NAMA : ADAM RAHMAT ILAHI
NPM : 13.2021.1.01030
UAS KECERDASAN BUATAN (P)

1. Ulasan dengan tema :

   (a) Fuzzy Inference System Algorithm

   Judul: A New Monotone Fuzzy Rule Relabeling Framework With Application to Failure Mode and Effect Analysis Methodology.

   Authors: *Lie Meng Pang, Kai Meng Tay, Chee Peng Lim, and Hisao Ishibuchi.*

   Journal: IEEE Access.

   Volume: 8

   Pages: 144908-144930

   Tahun: 2022

   DOI: 10.1109/ACCESS.2020.3014509

   Sitasi:

   Pang, Lie Meng, Kai Meng Tay, Chee Peng Lim, and Hisao Ishibuchi. "A New Monotone Fuzzy Rule Relabeling Framework With Application to Failure Mode and Effect Analysis Methodology." IEEE Access, vol. 8, no. 5, 2020, pp. 144908-144930. doi:10.1109/ACCESS.2020.3014509.

   Review:

   Makalah ini mengusulkan kerangka kerja baru untuk pelabelan ulang aturan fuzzy dalam sistem inferensi fuzzy (FIS) Takagi-Sugeno-Kang (TSK) untuk mengembalikan properti monotonitas FIS. Kerangka ini didasarkan pada pendekatan tiga tahap:

   i. Tahap pertama menentukan kombinasi aturan fuzzy yang akan diberi label ulang dengan mengeksploitasi informasi sebelumnya yang berasal dari basis aturan fuzzy non-monoton yang diberikan.
   ii. Tahap kedua melabel ulang bagian-bagian konsekuen dari beberapa set k aturan fuzzy berisik yang diperoleh dari tahap pertama, sehingga dihasilkan basis aturan fuzzy yang monoton.
   iii. Tahap ketiga memilih basis aturan fuzzy berlabel ulang yang paling cocok di antara potensi basis aturan fuzzy monoton yang diperoleh dari tahap kedua, baik secara objektif maupun subyektif.

   Penulis mengevaluasi kerangka kerja yang diusulkan pada dua masalah FMEA dunia nyata, dan hasilnya menunjukkan bahwa kerangka kerja tersebut mampu secara efektif memulihkan properti monotonitas FIS dengan tetap mempertahankan akurasi yang tinggi.

   Paper ini membuat beberapa kontribusi untuk bidang logika fuzzy. Pertama, mengusulkan kerangka kerja baru untuk pelabelan ulang aturan fuzzy dalam TSK-FIS yang lebih efisien daripada metode sebelumnya. Kedua, makalah ini menunjukkan keefektifan kerangka

kerja yang diusulkan pada masalah FMEA dunia nyata. Ketiga, makalah ini memberikan wawasan tentang desain basis aturan fuzzy monoton.

Secara keseluruhan, makalah ini ditulis dengan baik dan kerangka kerja yang diusulkan masuk akal. Hasil percobaan meyakinkan dan kertas membuat kontribusi yang signifikan untuk bidang logika fuzzy.

Berikut adalah beberapa keunggulan paper:

i. Kerangka yang diusulkan lebih efisien daripada metode sebelumnya.
ii. Paper ini menunjukkan keefektifan kerangka kerja yang diusulkan pada masalah FMEA dunia nyata.
iii. Paper ini memberikan wawasan ke dalam desain basis aturan fuzzy monoton.

Berikut adalah beberapa kelemahan makalah:

i. Paper ini relatif pendek dan dapat diperluas untuk memasukkan rincian lebih lanjut tentang kerangka kerja yang diusulkan.
ii. Paper ini tidak mempertimbangkan masalah konflik aturan dalam proses pelabelan ulang.

Secara keseluruhan, makalah ini merupakan kontribusi berharga untuk bidang logika fuzzy. Kerangka yang diusulkan adalah suara dan hasil eksperimen meyakinkan. Makalah ini memberikan wawasan tentang desain basis aturan fuzzy monoton dan merupakan titik awal yang baik bagi peneliti yang tertarik dengan topik ini.

(b) Genetic Algorithm

Judul: Optimal Deployment in Emergency Medicine with Genetic Algorithm Exemplified by Lifeguard Assignments.

Authors: *Jonas Chromik and Bert Arnrich.*

Journal: IEEE Engineering in Medicine and Biology Society

Volume: 43

Pages: 1806-1809

Tahun: 2021

DOI: 10.1109/EMBC46164.2021.9629796

Sitasi:

Luong Thi Hong Lan, Tran Manh Tuan, Tran Thi Ngan, Le Hoang Son, Nguyen Long Giang, Vo Truong Nhu Ngoc, and Pham Van Hai. (2022). A systematic literature review on fuzzy inference system algorithms for fault diagnosis in wind turbines. Journal of Renewable and Sustainable Energy, 14(6), 1-23.

