteraciones de pruebas, esto con el fin de obtener las métricas que permitan concluir la calidad del sistema de clasificación implementado. Para esto se tomó una muestra aleatoria del 10% y con el fin de ser certeros en que se puede dar una generalización de interpretaciones en las variables, es decir, que pueda clasificar varios grupos de datos y no que se corrobore únicamente por coincidencia una pequeña muestra, se decidió iterar tres veces en tres muestras de datos aleatorias distintas. Se recalca que se escogieron las métricas de promedio ponderado con el fin de tener resultados que sean más representativos con la cantidad de observaciones evaluadas por cada muestra aleatoria, esto principalmente porque existe una disparidad notable en la cantidad de muestras de las diferentes categorías en los datos completos.

En la figura 8 se puede observar la matriz de confusión que representa los resultados obtenidos al predecir con la red neuronal escogida, esto para la primera iteración.

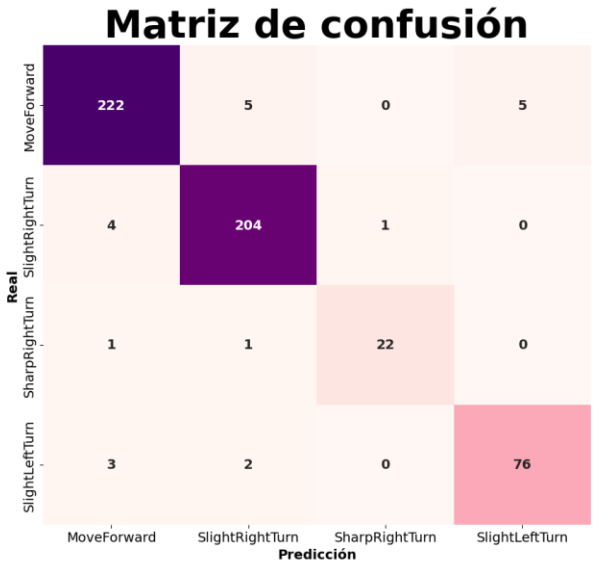

Figura 8. Matriz de confusión de la etapa de prueba de la red neuronal artificial entrenada con los hiperparámetros ganadores (Iteración 1).

Asimismo, en la tabla 18 se muestran distintas métricas que permiten discutir la calidad del modelo entrenado. Estos resultados perteneces a la primera iteración de prueba.

Tabla 18. Métricas obtenidas en la prueba de la iteración 1.

	Precisión	Recall	F1-score	Cantidad de datos total
Promedio ponderado	0.9597	0.9597	0.9597	546

Seguidamente, en la figura 9 se presenta la matriz de confusión que corresponde a la evaluación de los resultados obtenidos con la segunda muestra de datos para prueba aleatoria.

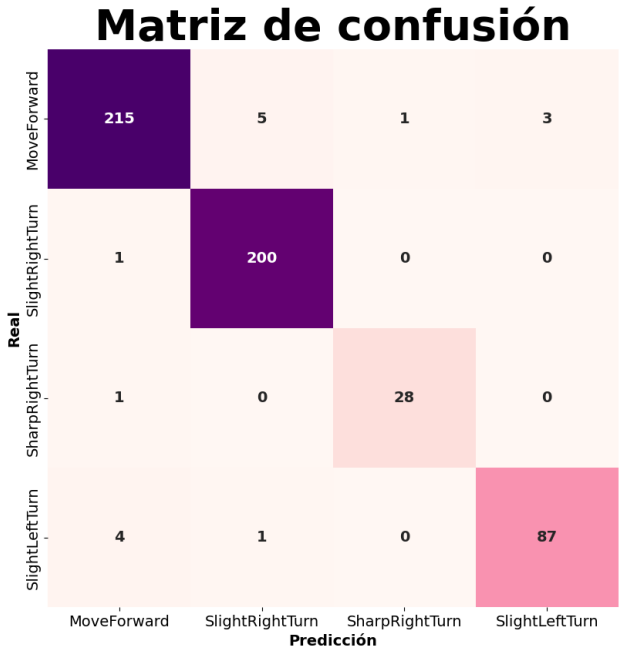

Figura 9. Matriz de confusión de la etapa de prueba de la red neuronal artificial entrenada con los hiperparámetros ganadores (Iteración 2).

A continuación, en la tabla 19, se muestran las métricas obtenidas para la muestra de datos aleatoria de prueba de la segunda iteración.

Tabla 19. Métricas obtenidas en la prueba de la iteración 2.

	Precisión	Recall	F1-score	Cantidad de datos total
Promedio ponderado	0.9707	0.9707	0.9706	546

Asimismo, en la figura 10 se presenta la matriz de confusión que corresponde a la evaluación de los resultados obtenidos con la segunda muestra de datos para prueba aleatoria.

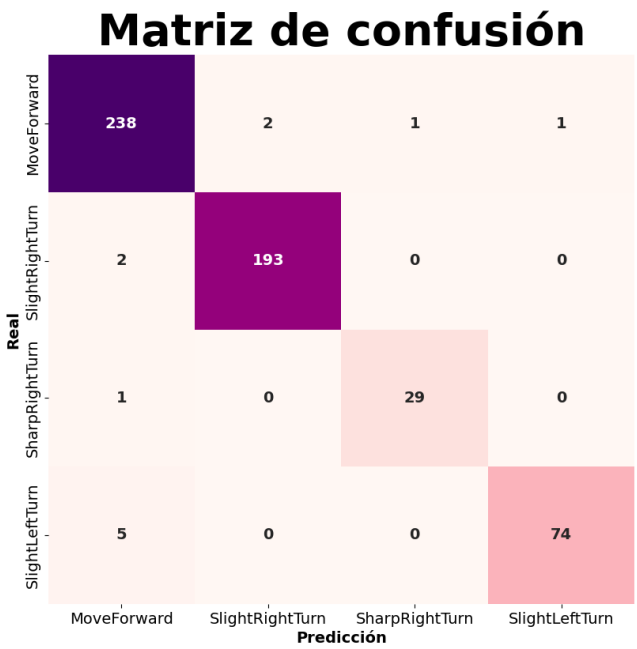

Figura 10. Matriz de confusión de la etapa de prueba de la red neuronal artificial entrenada con los hiperparámetros ganadores (Iteración 3).

Finalmente, en la tabla 20 se presentan las métricas obtenidas para la iteración 3 de muestras de prueba aleatoria.

Tabla 20. Métricas obtenidas en la prueba de la iteración 3.

