Samoorganizująca się mapa Kohonena

Wstęp

Tu omówiono jeden rodzaj nienadzorowanego, konkurencyjnego uczenia się, mapę cech Kohonena lub mapę samoorganizującą się (SOM). Jak pamiętasz, w uczeniu nienadzorowanym nie ma oczekiwanych wyników przedstawianych sieci neuronowej, jak w przypadku nadzorowanego algorytmu uczenia, takiego jak propagacja wsteczna. Zamiast tego sieć, dzięki swoim właściwościom samoorganizującym się, jest w stanie wywnioskować relacje i dowiedzieć się więcej, gdy przedstawia się jej więcej danych wejściowych. Jedną z zalet tego schematu jest to, że można oczekiwać, że system będzie się zmieniał wraz ze zmieniającymi się warunkami i danymi wejściowymi. System nieustannie się uczy. Kohonen SOM to system sieci neuronowej opracowany przez Teuvo Kohonena z Politechniki Helsińskiej i jest często używany do klasyfikowania wejść do różnych kategorii. Aplikacje do map funkcji można prześledzić do wielu obszarów, w tym rozpoznawania mowy i sterowania silnikiem robota.

Konkurencyjne uczenie się

Mapa obiektów Kohonena może być używana samodzielnie lub jako warstwa innej sieci neuronowej. Warstwa Kohonena składa się z neuronów, które konkurują ze sobą. Podobnie jak w adaptacyjnej teorii rezonansu, Kohonen SOM jest kolejnym przykładem zastosowania strategii zwycięzca bierze wszystko. Dane wejściowe są podawane do każdego neuronu w warstwie Kohonena (z warstwy wejściowej). Każdy neuron określa swój wynik na podstawie wzoru sumy ważonej:

$$\text{Output} = \Sigma\, w_{ij}\, x_i$$

Wagi i dane wejściowe są zwykle znormalizowane, co oznacza, że wielkość wektorów wagi i wejściowych jest równa jeden. Wygrywa neuron z największą mocą wyjściową. Ten neuron ma końcową wartość wyjściową 1. Wszystkie inne neurony w warstwie mają wyjście równe zero. Różne wzorce wejściowe kończą się odpalaniem różnych zwycięskich neuronów. Podobne lub identyczne wzorce wejściowe są klasyfikowane do tego samego neuronu wyjściowego. Otrzymujesz podobne dane wejściowe zgrupowane razem. W Części 12 zobaczysz użycie sieci Kohonena w klasyfikacji wzorców.

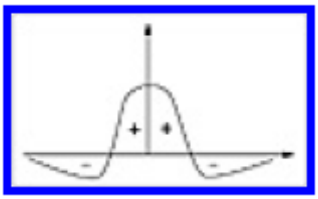
Normalizacja wektora Rozważmy wektor, A = ax + przez + cz. Znormalizowany wektor A’ otrzymuje się, dzieląc każdy składnik A przez pierwiastek kwadratowy z sumy kwadratów wszystkich składników. Innymi słowy, każdy składnik jest mnożony przez 1/[pierwiastek](a^2 + b^2 + c^2). Zarówno wektor wagowy, jak i wektor wejściowy są normalizowane podczas działania mapy cech Kohonena. Powodem tego jest to, że prawo treningu wykorzystuje odejmowanie wektora wagi od wektora wejściowego. Użycie normalizacji wartości w odejmowaniu redukuje oba wektory do stanu bez jednostek, a tym samym umożliwia odejmowanie podobnych wielkości. Już niedługo dowiesz się więcej o prawie szkoleń.

Hamowanie boczne

Hamowanie boczne to proces zachodzący w niektórych biologicznych sieciach neuronowych. Tworzą się boczne połączenia neuronów w danej warstwie i zgniatają odległych sąsiadów. Siła połączeń jest odwrotnie proporcjonalna do odległości. Pozytywne, wspierające połączenia są określane jako pobudzające, podczas gdy negatywne, zgniatające połączenia są określane jako hamujące. Biologiczny przykład hamowania bocznego występuje w ludzkim systemie widzenia.

Funkcja meksykańskiego kapelusza

Rysunek przedstawia funkcję zwaną meksykańską funkcją kapelusza, która pokazuje zależność między siłą połączenia a odległością od wygrywającego neuronu. Efektem tej funkcji jest stworzenie konkurencyjnego środowiska do nauki. Tylko wygrywające neurony i ich sąsiedzi uczestniczą w uczeniu się dla danego wzorca wejściowego.



Prawo szkoleniowe dla mapy Kohonen

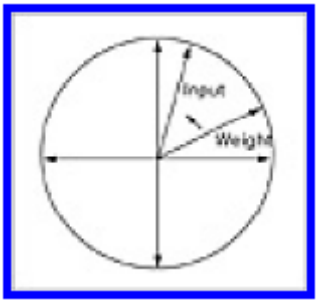
Prawo szkolenia dla mapy obiektów Kohonena jest proste. Zmiana wektora wagi dla danego neuronu wyjściowego to stała wzmocnienia alfa pomnożona przez różnicę między wektorem wejściowym a starym wektorem wagowym:

$$W_{new} = W_{old} + alpha * (Input - W_{old})$$

Zarówno stary wektor wagi, jak i wektor wejściowy są znormalizowane do długości jednostki. Alfa to stała wzmocnienia między 0 a 1.

