UNCUYO
UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

Trabajo Práctico 3
Machine Learning

Inteligencia Artificial II
Grupo 6
2023

ALUMNO	LEGAJO
Herrera, Martín	12416
Llano, Javier	11711
Perulan, Fernando	11322

Índice

1. Introducción

Introducción: En este informe, se presentarán los resultados obtenidos al resolver los ejercicios propuestos en el trabajo práctico sobre redes neuronales. Los objetivos del trabajo práctico incluían la implementación de medidas de precisión de clasificación (accuracy), el uso de conjuntos de prueba independientes, la incorporación de la técnica de parada temprana utilizando conjuntos de validación, modificación de distintos parámetros, resolución de problemas de regresión y realización de un barrido de parámetros. A continuación, se detallarán los resultados obtenidos y se incluirán gráficas relevantes.

2. Precisión

2.1. Implementación

Para abordar el ejercicio, se utilizaron tres códigos principales:

- `MLP_Clasificacion.py`
- `graficos.py`
- `dividir_datos.py`

Estos códigos se utilizaron en conjunto para implementar las funcionalidades requeridas.

2.2. Desarrollo

Al código proporcionado por la cátedra se le agregó la medida de precisión de clasificación para evaluar el rendimiento del modelo. Esta medida es una métrica importante para determinar qué tan bien clasifica el modelo las muestras de prueba. Para ello, se generó un conjunto de prueba independiente y se calculó la precisión utilizando dicho conjunto. En la figura 1 se aprecian ambos conjuntos.

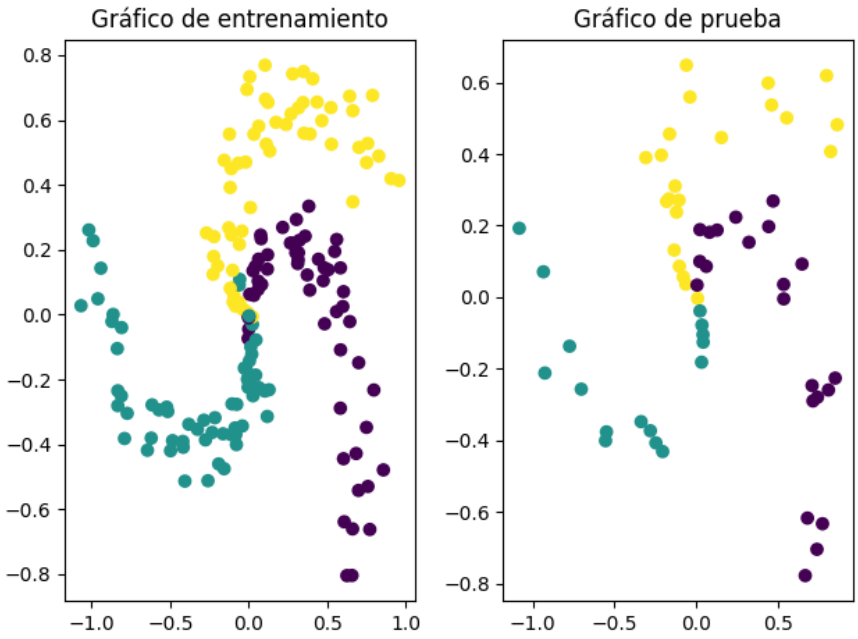


Figura 1: Datos de entrenamiento y de prueba

El cálculo de la precisión se realizó en base a las predicciones del modelo en el conjunto de prueba y las etiquetas reales de las muestras. Se utilizó la función `calcular_precision` para determinarla. La precisión se expresó como un porcentaje de muestras correctamente clasificadas. Los resultados obtenidos mostraron una precisión por encima del 0.94. Esta medida indica qué tan bien el modelo es capaz de clasificar nuevas muestras y es una medida fundamental para evaluar su desempeño en tareas de clasificación.

3. Parada Temprana

3.1. Implementación

Se implementó la técnica de parada temprana para evitar el sobreajuste y mejorar la eficiencia del entrenamiento. Para esto se utilizaron 4 códigos:

- `MLP_Clasificacion.py`
- `graficos.py`
- `dividir_datos.py`
- `parada_temprana.py`

3.2. Desarrollo

La parada temprana consiste en verificar el valor de pérdida o precisión en un conjunto de validación independiente cada cierto número de épocas. Si estos valores empeoran después de

una cierta tolerancia, se detiene el entrenamiento para evitar un rendimiento deficiente o el sobreajuste. En este caso se analizó el valor de pérdida.

En el código proporcionado en `parada_temprana.py`, se utilizó la función `train_parada_temprana` para implementar la lógica de la parada temprana. Se generó un conjunto de validación independiente utilizando la función `dividir_entrenamiento_validacion_prueba` del código `dividir_datos.py`.

Los resultados obtenidos en la figura 2 mostraron que el entrenamiento se detuvo debido a la activación de la parada temprana. Esto indica que el modelo alcanzó su mejor rendimiento en términos de pérdida y evitará el sobreajuste.

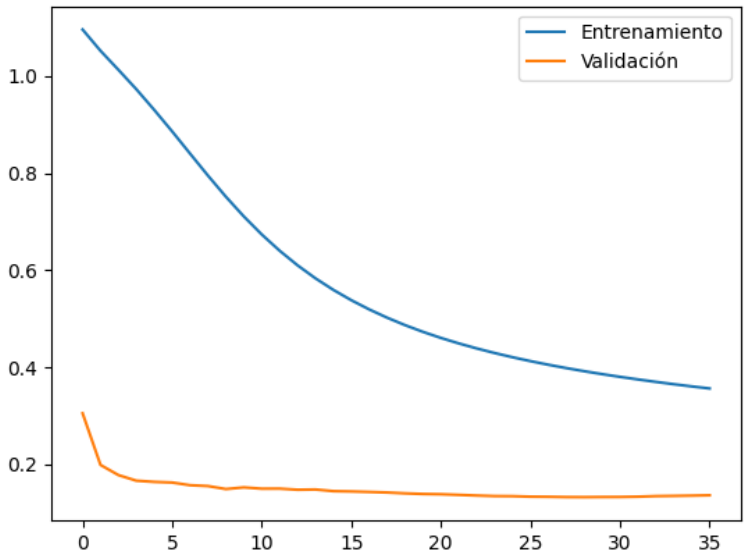


Figura 2: Comparación de loss entre conjunto de entrenamiento y de validación

4. Generador de Datos

Por medio de las pruebas solicitadas se analizó el comportamiento de una red neuronal al experimentar con diferentes configuraciones del generador de datos. Se consideraron dos generadores diferentes: uno básico y otro más complejo.

Para cuantificar el efecto de los cambios y poder medir su impacto se utilizará el cambio en el loss en función de los epoch. El término "loss"(también conocido como función de pérdida o función de costo) se refiere a una medida que indica qué tan lejos está la salida predicha de la red neuronal de la salida deseada o esperada. Representa la discrepancia entre la salida real de la red y la salida deseada, y se utiliza como una medida cuantitativa de qué tan bien está funcionando la red neuronal en una tarea específica.

4.1. Generador Básico

El generador de datos básico se basa en la ecuación paramétrica del círculo (x = r * cos(t), y = r * sin(t)). Se generaron datos distribuidos uniformemente en el círculo, con radios variando entre 0 y 1. Los parámetros que se pueden variar en este generador son:

- Número de Clases: representa la cantidad de clases en las que se divide el círculo. El valor por defecto es 3.
- Número de Ejemplos: indica la cantidad total de ejemplos generados. El valor por defecto es 300.
- Factor de Ángulo: determina la separación entre las clases generadas. Un valor mayor implica una mayor separación angular entre las clases. El valor por defecto es 0.79.
- Amplitud de Aleatoriedad: controla la dispersión de los puntos dentro de cada clase. Un valor mayor genera una mayor dispersión. El valor por defecto es 0.1.

Caso 0: Valores por defecto

En el caso de análisis con los valores por defecto del generador de datos básico, se generarán 300 ejemplos distribuidos en 3 clases en el círculo. El factor de ángulo será de 0.79 y la amplitud de aleatoriedad de 0.1. Esto significa que los puntos se distribuirán uniformemente en el círculo, con radios variando entre 0 y 1. Cada clase estará separada angularmente por un factor de 0.79. Además, se agregará una amplitud de aleatoriedad de 0.1 para generar dispersión en los puntos dentro de cada clase.

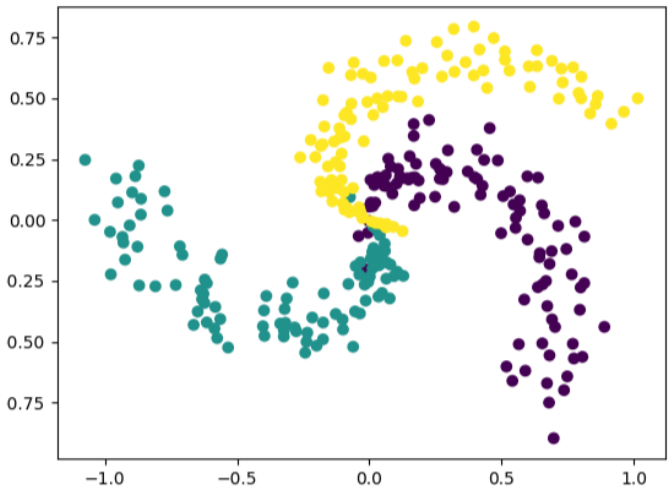


Figura 3: Valores generados por defecto

Para medir el rendimiento del algoritmo y visualizar cómo evoluciona el error durante el entrenamiento, se puede generar una gráfica de "loss"(pérdida) versus .epoch"(época).

