TIA

Soluciones metaheurísticas para problemas complejos

Jaime Ferrando Huertas

Octubre 2020

Contents

1 Introducción

Esta practica consiste en la evaluación de técnicas metaheurísticas para la resolución de problemas complejos. La mayoría de estas técnicas metaheurísticas se basan en conceptos que encontramos en la naturaleza y en la teoría de la evolución, suelen ser técnicas generalistas que no están diseñadas a medida para un problema en especifico por lo que deben ser adaptadas para cada uso. Son usadas en problemas en las que no existe una solución óptima y el mero hecho de encontrar una solución es complicado. En esta practica se han implementado algoritmos genéticos y técnicas de entrenamiento en frío.

1.1 Problema

El problema escogido para la evaluación de estas técnicas ha sido Cart-poleV1 **?**, un problema definido por OpenAI para reinforment learning. Este problema consiste en darle mover una plataforma en el espacio para que el palo que se encuentra encima de ella se mantenga equilibrado y no caiga a ninguno de los lados. Se considera resuelto si consigues mantenerlo estable durante 25 segundos.

Este problema suele ser usado para algoritmos de reinforcement learning con técnicas de aprendizaje profundo, donde la red neuronal se enfrenta al problema de manera iterativa y actualiza sus pesos dependiendo de sus resultados en cada iteración. Nosotros hemos propuesto un modo alternativo de uso de estas redes neuronales adaptado a métodos metaheurísticos donde tratamos una red neuronal como una solución definida por los valores de sus pesos en cada neurona, es decir que la red es una matriz de valores de pesos. En vez de entrenar una red neuronal y viajar por el gradiente de la función objetivo nosotros aplicamos estas técnicas metaheurísticas a la red neuronal para viajar entre múltiples soluciones(redes neuronales).

2 Técnicas usadas

2.1 Algoritmo genético

Un algoritmo genético es una técnica metaheurística inspirada en el proceso de selección natural,pertenece a la clase más amplia de algoritmos evolutivos. Los algoritmos genéticos se utilizan comúnmente para generar soluciones de alta calidad para problemas de optimización y búsqueda basándose en operadores inspirados biológicamente como mutación, cruce y selección. Los algoritmos genéticos hacen uso de estas teorías biológicas para crear poblaciones de posibles soluciones y iterar sobre ellas con las teorías evolutivas para conseguir soluciones al problema planteado. El problema se debe adaptar para poder crear soluciones únicas y aleatorias que se puedan combinar con otras o sufrir modificaciones. También se debe crear una función *fitness* que evalué la viabilidad de una solución, esta función se usara junto con las teorías biológicas para la selección de las soluciones.

Podemos definir un algoritmo genético como una función que inicializa una población aleatoria de cromosomas(soluciones) sobre las cuales se evalúa su viabilidad con la función *fitness*. Una vez con los resultados de la población se seleccionan un conjunto de soluciones mediante un criterio definido. Una vez seleccionadas un subconjunto de soluciones se aplican cruces entre ellas, este cruce también puede ser modificado a criterio definido. Tras el cruce se aplican mutaciones sobre la población (modificaciones aleatorias de los cromosomas/codificación de la solución) para evitar estancamientos y favorecer la búsqueda de nuevas soluciones. Por ultimo se reemplazan soluciones de la población inicial con las que acabamos de obtener mediante una función de reemplazo que toma en cuenta sus valores de fitness de las soluciones para seleccionar cuales son descartadas. Este proceso es iterativo hasta que satisfacemos una condición de parada en la evaluación de la población, esta condición suele ser si se ha encontrado una solución con un valor de *fitness* mínimo.

A continuación se describe nuestro enfoque para cada una de estas partes del algoritmo genético.

2.1.1 Codificación de las soluciones

Los algoritmos genéticos codifican soluciones en cromosomas, el símil para nuestras redes neuronales son los pesos que encontramos en cada neurona de la red. Cada red consiste de un conjunto de capas de neuronas por lo que podemos codificar nuestra red como una matriz de valores numéricos donde cada fila corresponde a una capa y los valores de cada fila son los pesos de la misma capa.

2.1.2 Población inicial

La población inicial es una lista N de redes neuronales representadas por su matriz de cromosomas. Estas redes neuronales se han creado de manera aleatoria donde los pesos siguen una distribución uniforme $[0, 1]$. El tamaño de N es uno de los parámetros a evaluar de nuestro algoritmo genético, un valor muy alto incrementara el tiempo requerido para evaluar cada generación de soluciones pero nos dará mas posibilidades de encontrar soluciones aleatorias gracias a mutaciones ya que contamos con mas individuos.

