FakeNEAT: Neuroevolución mediante algoritmos genéticos y enfriamiento simulado

Hugo Albert Bonet
Técnicas Metaheurísticas
Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital.
Universitat Politècnica de València, España
halbbon@etsinf.upv.es

***Resumen*—El presente trabajo desarrolla FakeNEAT, un método de neuroevolución basado en NEAT que combina algoritmos genéticos y de enfriamiento simulado para modificar tanto la arquitectura como los pesos de redes neuronales al mismo tiempo. En el documento se detalla el funcionamiento de los distintos elementos del algoritmo, su implementación y su uso, y se muestran resultados en distintos conjuntos de datos.**

***Palabras clave*—Neuroevolución, Redes Neuronales, Algoritmos Genéticos, Enfriamiento Simulado, Optimización, Técnicas Metaheurísticas**

I. Descripción del problema

Actualmente, las redes neuronales forman una parte fundamental del aprendizaje automático. Desde sus inicios como perceptrón multicapa, han ido avanzando y superándose continuamente, a través de la creación de backpropagation [1] para redes profundas, funciones de activación no lineales y no saturantes, conexiones residuales, capas convolucionales o la arquitectura de transformers. Gracias a ellas, nos encontramos en un momento en el que la inteligencia artificial es conocida en todo el mundo (y adorada o temida) por su potencial. Han demostrado grandes capacidades para enfrentarse a datos estructurados, series temporales, imagen, texto, vídeo, audio...

Su proceso de entrenamiento, sin adentrarnos en redes como los *Generative Pre-Trained Transformers (GPT)* [2] que incorporan procesos de aprendizaje por refuerzo a partir de realimentación humana (RLHF) [3], está dividido en dos pasos que se iteran de forma consecutiva.

En primer lugar, encontramos una búsqueda de arquitecturas. Este proceso se realiza a través de una búsqueda principalemente humana, en la que se varía la arquitectura de la red en función del conocimiento e intuición del experto. Este proceso se puede complicar tanto como se desee. A nivel básico, se varía el número de capas ocultas de la red y el número de neuronas de cada una de estas capas, pero también se pueden variar las funciones de activación de cada capa, decidir si se crea una conexión residual, añadir bloques especiales como capas convolucionales o mecanismos de atención, concatenar resultados de varias capas y muchas opciones más.

El segundo paso consiste en entrenar la red neuronal. Este proceso está automatizado. Encontramos numerosos optimizadores (Adam [4], Stochastic Gradient Descend (SGD), AdamW, AdEMAMix [5]...) que se encargan de variar los pesos de la red atendiendo al rendimiento actual de la misma ante las muestras de entrenamiento, mediante variaciones de algoritmos bien conocidos como descenso por gradiente y backpropagation. Se trata de un proceso que, pese a ser costoso computacionalmente cuanto mayor sea el conjunto de entrenamiento, se realiza sin necesidad de una gran atención del experto más allá de seleccionar los hiperparámetros adecuados.

Esta división del perfeccionamiento de una red neuronal en dos pasos separados, y sobre todo debido a la necesidad de una gran atención humana en uno de ellos, complica y ralentiza el proceso. Por ello, se han inventado procedimientos que consiguen paliar hasta un cierto grado esta situación. Los *tuners*, por ejemplo, permiten una búsqueda reducida en el infinito espacio de posibilidades que se abre ante esta situación. Incluyen tanto búsquedas aleatorias como otras más inteligentes, como por ejemplo bayesianas. Sin embargo, el proceso sigue dividiéndose en dos pasos: el *tuner* varía ciertos valores y después la red se entrena, se evaluan los resultados y se deciden las siguientes variaciones, y así sucesivamente.

Otra solución propuesta es la neuroevolución. Este concepto emplea algoritmos evolutivos para entrenar los pesos de la red al mismo tiempo que varía su arquitectura. De esta forma, unifica ambos pasos en uno, consiguiendo una gran exploración de posibilidades para alcanzar la mejor red neuronal posible. Un gran ejemplo del estado del arte de la neuroevolución es *NeuroEvolution of Augmented Topologies (NEAT)* [6], con ciertas variaciones como *HyperNEAT* [7]. Este método emplea algoritmos genéticos para partir de la red más sencilla posible e iterativamente añadir neuronas y conexiones hasta alcanzar el mejor resultado posible.

NEAT genera redes neuronales tan singulares como la que se muestra en la Figura 1. Podemos observar que se trata de una red neuronal heterogénea, diferente a las que estamos acostumbrados a encontrarnos divididas en capas bien diferenciadas y densamente conectadas. Esta forma de construir las redes neuronales, pese a conseguir un alto grado de diversidad y simplicidad, presenta un problema en el momento de la inferencia y el entrenamiento (pese a que NEAT no está pensado para que las redes sean entrenadas con backpropagation, pero siempre es interesante contar con esa posibilidad para procesos como el *fine-tuning*). Esto se debe a que no permite el tratamiento de la red neuronal como un conjunto de matrices que se multiplican entre sí y a las que

se les aplica una función no lineal, proceso para el que se ha optimizado el *hardware* actual como las GPUs.

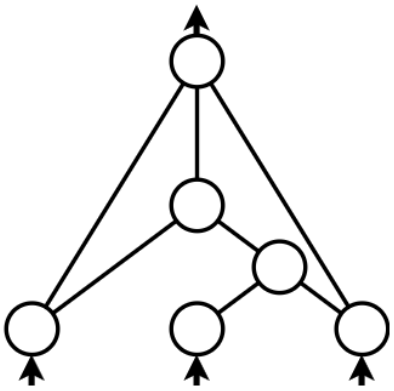


Fig. 1: Ejemplo de red neuronal generada por NEAT

Es por esta razón, que el presente trabajo propone una variación de NEAT ajustado a los procesos actuales de entrenamiento e inferencia llamado **FakeNEAT**. FakeNEAT combina algoritmos genéticos y enfriamiento simulado para obtener los mejores resultados, manteniendo siempre una estructura de red neuronal compatible con los procesos optimizados actuales para inferencia y entrenamiento.

Debido a las restricciones temporales ligadas a este trabajo, FakeNEAT se limita por el momento a optimizar el número de neuronas por cada capa y los pesos de la propia red. Actualmente deja de lado aspectos a optimizar como el número de capas (viene dado como hiperparámetro), la función de activación de cada capa (ReLU por defecto excepto para la capa de salida, que se ajusta automáticamente según sea una tarea de clasificación o regresión), ni capas especiales como pueden ser las convolucionales o las de normalización. Todas estas mejoras se proponen como trabajo futuro.