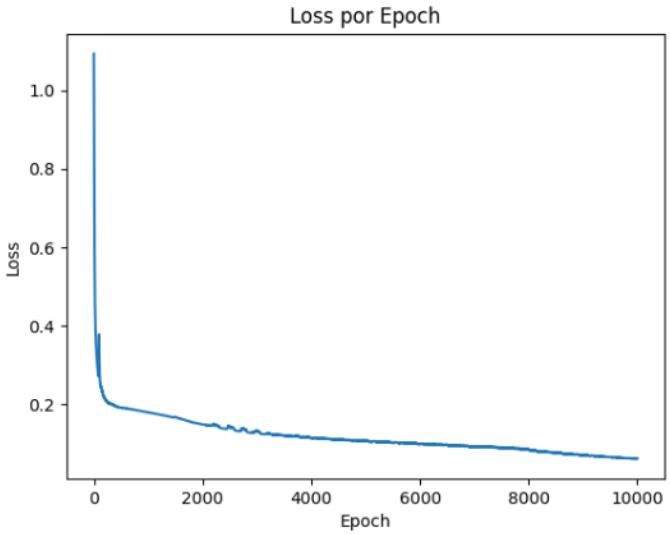


Figura 4: Loss vs Epochs - Parámetros estándar

La figura 4 mostrará cómo evoluciona la pérdida a medida que el modelo se entrena a lo largo de los epochs. Si el modelo está mejorando, esperamos que la pérdida disminuya gradualmente en cada epochs. Por otro lado, si la pérdida no disminuye o aumenta, puede ser un indicio de que el modelo no está aprendiendo correctamente y puede requerir ajustes en los hiperparámetros o en la arquitectura de la red neuronal.

Además de analizar la gráfica de loss vs epochs, se tomaron medidas adicionales para evaluar el rendimiento del algoritmo. Específicamente, se registró el tiempo necesario para completar la etapa de entrenamiento del algoritmo. Esta medida de tiempo nos proporciona información adicional sobre la eficiencia y velocidad de ejecución del algoritmo durante el proceso de entrenamiento.

Es importante destacar que el tiempo de entrenamiento puede verse afectado por varios factores, como el tamaño del conjunto de datos, la complejidad del modelo, los recursos computacionales disponibles y otros parámetros de configuración. Por lo tanto, se realizaron las mediciones en un entorno controlado y se utilizó el mismo conjunto de datos en todas las pruebas para garantizar comparaciones justas y consistentes.

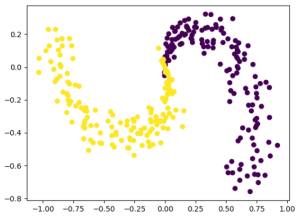
```
#Tiempos#
15.54615330696106
```

Figura 5: Tiempo de entrenamiento Casó 0

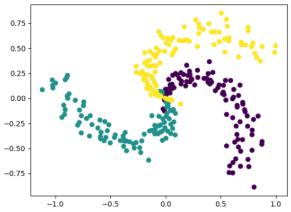
Caso 1: Análisis de la Cantidad de Clases

En el caso 1, se llevó a cabo un análisis para determinar el impacto del cambio en la cantidad de clases en el algoritmo. Para esta prueba, se seleccionaron diferentes valores de ‘numero_clases_test‘, que incluyen 2, 3, 4 y 10. El propósito de este análisis es evaluar cómo varía el comportamiento del algoritmo cuando se divide el círculo en diferentes cantidades de clases.

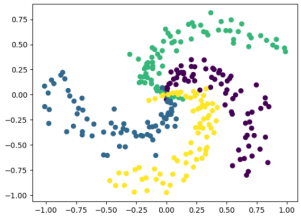
Para visualizar los resultados, se generaron dos tipos de gráficos. En primer lugar, se generó un gráfico que representa la distribución de los datos generados en el círculo para cada cantidad de clases (Figura 6). Esto permite visualizar cómo se distribuyen los puntos generados en el espacio bidimensional del círculo para cada configuración de clases. Al observar estos gráficos, es posible identificar patrones y variaciones en la distribución de los datos en función de la cantidad de clases utilizadas.



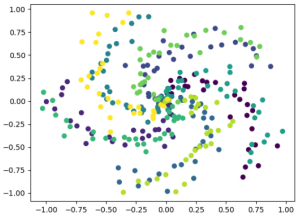
(a) 2 clases



(b) 3 clases



(c) 4 clases



(d) 10 clases

Figura 6: Distribución de datos dependiendo de la cantidad de clases

Por otro lado, se generó un gráfico que relaciona el valor de pérdida (loss) del algoritmo con respecto a los Epoch. La figura 7 proporciona información sobre cómo el algoritmo converge y se ajusta a los datos a medida que avanza el entrenamiento. Al analizar este gráfico para cada cantidad de clases, se pueden obtener conclusiones sobre la estabilidad y el rendimiento del algoritmo en diferentes configuraciones.

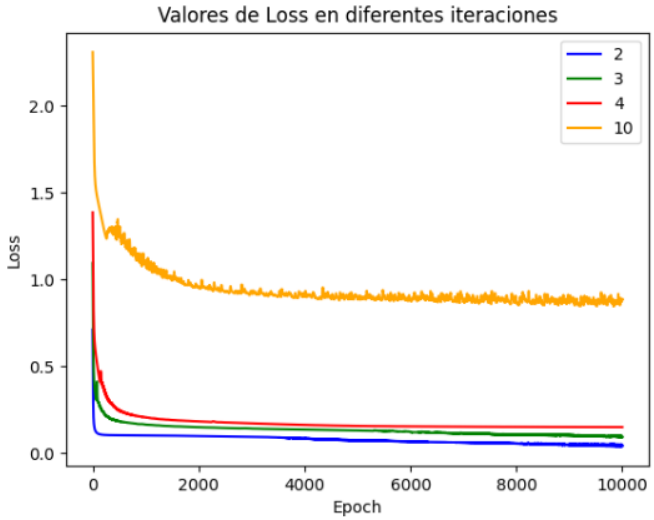


Figura 7

El análisis de estos gráficos nos permitirá comprender cómo el cambio en la cantidad de clases afecta la capacidad del algoritmo para clasificar los datos y cómo influye en su rendimiento general. Estos resultados nos ayudarán a tomar decisiones informadas sobre la configuración óptima del algoritmo en términos de la cantidad de clases a utilizar.

Es importante destacar que el análisis de la cantidad de clases es solo uno de los aspectos a considerar al evaluar el rendimiento del algoritmo. Otros factores, como la precisión de la clasificación y la eficiencia en tiempo de ejecución, también deben tenerse en cuenta al tomar decisiones sobre la configuración adecuada del algoritmo.

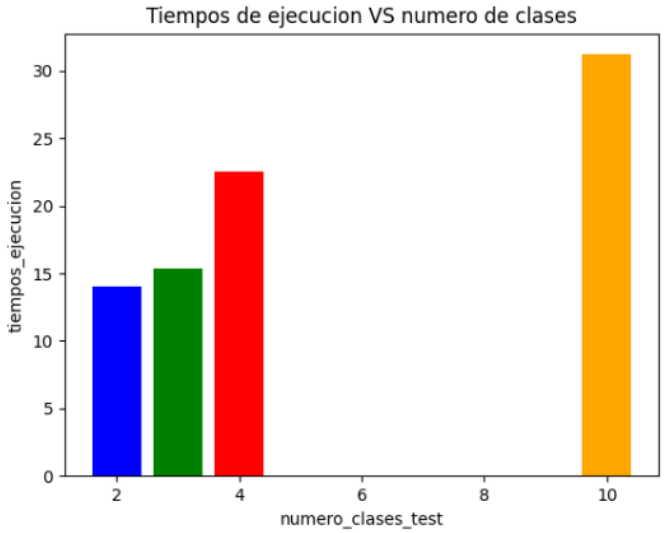
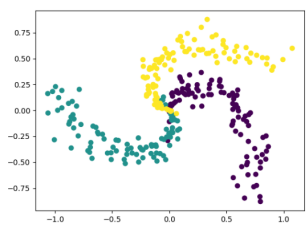
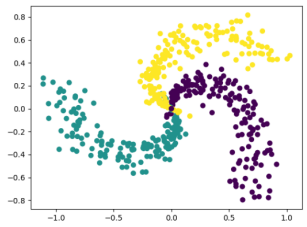


Figura 8

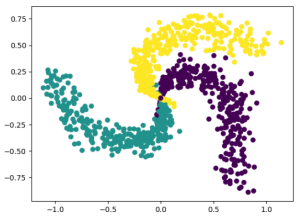
Caso 2: Análisis de la cantidad de ejemplos En el caso 2, se analiza el impacto del cambio en la cantidad de ejemplos en el algoritmo. Para esta prueba, se seleccionaron diferentes valores de ‘numero_ejemplos‘ para evaluar su efecto en el rendimiento del modelo. Los valores de ‘numero_ejemplos_test‘ utilizados en este análisis fueron [300, 500, 1000, 10000].



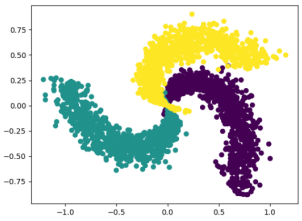
(a) 300 ejemplos



(b) 500 ejemplos



(c) 1000 ejemplos



(d) 10000 ejemplos

Figura 9: Distribución de datos dependiendo de la cantidad de ejemplos

El objetivo de este análisis es determinar cómo la cantidad de ejemplos afecta la capacidad del algoritmo para aprender y generalizar correctamente. Se espera que un mayor número de ejemplos proporcione más información al modelo y, en teoría, debería conducir a un mejor rendimiento. Sin embargo, también se debe considerar el riesgo de sobreajuste cuando se tiene una gran cantidad de ejemplos.

