UNIWERSYTET GDAŃSKI
Wydział Matematyki, Fizyki i Informatyki

Wojciech Denejko
nr albumu: 214 300

Rozpoznawanie tekstu w aplikacjach mobilnych

Praca magisterska na kierunku:
INFORMATYKA
Promotor:
dr Tomasz Borzyszkowski

Gdańsk 2017

Streszczenie

Niniejsza praca ma na celu stworzenie aplikacji rozpoznającej litery oraz cyfry pisane odręcznie, która charakteryzuje się kompatybilnością z systemami iOS oraz Android. Do wytworzenia aplikacji zostanie użyte narzędzie Xamarin, które służy do tworzenia aplikacji wieloplatformowych. Zbadane zostaną różne metody połączenia technologii wieloplatformowej z istniejącymi rozwiązaniami OCR. Przedstawiona konwolucyjna sieć neuronowa zaprezentuje klasyfikacje liter alfabetu oraz cyfr oraz zostaną przeprowadzone badania pokazujące przewagę konwolucyjnych sieci neoronowych nad innymi algorytmami służącymi do uczenia maszynowego.

Integralną częścią pracy jest aplikacja ImageResizer, która pozwala na przygotowanie modelu danych z podzielonego na poszczególne znaki oraz cyfry zbioru danych.

Słowa kluczowe

C#, Xamarin, .NET, Uczenie maszynowe, Sieci neuronowe, kNN, Random Forest,

Spis treści

Wprowadzenie 6

1. Rozpoznawanie tekstu w aplikacjach wieloplatformowych 7
- 1.1. Przedstawienie problemu 8
- 1.2. Sposób wytworzenia zbioru treningowego 9
- 1.3. Algorytm k-NN 11
- 1.4. Random Forest 13
- 1.5. Wielowarstwowe sieci neuronowe 16
- 1.6. Konwolucyjne sieci neuronowe - CNN 20

2. Przegląd bibliotek uczenia maszynowego 26
- 2.1. Microsoft Computer Vision API 26
- 2.2. TesseractAPI 27
- 2.3. TensorFlow 28
- 2.4. Scikit-learn 28
- 2.5. EmguCV 29

3. Przeprowadzone badania 31
- 3.1. Xamarin 31
- 3.2. Zbiór testowy 33
- 3.3. Skuteczność klasyfikacji 33
- 3.4. Czas treningu 34
- 3.5. Porównanie modeli danych 34
- 3.6. Rozmiar modelu, a skuteczność terningu 36

4. Podsumowanie i wnioski 37
- 4.1. Zalety aplikacji wieloplatformowych 37
- 4.2. Wady aplikacji wieloplatformowych 38
- 4.3. Uczenie maszynowe w aplikacjach mobilnych 38
- 4.4. Najlepsza metoda rozpoznawania pisma odręcznego 39

4.5. Koszt 39

Zakończenie 40

Bibliografia 41

Oświadczenie 42

Wprowadzenie

Xamarin to platforma deweloperska służąca do tworzenia natywnych aplikacji mobilnych dla systemów iOS, Android oraz Windows, za pomocą wspólnej technologii .NET i języka C# . Dzięki temu możliwe jest uzyskanie do stu procent wspólnego kodu między platformami Android oraz iOS. Aplikacje napisane przy użyciu technologii Xamarin i C# mają pełny dostęp do interfejsów, API oraz możliwość tworzenia natywnych interfejsów użytkownika.

Ze względu na dynamiczny rozwój rynku IT, uczenie maszynowe staje się coraz bardziej popularne a algorytmy zyskują lepszą skuteczność dzięki dostępności danych oraz szybszych podzespołów komputerowych.

Urządzenia przenośne mają stosunkowo ograniczone zasoby w związku z tym istnieje problem wykorzystania technologii uczenia maszynowego z aplikacjami mobilnymi. Algorytmy systemów uczących się wymagają dużej mocy obliczeniowej. Aplikacje wieloplatformowe pozwalają zaoszczędzić czas na implementacji oraz skuteczniej tworzyć funkcjonalności rozpoznawania tekstu. Połączenie tej technologi z algorytmem służącym do klasyfikacji znaków w obrazie jest bardziej optymalne niż ich natywne odpowiedniki.

Celem pracy jest zbadanie istniejących rozwiązań służących do rozpoznawania tekstu oraz stworzenie sieci neuronowej pozwalającej na klasyfikację znaków pisanych charakterystycznych dla współczesnego języka polskiego. Dane pozwalające na przeprowadzenie wymaganego treningu sieci neoronowego zostały udostępnione przez portal NIST.GOV. Zbiór Special Database 19 zawiera podzielone na klasy grupy liter od A do Z oraz cyfry od 0 do 9, następnie zostało swtorzone narzędzie do odczytywania znaków i zapisania ich jako model danych.

ROZDZIAŁ 1

Rozpoznawanie tekstu w aplikacjach wieloplatformowych

OCR (ang. Optical Character Recognition) jest to technika lub część oprogramowania służąca do rozpoznawania znaków oraz całych tekstów zapisanych w pliku graficznym prezentowanym za pomocą pionowo-poziomej siatki odpowiednio kolorowanych pikseli. Przykładem takiej grafiki jest zdjęcie z aparatu cyfrowego.

Niegdyś pojęcie rozpoznawania znaków oznaczało samą klasyfikacje ciągów znaków drukowanych, które są łatwiejszym problemem do rozwiązania, dziś również pisma odręczne oraz cechy formatowania, takie jak krój pisma lub układy tabelaryczne (formularze).

Techniki OCR są głównie wykorzystywane do cyfryzacji zasobów bibliotek, a także jako ułatwienie przy odczytywaniu dokumentacji napisanych pismem odręcznym, w aplikacjach mobilnych rozpoznawanie znaków pomaga w takich zadaniach jak tworzenie notatek, a następnie tłumaczenie ich na tekst drukowany. Niestety, w obu przypadkach istniejące rozwiązania OCR nie są tak skuteczne jak człowiek, zatem w przypadkach trudności z klasyfikacją znaku lub fragmentu tekstu niezbędna jest weryfikacja wyniku przez człowieka celem uniknięcia błędu.

Postęp w metodach OCR jest bardzo widoczny gdyż w obecnych czasach produkty potrafią rozpoznawać mało dokładne skany.

Wykonane telefonami komórkowymi z szumami na obrazkach, z tekstem napisanym pod nienaturalnymi kątami w wielu językach, pozostaje jednak problem rozpoznawania znaków pisma odręcznego.

Rozpoznawanie pisma jest możliwe dzięki zastosowaniu metod z dziedziny rozpoznawania wzorców, czyli pola badawczego w obrębie uczenia maszynowego. Metoda ta może być definiowana jako działanie polegające na pobieraniu danych i podejmowaniu dalszych czynności zależnych od kategorii, do której należą te dane. By odpowiednio wyodrębnić poszczególne znaki z obrazu używane są

biblioteki pozwalające na profesjonalną obróbkę zdjęć pod zastosowania w celach uczenia maszynowego. Przykładem takiej biblioteki jest OpenCV. Po wyodrębnieniu potrzebnych informacji na temat danego znaku obrazy są klasyfikowane jako poszczególne litery. Zwykle w tym procesie używane są sieci neuronowe.

Kompletny system rozpoznawania wzorców składa się:

- Zbioru danych, które oferują możliwość klasyfikacji lub opisu
- Mechanizmu wydobywania cech, które najlepiej charakteryzują i separują daną klasę, do której dany element zbioru danych należy
- Mechanizmu przekształcenia elementu zbioru w symboliczną informację, łatwiejszą do wykorzystania przez algorytm
- Schematu decyzyjnego lub schematu opisu, który realizuje właściwą część procesu klasyfikacji w oparciu o wydobyte i przekształcone cechy obiektu.

1.1. Przedstawienie problemu

Wsród istniejących rozwiązań mogących służyć jako narzędzie potrzebne do wytworzenia aplikacji mobilnej, która rozpozna polskie znaki pisma odręcznego nie istnieje łatwy sposób zastosowania rozwiązania pozwalającego na skuteczną klasyfikacje polskiego pisma. Brakuje również dostępnych danych wymaganych do skutecznej klasyfikacji w oparciu o przekształcone informacje. Aby rozwiązać ten problem należy stworzyć zbiór treningowy lub rozszerzenie istniejącego zbioru danych o polskie znaki alfabetu.

