|  |  |  |
| --- | --- | --- |
|  | **POLITECHNIKA KRAKOWSKA im. T. Kościuszki**  Wydział Mechaniczny  **Instytut Informatyki Stosowanej (M7)** | **logo WM** |

Kierunek studiów: **Informatyka Stosowana**

STUDIA STACJONARNE

**PRACA DYPLOMOWA**

INŻYNIERSKA

**Tadeusz Baum-Azbum**

Wykorzystanie sieci Hopfielda do rozpoznawania znaków

Hopfield network in optical character recognition

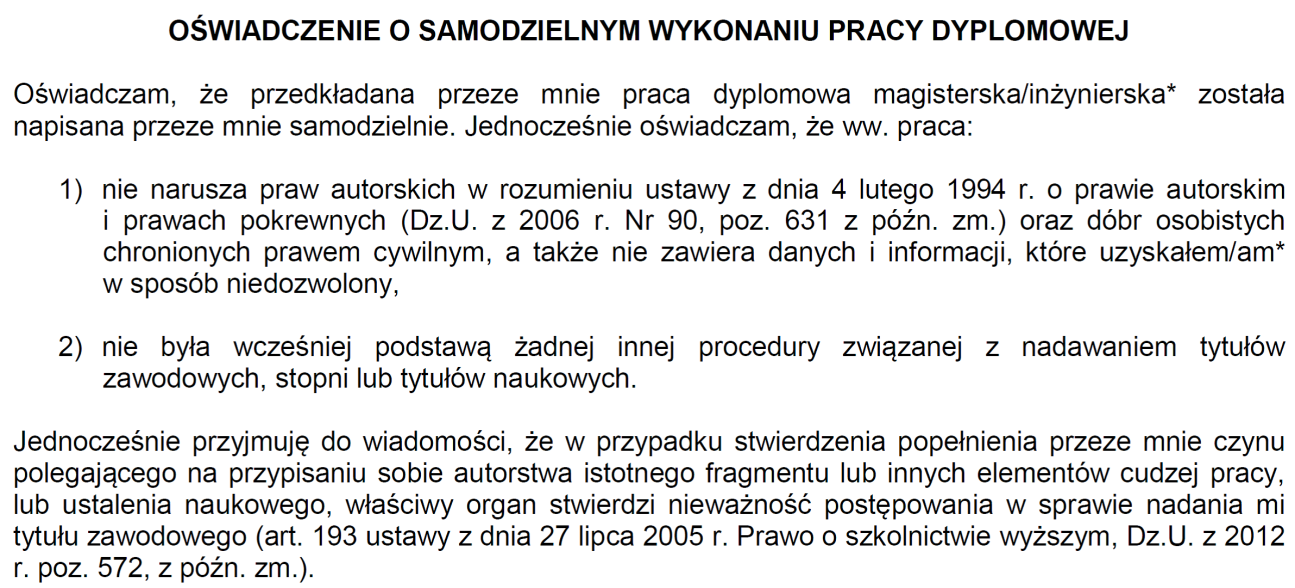
Promotor:

dr hab. inż. **Jacek Pietraszek**, prof. PK

Kraków, rok akad. 2017/2018

**Autor pracy: Tadeusz Baum-Azbum**

**Nr pracy: …………………………………………….**



………………………………

data i podpis

|  |
| --- |
| \* niepotrzebne skreślić |

|  |
| --- |
| **Uzgodniona ocena pracy: …………………………………………………………………..** |

**………………………………. .........................................**

podpis promotora podpis recenzenta

# SPIS TREŚCI

**1.** **Wstęp… 3**  
1.1. Cel pracy 5  
1.2. Zakres pracy 5

**2.** **Sieć Hopfielda 6**2.1. Opis i budowa sieci 6  
2.2. Nauka i odtwarzanie wzorców 7  
2.3. Implementacja sieci 9  
 2.3.1. Klasa Matrix 9  
 2.3.2. Klasa HopfieldNetwork 11

**3.** **Aplikacja rozpoznająca znaki 17**3.1. Specyfikacja aplikacji 17  
 3.1.1. Wymagania techniczne 17  
 3.1.2. Wymagania funkcjonalne 18  
3.2. Zastosowane technologie 18  
3.3. Architektura aplikacji 19  
3.4. Implementacja 20  
 3.4.1. Główne okno programu 21  
 3.4.2. Klasa NetworkViewModel 26

**4.** **Testy….. 28**

**5. Wnioski …….. 30**

**Literatura… 31**

**Summary…. 32**

# Wstęp

Przechowywanie wiedzy jest zagadnieniem towarzyszącym ludziom już od zarania dziejów. Można śmiało powiedzieć że gdyby nie wiedza, skrupulatnie utrwalane przez kolejne pokolenia naszych przodków, dzisiejsza cywilizacja wyglądała by zupełnie inaczej. Obecnie, dzięki postępowi techniki, każdy może z łatwością przechowywać wiedzę i dzielić się nią z innymi, a tradycyjne nośniki informacji powoli są wypierane przez ich cyfrowe odpowiedniki. Wciąż jednak dominująca w codziennym życiu formą zapisu informacji są nośniki papierowe.

Coraz częściej pojawia się więc potrzeba zamiany informacji zapisanej w formie papierowej, na informację możliwą do przetworzenia przez komputery. W niektórych sytuacjach może wystarczyć zwykłe użycie skanera, lub aparatu, w wyniku którego otrzymamy zdjęcie dokumentu w cyfrowej postaci, jednak stracimy możliwość jego modyfikacji. Do rozwiązania tego problemu wykorzystuje się techniki zwane OCR (ang. *Optical Character Recognition*).

W tej pracy podjęto się próby napisania aplikacji rozpoznającej pojedyncze litery zapisane w postaci cyfrowego obrazu i zamianę ich na odpowiedni kod znaku rozpoznawany przez komputer, przy pomocy rozwiązania opartego o sieci neuronowe. Jest to dobry punkt wyjścia do stworzenia systemu przetwarzającego całe słowa, a następnie całe strony i zamieniającego je na wirtualne dokumenty. Pełny kod projektu dostępny jest pod adresem https://github.com/Draeggiar/HopfieldNetworkOCR.

# Cel pracy

Celem pracy jest próba stworzenia aplikacji rozpoznającej znaki, w oparciu o implementację sieci Hopfielda. Aplikacja ta mogła by być później wykorzystana do budowy kompletnego systemu przetwarzania dokumentów, zamieniającego skan dokumentu na jego tekstowa reprezentację, możliwą do edycji na środowisku komputerowym.

### Zakres pracy

Praca składa się z dwóch części. Jedną jest zaprojektowanie i zaimplementowanie sieci neuronowej, zdolnej do zapamiętywania i odtwarzania wzorców, a następnie zaklasyfikowanie rozpoznanego wzorca do odpowiedniej kategorii. Drugą jest napisanie aplikacji przeznaczonej na systemy Windows, umożliwiającej wykorzystanie tej sieci do przetwarzania wp­­­­­rowadzonych przez użytkownika obrazów. W przypadku pomyślnego rozpoznania znaku wynikiem będzie komputerowo zakodowana litera.

