SPRAWOZDANIE

SIEĆ KOHONENA – ALGORYTM WTA

Przedmiot: *Podstawy sztucznej inteligencji*

Autor: *Dávid Ali*

Kontakt: *s26567@st.pans.nysa.pl*

Data: *26. 01. 2024 r.*

Spis treści

Spis treści 1
Cel 2
Realizacja 3
Spis plików 15

Cel

Celem niniejszego projektu jest implementacja algorytmu uczenia sieci neuronowej z wykorzystaniem Sieci Kohonena z algorytmem uczenia WTA (Winner-Takes-All). Projekt skupia się na dwóch różnych zestawach danych: własnych danych związanych z systemami ekspertowymi (SE1 i SE2) oraz znanymi danymi dotyczącymi irysów/klasy wina. Implementacja ma obejmować proces uczenia na obu zbiorach danych, a także ocenę poprawności generowanych grup/kategorii przez sieć na danych testowych.

Wykorzystane technologie

W projekcie wykorzystano język programowania C++ oraz środowisko Visual Studio Community Edition 2022 do implementacji algorytmu uczenia sieci neuronowej. Wybór tego środowiska umożliwia efektywne debugowanie, organizację kodu oraz zapewnia wygodne narzędzia do rozwijania projektu.

Zagadnienia wstępne

Sieć Kohonena

Sieć Kohonena to rodzaj sztucznej sieci neuronowej, której głównym zadaniem jest przekształcanie wejściowych danych wielowymiarowych do niższowymiarowej przestrzeni, co pozwala na uzyskanie struktury danych i ich grupowanie. Sieci tego typu są często wykorzystywane do analizy danych, redukcji wymiarów oraz klasyfikacji. Sieć Kohonena składa się z warstwy wejściowej i warstwy neuronów zwanej mapą Kohonena. Każdy neuron w mapie reprezentuje pewną grupę lub kategorię danych. Podczas procesu uczenia, neurony konkurują ze sobą o to, który z nich najlepiej odpowiada na aktualne dane wejściowe. Zwycięzca, czyli neuron z najbliższym wzorcem, zostaje dostosowany, co umożliwia adaptację sieci do struktury danych.

Algorytm WTA

Istnieje kilka algorytmów uczenia sieci Kohonena, a jednym z nich jest algorytm WTA (Winner-Takes-All). Algorytm ten polega na tym, że tylko jeden neuron staje się zwycięzcą i jest aktualizowany podczas każdej iteracji uczenia. Proces ten prowadzi do powstawania jednej dominującej grupy w wynikowej mapie. Podczas uczenia sieci, dla każdego wektora danych wejściowych, obliczane jest odległość pomiędzy wektorem a wagami każdego neuronu. Następnie neuron, którego wagi są najbliższe wektorowi wejściowemu, zostaje uznany za zwycięzcę. Wagi zwycięskiego neuronu (oraz w przypadku innych algorytmów uczenia [np. WTM] wag sąsiednich neuronów) są następnie aktualizowane w kierunku wektora wejściowego. Algorytm uczenia WTA ma za zadanie osiągnąć konwergencję, gdzie neurony mapy Kohonena reprezentują różne grupy danych. Proces ten powtarzany jest dla każdego wektora uczącego, aż do uzyskania zadowalającej jakości klasyfikacji.

W dalszej części sprawozdania zostaną przedstawione szczegóły implementacji algorytmu sieci Kohonena z wykorzystaniem algorytmu uczenia WTA w języku C++ z użyciem środowiska Visual Studio Community Edition 2022.