Para cada valor de ‘numero_ejemplos‘, se generaron datos de entrenamiento y se entrenó el modelo utilizando el generador de datos básico. Se registraron los resultados del entrenamiento, incluyendo el gráfico generado y la curva de pérdida (loss) versus los epochs.

Al analizar los gráficos generados, se observará cómo varía el rendimiento del algoritmo en función de la cantidad de ejemplos utilizados. Además, se prestará atención a la evolución de la curva de pérdida a lo largo de los epochs, lo que permitirá evaluar la capacidad del modelo para aprender y mejorar su rendimiento a medida que se incrementa la cantidad de ejemplos.

Estos resultados nos proporcionarán información valiosa sobre la importancia de la cantidad de ejemplos en el rendimiento del algoritmo, lo que permitirá tomar decisiones fundamentadas sobre el tamaño óptimo del conjunto de datos a utilizar en futuras aplicaciones del modelo.

Caso 3: Análisis Factor Angulo

En el caso 3, se realizó un análisis del factor de ángulo en el algoritmo. El factor de ángulo determina la separación entre las clases generadas en el círculo. Un valor mayor implica una mayor separación angular entre las clases, mientras que un valor menor resulta en una menor separación.

Para esta prueba, se tomaron los siguientes valores de factor de ángulo: 0.79, 0.86, 0.92 y 1. Se generaron los gráficos correspondientes para visualizar el resultado obtenido.

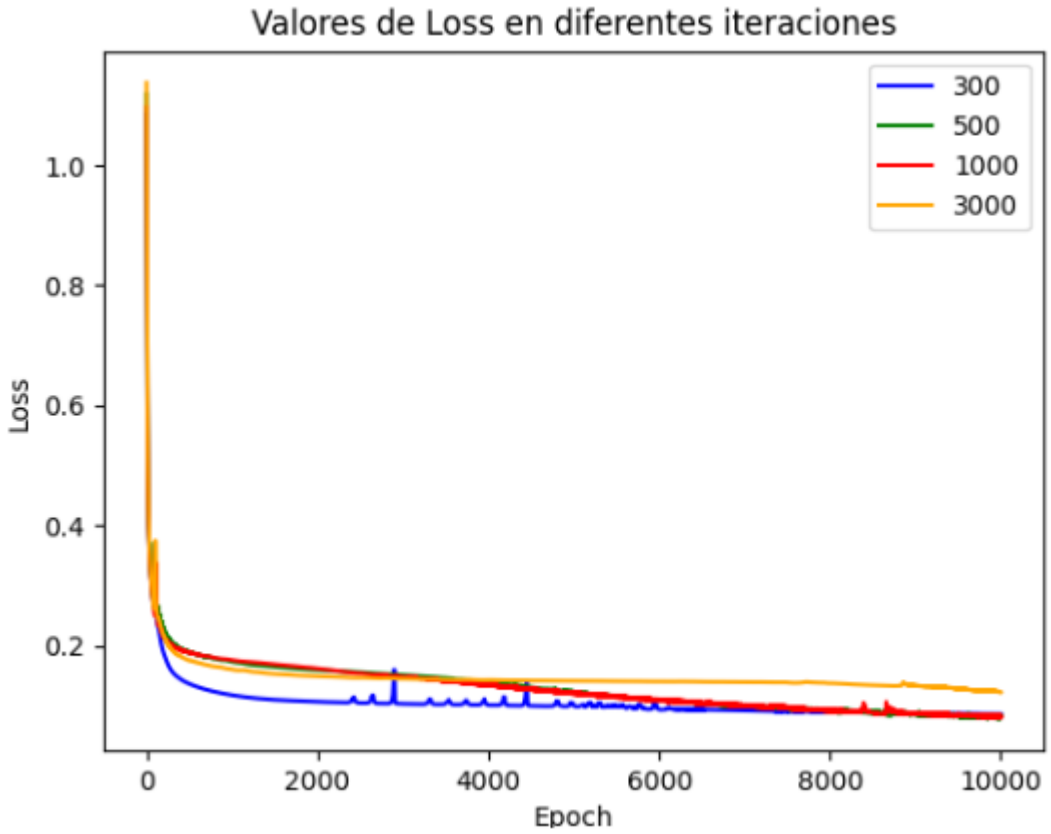

Valores de Loss en diferentes iteraciones
300
500
1000
3000
Loss
1.0
0.8
0.6
0.4
0.2
0
2000
4000
6000
8000
10000
Epoch

Figura 10

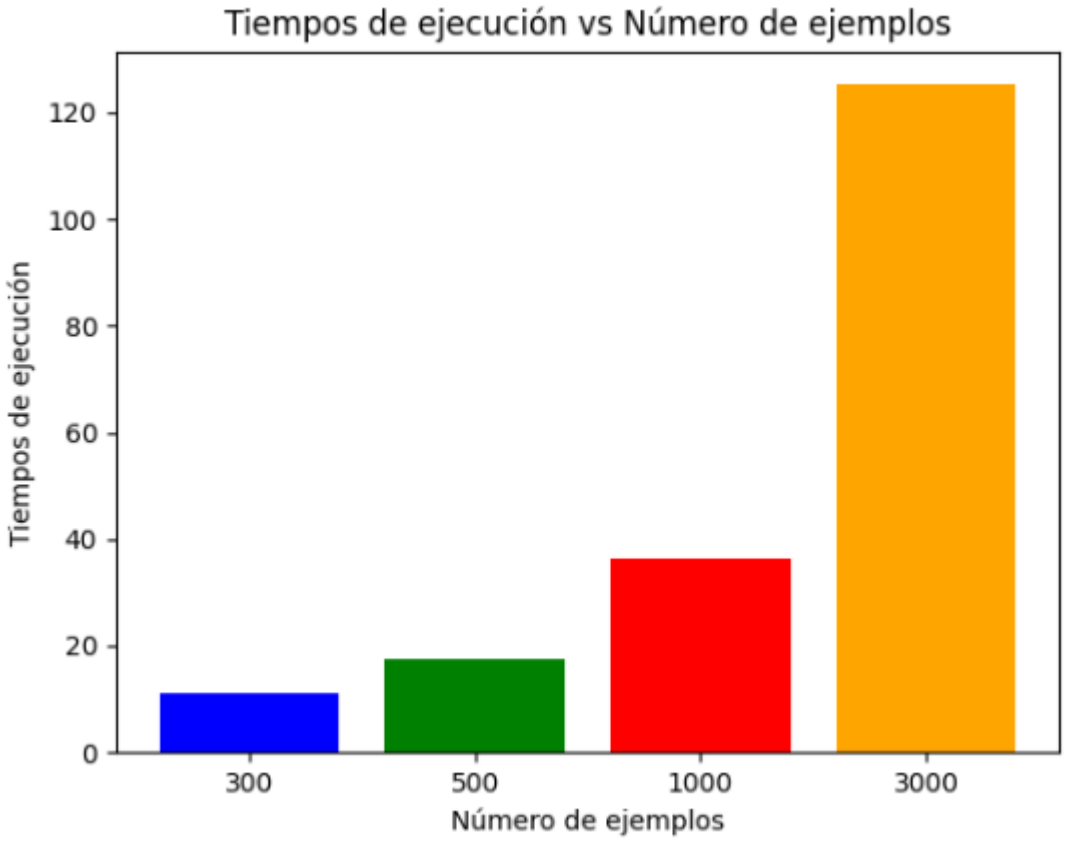
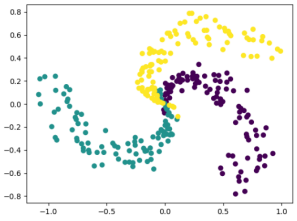
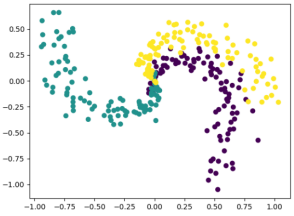

Tiempos de ejecución vs Número de ejemplos
Tiempos de ejecución
0
20
40
60
80
100
120
300
500
1000
3000
Número de ejemplos

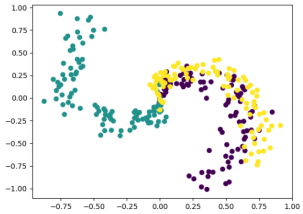
Figura 11



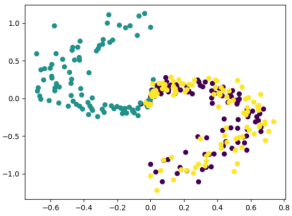
(a) Ángulo de 0,79



(b) Ángulo de 0,86



(c) Ángulo de 0,92



(d) Ángulo de 1

Figura 12: Distribución de datos variando el factor ángulo

Al analizar la figura 12, se observa cómo cambia la distribución de los puntos en el círculo para cada valor de factor de ángulo. Con un factor de ángulo mayor, se puede apreciar una mayor separación entre las clases, lo que implica una mayor discriminación entre ellas. Por otro lado, con un factor de ángulo menor, las clases tienden a estar más cercanas entre sí.

Además del análisis visual, se generó el gráfico que relaciona el loss (pérdida) con los epochs. La figura 13 muestra cómo el factor de ángulo afecta la convergencia y eficiencia del algoritmo durante el entrenamiento. Se pueden observar diferencias en la forma y velocidad de convergencia del loss según el valor del factor de ángulo utilizado.

