isec
Politécnico de Coimbra

TRAFFIC SIGNALS CLASSIFICATION

INTELIGÊNCIA COMPUTACIONAL

2024/2025
João Pedro Silveira da Costa, N.º 2022143368
Juliana Silva Teixeira, N.º 2022143370

ÍNDICE

1. EM QUE CONSISTE A COMPUTAÇÃO SWARM

Inspirada na inteligência coletiva de sistemas naturais (formigas, aves, peixes), envolve agentes simples e autónomos que interagem para resolver problemas complexos.

Aplicações em Redes Neuronais:

- Treino Distribuído: Agentes treinam partes da rede de forma independente e combinam os resultados, melhorando eficiência e reduzindo o tempo.
- Otimização de Híper-parâmetros: Testam diferentes configurações e partilham os melhores resultados, acelerando a procura por soluções ótimas.

Aproveita a interação coletiva para criar sistemas de IA distribuídos, eficientes e adaptáveis.

2. ANÁLISE E COMPARAÇÃO ENTRE O ALGORITMO GWO E PSO

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
- Menor risco de ficar preso em más soluções
- Menor número de parâmetros a ajustar

Desvantagens

- Menos flexível nos ajustes
- Menor eficiência em problemas grandes
- Computacionalmente intensivo

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}")
```

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 n_particles = 30 iters = 50	0.0303	(-0.0069,-0.0070)	0.5451	(-0.0251,0.0629,-0.1051)
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)

3. ANÁLISE DA INFLUÊNCIA DO ALGORITMO PSO NA FUNÇÃO DE ACKLEY

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)

4. COMPARAÇÃO DO ALGORITMO GWO E PSO NA OTIMIZAÇÃO DA ARQUITETURA

CÓDIGO PSO

```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)
```

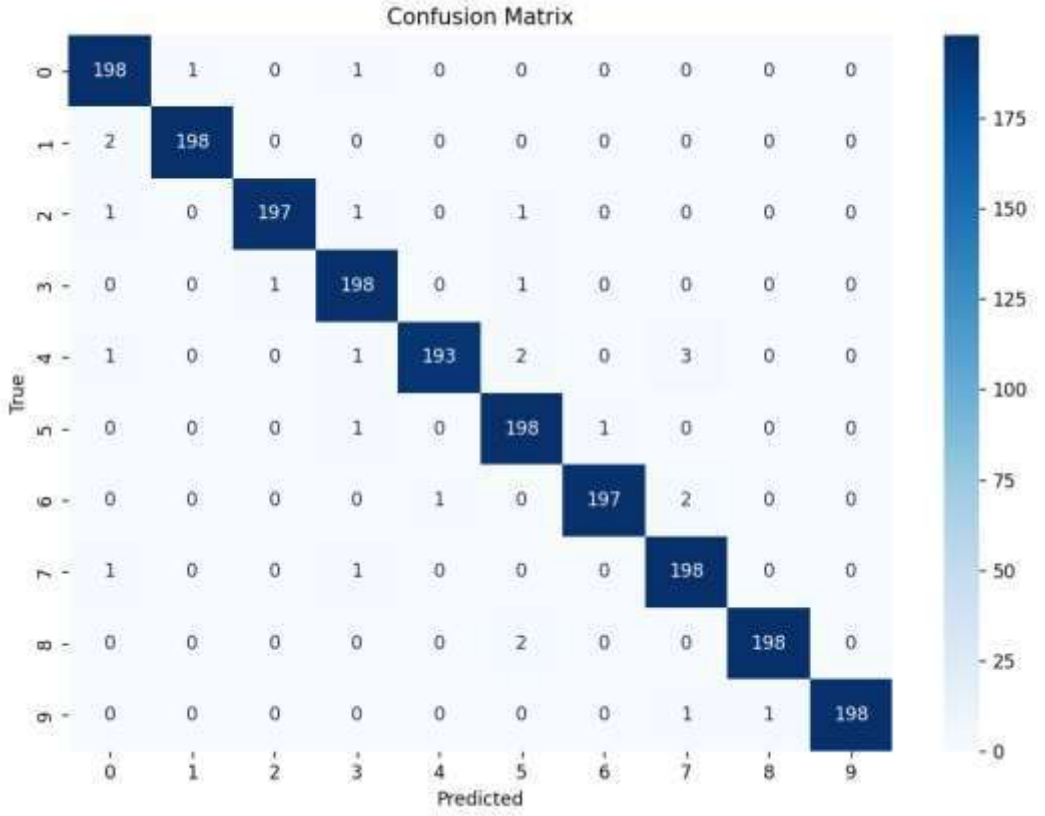
```python
# Definir limites e opções do PSO
bounds = [(0.0001, 0.01), (0.0, 0.5)]  # (min_learning_rate, max_learning_rate), (min_dropout_rate, max_dropout_rate)
options = {'c1': 0.5, 'c2': 0.3, 'w': 0.9}