	Precisión	Recall	F1-score	Cantidad de datos total
Promedio ponderado	0.9782	0.9780	0.9780	546

Estudio de sensibilidad de parámetros de entrada

Para realizar el estudio de sensibilidad de entradas se empleó el método “ceteris paribus” que consiste en mantener los valores de entrada constantes, con excepción a uno de los parámetros, esto se repite con cada una de las variables. El objetivo de este estudio es entender cuáles son las variables de entrada que afectan con mayor o menor intensa las decisiones de la red neuronal ya entrenada. Para realizar este estudio se plantearon variaciones porcentuales de 10, 15, 20 y 25 por ciento, una vez modificados únicamente los valores de un parámetro, se puso a predecir a la red y se cuantifico cuántos errores nuevos se obtenían.

En la figura 11 se puede observar la cantidad de errores nuevos que se obtuvieron al variar un 10% cada una de las entradas del problema planteado. Cabe resaltar que en el caso de que se presenten errores negativos, como son los casos de variaciones de 10%, 15% y 25%, esto representa que en realidad mejoraron las predicciones en comparación con lo analizado con los valores de entrada original.

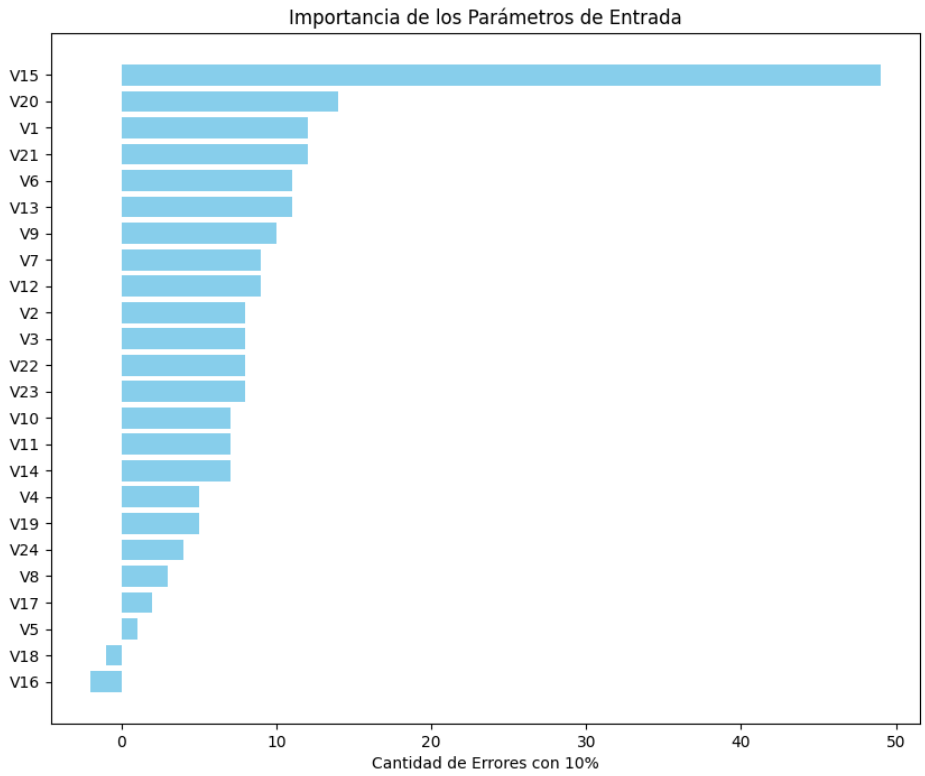

Figura 11. Cantidad de errores nuevos en la predicción al modificar una variable de entrada en 10%.

En la figura 12 se puede observar la cantidad de errores nuevos que se obtuvieron al variar un 15% cada una de las entradas del problema planteado.

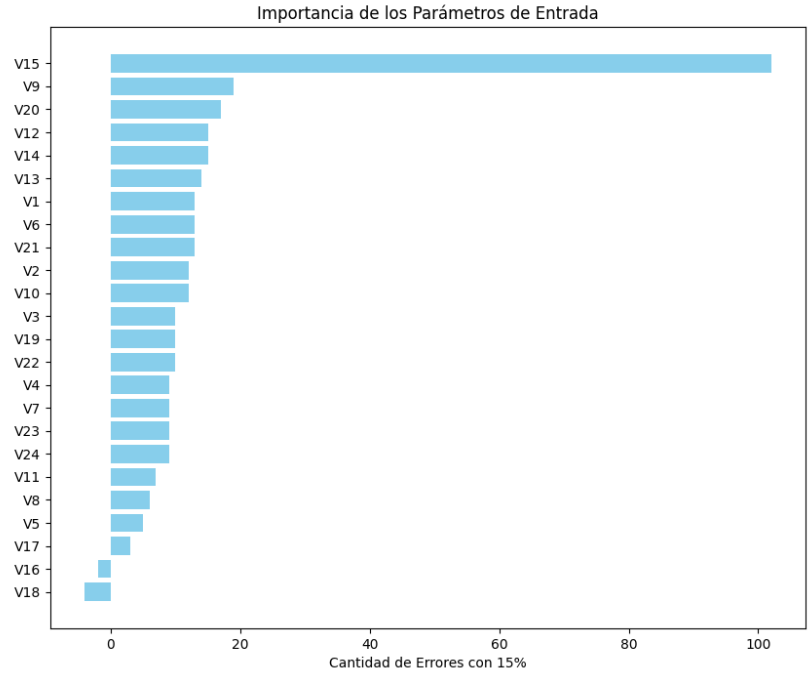

Figura 12. Cantidad de errores nuevos en la predicción al modificar una variable de entrada en 15%.

En la figura 13 se puede observar la cantidad de errores nuevos que se obtuvieron al variar un 20% cada una de las entradas del problema planteado.

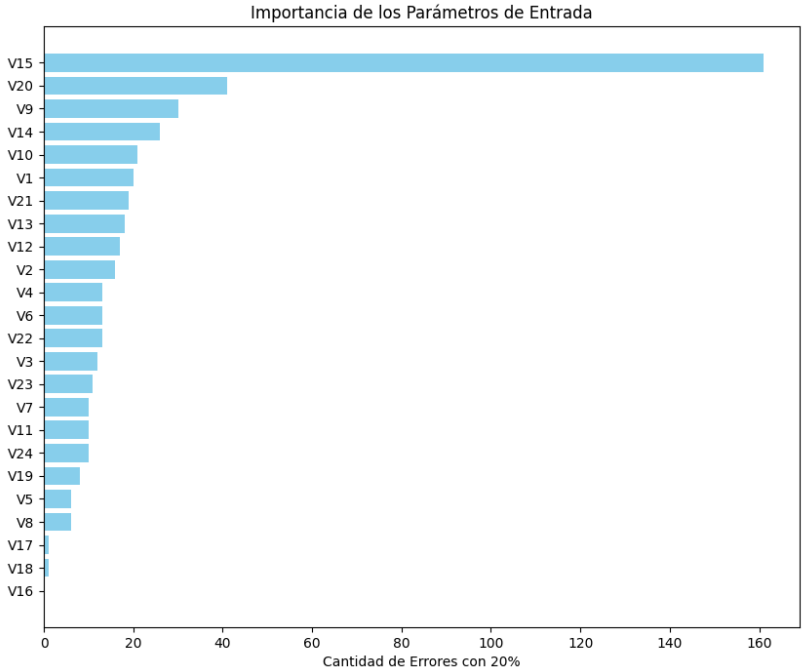

Figura 13. Cantidad de errores nuevos en la predicción al modificar una variable de entrada en 20%.

