Generación de una Red Neuronal Densa entrenada por un Algoritmo Evolutivo para el Control de un Péndulo Invertido

1st Gabriel Orlando Gonzalez Rodriguez, gabrielgr01@estudiantec.cr
2nd David Areus Fuentes Cerdas, davidareus@estudiantec.cr
Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Mecatrónica
MT8008 Inteligencia Artificial

Resumen

Este proyecto presenta la generación de una red neuronal MLP destinada al control de sistema no lineal: un péndulo invertido rotacional. La red recibe como entrada el ángulo y la velocidad angular del péndulo para predecir el torque necesario para mantener el sistema estable. Por lo general, el entrenamiento de una red neuronal MLP se realiza mediante el descenso de gradiente. Sin embargo, en este proyecto la red neuronal se optimiza por medio de un algoritmo evolutivo que asigna los pesos correspondientes de la red. Se diseña dicho algoritmo y se construye y simula la red neuronal para finalmente presentar los datos del control exitoso para el sistema planteado. Se obtuvo una red neuronal MLP de una sola capa oculta, 5 neuronas, capaz de predecir el torque necesario para mantener el sistema estable frente a perturbaciones, con un error de 2.62 Nm.

Palabras Clave

Inteligencia Artificial, Red Neuronal, Algoritmo Evolutivo, Control, Péndulo Invertido.

I. Introducción

Según [1] el péndulo invertido rotacional es un sistema mecánico no lineal subactuado que presenta una dinámica altamente no lineal, la cual dificulta el uso de modelos de control clásico. Es en este contexto en el que el aprendizaje reforzado representa una oportunidad alternativa atractiva para el control de sistemas no lineales. Los métodos clásicos de control requieren el uso de la linealización para aproximar el comportamiento de estos sistemas. La principal desventaja de la linealización yace en que se debe asumir un punto de operación conocido, respecto al cual, entre más se aleja el sistema de este punto de operación incrementa el error.

El principal objetivo de este proyecto es generara el control de un péndulo invertido rotacional, por medio de una red neuronal densa MLP. La red será entrenada por medio de un algoritmo evolutivo, en lugar del descenso de gradiente. Esto permitirá mayor robustez frente a mínimos locales y se evitará el cálculo de derivadas. El algoritmo evolutivo optimizará los pesos usando un conjunto de datos ("dataset") como referencia. El proyecto tiene la limitante de que tanto la obtención del "dataset" como la evaluación de la red se hacen por medio de una simulación, no en implementación física. Sin embargo, se toman en cuenta el efecto de perturbaciones, modeladas como impulsos en el tiempo.

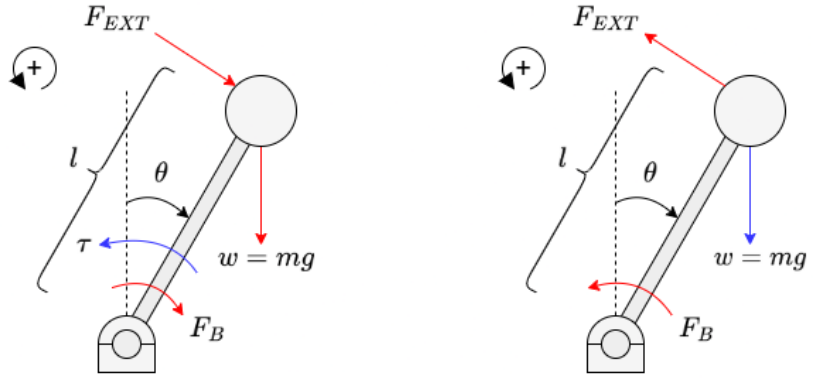
El presente proyecto se compone de 4 principales etapas. La generación de un "dataset" con datos de ángulo, velocidad angular y torque. El diseño de una red MLP, como modelo para la predicción del torque a partir de los datos de entrada de ángulo y velocidad angular. La optimización de los pesos y sesgos de la red neuronal a partir del algoritmo evolutivo. Y la evaluación de la red neuronal obtenida por medio de una simulación.

II. Análisis del Problema

El problema que se busca resolver es la generación de una red neuronal con pesos asignados por medio de un algoritmo evolutivo y que permita realizar el control de un péndulo invertido. Dicha red recibe como entrada un ángulo y una velocidad, y retorna el torque necesario para el control. Además, es requerida la generación de un conjunto de datos (ángulo, velocidad y torque) que representen el comportamiento del sistema para que sea utilizado en el entrenamiento de la red neuronal. Para todo el documento se respetan las siguientes unidades: angulo ($radianes$), velocidad ($\frac{m}{s}$), aceleración ($\frac{m}{s^2}$), torque ($N \cdot m$).

II-A. Modelo del Sistema

Se plantea el diagrama de cuerpo libre del sistema presente en la figura 1. Se divide en dos casos, uno para cuanto existe un torque de control aplicado al sistema (figura 1) y otro para cuando este torque es igual a cero, o en otras palabras, inexistente (figura 2). Esto se hace para poder modelar el sistema cuando su movimiento se debe al torque de control aplicado pero también para cuando se debe simplemente a su propio peso (y sin control alguno). Este último caso permite validar el comportamiento del sistema de una forma más intuitiva durante la implementación.



(a) Sistema Con Torque de Control (b) Sistema Sin Torque de Control

Figura 1: Modelo Dinámico del Péndulo Invertido

A partir de los diagramas de la figura 1 se obtienen las ecuaciones dinámicas del sistema, aplicando la Segunda Ley de Newton para movimiento angular, el cual indica que la "sumatoria de momentos es igual al momento de inercia multiplicado por la aceleración" [2]. De esta forma se obtiene la ecuación 1 que hace referencia a la figura 1a y que modela el comportamiento del péndulo con el efecto de un torque para el control.

$$\sum M_c = J\ddot{\theta}$$

$$\tau - Wl\sin(\theta) - B\dot{\theta} - F_{EXT} \cdot l = J\ddot{\theta}$$

$$\tau - mgl \cdot \sin(\theta) - B\dot{\theta} - F_{EXT} \cdot l = ml^2 \cdot \ddot{\theta}$$

$$\ddot{\theta} = (\frac{1}{ml^2}) \cdot \tau - (\frac{g}{l}) \cdot \sin\theta - (\frac{B}{ml^2}) \cdot \dot{\theta} - (\frac{1}{ml}) \cdot F_{EXT} \tag{1}$$

Para el modelo del péndulo sin el efecto del torque de control, se obtiene la ecuación 2 que hace referencia a la figura 1b.

$$\ddot{\theta} = -(\frac{g}{l}) \cdot \sin\theta + (\frac{B}{ml^2}) \cdot \dot{\theta} + (\frac{1}{ml}) \cdot F_{EXT} \tag{2}$$

II-B. Elección de la Estrategia para la Generación del Dataset

Para la generación del "dataset" se cuenta con dos opciones:

1. Utilizar un controlador PID y obtener los datos a partir de una simulación de su comportamiento.
2. Generar los datos a partir de acciones aleatorias con filtrado de estados estables y combinarlo con física inversa para mejorar la calidad de los resultados.

Se toma la decisión de utilizar el controlador PID para la generación de los datos por las siguientes razones:

- Presenta mayor precisión en los datos generados frente a las acciones aleatorias, ya que actúa alrededor del estado de estabilidad y no depende del correcto uso de procesados/filtrados posteriores para ello.
- Una vez calibrado el controlador, se obtienen los datos deseados directamente y no se requiere de procesos adicionales.
- Si bien el control PID requiere de un nivel más técnico para su implementación y calibración, su diseño es bastante estándar y existe suficiente documentación.
- El diseño de un simulador para el comportamiento del controlador PID puede ser de utilidad a la hora de querer simular los datos de la red neuronal, ya que ambos se podrían tratar como un módulo intercambiable con las mismas entradas y salidas.

Si bien se toma esta decisión como la más apropiada, existen también algunas desventajas:

- Al actuar alrededor del estado de estabilidad del sistema, los datos no representan lo suficientemente bien casos de inestabilidad, lo que puede generar que la red neuronal no se entrene bien para eventos de la vida real en donde exista este tipo de ruido en el sistema.
- Se requiere de una calibración inicial de las ganancias del controlador.

III. Diseño e Implementación de la Solución

III-A. Diseño de la Solución

Se busca obtener una **red neuronal** como la que se observa en la figura 2, la cual, al ingresarle un ángulo y velocidad específica, pueda proveer el torque necesario para el control de un péndulo invertido. Sin embargo, se tiene como requisito el encontrar los pesos de dicha red no por medio del descenso de gradiente, sino con el uso de un algoritmo evolutivo.

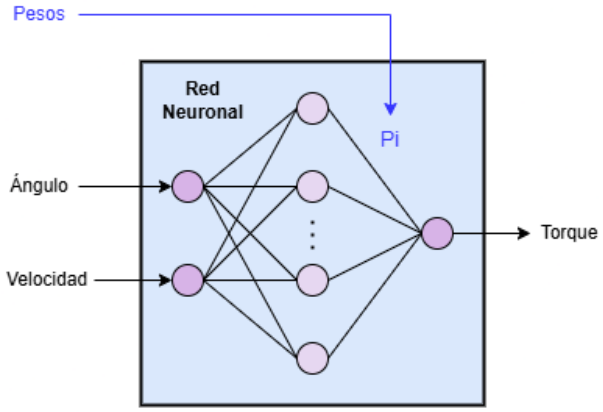


Figura 2: Diagrama de la Red Neuronal Requerida.

Para lograr obtener los pesos requeridos, se propone el diseño presente en la figura 3 y que contempla un **algoritmo evolutivo** al que se le define un cromosoma o individuo como una lista con los pesos requeridos por la red neuronal. Además, su **función de evaluación** compara el error entre los datos de torque obtenidos del "dataset" y los obtenidos de la predicción de la red. Este error es la calidad asignada para el conjunto de pesos de cada individuo.

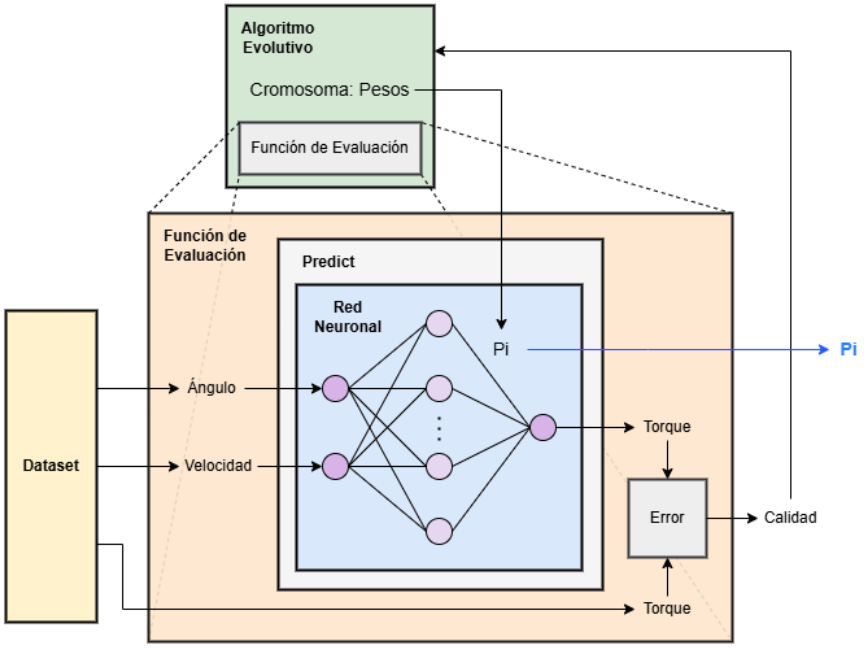


Figura 3: Diseño de la Red Neuronal Entrada por el Algoritmo Evolutivo.