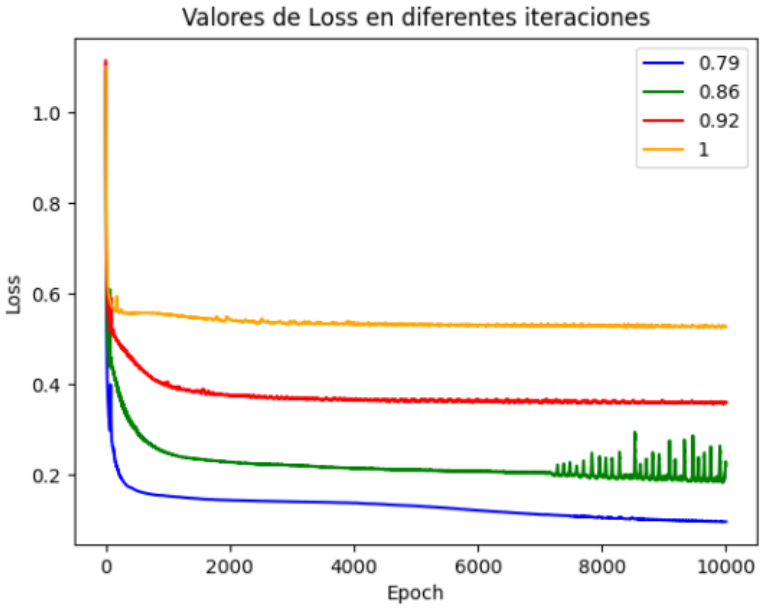


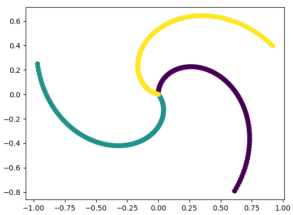
Figura 13

Estos resultados son importantes para comprender el impacto del factor de ángulo en el rendimiento del algoritmo y tomar decisiones adecuadas al ajustar este parámetro en futuras aplicaciones.

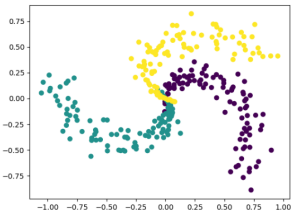
Caso 4 : Análisis Factor Amplitud Aleatoriedad

En el caso 3, se realiza un análisis de la amplitud de aleatoriedad en el algoritmo. Se exploran diferentes valores de AMPLITUD_ALEATORIEDAD_test, que determina la dispersión de los puntos dentro de cada clase. Esta prueba se lleva a cabo con los siguientes valores: AMPLITUD_ALEATORIEDAD_test = [0, 0.1, 0.2, 0.5].

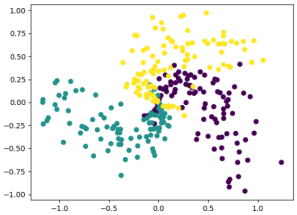
El objetivo es observar el impacto de la aleatoriedad en la distribución de los puntos y cómo afecta el rendimiento del algoritmo. Con un valor de AMPLITUD_ALEATORIEDAD igual a cero, los puntos estarán perfectamente distribuidos en cada clase, sin dispersión. A medida que aumenta la amplitud de aleatoriedad, se introduce un factor de dispersión que afectará la agrupación de los puntos.



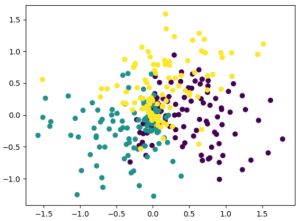
(a) Factor aleatorio de 0



(b) Factor aleatorio de 0,1



(c) Factor aleatorio de 0,2



(d) Factor aleatorio de 0,5

Figura 14: Distribución de datos variando el factor aleatorio

Se analizará el gráfico generado, que muestra la distribución de los puntos en el círculo, considerando los diferentes valores de amplitud de aleatoriedad. Además, se examinará la figura 15, para evaluar cómo varía el rendimiento del algoritmo en cada caso.

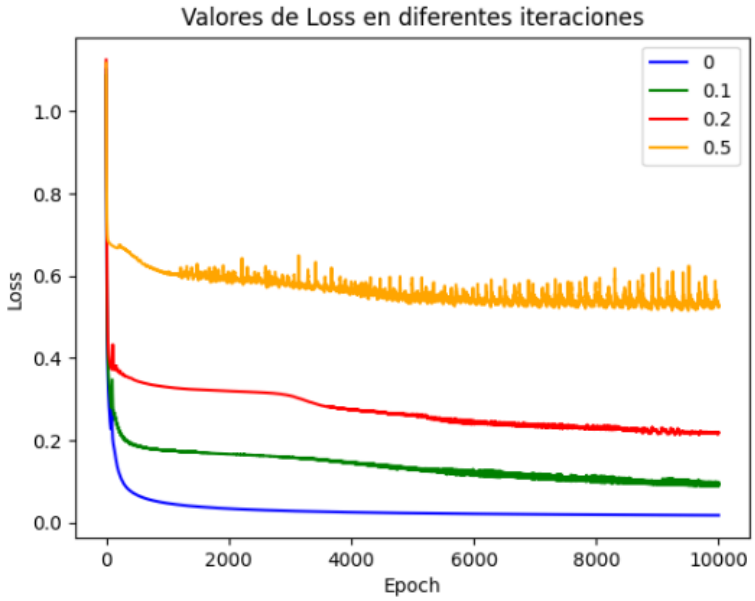


Figura 15

Este análisis permitirá comprender cómo la amplitud de aleatoriedad influye en la distribución y separación de las clases, así como en la capacidad del algoritmo para clasificar correctamente los puntos. Estos resultados proporcionarán información valiosa para la selección óptima del valor de amplitud de aleatoriedad en futuras aplicaciones del algoritmo.

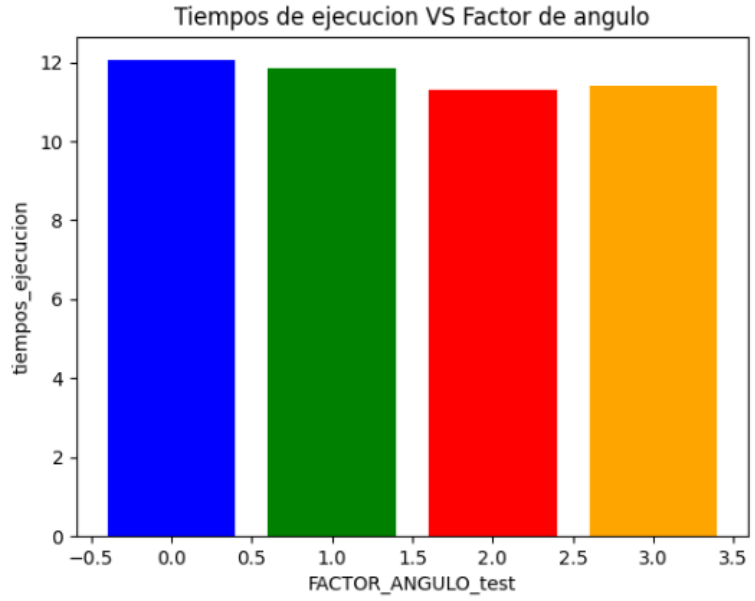


Figura 16

4.2. Generador Complejo

Al variar estos parámetros, se observó que un aumento en el número de clases o en la amplitud de aleatoriedad puede llevar a una mayor superposición entre las clases generadas. Esto dificulta la clasificación correcta de los puntos por parte de la red neuronal.

El generador de datos complejo se basa en la función $Z = -3X + Y^5$. Esta función genera datos con una relación no lineal entre las variables X e Y. Los parámetros que se pueden variar en este generador son los mismos que en el generador básico.

Al utilizar este generador de datos complejo, se observó que la red neuronal puede tener dificultades para aprender y clasificar los datos correctamente. La relación no lineal entre las variables X e Y presenta un desafío adicional para la red neuronal, ya que no puede modelarla de manera directa utilizando una única capa oculta.

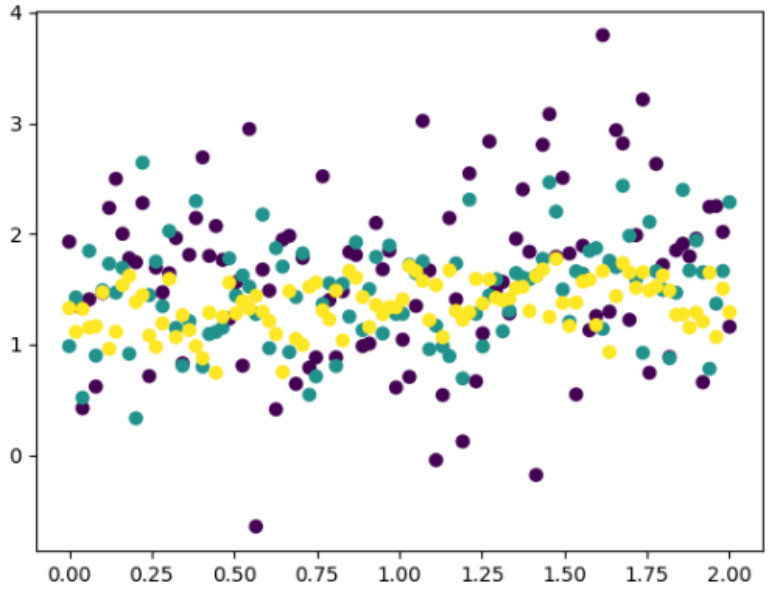


Figura 17: Valores obtenidos por generador complejo

En general, al experimentar con diferentes configuraciones del generador de datos, se encontró que el rendimiento de la red neuronal puede verse afectado por la complejidad y la superposición de las clases generadas. En casos donde las clases están más solapadas o la relación entre las variables es no lineal, la red neuronal puede tener dificultades para lograr una clasificación precisa.

