UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Implementación del algoritmo NEAT (neuroevolución por topologías aumentadas) para el control de un sistema caótico

TESIS

Que para obtener el título de

Ingeniero Mecatrónico

P R E S E N T A

Rodrigo Gamba Lavín

DIRECTOR(A) DE TESIS

Billy Arturo Flores Medero Navarro

Ciudad Universitaria, Cd. Mx., Ingresa 2019

"En las regiones del espíritu no tiene límites el descubrimiento de nuevos mundos."

- José Vasconcelos

Índice General

Resumen

En este proyecto se ha desarrollado una implementación del algoritmo NEAT (Neuroevolución por Topologías Aumentadas por sus siglas en inglés) en Python 3 con el objetivo de simular y controlar el comportamiento de un sistema dinámico no lineal como lo es un péndulo invertido doble sobre un carro.

Para realizar las simulaciones del algoritmo, se propone en un inicio obtener el modelo matemático del sistema que se pretende controlar, luego desarrollar el módulo en Python3 para finalmente utilizarlo en las pruebas de evaluación de desempeño de las propuestas de solución del algoritmo.

Se realizaron dos simulaciones del proceso de evolución, ajustando los parámetros de configuración del algoritmo, y finalmente se pudo obtener un comportamiento similar al que se obtendría mediante el diseño e implementación de modelos de control no lineal de alta complejidad con el objetivo de controlar el péndulo invertido doble sobre un carro móvil. Con estos resultados se pudo comprobar que el algoritmo NEAT y el aprendizaje de máquina basado en comportamiento puede servir para controlar cierto tipo de sistemas caóticos.

Capítulo 1. **Introducción**

1.1 Generalidades

El cerebro humano, fuente de creatividad y de las creaciones más asombrosas, es el objeto de estudio más complejo que hemos podido encontrar en el universo hasta el momento. Cuenta con billones de células transmitiendo información a través de impulsos electroquímicos que por alguna razón que aún no conocemos, desembocan en grandes ideas, teoremas matemáticos u obras de arte inigualables. Pero aún sin entender totalmente la complejidad de su comportamiento, el hombre ha tratado de imitar su funcionamiento básico. Es así como nacieron las redes neuronales artificiales, una vez que entendimos el funcionamiento básico de las redes neuronales biológicas se obtuvo un modelo matemático que describía su comportamiento y éste fue aplicado a problemas de clasificación y regresión. En un principio con no muy buenos resultados, pero variando su topología, funciones de activación y tipos de entrenamiento se fueron obteniendo mejores resultados. Una de las desventajas de usar estos modelos en vez de otras herramientas matemáticas para la solución de problemas, es que requerían de una gran capacidad de procesamiento para encontrar soluciones acertadas. Con el paso de los años la capacidad de procesamiento ha incrementado de manera exponencial, lo que ha permitido que este tipo de técnicas sea, no simplemente factible, sino la mejor opción para la solución de ciertos problemas.

La evolución es otro proceso en donde el humano ha tratado de imitar a la naturaleza para la solución de problemas. Una forma muy simplista de entender la evolución es la selección natural propuesta por Darwin y una serie de cambios aleatorios que propician la misma ventaja o desventaja de cierto individuo contra sus similares. Este proceso también fue generalizado e implementado, lo conocemos hoy en día como algoritmos genéticos y plantea evaluar poblaciones de soluciones para seleccionar las más aptas, combinarlas para producir descendientes y mutarlas aleatoriamente para explorar todo el espacio de soluciones posibles.

Si combinamos estas dos herramientas, haciendo evolucionar poblaciones de redes neuronales artificiales, podemos llegar a solucionar problemas de grados altos de complejidad como lo es el control de un sistema caótico sin siquiera tener el modelo matemático de su comportamiento.

Generales

- Investigar y estudiar distintas implementaciones del algoritmo NEAT (*Neuroevolution of Augmented Topologies*) en Python y otros lenguajes de programación para detectar áreas de oportunidad.
- Desarrollar el modelo matemático del péndulo invertido doble sobre un carro e implementar módulos en Python para su simulación.
- Desarrollar una propuesta de implementación del algoritmo NEAT en Python para controlar el modelo del péndulo invertido doble sobre un carro y medir resultados de entrenamiento.

Particulares

- Aplicar los conocimientos adquiridos en algebra lineal, cálculo vectorial y métodos numéricos, para entender más a fondo el comportamiento y modelado de las redes neuronales artificiales y algoritmos genéticos.
- Modelar el sistema del péndulo caótico sobre un carro en dos dimensiones mediante el método de LaGrange.
- Medir y analizar los resultados de la implementación del algoritmo.
- Realizar las simulaciones del control neuroevolutivo entrenado con diversos parámetros y medir resultados.

Capítulo 2. **Antecedentes**

En este capítulo se pretende hacer una breve explicación de todos los temas y áreas de estudio necesarias para la comprensión del trabajo. Cada uno de los conceptos explicados en este capítulo es fundamental para entender el funcionamiento de la implementación del algoritmo. Comenzando con la definición de los sistemas dinámicos, control automático, algoritmos genéticos, hasta la descripción del funcionamiento básico de las redes neuronales artificiales.

2.1 Sistemas dinámicos

Para entender los sistemas dinámicos es importante primero definir un sistema. Un sistema es simplemente un conjunto de elementos en interacción. Un sistema dinámico se refiere a un sistema en el cual una función describe la dependencia temporal de un punto en el espacio, esto quiere decir que un sistema dinámico es un sistema que varía su estado en el tiempo y que a su vez su estado puede ser descrito en el espacio.

Algunos ejemplos de sistemas dinámicos incluyen: modelos matemáticos que describen el comportamiento del vaivén de un péndulo, el flujo de agua a través de una pipa o incluso el incremento de la población de una especie a lo largo del año.

Existen diversas maneras de clasificar los sistemas dinámicos, una de ellas es por el tipo de evolución temporal del sistema, estos pueden ser de manera continua o de manera discreta. Es preciso distinguir estos dos tipos de sistemas dinámicos:

Sistemas dinámicos discretos: En este tipo de sistemas, el tiempo no varía de forma continua, es decir mediante saltos finitos. Un ejemplo claro puede ser el estudio del intervalo temporal entre una gota y otra, de manera sucesiva, de un grifo que gotea. [1]

Sistemas dinámicos continuos: En este tipo de sistemas se tiene en cuenta todos los valores posibles de la variable independiente que es el tiempo. Por ejemplo en nuestro caso, un péndulo que se mueve continuamente de un estado a otro. [1]

El análisis de los sistemas dinámicos continuos, requiere cálculo diferencial, dado que, se desea que la distancia entre el paso k y k+1 tienda a 0. Es decir, al aumentar infinitesimalmente el tiempo, la variación del sistema puede expresarse como una función del estado anterior. [1]

Referencias

[1] Modern Control Engineering". Fifth Edition. Pearson Education Inc., New Jersey, 2010.

2.2 Caos

El caos en términos generales, es el área que estudia el comportamiento de sistemas dinámicos que son altamente sensibles a condiciones iniciales. Esto quiere decir que variaciones minúsculas en las condiciones iniciales de un sistema pueden resultar en comportamientos totalmente distintos, y a su vez, en salidas completamente divergente, haciendo las predicciones a largo plazo prácticamente imposibles.

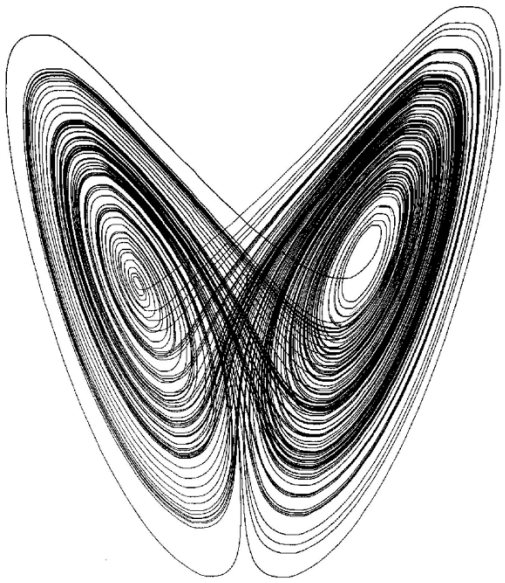


Fig. 2A Forma del atractor de Lorenz en 3 dimensiones

El 'caos determinista' es un término utilizado para describir el comportamiento irregular o no periódico de sistemas dinámicos producidos por una evolución en el tiempo estrictamente determinista, sin ninguna aleatoriedad externa. Esta irregularidad se traduce en una extrema dependencia de las condiciones iniciales que dificulta cualquier predicción a lo largo del tiempo de la dinámica del sistema. Existen sistemas caóticos con muy pocos grados de libertad.

Comúnmente "caos" significa "estado de desorden". Pero en la teoría del caos, se define con mayor precisión. Aunque no existe una definición matemática universalmente aceptada del caos, una definición frecuentemente utilizada fue dada por Robert L. Devaney, quien dice que para clasificar un sistema como caótico, debe tener las siguiente tres propiedades:

1. Movimiento Oscilante.
2. Determinismo: El sistema no es azaroso sino determinista. El comportamiento irregular, en dimensión finita surge de la no linealidad.
3. Sensibilidad a las condiciones iniciales: Las trayectorias que comienzan cercanas, con el tiempo se separan exponencialmente. En otras palabras, condiciones iniciales muy parecidas acaban dando lugar a comportamientos distintos después de cierto tiempo.

2.3 Control

El control automático ha tomado un rol de vital importancia en el avance de la ciencia y la ingeniería. El control en términos generales es la serie de operaciones necesarias que se deben de realizar a partir del estado de un sistema para que éste se comporte de la manera deseada. Para llegar a este objetivo se debe de tener muy claro el comportamiento del sistema que se quiere controlar, éste puede o no ser un sistema físico. En nuestro caso nos enfocaremos únicamente a sistemas físicos. Para llegar al modelo matemático de un sistema, se deben utilizar diversas herramientas matemáticas que en última instancia pretenden simular el comportamiento del sistema en términos cuantificables.

La rama de control automático es esencial en cualquier área de ciencias e ingeniería. La teoría de control se encarga del comportamiento de los sistemas dinámicos y de cómo modificar su estado para que tenga una respuesta deseada. Por ejemplo, el control es una parte primordial en sistemas de vehículos espaciales, sistemas robóticos, sistemas de manufactura moderna y cualquier proceso industrial en el cual se requiera controlar temperatura, presión, humedad, cantidad de flujo, etc. Dados los objetivos del proyecto nos limitaremos a explicar brevemente dos tipos de control: teoría de control clásico y teoría de control moderno.

Control clásico

La teoría de control clásico se enfoca en controlar un sistema, regularmente llamado planta, para que sus salidas (variables controladas) sigan el valor de una señal

controlada deseada llamada referencia, éste puede ser un valor estático o que cambie con el tiempo. Para llegar a este objetivo se diseña un controlador, el cual monitorea la salida del sistema y la compara con el valor de referencia. La diferencia entre la salida y el valor de referencia (la salida deseada), es la señal de error la cual es aplicada como realimentación a la(s) variable(s) entrada del sistema para acercar el valor de salida al de referencia, queriendo lograr así la reducción del error, siempre y cuando sea un sistema de control cerrado.

El control clásico se enfoca en sistemas lineales, invariantes en el tiempo de una sola entrada y una sola salida. Desafortunadamente para nosotros, existen muy pocos sistemas así en la naturaleza y por lo tanto su aplicación es limitada. Se utiliza la transformada de Laplace para cambiar una Ecuación Diferencial Ordinaria en el dominio del tiempo a un polinomio algebraico regular en el dominio de la frecuencia. Una vez que el sistema ha sido transformado, éste se puede manipular con mucho mayor facilidad.

En el diseño de controladores clásicos, se utiliza una función de transferencia que relaciona la transformada de Laplace de las salidas con las entradas en el dominio de la frecuencia.

Para describir el sistema es necesario diseñar: los controles usando los métodos de Nyquist y Bode, la magnitud y la fase de la frecuencia de respuesta. La función de transferencia también puede ser calculada y es necesaria para el cálculo de las raíces. El diagrama de bloque se usa para definir la función de transferencia de los sistemas compuestos. Para el diseño clásico no es necesaria la descripción de las dinámicas internas del sistema.

La teoría de control clásico es difícil de aplicar en sistemas con múltiples entradas o múltiples salidas (MIMO).

Control moderno

La teoría moderna de control se basa en el análisis del dominio del tiempo de sistemas de ecuaciones diferenciales. En la teoría de control moderno, en vez de cambiar de dominio para evitar complejidades matemáticas, convierte las ecuaciones diferenciales

ordinarias en un sistema de ecuaciones en el dominio del tiempo de menor orden llamado ecuaciones de estado, el cual puede ser manipulado usando técnicas de álgebra lineal.

Para realizar diseños de control moderno es fundamental la técnica del dominio del tiempo. Se requiere un modelo exacto del espacio de estados el sistema para poder controlarlo.

Muchos diseños de ingeniería buscan evitar o eliminar la no linealidad del sistema, pero normalmente a expensas de modificar radicalmente el sistema original. Si el sistema físico puede ser descrito satisfactoriamente por un modelo matemático no lineal, entonces explorando el espacio de parámetros del modelo , analítica o numéricamente, es posible conocer cómo puede evitarse el caos ajustando sólo unos pocos parámetros clave.

2.4 Inteligencia artificial

Desde hace muchos años, el hombre ha tratado de entender el funcionamiento del cerebro, el cual hoy en día se podría catalogar como el objeto con el funcionamiento más complejo en el universo entero.

El área de estudio llamada inteligencia artificial, nace a raíz de la curiosidad del humano por entender e imitar el funcionamiento de la razón. Responder preguntas esenciales como por ejemplo: ¿Cómo pensamos? ¿Cómo aprendemos? han despertado el interés de miles de científicos las últimas décadas, desembocando en el desarrollo de áreas de investigación como el aprendizaje de máquina, computación evolutiva, lógica difusa, entre otras.

Inteligencia artificial es el proceso de crear máquinas que puedan actuar de tal manera que sea considerada para un humano como inteligente. Se puede llegar a este objetivo exhibiendo características humanas, o comportamientos más sencillos como la habilidad de sobrevivir en un ambiente dinámico. Otra forma de definir la inteligencia artificial es como el campo de estudio de sistemas que actúan de forma tal que para el observador podría aparentar como inteligente.

El término Inteligencia Artificial fue usado por primera vez por John McCarthy en una conferencia en el *Dartmouth College,* en *New Hampshire*. El matemático Alan Turing diseñó una prueba llamada la prueba de Turing como una manera de juzgar el éxito o fracaso de un intento por producir inteligencia artificial. En resumen, la prueba consiste de tres personajes, el personaje A es quien interroga a los personajes B y C. El personaje B es un hombre y el personaje C es una máquina. El personaje A sólo puede comunicarse con los personajes B y C a través de un monitor y estos deberán intentar engañar al personaje A para que, al finalizar el interrogatorio, no pueda decidir quién es el hombre y quién es la máquina. Hasta la fecha ningún programa ha logrado pasar la prueba de Turing.

Algunas de las aplicaciones más relevantes de la inteligencia artificial hoy en día, incluyen:

- Reconocimiento del habla: Algunos sistemas de reconocimiento del habla requieren de algún tipo de entrenamiento donde individuos leen textos o vocablos aislados al sistema. Algunos de los modelos y técnicas de AI (Inteligencia Artificial) para reconocimiento del habla incluyen el modelo oculto de Markov, redes neuronales recurrentes, redes neuronales profundas, entre otras...
- Diseño de producto: Se han utilizado algoritmos genéticos para encontrar soluciones óptimas al diseño de productos, por ejemplo la NASA utilizó un algoritmo genético para encontrar el diseño óptimo de una antena.
- Visión artificial: Los sistemas de reconocimiento de imágenes utilizan en su mayoría modelos de redes neuronales y técnicas de aprendizaje profundo para reconocer objetos dentro de la imagen. Normalmente se aplican técnicas de transformaciones lineales para el pre procesamiento de las imágenes.
- Robótica: Uno de los problemas más grandes de la inteligencia artificial es el de la percepción y razonamiento lógico. En robótica se busca desarrollar robots capaces de percibir el ambiente en el que se encuentran y así tomar decisiones certeras. Por ejemplo un dron que identifica un objetivo y lo sigue, necesita percibir el entorno que lo rodea para esquivar objetos que se encuentren en su camino.
- Minería de datos: Uno de los grandes retos que tienen hoy en día grandes empresas tecnológicas, es encontrarle significado a la enorme cantidad de datos que obtienen día con día, por ejemplo solamente en Facebook cada minuto

ocurre lo siguiente: 510,000 nuevos comentarios, 293,000 actualizaciones de estado y 136,000 cargas de fotos. Encontrar significado a todos estos datos se da a partir del diseño de modelos y técnicas de minería de datos.