Es requerido que el "dataset" mencionado anteriormente sea generado de forma independiente, para garantizar la validez de los datos. Para esto, se propone el diseño presente en la figura 4. Este se compone de varias partes: Primero, se obtiene el **modelo dinámico** del sistema péndulo invertido y un controlador **PID**. Posterior a esto, se genera el **simulador**, el cual retorna un diccionario con los datos de la simulación del sistema controlado en el tiempo. Para este punto ya el PID debe de estar calibrado con las ganancias k_P, k_I y k_D correctas.

Por último, se crea un **generador del dataset**, que obtiene "n" simulaciones del sistema controlado, con condiciones iniciales distintas y generadas aleatoriamente entre un rango previamente establecido. Se incorporan también perturbaciones a las simulaciones generadas por el generador del dataset. Estas son aleatorias tanto en su generación, como en el tiempo de simulación en el que aplican y en su magnitud. Se limita su persencia a un máximo del 25 % de las simulaciones para mantener siempre una mayoría de datos en las condiciones de estabilidad y que el algoritmo evolutivo logre asignar los pesos correctos para el objetivo que se busca, que es el control del péndulo.

El **"dataset"** generado consiste entonces de:

- Guarda: Conjunto de gráficas de cada simulación.
- Guarda: Archivo .csv con los datos de las "n" simulaciones.
- Guarda: Dos archivos .csv de configuración con los máximos y mínimos del dataset, utilizados para normalizar los datos y para normalizar posteriores entradas a la red neuronal.
- Retorna: Ruta de los datos guardados.
- Retorna: "DataFrame" con los datos de las "n" simulaciones.

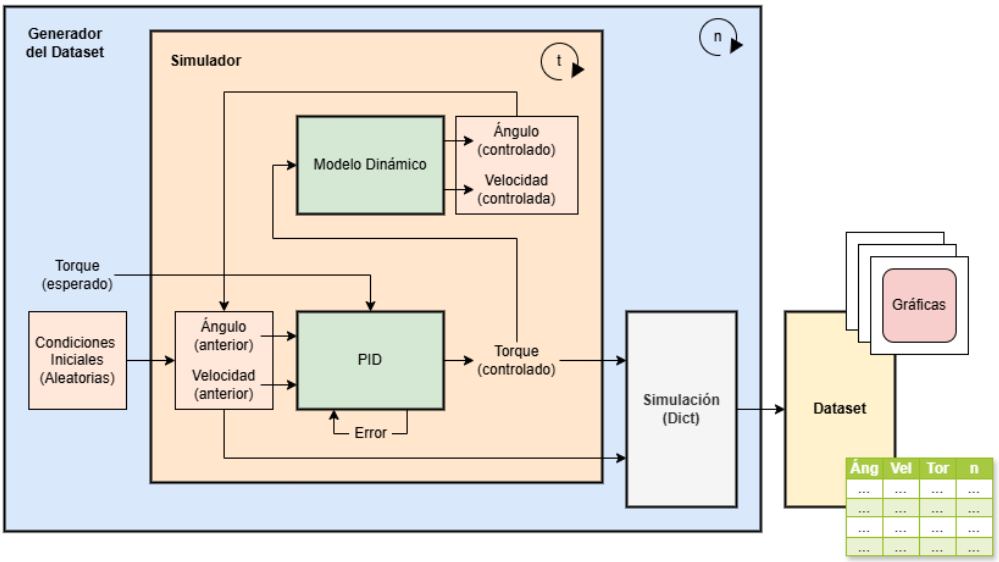


Figura 4: Diseño de la Simulación y Generación del Dataset

III-B. Estructura del Código

Se busca desarrollar un programa en el lenguaje de programación Python que nos permita obtener la red neuronal para resolver nuestro problema. Para procurar una implementación ordenada y modular, se organiza el código de la siguiente forma, separando en módulos las funciones según su objetivo:

- main.py : Posee la ejecución general del programa. (Código en el ANEXO H)
- modules/config.py : Posee las definiciones de variables globales y parámetros a utilizar en el programa. (Código en el ANEXO I)
- modules/simulation.py : Funciones relacionadas a la generación, resolución y simulación del modelo del sistema, así como la generación del "dataset". (Código en el ANEXO J)
- modules/network.py : Funciones relacionadas a la generación de la red neuronal. (Código en el ANEXO K)
- modules/evolution.py : Funciones relacionadas a la generación del algoritmo evolutivo. (Código en el ANEXO L)
- modules/utils.py : Funciones miscelaneas útiles para el programa. (Código en el ANEXO M)
- modules/__init__.py : Archivo de inicialización necesario para poder acceder a los modulos desde el main.py.

Además, en el anexo ANEXO G se puede observar una representación gráfica de la estructura de archivos utilizada y se incluyen además los archivos de imágenes generadas por el programa una vez ejecutado.

III-C. Implementación de la Red Neuronal

Ya que las entradas y salidas de la red están definidas se obtuvo una red **dos neuronas en la capa de entrada y una neurona en la capa de salida**. Además, para simplificar la implementación se estableció un **número fijo de capas ocultas de 1**. Además, se determinó que los únicos hiperparámetros que necesitan ser estudiados o justificados son los que se relacionan con la estructura intrínseca de la red y no con el entrenamiento de esta, ya este es reemplazado por el algoritmo evolutivo. Sin embargo, se analiza el efecto de variar el número de neuronas de la capa oculta sobre la validación de la solución de la red.

III-C1. Selección de Hiperparámetros de la Red Neuronal: Se seleccionó una **función de activación tanh** para la capa oculta y la capa de entrada por la compatibilidad de su rango con datos normalizados. Para la capa de salida se seleccionó una **función de activación lineal** para poder predecir cualquier valor de torque, ya sea positivo o negativo, y evitar distorsionar la salida debido a un rango limitado, como pasa con tanh.

Para determinar el número mínimo de neuronas de la capa oculta que permite el correcto funcionamiento de la red, se tomó un valor inicial de 3 neuronas y se simuló la capacidad de la red para controlar el péndulo invertido. Posteriormente, se incrementó el número de neuronas hasta alcanzar una red que es capaz de controlar el péndulo. El número mínimo de neuronas requerido para lograr el control fue de 5.

III-D. Implementación del Algoritmo Evolutivo

La **codificación del cromosoma** del algoritmo evolutivo consiste en los pesos y los sesgos de la red neuronal. El número total de genes, para una red de una sola capa oculta, se pude calcular con la ecuación 3, donde n_i es el número de neuronas de la capa de entrada, n_h es el número de neuronas de la capa oculta y n_o es el número de neuronas de la capa de salida. El orden en el que están codificados los genes respecto de los pesos y sesgos de la red neuronal es: pesos $n_i \rightarrow n_h$, sesgos n_h, pesos $n_h \rightarrow n_o$ y sesgos n_o.

$$n_c = n_i n_h + n_h + n_h n_o + n_o \tag{3}$$

Se definió que el tipo de problema a tratar con el algoritmo evolutivo es **monobjetivo**. Ya que se cuenta con un único objetivo de optimización, el error entre el torque predicho por la red neuronal y el torque del dataset. La **función de evaluación** seleccionada que permite representar el error entre la red neuronal y el dataset es el error cuadrático medio, el cual cuantifica la diferencia entre las magnitudes de torque y asegura que los individuos siempre tengan calidad positiva. La población inicial se genera con genes en el rango de -1 a 1, que es donde se encuentran las magnitudes típicas de pesos para redes neuronales. Sin embargo, no se restringe el **espacio de alelos** durante la mutación para permitir que el algoritmo evolutivo explore soluciones no convencionales.

III-D1. Selección de Hiperparámetros y Operadores del Algoritmo Evolutivo: Los **operadores seleccionados** se mencionan a continuación. Como operador de selección se tomó el operador de **torneo**, porque permite trabajar con un objetivo de minimización y proporciona control adicional sobre la presión selectiva. Se seleccionó el **cruce por mezcla** $BLX-\alpha$ porque es ideal para los genes del algoritmo evolutivo, los cuales toman valores reales. Se selecciona un operador de **mutación Gaussiana** porque es ideal para mutar valores continuos y brinda control sobre la exploración por medio del parámetro σ.

Se selecciona una **población inicial de 100 individuos** para evitar la convergencia prematura pero tampoco exceder el costo computacional. Se usa un **número de padres de 25** para incrementar ligeramente la presión selectiva y ahorrar recursos computacionales. Ya que el número de hijos debe ser como mínimo igual al número de padres se toma un **número de hijos de 25**. Se seleccionó una **probabilidad de cruce de 0.7** para reforzar la probabilidad de combinar genes exitosos. Se selecciona una **probabilidad de mutación de 0.2**, un valor típico que permite la exploración de nuevas soluciones y evita el estancamiento en óptimos locales. Se selecciona un **coeficiente de mezcla del cruce** $\alpha = 0{,}5$, un valor moderado que permite la variación de los genes sin crear pesos muy grandes que saturen las funciones de activación de las neuronas. Se seleccionó una **media de mutación de 0**, para no introducir sesgo en la exploración de las soluciones y que los cambios introducidos sean simétricos. Se seleccionó una **desviación estándar de la mutación** $\sigma = 1\,\%$ del rango del espacio de alelos en el que se crea la población inicial para una búsqueda más precisa de la solución.

IV. Resultados

La generación de resultados se dividió en varias etapas para garantizar el correcto funcionamiento de cada módulo del sistema. Primero, se graficó el comportamiento del controlador PID con sus ganancias ya calibradas, a como se puede observar en el ANEXO A.

Una vez comprobado el correcto control por parte del PID, se procede con la generación del "dataset". Se obtienen las gráficas para cada simulación presente. Un ejemplo de una de las simulaciones obtenidas se puede observar en la figura 11, en donde se observa el control del péndulo con el controlador PID y la presencia de una perturbación aleatoria. En el ANEXO B se puede observar la gráfica de las demás variables (torque, velocidad, aceleración) para la misma simulación. Además, en el ANEXO C se observa también otra de las simulaciones pero sin presencia de perturbaciones.

Para los resultados del algoritmo evolutivo, se obtiene la curva de error presente en la figura 5. La cual muestra el error cuadrático medio durante todas las generaciones y de la cual se obtienen los siguientes datos para la última generación, para el individuo seleccionado (sus pesos y sesgos) y para el error de este mejor individuo.

```
Last generation:
    gen  nevals       avg       std       min       max
30   30      22  8.274935  0.289476  7.981443  9.490844
```

```
Best Individual:  [0.2695260670788988, 0.15992585699787942, 0.16689584730954232, -0.5521598716677538,
    0.07483057085798772, 0.3578966845363399, -0.6507886173703904, 0.029083454524304553,
    -0.24421716446990677, 0.9748134673653231, 0.3645767014724366, -0.8397300160474149, 0.172394687933997,
    -0.12637757727364812, -0.423829490133893, -0.13577323924558904, -0.9909854724625808,
    0.4586532250948905, 0.2171977734045767, -0.49283932768933425, -0.1639220131875596]

Fitness:  6.873251522419229
```

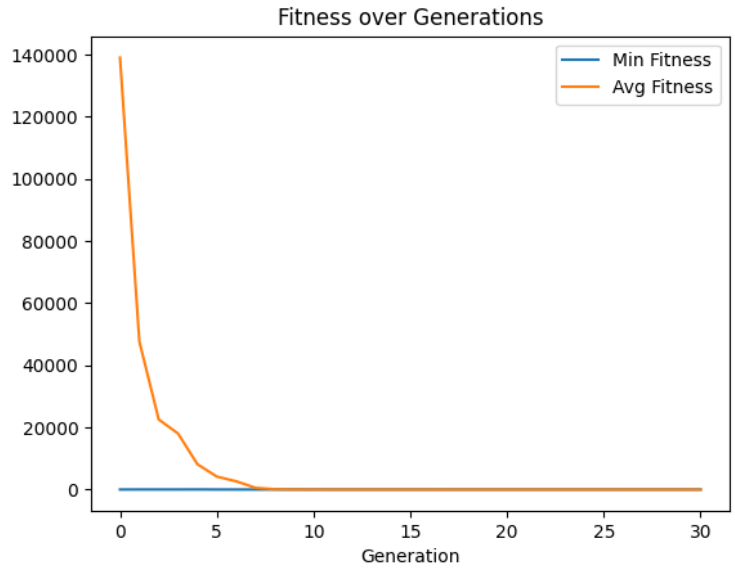


Figura 5: Valores de Calidad para Cada Generación del Algoritmo Evolutivo.