# 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 DOS RESULTADOS

```
Melhor learning rate: 0.0001955094894197907 3
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
```



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
```

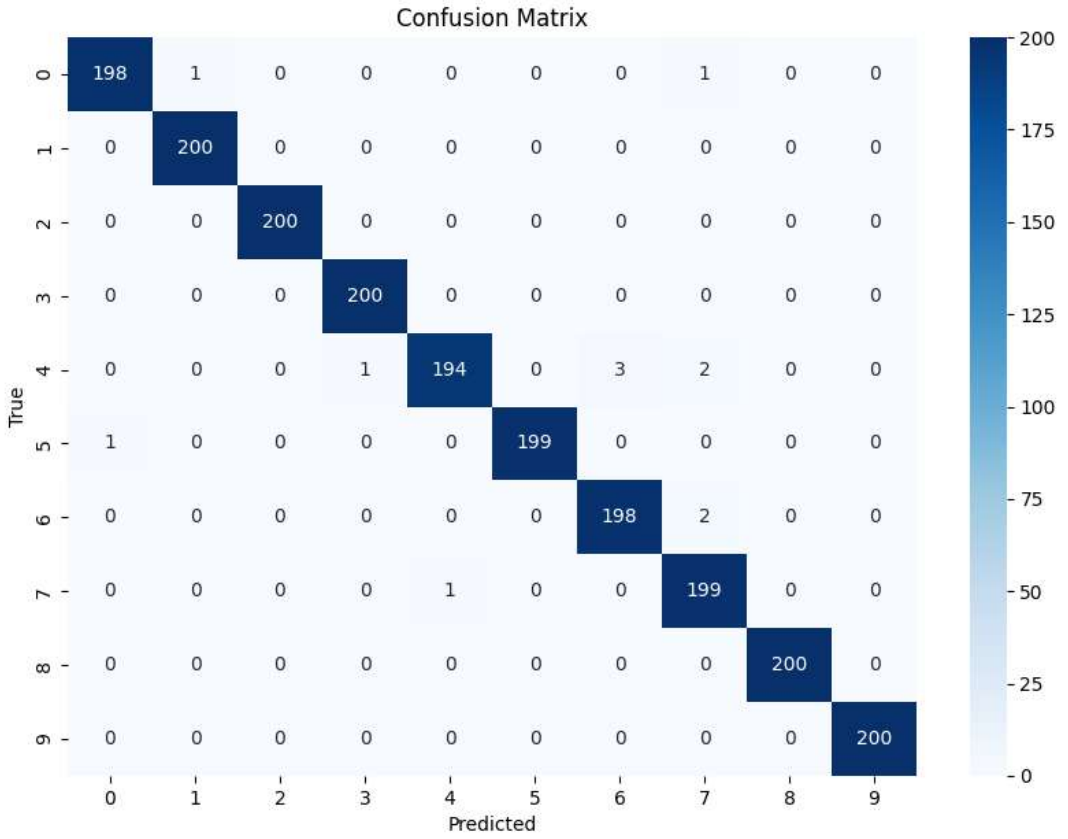
```python
# Inicializar e executar o GWO
optimizer = gwo(n_agents, fitness_function, lower_bound, upper_bound, dimensions, n_iterations)
best_learning_rate, best_dropout_rate = optimizer.get_Gbest()