2.1.3 Fitness

La función de *fitness* la hemos definido como los segundos que una solución es capaz de mantener la el palo sobre la plataforma del problema estable, para ello hacemos una llamada a la red neuronal en cada paso donde se le pasa una serie de características del entorno como posición de la plataforma y orientación del palo.

Se considera solución óptima cuando consigue aguantar 25 segundos. Hemos considerado un experimento resuelto cuando 5 soluciones son óptimas.

2.1.4 Selección

La función de selección determina las soluciones de la población actual que van a ser elegidos para la función de cruce. Normalmente esta función recibe el resultado de la función *fitness* para cada solución y elige los candidatos en base a esta. Nosotros hemos evaluado dos funciones de selección, proceso elitista y proceso de selección proporcional/ruleta.

El proceso elitista selecciona las $0.2 * N$ mejores soluciones de la población. El proceso de selección proporcional crea una tabla de probabilidades donde cada solución es asignada una probabilidad en valor de su resultado de la función fitness, entonces selecciona $0.6 * N$ soluciones dadas las probabilidades obtenidas.

2.1.5 Cruce

La función de cruce recibe las soluciones previamente seleccionadas y crea cruces en base a estas. Nosotros hemos creado tres funciones de cruce a evaluar para este problema.

1. Cross: Técnica de cruzado de soluciones donde dadas dos soluciones A y B partimos su matriz de cromosomas en dos para crear soluciones hijas AB y BA.

2. Cross-old: Técnica de cruzado con antigüedad donde dadas dos soluciones A y B crea una única solución AB y devuelve AB mas un padre A o B elegido aleatoriamente.

3. None: Técnica de cruzado nula donde devolvemos los mismos padres A y B sin cruce alguno, esta técnica tiene peligro de caer en mínimos locales ya que no genera soluciones nuevas y solo itera sobre las mismas continuamente.

Las dos ultimas técnicas de cruzado ejercen en parte como función de reemplazo ya que devuelve soluciones antiguas como nuevas.

2.1.6 Mutación

Las mutaciones nos ayudan a evitar estancamientos en nuestras soluciones en mínimos locales añadiendo aleatoriedad en los cromosomas. Nosotros hemos implementado nuestra mutación de cada cromosoma como $cromosoma+ = poder_mutación * random(0, 1)$ con un poder de mutación de 0.02.

2.1.7 Reemplazo

Una vez tenemos los cruces ya mutados es hora de reemplazar la población antigua con nuestras nuevas soluciones, para ello hemos probado tres metodologías.

1. Reemplazo estado-estacionario, donde solo se reemplazan las L peores soluciones según su valor de fitness con el conjunto de soluciones mutadas de longitud L.

2. Reemplazo generacional, donde se reemplazan todas las soluciones de la población con las que acabamos de obtener.

3. Doom-day, donde solo mantenemos la mejor solución resultante del cruce y generamos nuevas soluciones aleatorias para el resto de individuos en la población.

Si en algún caso no tenemos suficientes soluciones en la población para llegar al numero de soluciones iniciales N se duplican soluciones aleatorias de la población.

2.1.8 Entrenamiento e implementación

Hemos usado el lenguaje de programación python para el total desarrollo de este proyecto. Se han usado librerías de terceros como *Pytorch* para redes neuronales y *Openai-gym* para la ejecución del problema. Se ha paralelizado el código para evaluar el fitness de la población y el código para aplicar mutaciones a la población, esto nos ha dado grandes incrementos en velocidad de computo ya que evaluar el fitness de toda la población es la parte mas costosa ya que se incrementa su tiempo de computo linealmente con el valor devuelto.

Para evaluar como afecta el tamaño de la población N se ha entrenado la combinación $Selección :, Cruce :, Reemplazo :$ con distintos valores de .

Hemos entrenado todas las combinaciones posibles para las funciones de selección, cruce y reemplazo para distintos valores N: 20,50,100,200,500,1000,2000,5000. La ejecución de cada combinación era cancelada si la media del valor fitness de la población no incrementa en 2 segundos en las ultimas 150 generaciones.

2.2 Enfriamiento Simulado

La segunda técnica metaheurística que hemos evaluado para este problema ha sido el enfriamiento simulado, esta técnica se inspira en conceptos físicos. La técnica de enfriamiento se utiliza en metalurgia para conseguir materiales de mayor calidad, su fundamento es el calentar el material a temperaturas muy altas y bajar la temperatura lentamente. Al calentar el material hacemos que las partículas del mismo se desordenen y aumente la entropía pero cuando lo dejamos enfriar lentamente estas partículas se ordenan poco a poco y se consigue una estructura mas cristalina y de mayor calidad.

