isec
Politécnico de Coimbra

Inteligência Computacional

2024/2025

Projeto – Meta 2

João Pedro Silveira da Costa, N.º 2022143368

Juliana Silva Teixeira, N.º 2022143379

Índice

1. Em que Consiste a Computação Swarm

A computação swarm ou de enxame é um paradigma inspirado na inteligência de enxame, um conceito derivado de sistemas naturais como colónias de formigas, bandos de aves ou cardumes de peixes. Nestes grupos, indivíduos simples agem de forma independente, mas as suas interações coletivas resultam em comportamentos complexos e em capacidades de resolução de problemas. Aplicado à computação, o paradigma swarm envolve um conjunto estruturado de agentes (entidades de software, como modelos de linguagem ou outras inteligências artificiais) que colaboram para alcançar um objetivo comum. A grande vantagem deste sistema reside na sinergia entre os agentes, ao interagirem entre si, conseguindo assim resolver tarefas de forma mais eficiente e robusta do que se trabalhassem isoladamente.

Na computação swarm, cada agente é relativamente simples e autónomo, tomando decisões locais com base nas informações do ambiente imediato ou nas comunicações com agentes próximos. Esta abordagem resulta em descentralização, escalabilidade e robustez, fazendo assim com que o sistema não dependa de uma unidade central e não seja tão vulnerável a falhas isoladas, uma vez que os agentes podem adaptar-se e reorganizar-se conforme os diferentes ambientes. Este tipo de organização é especialmente útil em situações que exigem grande adaptabilidade e resiliência, como o treino de redes neuronais distribuídas e a otimização de híper-parâmetros.

No contexto do treino de redes neuronais, a computação swarm pode ser aplicada de diversas formas:

- Treino Distribuído: Múltiplos agentes treinam partes de uma rede neuronal de forma independente, para depois combinarem os resultados, conseguindo assim melhorar a eficiência e reduzir o tempo necessário para o treino.
- Otimização de Híper-parâmetros: Cada agente testa diferentes configurações de híper-parâmetros, como a taxa de aprendizagem e a arquitetura da rede, e partilha os melhores resultados com os demais, acelerando a procura por configurações ótimas.

Assim, a computação swarm explora a inteligência coletiva e a auto-organização, aproveitando a força das interações entre agentes individuais para resolver problemas complexos de forma eficiente e escalável, revelando-se uma abordagem poderosa e inspirada na natureza para sistemas de IA distribuídos.

2. Análise e Comparação entre o Algoritmo GWO e PSO

O algoritmo Gray Wolf Optimization inspira-se na estrutura hierárquica e na estratégia de caça dos lobos cinzentos na natureza. Os lobos seguem uma organização social rígida, dividindo a alcateia em quatro níveis: alfa, beta, delta e ómega.

Os lobos alfa são os líderes da sua alcateia. São responsáveis pela tomada de decisões, mas por vezes os alfas podem obedecer a outros lobos da alcateia.

Os lobos beta ajudam os alfas a tomar decisões e todos os beta são candidatos a alfa se um alfa tiver morrido ou envelhecido. Um lobo beta respeita um alfa e transfere comandos para a alcateia, assegura a disciplina entre os lobos inferiores e fornece um feedback da alcateia a um alfa.

Os lobos delta têm de se submeter aos alfas e aos betas, mas dominam os ómega.

Finalmente, os lobos ómega têm de obedecer a todos os outros lobos. Por vezes, desempenham um papel de protetores.

No modelo matemático, a hierarquia social dos lobos é mapeada para a solução adequada. A solução mais apta é considerada a alfa. Beta e delta são a segunda e terceira melhores soluções, respetivamente. As restantes soluções candidatas são consideradas ómega.