II. Representación de los datos

En este problema, una solución viene representa una red neuronal. Concretamente, una solución está representada en dos apartados. Por un lado, la arquitectura de la red neuronal se representa como una lista de valores, en las que cada valor representa el número de neuronas de una capa oculta como se muestra a continuación:

$$N = [n_1, n_2, ..., n_H],\ n_h \in \mathbb{N},\ n_h > 0 \tag{1}$$

Donde H se corresponde con el número de capas ocultas establecidas como hiperparámetro, y no incluye ni la capa de entrada ni la de salida puesto que se detectan y ajustan automáticamente. Esta lista únicamente cumple la función de un índice al que acudir cuando se requiera una inspección de la arquitectura de una red neuronal, de manera informativa. Por ello, la solución se acompaña de las matrices de pesos que se encuentran entre cada par de capas (incluyendo las de entrada y salida), además de las funciones de activación que acompañan a cada capa. Estos pesos están representados como una red neuronal de la librería PyTorch para facilitar funciones como la inferencia, y deben cumplir que:

$$\forall i, j \in \{0, 1, ..., H, H+1\}, dim(W_{ij}) = n_i \times n_j \tag{2}$$

Interpretando n_0 y n_{H+1} como el número de neuronas de la capa de entrada y salida respectivamente. De esta manera, se asegura que siempre podemos multiplicar dos matrices contiguas.

Teniendo en cuenta ambas representaciones en conjunto, una solución será factible siempre que cumpla con las restricciones mostradas en 1 y 2.

III. Solución del Algoritmo Genético

Los algoritmos genéticos pertenecen a la familia de algoritmos evolutivos, y son la aproximación sobre la que se construyó NEAT. Esta clase de algoritmos se basan en la teoría de la evolución de las especies de Charles Darwin. De este modo, parten de una población de soluciones iniciales, con sus semejanzas y sus diferencias, cuya adaptación al entorno es evaluada mediante la llamada función *fitness*. Dentro de la población, se producen cruces entre soluciones y mutaciones de las mismas que crean nuevas soluciones. Las soluciones con mejor adaptación al entorno, mejor fitness, tendrán más probabilidad de perpetuar sus características. En estos algoritmos, el balance entre *exploración* y *explotación* es un concepto clave para obtener los mejores resultados posibles.

En FakeNEAT, el algoritmo genético (NeuroEvolution Genetic Algorithm, NEGA) es el primer método con el que encontrar una solución optimizada (no necesariamente óptima, al tratarse de una técnica metaheurística). Esta sección describe las distintas características, decisiones y detalles de diseño del NEGA.

A. Población inicial

Una de las decisiones importantes del diseño de un algoritmo genético es la creación de la población inicial de soluciones. El proceso mediante el que se creen estas soluciones tendrá un gran impacto en los resultados finales obtenidos. El proceso a seguir para crear la población inicial depende en gran medida del problema a resolver, aunque dentro de un mismo problema se puede encontrar formas distintas de inicializar la población.

En el caso de FakeNEAT, la creación de una solución inicial es un proceso muy sencillo en el que simplemente se deben cumplir las restricciones señaladas en 1 y 2. De este modo, el NEGA ha sido diseñado para permitir dos tipos de población inicial según el objetivo final:

- **Búsqueda de simplicidad**: La primera opción trata de responder a problemas en los que la simplicidad y sencillez de la red neuronal final es un aspecto clave. En este caso, se parte de la solución más simple y eficiente en términos de computación en inferencia: una red neuronal con una única neurona por cada capa. $n_i = 1\ \forall i \in \{1, ..., H\}$. Observemos que las capas de entrada y salida mantienen el número de neuronas necesarias para ser compatible con

la tarea a resolver. De esta manera, todas las matrices de pesos W_{ij} tendrán una dimensión de 1×1 excepto $dim(W_{01}) = n_0 \times 1$ y $dim(W_{H,H+1}) = 1 \times n_{H+1}$. De esta manera, se generan redes neuronales de grandes dimensiones únicamente cuando producen un resultado mejor.

- **Búsqueda de variabilidad**: Esta opción trata de producir una población más variada en el inicio, favoreciendo la *exploración* al inicio pero sacrificando simplicidad ya que podemos encontrar redes muy grandes desde el inicio. Se parte de redes neuronales con un número aleatorio de neuronas por capa, estableciendo un número máximo de neuronas por capa al inicio ($maxN$): $n_i = randint(1, maxN)\ \forall i \in \{1, ..., H\}$. De nuevo, las capas inicial y final se ajustan al problema. Las matrices de pesos se inicializan con valores aleatorios, atendiendo siempre a las dimensiones de las capas entre las que se encuentran de forma que cumplan las restricciones en 2.

Cualquiera de las dos inicializaciones puede determinarse antes de ejecutar el algoritmo.

B. Función fitness

Otro aspecto realmente importante a la hora de diseñar un algoritmo genético, es la función fitness. Esta función no solamente nos permitirá comparar la bondad de las soluciones entre sí, sino que tiende a cobrar un papel fundamental en el proceso de selección natural para el cruce, la mutación y el reemplazo. Además, su naturaleza determinará si se trata de un problema de maximización o minimización.

El NEGA de FakeNEAT cuenta con dos opciones distintas de función fitness, en función de si se trata de un problema de regresión o clasificación:

- **Regresión**: Se emplea el error cuadrático medio:

$$F(s) = MSE(y, \hat{y}) = (y - \hat{y})^2 \tag{3}$$

- **Clasificación**: Para conseguir una homogeneidad en la optimización del problema, se ha buscado una función fitness que requiera también de una minimización. Por lo tanto, se emplea la inversa del accuracy:

$$F(s) = 1 - \frac{Verdaderos\ Positivos}{|Datos|} = 1 - accuracy \tag{4}$$

Esto permite que la función, además, esté acotada al rango $[0, 1]$.

A la hora de evaluar la función fitness en un individuo, se procede de formas distintas según la situación. Previo a la ejecución del algoritmo, se divide el conjunto de datos en tres particiones (entrenamiento, validación y test). Durante el proceso de optimización, el fitness de un individuo se evalúa sobre un batch (de tamaño previamente establecido como hiperparámetro) del conjunto de validación. Una vez terminado el algoritmo, la solución final se evalúa en el conjunto de test para devolver el fitness final (por defecto se evalúa al individuo sobre todo el conjunto de test, pero es posible reducirlo a un tamaño de batch menor).

C. Mutaciones

La mutación es el proceso en el que una solución sufre una variación aleatoria. Es un proceso que permite una gran flexibilidad en términos de creatividad en el diseño, ya que se puede decidir si aplicar a todas las soluciones, a una o a un grupo en cada generación, qué tipo de variación se sufre, si genera un individuo nuevo o modifica al anterior...