Dostępne biblioteki na rynku, takie jak TesseractAPI oraz Microsoft Computer Vision API oferują wysoką skuteczność w rozpoznawaniu polskich oraz angielskich obrazów tekstu drukowanego lecz zarazem brak możliwości rozpoznawania pisma odręcznego. Wymagane jest więc stworzenie systemu rozpoznawania wzorców, który pozwalałby na skuteczną klasyfikacje znaków pisma odręcznego.

Kolejnym problemem są znacząco ograniczone zasoby urządzeń mobilnych. Systemy rozpoznawania wzorców wymagają mocy obliczeniowej potrzebnej do przekształcenia obrazów w postać pozwalającą na wyodrębnianie cech, a następnie

przeprowadzenie procesu klasyfikacji. Rozwiązaniem tego problemu jest wykorzystanie systemu rozpoznawania wzorców jako serwisu internetowego działającego w oparciu o architekturę REST.

1.2. Sposób wytworzenia zbioru treningowego

Zbiór treningowy jest kontenerem krotek (przykładów, obserwacji, próbek), będących lista właściwości atrybutów opisowych (tzw. deskryptorów) i wybranego atrybutu decyzyjnego (ang. class label attribute). Głównym jego celem jest zbudowanie formalnego modelu zwanego klasyfikatorem. Wynikiem procesu klasyfikacji jest pewien otrzymany model (klasyfikator), który przydziela każdemu przykładowi wartość atrybutu decyzyjnego w oparciu o właściwości pozostałych atrybutów.

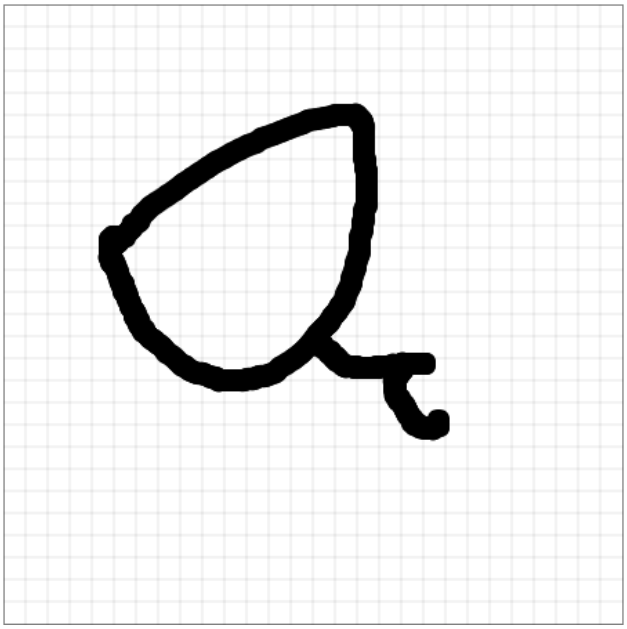
W przypadku systemu rozpoznawania wzorów zbiorem treningowym są zdjęcia obrazów zawierajace odpowiednio wszystkie litery polskiego alfabetu oraz cyfry. Wszystkie zdjęcia liter, które istnieją w zbiorze należy przeformatować do postaci najlepiej rozumianą przez wykorzystywane algorytmy.

Do transformacji zdjęć zastosowano EmguCV, jest to wieloplatformowa implementacja (ang. wrapper) w technologi .NET biblioteki OpenCV, pozwalająca na wykorzystanie funkcjonalności OpenCV w środowisku .NET we wszystkich jego językach programowania takich jak C#, VB, F#. Można ją zainstalować używając menadżera pakietów Nuget w programie Visual Sutdio, Xamarin Studio lub Unity, a więc jest również kompatybilna z platformami mobilnymi Android oraz iOS.

Transformacja zdjęcia przebiega następująco:

- Odczytaj zdjęcie w formacie .png
- Przeprowadź konwersje kolorów RGB na odcienie szarości
- Przetwórz obraz do formatu 28 x 28 pikseli
- Odczytaj stopień jasności każdego piksela w skali od 0 do 255 i zapisz je w tablicy

Rezultatem działania programu do konwersji zdjęć jest plik train.csv. Zawiera ona 785 kolumn. Pierwsza kolumna, nazwana ”label”, określa znak, który jest narysowany. Reszta kolumn zawiera informacje na temat jasności każdego piksela.



Rysunek 1.1. Przykład zdjęcia znaku

Każda kolumna w zbiorze treningowym ma ustawioną nazwę pixelx, gdzie x jest liczbą między 0 a 783. By znaleźć dany piksel na obrazie, należy rozłożyć x jako x = a * 28 + b, gdzie a i b to liczby między 0 a 27. Wtedy pixelx jest umieszczony w a-tym rzędzie b-tej kolumnie w macierzy 28 x 28, indeksowanej od zera. Na przykład, pixel31 wskazuje na to, piksel w czwartej kolumnie od lewej i drugim wierszu od góry. Tak jak pokazane na diagramie poniżej:

```
000  001  002  003  ...  026   027
028  029  030  031  ...  054   055
 |    |    |    |   ...   |     |
728  729  730  731  ...  754   755
956  957  958  959  ...  1022  1023
```

Fragment aplikacji pozwalającej na stworzenie modelu danych załączony poniżej:

```
var csv = new StringBuilder();
var csv2 = new StringBuilder();
```

```
var fileName = files[i].DirectoryName
                .Replace(imagesPath + @"\", string.Empty);
csv.Append(fileName);
csv.Append(',');
var originalImage = new Image<Gray, byte>(files[i].FullName)
                    .Not();
var img = originalImage.Resize(28, 28, Inter.Linear);
for (var k = 0; k < img.Height; k++)
{
    for (var j = 0; j < img.Width; j++)
    {
        csv.Append(img[k, j].Intensity);
        csv.Append(',');
    }
}

csv2.AppendLine(csv.ToString());
File.AppendAllText(csvFilePath, csv2.ToString());
```

Pełna wersja programu znajduje się w dołączonej dokumentacji w solucji pod nazwą ImageResizer

1.3. Algorytm k-NN

Algorytm k-najbliższych sąsiadów (ang. k nearest neighbours) - algorytm regresji nieparametrycznej najczęściej używany w statystyce do prognozowania pewnej wartości zmiennej losowe.

Założenia:

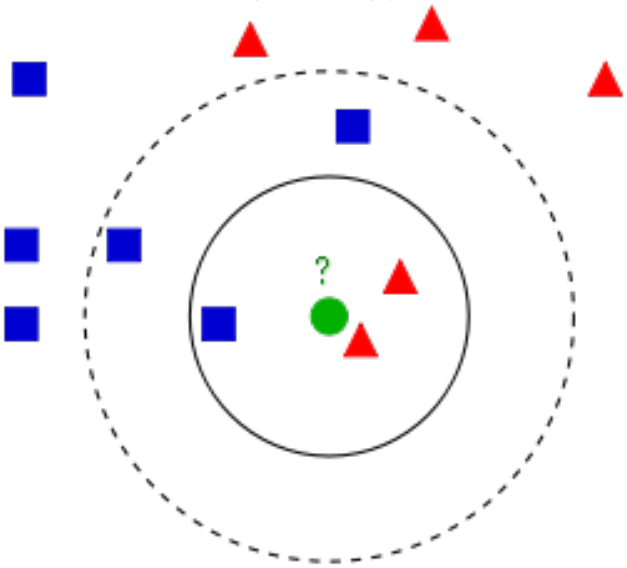
- Dany jest zbiór teningowy, który stworzony został w oparciu o narzędzie TraningSetGenerator.
- Dana jest obserwacja C, zawierająca wektor zmiennych pixelo ... pixel1024, dla której chcemy prognozować wartość zmiennej objaśnianej label.

Ilustracja przedstawiająca przykład działania algorytmu k najbliższych sąsiadów:

Algorytm działa następująco:

- Porównaj wartości zmiennych objaśniających dla obserwacji C, z każdym wektorem w zbiorze treningowy.
- Wybierz k (ustalonej z góry liczby) najbliższych do C obserwacji ze zbioru treningowego.
- Uśrednieniu wartości zmiennej objaśnianej dla wybranych obserwacji, w wyniku czego uzyskujemy prognozę.

Powyższy przykład demonstrukcje problem klasyfikacji przy użyciu algorytmu kNN, obiekt zaznaczony kołem dla k najbliższych sąsiadów równe 2, zostanie sklasyfikowany jako trójkąt, natomiast jeżeli do klasyfikacji zostanie użytych 3 najbliższych sąsiadów obiekt zostanie sklasyfikowany jako kwadrat.



