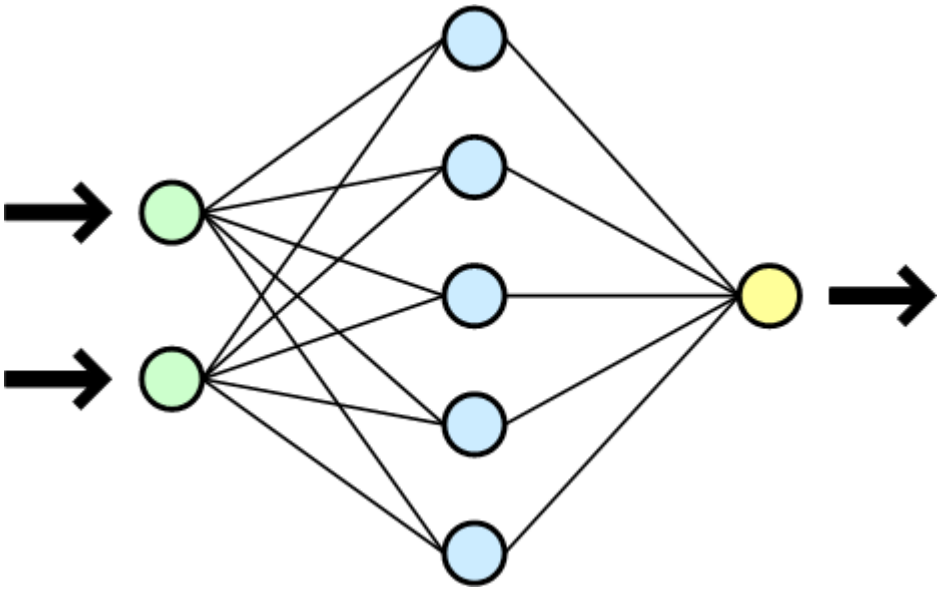
**Oznaczenia:**
SSN – Sztuczna Sieć Neuronowa
Ten kolor oznacza informacje dotyczące konkretnej implementacji SSN (dany projekt).

**PROGRAM ROZWIĄZUJE NASTEPUJĄCY PROBLEM**
Rozpoznawanie kodów pocztowych z podanego obrazu w pliku BMP w 256-kolorach (szarości lub zwykłych). Obraz z bitmapy może być różnych rozmiarów (maksymalnie 380x160).
Rozpoznawanie poszczególnych obrazów cyfr odbywa się za pomocą SSN.

**SZTUCZNA SIEĆ NEURONOWA**
Wynikowy obraz cyfry podawany na wejście SSN jest rozmiaru 16x16 pikseli o wartościach kolorów 0 – 255. Kolory są 'zinwersowane', tzn. wartości bliskie 0 reprezentują tło, a wartości bliskie 255 reprezentują znak. Wartości jasności pikseli są przeskalowane do przedziału [0.0 – 1.0]. Mamy zatem16*16 = 256 neuronów w warstwie wejściowej. Warstwa wyjściowa zawiera 10 neuronów – po jednym dla każdej cyfry.



Uproszczony schemat jednokierunkowej sieci neuronowej

**Sieci jednokierunkowe** to sieci neuronowe, w których nie występuje sprzężenie zwrotne, czyli pojedynczy wzorzec lub sygnał przechodzi przez każdy neuron dokładnie raz w swoim cyklu.
Najprostszą siecią neuronową jest pojedynczy **perceptron progowy**, opracowany przez McCullocha i Pittsa w roku 1943.
W programie uczącym SSN zaimplementowano **czterowarstwową nieliniową sztuczną sieć neuronową jednokierunkową** (wielowarstwowa sieć perceptronowa (ang. Multi-Layered Perceptron **MLP**)).
Zgodnie z teorią sieć o dwóch lub trzech warstwach perceptronowych może być użyta do aproksymacji większości funkcji.

Kolejne warstwy sieci, to: warstwa danych wejściowych, **dwie warstwy ukryte** (wewnętrzne) i warstwa danych wyjściowych. Użycie dwóch warstw ukrytych zamiast tylko jednej znacząco poprawia uczenie się SSN i jej możliwości generalizacji. Trzy warstwy to już przesada, a dwie w zupełności wystarczają do poprawnej nauki.
Połączenia neuronów między warstwami - każdy z każdym.
Wektorem uczącym są następujące dwa ciągi danych: uczący **(wejściowy) i weryfikujący (wyjściowy)**.

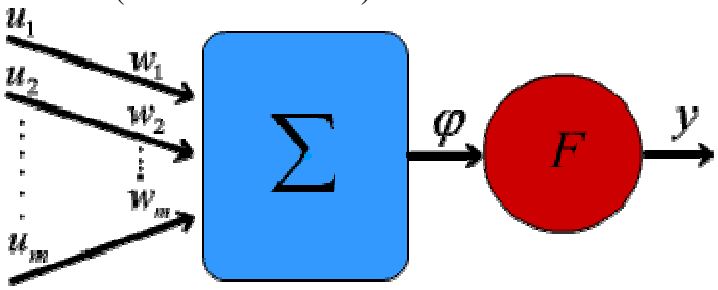
**Wektorem wejściowym** jest 256 liczb rzeczywistych, o wartościach z przedziału [0.0 ; 1.0].
**Wektorem wyjściowym** jest 10 liczb. W przypadku obrazu cyfry, 9 z nich ma wartość równą 0.0, a jedna, której numer reprezentuje daną cyfrę ma wartość równą 1.0. Jeżeli obraz nie jest cyfrą, to wszystkie neurony mają wartość 0.0.
Przykładowe wektory wyjściowe:

- [**1**,0,0,0,0,0,0,0,0,0] oznacza odpowiedź „cyfra 0”
- [0,0,0,0,0,0,0,0,0,**1**] oznacza odpowiedź „cyfra 9”
- [0,0,0,0,0,0,0,0,0,0] oznacza odpowiedź „nierozpoznany znak”

**Warstwy ukryte** (wewnętrzne) zawierają po około 60 neuronów – w tym wypadku sieć wydaje się uczyć najszybciej.
Liczbę neuronów w każdej warstwie ukrytej można ustalać w programie uczącym SSN.
Dla wielu warstw ukrytych stosuje się zwykle **metodę piramidy geometrycznej**, która zakłada, że liczba neuronów w kolejnych warstwach tworzy kształt piramidy i maleje od wejścia do wyjścia. Zatem liczba neuronów w kolejnych warstwach powinna być coraz mniejsza (większa liczba neuronów w kolejnej warstwie nie zwiększa możliwości nauki SSN).
Zatem SSN może ostatecznie mieć rozmiar np.: 256 - 70 - 50 - 10.
Generalnie jednak uczenie rozpoczyna się z małą liczbą neuronów w warstwach ukrytych a następnie, obserwując postępy tego procesu, doświadczalnie zwiększa się ich liczbę.

**Funkcją aktywacji** neuronu jest funkcja sigmoidalna unipolarna.
Wartość funkcji aktywacji jest sygnałem wyjściowym neuronu i propagowana jest do neuronów warstwy następnej. Funkcja aktywacji może przybierać jedną z trzech postaci:
- nieliniowa
- liniowa
- skoku jednostkowego tzw. funkcja progowa

Argumentem funkcji aktywacji neuronu są zsumowane iloczyny sygnałów wejściowych i wag tego neuronu (tzw. suma ważona).



