Sieci Neuronowe

K.Oratowski Sz.Kuliński J.Jurczak

13. Februar 2020

Sieci neuronowe są modelami procesu poznawczego mózgu. Mózg ma architekturę wieloprocesorową, która jest ze sobą ściśle powiązana. Sieci neuronowe mają niewiarygodny potencjał do rozwijania różnych rodzajów problemów, które są rozwiązywane przez komputery. Neuron jest podstawowym procesorem w sieciach neuronowych. Każdy neuron ma jedno wyjście, które jest generalnie związane ze stanem aktywacji neuronu - i które może rozwinąć się w kilka innych neuronów. Każdy neuron otrzymuje przez te połączenia kilka sygnałów wejściowych, zwanych synapsami. Dane wejściowe są aktywacjami przychodzących neuronów pomnożonymi przez masy synaps. Aktywacja neuronu jest obliczana przez zastosowanie funkcji progowej dla tego produktu. Ta funkcja progowa jest ogólnie jakąś formą funkcji nieliniowej.

Podstawowy sztuczny neuron można modelować jako wielo wejściowe urządzenie nieliniowe z ważonymi połączeniami w_{ij}, zwanymi również ciężarami synaptycznymi. Ciało komórki jest reprezentowane przez nieliniową funkcję ograniczającą lub progową funkcje f. Najprostszy model sztucznego neuronu sumuje n ważonych danych wejściowych i przekazuje wynik przez nieliniowość zgodnie z równaniem

$$y_j = f\left(\sum_{i=1}^{n} w_{ji}x_i - \theta_j\right)$$

gdzie f jest funkcją progową, zwaną także funkcją aktywacyjną, Θ_j; (Θ_j ϵ R) jest zewnętrznym progiem, zwanym także przesunięciem lub odchyleniem, w_{ij} to waga lub siły synaptyczne, x_i są wejściami (i = 1, 2, ..., n), n jest liczbą wejść y_j; reprezentuje wynik. Funkcja aktywacji jest również nazywana nieliniową charakterystyką przenoszenia lub funkcją zgniatania. Funkcja aktywacji f jest funkcją monotonicznie rosnącą.

A threshold value θ_j may be introduced by employing an additional input x_0 equal to $+1$ and the corresponding weight w_{j0} equal to minus the threshold value. Thus we can write

$$y_j = f\left(\sum_{i=0}^{n} w_{ji} x_i\right)$$

where

$$w_{j0} = -\theta_j, \qquad x_0 = 1.$$

The basic artificial neuron is characterized by its nonlinearity and the threshold θ_j. The *McCulloch-Pitts model* of the neuron used only the binary (hard-limiting) function (step function or *Heaviside function*), i.e.,

$$H(x) := \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 . \end{cases}$$

In this model a weighted sum of all inputs is compared with a threshold θ_j. If this sum exceeds the threshold, the neuron output is set to 1, otherwise to 0. For bipolar representation we can use the *sign function*

$$\text{sign}(x) := \begin{cases} 1 \text{ if } x > 0 \\ 0 \text{ if } x = 0 \\ -1 \text{ if } x < 0 . \end{cases}$$

The function

$$f_\lambda(x) = \frac{1}{1+e^{-\lambda x}}, \quad \lambda > 0$$

satisfies the nonlinear differential equation

$$\frac{df_\lambda}{dx} = \lambda f_\lambda (1 - f_\lambda)$$

and the function

$$g_\lambda(x) = \tanh(\lambda x), \quad \lambda > 0$$

satisfies the nonlinear differential equation

$$\frac{dg_\lambda}{dx} = \lambda(1 - g_\lambda^2)$$

This is important for the backpropagation algorithm.

