Systemy sztucznej inteligencji

dokumentacja projektu Digit Recognizer

Jambor Daniel
Grupa 2D

Kozieł Wojtek
Grupa 2D

Matula Kamil
Grupa 2D

25 maja 2020

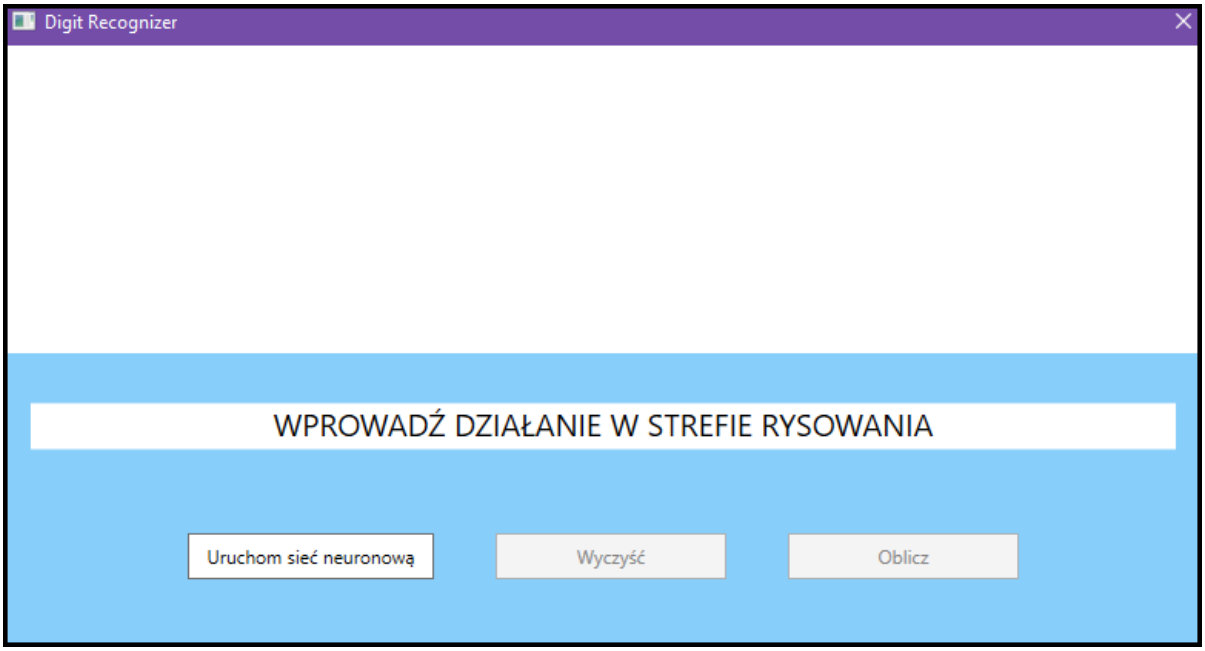
Część I

Opis programu

Program ***Digit Recognizer*** służy do rozpoznawania ręcznie napisanych działań i wyświetlania ich wyniku. Użytkownik pisze na specjalnym polu liczby naturalne oraz dowolne z czterech zaimplementowanych znaków arytmetycznych (odpowiadających działaniom dodawania, odejmowania, dzielenia i mnożenia), a program wyświetla końcowy wynik podanego wyrażenia. Program korzysta z bazy danych „MNIST", składającej się łącznie z 70 000 ręcznie napisanych cyfr, oraz z autorskiej bazy cyfr i oznaczeń matematycznych.

Instrukcja obsługi

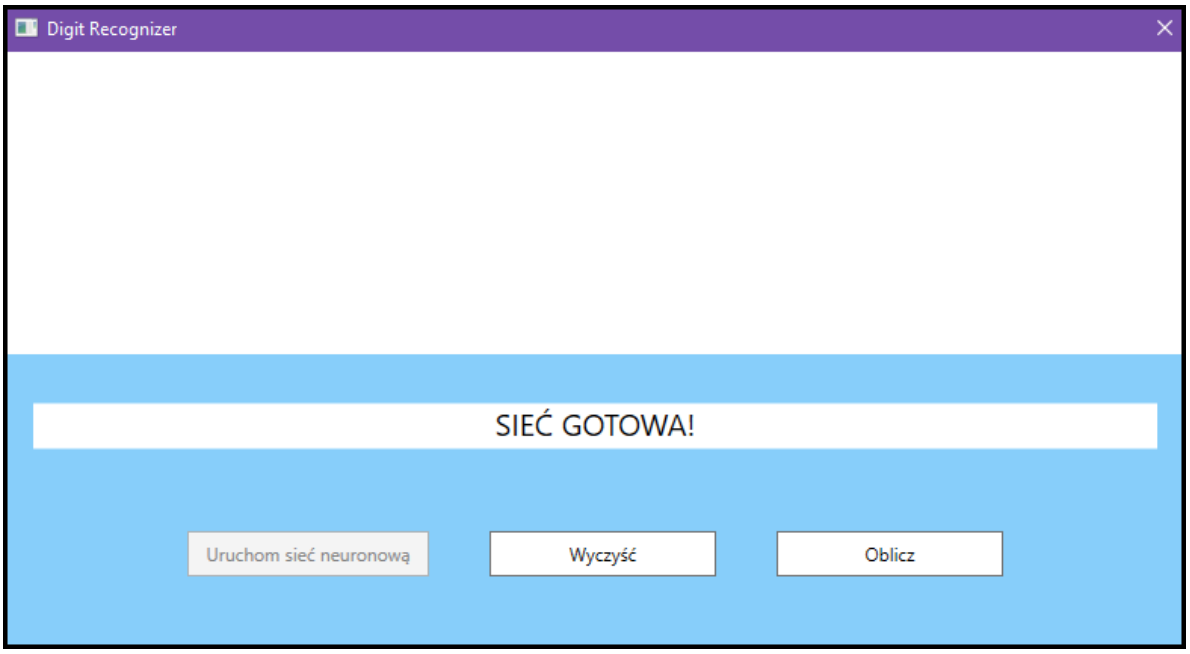
Po uruchomieniu pliku wykonawczego *DigitRecognizer.exe*, użytkownik powinien zobaczyć poniższy interfejs graficzny:



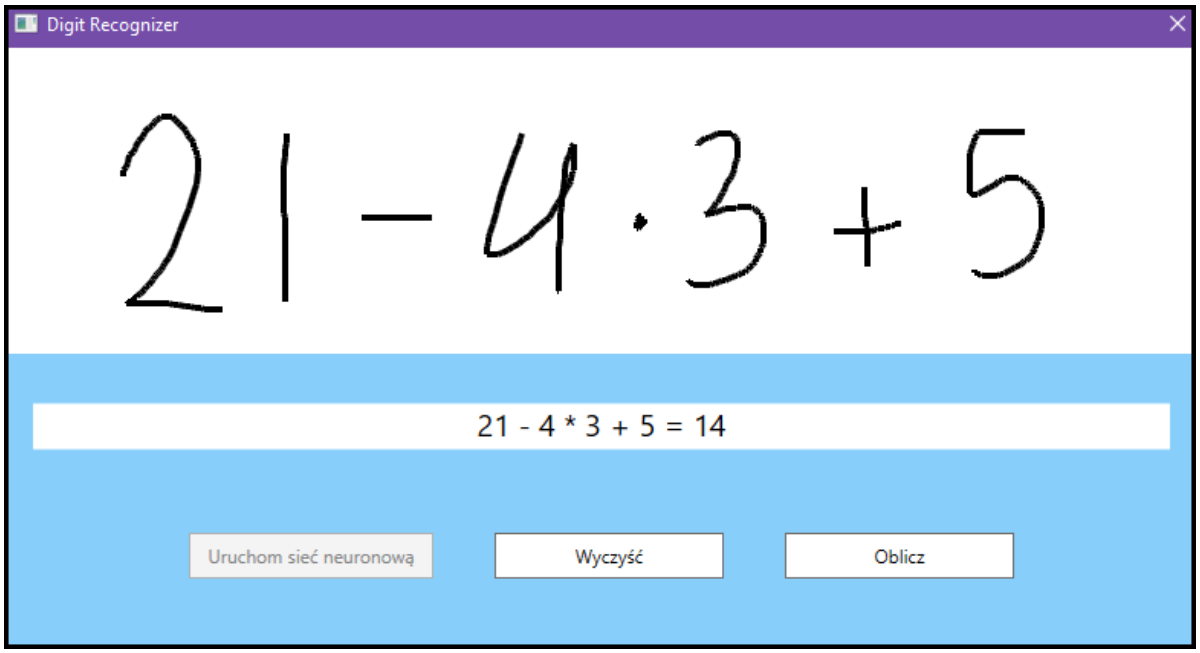
Jak widać na zamieszczonym wyżej zrzucie ekranu aplikacja składa się z:

- białego pola (tzw. „strefy rysowania"), na którym użytkownik może zapisywać działania,
- szerokiego pola tekstowego, gdzie wyświetlane są wszystkie komunikaty związane z działaniem programu włącznie z wynikiem zapisanego wyrażenia,
- przycisku „Uruchom sieć neuronową", który jako jedyny jest dostępny po uruchomieniu programu - pozwala on zbudować sieć neuronową i wczytać wcześniej wyuczone wagi,
- przycisku „Wyczyść", który czyści zawartość strefy rysowania,
- przycisku „Oblicz", który zleca sieci neuronowej pobranie narysowanych przez użytkownika cyfr i znaków, a także zwraca obliczony wynik.

Po wciśnięciu przycisku „Uruchom sieć neuronową” w polu tekstowym pojawi się informacja o tym, że sieć jest już gotowa. Ponadto przycisk ten stanie się nieaktywnym, a pozostałe dwa uaktywnią się, co widać na poniższym obrazku:



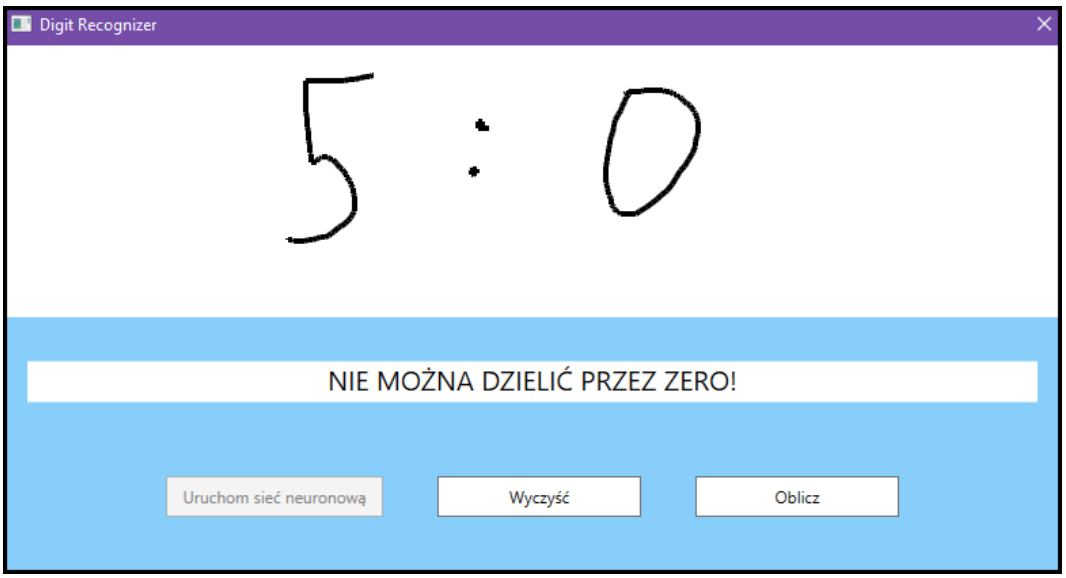
W celu obliczenia wartości pewnego wyrażenia wystarczy napisać je w strefie rysowania, a następnie wcisnąć przycisk „Oblicz” np.:



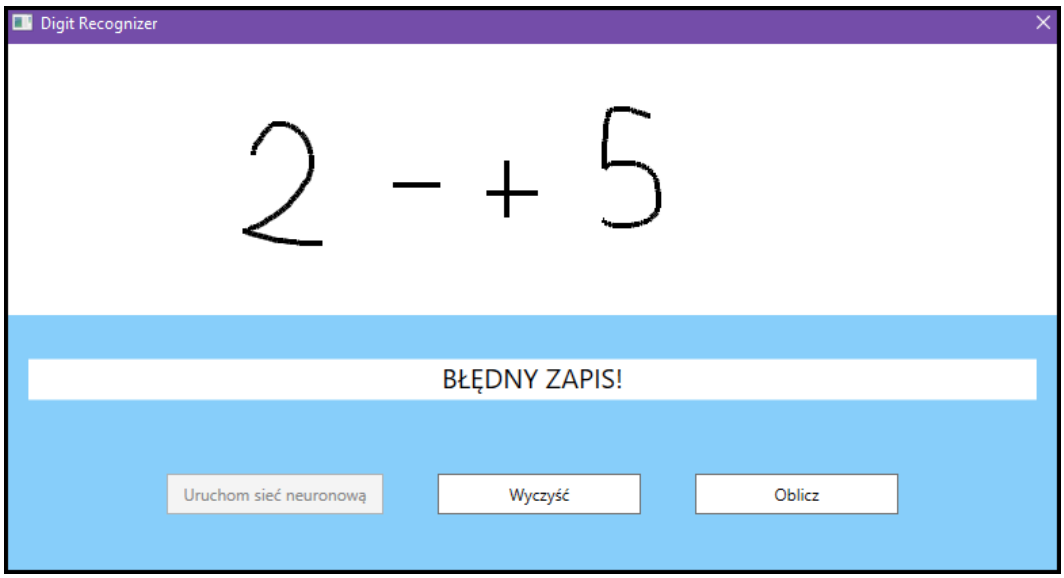
Dodatkowe informacje

Wymagania i zabezpieczenia

Do prawidłowego działania aplikacji wymagany jest plik *weights.txt*, który powinien znajdować się w tym samym folderze, co plik *DigitRecognizer.exe*. Zawiera on konfigurację sieci (współczynnik α funkcji aktywacji oraz ilość neuronów na poszczególnych warstwach), a także wagi synaps wejściowych każdej warstwy. W przypadku nieodnalezienia pliku z wagami lub niepoprawnej zawartości tego pliku pojawi się komunikat *„NIE ZNALEZIONO PRAWIDŁOWEGO PLIKU Z WAGAMI!”*. Program został także zabezpieczony przed możliwością dzielenia liczby przez zero co widać poniżej:



oraz przed niepoprawnym zapisem (tj. obecnością znaku arytmetycznego na początku wyrażenia, jego końcu lub obok innego znaku):



Przygotowanie sieci neuronowej

Sama aplikacja nie wymaga niczego oprócz dwóch wspomnianych plików, jednakże sieć neuronowa musiała wcześniej zostać nauczona rozpoznawania cyfr, zanim mogła zostać użyta w programie. W oddzielnym folderze projektu o podtytule *Learning Place* znajduje się zestaw plików źródłowych napisanych w języku C#, a także folder *Datasets* zawierający bazę MNIST oraz bazę autorską. Aby lepiej nauczyć sieć należy zmodyfikować ustawienia zapisane w metodzie *Main* w pliku *Program.cs*. Możliwe jest też douczenie sieci wstępnie nauczonej, dzięki zaimplementowanej funkcji wczytywania wag. Po nauczaniu plik z wagami powinien zostać skopiowany z folderu bieżącego do folderu, w którym znajduje się plik wykonawczy *DigitRecognizer.exe*.

Dodatkowe technologie i ich niedokładności

Do stworzenia pola zwanego „strefą rysowania" wykorzystano klasę *Canvas*, czyli obszar, na którym można pozycjonować elementy podrzędne, oraz klasę *Line* z przestrzeni nazw *Shapes*, z której pomocą zaimplementowano możliwość rysowania linii z użyciem myszy.
W połączeniu z algorytmem wycinającym każdy znak lub cyfra rozpoznawane są osobno.

Niestety połączenie Canvas oraz zaimplementowanych przez nas algorytmów do przetwarzania otrzymanego obrazu niesie za sobą pewne ryzyko wykrycia przez algorytm pojedynczych pikseli odłączonych od znaku, powstałych w skutek poszarpania linii rysowania. Piksele te znajdują się nad lub pod znakiem i mają niewielki wpływ na dokładność odczytywania znaku.

Algorytm przetwarzający obraz wykorzystuje klasę *Graphics* z przestrzeni nazw *System.Drawing*, której właściwości pozwoliły na wycięcie oraz przeskalowanie obrazów bez ich zniekształcenia. Dzięki strukturze *Rectangle* z tej przestrzeni możliwe było wycięcie figur z użyciem tylko punktów początkowych i końcowych cięcia.