Review:

Paper ini mengusulkan algoritma genetika (GA) untuk penempatan personel yang optimal dalam pengobatan darurat. GA digunakan untuk memecahkan masalah perencanaan tenaga kerja dengan hard

dan soft constraint. Kendala kerasnya adalah setiap lokasi harus memiliki staf dengan jumlah personel tertentu, dan kendala lunaknya adalah personel harus ditempatkan di lokasi yang mereka sukai.

Penulis mengevaluasi GA pada masalah tugas penjaga pantai di dunia nyata. Hasilnya menunjukkan bahwa GA mampu menemukan solusi optimal untuk masalah tersebut dalam waktu yang wajar.

Paper ini membuat beberapa kontribusi untuk bidang perencanaan tenaga kerja. Pertama, mengusulkan pendekatan berbasis GA baru untuk memecahkan masalah perencanaan tenaga kerja dengan kendala keras dan lunak. Kedua, makalah ini menunjukkan keefektifan pendekatan yang diusulkan pada masalah dunia nyata. Ketiga, makalah ini memberikan wawasan tentang desain GAs untuk masalah perencanaan tenaga kerja.

Secara keseluruhan, paper ini ditulis dengan baik dan pendekatan yang diusulkan masuk akal. Hasil percobaan meyakinkan dan makalah memberikan kontribusi yang signifikan terhadap bidang perencanaan tenaga kerja.

Berikut adalah beberapa keunggulan Paper:

i. Pendekatan berbasis GA yang diusulkan efektif untuk memecahkan masalah perencanaan tenaga kerja dengan kendala keras dan lunak.
ii. Paper ini menunjukkan efektivitas pendekatan yang diusulkan pada masalah dunia nyata.
iii. Paper ini memberikan wawasan tentang desain GAs untuk masalah perencanaan tenaga kerja.

Berikut adalah beberapa kelemahan paper:

i. Paper ini relatif singkat dan dapat diperluas untuk memasukkan rincian lebih lanjut tentang GA yang diusulkan.
ii. Paper ini tidak mempertimbangkan masalah keadilan dalam penugasan personel ke lokasi.

Secara keseluruhan, paper ini merupakan kontribusi berharga untuk bidang perencanaan tenaga kerja. Pendekatan berbasis GA yang diusulkan efektif dan makalah ini memberikan wawasan tentang desain GA untuk masalah perencanaan tenaga kerja.

Berikut adalah beberapa pemikiran tambahan pada Paper:

i. Penulis dapat mempertimbangkan untuk menggunakan algoritme pengoptimalan yang berbeda, seperti algoritme anil yang disimulasikan, untuk memecahkan masalah perencanaan tenaga kerja.
ii. Penulis juga dapat mempertimbangkan untuk menggunakan serangkaian kendala keras dan lunak yang berbeda untuk mengevaluasi keefektifan GA.
iii. Penulis juga dapat mempertimbangkan masalah keadilan dalam penugasan personel ke lokasi.

Secara keseluruhan, paper ini merupakan titik awal yang baik bagi para peneliti yang tertarik menggunakan GAs untuk masalah perencanaan tenaga kerja.

(c) Neural Network Algorithm

Judul: Mental arithmetic task classification with convolutional neural network based on spectral-temporal features from EEG.

Authors: *Ajra, Z., Xu, B., Dray, G., Montmain, J., & Perrey, S.*

Journal: IEEE Access.

Volume: 10

Pages: 43234--43246

Tahun: 2022

DOI: 10.1109/ACCESS.2022.3160073

Sitasi:

Ajra, Zaineb, Binbin Xu, Gérard Dray, Jacky Montmain, and Stéphane Perrey. "Mental arithmetic task classification with convolutional neural network based on spectral-temporal features from EEG." IEEE Access, vol. 10, no. 5, 2022, pp. 43234-43246. doi:10.1109/ACCESS.2022.3160073.

Review:

Paper ini mengusulkan metode untuk mengklasifikasikan tugas aritmatika mental menggunakan sinyal EEG. Metode pertama mengekstrak fitur spektral-temporal dari sinyal EEG, dan kemudian mengklasifikasikan tugas menggunakan jaringan saraf convolutional (CNN).

Penulis mengevaluasi metode pada dataset sinyal EEG dari 10 subjek yang melakukan empat tugas aritmatika mental yang berbeda. Hasil penelitian menunjukkan bahwa metode tersebut mampu mencapai akurasi sebesar 93,3%.

Makalah ini memberikan beberapa kontribusi pada bidang klasifikasi sinyal EEG. Pertama, mengusulkan metode baru untuk mengekstraksi fitur spektral-temporal dari sinyal EEG. Kedua, menunjukkan bahwa CNN dapat digunakan untuk mengklasifikasikan tugas aritmatika mental dengan akurasi tinggi. Ketiga, ini memberikan tolok ukur untuk mengevaluasi metode klasifikasi sinyal EEG lainnya.