Realizacja

```
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <cfloat>
#include <windows.h>
#pragma warning(disable : 4996)
#pragma warning(disable : 6031)
#pragma warning(disable : 4244)

//#define __VERBOSE
#define SALT_WHILE_TRAINING
#define SALT_WHILE_TRAINING_CHANCE 0.01
#define SALT_WHILE_TRAINING_MAX_CHANGE 0.005
//#define SALT_WHILE_TESTING
#define SALT_WHILE_TESTING_CHANCE 1
#define SALT_WHILE_TESTING_MAX_CHANGE 0.0001
//#define TRAIN_EVEN_THEN_TEST_ODD

#define USE_SIMPLIFIED_WTM_ALGORYTHM_IN_FIRST_N_EPOCHS 3
void fprintVector(FILE* fp, double* v, size_t n) {
 fprintf(fp, "{");
 for (size_t i = 0; i < n; i++) fprintf(fp, "%4.0lf", v[i] * 100);
 fprintf(fp, " }");
}

void sprintVector(char* buffer, double* v, size_t n) {
 sprintf(buffer, "{");
 for (size_t i = 0; i < n; i++) sprintf(buffer + strlen(buffer), "%4.0lf",
v[i] * 100);
 sprintf(buffer + strlen(buffer), " }");
}

void fprintWeights(FILE* fp, double** weights, size_t h, size_t w, size_t
epoch) {
 fprintf(fp, "WEIGHTS AFTER %Id EPOCHS:\n", epoch);

 for (size_t y = 0; y < h; y++) {
   fprintf(fp, "Neuron %*Iu: ", -((int)log10(h) + 1), y);
   fprintVector(fp, weights[y], w);
   fprintf(fp, "\n");
 }
}
#ifdef __VERBOSE
void fprintWeights2(FILE* fp, double** weights, size_t h, size_t w, size_t
trainIndex, double learningRate, size_t epochIndex) {
 fprintf(fp, "Epoch index: %5Id | Train index: %5Id | Learning rate: %lf
\n", epochIndex, trainIndex, learningRate);

 for (size_t y = 0; y < h; y++) {
   fprintf(fp, "Neuron %*Iu: ", -((int)log10(h) + 1), y);
   fprintVector(fp, weights[y], w);
   fprintf(fp, "\n");
 }
}
```

```
#endif
void fprintInfo(FILE* fp, unsigned int seed, char* inFileName, size_t
inputNum, size_t inputSize, size_t outputSize, size_t numEpochs, double
learningRate) {
 fprintf(fp, "      Random seed: 0x%08x\n", seed);
 fprintf(fp, " Number of inputs: %Id\n", inputNum);
 fprintf(fp, "   Size of inputs: %Id\n", inputSize);
 fprintf(fp, "Number of neurons: %Id\n", outputSize);
 fprintf(fp, " Number of epochs: %Id\n", numEpochs);
 fprintf(fp, "    Learning rate: %lf\n", learningRate);
}
void normalizeColumns(double** m, size_t h, size_t w) {
 for (size_t x = 0; x < w; x++) {
   double min = m[0][x];
   double max = m[0][x];
   for (size_t y = 1; y < h; y++) {
     if (min > m[y][x]) min = m[y][x];
     if (max < m[y][x]) max = m[y][x];
   }
   for (size_t y = 0; y < h; y++) {
     m[y][x] = (m[y][x] - min) / (max - min);
   }
 }
}
void normalizeRow(double* v, size_t w) {
 double sum = 0;
 for (size_t x = 0; x < w; x++) {
   sum += v[x] * v[x];
 }
 double sq = sqrt(sum);
 for (size_t x = 0; x < w; x++) {
   v[x] /= sq;
 }
}
void normalizeRows(double** m, size_t h, size_t w) {
 for (size_t y = 0; y < h; y++) {
   normalizeRow(m[y], w);
 }
}

double distance(double* v1, double* v2, size_t w) {
 double sum = 0;
 for (size_t x = 0; x < w; x++) {
   sum += pow(v1[x] - v2[x], 2);
 }
 return sqrt(sum);
}

inline size_t randomI(size_t upperExclusiveLimit) {
 return rand() % upperExclusiveLimit;
}

inline double randomD() {
