MODELOWANIE DANYCH POMIAROWYCH ZA POMOCĄ SZTUCZNYCH SIECI NEURONOWYCH OPTYMALIZOWANYCH ALGORYTMEM WSTECZNEJ PROPAGACJI BŁĘDÓW

dr inż. Igor Skalski

Gdańsk 2012

Spis treści

1. Wstęp **3**

2. Pojęcia podstawowe **3**

3. Podstawy teoretyczne **6**

4. Implementacja sieci neuronowej **8**

5. Przykład użycia sieci neuronowej **20**

6. Model zależności **24**

7. Prawa autorskie **24**

8. Literatura **27**

1. Wstęp

Sztuczne sieci neuronowe (*ang*: artificial neural networks), obok logiki rozmytej i algorytmów genetycznych, stanowią podstawowe składniki systemów sztucznej inteligencji. Matematyczne algorytmy sztucznych sieci neuronowych są oparte na zasadach, według których –jak się wydaje – funkcjonują biologiczne sieci neuronowe. Sztuczne sieci neuronowe są w szczególności stosowane do analizy złożonych problemów, trudnych lub niemożliwych do opisania równaniami matematycznymi. Są używane do tworzenie modeli i rozwiązywania problemów o nieznanej liczbie czynników wejściowych wpływających na wartości wyjściowe, gdy posiadana wiedza o badanych zależnościach jest niekompletna albo niepewna, w szczególności gdy dane nie są podane w postaci numerycznej. W odróżnieniu od metod klasycznych, sieci neuronowe pozwalają na tworzenie modeli uwzględniających wpływ nieznanych czynników na wyniki uzyskane na drodze pomiarów.

Modele utworzone przez sieci neuronowe mają charakter niejawny – nie stanowią równania, lecz zestaw danych, które w komplecie z siecią służą do wyznaczania modelowanych wartości.

Czas, który jest potrzebny do stworzenia modelu, czyli nauczenia sieci neuronowej właściwych odpowiedzi, jest zależny od liczby zestawów danych oraz złożoności analizowanego problemu. Najczęściej nauczenie sieci odpowiednich reakcji trwa od kilkunastu minut do kilku godzin. W złożonych przypadkach osiągnięcie przez sieć znacznego dopasowania trwa kilka dni.

Łatwość nauczenia sieci jest zależna przede wszystkim od jednoznaczności analizowanych zależności.

Tworzenie modelu w sieci neuronowej polega na optymalizowaniu wartości wag w taki sposób, aby ich wstępnie losowe wartości zmieniły się w sposób umożliwiający uzyskanie rozwiązania, które – w ujęciu statystycznym – zagwarantuje osiągnięcie minimum błędu globalnego.

Sieci neuronowe mogą być optymalizowane różnymi metodami. Mogą być tu stosowane między innymi metody przeszukiwania przestrzeni parametrów oraz algorytmy genetyczne. Do optymalizacji najczęściej jednak stosowana jest metoda wstecznej propagacji błędów (*ang*: error back propagation method). Metoda ta jest stosunkowo szybka i w zdecydowanej większości przypadków zbieżna. Tak, jak w przypadku klasycznych metod optymalizacji, w przypadku sztucznych sieci neuronowych przyspieszenie obliczeń jest okupione ryzykiem braku zbieżności albo oscylacjami wokół rozwiązania optymalnego.

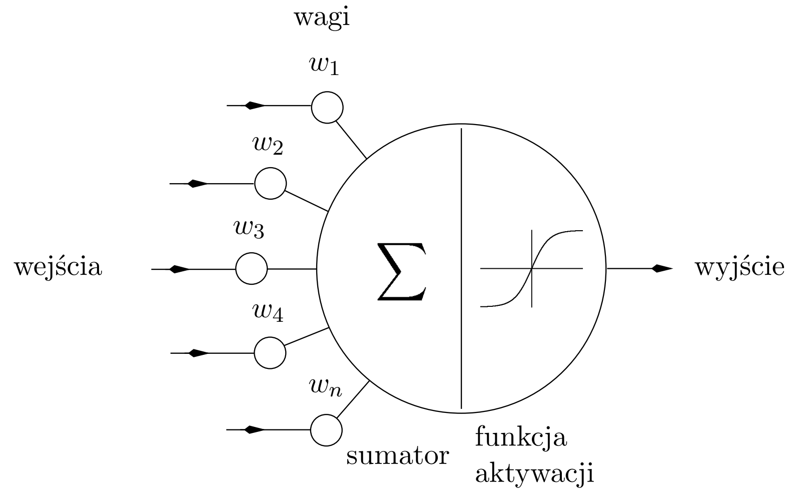
Dla odróżnienia od metod numerycznych model utworzony przez sztuczną sieć neuronową powinien mieć charakter ogólny – generalny, a dopasowanie nie powinno być zbyt dokładne. Dla podkreślenia tej różnicy w miejsce tradycyjnego pojęcia „aproksymacja" stosowane jest pojęcie „generalizacja".

Należy zaznaczyć, że użycie sztucznych sieci neuronowych nie gwarantuje osiągnięcia minimum globalnego, a więc i optymalnego rozwiązania. W dodatku kolejne uruchomienia sieci neuronowej mogą prowadzić do odmiennych rozwiązań, co wynika z losowego stanu początkowego sieci.

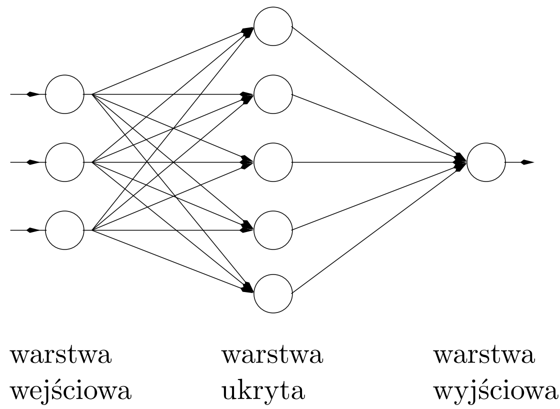
2. Pojęcia podstawowe

Podstawową jednostką logiczną służącą do budowy sztucznych sieci neuronowych jest neuron — rys. 1. Odwzorowuje on w sposób matematyczny wyobrażenie logiki działania neuronów biologicznych. Do neuronu przyłączone są elementy wejściowe — dendryty — doprowadzające sygnały wejściowe. Wartości te są sumowane z uwzględnieniem wartości wag synaptycznych i, po przetworzeniu przez funkcję aktywacji, są przekazywane do innych neuronów za pośrednictwem elementu wyjściowego — neurytu.

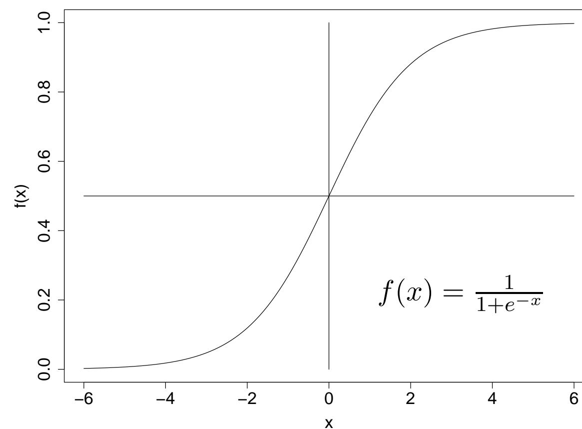
Sieć neuronowa składa się z neuronów rozmieszczonych w warstwach — rys. 2. Najczęściej



Rys. 1. Neuron — podstawowy element sieci neuronowej



Rys. 2. Budowa sztucznej sieci neuronowej



Rys. 3. Sigmoidalna funkcja aktywacji

stosowane sieci są zbudowane z trzech warstw: warstwy wejściowej, warstwy ukrytej i warstwy wyjściowej. Do analizy wyjątkowo złożonych problemów stosowane są sieci czterowarstwowe zawierające dwie warstwy ukryte, doświadczalnie stwierdzono jednak, że w większości przypadków sieci trójwarstwowe zawierające odpowiednią liczbę neuronów w warstwie ukrytej pozwalają z powodzeniem odwzorować analizowane zależności. Liczby neuronów stosowanych w warstwach wejściowej i wyjściowej są zależne od liczby danych wejściowych i wyjściowych zawartych w zestawach stosowanych do nauczania sieci. Liczba neuronów w warstwie ukrytej jest dobierana doświadczalnie i decyduje o dokładności odwzorowania zależności. Zbyt mała liczba neuronów prowadzi do przesadnej generalizacji zależności i nie pozwala na osiągnięcie optymalnego rozwiązania. Jest to spowodowane zbyt małą liczbą stopni swobody układu. Zbyt duża liczba neuronów może skutkować przeuczeniem sieci, wprowadzając zależności nie występujące w analizowanym przypadku.

