Instituto Superior de Engenharia de Coimbra

LICENCIATURA EM ENGENHARIA INFORMÁTICA

INTELIGÊNCIA COMPUTACIONAL
23/24

AULA LABORATORIAL N.º 06

INTELIGÊNCIA DE ENXAME
PSO - Particle Swarm Optimization

---

1. Objetivos

Pretende-se com este trabalho atingir-se os seguintes resultados de aprendizagem:

- Compreender os mecanismos de funcionamento dos algoritmos de inteligência de enxame, em particular o algoritmo conhecido por PSO - *Particle Swarm Optimization*.
- Aplicar o algoritmo a um problema de otimização e analisar o seu desempenho.
- Aplicar o algoritmo a um problema de treino de uma rede neuronal.
- Aplicar ao algoritmo a um problema de seleção de "características" - atribuição de pesos às entradas dos modelos para problemas de classificação e regressão.

2. Algoritmo PSO

Um "swarm" define-se como um conjunto estruturado de indivíduos (ou agentes) que interagem entre si para atingirem um **objetivo comum**, de forma mais eficiente do que agindo individualmente.

Os indivíduos são relativamente simples, contudo o **comportamento coletivo pode ser complexo** – analogia com colónias de formigas, enxames, bandos de pássaros, cardumes, etc.

O algoritmo funciona de acordo com os seguintes princípios:

- Cada partícula representa uma possível solução para um problema de otimização;
- A posição de uma partícula é determinada iterativamente de acordo com a sua inércia, "experiência individual" e "experiência das partículas vizinhas" (Figura 1);

Em anexo (anexo A) apresenta-se o pseudo - código com a variante mais usual do algoritmo, em que todas as partículas comunicam entre si – versão conhecida como “global best”.

Disponibiliza-se ainda uma implementação do algoritmo básico (Rooy, 2016) com aplicação a um problema simples de otimização – função quadrática.

Analise a implementação básica do algoritmo.

1. Represente o diagrama de classes;
2. Avalie o comportamento do algoritmo para as seguintes situações:
   - Constante social = 0
   - Constante cognitiva = 0
   - Constante de inércia=0
   - Quantas partículas necessita para atingir uma boa solução em menos de 10 iterações?
3. Que conclusão retira acerca do papel das constantes cognitiva, social, inércia e número de partículas a utilizar?

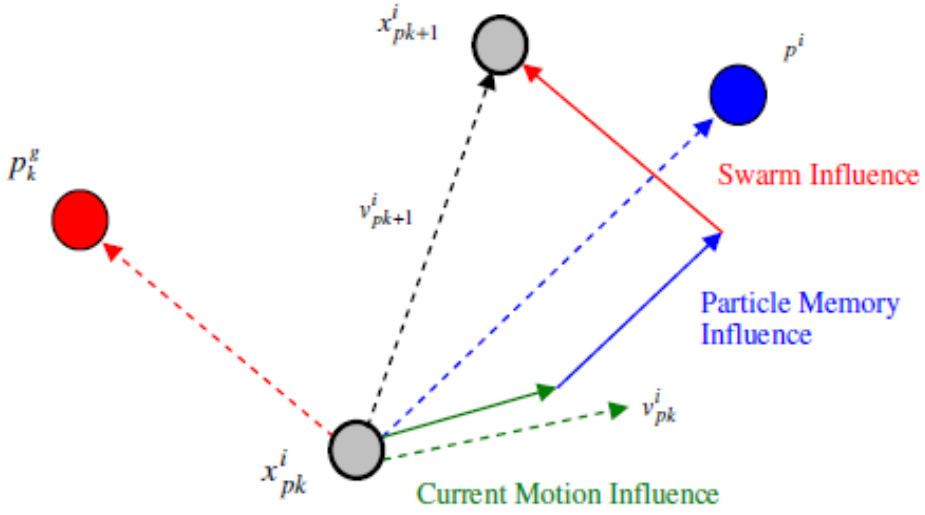

Figura 1. Cálculo da posição de uma partícula.

2.1 Biblioteca ‘PySwarms’

A biblioteca Pyswarms disponibiliza a implementação do algoritmo PSO e algumas das suas variantes - https://pyswarms.readthedocs.io/en/latest/installation.html.

Adicione esta package ao seu ambiente, por exemplo a partir do Anaconda *prompt*:

```
(base) C:\Users\cpereira\Documents>pip install pyswarms
```

3. Problema de otimização

Resolva novamente o problema de otimização da função “quadrática”.