Como validación final, se realiza una simulación de la red neuronal ya con los pesos obtenidos del algoritmo evolutivo, en donde se le proporcionan valores aleatorios de entrada y para las perturbaciones bajo las mismas condiciones de la generación del "dataset". Un ejemplo de una de las simulaciones obtenidas se puede observar en la figura 6, en donde se observa el control del péndulo con la red neuronal y la presencia de una perturbación aleatoria. En el ANEXO E se puede observar la gráfica de las demás variables (torque, velocidad, aceleración) para la misma simulación. Además, en el ANEXO F se observa también otra de las simulaciones pero sin presencia de perturbaciones.

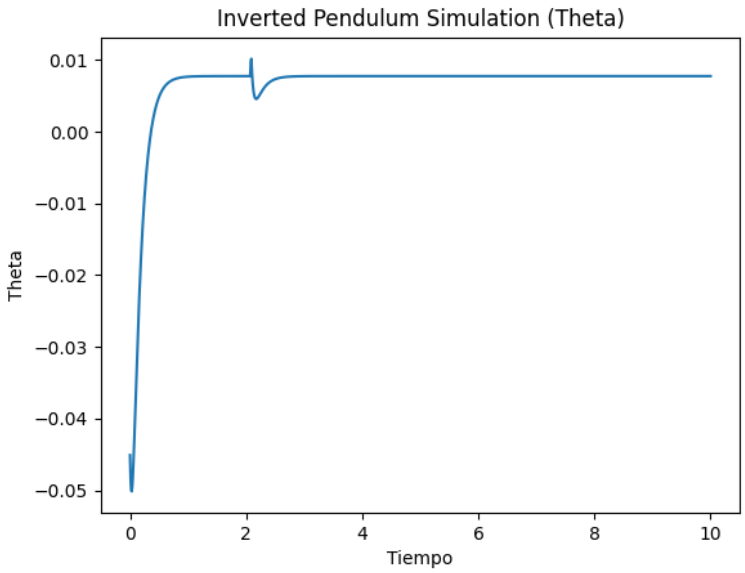


Figura 6: Ejemplo de Una Simulación del Control de Red Neuronal para el Angulo del Sistema.

V. Conclusiones

- Se lograron obtener los pesos y sesgos adecuados, a partir de un algoritmo evolutivo, para una red neuronal capaz de controlar un péndulo invertido.
- La calidad del individuo seleccionado fue de $6{,}87$, que corresponde al error cuadrático medio, por lo que representa un torque de $2{,}62\ N \cdot m$.
- Se obtuvo que el modelo de la red neuronal necesita un mínimo de 5 neuronas para ser capaz de llevar a cabo del control del péndulo invertido.
- La red neuronal obtenida realiza bien su control frente a perturbaciones de entre $-100\ N$ y $100\ N$.
- Se demuestra que el uso combinado de redes neuronales y computación evolutiva es una alternativa viable para el control de sistemas dinámicos con un comportamiento no lineal marcado.

Referencias

[1] Zied Ben Hazem, "Study of Q-learning and deep Q-network learning control for a rotary inverted pendulum system," Deleted Journal, vol. 6, no. 2, Feb. 2024, doi: https://doi.org/10.1007/s42452-024-05690-y.

[2] Openstax, s.f. "10.7 Segunda ley de Newton para la rotación" Recuperado de: https://openstax.org/books/f%C3%ADsica-universitaria-volumen-1/pages/10-7-segunda-ley-de-newton-para-la-rotacion (accedido el 22 de Junio, 2025).

ANEXOS

ANEXO A. SIMULACIÓN DEL CONTROLADOR PID PARA LA GENERACIÓN DE DATOS

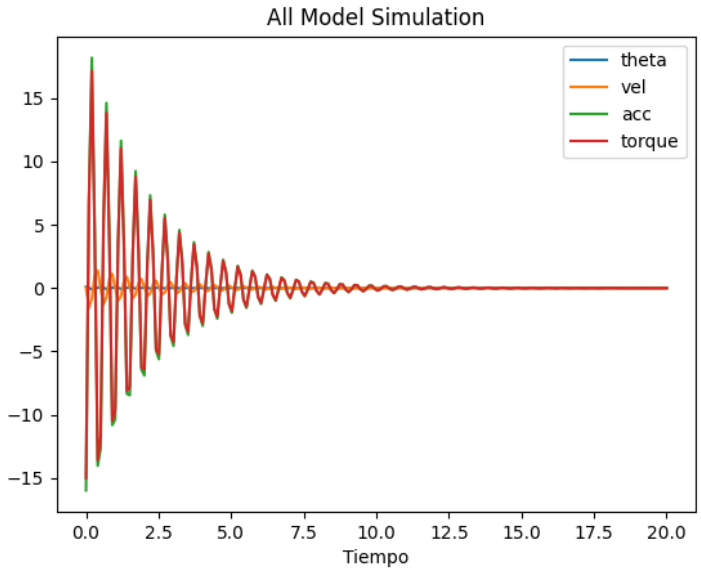


Figura 7: Simulación Completa de las Variables del Modelo Controladas por el PID. Numero de Muestras: 10 Veces el Tiempo de Simulación.

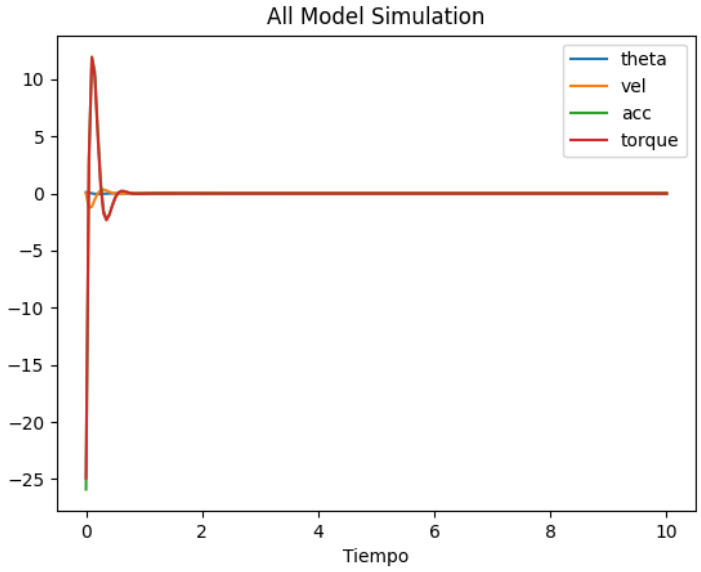


Figura 8: Simulación Completa de las Variables del Modelo Controladas por el PID. Numero de Muestras: 20 Veces el Tiempo de Simulación.

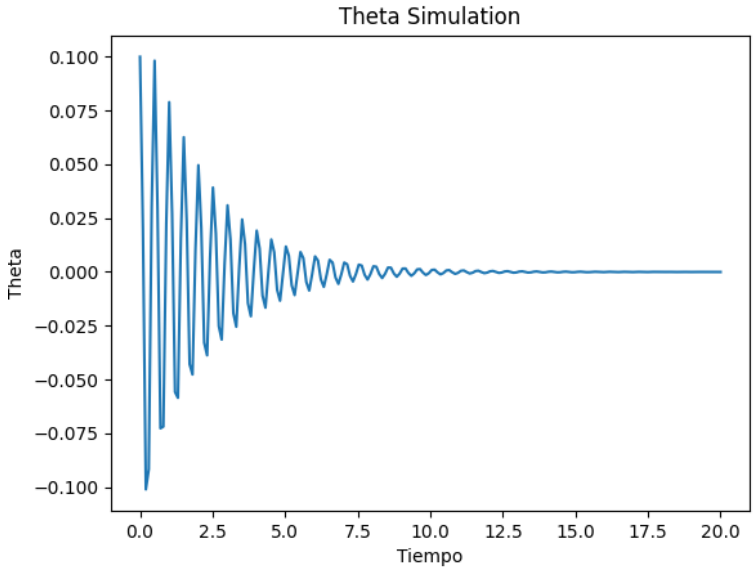


Figura 9: Simulación del Angulo del Modelo Controlado por el PID.
Numero de Muestras: 10 Veces el Tiempo de Simulación.

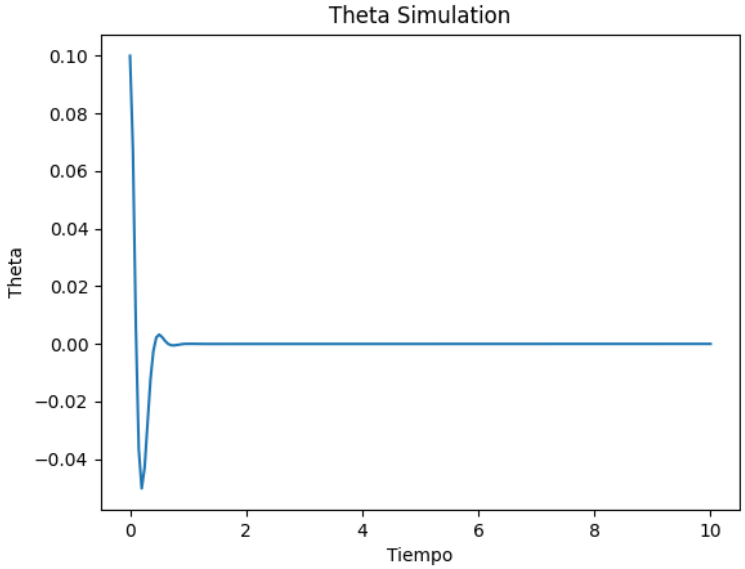


Figura 10: Simulación del Angulo del Modelo Controlado por el PID.
Numero de Muestras: 20 Veces el Tiempo de Simulación.

ANEXO B. EJEMPLO DE UNA SIMULACIÓN DEL DATASET CON PERTURBACIONES.

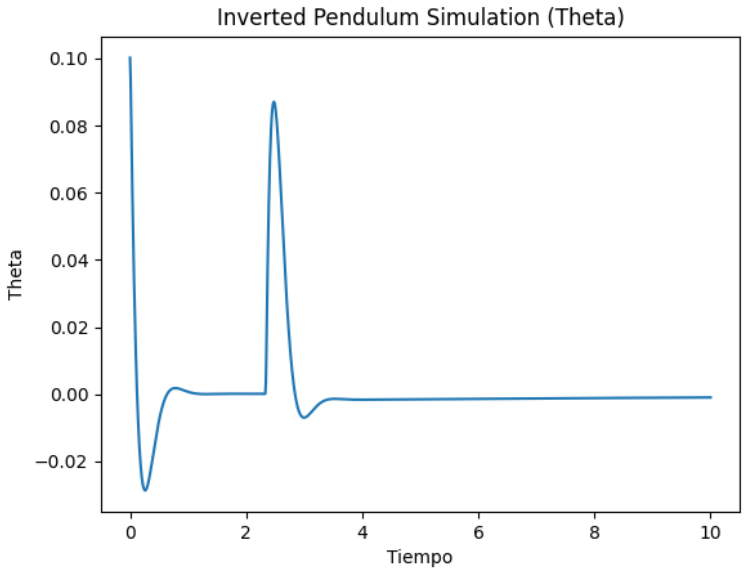


Figura 11: Ejemplo de Una Simulación del Dataset para el Angulo del Sistema.