En el caso de FakeNEAT, las mutaciones se producen sobre un porcentaje de la población seleccionada de acuerdo al proceso de selección explicado más adelante. Estas mutaciones siempre generan un individuo nuevo que se añadirá a la población existente de soluciones. Las tres mutaciones distintas son:

- **Eliminar neuronas**: Esta mutación elimina un número aleatorio de neuronas de una capa aleatoria, i, de forma que $n'_i = n_i - randint(1, n_i - 1)$, asegurando que $n'_i > 0$. Para conseguirlo, debemos modificar también las matrices de pesos W_{hi} y W_{ij} de forma que $dim(W_{hi}) = n_h \times n'_i$ y $dim(W_{ij}) = n'_i \times n_j$. Esto quiere decir que se debe eliminar tantas columnas de la matriz anterior y filas de la posterior como neuronas se hayan eliminado de la capa i.
- **Añadir neuronas**: Se produce el proceso inverso, ya que se añaden un número aleatorio de neuronas (con un valor máximo fijado con anterioridad, $maxNew$) a una capa aleatoria. Por lo tanto, $n'_i = n_i + randint(1, maxNew)$. Además, se debe actualizar de nuevo las matrices de pesos, de forma que $dim(W_{hi}) = n_h \times n'_i$ y $dim(W_{ij}) = n'_i \times n_j$. Esto quiere decir que se debe añadir tantas columnas a la matriz anterior y filas a la posterior como neuronas se hayan añadido a la capa i.
- **Mini-train**: Esta mutación ejecuta un mini proceso de entrenamiento tradicional (descenso por gradiente y back-propagation) sobre el conjunto de entrenamiento. Con la posibilidad de ser modificado, el entrenamiento se realiza con una única iteración sobre un único batch del conjunto de entrenamiento. De esta manera, conseguimos ahorrar tiempo mientras realizamos una pequeña modificación en los pesos mediante un método inteligente en lugar de aleatorio. Además, al utilizar una única iteración se evitan cambios muy agresivos y, por tanto, también el sobre-entrenamiento y la sobreoscilación. En este proceso de entrenamiento se utiliza la función de fitness como función de pérdida. Sin embargo, en el caso de una tarea de clasificación, se puede seleccionar la entropía cruzada como función de pérdida.

En cada generación, siendo m la proporción de individuos que mutan y pop el tamaño de la población, se seleccionan $m \cdot pop$ individuos de acuerdo al proceso de selección explicado más adelante para cada una de las mutaciones (de forma no excluyente, por lo que un mismo individuo podría generar un individuo nuevo por cada mutación en una misma generación). Por lo tanto, el número de individuos nuevos generados a partir de la mutación en cada generación es $3(m \cdot pop)$.

D. Cruce

El cruces es otro tipo de modificación de individuos que combina las características de dos (generalmente) o más individuos de la población. De esta forma, mientras que la mutación crea individuos nuevos en un espacio de búsqueda cercano al individuo que la sufre, el cruce combina dos individuos para explorar un espacio distinto del de cada uno, un espacio combinado. Utilizando un método de selección que favorezca a individuos mejor adaptados, se combinan y perpetran características beneficiosas para el problema.

El NEGA de FakeNEAT proporciona un único método de cruce, llamado **encoder-decoder crossover**. Este mecanismo de cruce consiste en seleccionar una sección inicial de una solución progenitora (encoder) y unirlo a una sección final (decoder) de la otra solución progenitora, y viceversa. Con el objetivo de que ambas soluciones resultantes sean factibles, el mecanismo de cruce se produce de la siguiente manera:

Algorithm 1 Encoder-Decoder Crossover

Require: $parent_1, parent_2$
  $idx = randint(1, len(parent_1))$
  if $parent_1[idx] > parent_2[idx]$ **then**
    $añadir\_neuronas(parent_2, idx, parent_1[idx] - parent_2[idx]$
  else if $parent_1[idx] < parent_2[idx]$ **then**
    $añadir\_neuronas(parent_1, idx, parent_2[idx] - parent_1[idx]$
  end if
  $child_1 = combina(parent_1[: idx], parent_2[idx :])$
  $child_2 = combina(parent_2[: idx], parent_1[idx :])$
  return $child_1, child_2$

De esta manera, se selecciona una capa aleatoria, se comprueba si ambos progenitores tienen el mismo número de neuronas en dicha capa y, de lo contrario, se añaden neuronas al que lo necesite. A continuación, dicha capa sirve de separación entre lo que consideramos encoder y decoder, transmitiendo todas las matrices de pesos anteriores a un descendiente y todas las posteriores al otro.

E. Selección y Reemplazo

Entendemos por selección al mecanismo utilizado para elegir a qué individuos se le aplica cada mutación y cruce. Esta decisión es clave a la hora de equilibrar los conceptos de *exploración* y *explotación*. Una selección demasiado elitista nos hará caer en mínimos locales, mientras que una muy permisiva evitará que la población avance hacia individuos mejor adaptados.

En cuanto al reemplazo, podría entenderse como una selección a la inversa. Se refiere al mecanismo mediante el que se selecciona a los individuos que desaparecerán de una generación a la siguiente. Además de balancear *exploración* y *explotación*, es muy importante que el reemplazo no elimine al mejor individuo de la población, de forma que siempre se mantenga la mejor solución encontrada hasta ese momento.

En el caso de FakeNEAT, el criterio de selección será proporcional al accuracy (inverso al fitness) siguiendo el método de selección por rueda de ruleta, atendiendo a la siguiente fórmula donde $P(i)$ es la probabilidad de seleccionar al individuo i:

$$P(i) = \frac{(Accuracy_i)^\alpha}{\sum_{j=1}^{N}(Accuracy_j)^\alpha} \tag{5}$$

Donde α es un factor cuyo valor se ha determinado experimentalmente, resultando finalmente en $\alpha = 1$ como el valor que produce los mejores resultados para los datasets utilizados en los experimentos.

En cuanto al criterio de reemplazo, es inverso al de selección. Se trata de un reemplazo proporcional al fitness (inverso al accuracy) por medio del método de selección por rueda de ruleta de nuevo. Su fórmula se muestra a continuación:

$$P(i) = \frac{(f_i - min\_fitness)^\alpha}{\sum_{j=1}^{N}(f_j - min\_fitness)^\alpha} \tag{6}$$

De nuevo, el factor α se ha seleccionado en base a resultados experimentales y también ha resultado obtener el mejor rendimiento con $\alpha = 1$.

Gracias a estos criterios, se favorece la elección de los mejores individuos pero se permite perpetuar características de individuos menos adaptados para aumentar la variabilidad y favorecer el descubrimiento de nuevas soluciones en un espacio de búsqueda no explorado. Además, se incluye un parámetro $topN$ que indica cuántos de los mejores individuos se mantienen externos al reemplazo. Por ejemplo, con $topN = 3$ los 3 mejores individuos siempre estarán presentes en la población.

F. Visión Completa del NEGA

Una vez explicado cada elemento del NEGA de FakeNEAT, se pretende proporcionar una visión global del algoritmo para mejorar su comprensión. Esta sección proporcionará esta vista, además de compartir los hiperparámetros más importantes a la hora de ejecutar el algoritmo genético.