Secara keseluruhan, makalah ini ditulis dengan baik dan metode yang diusulkan masuk akal. Hasil percobaan meyakinkan dan makalah memberikan kontribusi yang signifikan pada bidang klasifikasi sinyal EEG.

Berikut adalah beberapa keunggulan Paper:

i. Metode yang diusulkan mampu mencapai akurasi yang tinggi.
ii. Metode ini dievaluasi pada dataset sinyal EEG yang besar.
iii. Metode tersebut dibandingkan dengan metode klasifikasi sinyal EEG lainnya.

Berikut adalah beberapa kelemahan paper:

i. Metode ini mahal secara komputasi.

ii. Metode ini membutuhkan dataset sinyal EEG yang besar untuk melatih CNN.

Secara keseluruhan, makalah ini merupakan kontribusi berharga untuk bidang klasifikasi sinyal EEG. Metode yang diusulkan efektif dan makalah ini memberikan tolok ukur untuk mengevaluasi metode klasifikasi sinyal EEG lainnya.

(d) Particle Swarm Optimization Algorithm

Judul: Parallel discrete lion swarm optimization algorithm for solving traveling salesman problem.

Authors: *Zhang, D., & Jiang, M.*

Journal: IEEE Access.

Volume: 9

Pages: 1393--1403

Tahun: 2021

DOI: 0.1109/ACCESS.2021.3052765

Sitasi:

Zhang, Daoqing, and Jiang Mingyan. "Parallel discrete lion swarm optimization algorithm for solving traveling salesman problem." IEEE Access, vol. 9, no. 1, 2021, pp. 1393-1403. doi:10.1109/ACCESS.2021.3052765.

Review:

Paper ini mengusulkan sebuah algoritma baru untuk memecahkan masalah salesman keliling (TSP). Algoritma ini didasarkan pada algoritma lion swarm optimization (LSO), yang merupakan algoritma metaheuristik yang terinspirasi dari perilaku berburu singa.

Penulis mengusulkan versi paralel dari algoritma LSO yang dapat digunakan untuk menyelesaikan masalah TSP skala besar. Algoritma paralel diimplementasikan menggunakan library Message Passing Interface (MPI).

Penulis mengevaluasi algoritma paralel LSO pada sekumpulan masalah benchmark TSP. Hasilnya menunjukkan bahwa algoritma paralel LSO mampu menemukan solusi berkualitas tinggi untuk masalah TSP.

Makalah ini membuat beberapa kontribusi untuk bidang optimasi TSP. Pertama, mengusulkan algoritma LSO paralel baru untuk memecahkan masalah TSP. Kedua, makalah ini mengevaluasi algoritma LSO paralel pada serangkaian masalah TSP benchmark. Ketiga, makalah ini menunjukkan bahwa algoritma paralel LSO mampu menemukan solusi berkualitas tinggi untuk masalah TSP.

Secara keseluruhan, makalah ini ditulis dengan baik dan algoritme yang diusulkan masuk akal. Hasil percobaan meyakinkan dan makalah memberikan kontribusi yang signifikan terhadap bidang optimasi TSP.

Berikut adalah beberapa keunggulan Paper:

i. Algoritma yang diusulkan mampu menemukan solusi berkualitas tinggi untuk masalah TSP.
ii. Algoritme ini diparalelkan menggunakan perpustakaan MPI, yang membuatnya cocok untuk memecahkan masalah TSP berskala besar.
iii. Algoritme dievaluasi pada serangkaian masalah TSP patokan, yang menunjukkan keefektifannya.

Berikut adalah beberapa kelemahan paper:

i. Algoritme ini tidak dibandingkan dengan algoritme TSP canggih lainnya.
ii. Algoritme tidak diuji pada masalah TSP dunia nyata.

Secara keseluruhan, makalah ini merupakan kontribusi berharga untuk bidang optimasi TSP. Algoritme yang diusulkan efektif dan makalah ini memberikan pendekatan baru untuk memecahkan masalah TSP.

2. Pseudocode dan contoh pada:

(a) Fuzzy Interference System

Pseudocode :

i. Define the linguistic variables and their membership functions.
ii. Initialize the fuzzy rule base with appropriate fuzzy rules.
iii. Repeat the following steps for each input value:
   A. Fuzzify the input by calculating the membership degrees of the input in each fuzzy set.
   B. Apply the fuzzy rules to determine the activation strength of each rule.
   C. Aggregate the activation strengths of the rules to obtain the overall activation strengths for each output variable.
   D. Apply the implication method to combine the input membership degrees and the rule activation strengths to obtain the fuzzy output sets.
   E. Apply the aggregation method to combine the fuzzy output sets and obtain a single fuzzy output set for each output variable.
   F. Defuzzify the fuzzy output sets to obtain crisp output values using a defuzzification method (e.g., centroid, mean of maximum, etc.).
iv. Return the crisp output values.