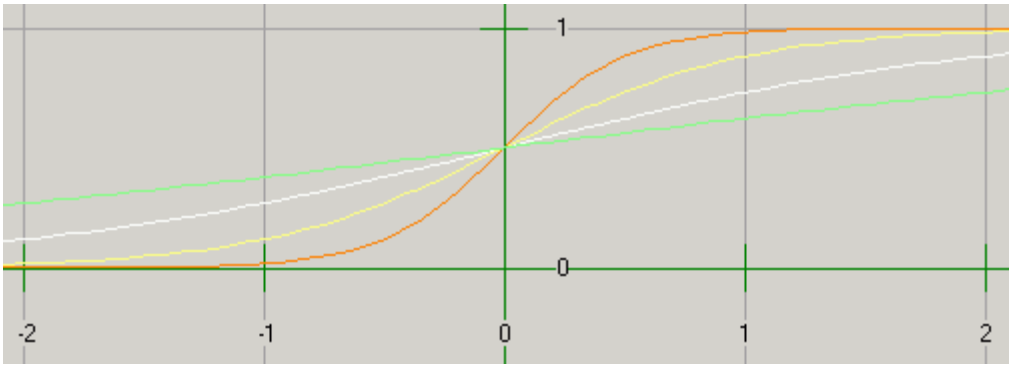

Wybór funkcji aktywacji zależy od rodzaju problemu, jaki stawiamy przed siecią do rozwiązania. Dla sieci wielowarstwowych najczęściej stosowane są **funkcje nieliniowe**, gdyż neurony o takich charakterystykach wykazują największe zdolności do nauki, polegające na możliwości odwzorowania w sposób płynny dowolnej zależności pomiędzy wejściem a wyjściem sieci. Umożliwia to otrzymanie na wyjściu sieci informacji ciągłej a nie tylko postaci: TAK - NIE.

Wymagane cechy funkcji aktywacji to:
- ciągłe przejście pomiędzy swoją wartością maksymalną a minimalną (np. 0.0 - 1.0),
- łatwa do obliczenia i ciągła pochodna,
- możliwość wprowadzenia do argumentu parametru beta do ustalania kształtu krzywej.

Użytą funkcją aktywacji neuronu jest funkcja **sigmoidalna unipolarna:**

$$f(x,beta) = \frac{1.0}{1.0 + e^{-beta*x}} \in [0.0\ ;\ 1.0]$$; gdzie $x = u_1*w_1+\ldots+ u_m*w_m$

Funkcja ta charakteryzuje się tym, że wartość jej pochodnej można obliczyć z obliczonej jej wartości:
**f’(x,beta)** = beta*F*(1.0-F); gdzie F = f(x,beta).



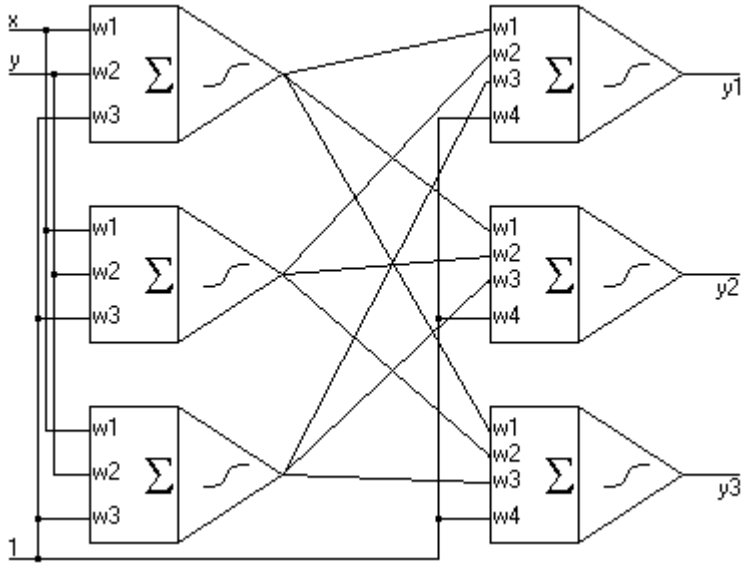

(białe) beta = 1.0;
beta = 2.0;
beta = 4.0;
beta = 0.5;

Funkcja sigmoidalna unipolarna $\epsilon$ [0.0 ; 1.0]

Każda warstwa sieci ma własny współczynnik **beta** dla funkcji aktywacji (można go ustalać w programie uczącym SSN; np. na kolejno: 1.0; 1.0; 1.0).
Jest to współczynnik determinujący nachylenie sigmoidalnej funkcji aktywacji.

Uwaga. Wpływ parametru beta może być zastąpiony odpowiednim doborem innych parametrów i wielkości. W przypadku przetwarzania zmiennych wejściowych na zmienne wyjściowe, efekt jaki powoduje zmiana nachylenia krzywej można uzyskać mnożąc wartości wag sieci przez wielkość parametru beta.

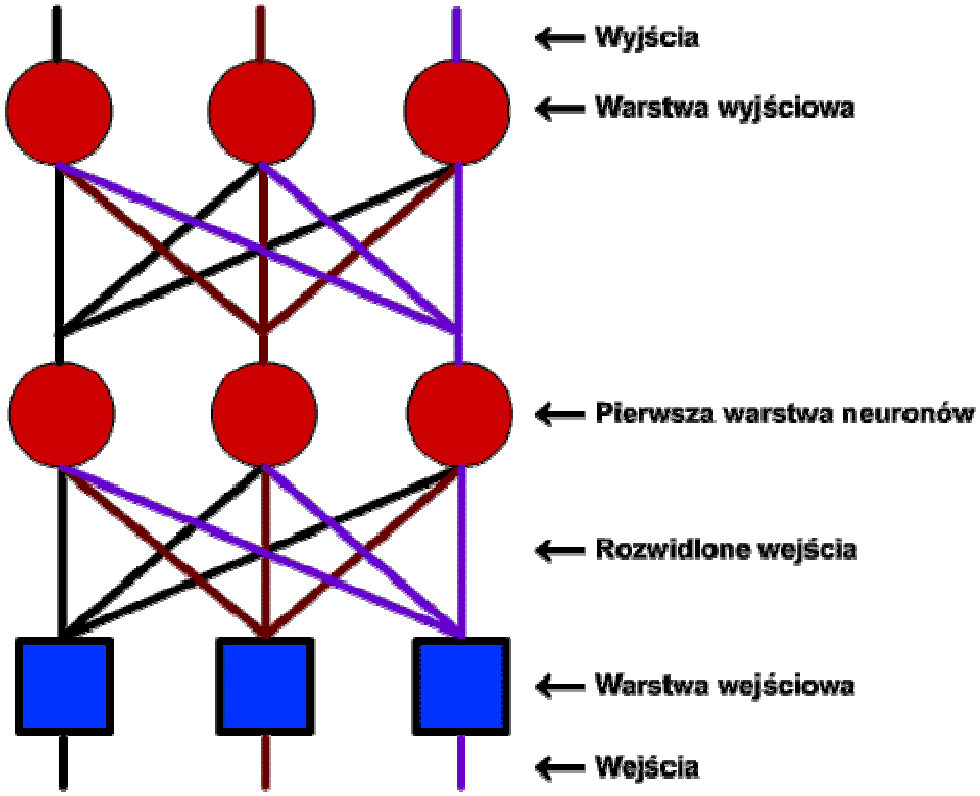
Przykładowa wielowarstwowa sieć neuronowa wygląda następująco:




Po obliczeniu wyniku SSN, neurony na wyjściu mają wartość z przedziału [0.0; 1.0].