Rysunek 1.2. Przykład problemu k-NN

Dla k = 3, niewiadoma oznaczona zielonym punktem będzie sklasyfikowana jako czerwony trójkąt w oparciu o trzech najbliższych sąsiadów, jednak jeśli k = 5, zostałaby sklasyfikowana jako niebieski kwadrat ponieważ algorytm działałby w oparciu o pięciu sąsiadów. Najbliżsi sąsiedzi są określani przy pomocy metryki euklidesowej określonej wzorem:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_i - q_i)^2 + ... + (p_n - q_n)^2}.$$

Fragment przykładowej implementacji powyższego algorytmu, który klasyfikuje przykładowy znak alfabetu dla k = 3 znajduje się poniżej:

```
labeled_images = pd.read_csv('output.csv')
images = labeled_images.iloc[0:5000,1:]
labels = labeled_images.iloc[0:5000,:1]
train_images, test_images, train_labels, test_labels = train_test_split(
knn = KNeighborsClassifier(n_neighbors=3, algorithm="kd_tree")
knn.fit(train_images, train_labels.values.ravel())
print knn.score(test_images,test_labels)
```

Pełna implementacja algorytmu k najbliższych sąsiadów jest dołączona w dokumentacji.

1.4. Random Forest

Drzewa decyzyjne to graficzna metoda wspomagania procesu decyzyjnego. Algorytm drzew decyzyjnych jest również stosowany w uczeniu maszynowym do pozyskiwania wiedzy na podstawie przykładów.

Koncepcja Bugging polega na budowie ekspertów dla podzbioru zadań. W tym przypadku, ze wszystkich problemów do rozwiązania losowany jest jeden ze zwracaniem podzbioru problemów, a następnie dla tego podzbioru szukany jest ekspert. W algorytmie tym z całego zbioru danych uczących losowany jest podzbiór - przez losowanie ze zwracaniem - i dla tego podzbioru budowany jest model predykcyjny. Następnie po raz kolejny ze zwracaniem losowany jest inny podzbiór wektorów i dla niego budowany jest kolejny model. Całość powtarzana jest k krotnie a na koniec wszystkie zbudowane modele użyte są do głosowania.

Algorytm Random Forest to metoda klasyfikacji polegająca na połączeniu drzew decyzyjnych oraz koncepcji bugging, tworzy ona wiele drzew decyzyjnych na podstawie losowego zestawu danych. Ideą tego algorytmu polega na zbudowaniu rady ekspertów z losowych drzew decyzyjnych, gdzie w odróżnieniu od klasycznych drzew decyzji, losowe drzewa budowane są na zasadzie, iż zbiór analizowanych cech w węźle dobierany jest losowo. Ponadto, poszczególne drzewa z losowych lasów drzew budowane są zgodnie z koncepcją Bugging.

Cechy algorytmu Random Forest:

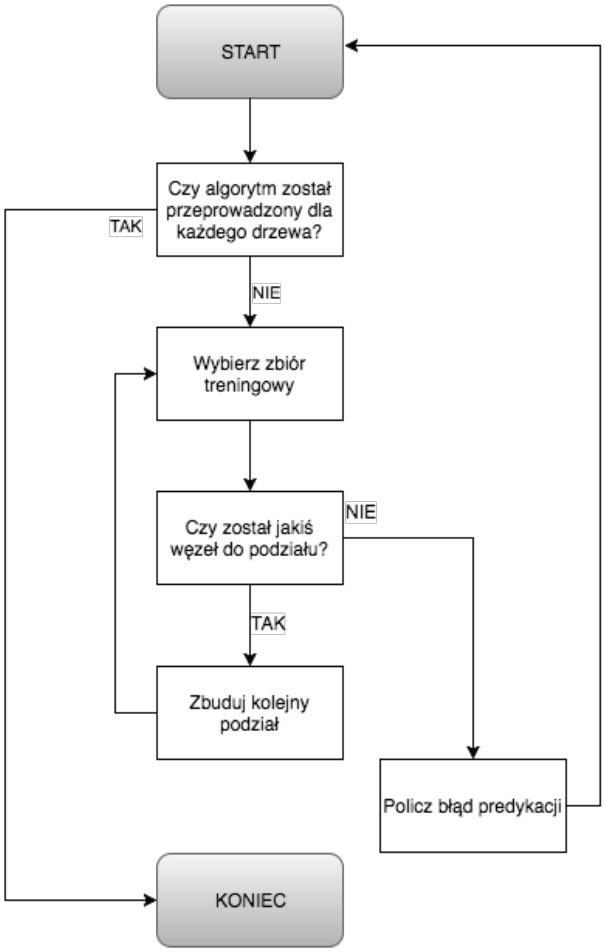
- Działa skutecznie na dużych zbiorach treningowych
- Utrzymuje dokładność w przypadku braku danych
- Daje oszacowanie, które zmienne są istotne w klasyfikacji
- Lasy drzew mogą być zapisane i wykorzystane w przyszłości dla innego zbioru danych
- Nie wymaga wiedzy eksperckiej

Algorytm działa następująco:

- Losujemy ze zwracaniem z n-elementowego zbioru treningowego n wektorów. Na podstawie takiej próby zostanie stworzone drzewo.
- W każdym węźle podział odbywa się poprzez wylosowanie bez zwracania m spośród p atrybutów, następnie w kolejnym węźle k spośród m atrybutów
- Proces budowania drzewa bez przycinania trwa, jeśli to możliwe do momentu uzyskania w liściach elementów z tylko jednej klasy.

Proces klasyfikacji:

- Dany wektor obserwacji jest klasyfikowany przez wszystkie drzewa, ostatecznie zaklasyfikowany do klasy, w której wystąpił najczęściej.
- W przypadku elementów niewylosowanych z oryginalnej podpróby, każdy taki i-ty element zostaje poddany klasyfikacji przez drzewa, w których budowie nie brał udziału. Taki element zostaje następnie przyporządkowany klasie, która osiągana była najczęściej.



Rysunek 1.3. Diagram przepływu algorytmu Random Forest

Fragment przykładowej implementacji powyższego algorytmu, który klasyfikuje przykładowy znak alfabetu znajduje się poniżej:

```
labeled_images = pd.read_csv('output.csv')
images = labeled_images.iloc[0:5000,1:]
labels = labeled_images.iloc[0:5000,:1]
train_images, test_images, train_labels, test_labels = train_test_split(
clf = RandomForestClassifier(n_jobs=2)
clf.fit(train_images, train_labels.values.ravel())
print clf.score(test_images,test_labels)
```

Pełna implementacja algorytmu random forrest jest dołączona w dokumentacji.

1.5. Wielowarstwowe sieci neuronowe

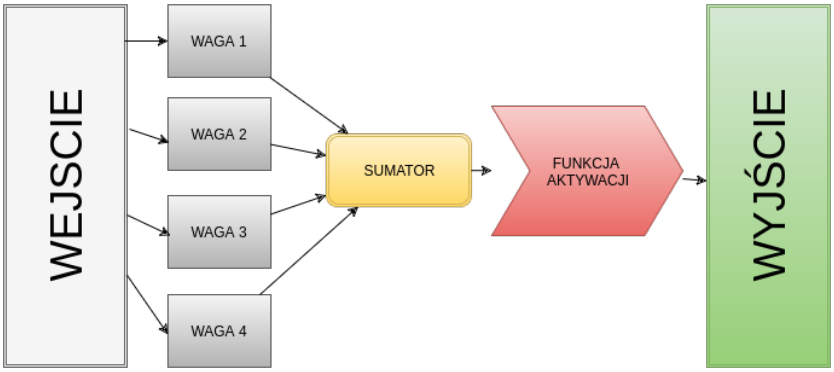
Siecią neuronową nazywa się programową lub sprzętową strukturę modeli, realizującą obliczenia lub przetwarzającą sygnały poprzez rzędy elementów, zwanych sztucznymi neuronami. Emulują one niektóre spośród zaobserwowanych właściwości biologicznych układów nerwowych. Sztuczne sieci neuronowe są swoistym systemem inspirowanym przez to, w jaki sposób gęsto połączone między sobą struktury mózgu, odbierają i przetwarzają dane które docierają w różny sposób z otoczenia. Kluczowym elementem jest zatem struktura systemu przetwarzania informacji. Sieć taka składa się z dużej liczby rozlegle połączonych ze sobą elementów przetwarzających, które są powiązane ze sobą ważonymi połączeniami.