O Gray Wolf Optimization (GWO) e o Particle Swarm Optimization (PSO) são algoritmos baseados em comportamentos sociais. O PSO simula o movimento de pássaros ou peixes, ajustando as soluções com base na melhor experiência individual e global. Já o GWO, inspirado na hierarquia dos lobos, utiliza uma estrutura social mais complexa (alfa, beta, delta, ómega) para guiar a procura por soluções. Enquanto o PSO prioriza interações simples entre partículas, o GWO reflete relações hierárquicas e estratégias cooperativas para adaptação das soluções.

Vantagens do GWO em relação ao PSO incluem:

- Equilíbrio entre busca ampla e refinamento: O GWO adapta-se melhor entre explorar novas soluções e melhorar as existentes.
- Menor risco de ficar preso em más soluções: A influência de várias "lideranças" (alfa, beta, delta) ajuda a evitar armadilhas de ótimos locais.
- Menor número de parâmetros a ajustar: No PSO, deve-se definir a inércia, o fator cognitivo e o fator social.

Desvantagens

- Menos flexível nos ajustes: O PSO permite mais personalizações para se adaptar a diferentes problemas.
- Menor eficiência em problemas grandes: O GWO pode ser menos eficaz em problemas com muitas variáveis.
- Computacionalmente intensivo: Devido à complexidade das interações sociais, o GWO torna-se mais exigente em termos de processamento.

3. Análise da Influência do Algoritmo PSO na função de Ackley

```python
import pyswarms as ps
import numpy as np


def ackley_function(x):
    a = 20
    b = 0.2
    c = 2 * np.pi
    d = x.shape[1]  # Número de dimensões (colunas da matriz x)

    sum_sq_term = -a * np.exp(-b * np.sqrt(np.sum(x**2, axis=1) / d))
    cos_term = -np.exp(np.sum(np.cos(c * x), axis=1) / d)
    result = sum_sq_term + cos_term + a + np.exp(1)
    return result


# Definindo os limites do espaço de busca para a função de Ackley
bounds = (np.array([-32.768] * 3), np.array([32.768] * 3))

# Configurações de PSO
options = {'c1': 0.5, 'c2': 0.3, 'w': 0.4}

# Inicialização do PSO
optimizer = ps.single.GlobalBestPSO(n_particles=30,
                                    dimensions=3,
                                    options=options,
                                    bounds=bounds)

# Execução do PSO
cost, pos = optimizer.optimize(ackley_function, iters=100)

# Resultado
print(f"Melhor custo encontrado: {cost}")
print(f"Melhor posição encontrada: {pos}")
```

Este código implementa o algoritmo de Otimização por Enxame de Partículas (PSO) para encontrar o mínimo global da função de Ackley.

O objetivo é minimizar o custo calculado pela função, com base na posição das partículas dentro do espaço de busca, que pode ter duas ou três dimensões. O PSO utiliza parâmetros como componente cognitiva (c1), componente social (c2) e peso da inércia (w) para equilibrar a exploração e a exploração do espaço de busca. O algoritmo começa com 30 partículas ao longo de 100 iterações.

O resultado demonstra a eficácia do PSO em alcançar o mínimo global da função, que é 0 nas coordenadas (0,0) para o espaço bidimensional ou (0,0,0) para o espaço tridimensional.

Análise dos diferentes parâmetros

Parâmetros	Melhor Custo (2 Dimensões)	Melhor Posição (2 Dimensões)	Melhor Custo (3 Dimensões)	Melhor Posição (3 Dimensões)
c1 = 0.5 c2 = 0.3 w = 0.9 n_particles = 30 iters = 100	0.0015	(0.0005,0.0001)	0.0150	(0.0042,0.0012,0.0044)
c1 = 1.0 c2 = 0.3 w = 0.4 n_particles = 30 iters = 100	0.0107	(0.0033, -0.0015)	0.7991	(-0.1404,0.0138,-0.0849)
c1 = 0.5 c2 = 0.3 w = 0.4 n_particles = 30 iters = 100	3.9968	(-7.4600e-16,1.0799e-15)	5.9519	(1.7393e-06,-1.8000e-06,-6.1390e-07)
c1 = 0.5 c2 = 0.3 w = 0.9 n_particles = 100 iters = 100	0.0064	(0.0012,0.0019)	0.0052	(0.0014,0.0004,-0.0017)
c1 = 0.5 c2 = 0.3 w = 0.9	0.0303	(-0.0069,-0.0070)	0.5451	(-0.0251,0.0629,-0.1051)