Numer neuronu o największej wartości, o ile ta wartość jest > 0.5 oznacza numer cyfry. Gdy wszystkie neurony mają wartości < 0.5 (w tym i największa), to oznacza odpowiedź „nie rozpoznany”.

**UCZENIE SSN**
Działanie SSN polega na tym, że sygnały pobudzające zawarte w wektorze wejściowym podawane na wejścia sieci, przetwarzane są w poszczególnych neuronach. Po tej projekcji na wyjściach sieci otrzymuje się wartości liczbowe, które stanowią odpowiedź sieci na pobudzenie i stanowią rozwiązanie postawionego problemu.
Sieć uczymy „na przykładach”.
Celem jest uzyskanie oczekiwanych zadawalających (tzn. nie przekraczających dopuszczalnego błędu) wartości na wyjściu przy danych wartościach wejściowych.
Gdy błąd jest zbyt duży należy skorygować odpowiednio wagi.




Jednak, aby takie rozwiązanie uzyskać, należy przejść żmudną drogę uczenia sieci. Jedną z metod jest **uczenie metodą wstecznej propagacji błędów – EBP** (ang. Error Back Propagation)**.** Jest to podstawowa (i jedna z najskuteczniejszych) metoda uczenia sieci wielowarstwowych. Została opracowana w 1974 roku przez P. J. Werbosa jako uczenie z nadzorem lub inaczej - z nauczycielem. Taką też metodę zastosowałem w programie.
Algorytm opiera się na sprowadzeniu wartości funkcji błędu sieci poniżej pewnego założonego minimum. Jako miarę błędu stosuje się błąd średniokwadratowy na neuronach wyjściowych (E).
Poprawianie wag odbywa się **po prezentacji każdego wzorca**, o ile błąd dla tego wzorca jest za duży (parametr epsilon).
Nie jest natomiast użyta tutaj inna metoda, tzw. reguła skumulowanej delty, czyli poprawianie wag po prezentacji wszystkich wzorców, (czyli po całej epoce) – tzw. BATCH propagation.

**OD STRONY MATEMATYCZNEJ**
Proces uczenia SSN to **minimalizacja funkcji błędu**, której dokonujemy za pomocą gradientu.
Kierunek największego spadku dowolnej funkcji wskazuje ujemny gradient tej funkcji.
Dlatego wektor wag neuronu jest przesuwany w kierunku ujemnego gradientu. Dodatkowo wektor ten jest przemnażany przez **współczynnik eta**, który nazywany jest **współczynnikiem szybkości uczenia** (współczynnik korekcji).
**Gradient F** - wektor utworzony z pochodnych funkcji F po wszystkich jej zmiennych.

Załóżmy, że mamy SSN złożoną z 3 warstw:
- wejściowa: **U**: u[0] ÷ u[m-1], to dane wejściowe, a u[m] – to stała wartość = 1.0 – tzw. bias
- jedna ukryta: **X**: x[0] ÷ x[n-1] oraz stałe x[n] = 1.0 – tzw. bias
- wyjściowa: **Y**: y[0] ÷ y[r-1]

Rozmiary sieci:
**m** – liczba danych wejściowych
**n** – liczba neuronów w warstwie ukrytej
**r** – liczba neuronów w warstwie wyjściowej

**U, X, Y** – wektory danych
Mamy zatem tutaj dwie warstwy wag:

- między warstwą wejściową i ukrytą: **w[i,j]**; i – nr neuronu warstwy ukrytej; j – nr danej wejściowej; czyli W[i]: W[0] ÷ W[m]
- między warstwą ukrytą i wyjściową: **s[i,j]**; i – nr neuronu warstwy wyjściowej; j – nr neuronu warstwy ukrytej; czyli S[i]: S[0] ÷ S[n]
**W[i], S[i]** – wektory wag w i s

Mamy osobne funkcje aktywacji dla każdej warstwy sieci:
$\mathbf{f^{Uk}(x)}$ = f(x, BetaUk) – funkcja aktywacji w warstwie ukrytej
$\mathbf{f^{Wy}(x)}$ = f(x, BetaWy) – funkcja aktywacji w warstwie wyjściowej

Sygnały w SSN płyną od wejścia do wyjścia, czyli **wynik SSN** jest obliczany kolejno:
Najpierw:
$x[i] = f^{Uk}(U*W[i]) = f^{Uk}(u[0]*w[i,0] + u[1]*w[i,1] + ...+ u[m-1]*w[i,m-1] + 1*w[i,m])$
(x[n] = 1)
czyli wektor $X = f^{Uk}(U*W)$
Potem:
$y[i] = f^{Wy}(X*S[i]) = f^{Wy}(x[0]*s[i,0] + x[1]*s[i,1] + ...+ x[n-1]*s[i,n-1] + 1*s[i,n])$
czyli wektor $Y = f^{Wy}(X*S)$

Niech **P** – zbiór wszystkich wzorców; $\mathbf{Z^{(p)}}$ – oczekiwany wynik dla danego wzorca p € P
Dla ustalonego p niech **z** = $z^{(p)}$, **x** = $x^{(p)}$, **y** = $y^{(p)}$.

Nasz cel: Dla każdego wzorca p ϵ P minimalizacja wartości: $\sum_{k=0}^{r-1}(y[k] - z[k])^2 = E$

$E = \sum_{k=0}^{r-1}(f^{Wy}(X*S[k]) - z[k])^2$ , czyli
E = F(S[0], S[1], …, S[r-1], X, Z) – funkcja r zmiennych S[i] oraz X i Z

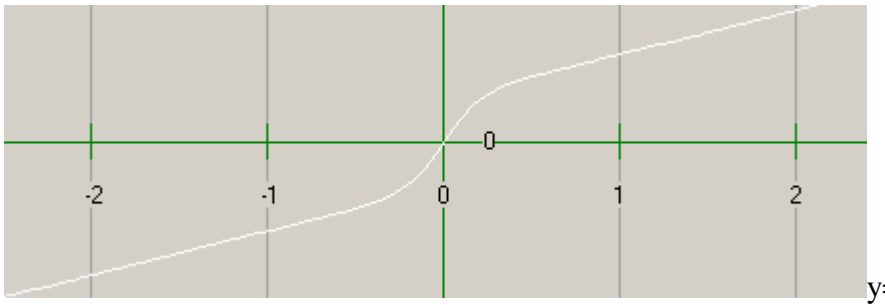
Chcemy znaleźć minimum funkcji F, czyli odpowiednich wartości zmiennych S[i].
Używamy **metody gradientu**:
**Gradient** $F = [\frac{\partial F}{\partial S[0]}, \frac{\partial F}{\partial S[1]}, ..., \frac{\partial F}{\partial S[r-1]}]$ t.j. wektor pochodnych F po wszystkich zmiennych S[i]
Oznaczmy potencjał wejściowy neuronu warstwy wyjściowej i jako $\mathbf{PotWej^{Wy}[i]}$ = X*S[i]
Zatem:
$$\frac{\partial F}{\partial S[0]} = 2 * \sum_{k=0}^{r-1}(f^{Wy}(X * S[k]) - z[k])^1 * \frac{\partial f^{Wy}(X * S[k] - z[k])}{\partial S[0]} =$$
$= 2 * (f^{Wy}(X*S[0]) - z[k]) * f^{Wy}{}'(X*S[0]) * X =$
(bo dla k ≠ 0 $\partial f^{Wy}(X*S[k] - z[k])/\partial S[0] = 0$)
$= 2 * (y[0] - z[0]) * f^{Wy}{}'(X*S[0]) * X$
$= 2 * (y[0] - z[0]) * f^{Wy}{}'(PotWej^{Wy}[0]) * X.$
$\mathbf{f^{Wy}{}'(x)}$ – pochodna funkcji $f^{Wy}(x)$

