POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka (AIR)
SPECJALNOŚĆ: Technologie informacyjne w systemach automatyki (ART)

PRACA DYPLOMOWA INŻYNIERSKA

Rozpoznawanie tekstu przy pomocy konwolucyjnych sieci neuronowych i korekty słownikowej

Convolutional neural networks with dictionary validation for text recognition

AUTOR:
Michał Banach

PROWADZĄCY PRACĘ:
dr inż. Piotr Ciskowski, K-9

OCENA PRACY:

WROCŁAW 2017

Spis treści

Rozdział 1

Wstęp

Niniejsza praca została wykonana na potrzeby projektu zespołowego *"Rozpoznawanie tekstu w czasie rzeczywistym z wykorzystaniem uczenia maszynowego na platformie mobilnej Android"* stworzonego przez Kamila Kuczaja, Jana Bednarskiego i Michała Banacha. Podział obowiązków wyglądał następująco:

Jan Bednarski - Stworzenie aplikacji na platformę Android, która będzie umożliwiała zrobienie zdjęcia oraz odnalezienie na nim słów.

Michał Banach - Rozpoznawania znalezionych słów przy użyciu konwolucyjnej sieci neuronowej.

Kamil Kuczaj - Rozpoznawanie tekstu przy pomocy innych technik klasyfikowania obrazu.

Jej celem jest rozpoznanie tekstu widocznego na obrazie wejściowym. Zostało to osiągnięte w trzech krokach:

1. segmentacja znaków,
2. rozpoznawanie znaków,
3. korekcja metodą słownikową.

Na potrzeby dzielenia obrazu na części zawierające po jednym znaku został opracowany i zaimplementowany oddzielny algorytm, który również jest opisany w niniejszej pracy. Najbardziej rozbudowaną częścią pracy jest ta odpowiedzialna za rozpoznawanie liter i cyfr. Została ona zaimplementowana przy użyciu głębokiej konwolucyjnej sieci neuronowej na podstawie sieci LeNet [3] z wykorzystaniem nowoczesnych metod uczenia maszynowego. Trzecia część została wykonana na podstawie programu napisanego przez Peter'a Norvig'a i jest wykorzystywana na licencji MIT.

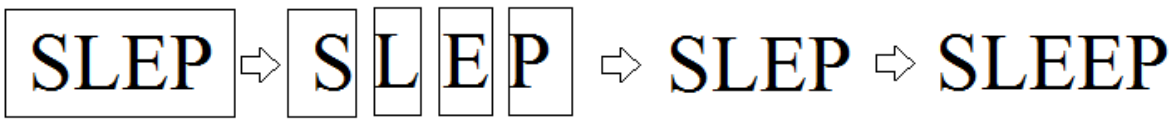
Rozdział 2

Cele pracy

Głównym celem niniejszej pracy jest poprawne rozpoznanie tekstu przedstawionego na zdjęciu, ale został on podzielony na mniejsze części, aby ułatwić etap projektowania oraz podzielić implementację na mniejsze logiczne i łatwe do zrozumienia części. Zostały wyodrębnione następujące części:

1. wstępne przetworzenie danych,
   (a) odfiltrowanie szumów,
   (b) utworzenie obrazu binarnego,
2. stworzenie algorytmu do segmentacji znaków na obrazie,
3. rozpoznawanie znaków,
   (a) utworzenie ciągu uczącego,
   (b) zaprojektowanie konwolucyjnej sieci neuronowej,
   (c) nauczenie sieci,
4. zaimplementowanie algorytmu słownikowej korekcji tekstu.

Wszystkie powyższe punkty zostały opisane w kolejnych rozdziałach oraz przedstawione schematycznie na rysunku 2.1.



Rysunek 2.1: Kolejne etapy przetwarzania: obraz wejściowy, obrazy po segmentacji, tekst po rozpoznaniu znaków, tekst poprawiony metodą słownikową.

Rozdział 3

Zakres pracy

W rozdziale 4. została opisana architektura konwolucyjnych sieci neuronowych na podstawie sieci LeNet-5. Wyjaśnione zostało również działanie poszczególnych warstw:

- warstwa gęsta,
- warstwa konwolucyjna,
- warstwa max pooling,
- dropout.

W rozdziale 5. opisane zostały algorytmy optymalizacji, które były testowane w ramach niniejszej pracy tj.

- algorytm AdaGrad,
- algorytm Adam,
- algorytm NAdam.

W rozdziale 6. przedstawione zostało wstępne przetworzenie obrazu wejściowego, obejmujące usuwanie szumów ze zdjęcia i konwersję do obrazu binarnego, oraz algorytm segmentujący. Został on zaprojektowany i zaimplementowany na potrzeby niniejszej pracy. Jego zadaniem jest wyznaczenie części obrazu, na których znajduje się tylko jeden znak.

Główna część pracy tj. rozpoznawanie liter została opisana w rozdziale 7. Opisane zostały etapy wybory architektury konwolucyjnej sieci neuronowej, a także wyboru algorytmu optymalizacji do uczenia sieci.

Algorytm korekcji słownikowej został przedstawiony w rozdziale 8. Przedstawiona została główna logika tej części programu, baza danych na podstawie, której wybierana jest odpowiedź oraz skuteczność algorytmu.

W 9. rodziale opisane zostały moduły stworzone w ramach niniejszej pracy. Przedstawiono listingi kodu zawierające główną logikę oraz przykłady wywołań.

W rozdziale 10. znajduje się podsumowanie pracy oraz wypełnionych celów.

Rozdział 4

Architektura konwolucyjnych sieci neuronowych

Rozpoznawanie liter oraz cyfr na zdjęciach za pomocą uczenia maszynowego nie jest łatwym zadaniem. Standardowe sieci neuronowe są bardzo czułe na zmiany obrazu wejściowego np. translacje i rotacje obiektu. Już niewielkie zakłócenia tego typu potrafią uniemożliwić odpowiednią klasyfikację. Do tego zwyczajne sieci neuronowe przez swoją architekturę mają bardzo dużo wag (każdy piksel do każdego neuronu w pierwszej warstwie) i przez to są kosztowne obliczeniowo. Wyspecjalizowaną wersją sieci wykorzystywaną dla przetwarzania zdjęć są konwolucyjne sieci neuronowe, które wykorzystują tzw. mapy cech albo filtry dopasowywane do obrazu. W ten sposób pojedynczą cechę można wykryć w dowolnym miejscu zdjęcia przy małym nakładzie obliczeniowym, ponieważ wykorzystuje się dwuwymiarową konwolucję z tym samym filtrem.