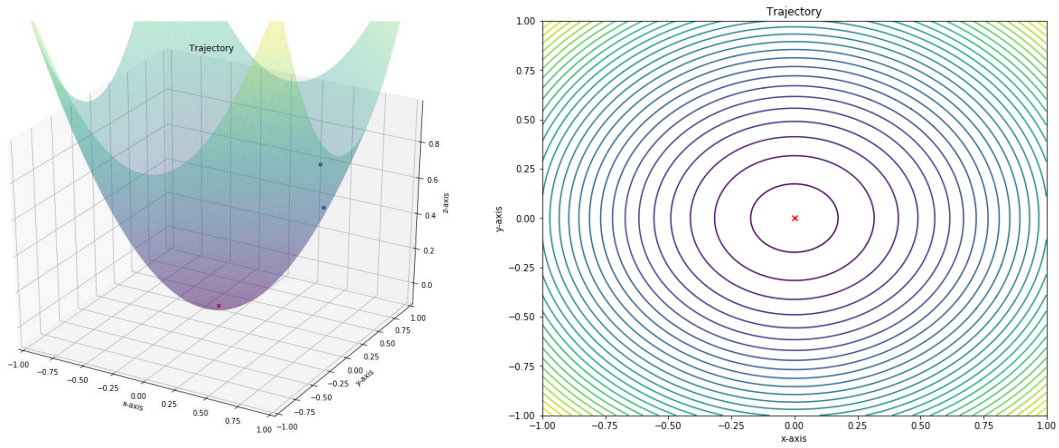

Figura 2. Função quadrática.

Defina as opções e modelo de otimização do PSO, considerando duas dimensões do problema e variando o número de partículas e opções.

```python
# Import modules
import numpy as np

# Import PySwarms
import pyswarms as ps
from pyswarms.utils.functions import single_obj as fx


# Set-up hyperparameters
options = {'c1': 0.5, 'c2': 0.3, 'w':0.9}

# Call instance of PSO
optimizer = ps.single.GlobalBestPSO(n_particles=10, dimensions=2, options=options)

# Perform optimization
cost, pos = optimizer.optimize(fx.sphere, iters=1000, verbose=3)
```

Resolva o mesmo problema na versão *localbest*, que implica a definição de uma vizinhança.

Avalie o impacto do tamanho da vizinhança para convergência do algoritmo (relação número de iterações/tamanho da vizinhança).

```
# Set-up hyperparameters
options = {'c1': 0.5, 'c2': 0.3, 'w':0.9, 'k': 2, 'p': 2}

# Call instance of PSO
optimizer = ps.single.LocalBestPSO(n_particles=10, dimensions=2, options=options)

# Perform optimization
cost, pos = optimizer.optimize(fx.sphere, iters=1000)
```

Teste diferentes topologias de vizinhanças:

```
import pyswarms as ps
from pyswarms.backend.topology import Pyramid
from pyswarms.utils.functions import single_obj as fx

# Set-up hyperparameters and topology
options = {'c1': 0.5, 'c2': 0.3, 'w':0.9}
my_topology = Pyramid(static=False)

# Call instance of GlobalBestPSO
optimizer = ps.single.GeneralOptimizerPSO(n_particles=10, dimensions=2,
                                    options=options, topology=my_topology)

# Perform optimization
stats = optimizer.optimize(fx.sphere, iters=100)
```

Estão definidas as seguintes topologias nas versões estáticas (vizinhos fixos á partida) e dinâmicas (Figura 3):

- Estrela – todas as partículas conectadas
- Anel (versão estática e dinâmica). Consiste na conexão de cada partícula a ‘k’ vizinhos.
- VonNeumann
- Pirâmide (versão estática e dinâmica)
- Aleatória (versão estática e dinâmica) – vizinhança com k partículas escolhidas aleatoriamente.