- Sistemas de recomendación: Grandes compañías como Amazon, Google, Spotify, etc… cuentan con sistemas de recomendación "inteligentes" que aprenden de los gustos de cada usuario para hacer sugerencias con base en los datos previamente buscados por el mismo. Estos sistemas de recomendación pueden ser sumamente complejos y se utilizan técnicas de aprendizaje profundo para procesar los grandes volúmenes de datos almacenados de los históricos de búsquedas y compras.
- Predicción de precios: Hoy en día más del 70% del volumen total de operaciones bursátiles en las bolsas de valores en el mundo se realizan de manera automatizada, esto gracias a la predictibilidad de los modelos obtenidos mediante técnicas de inteligencia artificial y aprendizaje de máquina. En términos generales se utiliza una red neuronal que se entrena con los datos históricos de los precios de las acciones para predecir el valor futuro de las mismas y con base en eso tomar decisiones de compra o venta.

Aprendizaje de máquina

El aprendizaje de máquina es en sí una forma más específica de aproximarse a la inteligencia artificial. En términos generales, es el conjunto de prácticas que utilizan algoritmos para procesar datos, aprender de ellos y finalmente obtener un modelo o una predicción. Por lo tanto, en vez de programar una rutina de software con una serie específica de instrucciones para realizar cierta tarea, la máquina se "entrena" utilizando grandes volúmenes de información y algoritmos que le dan la habilidad de aprender cómo realizar la tarea. Por ejemplo si quisiéramos desarrollar un sistemas para clasificar formas y agruparlas (por ejemplo triángulos, cuadrados y círculos), teniendo como entrada del sistema una forma cualquiera y como salida la decisión del grupo al que pertenece, podríamos hacerlo de dos formas:

1. Desarrollar un programa definiendo reglas y condiciones para que, mediante una serie de pasos lógicos, sea posible determinar si el objeto de entrada es un círculo, un cuadrado o un triángulo.
2. Extraer las características esenciales de cada grupo creando un conjunto de datos para entrenar una red neuronal. Con base en a las características de una

nueva forma, la red previamente entrenada con el conjunto de características, decidirá a qué grupo pertenece: triángulos, rectángulos o círculos.

El término "aprendizaje de máquina" fue acuñado en 1959 por Arthur Samuel. Evolucionó a partir del estudio de reconocimiento de patrones y la teoría de aprendizaje computacional. El aprendizaje de máquina explora el estudio y desarrollo de algoritmos que puedan aprender y hacer predicciones a partir de datos.

2.5 Algoritmos genéticos

Los algoritmos genéticos están inspirados en el concepto de la selección natural de Darwin para la discriminación de soluciones de una función dada. De forma general, el objetivo de utilizar un algoritmo genético es encontrar una solución óptima a un problema, definiendo una función de aptitud con la cual serán evaluadas las diversas soluciones. Para encontrar la solución óptima se itera a través de múltiples generaciones que estarán constituidas por varios "individuos", mismos que hacen referencia a la evaluación de la función que se quiere optimizar con distintos parámetros. Los individuos de la generación son evaluados con base en la función de aptitud, las soluciones con mejor desempeño serán seleccionadas para recombinarse y mutarse para producir soluciones hijo, que a su vez volverán a ser evaluadas en un proceso iterativo hasta encontrar la mejor solución posible.

Los algoritmos genéticos fueron inventados por John Holland en los años 60s y fueron desarrollados por Holland y sus estudiantes y colegas en la universidad de Michigan entre los años 1960s y 1970s. En contraste con las estrategias de evolución y la programación evolutiva, la meta principal de Holland no era diseñar un algoritmo para resolver problemas específicos, sino estudiar formalmente el fenómeno de adaptación como sucede en la naturaleza y así desarrollar formas en las que los mecanismos de adaptación natural puedan ser representados en sistemas computacionales. Holland presentó los algoritmos genéticos en su libro sobre los sistemas naturales y artificiales como una abstracción de la evolución biológica y dio un marco teórico para la adaptación.

Terminología:

- Individuo: Cualquier posible solución. También se usa el término cromosoma para referirse a cualquier solución dentro de una población.
- Población: Un grupo de individuos los cuales serán evaluados entre sí para identificar a los más aptos y seleccionarlos para la fase de reproducción.
- Desempeño: Valor que se le asigna al individuo al ser evaluado como propuesta de solución al problema. Los individuos con muy bajo desempeño son discriminados y no producen descendientes.
- Genotipo: La representación codificada de un individuo.
- Fenotipo: Es la representación decodificada del individuo, por ejemplo una red neuronal en sí es el fenotipo, mientras una representación binaria de los nodos y conexiones es su genotipo.

Los algoritmos genéticos se pueden utilizar para encontrar la solución al problema del vendedor viajero el cual plantea el siguiente caso: Dada una lista de ciudades y la distancia entre ellas, se necesita encontrar la ruta más corta posible para que un vendedor visite cada ciudad una vez y al finalizar regrese a la ciudad de origen.

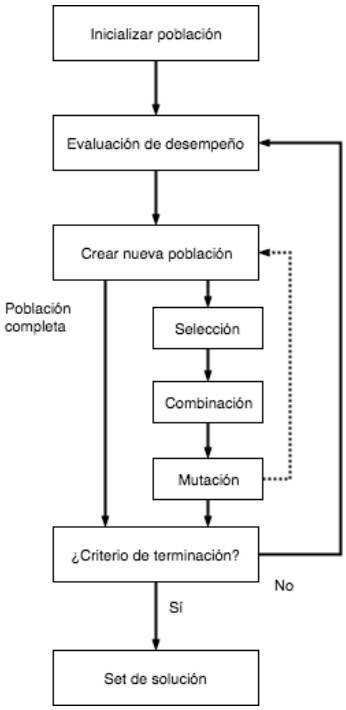


Fig 2B. Diagrama de flujo de un algoritmo genético básico.

Operadores genéticos

La forma más simple de algoritmos genéticos involucran tres tipos de operadores: selección, combinación y mutación.

- Selección: Este operador selecciona el cromosoma en la población para la reproducción. Entre más apto sea el cromosoma, es más probable que sea seleccionado más veces para su reproducción.
- Combinación: Este operador selecciona de manera aleatoria un bit e intercambia subsecuencias antes y después del lugar entre dos cromosomas para crear dos hijos. Por ejemplo, las cadenas de caracteres 10000100 y 11111111 podrían ser combinadas después del lugar en cada uno para producir los hijos 10011111 y 11100100. El operador de combinación imita de manera muy general la combinación entre dos organismos de cromosomas simples (haploide).
- Mutación: Este operador modifica de manera aleatoria algún bit en el cromosoma con el objetivo de asegurar la diversidad de individuos en la generación. Por ejemplo, la cadena 00000100 podría ser mutada en su segunda posición para producir 01000100. La mutación puede ocurrir en cada posición de bit según una probabilidad definida, normalmente muy reducida (ej. 0.001).

Implementación

Los pasos para implementar un algoritmo genético simple son los siguientes:

1. Generar una población aleatoria de n cantidad de cromosomas o individuos (soluciones candidato).
2. Calcular el desempeño del cada cromosoma x en la población.
3. Repetir lo siguiente hasta que n descendientes han sido calculados:
    a. Selección de un par de cromosomas, la probabilidad de la selección es una función de su desempeño. El mismo cromosoma puede ser seleccionado más de una vez para ser padre.
    b. Con la probabilidad de combinación (pc), se combina el par de cromosomas en un punto aleatorio, si no hay combinación los descendientes serán copias exactas de alguno o ambos padres.

c. Mutar los descendientes dada una probabilidad de mutación (pm), y colocar los cromosomas resultantes en la nueva población.
4. Reemplazar la población actual con la nueva.
5. Repetir el paso 2.

2.6 Redes neuronales artificiales

Las redes neuronales son modelos simplificados que tratan de imitar el comportamiento de una red neuronal biológica que se puede encontrar en nuestro sistema nervioso.

El cerebro humano es el objeto más complejo que se ha podido encontrar en el universo, cuenta con más de cien millones de neuronas conectadas entre sí a través de más de cien millones de conexiones.

Una neurona es una célula que ha creado una función de relación con otras células del mismo tipo, excitadas mediante la conducción electroquímica. Las neuronas biológicas constan de tres partes esenciales: El axón, las dendritas y el cuerpo celular o soma.

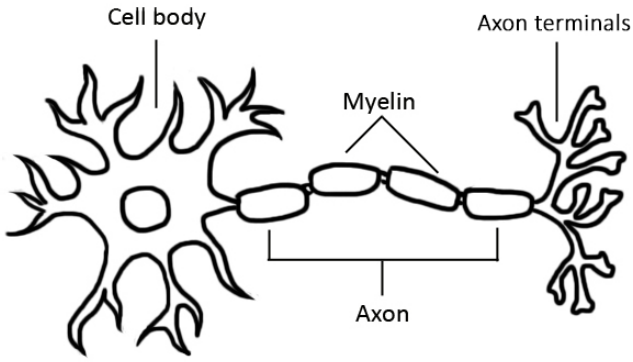


Fig 2C. Neurona biológica.

En el cuerpo celular se encuentra el núcleo, en donde se realizan acciones bioquímicas necesarias para la supervivencia de la célula.

Las dendritas son ramificaciones del cuerpo celular que permiten la interacción con otras neuronas, el número de comunicaciones de una neurona por medio de sus dendritas, se encuentra en un rango de cientos hasta decenas de miles.

El axón es el canal de comunicación de una neurona con el resto, es el medio por el cual se transfiere información de una neurona a otra. La comunicación entre las neuronas es de un sólo sentido, esto quiere decir que las señales se reciben por las dendritas y se propagan por el axón hasta llegar a las próximas neuronas dentro de la red. La combinación de señales de entrada dará como resultado la excitación, y por ende, el valor de salida de la neurona. A la interacción entre el axón de una neurona y las dendritas de otra se le denomina sinapsis.

Una neurona artificial pretende imitar el comportamiento de una neurona biológica. Tomando una serie de entradas a las cuales se les pondera mediante una función de red la cual es normalmente una sumatoria de los valores de entrada multiplicados por los pesos, una función de activación la cual define el umbral de activación para la siguiente capa, el sesgo que es el valor de corrección de la neurona y finalmente la salida que es el valor que recibirá la siguiente capa de neuronas o la salida misma de la red.

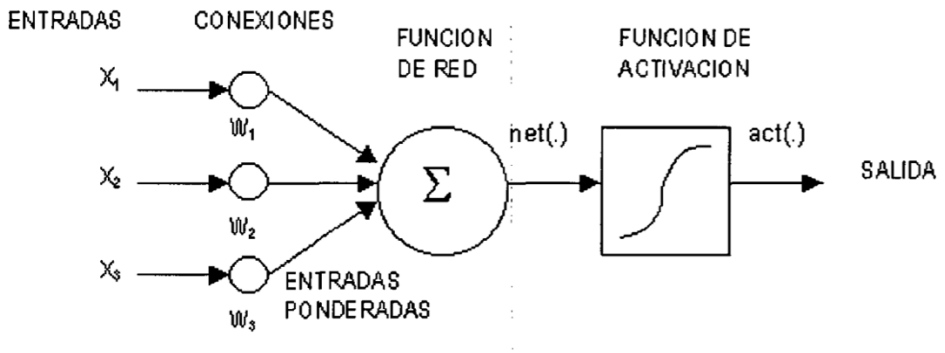


Fig 2D. Modelo de una neurona artificial.

Normalmente los nodos de entrada y salida se establecen dependiendo del tipo de tarea para la cual se pretende diseñar la red. Por ejemplo, consideremos una red que se usa para controlar un robot artificial que busca comida. Los nodos de entrada podrían ser asignados por diferentes sensores del robot, los nodos de salida podrían ser asignados a acciones que puede hacer el robot como comer, moverse en cierta dirección o quedarse quieto.

Los componentes de una red neuronal artificial son los siguientes:

- Neuronas: Usualmente llamados "nodos", cuenta con los elementos mencionados anteriormente, éstas a su vez forman capas por las cuales se propagan las señales de entrada para producir una salida.
- Conexiones y pesos: La red consiste de conexiones entre sus neuronas, cada salida de las neuronas en las capas ocultas está conectada a la entrada de las neuronas de la siguiente capa. A cada una de estas conexiones le corresponde un peso por el cual se multiplicará el valor de la salida. La combinación entre pesos, entradas y activación es lo que resultará en la salida de la neurona.
- Función de activación: Es una función que se usa para trasportar valores a través de las capas de las neuronas de la red. Usualmente, las entradas se suman para post Regla de aprendizaje: Es la regla del algoritmo utilizado para modificar los parámetros de la red neuronal con el fin de producir una salida deseada para una entrada dada. El proceso de aprendizaje normalmente se enfoca en modificar los pesos y umbrales de las variables dentro de la red, para posteriormente ser evaluadas por una función de activación.

En una red neuronal simple, los nodos de salida y los nodos ocultos tienen conexiones que los enlazan con otros nodos, todos estos enlaces tienen un peso específico, esto quiere decir que la conexión entre los nodos será más fuerte o más débil dependiendo del valor del peso de la conexión. A mayor peso la neurona previa tiene más influencia sobre la activación de la neurona sucesora.

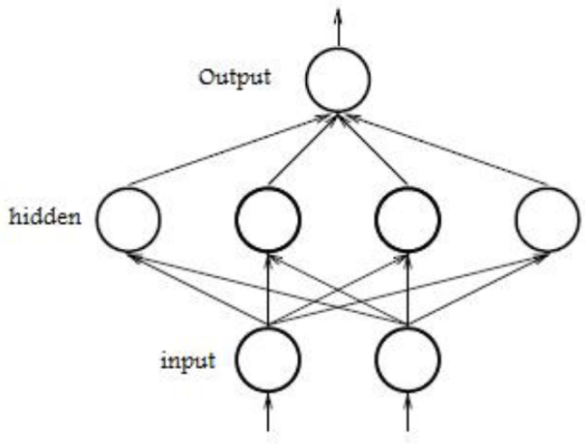


Fig 2E. Ejemplo de una red neuronal simple con 2 entradas, 1 salida y 1 capa oculta.

El valor de los nodos ocultos o nodos de salida se calcula por una función específica que toma como entrada la suma multiplicada por el peso de todos los nodos conectados con ese nodo específico, esta función se llama *función de activación*. Al entrenar cada peso de la red, ésta puede devolver la salida deseada dados ciertos valores de entrada.

Hay múltiples características que describen una red neuronal artificial. El opuesto a una red tipo *feedforward* es una recurrente, las redes recurrentes pueden tener ciclos de conexiones y realimentar las entradas de algunas capas con las salidas de otras.

El concepto de las redes neuronales artificiales nace de la idea de imitar el comportamiento de las neuronas bilógicas y ha estado en el aire desde 1940 con el trabajo de Warren McCulloch y Walter Pits, quienes pudieron demostrar que redes de neuronas artificiales podían procesar cualquier función aritmética o lógica. Luego de ellos surgieron varios investigadores quienes realizaron grandes aportes al área como Donald Hebb, Frank Rosenblatt, Bernard Widrow, John Hopfield, por mencionar algunos.

Aprendizaje

Existen tres principales paradigmas de aprendizaje:

Supervisado

El aprendizaje supervisado utiliza una serie de valores de entradas y salidas, de los cuales, para cada valor de entrada corresponde una salida específica. Este tipo de aprendizaje desea inferir el mapeo de los datos. Se utiliza una función de error (normalmente *mean-square error*) para tratar de minimizar el error entre las salidas de la red y el valor deseado de cada uno de los pares del set de entrenamiento. Minimizar el error usando gradiente descendiente para una red de perceptrón multicapa (MLP) da como resultado el algoritmo de entrenamiento de propagación hacia atrás. Este tipo de aprendizaje sirve para resolver problemas de reconocimiento de patrones (clasificación) y regresión (aproximación de función).