Znaczenie ustawy o szkoleniach

Rozważmy przypadek dwuwymiarowego wektora wejściowego. Jeśli spojrzysz na okrąg jednostkowy, jak pokazano na rysunku poniżej, efektem prawa uczenia jest próba wyrównania wektora wagowego i wektora wejściowego. Każdy wzorzec próbuje przybliżyć wektor wagowy o ułamek określony przez alfa. Dla trzech wymiarów powierzchnia staje się jednostkową sferą zamiast okręgiem. Dla wyższych wymiarów powierzchnię nazywasz hipersferą. Niekoniecznie idealne jest idealne wyrównanie wektorów wejściowych i wag. Używasz sieci neuronowych ze względu na ich zdolność do rozpoznawania wzorców, ale także do uogólniania zbiorów danych wejściowych. Dopasowując wszystkie wektory wejściowe do odpowiednich wektorów wagi zwycięzcy, zasadniczo zapamiętujesz klasy zestawów danych wejściowych. Bardziej pożądane może być zbliżenie się do siebie, aby zaszumione lub niekompletne dane wejściowe nadal mogły powodować poprawną klasyfikację.



Wielkość sąsiedztwa i alfa

Na mapie Kohonena parametr zwany rozmiarem sąsiedztwa służy do modelowania efektu funkcji meksykańskiego kapelusza. Te neurony, które znajdują się w odległości określonej przez wielkość sąsiedztwa, uczestniczą w treningu i zmianach wektora ciężaru; ci, którzy znajdują się poza tym

dystansem, nie uczestniczą w nauce. Rozmiar sąsiedztwa zazwyczaj rozpoczyna się jako wartość początkowa i zmniejsza się w miarę kontynuacji cykli wzorca wejściowego. Ten proces ma tendencję do wspierania strategii „zwycięzca bierze wszystko” poprzez ostatecznie wyodrębnienie zwycięskiego neuronu dla danego wzorca. Rysunek przedstawia liniowy układ neuronów o wielkości sąsiedztwa równej 2. Zwycięża zaszyfrowany neuron centralny. Zaciemnione sąsiednie neurony to te, które będą uczestniczyć w treningu.

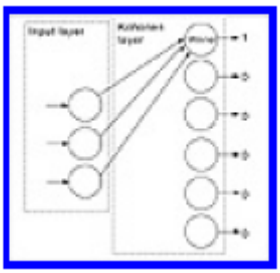
Oprócz rozmiaru sąsiedztwa, podczas symulacji zazwyczaj zmniejsza się również alfa. Zobaczysz te cechy, gdy opracujemy program map Kohonen.

Kod C++ do implementacji mapy Kohonena

Kod C++ dla mapy Kohonena opiera się na większości kodu opracowanego dla symulatora wstecznej propagacji błędów. Mapa Kohonena jest znacznie prostszym programem i może nie polegać na tak dużym zbiorze danych wejściowych. Program map Kohonena używa tylko dwóch plików, pliku wejściowego i pliku wyjściowego. Aby korzystać z programu, należy utworzyć zestaw danych wejściowych i zapisać go w pliku o nazwie input.dat. Plik wyjściowy nazywa się kohonen.dat i jest zapisywany w bieżącym katalogu roboczym. Wkrótce dowiesz się więcej o formatach tych plików.

Sieć Kohonena

Sieć Kohonen ma dwie warstwy, warstwę wejściową i warstwę wyjściową Kohonena. Warstwa wejściowa ma rozmiar określony przez użytkownika i musi odpowiadać rozmiarowi każdego wiersza (wzoru) w pliku danych wejściowych.



Modelowanie hamowania i wzbudzania bocznego

Funkcja kapelusza meksykańskiego pokazuje wartości dodatnie dla bezpośredniego sąsiedztwa neuronu i wartości ujemne dla neuronów odległych. Prawdziwa metoda modelowania obejmowałaby wzajemne wzbudzanie lub wsparcie dla neuronów znajdujących się w sąsiedztwie (przy czym to wzbudzenie wzrasta w przypadku neuronów bliższych) oraz hamowanie dla neuronów odległych poza sąsiedztwem. Ze względu na wydajność obliczeniową modelujemy boczne hamowanie i wzbudzenie, patrząc na maksymalne wyjście dla neuronów wyjściowych i sprawiając, że wyjście należy do neuronu zwycięskiego. Inne wyjścia są blokowane przez ustawienie ich wyjść na zero. Trening lub aktualizacja wagi jest wykonywana na wszystkich wyjściach, które znajdują się w odległości wielkości sąsiedztwa od zwycięskiego neuronu. Neurony spoza sąsiedztwa nie biorą udziału w treningu. Prawdziwy sposób modelowania hamowania bocznego byłby zbyt kosztowny, ponieważ liczba połączeń bocznych jest dość duża. Przekonasz się, że to przybliżenie doprowadzi do sieci z wieloma, jeśli nie wszystkimi, właściwościami prawdziwego podejścia do modelowania sieci Kohonena.