Danymi wejściowymi dla programu będą binarne obrazy w formacie .tiff w rozmiarze 10x12 pikseli. Dodatkowo aby program poprawnie przyporządkował rozpoznany znak do odpowiedniej litery, zakłada się że nazwą pliku w danych uczących musi być litera którą ten plik zawiera.

# Sieć Hopfielda

Praca oparta została na sieci Hopfielda, która odpowiada za zapamiętywanie i odtwarzanie obrazów liter. Łatwość budowy i stosowania tej sieci powoduje że jest ona dobrym rozwiązaniem przedstawionego problemu. Dodatkowo sieć Hopfielda działa w oparciu o autoasocjację, dzięki czemu potrafi ona odtworzyć nawet silnie zniekształcony, lub niekompletny obraz [1]. W tym rozdziale opisany został proces jej tworzenia, oraz implementacji.

### Opis i budowa sieci

Sieć Hopfielda jest przykładem sieci rekurencyjnej i jednym z podstawowych przedstawicieli pamięci autoasocjacyjnych. Jest siecią jednowarstwową, a jej cechą charakterystyczna jest to, że wszystkie sygnały wyjściowe są jednocześnie sygnałami wejściowymi, tzn. wszystkie połączenia w tej sieci są sprzężeniami zwrotnymi (Rys. 2.1). Sprzężenie neuronu z samym sobą jest pomijane. Ponieważ w niniejszej pracy operuje się na danych binarnych, opisy dotyczyć będą dyskretnego modelu sieci.



*Rys. 2.1 Schemat sieci Hopfielda*

Jako dane wejściowe dostarczone zostaną obrazy o rozmiarze 10x12 pikseli, czyli zbudowane w sumie ze 120 pikseli. Każdy neuron sieci odpowiadać będzie jednemu pikselowi. Sygnał **1** na wyjściu neuronu, oznacza że piksel jest czarny, a **0** oznacza piksel biały[1].

Sieć będzie się więc składać ze 120 neuronów. Połączenia pomiędzy poszczególnymi neuronami będą przechowywane w tzw. Macierzy wag, która jest symetryczna: **M = MT**. Ponieważ każdy neuron jest połączony z każdym, macierz ta będzie miała rozmiar 120x120, a w miejscu połączenia do samego siebie przyjmujemy *w­ii = 0* [2]. Do uczenia sieci została wykorzystana reguła Hebba, której dokładniejszy opis znajduje się w kolejnym podrozdziale.

Funkcja aktywacji dla poszczególnych neuronów ma postać:

|  |  |  |
| --- | --- | --- |
|  |  | (1.) |

dla i = 1,2, …, n, gdzie:  
n – ilość neuronów  
w*­ij* – waga połączenia neuronu *i*-tego z *j*-tym, zawarta w macierzy wag

Pojemność tak zbudowanej sieci, przy względnym błędzie εmax = 1% (1% bitów wzorca w niewłaściwym stanie) stanowi około 13,8 % liczby neuronów [2]. Sieć powinna więc być w stanie zapamiętać do 16 wzorców.

# Nauka i odtwarzanie wzorców

W działaniu sieci wyróżniamy tryb uczenia, w którym sieci prezentowany jest zbiór danych uczących i w którym ustalają się wartości wag pomiędzy poszczególnymi neuronami, oraz tryb odtworzeniowy, w którym na podstawie ustalonych wag, sieć stara się odtworzyć jeden z zapamiętanych wzorców.

Proces uczenia sieci powoduje powstawanie pewnych obszarów przyciągania wokół punktów równowagi, którymi są dane uczące. Danymi uczącymi w tym przypadku będzie zbiór binarnych wektorów o długości 120. Faza ucząca ma za zadanie takie ukształtowanie wag w macierzy (**wij**), aby w trybie odtworzeniowym, prezentacja wzorca (**y**) zbliżonego do jednego z wektorów uczących (**x**), doprowadziła do ustalonego stanu, w którym stan neuronów **y = x** jest niezmienny w dowolnej chwili [2].

Poszczególne wagi w macierzy zostały ustalone przy pomocy uogólnionej reguły Hebba, w postaci:

|  |  |  |
| --- | --- | --- |
|  |  | (2.) |

Gdzie:  
x(k) – zbiór k wektorów uczących

Po zakończeniu procesu uczenia, wartości poszczególnych wag w macierzy są już ustalone i nie zmienne. W fazie odtworzeniowej, inicjujemy sieć wektorem **x**, a następnie oczekujemy na odpowiedź sieci w postaci [3]:

|  |  |  |
| --- | --- | --- |
|  |  | (3.) |

dla y(0) = x, gdzie:  
W – macierz wag

Proces ten powtarza się aż do ustalenia się odpowiedzi, czyli osiągnięcia przez sieć lokalnego minimum funkcji energetycznej. Istnieje zazwyczaj wiele minimów lokalnych, każde reprezentujących inny, ustalony na etapie uczenia stan sieci. W przestrzeni stanów, te lokalne minima zwane są atraktorami i reprezentują punkty stabilne sieci [2]. Do określania energii sieci wykorzystana została funkcja energetyczna Lapunowa o następującej postaci [3]:

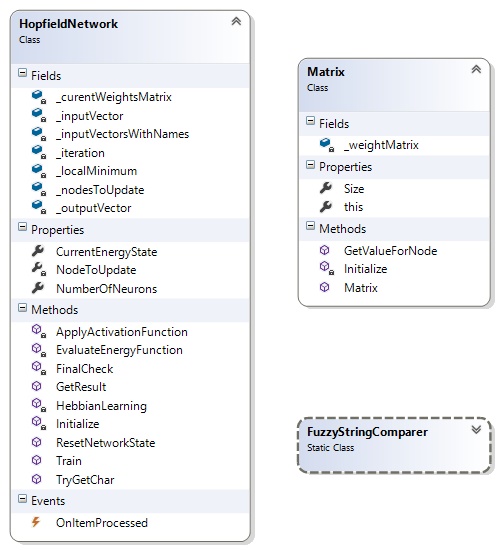
|  |  |  |
| --- | --- | --- |
|  |  | (4.) |

Gdzie:  
n – liczba neuronów w sieci  
x – wektor wejściowy  
y – wektor wyjściowy

Warto przy tym zaznaczyć że sieć od początku dążyć może do rozwiązania fałszywego. Może to wynikać np. z nakładania się na siebie różnych czynników danych wejściowych, w wyniku czego powstają fałszywe minima lokalne. Przy dużej liczbie wzorców mogą pojawić się również pośrednie minima lokalne, nie odpowiadające żadnemu z zapamiętanych wzorców. W trybie odtworzeniowym sieć może utknąć   
w jednym z takich minimów, powodując uzyskanie błędnego rozwiązania [2].