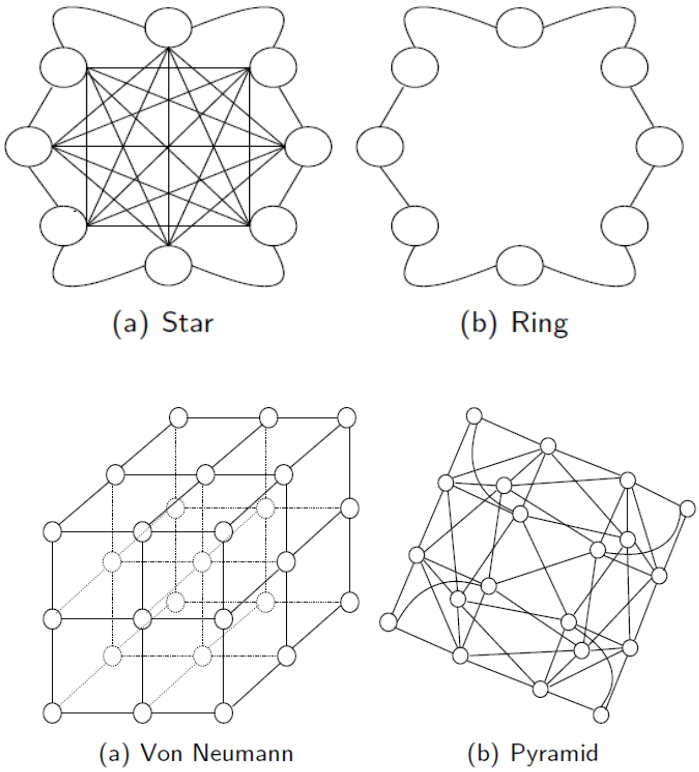

Figura 3. Topologias

4. Problema com ótimos locais

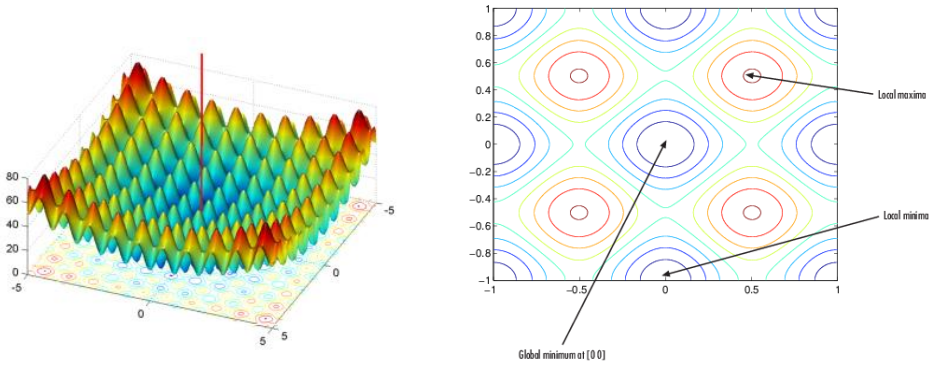

Figura 4. Função de Rastrigin.

Considere um problema de maior complexidade que o anterior, a função de Rastringin que se caracteria pelo elevado número de ótimos locais (Figura 4).

A bilblioteca disponibiliza vários métodos para representaçao de resultados:

```
"""
plot performance
"""
from pyswarms.utils.plotters import plot_cost_history, plot_contour, plot_surface
import matplotlib.pyplot as plt

plot_cost_history(optimizer.cost_history)
plt.show()
```

1. Avalie o algoritmo para 10, 100 e 1000 iterações.
2. Avalie o algoritmo para 1,2,5 e 10 partículas.
3. Repita cada experiência 10 vezes e analise os resultados registando o 'fitness' das melhores soluções encontradas (best, valor médio e desvio padrão).
4. Como pode adaptar o algoritmo para ter um comportamento do tipo 'hill-climbing'? Compare os resultados ao nível da capacidade de ultrapassar ótimos locais.

5. Treino de uma Rede Neuronal

Implemente o algoritmo PSO para treinar uma rede neuronal aplicada ao problema de classificação da "iris".

1. Importe os dados a partir da sklearn.datasets:

```
# Import modules
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris


# Import PySwarms
import pyswarms as ps


# Load the iris dataset
data = load_iris()
```

```
# Store the features as X and the labels as y
X = data.data
y = data.target
```

2. Defina uma rede MLP com 4 entradas (os atributos do problema da iris), uma camada interna com 20 neurónios e 3 saídas, que correspondem às classes de iris (setosa, viriginica, versicolor).