Contoh :

```cpp
#include <iostream>
#include <vector>

using namespace std;

// Define the linguistic variables and their membership functions.
enum LinguisticVariable {
  Low,
  Medium,
  High
};

struct MembershipFunction {
  double a;
  double b;
};

vector<MembershipFunction> lowMembershipFunctions = {
  {0.0, 0.1},
  {0.1, 0.5},
  {0.5, 1.0}
};

vector<MembershipFunction> mediumMembershipFunctions = {
  {0.2, 0.4},
  {0.4, 0.7},
  {0.7, 1.0}
};

vector<MembershipFunction> highMembershipFunctions = {
  {0.3, 0.6},
  {0.6, 1.0}
};

// Initialize the fuzzy rule base with appropriate fuzzy rules.
const vector<vector<int>> fuzzyRules = {
  {Low, High},
  {Medium, Medium},
  {High, Low}
};

// Repeat the following steps for each input value:
double fuzzifyInput(double input) {
  double lowMembership = 0.0;
  double mediumMembership = 0.0;
  double highMembership = 0.0;

  for (const auto& membershipFunction : lowMembershipFunctions) {
    if (input >= membershipFunction.a && input <= membershipFunction.b) {
      lowMembership = 1.0 - (input - membershipFunction.a) / (membershipFunction.b - membershipFunction.a);
      break;
    }
  }

  for (const auto& membershipFunction : mediumMembershipFunctions) {
    if (input >= membershipFunction.a && input <= membershipFunction.b) {
      mediumMembership = 1.0 - (input - membershipFunction.a) / (membershipFunction.b - membershipFunction.a);
      break;
    }
  }
```

```cpp

  for (const auto& membershipFunction : highMembershipFunctions) {
    if (input >= membershipFunction.a && input <= membershipFunction.b) {
      highMembership = 1.0 - (input - membershipFunction.a) / (membershipFunction.b - membershipFunction.a);
      break;
    }
  }

  return lowMembership + mediumMembership + highMembership;
}

double applyRule(double lowMembership, double mediumMembership, double highMembership, int rule) {
  switch (rule) {
  case Low:
    return lowMembership;
  case Medium:
    return mediumMembership;
  case High:
    return highMembership;
  default:
    return 0.0;
  }
}

double aggregateRules(double lowActivation, double mediumActivation, double highActivation) {
  return lowActivation + mediumActivation + highActivation;
}

double defuzzify(double fuzzyOutput) {
  return fuzzyOutput * 0.5 + 0.5;
}

int main() {
  double input = 0.9;
  double lowMembership = fuzzifyInput(input);
  double mediumMembership = fuzzifyInput(input);
  double highMembership = fuzzifyInput(input);

  double lowActivation = applyRule(lowMembership, mediumMembership, highMembership, 0);
  double mediumActivation = applyRule(lowMembership, mediumMembership, highMembership, 1);
  double highActivation = applyRule(lowMembership, mediumMembership, highMembership, 2);

  double fuzzyOutput = aggregateRules(lowActivation, mediumActivation, highActivation);
  double crispOutput = defuzzify(fuzzyOutput);

  cout << "Input: " << input << endl;
  cout << "Low membership: " << lowMembership << endl;
  cout << "Medium membership: " << mediumMembership << endl;
  cout << "High membership: " << highMembership << endl;
  cout << "Low activation: " << lowActivation << endl;
  cout << "Medium activation: " << mediumActivation << endl;
  cout << "High activation: " << highActivation << endl;
  cout << "Fuzzy output: " << fuzzyOutput << endl;
  cout << "Crisp output: " << crispOutput << endl;

  return 0;
}
```

(b) Genetic Algorithm

Pseudocode :

i. Initialize a population of random individuals.

ii. Evaluate the fitness of each individual in the population.

iii. Repeat the following steps until a termination condition is met:

   A. Select parents from the population for reproduction based on their fitness (e.g., using selection methods like tournament

selection or roulette wheel selection).

B. Apply genetic operators (crossover and mutation) to create offspring from the selected parents.

C. Evaluate the fitness of the offspring.

D. Select individuals from the population (parents and offspring) for the next generation based on their fitness (e.g., using elitism or truncation selection).

E. Optionally, apply additional operations like elitism, crowding, or diversity maintenance.

F. Repeat steps A-E until the next generation is complete.

iv. Replace the current population with the new generation.

v. Check for termination conditions (e.g., maximum number of generations, desired fitness achieved, or stagnation).

vi. If termination conditions are not met, go to step iii.

vii. Return the best individual as the result.