### Implementacja sieci

W tym rozdziale przedstawiona zostanie propozycja implementacji sieci Hopfielda w języku C#, która posłuży w kolejnym rozdziale jako moduł odpowiedzialny za rozpoznawanie znaków w kompletnej aplikacji. Jest ona jednak zawarta w osobnym pliku .dll, nic więc nie stoi na przeszkodzie, aby wykorzystać ją jako komponent w innym projekcie. Implementacja sieci składa się z dwóch głównych klas i zewnętrznej biblioteki, a jej ogólny schemat został przedstawiony na poniższym diagramie klas (Rys. 2.2).



*Rys.2.2 Diagram klas użytych do implementacji sieci*

### Klasa Matrix

Klasa ta jest reprezentacją macierzy wag, i będzie służyć do przechowywania wag połączeń pomiędzy poszczególnymi neuronami sieci. Wagi są przechowywane w dwuwymiarowej tablicy typu *double*, a łatwiejszy dostęp do macierzy został osiągnięty dzięki użyciu indeksera.

Taka konstrukcja zapewnia odpowiedni poziom hermetyzacji danych charakterystyczny dla programowania obiektowego, a przy tym ułatwia korzystanie z naszej klasy macierzy, dzięki możliwości stosowania przejrzystego zapisu przy odwoływania się do danych [4].

Tworzenie nowej macierzy współczynników odbywa się poprzez zainicjowanie jej wektorem początkowym w konstruktorze.

public Matrix(string inputVector)

{

Size = inputVector.Length;

\_weightMatrix = new double[Size, Size];

Initialize(inputVector);

}

private void Initialize(string inputVector)

{

for (int i = 0; i < inputVector.Length; i++)

{

for (int j = 0; j < inputVector.Length; j++)

{

if (i == j)

\_weightMatrix[i, j] = 0;

else

\_weightMatrix[i, j] =

(2 \* int.Parse(inputVector[i].ToString()) - 1)

\* (2 \* int.Parse(inputVector[j].ToString()) - 1);

}

}

}

Na podstawie długości wektora tworzymy macierz o odpowiedniej wielkości. Następnie metoda *Initialize* odpowiada za wypełnienie macierzy wartościami początkowymi na podstawie podanego wektora, korzystając z zależności dla wzorców binarnych [5]:

|  |  |  |
| --- | --- | --- |
|  |  | (5.) |

dla i ≠ j, gdzie:  
w – waga połączenia neuronu i-tego z j-tym  
x – wektor wejściowy

Miejsca sprzężenia neuronu z własnym wyjściem (i = j) wypełniane są zerami.

Metoda *GetValueForNode* jest wykorzystywana przy obliczaniu funkcji aktywacji zgodnie z wcześniej opisanym wzorem (1).

### Klasa HopfieldNetwork

W tej klasie została zawarta obsługa wszystkich operacji wykonywanych w sieci, dlatego jest ona bardziej rozbudowana, ale jej użycie jest bardzo intuicyjne. Ten rozdział skupia się na trybie uczenia i odtwarzania zaimplementowanych w tej klasie.

Pracę z siecią rozpoczynamy od utworzenia jej obiektu. Następnie, przy pomocy metody *Train* prezentujemy jej zbiór wzorców do zapamiętania, wraz z literami, które reprezentuje każdy wzorzec. Metoda przyjmuje ten zbiór w formie słownika, gdzie kluczem jest dana litera, a wartością, wzorzec jej odpowiadający.

public void Train(Dictionary<string, string> inputVectorsWithNames)

{

\_inputVectorsWithNames = inputVectorsWithNames;

var vectors = inputVectorsWithNames.Values.ToList();

Initialize(vectors.First());

HebbianLearning(vectors);

}

Przed rozpoczęciem faktycznego procesu nauki, za pomocą pierwszego wektora inicjalizowana jest macierz wag, wektor wejściowy, wyjściowy, oraz minimum lokalne:

private void Initialize(string input)

{

\_inputVector = input;

\_curentWeightsMatrix = new Matrix(input);

\_outputVector = input;

\_localMinimum = new KeyValuePair<double, string>(double.MaxValue,

string.Empty);

}

Tak przygotowana sieć rozpoczyna proces nauki zgodnie z uogólnioną regułą Hebba (2).

private void HebbianLearning(List<string> inputVectors)

{

var trainEventArgs =

new ProcessEventArgs(\_curentWeightsMatrix.Size);

for (int i = 0; i < \_curentWeightsMatrix.Size; i++)

{

for (int j = 0; j < \_curentWeightsMatrix.Size; j++)

{

var weights = 0.0;

if (i != j)

{

foreach (string vector in inputVectors)

{

weights +=

(2 \* double.Parse(vector[i].ToString()) - 1)

\* (2 \* double.Parse(vector[j].ToString()) - 1);

}

}

\_curentWeightsMatrix[i, j] =

weights / \_curentWeightsMatrix.Size;

}

trainEventArgs.CurrentItem = i + 1;

OnItemProcessed?.Invoke(this, trainEventArgs);

}

}

W miejscu połączenia neuronu z samym sobą (i = j) wpisywane jest 0.

Warto zwrócić uwagę na wykorzystanie zdarzenia *OnItemProcessed*, które po zasubskrybowaniu, może zostać użyte do wyświetlenia informacji o aktualnie przetwarzanym wzorcu, bez konieczności dostępu z zewnątrz do klasy HopfieldNetwork. Jest to praktyczne podejście, dzięki któremu warstwa logiki aplikacji, została oddzielona od warstwy widoku i może być użyta jako samodzielny komponent w innym projekcie [4].

public event EventHandler<ProcessEventArgs> OnItemProcessed;

public class ProcessEventArgs : EventArgs

{

public int ItemsCount;

public int CurrentItem;

internal ProcessEventArgs(int items)

{

ItemsCount = items;

}

}

Po zakończeniu nauki, można przejść do trybu odtworzeniowego, a sieć na podstawie zadanego wektora, spróbuje dopasować go do zapamiętanych wcześniej wzorców. Do tego celu służy funkcja *GetResult*. Jest to rekurencyjna funkcja, implementująca proces odtwarzania opisany w rozdziale 2.2, a jej wynikiem jest wektor wygenerowany przez sieć.

public string GetResult(string input)

{

if (\_curentWeightsMatrix.Size != input.Length)

throw new ArgumentException("Wrong image size");

OnItemProcessed?.Invoke(this,

new ProcessEventArgs(int.MaxValue) {

CurrentItem = \_iteration++});

\_inputVector = input;

var savedEnergyState = CurrentEnergyState;

var output = new StringBuilder(input);

var nodeToUpdate = NodeToUpdate;

output[nodeToUpdate] =

ApplyActivationFunction(input, nodeToUpdate);

\_outputVector = output.ToString();

if (CurrentEnergyState < \_localMinimum.Key)

\_localMinimum =

new KeyValuePair<double, string>(

CurrentEnergyState, \_outputVector);

if (CurrentEnergyState > savedEnergyState

|| \_nodesToUpdate.Count > 0

&& !FinalCheck(\_outputVector))

GetResult(\_outputVector);

return \_localMinimum.Value;

