

# **Inteligência Computacional - 2023/2024**

Otimização de hiper-parâmetros com Swarm  
intelligence

PROJETO FASE II

# Índice

Índice	1
1. Em que consiste a Computação Swarm?	2
2. Como funciona o algoritmo selecionado?	3
3. Aplicar e ilustrar o algoritmo para otimização	5
Resultados gerais	6
Melhores Resultados do PSO	6
Sensibilidade dos Parâmetros GSA	6
PSO: Número de Partículas	6
4. Otimização de hiperparâmetros	7
Análise das otimizações	9
5. Conclusão e Discussão de resultados	10

# 1. Em que consiste a Computação Swarm?

A Computação Swarm, ou computação de enxame, é um paradigma inspirado no comportamento coletivo de organismos sociais na natureza, como enxames de pássaros, colônias de formigas e cardumes de peixes, tendo como objetivo replicar a inteligência coletiva emergente desses sistemas para resolver problemas complexos.

Este paradigma utiliza múltiplos agentes autônomos que interagem entre si e com o ambiente seguindo regras simples. Estes agentes, também podem ser referidos como partículas, e colaboram de forma descentralizada para atingir objetivos globais.

No contexto de treino de uma rede neuronal, a computação Swarm pode ser aplicada para otimizar hiperparâmetros, como taxa de aprendizagem, número de neurônios, número de camadas, etc, onde cada uma destas configurações é representada por um agente no exame e a qualidade é avaliada com base no desempenho da rede neuronal treinada.

Esta abordagem de otimização permite explorar um grande espaço de hiperparâmetros de maneira eficiente, pois através da interação e troca de informações entre agentes pode se descobrir rapidamente regiões promissoras do espaço de pesquisa, adaptando-se a mudanças nas características do problema.

Resumindo, no treino de redes neurais, esta metodologia oferece uma maneira eficiente de descobrir configurações de hiperparâmetros que resultam em modelos mais robustos e eficazes, resolvendo problemas complexos onde acabamos por tirar partido da colaboração descentralizada e auto-organização que a computação Swarm oferece.

## 2. Como funciona o algoritmo selecionado?

O algoritmo selecionado foi o Gravitational Search Algorithm, este algoritmo é um algoritmo de otimização baseado na interação gravitacional entre massas em um sistema, onde foi inspirado nas leis da gravidade e no comportamento dos corpos celestes no espaço, tendo como objetivo principal encontrar a configuração otimizada de parâmetros para uma dada função de custo.

Inicialmente, cada agente passa por uma avaliação, onde é atribuída uma medida de aptidão com base em uma função objetivo. Soluções mais adequadas recebem massas maiores, enquanto soluções menos promissoras têm massas menores. A interação entre agentes é modelada usando a lei da gravidade. Agentes mais massivos exercem forças mais fortes, enquanto aqueles mais distantes experienciam forças mais fracas. Essa abordagem simula o movimento dos agentes no espaço de pesquisa, direcionando soluções mais promissoras para determinadas regiões. Ao longo de iterações, o algoritmo avalia, atribui massa, calcula forças e atualiza as posições dos corpos/agentes. Este processo permite que o enxame evolua, convergindo gradualmente para regiões do espaço onde soluções mais otimizadas podem ser encontradas.

O GSA é flexível e pode ser aplicado em diversas áreas. Em um contexto específico, como a otimização de hiperparâmetros para Redes Neurais Convolutivas (CNN), o GSA é utilizado para encontrar a configuração mais eficaz para a rede, maximizando seu desempenho em uma tarefa específica.

A otimização por Algoritmo de Enxame de Gravidade (GSA) e Otimização por Enxame de Partículas (PSO) são duas abordagens notáveis em problemas de otimização, cada uma trazendo suas próprias vantagens e desvantagens.

Sendo que o SGA é inspirado na interação gravitacional entre massas celestes, destaca-se por sua capacidade de explorar eficientemente o espaço de pesquisa, onde a sua força reside na forte ênfase na exploração, tornando-o particularmente útil em problemas complexos com espaços de busca extensos. No entanto, sua sensibilidade aos parâmetros pode ser uma desvantagem, requerendo uma cuidadosa calibração para obter o desempenho ideal. Por outro lado, o PSO baseia-se no comportamento coletivo de partículas em busca de soluções ótimas, onde a sua flexibilidade o torna adaptável a uma variedade de problemas. A facilidade de implementação e a capacidade de lidar com convergência prematura são vantagens distintas do PSO, no entanto, como desvantagem, também exige uma escolha criteriosa de parâmetros, embora seja geralmente menos sensível que o GSA.