Klasy do wykorzystania

Korzystamy z wielu klas z symulatora backpropagation. Potrzebujemy tylko dwóch warstw, warstwy wejściowej i warstwy Kohonena. Tworzymy nową klasę warstwy o nazwie Kohonen i nową klasę sieci o nazwie Kohonen_network.

Ponowne odwiedzanie klasy warstw

Klasa warstwy musi zostać nieznacznie zmodyfikowana, jak pokazano na listingu 1.

Listing 1 Modyfikacja warstwy layer.h

```
// layer.h V.Rao, H. Rao

// header file for the layer class hierarchy and

// the network class

#define MAX_LAYERS 5

#define MAX_VECTORS 100

class network;

class Kohonen_network;

class layer

{

protected:

int num_inputs;

int num_outputs;

float *outputs;// pointer to array of outputs

float *inputs; // pointer to array of inputs, which

// are outputs of some other layer

friend network;

friend Kohonen_network; // update for Kohonen model

public:

virtual void calc_out()=0;

};

...
```

Zauważysz, że Kohonen_network staje się przyjacielem klasy warstwy, dzięki czemu Kohonen_network może mieć dostęp do danych warstwy.

Nowa klasa warstwy dla warstwy Kohonena

Następnym krokiem, który należy wykonać, jest utworzenie klasy Kohonen_layer i Kohonen_network. Pokazano to na listingu 2.

Listing 2 Klasa Kohonen_layer i Kohonen_network w pliku layerk.h

```
// layerk.h V.Rao, H. Rao
// header file for the Kohonen layer and
// the Kohonen network
class Kohonen_network;
class Kohonen_layer: public layer
{
protected:
float * weights;
int winner_index;
float win_distance;
int neighborhood_size;
friend Kohonen_network;
public:
Kohonen_layer(int, int, int);
~Kohonen_layer();
virtual void calc_out();
void randomize_weights();
void update_neigh_size(int);
void update_weights(const float);
void list_weights();
void list_outputs();
float get_win_dist();
};
class Kohonen_network
{
private:
layer *layer_ptr[2];
int layer_size[2];
```

```
int neighborhood_size;
public:
Kohonen_network();
~Kohonen_network();
void get_layer_info();
void set_up_network(int);
void randomize_weights();
void update_neigh_size(int);
void update_weights(const float);
void list_weights();
void list_outputs();
void get_next_vector(FILE *);
void process_next_pattern();
float get_win_dist();
int get_win_index();
};
```

Kohonen_layer pochodzi z klasy warstwy, więc ma wskaźniki odziedziczone, które wskazują na zestaw wyjść i zestaw danych wejściowych. Przyjrzyjmy się niektórym funkcjom i zmiennym składowym.

Kohonen_layer:

• float * weights Wskaźnik do macierzy wag.

• int winner_index Wartość indeksu wyjścia, które jest zwycięzcą.

• float win_distance Odległość euklidesowa wektora wagi zwycięzcy od wektora wejściowego.

• int neighborhood _size Wielkość otoczenia.

• Kohonen_layer(int, int, int) Konstruktor warstwy: wejścia, wyjścia i rozmiar otoczenia.

• ~Kohonen_layer().Destructor

• virtual void calc_out() Funkcja obliczania wyników; dla warstwy Kohonena modeluje to boczną konkurencję.

• void randomize_weights() Funkcja inicjująca wagi z losowymi wartościami normalnymi.

• void update_neigh_size(nt) Ta funkcja aktualizuje rozmiar otoczenia o nową wartość.

• void update_weights(const float) Ta funkcja aktualizuje wagi zgodnie z prawem treningu przy użyciu przekazanego parametru alpha.

• void list_weights() Ta funkcja może służyć do wylistowania macierzy wag.

• void list_outputs() Ta funkcja służy do zapisywania danych wyjściowych do pliku wyjściowego.

• float get_win_dist() Zwraca odległość euklidesową między wektorem wagi zwycięzcy a wektorem wejściowym.

Kohonen_network:

• layer *layer_ptr[2] Tablica wskaźników; element 0 wskazuje na warstwę wejściową, element 1 wskazuje na warstwę Kohonena.

• int layer_size[2] Tablica rozmiarów warstw dla dwóch warstw.

• int neighborhood_size Bieżący rozmiar sąsiedztwa.

• Kohonen_network().Constructor

• ~Kohonen_network().Destructor

• void get_layer_info() Pobiera informacje o rozmiarach warstw.

• void set_up_network(int) Łączy warstwy i konfiguruje mapę Kohonena.

• void randomize_weights() Tworzy losowe znormalizowane wagi.

• void update_neigh_size(int) Zmienia rozmiar otoczenia.

• void update_weights(const float) Wykonuje aktualizację wagi zgodnie z przepisami dotyczącymi treningu.

• void list_weights() Może służyć do wylistowania macierzy wag.

• void list_outputs() Może być użyty do wylistowania wyjść.

• void get_next_vector(FILE*) Funkcja pobiera inny wektor wejściowy z pliku wejściowego.

• void process_next_pattern() Stosuje wzorzec do mapy Kohonena.

• float get_win_dist() Zwraca odległość zwycięzcy od wektora wejściowego.