}

Na początku długość wektora jest porównywana z wielkością macierzy, ponieważ obraz który chcemy rozpoznać musi być takiego samego rozmiaru jak obrazy użyte w procesie nauki. Jeżeli rozmiar się zgadza, wywołujemy zdarzenie *OnItemProcessed*, które zawiera informację o aktualnej iteracji. Następnie obliczana jest energia sieci, zgodnie ze wzorem funkcji energetycznej Lapunowa (4) :

public double CurrentEnergyState => EvaluateEnergyFunction();

private double EvaluateEnergyFunction()

{

double energy = 0;

for (int i = 0; i < \_curentWeightsMatrix.Size; i++)

{

for (int j = 0; j < \_curentWeightsMatrix.Size; j++)

{

energy += double.Parse(\_outputVector[i].ToString())

\* double.Parse(\_outputVector[j].ToString())

\* \_curentWeightsMatrix[i, j];

}

}

energy = energy \* (1.0 / 2.0);

for (int i = 0; i < \_curentWeightsMatrix.Size; i++)

{

energy -= double.Parse(\_inputVector[i].ToString())

\* double.Parse(\_outputVector[i].ToString());

}

return energy;

}

Kolejnym krokiem jest uzyskanie odpowiedzi od sieci dla konkretnego neuronu, wybranego w sposób losowy. Odpowiedź sieci ustala się poprzez nałożenie funkcji aktywacji (3) na element wektora wejściowego odpowiadający wybranemu neuronowi w następujący sposób:

private char ApplyActivationFunction(string input, int nodeToUpdate)

{

var neuronWeight =

\_curentWeightsMatrix.GetValueForNode(input, nodeToUpdate);

if (neuronWeight > 0)

return '1';

else if (neuronWeight < 0)

return '0';

return input[nodeToUpdate];

}

Wiedząc że nasze rozwiązanie powinno znajdować się w miejscu gdzie funkcja energii ma najmniejszą wartość, ponownie obliczamy energię sieci i porównujemy ją z dotychczasowym lokalnym minimum. Jeżeli jej wartość jest mniejsza, zapisujemy ją jako nowe lokalne minimum, wraz z uzyskanym wektorem. Ostatnim krokiem jest uwzględnienie warunków końcowych. Nie spełnienie któregoś z warunków skutkuje rekurencyjnym wywołaniem funkcji *GetResult*, gdzie wektorem wejściowym będzie wektor uzyskany w tej iteracji. Najpierw aktualny stan energii sieci jest porównywany ze stanem sprzed zmiany. Jeżeli jej stan zwiększył się, to znaczy że oddalamy się od rozwiązania. Następnie sprawdzamy, każdy neuron w sieci nałożył swoją odpowiedź na wektor, ponieważ uzyskane rozwiązanie powinno być zbudowane w całości z danych zwróconych przez sieć. Na koniec pobierana jest jeszcze raz odpowiedź dla każdego neuronu po kolei i jeżeli nic się nie zmieniło, to rozwiązanie można uznać za stabilne i funkcja zwraca zapisaną wartość wektora z lokalnego minimum.

private bool FinalCheck(string vectorToCheck)

{

var output = new StringBuilder(vectorToCheck);

for (int i = 0; i < \_curentWeightsMatrix.Size; i++)

{

output[i] = ApplyActivationFunction(vectorToCheck, i);

if (output.ToString() != vectorToCheck)

return false;

}

return true;

}

Tak uzyskany wektor jest gotowy do zapisania jako obraz i wyświetlenia, ale nie niesie żadnej informacji o znaku jaki przedstawia. Zwykłe porównanie czy wektor jest identyczny z jednym z wektorów uczących nie wystarczy, ponieważ odpowiedź wygenerowana przez sieć może się różnić, mimo że dla człowieka wciąż będzie rozpoznawalna jako konkretna litera. Dlatego do próby odczytania litery z odpowiedzi sieci służy osobna funkcja *TryGetChar*. Do poprawnego działania wymaga ona aby nazwy obrazów w zbiorze danych uczących odpowiadały właściwym literą zawartym na obrazach.

public bool TryGetChar(string input, out char recognizedCharacter)

{

recognizedCharacter = '\0';

var isKnownChar = false;

foreach (var vector in \_inputVectorsWithNames)

{

if (string.IsNullOrEmpty(vector.Key) || vector.Key.Length > 1)

throw new ArgumentException(

"Cannot associate output with letter.

Check file names.");

if(!string.Equals(input, vector.Value))

if (!FuzzyStringComparer.Equals(input, vector.Value))

continue;

isKnownChar = true;

recognizedCharacter = char.Parse(vector.Key);

}

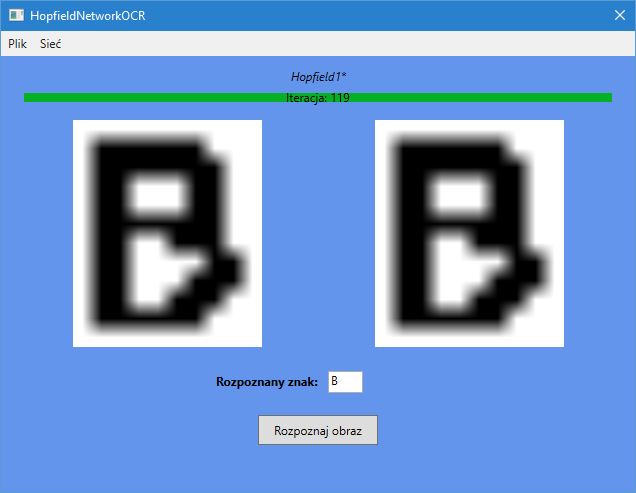
return isKnownChar;

}

Jako argumenty przyjmuje ona wektor uzyskany w wyniku działania funkcji *GetResult*, oraz zmienną typu *char*, w której zapisana zostanie rozpoznana litera. Działanie funkcji jest bardzo proste. Najpierw w zbiorze danych uczących szukane jest dokładne dopasowanie do wprowadzonego wzorca, a w przypadku nie powodzenia szukane jest dopasowanie oparte na aproksymacji. Do tego celu została użyta zewnętrzna biblioteka open source *FuzzyString*, dostępna pod adresem https://github.com/kdjones/fuzzystring w ramach licencji EPL-1.0. Zawiera ona implementacje algorytmów służących do porównywania obiektów tekstowych. Do porównywania wektorów w tej implementacji wykorzystane zostały: odległość Hamminga (ang. *Hamming Distance*), współczynnik nakładania się (ang. *Overlap Coefficient*), najdłuższy wspólny podciąg (ang. *Longest Common Subsequence*), najdłuższy wspólny podłańcuch (ang. *Longest Common Substring*), oraz podobieństwo Ratcliff’a-Obershelp’a (ang. *Ratcliff-Obershelp Similarity)*[6]. W przypadku udanego dopasowania funkcja zwróci *true* a do podanej zmiennej zostanie zapisana dopasowana litera. W przeciwnym wypadku funkcja zwróci *false,* a do zmiennej zostanie wpisany symbol pustego znaku.