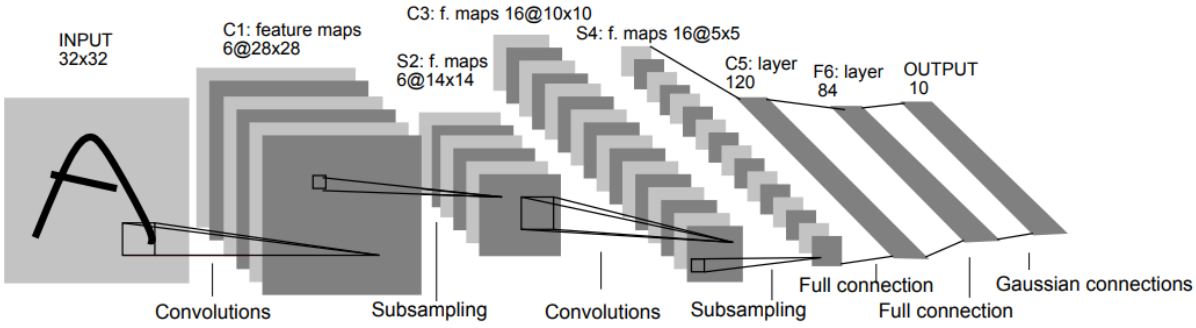
W niniejszej pracy wykorzystano architekturę konwolucyjnej sieci neuronowej na podstawie sieci LeNet-5 opisanej w [3], która została przedstawiona na rysunku 4.1. Składała się ona z następujących 7 warstw:

1. warstwa konwolucyjna $6 \times 5 \times 5$,
2. warstwa poolingowa 2×2,
3. warstwa konwolucyjna $16 \times 5 \times 5$,
4. warstwa poolingowa 2×2,
5. warstwa gęsta o 120 neuronach,
6. warstwa gęsta o 84 neuronach,
7. warstwa gęsta o 10 neuronach.

W kolejnych podrozdziałach zostały opisane warstwy konwolucyjnej sieci neuronowej stworzonej w ramach tej pracy.

4.1 Max pooling

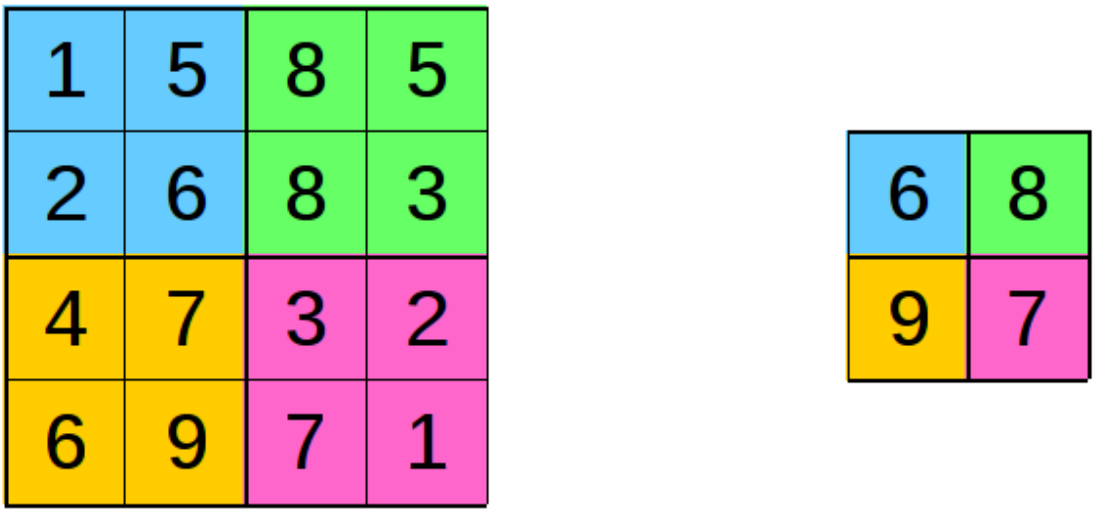
Rozmiar $a \times b$.
Warstwa ta dzieli obraz na n prostokątów o rozmiarze $a \times b$, przy czym najczęściej spełniony jest warunek $a = b$. W obrazie wyjściowym każdy piksel odpowiada takiemu prostokątowi w obrazie wejściowym, a jego wartość jest równa maksymalnej wartości wejściowej. Warstwa max poolingowa ma kilka skutków:



Rysunek 4.1: Architektura konwolucyjnej sieci neuronowej LeNet-5 wykorzystywanej do rozpoznawania cyfr [3].

- lepsza generalizacja,
- mniejsze prawdopodobieństwo przeuczenia sieci,
- zmniejszenie ilości wag do nauczenia,
- zmniejszenie ilości przetwarzanych danych,
- niwelowanie lekkich rotacji w obrazie wejściowym.

Konsekwencją otrzymania wszystkich powyższych zalet jest utrata danych, co, jeśli nastąpi na zbyt wczesnym etapie w sieci, może spowodować, że nie będzie się ona mogła się nauczyć odpowiedniego klasyfikowania obrazów. Innym skutkiem ubocznym jest przyspieszenie uczenia i działania sieci ze względu na mniejszą ilość danych do przetworzenia.



Rysunek 4.2: Sposób działania warstwy max pooling: obraz wejściowy, obraz wyjściowy.