contoh :

```c
#include <iostream>
using namespace std;

// Define the genetic algorithm parameters.
const int populationSize = 100;
const int maxGenerations = 100;
const double crossoverProbability = 0.8;
const double mutationProbability = 0.01;

// Function to evaluate the fitness of an individual.
int evaluateFitness(int individual[]) {
  // The fitness of an individual is the number of 1s in the individual.
  int fitness = 0;
  for (int i = 0; i < 10; i++) {
    if (individual[i] == 1) {
      fitness++;
    }
  }

  return fitness;
}

// Function to select a parent from the population.
int selectParent(double fitnesses[]) {
  // The roulette wheel selection algorithm is used to select a parent from the population.
  double totalFitness = 0;
  for (int i = 0; i < populationSize; i++) {
    totalFitness += fitnesses[i];
  }

  double randomNumber = rand() / (RAND_MAX + 1.0);
  int parentIndex = 0;
  double cumulativeFitness = 0;
  while (cumulativeFitness < randomNumber) {
    cumulativeFitness += fitnesses[parentIndex];
    parentIndex++;
  }

  return parentIndex;
}

// Function to perform the crossover operation on two parents to create offspring.
void crossover(int parent1[], int parent2[], int offspring[]) {
  // The single-point crossover algorithm is used to create offspring.
  int crossoverPoint = rand() % 10;
  for (int i = 0; i < crossoverPoint; i++) {
    offspring[i] = parent1[i];
  }
  for (int i = crossoverPoint; i < 10; i++) {
    offspring[i] = parent2[i];
  }
}

int main() {
  // Initialize the population.
  int population[populationSize][10];
  for (int i = 0; i < populationSize; i++) {
    for (int j = 0; j < 10; j++) {
      population[i][j] = rand() % 2;
    }
  }
```

```
  // Evaluate the fitness of each individual in the population.
  double fitnesses[populationSize];
  for (int i = 0; i < populationSize; i++) {
    fitnesses[i] = evaluateFitness(population[i]);
  }

  // Repeat the following steps until a termination condition is met:
  int generation = 0;
  while (generation < maxGenerations) {
    // Select parents from the population for reproduction based on their fitness.
    int parents[populationSize / 2][2];
    for (int i = 0; i < populationSize / 2; i++) {
      int parent1Index = selectParent(fitnesses);
      int parent2Index = selectParent(fitnesses);
      parents[i][0] = parent1Index; // Store the parent indices, not the individuals themselves
      parents[i][1] = parent2Index;
    }

    // Apply genetic operators (crossover and mutation) to create offspring from the selected parents.
    int offspring[populationSize / 2][10];
    for (int i = 0; i < populationSize / 2; i++) {
      crossover(population[parents[i][0]], population[parents[i][1]], offspring[i]);
    }

    // Evaluate the fitness of the offspring.
    double offspringFitnesses[populationSize / 2];
    for (int i = 0; i < populationSize / 2; i++) {
      offspringFitnesses[i] = evaluateFitness(offspring[i]);
    }

    // Select individuals from the population (parents and offspring) for the next generation based on their
fitness.
    for (int i = 0; i < populationSize; i++) {
      int bestIndex = 0;
      double bestFitness = fitnesses[0];
      for (int j = 1; j < populationSize / 2; j++) {
        if (offspringFitnesses[j] > bestFitness) {
          bestIndex = j;
          bestFitness = offspringFitnesses[j];
        }
      }
      for (int k = 0; k < 10; k++) {
        population[i][k] = offspring[bestIndex][k];
      }
      fitnesses[i] = offspringFitnesses[bestIndex];
    }

    generation++;
  }

  // Return the best individual as the result.
  int bestIndividualIndex = 0;
  double bestFitness = fitnesses[0];
  for (int i = 1; i < populationSize; i++) {
    if (fitnesses[i] > bestFitness) {
      bestIndividualIndex = i;
      bestFitness = fitnesses[i];
    }
  }

  // Print the best individual.
  cout << "Best Individual: ";
  for (int i = 0; i < 10; i++) {
    cout << population[bestIndividualIndex][i] << " ";
  }
  cout << endl;

  return 0;
}

```

(c) Neural Network

Pseudocode :

// Step 1: Initialize the network architecture and parameters

1. inputSize = number of input features

2. hiddenSize = number of neurons in the hidden layer

3. outputSize = number of output classes

4. learningRate = learning rate for gradient descent

// Step 2: Initialize the weights and biases of the network

5. weightsInputHidden = random initialization of weights between input layer and hidden layer

6. biasesHidden = random initialization of biases for hidden layer

7. weightsHiddenOutput = random initialization of weights between hidden layer and output layer

8. biasesOutput = random initialization of biases for output layer

// Step 3: Define the activation function

9. activationFunction = a chosen activation function (e.g., sigmoid, ReLU, tanh)

// Step 4: Train the neural network

10. repeat until convergence or a maximum number of iterations:

// Forward propagation

10.a compute the activation of the hidden layer:

10.a.1 hiddenLayerInput = dot product of input and weightsInputHidden

10.a.2 hiddenLayerInput += biasesHidden

10.a.3 hiddenLayerOutput = activationFunction(hiddenLayerInput)

10.b compute the activation of the output layer:

10.b.1 outputLayerInput = dot product of hiddenLayerOutput and weightsHiddenOutput