W wyniku nauczania sieć neuronowa tworzy niejawny model analizowanych procesów. Model nie może wprawdzie być zapisany w postaci parametrów równania, jednak stan sieci neuronowej zapisuje się zwykle w postaci wartości wag synaptycznych, dzięki czemu model może być stosowany do obliczeń i symulacji.

Nauczanie sieci jest procesem żmudnym, czasochłonnym i wymaga ze strony eksperymentatora prowadzącego obliczenia pewnego doświadczenia i biegłości. Osiągnięcie optymalnego rozwiązania sprowadza się do wyznaczenia takich wartości wag synaptycznych aby po przypisaniu do wejścia sieci odpowiednich wartości, na wyjściu sieci pojawiła się prawidłowa odpowiedź. Jakość uzyskanego modelu jest zależna od wartości losowych, przypisanych początkowo wartościom wag synaptycznych. Odpowiedni stopień generalizacji modelu wymaga zastosowania możliwie niewielkiej liczby neuronów w warstwie ukrytej. Generalizacja jest też czasami osiągana na drodze ograniczenia liczby cykli obliczeniowych. W szczególności wygładzenie modelu można osiągnąć stosując odpowiednie wartości dzielnika w sigmoidalnej funkcji aktywacji lub przez dobranie odpowiednich zakresów podczas wstępnego skalowania wartości wejściowych i wyjściowych zastosowanych do nauczania sieci.

Optymalizacja sztucznych sieci neuronowych polega na poszukiwaniu rozwiązania na drodze wykonywania cyklicznych obliczeń oraz uwzględnianiu błędów występujących na wyjściach sieci. Błędy te zazwyczaj stanowią różnicę pomiędzy wartościami obliczonymi i danymi uzyskanymi na drodze pomiarów. Do optymalizacji mogą służyć proste metody, oparte na przeszukiwaniu przestrzeni wartości wag synaptycznych. Można też zastosować rozwiązania zaawansowane takie, jak opisana w dalszej części pracy metoda wstecznej propagacji błędów. Dobre wyniki, szczególnie w przypadkach, gdy korzystne jest wyznaczenie wartości zgrubnych, można też otrzymać stosując algorytmy genetyczne.

Stosunkowo trudnym do rozwiązania problemem w stosowaniu sztucznych sieci neuronowych jest ocena osiągniętego rozwiązania. Stosowanie do tego celu metod używanych w klasycznych metodach aproksymacji zwykle nie jest możliwe. Jakkolwiek, jak już wcześniej wspomniano, nauczanie sieci polega na zmniejszaniu zarówno błędów liczonych na poszczególnych wyjściach, jak i błędu globalnego sieci, to nadrzędnym celem nie jest minimalizowanie tych błędów. Model stanowiący generalizację analizowanych zależności może się bowiem charakteryzować znacznie większym błędem, niż model pozwalający na dosłowne odwzorowanie danych zastosowanych do nauczania sieci. Z powyższego wynika konieczność oceny intuicyjnej wyników obliczeń, opartej na obserwacji zarówno wartości błędów, jak i ogólnego obrazu modelu. Jak się wydaje możliwa jest też ocena osiągniętego rozwiązania w oparciu o bazę wiedzy i systemy logiki rozmytej.

Stosując sieci neuronowe należy pamiętać, że nie powinny one być stosowane dla ekstrapolacji do obszarów znacząco odległych od wartości stosowanych do nauki. Stosując sieci neuronowe należy też mieć na uwadze, że modele uzyskane na drodze nauczania nie przewyższają swoją jakością jakości danych użytych do nauki.

Nauczanie (strojenie) sieci neuronowych ze wsteczną propagacją błędów, pracujących

w trybie uczenia nadzorowanego, polega na cyklicznym podawaniu zestawów danych na wejście sieci, propagacji tych danych w kierunku wyjścia oraz porównywaniu wyników uzyskanych na wyjściu z wartościami rzeczywistymi. Błędy występujące na wyjściu są wprowadzane, w postaci poprawek, do wag synaptycznych w warstwie wyjściowej i warstwach poprzedzających. Wielokrotne powtórzenie procesu uczenia prowadzi do obliczenia odpowiednich wartości wag synaptycznych. Uzyskanie optymalnych wartości wag opiera się na statystycznych właściwościach zestawów danych stosowanych do nauczania sieci [1].

Proces uczenia jest prowadzony do chwili uzyskania wyników obarczonych dopuszczalnym błędem lub osiągnięcia odpowiedniego poziomu generalizacji zależności.

3. Podstawy teoretyczne

W ogólności tworzenie sieci neuronowej ze wsteczną propagacją błędów oraz nadzorowane uczenie sieci składa się z następujących kroków [2]:

1. Zaprojektowanie struktury sieci.

2. Przypisanie losowych wartości wagom synaptycznym.
   Wstępne wartości wag synaptycznych powinny należeć do zakresu

   $$\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right] \tag{1}$$

   gdzie: n — liczba wag synaptycznych należących do neuronu.

3. Przypisanie wartości wejściowych i wyjściowych do wejścia i wyjścia sieci.

4. Propagacja wartości z neuronów wejściowych w kierunku wyjścia sieci.
   Dla warstwy wejściowej, ukrytej i wyjściowej przyjęto odpowiednio: oznaczenia indeksów i j i k oraz warstw $[i]$, $[j]$ i $[k]$.
   Neurony należące do warstwy wejściowej zawierają pojedyncze wejście, zwykle bez wag synaptycznych. Wartość wyjściowa $O_i^{[i]}$ neuronu jest określona funkcją aktywacji neuronu (zwaną też funkcją przejścia neuronu) $f()$

   $$O_i = f(I_i) \tag{2}$$

   gdzie: I_i — sygnał wejściowy.
   Powszechnie stosowaną funkcją aktywacji jest funkcja sigmoidalna — rys. 3

   $$f(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

   Pochodna funkcji sigmoidalnej, wyrażona z wykorzystaniem wartości funkcji [1], jest określona równaniem

   $$f'(x) = f(x) \cdot [1 - f(x)] \tag{4}$$

   Pochodna funkcji aktywacji jest używana do obliczania wartości błędu neuronów warstwy wyjściowej.
   Kontynuacja propagacji sygnału polega na obliczeniu wartości wyjściowych warstwy ukrytej

   $$O_j^{[j]} = f\left(\sum_i (w_{ji}^{[j]} \cdot O_i^{[i]})\right) \tag{5}$$

   gdzie: $f()$ — funkcja aktywacji, w_{ji} — wartość wagi synaptycznej łączącej neuron i warstwy poprzedzającej $[i]$ z neuronem j warstwy bieżącej $[j]$, O_i wartość wyjściowa neuronu i w warstwie poprzedzającej $[i]$.