• int get_win_index() Zwraca indeks zwycięzcy.

Implementacja warstwy Kohonen i sieci Kohonen

Listing 3 pokazuje plik layerk.cpp, który zawiera implementację przedstawionych funkcji.

Listing 3 Plik implementacyjny dla warstwy Kohonen i sieci Kohonen :layerk.cpp

```
// layerk.cpp V.Rao, H.Rao

// compile for floating point hardware if available

#include “layer.cpp”

#include “layerk.h”

// ——————————————————————-

// Kohonen layer

//———————————————————————
```

```
Kohonen_layer::Kohonen_layer(int i, int o, int
init_neigh_size)
{
num_inputs=i;
num_outputs=o;
neighborhood_size=init_neigh_size;
weights = new float[num_inputs*num_outputs];
outputs = new float[num_outputs];
}
Kohonen_layer::~Kohonen_layer()
{
delete [num_outputs*num_inputs] weights;
delete [num_outputs] outputs;
}
void Kohonen_layer::calc_out()
{
// implement lateral competition
// choose the output with the largest
// value as the winner; neighboring
// outputs participate in next weight
// update. Winner's output is 1 while
// all other outputs are zero
int i,j,k;
float accumulator=0.0;
float maxval;
winner_index=0;
maxval=-1000000;
for (j=0; j<num_outputs; j++)
{
for (i=0; i<num_inputs; i++)
{
```

```
k=i*num_outputs;
if (weights[k+j]*weights[k+j] > 1000000.0)
{
cout << “weights are blowing up\n”;
cout << “try a smaller learning constant\n”;
cout << “e.g. beta=0.02
aborting...\n”;
exit(1);
}
outputs[j]=weights[k+j]*(*(inputs+i));
accumulator+=outputs[j];
}
// no squash function
outputs[j]=accumulator;
if (outputs[j] > maxval)
{
maxval=outputs[j];
winner_index=j;
}
accumulator=0;
}
// set winner output to 1
outputs[winner_index]=1.0;
// now zero out all other outputs
for (j=0; j< winner_index; j++)
outputs[j]=0;
for (j=num_outputs-1; j>winner_index; j—)
outputs[j]=0;
}
void Kohonen_layer::randomize_weights()
{
```

```
int i, j, k;
const unsigned first_time=1;
const unsigned not_first_time=0;
float discard;
float norm;
discard=randomweight(first_time);
for (i=0; i< num_inputs; i++)
{
k=i*num_outputs;
for (j=0; j< num_outputs; j++)
{
weights[k+j]=randomweight(not_first_time);
}
}
// now need to normalize the weight vectors
// to unit length
// a weight vector is the set of weights for
// a given output
for (j=0; j< num_outputs; j++)
{
norm=0;
for (i=0; i< num_inputs; i++)
{
k=i*num_outputs;
norm+=weights[k+j]*weights[k+j];
}
norm = 1/((float)sqrt((double)norm));
for (i=0; i< num_inputs; i++)
{
k=i*num_outputs;
weights[k+j]*=norm;
```

```
}
}
}
void Kohonen_layer::update_neigh_size(int new_neigh_size)
{
neighborhood_size=new_neigh_size;
}
void Kohonen_layer::update_weights(const float alpha)
{
int i, j, k;
int start_index, stop_index;
// learning law: weight_change =
// alpha*(input-weight)
// zero change if input and weight
// vectors are aligned
// only update those outputs that
// are within a neighborhood's distance
// from the last winner
start_index = winner_index -
neighborhood_size;
if (start_index < 0)
start_index =0;
stop_index = winner_index +
neighborhood_size;
if (stop_index > num_outputs-1)
stop_index = num_outputs-1;
for (i=0; i< num_inputs; i++)
{
k=i*num_outputs;
for (j=start_index; j<=stop_index; j++)
weights[k+j] +=
```

```
alpha*((*(inputs+i))-weights[k+j]);
}
}