En la figura 14 se puede observar la cantidad de errores nuevos que se obtuvieron al variar un 25% cada una de las entradas del problema planteado.

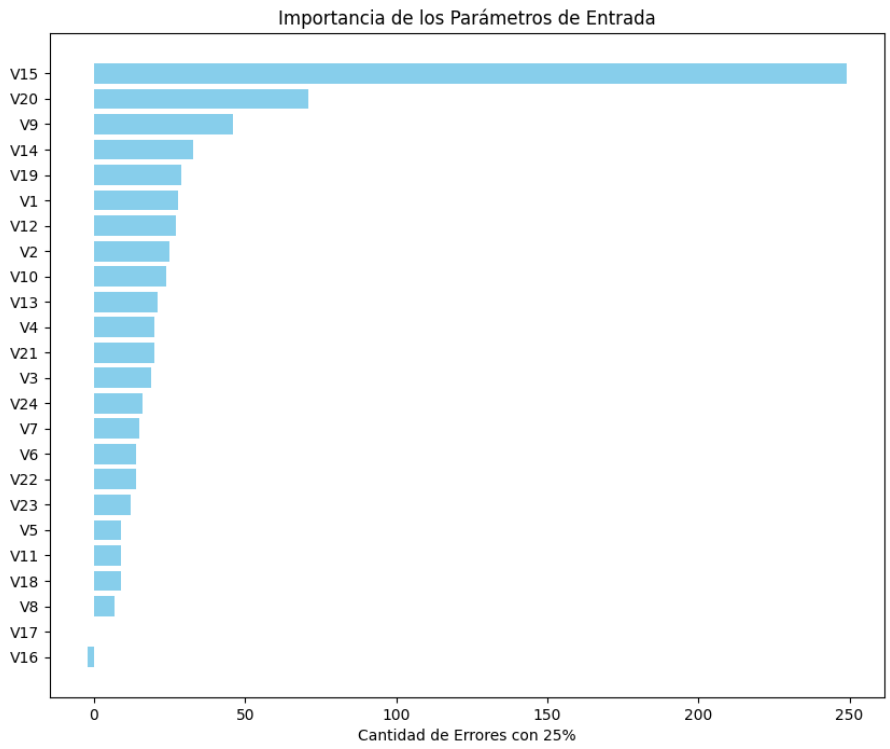

Figura 14. Cantidad de errores nuevos en la predicción al modificar una variable de entrada en 25%.

Análisis de resultados

Estudio de hiperparámetros

En lo que respecta al estudio de hiperparámetros del algoritmo evolutivo, se evaluaron únicamente 3 hiperparámetros, estos corresponden a la población inicial, número de generación y probabilidad de mutación. Se destacan otras elecciones que, bajo circunstancias diferentes, podrían haber sido parte de las variaciones en el estudio de hiperparámetros, sin embargo, por la implicación en recursos computacionales que conlleva este problema en específico, se decidió definir el método de mutación, el método de selección, el método de cruzamiento y la probabilidad de cruzamiento basado en recomendaciones de la literatura y en la naturaleza del problema como tal. Se realizaron 5 iteraciones con distintas combinaciones de valores, dichas iteraciones se plantearon de tal manera que los resultados evidencien un balance entre exploración y explotación.

En la iteración 1, como se puede observar en la tabla 3, se utilizó una población inicial de 70 individuos y una probabilidad de mutación de 15%, mientras que en la iteración 2, como se muestra en la tabla 6, se utilizó la misma muestra de población inicial, pero con una probabilidad de mutación considerablemente menor, únicamente un 5%. Al comparar la figura 3 y la figura 4 se puede observar la notable diferencia en el comportamiento puesto que la primera tiene una desviación estándar que no logra estabilizarse, comportamiento propio de un algoritmo genético con demasiada variabilidad que eventualmente tiene un comportamiento más aleatorio que lo que se podría desear. Por otro lado, en la figura 4 de la iteración 2 se muestra un comportamiento que comienza a converger en las últimas generaciones a una desviación estándar menor, lo que implica una población final con soluciones similares en calidad. En lo que respecta a los resultados puntuales la iteración 1 obtuvo una solución con mejor calidad que la segunda iteración, sin embargo, al realizar iteraciones, documentadas en las tablas 5 y 8, se pudo observar que la segunda iteración obtuvo una solución más estable que la primera. Esto se puede atribuir primero a la naturaleza del problema en que una combinación de hiperparámetros puede converger a diferentes calidades puesto a que la red se entrena y prueba cada vez con grupos de datos distintos y segundo, porque es posible que, en la primera iteración, la solución sobresaliente fuera una excepción que sobresalió de la mayoría de los individuos, pero que no corresponde a un proceso de evolución paulatino como puede ser el caso de la iteración 2.

Del caso de las dos primeras iteraciones, fue posible entender que una alta tasa de mutación no es favorable, por lo que se planteó como estrategia probar aumentar la población inicial y mantener la tasa de mutación baja. Además, se planteó aumentar las generaciones a 15 con el fin de darle una mayor oportunidad al algoritmo para converger a la mejor calidad. En la tercera iteración, con la combinación que se puede observar en la tabla 9, se obtuvo una calidad similar a la de la iteración 1, sin embargo, al analizar la mejor solución en reiteradas ocasiones, se puede observar en la tabla 11, que es una solución aún más estable, con precisiones más altas en todas las iteraciones y pérdida de entrenamiento y validación menor. Esta tercera iteración se puede catalogar como una solución factible para aplicar en la

combinación de hiperparámetros del entrenamiento de la red neuronal artificial, sin embargo, se planteó realizar más iteraciones con el objetivo de buscar una solución aún más acertada.