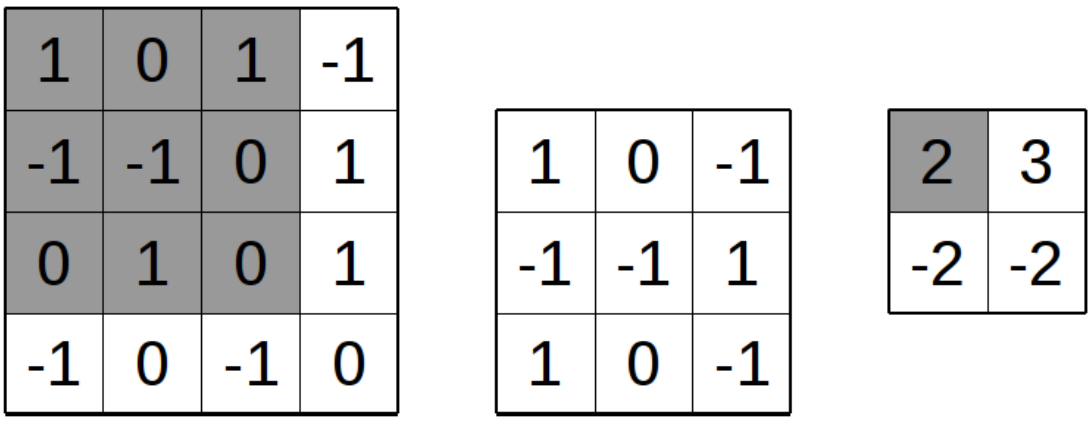
Innym stosowanym typem warstwy poolingowej jest tzw. average pooling, który zamiast wybierania maksymalnej wartości oblicza średnią ze wszystkich wartości w macierzy wejściowej.

4.2 Warstwa konwolucyjna

Rozmiar $a \times b \times n$.
Warstwa konwolucyjna powstała na skutek założenia, że wejściem konwolucyjnej sieci neuronowej może być tylko obraz. W takich warunkach sensownym algorytmem jest szukanie zależności między danym pikselem, a jego sąsiadami. Dzięki temu sieć uczy się rozpoznawać wzory na obrazie np. krawędzie w pierwszych warstwach, prostokąty i trójkąty w następnych aż do bardzo skomplikowanych kształtów w głębszych warstwach. Warstwa ta składa się z n filtrów o rozmiarze $a \times b$, dla obrazów dwuwymiarowych. Dla obrazów o trzech wymiarach (np. RGB) filtry mają też trzeci wymiar (np. $a \times b \times 3$ dla przypadku RGB). Dla każdego filtru tworzony jest obraz wyjściowy tak, że filtr jest splatany z obrazem wg wzoru:

$$Y_k = \sum_{i=1}^{a} \sum_{j=1}^{b} (X_{ij} F_{ij}),$$

gdzie Y_k - komórka obrazu wyjściowego,
X_{ij} - komórka o współrzędnych i oraz j w danych wejściowych,
F_{ij} - komórka o współrzędnych i oraz j filtru.
Przesunięcie domyślne filtru jest równe 1, ale je też można zmienić.



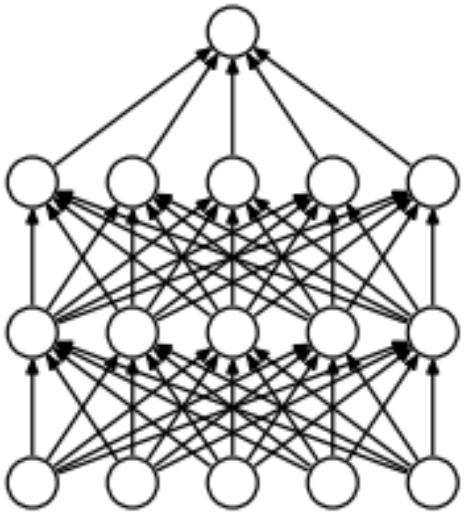
Rysunek 4.3: Sposób działania filtru w warstwie konwolucyjnej: obraz wejściowy, filtr, obraz wyjściowy.

Każda warstwa konwolucyjna po pierwszej ma trzeci wymiar filtru przystosowany do danych wejściowych tzn. kiedy pierwsza warstwa miała rozmiar $3 \times 3 \times 32$ to trzeci wymiar filtrów w warstwie drugiej będzie równy 32, aby wyciągnąć dane ze wszystkich warstw obrazu wejściowego. Każdy z filtrów odpowiada jednemu neuronowi w warstwie gęstej.

4.3 Warstwa gęsta

Rozmiar n.
Warstwa gęsta to standardowa warstwa w sieci neuronowej, w której każdy z neuronów w jednej warstwie jest połączony wagą z każdym neuronem w kolejnej warstwie. Neurony te wprowadzają wiele kolejnych wymiarów do problemu optymalizacji przy uczeniu sieci oraz nie mają zdolności do przenoszenia danych zgodnie z miejscem ich pojawienia się

w dwuwymiarowym obrazie wejściowym, dlatego stosuje się je dopiero na końcu, kiedy warstwy konwolucyjne zdążyły już wyodrębnić konkretne cechy z obrazu wejściowego i to właśnie na ich podstawie warstwy gęste wyciągają końcowe wnioski, co do np. klasyfikacji danych wejściowych.



(a) Standard Neural Net

Rysunek 4.4: Standardowa sieć neuronowa zbudowana z warstw gęstych. [2]