Model powyższej funkcji sigmoidalnej można zbudować przy użyciu tradycyjnych elektronicznych komponentów układu elektrycznego . Wzmacniacz napięcia symuluje ciało komórki, przewody zastępują strukturę wejściową i strukturę wyjściową, a rezystory zmienne modelują masy synaptyczne (synapsy). Napięcie wyjściowe wzmacniacza y_j zastępuje zmienną częstość tętna prawdziwego neuronu. Funkcja aktywacji sigmoidalnej jest naturalnie zapewniona przez charakterystykę nasycenia wzmacniacza. Sygnały napięcia wejściowego x_i dostarczają prąd do dendrytów drutowych proporcjonalnie do sumy iloczynów napięć wejściowych i odpowiednich przewodności. Stosując prawo prądu Kirchhoffa na węźle wejściowym wzmacniacza uzyskujemy wyrażenie

$$y_j = f(u_j), \qquad u_j := \frac{\sum_{i=0}^{n} G_{ji} x_i}{\sum_{i=0}^{n} G_{ji}}$$

```cpp
// threshold.cpp

#include <iostream>
#include <cmath>   // for exp, tanh
using namespace std;

int H(double* w,double* x,int n)
{
  double sum = 0.0;
  for(int i=0;i<=n;i++) { sum += w[i]*x[i]; }
  if(sum >= 0.0) return 1;
  else return 0;
}

int sign(double* w,double* x,int n)
{
  double sum = 0.0;
  for(int i=0;i<=n;i++) { sum += w[i]*x[i]; }
  if(sum >= 0.0) return 1;
  else return 0;
}

double unipolar(double* w,double* x,int n)
{
  double lambda = 1.0;
  double sum = 0.0;
  for(int i=0;i<=n;i++) { sum += w[i]*x[i]; }
  return 1.0/(1.0 + exp(-lambda*sum));
}

double bipolar(double* w,double* x,int n)
{
  double lambda = 1.0;
  double sum = 0.0;
  for(int i=0;i<=n;i++) { sum += w[i]*x[i]; }
  return tanh(lambda*sum);
}
```

```
    w[1] = 0.7; w[2] = -1.1; w[3] = 4.5; w[4] = 1.5;

    // memory allocation for input vector x
    double* x = new double[n];
    x[0] = 1.0;  // bias
    x[1] = 0.7; x[2] = 1.2; x[3] = 1.5; x[4] = -4.5;

    int r1 = H(w,x,n);
    cout << "r1 = " << r1 << endl;
    int r2 = sign(w,x,n);
    cout << "r2 = " << r2 << endl;
    double r3 = unipolar(w,x,n);
    cout << "r3 = " << r3 << endl;
    double r4 = bipolar(w,x,n);
    cout << "r4 = " << r4 << endl;
    delete[] w; delete[] x;
    return 0;
}
```

Hopfiled model

Problem sformułowano w następujący sposób. Przechowuj zestaw wzorców p

$$\mathbf{x}_k, \qquad k = 0, 1, 2, \ldots, p-1$$

w taki sposób, aby po przedstawieniu nowego wzorca s sieć zareagowała, wytwarzając którykolwiek z wzorców składowania, który najbardziej przypomina s. Binarna sieć Hopfielda może być wykorzystana do ustalenia, czy wektor wejściowy (wzór) jest znanym wektorem, czy nieznanym wektorem dalsze zmiany stanu do raportu. W bardziej rozbudowanej wersji sieci Hopfielda mechanizm wyzwalania neuronów (tj. Włączanie lub wyłączanie) jest zgodny z prawem probabilistycznym. W takiej sytuacji neurony nazywamy neuronami stochastycznymi. . Sieć rozpoznaje znany wektor, wytwarzając wzór aktywacji na jednostkach sieci, który jest taki sam jak wektor przechowywany w sieci. Może się również zdarzyć, że wektor wejściowy zbiega się z wektorem aktywacyjnym, który nie jest żadnym z zapisanych wzorów. Taki wzór nazywa się fałszywym stanem stabilnym. Sieć Hopfield jest siecią rekurencyjną, która ucieleśnia głęboką fizyczną zasadę, a mianowicie przechowywanie informacji w dynamicznie stabilnej konfiguracji. Pomysł Hopfielda polegał na zlokalizowaniu wzoru skrzynki, który ma być przechowywany na dnie „doliny” krajobrazu energetycznego, a następnie umożliwieniu dynamicznej procedury minimalizacji energii sieci w taki sposób, że dolina staje się zagłębiem przyciągania . Standardowa wersja sieci Hopfield z czasem dyskretnym wykorzystuje model McCullocha-Pittsa dla neuronów. Wyszukiwanie informacji przechowywanych w sieci odbywa się za pomocą dynamicznej procedury aktualizacji stanu neuronu wybranego spośród tych, które chcą się zmienić, przy czym ten konkretny neuron jest wybierany losowo i jeden naraz. Ta asynchroniczna procedura dynamiczna jest powtarzana do momentu braku dalszej zmiany stanu. W bardziej rozbudowanej wersji sieci Hopfielda mechanizm wyzwalania neuronów jest zgodny z prawem probabilistycznym. W takiej sytuacji neurony nazywamy neuronami stochastycznymi.