Cechą charakterystyczna sieci neuronowych od algorytmów realizujących przetwarzanie informacji przy użyciu algorytmów jest umiejetność generalizacji, czyli zdolność uogólniania wiedzy dla nieznanych wcześniej wzorców. Innym atutem jest także zdolność do aproksymacji wartości funkcji wielu zmiennych w przeciwieństwie do interpolacji, która jest możliwa do uzyskania używając przetwarzania algorytmicznego.

Uczenie sieci neuronowych zmienia liczbowe wartości wag znajdujących się pomiędzy neuronami. Następuje to poprzez bezpośrednią ekspozycje rzeczywistego zestawu danych, gdzie algorytm uczący modeluje wagi polaczeń. W dziecinie nauk technicznych sieci neuronowe wykorzystuje się do rozwiązywania między innymi problemów:

- Aproksymacji, prognozowania
- Klasyfikacji i rozpoznawania
- Kojarzenia danych, sieci neuronowe pozwalają zautomatyzować procesy wnioskowania i pomagają wykrywać istotne powiązania między danymi
- Analizy danych czyli poszukiwania związków między danymi

Podstawowym elementem sieci neuronowej jest neuron. Jego schemat został opracowany przez McCullocha i Pittsa w roku 1943, został on oparty na budowie biologicznej komórki nerwowej.



Rysunek 1.4. Schemat sztucznego neuronu

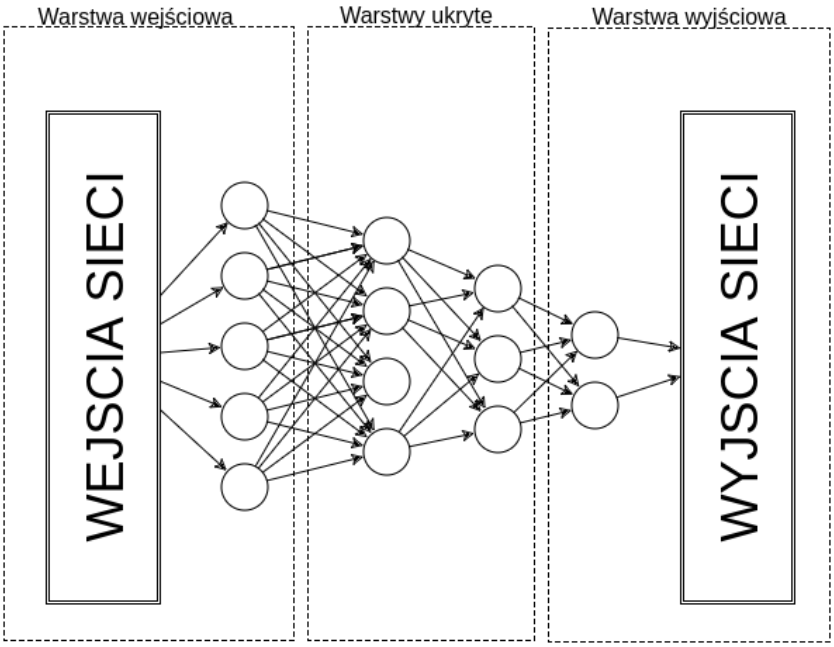
Do wejść doprowadzane są sygnały z wejść sieci lub neuronów warstwy poprzedniej. Każdy sygnał mnożony jest przez odpowiadającą mu wartość liczbowa zwaną wagą. Wpływa ona na percepcje danego sygnału wejściowego i jego udział w sygnale wyjściowym przez neuron. Waga może być dodania lub ujemna, jeżeli nie ma połączenia miedzy neuronami to waga jest równa zero. Zsumowane iloczyny wag i sygnałów są argumentem funkcji zwanej funkcją aktywacji neuronu.

Wartość funkcji aktywacji jest wyjściem neuronu i propagowana jest do neuronów warstwy następnej. Może ona przybierać jedną z trzech postaci:

- nieliniowa
- liniowa
- skoku jednostkowego

Należy zauważyć, iż jest to podział bardziej formalny niż merytoryczny. Różnice funkcjonalne między tymi typami raczej nie występują, natomiast można stosować je naprzemiennie w różnych warstwach sieci.

Najbardziej popularnym typem sieci neuronowej jest sieć wielowarstwowa (ang. Multi-Layer Neural Network). Jej cecha charakterystyczna jest występowanie co najmniej jednej warstwy ukrytej neuronów, pośredniczącej w przekazywaniu sygnałów pomiędzy wejściami a wyjściami sieci.



Rysunek 1.5. Schemat budowy sieci wielowarstwowej

Do rozpoznania polskich znaków pisma odręcznego użyta została sieć posiadająca trzy warstwy:

- Warstwa wejściowa sieci składa się z neuronów zawierających informacje na temat każdego piksela. Zbiór treningowy składa się z obrazów 32 x 32 pikseli. Zgodnie z tym założeniem pierwsza warstwa sieci składa się z 1024 neuronów. Każdy z nich przechowuje wartość skali szarości piksela, gdzie 0.0 oznacza kolor biały, a 1.0 czarny.
- Druga warstwa zawiera n neuronów, liczba n jest używana w kontekscie eksperymentalnym.
- Ostatnia warstwa, zawiera 36 neurony, ponieważ w alfabet składa się z 26 liter, rozpatrywane są zarówno litery oraz cyfry. Implementacja sieci:

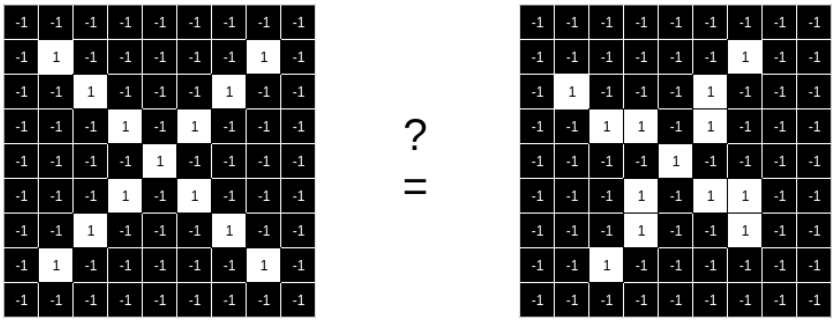
```
xo = x[:split]; x1 = x[split:]
yo = y[:split]; y1 = y[split:]
```

```
mlp = MLPClassifier(solver='sgd', activation='relu',
                    hidden_layer_sizes=(100,30),
                    learning_rate_init=0.3, learning_rate='adaptiv
                    momentum=0.9, nesterovs_momentum=True,
                    tol=1e-4, max_iter=200,
                    shuffle=True, batch_size=300,
                    early_stopping = False, validation_fraction =
                    verbose=True)
mlp.fit(xo, yo)
```

Kompletny skrypt implementujący sieć, dostępny jest w dokumentacji.

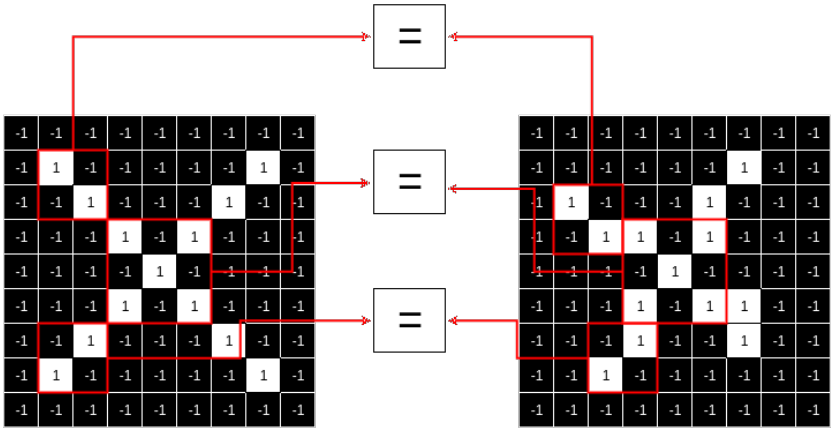
1.6. Konwolucyjne sieci neuronowe - CNN

Konwolucyjne sieci neuronowe (ang. Convolutional Neural Networks) są podobne do klasycznych sieci neuronowych. Aby dokładnie przeanalizować budowę oraz działanie CNN przedstawiony zostanie problem klasyfikacji dwóch liter X i O. Ten przykład demonstruje charakterystyczne reguły konwolucji.