n_particles = 30 iters = 50				
c1 = 1.0 c2 = 1.0 w = 0.9 n_particles = 100 iters = 100	0.0091	(-1.0565e-05,3.1127e-03)	0.0348	(0.0051,-0.0125,-0.0022)

Para iniciar a análise, foram escolhidos valores iniciais aleatórios apenas para servir de ponto de partida. A análise incluiu também uma avaliação do impacto de um equilíbrio maior entre a componente cognitiva e a componente social no desempenho do algoritmo. Além disso, testou-se como uma inércia menor, que promove uma exploração mais ampla do espaço de busca, afeta a capacidade do algoritmo de encontrar o mínimo global.

Outro aspeto observado foi o impacto de um enxame maior na convergência para o mínimo global, considerando que uma maior diversidade de soluções pode favorecer a exploração de diferentes áreas do espaço de busca. Também foi verificado se o algoritmo consegue encontrar uma solução satisfatória com um menor número de iterações, analisando, assim, a eficiência em termos de velocidade de convergência.

Com base nos resultados anteriores, a análise final foi aumentar os valores de c1, c2 e n_particles, mantendo constantes os valores de w e iters, para avaliar o impacto dessas configurações na capacidade do PSO de explorar e convergir de forma eficaz para o mínimo global.

Através da realização de vários testes com o algoritmo PSO aplicado à função de Ackley, foi possível observar que ao variar os diferentes parâmetros influenciam a eficácia do algoritmo na busca pelo mínimo global. Observou-se que aumentar o número de partículas e os valores de c1 e c2, que intensificam as influências das componentes cognitiva e social, respetivamente, tende a melhorar a precisão da busca tanto em duas dimensões como em três.

Análise dos resultados do GWO

```python
from SwarmPackagePy import gwo
import numpy as np


# Definir a função objetivo, por exemplo, a função Ackley
def ackley_function(x):
    a = 20
    b = 0.2
    c = 2 * np.pi
    d = len(x)  # Dimensão do vetor de entrada
    sum_sq_term = -a * np.exp(-b * np.sqrt(sum(x**2) / d))
    cos_term = -np.exp(sum(np.cos(c * x) / d))
    result = a + np.exp(1) + sum_sq_term + cos_term
    return result


# Parâmetros do GWO
n_agents = 100
n_iterations = 100
dimension = 3  # Duas dimensões
lower_bound = -32.768
upper_bound = 32.768


# Executar o algoritmo GWO
optimizer = gwo(n_agents, ackley_function, lower_bound, upper_bound, dimension, n_iterations)
best_position = optimizer.get_Gbest()


# Exibir os resultados
print(f"Melhor posição: {best_position}")
```

Parâmetros	**Melhor Posição (2 Dimensões)**	**Melhor Posição (3 Dimensões)**
n_agents = 100 n_iterations = 100	(-1.9060e-16,-2.9420e-16)	(1.5602e-16,-1.0096e-16,-2.9401e-17)

Comparando os parâmetros do algoritmo PSO no algoritmo GWO podemos observar que o GWO se aproxima mais do resultado pretendido, ainda que possua menos parâmetros.

Com esta comparação, podemos concluir que o GWO tem um melhor desempenho relativamente ao PSO.

4. Comparação do Algoritmo GWO e PSO na Otimização da Arquitetura

Código PSO

O código apresentado descreve o algoritmo de Otimização por Enxame de Partículas para treinar uma rede neuronal convolucional (CNN) e otimizar os seus parâmetros, que neste caso é a taxa de desativação e a taxa de aprendizagem.