Definition. Let $\mathbf{x}$ and $\mathbf{y}$ be two vectors of the same length N with $x_i, y_i \in \{1, -1\}$. Then the Hamming distance is defined as

$$d(\mathbf{x}, \mathbf{y}) := \frac{1}{2} \sum_{i=0}^{N-1} |x_i - y_i|.$$

Example. Consider $\mathbf{x} = (1, -1, 1, 1)^T$, $\mathbf{y} = (-1, 1, 1, -1)^T$. Then $d(\mathbf{x}, \mathbf{y}) = 3$.

```cpp
// Hamming.cpp

#include <iostream>
using namespace std;

int distance(int* x,int* y,int n)
{
  int d = 0;
  for(int i=0;i<n;i++)
  {
  if(x[i] != y[i]) d++;
  }
  return d;
}

int main(void)
{
  int n = 4;
  int* x = new int[n];
  x[0] = 1; x[1] = -1; x[2] = 1; x[3] = 1;

  int* y = new int[n];
  y[0] = -1; y[1] = 1; y[2] = 1; y[3] = -1;

  int result = distance(x,y,n);
  cout << "result = " << result << endl;

  delete[] x; delete[] y;
  return 0;
}
```

Funkcja energetyczna

Funkcja energetyczna Hopfield udowodnił, że dyskretna sieć Hopfielda zbiegnie się do stabilnego punktu granicznego (wzorzec aktywacji jednostek) poprzez rozważenie funkcji energetycznej dla systemu. Funkcja energii jest również nazywana funkcją Liapunowa. Funkcja energetyczna jest funkcją ograniczoną poniżej i stanowi nie rosnącą funkcję stanu systemu. Dla sieci Hopfielda funkcja energii E jest

$$E(\mathbf{s}(t)) := -\frac{1}{2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} W_{ij}s_i(t)s_j(t) \equiv -\frac{1}{2}\mathbf{s}^T(t)W\mathbf{s}(t).$$

Prawo Hebba

Na podstawie prawa Hebba najszerzej akceptowaną hipotezą wyjaśniającą mechanizm uczenia się osiągniętą przez wspomnienia asocjacyjne jest to, że pewne modyfikacje funkcjonalne zachodzą w połączeniach synaptycznych między neuronami. W szczególności zakłada się, że skorelowane aktywności neuronów zwiększają siłę połączenia synaptycznego. Zwykle ta hipoteza plastyczności synaptycznej jest wyrażona ilościowo przez stwierdzenie, że ciężar synaptyczny W powinien wzrastać, ilekroć neurony i j mają jednocześnie ten sam poziom aktywności, i że w przeciwnym przypadku powinien się zmniejszyć.

Consequently, in order to store a prototype $\mathbf{x}$ according to Hebb's hypothesis, the synaptic weight should be modified by an amount

$$\Delta W_{ij} = \Delta W_{ji} = \eta x_i x_j$$

where η is a positive *learning factor*. For the complete synaptic matrix the modification will thus be

$$\Delta W = \eta \mathbf{x}\mathbf{x}^T .$$

Notice that $\mathbf{x}$ is a column vector with N components and therefore $\mathbf{x}^T$ is a row vector with N components. Thus W is an $N \times N$ matrix. If p prototype vectors $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{p-1}$ have to be stored, one considers that the resulting synaptic matrix is given by Hebb's law

$$W = \frac{1}{p} \sum_{k=0}^{p-1} \mathbf{x}_k \mathbf{x}_k^T .$$

Since the sign function applies to the vector $W\mathbf{s}$ the factor $1/p$ can also be omitted. Furthermore it is assumed that $W_{ii} = 0$ for $i = 0, 1, \ldots, N-1$. With this assumption we have

$$W = \sum_{k=0}^{p-1} \mathbf{x}_k \mathbf{x}_k^T - pI_N$$

where I_N is the $N \times N$ unit matrix. When the prototypes (stored pattern) are orthogonal, i.e.