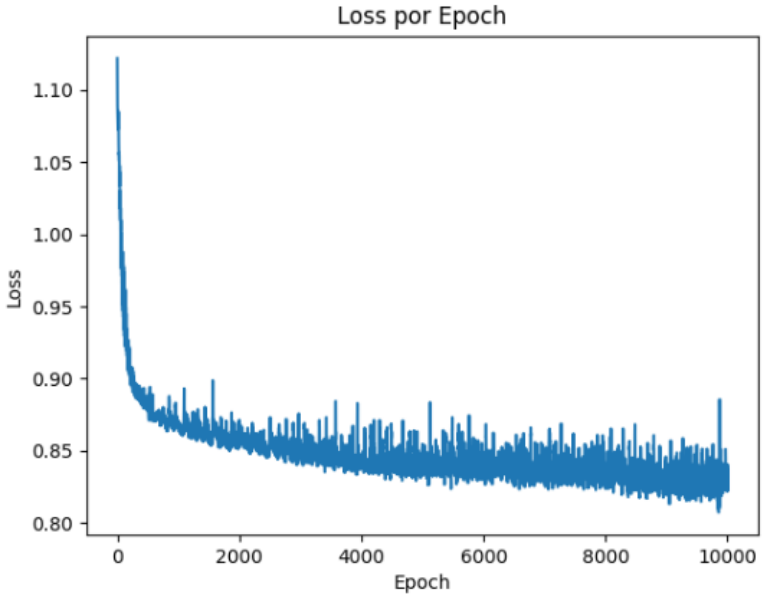


Figura 18

Para abordar estos desafíos, se recomienda ajustar los parámetros del generador de datos de acuerdo con la naturaleza de los datos y el problema en cuestión. Además, se pueden explorar técnicas más avanzadas de redes neuronales, como el uso de capas ocultas adicionales o el uso de funciones de activación no lineales, para abordar problemas más complejos de clasificación.

```
#Tiempos#
13.097317457199097
```

Figura 19

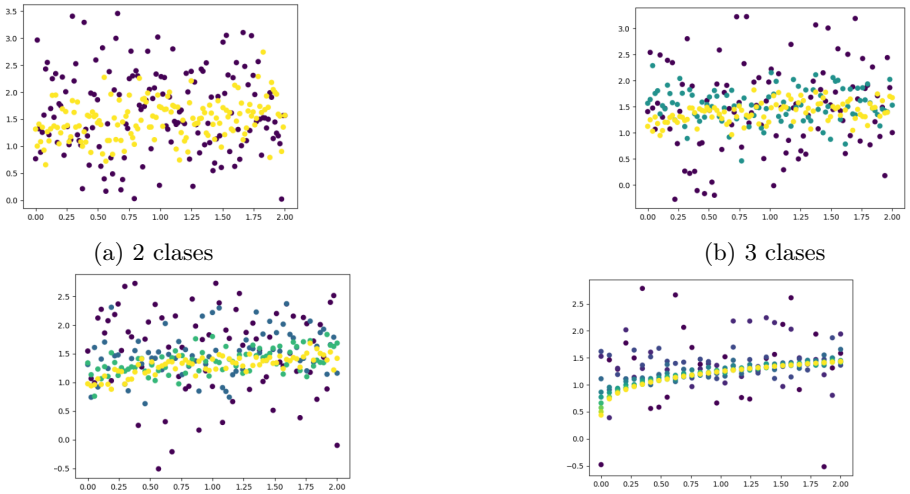
En el análisis con el generador complejo, se utilizaron los mismos parámetros que en el generador básico, que son los valores por defecto establecidos:

- FACTOR_ANGULO: 0.79
- AMPLITUD_ALEATORIEDAD: 0.1
- numero_clases: 3
- numero_ejemplos: 300

Estos parámetros fueron utilizados como base para generar los datos en el generador complejo, donde se introdujo una función más compleja para la generación de puntos dentro del círculo. Sin embargo, se mantuvieron constantes los valores de los parámetros mencionados para realizar una comparación directa y por simplicidad.

Caso 1: Análisis de la Cantidad de Clases

En el caso 1, al igual que en el caso 1 del generador simple, se llevó a cabo un análisis para determinar el impacto del cambio en la cantidad de clases en el algoritmo. Para esta prueba, se seleccionaron diferentes valores de 'numero_clases_test', que incluyen 2, 3, 4 y 10. El propósito de este análisis es evaluar cómo varía el comportamiento del algoritmo cuando se divide el círculo en diferentes cantidades de clases.



(a) 2 clases

(b) 3 clases

(c) 4 clases

(d) 10 clases

Figura 20: Distribución de datos comlejos dependiendo de la cantidad de clases

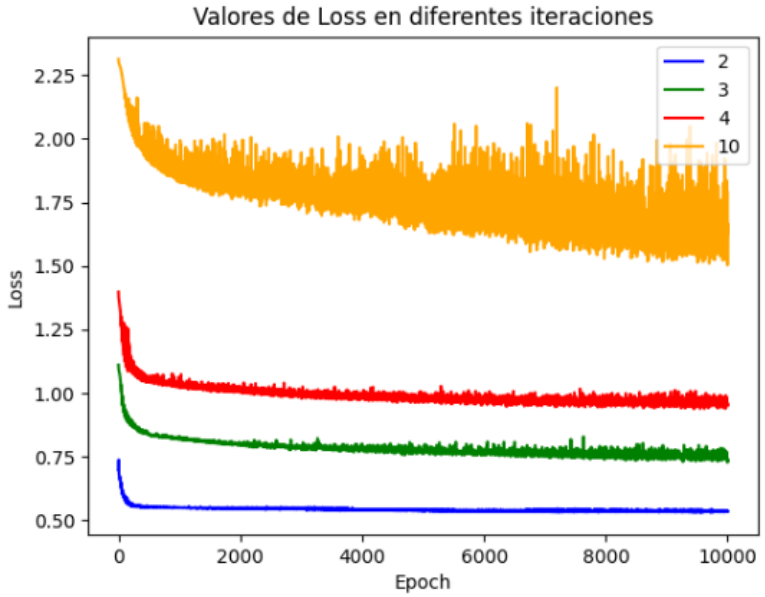


Figura 21

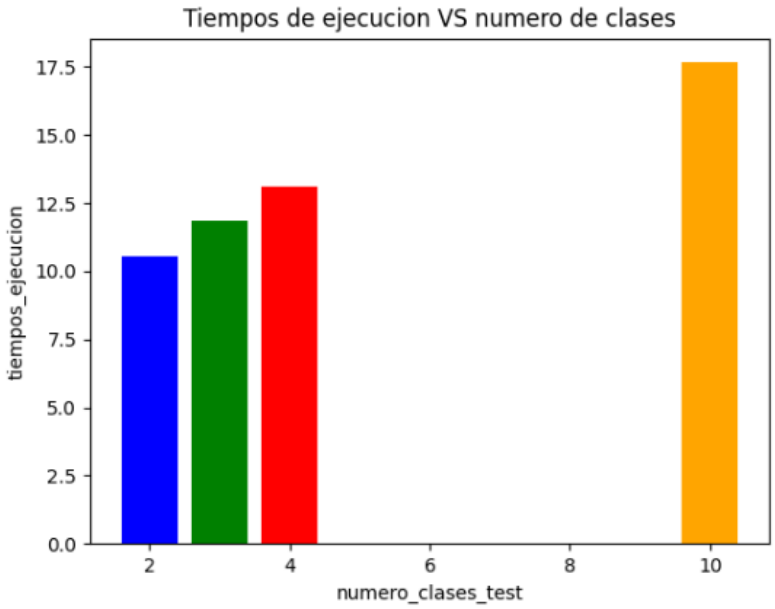


Figura 22

Caso 2: Análisis de la cantidad de ejemplos

En el caso 2, se analiza el impacto del cambio en la cantidad de ejemplos en el algoritmo. Para esta prueba, se seleccionaron diferentes valores de 'numero_ejemplos' para evaluar su efecto en el rendimiento del modelo. Los valores de 'numero_ejemplos_test' utilizados en este análisis fueron [300, 500, 1000, 10000]. El objetivo de este análisis es determinar cómo la cantidad de ejemplos afecta la capacidad del algoritmo para aprender y generalizar correctamente.

Caso 3: Análisis Factor Angulo

En el caso 3, se realizó un análisis del factor de ángulo en el algoritmo. El factor de ángulo determina la separación entre las clases generadas en el círculo. Un valor mayor implica una mayor separación angular entre las clases, mientras que un valor menor resulta en una menor separación.

Para esta prueba, se tomaron los siguientes valores de factor de ángulo: 0.79, 0.86, 0.92 y 1. Se generaron los gráficos correspondientes para visualizar el resultado obtenido.

Caso 4 : Análisis Factor Amplitud Aleatoriedad

En el caso 3, se realiza un análisis de la amplitud de aleatoriedad en el algoritmo. Se exploran diferentes valores de AMPLITUD_ALEATORIEDAD_test, que determina la dispersión de los puntos dentro de cada clase. Esta prueba se lleva a cabo con los siguientes valores: AMPLITUD_ALEATORIEDAD_test = [0, 0.1, 0.2, 0.5].

El objetivo es observar el impacto de la aleatoriedad en la distribución de los puntos y cómo afecta el rendimiento del algoritmo. Con un valor de AMPLITUD_ALEATORIEDAD igual a cero, los puntos estarán perfectamente distribuidos en cada clase, sin dispersión. A medida que aumenta la amplitud de aleatoriedad, se introduce un factor de dispersión que afectará la agrupación de los puntos.