Los hiperparámetros más importantes, muchos explicados en secciones anteriores, se muestran en la Tabla I:

Estos hiperparámetros proporcionan la personalización necesaria para adaptarse a los distintos contextos posibles. Una vez conocidos los hiperparámetros disponibles, se muestra el algoritmo completo en 2:

Hiperparámetro	Rango	Descripción
hidden_layers	[1, ∞]	Número de capas ocultas
maxN	[1, ∞]	Máx. nº de neuronas por capa al inicio
pop_size	[1, ∞]	Tamaño de la población
num_gen	[1, ∞]	Máx. nº de generaciones/iteraciones
mutation_rate	[0, 1]	Proporción de individuos mutados
topN	[1, pop_size]	Nº de mejores soluciones mantenidas
crossovers	[1, ∞]	Nº de cruces por generación
limit	[0, 1]	Fitness de terminación temprana
classifier	bool	V si es una tarea de clasificación
acc	bool	V para fitness como pérdida en mini-train
lr	[0, 1]	Learning rate en mini-train
train$_iters$	[1, ∞]	Nº iteraciones en mini-train
eval$_iters$	[1, ∞]	Nº iteraciones en evaluación del fitness
batch_size	[1, ∞]	Tamaño del batch en mini-tran

TABLA I: Hiperparámetros del NeuroEvolution Genetic Algorithm

Algorithm 2 FakeNEAT NeuroEvolution Genetic Algorithm

Require: hiperparámetros de la Tabla I
  $population = crear\_poblacion(pop\_size)$
  $fitness, topN\_fitness, topN\_sol = eval\_poblacion(topN)$
  for $i \in [1, num\_gen]$ **do**
    $nuevos\_indiv = []$
    for $j \in [1, crossovers]$ **do**
      $parent_1, parent_2 = seleccion(2)$
      $child_1, child_2 = crossover(parent_1, parent_2)$
      $nuevos\_indiv = nuevos\_indiv \cup [child_1, child_2]$
    end for
    $selec\_añadir\_neuronas = seleccion(mutation\_rate * pop\_size)$
    $nuevos\_indiv = nuevos\_indiv \cup añadir\_neuronas(selec\_añadir\_neuronas)$
    $selec\_eliminar\_neuronas = seleccion(mutation\_rate * pop\_size)$
    $nuevos\_indiv = nuevos\_indiv \cup eliminar\_neuronas(selec\_eliminar\_neuronas)$

    $selec\_minitrain = seleccion(mutation\_rate * pop\_size)$
    $nuevos\_indiv = nuevos\_indiv \cup minitrain(selec\_minitrain)$
    $nuevos\_fitness = eval\_poblacion(nuevos\_indiv)$
    $topN\_fitness, topN\_sol = comparar\_topN(nuevos\_fitness, nuevos\_indiv)$

    $population = population \cup nuevos\_indiv$
    $fitness = fitness \cup nuevos\_fitness$
    $population, fitness = reemplazo()$
    if $min\_fitness < limit$ **then**
      $break$
    end if
  end for
  return $topN\_sol, topN\_fitness$

El algoritmo se muestra también a modo de esquema visual en la Figura 2.

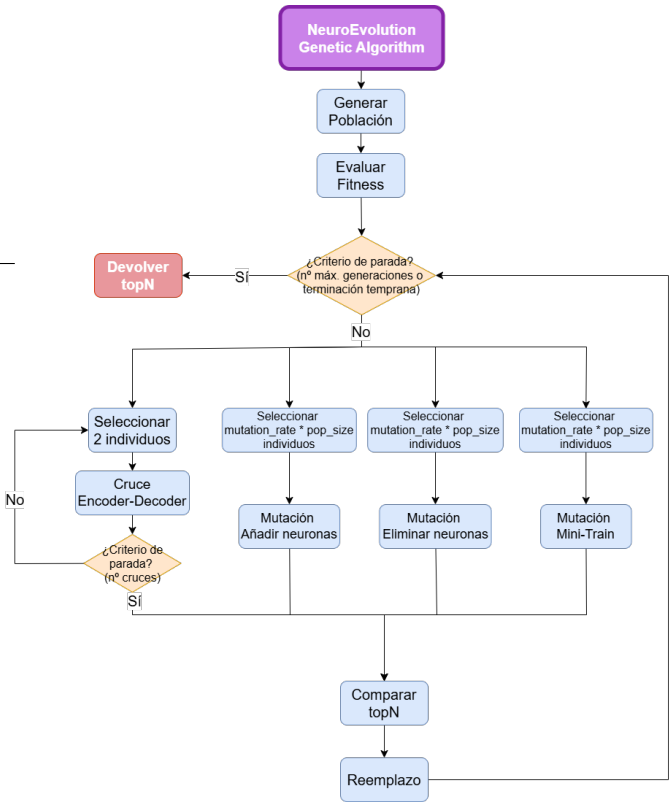


Fig. 2: FakeNEAT NeuroEvolution Genetic Algorithm

G. Experimentación

Una vez diseñado el algoritmo genético, es necesario ponerlo a prueba en casos reales para evaluar su capacidad. Para conseguirlo, se han empleado los *toy datasets* de la librería Scikit-learn por su facilidad de uso sin necesidad de preprocesado de los datos. De esta manera, podemos evaluar el algoritmo ante distintos contextos mayores adaptaciones de uno a otro que la modificación de hiperparámetros.

El dataset *Iris* ha servido como base para el proceso inicial de creación del algoritmo. Con él, se han realizado distintas pruebas variando los hiperparámetros para determinar cuáles son los mejores valores. Una vez se ha conseguido hacer funcionar el algoritmo en las mejores condiciones posibles, se ha repetido el proceso con los datasets *Digits* (sobre imágenes de dígitos manuscritos), *BreastCancer* (con datos sobre cáncer de mama) y *Wine* (una base de datos desbalanceada sobre vinos).

La Tabla II muestra las distintas variaciones llevadas a cabo en el dataset *Iris*, variando parámetros uno a uno mientras el resto se mantenían constantes para observar el efecto marginal. En esta tabla, *PI* corresponde con el tipo de inicialización de la población (S corresponde con simple y V con variado), *CO* con el número de capas ocultas, *maxN* con el número máximo de neuronas por capa al inicio, *PS* con el tamaño de población, *MR* es el ratio de mutación, *T* el tiempo de

ejecución en segundos, *F* el fitness en validación y *FT* el fitness en test. Todos los experimentos señalados en la tabla se han realizado con un único cruce por mutación y con un máximo de 2000 iteraciones. En las columnas de tiempo y fitness (tanto validación como test) se encuentran destacados en negrita los mejores valores.