$$\mathbf{x}_k^T \mathbf{x}_l = 0 \quad \text{if} \quad k \neq l$$

then the stored patterns are fixed points of the nonlinear map since

$$W\mathbf{x}_k = (N-p)\mathbf{x}_k, \qquad k = 0, 1, 2, \ldots, p-1 .$$

As an example we consider the case with $N = 5$ neurons. The input vectors (patterns) to be stored are

$$\mathbf{x}_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}, \qquad \mathbf{x}_1 = \begin{pmatrix} -1 \\ 1 \\ -1 \\ 1 \\ 1 \end{pmatrix}.$$

We note that

$$\mathbf{x}_0^T \mathbf{x}_1 = -1$$

i.e., the two input patterns are not orthogonal. Obviously, if the length of the vectors is an odd number the vectors cannot be orthogonal. For the entries of the connection matrix W we have

$$W = \mathbf{x}_0 \mathbf{x}_0^T + \mathbf{x}_1 \mathbf{x}_1^T - pI$$

where $p = 2$ (number of pattern stored) and I is the 5×5 unit matrix. The term $-pI$ is necessary so that the diagonal elements W_{jj} cancel out. Thus the symmetric connection matrix W is

$$W = \begin{pmatrix} 0 & 0 & 2 & 0 & -2 \\ 0 & 0 & 0 & 2 & 0 \\ 2 & 0 & 0 & 0 & -2 \\ 0 & 2 & 0 & 0 & 0 \\ -2 & 0 & -2 & 0 & 0 \end{pmatrix}.$$

Next we show that the two stored patterns are fixed points, i.e., we have to prove that

$$\mathbf{s}^* = \text{sign}(W\mathbf{s}^*)\,.$$

For the first stored pattern $\mathbf{x}_0$ we have

$$\text{sign}(W\mathbf{x}_0) = \text{sign}\begin{pmatrix} 4 \\ 2 \\ 4 \\ 2 \\ -4 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ -1 \end{pmatrix} = \mathbf{x}_0\,.$$

Analogously we prove that $\mathbf{x}_1$ is a fixed point, i.e.

$$\text{sign}(W\mathbf{x}_1) = \text{sign}\begin{pmatrix} -4 \\ 2 \\ -4 \\ 2 \\ 4 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \\ -1 \\ 1 \\ 1 \end{pmatrix} = \mathbf{x}_1\,.$$

Next we find the energy values from the energy function. From

$$E = -\frac{1}{2}\mathbf{s}^T W \mathbf{s}$$

we obtain for the first stored pattern $\mathbf{x}_0$ the energy $E = -8$. Analogously, for the second stored pattern $\mathbf{x}_1$ we find $E = -8$. Recall that there are $2^5 = 32$ configurations.

Next we look at the time evolution of the initial state

$$\mathbf{s}(0) = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}\,.$$

We find

$$W\mathbf{s}(0) = \begin{pmatrix} 0 \\ 2 \\ 0 \\ 2 \\ -4 \end{pmatrix}, \qquad \mathbf{s}(1) = \text{sign}(W\mathbf{s}(0)) = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}$$

where we used the rule that sign(0) is assigned to the corresponding value in the vector $\mathbf{s}(0)$. We find that $\mathbf{s}(1)$ is the stored pattern $\mathbf{x}_0$. This means we have reached a stable fixed point. The energy value of the initial state $\mathbf{s}(0)$ is $E = 0$. $E = -8$ is the lowest energy state the system has.

Kohonen network

Ciekawą cechą układów neuronowych jest możliwość tworzenia samoorganizujących się, zachowujących topologię mapowań dowolnej przestrzeni cech, np. Powierzchni ciała. Samoorganizacja oznacza, że mapowanie na korze neuronów powstaje tylko poprzez stochastyczne przedstawienie neuronów cechami wejściowymi. Mapowanie nazywa się zachowaniem topologii, gdy sąsiednie neurony reprezentują sąsiednie regiony w przestrzeni cech. Aby jak najlepiej wykorzystać podaną liczbę neuronów, ważne obszary przestrzeni cech (wysoka gęstość prezentacji) są reprezentowane przez więcej neuronów, tj. Rozdzielczość mapowania jest zoptymalizowana. Samoorganizująca się sieć neuronowa (zwana także mapą cech Kohonena lub mapą zachowującą topologię) przyjmuje strukturę topologiczną wśród jednostek klastra. Ta właściwość jest obserwowana w mózgu, ale nie występuje w innych sztucznych sieciach. Istnieją p jednostek klastrowych ułożonych w jedno- lub dwuwymiarowy układ.