En la cuarta iteración, como se puede observar en la tabla 12, se aumentó levemente la población inicial y se disminuyó la probabilidad de mutación, esto con el fin de balancear la explotación y la exploración. En el comportamiento de las curvas, observadas en la figura 6, se puede comparar con las curvas de la iteración anterior expuestas en la figura 5, y se puede apreciar una mayor tendencia a la explotación. Esto se puede concluir al observar como la desviación estándar se comporta de manera más gradual conforme la generación empieza a converger a un grupo de soluciones con calidades similares. La calidad del mejor individuo es similar a la iteración anterior, sin embargo, al ponerlo a prueba se pudo observar como se obtienen precisiones levemente más altas y pérdidas prácticamente idénticas. Se puede concluir que se obtuvo una solución levemente mejor a la anterior.

Por último, se planteó una última iteración en la que se aumenta considerablemente la variabilidad con el fin de explorar una solución con calidad mucha mejor. Se aumento un 50% la población inicial planteada en la cuarta iteración, y se aumentó a 2.5%, esto se puede observar en la tabla 15. Al realizar este cambio se obtuvo una solución con mejor calidad, sin embargo, al probar las combinaciones se pudo observar que las pérdidas y la precisión no son mejores a las distintas soluciones. Al observar el historial del algoritmo evolutivo ejecutado en esta iteración, se vio que esta exacta combinación fue evaluada en distintas oportunidades y las calidades resultantes no fueron tan buenas como la evaluación incorporada como mejor solución. Se puede argumentar entonces que la calidad tan buena obtenida fue una excepción resultante de la aleatoriedad presente en el entrenamiento de una red neuronal artificial.

En conclusión, del estudio de hiperparámetros del algoritmo evolutivo, se decidió elegir la mejor solución obtenida por la iteración 4, esto por la estabilidad que presentó al realizar iteraciones independientes para evaluar el comportamiento de la red. Los hiperparámetros de la red neuronal evolutiva se pueden encontrar en la tabla 13.

Resultados del modelo escogido

A continuación, se va a analizar la red escogida como mejor. Para esto, como se mencionó anteriormente se probó la red ya entrenada con tres conjuntos de datos diferentes y seleccionados de manera aleatoria. Como se puede apreciar en las figuras 8, 9 y 10, en los tres casos, una cantidad significativa de los datos fueron clasificados correctamente. Además, en las tablas 18, 19 y 20, lo primero que se puede apreciar es que el valor de precisión más bajo obtenido fue de 0.9597, lo cual indica que la red logró clasificar de forma correcta más del 95% de los datos y que, por ende, limitó el valor de falsos negativos a un valor menor al 5%.

Por otro lado, dentro de las métricas obtenidas, también se puede notar que los valores de recall de las tres iteraciones son bastante buenos e igualmente, el caso más crítico es de aproximadamente un 95%. Este valor lo que indica es que la red tiene un alto grado de

sensibilidad o exhaustividad y que evita de forma acertada la omisión de casos positivos (falsos negativos).

Además de estos, se tiene el f1-score, que igualmente, el caso más crítico fue cercano a un 95%. Este parámetro lo que indica es el grado de balance que tiene la red neuronal en su precisión y exhaustividad. En la mayoría de los casos es deseable que este valor sea lo más alto posible, pues esto lo que indica es que la red tiene una alta capacidad de limitar los falsos positivos, capturando al mismo tiempo la mayoría de las instancias positivas. Para este caso, se puede considerar que un valor del 95% es lo suficiente mente bueno para esta red.

Entonces, a partir de las métricas obtenidas, se puede afirmar que la red tiene la capacidad de clasificar con precisión la mayoría de las muestras, minimizando tanto los falsos positivos como los falsos negativos, lo cual lleva a un rendimiento sólido y equilibrado. Con esto, se puede concluir que el modelo tomado como óptimo tiene una alta capacidad para generalizar a datos diferentes a los usados en el entrenamiento y con un porcentaje de error aceptable para el problema tratado.

Estudio de sensibilidad de parámetros de entrada

En las figuras 11, 12, 13 y 14 se pueden observar gráficos de barras horizontales ordenados que permiten identificar la incidencia de las variables de entrada en las predicciones de la red neuronal. En la figura 11 se muestra el comportamiento de los errores para una variación porcentual de 10%, en este caso se puede observar que las variables con mayor incidencia corresponden a V15, V20, V1, V21 y V6, con una superioridad notable de V15 en lo que respecta a relevancia y las menos relevantes son V5, V17 y V18. Para la figura 12 se muestran los errores nuevos para una variación porcentual de 15%, en este caso se observan con claridad que la más incidente es V15, seguido por, con amplio margen de diferencia, V9, V20 y V12, asimismo, los menos incidentes son V5, V16, V17 y V18. Con respecto a la figura 13 se tienen los resultados para variaciones de 20% y se puede ver que V15, V20, V9 y V14 sobresalen en incidencia, mientras que V16, V18 y V17 prácticamente no tienen afectación en los resultados predichos. Finalmente, en la figura 14 se muestran los resultados al variar las entradas un 25%. Como se puede observar, las variables más relevantes son V15, V20, V9 y V14 y las menos incidentes son V16, V8, V17 y V18. Por lo que al analizar la coincidencia en los distintos porcentajes de variación se puede llegar a la conclusión de que las variables más relevantes para la predicción de resultados son V15 (la más relevante por mucho), V20 y V9, mientras que las variables menos incidentes son V16, V17 y V18.

Conclusiones

1. A partir del estudio de hiperparámetros del algoritmo evolutivo se concluye que la combinación de hiperparámetros que genera la mejor red neuronal es: una población inicial de 100 individuos y con 15 generaciones, torneo de tamaño 2 como método de selección y una probabilidad de mutación de 0.01. Esta combinación permitió una mejor exploración del espacio de búsqueda y una convergencia hacia soluciones de alta calidad.
2. Se obtuvo que una alta tasa de mutación genera resultados menos estables, mientras que una tasa más baja permite una convergencia a soluciones de mayor calidad y estabilidad.
3. Se determinó que era de vital importancia mantener un equilibrio entre la exploración y explotación en el algoritmo evolutivo, de modo que se puedan generar una amplia variedad de soluciones diferentes, pero que con el paso de las generaciones logre converger a una solución con buena calidad y con uso de recursos computacionales razonable.
4. Se demostró que, para una red neuronal, una solución del evolutivo con mejor calidad no siempre significa que genera una mejor red neuronal, ya que el proceso de entrenamiento de una red neuronal tiene factores aleatorios, como la escogencia de las muestras de entrenamiento y prueba, que pueden generar valores de pérdida y validación muy diferentes entre dos o más iteraciones.
5. Se determinó que la red que mejor mapea la tendencia descrita por el conjunto de datos se compone de los siguientes hiperparámetros: 1 capa oculta, 14 neuronas por capa oculta, optimizador ADAM y un valor de momentum de 1, esto respaldado por las métricas de precisión, *recall* y *f1-score*.
6. A partir de las métricas generadas en la etapa de prueba de la red neuronal seleccionada como óptima se puede concluir que el modelo tiene una alta capacidad para generalizar a datos diferentes a los usados en el entrenamiento y con un porcentaje de error mínimo para el problema tratado.
7. Al realizar el análisis de sensibilidad de entradas en el modelo óptimo de la red neuronal se obtuvo que la entrada con mayor influencia es V15, y con una diferencia significativa, esta es seguida por V20 y V9; mientras que las variables menos influyentes son V16, V17 y V18.