5. Regresión

Se modificó el código para que resuelva problemas de regresión, las principales modificaciones necesarias fueron:

- Las funciones fun y generar_datos_regresion

```
def fun(x):
    return 0.5 * np.sin(3 * x)

def generar_datos_regresion(cantidad_ejemplos, funcion, rango_min=-1,
rango_max=1):
    x = np.linspace(rango_min, rango_max, cantidad_ejemplos).reshape(-1, 1)
    amplitud = funcion(x)
    escala = np.random.uniform(0.85, 1.15, cantidad_ejemplos)
    t = amplitud * escala.reshape(-1, 1)
    return x, t
```

La función 'fun(x)' define la función generadora de los datos de entrenamiento y prueba. Luego, la función 'generar_datos_regresion(cantidad_ejemplos, funcion, rango_min, rango_max)' genera los datos de entrenamiento y prueba a partir de la función 'fun(x)'. Se crea un arreglo x y se aplica la función correspondiente para obtener la amplitud. Al multiplicar la amplitud por el arreglo escala, se obtiene un conjunto de datos ligeramente dispersos en lugar de seguir exactamente la función dada. El retorno de la función son las entradas x y las salidas deseadas t.

- La función de pérdida: calculamos el Error Cuadrático Medio.

```
def regresion_loss(t, y):
    mse = np.mean((t - y) ** 2)
    deriv = -2 * (t - y) / len(t)
    return mse, deriv
```

- La función **hacer_regresion** que implementa el *feedforward* (paso hacia adelante) y retorna las salidas **y**.

```
def hacer_regresion(x, pesos):
    resultados_feed_forward = ejecutar_adelante(x, pesos)
    y = resultados_feed_forward["y"]
    return y
```

5.1. Proceso de entrenamiento

1. Generación de datos de regresión
2. Inicialización de pesos:
   - La función inicializar_pesos(n_entrada, n_capa_oculta) inicializa los pesos de la red neuronal. Recibe el número de neuronas en la capa de entrada (n_entrada) y el número de neuronas en la capa oculta (n_capa_oculta). Devuelve un diccionario pesos que contiene las matrices de pesos w1, w2 y los sesgos b1, b2.
   - La función inicializar_pesos(n_entrada, n_capa_oculta) inicializa los pesos de la red neuronal. Recibe el número de neuronas en la capa de entrada (n_entrada) y el número de neuronas en la capa oculta (n_capa_oculta).
   - Devuelve un diccionario pesos que contiene las matrices de pesos w1, w2 y los sesgos b1, b2.
3. Feed-forward de la red neuronal:
   - La función ejecutar_adelante(x, pesos) realiza el cálculo del feedforward de la red neuronal. Recibe una matriz de entrada x y el diccionario pesos. Calcula las salidas de la capa oculta (h) y la capa de salida (y). Utiliza la función de activación ReLU en la capa oculta, y en la salida la función identidad.
   - Devuelve un diccionario con las salidas de cada capa:

$$"z" : z, "h" : h, "y" : y$$

4. Cálculo de la pérdida y derivadas: Se calcula el error cuadrático medio (MSE) entre los valores objetivo (t) y las predicciones (y). Y también la derivada del error con respecto a las salidas (y) para su posterior uso en la retropropagación.
5. Retropropagación y actualización de pesos:

   La función train realiza el entrenamiento de la red neuronal. Recibe:

   - las matrices de entrenamiento x y t,
   - las matrices de validación x_val y t_val, los pesos iniciales pesos,
   - la tasa de aprendizaje (learning_rate),
   - el número de épocas (epochs),
   - el intervalo de validación (validation_interval)
   - una tolerancia para detener el entrenamiento tempranamente.

   Durante cada época, realiza el feed-forward, calcula la pérdida y las derivadas, realiza la retropropagación para actualizar los pesos de la red utilizando el descenso del gradiente.

   Al final de cada intervalo de validación, calcula la pérdida en el conjunto de validación y verifica si ha habido una mejora significativa en la pérdida. Si no hay mejora, se detiene el entrenamiento tempranamente.

   Devuelve listas con el historial de pérdida de entrenamiento, pérdida de validación y las iteraciones de validación.
6. Visualización de datos y pérdida:

- La función visualizar_datos(x, t, y) muestra una gráfica con los datos de entrenamiento (x, t) y la curva de regresión generada (y).
- La función visualizar_loss(loss_history, val_loss_history, it_list) muestra una gráfica que representa la evolución de la pérdida de entrenamiento y validación a lo largo de las iteraciones.

7. Función principal:

   - La función iniciar(numero_ejemplos, n_entrada, n_capa_oculta, epochs, learning_rate, graficar_datos, graficar_loss) es la función principal que se encarga de iniciar el entrenamiento y controlar la ejecución del programa.
   - Genera los datos de entrenamiento y validación, inicializa los pesos de la red, llama a la función de entrenamiento (train), y opcionalmente muestra las gráficas de datos y pérdida.
   - Devuelve los pesos entrenados.

Testeo

En el fragmento final del código, se llama a la función principal iniciar con los parámetros deseados para iniciar el entrenamiento y realizar una prueba con un conjunto de datos de prueba. Se imprimen el MSE en el conjunto de prueba y se muestran las gráficas correspondientes.

- La función test(x_test, t_test, pesos) realiza la prueba del modelo entrenado utilizando un conjunto de datos de prueba (x_test, t_test). Calcula las predicciones y el MSE en el conjunto de prueba, y muestra una gráfica con los datos de prueba y las predicciones generadas.
- Devuelve el MSE en el conjunto de prueba.

5.2. Análisis de resultados

Se implementaron distintas funciones al momento de ejecutar el algoritmo, también se probó con distintos rangos para los datos de entrada y de acuerdo a esto se realizaron algunas variaciones "intuitivas" en los hiperparámetros para conseguir una salida razonable.