Sygnały wejściowe x_k to wektory. Wektor ciężaru dla jednostki skupienia służy jako przykład wzorców wejściowych powiązanych z tym skupieniem. Podczas procesu samoorganizacji wybrano jednostkę klastra, której wektor masy najbardziej odpowiada wzorowi wejściowemu (zazwyczaj kwadratowi minimalnej odległości euklidesowej) zwycięzcy. Zwycięska jednostka i jej jednostki sąsiednia (pod względem topologii jednostek klastra) aktualizują swoje wagi. Wektory ciężaru w sąsiednich jednostek na ogół nie są zbliżone do wzorca wejściowego. Na przykład dla liniowego układu jednostek skupień sąsiedztwo promienia r wokół jednostki skupienia składa się ze wszystkich $\max(1, j^* - r) \leq j \leq \min(j^* + r, p)$.

Tam, gdzie dane wejściowe pasują do wektorów węzłów, tam obszar mapy jest selektywnie optymalizowany, aby reprezentować średnią danych treningowych dla tej klasy. Z losowo zorganizowanego zestawu węzłów siatka osiada na mapie obiektów, która ma lokalną reprezentację i jest samoorganizująca. Samoorganizująca mapa obiektów ma na celu uchwycenie topologii i rozkładu prawdopodobieństwa danych wejściowych. Odwzorowania od wyższych do niższych wymiarów są również możliwe z samoorganizującą się mapą obiektów i są ogólnie przydatne do wymiarowania danych wejściowych.

```
// Fausett.cpp

#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

double euclidean(double *vec1,double *vec2,int n)
{
  double dist = 0.0;
  for(int i=0;i<n;i++)
  dist += (vec1[i]-vec2[i])*(vec1[i]-vec2[i]);
  return dist;
}
```

```
double distance(int i,int jstar)
{
  return double(i!=jstar);
  // returns 1.0 if i != jstar
  // returns 0.0 if i == jstar
}

double h(double d) { return 1.0 - d; }

void train(double **W,int n,int cols,double *vec,double rate)
{
  int i,j;
  int win = 0;
  double windist = euclidean(W[0],vec,n),edist;
  for(i=0;i<cols;i++)
  if((edist=euclidean(W[i],vec,n)) < windist)
  { win = i; windist = edist; }
  for(i=0;i<cols;i++)
  for(j=0;j<n;j++)
  W[i][j] += rate*h(distance(i,win))*(vec[j]-W[i][j]);
}

int main(void)
{
  int i, j;
  int T = 10000; // number of iterations
  const int m = 4;
  int cols;
  double eta = 0.6;  // learning rate

  // training vectors
  double x0[m] = { 1.0,1.0,0.0,0.0 };
  double x1[m] = { 0.0,0.0,0.0,1.0 };
  double x2[m] = { 1.0,0.0,0.0,0.0 };
  double x3[m] = { 0.0,0.0,1.0,1.0 };

  cout << "Enter number of columns for weight matrix: ";
  cin >> cols;
```

```
    for(j=0;j<m;j++)
    W[i][j] = rand()/double(RAND_MAX);

    for(i=0;i<T;i++)
    {
    train(W,m,cols,x0,eta);
    train(W,m,cols,x1,eta);
    train(W,m,cols,x2,eta);
    train(W,m,cols,x3,eta);
    eta /= 1.05;      // learning rate decreased
    }
    for(i=0;i<cols;i++)
    {
    cout << "W[" << i << "]= [";
    for(j=0;j<m;j++)
    cout << W[i][j] << " ";
    cout << "]" << endl;
    }
    for(i=0;i<cols;i++)
    delete[] W[i];
    delete[] W;

    return 0;
}
```