PI	CO	maxN	PS	MR	T	F	FT
S	5	500	100	0.1	**45**	0.36	0.37
V	5	500	100	0.1	**45**	0.03	**0.0**
V	5	500	200	0.1	404	0.16	0.13
V	5	500	100	0.2	479	0.33	0.4
V	5	1000	100	0.1	330	0.18	0.13
V	10	500	100	0.1	75	**0.02**	**0.0**

TABLA II: Experimentos en Iris

La Figura 3 muestra la evolución del fitness a lo largo de las distintas iteraciones del algoritmo con la mejor configuración encontrada. Podemos apreciar que finaliza por terminación temprana en poco más de 350 iteraciones.

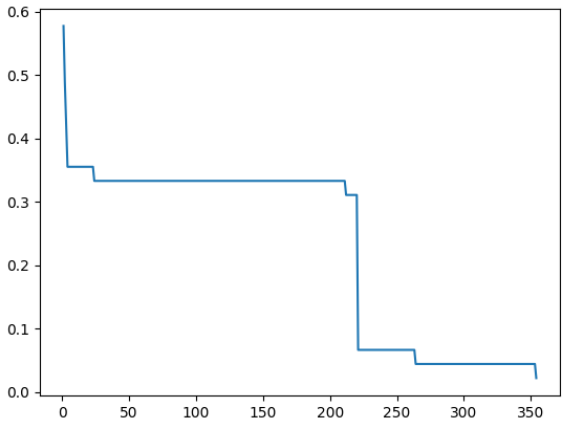


Fig. 3: Evolución del fitness en Iris

Podemos observar que, al tratarse de un problema complejo, encontramos grandes bajadas en ciertos puntos y grandes tramos sin mejoras, en lugar de pequeños descensos con poca separación entre ellos. Téngase en cuenta que en esta figura y las siguientes, el eje Y representa el fitness y el X el número de iteraciones llevadas a cabo.

Siguiendo un proceso similar, las Figuras 4, 5 y 6 muestran la evolución en los datasets de Digits, BreastCancer y Wine, respectivamente.

Observamos que el algoritmo consigue también muy buenos resultados en Digits y BreastCancer. Sin embargo, en Digits se produce una red neuronal sobre-entrenada ya que consigue un fitness inferior a 0.1 en validación, pero al evaluarlo en el conjunto de test observamos un fitness de 0.31. Esto nos indica la necesidad de añadir capas como las convolucionales si buscamos trabajar con imágenes. En el caso de Wine, observamos que no consigue resultados deseables, ya que se mantiene por encima del 18% de error desde la iteración

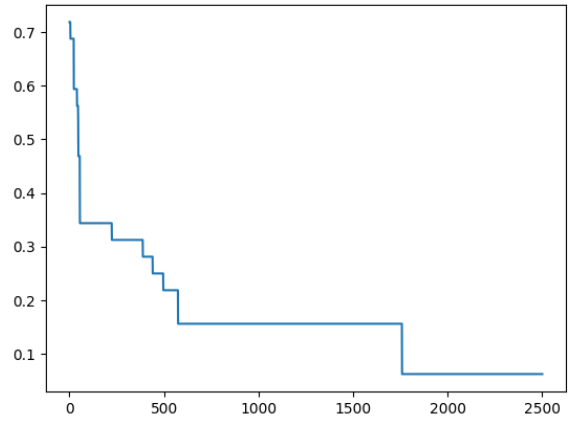


Fig. 4: Evolución del fitness en Digits (test: 0.31)

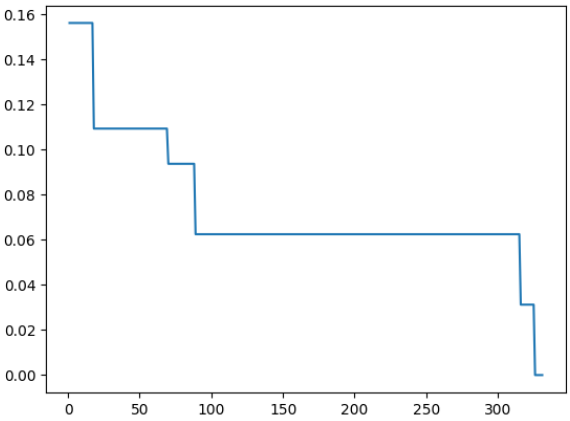


Fig. 5: Evolución del fitness en BreastCancer (test: 0.09)

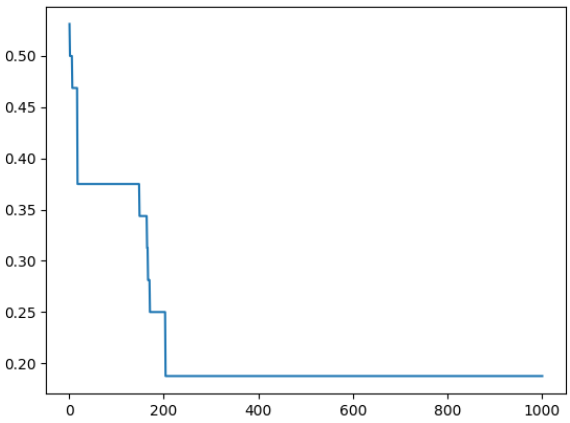


Fig. 6: Evolución del fitness en Wine (test: 0.34)

número 200 y no consigue mejorarla hasta que se termina la ejecución al alcanzar el número máximo de iteraciones (1000 en este caso).

IV. Solución mediante Enfriamiento Simulado

FakeNEAT no se limita únicamente a los algoritmos genéticos para encontrar la mejor solución posible, sino que además implementa un módulo de enfriamiento simulado (NeuroEvolution Simulated Annealing, NESA).

El enfriamiento simulado es una técnica algorítmica que se basa en la técnica empleada a la hora de forjar herramientas de metal. Este proceso de *annealing* se emplea cuando se enfría un material que ha transicionado al estado líquido tras ser calentado, para que vuelva a un estado cristalino. Este estado final será el de más baja energía u ordenación perfecta si la temperatura se reduce de la manera adecuada a lo largo del proceso.

El enfriamiento simulado consta de una búsqueda local, partiendo de una solución inicial y modificándola para obtener nuevas soluciones próximas en el espacio de búsqueda. Al inicio de la búsqueda, acepta soluciones peores a la inicial y a medida va avanzando reduce su tolerancia. Esto permite partir de una filosofía más centrada en *exploración*, y poco a poco transicionar a una estrategia centrada en *explotación*.

A. Solución inicial

La solución inicial es un elemento clave en un algoritmo de enfriamiento simulado, puesto que marcará el punto de inicio de nuestra búsqueda local. El NESA de FakeNEAT permite dos tipos de inicialización de la solución.

La primera opción consiste en proporcionar una solución externa al algoritmo. De esta manera, el programador tiene total control sobre su punto de partida, pudiendo elegir una solución lo más simple posible, una aleatoria o incluso una red previamente entrenada para un contexto similar, siendo compatible con un proceso similar al *fine-tuning*.