A função train_network calcula a perda de validação para cada conjunto de híper-parâmetros fornecido pelo PSO, ajudando-o a encontrar os melhores valores.

De seguida, é configurado o PSO, definindo os limites dos híper-parâmetros e executa-o para otimizar a taxa de aprendizagem e a taxa de desativação.

Por último, usa os melhores híper-parâmetros encontrados para treinar o modelo final, avalia o modelo nos dados de teste, calcula métricas como accuracy, recall, precision e F1 score, e gera uma matriz de confusão para análise dos resultados.

```python
# Função para criar o modelo CNN
def create_cnn_model(input_shape, num_classes, learning_rate, dropout_rate):
    model = Sequential()

    model.add(Conv2D(32, (3, 3), activation='relu', input_shape=input_shape))
    model.add(MaxPooling2D((2, 2)))

    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))

    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))

    model.add(Flatten())
    model.add(Dropout(dropout_rate))
    model.add(Dense(num_classes, activation='softmax'))

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

```python
# Função de treinamento da rede neural
def train_network(hyperparameters, train_x, train_y, val_x, val_y):
    learning_rate, dropout_rate = hyperparameters
    model = create_cnn_model((32, 32, 3), len(classes), learning_rate, dropout_rate)

    model.fit(train_x, train_y, epochs=5, validation_data=(val_x, val_y), verbose=0)
    val_loss, val_accuracy = model.evaluate(val_x, val_y, verbose=0)
    return val_loss

# Função de fitness para o PSO
def fitness_function(x, train_x, train_y, val_x, val_y):
    n_particles = x.shape[0]
    losses = []

    for i in range(n_particles):
        hyperparameters = x[i]
        loss = train_network(hyperparameters, train_x, train_y, val_x, val_y)
        losses.append(loss)

    return np.array(losses)
```

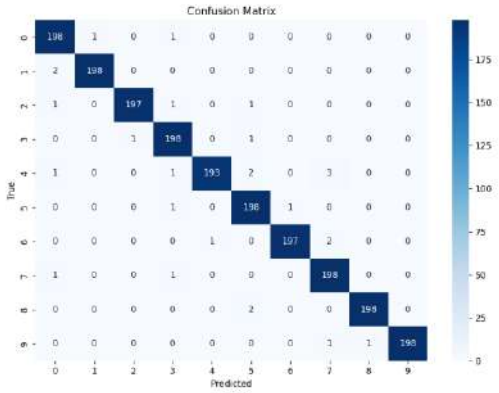
```python
# Definir limites e opções do PSO
bounds = [(0.0001, 0.01), (0.0, 0.5)]  # (min_learning_rate, max_learning_rate), (min_dropout_rate, max_dropout_rate)
options = {'c1': 0.5, 'c2': 0.3, 'w': 0.9}

# Inicializar o otimizador PSO
optimizer = ps.single.GlobalBestPSO(n_particles=20, dimensions=2, options=options, bounds=bounds)

# Executar a otimização
cost, best_pos = optimizer.optimize(fitness_function, iters=20, train_x=X_train, train_y=y_train, val_x=X_test, val_y=y_test)

best_learning_rate, best_dropout_rate = best_pos
print(f"Melhor learning rate: {best_learning_rate}")
print(f"Melhor dropout rate: {best_dropout_rate}")

# Treinar o modelo final com os melhores hiperparâmetros
final_model = create_cnn_model((32, 32, 3), len(classes), best_learning_rate, best_dropout_rate)
history = final_model.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test))
```

```python
# Avaliação e Métricas
test_loss, test_accuracy = final_model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_accuracy:.4f}')

y_probs = final_model.predict(X_test)
y_pred = np.argmax(y_probs, axis=1)

# Métricas de desempenho
accuracy = accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred, average='weighted')
precision = precision_score(y_test, y_pred, average='weighted')
fmeasure = f1_score(y_test, y_pred, average='weighted')

print(f"Accuracy: {accuracy}")
print(f"Recall: {recall}")
print(f"Precision: {precision}")
print(f"F-measure: {fmeasure}")