Zatem w celu minimalizacji F zmieniamy argumenty z szybkością eta:
𝕊 := 𝕊 – eta * Gradient F, czyli
$S[i] := S[i] - eta * \frac{\partial F}{\partial S[i]}$, czyli np. dla neuronu nr 0 i wagi nr i mamy:
$s[0,i] := s[0,i] - eta * 2 * (y[0] - z[0]) * f^{Wy}{}'(PotWej^{Wy}[0]) * x[i] =$
$s[0,i] := s[0,i] + eta * 2 * (z[0] - y[0]) * f^{Wy}{}'(PotWej^{Wy}[0]) * x[i].$
W celu przyspieszenia obliczeń, można dla danego neuronu k obliczyć na początku stałą wartość (mała delta):
**δ(k)** = $2 * (z[k] - y[k]) * f^{Wy}{}'(PotWej^{Wy}[k])$, jest to **wartość błędu neuronu k**, wtedy
$s[k,i] := s[k,i] + eta * δ(k) * x[i]$

Zatem aktualizacja wag neuronów odbywa się zgodnie ze wzorem (to wersja bez członu momentum):
s[k,i] = s[k,i] + Δs[k,i],
gdzie Δs[k,i] = eta * δ(k) * x[i] – dla warstwy wyjściowej

Aby zmiany były nieco łagodniejsze można Δs[k,i] nieco zmienić tzn. sugerować się bardziej znakiem gradientu (podobnie jak w metodzie **Resilient BackPropagation** RPROP) np.

Δs[k,i] = eta * sign(δ(k) * x[i]), gdzie np. $sign(x) = \frac{0.5}{1+e^{-10x}} - \frac{1}{4} + \frac{x}{4}$



y=sign(x)

Minimalizacja funkcji F powinna odbywać się także poprzez zmianę X, czyli poprzez zmianę wag W[i]: W[0] ÷ W[n-1].

Okazuje się, że dla dowolnej warstwy (L) δ(k) = sumie ważonej δ (delt) z warstwy następnej (L+1) razy pochodna funkcji aktywacji, tzn.:
W naszym przypadku dla warstwy ukrytej i wyjściowej:

$$\delta^{Uk}(i) = \sum_{j=0}^{r-1} (\delta^{Wy}(j) * s[j,i]) * f^{Uk\,\prime}(PotWej^{Uk}[i]) \; ; \; \delta^{Uk}(0) \div \delta^{Uk}(n-1)$$

Ogólnie dla warstw L i L+1:

$$\boldsymbol{\delta^{(L)}(i)} = \sum_{j=0}^{r(L+1)-1} (\delta^{(L+1)}(j) * w^{(L+1)}[j,i]) * f^{(L)\prime}(PotWej^{(L)}[i])$$

**r(L+1)** – liczna neuronów w warstwie L+1
**$w^{(L+1)}$[j,i]** – waga między neuronem j w warstwie L+1 a neuronem i w warstwie L

Kod C++ poprawy wag:

```
for (i=0; i<LNeuronowUk2; ++i) //po wsz. neuronach z warstwy Uk2; licz d(i) dla NeuronUk2
 {E = 0.0;  for (j=0; j<LNeuronowWy; ++j) E += DaneWy[j] * WagaWy[j][i]; //suma ważona d[] z warstwy następnej
  DaneUk2[i] = FunP(PotWejUk2[i], BetaUk2) * E; //d(i) i-ty neuron
  //--Korygowanie wag nauronu [i] warstwy Uk2 (dopiero teraz można!, bo WagaUk2[k][i] jest wcześniej czytana)
  for (j=0; j<LNeuronowUk1; ++j) //po wsz. wagach neuronu z Uk2
   {delta = etaUk2 * sign(DaneUk2[i] * DaneUk1[j]) + alfaUk2 * WagaUk2S[i][j];
    WagaUk2[i][j] += delta; //zmiana wag
   }
  delta = etaUk2 * sign(DaneUk2[i]) + alfaUk2 * WagaUk2S[i][LNeuronowUk1];
  WagaUk2[i][LNeuronowUk1] += delta; //oraz Bias <małe znaczenie>
 }
```

**PROCES UCZENIA SSN**
Zastosowany sposób uczenia SSN wygląda w ogólności następująco:
Pętla po wszystkich wzorcach (tzw. **epoka**):
- Oblicz błąd dla danego wzorca.
- Jeżeli błąd ten jest większy niż parametr epsilon, to popraw wagi **jeden RAZ!**,
(czyli nie poprawiaj wag, aż nie będzie błędu dla tego wzorca (tzw. **cykl**) i dopiero badaj następny).
- Sprawdź następny wzorzec.

Powtarzaj tę pętlę, aż nie będzie błędu większego niż '**epsilon**' dla wszystkich wzorców (w całej epoce) oraz suma tych błędów nie będzie większa od parametru '**Suma epsilon**'.

(Zatem tutaj cykl składa się z jednej poprawy wag. Taki sposób uczenia wydaje się w praktyce szybszy, niż stosowanie pełnych cykli.)

**Błąd sieci** to suma po wszystkich neuronach wyjściowych kwadratów różnicy między obliczonym wynikiem sieci **y[]** (dla podanych danych wejściowych – wzorzec p) a oczekiwanym wynikiem **z[]** (dla tych danych) czyli:

$$E=\sum_{k=0}^{r-1}(y[k]-z[k])^2$$

Np. Dla pewnych danych wejściowych i dwóch neuronów w warstwie wyjściowej sieć daje na wyjściu wartości 0.4 i 0.9, a powinna dawać wartości odpowiednio: 0.0 i 1.0. Zatem błąd dla tego wzorca wynosi $(0.4-0.0)^2+(0.9-1.0)^2 = 0.4^2 + 0.1^2 = 0.16 + 0.01 = 0.17$.

**PARAMETRY UCZENIA SSN**

Implementacja SSN została napisana od podstaw na podstawie własnej wiedzy i doświadczeń. Zawarta jest w plikach: KlasaSiecNmom2H.h/cpp

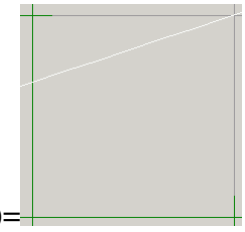
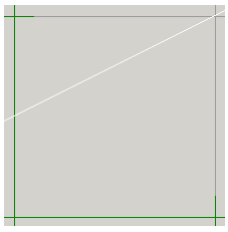
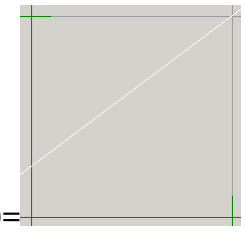
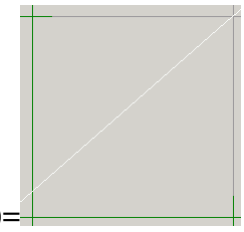
**Liczba wzorców** – to liczba obrazów znaków, których SSN ma się uczyć i na ich podstawie rozpoznawać inne obrazy.
Dodatkowo jako wzorce dla odpowiedzi „nie rozpoznano" – generowane są losowe obrazy (tzn. losowe kolory pikseli) oraz pewne szablonowe np. cały czarny – wszystkie wartości pikseli równe 255 lub cały biały – wszystkie wartości równe 0.