Część II

Opis działania sieci neuronowej

Jak zostało wcześniej wspomniane program opiera się na sztucznej sieci neuronowej (SSN), czyli matematycznym modelu sieci nerwowej działającej w mózgu. Podobnie jak ludzka sieć neuronowa, SSN zbudowana jest z neuronów ułożonych w warstwy. Każda komórka nerwowa danej warstwy połączona jest ze wszystkimi komórkami warstwy poprzedniej i warstwy następnej za pomocą synaps posiadających pewne losowo zainicjowane wagi w postaci liczb. Są one modyfikowane w procesie uczenia sieci neuronowej.

Pierwszą warstwę sieci, odpowiedzialną za przyjmowanie danych wejściowych, nazywamy warstwą wejściową. Analogicznie ostatnia warstwa sieci to warstwa wyjściowa, odpowiadająca za zwracanie wyniku. Pomiędzy nimi mogą (lecz nie muszą) znajdować się tzw. warstwy ukryte. Zadaniem projektanta sieci neuronowej jest znalezienie optymalnej ilości i wielkości tych warstw, dzięki czemu nauczanie będzie przebiegało efektywnie. Z kolei ilość neuronów na warstwach skrajnych zależy od tego, ile cech posiada obiekt wejściowy oraz do ilu klas można go zaklasyfikować na wyjściu - w przypadku tego projektu jest to 784 neuronów wejściowych (z każdego piksela pobieramy osobną wartość) oraz 14 neuronów wyjściowych (tyle różnych znaków można uzyskać dzięki rozpoznawaniu).

Każdy z neuronów przyjmuje pewną wartość na wejściu, a następnie przetwarza ją dzięki funkcji aktywacji. Sygnał wejściowy i-tego neuronu k-tej warstwy można opisać równaniem:

$$s_i^k = \sum_{j=1}^{n} w_{ij}^k y_j^{k-1} + b,$$

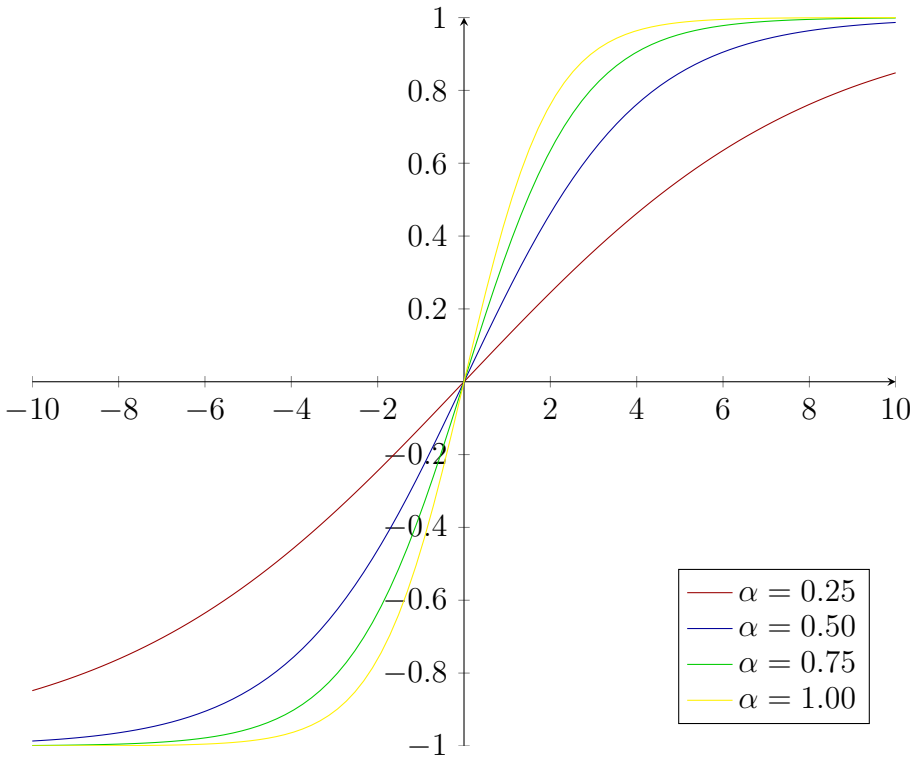
gdzie w_{ij}^k - waga synapsy pomiędzy i-tym neuronem k-tej warstwy a j-tym neuronem warstwy poprzedniej, y_j^{k-1} - wartość sygnału wyjściowego j-tego neuronu warstwy poprzedniej, b - zakłócenia sieci (tzw. bias). Najczęściej we wzorze tym nie uwzględnia się ostatniego czynnika (zakłada się, że sieć nie posiada zakłóceń tj. $b = 0$). Z kolei sygnał wyjściowy i-tego neuronu to:

$$y_i^k = f(s_i^k) = f(\sum_{j=1}^{n} w_{ij}^k y_j^{k-1} + b).$$

Wyróżniamy wiele funkcji aktywacji, jednak najczęściej wykorzystywaną (i wykorzystaną również w tym projekcie) jest funkcja bipolarna liniowa, której wzór wygląda następująco:

$$f(s_i^k) = \frac{2}{1 + e^{-\alpha s_i^k}} - 1 = \frac{1 - e^{-\alpha s_i^k}}{1 + e^{-\alpha s_i^k}}$$

gdzie α jest współczynnikiem korygującym rozpiętość funkcji aktywacji w przestrzeni decyzyjnej. Jej wykres zamieszczono na następnej stronie.



Bipolarna liniowa funkcja aktywacji

Kiedy sztuczna sieć neuronowa jest już odpowiednio zbudowana, należy ją nauczyć tego, czego od niej oczekujemy. Polega to na modyfikowaniu wag synaps w ściśle określony sposób. Jest wiele metod uczenia z czego część wymaga nauczyciela w postaci zbioru treningowego z danymi wejściowymi i oczekiwanymi danymi wyjściowymi, a część nie - wtedy sieć dostaje tylko dane wejściowe. W przypadku uczenia rozpoznawania obiektów stosuje się metody uczenia z nauczycielem. Jedną z takich strategii jest algorytm wstecznej propagacji. Jej zadaniem jest zminimalizowanie wartości funkcji błędu dla wszystkich elementów zbioru treningowego T, co opisuje wzór:

$$B(T) = \sum_{T} \sum_{i=1}^{n} (d_i - y_i)^2,$$

gdzie n to wymiar wektora wyjściowego / liczba neuronów wyjściowych, d_i to wartość oczekiwana na i-tej pozycji wektora wyjściowego, a y_i to wartość uzyskana na i-tej pozycji wektora wyjściowego. Korekcja wag synaps wejściowych poszczególnych neuronów zaczyna się w warstwie wyjściowej oznaczanej literą K i przebiega wstecz przez wszystkie wcześniejsze warstwy aż dotrze do warstwy wejściowej. Równanie korekcji wag wygląda następująco:

$$w_{ij}^k = w_{ij}^k + \eta \nabla w_{ij}^k,$$

gdzie η jest współczynnikiem korekcji powszechnie nazywanym „Learning Rate", a ∇w_{ij}^k to wartość gradientu błędu wagi synapsy opisywana wzorem:

$$\nabla w_{ij}^k = \frac{\partial B(T)}{\partial w_{ij}^k} = \frac{1}{2} \cdot \frac{\partial B(T)}{\partial s_i^k} \cdot 2 \cdot \frac{\partial s_i^k}{\partial w_{ij}^k} = 2\delta_i^k y_j^{k-1},$$

gdzie δ_i^k to zmiana funkcji błędu dla sygnału wejściowego i-tego neuronu k-tej warstwy, a y_j^{k-1} to sygnał wyjściowy j-tego neuronu warstwy poprzedniej. Wspomniana wartość δ liczona jest inaczej na warstwie wyjściowej i inaczej na pozostałych. Na K-tej warstwie wynosi:

$$\delta_i^K = \frac{1}{2} \cdot \frac{\partial B(T)}{\partial s_i^k} = \frac{1}{2} \cdot \frac{\partial (d_i^K - y_i^K)^2}{\partial s_i^k} = f'(s_i^K) \cdot (d_i^K - y_i^K),$$

gdzie $f'(s_i^K)$ to pochodna funkcja aktywacji na K-tej warstwie (wyjściowej). Wartość zmiany funkcji błędu na pozostałych warstwach jest zależna od wartości uzyskanej na warstwie następnej i jest równa:

$$\delta_i^k = \frac{1}{2} \cdot \frac{\partial B(T)}{\partial s_i^k} = \frac{1}{2} \cdot \sum_{j=1}^{N_{k+1}} \frac{\partial B(T)}{\partial s_j^{k+1}} \frac{\partial s_j^{k+1}}{\partial s_i^k} = f'(s_i^k) \sum_{j=1}^{N_{k+1}} \delta_j^{k+1} w_{ji}^{k+1},$$

gdzie N_{k+1} to liczba neuronów warstwy następnej.

Opis sposobu wyodrębniania znaków

W celu rozpoznania cyfr i znaków zawartych w strefie rysowania, cały jej obszar wysyłany jest do klasy DigitDetection gdzie rozpoczyna się jego przetwarzanie. Początkowo algorytm przeszukuje obraz kolumna po kolumnie szukając pikseli, które zostały zmienione. Powstaje wtedy lista kolumn w których zawarte są znaki. Z listy obliczane są kolumny stanowiące początek i koniec znaków w osi X. Następnie tak samo działający algorytm przeszukuje obliczone uprzednio obszary w osi Y. Na skutek działania obu algorytmów powstają dwie listy posiadające punkty pozwalające skopiować obszary zawierające cyfry bądź znaki. Każda z osobna figura jest środkowana oraz jej rozmiar jest transformowany do 28x28 pikseli. Tak przetworzone obrazy zamieniane są na wartości liczbowe, które ostatecznie wysyłane są do sieci neuronowej.

Opis sposobu wyznaczania wyniku zapisanych wyrażeń

Aby program mógł poprawnie obliczać podane wyrażenia, zapis musi być przekonwertowany na ONP (Odwrócona Notacja Polska). ONP to sposób zapisu wyrażeń arytmetycznych, w którym znak wykonywanej operacji umieszczony jest po operandach (zapis postfiksowy), a nie pomiędzy nimi jak w konwencjonalnym zapisie algebraicznym lub przed operandami jak w zwykłym zapisie prefiksowym. Dzięki takiemu zapisowi, program potrafi rozpoznać kolejność wykonywania działań i podać prawidłowy wynik. Aby uzyskać wynik z wyrażenia, wywoływana jest metoda, która konwertuje zapis klasyczny, na zapis postfiksowy. Następnie wynik tej metody jest wysyłany do algorytmu, który liczy wartość wyrażenia, korzystając już z zapisu ONP. Metoda licząca wykonuje się rekurencyjnie - pojedyncze znaki z wyrażenia są wyciągane ze stosu, a następnie, jeżeli program nie będzie mógł przekonwertować znaku na liczbę - metoda jest wyzwalana ponownie. Jeżeli metoda napotka znak, to algorytm sumuje, odejmuje, mnoży bądź dzieli liczby.

Algorytmy

Uczenie sieci

Data: ilość iteracji - *EpochsCount*, dane wejściowe zbioru treningowego - $Inputs$, oczekiwane dane wyjściowe zbioru treningowego - *ExpectedOutputs*
Result: Większa dokładność sieci
$L :=$ ilość warstw sieci neuronowej;
$Deltas :=$ pusta tablica poszarpana o L wierszach i tylu kolumnach w danym wierszu, ile neuronów ma dana warstwa; będzie przetrzymywać wartości δ;
for $i = 0$ **to** *EpochsCount* **do**
  for $j = 0$ **to** *wielkość zbioru treningowego* **do**
    Wprowadź j-ty wektor wejściowy zbioru treningowego ($Inputs[j]$) do synaps wchodzących neuronów pierwszej warstwy;
    for $k = 0$ **to** L **do**
      Wyznacz s^k na wszystkich neuronach k-tej warstwy, sumując iloczyny wag synaps wchodzących i y^k neuronów warstwy poprzedniej (lub synaps w przypadku pierwszej warstwy);
      Wyznacz y^k na wszystkich neuronach k-tej warstwy poprzez zastosowanie funkcji aktywacji;
    end
    $Output :=$ wektor złożony z wartości wyjściowych ostatniej warstwy;
    for $n = 0$ **to** *ilość neuronów wyjściowych* **do**
      $Deltas[L-1][n] = (ExpectedOutputs[j][n] - Output[j]) \cdot f'(s_n^{L-1})$;
    end
    for $k = L - 2$ **to** 0 **by** -1 **do**
      for $n = 0$ **to** *ilość neuronów na k-tej warstwie* **do**
        $Deltas[k][n] = 0$;
        for $m = 0$ **to** *ilość neuronów na (k+1)-tej warstwie* **do**
          $Deltas[k][n] = Deltas[k][n] + Deltas[k+1][m] \cdot w_{mn}^{k+1}$;
        end
        $Deltas[k][n] = Deltas[k][n] \cdot f'(s_n^k)$;
      end
    end
    for $k = L - 2$ **to** 0 **by** -1 **do**
      for $n = 0$ **to** *ilość neuronów na k-tej warstwie* **do**
        for $m = 0$ **to** *ilość neuronów na (k-1)-tej warstwie* **do**
          $w_{nm}^k = 2 \cdot LR \cdot Deltas[k][n] \cdot y_m^{k-1}$;
        end
      end
    end
  end
end

Algorithm 1: Algorytm trenowania sztucznej sieci neuronowej.

Widoczny na poprzedniej stronie algorytm przedstawia pełny proces trenowania sieci neuronowej z wykorzystaniem zbioru treningowego i algorytmu wstecznej propagacji. Przedziały liczbowe, przez które przebiegają zapisane pętle **for** są jednostronnie domknięte (liczba po **to** nie jest brana pod uwagę). Pojawiają się też zapisy LR, s_i^k, y_i^k i w_{ij}^k - są to kolejno: wartość współczynnika nauczania (Learning Rate), wartość wejściowa i wartość wyjściowa i-tego neuronu na k-tej warstwie oraz waga synapsy pomiędzy i-tym neuronem k-tej warstwy a j-tym neuronem warstwy poprzedniej. Ponadto $f'(\cdot)$ oznacza wartość pochodnej funkcji aktywacji - dla funkcji bipolarnej liniowej o wzorze $f(x) = \frac{1-e^{-\alpha x}}{1+e^{-\alpha x}}$ pochodna wynosi $f'(x) = \frac{2\alpha e^{-\alpha x}}{(1+e^{-\alpha x})^2}$.

Odwrotna Notacja Polska

Poniższy pseudokod przedstawia proces zamiany wyrażenia zapisanego w standardowej notacji infiksowej na postfiksową Odwrotną Notację Polską.

Data: Wyrażenie w notacji infiksowej - *tokens*
Result: Wyrażenie zapisane w ONP
precendence ← słownik przechowujący symbole arytmetyczne i ich wagi;
Zdefiniuj nowy stos;
Zdefiniuj zmienną przechowującą wynik;
Rozbij wyrażenie *token* na listę typu *string*;
for *znak* ***in*** *lista wyrażeń* **do**
  try:
    Przekonwertuj znak na typ *double*;
    Dodaj przekonwertowany znak do wyniku;
  catch:
    while *stos nie jest pusty* ***and*** *waga znaku* $\leq$ *waga elem. na szczycie stosu* **do**
      Dodaj do wyniku pierwszy element ze stosu;
    end
    Dodaj znak ponownie do stosu;
  end
end
while *stos nie jest pusty* **do**
  Dodaj do wyniku element ze szczytu stosu;
end

Algorithm 2: Algorytm zmieniający zapis tradycyjny na zapis ONP

Po konwersji na Odwrotną Notację Polską wynik wyrażenia jest obliczany za pomocą poniższego algorytmu rekurencyjnego:

Data: Wyrażenie zapisane w ONP - *tokens* (w postaci stosu)
Result: Liczba reprezentująca wynik wyrażenia - $firstNumber$
Przypisz element ze szczytu stosu do zmiennej *token*;
Zdefiniuj dwie zmienne $firstNumber$ i *secondNumber*;
if *konwersja zmiennej token na double i przypisanie do zmiennej* $firstNumber$ *nie powiedzie się* **then**
    Wywołaj metodę ponownie dla zmiennych $firstNumber$ i *secondNumber*;
    if *token* == +, −, ∗ *lub* / **then**
        Wykonaj odpowiednie działanie na zmiennych $firstNumber$ i *secondNumber*;
    end
end

Algorithm 3: Algorytm liczący wartość wyrażenia zapisanego w ONP.