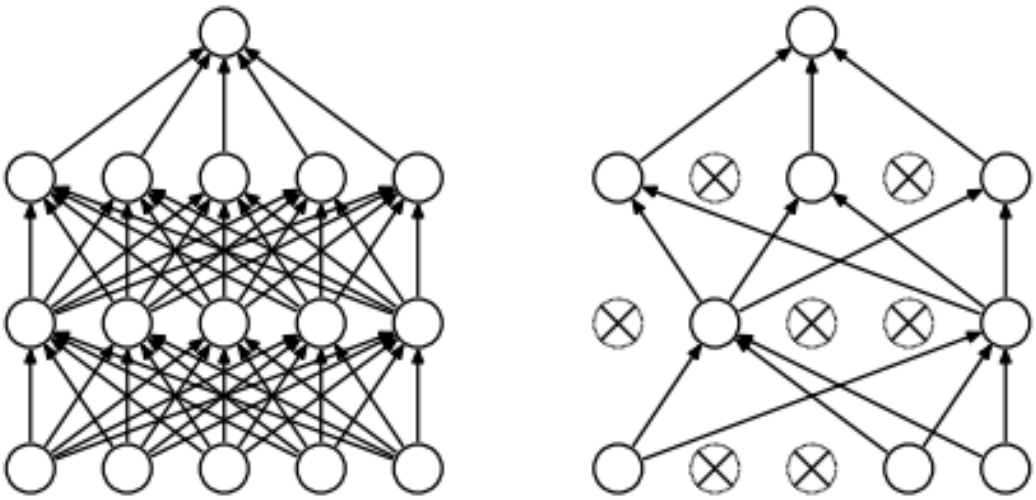
4.4 Dropout

Dropout jest to metoda unikania przeuczenia sieci. Polega na tym, że podczas uczenia sieci pewien procent neuronów (ustawiany podczas tworzenia sieci) jest nieaktywny. W skutek tego nie bierze również udziału w poprawianiu wag po danym kroku uczenia. Działanie dropoutu można rozumieć intuicyjnie na kilka sposobów:

1. Uczenie sieci z użyciem dropoutu działa jak uczenie pewnej ilości różnych sieci neuronowych na tych samych danych wejściowych. Następnie wszystkie sieci "głosują" na temat tego, jaki powinien być wynik. W skutek tego otrzymujemy wiele nauczonych sieci, dzięki czemu lepiej generalizują oraz lepiej klasyfikują.

2. Dropout wymusza na sieci rozwijanie wielu ścieżek do rozpoznania każdej cechy, ponieważ podczas każdego kroku taka ścieżka może zostać przerwana. Powstałe ścieżki są redundantne, co już podczas działania sieci sprawia, że nie wszystkie cechy muszą zostać wykryte, aby obraz został poprawnie zaklasyfikowany.

Dropout jest używany wyłącznie podczas uczenia sieci. Po zakończonym treningui jej wyjścia muszą zostać przeskalowane o tyle, ile neuronów było nieaktywnych przy każdym przebiegu np. jeśli sieć jest uczona z wykorzystaniem dropoutu 50% (połowa neuronów przy każdym kroku uczenia było dezaktywowanych) to muszą być podzielone przez 2.



(a) Standard Neural Net (b) After applying dropout.

Rysunek 4.5: Modele sieci neuronowej z dropout'em. **a)** Standardowa sieć neuronowa z dwiema warstwami ukrytymi. **b)** Przykład sieci odchudzonej przez zastosowanie dropout'u do sieci po lewej. Przekreślone neurony zostały opuszczone. [2]

Rozdział 5

Algorytmy optymalizacji

Proces uczenia konwolucyjnej sieci neuronowej polega na takim dobraniu jej wag, aby zminimalizować ustaloną funkcję kosztu. Jej wartość jest tym wyższa im mniej odpowiedź sieci zgadza się z żądanym wyjściem tj. kiedy sieć błędnie klasyfikuje obraz wejściowy. W tym procesie potrzebne są algorytmy, które pozwolą znaleźć wektor wag zbliżony do optymalnego. W następujących podrozdziałach zostały opisane 3 przetestowane w ramach tej pracy algorytmy oparte na metodach największego spadku. Są one rozwinięciami algorytmu wykorzystującego zwykły gradient stochastyczny, co sprawia, że szybciej znajdują optimum.

Istnieją różne sposoby numerycznego obliczania gradientu, ale w praktyce najlepiej sprawdza się formuła dana wzorem

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h},$$

gdzie h - promień otoczenia, w którym gradient jest obliczany.
Pomimo tego, że powyższa metoda jest kosztowna obliczeniowo to pozwala dokładniej obliczyć wartość gradientu.

Tylko metody pierwszego rzędu zostały wybrane, ponieważ algorytmy wyższego rzędu są nieskuteczne przy dużych zestawach danych. Takie ciągi uczące trzeba wczytywać częściami ze względu na ograniczenia pamięci komputera, a metody drugiego rzędu potrzebują całego zestawu uczącego, aby obliczyć poprawny krok.

5.1 Algorytm AdaGrad

Algorytm ten polega na kumulowaniu kwadratów gradientów w poszczególnych kierunkach przy każdym kroku. W warunkach kiedy przestrzeń, po której się poruszamy jest szybkozmienna w pewnych kierunkach, a wolnozmienna w innych, AdaGrad pozwala dostosować szybkość uczenia w każdym z nich. Wielkość kroku w stronę gwałtownych zmian w przestrzeni jest zmniejszana, a zwiększana w kierunku łagodnych zmian. Minusem tej metody jest to, że przy dużej liczbie kroków skumulowana wartość kwadratów gradientów może sprawić, że efektywne tempo uczenia zostanie zmniejszone do zera, a algorytm nigdy nie znajdzie minimum.

Poniżej przedstawiono pseudokod pojedynczego kroku w algorytmie AdaGrad.

```
allPrevious = dx^2
x += - learningRate * dx / sqrt(allPrevious)
```

5.2 Algorytm Adam

Jest to połączenie algorytmu AdaGrad z momentum oraz zmianą sposobu kumulowania poprzednich gradientów. Wykorzystany jest tu efekt zapominania, który sprawia, że nawet przy bardzo dużych liczbach kroków algorytm nie zatrzyma się ze względu na tempo uczenia zmniejszone do zera.