void Kohonen_layer::list_weights()
{
int i, j, k;
for (i=0; i< num_inputs; i++)
{
k=i*num_outputs;
for (j=0; j< num_outputs; j++)
cout << “weight[“<<i<<”,”<<
j<<”] is: “<<weights[k+j];
}
}
void Kohonen_layer::list_outputs()
{
int i;
for (i=0; i< num_outputs; i++)
{
cout << “outputs[“<<i<<
“] is: “<<outputs[i];
}
}
float Kohonen_layer::get_win_dist()
{
int i, j, k;
j=winner_index;
float accumulator=0;
float * win_dist_vec = new float [num_inputs];
for (i=0; i< num_inputs; i++)
{
```

```
k=i*num_outputs;
win_dist_vec[i]=(*(inputs+i))-weights[k+j];
accumulator+=win_dist_vec[i]*win_dist_vec[i];
}
win_distance =(float)sqrt((double)accumulator);
delete [num_inputs]win_dist_vec;
return win_distance;
}
Kohonen_network::Kohonen_network()
{
}
Kohonen_network::~Kohonen_network()
{}
void Kohonen_network::get_layer_info()
{
int i;
//————————————————————————
//
// Get layer sizes for the Kohonen network
//
// ————————————————————————-
cout << " Enter in the layer sizes separated by spaces.\n";
cout << " A Kohonen network has an input layer \n";
cout << " followed by a Kohonen (output) layer \n";
for (i=0; i<2; i++)
{
cin >> layer_size[i];
}
// ————————————————————————————
// size of layers:
// input_layer layer_size[0]
```

```
// Kohonen_layer layer_size[1]
//————————————————————————————————-
}
void Kohonen_network::set_up_network(int nsz)
{
int i;
// set up neighborhood size
neighborhood_size = nsz;
//————————————————————————————————-
// Construct the layers
//
//————————————————————————————————-
layer_ptr[0] = new input_layer(0,layer_size[0]);
layer_ptr[1] =
new Kohonen_layer(layer_size[0],
layer_size[1],neighborhood_size);
for (i=0;i<2;i++)
{
if (layer_ptr[i] == 0)
{
cout << “insufficient memory\n”;
cout << “use a smaller architecture\n”;
exit(1);
}
}
//————————————————————————————————-
// Connect the layers
//
//————————————————————————————————-
// set inputs to previous layer outputs for the Kohonen layer
layer_ptr[1]->inputs = layer_ptr[0]->outputs;
```

```
}
void Kohonen_network::randomize_weights()
{
((Kohonen_layer *)layer_ptr[1])
->randomize_weights();
}
void Kohonen_network::update_neigh_size(int n)
{
((Kohonen_layer *)layer_ptr[1])
->update_neigh_size(n);
}
void Kohonen_network::update_weights(const float a)
{
((Kohonen_layer *)layer_ptr[1])
->update_weights(a);
}
void Kohonen_network::list_weights()
{
((Kohonen_layer *)layer_ptr[1])
->list_weights();
}
void Kohonen_network::list_outputs()
{
((Kohonen_layer *)layer_ptr[1])
->list_outputs();
}
void Kohonen_network::get_next_vector(FILE * ifile)
{
int i;
float normlength=0;
int num_inputs=layer_ptr[1]->num_inputs;
```

```
float *in = layer_ptr[1]->inputs;
// get a vector and normalize it
for (i=0; i<num_inputs; i++)
{
fscanf(ifile,”%f”,(in+i));
normlength += (*(in+i))*(*(in+i));
}
fscanf(ifile,”\n”);
normlength = 1/(float)sqrt((double)normlength);
for (i=0; i< num_inputs; i++)
{
(*(in+i)) *= normlength;
}
}
void Kohonen_network::process_next_pattern()
{
layer_ptr[1]->calc_out();
}
float Kohonen_network::get_win_dist()
{
float retval;
retval=((Kohonen_layer *)layer_ptr[1])
->get_win_dist();
return retval;
}
int Kohonen_network::get_win_index()
{
return ((Kohonen_layer *)layer_ptr[1])
->winner_index;
}
```