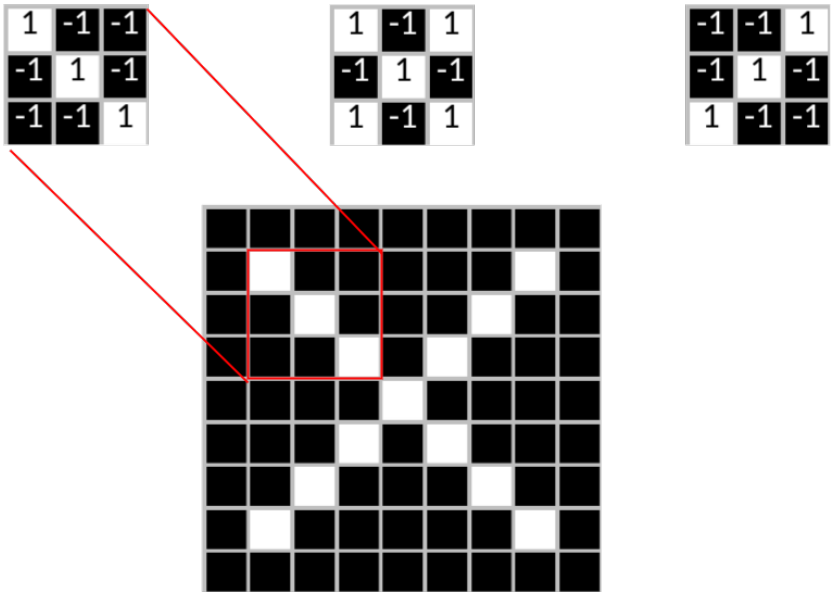
Rysunek 1.6. Problem klasyfikacji

CNN porównuje obrazy w kawałkach. Każda część nazywana jest cecha (ang. feature). Następnie zdjęcia przeszukiwane są na podobnych pozycjach by uzyskać maksymalną liczbę cech wspólnych. Sieci konwolucyjne wykorzystują podobieństwa na obrazach.



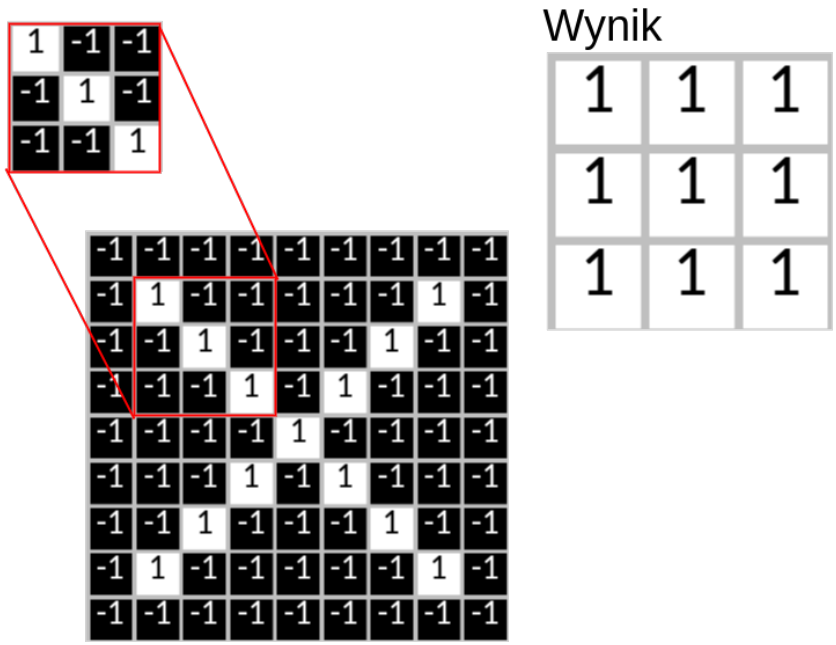
Rysunek 1.7. Wybrane cechy i ich odpowiedniki w zdjęciu do klasyfikacji

Każdą cechę można scharakteryzować jako mniejsze zdjęcie lub dwuwymiarową tablice wartości. W przypadku litery X, cechami będą ukośne linie i znak krzyża.



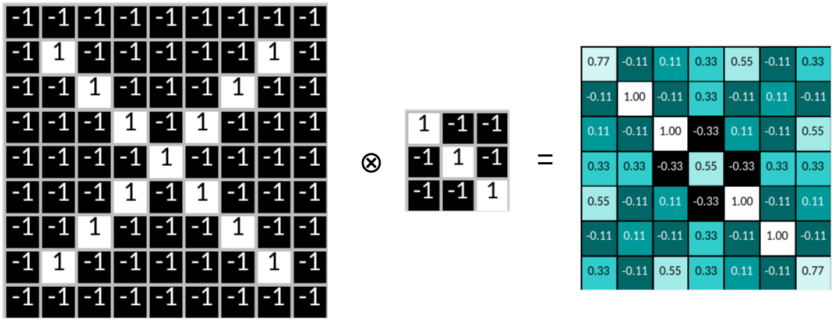
Rysunek 1.8. Wartości liczbowe pikseli różnych cech

Kiedy rozpatrywany jest nowy obraz, sieć poszukuje cech zdjęcia w każdej możliwej pozycji. Obliczając wartości pasujące do cech tworzymy filtr. Działania matematyczne kryjące się za tym działaniem nazywane są splotem całkowitym. Aby obliczyć parę cech obrazu, należy pomnożyć każdą wartość piksela cechy przez odpowiadający mu piksel danego zdjęcia. Następnie należy dodać wszystkie wyniki z poprzednich operacji i podzielić przez łączną liczbę pikseli cechy. Jeśli oba piksele były białe, a ich wartość była reprezentowana przez 1, wynik wynosi 1 - piksel jest biały, w przeciwnym wypadku piksel jest czarny a jego wartość jest równa -1. Jeżeli wszystkie wartości cechy są takie same, wynikiem dodawania i podzielenia przez liczbę pikseli będzie 1.

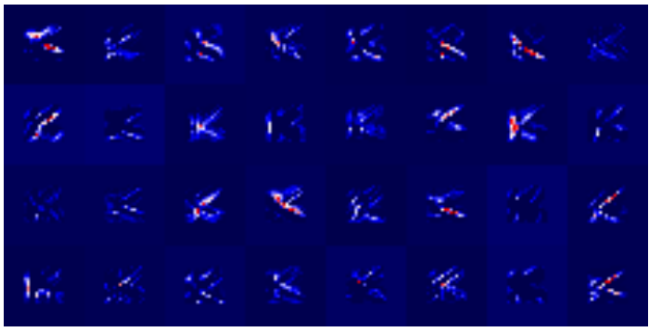


Rysunek 1.9. Wynik przykładowej operacji splotu

Następnym krokiem jest powtórzenie operacji splotu na kompletnym obrazie używając wszystkich dostępnych cech. Wynikiem jest szereg obrazów z filtrami, na każdym z dostępnych wyników nałożony jest jeden filtr. W konwolucyjnych sieciach neuronowych, warstwa ta nazywana jest warstwą konwolucyjną bądź też splotową (ang. convolution layer).