W przypadku wyniku programu należy pamiętać, że jest on wrażliwy na warunki początkowe dla wektorów masy, które wybraliśmy za pomocą generatora liczb losowych. Ponadto wynik jest wrażliwy na wartość początkową i spadek szybkości uczenia się η. Typowy wynik to

```
// Fausett.cpp

#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

double euclidean(double *vec1,double *vec2,int n)
{
  double dist = 0.0;
  for(int i=0;i<n;i++)
  dist += (vec1[i]-vec2[i])*(vec1[i]-vec2[i]);
  return dist;
}
```

```
double distance(int i,int jstar)
{
  return double(i!=jstar);
  // returns 1.0 if i != jstar
  // returns 0.0 if i == jstar
}

double h(double d) { return 1.0 - d; }

void train(double **W,int n,int cols,double *vec,double rate)
{
  int i,j;
  int win = 0;
  double windist = euclidean(W[0],vec,n),edist;
  for(i=0;i<cols;i++)
  if((edist=euclidean(W[i],vec,n)) < windist)
  { win = i; windist = edist; }
  for(i=0;i<cols;i++)
  for(j=0;j<n;j++)
  W[i][j] += rate*h(distance(i,win))*(vec[j]-W[i][j]);
}

int main(void)
{
  int i, j;
  int T = 10000; // number of iterations
  const int m = 4;
  int cols;
  double eta = 0.6;  // learning rate

  // training vectors
  double x0[m] = { 1.0,1.0,0.0,0.0 };
  double x1[m] = { 0.0,0.0,0.0,1.0 };
  double x2[m] = { 1.0,0.0,0.0,0.0 };
  double x3[m] = { 0.0,0.0,1.0,1.0 };

  cout << "Enter number of columns for weight matrix: ";
  cin >> cols;
```

Problem wędrownego sprzedawcy jest chyba najbardziej znanym problemem we wszystkich optymalizacjach sieciowych i kombinatorycznych. Problem jest łatwy do stwierdzenia. Zaczynając od swojej bazy macierzystej, węzła 1, sprzedawca chce odwiedzić każde z kilku miast reprezentowanych przez węzły 2, ..., n, dokładnie raz i wrócić do domu, robiąc to przy możliwie najniższych kosztach podróży. Każde możliwe rozwiązanie tego problemu nazywamy wycieczką (po miastach). Jest n! możliwych wycieczek, jeśli jest n miast, ale niektóre z nich są takie same. Problem wędrownego sprzedawcy jest ogólnym modelem rdzenia, który uchwycił kombinacyjną istotę większości problemów z routingiem, a w rzeczywistości większość innych problemów z routingiem stanowi jego rozszerzenie. Na przykład, w klasycznym problemie z trasowaniem pojazdów, zestaw pojazdów, każdy o stałej pojemności, musi odwiedzić zestaw klientów (np. Sklepy spożywcze), aby dostarczyć (lub odebrać) zestaw towarów. Chcemy ustalić najlepszy możliwy zestaw tras dostaw.

Po przypisaniu zestawu klientów do pojazdu, pojazd ten powinien odbyć minimalną wycieczkę przez zestaw przypisanych do niego klientów; oznacza to, że powinien odwiedzić tych klientów podczas optymalnej wycieczki dla sprzedawców. Problem wędrownego sprzedawcy pojawia się również w przypadku problemów, które na powierzchni nie mają związku z routingiem. Załóżmy na przykład, że chcemy znaleźć sekwencję ładowania zadań na maszynie i że ilekroć proces maszyny wykonuje zadanie i po zadaniu j, musimy zresetować maszynę, co pociąga za sobą czas konfiguracji c_{ij}. Następnie, aby znaleźć sekwencję przetwarzania, która minimalizuje całkowity czas konfiguracji, musimy rozwiązać problem podróżnego sprzedawcy - maszyna, która działa jako „sprzedawca", musi „odwiedzić" zlecenia w najbardziej opłacalny sposób. Prezentujemy problem podróżnego sprzedawcy jako wbudowaną (ukierunkowaną) strukturę przepływu sieci Niech a, oznacza koszt podróży z miasta i do miasta j, a y_{ij} będzie zmienną zero-jedynkową, wskazującą, czy sprzedawca podróżuje z miasta i do miasta j.