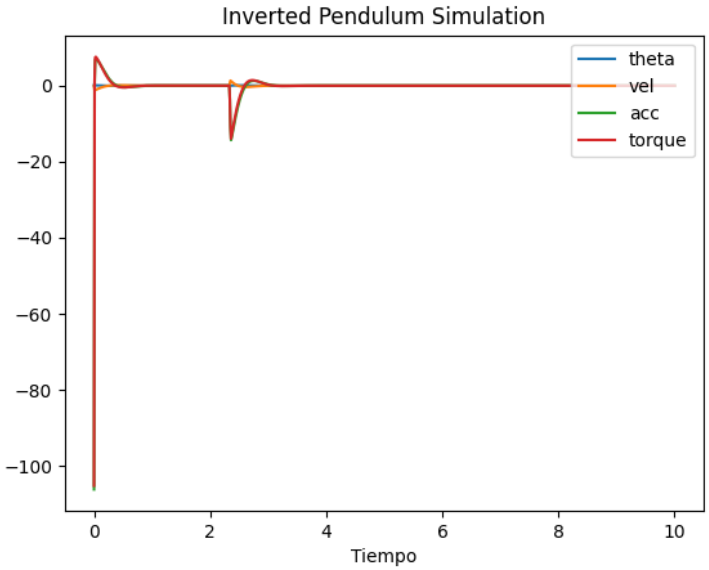


Figura 12: Ejemplo de una Simulación del Dataset con Todas las Variables del Sistema.

ANEXO C. EJEMPLO DE UNA SIMULACIÓN DEL DATASET SIN PERTURBACIONES.

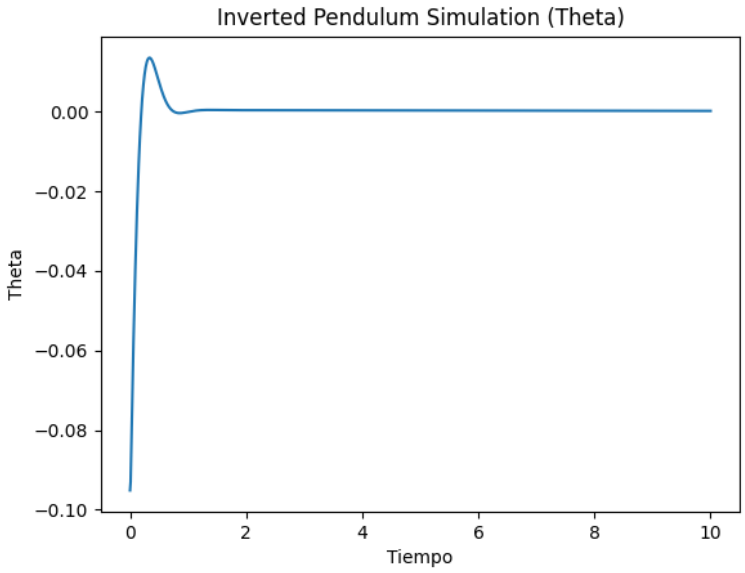


Figura 13: Ejemplo de Una Simulación del Dataset para el Angulo del Sistema. Sin Perturbaciones.

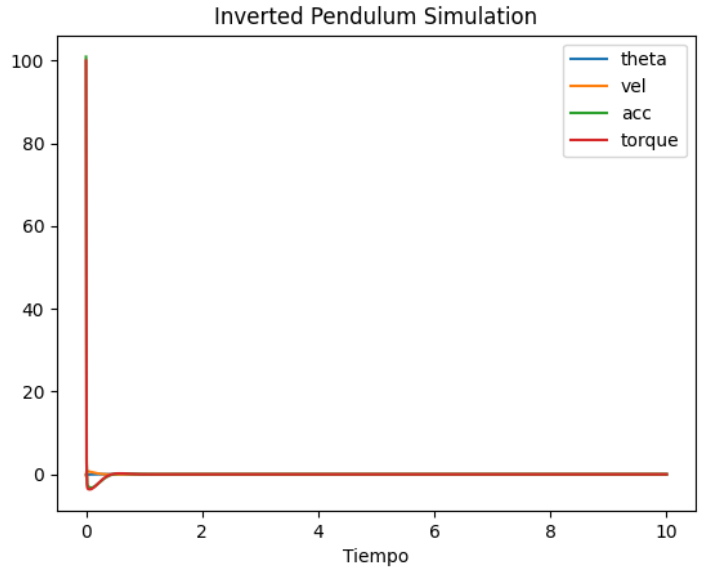


Figura 14: Ejemplo de una Simulación del Dataset con Todas las Variables del Sistema. Sin Perturbaciones.

ANEXO D. Estudio De Hiper-parámetros de la Red Neuronal: Número de Neuronas de la Capa Oculta

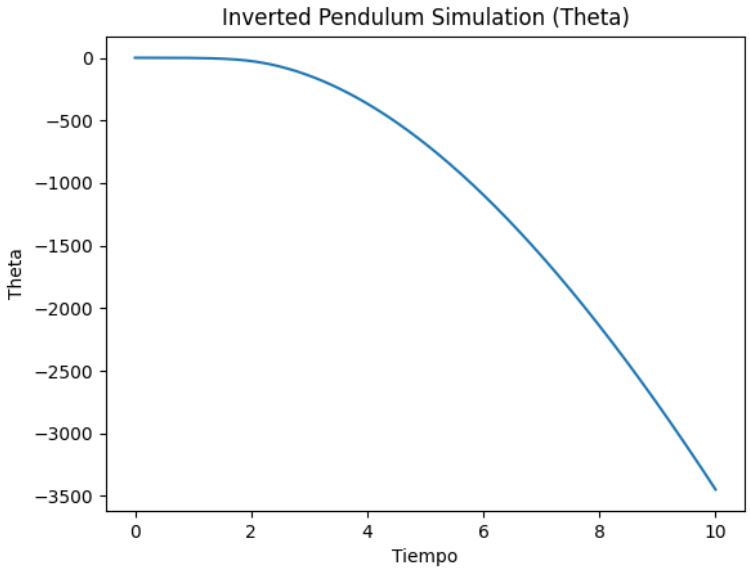


Figura 15: Simulación del Control del Péndulo con una Red de 3 Neuronas.

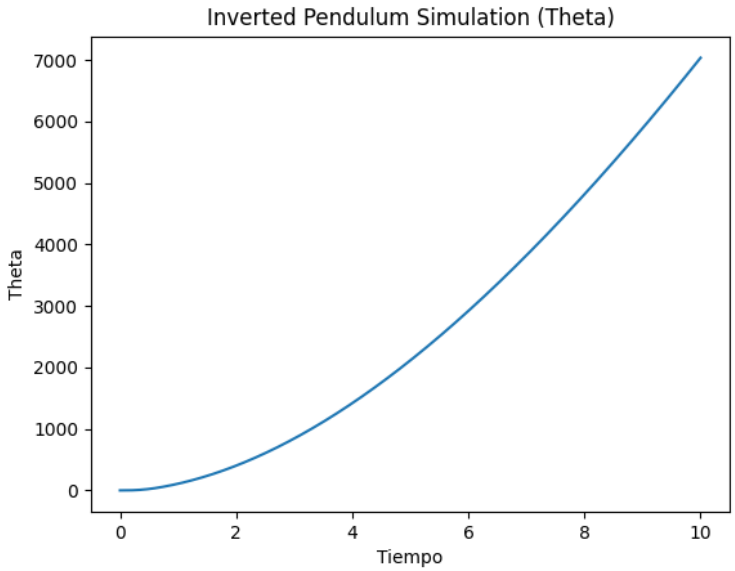


Figura 16: Simulación del Control del Péndulo con una Red de 4 Neuronas.

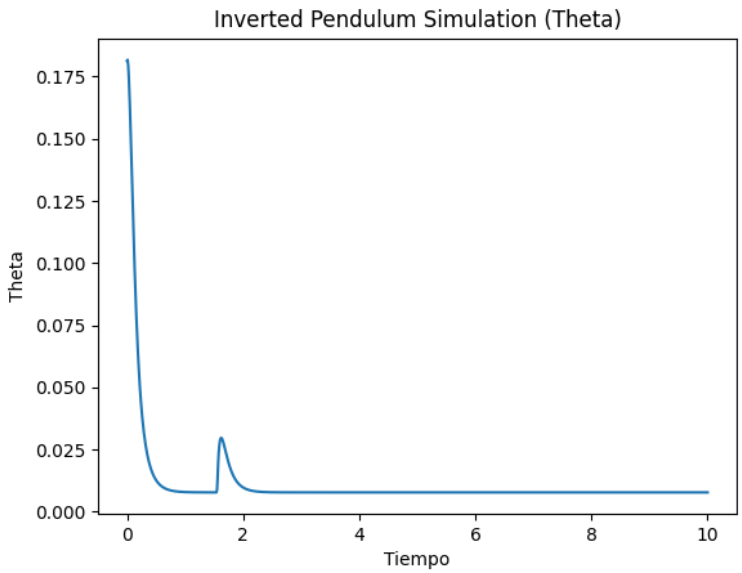


Figura 17: Simulación del Control del Péndulo con una Red de 5 Neuronas (Incluyendo Perturbaciones).

ANEXO E. EJEMPLO DE UNA SIMULACIÓN DE LA VALIDACIÓN DE LA RED NEURONAL.

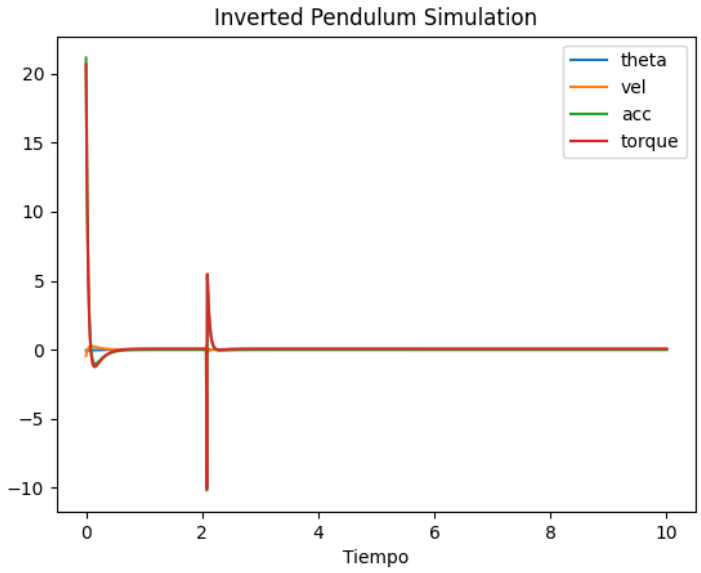


Figura 18: Ejemplo de Una Simulación del Control de Red Neuronal con Todas las Variables del Sistema.

ANEXO F. EJEMPLO DE UNA SIMULACIÓN DE LA VALIDACIÓN DE LA RED NEURONAL. SIN PERTURBACIONES.

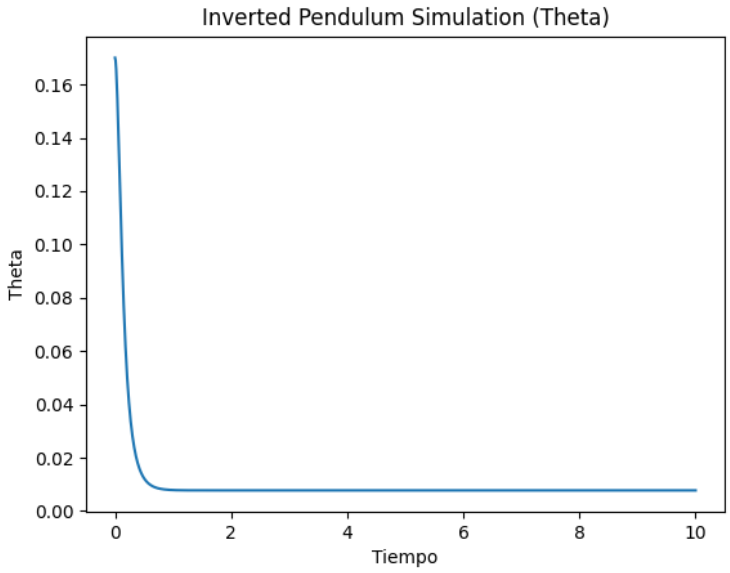


Figura 19: Ejemplo de Una Simulación del Control de Red Neuronal para el Angulo del Sistema. Sin Perturbaciones.