Przetwarzanie grafiki

Pseudokod dla algorytmu odpowiedzialnego za obróbkę zrzutu strefy rysowania w celu wysłania pojedynczych figur do identyfikacji przez sieć neuronową. Najpierw przeszukujemy zrzut strefy rysowania w celu znalezienia kolumn, w których znajdują się figury:

Data: Bitmapa zapisana w zmiennej *btm*
Result: Lista indeksów kolumn, w których znajdują się figury

Cols ← pusta lista liczb całkowitych;

for *j* ← 0 ***to*** *btm.width* **do**
    for *i* ← 0 ***to*** *btm.height* **do**
        Color ← wartość piksela w punkcie (j, i)
        if *Color nie jest biały* **then**
            Dodaj j do *Cols*;
        end
    end
end

Algorithm 4: Algorytm badający kolumny obrazu.

Następnie algorytm przeszukuje uzyskane przedziały kolumn w celu znalezienia punktów początkowych i końcowych w osi X i dodaje ich indeksy do list StartX oraz StopX. Kolejnym krokiem jest wywołanie części algorytmu przeszukującej wiersze, w których znajdują się figury, dla każdego przedziału z osobna. Działa to analogicznie do powyższego algorytmu i skutkuje pozyskaniem list indeksów wierszy StartY i StopY będących odpowiednio punktami początkowymi i końcowymi w osi Y.

Po uzyskaniu przedziałów dla obu osi możemy przystąpić do wycięcia figur ze zrzutu strefy rysowania oraz przetworzenia wyciętych obrazów do formy wymaganej przez wejście sieci. Pseudokod algorytmu przetwarzania:

Data: Zrzut obrazu ze strefy rysowania *btm* oraz listy
punktów cięcia $StartX,\ StartY,\ StopX,\ StopY$
Result: Cyfry i znaki arytmetyczne w postaci tablic liczb typu *double* - *digits*

digits ← pusta lista tablic liczb zmiennoprzecinkowych;

for $i = 0$ ***to*** $StartX.Length$ **do**
    width ← StopX[i] - StartX[i];
    height ← StopY[i] - StartY[i];
    if $width > 0$ ***and*** $height > 0$ **then**
        bmpCrop ← obraz wyznaczony przez punkty cięcia;
        if $bmpCrop.height < 0.15 \cdot btm.height$
        and $bmpCrop.width < 0.15 \cdot btm.height$ **then**
            Przypisz zmiennym *width* i *height* wartość $8 \cdot bmpCrop.height$;
        else
            if $bmpCrop.height > bmpCrop.width$ **then**
                Przypisz zmiennym *width* i *height* wartość $1.5 \cdot bmpCrop.height$;
            else
                Przypisz zmiennym *width* i *height* wartość $1.5 \cdot bmpCrop.width$;
            end
        end
        Naklej obraz *bmpCrop* na środek białego kwadratu o wym. *height* x *width*;
        Przeskaluj *bmpCrop* do wymiarów 28x28 pikseli;
        Zamień bitmapę *bmpCrop* na dwuwymiarową tablicę zmiennych typu double
        i dodaj ją do listy *digits*;
    end
end

Algorithm 5: Algorytm wycinania i przetwarzania obrazów.

Tak przetworzone obrazy w formie tablic mogą zostać użyte do klasyfikacji przez sieć neuronową.

Bazy danych

Nauka sieci neuronowej wykorzystywała dwie bazy danych.

Baza MNIST

THE MNIST DATABASE of handwritten digits to baza danych składająca się z 60 000 próbek treningowych oraz 10 000 próbek walidacyjnych. Dane są przechowywane w plikach zapisanych w formatach *.idx3-ubyte* (w przypadku samych obrazków) oraz *.idx1-ubyte* (w przypadku etykiet). Zawierają one wartości typu ubyte i 32-bit Integer. W przypadku plików przechowujących grafiki, pierwsze cztery wartości oznaczają kolejno:

- liczbę kontrolną zwaną „Magic number",
- liczbę zdjęć,
- szerokość jednego zdjęcia
- wysokość jednego zdjęcia

Kolejne liczby oznaczają wartości pikseli poszczególnych zdjęć. Przyjmują one wartości od 0 (które reprezentuje kolor biały) do 255 (reprezentacja koloru czarnego).

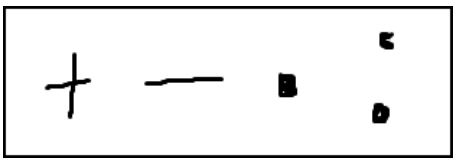
Pliki przechowujące etykiety zbudowane są na podobnej zasadzie - są to kolejno:

- liczba kontrolna „Magic number",
- liczba etykiet,
- kolejne etykiety (cyfry od 0 do 9).

Standardowe wczytywanie tej bazy danych polega na przetworzeniu pierwszych czterech wartości z pliku zawierającego zdjęcia oraz pierwszych dwóch z pliku zawierającego etykiety, a następnie wczytaniu do tablicy dwuwymiarowej 784 pikseli (grafiki mają wymiary 28 x 28) i odpowiadającej zdjęciu etykiety (dla jedynki będzie to 1 itd.). Takie dane są już gotowe do przesłania ich do sieci, jednak na potrzeby projektu wczytywanie bazy danych zostało przebudowane. Wykorzystywana jest tablica trójwymiarowa składająca się z czterech tablic dwuwymiarowych: tablicy zdjęć treningowych, tablicy etykiet treningowych, tablicy zdjęć walidacyjnych i tablicy etykiet walidacyjnych. W tablicach zdjęć wiersze składają się z 784 liczb, a każdy wiersz odpowiada jednemu obrazkowi. Z kolei etykiety zapisane są w postaci wierszy 14-kolumnowych składających się z samych zer i jednej jedynki na indeksie odpowiadającym tej liczbie.

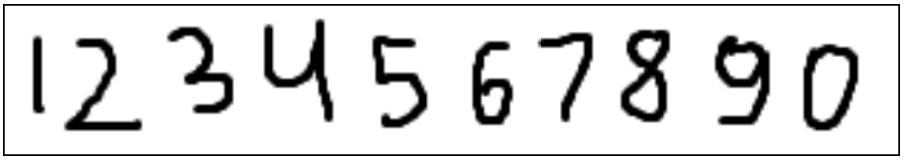
Autorska baza cyfr i oznaczeń matematycznych

Baza cyfr i oznaczeń matematycznych została zrobiona na potrzeby projektu. Zapisana jest w postaci kilku plików graficznych w formacie .png. Część z nich zawiera zestawy czterech znaków arytmetycznych jak na poniższym obrazku:



Jeden taki plik zawiera 50 zestawów znaków umieszczonych w jednym wierszu. Łącznie baza danych składa się z 1200 znaków arytmetycznych, z czego 10% jest traktowana jako część walidacyjna. Za pomocą napisanego algorytmu, każdy plik jest dzielony na 200 osobnych obrazków, których piksele są zapisywane w dwuwymiarowej tablicy tak samo jak w przypadku bazy danych MNIST. Z racji tego, że możliwe są tylko cztery etykiety, dodawane są naprzemiennie, gdyż kolejność oznaczeń jest taka sama (plus, minus, mnożenie, dzielenie). Występują w tablicy jako wiersze zer z jedynką na jednym z czterech ostatnich indeksów.

Pozostałe pliki składają się z zestawów dziesięciu cyfr:



W ich przypadku każdy z plików zawiera 15 zestawów umieszczonych w jednym wierszu. Łącznie baza danych składa się z 1200 cyfr, z czego 20% jest traktowana jako część walidacyjna. Tu ponownie wykorzystywany jest algorytm do dzielenia obrazu, a cyfry etykietowane są w postaci wierszy składających się z zer z jedną jeydnką na jednym z dziesięciu pierwszych indeksów.

Baza ta w całości (2400 znaków) jest dołączona do wczytanej bazy MNIST, a następnie tasowana w celu zwiększenia efektywności nauczania.

Implementacja

Informacje techniczne

Program został w całości napisany w języku C# przy użyciu środowiska programistycznego Visual Studio z wykorzystaniem technologii WPF odpowiedzialnej za graficzny interfejs aplikacji.

Podział na pliki

Program został stworzony w dwóch wersjach. Pierwsza z nich, nazywana „Neural Network - Learning Place", służy do uczenia sieci, zaś druga jest już właściwą aplikacją „Digit Recognizer". W poniższym opisie skupiono się na wersji głównej. Projekt podzielony jest na 9 plików źródłowych:

- Folder NeuralNetwork - w tym folderze znajdują się wszystkie pliki odpowiadające za działanie sieci neuronowej:
  - Data.cs - plik zawierający wszystkie metody odpowiadające za przygotowanie danych i ich wczytywanie,
  - Functions.cs - plik zawierający funkcje wykorzystywane przez neurony,
  - Layer.cs - plik zawierający klasę symulującą warstwę sieci neuronowej,
  - Network.cs - plik z główną klasą symulującą sieć neuronową,
  - Neuron.cs - plik z klasą symulującą neuron,
  - Synapse.cs - plik z klasą symulującą synapsę.
- Calculation.cs - plik z metodami odpowiedzialnymi za prawidłowe obliczenia; zawiera implementację Odwróconej Notacji Polskiej,
- DigitDetection.cs - plik zawierający przede wszystkim metody przetwarzające obraz ze „strefy rysowania".
- MainWindow.xaml.cs - plik zawierający graficzny interfejs aplikacji, implementację strefy rysowania wraz z metodą rysującą oraz inicjujący działanie sieci neuronowej.

Data.cs

Metody:

- `public static double[][][] PrepareDatasets(int MNISTDatasetSizeDivider)` - metoda ta odpowiedzialna jest za wczytanie baz danych i zwrócenia ich w postaci trójwymiarowej tablicy składającej się z czterech tablic wymiarowych. Są to kolejno: tablica danych wejściowych treningowych, tablica oczekiwanych danych wyjściowych treningowych, tablica danych wejściowych walidacyjnych i tablica oczekiwanych danych wyjściowych walidacyjnych. Na samym początku wczytane zostają pliki zawierające bazy danych. Ilość wczytywanych danych jest kontrolowana przez nas, ponieważ stwierdziliśmy, że wykorzystanie całej bazy MNIST nie jest konieczne do prawidłowego działania programu.

W tym celu używany jest jedyny argument tej funkcji - `MNISTDatasetSizeDivider`. Po wczytaniu danych są one dzielone na zestaw treningowy oraz walidacyjny, a następnie są tasowane za pomocą metody `Shuffle()`.

- `public static void LoadMNISTDataset(string imagesName, string labelsName, double[][] Images, double[][] Labels, int MNISTDatasetSizeDivider)` - metoda wczytująca fragment bazy danych MNIST. W celu wczytania pierwszych wartości, o których mowa w sekcji „Bazy danych", korzysta z pomocniczej metody `ReadBigInt32()`. Następnie wczytuje pozostałe liczby zawarte w plikach, konwertuje je na zmienne typu double i zapisuje w tablicy. Podobnie działa dla zapisanych w bazie etykiet.

- `private static void LoadOwnDatasets(double[][] trainImages, double[][] trainLabels, double[][] testImages, double[][] testLabels, string[] arithmeticFilePaths, string[] digitFilePaths, int MNISTDatasetSizeDivider)` - metoda ta jest odpowiedzialna za prawidłowe wczytanie naszej autorskiej bazy danych. Metoda działa zarówno dla cyfr, jak i znaków. Dla każdego zdjęcia wywoływana jest metoda konwertująca dane na tablice zmiennych typu double. Z powodu powtarzalnej sekwencji ułożenia cyfr i znaków arytmetycznych w bazie danych przypisywanie etykiet jest oparte na prostym działaniu modulo. Metoda ta operuje na wcześniej podanych tablicach zawierających dane walidayjne jak i dane treningowe, „doklejając" bazę autorską do wczytanej bazy MNIST, dlatego nie zwraca żadnych wartości.

- `public static void Shuffle(double[][] arr1, double[][] arr2)` - metoda tasująca elementy dwóch podanych tablic: zawierającej zdjęcia i zawierającej etykiety. Działanie wymaga dwóch tablic, aby zapobiec sytuacji, w której etykieta jest przypisana złemu obrazkowi i na odwrót - elementy z obu struktur są tasowane równocześnie.

- `public static double[][] BitmapToArray(Bitmap bitmap)` - metoda konwerująca bitmapę na tablicę dwuwymiarową. Każdemu pikselowi przypisywana jest wartość różnicy 255 oraz średniej arytmetycznej wartości RGB. Możliwa jest też pewna modyfikacja, zamieniająca wszystkie liczby dodatnie na liczbę 1. Na koniec zwracana jest tablica reprezentująca bitmapę (obraz).

- `public static List<double[]> RemoveSecondDimensions(List<double[][]> digits)` - metoda konwertująca tablicę dwuwymiarową na tablicę jednowymiarową. Jest to niezbędne do prawidłowego działania sieci neuronowej, gdyż sieć przyjmuje dane jako tablice jednowymiarowe.

- `public static class Extensions, public static int ReadBigInt32(this BinaryReader br)` - statyczna klasa zawierająca metodę odpowiadającą za prawidłowe wczytanie bazy MNIST. Bajty w bazie MNIST są zapisane w tzw. big-endian. Jest to forma zapisu, w której najbardziej znaczący bajt jest zapisany jako pierwszy. Jeśli więc pracujemy na procesorach Intela (które wykorzystują tzw. little-endian), musimy odwrócić bajty. Metoda ta sprawdza, czy jest potrzeba konwersji, a jeśli tak, to konwertuje big-endian na little-endian.

Spośród powyższych metod tylko dwie są wykorzystywane w głównym projekcie: `BitmapToArray` oraz `RemoveSecondDimensions`. Pozostałe są zaimplementowane jedynie w wersji „Learning Place".

Functions.cs

Klasa ta zawiera wszystkie funkcje matematyczne wykorzystywane w programie przez sieć neuronową. Posiada właściwość `public double Alpha` będącą współczynnikiem korekcji wykorzystywanym podczas liczenia wartości funkcji aktywacji oraz jej pochodnej; jest on ustawiony na 0.8. Klasa zawiera następujące metody:

- `public static double InputSumFunction(List<Synapse> Inputs, double bias = 0)` - metoda zwracająca sumę wartości zwróconych przez synapsy wejściowe danego neuronu. Do końcowego wyniku dodawany jest również bias (domyślnie ustawiony na 0),
- `public static double BipolarLinearFunction(double input)` - metoda zwracająca wartość funkcji aktywacji (funkcja bipolarna liniowa). Jako argument przyjmuje wartość zwróconą przez metodę `InputSumFunction()`,
- `public static double BipolarDifferential(double input)` - metoda zwracająca wartość pochodnej funkcji aktywacji. Wykorzystywana jest w algorytmie wstecznej propagacji,
- `public static double CalculateError(List<double> outputs, int row, double[][] expectedresults)` - metoda odpowiedzialna za wyznaczenie błędu średniokwadratowego na podstawie uzyskanego wektora wyjściowego i oczekiwanego wektora wyjściowego. Funkcja ta jest używana jedynie do celów testowych podczas trenowania.

Neuron.cs

Klasa reprezentująca neuron sieci neuronowej. Jej polami/właściwościami są: lista synaps wchodzących (`public List<Synapse> Inputs`), lista synaps wychodzących (`public List<Synapse > Outputs`), wartość na wejściu (`public double InputValue`) oraz wartość na wyjściu (`public double OutputValue`). Zawiera następujące metody:

- `public Neuron()` - konstruktor klasy, który inicjuje puste listy synaps.
- `public void AddOutputNeuron(Neuron outputneuron)` - metoda, która łączy neuron bieżący z innym za pomocą synapsy. Tworzona jest nowa synapsa, która następnie jest dodawana do listy synaps wychodzących bieżącego neuronu oraz do listy synaps wchodzących neuronu następnego.
- `public void AddInputSynapse(double input)` - metoda, która dodaje synapsę wejściową do neuronu bez łączenia z innym neuronem. Występuje jedynie w wejściowej warstwie sieci. Podobnie jak pozostałe synapsy, jest dodawana do listy synaps wchodzących.
- `public void CalculateOutput()` - metoda przypisująca właściwości `InputValue` wynik funkcji `InputSumFunction()` oraz właściwości `OutputValue` wynik funkcji `BipolarLinearFunction ()`.
- `public void PushValueOnInput(double input)` - metoda, która przypisuje podany argument zmiennej `PushedData` synapsie wejściowej pierwszej warstwy.

Synapse.cs

Klasa reprezentująca synapsę sieci neuronowej. Zawiera ona zmienną `static readonly Random tmp` potrzebną do generowania liczby pseudolosowej, dwie zmienne klasy Neuron (`internal Neuron FromNeuron, ToNeuron`) reprezentujące, od którego do którego neuronu synapsa została połączona, a także właściwości `public double Weight`, przechowującą wagę synapsy, oraz `public double PushedData`, która przechowuje wartości podane synapsie (jeśli jest to pierwsza warstwa). Klasa zawiera następujące metody:

- `public Synapse(Neuron fromneuron, Neuron toneuron)` - konstruktor przyjmujący za argumenty dwa neurony, które mają zostać połączone daną synapsą. W konstruktorze zmiennym `FromNeuron` i `ToNeuron` przypisywane są wartości podane jako argumenty, a właściwości `Weight` przypisywana jest wartość losowana przy użyciu zmiennej `tmp`.