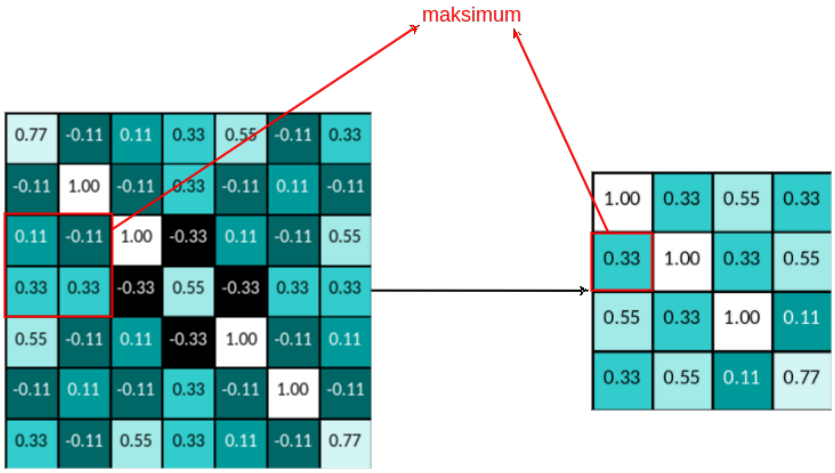


Rysunek 1.10. Wynik operacji splotu danej cechy



Rysunek 1.11. Przykład warstwy konwolucyjnej dla litery k

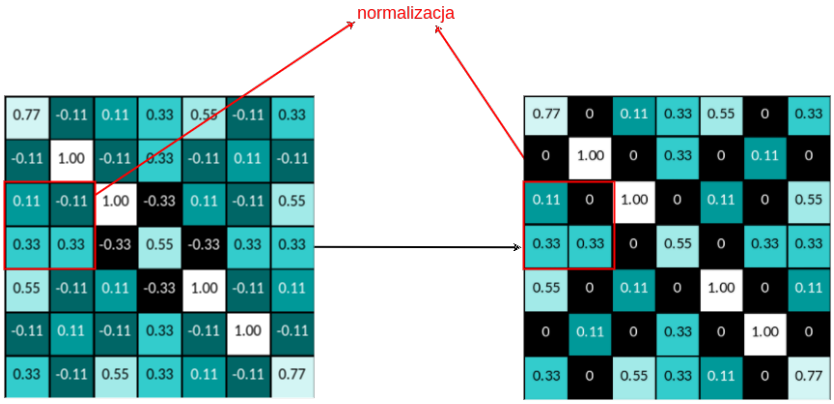
Kolejnym narzędziem oferowanym przez CNN jest warstwa sumowania (ang. pooling layer). Pooling daje możliwość zmniejszenia dużych obrazów bez utraty ważnych informacji na ich temat. Operacje wykonywane w tej warstwie to podział zdjęcia z poprzedniej warstwy na mniejsze, a następnie pobranie maksimum z danej części. Operacje te należy powtórzyć na całym zdjęciu.



Rysunek 1.12. Przykład sumowania

Po sumowaniu, wielkość danego zdjęcia jest cztery razy mniejsza. Ponieważ przetrzymywane są wartości maksymalne z każdego okna, przechowywane zostają najlepsze pasujące wartości danej cechy w oknie. To oznacza, że nie ważne jest gdzie dana cecha pasuje do momentu, aż znajdziemy pasującą odpowiedź w oknie.

Dodatkowym narzędziem jest warstwa normalizacyjna (ang. Rectified Linear Units Layer, ReLU), użycie tej funkcji pozwala ustawić wszystkie ujemne wartości liczbowe na zero. Jeśli wartość jest większa od zero, funkcja ReLU zwróci jeden. To ważny proces w którym wszystkie ujemne wartości zostają zastąpione zerami. Wspomaga to funkcjonowaniu sieci trzymając wszystkie wartości na których wykonywane są operacje by te były równe o lub dodatnie.



Rysunek 1.13. Przykład normalizacji danych

Ostatnim narzędziem dostępnym do wykorzystania jest warstwa FCL (ang. Fully Connected Layer). Przyjmuje ona przefiltrowane obrazy, a następnie tłumaczy je na wartości z przedziału od 0 do 1. W przypadku klasyfikacji znaków pisma odręcznego należy zdecydować do której kategorii należy znak X czy O. Kiedy nowy obraz zostaje przekazany do klasyfikacji, wykonywane są operacje na wszystkich wymienionych warstwach, a ostatecznie FCL decyduję ,do której grupy należy znak.

ROZDZIAŁ 2

Przegląd bibliotek uczenia maszynowego

Aby dobrze zobrazować problem rozpoznawania pisma odręcznego w aplikacjach mobilnych, wykorzystanych zostało kilka technologii oraz bibliotek dostępnych w językach programowania Python oraz C#. Jednocześnie wybrane przykłady demonstrują gotowe implementacje algorytmów, które pozwalają wykorzystywać uczenie maszynowe do własnych potrzeb bez koniecznej wiedzy z zakresu uczenia maszynowego, takimi technologiami są Microsoft Computer Vision oraz TesseractAPI. Drugim przykładem są biblioteki dostępne do użycia dla programistów uczenia maszynowego, pozwalające na precyzyjne skonfigurowanie dostępnych algorytmów do własnych potrzeb, są to biblioteki Scikit-learn oraz Tensorflow. Dodatkowo, wykorzystana została biblioteka EmguCV pozwalająca na wykrywanie znaków na zdjęciach oraz odpowiednie formatowanie go na potrzeby stworzenia modelu danych do treningu.

2.1. Microsoft Computer Vision API

Usługi Microsoft Cognitive Services umożliwiają kompilowanie aplikacji z zaawansowanymi algorytmami przy użyciu zaledwie kilku wierszy kodu. Działają one na różnych urządzeniach i platformach, takich jak systemy iOS, Android i Windows, wciąż są ulepszane i łatwo je skonfigurować. Usług uruchomionych w ramach tego API jest dokładnie 22. Począwszy od lokalizowania twarzy na zdjęciach, rozpoznawania ich emocji, skończywszy na usłudze wykrywania tekstu na fotografiach, konwertowania mowy na tekst lub autosugestii wprowadzanego tekstu prosto z wyszukiwarki Bing.

Computer Vision API, jest system bazowanym na rozwiązaniach Azure, dostarcza deweloperom dostęp do algorytmów pozwalających na przetwarzanie obrazów i zwracanie informacji na ich temat. Poprzez wysłanie obrazu do serwera

algorytmy umieszczone w chmurze firmy Microsoft są w stanie zanalizować kontekst wizualny w wielu różnych sposobach na podstawie wyborów użytkowników.

Między innymi technologia zawiera narzędzia do rozpoznawania znaków oraz całych tekstów. Działa ona w 21 językach, dostępny jest również język polski. Usługa posiada możliwość dostosowania kątu nachylenia tekstu, horyzontalnego obrotu osią by rozpoznania dawały lepszy efekt.

Skuteczność rozpoznanego tekstu zależy od jakości zdjęcia. Nie dokładne rozpoznania mogą być spowodowane dostarczeniem zdjęcia które:

- Jest rozmazane
- Zawiera odręczne pismo
- Czcionka tekstu jest rzadko spotykana
- Posiada złożone tła, cienie, odbicia na tekscie

Od marca 2017r. technologia ta pozwala na rozpoznawanie ręcznie pisanego tekstu w języku angielskim, dostępna jest ona w wersji demonstracyjnej i rozpoznaje jedynie zdjęcia w języku angielskim.

2.2. TesseractAPI

Tesseract jest open source'owym silnikiem służącym do rozpoznawania znaków, dostępny jest on na githubie[1] . Może być on używany bezpośrednio lub używając API by wyodrębnić napisane odręcznie bądź też drukowane znaki ze zdjęć. Obsługuje on ponad 20 języków w tym też Polski.

Zaletą dostępnego API jest prosta obsługa, brak konieczności posiadania wiedzy z zakresu modelowania algorytmów do uczenia maszynowego oraz gotowe modele języka, które dostępne są do pobrania na stronie projektu.

Wadami tego systemu jest brak możliwości ingerencji w model danych którymi dysponuje API, nie ma możliwości modyfikacji algorytmów, które wykorzystywane są do rozpoznawania znaków. Skutkuje to niską skutecznością rozpoznania znaków pisma odręcznego.

2.3. TensorFlow

Najważniejsze komponenty TensorFlow, stworzonej przez Google biblioteki sztucznej inteligencji, która głównie wykorzystywana jest do rozpoznawania obrazów i mowy, maszynowych tłumaczeń oraz dostarczania lepszych wyników wyszukiwania w danych, technolgoia ta jest otwarta i udostępniona na wolnej licencji Apache 2.0.

Działanie TensorFlow przypomina sposób wykonywania obliczeń za pomocą grafu skierowanego, w którym wierzchołki to matematyczne operacje lub punkty, w których dochodzi do wymiany danych, zaś krawędzie opisują relacje wejścia i wyjścia pomiędzy węzłami, przedstawione w formie dynamicznych wielowymiarowych macierzy danych, czyli tensorów.

ZDJĘCIE!

Biblioteka oferowana przez firmę Google pozwala na obliczenia numeryczne z wykorzystywaniem takich grafów. Można tu korzystać z ogromnych klastrów superkomputerowych, stacji roboczych z wieloma procesorami graficznymi, a nawet wielordzeniowych mobilnych procesorów w smartfonach.

W badaniach nad problemem rozpoznawania znaków, biblioteka TensorFlow została użyta by stworzyć Splotową Sieć Neuronową, ponieważ sposób wykonywania operacji na macierzach, elastyczność oraz bogata dokumentacja pozwalają na proste zamodelowanie sieci, która przy dużej skuteczności potrafi rozpoznać litery i cyfry pisane odręcznie.