 return (double)rand() / RAND_MAX;
}

inline double randomDaround0(double plusMinus) {
```

```
 return ((double)rand() / RAND_MAX - 0.5) * 2 * plusMinus;
}

void initWeights(double*** weights, size_t h, size_t w) {
 *weights = (double**)malloc(h * sizeof(double*));
 if (*weights == nullptr) exit(1);
 for (size_t y = 0; y < h; y++) {
   (*weights)[y] = (double*)malloc(w * sizeof(double));
   if ((*weights)[y] == nullptr) exit(1);
   for (size_t x = 0; x < w; x++) {
     (*weights)[y][x] = randomD();
   }
 }
 normalizeRows(*weights, h, w);
}

void initWeights2(double*** weights, double** inputs, size_t inputNum,
size_t inputSize, size_t outputSize) {
 *weights = (double**)malloc(outputSize * sizeof(double*));
 if (*weights == nullptr) exit(1);
 for (size_t y = 0; y < outputSize; y++) {
   (*weights)[y] = (double*)malloc(inputSize * sizeof(double));
   if ((*weights)[y] == nullptr) exit(1);
   for (size_t x = 0; x < inputSize; x++) {
     (*weights)[y][x] = inputs[randomI(inputNum)][x];
   }
 }
 normalizeRows(*weights, outputSize, inputSize);
}

void freeDoubleMatrix(double** m, size_t h) {
 for (size_t y = 0; y < h; y++) {
   free(m[y]);
 }
 free(m);
}
void freeCharMatrix(char** m, size_t h) {
 for (size_t y = 0; y < h; y++) {
   free(m[y]);
 }
 free(m);
}

size_t findWinner(double** weights, double* in, size_t h, size_t w) {
 size_t winnerIndex = 0;
 double minDistance = DBL_MAX;
 for (size_t y = 0; y < h; y++) {
   double dist = distance(weights[y], in, w);
   if (dist < minDistance) {
     minDistance = dist;
     winnerIndex = y;
   }
 }
 return winnerIndex;
}
```

```
void moveAtowardsB(double* A, double* B, double rate, size_t w, size_t
winnerIndex = 0) {
 for (size_t x = 0; x < w; x++) {
   A[x] += rate * (B[x] - A[x]);
 }
}

void salt(double* in, double** copyWithSalt, size_t size, double chance =
0, double maxChange = 0) {
 (*copyWithSalt) = (double*)malloc(size * sizeof(double));
 if ((*copyWithSalt) == nullptr) exit(0);
 memcpy((*copyWithSalt), in, size * sizeof(double));
 for (size_t i = 0; i < size; i++) {
   if (randomD() >= chance) continue;
   (*copyWithSalt)[i] += randomDaround0(maxChange);
   if ((*copyWithSalt)[i] < 0) (*copyWithSalt)[i] = 0;
   if ((*copyWithSalt)[i] > 1) (*copyWithSalt)[i] = 1;
 }
}

void train(double** weights, double* in, double rate, size_t h, size_t w,
int useSimplifiedWTMAlgorytm = 0) {
#ifdef SALT_WHILE_TRAINING
 double* inp;
 salt(in, &inp, w, SALT_WHILE_TRAINING_CHANCE,
SALT_WHILE_TRAINING_MAX_CHANGE);
 size_t winnerIndex = findWinner(weights, inp, h, w);
 moveAtowardsB(weights[winnerIndex], inp, rate, w);
 free(inp);
#else
 size_t winnerIndex = findWinner(weights, in, h, w);
 moveAtowardsB(weights[winnerIndex], in, rate, w);
#endif
 normalizeRow(weights[winnerIndex], w);
 if (useSimplifiedWTMAlgorytm) {
   double* lowerNeighbor = weights[(winnerIndex + h - 1) % h];
   double* lowerlowerNeighbor = weights[(winnerIndex + h - 2) % h];
   double* upperNeighbor = weights[(winnerIndex + 1) % h];
   double* upperupperNeighbor = weights[(winnerIndex + 2) % h];