- `public Synapse(Neuron toNeuron, double output)` - konstruktor, który przyjmuje za argument neuron oraz dane liczbowe); wykorzystywany przy pierwszej warstwie , z tego samego powodu waga jest ustawiona na wartość 1.

- `public double GetOutput()` - metoda zwracająca wynik z danej synapsy. Metoda sprawdza, czy wyowałana została z pierwszej warstwy - jeśli tak, to zwraca niezmienione dane, jeśli nie, to zmienna `OutputValue` ze zmiennej `FromNeuron` jest mnożona razy wagę.

Layer.cs

Klasa ta definiuje warstwę zaimplementowanej sieci neuronowej. Jej jedynym polem/właściwością jest `public List<Neuron> Neurons`, czyli lista neuronów wchodzących w skład warstwy. Klasa posiada następujące metody:

- `public Layer(int numberofneurons)` - konstruktor klasy, inicjuje on listę Neuronów, dodając do niej podaną jako argument liczbę neuronów,

- `public void ConnectLayers(Layer outputlayer)` - metoda, która łączy wszystkie neurony w warstwie z innymi neuronami w innej warstwie poprzez wielokrotne wywołanie metody `AddOutputNeuron` z klasy `Neuron`.

- `public void CalculateOutputLayer()` - metoda wywołująca funkcję `CalculateOutput()` z klasy `Neuron` dla wszystkich neuronów z listy.

Network.cs

Klasa reprezentująca sieć neuronową. Klasa zawiera statyczną zmienną `static readonly double LearningRate`, która reprezentuje wielkość kroku w każdej iteracji podczas osiągania minimum tzw. funkcji celu (ang. loss function) - jest wykorzystywana w algorytmie wstecznej propagacji. Jej wartość ustawiono na 0.05. Następna zmienna, `static double SynapsesCount`, przechowuje informację o tym, ile synaps liczy cała sieć neuronowa. Lista `internal List<Layer> Layers` przechowuje wszystkie zdefiniowane warstwy sieci. Tablica dwuwymiarowa `internal double[][] ExpectedResults` przechowuje etykiety. a `double[][] ErrorFunctionChanges` przechowuje wartości zmiany funkcji błędu δ dla każdego neuronu w sieci. Zmienna `LearningRate` oraz obie tablice występują jedynie w wersji „Learning Place”. Metody w klasie:

- `public Network(double alpha, int inputneuronscount, int[] hiddenlayerssizes, int outputneuronscount)` - konstruktor sieci neuronowej. Na samym początku sprawdzana jest poprawność podanych argumentów, które później przypisywane są do pól. Następnie za pomocą metod `AddFirstLayer()` i `AddNextLayer()` tworzone są wszystkie warstwy sieci.

- `private void AddFirstLayer(int inputneuronscount)` - prywatna metoda, która tworzy pierwszą warstwę sieci neuronowej, zawierającą `inputneuronscount` neuronów. Następnie za pomocą pętli do każdego neuronu w warstwie dodawana jest synapsa wejściowa z wartością 0 za pomocą funkcji `AddInputSynapse()`, a sama warstwa jest dodawana do listy warstw sieci `Layers`.

- `private void AddNextLayer(Layer newLayer)` - metoda łącząca ostatnią warstwę sieci z kolejną warstwą neuronów. Nowa warstwa jest dodawana do listy `Layers`.

- `public void PushInputValues(double[] inputs)` - metoda przekazująca sieci neuronowej dane wejściowe. Metoda sprawdza, czy dane są poprawne tzn. czy wektor wejściowy ma taki sam rozmiar co pierwsza warstwa sieci, po czym umieszcza je w synapsach wejściowych neuronów pierwszej warstwy za pomocą funkcji `PushValueOnInput()` klasy `Neuron`.

- `public void PushExpectedValues(double[][] expectedvalues)` - metoda przekazująca sieci neuronowej poprawne dane wyjściowe. Po sprawdzeniu poprawności danych, metoda przypisuje zmiennej prywatnej `ExpectedResults` podany argument.

- `public List<double> GetOutput()` - metoda zwracająca uzyskane dane wyjściowe w postaci listy liczb typu `double`. Po zainicjowaniu listy `output` na każdej z warstw zostaje wykonana metoda `CalculateOutputOnLayer()` z klasy `Layer`, aktywująca neurony poszczególnych warstw w sposób sekwencyjny. Dzięki temu neurony na warstwie wyjściowej będą miały przypisaną aktualną wartość `OutputValue`. Wartości każdego neuronu z tej warstwy zostają dodane do listy `output`, po czym jest ona zwracana.

- `public void Train(double[][][] datasets, double epochscount, bool showinfo = false, bool breaking = false)` - metoda trenująca sieć neuronową, przyjmująca za argument m.in. zestaw 4 tablic dwuwymiarowych (treningowych i testowych). Po wywołaniu metody `PushExpectedValues()` wykonywana jest `epochscount` razy pętla odpowiadająca za właściwe trenowanie sieci. W każdej iteracji do sieci wysyłane są wszystkie treningowe wektory wejściowe, do zmiennej `outputs` przypisywane są uzyskane dane wyjściowe, a następnie zostaje wywołana metoda `ChangeWeights()`. Liczony jest też błąd średniokwadratowy przy użyciu funkcji `CalculateMeanSquareError()`. Jest on wypisywany na ekranie tylko wtedy, gdy podany argument `showinfo` ma ustawioną wartość `true`. Ponadto istnieje dodatkowe zabezpieczenie, które sprawdza, czy występuje tzw. overfitting, czyli przeuczenie sieci. Jeśli metoda wykryje przeuczenie, to kończy swoje działanie. Zabezpieczenie to jest aktywne tylko wtedy, gdy podany argument `breaking` ma ustawioną wartość `true`. Po zakończeniu uczenia wagi każdej synapsy są zapisywane do pliku `weights.txt` za pomocą metody `SaveNetworkToFile()`.

- `private double CalculateMeanSquareError(double[][] inputs, double[][] expectedoutputs, bool showerror = false)` - metoda licząca i zwracająca średnią wartość błędu średniokwadratowego dla podanego zbioru testowego przy użyciu funkcji `CalculateError()` z klasy `Functions`. Błąd jest wypisywany (o ile zmienna `showerror` ma wartość `true`) z dokładnością do 5 miejsc po przecinku.

- `private void ChangeWeights(List<double> outputs, int row)` - metoda modyfikująca wagi wszystkich synaps sieci neuronowej zgodnie z algorytmem wstecznej propagacji (opisanym w sekcji „Opis działania sieci neuronowej"). Po zaktualizowaniu tablicy `ErrorFunctionChanges` przy użyciu metody `CalculateErrorFunctionChanges()` dla każdej synapsy wejściowej każdego neuronu każdej warstwy (poza pierwszą) zaczynając od wyjściowej waga zostaje zmieniona zgodnie ze wzorem: $w_{ij}^k = w_{ij}^k + LearningRate \cdot 2 \cdot \delta_i^k \cdot y_j^{k-1}$.

- `private void CalculateErrorFunctionChanges(List<double> outputs, int row)` - metoda aktualizująca tablicę `ErrorFunctionChanges`. Na początku uzupełniana jest kolumna odpowiadająca ostatniej warstwie sieci - przebiega to zgodnie ze wzorem $\delta_i^K = f'(s_i^K) \cdot (d_i^K - y_i^K)$ - a następnie kolumny odpowiadające wszystkim warstwom ukrytym zaczynając od końca zgodnie ze wzorem: $\delta_i^k = f'(s_i^k) \sum_{j=1}^{N_{k+1}} \delta_j^{k+1} w_{ij}^{k+1}$.

- `private void SaveNetworkToFile(string path)` - prosta metoda zapisująca wagi synaps do pliku, którego ścieżkę podano jako argument. Kilka pętli przeskakuje przez wszystkie warstwy, neurony i synapsy, zapisując wagi do zainicjowanej wcześniej listy `tmp`. Zapisywane są również ustawienia sieci takie jak zmienna `alpha` i liczba neuronów w każdej warstwie - znajdują się one na samym początku listy. Następnie lista zapisywana jest do pliku.

- `public static Network LoadNetworkFromFile(string path)` - metoda wczytująca sieć neuronową z pliku, którego ścieżkę podano jako argument. Metoda tworzy nową sieć neuronową, podając w konstruktorze dane wczytane z podanego pliku, a następnie pętle przechodzą przez wszystkie warstwy, neurony i synapsy, wczytując wagi. Metoda zwraca obiekt klasy `Network`.

- `public void CalculatePrecision(double[][][] datasets, bool shownumbers = false)` - metoda sprawdzająca precyzję sieci neuronowej. Do sieci wprowadzane są testowe wektory wejściowe, a uzyskane wartości wyjściowe sprawdzane są z oczekiwanymi etykietami z wykorzystaniem pomocniczej metody `Classify()`. Na końcu wyświetlany jest procent poprawnych odpowiedzi zaokrąglony do czterech miejsc po przecinku.

- `public void Classify(double[][] testingOutputs, List<double> trueOutputs)` - pomocnicza metoda wyświetlająca oczekiwane etykiety oraz dane wyjściowe zwrócone przez sieć neuronową.

- `private double CountSynapse()` - metoda licząca i zwracająca liczbę synaps w sieci neuronowej.

Wiele z powyższych metod jest wykorzystywanych jedynie w wersji „Learning Place" - są to metody związane z uczeniem i klasyfikacją: `Train`, `CalculateErrorFunctionChanges`, `ChangeWeights`, `CalculateMeanSquareError`, `CalculatePrecision`, `PushExpectedValues`, `Classify`, `SaveNetworkToFile`.

Calculation.cs

Klasa ta zawiera implementację ONP i jest używana jedynie w głównej wersji programu. Jej metody to:

- `public static string Calculate(string equation)` - metoda przyjmująca działanie zapisane w zmiennej tekstowej, konwertująca na zapis ONP i zwracająca wynik wyrażenia. Korzysta z trzech metod pomocniczych: `toRPN()`, `ConvertToStack()` oraz `evalRPN()`. Na samym początku jednak sprawdzane jest, czy działanie jest poprawnie zapisane tzn. czy nie kończy i nie zaczyna się na znaku arytmetycznym i czy nie nastąpiło dzielenie przez zero.

- `public static string toRPN(string token)` - algorytm zmieniający zapis normalny na zapis w odwróconej notacji polskiej. Każde działanie jest zdefiniowane w słowniku i przypisana zostaje mu odpowiednia waga (dodawaniu i odejmowaniu - 1; mnożeniu i dzieleniu - 2). Następnie tworzona jest zmienna reprezentująca stos. Pętla przeskakuje po każdym znaku w zmiennej i próbuje ją przekonwertować na typ `double`. Jeśli operacja powiedzie się, to znaczy, cyfra dodawana jest do zmiennej `result`. Jeśli natomiast nastąpi błąd, znak arytmetyczny dodawany jest do stosu.

- `public static Stack<string> ConvertToStack(string tokens)` - prosta metoda pomocnicza, która konwertuje typ `string` na `Stack`. Jest to potrzebne do prawidłowego i łatwego korzystania z innych metod tej klasy.

- `public static double evalRPN(Stack<string> tokens)` - metoda wyliczająca wynik z odwróconej notacji polskiej. Przyjmuje za argument stos, a zwraca typ `double`. Metoda jest wykonywana rekurencyjnie, gdzie sprawdzane są kolejne znaki arytmetyczne i na tej zasadzie liczby są dodawane, odejmowane, dzielone lub mnożone.

DigitDetection.cs

- `private static List<int> ColumnSearch(Bitmap btm)` - metoda odpowiedzialna jest za przeszukiwanie bitmapy w osi X w poszukiwaniu kolumn, w których znajdują się znaki/cyfry. Wykrywanie zrealizowane jest za pomocą funkcji `GetPixel()`, która zwraca wartości RGB pikseli. Dwie pętle przeszukują każdą kolumnę od góry do dołu. W przypadku, w którym zostanie wykryty piksel, który nie jest biały, dana kolumna zostaje dodana do listy. Funkcja zwraca listę zawierającą indeksy tych kolumn.

- `private static List<int> RowSearch(List<int> StartX, List<int> StopX, Bitmap btm, int digit)` - metoda ta działa podobnie do `ColumnSearch()`, lecz skupia się na przeszukiwaniu wierszy w każdym z przedziałów kolumn, w których wykryto znaki. Wywoływana jest dla każdego przedziału z osobna i zwraca listę indeksów wierszy, w których znajduje się znak dla konkretnego przedziału.

- `private static List<double[][]> IntervalsCounting(List<int> columnsWithBlackPoints, Bitmap btm)` - metoda, która wyznacza punkty kluczowe dla wycinania cyfr i znaków, zwraca wartość zwracaną przez `VericalCropping()`.

- `private static List<double[][]> VerticalCropping(List<int> StartX, List<int> StopX, List<int> StartY, List<int> StopY, Bitmap btm)` - metoda ta kopiuje obszary ograniczone przez obliczone wcześniej punkty kluczowe i, wywołując metody `ResizeImage()` oraz `TransformToSquare()`, finalnie zwraca listę tablic dwuwymiarowych typu `double` odpowiadających wykrytym znakom.

- `ResizeImage(Image image)` - metoda skalująca obraz do wymiarów 28x28 pikseli w celu prawidłowego działania sieci neuronowej. Kwadratowy obraz znaku na końcu jest przetwarzany przez funkcję `BitmapToArray()` z klasy `Data` i zwracany w postaci tablicy dwuwymiarowej typu `double`.

- `TransformToSquare(Bitmap bmpCrop, Bitmap btm)` - metoda generuje kwadratową płaszczyznę z białym tłem na podstawie wymiarów przesłanej bitmapy, a następnie umieszcza ją w jej centrum - płaszczyzna i bitmapa są nakładane na siebie. Metoda przewiduje dodatkowe warunki generowania płaszczyzny dla mnożenia oraz odejmowania i zwraca przetransformowane obrazy.

- `public static List<double[][]> DetectDigits(MemoryStream picture)` - główna metoda wywołująca całą sekwencję. Jej argumentem jest zapisany w pamięci obraz, zwraca listę cyfr w postaci tablic dwuwymiarowych. Metoda ta występuje jedynie w głównej wersji projektu i jest wykorzystywana do analizy „strefy rysowania".

- `public static List<double[][]> DetectDigits(Bitmap picture)` - inny wariant powyższej metody. Tym razem argumentem jest bitmapa. Metoda ta jest wykorzystywana w wersji „Learning Place" podczas przetwarzania graficznej bazy danych.

- `public static string RecognizeDigits(List<double[]> digits, Network network)` - metoda przepuszczająca przez podaną w argumencie sieć kolejne tablice przekonwertowanych obrazów i zwracająca uzyskany od sieci ciąg znaków w postaci zmiennej typu `string`.

MainWindow.xaml.cs

- `private void CanvasMouseDown(object sender, MouseButtonEventArgs e)` - metoda śledząca pozycję myszy w czasie, kiedy lewy przycisk jest wciśnięty.

- `private void CanvasMouseMove(object sender, MouseEventArgs e)` - metoda śledząca pozycję myszy w trakcie przesuwania i rysująca linię pomiędzy punktem początkowym - z momentu wywołania - a nowym - po przesunięciu. Pomiędzy tymi punktami rysowana jest linia, co w efekcie ciągłego wywoływania metody w trakcie przesuwania myszy daje linię wzdłuż toru ruchu kursora.

- `private void LaunchNetworkButtonClick(object sender, RoutedEventArgs e)` - metoda wywoływana w trakcie naciśnięcia przycisku „Uruchom sieć neuronową". Powoduje załadowanie parametrów sieci z pliku. Jeśli operacja ładowania wag nie powiedzie się, pozostałe dwa przyciski nadal będą nieaktywne.

- `private void ClearButtonClick(object sender, RoutedEventArgs e)` - metoda wywoływana w trakcie naciśnięcia przycisku „Wyczyść". Powoduje odświeżenie strefy rysowania i usunięcie istniejących na niej linii.

- `private void CalculateButtonClick(object sender, RoutedEventArgs e)` - metoda wywoływana w trakcie naciśnięcia przycisku „Oblicz”. Powoduje wysłanie grafiki strefy rysowania do wycięcia oraz uruchamia klasyfikację. Otrzymany wynik wyświetlany jest w oknie aplikacji.

- `private MemoryStream SaveCanvas(Canvas canvas)` - zapisuje grafikę ze strefy rysowania w pamięci RAM.