Bibliografía

[1] A. P. Engelbrecht, “Computational Intelligence: An Introduction”. (2nd edition, 2007, Wiley).

[2] S. Haykin, “Neural Networks and Learning Machines “. (3th edition, 2009, Pearson).

[3] J. L. Crespo Mariño y F. Meza Obando, «Funciones de pérdida y optimizadores» de Tema 2: Redes Neuronales Artificiales (II.2), 2023.

[4] A. E. Eiben and J. E. Smith, Introduction to Evolutionary Computing, Second. Amsterdam: Springer, 2015.

[5] G. Del Valle Martinez, “Introducción Al Deep Learning III: Métodos de Optimización II,” Medium, https://medium.com/soldai/introducci%C3%B3n-al-deep-learning-iii-m%C3%A9todos-de-optimizaci%C3%B3n-ii-c80b356a7294 (Recuperado: nov. 15, 2023).

[6] “Evolutionary tools,” Evolutionary Tools - DEAP 1.4.1 documentation, https://deap.readthedocs.io/en/master/api/tools.html#deap.tools.mutUniformInt (Recuperado: nov. 17, 2023).

Anexos

Anexo 1

Código del algoritmo evolutivo completo con la red neuronal como parte de la función de calidad.

```
from deap import base, creator, tools, algorithms
import random
import numpy
import array
import pandas as pd
import tensorflow as tf
from keras import models
from keras import layers
from keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix   , accuracy_score,
recall_score, f1_score

# Definición de hiperparámetros


train_percentage = 0.75
iteraciones = 300
LABELS = ['MoveForward', 'SlightRightTurn', 'SharpRightTurn',
'SlightLeftTurn']


def customMutation(individuo, indpb=0.2):
    # Diccionario que contiene los límites de los espacios de alelos
    lim_dict = {0: [1, 6], 1: [1, 14], 2: [0.00001, 0.5], 3: [1, 3], 4: [0,
1]}
    # Se itera a través de cada uno de lo genes
    for index, gen in enumerate(individuo):
        prob = random.random()
        # Si el valor generado con anterioridad es menor a la probabilidad
        # definida por el usuario, se cambia (muta) el gen.
        if prob < indpb:
            new_gen = gen
            # Se repite hasta que se encuentra un gen de valor diferente
            while new_gen == gen:
                # Todos los genes son enteros con excepción del tercero
(index 2)
```

```
                if index != 2:
                    new_gen = random.randint(lim_dict[index][0],
lim_dict[index][1])
                else:
                    new_gen = random.uniform(lim_dict[index][0],
lim_dict[index][1])
                # Se asigna como float porque todos los genes son tipo float
                individuo[index] = float(new_gen)
    # Se retorna el individuo mutado
    return individuo,


def normalizing(df):
    max_val = df.max(axis=0)  # Se obtiene el máximo de cada columna
    min_val = df.min(axis=0)  # Se obtiene el mínimo de cada columna
    range = max_val - min_val  # Se obtiene la diferencia de los dos
    df = (df - min_val)/(range)  # Y se utiliza para normalizarlas
    df = df.astype(float)  # Se asegura que los datos sean tipo float
    return df


# Carga de datos
def load_data():
    data = pd.read_excel("datos.xlsx")  # Se lee el archivo de excel con los
datos formateados
    resultado = data["Class"]
    data.pop("Class")
    data = normalizing(data)
    data = data.replace(0.000000, 0.00001)
    data.insert(0, "Class", resultado)
    return data


def Report(dato, Prediction):
    true_labels = dato["Class"]
    report_final = classification_report(true_labels, Prediction,
                                         target_names=LABELS,
output_dict=True, zero_division=0)
    precision = []
    for label in LABELS:
        precision.append(report_final[label]["precision"])
    return precision


def define_model(network, n_layers, number_neurons, LR, optimizer,
Momentum):
```

```
    # Declaración de la capa de entrada y la primera capa oculta
    network.add(tf.keras.layers.Dense(
        units=number_neurons,      # número de neuronas en la capa oculta
        activation="sigmoid",   # función de activación
        input_shape=(24,)))     # cantidad de neuronas en la capa de entrada

    # Ciclo para crear las capas de neuronas intermedias
    i = 0
    while i < n_layers-1:           # se repite el ciclo para completar el
número de capas deseadas
        network.add(layers.Dense(
            units=number_neurons,      # número de neuronas
            activation="sigmoid"))  # funsión de activación
        i += 1

    # Declaración de la capa de salidas
    network.add(layers.Dense(
        units=4,                # cantidad de neuronas de la capa de salida
        activation="sigmoid"))  # función de activación
    if Momentum == 1:
        Momentum = 0.90
    # Compilación del modelo
    if optimizer == 1:
        # Cambiar por RMSprop
        network.compile(
            optimizer=tf.keras.optimizers.Adam(learning_rate=LR),  # se
define el optimizador y la tasa de aprendizaje
            loss=tf.keras.losses.CategoricalCrossentropy())         # se
define la función de pérdida
    elif optimizer == 2:
        network.compile(
            optimizer=tf.keras.optimizers.SGD(learning_rate=LR,
momentum=Momentum),  # se define el optimizador y la tasa de aprendizaje
            loss=tf.keras.losses.CategoricalCrossentropy())         # se
define la función de pérdida
    elif optimizer == 3:
        network.compile(
            optimizer=tf.keras.optimizers.RMSprop(learning_rate=LR,
momentum=Momentum),  # se define el optimizador y la tasa de aprendizaje
            loss=tf.keras.losses.CategoricalCrossentropy())         # se
define la función de pérdida


def red_neuronal(num_neurons, n_layers, learning_rate, optimizador,
Momentum, data):
```

```
    # Distribución de los datos en sets de entrenamiento y prueba
    trainset = data.sample(frac=train_percentage)  # Se extraen datos para
el entrenamiento
    testset = data.drop(trainset.index)  # Y se le quitan esos mismos al
dataset para crear los datos de prueba
    # Conversión de los vectores de resultado a matrices de clasificación