   moveAtowardsB(lowerNeighbor, weights[winnerIndex], rate / 2, w);
   normalizeRow(lowerNeighbor, w);
   moveAtowardsB(upperNeighbor, weights[winnerIndex], rate / 2, w);
   normalizeRow(upperNeighbor, w);
   moveAtowardsB(lowerlowerNeighbor, lowerNeighbor, rate / 5, w);
   normalizeRow(lowerlowerNeighbor, w);
   moveAtowardsB(upperupperNeighbor, upperNeighbor, rate / 5, w);
   normalizeRow(upperupperNeighbor, w);
 }
}

void initInputs(const char* inFileName, double*** inputs, size_t*
inputNum, size_t* inputSize, size_t* outputSize, size_t* numEpochs,
double* learningRate) {
 FILE* fp = fopen(inFileName, "r");
 if (fp == nullptr) {
   fprintf(stderr, "Error opening file \"%s\". Exiting.", inFileName);
```

```
    exit(1);
  }
  fscanf(fp, "%Id", inputNum);
  fscanf(fp, "%Id", inputSize);
  fscanf(fp, "%Id", outputSize);
  fscanf(fp, "%Id", numEpochs);
  fscanf(fp, "%lf", learningRate);

  *inputs = (double**)malloc(*inputNum * sizeof(double*));
  if (*inputs == nullptr) exit(1);

  for (size_t y = 0; y < *inputNum; y++) {
    (*inputs)[y] = (double*)malloc(*inputSize * sizeof(double));
    if ((*inputs)[y] == nullptr) exit(1);
    for (size_t x = 0; x < *inputSize; x++) {
      fscanf(fp, "%lf", &((*inputs)[y][x]));
    }
  }
  fclose(fp);
}

#ifdef __VERBOSE
void gotoXY(size_t x, size_t y) {
  COORD destCoord;
  destCoord.X = x;
  destCoord.Y = y;
  SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), destCoord);
}
void rememberXY(SHORT* x, SHORT* y) {
  CONSOLE_SCREEN_BUFFER_INFO info;
  GetConsoleScreenBufferInfo(GetStdHandle(STD_OUTPUT_HANDLE), &info);
  *x = info.dwCursorPosition.X;
  *y = info.dwCursorPosition.Y;
}
void clearConsoleRow() {
  SHORT cursorX;
  SHORT cursorY;
  rememberXY(&cursorX, &cursorY);
  printf("
");
  gotoXY(cursorX, cursorY);
}
#endif
void test(char*** output, double** weights, double** inputs, size_t
inputNum, size_t inputSize, size_t outputSize) {
  (*output) = (char**)malloc(outputSize * sizeof(char*));
  if ((*output) == nullptr) exit(1);
  size_t neuronNumberWidth = (size_t)(log10(outputSize)) + 1;
  size_t numberWidth = (size_t)(log10(inputNum)) + 2;
  for (size_t i = 0; i < outputSize; i++) {
    (*output)[i] = (char*)malloc((10 + neuronNumberWidth + inputNum *
numberWidth) * sizeof(char));
    if ((*output)[i] == nullptr) exit(1);
    sprintf((*output)[i], "Neuron %*Id: ", -(int)neuronNumberWidth, i);
  }
#ifdef TRAIN_EVEN_THEN_TEST_ODD
  for (int testIndex = 1; testIndex < inputNum; testIndex += 2) {
```

```
#else
 for (int testIndex = 0; testIndex < inputNum; testIndex += 1) {
#endif
#ifdef SALT_WHILE_TESTING
   double* inp;
   salt(inputs[testIndex], &inp, inputSize, SALT_WHILE_TESTING_CHANCE,
SALT_WHILE_TESTING_MAX_CHANGE);
   size_t winner = findWinner(weights, inp, outputSize, inputSize);
   free(inp);
#else
   size_t winner = findWinner(weights, inputs[testIndex], outputSize,
inputSize);
#endif
   sprintf((*output)[winner] + strlen((*output)[winner]), "%*d",
(int)numberWidth, testIndex);
 }
}

void doEpoch(double** weights, double** inputs, double rate, size_t