Następnie obliczane są wartości wyjściowe neuronów w warstwie wyjściowej

$$O_k^{[k]} = f\left(\sum_j (w_{kj}^{[k]} \cdot O_j^{[j]})\right) \tag{6}$$

Zastosowanie sigmoidalnej funkcji aktywacji w warstwie wejściowej powoduje ograniczenie zakresu stosowanych wartości do $-5 \ldots 5$. Użycie sigmoidalnej funkcji aktywacji w warstwie wyjściowej skutkuje ograniczeniem wartości wyjściowych do zakresu $0 \ldots 1$. Dane podawane zarówno do wejścia, jak i do wyjścia sieci należy dostosować od tych zakresów.

5. Obliczenie i propagacja wsteczna błędów
   Obliczenia wartości błędów neuronów wyjściowych można dokonać na różne sposoby. W prostym przypadku możliwe jest użycie różnicy wartości wymaganej $R_k^{[k]}$ i wartości obliczonej $O_k^{[k]}$

$$e_k^{[k]} = R_k^{[k]} - O_k^{[k]} \tag{7}$$

gdzie: $e_k^{[k]}$ — wartość błędu neuronu k.
Częściej jest stosowane równanie zawierające pochodną funkcji aktywacji — w przedstawionym przypadku funkcji sigmoidalnej

$$e_k^{[k]} = O_k^{[k]} \cdot (1 - O_k^{[k]}) \cdot (R_k^{[k]} - O_k^{[k]}) \tag{8}$$

Modyfikacja funkcji liczącej błędy neuronów wyjściowych pozwala na znaczne zwiększenie szybkości obliczeń [3]

$$e_k^{[k]} = \begin{cases} 1 + e^{(R_k^{[k]} - O_k^{[k]})^2} & \text{gdy} \quad (R_k^{[k]} - O_k^{[k]}) \geqslant 0 \\ -(1 + e^{(R_k^{[k]} - O_k^{[k]})^2}) & \text{gdy} \quad (R_k^{[k]} - O_k^{[k]}) < 0 \end{cases} \tag{9}$$

Błędy neuronów wyjściowych ulegają propagacji wstecznej do warstwy ukrytej

$$e_j^{[j]} = O_j^{[j]} \cdot (1 - O_j^{[j]}) \cdot \sum_k \left(e_k^{[k]} \cdot w_{kj}^{[k]}\right) \tag{10}$$

i do warstwy wejściowej

$$e_i^{[i]} = O_i^{[i]} \cdot (1 - O_i^{[i]}) \cdot \sum_j \left(e_j^{[j]} \cdot w_{ji}^{[j]}\right) \tag{11}$$

Korzystając z obliczonych wartości błędów wprowadza się poprawki do wag synaptycznych. Poprawki liczone są z wykorzystaniem współczynnika l, określającego szybkość nauczania. Zazwyczaj stosuje się $l = 0{,}2$. Zmniejsza to szybkość zbieżności, lecz zwiększa stabilność sieci

$$\Delta w_{ji}^{[j]} = l \cdot e_j^{[j]} \cdot O_i^{[i]} \tag{12}$$

$$w_{ji}^{[j]} = w_{ji}^{[j]} + \Delta w_{ji}^{[j]} \tag{13}$$

$$\Delta w_{kj}^{[k]} = l \cdot e_k^{[k]} \cdot O_j^{[j]} \tag{14}$$

$$w_{kj}^{[k]} = w_{kj}^{[k]} + \Delta w_{kj}^{[k]} \tag{15}$$

W celu zwiększenia szybkości nauczania sieci oraz zmniejszenia tendencji do poszukiwania minimum lokalnego, często stosowany jest dodatkowy składnik poprawki, nazywany *momentum*. Stanowi on ułamek wartości poprawki błędu Δw, obliczonej podczas poprzedniej iteracji

$$\beta \cdot \Delta w_{(t-1)} \tag{16}$$

gdzie: $\Delta w_{(t-1)}$ — wartość poprzedniej poprawki, β — stała *momentum* przyjmująca wartość od 0 do 0,95.
Wartość poprawki zawierająca *momentum* jest określona równaniem

$$\Delta w_{(t)} = \Delta w_{(t)} + \beta \cdot \Delta w_{(t-1)} \tag{17}$$

gdzie: $\Delta w_{(t)}$ — wartość bieżącej poprawki.

6. Błąd globalny E sieci neuronowej jest określony równaniem

$$E = \frac{1}{2} \sum_{k} \left(R_k^{[k]} - O_k^{[k]} \right)^2 \tag{18}$$

7. Powrót do punktu 3.
Zakończenie nauczania sieci następuje po osiągnięciu dopuszczalnej wartości błędu $R_k^{[k]} - O_k^{[k]}$ dla każdego zestawu danych lub po osiągnięciu odpowiedniego poziomu generalizacji modelu.

4. Implementacja sieci neuronowej

Program zapisany w języku C należy rozpocząć od przyłączenia wykorzystywanych bibliotek oraz zadeklarowania niektórych wykorzystywanych funkcji. W przykładzie ograniczono się do wykorzystania najbardziej standardowych bibliotek tak, aby kod programu był przenośny.

```
#include<stdio.h>
#include<stdlib.h>
 long int random(void);
 void srandom(unsigned int seed);
#include<math.h>
```

Strukturę trójwarstwowej sztucznej sieci neuronowej określa się liczbą neuronów składających się na warstwy: wejściową, ukrytą i wyjściową. Liczba neuronów w warstwach wejściowej i wyjściowej jest zależna od struktury danych użytych do obliczeń. Od liczby neuronów zawartych w warstwie ukrytej zależy poziom generalizacji. Liczbę tą dobiera się doświadczalnie.

```
#define INUM 2
#define HNUM 5
#define ONUM 1
```

Współczynnik szybkości uczenia sieci ma bardzo duży wpływ na proces osiągania optymalnego modelu. Zbyt mała wartość wydłuża czas obliczeń. Zbyt duża może prowadzić do braku zbieżności. W praktyce stosuje się wartości od 0,01 do 0,8.

```
#define LCOEF 0.1
```

Współczynnikiem pozwalającym na przyspieszenie obliczeń jest *momentum*. Zwiększa on zbieżność obliczeń każdego kroku wykorzystując dane uzyskane podczas kroku poprzedniego. W przypadku niestabilności sieci należy zrezygnować z przyspieszenia przypisując tej stałej wartość zerową.

Wartość *momentum* jest zazwyczaj mniejsza od wartości współczynnika szybkości uczenia.

```
#define MOMENTUM 0.1
```

Struktura neuronu wejściowego zawiera zmienne, w których umieszczone są wartości wejściowe sieci oraz zmienne pomocnicze służące do skalowania tych wartości do zakresu pracy sieci. W strukturze przechowywane są również obliczone wartości wyjścia i błędu.

```
typedef struct
{
  double input;
  double input_offset;
  double input_gain;
  double input_scaled;
  double output;
  double error;
}input_node_t;
```

W strukturze neuronu należącego do warstwy ukrytej przechowywane są wartości wag oraz obliczone wartości wyjścia i błędu.

```
typedef struct
{
  double weight[INUM];
  double dweight[INUM];
  double output;
  double error;
}hiden_node_t;
```

Struktura wyjściowa oprócz wartości obliczonych zawiera zmienne pomocnicze umożliwiające skalowanie wartości wyjściowej. Zawiera też wartość optymalną.

```
typedef struct
{
  double weight[HNUM];
  double dweight[HNUM];
  double output;
  double required_output;
  double output_offset;
  double required_output_scaled;
  double output_gain;
  double error;
}output_node_t;
```

Tablice struktur zawierają trzy warstwy: wejściową, ukrytą i wyjściową.

```
input_node_t input_layer[INUM];
hiden_node_t hiden_layer[HNUM];
output_node_t output_layer[ONUM];
```