Cuando aplicamos esta teoría a la búsqueda de soluciones de nuestro problema podemos crear una función de entrenamiento. En esta función usaremos una solución inicial en analogía al material y reduciremos la temperatura cada iteración hasta encontrar una solución óptima o alcanzar el numero máximo de iteraciones y devolver la mejor solución encontrada. Empezamos con una solución inicial generada de manera aleatoria y en cada iteración crearemos una solución cercana a esta (similar a la mutación en los algoritmos genéticos) que aceptaremos en función de la temperatura y en caso de aceptar la solución sustituiremos la solución inicial por la nueva solución . Aceptaremos la solución

cuando nos de una mejora en la función de fitness, en el caso contrario aun podemos aceptar la solución en función de la probabilidad:

$$p(aceptar) = e^{\frac{(fitness(new) - fitness(sol)) - fitness(sol)}{T}}$$

Esta formula nos dará mayores probabilidades de aceptar soluciones que empeoren el valor de fitness cuanto mayor sea la temperatura. La temperatura se ira reduciendo a medida que buscamos soluciones con la siguiente formula:

$$T = T/(i + K * T)$$

Donde K es una constante e i es el numero de la iteración en la que nos encontramos.

El algoritmo de enfriamiento simulado nos permitirá explorar soluciones no óptimas en las iteraciones iniciales y a medida que disminuye la temperatura nos centraremos en soluciones que nos reportan una mejora en el valor de fitness.

2.2.1 Entrenamiento e implementación

Hemos usado el lenguaje de programación python para el total desarrollo de este proyecto. Hemos usado *Openai-gym* para la ejecución del problema y la implementación del enfriamiento simulado ha sido posible con las funciones nativas de python. Hemos generado la solución inicial de manera aleatoria y hemos generado las soluciones vecinas de la misma forma que hemos aplicado mutaciones en el algoritmo genético (aplicando la siguiente formula a todos los pesos de la red neuronal $peso+ = 0.02 * random(0, 1)$.

Hemos evaluado distintos parámetros en nuestra función de enfriamiento simulado: K, T ejecutando veinte experimentos para cada combinación de parámetros y devolviendo la media de iteraciones (si converge) que ha necesitado la función para alcanzar un valor de fitness ¿250. Los valores de K y T que hemos usado para generar todas las combinaciones posibles han sido: $T[1, 10, 100, 1000, 10000]$ y $K[0.001, 0.01, 0.1, 1, 10]$. Hemos puesto el limite de iteraciones para encontrar una solución óptima en 10.000.

3 Resultados

3.1 Algoritmo genético

Para el algoritmo genético hemos evaluado todas las combinaciones de la función de selección,unión y reemplazo con distintos tamaños de poblacion N. Aquí se encuentra la tabla con el numero de iteraciones que ha necesitado cada combinación para encontrar una solución aceptable al problema en función del tamaño de la población N, con valor -1 si no lo consigue.

F.selección	F.Cruce	F.reemplazo	N=20	N=50	N=100	N=200	N=500	N=1000
S.Proporcional	Cross	Generacional	438.0	331.0	165.0	125.0	116.0	119.0
	None	Generacional	407.0	269.0	175.0	145.0	116.0	111.0
	Cross-old	Generacional	469.0	219.0	240.0	157.0	115.0	100.0
Elitista	Cross	Generacional	313.0	163.0	147.0	108.0	126.0	107.0
	None	Generacional	218.0	180.0	140.0	146.0	100.0	91.0
	Cross-old	Generacional	441.0	202.0	157.0	140.0	100.0	106.0
S.Proporcional	Cross	E.Estacionario	278.0	157.0	136.0	123.0	110.0	112.0
	None	E.Estacionario	327.0	209.0	150.0	126.0	95.0	106.0
	Cross-old	E.Estacionario	316.0	153.0	149.0	138.0	110.0	104.0
Elitista	Cross	E.Estacionario	200.0	165.0	135.0	145.0	120.0	85.0
	None	E.Estacionario	245.0	174.0	148.0	122.0	112.0	80.0
	Cross-old	E.Estacionario	240.0	147.0	148.0	123.0	112.0	114.0
S.Proporcional	Cross	Doom-day	NaN	NaN	NaN	NaN	NaN	NaN
	None	Doom-day	NaN	NaN	NaN	NaN	NaN	NaN
	Cross-old	Doom-day	NaN	NaN	NaN	NaN	NaN	NaN
Elitista	Cross	Doom-day	NaN	NaN	NaN	NaN	NaN	NaN
	None	Doom-day	NaN	NaN	NaN	NaN	NaN	NaN
	Cross-old	Doom-day	NaN	NaN	NaN	NaN	NaN	NaN

Para evaluar como afecta el tamaño de la población podemos ver la media de iteraciones necesarias entre todas las combinaciones para cada tamaño de N. También añadimos el numero de soluciones visitadas (numero de soluciones sobre las que hemos evaluado el fitness) como N * Numero de iteraciones medio.