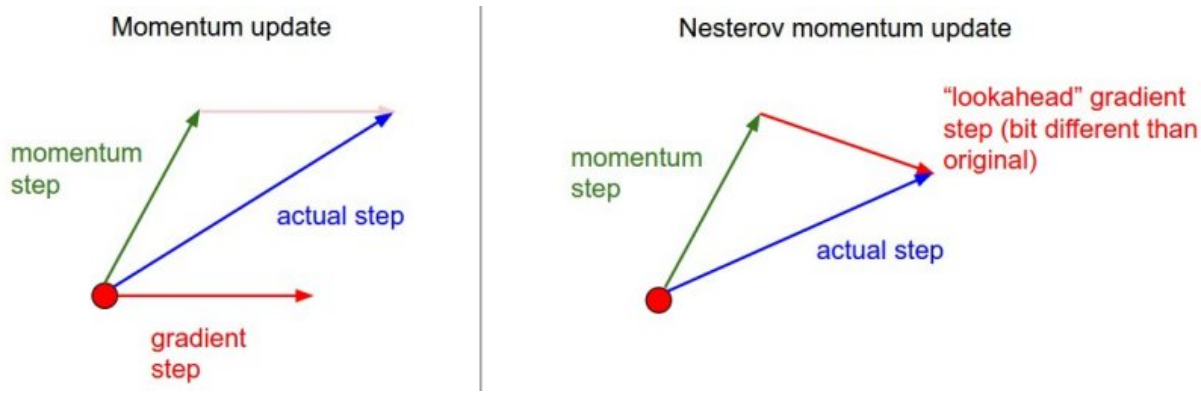
Poniżej przedstawiono pseudokod pojedynczego kroku w algorytmie Adam.

```
momentum = alfa * momentum + (1 - alfa) * dx
allPrevious = beta * allPrevious + (1 - beta) * dx^2
x += - learningRate * momentum / sqrt(allPrevious)
```

Parametry alfa i beta są stałymi ustawianymi przed rozpoczęciem uczenia. Nie są one obliczane według wzorów, ale zgadywane na podstawie doświadczenia. Można wyszukać najlepszą wartość dzięki cross-walidacji tzn. wyszkoleniu wielu konwolucyjnych sieci neuronowych i sprawdzeniu, która z nich ma najlepszą skuteczność.

5.3 Algorytm NAdam

Jest to modyfikacja algorytmu Adam wykorzystująca momentum Nesterova zamiast zwyczajnego momentum. Zmiana ta polega na tym, że w kroku nie jest stosowany gradient w miejscu, w którym znajduje się algorytm, ale gradient w miejscu, w którym algorytm by się znalazł gdyby poruszył się zgodnie z obecnym momentum.



Rysunek 5.1: Schemat działania algorytmu z momentum Nesterova. [5]

Rozdział 6

Wstępne przetwarzanie i segmentacja

Kroki opisane w niniejszym rozdziale służą przygotowaniu danych do etapu rozpoznawania przez konwolucyjną sieć neuronową. Jeśli obraz wejściowy jest zgodny z założeniami, to usunięcie szumów oraz utworzenie obrazu binarnego pozwala prawidłowo działać algorytmowi segmentującemu. Prawidłowa praca tego algorytmu jest niezbędna, ponieważ złe podzielenie zdjęcia całkowicie uniemożliwia poprawne rozpoznanie oraz poprawianie tekstu.

6.1 Założenia wobec danych wejściowych

Praca ta została wykonana jako część projektu zespołowego, dlatego nie jest przeznaczona do tego żeby działać samodzielnie. Obraz wejściowy powinien być dostarczany przez inny moduł. Obraz ten powinien spełniać następujące założenia, aby zapewnić prawidłową pracę programu:

1. na każdym obrazie wejściowym znajduje się dokładnie jedno słowo,
2. słowo to jest w języku angielskim,
3. tekst jest ciemny na jasnym tle,
4. obraz ma wysokość 76 pikseli,

Pierwsze założenie wynika to z tego, że jeśli na obrazie będzie więcej słów to odstępy między nimi nie zostaną wydzielone przez algorytm segmentujący jako oddzielne znaki, co z kolei uniemożliwia ich przetworzenie oraz utrudnia rozpoznanie znaku, do którego ten odstęp został przyporządkowany, ponieważ znak ten będzie przesunięty w prawo lub w lewo.

Drugie ograniczenie to skutek wyboru metody korekcji słownikowej. Metoda ta polega na wykorzystaniu bazy danych stworzonych ze słów występujących w języku angielskim i na generowaniu różnych kombinacji tego słowa ze wszystkimi literami w celu wytypowania odpowiedniej poprawki. Mechanizm ten mógłby być niewystarczający, żeby podołać wyzwaniom stawianym przez polską ortografię.

Trzecie założenie jest efektem metod używanych przy wstępnym przetwarzaniu obrazu oraz algorytmu segmentacji. Podczas przetwarzania używane jest progowanie, a następnie algorytm dzieli znaki, znajdując miedzy nimi pionowe białe linie. Jeśli tło na tym etapie będzie czarne to podział na segmenty będzie niemożliwy.

Czwarte założenie to kolejne, które wynika z działania algorytmu segmentującego. Algorytm ten ma założone maksymalne możliwe odchylenie wyznaczanej granicy o 4 piksele.

Dla obrazów o różnych wysokościach ten limit musiałby się zmieniać, aby jednocześnie zapewnić efektywny krok na boki oraz zapobiec sytuacji, w której znak nie zostanie wykryty z powodu 'przeskoczenia' go przez ten algorytm.

6.2 Odszumianie i binaryzacja obrazu

Zanim obraz wejściowy zostanie podzielony na części zawierające po jednym znaku, musi on przejść wstępne przetworzenie, żeby zmniejszyć wpływ zakłóceń na rozpoznawanie oraz ułatwić pracę algorytmowi segmentującemu.

Na początku obraz wejściowy jest poddawany działaniu filtru Gaussa. Dzięki temu, że jest to filtr dolnoprzepustowy, obraz jest lekko uśredniany oraz usuwane są zakłócenia o wysokich częstotliwościach, takie jak np. "pieprz i sól". Dzięki temu zwiększone jest prawdopodobieństwo, że dwa sąsiednie znaki nie zostaną połączone, ponieważ czarny piksel, który pojawił się tam wskutek zakłóceń, prawdopodobnie będzie miał obniżoną wartość wskutek filtrowania. Gdyby to nie miało miejsca, to rozdzielenie danych znaków na późniejszym etapie stanie się niemożliwe.

Kolejną korzyścią wynikającą z zastosowania takiego filtru jest usunięcie jasnych pikseli, które mogły się pojawić w literach i cyfrach, ponieważ takie zakłócenie mogłoby utrudnić poprawne rozpoznanie danego znaku przez konwolucyjną sieć neuronową. Również czarne piksele, które pojawiły się na tle mogłyby mieć taki skutek, a dodatkowo, ze względu na sposób działania algorytmu segmentującego, mogłoby to doprowadzić do wygenerowaniu wielu niepoprawnych segmentów, które nie zawierałyby liter ani cyfr, a tylko np. jeden piksel, który znalazł się tam wskutek zakłóceń.