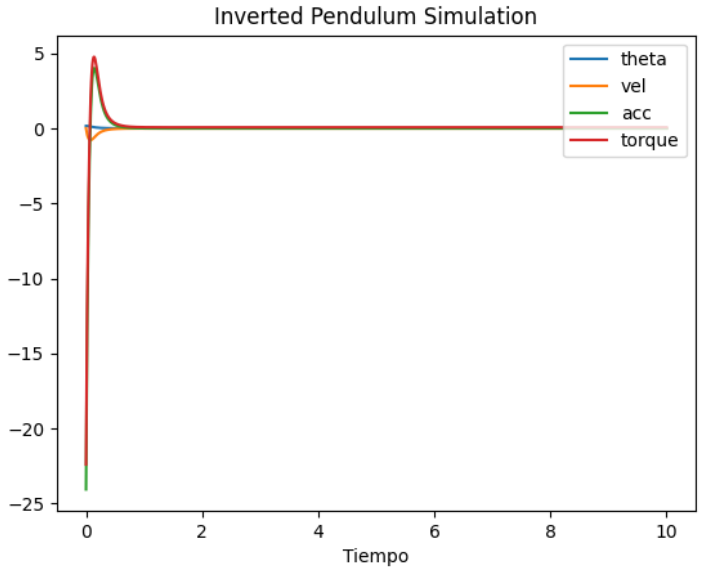


Figura 20: Ejemplo de una Simulación de la Red Neuronal con Todas las Variables del Sistema. Sin Perturbaciones.

ANEXO G. ESTRUCTURA COMPLETA DE LOS ARCHIVOS DEL PROGRAMA

```
Program File Tree:
C:.
|   main.py
|
|
|-- data
|       dataset.csv
|       inv_pen_sim_1.png
|       inv_pen_sim_2.png
|       ...
|
|-- modules
|       config.py
|       evolution.py
|       network.py
|       simulation.py
|       utils.py
|       __init__.py
|
|-- results
        evolution_fitness.png
        inv_pen_sim_1.png
        inv_pen_sim_2.png
        ...
        max_config.csv
        min_config.csv
        model.keras
        network_sim.csv
```

ANEXO H. CÓDIGO MAIN.PY

```
##### IMPORTS #####

# Third party imports

# Built-in imports

# Local imports
from modules.config import *
import modules.utils as util
import modules.simulation as sim
import modules.network as net
import modules.evolution as evol


##### GLOBAL VARIABLES #####


##### FUNCTIONS DEFINITION #####

def train_new_model(dataset_dataframe):
    util.create_directory(RESULTS_DIR_PATH, backup=False, overwrite=False)

    # Creates the neural network model
    net_model = net.build_model(num_input_neurons=NUM_INPUT_NEURONS,
                                num_hidden_neurons=NUM_HIDDEN_NEURONS,
                                num_output_neurons=NUM_OUTPUT_NEURONS,
                                input_act_func=INPUT_ACT_FUNCTION,
                                hidden_act_func=HIDDEN_ACT_FUNCTION,
                                output_act_func=OUTPUT_ACT_FUNCTION)
    # Runs the evolutionary algorithm to get the network's weigths
    best_ind, best_ind_fitness = evol.run_evolutionary_algorithm(net_model, dataset_dataframe)

    net.set_model_weights(net_model, best_ind)

    print("Best Individual: ", best_ind)
    print("Fitness: ", best_ind_fitness[0])

    return net_model


##### MAIN EXECUTION #####

## Generates new dataset of n_sims simulations
#_, dataset_df = sim.generate_dataset(n_sims = 20, data_path=DATA_DIR_PATH, option="PID")

## Loads a dataset from a .csv file
#dataset_df = sim.load_dataset(DATASET_CSV_PATH)

## Trains the new neural network model
#new_model = train_new_model(dataset_df)

## Saves the neural network model
#net.save_model(new_model, MODEL_SAVE_PATH)

## Loads the neural network model
net_model = net.load_model(MODEL_SAVE_PATH)

## Simulates the neural network model
_, network_sim_df = sim.generate_dataset(n_sims = 10, data_path=RESULTS_DIR_PATH, option="NETWORK", model=
    net_model)
```

ANEXO I. CÓDIGO CONFIG.PY

```
##### IMPORTS #####

# Third party imports

# Built-in imports
import os

# Local imports


##### GLOBAL VARIABLES #####

RUN_AREA = os.getcwd() # Directory where the program is run
RESULTS_DIR_NAME = "results"
RESULTS_DIR_PATH = os.path.join(RUN_AREA, RESULTS_DIR_NAME)                  # Results directory path.
DATA_DIR_NAME = "data"
DATA_DIR_PATH = os.path.join(RUN_AREA, DATA_DIR_NAME)                        # Data directory path.

DATASET_CSV_NAME = "dataset.csv"
DATASET_CSV_PATH = os.path.join(DATA_DIR_PATH, DATASET_CSV_NAME)             # Dataset path.

NETWORK_SIM_CSV_NAME = "network_sim.csv"
NETWORK_SIM_CSV_PATH = os.path.join(RESULTS_DIR_PATH, NETWORK_SIM_CSV_NAME) # Neural network simulation
    data path.

MODEL_SAVE_NAME = "model.keras"
MODEL_SAVE_PATH = os.path.join(RESULTS_DIR_PATH, MODEL_SAVE_NAME)            # Neural network model path.
MAX_CONFIG_SAVE_NAME = "max_config.csv"
MAX_CONFIG_SAVE_PATH = os.path.join(RESULTS_DIR_PATH, MAX_CONFIG_SAVE_NAME) # Config path with max values
    for normalization.
MIN_CONFIG_SAVE_NAME = "min_config.csv"
MIN_CONFIG_SAVE_PATH = os.path.join(RESULTS_DIR_PATH, MIN_CONFIG_SAVE_NAME) # Config path with min values
    for normalization.
MODEL_CONFIG_PATH_LIST = [MAX_CONFIG_SAVE_PATH, MIN_CONFIG_SAVE_PATH]       # List with max_config.csv and
    min_condig.csv paths.

SIM_T_STOP = 10 # Maximum simulation time
DT = 0.01       # Intervals length in the simulation
SIM_T_SAMPLES = int((SIM_T_STOP / DT) + 1) # Num of samples in the simulation.
PERTURBANCE_SIM_LIMIT = 5   # Max number of simulations with perturbance.
PERTURBANCE_SIM_COUNT = 0   # Count of simulations with perturbance.

VERBOSE = False     # To enable prints in the function.
DEBUG = False       # To print detailed messages useful for debugging.

PID_KP = 50         # Proportional Gain for the PID.
PID_KI = 5          # Integral Gain for the PID.
PID_KD = 10         # Derivative Gain for the PID.
TARGET_ANGLE = 0    # [rad] Target angle for the pendulum control.
DYNAMIC_INPUT_PARAMS = [1.0,    # mass [kg]
                        1.0,    # length [m]
                        9.81,   # gravity [m/s^2]
                        0.1     # friction coefficient
                        ]


##### HYPERPARAMETERS #####

# Neural Network
NUM_INPUT_NEURONS = 2       # Number of neurons of the input layer
NUM_OUTPUT_NEURONS = 1      # Number of neurons of the output put layer
NUM_HIDDEN_NEURONS = [5,]   # List of neurons of each hidden layer.
                            # Example: [5, 5] represents two hidden
                            # layers of 5 neurons each.
INPUT_ACT_FUNCTION = "tanh"     # Activation function for the neurons in the input layer
HIDDEN_ACT_FUNCTION = "tanh"    # Activation function for the neurons in the hidden layers
OUTPUT_ACT_FUNCTION = "linear"  # Activation function for the neurons in the output layer

# Evolutionary Algorithm
POPULATION_SIZE = 100   # Initial population size.
NUM_GENERATIONS = 30    # Number of generations to evolve.
PARENT_POPU_SIZE = 25   # Number of parents for each generation.
CHILD_POPU_SIZE = 25    # Number of childs for each generation.
MATE_CHANCE = 0.7       # Chance of mating to individuals.
MUTATE_CHANCE = 0.2     # Chance of mutating a new individual.
```

```
MU = 0.0                # Average of the normal distribution used in the mutation operator.
GENE_RANGE = [-1, 1]    # Valid range of values for the genes.
ALPHA = 0.5             # Recombination coeffient for mate operator.
SIGMA = (GENE_RANGE[1] - GENE_RANGE[0]) * 0.01 # Standard deviation for the mutation operator.
```