No Supervisado

El aprendizaje no supervisado se utiliza cuando tenemos datos que no están clasificados o etiquetados y se pretende extraer información útil mediante la clasificación o agrupación de la información. El agrupamiento o *"clustering"* es una de

las formas más utilizadas de aprendizaje no supervisado y su objetivo es encontrar grupos de instancias tales que las instancias de un clúster tengan características similares entre sí, y diferentes de las instancias de otros clúster. Por ejemplo si contamos con un arreglo de datos con la información de estatura y peso de mil personas, probablemente al graficar su peso y estatura encontraremos dos principales clúster los cuales nos indicaría el género de la persona basado en esas 2 variables.

Por refuerzo

El aprendizaje por refuerzo se utiliza para maximizar una recompensa numérica obtenida por algún objeto, comúnmente denominado agente, que aprende a navegar un ambiente desconocido. El objetivo principal es que el agente explore un espacio desconocido de estados, obteniendo puntos cada vez que el agente toma una decisión acertada y perdiéndolos cada vez que tome una decisión errónea. Se utiliza el proceso de decisión de Markov, el cual provee un marco matemático para modelar la toma de decisiones en situaciones donde los resultados son en parte aleatorios y en parte bajo control de quien toma las decisiones.

Capítulo 3. **NEAT**

Este capítulo describe la historia, aplicaciones, funcionamiento básico y ventajas del algoritmo de Neuroevolución por Topologías Aumentadas, el cual propone, en términos generales, un proceso de evolución de poblaciones de redes neuronales para llegar a un comportamiento deseado del sistema.

3.1 Generalidades

El algoritmo NEAT (Neuroevolución por Topologías Aumentadas) fue desarrollado por Kenneth O. Stanley en el año 2002 y propone, en términos generales, la implementación de un algoritmo genético para evolucionar poblaciones de redes neuronales y llegar así a una solución óptima.

El objetivo principal del algoritmo de Neuroevolución por Topologías Aumentadas (NEAT) es utilizar un algoritmo genético que busque la solución óptima de una red neuronal modificando tanto sus pesos como su topología y tiene las siguientes propiedades:

1. Existe una representación genética de la red que permite que la combinación de las estructuras sea de una manera significativa.
2. Protege la innovación topológica que necesita evolucionar algunas generaciones para ser optimizada y así no desaparezca del pool de genomas prematuramente.
3. Minimizar topologías mediante entrenamiento y funciones de penalización.

Terminología

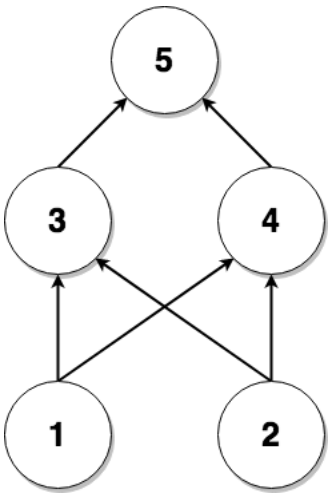
Gen: Cada individuo o red consta de dos tipos de genes: genes de nodos y genes de conexiones
Genoma: Individuo de la población codificado para que a partir de él se cree una red neuronal y pueda ser evaluada
Fenotipo: Representación "real" de un genoma, la red neuronal que le corresponde al genoma decodificado.
Población: Hace referencia a todos los individuos de una generación.
Marcadores históricos: Permiten saber el origen de un gen y ayuda a resolver el problema de convenciones competidoras.

3.2 Codificación

Existen diversos métodos para codificar y representar la topología de una red neuronal artificial, en el caso del algoritmo NEAT, Stanley decidió utilizar la codificación binaria o

directa ya que se necesita seguir el rastro de la evolución de la topología de una red, cosa que con codificación indirecta no se podría dado que es posible representar la misma red de múltiples maneras (n!), por lo tanto al momento de combinar los individuos habría pérdida de información de la topología.

El método de codificación directa para redes neuronales es ilustrada en el trabajo de Geoffrey Miller, Peter Tod, and Shilesh Hedge (1989), quienes restringieron su proyecto inicial con redes tipo prealimentada (*feedforward*) y un número fijo de neuronas con la cual el algoritmo genético debía de evolucionar la topología de las conexiones.



	1	**2**	**3**	**4**	**5**
1	0	0	0	0	0
2	0	0	0	0	0
3	L	L	0	0	0
4	L	L	0	0	0
5	0	0	L	L	0

Cromosoma: 000000000011000011000001l0

Fig 3A. Ejemplo de una matriz de codificación directa.

En codificación o representación directa el número de nodos, conexiones y pesos de un genotipo están expresados de manera literal en la representación. Por ejemplo la codificación en NEAT de la red neuronal solución al problema de OR exclusivo con el mínimo número de neuronas es, en código de Python:

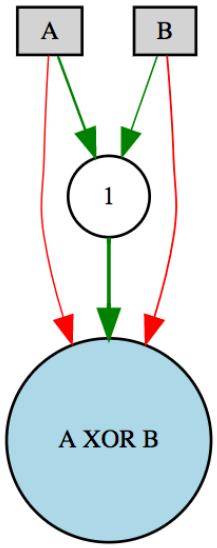


Fig 3B. Fenotipo de la solución al problema XOR obtenido mediante NEAT.

```
Genome(
    nodes=[
        { "key" : 0, "bias" : b0, "activation" : "sigmoid"} # Output
        { "key" : 1, "bias" : b1, "activation" : "sigmoid"} # Hidden neuron
    ],
    links=[
        { "key" : (-2,0), "weight" : w0, "enabled" : True }
        { "key" : (-2,1), "weight" : w1, "enabled" : True }
        { "key" : (-1,0), "weight" : w2, "enabled" : True }
        { "key" : (-1,1), "weight" : w3, "enabled" : True }
        { "key" : (1,0), "weight" : w4, "enabled" : True }
    ]
)
```

Fig 3C. Representación en Python3 del Fenotipo de la solución al problema XOR. En esta representación, las entradas son valores enteros negativos, desde -1 hasta -infinito. No son considerados dentro de los nodos de la clase de genotipo porque no tienen sesgo ni activación).

Genes de Nodos

{ key=0, bias=-1.43, resp=1.0, activation=sigmoid } - Neurona de salida

{ key=1, bias=-2.3, resp=1.0, activation=sigmoid } - Neurona de la capa oculta

(representa dos neuronas con sesgo de -1.43 y -2.3, ambas con función de activación sigmoidal)

Genes de Conexiones

{ key=(-2,0), weight=2.5, enabled=True } - Neurona de input B a salida

{ key=(-2,1), weight=-2.31, enabled=True } - Neurona de input A a salida

{ key=(-1,0), weight=-2.67, enabled=True } - Neurona de input B a salida

{ key=(-1,1), weight=4.04, enabled=True } - Neurona de input B a salida

{ key=(1,0), weight=6.155, enabled=True } - Neurona de input B a salida

3.3 Convenciones competidoras

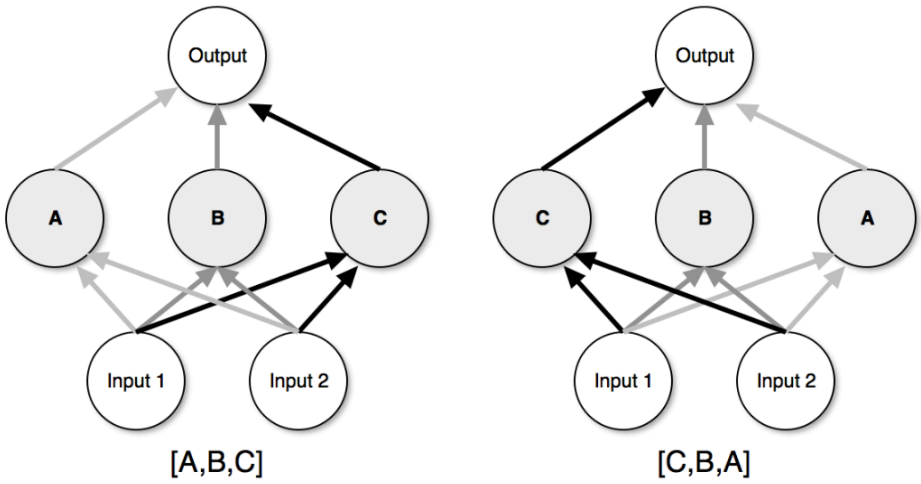


Fig 3D. Dos redes neuronales que son funcionalmente equivalentes pueden ser codificadas de maneras distintas. Por ejemplo la combinación de las dos redes [A,B,C] x [C,B,A] producen los genotipos [A,B,A] y [C,B,C]. Existe pérdida de un tercio de información en ambos descendientes. Existen n! de formas distintas para representar la misma red, donde n es el número de nodos en las capas ocultas.

Convenciones competidoras significa tener más de una manera para expresar una misma solución a un problema de optimización de pesos con una red neuronal. Cuando los genomas representando la misma solución no tienen la misma codificación, el cruzamiento puede producir descendientes o "hijos" defectuosos. El principal indicador en NEAT es que el origen histórico de dos genes, es evidencia directa de la homología

si los genes comparten el mismo origen. Por lo tanto, NEAT realiza sinapsis artificiales basadas en marcadores históricos, permitiendo cambiar la topología de la red sin perder la pista de cada gen en el curso de una simulación.

- Más de una manera de expresar una misma solución de optimización de pesos.
- NEAT lo resuelve colocando marcadores históricos a los genes, permitiendo cambiar la estructura de la red sin perder la pista de los genes a lo largo de la simulación.

3.4 Mutaciones

Existen tres diferentes tipos de mutaciones, algunas estructurales y otras no estructurales, esto quiere decir que pueden afectar o no la topología de la red.

Mutación de pesos

En la mutación de pesos únicamente se modifica el valor del peso entre las conexiones de la red de manera aleatoria. La probabilidad de que suceda una mutación de pesos dependerá de un valor que asignemos en el archivo de configuración

Mutación de nodos

Siempre que se agrega un nodo en la red, se hace añadiéndolo entre dos nodos conectados, asignando peso de 1 para que no haya tanta afectación al sistema. Por lo tanto cuando se hace una mutación de nodo se agregan 3 genes: un gen de nodo y dos genes de conexión. Tomemos como ejemplo una red simple a la que se le quiere agregar un nuevo nodo. Los genes quedarían de la siguiente manera:

Ejemplo de agregar un nodo entre Nodo0 (output) y NodoC (input):

- Se deshabilita el link entre los nodos 0 y C
- Se agrega el nuevo nodo (Nodo2)
- Se agrega gen de conexión entre NodoC y Nodo2 con peso de 1
- Se agrega gen de conexión entre Nodo2 y Nodo0 con el peso que tiene el gen deshabilitado entre Nodo0 y NodoC

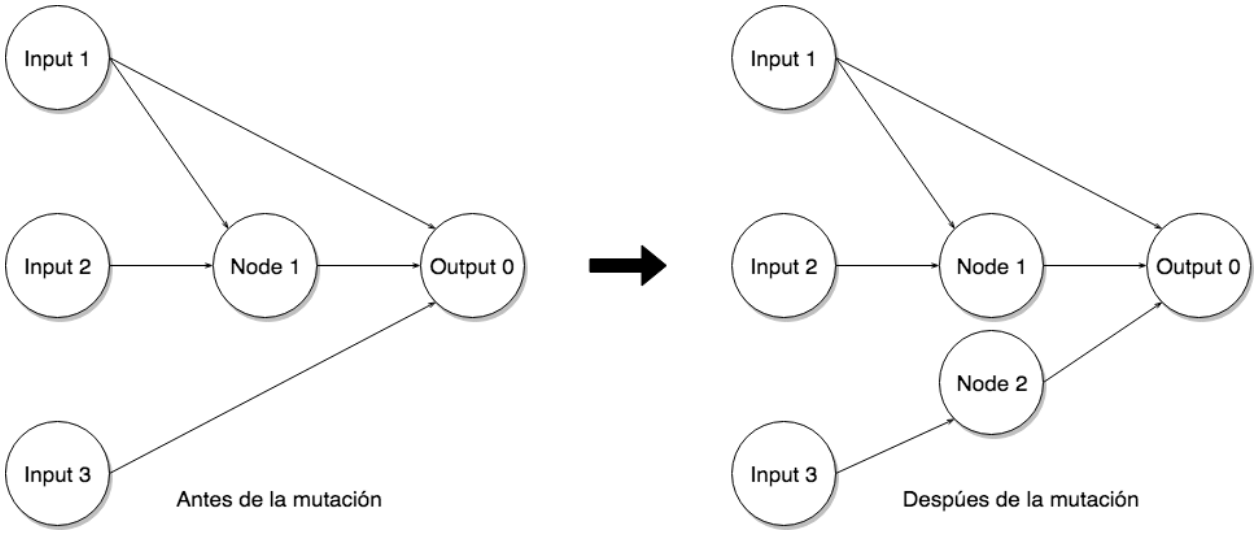


Fig 3E. Ejemplo de mutación estructural de genoma

Genoma antes de la mutación de nodos

```
Genome(
    nodes=[
        { "key" : 0, "bias" : b0, "activation" : "sigmoid"},    # Output
        { "key" : 1, "bias" : b1, "activation" : "sigmoid"}     # Hidden
    ],
    links=[
        { "key" : (-3,0), "weight" : w0, "enabled" : True  },
        { "key" : (-2,1), "weight" : w1, "enabled" : True  },
        { "key" : (-1,1), "weight" : w2, "enabled" : True  },
        { "key" : (-1,0), "weight" : w3, "enabled" : True  },
        { "key" : (1,0), "weight" : w4, "enabled" : True  },
    ]
)
```

Fig 3F. Representación en Python3 de un genoma previo a la mutación aleatoria de sus nodos.

Genoma después de la mutación de nodos

```
Genome(
    nodes=[
        { "key" : 0, "bias" : b0, "activation" : "sigmoid"},# Output
        { "key" : 1, "bias" : b1, "activation" : "sigmoid"} # Hidden
        { "key" : 2, "bias" : b2, "activation" : "sigmoid"} # New Hidden
    ],
    links=[
        { "key" : (-3,0), "weight" : w0, "enabled" : False}, # Disabled
        { "key" : (-2,1), "weight" : w1, "enabled" : True},
        { "key" : (-1,1), "weight" : w2, "enabled" : True},
        { "key" : (-1,0), "weight" : w3, "enabled" : True},
        { "key" : (1,0), "weight" : w4, "enabled" : True},
        { "key" : (-3,2), "weight" : 1, "enabled" : True},# Peso de 1
inno+1
        { "key" : (2,0), "weight" : w0, "enabled" : True  } # Peso de C a 0
innov+2
    ]
)
```

Fig 3G. Representación en Python3 de la mutación de nodos en una red. En este caso se añadió un nodo y se puso peso de 1 a uno de sus enlaces.

Mutación de enlaces

La mutación de enlaces se lleva acabo simplemente creado o quitando enlaces entre los nodos de los genomas.

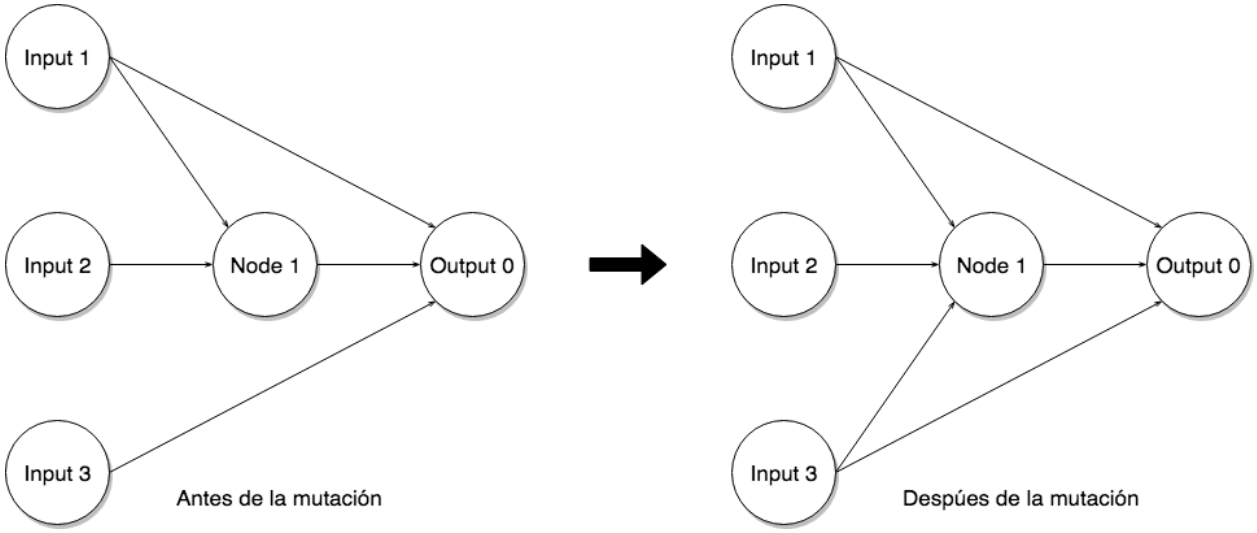


Fig 3H. Ejemplo de mutación de enlaces de genoma.