Testy

Uczenie sieci

Trenowanie sieci z wykorzystaniem bazy MNIST już od samego początku było bardzo efektywne, a dokładność zaimplementowanej sztucznej sieci neuronowej wynosiła powyżej 90%. Mimo to skuteczność rozpoznawania cyfr pisanych w „strefie rysowania” nie była wystarczająco wysoka, dlatego dodano bazę autorską i przetestowano kilka konfiguracji sieci. Przy wykorzystaniu 1% bazy MNIST i ograniczonej bazie autorskiej uzyskano następujące wyniki:

L. w. ukrytych	L. neuronów	LR	α	Błąd średniokwadratowy
2	100	0.05	0.5	0.207
2	100	0.05	0.75	0.186

W powyższej tabeli LR oznacza współczynnik nauczania wykorzystywany w algorytmie wstecznej propagacji, a α to współczynnik korekcji wykorzystywany w funkcji aktywacji w neuronach. Błąd średniokwadratowy jest liczony na podstawie wzoru $E_1 = \sum_{i=1}^{n}(d_i - y_i)^2$, gdzie n to wymiar wektora wyjściowego / liczba neuronów wyjściowych, d to wartość oczekiwana, a y to wartość uzyskana (na i-tej pozycji wektora wyjściowego). W tabeli zapisano średnią błędu wszystkich elementów należących do zbioru testowego.

Już po pierwszych dwóch testach zauważono, że wzrost współczynnika α działa pozytywnie na zmniejszenie się błędu średniokwadratowego, dlatego zdecydowano go zwiększyć do poziomu 0.8. Zdecydowano także zwiększyć wykorzystanie bazy MNIST do 2%. Uzyskano następujące wyniki:

L. w. ukrytych	L. neuronów	LR	α	Błąd średniokwadratowy
2	100	0.05	0.8	0.198
3	100	0.05	0.8	0.156
4	100	0.05	0.8	0.143

Następnie zdecydowano się powiększyć dwukrotnie autorską bazę danych i ostateczne osiągi sztucznej sieci neuronowej wyniosły:

L. w. ukrytych	L. neuronów	LR	α	Błąd średniokwadratowy
4	100	0.05	0.8	0.12

Niewątpliwie sieć można by było uczynić bardziej dokładną szukając lepszej konfiguracji sieci, jednak zdecydowano, że dalsze modyfikacje będą dotyczyły sposobu zapisu liczb w tablicy dwuwymiarowej. Mianowicie obrazki przekonwertowane na tablice liczb zmiennoprzecinkowych typu double z przedziału 0-255 zostały zmienione na tablice zer i jedynek w taki sposób, że liczby poniżej 10 zmieniono na zera, a liczby większe lub równe 10 na jedynki. Modyfikacja ta wpłynęła pozytywnie na rozpoznawanie większości cyfr.

Napotkane błędy i ich eliminacja

- **Niepoprawne skalowanie dla znaku mnożenia** - program, wycinając znak w celu zmiany jego rozmiaru, wkleja go na wygenerowane pole zależne od dłuższej krawędzi wycinka. Metoda ta wymagała dołożenia osobnego warunku skalowania dla znaku „*", gdyż użycie standardowego dla innych cyfr i znaków mnożnika 1.5, powodowało rozciągnięcie tego znaku na cały obszar, co uniemożliwiało jego poprawne rozpoznanie. W celu eliminacji błędu mnożnik dla znaków o rozmiarach mniejszych niż 10 procent wysokości pola rysowania ustawiono na wartość 8.

- **Wykrywanie przez program pojedynczych pikseli oderwanych od figury** - pojawianie się zagubionych pikseli wynikało z niedoskonałości linii powstałej w strefie rysowania. Program, wykrywając piksel, interpretował go jako znak mnożenia, co doprowadzało do wyświetlenia komunikatu o błędnym zapisie - dwa znaki obok siebie. Rozwiązaniem tego problemu okazało się zwiększenie minimalnego odstępu pomiędzy figurami z 1 do 3 pikseli.

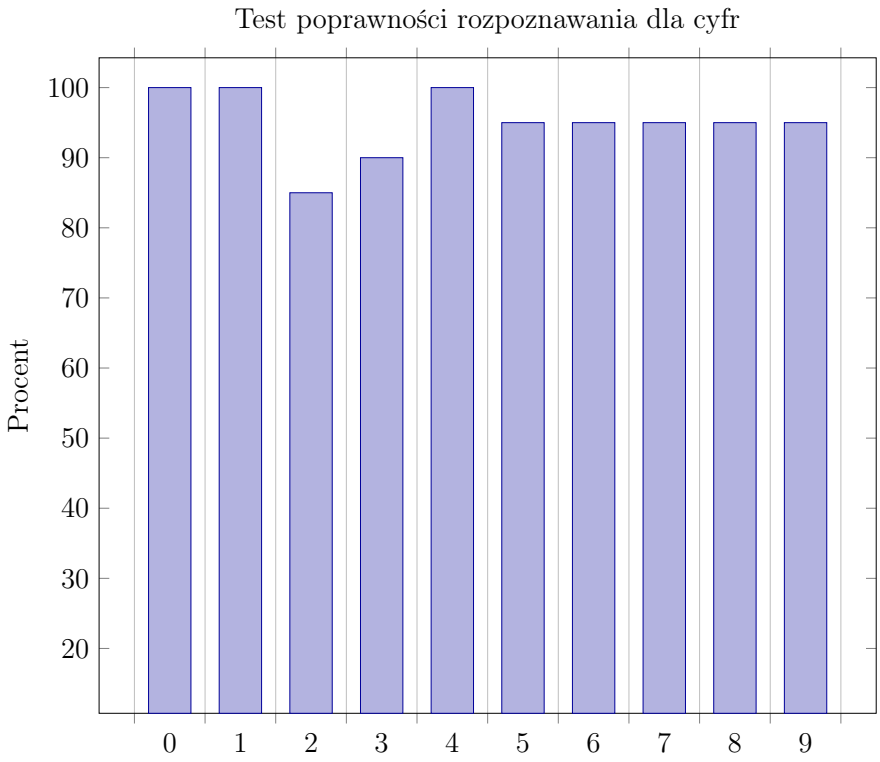
Inne napotkane problemy aplikacji i optymalizacja działania

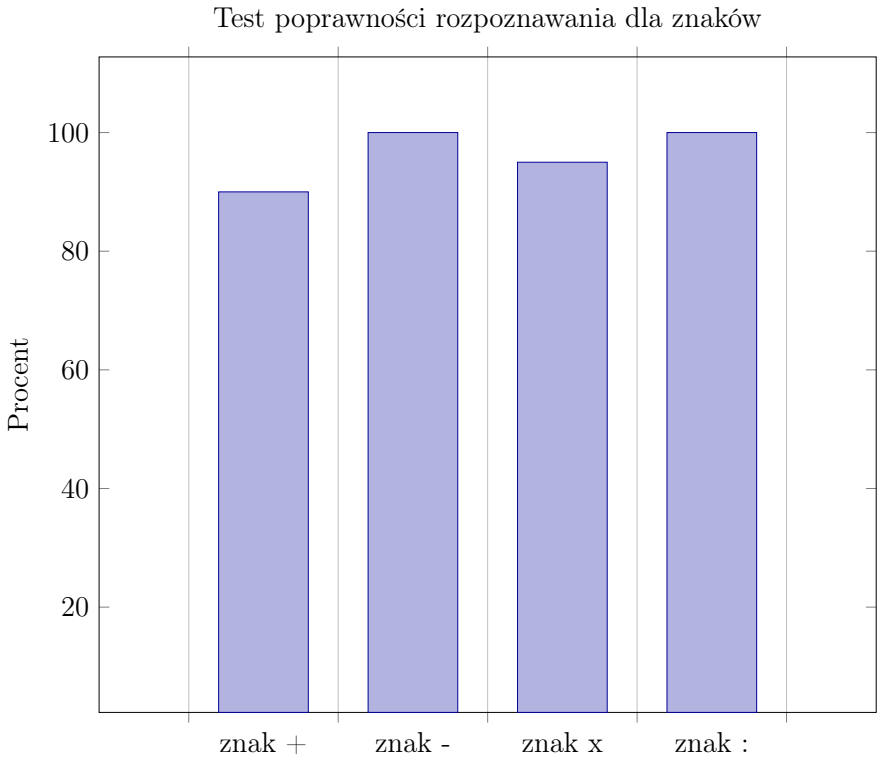
- **Duże wykorzystanie pamięci przez bitmapy** - algorytm wycinający wykorzystuje bitmapy, które przesyłane są pomiędzy kolejnymi funkcjami programu. W celu zminimalizowania zużycia program zmniejszono wykorzystanie bitmap do minimum i są one generowane tylko w miejscach, gdzie jest to konieczne. Pozwala to uniknąć przepełniania pamięci w przypadku dużej ilości figur wprowadzonej do strefy rysowania.

- **Duża złożoność obliczeniowo-czasowa podczas uruchamiania sieci neuronowej** - początkowo wciśnięcie przycisku „Uruchom sieć neuronową" skutkowało pobraniem całej bazy danych, zbudowaniem na jej podstawie sieci, wczytaniem wag z pliku i wyliczeniem dokładności - trwało to nawet kilkanaście sekund; zrezygnowano z tego pomysłu umieszczając w pierwszej linii pliku z wagami informację o budowie sieci, dzięki czemu bazy danych nie są już potrzebne aplikacji, a uruchamianie trwa ułamek sekundy.

Test aplikacji

Przeprowadzone przez nas testy polegały na wprowadzeniu do programu w celu rozpoznania kolejno każdego z 14 znaków dwudziestokrotnie. Zestawy wprowadzane były na bieżąco do strefy rysowania, a następnie dane wyjściowe porównywano z danymi wejściowymi. Sprawdzana była w ten sposób zdolność identyfikacji wprowadzonej figury oraz poprawność obliczeniowa programu. Na następnej stronie znajduje się wykres poprawności rozpoznania dla cyfr i znaków.





Zdolność obliczeniowa aplikacji okazała się w stu procentach precyzyjna, a sama aplikacja odporna na błędy logiczne. Ponadto zdolność rozpoznawania znaków również okazała się bardzo precyzyjna (błędy zdarzały się sporadycznie).

Pełen kod aplikacji

Digit Recognizer

MainWindow.xaml - szata graficzna

```
<Window x:Class="DigitRecognizer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        mc:Ignorable="d"
        Title="Digit Recognizer" Height="435" Width="800" ResizeMode="NoResize" Background="LightSkyBlue">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*" />
            <RowDefinition Height="1*"/>
            <RowDefinition Height="1*"/>
        </Grid.RowDefinitions>

        <Canvas x:Name="PaintSurface" Grid.Row="0" MouseDown="CanvasMouseDown" MouseMove="CanvasMouseMove" Height="200" Width="800"
                    HorizontalAlignment="Stretch" VerticalAlignment="Stretch">
            <Canvas.Background>
                <SolidColorBrush Opacity="100" Color="White" />
            </Canvas.Background>
        </Canvas>

        <TextBlock x:Name="MathTextBox" Text="WPROWADZ DZIALANIE W STREFIE RYSOWANIA" FontSize="20"
                   Margin="20" Height="30" TextAlignment="Center" Grid.Row="1" Background="White"/>

        <WrapPanel HorizontalAlignment="Center" Grid.Row="2">
            <Button x:Name="LaunchButton" Click="LaunchNetworkButtonClick" Content="Uruchom siec neuronowa" Width="160" Height="30" Margin="20" Background="White" />
            <Button x:Name="ClearButton" Click="ClearButtonClick" Content="Wyczysc" Width="150" Height="30" Margin="20"
                    IsEnabled="False" Background="White"/>
            <Button x:Name="CalculateButton" Click="CalculateButtonClick" Content="Oblicz" Width="150" Height="30" Margin="20"
                    IsEnabled="False" Background="White"/>
        </WrapPanel>
    </Grid>
</Window>
```

MainWindow.xaml.cs - interfejs graficzny

```
1  using System.IO;
2  using System.Windows;
3  using System.Windows.Controls;
4  using System.Windows.Input;
5  using System.Windows.Media;
6  using System.Windows.Media.Imaging;
7  using System.Windows.Shapes;
8  using System.Collections.Generic;
9  using NeuralNetwork;
10 //using System.Diagnostics;
11
12 namespace DigitRecognizer
13 {
14     public partial class MainWindow : Window
15     {
16         Point CurrentPoint = new Point();
17         Network network;
18
19         public MainWindow()
20         {
21             InitializeComponent();
22         }
23
24         #region Rysowanie
25         private void CanvasMouseDown(object sender, MouseButtonEventArgs
               e)
26         {
27             if (e.ButtonState == MouseButtonState.Pressed)
28                 CurrentPoint = e.GetPosition(this);
29         }
30
31         private void CanvasMouseMove(object sender, MouseEventArgs e)
32         {
33             if (e.LeftButton == MouseButtonState.Pressed)
34             {
35                 Line line = new Line
36                 {
37                     StrokeThickness = 4,
38                     Stroke = SystemColors.WindowTextBrush,
39                     X1 = CurrentPoint.X,
40                     Y1 = CurrentPoint.Y,
41                     X2 = e.GetPosition(this).X,
42                     Y2 = e.GetPosition(this).Y
43                 };
44
45                 CurrentPoint = e.GetPosition(this);
46                 PaintSurface.Children.Add(line);
47             }
48         }
49         #endregion
50
51
52
```

```
53          #region Przyciski
54          private void LaunchNetworkButtonClick(object sender,
                RoutedEventArgs e)
55          {
56              try
57              {
58                  network = Network.LoadNetworkFromFile("weights.txt");
59                  LaunchButton.IsEnabled = false;
60                  CalculateButton.IsEnabled = true;
61                  ClearButton.IsEnabled = true;
62                  MathTextBox.Text = "SIEC GOTOWA!";
63              }
64              catch { MathTextBox.Text = "NIE ZNALEZIONO PRAWIDLOWEGO
                    PLIKU Z WAGAMI!"; }
65          }
66
67          private void ClearButtonClick(object sender, RoutedEventArgs e)
68          {
69              PaintSurface.Children.Clear();
70              MathTextBox.Text = "WPROWADZ NOWE DZIALANIE W STREFIE
                    RYSOWANIA";
71          }
72
73          private void CalculateButtonClick(object sender, RoutedEventArgs
                e)
74          {
75              var picture = SaveCanvas(PaintSurface);       // Zapisuje
                    canvas w pamieci
76              var DigitsInTwoDimensions = DigitDetection.DetectDigits(
                    picture); // Wywolanie kolejnych funkcji do wycinania i
                    obrobki wczytanych cyfr, znakow
77
78              List<double[]> digits = Data.RemoveSecondDimensions(
                    DigitsInTwoDimensions); // Przygotowanie pod karmienie
                    sieci
79
80              string tmp = DigitDetection.RecognizeDigits(digits, network)
                    ;
81              if (tmp != "") MathTextBox.Text = tmp;
82              string result = Calculation.Calculate(tmp).ToString();
83              if (result == "BLEDNY ZAPIS!" || result == "NIE MOZNA
                    DZIELIC PRZEZ ZERO!") MathTextBox.Text = result;
84              else MathTextBox.Text += result;
85          }
86          #endregion
87
88          // Zapis Canvas:
89          private MemoryStream SaveCanvas(Canvas canvas)
90          {
91              RenderTargetBitmap renderBitmap = new RenderTargetBitmap((
                    int)canvas.Width, (int)canvas.Height, 96d, 96d,
                    PixelFormats.Pbgra32);
92
93              canvas.Measure(new Size((int)canvas.Width, (int)canvas.
                    Height));
```

```
94              canvas.Arrange(new Rect(new Size((int)canvas.Width, (int)
                    canvas.Height)));
95              renderBitmap.Render(canvas);
96              JpegBitmapEncoder encoder = new JpegBitmapEncoder();
97              encoder.Frames.Add(BitmapFrame.Create(renderBitmap));
98
99              MemoryStream stream = new MemoryStream();
100             encoder.Save(stream);
101
102             return stream;
103         }
104     }
105 }
```