ANEXO J. CÓDIGO SIMULATION.PY

```
##### IMPORTS #####

# Third party imports
import numpy as np
from scipy.integrate import odeint
import pandas as pd

# Built-in imports
import random

# Local imports
from .config import *
from .utils import *
from .network import *


##### FUNCTIONS DEFINITION #####

def get_inv_pendulum_acceleration(input_params, theta, vel, torque_control, f_ext):
    """
    Function:
        Defines the differential equation for a inverted pendulum system.
        Solves for acceleration.

    Parameters:
        input_params (list): Physical constants for the differential ecuation.
        theta (float): Angle value.
        vel (float): Velocity value.
        torque_control (float): Control torque applied to the system.
        f_ext (float): External force (perturbance).

    Returns:
        acc (float): Acceleration of the system.
    """

    m = input_params[0]
    l = input_params[1]
    g = input_params[2]
    B = input_params[3]

    if torque_control == 0:
        acc = + f_ext*(1/m*l) - np.sin(theta)*(g/l) + vel*(B/m*l**2)
    else:
        acc = torque_control*(1/m*l**2) - f_ext*(1/m*l) - np.sin(theta)*(g/l) - vel*(B/m*l**2)
    return acc


def get_inv_pendulum_torque(input_params, theta, vel, acc, f_ext):
    """
    Function:
        Defines the differential equation for a inverted pendulum system.
        Solves for torque.

    Parameters:
        input_params (list): Physical constants for the differential ecuation.
        theta (float): Angle value.
        vel (float): Velocity value.
        acc (float): Acceleration value.
        f_ext (float): External force (perturbance).

    Returns:
        torque (float): Applied control torque of the system.
    """

    m = input_params[0]
    l = input_params[1]
    g = input_params[2]
    B = input_params[3]
    f_ext = input_params[4]

    torque = m*l**2*acc + m*g*l*np.sin(theta) + B*vel + f_ext*l
    return torque


def get_inv_pendulum_model(S, t, input_params, torque_control, f_ext):
```

```
    """
    Function:
        Defines the differential state equations for a inverted pendulum system.

    Parameters:
        S (list): State vector / Initial conditions [angle 'theta', velocity 'vel'].
        t (float): Time variable (not used explicitly but required by odeint).
        input_params (list): Physical constants for the differential ecuation.
        torque_control (float): System control torque.

    Returns:
        List: Derivatives [dtheta/dt, dv/dt].
    """

    theta, vel = S
    return [vel, get_inv_pendulum_acceleration(input_params, theta, vel, torque_control, f_ext)]


def solve_inv_pendulum_model(input_params, torque_control, initial_state, t_start, t_stop, t_samples, f_ext
    ):
    """
    Function:
        Solves the inverted pendulum system using the given parameters.

    Parameters:
        input_params (list): Physical constants for the differential ecuation.
        torque_control (float): System control torque.
        t_start (float): Starting time for the simulation.
        t_stop (float): Maximum time for simulation.
        t_samples (int): Number of time samples.

    Returns:
        List: time (t), angle (theta), velocity (v), and acceleration (a).
    """

    S_0 = initial_state
    t = np.linspace(t_start, t_stop, t_samples)

    solution = odeint(get_inv_pendulum_model, y0=S_0, t=t, args=(input_params, torque_control, f_ext))
    theta_sol = solution.T[0]
    vel_sol = solution.T[1]
    acc_sol = get_inv_pendulum_acceleration(input_params, theta_sol, vel_sol, torque_control, f_ext)

    return t, theta_sol, vel_sol, acc_sol


def impulse(dt, current_t, t_impulse, magnitude):
    """
    Function:
        Impulse function.

    Parameters:
        dt (float): Time interval for each step of the simulation.
        current_t (float): Current simulation time.
        t_impulse (float): Time when the impulse should appear.
        magnitude (float): Magnitude of the impulse.

    Returns:
        _ (float): Impulse magnitude
    """

    if abs(current_t - t_impulse) <= dt:
        return magnitude
    else:
        return 0


def pid_control(kp, ki, kd, target, inputs, prev_integral_error, prev_error, dt):
    """
    Function:
        PID controller.

    Parameters:
        kp, ki, kd (float): PID gains.
        target (float): Desired/Reference value.
        inputs (list): current system value (ex. angle, velocity)
        prev_integral_error (float): cummulative sum of the error (integral)
```

```
        prev_error (float): previous step error (derivative)
        dt (float): time step

    Returns:
        output: controlled output
        integral_error: updated integral error
        error: updated error (prev_error for the next iteration)
    """
    error = target - inputs
    integral_error = prev_integral_error + error * dt
    derivative_error = (error - prev_error) / dt

    output = (kp * error) + (ki * integral_error) + (kd * derivative_error) # Output is torque

    return output, integral_error, error


def get_pid_gains(input_params, init_conditions, target = 0, t_max = 5, t_samples = 300):
    """
    Function:
        Plots a series of graphs with the PID behavior for several gain combinations.
        This function is only run once for the PID calibration.

    Parameters:
        input_params (list): Physical constants for the differential ecuation.
        init_conditions (list): Initial 'theta' and 'vel' values.
        target (float): Desired/Reference value.
        t_max (float): Maximum time for simulation.
        t_samples (int): Number of time samples.

    Returns:
        None
    """

    input_params = DYNAMIC_INPUT_PARAMS
    init_conditions = [0.2, 0.0]  # angle, velocity

    # Gain combinations to test (Kp, Ki, Kd)
    pid_tests = [
        (10, 0, 0),   # Pure proporcional gain
        (20, 0, 5),   # Proporcional + derivative
        (30, 2, 8),   # Agressive PID
        (50, 5, 10),  # Strong PID
        (15, 1, 2),   # Soft PID
    ]

    for i, pid_gains in enumerate(pid_tests):
        data = get_simulated_data(input_params, target, init_conditions, pid_gains, t_max, t_samples)

        t = data['t']
        theta = data['theta']
        vel = data['vel']
        torque = data['torque']

        model_solutions = {
            'theta': theta,
            'vel': vel,
            'torque': torque
        }

        create_multi_y_graph(
            x_values = t,
            x_title = "Tiempo [s]",
            y_values_dict = model_solutions,
            plot_type = "scatter",
            show_plot = True,
            graph_title = f"PID Test {i+1} - Kp={pid_gains[0]}, Ki={pid_gains[1]}, Kd={pid_gains[2]}",
            image_name = f"pid_test_{i+1}",
            image_path = RUN_AREA,
        )

        create_simple_graph(
            x_values = t,
            x_title = "Tiempo",
            y_values = model_solutions['theta'],
            y_title = "Theta",
            annotate_values = [],
```

```
            plot_type = "scatter",
            show_plot = True,
            graph_title = f"PID Test Theta {i+1} - Kp={pid_gains[0]}, Ki={pid_gains[1]}, Kd={pid_gains[2]}"
    ,
            image_name = f"pid_test_theta_{i+1}",
            image_path = RUN_AREA,
        )


def get_simulated_data(input_params, target, init_conditions, pid_gains, t_max, t_samples, perturbance):
    """
    Function:
        Simulates the inverted pendulum system control (PID).

    Parameters:
        input_params (list): Physical constants for the differential ecuation.
        target (float): Desired/Reference value.
        init_conditions (list): Initial 'theta' and 'vel' values.
        pid_gains (list): Gain constants for the PID controller.
        t_max (float): Maximum time for simulation.
        t_samples (int): Number of time samples.
        perturbance (bool): Indicates the presence of a perturbance in the simulation.

    Returns:
        dataset (dict):
            't': time (array)
            'theta': angle (array)
            'vel': angular velocity (array)
            'acc': angular acceleration (array)
            'torque': control torque (array)
    """
    global PERTURBANCE_SIM_COUNT

    dt = t_max / (t_samples - 1)
    t = np.linspace(0, t_max, t_samples)

    # Initial Conditions
    theta, vel = init_conditions

    # PID parameters
    integral_error = 0.0
    prev_error = 0.0
    kp = pid_gains[0]
    ki = pid_gains[1]
    kd = pid_gains[2]

    # Lists to save the results
    theta_list = []
    vel_list = []
    acc_list = []
    torque_list = []

    if perturbance == True and PERTURBANCE_SIM_COUNT < PERTURBANCE_SIM_LIMIT:
        t_impulse = random.uniform(0.0, t_max/2)
        PERTURBANCE_SIM_COUNT += 1
    else:
        t_impulse = t_max + 1

    for i in range(t_samples):
        t_seconds = (i/t_samples) * t_max
        magnitude = random.uniform(-100, 100)
        f_ext = impulse(dt, t_seconds, t_impulse, magnitude)

        torque, integral_error, prev_error = pid_control(
            kp, ki, kd, target, theta, integral_error, prev_error, dt
        )
        acc = get_inv_pendulum_acceleration(input_params, theta, vel, torque, 0)

        theta_list.append(theta)
        vel_list.append(vel)
        acc_list.append(acc)
        torque_list.append(torque)

        # Solve system with small time step
        S_0 = [theta, vel]
        _, theta_sol, vel_sol, acc_sol = solve_inv_pendulum_model(input_params, torque, S_0, 0, dt, 2,
    f_ext)
```

```
        theta = theta_sol[-1]   # Gets the last calculated theta
        vel = vel_sol[-1]       # Gets the last calculated velocity
        acc = acc_sol[-1]       # Gets the last calculated acceleration

    sim_data = {
        't': t,
        'theta': np.array(theta_list),
        'vel': np.array(vel_list),
        'acc': np.array(acc_list),
        'torque': np.array(torque_list)
    }

    return sim_data


def get_simulated_data_from_network(input_params, init_conditions, model, t_max, t_samples, perturbance):
    """
    Function:
        Simulates the inverted pendulum system control with the neural network.

    Parameters:
        input_params (list): Physical constants for the differential ecuation.
        init_conditions (list): Initial 'theta' and 'vel' values.
        model (model.keras): The neural network model.
        t_max (float): Maximum time for simulation.
        t_samples (int): Number of time samples.
        perturbance (bool): Indicates the presence of a perturbance in the simulation.

    Returns:
        dataset (dict):
            't': time (array)
            'theta': angle (array)
            'vel': angular velocity (array)
            'acc': angular acceleration (array)
            'torque': control torque (array)
    """
    global PERTURBANCE_SIM_COUNT

    dt = t_max / (t_samples - 1)
    t = np.linspace(0, t_max, t_samples)

    # Initial Conditions
    theta, vel = init_conditions

    # Lists to save the results
    theta_list = []
    vel_list = []
    acc_list = []
    torque_list = []

    if perturbance == True and PERTURBANCE_SIM_COUNT < PERTURBANCE_SIM_LIMIT:
        t_impulse = random.uniform(0.0, t_max/2)
        PERTURBANCE_SIM_COUNT += 1
    else:
        t_impulse = t_max + 1

    for i in range(t_samples):
        t_seconds = (i/t_samples) * t_max
        magnitude = random.uniform(-100, 100)
        f_ext = impulse(dt, t_seconds, t_impulse, magnitude)

        # Simulate model
        torque = model_predict(model, theta, vel)
        acc = get_inv_pendulum_acceleration(input_params, theta, vel, torque, 0)

        theta_list.append(theta)
        vel_list.append(vel)
        acc_list.append(acc)
        torque_list.append(torque)

        # Solve system with small time step
        S_0 = [theta, vel]
        _, theta_sol, vel_sol, acc_sol = solve_inv_pendulum_model(input_params, torque, S_0, 0, dt, 2,
    f_ext)

        theta = theta_sol[-1]   # Gets the last calculated theta
```

```
        vel = vel_sol[-1]       # Gets the last calculated velocity
        acc = acc_sol[-1]       # Gets the last calculated acceleration

    sim_data = {
        't': t,
        'theta': np.array(theta_list),
        'vel': np.array(vel_list),
        'acc': np.array(acc_list),
        'torque': np.array(torque_list)
    }

    return sim_data


def generate_dataset(n_sims, data_path, option, model=None, backup=False):
    """
    Function:
        Creates a dataset with multiple simulations of the
        inverted pendulum system control (PID).

    Parameters:
        n_sims (int): Number of simulations.
        data_path (str): Path of the directory to save the dataset.
        option (str): 'PID' or 'NETWORK' depending on the type of
                      simulation to execute.
        model (model.keras): Neural network model in case 'NETWORK'
                             option is provided.
        backup (bool): Whether to save or not a backup of the previous
                       dataset before creating the new one.

    Retorna:
        data_path (str): Full path where the dataset files were saved.
        norm_df_dataset (pd.DataFrame): Normalized dataset.
    """
    global PERTURBANCE_SIM_COUNT

    if option == "PID":
        print("\n--> Generating Dataset ...")
    elif option == "NETWORK":
        print("\n--> Simulating Network ...")
    else:
        print(f"-E-: {option} is not a valid option. Use 'PID' or 'NETWORK'.")
        return 1

    PERTURBANCE_SIM_COUNT = 0
    input_params = DYNAMIC_INPUT_PARAMS
    target = TARGET_ANGLE
    pid_gains = [PID_KP, PID_KI, PID_KD]
    t_stop = SIM_T_STOP
    t_samples = SIM_T_SAMPLES

    create_directory(data_path, backup, overwrite=False)

    df_list = [] # List of DataFrames/simulations

    for i in range(n_sims):
        id = i + 1 # Simulation Number
        init_conditions = [random.uniform(-0.2, 0.2),    # angle [rad]
                           random.uniform(-0.5, 0.5)     # velocity [rad/s]
                           ]

        perturbance = random.choice([True, False])

        if option == "PID":
            data = get_simulated_data(input_params, target, init_conditions, pid_gains, t_stop, t_samples,
    perturbance)
        elif option == "NETWORK":
            data = get_simulated_data_from_network(input_params, init_conditions, model, t_stop, t_samples,
     perturbance)

        # DataFrame generation (to save as .csv)
        df = pd.DataFrame({
            't': data['t'],
            'theta': data['theta'],
            'vel': data['vel'],
            'acc': data['acc'],
            'torque': data['torque']
```

```
        })
        df["simulacion"] = id # Simulation Number
        df_list.append(df)

        # Plot generation (to save as .png)
        t = data['t']
        model_solutions = {
            'theta': data['theta'],
            'vel': data['vel'],
            'acc': data['acc'],
            'torque': data['torque']
        }

        # Saves plots as .png
        create_simple_graph(
            x_values = t,
            x_title = "Tiempo",
            y_values = data['theta'],
            y_title = "Theta",
            annotate_values = [],
            plot_type = "plot",
            show_plot = False,
            graph_title = "Inverted Pendulum Simulation (Theta)",
            image_name = f"inv_pen_sim_theta{id}",
            image_path = data_path,
        )

        create_multi_y_graph(
            x_values = t,
            x_title = "Tiempo",
            y_values_dict = model_solutions,
            plot_type = "plot",
            show_plot = False,
            graph_title = "Inverted Pendulum Simulation",
            image_name = f"inv_pen_sim_{id}",
            image_path = data_path,
        )

    # Builds full dataframe
    df_dataset = pd.concat(df_list, ignore_index=True)

    # Variables for data normalization
    max_df_vals = df_dataset.max(axis=0) # Max of each dataset column
    min_df_vals = df_dataset.min(axis=0) # Min of each dataset column

    # Saves variables for data normalization into config file
    save_norm_config(max_df_vals, min_df_vals)

    # Normalizes dataframe
    norm_df_dataset = process_df("normalize", df_dataset, "", max_df_vals, min_df_vals)

    # Saves data in .csv
    if option == "PID":
        file_path = DATASET_CSV_PATH
        norm_df_dataset.to_csv(file_path, index=False)
        print(f"\nDataset saved as: '{file_path}'")
    elif option == "NETWORK":
        file_path = NETWORK_SIM_CSV_PATH
        norm_df_dataset.to_csv(file_path, index=False)
        print(f"\nNetwork simulation saved as: '{file_path}'")
    print("")

    return data_path, norm_df_dataset


def load_dataset(csv_path):
    """
    Function:
        Loads a saved dataset.

    Parameters:
        csv_path (str): Path of the .csv file to load.

    Retorna:
        dataset_df (pd.DataFrame): Dataset values in the form of a DataFrame.
    """
```

```
    print(f"\n--> Loading dataset from .csv file:\n{csv_path}\n")
    dataset_df = pd.read_csv(csv_path)
    return dataset_df


##### TEST FUNCTIONS (For Developers) #####

def test_solve_inv_pendulum_model():
    input_params = DYNAMIC_INPUT_PARAMS
    torque = 5.0
    initial_state = [0.1, 0.0]  # angle [rad], angular velocity [rad/s]

    t_start = 0
    t_stop = 20
    t_samples = 100

    t, theta, vel, acc = solve_inv_pendulum_model(input_params, torque, initial_state, t_start, t_stop,
    t_samples)

    model_solutions = {
        "theta [rad]" : theta,
        "vel [m/s]" : vel,
        "acc [m/s^2]" : acc
    }

    create_multi_y_graph(
        x_values = t,
        x_title = "Tiempo",
        y_values_dict = model_solutions,
        plot_type = "plot",
        show_plot = True,
        graph_title = "Inv Pendulum Model",
        image_name = "inv_pendulum_model_test",
        image_path = RUN_AREA,
    )


def test_pid_control():
    # Random values for testing
    kp = PID_KP
    ki = PID_KI
    kd = PID_KD
    dt = 0.1
    target = 0.0
    input_val = 0.8
    prev_integral_error = 0.05
    prev_error = 0.1

    torque, integral_error, error_out = pid_control(kp, ki, kd, target, input_val, prev_integral_error,
    prev_error, dt)

    print("Torque: ", torque)
    print("integral_error: ", integral_error)
    print("error_out: ", error_out)


def test_get_pid_gains():
    input_params = DYNAMIC_INPUT_PARAMS
    init_conditions = [0.2, 0.0]  # angle [rad], angular velocity [rad/s]
    get_pid_gains(input_params, init_conditions)


def test_get_simulated_data(option):
    input_params = DYNAMIC_INPUT_PARAMS
    initial_state = [0.1, 0.0] # angle [rad], angular velocity [rad/s]
    target = TARGET_ANGLE
    pid_gains = [PID_KP, PID_KI, PID_KD]
    t_stop = 10
    t_samples = 200

    if option == "PID":
        data = get_simulated_data(input_params, target, initial_state, pid_gains, t_stop, t_samples, False)
    elif option == "NETWORK":
        model = load_model(MODEL_SAVE_PATH)
        data = get_simulated_data_from_network(input_params, initial_state, model, t_stop, t_samples, False
    )
```

```
t = data['t']

model_solutions = {
    'theta': data['theta'],
    'vel': data['vel'],
    'acc': data['acc'],
    'torque': data['torque']
}

create_simple_graph(
    x_values = t,
    x_title = "Tiempo",
    y_values = data['theta'],
    y_title = "Theta",
    annotate_values = [],
    plot_type = "plot",
    show_plot = True,
    graph_title = "Theta Simulation",
    image_name = f"theta_simulation",
    image_path = RUN_AREA,
)

create_multi_y_graph(
    x_values = t,
    x_title = "Tiempo",
    y_values_dict = model_solutions,
    plot_type = "plot",
    show_plot = True,
    graph_title = "All Model Simulation",
    image_name = f"all_model_simulation_test",
    image_path = RUN_AREA,
)
```