Aby możliwe była kontynuacja obliczeń po przerwaniu pracy programu, bieżące wartości wag stanowiące stan sieci należy cyklicznie zapisywać na dysku. Stan ten można później przywracać odczytując wartości wag i odpowiednio je przypisując poszczególnym neuronom. Temu właśnie służą funkcje `write_nn` i `read_nn`.

```
int write_nn()
{
  int i,j;
  FILE *f;
  if(!(f=fopen("nn.dat","w+")))
    return(1);
  for(i=0;i<HNUM;i++)
    for(j=0;j<INUM;j++)
      fprintf(f,"%g ",hiden_layer[i].weight[j]);
  fprintf(f,"\n");
  for(i=0;i<ONUM;i++)
    for(j=0;j<HNUM;j++)
      fprintf(f,"%g ",output_layer[i].weight[j]);
  fprintf(f,"\n");
  if(fclose(f))
    return(1);
  return(0);
}

int read_nn()
{
  int i,j;
  FILE *f;
  if(!(f=fopen("nn.dat","r")))
    return(1);
  for(i=0;i<HNUM;i++)
    for(j=0;j<INUM;j++)
      fscanf(f,"%lg",&hiden_layer[i].weight[j]);
  for(i=0;i<ONUM;i++)
    for(j=0;j<HNUM;j++)
      fscanf(f,"%lg",&output_layer[i].weight[j]);
  if(fclose(f))
    return(1);
  fprintf(stderr,"read_nn: ok.\n");
  return(0);
}
```

Każdorazowo przed rozpoczęciem obliczeń należy wyzerować wartości zmiennych wykorzystywanych przez *momentum* do przyspieszania zbieżności.

```
void reset_dweights()
{
  int i,j,k;
  for(i=0;i<INUM;i++)
    for(j=0;j<HNUM;j++)
      hiden_layer[j].dweight[i]=0;
  for(j=0;j<HNUM;j++)
```

```
    for(k=0;k<ONUM;k++)
      output_layer[k].dweight[j]=0;
}
```

Jeżeli obliczenia nie stanowią kontynuacji wagom należy przypisać wartości losowe.

```
void randomize_weights()
{
  int i,j,k;
  double d;
  FILE *fd;
  if((fd=fopen("/dev/random","r")))
  {
    srandom(getc(fd));
    if(fclose(fd));
  }
  d=1.0/pow(INUM,0.5);
  for(i=0;i<INUM;i++)
    for(j=0;j<HNUM;j++)
      hiden_layer[j].weight[i]=(d*2.0*(double)rand()/(double)RAND_MAX)-d;
  d=1.0/pow(HNUM,0.5);
  for(j=0;j<HNUM;j++)
    for(k=0;k<ONUM;k++)
      output_layer[k].weight[j]=(d*2.0*(double)rand()/(double)RAND_MAX)-d;
  reset_dweights();
}
```

Najczęściej stosowane funkcje aktywacji ograniczają poprawność pracy sieci neuronowej do zakresu wejściowego $-5\ldots5$. Funkcja `prescale_input` analizuje dane wejściowe i oblicza odpowiednie współczynniki służące do wzmacnianie albo osłabianie i przesuwania w skali. Współczynniki zostają umieszczone w odpowiednich zmiennych w strukturze wskazanego neuronu wejściowego.

```
void prescale_input(int input,double * table,int values)
{
  int i;
  double mean=0;
  double maxd=0;
  for(i=0;i<values;i++)
    mean+=table[i];
  mean/=values;
  input_layer[input].input_offset=-mean;
  for(i=0;i<values;i++)
    if(fabs(table[i]-mean)>maxd)
      maxd=fabs(table[i]-mean);
  if(maxd==0)
    input_layer[input].input_gain=1.0;
  else
    input_layer[input].input_gain=4.0/maxd;
}
```

Również dane wyjściowe podlegają ograniczeniom. Zazwyczaj akceptowane są wartości z zakresu $0\ldots1$. Funkcja `prescale_output` po dokonaniu analizy danych oblicza współczynniki korekcyjne dla wskazanego neuronu wyjściowego.

```
void prescale_output(int output,double * table,int values)
{
  int i;
  double mean=0;
  double maxd=0;
  for(i=0;i<values;i++)
    mean+=table[i];
  mean/=values;
  output_layer[output].output_offset=-mean;
  for(i=0;i<values;i++)
    if(fabs(table[i]-mean)>maxd)
      maxd=fabs(table[i]-mean);
  if(maxd==0)
    output_layer[output].output_gain=1.0;
  else
    output_layer[output].output_gain=0.25/maxd;
}
```

Wartości podane na wejścia i wyjścia sieci muszą być każdorazowo przed rozpoczęciem kroku obliczeniowego przeskalowane z wykorzystaniem współczynników obliczonych za pomocą dwóch wcześniej przedstawionych funkcji.

```
void scale_values()
{
  int i;
  /* scale inputs */
  for(i=0;i<INUM;i++)
  {
    input_layer[i].input_scaled=input_layer[i].input+
      input_layer[i].input_offset;
    input_layer[i].input_scaled*=input_layer[i].input_gain;
  }
  /* scale outputs */
  for(i=0;i<ONUM;i++)
  {
    output_layer[i].required_output_scaled=
      output_layer[i].required_output+output_layer[i].output_offset;
    output_layer[i].required_output_scaled*=output_layer[i].output_gain;
    output_layer[i].required_output_scaled+=0.5;
  }
}
```

Najczęściej stosowaną funkcją aktywacji w sztucznych sieciach neuronowych jest funkcja sigmoidalna. W zależności od charakteru danych wejściowych stosuje się również inne funkcje, które mogą skutkować lepszym odwzorowaniem analizowanych zależności. Oprócz funkcji sigmoidalnej powszechnie stosowane są również funkcje: o kształcie tangensa hiperbolicznego, sinusoidalna, funkcja Gaussa, schodkowa i liniowa z nasyceniem.

Wyboru funkcji w programie dokonuje się definiując jedną ze stałych: `SIGMOIDAL`, `TANH`, `SIN`, `MPOW` albo `LINEAR_WITH_SATURATION`.

```
#define SIGMOIDAL
```

W funkcjach aktywacji: sigmoidalnej, o kształcie tangensa hiperbolicznego i sinusoidalnej stosowana jest zmienna modyfikująca sposób przebieg zależności.

```
#define T 1.0
```

Funkcja sigmoidalna jest najczęściej stosowaną funkcją aktywacji w sztucznych sieciach neuronowych. Jej kształt może być modyfikowany przez zmianę współczynnika T – rys. 4. Większe wartości współczynnika prowadzą do uzyskania większej generalizacji, a więc „gładszej" zależności.

```
#ifdef SIGMOIDAL
double activation_function(double x)
{
  return(1.0/(1.0+exp(-x/T)));
}
#endif
```

Krzywa opisana tangensem hiperbolicznym jest zbliżona do funkcji sigmoidalnej, jednak wartość współczynnika T inaczej wpływa na kształt funkcji – rys. 5.

```
#ifdef TANH
double activation_function(double x)
{
  return(0.5+tanh(x/T)/2.0);
}
#endif
```

Funkcja sinusoidalna jest stosowana w przypadkach, gdy oczekiwany jest znaczny poziom generalizacji – rys. 6.