Ponadto zdefiniujmy zmienne przepływu x_{ij} dla każdego arc(i, j) i załóżmy, że sprzedawca ma n-1 jednostek dostępnych w węźle 1, które arbitralnie wybieramy jako „węzeł źródłowy”, i że musi dostarczyć 1 jednostkę do buforowania innych węzłów.

$$\text{Minimize} \quad \sum_{(i,j)\in A} c_{ij}y_{ij} \tag{a}$$

subject to

$$\sum_{1\le j\le n} y_{ij} = 1 \qquad \text{for all } i = 1, 2, \ldots, n \tag{b}$$

$$\sum_{1\le i\le n} y_{ij} = 1 \qquad \text{for all } j = 1, 2, \ldots, n \tag{c}$$

$$\mathcal{N}x = b \tag{d}$$

$$x_{ij} \le (n-1)y_{ij} \quad \text{for all } (i,j) \in A \tag{e}$$

$$x_{ij} \ge 0 \quad \text{for all } (i,j) \in A \tag{f}$$

$$y_{ij} = 0 \text{ or } 1 \quad \text{for all } (i,j) \in A. \tag{g}$$

To interpret this formulation, let

$$A' := \{\,(i,j) : y_{ij} = 1\,\}, \qquad A'' := \{\,(i,j) : x_{ij} > 0\,\}.$$

```
// kohonen.cpp

#include <iostream>
#include <cmath>     // for fabs, exp
#include <cstdlib>
#include <ctime>
#include <iomanip>   // for setprecision
using namespace std;

// find the position in the array d
// which possesses the smallest number
int minimum(double* d,int M)
{
  int m = 0;
  double r = d[0];
  for(int j=1;j<M;j++)
  {
  if(d[j] < r) { r = d[j]; m = j; }
  }
  return m;
}

// lateral interaction of neurons
double map(double a,double b,double c,double d,double
{
  double result;
  result = exp(-(fabs(a-c) + fabs(b-d))/(2*s*s));
  return result;
}

int main(void)
{
  int N = 8;  // number of cities
  double* x = new double[N];
  double* y = new double[N];
  // coordinates of cities
  x[0] = 10.1; x[1] = 4.0;  x[2] = 0.1;  x[3] = 0.5;
  x[4] = 10.5; x[5] = 8.0;  x[6] = 12.0; x[7] = 18.0;
  y[0] = 20.2; y[1] = 7.0;  y[2] = 2.0;  y[3] = 10.7;
  y[4] = 0.6;  y[5] = 10.0; y[6] = 16.0; y[7] = 8.0;

  int M = 3*N; // M number of neurons
  double* u = new double[M];
  double* v = new double[M];
```

```
const double PI = 3.14159;

// neuron pattern at t = 0
for(int j=0;j<M;j++)
{
double k = j; double K = M;
u[j] = 10.0 + 10.0*sin(2.0*PI*k/K);   // neuron pattern
v[j] = 10.0 + 10.0*cos(2.0*PI*k/K);   // neuron pattern
}

double* dist = new double[M];
double* delW1 = new double[M];
double* delW2 = new double[M];

double eta0 = 0.8;   // learning rate at t = 0
long T = 15000;      // number of iterations

srand((unsigned long) time(NULL)); // seed for random numbers

// iteration starts here
for(long cnt=0;cnt<T;cnt++)
{
int nrand = rand()%N;
for(j= 0;j<M;j++)
{
dist[j] = fabs(u[j]-x[nrand]) + fabs(v[j]-y[nrand]);
}

int imin = minimum(dist,M);
double c = cnt;  double it = T;
double s;
s = eta0*(1.0 - c/(it + 10000.0));

for(j=0;j<M;j++)
{
delW1[j] = 1.0*map(u[j],v[j],u[imin],v[imin],s)*(x[nrand]-u[j]);
delW2[j] = 1.0*map(u[j],v[j],u[imin],v[imin],s)*(y[nrand]-v[j]);
}

for(j=0;j<M;j++)
{
u[j] += delW1[j]; v[j] += delW2[j];
}
}  // end iteration
```

```
    // display of output
    cout << endl << endl;
    for(j=0;j<M;j++)
    {
    cout << "u[" << j << "] = " << setprecision(3) << u[j] << "    ";
    cout << "v[" << j << "] = " << setprecision(3) << v[j] << endl;
    }

    delete[] x; delete[] y;
    delete[] u; delete[] v;
    delete[] delW1; delete[] delW2;
    delete[] dist;
    return 0;
}
```