Ambos os algoritmos têm aplicação em diversas áreas. O GSA destaca-se em problemas que exigem uma exploração profunda, como otimização de estruturas, enquanto o PSO é frequentemente aplicado em problemas complexos, como otimização de redes neurais e otimização combinatória.

### 3. Aplicar e ilustrar o algoritmo para otimização

Nesta etapa, realizei um desenvolvimento para avaliar o desempenho e a sensibilidade dos parâmetros do GSA (Gravitational Search Algorithm) e do PSO (Particle Swarm Optimization), apresentando e comparando os resultados. Para isso, foi utilizada uma função 'benchmark' baseada na função Ackley para dimensões 2 e 3.

As funções de Ackley são uma classe de funções de benchmark amplamente utilizadas para avaliar o desempenho de algoritmos de otimização. Elas foram propostas pelo matemático David Ackley e são conhecidas por apresentar características desafiadoras, como múltiplos mínimos locais e ampla região de pesquisa.

Dado isto, o objetivo nesta parte foi comparar os dois algoritmos e entender qual deles é mais eficaz na resolução de problemas de otimização, neste caso específico, na minimização da função de Ackley em diferentes configurações. O "melhor custo" neste caso é o valor mínimo da função objetivo que o algoritmo consegue encontrar.

```
class AckleyBenchmark:
    @staticmethod
    def ackley_2d(x):
        return -20 * np.exp(-0.2 * np.sqrt(0.5 * (x[0]**2 + x[1]**2))) - np.exp(0.5 * (np.cos(2 * np.pi * x[0]) + np.cos(2 * np.pi * x[1]))) + np.exp(1) + 20

    @staticmethod
    def ackley_3d(x):
        if len(x) == 2:
            return AckleyBenchmark.ackley_2d(x)
        return -20 * np.exp(-0.2 * np.sqrt(1/3 * (x[0]**2 + x[1]**2 + x[2]**2))) - np.exp(1/3 * (np.cos(2 * np.pi * x[0]) + np.cos(2 * np.pi * x[1]) + np.cos(2 * np.pi * x[2]))) + np.exp(1) + 20

def run_gsa_benchmark(dimensions, max_iter, num_agents, benchmark_function):
    gsa_obj = gsa(n=num_agents, function=benchmark_function, lb=-5, ub=5, dimension=dimensions, iteration=max_iter, G0=5)
    best_cost = gsa_obj.get_gbest()
    best_position = gsa_obj.get_agents()
    return best_cost, best_position

def run_pso_benchmark(dimensions, max_iter, num_particles, benchmark_function):
    pso_obj = pso(n=num_particles, function=benchmark_function, lb=-5, ub=5, dimension=dimensions, iteration=max_iter)
    best_cost = pso_obj.get_gbest()
    best_position = pso_obj.get_agents()
    return best_cost, best_position

dimensions_list = [2, 3]
max_iter_list = [50, 100, 200, 300]
num_agents_particles_list = [10, 20, 30]
```

Esta classe AckleyBenchmark foi utilizada para definir as funções de 2 e 3 dimensões de Ackley e são funções comuns para avaliar a performance de algoritmos de otimização e que foram utilizadas para analisar a performance tanto do GSA como do PSO.

## Resultados gerais

- O GSA tende a alcançar melhores resultados em termos de melhor custo em comparação com o PSO na função de Ackley.
- Os valores obtidos pelo GSA foram consistentemente mais próximos de zero em diversas configurações.

## Melhores Resultados do PSO

- Em alguns casos, o PSO obteve resultados muito próximos de zero, indicando que, em situações específicas, o PSO também pode fornecer soluções de alta qualidade.

## Sensibilidade dos Parâmetros GSA

- Número de Agentes (G): Aumentar o número de agentes geralmente permitiu explorar mais efetivamente o espaço de pesquisa, resultando em melhores soluções.
- Número de Iterações: Aumentar o número de iterações permitiu que o algoritmo continuasse a busca, melhorando gradualmente os resultados.

## PSO: Número de Partículas

- Aumentar o número de partículas melhorou a capacidade de explorar o espaço de pesquisa, e obteve soluções mais precisas.
- Número de Iterações: Um aumento nas iterações permitiu que o PSO alcançasse melhores soluções, especialmente se a convergência for lenta.
- Dimensões: Aumentar as dimensões pode aumentar a complexidade do problema, tornando-o mais desafiador para o PSO.

## 4. Otimização de hiperparâmetros

Nesta fase concentrei-me na otimização dos hiperparâmetros, tendo também alterado para redes neurais convolucionais (CNNs), e utilizando para tal algoritmos Swarm, especificamente o Particle Swarm Optimization (PSO) e o Gravitational Search Algorithm (GSA).