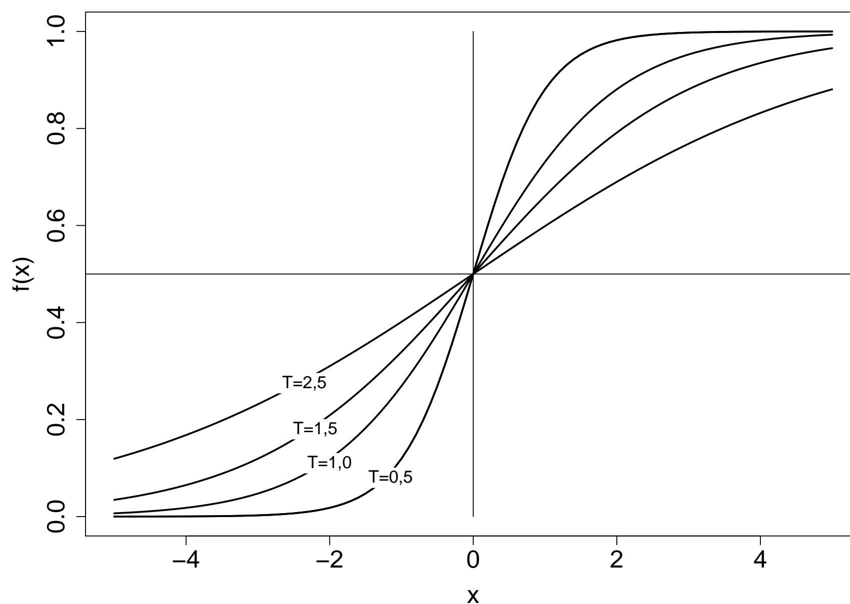
```
#ifdef SIN
double activation_function(double x)
{
  return(0.5+sin(x/T)/2.0);
}
#endif
```

Inne efekty można uzyskać stosując potęgową funkcję aktywacji o kształcie „odwrotnym" w stosunku do krzywych sigmoidanych – rys. 7.

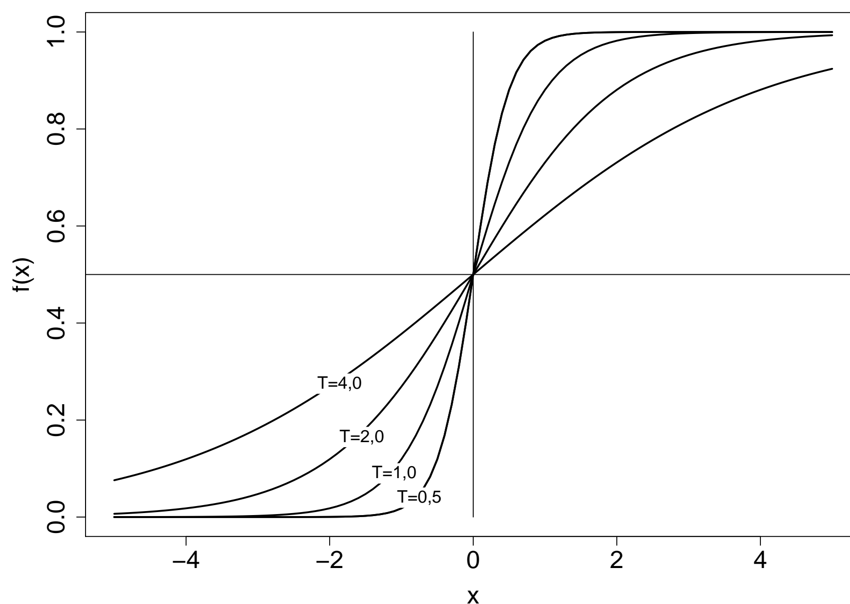
```
#ifdef MPOW
double activation_function(double x)
{
  double d=0.5+x/T*fabs(x/T);
  if(d<0)
    d=0;
  if(d>1)
    d=1;
  return(d);
}
#endif
```

Proste modele zawierające płaskie obszary otrzymuje się za pomocą funkcji aktywacji liniowej z nasyceniem przedstawionej na rys. 8.

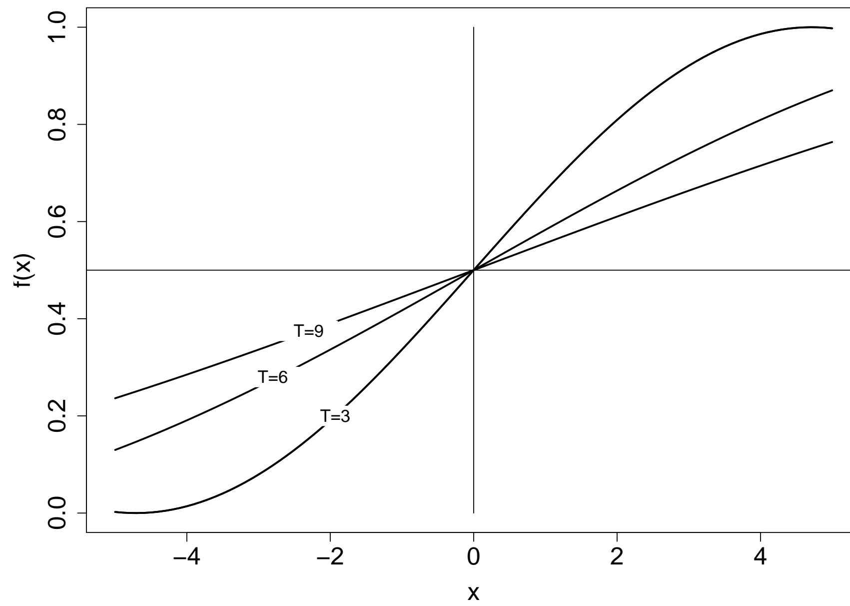
```
#ifdef LINEAR_WITH_SATURATION
double activation_function(double x)
{
  if(x<=-5.0)
    return(0);
  else
    if(x>=5.0)
      return(1.0);
    else
      return(0.5+x/10.0);
}
#endif
```



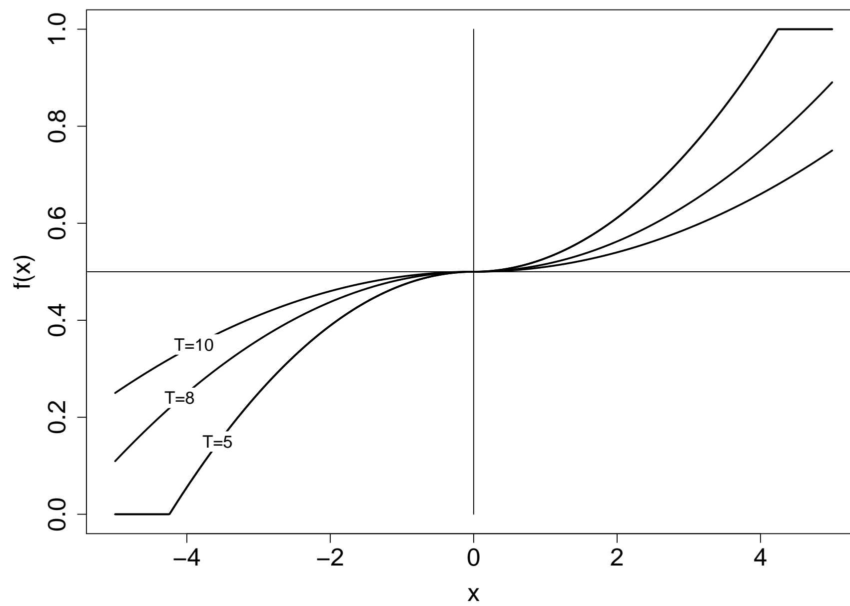
Rys. 4. Zależność kształtu sigmoidalnej funkcji aktywacji od wartości T