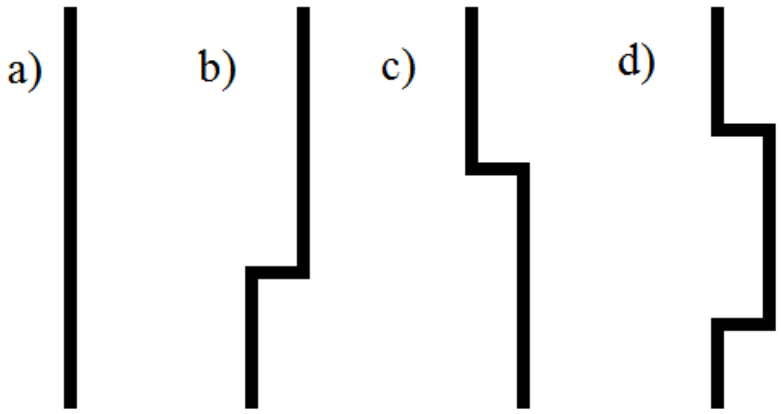
1	2	1
2	4	2
1	2	1

Rysunek 6.1: Przykładowa maska filtru Gaussa.

Następnym etapem wstępnego przetwarzania jest progowanie obrazu tak, aby uzyskać obraz binarny, na którym litery i cyfry są czarne, a tło - białe. Jest to progowanie globalne tzn. dla całego obrazu ustalony jest jeden próg, a wszystkie piksele, których wartość jest mniejsza niż ten próg są ustawiane jako czarne, a pozostałe piksele zostają białe. Ten krok pozwala oddzielić od siebie znaki, które na obrazie wejściowym mogły być połączone ciemnymi pikselami wynikającymi z zakłóceń. Progowanie w tym przypadku jest używane komplementarnie do zastosowania filtru Gaussa. Wszystkie piksele, których wartość została podwyższona podczas filtrowania powyżej zadanego progu zostają usuwane z obrazu, dzięki czemu możliwe jest usunięcie z danych wejściowych zakłóceń o wysokiej częstotliwości.

6.3 Podział na segmenty

Algorytm rozdzielający poszczególne znaki, który został zaprojektowany i zaimplementowany na potrzeby tej pracy, opiera się na założeniu, że jeśli istnieje na obrazie wejściowym pionowa biała linia z maksymalnym odchyleniem od pionu wynoszącym 4 piksele, to ta linia oddziela od siebie 2 różne znaki. Linia ta może być prosta, odchylona w prawo, odchylona w lewo jak również w obie strony, jak pokazano na rysunku 6.2.



Rysunek 6.2: Linie: a) prosta, b) odchylona w prawo, c) odchylona w lewo oraz d) w obie strony.

Każda wyznaczona granica jest zapamiętywana jako współrzędna jej wystąpienia tzn. jeśli pionową linię dało się przeprowadzić nad 26-tym pikselem licząc od lewej strony to zapamiętywana jest liczba 26. Trochę bardziej skomplikowane jest wyznaczanie granicy, kiedy nie jest ona prosta. W takim wypadku zapisywana jest średnia arytmetyczna współrzędnych: tej, od której linia się rozpoczęła oraz tej, na której linia się zakończyła.

Na rysunku 6.5 przedstawiono algorytm wyznaczający granice na obrazie wejściowym.

Takie postępowanie skutkuje tym, że zostaje wyznaczonych wiele granic między danymi dwoma znakami, z których wszystkie są przyległe do siebie nawzajem, co pokazano na rysunku 6.3.

Rysunek 6.3: Wszystkie wyznaczone granice.

Dzielenie obrazu wejściowego w miejscu każdej wykrytej granicy byłoby bezcelowe, gdyż prowadziłoby do stworzenia wielu pustych obrazów np. obraz pomiędzy granicą o współrzędnej szerokości równej 25, a granicą o tej współrzędnej równej 26 miałby rozmiary: wysokość = wysokość obrazu wejściowego, szerokość = 0. Z tego względu przed dzieleniem obrazu następuje grupowanie granic. Wszystkie granice, które są przyległe do siebie, tj. ich współrzędna szerokości nie różni się o więcej niż 1, są przyporządkowywane do jednego zbioru. Dopiero na podstawie tych wszystkich zbiorów określana jest jedna linia wzdłuż, której obraz zostanie rozdzielony. Dzieje się to według wzoru:

$$g = \frac{1}{N} \sum_{i=1}^{N} X_i,$$

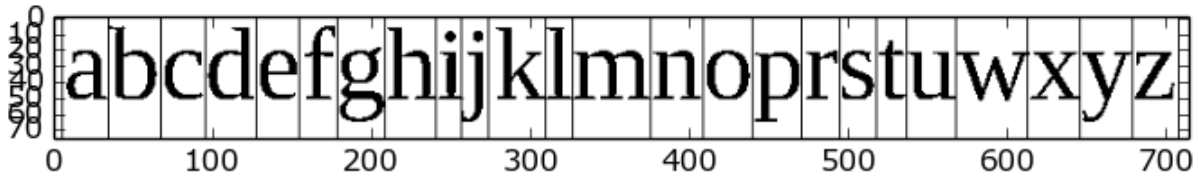
gdzie g - współrzędna wyznaczonej linii podziału obrazu,
N - komórka o współrzędnych i oraz j w danych wejściowych,
X_i - współrzędna i-tej granicy.

Po wyznaczeniu granic obraz jest wzdłuż nich dzielony, tak jak pokazano na rysunku 6.4.