Genoma antes de la mutación de enlace

```
Genome(
    nodes=[
        { "key" : 0, "bias" : b0, "activation" : "sigmoid"},    # Output
        { "key" : 1, "bias" : b1, "activation" : "sigmoid"}     # Hidden
    ],
    links=[
        { "key" : (-3,0), "weight" : w0, "enabled" : True  },
        { "key" : (-2,1), "weight" : w1, "enabled" : True  },
        { "key" : (-1,1), "weight" : w2, "enabled" : True  },
        { "key" : (-1,0), "weight" : w3, "enabled" : True  },
        { "key" : (1,0), "weight" : w4, "enabled" : True  },
    ]
)
```

Fig 3I. Representación en Python3 de un Fenotipo antes de una mutación de enlaces.

Genoma después de la mutación de enlace

```
Genome(
    nodes=[
        { "key" : 0, "bias" : b0, "activation" : "sigmoid"},    # Output
        { "key" : 1, "bias" : b1, "activation" : "sigmoid"}     # Hidden
    ],
    links=[
        { "key" : (-3,0), "weight" : w0, "enabled" : True  },
        { "key" : (-3,1), "weight" : w5, "enabled" : True  },
        { "key" : (-2,1), "weight" : w1, "enabled" : True  },
        { "key" : (-1,1), "weight" : w2, "enabled" : True  },
        { "key" : (-1,0), "weight" : w3, "enabled" : True  },
        { "key" : (1,0), "weight" : w4, "enabled" : True  },
    ]
)
```

Fig 3J. Representación en Python3 de un Fenotipo después de una mutación en los pesos de sus enlaces.

3.5 Reproducción

Dos genes con el mismo origen histórico deben representar la misma estructura (aunque posiblemente con pesos distintos), pues los dos son derivados del mismo gen ancestro de algún punto en el pasado. Por lo tanto la necesidad de alinear los genes entre ellos es para seguir la pista del origen histórico de cada gen del sistema.
Rastrear el origen histórico requiere muy poca computación. Cuando un nuevo gen aparece (por medio de mutación estructural), un *global innovation number* se incrementa y se asigna a ese gen. El *innovation number* por lo tanto representa una cronología de las apariciones de cada gen en el sistema

- Cuando se agrega un nuevo gen al genoma por mutación estructural (*node/link mutation*), se incrementa un *global innovation number* y se le asigna al nuevo gen creado.
  Un problema que se puede presentar es que la misma innovación estructural reciba diferentes *innovation numbers* en la misma generación

Para hacer la recombinación entre los genomas, éstos se alinean conforme el *innovation number* de los genes y considerando únicamente los genes de las conexiones. Por ejemplo si un genotipo tiene un *innovation number* y otro no, se considera una desarticulación (*disjoint*) si está dentro del rango del *innovation number* de ambos, si no está dentro del rango del *innovation number* de alguno, éste se considera un exceso (*excess)*.

Cuando se conforma el genoma hijo, los genes son seleccionados aleatoriamente de cualquier padre en los genes alineados, y los genes desarticulados o en exceso siempre son incluidos del padre más apto (con mejor desempeño).

- Los genes que están alineados se heredan de manera aleatoria.
- Los genes no alineados (desarticulados o en exceso) son heredados del padre más apto (todos). Si el desempeño de ambos es igual se realiza de manera aleatoria.

Ejemplo de reproducción entre genomas:

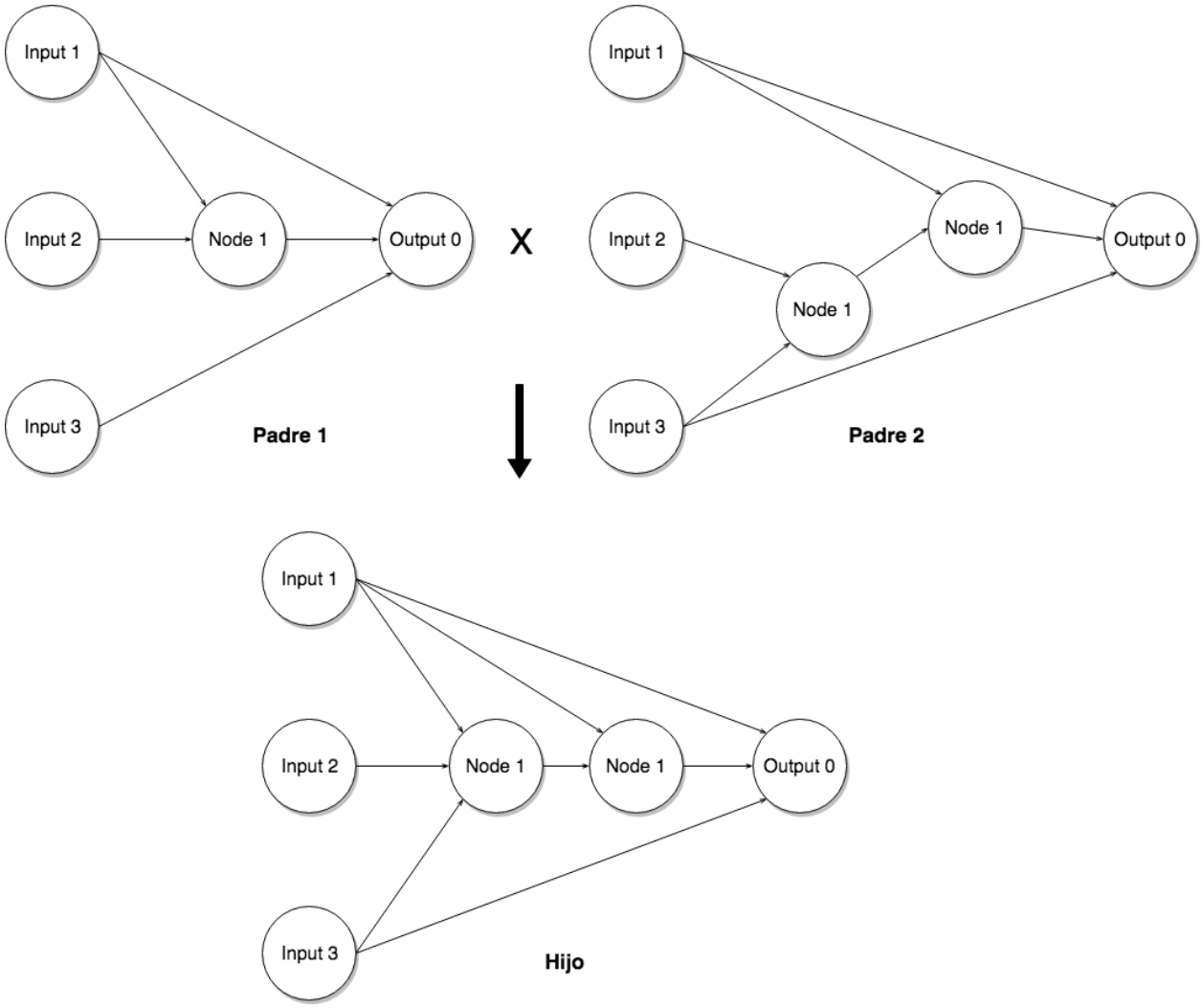


Fig 3K. Reproducción de 2 genomas

Representación de los genomas de los padres y el hijo resultante:

```
# Padre 1
parent1 = Genome(
    nodes=[
        { "key" : 0, "bias" : b0, "activation" : "sigmoid"},
        { "key" : 1, "bias" : b1, "activation" : "sigmoid"}
    ],
    links=[
        { "key" : (-3,0), "weight" : w10, "enabled" : True  },
        { "key" : (-2,1), "weight" : w11, "enabled" : True  },
        { "key" : (-1,1), "weight" : w12, "enabled" : True  },
        { "key" : (-1,0), "weight" : w13, "enabled" : True  },
        { "key" : (1,0),  "weight" : w14, "enabled" : True  },
    ]
)
# Padre 2
parent2 = Genome(
    nodes=[
        { "key" : 0, "bias" : b0, "activation" : "sigmoid"},
        { "key" : 1, "bias" : b1, "activation" : "sigmoid"},
        { "key" : 2, "bias" : b2, "activation" : "sigmoid"}
    ],
    links=[
        { "key" : (-3,1), "weight" : w20, "enabled" : True  },
        { "key" : (-3,0), "weight" : w21, "enabled" : True  },
        { "key" : (-2,1), "weight" : w22, "enabled" : True  },
        { "key" : (-1,2), "weight" : w23, "enabled" : True  },
        { "key" : (-1,0), "weight" : w24, "enabled" : True  },
        { "key" : (1,2),  "weight" : w25, "enabled" : True  },
        { "key" : (2,0),  "weight" : w26, "enabled" : True  }
    ]
)
```

Fig 3L. Representación en Python3 de dos Genomas previos a la combinación para producir un hijo.

Resultado:

```
# Hijo
offspring = Genome(
    nodes=[
        { "key" : 0, "bias" : b0, "activation" : "sigmoid"},   # Random
        { "key" : 1, "bias" : b1, "activation" : "sigmoid"},   # Random
        { "key" : 2, "bias" : b2, "activation" : "sigmoid"}    # Random
    ],
    links=[
        { "key" : (-3,1), "weight" : w20, "enabled" : True  }, # Parent 2
        { "key" : (-3,0), "weight" : w21, "enabled" : True  }, # Parent 2
        { "key" : (-2,1), "weight" : w11, "enabled" : True  }, # Parent 1
        { "key" : (-1,2), "weight" : w23, "enabled" : True  }, # Parent 2
        { "key" : (-1,1), "weight" : w12, "enabled" : True  }, # Parent 1
        { "key" : (-1,0), "weight" : w13, "enabled" : True  }, # Parent 1
        { "key" : (1,2),  "weight" : w25, "enabled" : True  }, # Parent 2
        { "key" : (2,0),  "weight" : w20, "enabled" : True  }  # Parent 2
    ]
)
```

Fig 3M. Representación en Python3 del genoma resultante de la combinación de los dos genomas previamente vistos en la figura Y

3.6 Especiación

Frecuentemente, agregar nodos en una red hace que inicialmente baje su desempeño antes de permitir que sus pesos se optimicen. Por ejemplo aumentar la estructura introduce no linealidades en donde no las había antes. Es improbable que un nuevo nodo o conexión expresen una buena respuesta tan pronto como son introducidos a la red. Desafortunadamente, como inicialmente hay una pérdida de desempeño del genoma, la innovación es improbable que sobreviva en una población que ya se ha optimizado previamente. Por tal motivo, es necesario proteger de alguna manera las innovaciones estructurales de la red para que puedan hacer uso y optimizar esa nueva estructura.

En la naturaleza, diferentes estructuras tienden a pertenecer a diferentes grupos o especies que compiten en diferentes nichos. Por lo tanto, innovación es implícitamente protegida dentro del nicho. De manera similar, si las redes con estructuras novedosas

pudieran estar aisladas en su propia especie, tendrían oportunidad de optimizar sus estructuras antes de tener que competir con el grueso de población.

La especiación requiere de una función de compatibilidad que indique si dos genomas deberían estar en la misma especie o no. Es difícil formular esta función de compatibilidad entre redes de distintas topologías.

Como NEAT guarda información histórica del gen, la población en NEAT puede ser fácilmente separada en especies. Se usa la división explícita de desempeño (*explicit fitness sharing),* la cual obliga individuos con genomas similares a compartir su desempeño.

Crear especies dentro de la población permite que los organismos compitan primordialmente con los de su mismo nicho en vez de competir con el grueso de la población. La idea es dividir la población en especies, de tal forma que las topologías similares queden en la misma especie. Parecería que esta acción corresponde enteramente a un pareo de topologías, sin embargo los marcadores históricos proveen una solución eficiente al problema. Entre más desarticulados sean dos genomas, menos historia de evolución comparten y por lo tanto son menos compatibles entre sí. Podemos medir la distancia de compatibilidad de distintas estructuras como una simple combinación lineal del número de genes de excesos y desarticulaciones , así como el promedio de la diferencia entre pesos de los genes pareados (alineados), incluyendo los genes deshabilitados.

Distancia de Compatibilidad

```
d = c1*(E/N) + c2*(D/N) + c3 * avg(W)
```

Donde:

d: distancia de compatibilidad
E : número de genes de exceso
D : número de genes desarticulados
W : pesos de la red
ci : coeficientes para ajustar la importancia de los tres factores
N : número de genes en el genoma más grande

La distancia de compatibilidad nos permite comparar contra un umbral de compatibilidad definido por nosotros. En cada generación, los genomas son secuencialmente incluidos en las especies. Cada especie tiene un organismo representante que es seleccionado de manera aleatoria de la generación anterior.
Un genoma g en la generación en curso es colocado en la primer especie con la cual sea compatible, midiendo la distancia entre él y cada representante de esa especie. De esta manera las especies no se traslapan. Si g no es compatible con ningún representante de las especies, una nueva especie es creada

División de desempeño (*Fitness sharing*):

```
adjusted_fitness_j = fitness_j / sum( sh(d(i,j)), range(i,n) )

# Otra forma de expresarlo
adjusted_fitness_j = fitness_j / num_organisms_same_species_of_j
```

Donde:
j : número de genoma específico.
i : número total de neuronas.
d : función que regresa la distancia de compatibilidad entre i y j.
sh : función de división de desempeño (*sharing function*) (regresa 1 o 0).

Función de división de desempeño (*fitness sharing function*)

Si la distancia entre el genoma i y j es mayor que el umbral de compatibilidad, la función regresa 0. Si la distancia es menor que el umbral, la función regresa 1.

- Si hay una especie con 3 individuos, se divide entre 3
- Por ejemplo si hay una especie con un sólo individuo la función de división de desempeño (*fitness sharing function*) sería:

```
adjusted_fitness = fitness / 1
```

Las especies entonces se reproducen primero eliminando los genomas menos aptos de la población. La población entera es reemplazada por los hijos de los organismos remanentes de cada especie.

Algunas consideraciones importantes:

- El número de excesos y desarticulaciones entre un par de genomas es una medida natural para medir su compatibilidad.
- Se toma un representante aleatorio de la especie de la generación previa.
- Para colocar a un organismo en una especie se mide la distancia vs el representante de cada especie, si es compatible se asigna a esa especie.
- Si el genoma a especiar no es compatible con ningún representante, se crea una nueva especie.
- El fitness sharing entre individuos de la misma especie tiene como propósito penalizar a las especies con muchos individos, así una sola especie no toma todo el control.
- Función de division de desempeño (*fitness sharing function*): *fitness* = *fitness* / num_organismos_de_la_especie.
- Toda la especiación se reduce a ajustar el desempeño (*fitness*) considerando el número de individuos parecidos.

3.7 Población inicial

En NEAT, a diferencia de otros algoritmos de neuroevolución, establece los individuos de su población inicial con pocas neuronas y las cuales están interconectadas entre sí, sin capas de neuronas ocultas.

Una manera de minimizar las redes, es incorporando el tamaño de la red a la función de desempeño. En tales casos, el desempeño será penalizado para las redes más grandes.

La similaridad puede ser fácilmente medida basado en información histórica en los genes. Una alternativa para no tener que modificar la función de desempeño, es que el mismo método de neuroevolución tienda a la minimalidad. Si la población empieza sin nodos ocultos y la estructura crece sólo conforme beneficie a la solución, no hay necesidad de modificar la función de desempeño. Por lo tanto, empezar con una población minimalista e ir creciendo su estructura es un principio de diseño en NEAT.