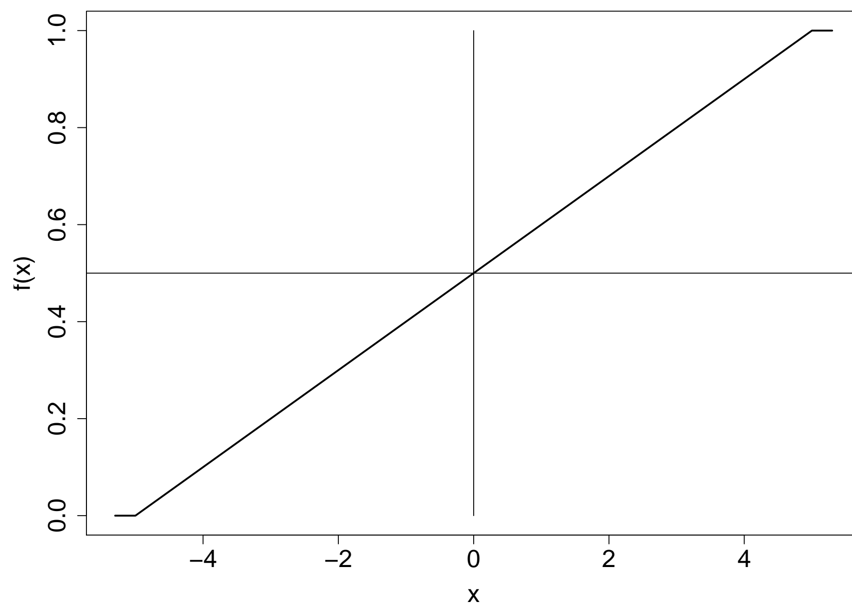
Rys. 5. Zależność kształtu funkcji aktywacji opisanej kształtem tangensa hiperbolicznego od wartości T



Rys. 6. Zależność kształtu sinusoidalnej funkcji aktywacji od wartości T



Rys. 7. Zależność kształtu potęgowej funkcji aktywacji od wartości T



Rys. 8. Kształt funkcji aktywacji liniowej z nasyceniem

Obserwowanie wartości błędu globalnego sieci neuronowej pozwala na szacunkową ocenę postępów pracy sieci. Należy jednak mieć na uwadze, że wartość błędu obliczona według równania 18 dotyczy ostatnio obliczonego zestawu danych, dlatego nie stanowi wartości właściwej dla wszystkich danych.

```
double global_error()
{
  int k;
  double e=0;
  for(k=0;k<ONUM;k++)
    e+=pow(output_layer[k].required_output_scaled-
           output_layer[k].output,2.0);
  return(e/0.5);
}
```

Propagacja wartości podanych na wejścia sieci przebiega zgodnie z równaniami 2, 5 i 6.

```
void forward_propagation()
{
  int i,j,k;
  double x;
  /* forward propagation */
  for(i=0;i<INUM;i++)
    input_layer[i].output=
      activation_function(input_layer[i].input_scaled);
  for(j=0;j<HNUM;j++)
  {
    x=0;
    for(i=0;i<INUM;i++)
      x+=(hiden_layer[j].weight[i]*input_layer[i].output);
    hiden_layer[j].output=activation_function(x);
  }
  for(k=0;k<ONUM;k++)
  {
    x=0;
    for(j=0;j<HNUM;j++)
      x+=(output_layer[k].weight[j]*hiden_layer[j].output);
    output_layer[k].output=activation_function(x);
  }
}
```

Do obliczenia błędu sieci wykorzystano równanie zawierające pochodną sigmoidalnej funkcji aktywacji określoną równaniem 8. Błąd liczony jest za pomocą tego równania niezależnie od zastosowanej funkcji aktywacji. W przypadku funkcji innych niż sigmoidalna stanowi przybliżenie wartości optymalnej.

Obliczone wartości błędu ulegają propagacji do warstwy ukrytej zgodnie z równaniem 10 i do warstwy wejściowej zgodnie z równaniem 11.

```
void back_propagate_errors()
{
  int i,j,k;
  /* calculation of output layer local errors */
```

```
  for(k=0;k<ONUM;k++)
  {
    /* back propagation error calculation */
    output_layer[k].error=output_layer[k].output
      *(1.0-output_layer[k].output)
      *(output_layer[k].required_output_scaled-output_layer[k].output);
  }
  /* error back propagation */
  for(j=0;j<HNUM;j++)
  {
    double s=0;
    for(k=0;k<ONUM;k++)
      s+=(output_layer[k].error*output_layer[k].weight[j]);
    hiden_layer[j].error=hiden_layer[j].output*
      (1.0-hiden_layer[j].output)*s;
  }
  for(i=0;i<INUM;i++)
  {
    double s=0;
    for(j=0;j<HNUM;j++)
      s+=(hiden_layer[j].error*hiden_layer[j].weight[i]);
    input_layer[i].error=input_layer[i].output*
      (1.0-input_layer[i].output)*s;
  }
}
```

Obliczone wartości błędów zmieniają stan sieci wprowadzając zmian poprawki do wag. Nowe wagi są liczone według równań 12 – 15 z uwzględnieniem wartości *momentum* liczonej zgodnie z równaniem 17.

```
void forward_propagate_weights()
{
  int i,j,k;
  double dw;
  for(j=0;j<HNUM;j++)
    for(i=0;i<INUM;i++)
    {
      dw=LCOEF*hiden_layer[j].error*input_layer[i].output;
      dw+=MOMENTUM*hiden_layer[j].dweight[i];
      hiden_layer[j].dweight[i]=dw;
      hiden_layer[j].weight[i]+=dw;
    }
  for(k=0;k<ONUM;k++)
    for(j=0;j<HNUM;j++)
    {
      dw=LCOEF*output_layer[k].error*hiden_layer[j].output;
      dw+=MOMENTUM*output_layer[k].dweight[j];
      output_layer[k].dweight[j]=dw;
      output_layer[k].weight[j]+=dw;
    }
}
```

Wykonanie kroku obliczeniowego podczas nauczania sieci polega na przypisaniu wejściom i wyjściom sieci odpowiednich wartości oraz wywołaniu kolejnych funkcji: skalującej, przenoszącej dane wprzód, obliczającej i przenoszącej błędy wstecz oraz modyfikującej stan sieci.

```
void bp_step()
{
  scale_values();
  forward_propagation();
  back_propagate_errors();
  forward_propagate_weights();
}
```

Funkcje `set_input`, `get_input`, `set_output`, `get_output`, `get_required_output` (zwrócenie wartości rzeczywistej) pozwalają na przypisywanie i odczytywanie wartości wejść i wyjść sieci, przy czym wejścia i wyjścia sieci liczone są od 0.

```
void set_input(int input,double value)
{
  input_layer[input].input=value;
}

double get_input(int input)
{
  return(input_layer[input].input);
}

void set_output(int output,double value)
{
  output_layer[output].required_output=value;
}

double get_output(int output)
{
  double d;
  d=output_layer[output].output-0.5;
  d/=output_layer[output].output_gain;
  d-=output_layer[output].output_offset;
  return(d);
}

double get_required_output(int output)
{
  return(output_layer[output].required_output);
}
```

5. Przykład użycia sieci neuronowej

W przedstawionym przykładzie użyto wartości wyznaczonych podczas badania zależności szybkości korozji stali węglowej St3SX, w roztworze chlorku sodowego NaCl mieszanym za pomocą mieszadła magnetycznego, od stężenia i temperatury. Badania przeprowadzono w komorze klimatycznej. Każde badanie trwało 24 godziny. Szybkość korozji określano metodą grawimetryczną (wagową), a wyniki przedstawiono w jednostce korozji liniowej – tablica 1.