# Aplikacja rozpoznająca znaki

W tym rozdziale znajduje się opis głównego przedmiotu pracy, jakim jest utworzenie działającej aplikacji, będącej w stanie rozpoznać i wyświetlić komputerowo zakodowaną literę, na podstawie zadanego obrazu.



*Rys. 3.1 Główne okno programu*

### Specyfikacja aplikacji

W tym podrozdziale zawarta jest specyfikacja aplikacji. Ponieważ program nie jest skomplikowany i założenia ma być prosty w obsłudze, przedstawione wymagania zdefiniowane są bardzo ogólnie.

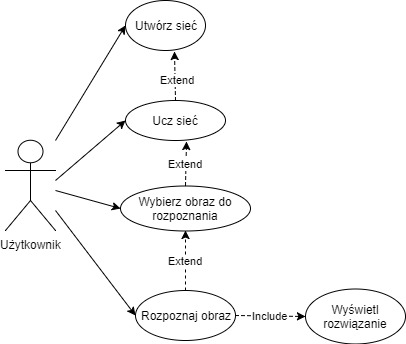
### Wymagania techniczne

Aplikacja jest przeznaczona na komputery z systemem Windows, oraz zainstalowanym zestawem bibliotek .Net Framework w wersji 4.5.2. Jest samodzielna i jej instalacja nie jest konieczna.

### Wymagania funkcjonalne

Aplikacja powinna posiadać interfejs graficzny, a interakcja powinna odbywać się wyłącznie za pomocą myszki. Użytkownik aplikacji powinien móc wykonywać następujące czynności przedstawione na diagramie przypadków użycia (Rys. 3.2):

* + - Utworzenie nowej sieci
    - Wybranie danych uczących i rozpoczęcie nauki istniejącej sieci
    - Wybór obrazu do rozpoznania
    - Rozpoczęcie procesu rozpoznawania, którego wynik powinien być wyświetlony na ekranie



*Rys. 3.2 Diagram przypadków użycia*

Ponadto interfejs aplikacji powinien być uproszczony i przejrzysty, aby praca z programem była intuicyjna i nie wymagała dodatkowych instrukcji. Program powinien być stabilny, a w przypadku wystąpienia błędu, informować o nim użytkownika w przejrzystym komunikacie.

### Zastosowane technologie

Całości implementacji wykonana jest w języku **C#**, w programie **Microsoft Visual Studio 2015 Community Edition** i oparta jest o platformę **.Net**. Wykorzystuje zestaw bibliotek i komponentów **Microsoft .Net Framework 4.5.2**. Do utworzenia interfejsu graficznego wykorzystany został silnik graficzny i API **Windows Presentation Foundation**, będący częścią platformy .Net [7]. Do aproksymacji łańcuchów znaków została wykorzystana zewnętrzna biblioteka **FuzzyString**, o której więcej można przeczytać w rozdziale 3.3.2. Testy jednostkowe zostały napisane w oparciu o zestaw narzędzi **Visual Studio Unit Testing Framework**, zintegrowany z programem Visual Studio.

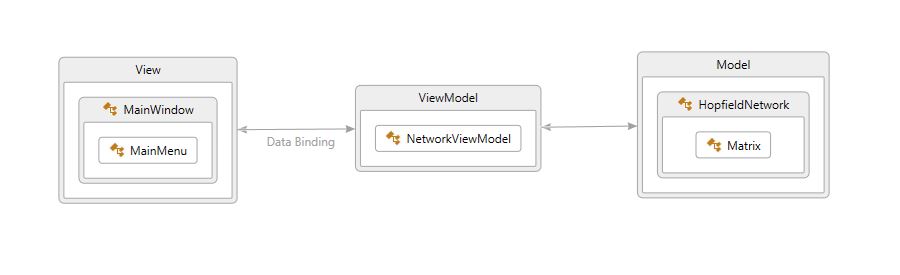
### Architektura aplikacji

Aplikacja została oparta o wzorzec MVVM (ang. *Model-View-Viewmodel)* (Rys. 3.3)*.* Dzięki takiemu podejściu warstwa logiki została oddzielona od warstwy widoku, co umożliwia łatwiejsze zarządzanie kodem i dodawanie nowych modułów w przyszłości, a ponadto przyspiesza implementację [8].



*Rys. 3.3 Schemat wzorca MVVM (źródło: https://en.wikipedia.org/wiki/Model-view-viewmodel#/media/File:MVVMPattern.png)*

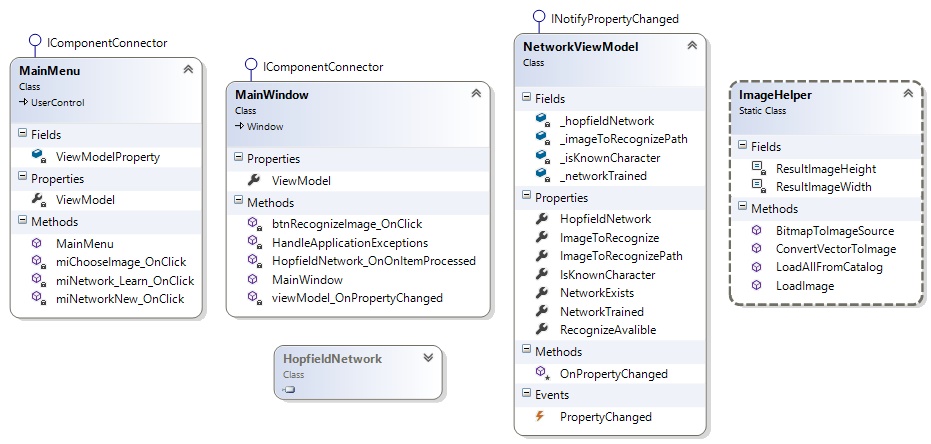
Modelem w tym wypadku jest implementacja sieci Hopfielda, widokiem będzie interfejs graficzny stworzony w oparciu o WPF, a pośredniczyć pomiędzy nimi będzie specjalnie napisana klasa *NetworkViewModel*, reprezentująca ViewModel. Ogólną architekturę tego wzorca w odniesieniu do aplikacji można zobaczyć na rysunku 3.4.