DigitDetection.cs - wyodrębnianie i obróbka liczb z obrazów

```
1  using System.Collections.Generic;
2  using System.Drawing;
3  using System.Drawing.Drawing2D;
4  using System.Drawing.Imaging;
5  using System.IO;
6  using System.Linq;
7  using System.Diagnostics;
8  using NeuralNetwork;
9
10 namespace DigitRecognizer
11 {
12     class DigitDetection
13     {
14         // Przeszukuje kolumny w celu znalezienia punktow innych niz
               biale:
15         private static List<int> ColumnSearch(Bitmap btm)
16         {
17             List<int> Cols = new List<int>();
18             Color color;
19             for (int j = 0; j < btm.Width; j++)
20                 for (int i = 0; i < btm.Height; i++)
21                 {
22                     color = btm.GetPixel(j, i);
23                     if (color != Color.FromArgb(255, 255, 255))
24                     {
25                         Cols.Add(j);
26                         break;
27                     }
28                 }
29             return Cols;
30         }
31
32
33         //Przeszukuje przedzialy znalezione przez ColumnSearch w
               poszukiwaniu punktow innych niz biale:
34         private static List<int> RowSearch(List<int> StartX, List<int>
               StopX, Bitmap btm, int digit)
```

```
35          {
36              List<int> Rows = new List<int>();
37              Color color;
38              for (int k = 0; k < btm.Height; k++)
39                  for (int j = StartX[digit]; j < StopX[digit]; j++)
40                  {
41                      color = btm.GetPixel(j, k);
42                      if (color != Color.FromArgb(255, 255, 255))
43                      {
44                          Rows.Add(k);
45                          break;
46                      }
47                  }
48
49              if (Rows.Count == 0) //Zabezpieczenie
50              {
51                  Rows.Add(0);
52                  Rows.Add(btm.Height);
53              }
54
55              return Rows;
56          }
57
58
59          private static List<double[][]> IntervalsCounting(List<int>
                columnsWithBlackPoints, Bitmap btm)
60          {
61              if (columnsWithBlackPoints.Count == 0)
62                  return new List<double[][]>();
63
64              //Przedzialy miedzy kolumnami
65              List<int> StartX = new List<int>();
66              List<int> StopX = new List<int>();
67
68              StartX.Add(columnsWithBlackPoints[0]);
69              for (int i = 1; i < columnsWithBlackPoints.Count - 1; i++)
70                  if (columnsWithBlackPoints[i + 1] -
                        columnsWithBlackPoints[i] > 3)
71                  {
72                      StartX.Add(columnsWithBlackPoints[i + 1]);
73                      StopX.Add(columnsWithBlackPoints[i]);
74                  }
75              StopX.Add(columnsWithBlackPoints[columnsWithBlackPoints.
                    Count - 1]);
76
77
78              //Przedzialy miedzy wierszami
79              List<int> StartY = new List<int>();
80              List<int> StopY = new List<int>();
81              int digits = StartX.Count; //Tyle znaleziono znakow
82              for (int i = 0; i < digits; i++)
83              {
84                  List<int> Rows = RowSearch(StartX, StopX, btm, i); //Dla
                        kazdego przedzialu kolumn z osobna
85                  if (Rows.Count != 0)
```

```
86                  {
87                      StartY.Add(Rows[0]);
88                      StopY.Add(Rows[Rows.Count - 1]);
89                  }
90              }
91              return VerticalCropping(StartX, StopX, StartY, StopY, btm);
92          }
93
94          // Dla obliczonych przedzialow wycinamy obrazy i wywolujemy
                funkcje skalujaca wyciete obrazy:
95          private static List<double[][]> VerticalCropping(List<int>
                StartX, List<int> StopX, List<int> StartY, List<int> StopY,
                Bitmap btm)
96          {
97              int width, height;
98              Bitmap bmpImage = new Bitmap(btm);
99              List<double[][]> digits = new List<double[][]>();
100
101             for (int i = 0; i < StartX.Count; i++)
102             {
103                 width = StopX[i] - StartX[i];
104                 height = StopY[i] - StartY[i];
105                 if (width > 0 && height > 0)
106                 {
107                     Bitmap bmpCrop = bmpImage.Clone(new Rectangle(StartX
                            [i], StartY[i], width, height), bmpImage.
                            PixelFormat);
108                     digits.Add(ResizeImage(TransformToSquare(bmpCrop,
                            btm)));
109                 }
110             }
111             return digits;
112         }
113
114         private static Bitmap TransformToSquare(Bitmap bmpCrop, Bitmap
                btm)
115         {
116             int height, width;
117             if(bmpCrop.Height < btm.Height*0.15 && bmpCrop.Width < btm.
                    Height * 0.15) // "*"
118             {
119                 height = (int)(bmpCrop.Height * 8);
120                 width = (int)(bmpCrop.Height * 8);
121             }
122             else if (bmpCrop.Height > bmpCrop.Width) // "-"
123             {
124                 height = (int)(bmpCrop.Height * 1.5);
125                 width = (int)(bmpCrop.Height * 1.5);
126             }
127             else
128             {
129                 height = (int)(bmpCrop.Width * 1.5);
130                 width = (int)(bmpCrop.Width * 1.5);
131             }
132             Bitmap bitmap = new Bitmap(width, height);
```

```
133             using (var g = Graphics.FromImage(bitmap))
134             {
135                 g.FillRectangle(Brushes.White, 0, 0, width, height);
136                 int x = width / 2 - bmpCrop.Width / 2;
137                 int y = height / 2 - bmpCrop.Height / 2;
138                 g.DrawImage(bmpCrop, x, y);
139             }
140             return bitmap;
141         }
142 
143 
144         // Funkcja zmieniajaca rozdzielczosc na 28x28 i zwracajaca
                bitmape w postaci tablicy dwuwymiarowej:
145         private static double[][] ResizeImage(Image image)
146         {
147             int width = 28, height = 28;
148             Rectangle croppSize = new Rectangle(0, 0, width, height);
149             Bitmap resizedImage = new Bitmap(width, height);
150             resizedImage.SetResolution(image.HorizontalResolution, image
                    .VerticalResolution);
151 
152             using (var graphics = Graphics.FromImage(resizedImage))
153             {
154                 graphics.CompositingMode = CompositingMode.SourceCopy;
155                 graphics.CompositingQuality = CompositingQuality.
                        HighQuality;
156                 graphics.InterpolationMode = InterpolationMode.
                        HighQualityBicubic;
157                 graphics.SmoothingMode = SmoothingMode.HighQuality;
158                 graphics.PixelOffsetMode = PixelOffsetMode.HighQuality;
159 
160                 using (var wrapMode = new ImageAttributes())
161                 {
162                     wrapMode.SetWrapMode(WrapMode.TileFlipXY);
163                     graphics.DrawImage(image, croppSize, 0, 0, image.
                            Width, image.Height, GraphicsUnit.Pixel, wrapMode
                            );
164                 }
165             }
166             return Data.BitmapToArray(resizedImage);
167         }
168 
169         // Glowna funkcja wywolujaca sekwencje:
170         public static List<double[]> DetectDigits(MemoryStream picture
                )
171         {
172             Bitmap btm = new Bitmap(picture);
173             return IntervalsCounting(ColumnSearch(btm), btm);  //
                    Analiza dzialania, wyciecie i zapis
174         }
175 
176         public static string RecognizeDigits(List<double[]> digits,
                Network network)
177         {
178             string tmp = "";
```

```
179                 foreach (double[] digit in digits)
180                 {
181                     network.PushInputValues(digit);
182                     var output = network.GetOutput();
183 
184                     for (int i = 0; i < output.Count; i++)
185                         Debug.WriteLine(output[i] + " ");
186                     Debug.WriteLine("");
187 
188 
189                     double max = output.Max();
190                     //if (max < 0.5) return $"NIE POTRAFIE ROZPOZNAC {digits
                            .IndexOf(digit) + 1}. ZNAKU";
191 
192                     //int count = 0;
193                     //foreach (double i in output) if (i > 0.7) count++;
194                     //if (count > 1) return $"NIE POTRAFIE ROZPOZNAC {digits
                            .IndexOf(digit) + 1}. ZNAKU";
195 
196                     string[] signs = { "0", "1", "2", "3", "4", "5", "6", "7
                            ", "8", "9", "+", "-", "*", "/" };
197                     int index = output.IndexOf(max);
198                     if (index >= 10) tmp += " " + signs[index] + " ";
199                     else tmp += signs[index];
200                 }
201                 return tmp;
202             }
203         }
204 }
```

Calculation.cs - funkcje wyliczeniowe

```
1  using System;
2  using System.Collections.Generic;
3 
4  namespace DigitRecognizer
5  {
6      class Calculation
7      {
8          public static string Calculate(string equation)
9          {
10             if (equation.EndsWith(" - ") || equation.EndsWith(" + ") ||
                   equation.EndsWith(" * ") || equation.EndsWith(" / ") ||
                   equation.Contains("  "))
11                 return "BLEDNY ZAPIS!";
12             if (equation.StartsWith(" - ") || equation.StartsWith(" + ")
                   || equation.StartsWith(" * ") || equation.StartsWith(" /
                   "))
13                 return "BLEDNY ZAPIS!";
14             if (equation.Contains(" / 0")) return "NIE MOZNA DZIELIC
                   PRZEZ ZERO!";
15             Stack<string> temp = ConvertToStack(toRPN(equation));
16             return " = " + evalRPN(temp).ToString();
```

```
17          }
18
19          public static string toRPN(string token) //metoda zwracajaca
                wyrazenie w RPN w stringu
20          {
21              Dictionary<string, int> precedence = new Dictionary<string,
                    int>
22              {
23                  { "+", 1 }, { "-", 1 }, { "/", 2 }, { "*", 2 }
24              };
25
26              Stack<string> stack = new Stack<string>();
27
28              string result = "";
29              string[] equation = token.Split(' ');
30
31              foreach (string item in equation)
32              {
33                  try
34                  {
35                      double temp = Convert.ToDouble(item);
36                      result += " " + item;
37                  }
38                  catch
39                  {
40                      while (stack.Count != 0 && precedence[item] <=
                            precedence[stack.Peek()])
41                          result += " " + stack.Pop();
42
43                      stack.Push(item);
44                  }
45              }
46
47              while (stack.Count != 0)
48                  result += " " + stack.Pop();
49
50              result = result.Remove(0, 1);
51
52              return result;
53          }
54
55          public static Stack<string> ConvertToStack(string tokens) //
                metoda konwertujaca string na Stack
56          {
57              string[] result = tokens.Split();
58              Stack<string> stack = new Stack<string>();
59
60              foreach (string token in result)
61                  stack.Push(token);
62
63              return stack;
64          }
65
66          public static double evalRPN(Stack<string> tokens) //metoda
                zwracajaca wynik wyrazenia w RPN
```

```
67          {
68              string token = tokens.Pop();
69              double firstNumber, secondNumber;
70
71              if (!Double.TryParse(token, out firstNumber))
72              {
73                  secondNumber = evalRPN(tokens);
74                  firstNumber = evalRPN(tokens);
75
76                  if (token == "+")
77                      firstNumber += secondNumber;
78                  else if (token == "-")
79                      firstNumber -= secondNumber;
80                  else if (token == "*")
81                      firstNumber *= secondNumber;
82                  else if (token == "/")
83                      firstNumber /= secondNumber;
84                  else throw new Exception();
85              }
86
87              return firstNumber;
88          }
89      }
90  }
```

NeuralNetwork/Data.cs

```
1  using System.Collections.Generic;
2  using System.Drawing;
3
4  namespace NeuralNetwork
5  {
6      class Data
7      {
8          public static double[][] BitmapToArray(Bitmap bitmap)
9          {
10             double[][] values = new double[bitmap.Height][];
11             for (int i = 0; i < values.Length; i++)
12                 values[i] = new double[bitmap.Width];
13
14             for (int i = 0; i < bitmap.Height; i++)
15                 for (int j = 0; j < bitmap.Width; j++)
16                 {
17                     values[i][j] = 255 - (bitmap.GetPixel(j, i).R +
                           bitmap.GetPixel(j, i).G + bitmap.GetPixel(j, i).B
                           ) / 3;
18                     //if (values[i][j] >= 10) values[i][j] = 1;
19                     //else values[i][j] = 0;
20                 }
21
22             return values;
23         }
24
```

```
25          public static List<double[]> RemoveSecondDimensions(List<double
                [][]> digits)
26          {
27              List<double[]> tmp = new List<double[]>();
28              foreach (double[][] digit in digits)
29              {
30                  List<double> newlist = new List<double>();
31                  for (int i = 0; i < digit.Length; i++)
32                      for (int j = 0; j < digit[i].Length; j++)
33                          newlist.Add(digit[i][j]);
34
35                  tmp.Add(newlist.ToArray());
36              }
37              return tmp;
38          }
39      }
40  }
```

NeuralNetwork/Network.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.IO;
4
5   namespace NeuralNetwork
6   {
7       class Network
8       {
9           static double SynapsesCount;
10          internal List<Layer> Layers;
11
12          public Network(double alpha, int inputneuronscount, int[]
                hiddenlayerssizes, int outputneuronscount)
13          {
14              if (inputneuronscount < 1 || hiddenlayerssizes.Length < 1 ||
                    outputneuronscount < 1)
15                  throw new Exception("Incorrect Network Parameters");
16
17              Functions.Alpha = alpha;
18
19              Layers = new List<Layer>();
20              AddFirstLayer(inputneuronscount);
21              for (int i = 0; i < hiddenlayerssizes.Length; i++)
22                  AddNextLayer(new Layer(hiddenlayerssizes[i]));
23              AddNextLayer(new Layer(outputneuronscount));
24
25              SynapsesCount = CountSynapses();
26          }
27
28          private void AddFirstLayer(int inputneuronscount)
29          {
30              Layer inputlayer = new Layer(inputneuronscount);
31              foreach (Neuron neuron in inputlayer.Neurons)
```

```
32                  neuron.AddInputSynapse(0);
33              Layers.Add(inputlayer);
34          }
35
36          private void AddNextLayer(Layer newlayer)
37          {
38              Layer lastlayer = Layers[Layers.Count - 1];
39              lastlayer.ConnectLayers(newlayer);
40              Layers.Add(newlayer);
41          }
42
43          public void PushInputValues(double[] inputs)
44          {
45              if (inputs.Length != Layers[0].Neurons.Count)
46                  throw new Exception("Incorrect Input Size");
47
48              for (int i = 0; i < inputs.Length; i++)
49                  Layers[0].Neurons[i].PushValueOnInput(inputs[i]);
50          }
51
52          public List<double> GetOutput()
53          {
54              List<double> output = new List<double>();
55              for (int i = 0; i < Layers.Count; i++)
56                  Layers[i].CalculateOutputOnLayer();
57              foreach (Neuron neuron in Layers[Layers.Count - 1].Neurons)
58                  output.Add(neuron.OutputValue);
59              return output;
60          }
61
62          public static Network LoadNetworkFromFile(string path)
63          {
64              string[] lines = File.ReadAllLines(path);
65              string[] firstLine = lines[0].Split();
66              List<int> hiddenLayerSizes = new List<int>();
67              for (int i = 2; i < firstLine.Length - 1; i++)
68                  hiddenLayerSizes.Add(Convert.ToInt32(firstLine[i]));
69
70              Network net = new Network(double.Parse(firstLine[0]),
                    Convert.ToInt32(firstLine[1]),
71                  hiddenLayerSizes.ToArray(), Convert.ToInt32(firstLine[
                        firstLine.Length - 1]));
72
73              if (lines.Length - 1 != SynapsesCount)
74                  throw new Exception("Incorrect Input File");
75              else
76              {
77                  try
78                  {
79                      int i = 1;
80                      for (int j = 1; j < net.Layers.Count; j++)
81                          foreach (Neuron neuron in net.Layers[j].Neurons)
82                              foreach (Synapse synapse in neuron.Inputs)
83                                  synapse.Weight = double.Parse(lines[i
                                        ++]);
```

```
84                      }
85                      catch (Exception) { throw new Exception("Incorrect Input
                            File"); }
86                  }
87                  return net;
88              }
89
90              private double CountSynapses()
91              {
92                  double count = 0;
93                  for (int i = 1; i < Layers.Count; i++)
94                      foreach (Neuron neuron in Layers[i].Neurons)
95                          foreach (Synapse synapse in neuron.Inputs)
96                              count++;
97                  return count;
98              }
99      }
100 }
```

NeuralNetwork/Layer.cs

```
1  using System.Collections.Generic;
2
3  namespace NeuralNetwork
4  {
5      class Layer
6      {
7          public List<Neuron> Neurons;
8
9          public Layer(int numberofneurons)
10         {
11             Neurons = new List<Neuron>();
12             for (int i = 0; i < numberofneurons; i++)
13                 Neurons.Add(new Neuron());
14         }
15
16         public void ConnectLayers(Layer outputlayer)
17         {
18             foreach (Neuron thisneuron in Neurons)
19                 foreach (Neuron thatneuron in outputlayer.Neurons)
20                     thisneuron.AddOutputNeuron(thatneuron);
21         }
22
23         public void CalculateOutputOnLayer()
24         {
25             foreach (Neuron neuron in Neurons)
26                 neuron.CalculateOutput();
27         }
28     }
29 }
```

NeuralNetwork/Neuron.cs