Przebieg programu i funkcji main()

Funkcja main() jest zawarta w pliku o nazwie kohonen.cpp, który pokazano na listingu 11.4. Aby skompilować ten program, wystarczy skompilować i utworzyć główny plik, kohonen.cpp. Inne pliki są w tym zawarte.

Listing 4 Główny plik implementacyjny, kohonen.cpp dla programu Kohonen Map

```
// kohonen.cpp V. Rao, H. Rao

// Program to simulate a Kohonen map

#include “layerk.cpp”

#define INPUT_FILE “input.dat”

#define OUTPUT_FILE “kohonen.dat”

#define dist_tol 0.05

void main()

{

int neighborhood_size, period;

float avg_dist_per_cycle=0.0;

float dist_last_cycle=0.0;

float avg_dist_per_pattern=100.0; // for the latest cycle

float dist_last_pattern=0.0;

float total_dist;

float alpha;

unsigned startup;

int max_cycles;

int patterns_per_cycle=0;

int total_cycles, total_patterns;

// create a network object

Kohonen_network knet;

FILE * input_file_ptr, * output_file_ptr;

// open input file for reading

if ((input_file_ptr=fopen(INPUT_FILE,”r”))==NULL)

{

cout << “problem opening input file\n”;

exit(1);

}
```

```
// open writing file for writing
if ((output_file_ptr=fopen(OUTPUT_FILE,"w"))==NULL)
{
cout << "problem opening output file\n";
exit(1);
}
// ————————————————————————-
// Read in an initial values for alpha, and the
// neighborhood size.
// Both of these parameters are decreased with
// time. The number of cycles to execute before
// decreasing the value of these parameters is
// called the period. Read in a value for the
// period.
// ————————————————————————-
cout << " Please enter initial values for:\n";
cout << "alpha (0.01-1.0),\n";
cout << "and the neighborhood size (integer between 0
and
50)\n";
cout << "separated by spaces, e.g. 0.3 5 \n ";
cin >> alpha >> neighborhood_size ;
cout << "\nNow enter the period, which is the\n";
cout << "number of cycles after which the values\n";
cout << "for alpha the neighborhood size are
decremented\n";
cout << "choose an integer between 1 and 500 , e.g. 50 \n";
cin >> period;
// Read in the maximum number of cycles
// each pass through the input data file is a cycle
cout << "\nPlease enter the maximum cycles for the
```

```
simulation\n";
cout << "A cycle is one pass through the data set.\n";
cout << "Try a value of 500 to start with\n\n";
cin >> max_cycles;
// the main loop
//
// continue looping until the average distance is less
// than the tolerance specified at the top of this file
// , or the maximum number of
// cycles is exceeded;
// initialize counters
total_cycles=0; // a cycle is once through all the input data
total_patterns=0; // a pattern is one entry in the input data
// get layer information
knet.get_layer_info();
// set up the network connections
knet.set_up_network(neighborhood_size);
// initialize the weights
// randomize weights for the Kohonen layer
// note that the randomize function for the
// Kohonen simulator generates
// weights that are normalized to length = 1
knet.randomize_weights();
// write header to output file
fprintf(output_file_ptr,
"cycle\tpattern\twin index\tneigh_size\tavg_dist_per_pa
tern\n");
fprintf(output_file_ptr,
"————————————————————————————————\n");
// main loop
startup=1;
```

```
total_dist=0;
while (
(avg_dist_per_pattern > dist_tol)
&& (total_cycles < max_cycles)
|| (startup==1)
)
{
startup=0;
dist_last_cycle=0; // reset for each cycle
patterns_per_cycle=0;
// process all the vectors in the datafile
while (!feof(input_file_ptr))
{
knet.get_next_vector(input_file_ptr);
// now apply it to the Kohonen network
knet.process_next_pattern();
dist_last_pattern=knet.get_win_dist();
// print result to output file
fprintf(output_file_ptr,"%i\t%i\t%i\t\t%i\t\t%f\n",
total_cycles,total_patterns,knet.get_win_index(),
neighborhood_size,avg_dist_per_pattern);
total_patterns++;
// gradually reduce the neighborhood size
// and the gain, alpha
if (((total_cycles+1) % period) == 0)
{
if (neighborhood_size > 0)
neighborhood_size —;
knet.update_neigh_size(neighborhood_size);
if (alpha>0.1)
alpha -= (float)0.1;
```

```
}
patterns_per_cycle++;
dist_last_cycle += dist_last_pattern;
knet.update_weights(alpha);
dist_last_pattern = 0;
}
avg_dist_per_pattern= dist_last_cycle/patterns_per_cycle;
total_dist += dist_last_cycle;
total_cycles++;
fseek(input_file_ptr, 0L, SEEK_SET); // reset the file
pointer
// to the beginning of
// the file
} // end main loop
cout << "\n\n\n\n\n\n\n\n\n\n\n";
cout << "————————————————————————————————————-\n";
cout << " done \n";
avg_dist_per_cycle= total_dist/total_cycles;
cout << "\n";
cout << "——>average dist per cycle = " << avg_dist_per_cycle
<< " <—-\n";
cout << "——>dist last cycle = " << dist_last_cycle << " <—-
\n";
cout << "->dist last cycle per pattern= " <<
avg_dist_per_pattern << " <—-\n";
cout << "————————>total cycles = " << total_cycles << " <—-\n";
cout << "————————>total patterns = " << total_patterns << " <—-
\n";
cout << "————————————————————————————————————-\n";
// close the input file
fclose(input_file_ptr);
```

}

Przebieg programu

Przebieg programu jest bardzo podobny do symulatora propagacji wstecznej. Kryterium zakończenia symulacji w programie Kohonen jest średnia odległość zwycięzcy. Jest to miara odległości euklidesowej między wektorem wejściowym a wektorem wagi zwycięzcy. Odległość ta jest pierwiastkiem kwadratowym z sumy kwadratów różnic między poszczególnymi składowymi wektora między dwoma wektorami.

Wyniki z uruchomienia programu Kohonena

Po skompilowaniu programu musisz utworzyć plik wejściowy, aby go wypróbować. Najpierw użyjemy bardzo prostego pliku wejściowego i zbadamy wyniki.

Prosty pierwszy przykład

Utwórzmy plik wejściowy input.dat, który zawiera tylko dwa dowolne wektory:

0,4 0,98 0,1 0,2

0,5 0,22 0,8 0,9

Plik zawiera dwa czterowymiarowe wektory. Spodziewamy się, że dane wyjściowe zawierają inny neuron zwycięzcy dla każdego z tych wzorców. Jeśli tak jest, to odwzorowanie Kohonena przypisało różne kategorie dla każdego z wektorów wejściowych iw przyszłości można spodziewać się takiej samej klasyfikacji zwycięzcy dla wektorów bliskich lub równych tym wektorom. Uruchamiając program Kohonen map, zobaczysz następujące dane wyjściowe:

Please enter initial values for:

alpha (0.01-1.0),

and the neighborhood size (integer between 0 and 50)

separated by spaces, e.g. 0.3 5

0.3 5

Now enter the period, which is the

number of cycles after which the values

for alpha the neighborhood size are decremented

choose an integer between 1 and 500 , e.g. 50

50

Please enter the maximum cycles for the simulation

A cycle is one pass through the data set.

Try a value of 500 to start with

500

Enter in the layer sizes separated by spaces.