Tablica 1. Doświadczalnie wyznaczone wartości szybkości korozji stali St3SX od stężenia chlorku sodowego i temperatury

Stężenie NaCl [% masy]	Temperatura [°C]	Szybkość korozji [mm/rok]
0,5	9,5	12,6432
1	9,5	14,8899
2	9,5	17,7093
4	9,5	16,8722
0,5	21,5	21,1017
1	21,5	24,1972
2	21,5	21,7993
4	21,5	21,7121
0,5	30,2	30,2203
1	30,2	28,9427
2	30,2	38,4141
4	30,2	32,2907
0,5	36,5	44,4934
1	36,5	37,3568
2	36,5	37,3128
4	36,5	34,7577

Wartości stężenia, temperatury i szybkości korozji zawarto w tablicach. Stała `N` określa liczbę zestawów danych.

```
#define N 16

  double conc[N]={0.5, 1, 2, 4,
                  0.5, 1, 2, 4,
                  0.5, 1, 2, 4,
                  0.5, 1, 2, 4};
  double temp[N]={9.5, 9.5, 9.5, 9.5,
                  21.5,  21.5,  21.5,  21.5,
                  30.2, 30.2, 30.2, 30.2,
                  36.5, 36.5, 36.5, 36.5};
  double vcor[N]={12.6432, 14.8899, 17.7093, 16.8722,
                  21.1017, 24.1972, 21.7993, 21.7121,
                  30.2203, 28.9427, 38.4141, 32.2907,
                  44.4934, 37.3568, 37.3128, 34.7577};
```

W głównej pętli programu zadeklarowano zmienne pomocnicze.

```
int main(int argc,char *argv[])
{
  int i,j;
```

Gdy jest możliwe odczytanie stanu, sieć kontynuuje przerwaną wcześniej pracę. W innym przypadku wymagane jest przypisanie wagom sieci wartości losowych.

```
  if(read_nn())
    randomize_weights();
```

W każdym przypadku należy wyzerować zmienne odpowiedzialne za modyfikację wartości wag.

```
  reset_dweights();
```

Ze względu na ściśle określony zakres wartości, w jakim możliwa jest poprawna praca sieci neuronowej, konieczne jest odpowiednie skalowanie zakresów dla danych wejściowych i wyjściowych. Każde wejście i wyjście sieci należy wyskalować oddzielnie. Odpowiednie funkcje przeszukują wartości w tablicach i dokonują odpowiednich obliczeń. Nic jednak nie stoi na przeszkodzie, aby dobrać wartości użyte do skalowania w sposób arbitralny. Najłatwiej jest w tym celu stworzyć dwuelementową tablicę zawierającą wartość minimalną i maksymalną oraz przekazać tę tablicę do odpowiedniej funkcji `prescale_`. Odpowiednie dobranie zakresów ma istotny wpływ na kształt uzyskanego modelu.

```
  prescale_input(0,conc,N);
  prescale_input(1,temp,N);
  prescale_output(0,vcor,N);
```

Główna pętla, w której dokonywane są obliczenia, może trwać określoną liczbę cykli. Może też to być pętla, którą program opuści po spełnieniu określonego warunku. W praktyce często jest stosowana pętla o nieskończonym trwaniu, gdzie o zakończeniu obliczeń decyduje człowiek.

```
  for(j=0;j<1000000;j++)
  {
```

Zestawy danych można pobierać w kolejności, w jakiej zostały wpisane do programu. W bardziej złożonych przypadkach lepszym rozwiązaniem jest pobieranie zestawów danych w kolejności losowej.

Poszczególne dane przypisywane są odpowiednim wejściom i wyjściom sztucznej sieci neuronowej.

```
for(i=0;i<N;i++)
{
  set_input(0,conc[i]);
  set_input(1,temp[i]);
  set_output(0,vcor[i]);
```

Wywołanie funkcji `bp_step()` uruchamia sekwencję obliczeniową skutkującą – jeżeli jest to możliwe – bardziej optymalnym dopasowaniem modelu do zestawu danych.

```
  bp_step();
```

Postępy procesu obliczeniowego wygodnie jest co pewien czas obserwować wyświetlając zestawy danych i porównując obliczone wartości z danymi. Zmiany wartości błędu globalnego sieci obliczone dla zestawów danych pozwalają na ocenę zbieżności obliczeń.

```
  if(!(j%1000))
  {
    fprintf(stderr,"%1.0f, %1.1f, %1.6f, %1.6f, ",
            get_input(0),
            get_input(1),
            get_required_output(0),
            get_output(0));
    forward_propagation();
    fprintf(stderr,"%6g\n",global_error());
  }
}
```

Okresowo należy zapisywać stan sieci oraz wywoływać funkcję zapisującą model w postaci dogodnej do dalszych obliczeń za pomocą innych programów. W tym miejscu często umieszcza się też funkcję `system(...)` w celu wywołania programów zewnętrznych do przetwarzania i wizualizacji.

```
  if(!(j%10000))
  {
    write_nn();
    output();
  }
}
```

Przed zakończeniem pracy program przesyła do systemu operacyjnego kod zakończenia.

```
  return(0);
}
```

Funkcja `output()` wywoływana cyklicznie nie została wcześniej przedstawiona. Jej działanie jest zależne od wejściowego formatu programu stosowanego do wizualizacji. Przykładowo funkcja ta może zapisywać dane w postaci zestawów wartości dla określonej przestrzeni może mieć następującą postać:

```
void output()
{
  double c,t;
  FILE * f;
  int n;

  if(!(f=fopen("nn.out","w+")))
    return;

  for(c=0.5;c<=4;c+=0.1)
  {
    n=0;
    for(t=5;t<=45;t+=1)
    {
      set_input(0,c);
      set_input(1,t);
      scale_values();
      forward_propagation();
      fprintf(f,"%g %g %g\n",t,c,get_output(0));
      n++;
    }
  }
  if(fclose(f))
    return;
}
```

Kod programu należy skompilować w systemie Unixowym za pomocą kompilatora `gcc` stosując polecenie:

```
gcc -ansi -pedantic -Wall -lm -O2 nn.c -o nn
```

6. Model zależności

Szybkość dopasowania modelu tworzonego przez sztuczną sieć neuronową do danych jest zależna od wielu czynników. Największy wpływ na ten proces ma liczba zestawów danych i jednoznaczność obserwowanych zależności. Wartość stałej nauczania wpływa nie tylko na szybkość uczenia, ale też na umiejętność pomijania minimów lokalnych. Inaczej, niż w przypadku klasycznych metod aproksymacji, wielość zmiennych zwykle nie wpływa ograniczająco na jakość pracy sieci, lecz wręcz przeciwnie – ułatwia osiągnięcie rozwiązania globalnego.

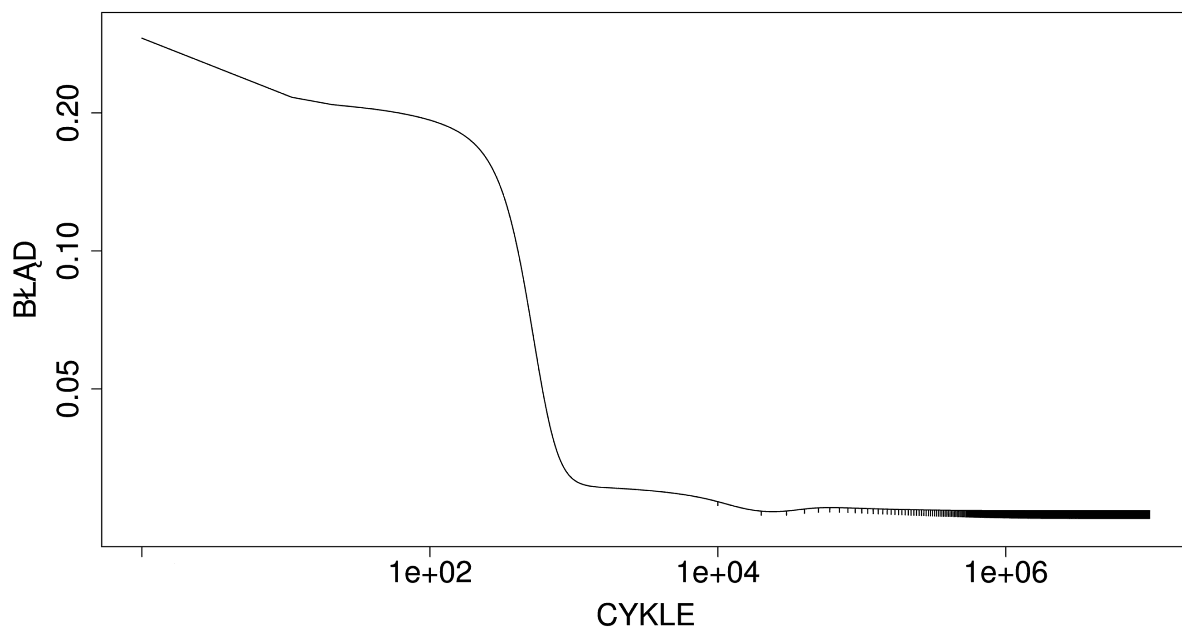
Przebieg poszukiwania rozwiązania przedstawiono na rys. 9.