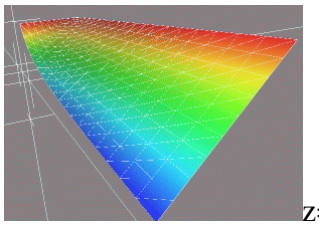
**eta** – współczynnik szybkości uczenia – określa jak bardzo w danym kroku zmieniane są wagi (kierunek uczenia sieci).
Zbyt mała jego wartość powoduje zwiększenie się liczby potrzebnych iteracji i tym samym czasu wymaganego na zakończenie treningu. Algorytm ma ponadto w takim przypadku tendencję to wygasania w minimach lokalnych funkcji celu. Ustalenie z kolei zbyt dużej wartości kroku grozi wystąpieniem oscylacji wokół poszukiwanego minimum, prowadzącej ostatecznie do zatrzymania procesu uczenia bez osiągniecie wymaganej wartości funkcji celu.
Zwiększenie tej wartości może zwiększać szybkość uczenia się SSN pojedynczych wzorców, ale może to powodować, że w znajdowaniu globalnego minimum funkcji, algorytm wpadnie w minimum lokalne i nie będzie możliwe nauczenie się wszystkich wzorców.
Mała wartość powoduje, że SSN uczy się małymi kroczkami, ale za to konsekwentnie dąży do celu i nierzadko przez to zwiększa się całościowa szybkość nauczenia się wszystkich wzorców.

W programie zastosowano dynamiczne ustalanie parametru eta proporcjonalnie od aktualnej liczby błędów sieci. Ponadto można ustalić wartości eta dla każdej warstwy osobno, przykładowo:
EtaUk1 = 0.3, A_EtaUk1 = 10, AlfaUk1 = 0.8,
EtaUk2 = 0.2, A_EtaUk2 = 8, AlfaUk2 = 0.8,
EtaWy = 0.1, A_EtaWy = 6, AlfaWy = 0.8,

**A_Eta** – określa jak bardzo ma maleć wartość Eta w zależności od liczby błędów w ostatniej epoce.
Im większa wartość A_Eta tym eta bardziej maleje, proporcjonalnie do liczby błędów. A_Eta musi być >= 1.0. Gdy A_Eta = 1.0, Eta nie zmienia się.
Dokładnie obliczanie końcowego parametru eta wygląda następująco:

eta = Eta * F(LBłędów/Wzorców, A_Eta), gdzie $F(x, n) = \frac{1+(n-1)*x}{n}$

F(x,1.5)=  F(x,2)=  F(x,4)=  F(x,8)= 

 z=F(x,n); 0<=x<=1; 1<=n<=6

**Alfa** – parametr określający wpływ **członu meomentum** dla każdej z warstw osobno.
Im większa jego wartość, tym większy wpływ na kierunek uczenia mają poprzednie wartości popraw wag sieci. Odpowiednio użyty może bardzo poprawić szybkość uczenia. Najczęściej przyjmuje się wartość 0.7 - 0.95, np. 0.88. Wartość równa 0.0 oznacza brak wpływu członu momentum na zmianę wag.

**epsilon** – maksymalny błąd SSN dla pojedynczego wzorca. Zwykle ustala się jego wartość na nie większą niż **0,5**.
Zmniejszanie tej wartości powoduje zwiększanie dokładności sieci i co za tym idzie precyzji predykcji lub generalizacji, ale też nieuchronnie wydłuża czas trwania procesu uczenia lub całkowicie uniemożliwia poprawną naukę SSN przy dużej liczbie wzorców do nauczenia.
Zwiększanie wartości tego parametru powoduje natomiast szybsze zakończenie uczenia sieci, ale tak nauczona sieć może dawać złe odpowiedzi nawet dla uczonych wzorców.

**Suma epsilon** – maksymalna suma błędów SSN dla wszystkich uczonych wzorców. Dodatkowy parametr określający jakość nauczonej SSN. Jeżeli nie interesuje nas ten parametr, to należy go ustawić na odpowiednio dużą wartość (> liczby wzorców).

**Maksymalna liczba popraw wag** – określa maksymalną liczbę kroków, tzn. popraw wag sieci, jaką można przeprowadzić w procesie uczenia, po to, aby proces mógł się zatrzymać, gdy SSN nie chce się nauczyć (, bo np. wpadła w minimum lokalne).

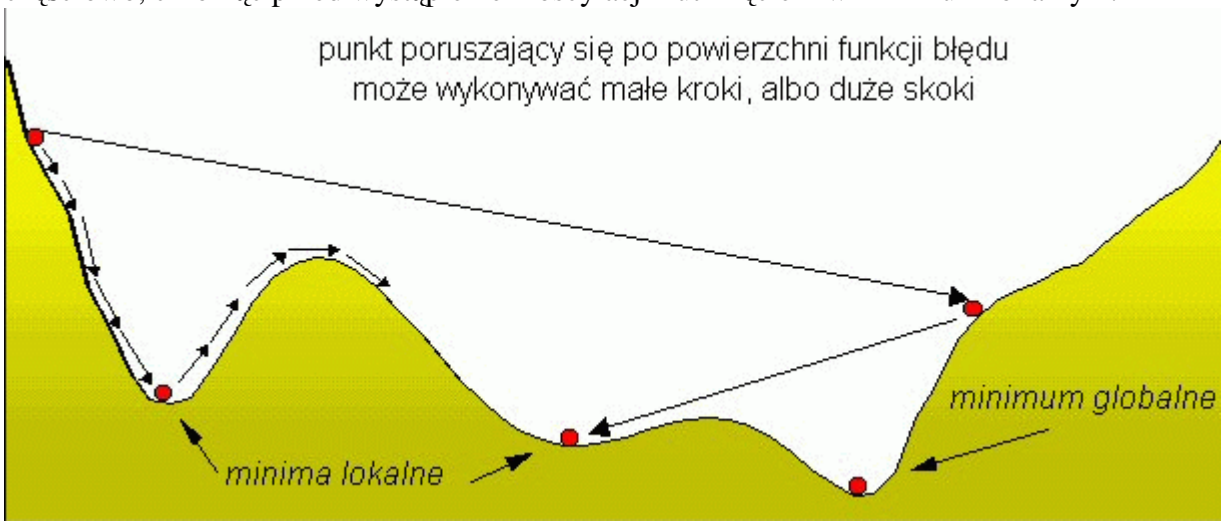
**Maksymalna liczba epok** – określa maksymalną liczbę epok, jaka może być w procesie uczenia – również dla zatrzymania tego procesu.