2.4. Scikit-learn

Scikit-learn is biblioteką dostępną w języku Python, służy ona do uczenia maszynowego, oferuje ona szereg algorytmów uczenia szynowego. Biblioteka zbudowana jest na podstawie SciPy, który jest wymaganym elementem do instalacji i używania scikit-learn. W stos SciPy wchodzą biblioteki:

- NumPy - dostarcza on n-wymiarowe tablice
- SciPy - służy do wykonywania obliczeń naukowych

- Matplotlib - narzędzie do rysowania grafów 2D/3D
- Pandas - struktury oraz analiza danych

Celem autorów biblioteki było dostarczenie solidnego i dobrze wspomaganego przez społeczność zestawu narzędzi, które działałyby na systemach produkcyjnych. To oznacza, prostota, jakość kodu, dokumentacja oraz wydajność biblioteki jest bardzo wysoka.

Scikit-learn oferuje dużą liczbe klasyfikatorów takich jak:

- Algorytm k najbliższych sąsiadów
- Drzewa decyzyjne
- Random forrest
- Regresja liniowa
- Sieci neuronowe

Na ten moment, biblioteka nie oferuje wsparcia dla własnych modeli sieci neuronowych oraz wykorzystywania klastrów procesorów graficznych.

2.5. EmguCV

Emgu CV jest wieloplatformową implementacją biblioteki OpenCV służącą do przetwarzania zdjęć. Pozwala ona na wywoływanie funkcji OpenCV z poziomu .NET i językami, które są wspierane takie jak C#, VC++, IronPython czy F#. Implementacja ta może być skompilowana z poziomu Visual Studio, Xamarin Studio oraz Unity. Działa na platformach Windows, Linux, Mac OS X, iOS, Android oraz Windows Phone.

Zaletami użycia EmguCV jest przedewszystkim możliwość implementacji na wielu platformach używając jednej technologi, gdyż EmguCV jest napisna w języku C#. To oznacza że może być kompilowana za pomocą Mono, czyli kompilatora C# dostępnego pod platformami Linux oraz Mac OS X. Dzięki temu, przy użyciu platformy Xamarin oraz EmguCV, wszystkie operacje związane z obróbką zdjęcia

i przygotowaniem go do klasyfikacji, mogą zostać wykonane z poziomu aplikacji, jest to ważny aspekt, gdyż w wypadku ograniczonych zasobów, koszt przygotowania zdjęcia do klasyfikacji jest duży.

ROZDZIAŁ 3

Przeprowadzone badania

Aby przeprowadzić badania wskazujące czy przedstawiona propozycja konwolucyjnej sieci neuronowej autora rezultuje lepszymi wynikami rozpoznania od innych, istniejących rozwiązań, stworzona została wieloplatformowa aplikacja mobilna, która po narysowaniu dowolnego znaku lub cyfry podejmuje próbę rozpoznania. Porównane zostaną implementacje Microsoft Computer Vision API, TesseractAPI oraz Webservice'u stworzonego przez autora, który rozpoznanie określa na podstawie modelu sieci CNN.

Podjęta zostanie również próba porównania treningu modeli o różnym rozmiarze jednej cechy, punktem wejściowym do pomiaru będzie pełny model danych, ok. 1.5mln zdjęć cyfr i liter, przedstawione zostaną warianty tego modelu posiadające inna liczbe elementów, bądź inny rozmiar zdjęć.

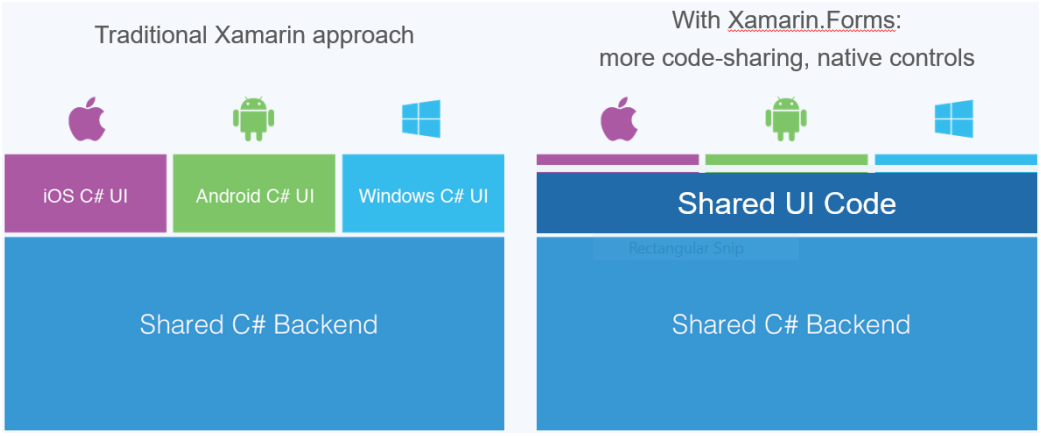
3.1. Xamarin

Xamarin pozwala na tworzenie natywnych aplikacji Android/iOS z wykorzystaniem języka C# i platformy .NET. Same aplikacje tworzymy w Visual Studio lub Xamarin Studio, a możemy do tego podejść na dwa sposoby: Xamarin.Native lub Xamarin.Forms (więcej o tych dwóch podejściach będzie w dalszej części posta). Xamarin od zeszłego roku jest dostępny dla wszystkich zainteresowanych, dzięki jego przejęciu przez Microsoft. Natomiast, sam kod źródłowy Xamarina został udostępniony na GitHub.

Kompilator Xamarin.iOS kompiluje AOT(Ahead of time) aplikacje do natywnego kodu dla ARM, który jest uruchamiany bezpośrednio na iPhone'ach. Natomiast kompilator Xamarin.Android kompiluje do języka IL (Intermediate Language), następnie by uruchomić aplikację, korzysta z JIT (Just in time). Dzięki tym

zabiegom, Xamarin jest transparentny dla docelowych systemów mobilnych, tworząc natywne, w pełni funkcjonalne aplikacje.

W Xamarin.Forms można pisać kod na dwa sposoby, w tzw. code behind w C# lub za pomocą XAML. Drugi sposób dla osób z doświadczeniem w WPF jest bardzo wygodnym rozwiązaniem. Xamarin.Forms tzw. warpery na natywne kontrolki. Oznacza to, że np. odwołując się do kontrolki Label w Formsa-ch odwołujemy się do ich natywnych odpowiedników w poszczególnych platformach pisząc tylko jeden kod. Ponadto kontrolki Xamarin.Forms są w pełni edytowalne i konfigurowalne tzn. w przypadku gdy czegoś brakuje, można poszukać gotowego rozwiązania lub stworzyć własne.



Rysunek 3.1. architektura platformy Xamarin

3.2. Zbiór testowy

W uczeniu maszynowym istnieje wiele metod testowania skuteczności klasyfikatorów. W pracy zawarto metodę losowego podziału na zbiory treningowe i testowe. Oznacza to, iż zbiór danych jest dzielony na dwie części. Uczenie klasyfikatora przebiega z wykorzystaniem zbioru treningowego, natomiast w celu sprawdzenia generalizacji klasyfikatora wykorzystywana jest testowa część danych. Metoda ta wydaje się być najbardziej zbliżona do rzeczywistości. Wadą w tym wypadku jest ograniczenie ilości danych treningowych, co w przypadku, gdy do dyspozycji jest niewielka liczba próbek lub występują ograniczenia sprzętowe, prowadzi do pogorszenia skuteczności klasyfikatora.

3.3. Skuteczność klasyfikacji

Używając zbioru treningowego stworzonego za pomocą narzędzia przedstawionego w sekcji 1.2 oraz klasyfikatorów problemu rozpoznania znaków pisma odręcznego wybranych w rodziale 1 przeprowadzone zostaną badania wykorzystujące algorytmy wymienione w sekcjach 1.3, 1.4 oraz sieci neuronowe 1.5, 1.6. Badanie zostało oparte o zbiór testowy zawierający 3600 elementów, uzyskane zostały nastepujące rezulataty:

Nazwa	Dokładność
kNN	38%
Random Forest	28%
Sieć neuronowa	73%
CNN	85%

Tabela 3.1. Wyniki badań skuteczności klasyfikatorów

Następnie używając zbioru testowego z poprzednich badań wykonane zostało porównanie rozpoznań aplikacji korzystających z gotowych bibliotek

Według wyników badań, najlepszą metodą klasyfikacji znaków pisma odręcz-

Nazwa	Dokładność
Computer Vision API	29%
Tesseract API	32%

Tabela 3.2. Wyniki badań skuteczności klasyfikatorów

nego jest wykorzystanie konwolucyjnych sieci neuronowych proponowanych przez autora.