   As características da rede MLP são assim:
   - Número de entradas: 4
   - Camada interna: 20 neurónios com função de ativação “tanh”
   - Saídas: 3 neurónios de saída com função “softmax”

Temos neste caso (4∗20)+(20∗3)+20+3 = **163 parâmetros**, o que representa um problema com 163 dimensões!

```
# Forward propagation
def forward_prop(params):

# Neural network architecture

    n_inputs = 4
    n_hidden = 20
    n_classes = 3

    # Roll-back the weights and biases
    W1 = params[0:80].reshape((n_inputs,n_hidden))
    b1 = params[80:100].reshape((n_hidden,))
    W2 = params[100:160].reshape((n_hidden,n_classes))
    b2 = params[160:163].reshape((n_classes,))

    # Perform forward propagation
    z1 = X.dot(W1) + b1  # Pre-activation in Layer 1
    a1 = np.tanh(z1)     # Activation in Layer 1
    z2 = a1.dot(W2) + b2 # Pre-activation in Layer 2
    logits = z2          # Logits for Layer 2

    # Compute for the softmax of the logits
    exp_scores = np.exp(logits)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    # Compute for the negative log likelihood
    N = 150 # Number of samples
    corect_logprobs = -np.log(probs[range(N), y])
    loss = np.sum(corect_logprobs) / N

    return loss
```

Cada partícula corresponde ao conjunto de parâmetros (pesos) da rede neuronal.

A qualidade - “fitness” de cada partícula corresponde ao erro de treino da rede neuronal.

```
def f(x):
    """Higher-level method to do forward_prop in the
    whole swarm.

    Inputs
    ------
    x: numpy.ndarray of shape (n_particles, dimensions)
        The swarm that will perform the search

    Returns
    -------
    numpy.ndarray of shape (n_particles, )
        The computed loss for each particle
    """
    n_particles = x.shape[0]
    j = [forward_prop(x[i]) for i in range(n_particles)]
    return np.array(j)
```

3. Treine o modelo com a versão PSO “Global best”

```
# Initialize swarm
options = {'c1': 0.5, 'c2': 0.3, 'w':0.9}

# Call instance of PSO
dimensions = (4 * 20) + (20 * 3) + 20 + 3
optimizer = ps.single.GlobalBestPSO(n_particles=100, dimensions=dimensions,
options=options)

# Perform optimization
cost, pos = optimizer.optimize(f, iters=1000)
```

4. Podemos agora testar a rede com novos exemplos:

```
def predict(X, pos):
    """
    Use the trained weights to perform class predictions.

    Inputs
    ------
    X: numpy.ndarray
        Input Iris dataset
    pos: numpy.ndarray
        Position matrix found by the swarm. Will be rolled
        into weights and biases.
    """
    # Neural network architecture
    n_inputs = 4
    n_hidden = 20
    n_classes = 3

    # Roll-back the weights and biases
    W1 = pos[0:80].reshape((n_inputs,n_hidden))
    b1 = pos[80:100].reshape((n_hidden,))
    W2 = pos[100:160].reshape((n_hidden,n_classes))
    b2 = pos[160:163].reshape((n_classes,))

    # Perform forward propagation
```

```
    z1 = X.dot(W1) + b1  # Pre-activation in Layer 1
    a1 = np.tanh(z1)     # Activation in Layer 1
    z2 = a1.dot(W2) + b2 # Pre-activation in Layer 2
    logits = z2          # Logits for Layer 2

    y_pred = np.argmax(logits, axis=1)
    return y_pred
```

5. A avaliação de desempenho da rede treinada com o PSO pode ser avaliação por:

```
acc=(predict(X, pos) == y).mean()
print(acc)
```

Para o problema da Iris, forme um conjunto de treino e teste (20%).

1. Treine a rede neuronal com o PSO e apresente a matriz de confusão.
2. Varie os parâmetros do PSO e procure encontrar o melhor balanço entre precisão e recall (f1-measure).

6. PSO Binário para seleção de características

Vamos aplicar a implementação da versão discreta do PSO para selecionar as características (*features*) de um problema de classificação.
No PSO binário a posição das partículas é representada por um array de variáveis binárias (0 ou 1);

$$X=[x1,x2,x3,\ldots,xd] \quad onde \quad xi \in 0,1$$

Cada posição representa uma característica (*feature*) do problema. O valor de '1' significa que essa característica é considerada como "entrada" para o classificador.