10.b.2 outputLayerInput += biasesOutput

10.b.3 outputLayerOutput = activationFunction(outputLayerInput)

// Compute the loss/error

10.c compute the loss between predicted output and target output using a suitable loss function (e.g., mean squared error, cross-entropy)

// Backpropagation

10.d compute the gradient of the loss with respect to the output layer activations:

10.d.1 outputGradient = derivative of the loss function with respect to the output layer activations

10.e compute the gradient of the loss with respect to the hidden layer activations:

10.e.1 hiddenGradient = dot product of outputGradient and transpose of weightsHiddenOutput

10.e.2 hiddenGradient *= derivative of activationFunction with respect to hiddenLayerInput

10.f compute the gradients of the weights and biases:

10.f.1 weightsHiddenOutputGradient = dot product of transpose of hiddenLayerOutput and outputGradient

10.f.2 biasesOutputGradient = sum of outputGradient along the rows (axis=0)

10.f.3 weightsInputHiddenGradient = dot product of transpose of input and hiddenGradient

10.f.4 biasesHiddenGradient = sum of hiddenGradient along the rows (axis=0)

// Update the weights and biases

10.g weightsHiddenOutput -= learningRate * weightsHiddenOutputGradient

10.h biasesOutput -= learningRate * biasesOutputGradient

10.i weightsInputHidden -= learningRate * weightsInputHiddenGradient

10.j biasesHidden -= learningRate * biasesHiddenGradient

// Step 5: Use the trained network for prediction

11 function predict(inputs):

11.a compute the activation of the hidden layer:

11.a.1 hiddenLayerInput = dot product of inputs and weightsInputHidden

11.a.2 hiddenLayerInput += biasesHidden

11.a.3 hiddenLayerOutput = activationFunction(hiddenLayerInput)

11.b compute the activation of the output layer:

11.b.1 outputLayerInput = dot product of hiddenLayerOutput and weightsHiddenOutput

11.b.2 outputLayerInput += biasesOutput

11.b.3 outputLayerOutput = activationFunction(outputLayerInput)

11.c return outputLayerOutput

contoh :

```cpp
#include <iostream>
#include <vector>
#include <cmath>

// Define the activation function
double sigmoid(double x) {
    return 1.0 / (1.0 + exp(-x));
}

// Define the derivative of the activation function
double sigmoidDerivative(double x) {
    double sigmoidX = sigmoid(x);
    return sigmoidX * (1.0 - sigmoidX);
}

class NeuralNetwork {
private:
    int numInputNeurons;
    int numHiddenNeurons;
    int numOutputNeurons;
    std::vector<std::vector<double>> hiddenWeights;
    std::vector<double> hiddenBiases;
    std::vector<std::vector<double>> outputWeights;
    std::vector<double> outputBiases;
    double learningRate;
    int numIterations;

public:
    NeuralNetwork(int numInputs, int numHidden, int numOutputs, double learningRate, int numIterations)
        : numInputNeurons(numInputs), numHiddenNeurons(numHidden), numOutputNeurons(numOutputs),
          learningRate(learningRate), numIterations(numIterations) {

        // Initialize weights and biases randomly
        hiddenWeights.resize(numInputNeurons, std::vector<double>(numHiddenNeurons));
        hiddenBiases.resize(numHiddenNeurons);
        outputWeights.resize(numHiddenNeurons, std::vector<double>(numOutputNeurons));
        outputBiases.resize(numOutputNeurons);

        // Randomly initialize the weights and biases
        initializeWeightsAndBiases();
    }

    void initializeWeightsAndBiases() {
        // Initialize the weights and biases randomly between -1 and 1
        for (int i = 0; i < numInputNeurons; ++i) {
            for (int j = 0; j < numHiddenNeurons; ++j) {
                hiddenWeights[i][j] = (2.0 * rand() / RAND_MAX) - 1.0;
            }
        }

        for (int i = 0; i < numHiddenNeurons; ++i) {
            hiddenBiases[i] = (2.0 * rand() / RAND_MAX) - 1.0;
        }

        for (int i = 0; i < numHiddenNeurons; ++i) {
            for (int j = 0; j < numOutputNeurons; ++j) {
                outputWeights[i][j] = (2.0 * rand() / RAND_MAX) - 1.0;
            }
        }

        for (int i = 0; i < numOutputNeurons; ++i) {
            outputBiases[i] = (2.0 * rand() / RAND_MAX) - 1.0;
        }
    }