ANEXO K. CÓDIGO NETWORK.PY

```
##### IMPORTS #####


# Third party imports
from tensorflow.keras.models import Sequential
from tensorflow.keras.models import load_model as keras_load_model
from tensorflow.keras.layers import Dense
import numpy as np
import pandas as pd


# Local imports
from .config import *
from .utils import *


##### FUNCTIONS DEFINITION #####

def get_model_shapes(
        num_input_neurons,
        num_hidden_neurons,
        num_output_neurons
        ):
    """
    Function:
        Creates a list with the shapes of the weights and biases of the the model.

    Parameters:
        num_hidden_neurons (list): List of neurons of each hidden layer.
        num_input_neurons (int): Number of neurons of the input layer.
        num_output_neurons (int): Number of neurons of the output layer.

    Returns:
        shapes (list):  List of shapes of the model with the following format:
                        [(num of input neurons, num weigths for each neuron),
                        (bias of input layer,),
                        (num of neurons, num weigths for each neuron),
                        (bias of hidden layer,),
                        (num of output neurons, num weigths for each neuron),
                        (bias of output layer,)]
    """
    shapes = []
    layer_sizes = [num_input_neurons] + num_hidden_neurons + [num_output_neurons]

    for i in range(len(layer_sizes) - 1):
        in_dim = layer_sizes[i]
        out_dim = layer_sizes[i + 1]

        shapes.append((in_dim, out_dim))  # pesos
        shapes.append((out_dim,))         # bias

    return shapes


def build_model(
        num_input_neurons,
        num_hidden_neurons,
        num_output_neurons,
        input_act_func,
        hidden_act_func,
        output_act_func
        ):
    """
    Function:
        Builds a dense neural network model given a number of neurons
        for the dense layer.

    Parameters:
        num_hidden_neurons (list): Number of neurons of the dense layer.
        num_input_neurons (int): Number of neurons of the input layer.
        num_output_neurons (int): Number of neurons of the output layer.
        input_act_funct (str): Activation function for the network's input layer neurons.
        hidden_act_funct (str): Activation function for the network's hidden layer neurons.
        output_act_funct (str): Activation function for the network's output layer neurons.
```

```
    Returns:
        model: A neural network dense model.
    """

    print("\n--> Creating the Neural Network ...\n")

    model = Sequential()

    # First layer
    model.add(Dense(num_hidden_neurons[0],
                    input_shape=(num_input_neurons,),
                    activation=input_act_func))

    # Hidden layers
    for units in num_hidden_neurons[1:]:
        model.add(Dense(units, activation=hidden_act_func))

    # Output layer
    model.add(Dense(num_output_neurons, activation=output_act_func))

    return model


def set_model_weights(model, weights):
    """
    Function:
        Sets the model with the input weights.

    Parameters:
        model (model.keras): Neural network model.
        weights (list): Desired weights for the neural network.

    Returns:
        model (model.keras): Neural network model with the assigned weights.
    """

    shapes = get_model_shapes(
        NUM_INPUT_NEURONS,
        NUM_HIDDEN_NEURONS,
        NUM_OUTPUT_NEURONS
    )
    idx = 0
    new_weights = []
    for shape in shapes:
        size = np.prod(shape)
        array = np.array(weights[idx:idx+size]).reshape(shape)
        new_weights.append(array)
        idx += size

    model.set_weights(new_weights)

    return model


def model_predict(model, theta, vel):
    """
    Function:
        Predicts the control torque using the provided neural network
        model.

    Parameters:
        model (model.keras): Neural network model.
        theta (float): Angle value.
        vel (float): Velocity value.

    Returns:
        torque_denorm (float): Denormalized predicted control torque by
        the neural network.
    """

    max_df_vals, min_df_vals = get_norm_config()

    # Normalizing inputs
    input_df = pd.DataFrame([[theta, vel]], columns=["theta", "vel"])
    input_norm_df = process_df("normalize", input_df, "", max_df_vals, min_df_vals)
    theta_norm = input_norm_df["theta"].to_numpy()
    vel_norm = input_norm_df["vel"].to_numpy()
```

```
    model_input = np.column_stack((theta_norm, vel_norm))

    # Predicting the output torque
    torque = model.predict(model_input)

    # Denormalizing output
    output_df = pd.DataFrame([[torque]], columns=["torque"])
    output_norm_df = process_df("denormalize", output_df, "", max_df_vals, min_df_vals)
    torque_denorm = output_norm_df["torque"].to_numpy()

    return float(torque_denorm)


def save_model(model, model_save_path):
    """
    Function:
        Saves the model in the provided directory.

    Parameters:
        model (model.keras): Neural network model.
        model_save_path (str): Directory path to save the model.

    Returns:
        None
    """

    print ("\n--> Saving model ...\n")

    model.save(model_save_path)

    print("Model saved in:\n")
    print(model_save_path, "\n")


def load_model(model_save_path):
    """
    Function:
        Loads the model from the provided directory.

    Parameters:
        model_save_path (str): Directory path from which to load the model.

    Returns:
        model (model.keras): Neural network model.
    """

    print ("\n--> Loading model...\n")
    model = keras_load_model(model_save_path)
    print(f"Loaded model from:\n{model_save_path}\n")
    return model


##### TEST FUNCTIONS (For Developers) #####

def test_set_model_weights():

    # Test the function
    model = build_model(
        NUM_INPUT_NEURONS,
        NUM_HIDDEN_NEURONS,
        NUM_OUTPUT_NEURONS,
        INPUT_ACT_FUNCTION,
        HIDDEN_ACT_FUNCTION,
        OUTPUT_ACT_FUNCTION
        )


    shapes = get_model_shapes(
        NUM_INPUT_NEURONS,
        NUM_HIDDEN_NEURONS,
        NUM_OUTPUT_NEURONS
        )

    print("\nshapes: ", shapes)
    print("\n")

    # Generate flat weight list with the correct total size: 2*5 + 5 + 5*1 + 1 = 21
```

```
    weights = np.arange(1, 22)  # [1, 2, ..., 21]

    # Set weights using your function
    set_model_weights(model, weights, shapes)

    # Get weights from the model to verify
    model_weights = model.get_weights()

    print("Model weights: ")
    print(model_weights)
    print("\n")

    # Flatten model weights to compare
    flattened = np.concatenate([w.flatten() for w in model_weights])

    # Assertion
    assert np.array_equal(flattened, weights), "Weights were not set correctly"
    print("Test passed: Weights correctly set.")
```