3.4. Czas treningu

Używając zbioru treningowego stworzonego za pomocą narzędzia przedstawionego w sekcji 1.2 oraz klasyfikatorów problemu rozpoznania znaków pisma odręcznego wybranych w rodziale 1 przeprowadzone zostaną badania wykorzystujące algorytmy wymienione w sekcjach 1.3, 1.4 oraz sieci neuronowe 1.5, 1.6. Badanie zostało oparte o zbiór testowy zawierający 3600 elementów, uzyskane zostały nastepujące rezulataty:

Nazwa	Czas
kNN	12.3s
Random Forest	9.0s
Sieć neuronowa	159s
CNN	3969s

Tabela 3.3. Wyniki badań czasu treningu klasyfikatorów

Badanie bibliotek uczenia maszynowego zostało pominięte ze względu na to, iż w procesie nie występuje etap treningu.

3.5. Porównanie modeli danych

W celu przeprowadzenia porównania modeli danych treningowych problemu rozpoznawania polskich znaków pisanych zostało stworzonych 5 różnych zbiorów da-

nych, każdy z nich posiada konkretną ilość elementów każdego znaku oraz rozdzielczość w jakiej zapisano poszczególne obrazy. Badanie ma na celu sprawdzenie czy w warunkach mocno ograniczających przeprowadzanie treningu modelu, np. na urządzeniach mobilnych, istnieje znacząca różnica w wynikach danych. Trening przeprowadzono korzystając proponowaną przez autora splotową sieć neuronowa, ponieważ rezultaty w poprzednich badaniach pokazały, iż pokazuje ona najlepsze parametry wymagane do przeprowadzenia takiego testu.

Model	Ilość elementów	Rozdzielczość
1	72000	28 x 28 pikseli
2	250000	28 x 28 pikseli
3	500000	28 x 28 pikseli
4	750000	28 x 28 pikseli
5	125000	32 x 32 pikseli

Tabela 3.4. Dane wejściowe - porównanie modeli danych

Wyniki treningu poszczególnych modeli, dla liczby iteracji równej 5000, do testu skuteczności został użyty zbiór testowy używany w poprzednich badaniach.

Model	Skuteczność	Czas treningu
1	90%	1h 32m
2	88%	4h 51m
3	94%	6h 02m
4	—	—
5	92%	2h 42m

Tabela 3.5. Wyniki - porównanie modeli danych

Badanie bibliotek uczenia maszynowego zostało pominięte ze względu na to, iż w procesie nie występuje etap treningu.

3.6. Rozmiar modelu, a skuteczność terningu

Z przeprowadzonych badań wynika, iż skuteczność treningu w jakimś stopniu zależna jest od wielkości modelu treningowego. Wszystkie porównania przeprowadzane były w warunkach domowych co oznacza duże ograniczenia zasobów pamięci RAM, CPU oraz GPU. Trening na największym zbiorze danych nie był w stanie uruchomić się na środowisku testowym ponieważ zasoby potrzebne na operacje na macierzach wykonywane są w pamięci RAM, która jest znacznie ograniczona.

Jednak z obserwacji wynika, iż im bardziej precyzyjne zawarte są dane w modelu, trening sieci neuronowych resultuje lepszym wynikiem końcowym. Małe modele idealne są do celów sprawdzenia czy istnieje poprawa jakości wyodrębnionych cech poszczególnych obrazów i czy klasyfikacja zbioru testowego pokaże lepsze wyniki.

ROZDZIAŁ 4

Podsumowanie i wnioski

4.1. Zalety aplikacji wieloplatformowych

- Xamarin bazowo unifikuje przede wszystkim backendową część aplikacji. Jeśli tworzysz aplikacje z rozbudowaną logiką biznesową, to warto ją pisać tylko raz. Xamarin na pewno w tym pomoże
- Jeśli layout aplikacji opiera się o standardowe elementy i wzorce, to warto pokusić się o użycie Xamarin.Forms, który obsługuje kilkadziesiąt kontrolek, które później mapowane są na natywne rozwiązania
- Jeśli martwi Cię konieczność posiadania osobnego zestawu programistów na każdą platformę, to również warto dać szansę Xamarinowi. Nawet podejście native można w całości ograć w języku C#. Xamarin mapuje 100% API z Androida oraz iOS
- Nawet jeśli znasz rozwiązania natywne, wciąż możesz poczuć się jak w domu, używając Xamarina. Dla przykładu, w Androidzie wciąż można używać Activity, kod layoutu pisać w plikach AXML, a aplikację konfigurować w Manifeście. Jednocześnie przy okazji łatwo jest napisać kawał kodu, który można użyć na dowolnej inne platformie. Bajka? Niekoniecznie. Xamarin
- Jeśli pracowałeś z jakąkolwiek technologią operującą na XAMLu (WPF, Silverlight, Universal Apps, UWP), to również w tej sytuacji nie powinieneś mieć problemów by się odnaleźć w temacie
- Pod Xamarina można programować w Visual Studio [Community] dla Windows lub Xamarin Studio dla OS X
- Kod źródłowy całego rozwiązania dostępny jest na GitHubie!

4.2. Wady aplikacji wieloplatformowych

- Podejście Forms zazwyczaj nie pozwala na napisanie całej aplikacji. Część kodu i tak musi być pisania z użyciem dedykowanych konstrukcji warunkowych, a w przypadku złożonych interfejsów, może to być większa część kodu
- Do pracy z Xamarinem wymagana jest również przynajmniej bazowa znajomość C#. W podejściu Xamarin.Forms, trzeba się również nauczyć XAMLa.
- Ponadto przy wydaniu każdej nowej wersji wspieranych systemów, musimy czekać na wrappery. Oczywiście w chwili obecnej pojawiają się one praktycznie natychmiast, ale pytanie czy tak będzie zawsze?
- Xamarin bywa również niestabilny. Zdarzają się wersje lepsze i gorsze. Czasami pojawiają się problemy w sytuacji gdy odpalamy solucję przygotowaną w Visual Studio na Macu i odwrotnie. Zdarzają się również inne błędy.
- Ta technologia ma jeszcze jedno ryzyko. Obecnie jej właścicielem jest Microsoft, który czasem bywa nieprzewidywalny. Na szczęście tak jak pisałem wcześniej kod źródłowy Xamarina jest otwarty i można go znaleźć na GitHubie

4.3. Uczenie maszynowe w aplikacjach mobilnych

Wykorzystywanie technik uczenia maszynowego, czy to są algorytmy typu random forrest, czy też sieci neuronowe, wymagają olbżymich zasobów sprzętowych, w związku z tym urządzenia mobilne, których moc obliczeniowa nie może równać się z komputerami klasy PC czy klastrów procesorów graficznych nie nadają się do treningów. Istniejące gotowe rozwiązania jak na przykład przedstawione w pracu usługi Cognitive Services od firmy Microsoft idealnie pasują do rynku aplikacji mobilnych, ponieważ wszystkie obliczenia związane z klasyfikacją znajdują się po stronie serwera, wadą tego rozwiązania jest natomiast brak ingerencji w modele danych oraz brak jakiejkolwiek personalizacji, który mógłby pomóc w rozwiązaniu problemu.

Idealnym rozwiązaniem, również wykorzystanym w tej pracy przez autora jest połączenie mocy obliczeniowej komputerów PC lub laptopów by wytrenować odpowiedni model danych realizujący problem, a następnie stworzenie aplikacji webowej, która implementowałaby określony interfejst, tak aby była możliwość konsumowania go po stronie aplikacji mobilnej. W ten sposób zyskujemy nie zawodność po stronie mobilnej oraz personalizacje i wgląd w model danych po stronie serwera.

4.4. Najlepsza metoda rozpoznawania pisma odręcznego

4.5. Koszt

Zakończenie

Celem pracy autora było zaprezentowanie istniejących rozwiązań pozwalających szybkie przeprowadzenie rozpoznania znaków oraz przedstawienie własnej implementacji sieci neuronowej, która rozwiązuje problem należący do tej samej klasy.

Bibliografia

[1] Leslie Lamport, *LaTeX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.

Oświadczenie

Ja, niżej podpisany(a) oświadczam, iż przedłożona praca dyplomowa została wykonana przeze mnie samodzielnie, nie narusza praw autorskich, interesów prawnych i materialnych innych osób.

.....	.....
data	podpis