inputNum, size_t outputSize, size_t inputSize, size_t epochIndex, size_t
epochNum) {
 size_t* indices = (size_t*)malloc(inputNum * sizeof(size_t));
 if (indices == nullptr) exit(1);
 for (size_t i = 0; i < inputNum; i++) {
   indices[i] = i;
 }
 size_t temp, randomIndex;
 for (size_t i = 0; i < inputNum; i++) {
   randomIndex = (size_t)randomI(inputNum);
   temp = indices[i];
   indices[i] = indices[randomIndex];
   indices[randomIndex] = temp;
 }

 for (int trainIndex = 0; trainIndex < inputNum; trainIndex++) {
#ifdef TRAIN_EVEN_THEN_TEST_ODD
   if (indices[trainIndex] % 2 == 0)continue;
#endif

   double learningRate = rate * (1.0 - ((double)trainIndex / inputNum));
   train(weights, inputs[indices[trainIndex]], learningRate, outputSize,
inputSize, epochIndex < USE_SIMPLIFIED_WTM_ALGORYTHM_IN_FIRST_N_EPOCHS);

#ifdef __VERBOSE
   SHORT cursorX;
   SHORT cursorY;
   rememberXY(&cursorX, &cursorY);
   fprintWeights2(stdout, weights, outputSize, inputSize, trainIndex,
learningRate, epochIndex);
   gotoXY(cursorX, cursorY);
#endif
 }
 free(indices);
}
int main(int argc, char* argv[]) {
 unsigned int seed;
 double** inputs;
```

```
  size_t inputNum;
  size_t inputSize;
  size_t outputSize;
  size_t numEpochs;
  double learningRate;
  double** weights;
  char** output;
  char inFileName[128];
  char outFileName[128];

  seed = 0x416C6944;
  //seed = time(0);
  srand(seed);
  if (argc > 1) {
     sprintf(inFileName, "%s.in", argv[1]);
     sprintf(outFileName, "%s.out", argv[1]);
  }
  else {
     char buf[100];
     printf("Podaj nazwe pliku wejsciowego (bez rozszerzenia, nie moze
zawierac spacji i znakow specjalnych, nie moze byc dluzsza niz 100
znakow): \n");
     printf("Nazwa pliku: "); scanf("%s", buf);
     sprintf(inFileName, "%s.in", buf);
     sprintf(outFileName, "%s.out", buf);
  }
  initInputs(inFileName, &inputs, &inputNum, &inputSize, &outputSize,
&numEpochs, &learningRate);
  normalizeColumns(inputs, inputNum, inputSize);
  normalizeRows(inputs, inputNum, inputSize);
  initWeights2(&weights, inputs, inputNum, inputSize, outputSize);

  FILE* fp = fopen(outFileName, "w");
  if (fp == nullptr) exit(1);
  fprintInfo(stdout, seed, inFileName, inputNum, inputSize, outputSize,
numEpochs, learningRate);
  fprintInfo(fp, seed, inFileName, inputNum, inputSize, outputSize,
numEpochs, learningRate);


  for (int epoch = 0; epoch < numEpochs; epoch++) {
     doEpoch(weights, inputs, learningRate, inputNum, outputSize, inputSize,
epoch, numEpochs);
  }

#ifdef __VERBOSE
  clearConsoleRow();
#endif
  fprintWeights(stdout, weights, outputSize, inputSize, numEpochs);
  fprintWeights(fp, weights, outputSize, inputSize, numEpochs);
  test(&output, weights, inputs, inputNum, inputSize, outputSize);

  fprintf(stdout, "TEST RESULTS:\n");
  fprintf(fp, "TEST RESULTS:\n");

  for (size_t i = 0; i < outputSize; i++) {
     fprintf(stdout, "%s\n", output[i]);
```

```
    fprintf(fp, "%s\n", output[i]);
  }
  fclose(fp);

  freeCharMatrix(output, outputSize);
  freeDoubleMatrix(weights, outputSize);
  freeDoubleMatrix(inputs, inputNum);

 }
```