*Rys. 3.4 Wzorzec MVVM w projekcie*

### Implementacja

Najważniejszym elementem implementacji jest model, to wokół niego skupia się cała architektura aplikacji. Modelem jest implementacja sieci Hopfielda, której został poświęcony osobny rozdział 3.3, tutaj znajdować się będą jedynie opisy użycia gotowej klasy *HopfieldNetwork*. Interfejs graficzny składa się z dwóch komponentów: głównego okna programu (klasa *MainWindow*), oraz menu kontekstowego (klasa *MainMenu*). Ponieważ są one częścią standardu WPF, składają się z części code-behind napisanej w C#, oraz z części odpowiedzialnej za wygląd, napisanej w XAML (ang. *Extensible Application Markup Language*) [7]. Częścią pośredniczącą w wymianie danych pomiędzy interfejsem, a modelem, jest klasa *NetworkViewModel*, zgodnie ze standardem MVVM, będąca ViewModelem. Oprócz wyżej wymienionych klas, aplikacja składa się jeszcze ze statycznej klasy pomocniczej *ImageHelper*, zawierającej metody ułatwiające pracę z plikami obrazów (odczyt z dysku, zamiana ich na wektory itp.).   
Ogólny schemat implementacji widoczny jest na poniższym diagramie klas (Rys. 3.5):



*Rys. 3.5 Diagram klas wykorzystanych do budowy aplikacji*

### Główne okno programu

Wygląd głównego okna programu przedstawiony został na rysunku 4.1. Ma ono wymiary 500x650px. Projekt interfejsu oparty został o siatkę z 6 kolumnami i 9 wierszami. Tworzenie poszczególnych kontrolek w WPF odbywa się poprzez umieszczanie ich na wcześniej zdefiniowanej siatce i ustalaniu ich właściwości udostępnionych przez silnik. Poniżej znajduje się fragment kodu definiujący kontenery, w których wyświetlane są obrazki:

<Image Name="imgInput" Stretch="Uniform"

Grid.Column="1" Grid.ColumnSpan="2"

Grid.Row="4"

Margin="10px"

Source="{Binding ViewModel.ImageToRecognize}"/>

<Image Name="imgOutput" Stretch="Uniform"

Grid.Column="3" Grid.ColumnSpan="2"

Grid.Row="4"

Margin="10px"/>

Warto zwrócić uwagę na właściwość *Source* pierwszego obrazka. Zawiera ona aktualnie wyświetlany obraz jako obiekt klasy *BitmapImage*. Zapis *{Binding … }* oznacza że właściwość ta została połączona (dosł. zbindowana) do innej właściwości, znajdującej się w tym wypadku w ViewModelu. Gdy tylko obiekt *BitmapImage* zostanie przypisany do właściwości *ImageToRecognize* w znajdującej się w ViewModelu, dzięki temu połączeniu, zostanie on automatycznie wyświetlony na ekranie [9]. Więcej o tym mechanizmie przeczytać można w rozdziale 3.4.2.

W części *code-behind* głównego okna, znajduje się obsługa zdarzeń związanych z operacjami wykonywanymi w głównym oknie programu. Najważniejsza jest tutaj obsługa zdarzenia *btnRecognizeImage\_OnClick*, które wywoływane jest po kliknięciu przez użytkownika przycisku „*Rozpoznaj obraz*” (Rys. 3.1).

public static NetworkViewModel ViewModel { get; private set; }   
   
private void btnRecognizeImage\_OnClick(object sender, RoutedEventArgs e)

{

imgOutput.Source = null;

txtRecognizedChar.Text = string.Empty;

try

{

var imageContent = ImageHelper.LoadImage(

ViewModel.ImageToRecognizePath);

var resultImage =

ViewModel.HopfieldNetwork.GetResult(imageContent);

imgOutput.Source = ImageHelper.BitmapToImageSource(

ImageHelper.ConvertVectorToImage(resultImage));

char recognizedChar;

if (ViewModel.HopfieldNetwork.TryGetChar(

resultImage, out recognizedChar))

{

ViewModel.IsKnownCharacter = true;

txtRecognizedChar.Text = recognizedChar.ToString();

}

else

ViewModel.IsKnownCharacter = false;

ViewModel.HopfieldNetwork.ResetNetworkState();

}

catch (Exception ex)

{

txtOutput.Text = ex.Message;

}

}

Ponieważ widok nie operuje bezpośrednio na modelu, obiekt klasy *NetworkViewModel* będzie pośredniczyć w operacjach związanych z siecią. Dzięki odpowiedniemu bindingowi danych, przycisk jest widoczny tylko gdy sieć już istnieje i zakończyła proces uczenia, tak więc możliwe jest wczytanie obrazu wejściowego bez warunków sprawdzających stan sieci. W trakcie przetwarzania obrazu przez sieć, na pasku postępu wyświetlana jest aktualna iteracja w procesie odtwarzania, dzięki czemu wyeliminowano wrażenie że program się zawiesił. Odpowiedź wygenerowana przez sieć w formie wektora, jest zamieniania na obraz, a następnie wyświetlana na ekranie, po prawej stronie głównego okna (Rys. 3.1). W razie wyjątku w trakcie wykonywania którejkolwiek operacji, wyjątek zostanie złapany, a komunikat wyświetlony na ekranie, bez przerywania działania programu. W przypadku pomyślnego rozpoznania znaku, zostaje on wypisany w oknie dialogowym na dole (Rys. 3.1). Na koniec stan sieci jest resetowany, aby po kolejnym kliknięciu sieć mogła wygenerować nową odpowiedź.

W klasie głównego okna znajduje się również obsługa wspomnianego w rozdziale 2.3.2 zdarzenia *OnItemProcessed*. Służy ona do wyświetlania postępów operacji wykonywanych przez sieć. Nie musimy przy tym ingerować w wewnętrzny stan sieci, na bieżąco przesyła nam ona informacje o postępach wraz z kolejnymi wywołaniami wspomnianego zdarzenia. Otrzymane informacje są w asynchroniczny sposób wyświetlane na ekranie, dzięki czemu główne okno pozostaje responsywne, mimo ciągłego odświeżania.

private void HopfieldNetwork\_OnOnItemProcessed(object sender,

ProcessEventArgs processEventArgs)

{

if (processEventArgs.ItemsCount == int.MaxValue)

{

txtStatus.Text = "Iteracja: " + processEventArgs.CurrentItem;

pbStatus.Dispatcher.Invoke(

() => pbStatus.Value = 100,

DispatcherPriority.Background);

}

else

{

var progresPercentage = (double) processEventArgs.CurrentItem

/ (double) processEventArgs.ItemsCount

\* 100.0;

txtStatus.Text = "Postęp nauki: " + $"{progresPercentage:0}%";

pbStatus.Dispatcher.Invoke(

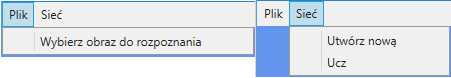
() => pbStatus.Value = progresPercentage,

DispatcherPriority.Background);

}

}

Na koniec pozostał najważniejszy element głównego okna programu, czyli górne menu kontekstowe (Rys. 3.6). To w nim odbywa się główna interakcja użytkownika z programem. Dostępne są w nim dwie zakładki: *Plik*, zawierająca opcję do wyboru obrazu wejściowego do przetworzenia, oraz *Sieć*, zawierająca opcje stworzenia nowej, oraz uczenia istniejącej sieci. Opcje sprzeczne z logią aplikacji są wyszarzone i zmieniają się na bieżąco, tzn. że nie możemy rozpocząć nauki do póki nie stworzymy sieci i nie możemy wybrać obrazu do rozpoznania, dopóki proces nauki nie zostanie zakończony.



*Rys. 3.6 Wygląd menu kontekstowego*

Menu jest osobną kontrolką, znajdującą się w klasie *MainMenu* i tak jak główne okno zbudowane jest z dwóch części. Wydzielenie menu jako osobnego komponentu daje większą kontrolę nad faktycznym wyglądem aplikacji i nie zaciemnia kodu głównego okna. Dodanie go do okna odbywa się tak jak dodanie każdego innego predefiniowanego komponentu.