```
using System.Collections.Generic;

namespace NeuralNetwork
{
    class Neuron
    {
        public List<Synapse> Inputs { get; set; }
        public List<Synapse> Outputs { get; set; }
        public double InputValue { get; set; }
        public double OutputValue { get; set; }

        public Neuron()
        {
            Inputs = new List<Synapse>();
            Outputs = new List<Synapse>();
        }

        public void AddOutputNeuron(Neuron outputneuron)
        {
            Synapse synapse = new Synapse(this, outputneuron);
            Outputs.Add(synapse); outputneuron.Inputs.Add(synapse);
        }

        public void AddInputSynapse(double input)
        {
            Synapse syn = new Synapse(this, input);
            Inputs.Add(syn);
        }

        public void CalculateOutput()
        {
            InputValue = Functions.InputSumFunction(Inputs);
            OutputValue = Functions.BipolarLinearFunction(InputValue);
        }

        public void PushValueOnInput(double input)
        {
            Inputs[0].PushedData = input;
        }
    }
}
```

NeuralNetwork/Synapse.cs

```
using System;

namespace NeuralNetwork
{
    class Synapse
    {
        static readonly Random tmp = new Random();
```

```
8           internal Neuron FromNeuron, ToNeuron;
9           public double Weight { get; set; }
10          public double PushedData { get; set; }
11
12          public Synapse(Neuron fromneuron, Neuron toneuron) // zwykla
                synapsa
13          {
14              FromNeuron = fromneuron; ToNeuron = toneuron;
15              Weight = tmp.NextDouble() - 0.5;
16          }
17
18          public Synapse(Neuron toneuron, double output)     // synapsa
                wejsciowa pierwszej warstwy
19          {
20              ToNeuron = toneuron; PushedData = output;
21              Weight = 1;
22          }
23
24          public double GetOutput()
25          {
26              if (FromNeuron == null) return PushedData;
27              return FromNeuron.OutputValue * Weight;
28          }
29      }
30  }
```

NeuralNetwork/Functions.cs

```
1   using System;
2   using System.Collections.Generic;
3
4   namespace NeuralNetwork
5   {
6       class Functions
7       {
8           public static double Alpha { get; set; } = 0.8;
9
10          public static double InputSumFunction(List<Synapse> Inputs)
11              // funkcja wejscia: suma iloczynow wag synaps wchodzacych i
                    wartosci wyjsciowych neuronow warstwy poprzedniej
12          {
13              double input = 0;
14              foreach (Synapse syn in Inputs)
15                  input += syn.GetOutput();
16              return input;
17          }
18
19          public static double BipolarLinearFunction(double input) //
                funkcja aktywacji: bipolarna liniowa
20              => (1 - Math.Pow(Math.E, -Alpha * input)) / (1 + Math.Pow(
                    Math.E, -Alpha * input));
21      }
22  }
```

Neural Network - Learning Place

Data.cs

```
1  using System;
2  using System.IO;
3  using System.Collections.Generic;
4  using System.Drawing;
5
6  namespace NeuralNetwork
7  {
8      class Data
9      {
10         public static double[][][] PrepareDatasets(int
               MNISTDatasetSizeDivider)
11         {
12             Console.WriteLine(" Loading datasets...");
13             string[] arithmeticFilePaths = Directory.GetFiles(@"Datasets
                   \", "signs*.png");
14             string[] digitFilePaths      = Directory.GetFiles(@"Datasets
                   \", "digits*.png");
15             double[][] trainImages = new double[60000 /
                   MNISTDatasetSizeDivider + arithmeticFilePaths.Length *
                   180 + digitFilePaths.Length * 120][];
16             double[][] trainLabels = new double[60000 /
                   MNISTDatasetSizeDivider + arithmeticFilePaths.Length *
                   180 + digitFilePaths.Length * 120][];
17             for (int i = 0; i < trainImages.Length; i++)
18                 trainImages[i] = new double[28 * 28];
19             for (int i = 0; i < trainLabels.Length; i++)
20                 trainLabels[i] = new double[14];
21
22             double[][] testImages = new double[10000 /
                   MNISTDatasetSizeDivider + arithmeticFilePaths.Length * 20
                    + digitFilePaths.Length * 30][];
23             double[][] testLabels = new double[10000 /
                   MNISTDatasetSizeDivider + arithmeticFilePaths.Length * 20
                    + digitFilePaths.Length * 30][];
24             for (int i = 0; i < testImages.Length; i++)
25                 testImages[i] = new double[28 * 28];
26             for (int i = 0; i < testLabels.Length; i++)
27                 testLabels[i] = new double[14];
28
29             LoadMNISTDataset(@"Datasets\train-images.idx3-ubyte", @"
                   Datasets\train-labels.idx1-ubyte", trainImages,
                   trainLabels, MNISTDatasetSizeDivider);
30             LoadMNISTDataset(@"Datasets\t10k-images.idx3-ubyte", @"
                   Datasets\t10k-labels.idx1-ubyte", testImages, testLabels,
                    MNISTDatasetSizeDivider);
31             LoadOwnDatasets(trainImages, trainLabels, testImages,
                   testLabels, arithmeticFilePaths, digitFilePaths,
```

```
            MNISTDatasetSizeDivider);
32          Shuffle(trainImages, trainLabels);
33
34          return new double[][][] { trainImages, trainLabels,
                testImages, testLabels };
35      }
36
37      private static void LoadOwnDatasets(double[][] trainImages,
            double[][] trainLabels,
38          double[][] testImages, double[][] testLabels, string[]
                arithmeticFilePaths, string[] digitFilePaths, int
                MNISTDatasetSizeDivider)
39      {
40          int trainIndex = 60000 / MNISTDatasetSizeDivider, testIndex
                = 10000 / MNISTDatasetSizeDivider;
41
42          // Znaki arytmetyczne:
43          List<double[]> arithmeticSigns; int tempIndex = 0;
44          for (int i = 0; i < arithmeticFilePaths.Length; i++)
45          {
46              arithmeticSigns = RemoveSecondDimensions(DigitDetection.
                    DetectDigits(new Bitmap(arithmeticFilePaths[i])));
47              for (int j = 0; j < arithmeticSigns.Count - 20; j++)
48              {
49                  trainImages[trainIndex] = arithmeticSigns[j];
50                  for (int k = 0; k < trainImages[trainIndex].Length;
                        k++)
51                  {
52                      if (trainImages[trainIndex][k] < 10) trainImages
                            [trainIndex][k] = 0;
53                      else trainImages[trainIndex][k] = 1;
54                  }
55                  trainLabels[trainIndex][(tempIndex++ % 4) + 10] = 1;
56                  trainIndex++;
57              }
58              for (int j = arithmeticSigns.Count - 20; j <
                    arithmeticSigns.Count; j++) // ostatnie 20 znakow (
                    czyli 10%, bo mamy pliki po 200 znakow) idzie do
                    testowego
59              {
60                  testImages[testIndex] = arithmeticSigns[j];
61                  for (int k = 0; k < testImages[testIndex].Length; k
                        ++)
62                  {
63                      if (testImages[testIndex][k] < 10) testImages[
                            testIndex][k] = 0;
64                      else testImages[testIndex][k] = 1;
65                  }
66                  testLabels[testIndex][(tempIndex++ % 4) + 10] = 1;
67                  testIndex++;
68              }
69          }
70
71          // Cyfry:
72          List<double[]> digits; tempIndex = 0;
```

```
73              for (int i = 0; i < digitFilePaths.Length; i++)
74              {
75                  digits = RemoveSecondDimensions(DigitDetection.
                        DetectDigits(new Bitmap(digitFilePaths[i])));
76                  for (int j = 0; j < digits.Count - 30; j++)
77                  {
78                      trainImages[trainIndex] = digits[j];
79                      for(int k = 0; k < trainImages[trainIndex].Length; k
                            ++)
80                      {
81                          if (trainImages[trainIndex][k] < 10) trainImages
                                [trainIndex][k] = 0;
82                          else trainImages[trainIndex][k] = 1;
83                      }
84                      trainLabels[trainIndex][(tempIndex++ + 1) % 10] = 1;
85                      trainIndex++;
86                  }
87                  for (int j = digits.Count - 30; j < digits.Count; j++)
                        // ostatnie 30 znakow (czyli 20%, bo mamy pliki po
                        150 znakow) idzie do testowego
88                  {
89                      testImages[testIndex] = digits[j];
90                      for (int k = 0; k < testImages[testIndex].Length; k
                            ++)
91                      {
92                          if (testImages[testIndex][k] < 10) testImages[
                                testIndex][k] = 0;
93                          else testImages[testIndex][k] = 1;
94                      }
95                      testLabels[testIndex][(tempIndex++ + 1) % 10] = 1;
96                      testIndex++;
97                  }
98              }
99          }
100
101         private static void Shuffle(double[][] arr1, double[][] arr2)
102         {
103             Random rand = new Random();
104             int j = arr1.Length;
105
106             while(j > 1)
107             {
108                 int k = rand.Next(j--);
109                 var temp1 = arr1[j];
110                 var temp2 = arr2[j];
111
112                 arr1[j] = arr1[k];
113                 arr1[k] = temp1;
114
115                 arr2[j] = arr2[k];
116                 arr2[k] = temp2;
117             }
118         }
119
120         public static double[][] BitmapToArray(Bitmap bitmap)
```

```
121         {
122             double[][] values = new double[bitmap.Height][];
123             for (int i = 0; i < values.Length; i++)
124                 values[i] = new double[bitmap.Width];
125 
126             for (int i = 0; i < bitmap.Height; i++)
127                 for (int j = 0; j < bitmap.Width; j++)
128                     values[i][j] = 255 - (bitmap.GetPixel(j, i).R +
                            bitmap.GetPixel(j, i).G + bitmap.GetPixel(j, i).B
                            ) / 3;
129 
130             return values;
131         }
132 
133         private static void LoadMNISTDataset(string imagesName, string
                labelsName, double[][] Images, double[][] Labels, int
                MNISTDatasetSizeDivider)
134         {
135             BinaryReader brImages = new BinaryReader(new FileStream(
                    imagesName, FileMode.Open));
136             BinaryReader brLabels = new BinaryReader(new FileStream(
                    labelsName, FileMode.Open));
137 
138             Extensions.ReadBigInt32(brImages);  // magic1
139             int numImages = Extensions.ReadBigInt32(brImages);
140             int numRows = Extensions.ReadBigInt32(brImages);
141             int numCols = Extensions.ReadBigInt32(brImages);
142 
143             Extensions.ReadBigInt32(brLabels);  // magic2
144             Extensions.ReadBigInt32(brLabels);  // numLabels
145 
146             for (int i = 0; i < numImages / MNISTDatasetSizeDivider; i
                    ++)
147             {
148                 for (int j = 0; j < numRows * numCols; j++)
149                 {
150                     Images[i][j] = Convert.ToDouble(brImages.ReadByte())
                            ;
151                     if (Images[i][j] < 10) Images[i][j] = 0;
152                     else Images[i][j] = 1;
153                 }
154 
155                 Labels[i][Convert.ToInt32(brLabels.ReadByte())] = 1;
156             }
157         }
158 
159         private static List<double[]> RemoveSecondDimensions(List<double
                []>> digits)
160         {
161             List<double[]> tmp = new List<double[]>();
162             foreach (double[][] digit in digits)
163             {
164                 List<double> newlist = new List<double>();
165                 for (int i = 0; i < digit.Length; i++)
166                     for (int j = 0; j < digit[i].Length; j++)
```

```
                        newlist.Add(digit[i][j]);

                tmp.Add(newlist.ToArray());
            }
            return tmp;
        }
    }

    public static class Extensions
    {
        public static int ReadBigInt32(this BinaryReader br)
        {
            var bytes = br.ReadBytes(sizeof(Int32));
            if (BitConverter.IsLittleEndian)
                Array.Reverse(bytes);
            return BitConverter.ToInt32(bytes, 0);
        }
    }
}
```

Network.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;

namespace NeuralNetwork
{
    class Network
    {
        static readonly double LearningRate = 0.05;
        static double SynapsesCount;
        internal List<Layer> Layers;
        internal double[][] ExpectedResults;
        double[][] ErrorFunctionChanges;

        public Network(double alpha, int inputneuronscount, int[]
            hiddenlayerssizes, int outputneuronscount)
        {
            Console.WriteLine(" Building neural network...");
            if (inputneuronscount < 1 || hiddenlayerssizes.Length < 1 ||
                outputneuronscount < 1)
                throw new Exception("Incorrect Network Parameters");

            Functions.Alpha = alpha;

            Layers = new List<Layer>();
            AddFirstLayer(inputneuronscount);
            for (int i = 0; i < hiddenlayerssizes.Length; i++)
                AddNextLayer(new Layer(hiddenlayerssizes[i]));
            AddNextLayer(new Layer(outputneuronscount));