# Matriz de Confusão
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

Análise de resultados

```
Melhor learning rate: 0.00019550948941979073
Melhor dropout rate: 0.11431650642068683
```

```
Accuracy: 0.9865
Recall: 0.9865
Precision: 0.986660948631
F-measure: 0.986518879485

AUC
AUC for class 2: 0.9999
AUC for class 8: 0.9999
AUC for class 9: 0.9999
AUC for class 10: 0.9996
AUC for class 11: 0.9991
AUC for class 13: 0.9998
AUC for class 18: 1.0000
AUC for class 25: 0.9997
AUC for class 35: 0.9999
AUC for class 38: 0.9999
```



Os resultados do algoritmo PSO indicam que foi atingida uma taxa de aprendizagem de aproximadamente de 0.0002 e uma taxa de desativação cerca de 0.1143. Alcançando uma accuracy, uma recall, um f-measure e uma precision de 98.7%.

Código GWO

```python
# Função para criar o modelo CNN
def create_cnn_model(input_shape, num_classes, learning_rate, dropout_rate):
    model = Sequential()

    model.add(Conv2D(32, (3, 3), activation='relu', input_shape=input_shape))
    model.add(MaxPooling2D((2, 2)))

    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))

    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))

    model.add(Flatten())
    model.add(Dropout(dropout_rate))
    model.add(Dense(num_classes, activation='softmax'))

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

    return model
```

```python
# Função de treinamento para o GWO
def fitness_function(hyperparameters):
    learning_rate, dropout_rate = hyperparameters
    input_shape = (32, 32, 3)
    num_classes = 10

    model = create_cnn_model(input_shape, num_classes, learning_rate, dropout_rate)

    history = model.fit(X_train, y_train, epochs=5, validation_data=(X_test, y_test), verbose=0)
    val_loss = model.evaluate(X_test, y_test, verbose=0)[0]

    print(f"Hyperparameters: {hyperparameters}, Validation Loss: {val_loss}")
    return val_loss

# Parâmetros do GWO
n_agents =10
n_iterations = 20
dimensions = 2
lower_bound = [0.0001, 0.0]  # Limites inferiores: learning rate e dropout
upper_bound = [0.2, 0.7]     # Limites superiores: learning rate e dropout
```

```python
# Inicializar e executar o GWO
optimizer = gwo(n_agents, fitness_function, lower_bound, upper_bound, dimensions, n_iterations)
best_learning_rate, best_dropout_rate = optimizer.get_Gbest()

print(f"\nMelhores hiperparâmetros encontrados:")
print(f"Learning Rate: {best_learning_rate}")
print(f"Dropout Rate: {best_dropout_rate}\n")

# Treinar o modelo final com os melhores hiperparâmetros
final_model = create_cnn_model((32, 32, 3), 10, best_learning_rate, best_dropout_rate)
history = final_model.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test))

# Avaliar o modelo final
test_loss, test_acc = final_model.evaluate(X_test, y_test)
print(f"\nTest Loss: {test_loss:.4f}, Test Accuracy: {test_acc:.4f}")
```

```python
# Previsões e métricas
y_probs = final_model.predict(X_test)
y_pred = np.argmax(y_probs, axis=1)

accuracy = accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred, average='weighted')
precision = precision_score(y_test, y_pred, average='weighted')
fmeasure = f1_score(y_test, y_pred, average='weighted')

print(f"\nAccuracy: {accuracy:.4f}")
print(f"Recall: {recall:.4f}")
print(f"Precision: {precision:.4f}")
print(f"F-measure: {fmeasure:.4f}")

# Matriz de Confusão
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

O código apresentado descreve o algoritmo de Gray Wolf Optimization para treinar uma rede neuronal convolucional (CNN) e otimizar os seus parâmetros, que neste caso é a taxa de desativação e a taxa de aprendizagem.

A função train_function avalia o desempenho de cada conjunto de híper-parâmetros no conjunto de validação, calculando a perda de validação.

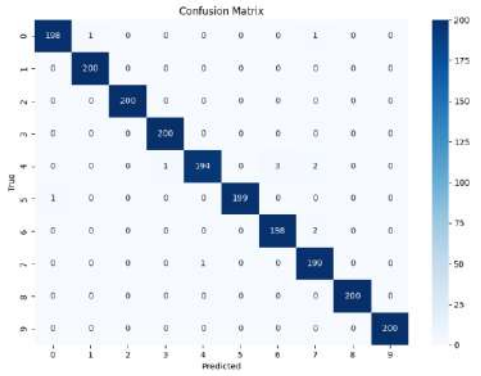
De seguida, é configurado o GWO, com 10 agentes e 20 iterações para encontrar os melhores valores para a taxa de aprendizagem e a taxa de desativação.

Por último, treina o modelo final com os melhores híper-parâmetros encontrados pelo GWO e avalia o modelo no conjunto de teste, calcula métricas de desempenho (accuracy, recall, precision, F-measure) e gera uma matriz de confusão para visualização dos resultados.

Análise de resultados