Jak widać na rysunku 6.4, efektem ubocznym działania tego algorytmu jest wydzielenie również części obrazu, które nie zawierają żadnego znaku na początku i na końcu obrazu wejściowego. Rozwiązaniem zastosowanym w tej pracy jest usunięcie pierwszego i ostatniego wydzielonego fragmentu. Taka decyzja została podjęta, ponieważ bardzo mało prawdopodobnym jest aby prawidłowy obraz wejściowy nie zaczynał i nie kończył się w taki sposób. Obraz, w którym już na wysokości pierwszego piksela zaczyna się litera lub, w którym na wysokości ostatnie piksela ostatnia litera się kończy spowodowałby, że kolejno pierwsza i ostatnia litera tego obrazu nie zostałaby oddzielona jako oddzielny znak do segmentacji, ale możliwe, że ten błąd udałoby się naprawić podczas korekty słownikowej. Obraz taki nie pojawił się podczas testowania ani razu i wobec tego przyjęto, że prawdopodobieństwo jego wystąpienia jest małe.

Innym sposobem na ominięcie tego problemu mogłoby być zmienienie algorytmu wyszukującego granice w taki sposób, aby nie wyznaczał ich, o ile wcześniej nie został wykryty żaden obiekt. Takie podejście pozwoliłoby uniknąć wydzielenia pustego fragmentu na początku obrazu, ale prawdopodobnie skutkowałoby trudnościami z rozpoznawaniem pierwszego znaku, gdyż litera byłaby potencjalnie znacznie przesunięta w prawo, co mogłoby nie zostać odpowiednio skategoryzowane przez konwolucyjną sieć neuronową.

Kolejnym sposobem może być nauczenie sieci neuronowej rozpoznawać, kiedy na obrazie nie ma żadnego znaku, co powinna skategoryzować to jako pusty element. To podejście wymaga wygenerowania wielu (ok. 1000) obrazów do ciągu uczącego, jednak byłoby potencjalnie najlepszym rozwiązaniem.



Rysunek 6.4: Podział obrazu po wyznaczeniu średniej współrzędnych granic w zbiorze.

Ze względu na to, że ustawiona jest sztywna granica 4 pikseli, obraz wejściowy powinien mieć wysokość 76 pikseli lub zbliżoną, żeby zachować proporcje liter. Gdyby obraz był za mały to może zdarzyć się, że wąskie litery, takie jak "i", nie zostaną w ogóle wykryte, ponieważ znak ten nie zablokował żadnej próby ustanowienia granicy, a co za tym idzie, granice te będą należały do jednego zbioru i z nich wszystkich zostanie obliczona tylko jedna linia wzdłuż, której obraz zostanie podzielony. Granica 4 pikseli została wyznaczona eksperymentalnie.

Użyty w niniejszej pracy algorytm do segmentacji znaków jest wrażliwy, na wszelkie pochyłości, podkreślenia oraz inne zakłócenia. Czcionki pochyłe są niemożliwe do rozdzielenia, kiedy poprowadzenie linii między dwoma znakami wymaga pochylenia jej o więcej

niż 4 piksele. Zwiększenie jednak tej granicy skutkowałoby problemami z prawidłowym pocięciem obrazu wejściowego, gdyż nie mogłoby być ono pionowe, ponieważ potencjalnie przeciętoby oba znaki, uniemożliwiając ich prawidłowe rozpoznanie.

Podobnie napisy podkreślone ciągła linią lub taką linią przekreślone całkowicie uniemożliwiają działanie algorytmu i w takim przypadku potrzebne byłoby zastosowanie wyrafinowanych technik wstępnej obróbki danych lub metody rozpoznawania całych słów od razu, zamiast po jednym znaku.

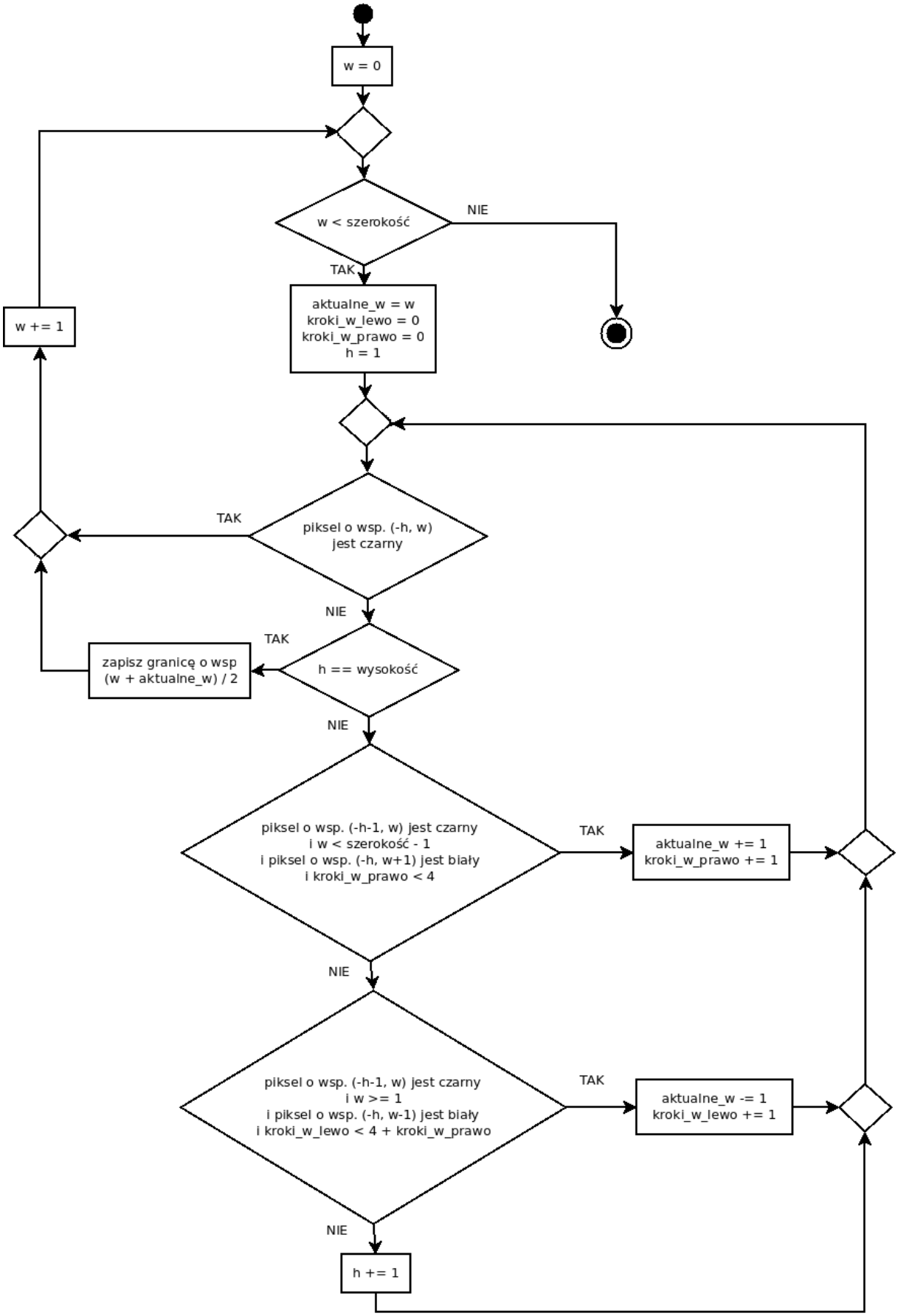
Po skończeniu dzielenia obrazu wejściowego na fragmenty zawierające poszczególne znaki, każdy z nich jest poddawany kolejnej obróbce przed podaniem na wejście konwolucyjnej sieci neuronowej. Uzyskane segmenty muszą mieć konkretny rozmiar, a najlepiej gdyby były również jak najbardziej podobne do obrazów w ciągu uczącym. Podczas pisania programu na potrzeby niniejszej pracy zauważono, że okno potrzebne do zawarcia całego słowa na zdjęciu sprawia, iż obraz, który jest podawany do algorytmu segmentacji zawiera dużo wolnej przestrzeni, zarówno na górze, jak i na dole. Spowodowane to jest tym, że niektóre litery takie, jak