A Kohonen network has an input layer

followed by a Kohonen (output) layer

4 10

————————————————————————————

done

——>average dist per cycle = 0.544275 <——-

——>dist last cycle = 0.0827523 <——-

->dist last cycle per pattern= 0.0413762 <——-

————————>total cycles = 11 <——-

————————>total patterns = 22 <——-

————————————————————————————

Rozmiary warstw są podane jako 4 dla warstwy wejściowej i 10 dla warstwy Kohonena. Powinieneś wybrać rozmiar warstwy Kohonena, aby był większy niż liczba odrębnych wzorców, które Twoim zdaniem znajdują się w wejściowym zestawie danych. Jednym z danych wyjściowych wyświetlanych na ekranie jest odległość dla ostatniego cyklu na wzór. Ta wartość jest wyświetlana jako 0.04, czyli mniejsza niż wartość końcowa ustawiona na górze pliku kohonen.cpp wynosząca 0.05. Mapa zbiegła się w poszukiwaniu rozwiązania. Spójrzmy na plik, kohonen.dat, plik wyjściowy, aby zobaczyć mapowanie do indeksów zwycięzców:

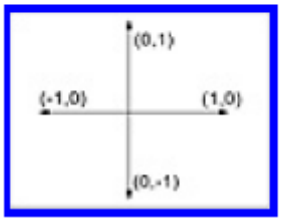
cycle	pattern	win index	neigh_size	avg_dist_per_pattern
0	0	1	5	100.000000
0	1	3	5	100.000000
1	2	1	5	0.304285
1	3	3	5	0.304285
2	4	1	5	0.568255
2	5	3	5	0.568255
3	6	1	5	0.542793
3	7	8	5	0.542793
4	8	1	5	0.502416
4	9	8	5	0.502416
5	10	1	5	0.351692
5	11	8	5	0.351692
6	12	1	5	0.246184
6	13	8	5	0.246184
7	14	1	5	0.172329
7	15	8	5	0.172329
8	16	1	5	0.120630
8	17	8	5	0.120630
9	18	1	5	0.084441
9	19	8	5	0.084441
10	20	1	5	0.059109

W tym przykładzie rozmiar sąsiedztwa pozostaje na początkowej wartości 5. W pierwszej kolumnie widać numer cyklu, aw drugiej numer wzoru. Ponieważ są dwa wzory na cykl, zobaczysz numer cyklu powtórzony dwukrotnie dla każdego cyklu. Mapa Kohonena była w stanie znaleźć dwa odrębne neurony zwycięskie dla każdego z wzorców. Jeden ma indeks zwycięzcy 1, a drugi indeks 8.

Przykład ortogonalnych wektorów wejściowych

Jako drugi przykład spójrz na rysunek poniżej, gdzie wybieramy wektory wejściowe na dwuwymiarowym okręgu jednostkowym, które są od siebie oddalone o 90 °. Plik input.dat powinien wyglądać następująco:

```
 1   0
 0   1
-1   0
 0  -1
```



Używając tych samych parametrów dla sieci Kohonena, ale z rozmiarami warstw 2 i 10, jakiego rezultatu można by się spodziewać? Plik wyjściowy, kohonen.dat, wygląda następująco:

cycle	pattern	win index	neigh_size	avg_dist_per_pattern
0	0	4	5	100.000000
0	1	0	5	100.000000
0	2	9	5	100.000000
0	3	3	5	100.000000
1	4	4	5	0.444558
1	5	0	5	0.444558
497	1991	6	0	0.707107
498	1992	0	0	0.707107
498	1993	0	0	0.707107
498	1994	6	0	0.707107
498	1995	6	0	0.707107
499	1996	0	0	0.707107
499	1997	0	0	0.707107
499	1998	6	0	0.707107
499	1999	6	0	0.707107

Widać, że ten przykład nie do końca działa. Mimo że wielkość sąsiedztwa stopniowo zmniejszała się do zera, cztery wejścia nie zostały sklasyfikowane jako różne wyjścia. Zwycięska odległość utknęła na wartości 0,707, co jest odległością od wektora przy 45°. Innymi słowy, mapa nieco za bardzo uogólnia, osiągając średnią wartość dla wszystkich wektorów wejściowych. Możesz rozwiązać ten problem, zaczynając od mniejszego rozmiaru otoczenia, co zapewnia mniejszą generalizację. Stosując te same parametry i wielkość sąsiedztwa wynoszącą 2, otrzymuje się następujące dane wyjściowe.

cycle	pattern	win index	neigh_size	avg_dist_per_pattern
0	0	5	2	100.000000
0	1	6	2	100.000000
0	2	4	2	100.000000
0	3	9	2	100.000000
1	4	0	2	0.431695
1	5	6	2	0.431695
1	6	3	2	0.431695
1	7	9	2	0.431695
2	8	0	2	0.504728
2	9	6	2	0.504728
2	10	3	2	0.504728
2	11	9	2	0.504728
3	12	0	2	0.353309
3	13	6	2	0.353309
3	14	3	2	0.353309
3	15	9	2	0.353309
4	16	0	2	0.247317
4	17	6	2	0.247317
4	18	3	2	0.247317
4	19	9	2	0.247317
5	20	0	2	0.173122
5	21	6	2	0.173122
5	22	3	2	0.173122
5	23	9	2	0.173122
6	24	0	2	0.121185
6	25	6	2	0.121185
6	26	3	2	0.121185
6	27	9	2	0.121185
7	28	0	2	0.084830
7	29	6	2	0.084830
7	30	3	2	0.084830
7	31	9	2	0.084830
8	32	0	2	0.059381
8	33	6	2	0.059381
8	34	3	2	0.059381
8	35	9	2	0.059381