```

```cpp
    std::vector<double> forwardPropagation(const std::vector<double>& inputs) {
        std::vector<double> hiddenActivations(numHiddenNeurons);
        std::vector<double> outputActivations(numOutputNeurons);

        // Compute the activations of the hidden layer
        for (int j = 0; j < numHiddenNeurons; ++j) {
            double weightedSum = 0.0;
            for (int i = 0; i < numInputNeurons; ++i) {
                weightedSum += inputs[i] * hiddenWeights[i][j];
            }
            weightedSum += hiddenBiases[j];
            hiddenActivations[j] = sigmoid(weightedSum);
        }

        // Compute the activations of the output layer
        for (int k = 0; k < numOutputNeurons; ++k) {
            double weightedSum = 0.0;
            for (int j = 0; j < numHiddenNeurons; ++j) {
                weightedSum += hiddenActivations[j] * outputWeights[j][k];
            }
            weightedSum += outputBiases[k];
            outputActivations[k] = sigmoid(weightedSum);
        }

        return outputActivations;
    }

    void backpropagation(const std::vector<double>& inputs, const std::vector<double>& targets) {
        std::vector<double> hiddenActivations(numHiddenNeurons);
        std::vector<double> outputActivations(numOutputNeurons);
        std::vector<double> outputDeltas(numOutputNeurons);
        std::vector<double> hiddenDeltas(numHiddenNeurons);

        // Compute the activations of the hidden layer
        for (int j = 0; j < numHiddenNeurons; ++j) {
            double weightedSum = 0.0;
            for (int i = 0; i < numInputNeurons; ++i) {
                weightedSum += inputs[i] * hiddenWeights[i][j];
            }
            weightedSum += hiddenBiases[j];
            hiddenActivations[j] = sigmoid(weightedSum);
        }

        // Compute the activations of the output layer
        for (int k = 0; k < numOutputNeurons; ++k) {
            double weightedSum = 0.0;
            for (int j = 0; j < numHiddenNeurons; ++j) {
                weightedSum += hiddenActivations[j] * outputWeights[j][k];
            }
            weightedSum += outputBiases[k];
            outputActivations[k] = sigmoid(weightedSum);
        }

        // Compute the deltas of the output layer
        for (int k = 0; k < numOutputNeurons; ++k) {
            double outputError = targets[k] - outputActivations[k];
            outputDeltas[k] = outputError * sigmoidDerivative(outputActivations[k]);
        }

```

```cpp
        // Compute the deltas of the hidden layer
        for (int j = 0; j < numHiddenNeurons; ++j) {
            double hiddenError = 0.0;
            for (int k = 0; k < numOutputNeurons; ++k) {
                hiddenError += outputDeltas[k] * outputWeights[j][k];
            }
            hiddenDeltas[j] = hiddenError * sigmoidDerivative(hiddenActivations[j]);
        }

        // Update the weights and biases of the output layer
        for (int j = 0; j < numHiddenNeurons; ++j) {
            for (int k = 0; k < numOutputNeurons; ++k) {
                outputWeights[j][k] += learningRate * hiddenActivations[j] * outputDeltas[k];
            }
        }

        for (int k = 0; k < numOutputNeurons; ++k) {
            outputBiases[k] += learningRate * outputDeltas[k];
        }

        // Update the weights and biases of the hidden layer
        for (int i = 0; i < numInputNeurons; ++i) {
            for (int j = 0; j < numHiddenNeurons; ++j) {
                hiddenWeights[i][j] += learningRate * inputs[i] * hiddenDeltas[j];
            }
        }

        for (int j = 0; j < numHiddenNeurons; ++j) {
            hiddenBiases[j] += learningRate * hiddenDeltas[j];
        }
    }

    void train(const std::vector<std::vector<double>>& trainingData) {
        for (int iteration = 0; iteration < numIterations; ++iteration) {
            for (const auto& data : trainingData) {
                std::vector<double> inputs(data.begin(), data.end() - numOutputNeurons);
                std::vector<double> targets(data.end() - numOutputNeurons, data.end());
                backpropagation(inputs, targets);
            }
        }
    }
};

int main() {
    // Example usage
    int numInputs = 2;
    int numHidden = 4;
    int numOutputs = 1;
    double learningRate = 0.1;
    int numIterations = 1000;

    NeuralNetwork network(numInputs, numHidden, numOutputs, learningRate, numIterations);

    // Training data: XOR problem
    std::vector<std::vector<double>> trainingData = {
        {0, 0, 0},
        {0, 1, 1},
        {1, 0, 1},
        {1, 1, 0}
    };

    network.train(trainingData);

    // Test the trained network
    std::vector<double> input = {1, 1};
    std::vector<double> output = network.forwardPropagation(input);

    std::cout << "Input: " << input[0] << ", " << input[1] << std::endl;
    std::cout << "Output: " << output[0] << std::endl;

    return 0;
}
```

(d) Particle Swarm Optimization

pseudocode :

Particle Swarm Optimization (PSO) Pseudocode

Step 1: Initialize the swarm

Initialize particles with random positions and velocities

Set the global best position and fitness to the initial position of a randomly selected particle

Step 2: Evaluate fitness

For each particle:

Evaluate the fitness of the current position

Step 3: Update particle's best position

For each particle:

If the current position has a better fitness than the particle's best position:

Update the particle's best position and fitness

Step 4: Update global best position

For each particle:

If the particle's best position has a better fitness than the global best position:

Update the global best position and fitness

Step 5: Update particle's velocity and position

For each particle:

Update the velocity using the previous velocity, the particle's best position, and the global best position

Update the position using the previous position and the new velocity

Step 6: Repeat steps 2-5 until a termination condition is met

Repeat steps 2-5 for a specified number of iterations or until a termination condition is met (e.g., reaching a maximum number of iterations, achieving a desired fitness level, etc.)