3.8 Parámetros de configuración

Los parámetros de configuración que se pueden variar en el algoritmo NEAT y en específico de la implementación del neurocontrolador Neuropy para obtener una convergencia más rápida y simplemente mejores resultados son:

- Número máximo de generaciones: <generations:int>
- Umbral de desempeño: <fitness_threshold:int>
- Criterio de desempeño: <fitness_criterion:str>
- Tamaño de población: <pop_size:int>
- Evaluaciones por genoma: <evals_per_genome:int>
- Función de activación: <activation_default:str>
- Valor de sesgo max: <bias_max_val:float>
- Valor de sesgo min: <bias_min_val:float>
- Número de neuronas de entrada: <num_inputs:int>
- Número de neuronas de salidas: <num_outputs:int>
- Número de neuronas en capa oculta: <num_hidden:int>
- Valor de pesos max: <weight_max_val:float>
- Valor de pesos min: <weight_min_val:float>
- Umbral de compatibilidad: <compatibility_threshold:str>
- Probabilidad de mutación de pesos: <weight_mutate_rate:float>
- Probabilidad de mutación de sesgo: <bias_mutate_rate:float>
- Probabilidad de agregar nodo: <node_add_prob:float>
- Probabilidad de borrar nodo: <node_delete_prob:float>
- Probabilidad de agregar enlace: <edge_add_prob:float>
- Probabilidad de borrar enlace: <edge_delete_prob:float>
- Probabilidad de activar/desactivar enlace: <enable_mutate_rate:float>
- Umbral de supervivencia: <survival_threshold:float>

Capítulo 4. **Modelo y control**

En este capítulo se obtendrá el modelo matemático con el método de Lagrange que describe el comportamiento del péndulo invertido doble sobre un carro móvil. El modelo matemático es necesario para luego desarrollar el módulo en Python3 y realizar las iteraciones de la simulación sin necesidad de construir el sistema físicamente. También se definen las entradas y la salida de la red, mismas que serán usadas como parámetros de configuración del algoritmo NEAT.

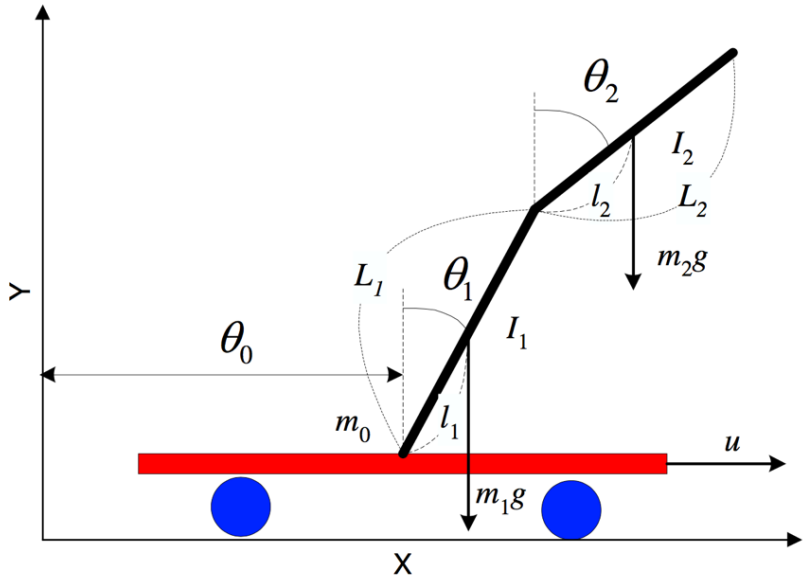


Fig 4A. Diagrama de cuerpo libre del sistema de péndulo invertido doble sobre un carro.

Nomenclatura

X	Posición en el eje X del carro
θ_1	Posición angular del péndulo 1
θ_2	Posición angular del péndulo 2
m_i	Masa del elemento i
I_i	Momento de inercia del eslabón *i*
g	Constante de gravedad [9.81 m/s^2]
l_i	Distancia de la junta al centro de masa del péndulo i
L_i	Longitud del péndulo i
u	Fuerza controlada
L	Lagrangiano
T	Energía cinética del sistema
P	Energía potencial del sistema

Lagrangiano

Se propone el método de Lagrange para obtener el modelo que describe el cambio de energía del sistema con base en las posiciones angulares de los péndulos y la posición horizontal del carro

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \frac{\partial L}{\partial \theta} = Q \qquad \text{... Ec. 1}$$

Donde $L = T - P$ es el Lagrangiano, Q es el vector generalizado de fuerzas (o momentos) actuando en dirección de las coordenadas generalizadas θ.

Las energías cinéticas y potenciales del sistemas están dadas por la suma de las energías de sus componentes individualmente (un carro y dos péndulos).

$$\begin{aligned} T &= T_0 + T_1 + T_2 \\ L &= L_0 + L_1 + L_2 \end{aligned} \qquad \text{... Ec. 2}$$

Energía cinética del carro:

$$T_0 = \frac{1}{2} m_0 \dot{X}^2 \qquad \text{... Ec. 3}$$

Obteniendo la energía cinética del péndulo 1, sustituyendo energías en Ec.2:

$$\begin{aligned} T_1 &= \frac{1}{2} m_1 [(\dot{X} + \dot{\theta_0} l_1 \cos\theta_1)^2 + (\dot{\theta_1} l_1 \sin\theta_1)^2] + \frac{1}{2} I_1 \dot{\theta_1}^2 \\ T_1 &= \frac{1}{2} m_1 [(\dot{X}^2 + 2\dot{X}\dot{\theta_1} \cos\theta_1 + \dot{\theta_1}^2 l_1^2 \cos^2\theta_1) + (\dot{\theta_1}^2 l_1^2 \sin^2\theta_1)] \\ T_1 &= \frac{1}{2} m_1 \dot{X}^2 + m_1 \dot{X}\dot{\theta} l_1 \cos\theta_1 + \frac{1}{2} m_1 \dot{\theta_1}^2 l_1^2 \cos^2\theta_1 + \frac{1}{2} m_1 \dot{\theta_1}^2 l_1^2 \sin^2\theta_1 + \frac{1}{2} m_1 I_1 \dot{\theta_1}^2 \\ T_1 &= \frac{1}{2} m_1 \dot{X}^2 + \frac{1}{2} m_1 \dot{\theta_1}^2 l_1^2 + m_1 \dot{X}\dot{\theta_1} l_1 \cos\theta_1 + \frac{1}{2} m_1 I_1 \dot{\theta_1}^2 \end{aligned} \qquad \text{... Ec. 4}$$

Obteniendo energía cinética del péndulo 2:

$$T_2 = \frac{1}{2}m_2[(\dot{X} + \dot{\theta_1}L_1\cos\theta_1 + \dot{\theta_2}l_2\cos\theta_2)^2 + (\dot{\theta_1}l_1\sin\theta_1 + \dot{\theta_2}l_2\sin\theta_2)^2] + \frac{1}{2}I_2 + \dot{\theta_2}$$

$$T_2 = \frac{1}{2}m_2[(\dot{X}^2 + \dot{\theta_1}^2L_1^2\cos^2\theta_1 + 2\dot{X}\dot{\theta_1}L_1\cos\theta_1 + 2\dot{X}\dot{\theta_2}l_2\cos\theta_2 + 2\dot{\theta_1}\dot{\theta_2}L_1l_2\cos\theta_1\cos\theta_2 + \dot{\theta_2}^2l_2^2\cos^2\theta_2)\ldots$$
$$+(\dot{\theta_2}^2L_2^2\sin^2\theta_1 + \dot{\theta_1}\dot{\theta_2}L_1l_2\sin\theta_1\sin\theta_2 + \dot{\theta_2}^2l_2^2\sin^2\theta_2)] + \frac{1}{2}I_2\dot{\theta_2}^2$$

$$T_2 = \frac{1}{2}m_2[\dot{\theta_1}^2L_1^2(\cos^2\theta_1 + \sin^2\theta_1) + \dot{\theta_2}^2l_2(\cos^2\theta_2 + \sin^2\theta_2) + 2\dot{\theta_1}\dot{\theta_2}L_1l_2(\cos\theta_1\cos\theta_2 + \sin\theta_1\sin\theta_2)\ldots$$
$$+2\dot{X}(\dot{\theta_1}L_1\cos\theta_1 + \dot{\theta_2}l_2\cos\theta_2) + \dot{X}^2] + \frac{1}{2}I_2\dot{\theta_2}^2$$

$$T_2 = \frac{1}{2}m_2[\dot{\theta_1}^2L_1^2 + \dot{\theta_2}^2l_2^2 + 2\dot{\theta_1}\dot{\theta_2}L_1l_2\cos(\theta_1 - \theta_2) + 2\dot{X}\dot{\theta_1}L_1\cos\theta_1 + 2\dot{X}\dot{\theta_2}l_2\cos\theta_2 + \dot{X}^2] + \frac{1}{2}I_2\dot{\theta_2}^2$$

$$T_2 = \frac{1}{2}m_2\dot{X}^2 + m_2\dot{\theta_1}\dot{\theta_2}L_1l_2\cos(\theta_1 - \theta_2) + m_2\dot{X}\dot{\theta_1}L_1\cos\theta_1 + m_2\dot{X}\dot{\theta_2}l_2\cos\theta_2 + \frac{1}{2}m_2\dot{\theta_1}^2L_1^2\ldots$$
$$+\frac{1}{2}\dot{\theta_2}^2(m_2l_2 + I_2)$$

… Ec. 5

Energías potenciales de los cuerpos del sistema:

$$P_0 = 0$$
$$P_1 = m_1g\cos\theta_1$$
$$P_2 = m_2g(L_1\cos\theta_1 + l_2cos\theta_2)$$

… Ec. 6

Para obtener el Lagrangiano, lo despejamos de Ec. 2:

$$L = (T_0 + T_1 + T_2) - (P_0 + P_1 + P_2)$$

… Ec. 7

Sustituyendo energías, Ec3, Ec4, Ec5 y Ec6 en Ec7:

$$L = \frac{1}{2}m_0\dot{X}^2 + \frac{1}{2}m_1\dot{X}^2 + m_1l_1\dot{X}\theta_1\cos\theta_1 + \frac{1}{2}\dot{\theta_1}(m_1l_1^2 + I_1)\ldots$$
$$\frac{1}{2}m_2\dot{X}^2 + m_2\dot{\theta_1}\dot{\theta_2}L_1l_2\cos(\theta_1 - \theta_2) + m_2\dot{X}\theta_1L_1\cos\theta_1 + m_2\dot{X}\dot{\theta_2}l_2\cos\theta_2\ldots$$
$$\frac{1}{2}m_2\dot{\theta_1}^2L_1^2 + \frac{1}{2}\dot{\theta_2}^2(m_2l_2^2 + I_2) - m_1g\cos\theta_1 - m_2g(L_1\cos\theta_1 + l_2cos\theta_2)$$

… Ec. 8

Agrupando términos:

$$L = \frac{1}{2}\dot{X}(m_0 + m_1 + m_2) + \frac{1}{2}\dot{\theta_1}^2(m_1l_1^2 + I_1 + m_2) + \frac{1}{2}\dot{\theta_2}^2(m_2l_2^2 + I_2) + \dot{X}\dot{\theta_1}\cos\theta_1(m_1l_1 + m_2L_1)$$
$$+ m_2\dot{X}\dot{\theta_2}l_2\cos\theta_2 + m_2\dot{\theta_1}\dot{\theta_2}L_1l_2\cos(\theta_1 - \theta_2) - g\cos\theta_1(m_1l_1 + m_2L_2) - m_2gl_2\cos\theta_2$$

… Ec. 9

Ecuaciones de Lagrange:

$$\frac{d}{dt}(\frac{\partial L}{\partial \dot{X}}) - \frac{\partial L}{\partial X} = u$$
$$\frac{d}{dt}(\frac{\partial L}{\partial \dot{\theta_1}}) - \frac{\partial L}{\partial \theta_1} = 0$$
$$\frac{d}{dt}(\frac{\partial L}{\partial \dot{\theta_2}}) - \frac{\partial L}{\partial \theta_2} = 0 \quad \text{... Ec. 10}$$

Explícitamente, derivando Ec9 y sustituyendo en Ec10:

$$u = (\sum m_i)\ddot{X} + (m_1 l_1 + m_2 L_1)\cos(\theta_1)\ddot{\theta_1} + m_2 l_2 \cos(\theta_2)\ddot{\theta_2}\ldots$$
$$-(m_1 l_1 + m_2 L_1)\sin(\theta_1)\dot{\theta_1}^2 - m_2 l_2 \sin(\theta_2)\dot{\theta_2}^2$$
$$0 = (m_1 l_1 + m_2 L_1)\cos(\theta_1)\ddot{X} + (m_1 l_1^2 + m_2 L_1^2 + I_1)\ddot{\theta_1} + m_2 L_1 l_2 \cos(\theta_1 - \theta_2)\ddot{\theta_2}\ldots$$
$$+m_2 L_1 l_2 \sin(\theta_1 - \theta_2)\dot{\theta_2}^2 - (m_1 l_1 + m_2 L_1) g \sin\theta_1$$
$$0 = m_2 l_2 \cos(\theta_2)\ddot{X} + m_2 L_1 l_2 \cos(\theta_1 - \theta_2)\ddot{\theta_1} + (m_2 l_2^2 + I_2)\ddot{\theta_2}\ldots$$
$$-m_2 L_1 l_2 \sin(\theta_1 - \theta_2)\dot{\theta_1}^2 - m_2 l_2 g \sin\theta_2 \quad \text{... Ec. 11}$$

Representado en forma matricial como:

$$D(\theta)\ddot{\theta_2} + C(\theta, \dot{\theta})\dot{\theta} + G(\theta) = Hu \quad \text{... Ec. 12}$$

Donde:

$$D(\theta) = \begin{bmatrix} d_1 & d_2\cos\theta_1 & d_3\cos\theta_2 \\ d_2\cos\theta_1 & d_4 & d_5\cos(\theta_1 - \theta_2) \\ d_3\cos\theta_2 & d_5\cos(\theta_1 - \theta_2) & d_6 \end{bmatrix}$$

$$C(\theta, \dot{\theta}) = \begin{bmatrix} 0 & -d_2\sin(\theta 1)\dot{\theta_1} & -d3\sin(\theta_2)\dot{\theta_2} \\ 0 & 0 & d_5\sin(\theta_1 - \theta_2)\dot{\theta_2} \\ 0 & -d5\sin(\theta_1 - \theta_2)\dot{\theta_2} & 0 \end{bmatrix}$$

$$G(\theta) = \begin{bmatrix} 0 \\ -f1\sin(\theta_1) \\ -f2\sin(\theta_2) \end{bmatrix}$$

$$H = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{... Ec. 13}$$

Asumiendo que los centros de masa de los péndulos se ubican en los centros geométricos de los vértices [1], los cuales a su vez asumimos que son barras sólidas, tenemos que $l_i = L_i/2$, $I_i = m_i L_i^2/12$. Por los tanto, para las matrices tenemos:

$$d_2 = m_1 l_1 + m_2 L_2 = (\frac{1}{2} m_1 + m_2) L_1$$
$$d_3 = m_2 l_2 = \frac{1}{2} m_2 L_2$$
$$d_4 = m_1 l_2^2 + m_2 L_1^2 + I_1 = (\frac{1}{3} m_1 + m_2) L_1^2$$
$$d_5 = m_2 L_1 l_2 = \frac{1}{2} m_2 L_1 L_2$$
$$d_6 = m_2 l_2^2 + I_2 = \frac{1}{3} m_2 L_2^2$$
$$f_1 = (m_1 l_1 + m_2 L_1) g = (\frac{1}{2} m1 + m_2) L_1 g$$
$$f_2 = m_2 l_2 g = \frac{1}{2} m_2 L_2 g \quad \text{… Ec. 14}$$

Mismas variables que se utilizarán dentro del modelo en Python3 para obtener el comportamiento del péndulo invertido doble sobre un carro.

4.2 Estabilidad

Como se ha mencionado con anterioridad, el sistema de un péndulo invertido doble sobre un carro es caótico.

La estabilidad de un sistema puede ser determinada por las oscilaciones en las trayectorias en los espacios de fases, éstas pueden ser oscilaciones estables e inestables. En donde, las oscilaciones estables presentan un decrecimiento en su amplitud hasta localizarse en un punto de equilibrio; en tanto que en las oscilaciones inestables se presenta un ligero movimiento del punto de equilibrio que lleva a oscilaciones que se alejan cada vez más del punto de equilibrio, es decir, produce trayectorias que nunca regresan a un ciclo permanente.

La predictibilidad de los sistemas está relacionada al problema de su estabilidad. De tal manera, que un sistema dinámico puede ser amplificado por influencias estocásticas iniciadas desde el medio externo, sin generar ruido o caos (*Deissler* y *Doyne*, 1992). La

estabilidad tanto local como global, pueden causar ruido ambiental y ser amplificadas a proporciones macroscópicas. Las inestabilidades locales causan fluctuaciones temporalmente amplificadas en el espacio de fases.