La segunda opción consiste en utilizar la solución resultante del NEGA. De esta manera, antes de ejecutar el NESA se buscará optimizar la solución mediante el algoritmo genético, para posteriormente partir de la mejor solución y explorar su vecindario. Por tanto, el algoritmo genético y el enfriamiento simulado no funcionan como dos técnicas separadas, sino que pueden combinarse para conseguir una solución mejor.

B. Generación del vecindario

La generación de vecinos determinará la forma en la que nuestro algoritmo explorará el espacio próximo de soluciones, generando vecinos nuevos alrededor de la mejor solución encontrada hasta el momento. Para ello, se debe seleccionar una estrategia de modificación de la solución actual que nos permita introducir cierta diversidad para encontrar nuevas soluciones semejantes.

En el caso de FakeNEAT, esta generación de nuevas soluciones a partir de la actual se centrará en aplicar las mutaciones del NEGA. Estas mutaciones generan variaciones de una única solución, por lo que son candidatas perfectas para este trabajo. En el NESA, nunca se aplicará más de una mutación al mismo tiempo para generar un nuevo vecino, sino que se seleccionará una de las tres de forma aleatoria cada vez que se desee encontrar un individuo nuevo de la vecindad.

C. Temperatura

La temperatura es el elemento clave en el que se basa el concepto del enfriamiento simulado. Se trata de un valor que se reducirá a lo largo de la ejecución, que al inicio permitirá una mayor *exploración* y que a medida se avance en el proceso hará la búsqueda más restrictiva, favoreciendo la *explotación*.

En el caso de la temperatura contamos con dos aspectos principales. El primero consiste en la estrategia a seguir a la hora de reducir la temperatura. En este punto se puede ser tan creativo como se desee, pero lo común es implementar una reducción exponencial o lineal. FakeNEAT implementa una reducción lineal de la temperatura, siguiendo siempre la misma pendiente a lo largo del entrenamiento. Esta pendiente se calcula como:

$$-\frac{T_{inicial} - T_{final}}{N^o\ iteraciones} \tag{7}$$

El segundo aspecto importante es la decisión sobre cuáles son la temperatura inicial y la temperatura fina (especialmente la inicial). Si contamos con temperaturas demasiado elevadas, corremos el riesgo de transicionar demasiado tarde a una estrategia de *explotación*, además de que permite seleccionar soluciones excesivamente poco adaptadas, con un resultado demasiado lejano del óptimo. Una buena práctica consiste en establecer la temperatura inicial con un orden de magnitud similar una variación común de nuestra función de fitness. Por ello, el mejor resultado de temperatura inicial en FakeNEAT se ha conseguido con un valor de 0.01.

D. Visión Completa del NESA

A continuación se muestran los detalles del algoritmo de enfriamiento simulado, NESA, de FakeNEAT como conjunto. Los hiperparámetros a modificar en este algoritmo se muestran en la Tabla III.

Hiperparámetro	Rango	Descripción
min_delta	[0, ∞]	Variación mínima aceptable
vecinos	[1, ∞]	nº vecinos por iteración
temp_ini	[0, ∞]	Temperatura inicial
temp_fin	[0, ∞]	Temperatura final
max_iters	[1, ∞]	Nº máx. de iteraciones
max_time	[0, ∞]	Tiempo máx. de ejecución (s)

TABLA III: Hiperparámetros del NeuroEvolution Simulated Annealing

El funcionamiento del NESA se detalla a continuación en el Algoritmo 3.

Algorithm 3 FakeNEAT NeuroEvolution Simulated Annealing

Require: hiperparámetros de la Tabla III
$T = T_0$
if use_NEGA **then**
$sol, fit = NEGA()$
end if
while $fit > min\_delta$ & $iteration < max\_iters$ & $T > T_{final}$ & $time < max\_time$ **do**
for $i \in [1, vecinos]$ **do**
$sol\_candidata, fit\_candidata = mutacion\_aleatoria(sol)$
$delta = fit\_candidata - fit$
if $delta < 0 \mid random() < e^{\frac{-delta}{T}}$ **then**
$sol = sol\_candidata$
$fit = fit\_candidata$
end if
end for
$T = T - \frac{T_0 - T_{final}}{N^o\ iteraciones}$
end while
$fit\_test = evaluar\_fitness\_test(sol)$
return sol, fit, fit_test

Es posible observar el algoritmo a modo de esquema visual en la Figura 7.

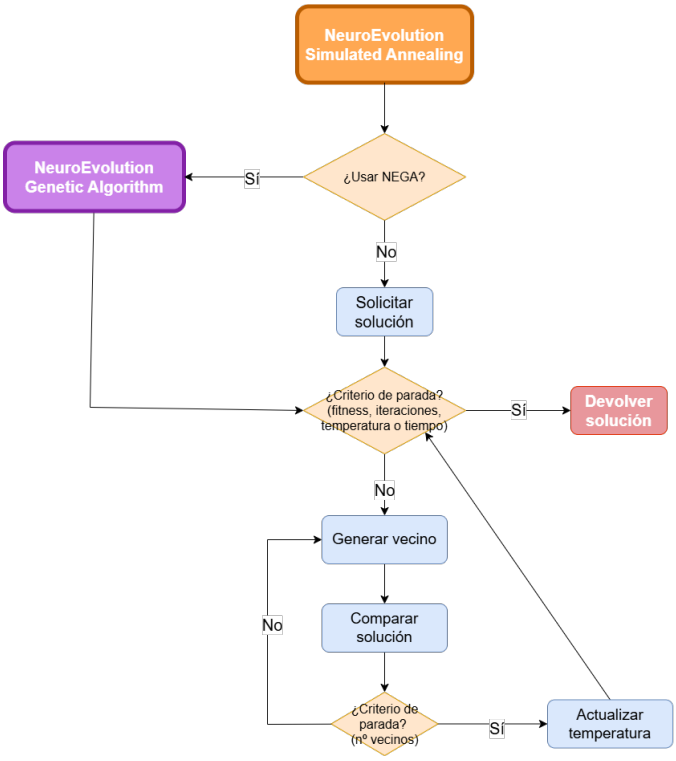


Fig. 7: NeuroEvolution Simulated Annealing

E. Experimentación

El algoritmo NESA se ha evaluado sobre los mismos datasets que el NEGA (Iris, Digits, BreastCancer, Wine) siguiendo el mismo procedimiento. La solución inicial del NESA se ha escogido como una solución *temprana* del NEGA debido a que únicamente con el algoritmo genético en varios datasets se conseguían resultados excelentes con poco margen de mejora. Con este procedimiento, ahorramos tiempo de ejecución del primer paso y conseguimos dejar un margen de mejora sobre el que evaluar nuestro algoritmo.

La Tabla IV muestra los resultados variando ciertos hiperparámetros del algoritmo sobre el dataset Iris mientras el resto de ellos se mantienen constantes. Todos estos experimentos se han realizado con una temperatura final de 0 y un número máximo de iteraciones de 3000.