<userControls:MainMenu x:Name="MainMenu" Grid.Row="0" Grid.ColumnSpan="6"

VerticalAlignment="Top" HorizontalAlignment="Left"

Width="650"/>

W pliku *code-behind* dla menu, znajdują się metody obsługujące zdarzenia wyboru poszczególnych opcji z menu. Przed dokładnym omówieniem ich warto jednak zwrócić uwagę na jeszcze jeden mechanizm wykorzystany w tej klasie.   
Jak wcześniej wspomniano, dostępne do wyboru opcje zmieniają się na bieżąco. Zostało to uzyskane dokładnie tak jak w głównym oknie, tzn. właściwość *IsEnabled* poszczególnych przycisków została zbindowana do odpowiednich właściwości ViewModelu. Różni się jednak sposób w jaki uzyskano dostęp do tych właściwości.

<MenuItem Name="miChooseImage" Header="\_Wybierz obraz do rozpoznania"

IsEnabled="{Binding ViewModel.NetworkTrained}"

Click="miChooseImage\_OnClick"/>

Ponieważ obiekt zawierający ViewModel znajduje się w głównym oknie i może często zmieniać stan, zastosowano tutaj tak zwaną właściwość zależną (ang. *Dependency property*).

private static readonly DependencyProperty ViewModelProperty =

DependencyProperty.Register("ViewModel",

typeof(NetworkViewModel),

typeof(MainMenu));

private NetworkViewModel ViewModel

{

get { return GetValue(ViewModelProperty) as NetworkViewModel; }

set { SetValue(ViewModelProperty, value); }

}

public MainMenu()

{

ViewModel = MainWindow.ViewModel;

InitializeComponent();

}

Dzięki takiej konstrukcji operacje przeprowadzamy bezpośrednio na ViewModelu z klasy głównego okna, bez konieczności tworzenia dodatkowej zmiennej. Oszczędzany jest więc czas, oraz miejsce w pamięci. Dodatkowo jeżeli którakolwiek z właściwości w ViewModelu ulegnie zmianie, widok natychmiast dostaje o tym informację i zbindowana właściwość również zmienia wartość, bez konieczności pisania dodatkowego kodu obsługującego tą zmianę stanu [10, 11].

Logika ukryta pod przyciskami menu prezentuje się następująco:

private void miNetworkNew\_OnClick(object sender, RoutedEventArgs e)

{

ViewModel.HopfieldNetwork = new HopfieldNetwork();

}

private void miNetwork\_Learn\_OnClick(object sender, RoutedEventArgs e)

{

using (var fbd = new FolderBrowserDialog())

{

fbd.Description = "Wybierz katalog z danymi uczącymi";

if (fbd.ShowDialog() !=

DialogResult.OK

|| string.IsNullOrWhiteSpace(fbd.SelectedPath)) return;

var leariningData = ImageHelper.

LoadAllFromCatalog(fbd.SelectedPath);

ViewModel.HopfieldNetwork.Train(leariningData);

ViewModel.NetworkTrained = true;

}

}

private void miChooseImage\_OnClick(object sender, RoutedEventArgs e)

{

FileDialog fileChoseDialog = new OpenFileDialog

{

Multiselect = false,

Filter = "Tiff files|\*.tiff",

Title = "Wybierz obraz do rozpoznania"

};

if (fileChoseDialog.ShowDialog() != true) return;

ViewModel.ImageToRecognizePath = fileChoseDialog.FileName;

}

Przycisk „*Utwórz nową*” tworzy nowy obiekt klasy *HopfieldNetwork*. Ponieważ obiekt ten jest przechowywany w ViewModelu, z każdego miejsca w programie mamy łatwy dostęp do tego samego obiektu sieci. Pomyślne utworzenie sieci program sygnalizuje poprzez pojawienie się pustego paska postępu nauki i nazwy sieci (Rys. 3.1). Przycisk „*Ucz*” otwiera okno dialogowe wyboru katalogu z danymi uczącymi. Po wybraniu folderu, metoda pomocnicza ładuje wszystkie obrazy o rozszerzeniu .tiff i zamienia je na zero-jedynkowe wektory. Następnie zbiór tych wektorów prezentowany jest sieci, która rozpoczyna proces nauki zgodnie z opisem z rozdziału 2.2. Postęp nauki widoczny jest na górze okna w postaci wartości procentowej, oraz paska postępu. Przycisk „*Wybierz obraz do rozpoznania*” otwiera okno dialogowe wyboru obrazu wejściowego w formacie .tiff. Ścieżka do wybranego pliku zapisywana jest we właściwości ViewModelu, a wybrany obraz pojawia się w głównym oknie, po jego lewej stronie (Rys.3.1).

### Klasa NetworkViewModel

Klasa ta jest zbiorem właściwości, wyciągniętymi z modelu, w celu zbindowania ich do odpowiednich elementów interfejsu graficznego. Jako przykład niech posłuży właściwość *NetworkExists*, od której zależy czy przycisk „*Ucz*” jest dostępny, oraz od której zależy widoczność paska postępów.

private HopfieldNetwork \_hopfieldNetwork;

public HopfieldNetwork HopfieldNetwork

{

get { return \_hopfieldNetwork; }

set

{

\_hopfieldNetwork = value;

OnPropertyChanged(nameof(NetworkExists));

}

}

public bool NetworkExists => HopfieldNetwork != null;

Zwraca ona *true* jeżeli przypisaliśmy my obiekt do właściwości *HopfieldNetwork*, w przeciwnym razie zwróci ona *false*. Jednak kod zostanie wykonany jedynie w momencie kompilacji, a w tym wypadku obiekt będziemy tworzyć klikając na przycisk, już w trakcie działania programu. Dlatego w setterze właściwości *HopfieldNetwork*, znajduje się wywołanie metody *OnPropertyChanged* gdzie argumentem jest nazwa właściwości która uległa zmianie w wyniku tego przypisania. Jest to metoda implementowana przez interfejs *INotifyPropertyChanged*, dzięki któremu wszystkie obiekty korzystające z właściwości *NetworkExists* zostaną poinformowane o zmianie jej wartości, i uaktualnią swój stan w czasie rzeczywistym. Implementacja interfejsu *INotifyPropertyChanged* wygląda następująco:

public class NetworkViewModel : INotifyPropertyChanged

{

public event PropertyChangedEventHandler PropertyChanged;

// ...

[NotifyPropertyChangedInvocator]

protected virtual void OnPropertyChanged(

[CallerMemberName] string propertyName = null)

{

PropertyChanged?.Invoke(this,

new PropertyChangedEventArgs(propertyName));

}

}

Gdy do tej właściwości zbindujemy przycisk w sposób pokazany poniżej, będzie on wyszarzony dopóki do *HopfieldNetwork* nie zostanie przypisany obiekt. W łatwy sposób i w niewielkiej ilości kodu, logika sprawdzająca czy obiekt istnieje została przeniesiona na faktyczny widok [11].