Caso 1
rango de entrada: [-2, 2]
learning rate: 0.15

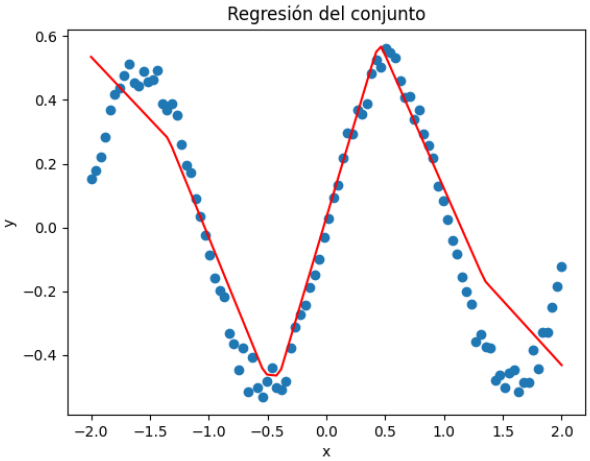


Figura 23

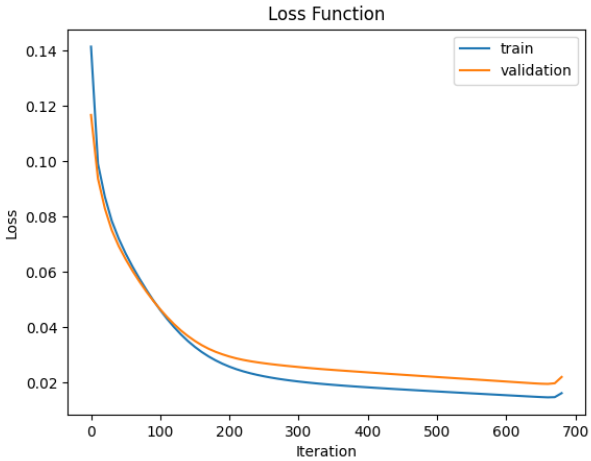


Figura 24

Caso 2
rango de entrada: [-1, 1]
learning rate: 0.15

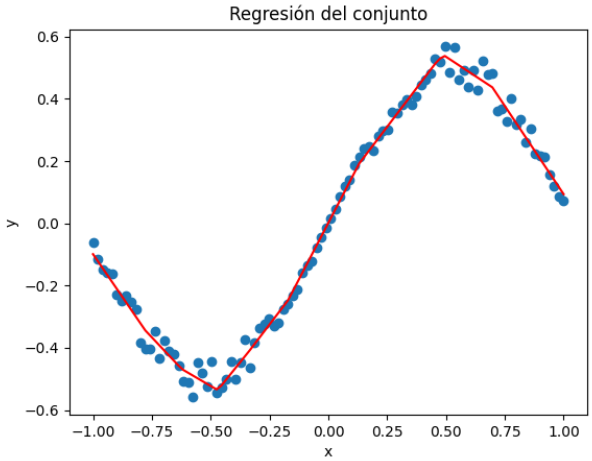


Figura 25

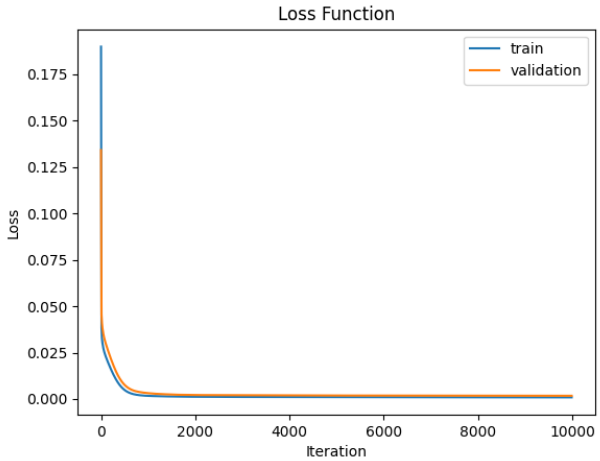


Figura 26

Caso 3
rango de entrada: [-2, 2]
learning rate: 0.05

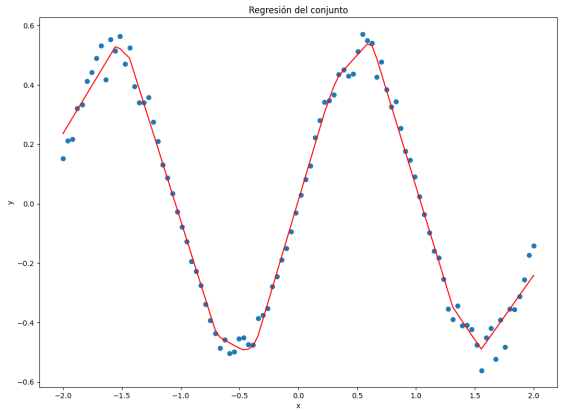


Figura 27

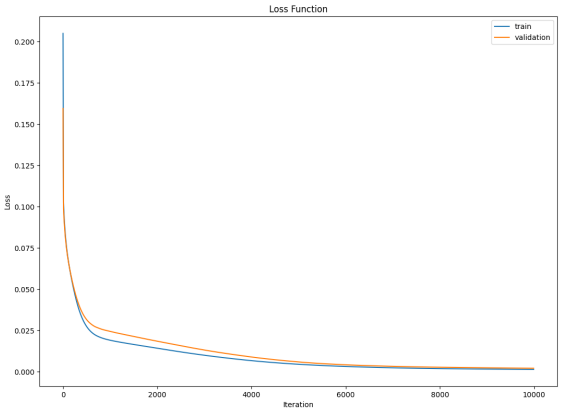


Figura 28

Lo observado se explica en que un aumento en el rango de los datos de entrada significa mayor separación entre los puntos de salida, por lo que para conseguir una regresión suave y adecuada el paso en la dirección del gradiente deben ser menores a la hora de actualizar los pesos, es decir, que la tasa de aprendizaje debe ser menor.

A continuación se presenta la salida del conjunto de prueba para el caso 3.

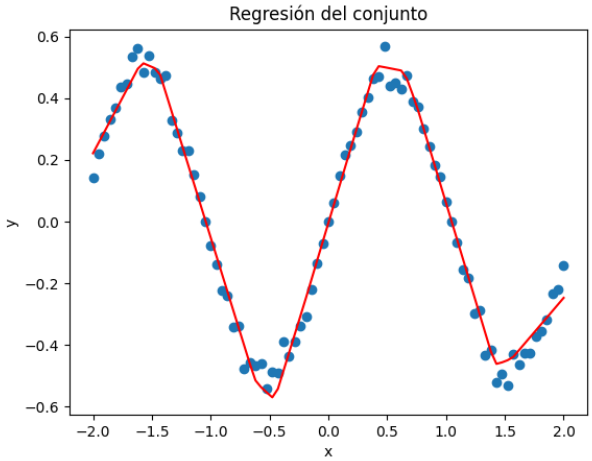


Figura 29

MSE en el conjunto de prueba: 0.0016506955461622335

Red Neuronal con dos capas de entrada

Luego de realizar el análisis anterior, se partió del correspondiente código y se lo modificó para lograr una implementación con dos capas ocultas, con el objetivo de, por un lado realizar una implementación práctica de una capa adicional, y por otro lado caracterizar y evaluar como varía la salida respecto de la red monocapa, principalmente en funciones algo más complejas y rangos mayores.

Las principales modificaciones:

- Se agregó la nueva capa n_capa_oculta2 en la función inicializar_pesos, y junto a los pesos y sesgos correspondientes

```
def inicializar_pesos(n_entrada, n_capa_oculta1, n_capa_oculta2, n_salida):
    randomgen = np.random.default_rng()
    w1 = 0.1 * randomgen.standard_normal((n_entrada, n_capa_oculta1))
    b1 = 0.1 * randomgen.standard_normal((1, n_capa_oculta1))
    w2 = 0.1 * randomgen.standard_normal((n_capa_oculta1, n_capa_oculta2))
    b2 = 0.1 * randomgen.standard_normal((1, n_capa_oculta2))
    w3 = 0.1 * randomgen.standard_normal((n_capa_oculta2, n_salida))
    b3 = 0.1 * randomgen.standard_normal((1, n_salida))
    return {"w1": w1, "b1": b1, "w2": w2, "b2": b2, "w3": w3, "b3": b3}
```

Figura 30

- En la función ejecutar_adelante, se añadió la activación ReLU después de cada capa oculta.

```python
def ejecutar_adelante(x, pesos):
    z1 = x.dot(pesos["w1"]) + pesos["b1"]
    h1 = np.maximum(0, z1)
    z2 = h1.dot(pesos["w2"]) + pesos["b2"]
    h2 = np.maximum(0, z2)
    y = h2.dot(pesos["w3"]) + pesos["b3"]
    return {"z1": z1, "h1": h1, "z2": z2, "h2": h2, "y": y}
```

Figura 31

- En la función train, se calcularon las derivadas dL_dz2 y dL_dz1 correspondientes a las capas ocultas adicionales.

```python
        dL_dh2 = d.dot(w3.T)
        dL_dz2 = dL_dh2.copy()
        dL_dz2[z2 <= 0] = 0
```

Figura 32

Se ejecutó el algoritmo de entrenamiento para: funcion de datos: $f(x) = (np.exp(-x**2)) * x$
rango: [-8, 8]
learning rate = 0.05
n_capa_oculta1 = 200 (igual a capa única)
n_capa_oculta2 = 200
numero_ejemplos = 300
Red de una única capa oculta

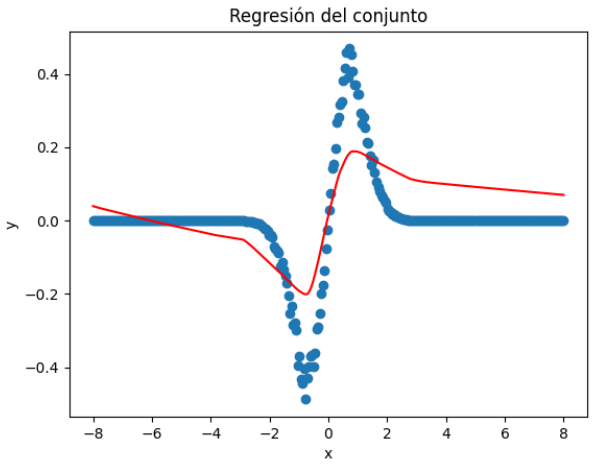


Figura 33

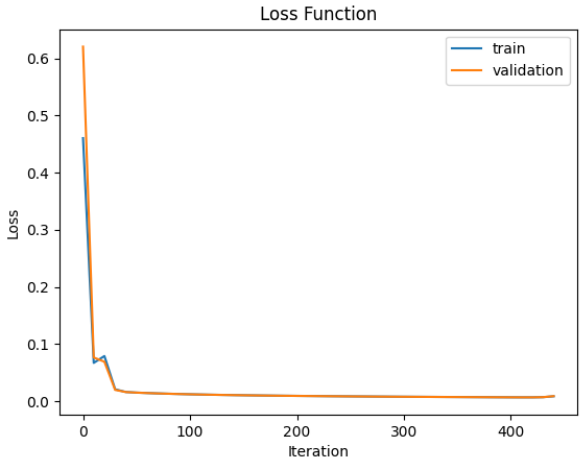


Figura 34

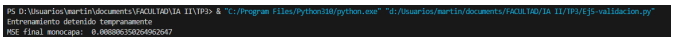


Figura 35

Red de dos capas ocultas

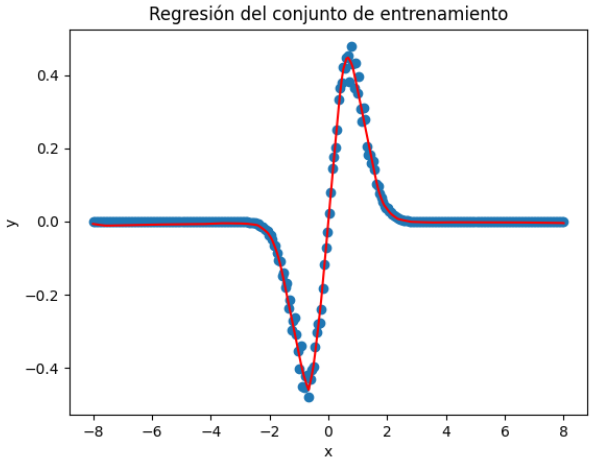


Figura 36

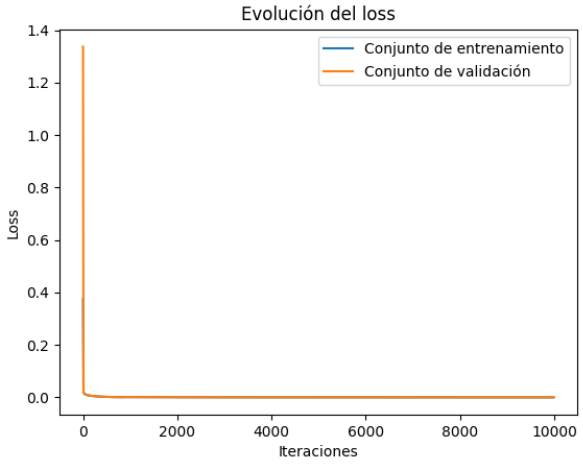


Figura 37

Figura 38

Los resultados obtenidos demuestran que una red neuronal de dos capas ocultas al tener una capa adicional de neuronas ocultas, que pueden procesar y combinar las activaciones de la capa oculta anterior, logra una mayor capacidad para capturar relaciones no lineales y aprender representaciones más sofisticadas de los datos. Las neuronas en la segunda capa oculta pueden combinar características aprendidas en la primera capa oculta y generar representaciones aún más abstractas y relevantes para la predicción final.

6. Barrido de parámetros

Se desarrolló un codigo python para optimizar los hiperparámetros de la red neuronal. Los hiperparámetros son valores que no se aprenden durante el entrenamiento, sino que se establecen antes de comenzar el proceso de entrenamiento.