Para isto escolhi dois hiperparâmetros críticos para a arquitetura da minha CNN: **o número de neurónios** na camada escondida e a **taxa de aprendizagem**, sendo estes parâmetros fundamentais porque influenciam diretamente a capacidade da rede em aprender padrões complexos e a sua velocidade de convergência durante o treino.

Para a otimização, recorri ao PSO e ao GSA, ambos implementados através da biblioteca SwarmPackagePy. Estes algoritmos realizam pesquisas no espaço de hiperparâmetros, ajustando-os de forma a maximizar a **área sob a curva ROC (AUC)** do modelo no conjunto de validação.

Dado isto, foi criada uma função de avaliação que será utilizada para determinar o desempenho do modelo criado para os determinados hiper-parâmetros, esta função chama-se **evaluate\_model\_GSA** e **evaluate\_model\_PSO**. Esta função inicia a sua execução extraindo os hiperparâmetros, o número de neurónios na camada escondida e a taxa de aprendizagem, após isto, com estes parâmetros, a função procede à criação e ao treino de uma rede neural convolucional (CNN), utilizando um conjunto fixo de épocas, mantendo o processo eficiente e livre de saídas verbosas desnecessárias.

Após o treino, a função avalia o desempenho do modelo utilizando o conjunto de validação, sendo que isto é feito através da comparação das previsões do modelo com os rótulos reais. A métrica escolhida para esta avaliação é a Área Sob a Curva Receiver Operating Characteristic (**AUC**), calculada através da função `roc_auc_score`. Esta métrica é particularmente útil para medir a capacidade do modelo de distinguir entre várias classes e a função retorna  $(1 - \text{auc\_score})$ , uma formulação que permite a maximização do AUC, visto que os algoritmos de otimização focam na minimização da função objetivo. Esta metodologia de avaliação foi a definida para o meu processo de otimização, facilitando a identificação da configuração ótima da rede CNN para a tarefa de classificação de imagens, e garantindo assim a escolha dos melhores hiperparâmetros para o modelo.



```

def create_cnn_model(num_neurons, learning_rate):
    input_shape = (64, 64, 3)
    num_classes = len(categories)

    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu',
input_shape=input_shape))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))

    model.add(tf.keras.layers.Conv2D(num_neurons, kernel_size=(3, 3),
activation='relu'))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))

    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(num_classes, activation='softmax'))

    optimizer = Adam(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

    return model

def evaluate_model(solution):
    num_neurons, learning_rate = solution
    model = create_cnn_model(num_neurons, learning_rate)
    model.fit(x_train, y_train, epochs=5, batch_size=32, verbose=0)

    y_pred = model.predict(x_validation)
    y_validation_categorical = to_categorical(y_validation,
num_classes=len(categories))
    auc_score = roc_auc_score(y_validation_categorical, y_pred, multi_class='ovr')
    return 1 - auc_score

num_agents = 2 # Número de agentes
max_iter = 3 # Número de iterações
dimension = 2 # Número de dimensões (neste caso, num_neurons e
learning_rate)
lb = [10, 0.0001] # Limites inferiores para num_neurons e learning_rate
ub = [100, 0.01] # Limites superiores para num_neurons e learning_rate

print("Iniciando a otimização com GSA...")
gsa_instance = gsa(n=num_agents, function=evaluate_model, lb=lb, ub=ub,
dimension=dimension, iteration=max_iter)

best_solution_gsa = gsa_instance.get_Gbest()