N	Iteraciones	Soluciones visitadas
20	324.33	6480
50	197.41	9850
100	157.50	15700
200	133.16	26600
500	111.00	55500
1000	102.91	118000

Podemos ver como a medida que aumentamos N se reduce el numero de iteraciones necesario ya que al contar con una población mayor visitamos mas soluciones en cada iteración. Sin embargo el aumentar N también aumenta considerablemente el tiempo de computo necesario para ejecutar el algoritmo y el numero de soluciones visitadas, hemos encontrado N=100 como valor óptimo para nuestro problema y nuestra capacidad de computo. Procedemos a evaluar ahora las distintas combinaciones.

Ahora nos podemos centrar en ver como afectan las distintas combinaciones en nuestros resultados. Volvemos a agrupar todas las combinaciones y a calcular valores medios de iteraciones necesarias para conseguir una solución óptima. Empezamos mirando a la función de reemplazo

F.Reemplazo	N=20	N=50	N=100	N=200	N=500	N=1000
Doom-day	NaN	NaN	NaN	NaN	NaN	NaN
Generacional	381.00	227.33	170.66	136.83	112.16	105.66
E.Estacionario	267.66	167.500	144.33	129.50	109.83	100.16

La función de doom-day no converge con ningún tamaño de población, eso se debe a que al quedarnos con 1 única solución cada iteración y crear el resto aleatoriamente le damos mucha influencia a las soluciones aleatorias y en nuestro caso para las redes neuronales estas tienen muy poca probabilidad de reportar un valor de fitness alto. Entre la generacional y state podemos apreciar que la state necesita menos iteraciones para cada tamaño de población por lo que es superior para este problema. Pasamos ahora a la función de unión:

F.Unión	N=20	N=50	N=100	N=200	N=500	N=1000
Cross	307.25	204.00	145.75	125.25	118.00	105.75
Cross-old	366.50	180.25	173.50	139.50	109.25	106.00
None	299.25	208.00	153.25	134.75	105.75	97.00

No podemos observar grandes diferencias entre las distintas funciones pero nos atrevemos a decir que la función de unión nula es la que mejores resultados de media. Esto significaría que las uniones de redes neuronales tal y como las hemos definido en nuestro problema no reportar mejores resultados, probablemente sea debido a que los pesos de una red neuronal están altamente correlados con el valor de los otros pesos y al generar cruces y mezclar pesos perdemos esa correlación y por lo tanto empeoramos la búsqueda. Pasamos ahora a la función de selección:

F.Selección	N=20	N=50	N=100	N=200	N=500	N=1000
S.Proporcional	372.50	223.00	169.16	135.66	110.33	108.66
Elitista	276.16	171.83	145.83	130.6	111.66	97.16

Se observa claramente como la función elitista nos da mejores resultados en nuestro problema, esto tiene sentido con el resto de resultados que hemos visto hasta ahora donde vemos peores resultados en las funciones que dan mas peso a nuevas soluciones generadas de manera aleatoria y no a soluciones producto de una solución anterior que tenia valor de fitness alto.

3.2 Enfriamiento simulado

Para el enfriamiento simulado hemos evaluado todas las combinaciones de temperatura y K para los valores $T[1, 10, 100, 1000, 10000]$ y $K[0.001, 0.01, 0.1, 1, 10]$. Se han ejecutado 100 experimentos para cada combinación y hemos agregado los resultados en la media de iteraciones necesarias para alcanzar una solución óptima y fitness medio obtenido. Cabe recordar que en enfriamiento simulado el numero de soluciones visitadas es igual al numero de iteraciones necesarias

por lo que esta columna no se incluye en los resultados de esta sección. Los resultados se pueden observar en la siguiente tabla.