**Dla przyspieszenia procesu uczenia zastosowałem następujące zabiegi:**

1. Dodanie **członu momentum,** czyli przy poprawianiu wag branie pod uwagę poprzedniej wartości poprawy danej wagi. Daje to bardzo duże przyspieszenie uczenia.
2. Dynamiczne ustalanie parametru eta: zmniejszanie jego wartości proporcjonalnie do liczby błędów w ostatniej epoce. Dodatkowy parametr A_Eta określa stopień tej zależności.
3. Losowo wybierany numer wzorca, od którego zaczyna się epoka.
4. Odpowiednio dobrane początkowo wylosowane wagi. Losowanie wag z pewnego zakresu (określane parametrami: rUk1,rUk2,rWy, mUk1,mUk2,mWy)
5. Ustalenie odpowiednio małych wartości początkowych biasów. Parametry: PoczBiasUk1 = 0.01, PoczBiasUk2 = 0.01, PoczBiasWy = 0.01
6. Możliwość zaburzenia wag podczas nauki, czyli pewnej niewielkiej losowej zmiany wag sieci, gdy sieć nie chce się uczyć, bo ugrzęzła w minimum lokalnym. Zaburzenie to jest określane przez parametry: MaxG = 20; MinR = 0.05; zaburzenie = 0.6, co oznacza, że jeżeli przez ostatnie 20 kroków popraw wag Suma epsilon nie zmieniła się o więcej niż 0.05, to należy zaburzyć wagi sieci o wartości losowe z przedziału [-0.6 ; 0.6].
7. Dodatkowy cykl popraw wag na końcu epoki dla jednego wzorca, który uporczywie powoduje błąd SSN.

**Momentowa metoda wstecznej propagacji błędów (MBP)**
Klasyczna metoda wstecznej propagacji błędów wymaga dużej liczby iteracji by osiągnąć zbieżność oraz jest wrażliwa na występowanie minimów lokalnych. Podstawowy algorytm BackPropagation może się okazać zbyt wolny, jeżeli przyjmie się za mały współczynnik uczenia, z kolei zbyt duża jego wartość grozi wystąpieniem **oscylacji**. Jednym z rozwiązań umożliwiających bezpieczne zwiększenie

efektywnego tempa uczenia bez pogarszania stabilności procesu jest zastosowanie momentowej metody wstecznej propagacji błędu – MBP lub MEBP (ang. Momentum Error BackPropagation).
Istotą metody jest wprowadzenie do procesu uaktualniania wagi pewnej **bezwładności** tzw. "momentu", proporcjonalnego do zmiany tej wagi w poprzedniej iteracji. Pojawienie się gradientu wymuszającego ruch w kierunku innym, niż aktualny nie od razu wpływa na zmianę trajektorii w przestrzeni wag. W przypadku, gdy kolejne gradienty są przeciwnie skierowane ich oddziaływanie częściowo się znosi. Powoduje to, że zmiana położenia w przestrzeni wag jest bardziej płynna i chwilowe zbyt duże zmiany są w rezultacie kompensowane.
Składnik momentu ma duże znaczenie dla płaskich odcinków funkcji błędu, dla których zwykle obserwuje się zmniejszenie szybkości uczenia. Dla płaskiego obszaru hiperpłaszczyzny funkcji miary błędu w przestrzeni wag, gdzie gradient w kolejnych iteracjach jest w przybliżeniu stały, przyrost wag w kolejnych iteracjach jest również stały. Gdy kolejne wektory gradientu mają ten sam kierunek ich działanie **kumuluje się**.
Moment odgrywa również pozytywną rolę w przypadku występowania na powierzchni funkcji błędu tzw. **wąwozów**, to znaczy wąskich obszarów o stromych ścianach bocznych i głębokich dnach. Moment pełni tu niejako rolę filtru dolnoprzepustowego dla zmian gradientu. Zmiany gradientu o wysokiej częstotliwości (oscylacje w poprzek ścian wąwozu) są eliminowane, a wzmacniany jest składnik gradientu wymuszający ruch w stronę dna. Modyfikacja momentowa BP ma ponadto korzystny wpływ na problem **minimów lokalnych**. W pobliżu takiego minimum składnik momentu, nie będąc związany z aktualną wartością gradientu, może spowodować zmianę wag prowadzącą do chwilowego wzrostu wartości funkcji błędu i w efekcie opuszczenia strefy "przyciągania" tego minimum.
Algorytm ten w rzeczywistości wykonuje wygładzanie poprawek zbioru wag, uzależnione od wartości współczynnika wygładzania - momentu.
Modyfikacja momentowa algorytmu propagacji wstecznej powoduje zatem przyspieszenie procesu uczenia przy pewnych stałych tendencjach zmian w przestrzeni wag jednocześnie, przynajmniej częściowo, chroniąc przed wystąpieniem oscylacji i utknięciem w minimum lokalnym.




**TESTOWANIE POPRAWNOŚCI NAUKI SSN**
Zakończenie się procesu nauki SSN może wystąpić z dwóch powodów

a) SSN nauczyła się, tzn. osiągnęła wymagane parametry, tzn. wartość **epsilon** i **Suma epsilon**. W tym wypadku program podaję liczbę przeprowadzonych kroków nauki.
b) SSN nie nauczyła się a osiągnięto już maksymalną liczbę iteracji (kroków nauki: popraw wag lub liczby epok). W tym wypadku program podaje liczbę błędów, tzn. przypadków, dla których błąd jest > epsilon.

Zakończenie procesu nauki nie musi oznaczać, że SSN nauczyła się poprawnie, tzn. że daje poprawne odpowiedzi dla uczonych wzorców (, bo np. parametr epsilon był za duży – np. >0,5).
Zatem po zakończeniu procesu nauki, sprawdzana jest POPRAWNOŚĆ rozpoznawania uczonych wzorców, tzn. czy SSN poprawnie się ich nauczyła.

Polega to na podaniu na wejście sieci kolejno wszystkich uczonych wzorców i sprawdzaniu czy sieć daje oczekiwane wyniki.

Następną sprawą jest test możliwości generalizacji SSN, tzn. jak sieć radzi sobie z rozpoznawaniem nieuczonych przypadków. Takiego testu nie przeprowadzam, gdyż uczę SSN wszystkich dostępnych wzorców (ponad 1000).
Jeżeli sieć jest nauczona poprawnie, jej parametry (rozmiary sieci, wagi, współczynniki Beta) są zapisywane do pliku '**parametry.ssn**', który później jest wczytywany przez program do rozpoznawania obrazów.

**IMPLEMENTACJA:**
Aplikacja została napisana w środowisku Borland C++ Builder 6.0 w sposób obiektowy.
Dlaczego C++:
Uczenie SSN jest procesem wymagającym bardzo dużej liczby obliczeń a język C++ zapewnia najszybszy kod wynikowy z języków wysokiego poziomu.

Oto ważniejsze elementy programu do uczenia SSN:
Plik **KlasaBazaZnakow.h/cpp** zawiera definicję klasy odpowiedzialnej za wczytanie obrazów znaków z przygotowanego wcześniej pliku bazy.
Baza obrazów zapamiętywana jest w obiekcie klasy **KlasaBazaZnakow.**
**Struktura bazy** (plik BazaObrazowZnakow.boz):
Każdy rekord składa się z 257 bajtów: pierwsze 256 bajtów to kolejne wartości pikseli (0-tło (biały), 255-znak (czarny)) ustawione wierszami od góry. Ostatni bajt, to etykieta (0-9 –oznacza cyfrę, 10-niecyfra).