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()
```

ANÁLISE DOS RESULTADOS

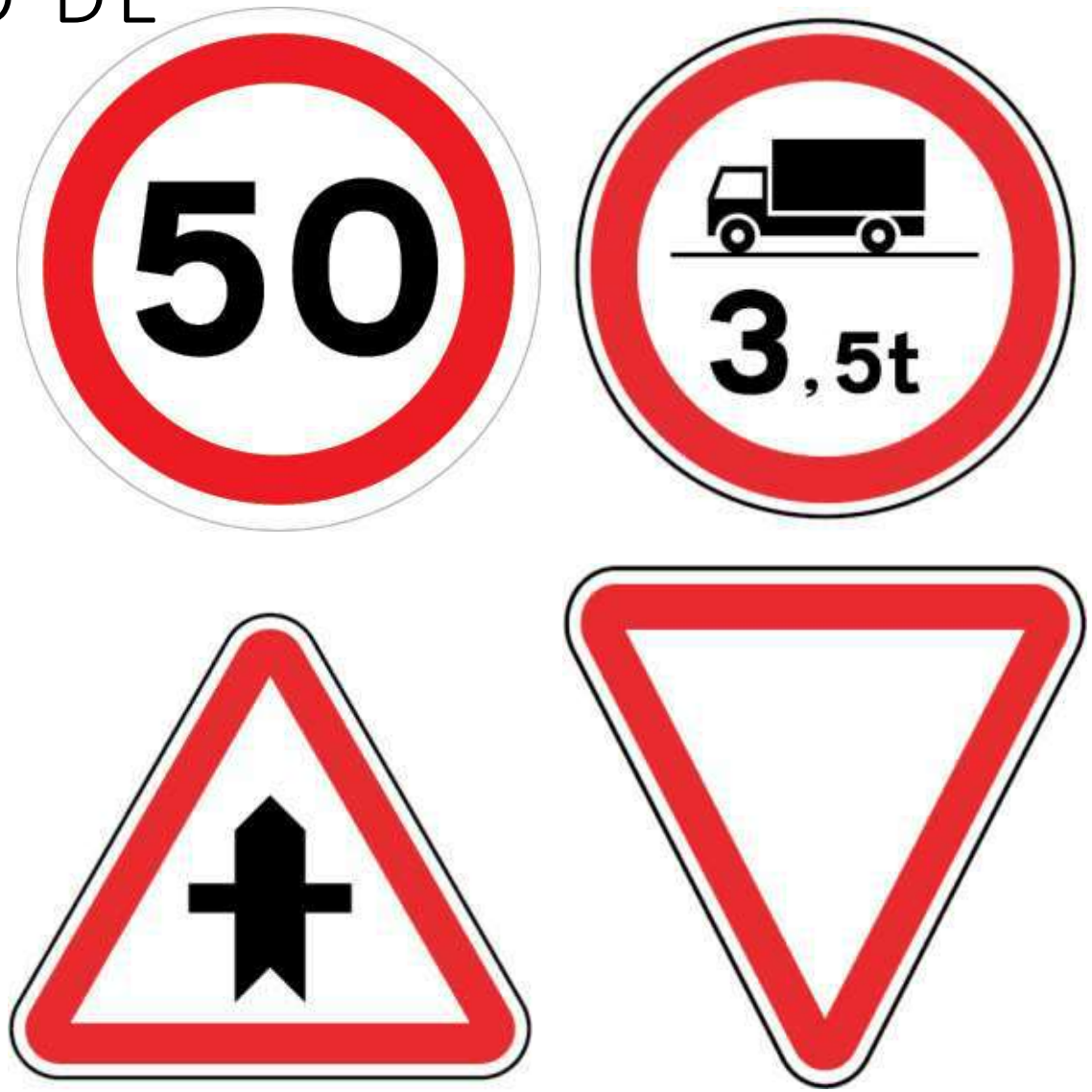
```
Melhores hiperparâmetros encontrados:
Learning Rate: 0.00371436721357021 8
Dropout Rate: 0.27767481630082097
```

```
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
```



4. CONCLUSÕES E DISCUSSÃO DE RESULTADOS

- GWO: 99,4% de accuracy
- PSO: 98,65% de accuracy
- AUC alta em ambos os algoritmos
- GWO e PSO são confiáveis para otimizar híper-parâmetros
- Desempenho consistente, destacando a eficácia das técnicas de otimização inspiradas em enxames para resolver problemas complexos



5. REFERÊNCIAS

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