Step 7: Return the best solution

Return the global best position as the optimized solution

End of Particle Swarm Optimization (PSO) Pseudocode

contoh :

```cpp
#include <iostream>
#include <vector>
#include <random>
#include <cmath>

// Define the fitness function to be optimized
double fitnessFunction(const std::vector<double>& position) {
    // Example fitness function: Sphere function
    double sum = 0.0;
    for (double x : position) {
        sum += x * x;
    }
    return sum;
}

class Particle {
public:
    std::vector<double> position;
    std::vector<double> velocity;
    std::vector<double> bestPosition;
    double bestFitness;

    Particle(int numDimensions) {
        position.resize(numDimensions);
        velocity.resize(numDimensions);
        bestPosition.resize(numDimensions);
        bestFitness = std::numeric_limits<double>::max();
    }
};

class PSO {
private:
    int numParticles;
    int numDimensions;
    std::vector<Particle> particles;
    std::vector<double> globalBestPosition;
    double globalBestFitness;
    double inertiaWeight;
    double cognitiveWeight;
    double socialWeight;
    double velocityLimit;
    int numIterations;

public:
    PSO(int numParticles, int numDimensions, double inertiaWeight, double cognitiveWeight, double socialWeight,
        double velocityLimit, int numIterations)
        : numParticles(numParticles), numDimensions(numDimensions), inertiaWeight(inertiaWeight),
          cognitiveWeight(cognitiveWeight), socialWeight(socialWeight), velocityLimit(velocityLimit),
          numIterations(numIterations) {
        particles.resize(numParticles, Particle(numDimensions));
        globalBestPosition.resize(numDimensions);
        globalBestFitness = std::numeric_limits<double>::max();
    }

    void initializeParticles() {
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_real_distribution<double> dist(-1.0, 1.0);

        for (int i = 0; i < numParticles; ++i) {
            for (int j = 0; j < numDimensions; ++j) {
                particles[i].position[j] = dist(gen);
                particles[i].velocity[j] = dist(gen);
                particles[i].bestPosition[j] = particles[i].position[j];
            }

            double fitness = fitnessFunction(particles[i].position);
            particles[i].bestFitness = fitness;

```

```
            if (fitness < globalBestFitness) {
                globalBestFitness = fitness;
                globalBestPosition = particles[i].position;
            }
        }
    }

    void updateParticles() {
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_real_distribution<double> dist(0.0, 1.0);

        for (int i = 0; i < numParticles; ++i) {
            for (int j = 0; j < numDimensions; ++j) {
                // Update velocity
                double r1 = dist(gen);
                double r2 = dist(gen);
                particles[i].velocity[j] =
                    inertiaWeight * particles[i].velocity[j] +
                    cognitiveWeight * r1 * (particles[i].bestPosition[j] - particles[i].position[j]) +
                    socialWeight * r2 * (globalBestPosition[j] - particles[i].position[j]);

                // Apply velocity limits
                particles[i].velocity[j] = std::min(std::max(particles[i].velocity[j], -velocityLimit),
velocityLimit);

                // Update position
                particles[i].position[j] += particles[i].velocity[j];

                // Clamp position within a range if necessary
                // Example: particles[i].position[j] = std::min(std::max(particles[i].position[j], minValue),
maxValue);
            }

            double fitness = fitnessFunction(particles[i].position);

            if (fitness < particles[i].bestFitness) {
                particles[i].bestFitness = fitness;
                particles[i].bestPosition = particles[i].position;
            }

            if (fitness < globalBestFitness) {
                globalBestFitness = fitness;
                globalBestPosition = particles[i].position;
            }
        }
    }

    std::vector<double> optimize() {
        initializeParticles();

        for (int iteration = 0; iteration < numIterations; ++iteration) {
            updateParticles();
        }

        return globalBestPosition;
    }
};

```

```cpp
int main() {
    // Example usage
    int numParticles = 30;
    int numDimensions = 2;
    double inertiaWeight = 0.5;
    double cognitiveWeight = 1.0;
    double socialWeight = 1.0;
    double velocityLimit = 0.1;
    int numIterations = 100;

    PSO pso(numParticles, numDimensions, inertiaWeight, cognitiveWeight, socialWeight, velocityLimit,
numIterations);

    std::vector<double> bestPosition = pso.optimize();

    std::cout << "Optimized solution: ";
    for (double x : bestPosition) {
        std::cout << x << " ";
    }
    std::cout << std::endl;

    return 0;
}
```