Powyższy kod implementuje prostą wersję algorytmu map samoorganizujących Kohonena. SOM to algorytm uczenia nienadzorowanego używany do grupowania i wizualizacji danych o wysokiej wymiarowości. Poniżej przedstawiam kilka kluczowych punktów dotyczących tego kodu:

Kluczowe punkty kodu

Nagłówki

Kod zawiera różne standardowe nagłówki C++ (<cmath>, <cstdio>, <cstdlib>, <ctime>, <cfloat>) oraz nagłówek specyficzny dla systemu Windows - <windows.h>, używany do funkcji związanych z konsolą.

Dyrektywy preprocesora i definicje

Istnieje wiele dyrektyw preprocesora (#define), które kontrolują zachowanie kodu, takie jak drukowanie szczegółów, opcje dodawania „soli” (szumu) podczas treningu i testowania oraz używanie uproszczonego algorytmu w pierwszych epokach.

Definicje funkcji

Zdefiniowano kilka funkcji, w tym funkcje do drukowania wektorów i wag, normalizacji macierzy, obliczania odległości, inicjalizacji wag oraz główne funkcje uczenia i testowania.

Algorytm uczący

Funkcja ucząca (train) aktualizuje wagi neuronów na podstawie danych wejściowych i neuronu-zwycięzcy (tego, którego wektor wag jest najbliższy wektorowi wejściowemu).

Generowanie liczb losowych

Funkcje generujące liczby losowe (randomI i randomD) są używane do inicjalizacji wag oraz wprowadzania szumu podczas treningu i testowania.

Obsługa pliku wejściowego

Kod czyta parametry wejściowe i dane z pliku, który jest określony jako argument wiersza poleceń lub wprowadzany przez użytkownika podczas działania programu.

Testowanie

Funkcja test ocenia wytrenowany SOM na danych wejściowych i generuje ciągi wyjściowe wskazujące, które wejścia są skojarzone z każdym neuronem.

Pętla Epok

Główna funkcja zawiera pętlę, która przechodzi przez epoki, wywołując funkcję uczącą (doEpoch) w każdej iteracji.

Wyjście
Ostateczne wagi i wyniki testów są drukowane na konsolę i zapisywane do pliku wyjściowego.

Interakcja z konsolą (opcjonalna)
Istnieją fragmenty kodu związane z interakcją z konsolą, takie jak pozycjonowanie kursora i czyszczenie wierszy konsoli. Są one używane do drukowania szczegółów i celów debugowania.

Zarządzanie pamięcią
Pamięć jest dynamicznie alokowana dla różnych macierzy i tablic, a na końcu programu zarezerwowana pamięć się zwalnia.

Algorytm
W tym konkretnym kodzie, używana jest jedna z podstawowych form algorytmu SOM, znana jako algorytm Winner-Takes-All (WTA). Oto ogólne kroki działania tego algorytmu:

Inicjalizacja wag
Na początku programu wagi neuronów są inicjalizowane losowo.

Normalizacja danych wejściowych
Dane wejściowe (wektory) są zazwyczaj normalizowane, aby uniezależnić algorytm od skali wartości.

Trening
Algorytm przechodzi przez kilka epok treningowych. W każdej epoce, dla każdego przykładu treningowego:

- Obliczana jest odległość między wektorem wejściowym a wagami wszystkich neuronów.
- Wybierany jest neuron-zwycięzca (ten, którego waga jest najbliższa wektorowi wejściowemu).
- Wagi tego zwycięzcy[1] są aktualizowane w kierunku wektora wejściowego.

Normalizacja wag
Po aktualizacji wag w każdej iteracji treningowej, wagi neuronów są normalizowane, aby zachować ich jednostkową długość (normalizacja wektorów wag).

Testowanie
Po zakończeniu treningu można przeprowadzić testowanie, gdzie dla każdego wektora testowego obliczany jest neuron-zwycięzca, czyli ten, którego waga jest najbliższa wektorowi testowemu.

Zmęczenie neuronów
Aby zapobiec zmęczenie neuronów istnieje opcja zastosowania uproszczonej wersji algorytmu WTM w pierwszych kilku epokach. Używając tej opcji można było zaobserwować większej skuteczności, dzięki temu że więcej neuronów się aktywuje.

Drugi pomysł na zapobieganie zmęczenia neuronów to dodawanie niewielkiego szumu do danych wejściowych podczas uczenia (i/lub testowania) sieci.

[1] W algorytmie Winner-Takes-All (WTA) wagi neuronów są aktualizowane tylko dla zwycięzcy, co oznacza, że tylko najbliższy neuron ma zmienione wagi w danym kroku treningowym. Kod zawiera także pewne modyfikację: uproszczony algorytm WTM (Winner-Takes-Most) polega na tym, że nie tylko zwycięzca zostaje „przyciągany” do wektora uczącego ale także neurony sąsiadujące w mniejszym stopniu.

Tryb __VERBOSE

Jeśli usuniemy „//” przed linii #define __VERBOSE, oznacza to, że ta dyrektywa preprocesora stanie się aktywna, co skutkuje włączeniem trybu szczegółowego w kodzie. Tryb ten służy do wypisywania dodatkowych informacji diagnostycznych na standardowe wyjście (konsolę). Wykorzystanie tego trybu jest związane z funkcjami fprintWeights2 i związanymi z nimi fragmentami kodu.