```

```
30              SynapsesCount = CountSynapses();
31
32              ErrorFunctionChanges = new double[Layers.Count][];
33              for (int i = 1; i < Layers.Count; i++)
34                  ErrorFunctionChanges[i] = new double[Layers[i].Neurons.
                       Count];
35          }
36
37          private void AddFirstLayer(int inputneuronscount)
38          {
39              Layer inputlayer = new Layer(inputneuronscount);
40              foreach (Neuron neuron in inputlayer.Neurons)
41                  neuron.AddInputSynapse(0);
42              Layers.Add(inputlayer);
43          }
44
45          private void AddNextLayer(Layer newlayer)
46          {
47              Layer lastlayer = Layers[Layers.Count - 1];
48              lastlayer.ConnectLayers(newlayer);
49              Layers.Add(newlayer);
50          }
51
52          public void PushInputValues(double[] inputs)
53          {
54              if (inputs.Length != Layers[0].Neurons.Count)
55                  throw new Exception("Incorrect Input Size");
56
57              for (int i = 0; i < inputs.Length; i++)
58                  Layers[0].Neurons[i].PushValueOnInput(inputs[i]);
59          }
60
61          public void PushExpectedValues(double[][] expectedvalues)
62          {
63              if (expectedvalues[0].Length != Layers[Layers.Count - 1].
                   Neurons.Count)
64                  throw new Exception("Incorrect Expected Output Size");
65
66              ExpectedResults = expectedvalues;
67          }
68
69          public List<double> GetOutput()
70          {
71              List<double> output = new List<double>();
72              for (int i = 0; i < Layers.Count; i++)
73                  Layers[i].CalculateOutputOnLayer();
74              foreach (Neuron neuron in Layers[Layers.Count - 1].Neurons)
75                  output.Add(neuron.OutputValue);
76              return output;
77          }
78
79          public void Train(double[][][] datasets, double epochscount,
               bool showinfo = false, bool breaking = false)
80          {
81              double[][] trainingInputs = datasets[0], trainingOutputs =
```

```
            datasets[1];
82          double recenterror = double.MaxValue, minerror = double.
                MaxValue;
83
84          PushExpectedValues(trainingOutputs);
85          Console.WriteLine(" Training neural network...");
86          for (int i = 0; i < epochscount; i++)
87          {
88              List<double> outputs = new List<double>();
89              for (int j = 0; j < trainingInputs.Length; j++)
90              {
91                  PushInputValues(trainingInputs[j]);
92                  outputs = GetOutput();
93                  ChangeWeights(outputs, j);
94              }
95
96              recenterror = CalculateMeanSquareError(datasets[2],
                    datasets[3], showinfo);
97              if (breaking == true && minerror < recenterror) break;
98              minerror = recenterror;
99          }
100
101         SaveNetworkToFile("weights.txt");
102         Console.WriteLine(" Done!");
103     }
104
105     private double CalculateMeanSquareError(double[][] inputs,
            double[][] expectedoutputs, bool showerror = false)
106     {
107         double error = 0;
108         List<double> outputs = new List<double>();
109         for (int i = 0; i < inputs.Length; i++)
110         {
111             PushInputValues(inputs[i]);
112             outputs = GetOutput();
113             error += Functions.CalculateError(outputs, i,
                        expectedoutputs);
114         }
115         error /= inputs.Length;
116         if (showerror == true) Console.WriteLine($" Average mean
                square error: {Math.Round(error, 5)}");
117         return error;
118     }
119
120     private void ChangeWeights(List<double> outputs, int row) //
            przy uzyciu algorytmu wstecznej propagacji
121     {
122         CalculateErrorFunctionChanges(outputs, row);
123         for (int k = Layers.Count - 1; k > 0; k--)
124             for (int i = 0; i < Layers[k].Neurons.Count; i++)
125                 for (int j = 0; j < Layers[k - 1].Neurons.Count; j
                        ++)
126                     Layers[k].Neurons[i].Inputs[j].Weight +=
127                         LearningRate * 2 * ErrorFunctionChanges[k][i
                            ] * Layers[k - 1].Neurons[j].OutputValue;
```

```
128         }
129
130         private void CalculateErrorFunctionChanges(List<double> outputs,
                int row)
131         {
132             for (int i = 0; i < Layers[Layers.Count - 1].Neurons.Count;
                    i++)
133                 ErrorFunctionChanges[Layers.Count - 1][i] = (
                        ExpectedResults[row][i] - outputs[i])
134                     * Functions.BipolarDifferential(Layers[Layers.Count
                        - 1].Neurons[i].InputValue);
135             for (int k = Layers.Count - 2; k > 0; k--)
136                 for (int i = 0; i < Layers[k].Neurons.Count; i++)
137                 {
138                     ErrorFunctionChanges[k][i] = 0;
139                     for (int j = 0; j < Layers[k + 1].Neurons.Count; j
                            ++)
140                         ErrorFunctionChanges[k][i] +=
                            ErrorFunctionChanges[k + 1][j] * Layers[k +
                            1].Neurons[j].Inputs[i].Weight;
141                     ErrorFunctionChanges[k][i] *= Functions.
                        BipolarDifferential(Layers[k].Neurons[i].
                        InputValue);
142                 }
143         }
144
145         private void SaveNetworkToFile(string path)
146         {
147             List<string> tmp = new List<string>();
148             for (int i = 1; i < Layers.Count; i++)
149                 foreach (Neuron neuron in Layers[i].Neurons)
150                     foreach (Synapse synapse in neuron.Inputs)
151                         tmp.Add(synapse.Weight.ToString());
152
153             string build = Functions.Alpha.ToString();
154             foreach (Layer layer in Layers) build += " " + layer.Neurons
                    .Count.ToString();
155             tmp.Insert(0, build);
156             File.WriteAllLines(path, tmp);
157         }
158
159         public static Network LoadNetworkFromFile(string path)
160         {
161             string[] lines = File.ReadAllLines(path);
162             string[] firstLine = lines[0].Split();
163             List<int> hiddenLayerSizes = new List<int>();
164             for (int i = 2; i < firstLine.Length - 1; i++)
165                 hiddenLayerSizes.Add(Convert.ToInt32(firstLine[i]));
166
167             Network net = new Network(double.Parse(firstLine[0]),
                    Convert.ToInt32(firstLine[1]),
168                 hiddenLayerSizes.ToArray(), Convert.ToInt32(firstLine[
                        firstLine.Length - 1]));
169
170             Console.WriteLine(" Loading weights...");
```

```
171             if (lines.Length - 1 != SynapsesCount)
172                 Console.WriteLine(" Incorrect input file.");
173             else
174             {
175                 try
176                 {
177                     int i = 1;
178                     for (int j = 1; j < net.Layers.Count; j++)
179                         foreach (Neuron neuron in net.Layers[j].Neurons)
180                             foreach (Synapse synapse in neuron.Inputs)
181                                 synapse.Weight = double.Parse(lines[i
                                    ++]);
182                 }
183                 catch (Exception) { Console.WriteLine(" Incorrect input
                    file."); }
184             }
185             return net;
186         }
187 
188         public void CalculatePrecision(double[][][] datasets, bool
            shownumbers = false) // z uzyciem zbioru testowego
189         {
190             double[][] testingInputs = datasets[2], testingOutputs =
                datasets[3];
191             List<double> outputs; int correct = 0;
192             for (int i = 0; i < testingInputs.Length; i++)
193             {
194                 PushInputValues(testingInputs[i]);
195                 outputs = GetOutput();
196                 if (shownumbers == true) Classify(testingOutputs[i],
                    outputs);
197                 if (outputs.IndexOf(outputs.Max()) == testingOutputs[i].
                    ToList().IndexOf(1)) correct += 1;
198             }
199             double precision = Math.Round((double)correct /
                testingInputs.Length, 4) * 100;
200             Console.WriteLine($" Precision: {precision.ToString()}%");
201         }
202 
203         public void Classify(double[] testingOutputs, List<double>
            trueOutputs)
204         {
205             string[] signs = { "0", "1", "2", "3", "4", "5", "6", "7", "
                8", "9", "+", "-", "*", ":" };
206             Console.Write("\n Should be: ");
207             for (int i = 0; i < testingOutputs.Length; i++) Console.
                Write(string.Format("{0, 4}", testingOutputs[i].ToString(
                "0.0")) + " ");
208             Console.Write($"-> {signs[testingOutputs.ToList().IndexOf(
                testingOutputs.Max())]}\n Got:       ");
209             for (int i = 0; i < trueOutputs.Count; i++) Console.Write(
                string.Format("{0, 4}", trueOutputs[i].ToString("0.0")) +
                 " ");
210             Console.WriteLine($"-> {signs[trueOutputs.ToList().IndexOf(
                trueOutputs.Max())]}\n");
```

```
211         }
212
213         private double CountSynapses()
214         {
215             double count = 0;
216             for (int i = 1; i < Layers.Count; i++)
217                 foreach (Neuron neuron in Layers[i].Neurons)
218                     foreach (Synapse synapse in neuron.Inputs)
219                         count++;
220             return count;
221         }
222     }
223 }
```

Layer.cs

```
1  using System.Collections.Generic;
2
3  namespace NeuralNetwork
4  {
5      class Layer
6      {
7          public List<Neuron> Neurons;
8
9          public Layer(int numberofneurons)
10         {
11             Neurons = new List<Neuron>();
12             for (int i = 0; i < numberofneurons; i++)
13                 Neurons.Add(new Neuron());
14         }
15
16         public void ConnectLayers(Layer outputlayer)
17         {
18             foreach (Neuron thisneuron in Neurons)
19                 foreach (Neuron thatneuron in outputlayer.Neurons)
20                     thisneuron.AddOutputNeuron(thatneuron);
21         }
22
23         public void CalculateOutputOnLayer()
24         {
25             foreach (Neuron neuron in Neurons)
26                 neuron.CalculateOutput();
27         }
28     }
29 }
```

Neuron.cs

```
1  using System.Collections.Generic;
2
3  namespace NeuralNetwork
```

```
{
    class Neuron
    {
        public List<Synapse> Inputs { get; set; }
        public List<Synapse> Outputs { get; set; }
        public double InputValue { get; set; }
        public double OutputValue { get; set; }

        public Neuron()
        {
            Inputs = new List<Synapse>();
            Outputs = new List<Synapse>();
        }

        public void AddOutputNeuron(Neuron outputneuron)
        {
            Synapse synapse = new Synapse(this, outputneuron);
            Outputs.Add(synapse); outputneuron.Inputs.Add(synapse);
        }

        public void AddInputSynapse(double input)
        {
            Synapse syn = new Synapse(this, input);
            Inputs.Add(syn);
        }

        public void CalculateOutput()
        {
            InputValue = Functions.InputSumFunction(Inputs);
            OutputValue = Functions.BipolarLinearFunction(InputValue);
        }

        public void PushValueOnInput(double input)
        {
            Inputs[0].PushedData = input;
        }
    }
}
```

Synapse.cs

```
using System;

namespace NeuralNetwork
{
    class Synapse
    {
        static readonly Random tmp = new Random();
        internal Neuron FromNeuron, ToNeuron;
        public double Weight { get; set; }
        public double PushedData { get; set; }

        public Synapse(Neuron fromneuron, Neuron toneuron) // zwykla
```

```
            synapsa
13          {
14              FromNeuron = fromneuron; ToNeuron = toneuron;
15              Weight = tmp.NextDouble() - 0.5;
16          }
17
18          public Synapse(Neuron toneuron, double output)      // synapsa
                wejsciowa pierwszej warstwy
19          {
20              ToNeuron = toneuron; PushedData = output;
21              Weight = 1;
22          }
23
24          public double GetOutput()
25          {
26              if (FromNeuron == null) return PushedData;      // if it is
                    first layer
27              return FromNeuron.OutputValue * Weight;
28          }
29      }
30  }
```

Functions.cs

```
1   using System;
2   using System.Collections.Generic;
3
4   namespace NeuralNetwork
5   {
6       class Functions
7       {
8           public static double Alpha { get; set; } = 0.8;
9
10          public static double CalculateError(List<double> outputs, int
                row, double[][] expectedresults) // funkcja celu
11          {
12              double error = 0;
13              for (int i = 0; i < outputs.Count; i++)
14                  error += Math.Pow(outputs[i] - expectedresults[row][i],
                        2);
15              return error;
16          }
17
18          public static double InputSumFunction(List<Synapse> Inputs,
                double bias = 0)
19              // funkcja wejscia: suma iloczynow wag synaps i wyjsc
                    neuronow + bias (zaklocenia)
20          {
21              double input = 0;
22              foreach (Synapse syn in Inputs)
23                  input += syn.GetOutput();
24              input += bias;
25              return input;
```

```
26          }
27
28          public static double BipolarLinearFunction(double input) //
                funkcja aktywacji: bipolarna liniowa...
29              => (1 - Math.Pow(Math.E, -Alpha * input)) / (1 + Math.Pow(
                    Math.E, -Alpha * input));
30
31          public static double BipolarDifferential(double input) // ... i
                jej pochodna
32              => (2 * Alpha * Math.Pow(Math.E, -Alpha * input)) / (Math.
                    Pow(1 + Math.Pow(Math.E, -Alpha * input), 2));
33      }
34  }
```

DigitDetection.cs

```
1  using System.Collections.Generic;
2  using System.Drawing;
3  using System.Drawing.Drawing2D;
4  using System.Drawing.Imaging;
5  using System.IO;
6
7  namespace NeuralNetwork
8  {
9      class DigitDetection
10     {
11         // Przeszukuje kolumny w celu znalezienia punktow innych niz
               biale:
12         private static List<int> ColumnSearch(Bitmap btm)
13         {
14             List<int> Cols = new List<int>();
15             Color color;
16             for (int j = 0; j < btm.Width; j++)
17                 for (int i = 0; i < btm.Height; i++)
18                 {
19                     color = btm.GetPixel(j, i);
20                     if (color != Color.FromArgb(255, 255, 255))
21                     {
22                         Cols.Add(j);
23                         break;
24                     }
25                 }
26             return Cols;
27         }
28
29         //Przeszukuje przedzialy znalezione przez ColumnSearch w
               poszukiwaniu punktow innych niz biale:
30         private static List<int> RowSearch(List<int> StartX, List<int>
               StopX, Bitmap btm, int digit)
31         {
32             List<int> Rows = new List<int>();
33             Color color;
34             for (int k = 0; k < btm.Height; k++)
```

```
35                  for (int j = StartX[digit]; j < StopX[digit]; j++)
36                  {
37                      color = btm.GetPixel(j, k);
38                      if (color != Color.FromArgb(255, 255, 255))
39                      {
40                          Rows.Add(k);
41                          break;
42                      }
43                  }
44
45              if (Rows.Count == 0) //Zabezpieczenie
46              {
47                  Rows.Add(0);
48                  Rows.Add(btm.Height);
49              }
50
51              return Rows;
52          }
53
54
55          private static List<double[][]> IntervalsCounting(List<int>
                columnsWithBlackPoints, Bitmap btm)
56          {
57              if (columnsWithBlackPoints.Count == 0)
58                  return new List<double[][]>();
59
60              //Przedzialy miedzy kolumnami
61              List<int> StartX = new List<int>();
62              List<int> StopX = new List<int>();
63
64              StartX.Add(columnsWithBlackPoints[0]);
65              for (int i = 1; i < columnsWithBlackPoints.Count - 1; i++)
66                  if (columnsWithBlackPoints[i + 1] -
                        columnsWithBlackPoints[i] > 3)
67                  {
68                      StartX.Add(columnsWithBlackPoints[i + 1]);
69                      StopX.Add(columnsWithBlackPoints[i]);
70                  }
71              StopX.Add(columnsWithBlackPoints[columnsWithBlackPoints.
                    Count - 1]);
72
73
74              //Przedzialy miedzy wierszami
75              List<int> StartY = new List<int>();
76              List<int> StopY = new List<int>();
77              int digits = StartX.Count; //Tyle znaleziono znakow
78              for (int i = 0; i < digits; i++)
79              {
80                  List<int> Rows = RowSearch(StartX, StopX, btm, i); //Dla
                        kazdego przedzialu kolumn z osobna
81                  if (Rows.Count != 0)
82                  {
83                      StartY.Add(Rows[0]);
84                      StopY.Add(Rows[Rows.Count - 1]);
85                  }
```

```
86              }
87              return VerticalCropping(StartX, StopX, StartY, StopY, btm);
88          }
89
90
91
92          // Dla obliczonych przedzialow wycinamy obrazy i wywolujemy
                funkcje skalujaca wyciete obrazy:
93          private static List<double[][]> VerticalCropping(List<int>
                StartX, List<int> StopX, List<int> StartY, List<int> StopY,
                Bitmap btm)
94          {
95              int width, height;
96              Bitmap bmpImage = new Bitmap(btm);
97              List<double[][]> digits = new List<double[][]>();
98
99              for (int i = 0; i < StartX.Count; i++)
100             {
101                 width = StopX[i] - StartX[i];
102                 height = StopY[i] - StartY[i];
103                 if (width > 0 && height > 0)
104                 {
105                     Bitmap bmpCrop = bmpImage.Clone(new Rectangle(StartX
                            [i], StartY[i], width, height), bmpImage.
                            PixelFormat);
106                     digits.Add(ResizeImage(TransformToSquare(bmpCrop,
                            btm)));
107                 }
108             }
109             return digits;
110         }
111
112         private static Bitmap TransformToSquare(Bitmap bmpCrop, Bitmap
                btm)
113         {
114             int height, width;
115             if (bmpCrop.Height < btm.Height * 0.15 && bmpCrop.Width <
                    btm.Height * 0.15)
116             {
117                 height = (int)(bmpCrop.Height * 8);
118                 width = (int)(bmpCrop.Height * 8);
119             }
120             else if (bmpCrop.Height > bmpCrop.Width)
121             {
122                 height = (int)(bmpCrop.Height * 1.5);
123                 width = (int)(bmpCrop.Height * 1.5);
124             }
125             else
126             {
127                 height = (int)(bmpCrop.Width * 1.5);
128                 width = (int)(bmpCrop.Width * 1.5);
129             }
130             Bitmap bitmap = new Bitmap(width, height);
131             using (var g = Graphics.FromImage(bitmap))
132             {
```

```
133                 g.FillRectangle(Brushes.White, 0, 0, width, height);
134                 int x = width / 2 - bmpCrop.Width / 2;
135                 int y = height / 2 - bmpCrop.Height / 2;
136                 g.DrawImage(bmpCrop, x, y);
137             }
138             return bitmap;
139         }
140
141
142
143         // Funkcja zmieniajaca rozdzielczosc na 28x28 i zwracajaca
                bitmape w postaci tablicy dwuwymiarowej:
144         private static double[][] ResizeImage(Image image)
145         {
146             int width = 28, height = 28;
147             Rectangle croppSize = new Rectangle(0, 0, width, height);
148             Bitmap resizedImage = new Bitmap(width, height);
149             resizedImage.SetResolution(image.HorizontalResolution, image
                    .VerticalResolution);
150
151             using (var graphics = Graphics.FromImage(resizedImage))
152             {
153                 graphics.CompositingMode = CompositingMode.SourceCopy;
154                 graphics.CompositingQuality = CompositingQuality.
                        HighQuality;
155                 graphics.InterpolationMode = InterpolationMode.
                        HighQualityBicubic;
156                 graphics.SmoothingMode = SmoothingMode.HighQuality;
157                 graphics.PixelOffsetMode = PixelOffsetMode.HighQuality;
158
159                 using (var wrapMode = new ImageAttributes())
160                 {
161                     wrapMode.SetWrapMode(WrapMode.TileFlipXY);
162                     graphics.DrawImage(image, croppSize, 0, 0, image.
                            Width, image.Height, GraphicsUnit.Pixel, wrapMode
                            );
163                 }
164             }
165             return Data.BitmapToArray(resizedImage);
166         }
167
168         public static List<double[]> DetectDigits(Bitmap picture) =>
                IntervalsCounting(ColumnSearch(picture), picture);
169     }
170 }
```

Program.cs

```
using System;

namespace NeuralNetwork
{
    class Program
    {
        static void Main()
        {
            int MNISTDatasetSizeDivider = 50; // 1 -> 60000+10000; 5 -> 12000+2000; 10 -> 6000+1000; itd.
            double[][][] datasets = Data.PrepareDatasets(MNISTDatasetSizeDivider);
            // datasets[0] - Training Set's Input
            // datasets[1] - Training Set's Expected Output
            // datasets[2] - Testing  Set's Input
            // datasets[3] - Testing  Set's Expected Output

            //Network network = new Network(0.5, datasets[0][0].Length, new int[] { 100, 100, 100, 100}, datasets[1][0].Length);
            Network network = Network.LoadNetworkFromFile("weights.txt");
            network.CalculatePrecision(datasets);
            network.Train(datasets, 5, true);
            network.CalculatePrecision(datasets);

            Console.ReadKey();
        }
    }
}
```