Osiągnięcie odpowiedniego dopasowania modelu do danych wymaga przeprowadzenia odpowiedniej liczby cykli obliczeniowych – rys. 10.

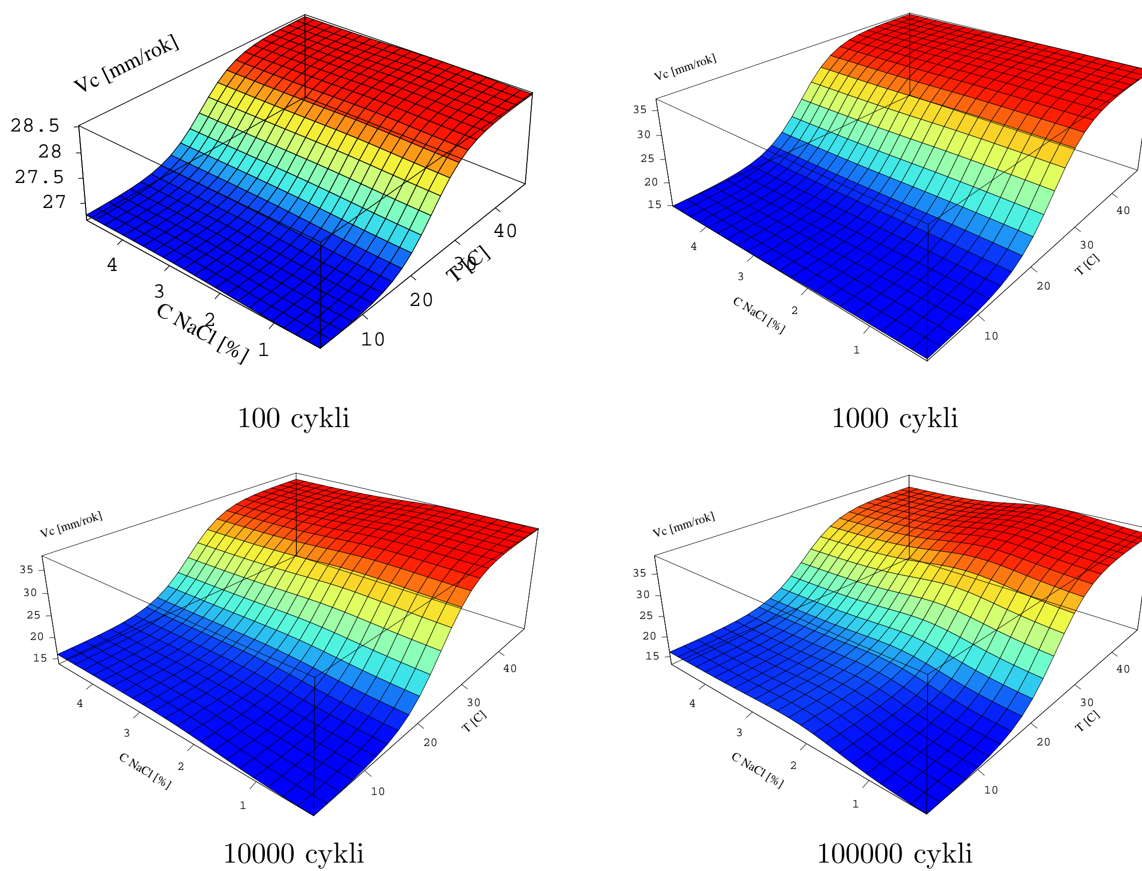
Model zależności szybkości korozji od stężenia chlorku sodowego i temperatury (rys. 11) stanowi zależność generalną i oprócz tego, że dostarcza ogólny obraz zależności, pozwala na oszacowanie szybkości korozji dla określonych warunków. Wyznaczenia wartości liczbowej dla wybranych warunków można dokonać w sposób przedstawiony w funkcji `output()`.

7. Prawa autorskie

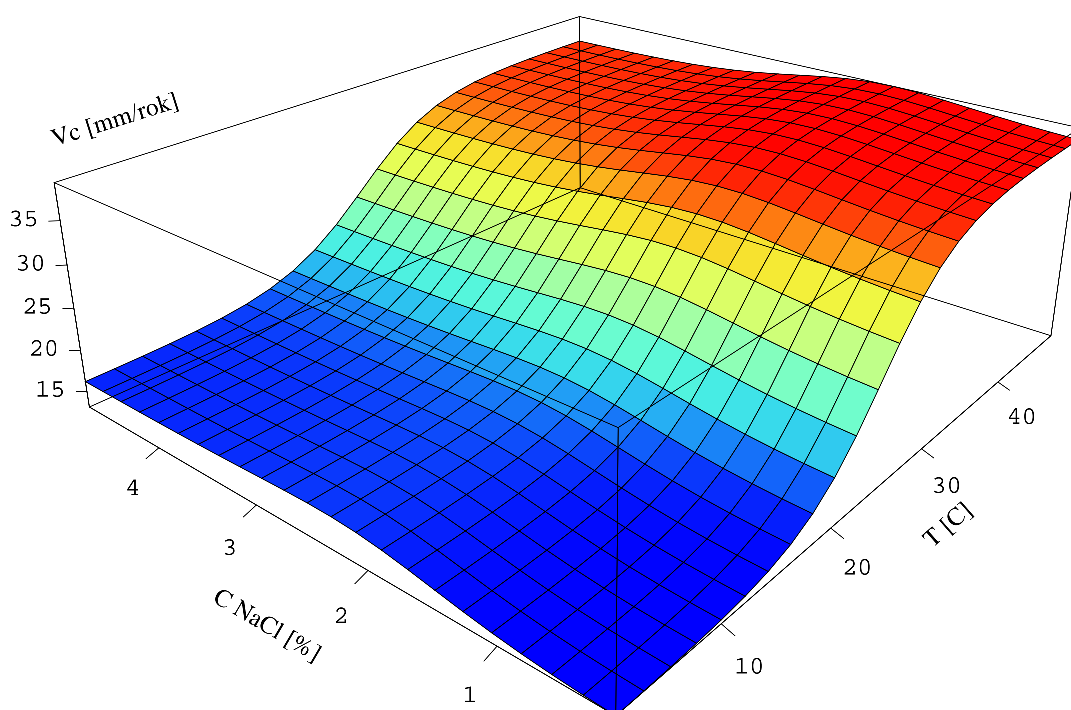
Kod programu przedstawionego w niniejszej pracy może być wykorzystywany zarówno do celów niekomercyjnych, jak i komercyjnych zgodnie z licencją GNU GPL.



Rys. 9. Zmiana błędu dopasowania modelu w funkcji liczby cykli obliczeniowych (wykres logarytmiczny)



Rys. 10. Zależność modelu od liczby cykli obliczeniowych



Rys. 11. Zależność szybkości korozji od stężenia chlorku sodowego i temperatury roztworu

8. Literatura

[1] Smith S. W.: *The Scientist and Engineer's Guide to Digital Signal Processing. 2nd edn.* California Technical Publishing, San Diego, USA, 1999.

[2] Papik K., Molnar B., Schaefer R., Dombowari Z., Tulassay Z., Feher J.: Application of neural networks in medicine – a review. *Medical Science Monitor, Diagnostics and Medical Technology*, 1998, 4(3):538–546.

[3] Otair M. A., Salameh W. A.: Speeding Up Back–Propagation Neural Networks. *Proceedings of the 2005 Informing Science and IT Education Joint Conference, Flagstaff, Arizona, USA*, June 2005, 168–173.