```

*(Exemplo usado para a avaliação e determinação dos melhores hiperparâmetros)*

	Tipo	Num_Agentes	Num_Iterações	Dimensão	Limite_Inferior	Limite_Superior	Num_Neurônios	Learning_Rate	Train_Accuracy	Validation_Accuracy	Test_Accuracy	Test_AUC
1	PSO	1	3	2	[10, 0.0001]	[100, 0.01]	47.87874676243591	0.0087733845747881	0.878333330154419	0.7909374833106995	0.8025000095367432	
2	PSO	2	3	2	[10, 0.0001]	[100, 0.01]	75.22757411441034	0.0054160404778483	0.9545633468437196	0.8712499737739563	0.893750011920929	
3	PSO	2	5	2	[10, 0.0001]	[100, 0.01]	52.0556527933356	0.0014489380436244	0.9762499928474426	0.9259374737739564	0.9375	
4	PSO	3	3	2	[10, 0.0001]	[100, 0.01]	81.0689346593836	0.0014025599233438	0.9768333435058594	0.9293749928474426	0.9412500262260436	
5	PSO	3	5	2	[10, 0.0001]	[100, 0.01]	17.307716508221	0.0059809363961371	0.9321666595947876	0.8500000238418579	0.8700000047683716	
6	PSO	3	10	2	[10, 0.0001]	[100, 0.01]	32.82104476338705	0.0018843320254153	0.9607499837675366	0.931249976158142	0.9387500286102296	
7	PSO	4	10	2	[10, 0.0001]	[100, 0.01]	47.96320103316841	0.0015117143188046	0.9670633345227904	0.9075000286102296	0.9162499904632568	
8	GSA	2	3	2	[10, 0.0001]	[100, 0.01]	47.27899332675588	0.0087431398830898	0.8589166402816772	0.7337499856948853	0.7350000143051147	0.9557892857142857
9	GSA	2	5	2	[10, 0.0001]	[100, 0.01]	77.86912753604364	0.007785604028644	0.8973333239555359	0.801562488079071	0.8087499737739563	0.9733446428571428
10	GSA	3	3	2	[10, 0.0001]	[100, 0.01]	21.971680107185872	0.0023550929124363	0.9631666541099548	0.8315625190734863	0.8212500214576721	0.9871321428571428
11	GSA	3	5	2	[10, 0.0001]	[100, 0.01]	28.13555613472179	0.0023181631727126	0.969666651725769	0.8987500071525574	0.9024999737739564	0.9948517857142856
12	GSA	3	10	2	[10, 0.0001]	[100, 0.01]	70.78772267244659	0.0011889281157621	0.972000002861023	0.9334375262260436	0.9325000047683716	0.9974214285714286

## Análise das otimizações

A variação nas métricas de acerto e AUC entre as diferentes execuções sugere que pode haver uma sensibilidade à inicialização e à escolha dos hiperparâmetros.

Ao analisar os resultados do uso do Gravitational Search Algorithm (GSA) em comparação com o Particle Swarm Optimization (PSO) para a otimização de hiperparâmetros em uma CNN, notamos algumas tendências e diferenças importantes:

- O GSA tende a ser mais demorado em comparação com o PSO e deve-se à natureza do algoritmo, que simula um sistema onde cada partícula (agente) é atraída por todas as outras com uma força que depende da massa e da distância entre elas e este cálculo pode ser computacionalmente intensivo, especialmente à medida que o número de agentes aumenta.
- Observei que o GSA mostra melhorias consistentes à medida que aumentamos o número de agentes e iterações. Isto sugere que o algoritmo é capaz de explorar mais eficientemente o espaço de soluções com uma pesquisa mais extensa comparativamente ao PSO.
- Apesar de mais rápido, o PSO pode não explorar o espaço de soluções tão extensivamente quanto o GSA, especialmente com um número menor de iterações ou agentes.
- O PSO mostrou-se útil em uma variedade de configurações, o que sugere que é um algoritmo versátil e pode ser aplicado em diferentes contextos e conjuntos de dados.
- A performance do modelo (em termos de acerto de treino, validação e teste) varia com diferentes configurações de número de agentes e iterações, sugerindo que o PSO é sensível à escolha destes parâmetros.
- Como o PSO é baseado no conceito de equilibrar a exploração de novas áreas do espaço de soluções e a exploração de áreas conhecidas, em alguns casos, um maior número de agentes e iterações pode não levar a uma melhoria significativa se o enxame já convergiu para uma solução ótima ou se houver risco de sobreajuste.

## 5. Conclusão e Discussão de resultados

A análise das otimizações realizadas com o Gravitational Search Algorithm (GSA) e o Particle Swarm Optimization (PSO) apresentou amostras valiosas sobre a otimização de hiperparâmetros em redes neurais convolucionais (CNNs).

A variação nas métricas de acerto e AUC entre diferentes execuções destacou a sensibilidade do modelo à inicialização e à escolha dos hiperparâmetros.

Observei que o GSA, embora mais demorado devido à sua natureza computacionalmente intensiva, mostrou melhorias consistentes com o aumento do número de agentes e iterações, sugerindo uma capacidade superior de explorar o espaço de soluções, em que em contraste, o PSO, sendo mais rápido, apresentou uma performance variável, indicando que pode não explorar tão profundamente o espaço de soluções, especialmente com configurações de menos agentes e iterações.

A transição de redes MLP para redes convolucionais resultou em um aumento significativo na taxa de acerto final do melhor modelo, alcançando facilmente os 94% sobre as imagens de teste, e isto ressalta a eficácia das CNNs em tarefas de classificação de imagens, onde são capazes de capturar padrões mais complexos.

O uso de algoritmos de otimização baseados em Swarm, como o GSA e o PSO, demonstrou as vantagens e as possibilidades existentes na definição automática e na descoberta dos melhores hiperparâmetros para treinar os modelos, o que não só economiza tempo e esforço significativos que seriam gastos em ajustes manuais, mas também revela configurações de hiperparâmetros que podem não ser imediatamente óbvias.