W tym przypadku sieć szybko zbliża się do jednego zwycięzcy dla każdego z czterech wzorców wejściowych, a kryterium odległości jest poniżej ustalonego kryterium w ciągu ośmiu cykli. Możesz eksperymentować z innymi zestawami danych wejściowych i kombinacjami parametrów sieci Kohonena.

Odmiany i zastosowania sieci Kohonena

Istnieje wiele odmian sieci Kohonena. Niektóre z nich zostaną krótko omówione w tej sekcji.

Korzystanie z sumienia

DeSieno wykorzystał czynnik sumienia w sieci Kohonena. W przypadku neuronu wygrywającego, jeśli neuron wygrywa przez większą część czasu (w przybliżeniu więcej niż 1/n, gdzie n to liczba neuronów), wówczas ten neuron ma próg, który jest stosowany tymczasowo, aby umożliwić innym neuronom

szansa by wygrać. Celem tej modyfikacji jest umożliwienie bardziej równomiernego rozłożenia ciężaru podczas nauki.

LVQ: Nauka kwantyzatora wektorów

O LVQ (uczącym się kwantyzatorze wektorowym) przeczytałeś w poprzednich rozdziałach. W świetle mapy Kohonena należy wskazać, że LVQ jest po prostu nadzorowaną wersją sieci Kohonena. Wejścia i oczekiwane kategorie wyników są prezentowane w sieci na trening. Dane są grupowane, podobnie jak sieć Kohonena, zgodnie z podobieństwem do innych danych wejściowych.

Sieć kontrpropagacji

Topologia sieci neuronowej, zwana siecią kontrpropagacji, to połączenie warstwy Kohonena z warstwą Grossberga. Ta sieć została opracowana przez Roberta Hechta-Nielsena i jest przydatna do prototypowania systemów, z dość szybkim czasem uczenia się w porównaniu z propagacją wsteczną. Warstwa Kohonena zapewnia kategoryzację, podczas gdy warstwa Grossberga umożliwia uczenie warunkowane Hebbem. Kontrpropagacja jest z powodzeniem stosowana w aplikacjach do kompresji danych dla obrazów. Uzyskano współczynniki kompresji od 10:1 do 100:1 przy użyciu schematu kompresji stratnej, który koduje obraz za pomocą techniki zwanej kwantyzacją wektorową, w której obraz jest dzielony na reprezentatywne wektory podobrazów. Statystyki tych wektorów pozwalają stwierdzić, że duża część obrazu może być odpowiednio reprezentowana przez podzbiór wszystkich wektorów. Wektory o największej częstotliwości występowania są kodowane przy użyciu najkrótszych ciągów bitów, dzięki czemu uzyskuje się kompresję danych.

Aplikacja do rozpoznawania mowy

Kohonen stworzył fonetyczną maszynę do pisania, klasyfikując mowę , fale różnych fonemów mowy fińskiej na różne kategorie przy użyciu SOM Kohonena. Mapa fonemów Kohonena wykorzystywała do kalibracji 50 próbek każdego fonemu. Próbki te powodowały pobudzenie w sąsiedztwie komórek silniej niż w innych komórkach. Sąsiedztwo zostało oznaczone konkretnym fonemem, który wywołał pobudzenie. W przypadku wypowiedzi wypowiedzianej w sieci, odnotowywano dokładne sąsiedztwa, które były aktywne podczas wypowiedzi, jak długo iw jakiej kolejności. Krótkie wzbudzenia zostały potraktowane jako dźwięki przejściowe. Informacje uzyskane z sieci zostały następnie połączone, aby znaleźć słowa w wypowiedzi skierowanej do sieci.

Podsumowanie

Dowiedziałeś się o jednym z ważnych typów konkurencyjnego uczenia się, zwanym mapą cech Kohonena. Najważniejsze punkty tej dyskusji są przedstawione w następujący sposób:

• Mapa cech Kohonena jest przykładem nienadzorowanej sieci neuronowej, która jest używana głównie jako system klasyfikatorów lub system klastrowania danych. Ponieważ więcej danych wejściowych jest prezentowanych w tej sieci, sieć usprawnia proces uczenia się i jest w stanie dostosować się do zmieniających się danych wejściowych.

• Prawo uczenia sieci Kohonena próbuje wyrównać wektory wag w tym samym kierunku, co wektory wejściowe.

• Sieć Kohonena modeluje konkurencję poprzeczną jako formę samoorganizacji. Dla każdego wzorca wejściowego wyprowadzany jest jeden neuron zwycięzcy, który kategoryzuje dane wejściowe.

• Tylko neurony znajdujące się w pewnej odległości (sąsiedztwie) od zwycięzcy mogą brać udział w treningu dla danego wzorca wejściowego.