Min.Delta	Vecinos	T_0	T (s)	Fit	Fit Test
0.01	50	0.01	309	0.2	0.3
0.01	50	0.1	235	0.64	0.63
0.01	50	0.05	317	0.36	0.36
0.01	100	0.01	688	0.2	0.2
0.01	20	0.01	**133**	**0.18**	**0.13**
0.1	20	0.01	**133**	0.33	0.4

TABLA IV: Experimentos en Iris

La Figura 8 muestra la evolución del fitness en el dataset Iris con la mejor configuración encontrada.

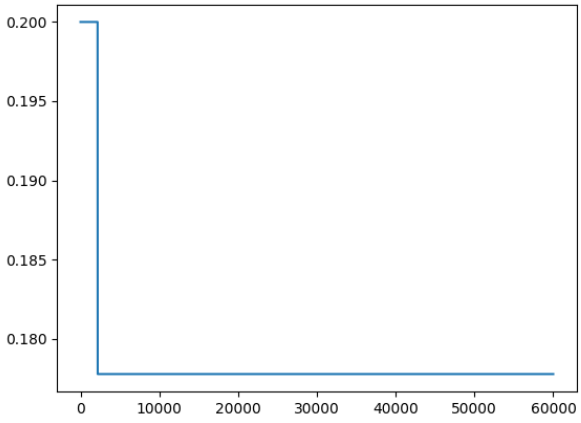


Fig. 8: Evolución del fitness en Iris

Podemos observar cómo el algoritmo consigue mejorar la solución de un fitness de 0.2 a uno de 0.177 y ya no logra mejorarlo a lo largo de las iteraciones restantes. Cabe destacar que el eje Y representa el fitness y el X representa los vecinos generados en total, es decir, $vecinos * iteraciones$.

Continuando con el mismo proceso, las Figuras 9, 10 y 11 muestran la evolución de los resultados en Digits, BreastCancer y Wine respectivamente.

Observamos, por lo general, un funcionamiento menos acertado que el algoritmo genético, a excepción de Wine. Un aspecto importante es que se está generando mayor diferencia

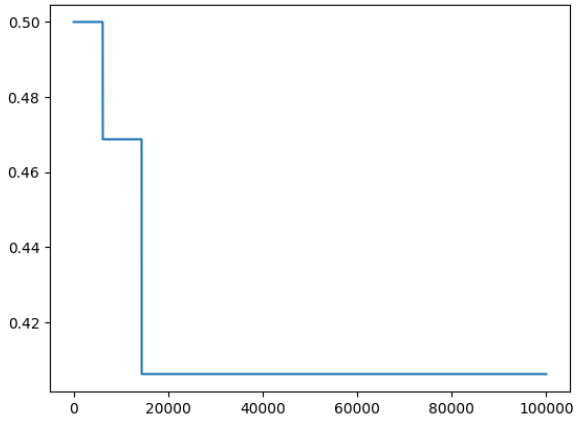


Fig. 9: Evolución del fitness en Digits (0.45 en test)

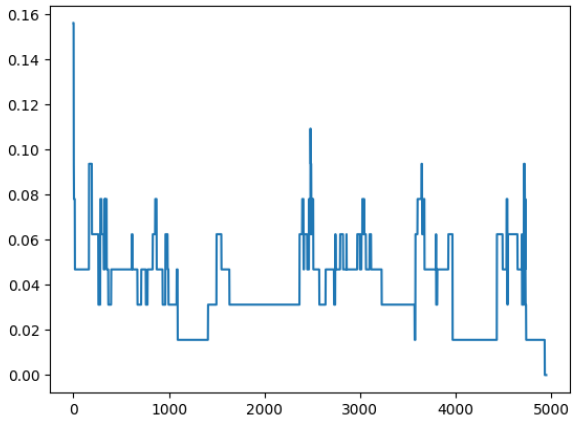


Fig. 10: Evolución del fitness en BreastCancer (0.12 en test)

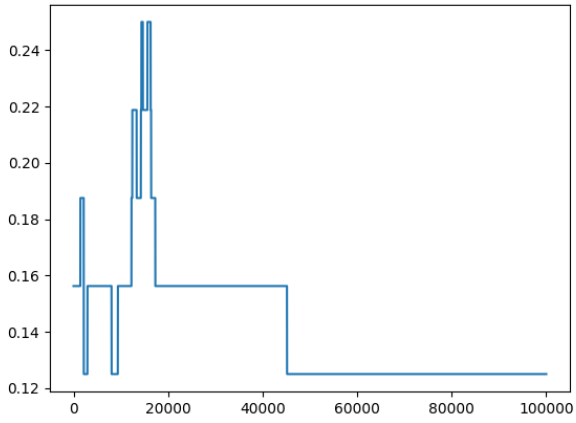


Fig. 11: Evolución del fitness en Wine (0.3 en test)

entre el fitness en validación y en test. Por último, observamos que para imágenes como el dataset Digits el rendimiento es significativamente erróneo.

V. Detalles de Implementación

Esta sección pretende compartir todos los detalles posibles de implementación de FakeNEAT a nivel práctico. Esto incluye el lenguaje de programación y librerías utilizadas, así como la organización del código y su utilización. De esta forma, no se consigue únicamente una mayor comprensión de los algoritmos sino que facilita una posterior contribución, su uso por parte de terceros para replicar los resultados o su adaptación a nuevos contextos.

El lenguaje escogido para la implementación del algoritmo es Python. Pese a que se planteó la elección de C++ como lenguaje debido a su velocidad de ejecución (clave en este tipo de algoritmos), finalmente se decidió Python por su amplio desarrollo en tratamiento de redes neuronales. Para ello, la librería utilizada fue PyTorch. El código está disponible en GitHub y todas las librerías empleadas en el desarrollo del código se muestran en la Figura 12.

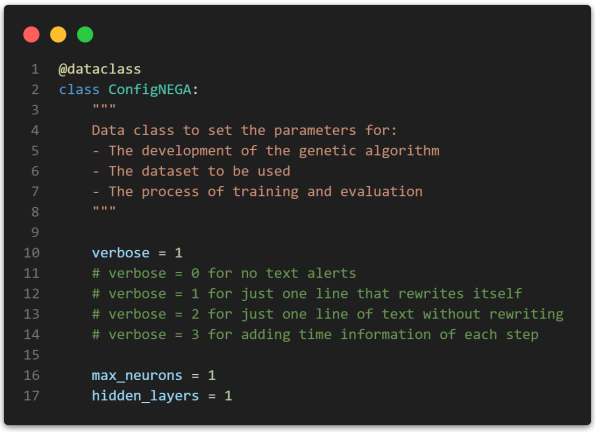
```python
#Librerías para manejo de redes neuronales
import torch
from torch import nn
from torch.nn import functional as F
#Para operaciones matemáticas
import numpy as np
import math
#Manejo de estructuras de datos
from dataclasses import dataclass
from sklearn import datasets
import random
import copy
import pandas as pd
import time #Control temporal
import matplotlib.pyplot as plt #Gráficos
```

Fig. 12: Librerías empleadas

Los hiperparámetros de los algoritmos se controlan desde dos clases distintas de tipo *dataclass* (similar a un struct en C++) para facilitar su modificación. La Figura 13 muestra el formato de esta clase junto con los primeros hiperparámetros que es posible manipular.