    y_train = to_categorical(trainset["Class"], num_classes=4)
    y_test = to_categorical(testset["Class"], num_classes=4)

    # Creación del modelo de red neuronal
    network = models.Sequential()
    define_model(network, n_layers=n_layers, number_neurons=num_neurons,
LR=learning_rate, optimizer=optimizador, Momentum=Momentum)
    callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=10)


    # Entrenamiento del modelo
    losses = network.fit(
        x=trainset[trainset.columns[1:]],
        y=y_train,
        validation_data=(
            testset[testset.columns[1:]],
            y_test),
        batch_size=100,  # cantidad de datos en un batch
        callbacks=[callback], epochs=iteraciones,
        verbose=0)     # número máximo de iteraciones

    # Se extrae el historial de error contra iteraciones de la clase
    loss_df = pd.DataFrame(losses.history)
    epochs = len(loss_df)
    loss = loss_df.loc[epochs-1, "loss"]
    val_loss = loss_df.loc[epochs-1, "val_loss"]

    # Se eligen 15 datos al azar
    dato = data.sample(frac=1)
    # print(f"dato: {dato}")
    datoPrueba = dato.drop(columns=["Class"])
    # Y se predice el resultado, luego de revertir la normalización
    result = network.predict(datoPrueba)
    # Se usa la función numpy.argmax para retornar el número de categoría
que predijo la red
    Prediction = np.argmax(result, axis=1)
    # print(Prediction)
    # Se sacan las precisiones para cada una de las categorías
    precision = Report(dato, Prediction)
```

```
    # print(f"La pérdida final es: {loss}")
    # print(f"La pérdida final de validación es: {val_loss}")
    # print(f"La precisión en las distintas categorías en la validación es:
{precision}")

    return precision, loss, val_loss
    # El return depende de lo que se necesite para la función de calidad

def funcionEval(individuo, data):
    Ncapas = individuo[0]
    Nneuronas = individuo[1]
    Learning_rate = individuo[2]
    Optimizador = individuo[3]
    Momentum = individuo[4]
    # Acá es donde se debe de llamar a la red neuronal y manipular los
valores que devuelve para darle una puntuación a cada una de las posibles
soluciones.
    if (Optimizador == 1) and (Momentum == 0):
        error = 10
        return [error, ]
    else:
        avg_precision, loss, val_loss = red_neuronal(Nneuronas, Ncapas,
Learning_rate, Optimizador, Momentum, data)
        sum_precision = sum(avg_precision)
        error = loss + val_loss + (4 - sum_precision)
        print(Ncapas, Nneuronas, Learning_rate, Optimizador, Momentum,
error)
        return [error, ]


def AG():
    data = load_data()
    generaciones = 10
    tamañoPoblación = 100
    mutation_gen = 0.1

    # Definición del toolbox
    toolbox = base.Toolbox()
    # El fitness que manejará el toolbox será una función con el peso 1
    # (maximización con peso unitario para cada atributo)
    creator.create('Fitness_funct', base.Fitness, weights=(-1.0,))
    # El individuo que manejará el toolbox será un array de floats
    creator.create('Individuo', array.array,
                   fitness=creator.Fitness_funct, typecode='f')
```

```
    # Registro de la función de evaluación, usando lo definido previamente
en el código
    toolbox.register('evaluate', funcionEval, data=data)

    # Acá es donde hay que definir distintos atributos con diferentes
rangos:
    toolbox.register('capas', random.randint, a=1, b=6)
    toolbox.register('neuronas', random.randint, a=1, b=14)
    toolbox.register('LR', random.uniform, a=0.00001, b=0.5)
    toolbox.register('optimizador', random.randint, a=1, b=3)
    toolbox.register('momento', random.randint, a=0, b=1)
    # Se genera un atributo de float al azar 3 veces, y se guarda en un
Individuo
    toolbox.register('individuo_gen', tools.initCycle, creator.Individuo,
                     (toolbox.capas, toolbox.neuronas, toolbox.LR,
toolbox.optimizador, toolbox.momento),
                     n=1)
    toolbox.register("Poblacion", tools.initRepeat, list,
toolbox.individuo_gen, n=tamañoPoblación)
    # Para ello, llama unas 30 veces a la función 'individuo_gen', de manera
que
    # queda generada una población de 'Individuo's.
    # Se utiliza la función registrada para generar una población
    popu = toolbox.Poblacion()
    print(popu)

    # Método de cruce de dos puntos
    toolbox.register('mate', tools.cxOnePoint)

    # Para la mutación, se utiliza el método de mutación Gaussiana
    toolbox.register('mutate', customMutation, indpb=0.2)

    # Para la mutación, se utiliza el método de torneo
    toolbox.register('select', tools.selTournament, tournsize=2)

    # Hall of Fame: presentación de los mejores 10 individuos
    hof = tools.HallOfFame(10)

    # Estadísticas del fitness general de la población
    stats = tools.Statistics(lambda indiv: indiv.fitness.values)
    stats.register('avg', numpy.mean)  # Promedio de la gen
    stats.register('std', numpy.std)  # Desviación estándar de los
individuos
    stats.register('min', numpy.min)  # Fitness mínimo de la gen
    stats.register('max', numpy.max)  # Fitness máximo de la gen
```

```
    # Una vez que todo está registrado y establecido, ya se puede comenzar
    # a correr el algoritmo evolutivo.
    popu, logbook = algorithms.eaSimple(popu, toolbox, cxpb=0.45,
                                        mutpb=mutation_gen,
ngen=generaciones, stats=stats, halloffame=hof, verbose=True)
    print('---------------------------')
    print(logbook)

    print(hof)
    # X axis parameter:
    generation = logbook.select("gen")
    # Y axis parameter:
    avg = logbook.select("avg")
    std = logbook.select("std")
    minn = logbook.select("min")
    plt.plot(generation, avg, label="Calidad promedio")
    plt.plot(generation, std, label="Desviación estándar")
    plt.plot(generation, minn, label="Mejor individuo")
    plt.xlabel('Generación')
    plt.ylabel('Valor')
    plt.legend()
    plt.show()


if __name__ == "__main__":
    AG()
```