4.3 Control

Una de las principales complicaciones para controlar los sistemas caóticos es que minúsculas variaciones entre el modelo matemático y el sistema real, puede resultar en comportamientos completamente divergentes. Con las técnicas de control neuroevolutivo, no es necesario contar con el modelo matemático del sistema. El algoritmo puede ir simplemente "aprendiendo" del comportamiento del sistema conforme se desarrolla el proceso de evolución. En nuestro caso desarrolló el modelo matemático y la simulación para representar al sistema real.

Una vez obtenido el modelo que describe el comportamiento del sistema que se desea controlar (en este caso un péndulo invertido doble sobre un carro), se deberá definir la función de desempeño con la que serán evaluadas cada una de las propuestas de solución dentro del proceso de evolución del algoritmo genético.

Los parámetros que se deben establecer dentro de las simulaciones para la evaluación de cada modelo son: ángulos máximos permitidos de ambos péndulos así como el desplazamiento máximo permitido del carro, si alguno de éstos parámetros es alcanzado, la evaluación del fenotipo será detenida y se obtendrá el índice de desempeño (*fitness*) del individuo evaluado. El proceso evolutivo se detendrá una vez que se haya alcanzado el número máximo de poblaciones definidas en la configuración del algoritmo, o se sobrepase el umbral de desempeño establecido. En este caso el desempeño de cada individuo será fijado por el tiempo que dure la simulación, esto quiere decir, el tiempo que los péndulos y el carro estén dentro del rango establecido permitido.

Proceso general

Para llegar a la solución, es necesario evaluar cada uno de los individuos de las diferentes generaciones, seleccionar a los más aptos, separarlos en especies y reproducirlos hasta llegar al umbral de desempeño definido en la configuración del algoritmo, en nuestro caso el umbral que definiremos para considerar un individuo como

la solución al problema será de 20 segundos, esto quiere decir que cualquier individuo dentro del proceso de evolución que llegue o sobrepase este valor de desempeño será seleccionado como la solución al problema y la simulación se detendrá. En términos generales, proceso consta de los siguientes pasos

1. Cargar variables de configuración.
2. Inicialización aleatoria de genomas de la primera generación.
3. Mientras la generación actual no sobrepase el número máximo de generaciones:
    a. Por cada uno de los genomas de la generación:
        i. Evaluar genoma y obtener valor de desempeño.
        ii. Si el genoma tiene desempeño igual o mejor del buscado, ir a paso 4.
    b. Reproducción de genomas.
    c. Especiación y mutación.
4. Simulación y almacenamiento de resultados.

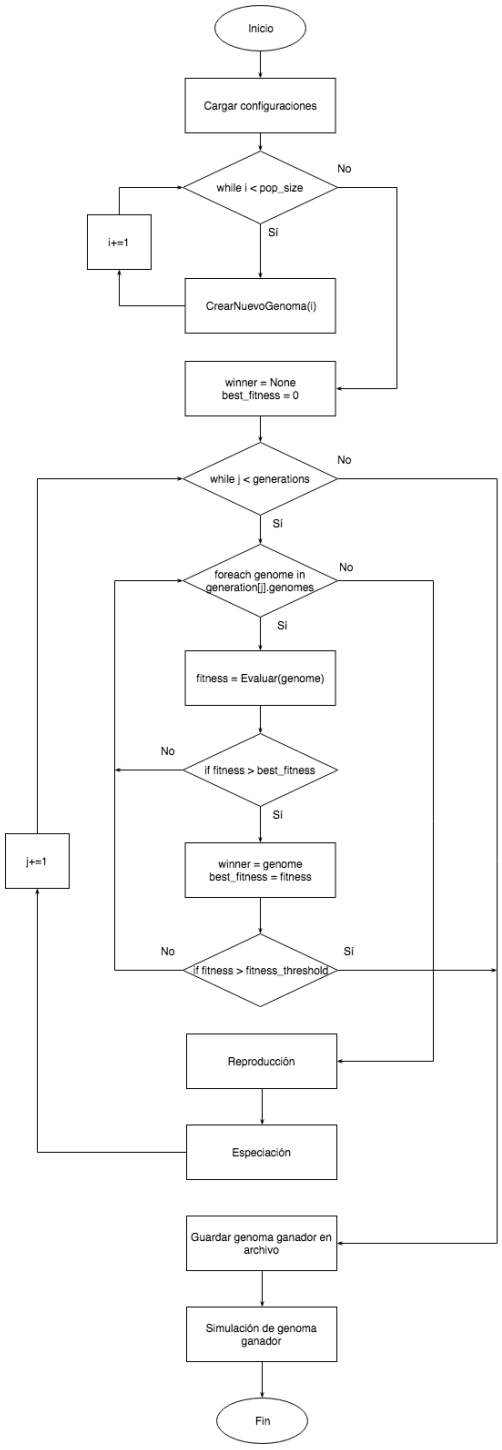


Fig 4B. Diagrama de flujo y pasos generales del proceso de evolución

Evaluación de genotipos

El genotipo es la unidad fundamental dentro del algoritmo NEAT, pues de él dependerá la definición de las entradas, salidas y funciones de transferencia de su fenotipo. El fenotipo es la representación del genoma como una red neuronal artificial y únicamente los fenotipos pueden ser evaluados.

En cada iteración del proceso de evolución se buscará evaluar a cada uno de los individuos por lo menos 2 veces y obtener como valor final el promedio del desempeño de las 2 evaluaciones.

El fenotipo (red neuronal artificial) de cada genotipo dentro del proceso de evolución tendrá 6 entradas y una sola salida. La topología y los valores de los pesos de las conexiones de la red determinarán la salida necesaria para controlar al sistema. Como salida de nuestro sistema controlado, buscamos obtener la fuerza "u" necesaria que, aplicada al carrito pueda estabilizar el péndulo invertido doble, de tal manera que pueda sobrepasar el umbral de 20 segundos, establecido en el archivo de configuración del algoritmo

Entradas a la red

X	Posición en el eje X del carro.
θ_1	Posición angular del péndulo 1.
θ_2	Posición angular del péndulo 2.
$\dot{X}$	Velocidad del carro.
$\dot{\theta}_1$	Velocidad angular del péndulo 1.
$\dot{\theta}_2$	Velocidad angular del péndulo 2.

Salidas de la red

u	Fuerza controlada.

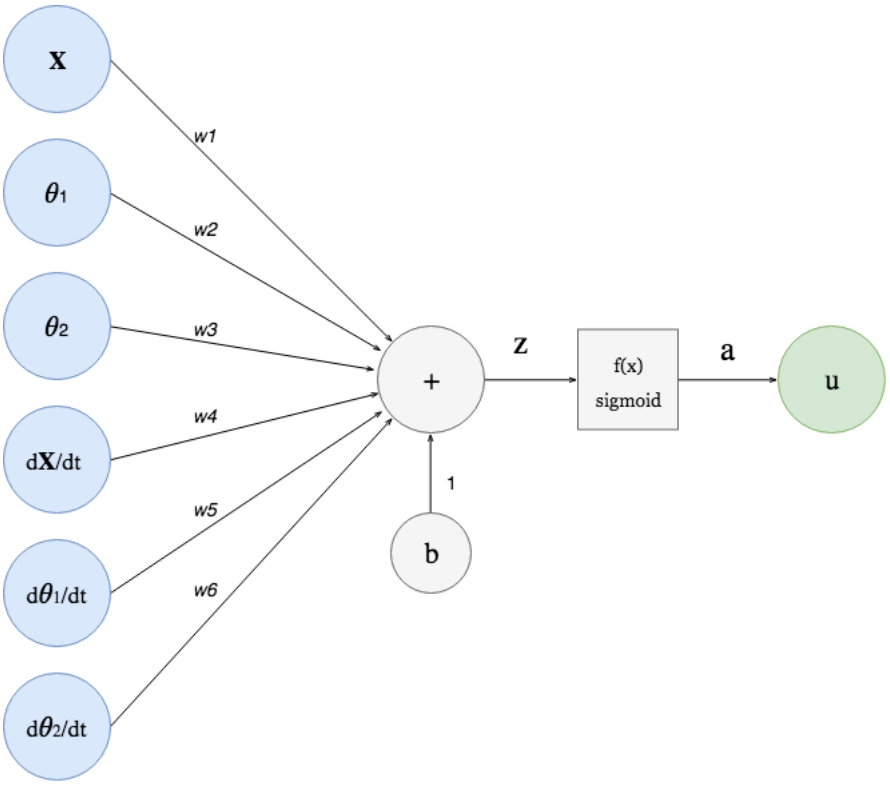


Fig 4C. Topología de individuo de la primera generación del algoritmo para el modelo del péndulo invertido doble sobre un carro.

El modelo del sistema se utiliza al mandar a llamar la función de evaluación del genoma, en donde primero de decodifica al genoma para obtener su fenotipo (red neuronal). Luego, se hará una evaluación por cada lapso de tiempo que definamos en el modelo (tau) para activar la red y obtener la salida y volver a evaluar hasta que la simulación termine, esta puede terminar por que alcanzó el máximo de tiempo o porque alguna de las variables del sistema está fuera de los límites permitidos.

Referencias

[1] Riley, F. William. “Ingeniería Mecánica, Estática”. Edición en español. Editorial Reverté S.A. 2002, España

Capítulo 5. **Neurocontrolador**

En este capítulo se describe el funcionamiento básico de la propuesta de implementación del algoritmo. Desde el proceso general de evolución, variables de configuración, hasta el diagrama de entidades y arquitectura del programa.

5.1 Visión general

Para el neurocontrolador se propone desarrollar neuropy, una biblioteca en Python que implementa una versión simplificada del algoritmo NEAT publicada por Kenneth O. Stanley in 2002, el cual, como se explicó en el capítulo anterior, propone un algoritmo genético para evolucionar redes neuronales artificiales. A grandes rasgos, las consideraciones principales de este algoritmos contra otros algoritmos de neuroevolución son:

- Empieza el proceso de evolución con una población de individuos con estructura o topología mínimas.
- Protege innovaciones topológicas (nuevas estructuras de redes neuronales artificiales) por medio de la especiación de los individuos.
- Usa marcadores históricos como una manera de resolver el problema de las convenciones competidoras, esto es que dos o más redes neuronales codificadas pueden representar la misma solución y por consiguiente podrían existir individuos que se repiten en una misma población.

El desarrollo del paquete tiene como objetivo proporcionar un mejor entendimiento del algoritmo NEAT y está inspirada en la biblioteca neat-python por *@CodeReclaimers* con varias modificaciones con el fin de obtener mayor claridad en el código.

5.2 Requerimientos

La lista de requerimientos para el desarrollo del paquete son los siguientes:

- Configuración de los parámetros del proceso de evolución por medio de un archivo de configuración config.json.
- Inicio de ejecución del proceso de evolución por medio de línea de comandos, dando como argumento el modelo o clase a probar y el modo de ejecución (evolución o prueba de solución).
- Indicar que se ha encontrado una solución cuando el desempeño de al menos un individuo sobrepasa el umbral de desempeño definido en el archivo de config.json.
- Una vez terminado el proceso de evolución, imprimir en la red neuronal con mejor desempeño y guardarla en un archivo .json.

- Posibilidad de evaluar la red solución a partir del archivo guardado.
- Diseño de la estructura de archivos tal que sea fácil incluir nuevos modelos o clases para probar con el algoritmo.

5.3 Entidades

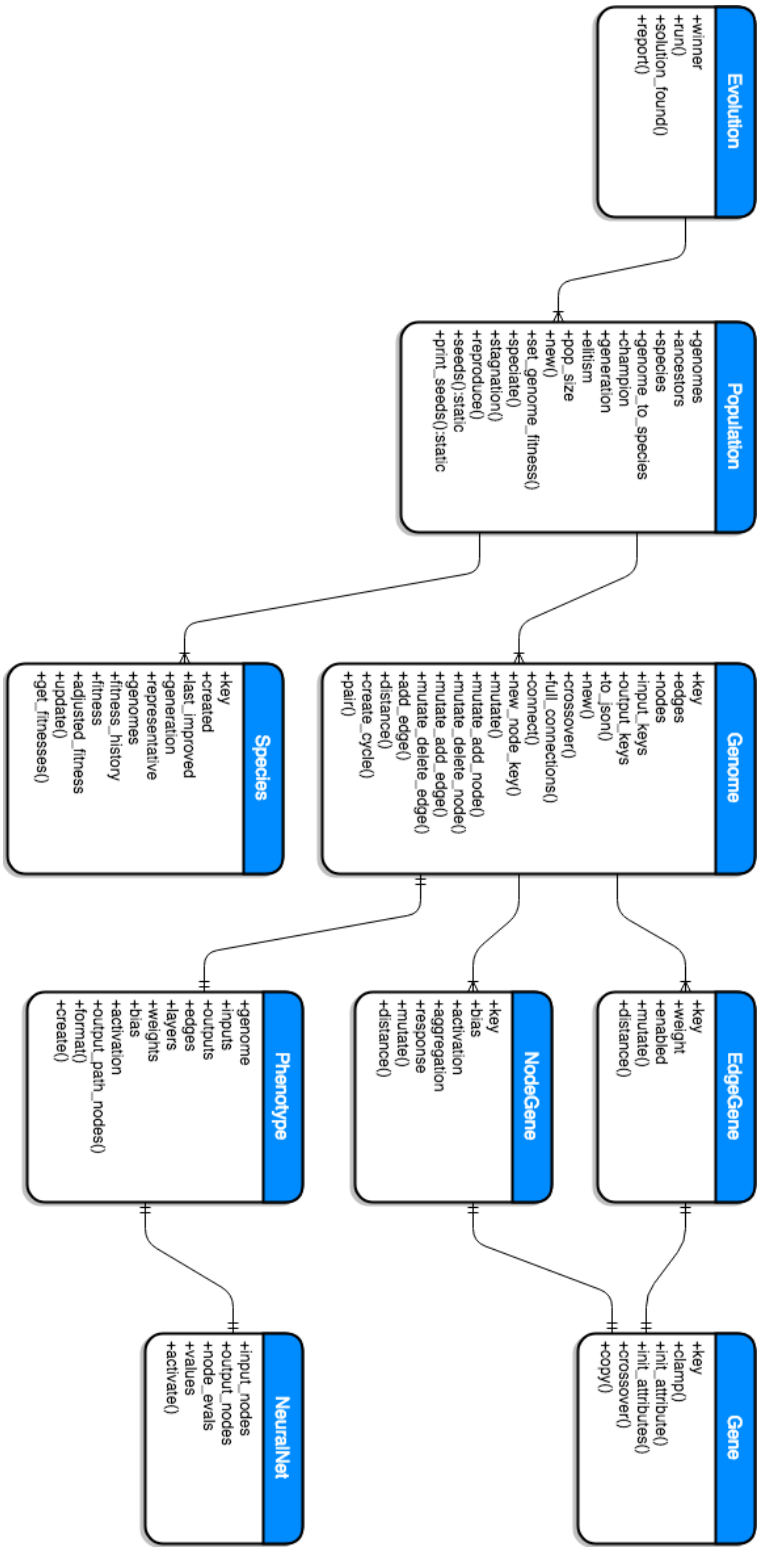


Fig 5A. Diagrama de relación de entidades con propiedades y métodos de cada clase

5.4 Arquitectura

La arquitectura de directorios del paquete es la siguiente:

```
./
├── /activations.py
├── /error.py
├── /evolution.py
├── /gene.py
├── /genome.py
├── /indexer.py
├── /neural_net.py
├── /phenotype.py
├── /population.py
├── /species.py
├── /tests.py
├── /utils.py
└── /visualize.py
└──/models
|    ├── /double_pendulum.py
|    ├── /pendulum.py
|    └── /rendering.py
└──/simulations
     ├── /double_pendulum
     |     ├── /plots
     |     |    ├── /winner.gv.svg
     |     ├── /__main__.py
     |     ├── /config.json
     |     └── /solutions.txt
     ├── /pendulum
     |     ├── /plots
     |     |     └── /winner.gv.svg
     |     ├── /__main__.py
     |     ├── /config.json
     |     └── /solutions.txt
     └── /xor
           ├── /plots
           ├── /winner.gv.svg
           ├── /__main__.py
           ├── /config.json
           └── /solutions.txt
```

Fig 5B. Árbol de módulos y archivos del sistema.