W tym trybie pojawiają się dodatkowe informacje diagnostyczne podczas treningu. W tych dodatkowych informacjach znajdują się szczegóły dotyczące wag neuronów, aktualnych epok treningowych, indeksów próbek treningowych oraz współczynnika uczenia.

Uwaga, ten tryb jest bardzo wolny w stosunku do trybu normalnego (poprzez wyświetlanie informacji).

Plik wejściowy

Struktura pliku wejściowego dla tego programu jest opisana w kodzie. Poniżej przedstawiam przykład, jak może wyglądać plik wejściowy:

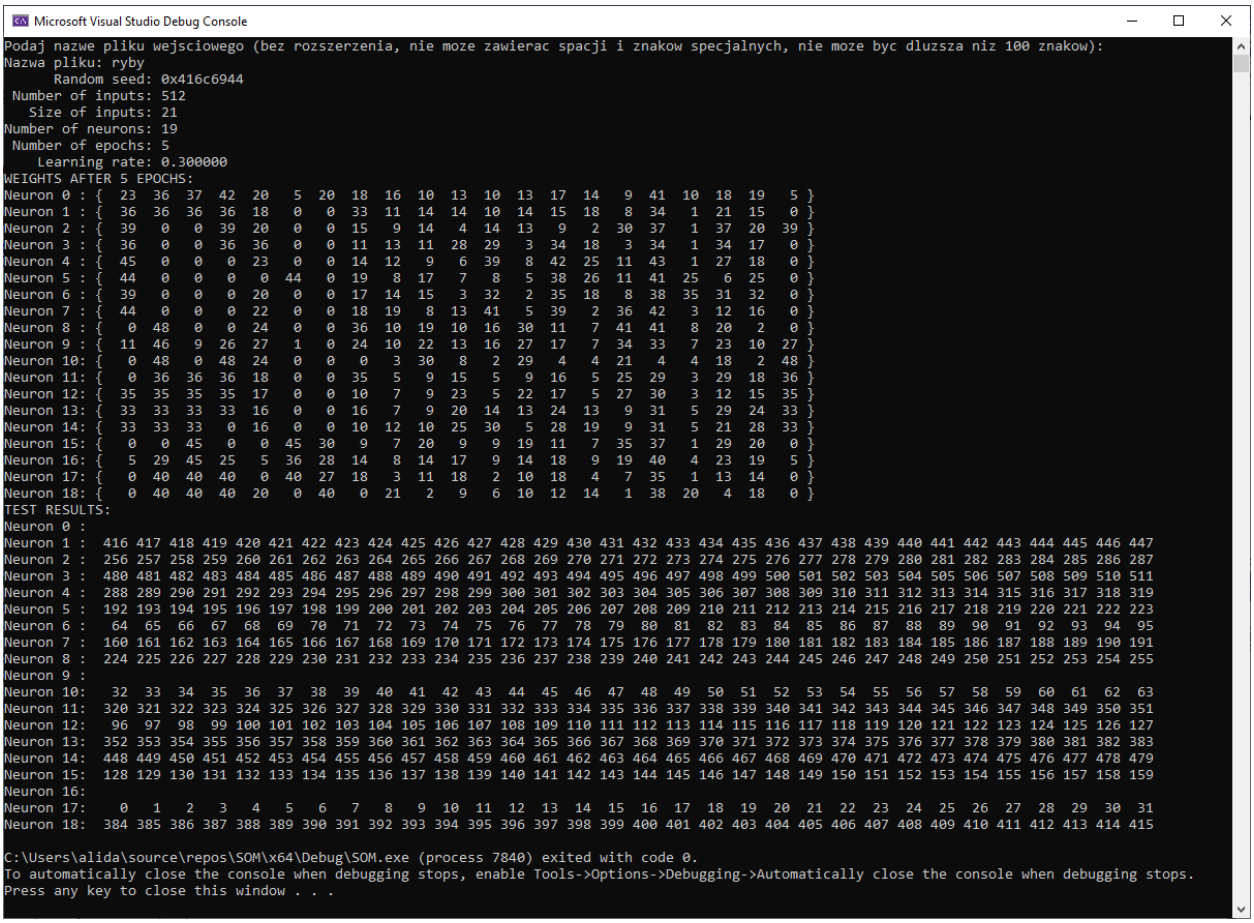
```
4       3       2       1       0.1
0.2     0.4     0.6
0.8     0.1     0.3
0.5     0.7     0.9
0.2     0.5     0.7
```

Gdzie:

- Pierwsza liczba (4) oznacza liczbę wektorów wejściowych.
- Druga liczba (3) to rozmiar wektora wejściowego.
- Trzecia liczba (2) to liczba neuronów.
- Czwarta liczba (1) to liczba epok treningu.
- Piąta liczba (0.1) to (początkowy) współczynnik uczenia.