Temperatura	k	Iteraciones	Fitness
1	0.001	1100.100000	268.600000
1	0.010	1230.066667	273.355556
1	0.100	1299.366667	275.600000
1	1.000	1195.066667	275.133333
1	10.000	1242.166667	274.822222
10	0.001	1267.100000	273.033333
10	0.010	1194.266667	270.688889
10	0.100	1236.700000	268.766667
10	1.000	1290.166667	269.533333
10	10.000	1163.566667	273.744444
100	0.001	1588.200000	276.144444
100	0.010	1421.733333	272.777778
100	0.100	1165.800000	275.322222
100	1.000	1492.433333	271.933333
100	10.000	1386.300000	260.822222
1000	0.001	1388.233333	275.700000
1000	0.010	1181.533333	277.755556
1000	0.100	1233.400000	278.577778
1000	1.000	1447.233333	269.500000
1000	10.000	1467.600000	271.366667
10000	0.001	1968.366667	264.011111
10000	0.010	1155.633333	278.722222
10000	0.100	1358.900000	273.311111
10000	1.000	1377.800000	274.444444
10000	10.000	1362.000000	282.544444

Podemos observar como todas las combinaciones han sido capaces de encontrar una solución con valor de fitness ¿250 (aguanta 250milisegundos con el palo recto). Vamos a indagar un poco mas en como T y K afectan el coste de la función y cuantas soluciones deben visitar. La mejor solución se da cuando T=1 y K=0.001, veamos como afectan estos parámetros en el numero de iteraciones necesario para poder entender el porque esta combinación nos da los mejores resultados. Si agrupamos los resultados en función de K y hacemos la media de los distintos experimentos de temperatura:

K	Iteraciones	Fitness
0.001	1462.400000	271.497778
0.010	1236.646667	274.660000
0.100	1258.833333	274.315556
1.000	1360.540000	272.108889
10.000	1324.326667	272.660000

Observamos que hay cierta tendencia a que cuanto mas pequeño es K menos iteraciones necesita nuestra función a excepción de K=0.001 donde creemos que el valor es demasiado pequeño y reduce la temperatura drásticamente a medida que aumentan las iteraciones. Recordemos que la constante K es multiplicada por la temperatura en la formula para calcular la temperatura dada una iteración i por lo que cuanto mas pequeña sea mas se reducirá nuestra temperatura. Si ahora agrupamos los resultados en función de T y hacemos la media de los distintos experimentos de K:

Temperatura	Iteraciones	Fitness
1	1213.353333	273.502222
10	1230.360000	271.153333
100	1410.893333	271.400000
1000	1343.600000	274.580000
10000	1444.540000	274.606667

Volvemos a observar una tendencia parecida a lo que vimos en K, cuanto mas pequeña sea nuestra temperatura (menos soluciones no óptimas que admitiremos) menos iteraciones necesita nuestra función para alcanzar una solución óptima. Esto hila con nuestra conclusión del efecto del valor K ya que menores valores de K y T ayudan a reducir la temperatura rápidamente y solo aceptar soluciones vecinas óptimas.

Los resultados vistos en K y T concuerdan con que la combinación que es capaz de encontrar una solución óptima mas rápidamente sea T=1 y K=0.001, con estos resultados podemos pensar que nuestro problema no se beneficia mucho de soluciones aleatorias no óptimas.

4 Conclusiones

Primero nos gustaría agradecer la posibilidad de proponer trabajos en la asignatura de TIA ya que nos ha dado la posibilidad de ver una aplicación de técnicas metaheurísticas a un entorno como las redes neuronales en las que prevalece otro tipo de entrenamiento o busqueda de soluciones. En cuanto a los resultados obtenidos por los algoritmos genéticos y el entrenamiento en frío podemos decir que el entrenamiento en frío es superior a los algoritmos genéticos en cuanto a numero de soluciones visitadas. Entrenamiento en frío necesita visitar entre 900-1300 soluciones para encontrar una solución óptima mientras que el algoritmo genético necesita mínimo 4000 soluciones (N=20, Elitista, None, Generacional) en su mejor entrenamiento y aumenta exponencialmente a medida que aumentamos N.

También mencionar que nos ha sorprendido que los dos enfoques fueran capaces de encontrar la solución óptima para el problema, no esperábamos que fueran capaces dado la talla del problema. Intentamos aumentar la talla del problema pero ha sido complicado debido a estar limitados al framework de

OpenAI, probamos a aumentar el numero de capas de la red pero se seguían encontrando las soluciones óptimas con no mucha diferencia al caso base. Creemos que para encontrar un problema de talla mayor deberíamos haber usado otro enfoque o problema y habría supuesto un cambio de framework por lo que es una gran opción para trabajos futuros.