Directorios:

- *root* (./) : Directorio del proyecto, en él viven los archivos principales que confirman el algoritmo
- *models/* : Directorio en donde viven los modelos para realizar las evaluaciones de desempeño de las distintas implementaciones del algoritmo. Por ejemplo aquí se ubican el modelo del Péndulo invertido doble sobre un carro, péndulo invertido sencillo sobre un carro, e incluso podríamos incluir el módulo para la evaluación de la función XOR, pero por simplicidad se incluyó en el mismo archivo en donde se encuentra el método principal.
- *simulations/* : Directorio en donde se encuentran los paquetes de las simulaciones. Cada implementación cuenta con su propio el cual a su vez contendrá el archivo de proceso inicial __main__.py, el archivo de configuración del algoritmos que veremos a detalle más adelante config.json, el archivo de texto que guarda la topología y pesos del individuo solución al modelo solutions.txt y el directorio que guarda la visualización de la red solución ./plots/.

Capítulo 6. **Código**

En este capítulo se describe la implementación del algoritmo en *Python3,* también se ilustran el proceso de instalación, así como los requerimientos mínimos, tanto de hardware como de software, para poder correr el algoritmo, y finalmente se muestra cómo configurar e iniciar el proceso de evolución para el modelo desarrollado.

6.1 Desarrollo

La librería fue desarrollada completamente en *python3.5*, el cual fue seleccionado por su simplicidad en la sintaxis, eficiencia y gran comunidad de soporte. Para iniciar con el desarrollo, fue necesario en un principio crear un ambiente de desarrollo aislado para que no existiera conflicto entre las dependencias de Python del sistema operativo y del nuevo paquete. Este ambiente de desarrollo es creado con el paquete de Python llamado virtualenv.

Los requerimientos técnicos para correr el paquete son mínimos y dependerán esencialmente de la complejidad del problema que se quiera resolver. En el caso específico del péndulo invertido doble con la configuración propuesta, los requisitos son los siguientes:

- Python 3.5 o versiones posteriores
- OS Ubuntu / CentOS / Debian / macOS
- Memoria RAM de al menos 500Mb

La lista de dependencias de Python necesarias para la ejecución de la simulación son:

- numpy: biblioteca especializada para el manejo eficiente de variables numéricas y operaciones matemáticas en Python.
- graphviz: impresión y generación de la gráfica de la red neuronal artificial (grafo).
- matplotlib: impresión de la gráfica de la red neuronal artificial.
- pyglet: biblioteca para generar la animación de los modelos del péndulo sencillo y doble.

6.2 Instalación

Para correr el algoritmo es necesario preparar un ambiente de pruebas, para esto requerimos instalar tanto el intérprete de Python, como todas sus dependencias.

Instalación de Python

Se asumirá que el sistema operativo del equipo en donde se correrá el algoritmo es Ubuntu 16 que está basado en la arquitectura Debian. En caso de no tener instaladas las dependencias necesarias para correr el algoritmo, habrá que seguir los siguientes pasos:

1. Actualizar repositorios:

```
$> sudo apt-get update
$> sudo apt-get -y upgrade
```

2. Instalación del administrador de paquetes de Python 3:

```
$> sudo apt-get install -y python3-pip
```

3. Instalar algunas herramientas de desarrollo:

```
$> sudo apt-get install build-essential libssl-dev libffi-dev python3-dev
```

4. Librería para crear ambientes virtuales:

```
$> sudo apt-get install -y python3-venv
```

Descarga del código

El código fuente se encuentra hospedado en una plataforma llamada github.com y puede ser descargado totalmente gratis pues es de código abierto y cuenta con una licencia tipo MIT, la cuál indica que el código puede ser distribuido y comercializado únicamente con la condición que se mantenga el crédito a el o los autores del mismo.

Para poder descargar el código es necesario primero instalar el programa de control de versiones git.

```
$> sudo apt-get install git
```

Git sirve para llevar un rastro de las versiones y los cambios que se realizan al código, para mas información consultar https://git-scm.com/. En nuestro caso **git** nos servirá únicamente para descargar y copiar el código fuente del paquete. Para descargarlo es necesario ubicarse en la carpeta en donde queramos que la aplicación viva y luego clonar el repositorio:

```
$> cd /path/to/your/dir
$> git clone https://github.com/rogamba/neuropy.git
```

Este comando descargará la última versión del paquete neuropy. Ahora se deberán instalar las dependencias de *python*.

Ambiente de desarrollo

Para instalar las dependencias de *python*, primero debemos aislar nuestro ambiente de desarrollo con las dependencias de *python* instaladas a nivel global, para esto debemos crear un ambiente virtual dentro de la carpeta del código.

```
$> virtualenv env –p python3
```

Este comando crea la carpeta **env** dentro del directorio de la aplicación. Dentro de esta carpeta se encuentra en archivo binario de la versión de *python* que correrá el algoritmo, así como todas las demás dependencias necesarias.

Una vez creado el ambiente virtual debemos activarlo, esto quiere decir, que se exporten las variables de ambiente necesarias para ejecutar la aplicación con la versión correcta de python junto con todas sus dependencias.

```
$> source env/bin/activate
```

Nos daremos cuenta que el ambiente está activo porque se le agregará (env) a nuestro cursor. Podemos desactivar el ambiente virtual con el comando **deactivate**

Activo nuestro ambiente virtual, ahora es necesario instalar los paquetes de *python* necesarios para correr el algoritmo ejecutando el comando:

```
(env)$> pip install -r requirements.txt
```

El comando *pip install* se utiliza para instalar cualquier paquete de *python* directo del administrador de paquetes PyPi. La banders -r indica que las dependencias se obtendrán de un archivo, en nuestro caso llamado requirements.txt.

6.3 Configuración

Una vez clonado e instalado las dependencias de la aplicación será necesario ajustar las variables de configuración para conseguir el comportamiento deseado.

Antes de iniciar las pruebas debemos configurar el algoritmo con las variables que se revisaron en el capítulo anterior las cuales se encuentran en el archivo `config.json` dentro del directorio de la simulación que queramos probar, en nuestro caso será la del péndulo invertido doble.

```
./
├── ...
└──/simulations
    └── /double_pendulum
        ├── /config.json
        ├── /plots
        ├── /winner.gv.svg
        ├── /winner.json
        ├── /__main__.py
        └── /solutions.txt
```

Esencialmente podemos variar los siguientes parámetros de configuración: número de generaciones dentro del proceso de evolución, número de individuos de la población, umbral del valor de desempeño para terminar el proceso de evolución, entre otros...

6.4 Ejecución

Para iniciar la ejecución del algoritmo, simplemente ejecutamos el comando de python haciendo referencia al módulo que queramos probar, en nuestro caso será la simulación del péndulo invertido doble sobre un carro.

```
$> cd /path/to/neuropy                                  # Cambio de directorio
$> source env/bin/activate                              # Activamos ambiente
(env)$> python -m simulations.double_pendulum           # Iniciamos ejecución
```

Al iniciar la ejecución se abre una ventana en donde podremos ver el desempeño de cada uno de los individuos de todas las generaciones del proceso de evolución:

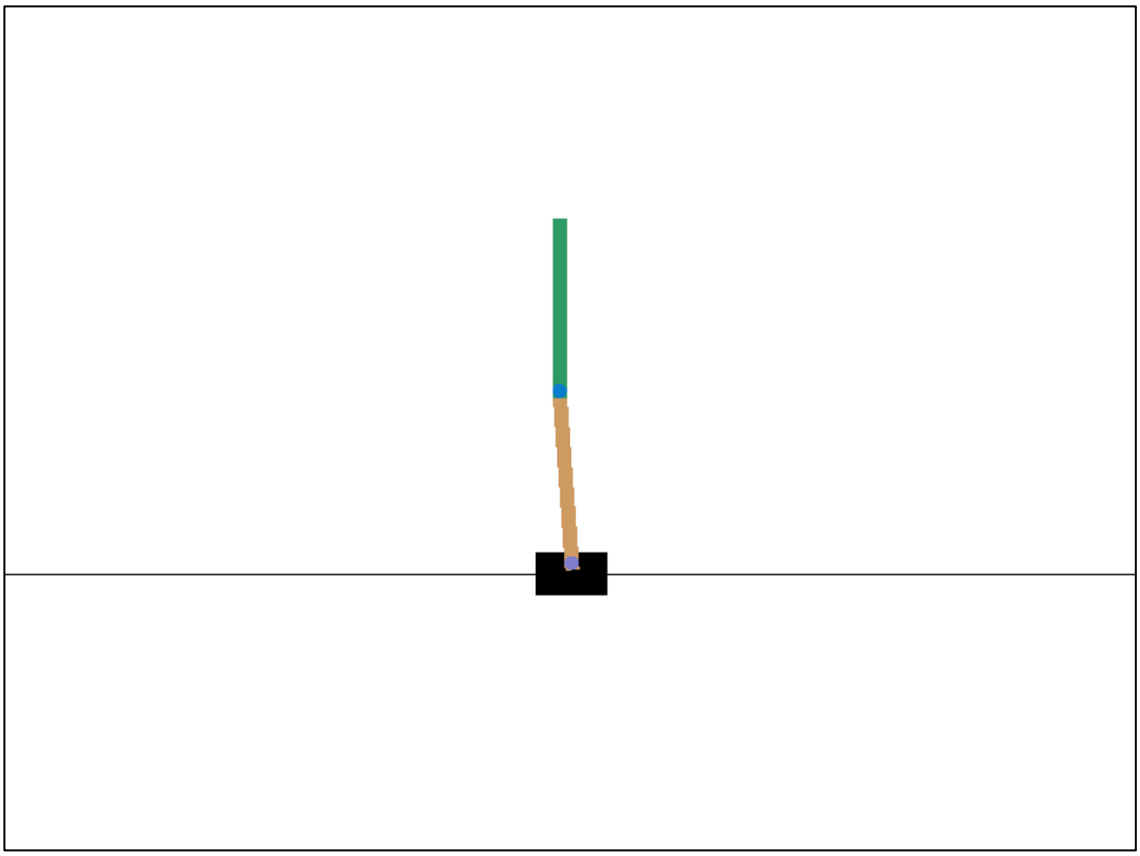


Fig 6A. Captura de la animación de la evaluación de un individuo con *pyglet*, biblioteca en Python para realizar animaciones.

Capítulo 7. **Experimentos**

7.1 Parámetros y variables de medición

Las pruebas se realizaron en un equipo Macbook pro 2014 con procesador core i7 con 16 GB de memoria RAM. Todas las pruebas se hicieron simulando el modelo del péndulo invertido doble y en cada una de ellas se variaron ligeramente algunos de los parámetros para analizar la velocidad de convergencia y tipo de solución. Los parámetros que se variaron son:

- Tamaño de población.
- Número de generaciones.
- Umbral de desempeño.
- Función de activación.
- Número de evaluaciones por genoma.

Las variables que se pretenden medir y analizar al finalizar los experimentos son:

- Tiempo de convergencia del algoritmo.
- Número de generaciones hasta la convergencia.
- Desempeño de la solución.
- Topología de la solución.

7.2 Evaluación

El cálculo del desempeño de los genomas del proceso de evolución dependerá del número de veces que queramos probamos a cada individuo. Idealmente debemos probar más de una vez cada individuo para asegurar que, en caso de tener buen desempeño, no haya sido un comportamiento aleatorio.

El delta de tiempo entre cuadros (tau) es de 0.02 segundos, esto quiere decir que si queremos obtener el desempeño en términos de segundos se deberán sumar los deltas de tiempo de cada cuadro hasta que la simulación termine y obtener un promedio de los tiempos. La función de evaluación es la siguiente:

```python
def eval_genome(genome):
    """ Receives a genome, converts it to it's
        phenotype for evaluation and returns
        a fitness value

            @Params:
                - genome: <obj>
            @Returns:
                - fitness: <float>
    """
    # Double pendulum on cart instance object
    global cart

    # Create phenotype from the genome
    phenotype = Phenotype(config.params, genome)
    net = phenotype.create()
    times = []

    # Loop evaluations per genome (2 or 3)
    for e in range(config.params['evals_per_genome']):
        step = 0
        done = False
        s = cart.reset()
        a = 0

        # Loop up to 100 seconds (5000 frames)
        for n in range(1,5000):

            # Cart state
            s_, r, done, info = cart.step(a)

            # Activate net, get the output
            step += 1
            output = net.activate(s_)
            a_ = output[0]

            # Update state and action
            s = s_
            a = a_

            if done or n >= 5000:
                times.append(step)
                break

    # Calculate fitness
    fitness = np.mean(times) / 50
    return fitness
```

Fig 7A. Parte del código del archivo */simulations/double_pendulum/__main__.py* en donde se ejecutan las iteraciones principales del proceso de evolución.

Por lo tanto el desempeño lo obtendremos sacando el promedio del número de cuadros y dividiendo entre 50.

7.3 Resultados

Prueba 1

La primera prueba se realizó con una función de activación **sigmoide** para los fenotipos, un máximo de **250** generaciones antes de terminar el proceso de evolución, un umbral de desempeño de **30** segundos, **150** individuos a evaluar por generación y **3** evaluaciones por cada individuo.

Configuración:

```
{
    "activation" : "sigmoid",
    "generations" : 250,
    "fitness_threshold" : 30,
    "pop_size" : 150,
    "evals_per_genome" : 3,
    ...
}
```

Fig 7B. Formato del archivo de configuración *config.json* de la prueba 1 en donde se definen variables como: función de activación, número máximo de generaciones, umbral de desempeño, tamaño de población, etc.

Resultados

- Tiempo de evolución: **7h 32m**.
- Generaciones: **65**.
- Desempeño alcanzado: **42.44 segundos**.

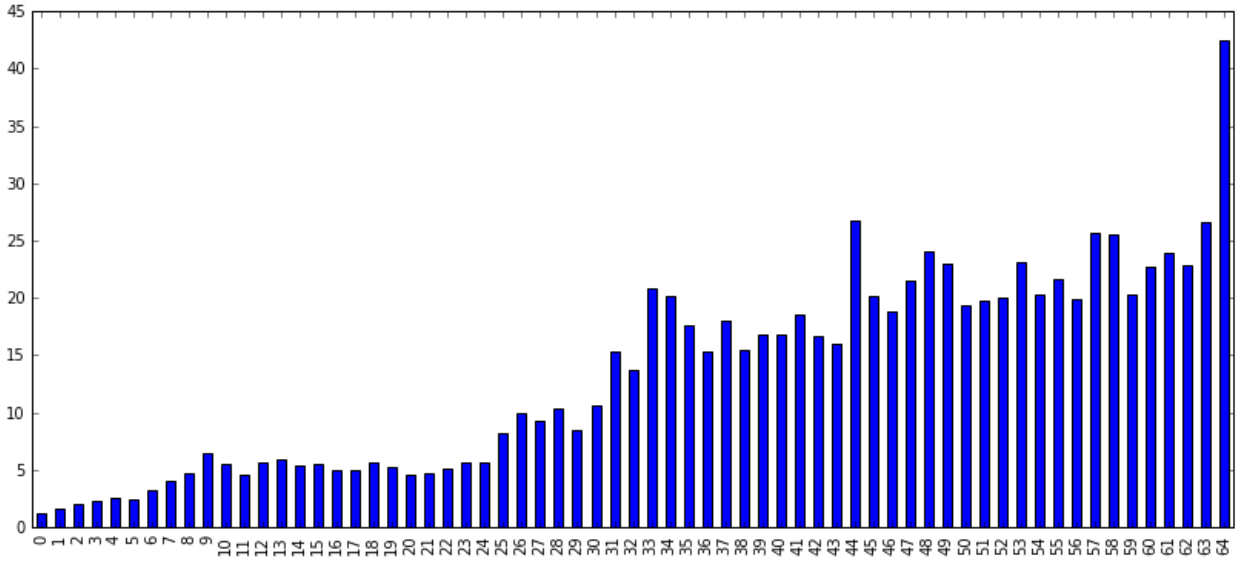


Fig 7C. Gráfica de desempeño del mejor genoma vs número de generación para la prueba 1