Przykłady działania

Ryby

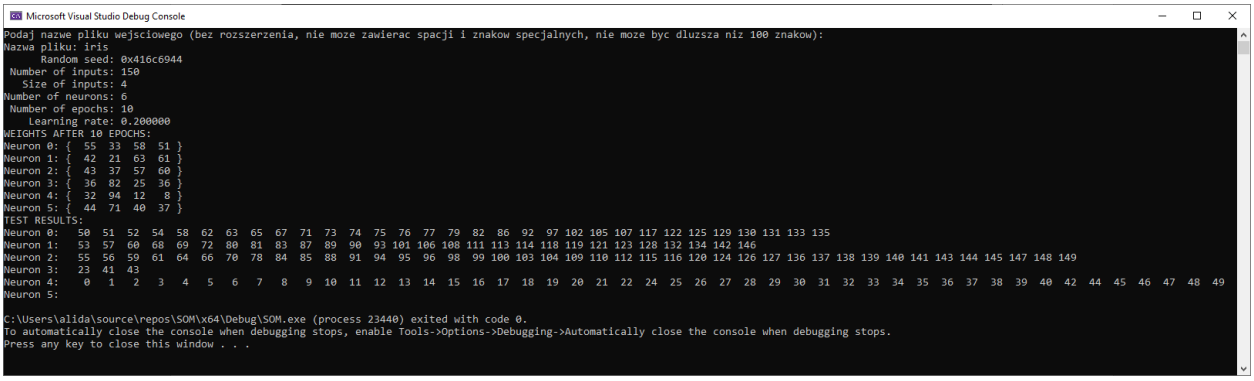


Plik ryby.in zawiera dane związane z 512 osobników 16 różnych gatunków ryb (każdy gatunek po 32 osobników). Każdy osobnik jest opisane 21 atrybutami. Zostaje tworzony 19 neuronów, proces uczenia trwa 5 epoki z początkującymi współczynnikami uczenia o wartości 0.3. Dane uczące są posortowane według gatunków.

Podczas działania programu oprócz wyświetlania wyników na konsoli także powstał plik ryby.out.

W tym analizując wyniki widzimy, że każdy neuron (który w ogóle się aktywował podczas testów) się aktywował 32 razy, i sieć 100%-ową skutecznością sklasyfikował każdego osobnika.

Iris



Na iris.in (źródło: https://en.wikipedia.org/wiki/Iris_flower_data_set) – z ustawieniami widocznymi na rzucie ekranu – wyniki są następujące:

	I. setosa [szt]	I. versicolor [szt]	I. virginica [szt]
Neuron 0	-	20	11
Neuron 1	-	13	15
Neuron 2	-	17	24
Neuron 3	3	-	-
Neuron 4	47	-	-
Neuron 5	-	-	-

Czyli w miarę skutecznie udało się rozpoznawać gatunki I. setosa, ale ogólnie sieć ma problemy z rozpoznawaniem gatunków I. versicolor oraz I. virginica. Aby doprowadzić do lepszych wyników trzeba będzie eksperymentować z różnymi ustawieniami.

Opinia

Bardzo ciekawy projekt.

Spis plików

Nazwa pliku	**Opis**
SOM.cpp	Źródło C++ programu
DAVID ALI - SOM - Sprawozdanie.docx	To sprawozdanie w formacie docx.
DAVID ALI - SOM - Sprawozdanie.pdf	To sprawozdanie w formacie pdf.
iris.in	Dane wejściowe „iris”
iris.out	Dane wyjściowe „iris”
ryby.in	Dane wejściowe „ryby”
ryby.out	Dane wyjściowe „ryby”