A 'fitness' de cada partícula é calculada através de uma fórmula que representa um compromisso entre o desempenho do classificador (P) e o número de features (Nf) relativamente valor total (Nt):

$$f(X)=\alpha(1-P)+(1-\alpha)(1-Nf/Nt)$$

```
# Import modules
import numpy as np
import random

# Import PySwarms
import pyswarms as ps

RANDOM_SEED = 42
random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
```

Considere-se um “dataset” virtual com 15 features: 5 informativas, 5 redundantes e 2 repetidas (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html):

```
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=100, n_features=15, n_classes=3,
                           n_informative=5, n_redundant=5, n_repeated=2,
                           random_state=1)
```

Defina-se o seguinte classificador,

```
from sklearn import linear_model
classifier = linear_model.LogisticRegression()
```

e a função objetivo:

```
def f_per_particle(m, alpha):
    """Computes for the objective function per particle

    Inputs
    ------
    m : numpy.ndarray
        Binary mask that can be obtained from BinaryPSO, will
        be used to mask features.
    alpha: float (default is 0.5)
        Constant weight for trading-off classifier performance
        and number of features

    Returns
    -------
    numpy.ndarray
        Computed objective function
    """
    total_features = 15
    # Get the subset of the features from the binary mask
    if np.count_nonzero(m) == 0:
        X_subset = X
    else:
        X_subset = X[:,m==1]
    # Perform classification and store performance in P
    classifier.fit(X_subset, y)
    P = (classifier.predict(X_subset) == y).mean()
    # Compute for the objective function
    j = (alpha * (1.0 - P)
        + (1.0 - alpha) * (1 - (X_subset.shape[1] / total_features)))

    return j
```

```
def f(x, alpha=0.88):
    """Higher-level method to do classification in the
    whole swarm.

    Inputs
    ------
    x: numpy.ndarray of shape (n_particles, dimensions)
        The swarm that will perform the search

    Returns
    -------
    numpy.ndarray of shape (n_particles, )
        The computed loss for each particle
    """
    n_particles = x.shape[0]
    j = [f_per_particle(x[i], alpha) for i in range(n_particles)]
    return np.array(j)
```

Considere-se a implementação do PSO binário:

```
# Initialize swarm, arbitrary
options = {'c1': 0.5, 'c2': 0.5, 'w':0.9, 'k': 30, 'p':2}

# Call instance of PSO
dimensions = 15 # dimensions should be the number of features
optimizer = ps.discrete.BinaryPSO(n_particles=30, dimensions=dimensions,
options=options)

cost, pos = optimizer.optimize(f, iters=1000)
```

Avalie-se finalmente o desempenho do classificador:

```
# Create two instances of LogisticRegression
classifier = linear_model.LogisticRegression()
# Get the selected features from the final positions
X_selected_features = X[:,pos==1]  # subset
# Perform classification and store performance in P
classifier.fit(X_selected_features, y)
# Compute performance
subset_performance = (classifier.predict(X_selected_features) == y).mean()
print('Subset performance: %.3f' % (subset_performance))
```

Para este problema:

1. Forme um conjunto independente de teste.
2. Compare com o desempenho do classificador obtido com um classificador completo.

3. Tente melhorar o resultado variando os parâmetros do classificador e função objetivo.

Referências

- Miranda L.J., (2018). PySwarms: a research toolkit for Particle Swarm Optimization in Python. Journal of Open Source Software, 3(21), 433, https://doi.org/joss.00433 https://pyswarms.readthedocs.io/en/latest/
- Engelbrecht, “Computational Intelligence, An Introduction” John Wiley & Sons, 2n edition.
- https://github.com/rapidminer/python-rapidminer
- https://niapy.org/en/stable/tutorials/hyperparameter_optimization.html

Anexo A

PSO versão “global best” – topologia em estrela.

1. Inicializar o *swarm* de partículas $P(t)$ para t=0, de tal forma que a posição $P_i \in P(t)$, de cada partícula $x_i(t)$ é aleatória, dentro do espaço multidimensional de procura.
2. Avaliar a performance de cada partícula na sua posição atual $F(x_i(t))$
3. Comparar a performance de cada indivíduo com a sua melhor performance:
   Se $F(x_i(t)) < pbest_i$ Então:
   (a) $pbest_i = F(x_i(t))$
   (b) $xbest_i = x_i(t)$
4. Comparar a performance de cada indivíduo com melhor performance global:
   Se $F(x_i(t)) < gbest$ Então:
   (a) $gbest = F(x_i(t))$
   (b) $xgbest = x_i(t)$
5. Atualiza a velocidade de cada partícula:
   $v_i(t) = v_i(t-1) + \rho_1(xbest_i - x_i(t)) + \rho_2(xgbest_i - x_i(t))$
   onde ρ_1 e ρ_2 representam valores aleatórios positivos, respetivamente designados de componente cognitiva e componente social.
6. Move cada partícula para a sua nova posição:
   (a) $x_i(t) = x_i(t-1) + v_i(t)$
   (b) $t = t+1$
7. Volta ao passo 2 e repete até convergência do algoritmo.