```
Melhores hiperparâmetros encontrados:
Learning Rate: 0.003714367213570218
Dropout Rate: 0.27767481630082097
```

```
Accuracy: 0.9940
Recall: 0.9940
Precision: 0.9940
F-measure: 0.9940

AUC
AUC for class 2: 1.0000
AUC for class 8: 1.0000
AUC for class 9: 1.0000
AUC for class 10: 1.0000
AUC for class 11: 0.9994
AUC for class 13: 1.0000
AUC for class 18: 1.0000
AUC for class 25: 0.9999
AUC for class 35: 1.0000
AUC for class 38: 1.0000
```



Os resultados do algoritmo GWO indicam que foi atingida uma taxa de aprendizagem de aproximadamente de 0.0037 e uma taxa de desativação cerca de 0.2777. Alcançando uma accuracy, um recall , um f-measure e uma precision de 99.4%.

As pontuações AUC para cada classe são de aproximadamente 1, o que mostra que o modelo tem uma excelente capacidade de distinguir entre as classes positivas e negativas em todas as categorias.

Em suma, o uso do algoritmo GWO levou ao desenvolvimento de um modelo altamente eficiente, com híper-parâmetros otimizados que garantiram um desempenho sólido e consistente na tarefa de classificação.

5. Conclusão e Discussão de Resultados

A análise dos resultados das otimizações realizadas com os algoritmos Gray Wolf Optimization (GWO) e Particle Swarm Optimization (PSO) demonstra a eficácia de ambos na afinação de híper-parâmetros para redes neuronais, resultando em modelos com elevada precisão nos testes. O PSO alcançou uma accuracy de 98.65%, enquanto o GWO obteve uma ligeiramente superior, de 99.4 %. Esta diferença pode ser explicada pelas particularidades dos métodos de busca dos algoritmos e pelas características específicas do conjunto de dados.

As pontuações AUC foram extremamente elevadas em ambos os casos, evidenciando a capacidade consistente dos modelos em distinguir entre classes positivas e negativas.

Em conclusão, os resultados indicam que tanto o GWO como o PSO são métodos eficazes para otimizar híper-parâmetros em redes neuronais aplicadas a problemas de classificação. Embora o PSO tenha alcançado uma ligeira vantagem em termos de accuracy, a diferença é mínima, tornando a escolha entre os dois mais dependente das características específicas do problema ou preferências do utilizador. Ambos os algoritmos mostraram desempenho consistente, destacando a eficácia das técnicas de otimização inspiradas em enxames para resolver problemas complexos.

6. Referências

- https://github.com/SISDevelop/SwarmPackagePy?tab=readme-ov-file#particle-swarm-optimization
- https://pt.wikipedia.org/wiki/Intelig%C3%AAncia_de_enxame