W programie uczącym SSN zdefiniowano w pliku **KlasaSiecNmom2H.h/cpp** klasę **SIECN2H**, która tworzy, uczy SSN i ma możliwość zapisu i wczytywania parametrów nauczonej sieci.

W głównym pliku **UnitFormMain.cpp**, gdzie jest tworzony obiekt klasy SIECN2H są dwie funkcje przygotowujące dane do wejścia i oczekiwanego wyjścia SSN:
double* **FunPX**(int n) //zwraca wskaźnik na tablicę danych wEjściowych n-tego wzorca
double* **FunPZ**(int n) //zwraca wskaźnik na tablicę danych wYjściowych n-tego wzorca

* Utworzenie SSN następuje przez utworzenie obiektu klasy SIECN2H konstruktorem:
**SIECN2H**(int _LDanychWe, int _LNeuronowUk1, int _LNeuronowUk2, int _LneuronowWy)
Parametrami są rozmiary w liczbach neuronów kolejnych warstw SSN.

* Po utworzeniu SSN należy ją zresetować, tzn. ustalić współczynniki Beta dla każdej z warstw oraz ustalić wartości wag na losowe. Zaczynając od **losowo** ustalonych wartości wag, SSN uczy się najszybciej!
Dokonujemy tego metodą:
Siec.**Resetuj**(BetaUk1, BetaUk2, BetaWy, PoczBiasUk1, PoczBiasUk2, PoczBiasWy,
rUk1,rUk2,rWy, mUk1,mUk2,mWy)
Parametry funkcji, to:
BetaUk1, BetaUk2, BetaWy – wartości beta funkcji aktywacji kolejnych trzech warstw sieci
PoczBiasUk1, PoczBiasUk2, PoczBiasWy – początkowe wartości bias dla wag kolejnych warstw sieci
rUk1,rUk2,rWy, mUk1,mUk2,mWy – określają sposób losowania wag: zakres i przesunięcie

* Proces nauki SSN rozpoczynamy wywołaniem funkcji:
Siec.**LearnAll**(LWzorcow, FunPX, FunPZ, EtaUk1, A_EtaUk1, AlfaUk1, EtaUk2, A_EtaUk2, AlfaUk2, EtaWy, A_EtaWy, AlfaWy, mC, MinR, MaxG, zaburzenie, eps, Sum_eps, MaxLPopraw, MaxLEpok, void PokazOpis(char *info))
Kolejne parametry, to:
- Lwzorcow – liczba wzorców do nauczenia
- FunPX, FunPZ – funkcje przygotowujące wartości wejściowe i wyjściowe funkcji

- EtaUk1, A_EtaUk1, AlfaUk1, EtaUk2, A_EtaUk2, AlfaUk2, EtaWy, A_EtaWy, AlfaWy – wektor nauki i sposób jego dynamicznej modyfikacji
- mC – (dla eksperymentu) zmienia wektor uczenia (tu ustalono na = 0.0 – nic nie zmienia)
- MinR, MaxG, zaburzenie – określają, kiedy i jak ma być dokonane zaburzenie na wagach sieci, (gdy sieć ugrzęźnie w jakimś minimum lokalnym i nie chce się dalej uczyć)
- **eps, Sum_eps** – epsilon i suma episolon – warunki, jakie ma spełnić nauczona sieć
- MaxLPopraw, MaxLEpok – kryterium zatrzymania procesu uczenia (aby przerwać naukę, gdy nie chce się nauczyć)
- PokazOpis(char *info) – funkcja która będzie wywoływana po każdej epoce; parametr info będzie zawierał łańcuch opisujacy aktualny stan nauki SSN

Proces nauki SSN polega na odpowiednim zmienianiu wag sieci, tzn. liczb oznaczających połączenie między neuronami (stopień wpływu jednego sygnału na inny). Wartości tych wag ustala się początkowo losowo, aby móc rozpocząć naukę od jakiegoś miejsca. Wpływ na sposób losowania tych wartości uzyskujemy przez ustalanie parametrów: rUk1, rUk2, rWy oraz mUk1, mUk2, mWy. Wpływają one na wartości losowanych początkowo wag sieci dla każdej warstwy osobno:
Uk1- pierwsza warstwa ukryta
Uk2 - druga warstwa ukryta
Wy - warstwa wyjściowa

Sam proces poprawiania wag względem obliczonych przez SSN danych wyjściowych a oczekiwanych odbywa się w funkcji:
**PoprawWagi**(const double *DaneOcz, double etaUk1, double alfaUk1, double etaUk2, double alfaUk2, double etaWy, double alfaWy, double mC)

Jest to najważniejsza funkcja (jądro) składająca się na naukę SSN. Tutaj zaimplementowany jest ważny algorytm **minimalizacji funkcji błędu** - metoda delty i tu obliczany jest gradient tej funkcji.

Funkcja obliczająca wynik SSN dla ustawionych wcześniej funkcją FunPX(nr_wzorca) danych wejściowych, to:
**Wynik**(void) – ustala wyjście SSN, tzn. wartości w tablicy **DaneWy[]**

W programie do **samego rozpoznawania** obrazów (bez nauki) użyto nieco zmienionej klasy SIECN2H, tzn. bez metod: LearnAll i służącej do zapisywania parametrów sieci.
Deklaracja i definicja tej kasy są zawarte w plikach **KlasaSiecNmom2HRozp.h/cpp.**

**KONKRETNE PROGRAMY**

**1. DopisDoBazyZnakow**
- wyszukuje w obrazie z podanego pliku BMP obrazy cyfr
- transformuje je: skalowanie, kontrast, normalizacja itp.
- dopisuje obraz z podaną etykietą do wskazanego pliku (BazaObrazowZnakow.boz)

**2. EdycjaBazyZnakow**
- umożliwia przeglądanie zapisanych w pliku bazy obrazów. Przyciski ‘<<’-poprzedni, ‘>>’-następny lub wpisanie numeru elementu i wciśnięcie Enter
- umożliwia zmianę etykiety danego obrazu, (gdy np. błędnie ją wpisaliśmy w programie DopisDoBazyZnakow): przycisk ‘Zmień etykietę na’; obok należy wpisać właściwą etykietę dla aktualnego obrazu znaku.
- umożliwia usunięcie danego obrazu z bazy: w jego miejsce zostaje wpisany ostatni element (aby nie buło ‘dziur’ w bazie)
- dokonane zmiany w bazie obrazów, należy zapisać przyciskiem ‘Zapisz bazę’