A continuación, se ha optado por un estilo de programación orientado a objetos de forma que se consiga una estructura modular que facilite si manipulación. De esta manera, contamos con 5 clases distintas:

- **LinearModule**: Hereda de *torch.nn.Module* y genera un bloque lineal de la red neuronal, es decir, una capa densa lineal y una función de activación ReLU.



```python
@dataclass
class ConfigNEGA:
    """
    Data class to set the parameters for:
    - The development of the genetic algorithm
    - The dataset to be used
    - The process of training and evaluation
    """

    verbose = 1
    # verbose = 0 for no text alerts
    # verbose = 1 for just one line that rewrites itself
    # verbose = 2 for just one line of text without rewriting
    # verbose = 3 for adding time information of each step

    max_neurons = 1
    hidden_layers = 1
```

Fig. 13: Snippet de la configuración del NEGA

```python
##############################
#         MAIN LOOP          #
##############################

if __name__ == "__main__":
    ga = NEGA()
    best_population, best_fitness = ga()
    sa = NESA()
    sol, fit = sa()
```

Fig. 14: Bucle principal

- **Network**: Hereda también de *torch.nn.Module*. Genera la red neuronal completa a partir del gen correspondiente, que consiste de una capa de entrada, un número detectado automáticamente de bloques lineales con tantas neuronas en cada una como indique el gen, una capa de salida y, opcionalmente si se trata de un problema de clasificación, la función softmax a la salida.
- **Individual**: Una clase con tres atributos: el gen, la red (módulo Network) y el fitness. Responde a una solución de la población. Su método *__call__* se ha sobreescrito para llamar al *forward* de la red neuronal.
- **NEGA**: Incluye la implementación completa del NEGA. Esto quiere decir que cuenta con cada método necesario para llevarse a cabo (y alguno no utilizado finalmente, como la destrucción de un porcentaje de la población y su reemplazo aleatorio cada cierto número de iteraciones). Su método *__call__* tambien ha sido sobreescrito para inicial la ejecución del algoritmo. Uno de sus parámetros opcionales es almacenar una gráfica de la evolución del fitness.
- **NESA**: Corresponde con la implementación del NESA, exactamente de la misma manera que el NEGA.

De esta manera, el bucle principal de ejecución es muy sencillo. El usuario simplemente necesita haber editado las dataclasses *ConfigNEGA* y *ConfigNESA*, inicializar la clase *NEGA* y/o *NESA* (según considere), y hacer una llamada al objeto. El proceso se muestra en la Figura 14.

VI. Conclusiones

Esta sección describe las conclusiones extraídas en base a la experiencia y experimentación a lo largo del desarrollo de FakeNEAT.Se analizan aspectos como la adecuación de estos algoritmos para el problema al que pretenden dar solución, el grado de complicación a la hora de adaptarlos a cada contexto o la comparación entre el algoritmo genético y el enfriamiento simulado, así como propuestas de otros algoritmos que podrían ser de ayuda.

En primer lugar, podemos observar que FakeNEAT, pese a estar en un nivel de desarrollo inicial con ciertas limitaciones que podrían suponer una gran mejora en cuanto a resultados, es capaz de adaptarse a distintos datasets y conseguir muy buenos resultados (incluso un 0% de error en test en algunos casos) con un tiempo de ejecución reducido (por debajo del minuto). Este aspecto es interesante, ya que en lugar de estar entrenando una única red neuronal se están entrenando decenas e incluso cientos de ellas al mismo tiempo y probando combinaciones de arquitecturas en el mismo grado de magnitud, y aún así se obtienen tiempos similares a un entrenamiento tradicional de una sola red, muy inferiores a métodos iterativos como los tuners. Si, con el tiempo, FakeNEAT se adaptara para añadir nuevas funciones de activación, variar el número de capas, añadir bloques especiales (atención o convolucionales), conexiones residuales o aumento de datos, las posibilidades son inimaginables.

En segundo lugar, cabe destacar la sencillez con la que se adapta a distintos datasets una vez formateados (en forma de columnas de variables explicativas y explicadas). Simplemente modificando los valores de los hiperparámetros, está listo para realizar ensayos en nuevos conjuntos de datos sin modificación del código en absoluto. Si se añadiesen todas las opciones de mejora mencionadas en el apartado anterior, debería trabajarse para mantener esta modularidad.

Es interesante mencionar la importancia del cruce observada en el algoritmo genético. Este problema ha demostrado requerir de un alto grado de elitismo a la hora de modificar las soluciones iniciales, y el cruce permite una gran modificación de la red al mismo tiempo que reduce la aleatoriedad de los valores añadidos, puesto que parten de otra red con buenos resultados (probablemente, debido a la selección). Como se puede observar en versiones anteriores del código de GitHub, la adición del método de cruce fue un punto de inflexión a la hora de obtener mejores resultados, y probablemente sea también la razón por la que el algoritmo genético funciona mejor que el enfriamiento simulado.

Una vez enfrentado el problema, otro algoritmo interesante

a aplicar sería una combinación del algoritmo de colonia de hormigas y el de enjambre de partículas. El algoritmo de colonia de hormigas debería centrarse en ajustar variables discretas, como el número de neuronas por capa o número de capas, mientras que el enjambre de partículas se debería de centrar en optimizar los pesos de la red. Sin embargo, la aplicación del enjambre de partículas sería especialmente complejo entre redes con número distinto de parámetros.

BIBLIOGRAFÍA

[1] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. nature, 323(6088), 533-536.

[2] Vaswani, A. (2017). Attention is all you need. Advances in Neural Information Processing Systems.

[3] Kirk, R., Mediratta, I., Nalmpantis, C., Luketina, J., Hambro, E., Grefenstette, E., & Raileanu, R. (2023). Understanding the effects of rlhf on llm generalisation and diversity. arXiv preprint arXiv:2310.06452.

[4] Kingma, D. P. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[5] Pagliardini, M., Ablin, P., & Grangier, D. (2024). The ademamix optimizer: Better, faster, older. arXiv preprint arXiv:2409.03137.

[6] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks Through Augmenting Topologies," Evolutionary Computation Journal, vol. 10, pp. 99–127, 2002.

[7] Stanley, K. O., D'Ambrosio, D. B., & Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. Artificial life, 15(2), 185-212.