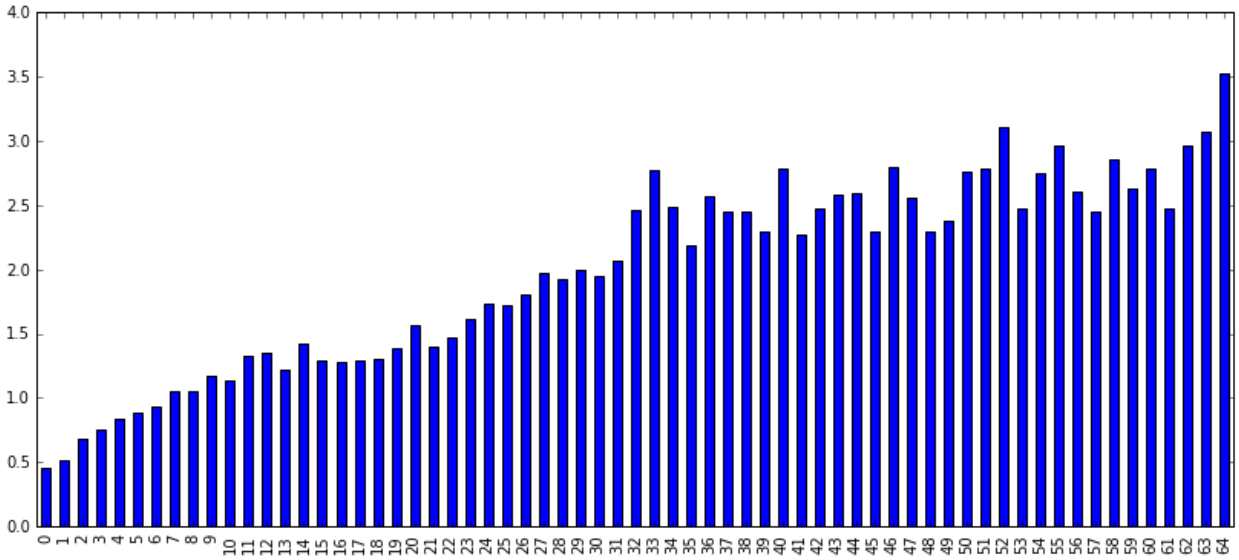


Fig 7D. Gráfica de el desempeño promedio de la generación vs número de generación para la prueba 1

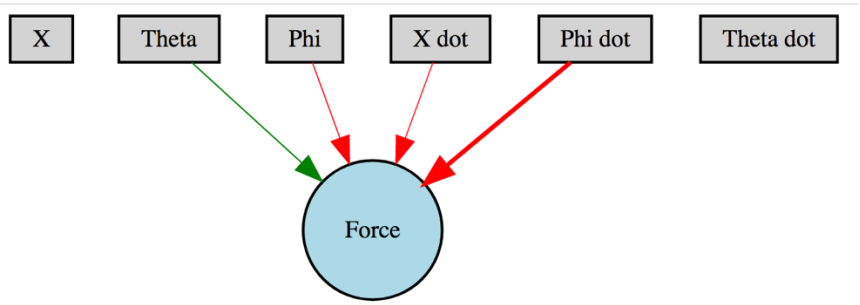


Fig 7E. Topología del fenotipo del genoma ganador de la prueba 1


```
{
    "nodes": [
        {
            "key": 0,
            "aggregation": "sum",
            "response": 1.0,
            "number": 0,
            "activation": "sigmoid",
            "bias": 0.04245775296790166
        }
    ],
    "key": 9344,
    "fitness": 42.44,
    "edges": [
        {
            "key": [-6, 0],
            "weight": -7.879693394235895,
            "enabled": true
        },
        {
            "key": [-4, 0],
            "weight": -1.5406142583148954,
            "enabled": true
        },
        {
            "key": [-3, 0],
            "weight": -1.7233804689689838,
            "enabled": true
        }, {
            "key": [-2, 0],
            "weight": 2.1950908040732275,
            "enabled": true
        }
    ]
}
```


Fig 7F. Representación en Python3 del genoma ganador de la prueba 1

Prueba 2

La segunda prueba se realizó con una función de activación purelin para los fenotipos, un máximo de 200 generaciones antes de terminar el proceso de evolución, un umbral de desempeño de 40 segundos, 100 individuos a evaluar por generación y 1 simulación para la evaluación de cada individuo.

Configuración

```
{
    "activation" : "purelin",
    "generations" : 200,
    "fitness_threshold" : 40,
    "pop_size" : 100,
    "evals_per_genome" : 1,
    ...
}
```

Fig 7G. Formato del archivo de configuración *config.json* de la prueba 2 en donde se definen variables como: función de activación, número máximo de generaciones, umbral de desempeño, tamaño de población, etc.

Resultados

- Tiempo de evolución: 29m 13s.
- Generaciones: 47.
- Desempeño alcanzado: 59.76 segundos.

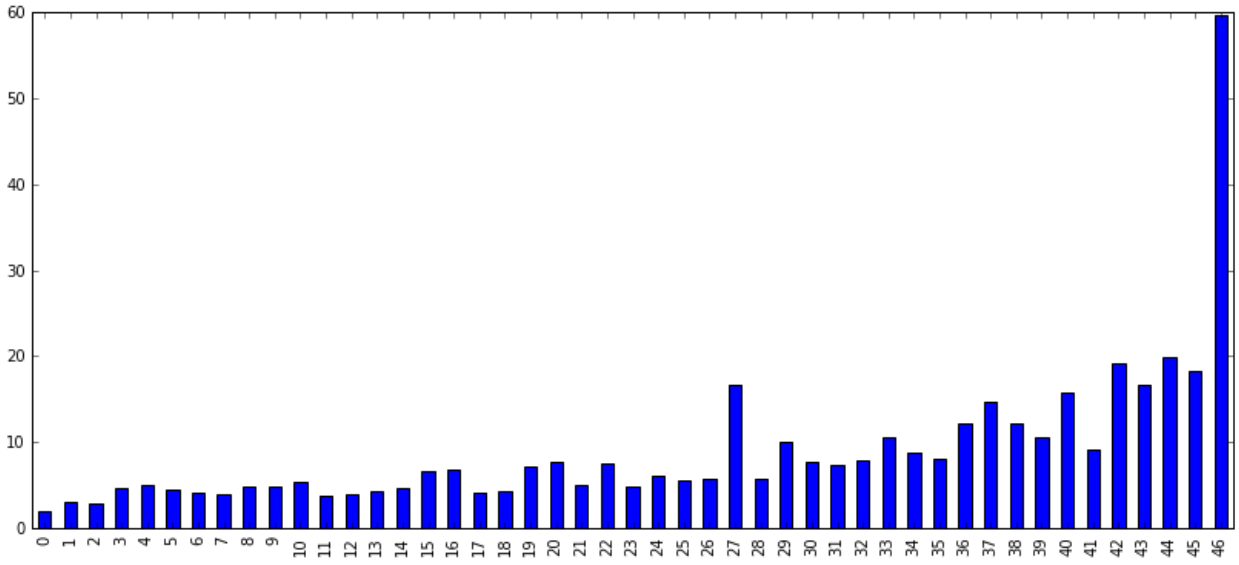


Fig 7H. Gráfica de el desempeño del mejor genoma vs número de generación para la prueba 2

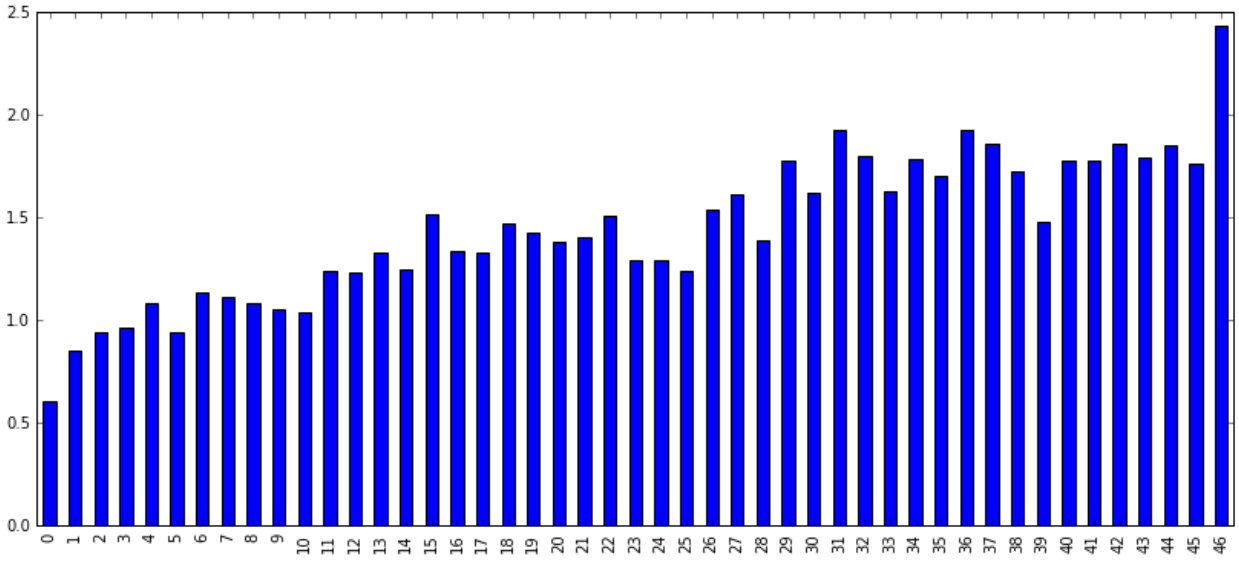


Fig 7I. Gráfica de el desempeño promedio de la generación vs número de generación para la prueba 2

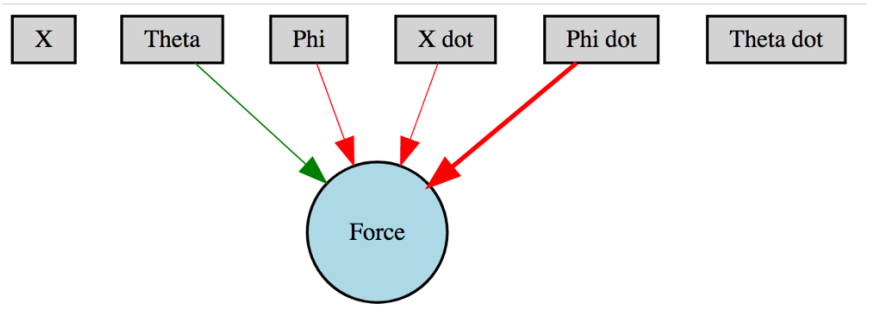


Fig 7J. Topología del fenotipo del genoma ganador para la prueba 2


```
{
    "nodes": [
        {
            "bias": 0.4918536727973282,
            "aggregation": "sum",
            "activation": "purelin",
            "key": 0,
            "response": 1.0,
            "number": 0
        }
    ],
    "key": 4414,
    "edges": [
        {
            "enabled": true,
            "key": [-6, 0],
            "weight": -2.185515559173712
        },
        {
            "enabled": true,
            "key": [-4, 0],
            "weight": -0.3442777272580208
        },
        {
            "enabled": true,
            "key": [-3, 0],
            "weight": -3.1377489260669575
        },
        {
            "enabled": true,
            "key": [-2, 0],
            "weight": 2.711322510428475
        }
    ],
    "fitness": 59.76
}
```


Fig 7K. Representación en Python3 del genoma ganador de la prueba 2

Capítulo 8. **Conclusiones**

Como primer objetivo del trabajo, se planteó investigar y estudiar distintas implementaciones del algoritmo NEAT en Python y otros lenguajes de programación para detectar áreas de oportunidad. Al realizar la investigación se encontró que el paquete *neat-python* podía servir como base para el proyecto porque adaptab a los requerimientos buscados. También se encontraron algunas desventajas como que el código no era suficientemente claro en sintaxis ni en el proceso de ejecución, por lo tanto se partió de ese proyecto para reconstruirlo y desarrollar el paquete *neuropy*.

El segundo objetivo consiste en modelar matemáticamente el sistema del péndulo invertido doble sobre un carro. Afortunadamente existe bastante información y publicaciones que estudian el péndulo invertido doble por ser uno de los sistemas caóticos más sencillos que se conocen, pero desafortunadamente no existe la misma documentación para uno sobre un carro móvil. Luego de haber realizado el modelo matemático, uno de los retos más grandes fue el de su implementación en Python y su animación, pues de la misma manera existe amplia documentación para péndulos sencillos pero no péndulos dobles sobre un carro.

Como último objetivo se propuso el desarrollo de la implementación de NEAT en Python3 para ejecutar el proceso de evolución y encontrar la solución al control de un sistema caótico para medir y analizar resultados del proceso de convergencia. Con base en los resultados obtenidos podemos concluir que el algoritmo NEAT, configurado con los parámetros correctos, puede servir para el control de sistemas caóticos de, por lo menos, un grado bajo de complejidad como lo es el del péndulo invertido doble sobre un carro.

Durante la realización de las primeras pruebas, al simular las soluciones obtenidas luego de haber ejecutado el proceso de evolución, se podía notar un comportamiento irregular en el sistema, éste se lograba estabilizar y balancear pero el péndulo 2 siempre terminaría con un ángulo de 180 o -180 grados. Esto se debe a que en un principio no se restringió el ángulo theta 2 el cual, al superar cierto rango, debía terminar las evaluaciones. Por lo tanto la solución del modelo efectivamente balanceaba el péndulo pero una vez que el péndulo 2 se encontraba totalmente invertido en un estado "estable".

Luego de haberse realizado las simulaciones con los parámetros correctamente ajustados, se pudo observar que el algoritmo convergió notablemente más rápido en la

segunda prueba, esto puede variar tanto a los parámetros de configuración, como a la aleatoriedad incluida en la mutación de los nodos. Una hipótesis propuesta es que se debe al cambio de función de activación de la sigmoide a la función purelin pero deberán realizarse más pruebas si se desea una comprobación exacta. Otra observación importante es que la topología de la red es prácticamente de una sola capa, esto quiere decir que no se evolucionó de manera importante la topología de la red y únicamente fue necesario ajustar los pesos de las entradas para poder obtener el controlador de el sistema propuesto.

Otra cosa que se pudo observar durante las pruebas es que el equipo no estaba utilizando el 100% de su capacidad de procesamiento, esto se debe a que el proceso del algoritmo como está diseñado actualmente, utiliza un solo núcleo de la unidad de procesamiento central. Si quisiéramos utilizar más núcleos se debieran de hacer las modificaciones necesarias al algoritmo y utilizar el módulo de *multiprocessing* de Python para manejar procesos concurrentes.

Para finalizar, es posible concluir que la implementación de un algoritmo NEAT puede ser muy útil si se cuenta con suficiente poder de procesamiento y no necesariamente se requiere entender por completo el modelo del sistema, sino simplemente se pretende controlarlo. El único requisito es definir correctamente las variables de entrada de la red para que sean relevantes al sistema y así obtener las salidas esperadas de las variables que se pretenden controlar.

Bibliografía

- **Bogdanov Alexander**. "Optimal Control of a Double Inverted Pendulum on a Cart". OGI School of Science & Engineering, Diciembre 2004.

- **O. Stanley Kenneth, Miikkulainen Risto.** "Evolving Neural Networks through Augmented Topologies". Department of Computer Sciences, The University of Texas at Austin, Austin, USA.

- **Hernández de la Sota, Cristina.** "Control de Sistemas Dinámicos Caóticos". Universidad Politécnica de Madrid, Facultad de Informática, Madrid 2004.

- **Mathew, Tom V.** "Genetic Algorithm". Department of Civil Engineering, Indian Institute of Technology Bombay, Mumbai-400076.

- **Hoekstra, Vincent.** "An overview of neuroevolution techniques". Diciembre 2011.

- **Kearney , William T.** "Using Genetic Algorithms to Evolve Artificial Neural Networks". Colby College, 2016.

- **Ogata, Katsuhiko.** "Modern Control Engineering". Fifth Edition. Pearson Education Inc., New Jersey, 2010.

- **Jones, M. Tim.** "AI Programming". Second Edition. Charles River Media, 2003.

- **Hagan, Martin T., Demuth, Howard B.** "Neural Network Design", 2nd Edition.

- **Mitchel, Melanie.** "An Introduction to Genetic Algorithms". The MIT Press, Massachusetts Institute of Technology, 1996.

- **Ben, Coppin.** "Artificial Intelligence Illuminated". Jones and Barlett Publishers, US, 2004.

9.1 Recursos

Neat-Python
http://repositorio.ipl.pt/bitstream/10400.21/1144/1/Disserta%C3%A7%C3%A3o%20Ingl%C3%AAs

Implementación de algoritmos genéticos
https://lethain.com/genetic-algorithms-cool-name-damn-simple/

http://gekkoquant.com/2016/03/13/evolving-neural-networks-through-augmenting-topologies-part-1-of-4/

Reinforcement learning in autonomous agents
http://repositorio.ipl.pt/bitstream/10400.21/1144/1/Disserta%C3%A7%C3%A3o%20Ingl%C3%AAs

Double pendulum modeling
http://blog.otoro.net/2015/02/11/cne-algorithm-to-train-self-balancing-double-inverted-pendulum/

Double Pendulum modeling2 **
http://scienceworld.wolfram.com/physics/DoublePendulum.html

Open AI - Reinforcement learning
https://gym.openai.com/