Anexo 2

Código de la red neuronal que permite entrenarla y verificar las métricas de calidad de la misma.

```
import pandas as pd
import tensorflow as tf
from keras import models
from keras import layers
from keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix, accuracy_score, recall_score,
f1_score

# Definición de hiperparámetros
num_neurons = 13
n_layers = 1
train_percentage = 0.70

def normalizing(df):
    max_val = df.max(axis=0) # Se obtiene el máximo de cada columna
    min_val = df.min(axis=0) # Se obtiene el mínimo de cada columna
    range = max_val - min_val # Se obtiene la diferencia de los dos
    df = (df - min_val)/(range) # Y se utiliza para normalizarlas
    df = df.astype(float) # Se asegura que los datos sean tipo float
    return df


# Carga de datos
data = pd.read_excel("datos.xlsx") # Se lee el archivo de excel con los
datos formateados
#print(data)
resultado = data["Class"]
data.pop("Class")
#print(data)
data = normalizing(data)
data = data.replace(0.000000, 0.00001)
data.insert(0, "Class", resultado)
#print(data)



# Distribución de los datos en sets de entrenamiento y prueba
```

```
trainset = data.sample(frac=train_percentage)  # Se extraen datos para el
entrenamiento
testset = data.drop(trainset.index)  # Y se le quitan esos mismos al dataset
para crear los datos de prueba

# Conversión de los vectores de resultado a matrices de clasificación
y_train = to_categorical(trainset["Class"], num_classes=4)
y_test = to_categorical(testset["Class"], num_classes=4)

# Creación del modelo de red neuronal
network = models.Sequential()

# Declaración de la capa de entrada y la primera capa oculta
network.add(tf.keras.layers.Dense(
    units=num_neurons,      # número de neuronas en la capa oculta
    activation="sigmoid",   # función de activación
    input_shape=(24,)))     # cantidad de neuronas en la capa de entrada

# Ciclo para crear las capas de neuronas intermedias
i = 0
while i < n_layers-1:           # se repite el ciclo para completar el
número de capas deseadas
    network.add(layers.Dense(
        units=num_neurons,      # número de neuronas
        activation="sigmoid"))  # funsión de activación
    i += 1

# Declaración de la capa de salidas
network.add(layers.Dense(
    units=4,                # cantidad de neuronas de la capa de salida
    activation="sigmoid"))  # función de activación

# Compilación del modelo
network.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.019652849063277245),
 # se define el optimizador y la tasa de aprendizaje
    loss=tf.keras.losses.CategoricalCrossentropy())          # se define la
función de pérdida
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=10)

# Entrenamiento del modelo
losses = network.fit(
    x=trainset[trainset.columns[1:]],
    y=y_train,
    validation_data=(
```

```
        testset[testset.columns[1:]],
        y_test),
    batch_size=100, # cantidad de datos en un batch
    callbacks=[callback], epochs=300)     # número máximo de iteraciones

# Se guarda la red creada
# network = models.load_model("red_neuronal_proyecto.keras")

# Se extrae el historial de error contra iteraciones de la clase
loss_df = pd.DataFrame(losses.history)
# Se grafican las curvas de pérdida
loss_df.loc[:, ['loss', 'val_loss']].plot()
# Y se pide que se muestre la ventana en que se graficó
plt.show()

dato = data.sample(frac=546/data.shape[0])
datoPrueba = dato.drop(columns=["Class"])
# Y se predice el resultado, luego de revertir la normalización
result = network.predict(datoPrueba)
# Se usa la función numpy.argmax para retornar el número de categoría que
predijo la red
Prediction = np.argmax(result, axis=1)

# Se muestran los resultados en consola
print("Resultado:")
dato.insert(0, "Predicción", Prediction)
print(dato)

# Display de la matriz de confusión
true_labels = dato["Class"]
plt.figure(figsize=(8,8))
plt.title('Confusion Matrix', size=40, weight='bold')
sns.heatmap(
 confusion_matrix(true_labels, Prediction),
 annot=True,
 annot_kws={'size':14, 'weight':'bold'},
 fmt='d',
 cbar=False,
 cmap='RdPu',
 xticklabels=['MoveForward', 'SlightRightTurn', 'SharpRightTurn',
'SlightLeftTurn'],
 yticklabels=['MoveForward', 'SlightRightTurn', 'SharpRightTurn',
'SlightLeftTurn'])
plt.tick_params(axis='both', labelsize=14)
plt.ylabel('Actual', size=14, weight='bold')
```

```
plt.xlabel('Predicted', size=14, weight='bold')
plt.show()

# Display del reporte de clasificación
print("\n-----------------------------------------------------")
print("Classification report:\n")
print(classification_report(true_labels, Prediction, digits=4,
                            target_names=['MoveForward', 'SlightRightTurn',
                                          'SharpRightTurn',
'SlightLeftTurn']))
print("-----------------------------------------------------")
```