**3. UczObrazy**
Implementacja SSN i uczenie jej wzorcami z bazy obrazów (plik BazaObrazowZnakow.boz).

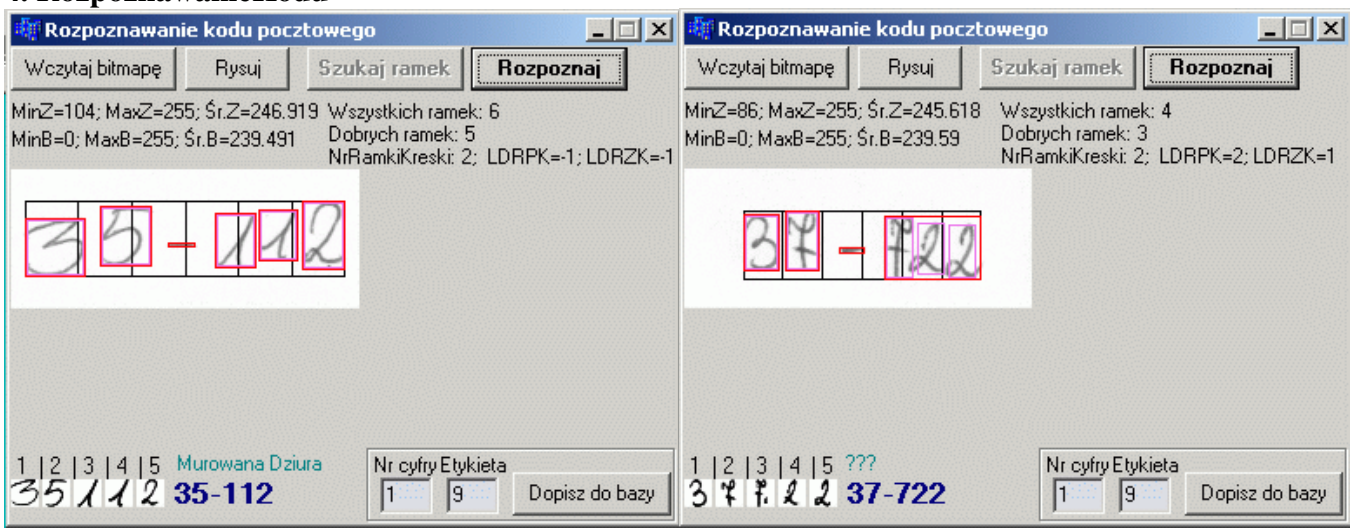
W trakcie jego działania, pokazywany jest proces uczenia się SSN.
Pozwala to śledzić jak zmieniają się niektóre kluczowe wartości sieci.
Kolejne wiersze zawierają:
**Epk** - Numer kolejnej epoki
**LB** – liczba błędów w ostatniej epoce
**Nrp** – nr ostatniego wzorca w epoce, dla którego był błąd (Uwaga: nr wzorca, od którego zaczynamy epokę jest losowy, wiec ten ostatni, to nie musi być ostatni w kolejności jak w bazie danych)
**SumE** – Suma błędów ze wszystkich wzorców w całej epoce
**SO-SE** – różnica między sumą błędów w poprzedniej epoce i obecnej
**EtaUk1, Uk2, Wy** – bieżące wartości parametrów eta dla każdej warstwy (zmienia się dynamicznie; zależy od liczby błędów w ostatniej epoce)

Po poprawnym nauczeniu SSN program zapisuje do pliku 'ParametryNaukiSSN.txt' wszystkie parametry SSN i wiele innych informacji o nauce.

**Szybkość nauczenia SSN - przykłady:**
* 914 wzorców, w tym 6 nie cyfr, Epsilon = **0.28**:
- 40 epok; 5660 popraw; 33 s
* 914 wzorców, w tym 6 nie cyfr, Epsilon = **0.27**:
- 91 epok; 6039 popraw; 85 s
- 37 epok; 5998 popraw; 42 s
* 908 wzorców, w tym 0 nie cyfr, Epsilon = **0.27**:
- 28 epok; 6294 popraw; 36 s
* 923 wzorców, w tym 15 nie cyfr, Epsilon = **0.27**:
- 51 epok; 6135 popraw; 53 s
* 915 wzorców, w tym 8 nie cyfr, Epsilon = **0.20**:
- 44 epoki; 6041 popraw; 47 s
* 1173 wzorców, w tym 100 nie cyfr, Epsilon = **0.25**:
- 69 epok; 7229 popraw; 106 s

Duże przyspieszenie uczenia daje zmniejszenie (nawet do 0) liczby wzorców niecyfr. Jednak może to pogorszyć możliwości generalizacji wiedzy SSN.

**4. RozpoznawanieKodu**




Ostateczny program do rozpoznawania 5 cyfr z ciągu cyfr kodu pocztowego z podanego obrazu z pliku BMP.
Program przy uruchamianiu od razu tworzy na podstawie pliku 'parametry.ssn' odpowiednich rozmiarów SSN i wczytuje, tam zapisane, parametry nauczonej wcześniej SSN. Plik ten musi być w katalogu z programem.

* Wiele wspólnych funkcji operujących na obrazie zostało zebranych w jednym pliku:
**UnitFunkcjeObraz.h/cpp**

**Ogólny proces przygotowania obrazu do wejścia SSN:**

1. Rozmycie wczytanego obrazu BMP dla zamazania ewentualnych zabrudzeń
2. Znalezienie ramki ciągu znaków
3. Znalezienie w tej ramce ramek poszczególnych cyfr
4. Wpisanie danej ramki znaku do kwadratu – dla zachowania proporcji rozmiarów
5. Normalizacja obrazu – przeskalowanie kolorów do przedziału [0 – 255]
6. Kontrast – rozjaśnienie jasnych pikseli i ściemnienie ciemnych – względem średniej jasności kolorów w ramce (korekcja Gamma)
7. Skalowanie do wielkości 16x16
8. Ponowna normalizacja
9. Ponowny kontrast – względem średniej jasności kolorów w ramce
10. Inwersja kolorów

* Funkcja szukająca ramki ciągu znaków to:
**SzukajRamkiR**(unsigned char *Obraz, int RozX, int RozY, unsigned char Prog, int Roznica, int MinX, int MinY, TRamka &Ramka, char *info)

* Po znalezieniu ramki ciągu znaków, szukamy w niej ramek poszczególnych cyfr, funkcją:
**SzukajZnakOdLewejR**(unsigned char *Obraz, int RozX, int RozY, //wsk. na obraz; jego rozmiary
TRamka RamkaO, //ramka obrazu
unsigned char ProgB, //próg bezwzględny jasności
int Roznica, //próg różnicy max-min na całej wysokości ramki
TRamka &RamkaZ, char *info) //wynik

* Potem ramka znaku może być jeszcze poobcinana funkcją:
**SzukajPodRamkiR**(unsigned char *Obraz, int ObrazRozX, int ObrazRozY, //wsk. na obraz;
unsigned char Prog, int Roznica,
int MinX, int MinY, TRamka RamkaZ,
TRamka &RamkaW, char *info) //wynik

* Po wyznaczeniu ramki danej cyfry jej obraz podawany jest do funkcji:
**PobierzObrazDoSSN**(unsigned char *ObrazZ, int ObrazRozX, int ObrazRozY, unsigned char *ObrazBuf, TRamka Ramka, unsigned char *ObrazZnak, int ZnakRozX, int ZnakRozY, char *info)

Samo znalezienie ramki znaków nie jest takie proste i nie wystarczy zwykłe rozróżnianie pikseli ciemnych od jasnych, gdyż obraz znaków może być na ciemnym tle lub znaki mogą być wyblakłe, rozjaśnione. Jest jeszcze wiele innych pułapek. Dlatego tak dużo jest użytych różnych funkcji i każda z nich tak bardzo rozbudowana, a i tak nie są one w stanie rozpoznać wszystkich możliwych przypadków. Wziąłem pod uwagę tylko te najczęściej występujące.
**
Wszystkie kody źródłowe zostały napisane od podstaw. Oprócz tej dokumentacji wiele opisów i wyjaśnień jest w samych komentarzach w kodach źródłowych.
Autor: Artur Czekalski; ARTUR@epokaY.net; www.epokay.net/artur; 11d-4m-2005