ANEXO L. CÓDIGO EVOLUTION.PY

```
##### IMPORTS #####


# Third party imports
import numpy as np
from deap import creator, base, tools, algorithms
import random
import pandas as pd
from functools import partial

# Local imports
from .config import *
from .utils import *
from .network import *
from .simulation import *


##### FUNCTIONS DEFINITION #####

toolbox = base.Toolbox()


def get_model_num_weights(net_model):
    """
    Function:
        Sums the total number of weights and biases of the neural network.

    Parameters:
        net_model (model.keras): Neural network model.

    Returns:
        total_weights_num (int): Total number of weights and biases.
    """

    weights = net_model.get_weights()
    total_weights_num = 0
    for weight in weights:
        total_weights_num += weight.size
    return total_weights_num


def evaluation_function(
        individual,
        model,
        dataset_df
        ):

    """
    Function:
        Evaluation function used to set the fitness of an individual.

    Parameters:
        individual (list): Defined with the neural network possible weights.
        model (model.keras): Neural network model.
        dataset_df (pd.DataFrame): The dataset to which compare the model prediction.

    Returns:
        fitness (array): Fitness value of the individual.
    """

    # Sets the model with the weights defined by the individual
    set_model_weights(model, individual)

    torque_array = dataset_df["torque"].to_numpy()

    max_df_vals, min_df_vals = get_norm_config()

    # Extract angle and velocity
    theta_norm = dataset_df["theta"].to_numpy()
    vel_norm = dataset_df["vel"].to_numpy()

    # Predict torque with the neural network
    model_input = np.column_stack((theta_norm, vel_norm))
    norm_network_torque = model.predict(model_input)
```

```
    # Denormalize the torque
    norm_net_torque_df = pd.DataFrame(norm_network_torque, columns=["torque"])
    norm_dataset = process_df("denormalize", norm_net_torque_df, "", max_df_vals, min_df_vals)
    network_torque = norm_dataset["torque"]

    # Calculate the fitness of the individual
    fitness = np.mean((np.array(torque_array) - np.array(network_torque)) ** 2)

    return (fitness,)


def run_evolutionary_algorithm(net_model, dataset_df):
    """
    Function:
        Runs a multi-objective evolutionary algorithm using the DEAP library.

        This function sets up and executes an evolutionary process to optimize two
        conflicting objectives (displacement and acceleration). The function registers
        genetic operators, initializes the population, executes the evolution, and
        visualizes the results.

    Parameters:
        None

    Returns:
        best_ind (list): List with the trained weights of the neural network.
        best_ind_fitness (float) : Fitness value of the 'best_ind'.
    """

    print("\n--> Running the Evolutionary Algorithm ...\n")

    # Negative weights for minimization
    pesos_fitness = (-1.0,)

    n_genes = get_model_num_weights(net_model)

    # Fitness function definition
    creator.create("fitness_function", base.Fitness, weights=pesos_fitness)
    # Individual definition
    creator.create("individual", list, fitness=creator.fitness_function, typecode="f")

    # Alleles
    toolbox.register("gene", random.uniform, GENE_RANGE[0], GENE_RANGE[1])

    # Individual generator
    toolbox.register(
        "individual_generation",
        tools.initRepeat,
        creator.individual,
        toolbox.gene,
        n_genes,
    )

    # Population generator
    toolbox.register(
        "population", tools.initRepeat, list, toolbox.individual_generation
    )

    toolbox.register("evaluate", partial(evaluation_function, model=net_model, dataset_df=dataset_df))
    # Evolution operators
    toolbox.register("select", tools.selTournament, tournsize=2)
    toolbox.register("mate", tools.cxBlend, alpha=ALPHA)
    toolbox.register("mutate", tools.mutGaussian, mu=MU, sigma=SIGMA, indpb=0.2)

    if DEBUG == True:
        # Test for the population and individuals generation
        population_test = toolbox.population(n=POPULATION_SIZE)
        individual_test = toolbox.individual_generation()
        print("Individuo: ", individual_test)
        print("Ejemplo de poblacion: ", population_test)

    # Statistics on the general fitness of the population
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", np.mean) # Generation 'Average'
    stats.register("std", np.std)  # Individuals 'Standard Deviation'
    stats.register("min", np.min)  # 'Min Fitness' of the generation
    stats.register("max", np.max)  # 'Max Fitness' of the generation
```

```
    hof = tools.HallOfFame(1)  # Hall of Fame
    popu = toolbox.population(n=POPULATION_SIZE)  # Defines the initial population

    # Runs the Evolutionary Algorithm
    popu, logbook = algorithms.eaMuPlusLambda(
        population=popu,
        toolbox=toolbox,
        mu=PARENT_POPU_SIZE,
        lambda_=CHILD_POPU_SIZE,
        cxpb=MATE_CHANCE,
        mutpb=MUTATE_CHANCE,
        ngen=NUM_GENERATIONS,
        stats=stats,
        halloffame=hof,
        verbose=VERBOSE,
    )

    best_ind = hof[0]
    best_ind_fitness = best_ind.fitness.values
    log_df = pd.DataFrame(logbook)

    # View last generation summary
    print()
    print("Last generation: ")
    print(log_df.tail(1))
    print("")

    # Plot fitness across generations
    y_fitness_dict = {
        "Min Fitness" :  log_df["min"],
        "Avg Fitness" : log_df["avg"]
    }
    create_multi_y_graph(
        x_values = log_df["gen"],
        x_title = "Generation",
        y_values_dict = y_fitness_dict,
        plot_type = "plot",
        show_plot = True,
        graph_title = "Fitness over Generations",
        image_name = "evolution_fitness",
        image_path = RESULTS_DIR_PATH
    )

    return best_ind, best_ind_fitness
```

ANEXO M. CÓDIGO UTILS.PY

```
##### IMPORTS #####

# Third party imports
import pandas as pd
import matplotlib.pyplot as plt

# Built-in imports
import shutil
import os

# Local imports
from .config import *


##### FUNCTIONS DEFINITION #####

def create_directory(full_path, backup=False, overwrite=True):
    """
    Function:
        Manages directories creation.

    Parameters:
        full_path (str): Full path of the directory to create.
        backup (bool): If the directory was to be overwritten, creates a
            backup with an "_old" suffix.
        overwrite (bool): Wheter to overwrite the directory or not if it
            exists.

    Returns:
        None
    """

    if backup == True:
        if os.path.exists(full_path):
            old_path = full_path + "_old"
            if os.path.exists(old_path):
                shutil.rmtree(old_path)
            os.replace(full_path, old_path)
        else:
            os.makedirs(full_path)
    else:
        if os.path.exists(full_path):
            if overwrite == True:
                shutil.rmtree(full_path)
                os.makedirs(full_path)
        else:
            os.makedirs(full_path)


def print_files_tree_short():
    """
    Function:
        Prints the hierarchical tree structure of the main
        program files and modules. Includes only code files.
        Used to explain the code in the written report.

    Parameters:
        None

    Returns:
        None
    """

    print("""
Program File Tree:
C:.
      main.py

      modules
        config.py
        evolution.py
        network.py
        simulation.py
        utils.py
        __init__.py
```

```
        """)


def process_df(operation, df, title, max_vals, min_vals):
    """
    Function:
        Normalizes or denormalizes values of a pd.DataFrame.

    Par metros:
        operation (str): To choose between 'normalize' o 'denormalize'.
        df (Pandas.DataFrame): DataFrame to process.
        title (str): Title to print in debug mode.
        max_vals (array): max values of the DataFrame for the normalization.
        min_vals (array): min values of the DataFrame for the normalization.

    Retorna:
        norm_df (Pandas.DataFrame): (De)normalized DataFrame.
    """

    difference = max_vals - min_vals

    if operation == "normalize":
        # Normalizaci n tomando en cuenta caso donde toda la columna tiene
        # elementos iguales
        norm_df = df.apply(
            lambda col: (
                (col - min_vals[col.name]) / difference[col.name]
                if difference[col.name] != 0
                else pd.Series([0.0] * len(col), index=col.index)
            )
        )
    elif operation == "denormalize":
        # Des-normalizaci n tomando en cuenta caso donde toda la columna tiene
        # elementos iguales
        norm_df = df.apply(
            lambda col: (
                col * difference[col.name] + min_vals[col.name]
                if difference[col.name] != 0
                else pd.Series([min_vals[col.name]] * len(col), index=col.index)
            )
        )
    else:
        raise ValueError(
            "-I-: 'operation' parameter should be one of the following: 'normalize' or 'denormalize'."
        )

    norm_df = norm_df.astype(float)

    if DEBUG == 1:
        title = operation.capitalize() + "d " + title + ":"
        print(title)
        print(norm_df)

    return norm_df


def save_norm_config(max_vals, min_vals):
    """
    Function:
        Saves a configuration file with the max and min values
        used in the dataset normalization for future reference.

        Files are saved in the MODEL_CONFIG_PATH_LIST paths
        ('results' directory).

    Parameters:
        max_vals (array): max values of the DataFrame for the normalization.
        min_vals (array): min values of the DataFrame for the normalization.

    Returns:
        None
    """

    print("\n--> Saving normalization data ...")

    max_config_path = MODEL_CONFIG_PATH_LIST[0]
    min_config_path = MODEL_CONFIG_PATH_LIST[1]
```

```
    max_vals = max_vals.to_frame().T
    min_vals = min_vals.to_frame().T
    max_vals.to_csv(max_config_path, index=False)
    min_vals.to_csv(min_config_path, index=False)

    print("Normalization configs created: ")
    print(MODEL_CONFIG_PATH_LIST[0])
    print(MODEL_CONFIG_PATH_LIST[1], "\n")


def get_norm_config():
    """
    Function:
        Returns the max and min values read from the normalization
        config files.

    Parameters:
        None

    Returns:
        max_df_vals (array): max values of the DataFrame for the normalization.
        min_df_vals (array): min values of the DataFrame for the normalization.
    """

    max_df_recuperado = pd.read_csv(MODEL_CONFIG_PATH_LIST[0])
    min_df_recuperado = pd.read_csv(MODEL_CONFIG_PATH_LIST[1])
    max_df_vals = max_df_recuperado.iloc[0]
    min_df_vals = min_df_recuperado.iloc[0]

    return max_df_vals, min_df_vals


def create_simple_graph(
    x_values,
    x_title,
    y_values,
    y_title,
    annotate_values,
    plot_type,
    show_plot,
    graph_title,
    image_name,
    image_path,
):
    """
    Function:
        Creates and saves a 2D graph (scatter or line).

    Parameters:
        x_values (list): Data points for the x-axis.
        x_title (str): Label for the x-axis.
        y_values (list): Data points for the y-axis.
        y_title (str): Label for the y-axis.
        annotate_values (list): Optional. List of two lists
                                [k_values, b_values] for annotating each
                                (k, b) point.
        plot_type (str): Type of plot to create. Valid options: 'scatter',
                         others default to line plot.
        show_plot (bool): Shows the plot while running if True.
        graph_title (str): Title of the graph.
        image_name (str): Name of the image to generate.
        image_path (str): Path to save the image.

    Returns:
        None
    """

    plt.figure()
    match plot_type:
        case "scatter":
            plt.scatter(x_values, y_values)
        case "plot":
            plt.plot(x_values, y_values)
        case _:
            print(
                "-W-: For 'create_multi_y_graph' attribute 'plot_type' use: scatter or plot"
```

```
            )

    # Annotate each point with k and b values
    if len(annotate_values) == 2:
        k_values = annotate_values[0]
        b_values = annotate_values[1]
        for i in range(len(x_values)):
            plt.annotate(
                f"k={k_values[i]:.1f}\nb={b_values[i]:.1f}",
                (x_values[i], y_values[i]),
                textcoords="offset points",
                xytext=(5, 5),
                ha="left",
                fontsize=8,
            )

    plt.xlabel(x_title)
    plt.ylabel(y_title)
    plt.title(graph_title)
    plt.savefig(f"{image_path}/{image_name}")
    if show_plot == True:
        plt.show()
    plt.close()


def create_multi_y_graph(
    x_values,
    x_title,
    y_values_dict,
    plot_type,
    show_plot,
    graph_title,
    image_name,
    image_path,
):
    """
    Function:
        Creates and saves a 2D graph with multiple y-axis functions
        (scatter or line).

    Parameters:
        x_values (list or array): Data points for the x-axis.
        x_title (str): Label for the x-axis.
        y_values_dict (dict): Dictionary where keys are labels and values are
                              lists of y data.
        plot_type (str): Type of plot to create. Valid options: 'scatter',
                         others default to line plot.
        show_plot (bool): Shows the plot while running if True.
        graph_title (str): Title of the graph.
        image_name (str): Name of the image to generate.
        image_path (str): Path to save the image.

    Returns:
        None
    """

    plt.figure()
    for y_title, y_values in y_values_dict.items():
        match plot_type:
            case "scatter":
                plt.scatter(x_values, y_values, label=y_title)
            case "plot":
                plt.plot(x_values, y_values, label=y_title)
            case _:
                print(
                    "-W-: For 'create_multi_y_graph' attribute 'plot_type' use: scatter or plot"
                )
    plt.legend(loc="upper right")
    plt.xlabel(x_title)
    plt.title(graph_title)
    plt.savefig(f"{image_path}/{image_name}")
    if show_plot == True:
        plt.show()
    plt.close()
```