Anexo 3

Código implementado para realizar el estudio de sensibilidad de la solución.

```
# Importación de librerías
import pandas as pd
from keras import models
import numpy as np
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns


def normalizing(df):
    max_val = df.max(axis=0)  # Se obtiene el máximo de cada columna
    min_val = df.min(axis=0)  # Se obtiene el mínimo de cada columna
    range = max_val - min_val  # Se obtiene la diferencia de los dos
    df = (df - min_val)/(range)  # Y se utiliza para normalizarlas
    df = df.astype(float)  # Se asegura que los datos sean tipo float
    return df


# Variables globales con el número de errores
errores = 0  # errores de la red original
errores_aux = 0  # errores de la red modificada

# Se carga el modelo ya definido anteriormente
network = models.load_model("red_neuronal_proyecto.keras")

df = pd.read_excel("datos.xlsx") # Se lee el archivo de excel con los datos
formateados
#print(data)
resultado = df["Class"]
df.pop("Class")
#print(data)
df = normalizing(df)
df = df.replace(0.000000, 0.00001)
df.insert(0, "Class", resultado)

datoPrueba = df.drop(columns=["Class"])  # Genera un df solo con entradas
result = network.predict(datoPrueba)  # se predicen los resultados con el
dataframe de prueba
Prediction = np.argmax(result, axis=1)  # se usa la función numpy.argmax
para extraer el número de clase predicho
# Se muestran los resultados en consola
df.insert(0, "Predicción", Prediction)
true_labels = df["Class"]  # Guarda los resultados correctos
```

```
# Se genera la matriz de confusión
Matrix = confusion_matrix(true_labels, Prediction).ravel()
# Descomentar el siguiente código si se quiere mostrar la matriz de
confusión

plt.figure(figsize=(4,4))
plt.title('Confusion Matrix', size=20, weight='bold')
sns.heatmap(
    confusion_matrix(true_labels, Prediction), annot=True,
annot_kws={'size':14, 'weight':'bold'},
    fmt='d', cbar=False, cmap='RdPu', xticklabels=['Move-Forward', 'Slight-
Right-Turn', 'Sharp-Right-Turn', 'Slight-Left-Turn'],
    yticklabels=['Move-Forward', 'Slight-Right-Turn', 'Sharp-Right-Turn',
'Slight-Left-Turn'])
plt.tick_params(axis='both', labelsize=14)
plt.ylabel('Actual', size=14, weight='bold')
plt.xlabel('Predicted', size=14, weight='bold')
plt.show()

# Ciclo para contar el número de errores en la clasificación de los datos
for a in range(len(Matrix)):
    if a not in [0, 5, 10, 15]:
        errores = errores + Matrix[a]  # se suman todos los datos de la
matriz que no estén lasificaciones incorrectas)
# Variables necesarias para el estudio de sensibilidad
df2 = pd.DataFrame([0], columns=["10%"], index=["Original"])  # Dataframe
para guardar el resultado
col_names = ["10%", "15%", "20%", "25%"]  # Nombres de las columnas
functions_list = [lambda x:x*1.10, lambda x:x*1.15, lambda x:x*1.20, lambda
x:x*1.25]  # lista de funciones anónimas usadas para
# escalar los datos del dataframe
# Ciclo para realizar el estudio de sensibilidad
for j in range(len(functions_list)):
    for i in range(2, len(df.columns)):
        df_aux = df.copy()  # Copia el dataframe original
        # Las próximas tres lineas cambian la columna correspondiente a los
valores modificados
        col = df[df.columns[i]]
        change_col = col.apply(functions_list[j])
        df_aux[df.columns[i]] = change_col
        entrada = df_aux.drop(columns=["Class", "Predicción"])    #  Genera
los valores de entrada a la red
        result_aux = network.predict(entrada, verbose=0)  # Hace la
predicción
```

```
        # Pasa a binario lo que predijo la red
        Prediction_aux = np.argmax(result_aux, axis=1)
        # Genera los valores correctos o de error
        Matrix_aux = confusion_matrix(true_labels, Prediction_aux).ravel()
        # Calcula la cantidad total de errores
        for b in range(len(Matrix_aux)):
            if b not in [0, 5, 10, 15]:
                errores_aux = errores_aux + Matrix_aux[b]
        # Saca la diferencia de errores entre los valores originales y con
la tabla modificada
        diferencia = errores - errores_aux  # Si es positivo quiere decir
que mejoró la red
        errores_aux = 0
        # Guarda el valor en el dataframe
        df2.at[df.columns[i], col_names[j]] = diferencia
    # Se imprimen los resultados
    print(f"Rubros con mayor cantidad de errores para una variación de
{col_names[j]}:")
    print(df2[col_names[j]].nsmallest(5))
    print(f"Rubros con menor cantidad de errores para una variación de
{col_names[j]}:")
    print(df2[col_names[j]].nlargest(5))
print(df2)
df2.to_excel("Estudio de sensibilidad.xlsx")
# Crear el gráfico de barras horizontal
plt.figure(figsize=(10, 8))  # Tamaño del gráfico
plt.barh(df.columns, df2[0], color='skyblue')  # Crear gráfico de barras
horizontal
plt.xlabel('Cantidad de Errores con 15%')  # Etiqueta del eje x
plt.title('Importancia de los Parámetros de Entrada')  # Título del gráfico
plt.gca().invert_yaxis()  # Invertir el eje y para mostrar el parámetro más
importante arriba
plt.show()
```

Anexo 4

Implementación en Python que permitió graficar las variables de mayor incidencia en la clasificación del conjunto de entradas.

```
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_excel("Estudio de sensibilidad.xlsx")
diez = df["10%"][1:]*-1
quince = df["15%"][1:]*-1
veinte = df["20%"][1:]*-1
veinticinco = df["25%"][1:]*-1

# Combinar y ordenar los datos en función de la cantidad de errores
datos_ordenados = sorted(zip(df["Unnamed: 0"][1:], diez), key=lambda x:
x[1], reverse=True)
parametros_ordenados, errores_ordenados = zip(*datos_ordenados)

plt.figure(figsize=(10, 8))  # Tamaño del gráfico
plt.barh(parametros_ordenados, errores_ordenados, color='skyblue')  # Crear
gráfico de barras horizontal
plt.xlabel('Cantidad de Errores con 10%')  # Etiqueta del eje x
plt.title('Importancia de los Parámetros de Entrada')  # Título del gráfico
plt.gca().invert_yaxis()  # Invertir el eje y para mostrar el parámetro más
importante arriba
plt.show()

datos_ordenados = sorted(zip(df["Unnamed: 0"][1:], quince), key=lambda x:
x[1], reverse=True)
parametros_ordenados, errores_ordenados = zip(*datos_ordenados)

plt.figure(figsize=(10, 8))  # Tamaño del gráfico
plt.barh(parametros_ordenados, errores_ordenados, color='skyblue')  # Crear
gráfico de barras horizontal
plt.xlabel('Cantidad de Errores con 15%')  # Etiqueta del eje x
plt.title('Importancia de los Parámetros de Entrada')  # Título del gráfico
plt.gca().invert_yaxis()  # Invertir el eje y para mostrar el parámetro más
importante arriba
plt.show()

datos_ordenados = sorted(zip(df["Unnamed: 0"][1:], veinte), key=lambda x:
x[1], reverse=True)
parametros_ordenados, errores_ordenados = zip(*datos_ordenados)

plt.figure(figsize=(10, 8))  # Tamaño del gráfico
```

```
plt.barh(parametros_ordenados, errores_ordenados, color='skyblue')  # Crear gráfico de barras horizontal
plt.xlabel('Cantidad de Errores con 20%')  # Etiqueta del eje x
plt.title('Importancia de los Parámetros de Entrada')  # Título del gráfico
plt.gca().invert_yaxis()  # Invertir el eje y para mostrar el parámetro más importante arriba
plt.show()

datos_ordenados = sorted(zip(df["Unnamed: 0"][1:], veinticinco), key=lambda x: x[1], reverse=True)
parametros_ordenados, errores_ordenados = zip(*datos_ordenados)

plt.figure(figsize=(10, 8))  # Tamaño del gráfico
plt.barh(parametros_ordenados, errores_ordenados, color='skyblue')  # Crear gráfico de barras horizontal
plt.xlabel('Cantidad de Errores con 25%')  # Etiqueta del eje x
plt.title('Importancia de los Parámetros de Entrada')  # Título del gráfico
plt.gca().invert_yaxis()  # Invertir el eje y para mostrar el parámetro más importante arriba
plt.show()
```