<MenuItem Name="miNetworkLearn" Header="\_Ucz"

IsEnabled="{Binding ViewModel.NetworkExists}"

Click="miNetwork\_Learn\_OnClick"/>

Pozostałe właściwości znajdujące się w tej klasie działają analogicznie. Ich wartości reprezentują pewne stany w modelu, które za pomocą bindingu przenoszone są na widok.

# Testy

Oprócz implementacji, projekt obejmował trzy rodzaje testów funkcjonalnych. Pierwszym z nich były testy jednostkowe, pisane na równolegle z implementacją poszczególnych modułów, co pozwoliło na bieżąco znajdować i eliminować błędy w kodzie. Testy zostały napisane zgodnie ze wzorcem 3A (Arrange, Act, Assert), dzięki czemu są przejrzyste i jasno określają przedmiot testów w danym fragmencie kodu.   
W sumie napisanych zostało 9 testów jednostkowych, testujących główne komponenty sieci i metody pomocnicze.

Po zakończeniu implementacji, gotowa aplikacja została skonfrontowana z wymaganiami zawartymi w specyfikacji. Aplikacja końcowa spełnia wszystkie te wymagania, oraz założenia projektowe.

Na koniec działanie aplikacji zostało przetestowane z następującymi danymi wejściowymi:

* Zestaw wszystkich liter wyeksportowanych z czcionki Arial (Rys. 4.1):

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| *Rys. 4.1 Litery w czcionce Arial* | | | | | | | | | | | | |

Program poprawnie odtworzył i zaklasyfikował 3 z 25 liter.

* Zestaw specjalnie przygotowanych liter (Rys. 4.2) [1]:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |
|  | *Rys. 4.2 Zestaw przygotowanych liter* | | | | | | | | | | |

program poprawnie odtworzył i zaklasyfikował 5 z 11 liter.

* Zestaw figur geometrycznych (prostokąt, trójkąt, koło) i ich zaburzone wersje (Rys. 4.3):

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| *Rys. 4.3 Figury geometryczne i ich zaburzone odpowiedniki* | | |

Głównym przeznaczeniem aplikacji jest rozpoznawanie znaków, ale sieć Hopfielda potrafi odtwarzać dowolne kształty, dla tego jej działanie zostało również przetestowane na prostych figurach geometrycznych. Program poprawnie odtworzył koło i trójkąt, z czego po zaprezentowaniu zaburzonego wzorca, program potrafił odtworzyć jedynie trójkąt.

Wszystkie wzorce użyte do testów znajdują się w katalogu *TrainingData,* w repozytorium z kodem.

# Wnioski

Celem niniejszej pracy było stworzenie aplikacji rozpoznającej znaki na obrazach w oparciu o sieć Hopfielda. O ile sama implementacja sieci i stworzenie na jej bazie programu przebiegło pomyślnie, to uzyskane rezultaty nie są zadowalające. Pojedyncza sieć Hopfielda o 120 neuronach jest za mała, aby pomieścić wszystkie wzorce uczące liter. W wyniku tego wykres funkcji energetycznej jest pełen zakłóceń związanych z nakładaniem na siebie wzorców i rozwiązaniem często są fałszywe minima lokalne. Sieć nie radzi sobie zbyt dobrze z rozpoznawaniem nawet wzorców zawartych bezpośrednio w danych uczących i użycie jej do rozpoznawania wzorców zaburzonych, np. pisma ręcznego nie było by zbyt efektywne. Jak pokazują testy aplikacji, wraz ze spadkiem liczby wzorców, zwiększa się ilość poprawnie rozpoznanych elementów.

Lepsze rezultaty mogły by zostać osiągnięte np. poprzez dodanie *biasu* do funkcji energetycznej, w celu lepszego nakierowania jej na rozwiązanie, lub podział sieci na kilka mniejszych, gdzie każda z nich zajmowała by się przetwarzaniem innego fragmentu obrazu, a wynikiem była by odpowiednia suma ich odpowiedzi. Trzeba jednak pamiętać że sieć Hopfielda w takiej postaci została zaproponowana w 1982 roku. Obecnie znacznie lepsze rezultaty w dziedzinie rozpoznawania znaków można osiągnąć używając sieci konwolucyjnych lub głębokich.

# Literatura

1. Tadeusiewicz R., Gąciarz T., Borowik B., Leper B.: *Odkrywanie właściwości sieci neuronowych przy użyciu programów w języku C#,* Polska Akademia Umiejętności, Kraków 2007
2. Stanisław Osowski: *Sieci neuronowe do przetwarzania informacji,* Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 2013.
3. *„Artificial Neural Network – Hopfield Networks”*, https://www.tutorialspoint.com/artificial\_neural\_network/artificial\_neural\_network\_hopfield.htm, [dostęp: 2.01.2018]
4. Griffiths I., Adams M., Liberty J.: *C# Programowanie,* Helion, 2012
5. *„Hopfield Networks”*, http://web.cs.ucla.edu/~rosen/161/notes/hopfield.html, [dostęp: 2.01.2018]
6. *„FuzzyString”*, https://github.com/kdjones/fuzzystring, [dostęp: 5.01.2018]
7. *„Windows Presentation Foundation”*, https://pl.wikipedia.org/wiki/Windows\_Presentation\_Foundation, [dostęp: 6.01.2018]
8. *„Model-View-Viewmodel”*, https://en.wikipedia.org/wiki/Model-view-viewmodel, [dostęp: 5.01.2018]
9. *„WPF Tutorial”*, http://www.wpf-tutorial.com/, [dostęp: 6.01.2018]
10. *„Dependency Properties”*, https://www.wpftutorial.net/DependencyProperties.html, [dostęp: 6.01.2018]
11. *„WPF Tutorial - Tworzenie kontrolek użytkownika”*, http://www.altcontroldelete.pl/artykuly/wpf-tutorial-tworzenie-kontrolek-uzytkownika/, [dostęp: 6.01.2018]
12. *„How to: Implement Property Change Notification”*, https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/how-to-implement-property-change-notification, [dostęp: 6.01.2018]

# Summary

The purpose of this paper, was to create application, able to recognize computer encoded character, basing on its image in .tiff format. The application uses Hopfield network, designed to store and recognize characters patterns. Although the implementation of the network and application design were successful, results are not satisfying. A single Hopfield network, made of 120 neurons is too small to store all letter patterns. In consequence, a flow the of energy function is full of noise from overlapping patterns stored in the network and the output is often false local minimum. Unfortunately the network is not capable of recognizing patterns used in learning process, therefore using it to recognize noisy patterns or handwriting would not be effective. As shown by the test, recognition capabilities grows as the number of training patterns decrease.

Better results could be achieved by applying bias to energy function, in order to more proficiently find the solution. Alternatively network could be dispersed on few smaller networks, where each of them would process only a part of the image and the result would be calculated based on sum of their outputs. On the other hand, Hopfield network in this form was originally presented in 1982. Modern methods, based on convolutional or deep networks, achieves far better results in the field of Optical Character Recognition.