Para el barrido, se dividió el conjunto de datos en tres partes: entrenamiento, validación y prueba. El conjunto de entrenamiento se utilizó para entrenar la red neuronal, mientras que el conjunto de validación para verificar la performance de los valores de los hiperparámetros. Una vez seleccionados los valores óptimos de los hiperparámetros, se prosiguió con la realización de la prueba el rendimiento con el conjunto de prueba.

Consideraciones:

1. La optimización se desarrolló mediante un barrido matricial de los hiperparámetros, lo que resultó en un algoritmo que debió realizar todas las combinaciones posibles, por lo que resultó lento.

2. Se utilizó la versión de una única capa oculta, para acortar el tiempo de ejecución.

3. La elección de la función de activación se definió como un hiperparámetro mas, siendo las funciones ReLu y Sigmoide las opciones.
4. Se presentan los demás hipermarámetros barridos, junto a sus respectivos rangos:
   - cantidad de neuronas de la capa oculta : [50, 100, 150, 200]
   - learning_rates = [0.05, 0.1, 0.15, 0.2, 0.25, 0.3]
   - epochs = [1000, 5000, 10000]
5. Además se consideró un conjunto de datos de entrada (única entrada) de 150 elementos
6. Para el ajuste de los hiperparámetros se utilizó la técnica K-Fold Cross Validation.

K-Fold Cross Validation: técnica que implica dividir el conjunto de datos en K-partes iguales. Luego, se entrena el modelo K-veces, cada vez utilizando una parte diferente como conjunto de validación y las otras K-1 partes como conjunto de entrenamiento. Por ejemplo, si se utiliza K=5, se divide el conjunto de datos en 5 partes iguales. Luego, se entrena el modelo 5 veces, cada vez utilizando una parte diferente como conjunto de validación y las otras 4 partes como conjunto de entrenamiento. El rendimiento del modelo se evalúa en cada una de las 5 iteraciones utilizando el conjunto de validación correspondiente.

Finalmente, se calcula el rendimiento promedio del modelo sobre los 5 conjuntos de validación. La ventaja de K-Fold Cross Validation es que permite evaluar el rendimiento del modelo de manera más robusta y confiable que utilizando un solo conjunto de validación.

6.1. Implementación

Las siguientes funciones fueron las que llevaron a cabo el proceso de barrido.
Función variar_hiperparametros:

- La función variar_hiperparametros

  Es responsable de explorar diferentes combinaciones de hiperparámetros con el fin de encontrar los óptimos para el modelo de regresión.

  Luego, se realizan bucles anidados para iterar sobre todas las combinaciones posibles de hiperparámetros. Para cada combinación, se realizan 10 iteraciones de validación cruzada (K-Fold Cross Validation) utilizando la clase KFold de scikit-learn. En cada iteración, se divide el conjunto de datos en conjuntos de entrenamiento y validación, y se entrena un modelo de regresión utilizando los hiperparámetros específicos. Se calcula el MSE (Error cuadrático medio) en el conjunto de validación y se registra en una lista mse_scores. Al final de las 10 iteraciones, se calcula el MSE promedio y se agrega a la lista de puntajes mse_scores.

  Los resultados se almacenan en un diccionario mse_resultados, donde las claves son tuplas que contienen los valores de los hiperparámetros y los valores son una lista que contiene los puntajes de MSE y la desviación estándar correspondiente.

- Función obtener_hiperparametros_optimos: Toma el diccionario de resultados del barrido de hiperparámetros y devuelve los hiperparámetros óptimos encontrados. Primero, se ordenan los resultados del barrido utilizando la función sorted, pasando como criterio de ordenamiento el MSE promedio y luego la desviación estándar. Luego, se seleccionan los hiperparámetros óptimos tomando los valores de la primera entrada del diccionario ordenado, y se retornan.

- Función plot_box_plots: Se encarga de graficar los resultados del barrido. Toma como entrada el diccionario de resultados y los hiperparámetros óptimos encontrados.

La función filtra los resultados del diccionario según los parámetros óptimos. Utiliza bucles para generar listas específicas de valores de hiperparámetros y puntajes de MSE correspondientes.

Luego muestra un gráfico de caja que representa la variación del MSE con respecto a un hiperparámetro específico.

En cada gráfico de caja, la línea verde representa el valor óptimo encontrado, y los puntos azules corresponden a los puntajes de MSE para cada combinación de hiperparámetros. Esta visualización permite observar la distribución de los puntajes de MSE y detectar posibles tendencias o diferencias significativas entre las combinaciones de los hiperparámetros.

Finalmente se encuentra la sección de prueba, donde testeamos lo hiperparámetros óptimos obtenidos, generando un conjunto de prueba y ejecutando el entrenamiento.

6.2. Resultados

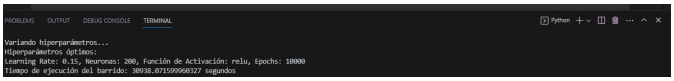


Figura 39: Resultados en terminal

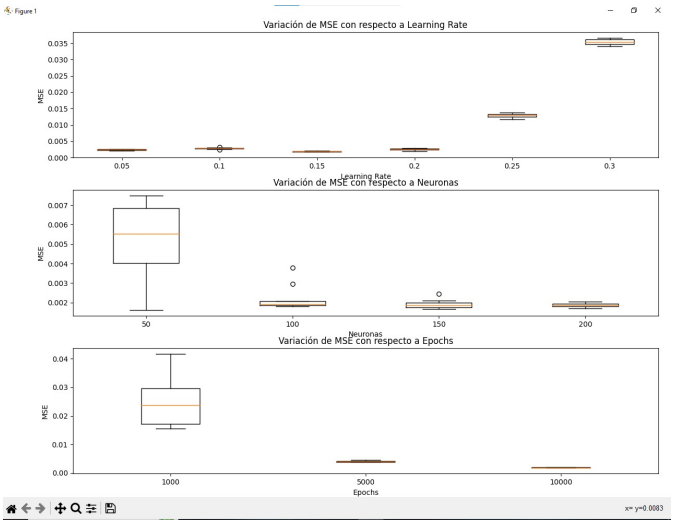


Figura 40: Variaciones de MSE

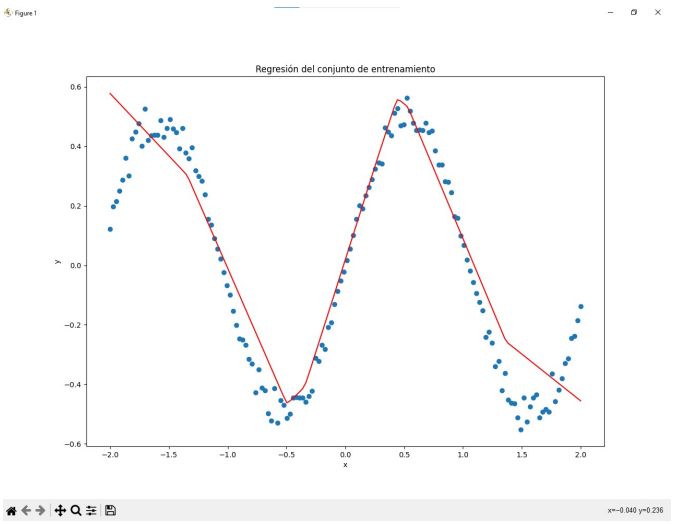


Figura 41: Regresión del conjunto de entrenamiento

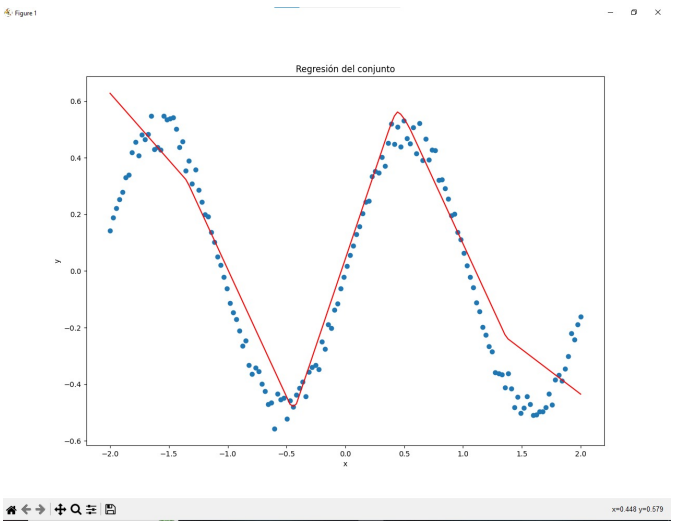


Figura 42: Conjunto de prueba

7. Conclusión

En base al trabajo realizado, se puede concluir que se logró cumplir con los objetivos propuestos en el trabajo práctico. Se implementaron mejoras significativas al código original de la red neuronal, permitiendo medir la precisión de clasificación y utilizar conjuntos de test y validación independientes. La implementación de la parada temprana brindó una herramienta efectiva para evitar el sobreajuste y mejorar la eficiencia del entrenamiento. Además, la experimentación con distintos parámetros de configuración permitió comprender su impacto en el desempeño de la red neuronal. En general, se obtuvieron resultados satisfactorios y se evidenció el potencial de las redes neuronales fully connected para resolver tanto problemas de clasificación como de

regresión. Las gráficas generadas proporcionan una visualización adicional del rendimiento del modelo.

El trabajo práctico proporcionó una sólida comprensión de los conceptos fundamentales de las redes neuronales fully connected y su aplicación en problemas de clasificación y regresión. Además, brindó la oportunidad de desarrollar habilidades en la manipulación y experimentación con los parámetros de configuración para optimizar el rendimiento de la red neuronal.