$$g, j, p, q, y$$

sięgają nisko podczas gdy np.

$$b, d, f, h, k, l, t$$

sięgają wysoko. Rzutuje to również na wszystkie pozostałe litery, ponieważ w skutek tego są skalowane inaczej niż ma to miejsce w zestawie uczącym. Żeby temu przeciwdziałać, przyjęto, że żadna litera nie może mieć mieć nad sobą lub pod sobą więcej niż 5 pustych pikseli. Po usunięciu zbędnych pikseli obraz jest przeskalowywany tak, aby pasował do żądanej wysokości tj. 32 piksele, a następnie prawa i lewa strona jest uzupełniana o puste kolumny, aby również szerokość była równa 32 pikselom.



Rysunek 6.5: Diagram algorytmu wyznaczającego granice.

Rozdział 7

Rozpoznawanie liter i cyfr

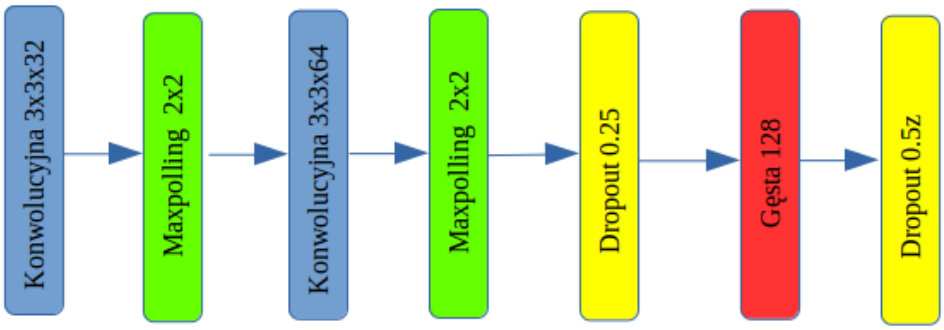
W niniejszej pracy została wykorzystana głęboka konwolucyjna sieć neuronowa zbudowana na podstawie sieci LeNet [3] z zastosowaniem dodatkowych metod polepszania klasyfikacji tj. dropout. Sieć ta rozpoznaje obrazy o rozmiarze 32x32 piksele, czarno-białe. Obrazy są klasyfikowane jako przynależące do jednej z 62 kategorii (10 cyfr, 26 liter małych, 26 liter dużych). Do stworzenia sieci oraz do jej nauczenia wykorzystany został framework Keras oraz język programowania Python.

Jako zestaw treningowy i testowy została użyta baza "The Chars74K dataset", która została stworzona na potrzeby pracy dotyczącej rozpoznawania liter, cyfr oraz znaków pisma kannada [1]. Użyte zostały tylko obrazy przedstawiające pismo drukowane angielskiego alfabetu. Każda klasa jest reprezentowana przez ok. 1000 przykładów. Obrazy w surowych danych mają różne wielkości, co uniemożliwia ich wykorzystanie, dlatego zostały przeskalowane do rozmiaru 32x32.

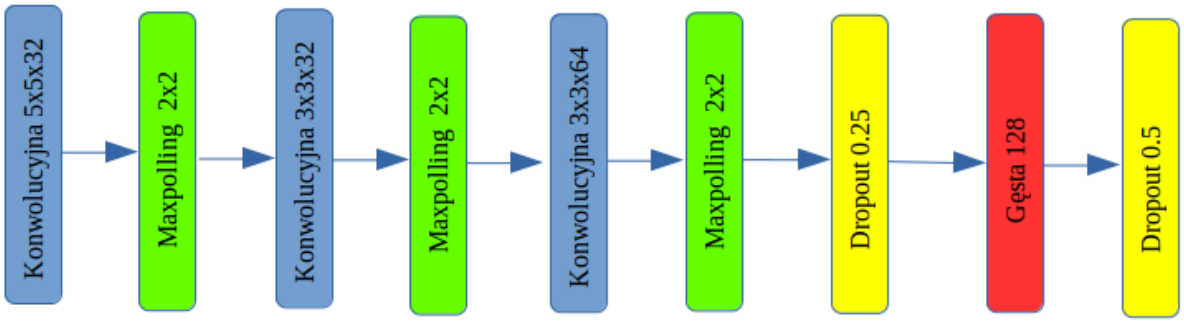
Rysunek 7.1: Przykładowe znaki z ciągu uczącego.

7.1 Wybór architektury sieci neuronowej

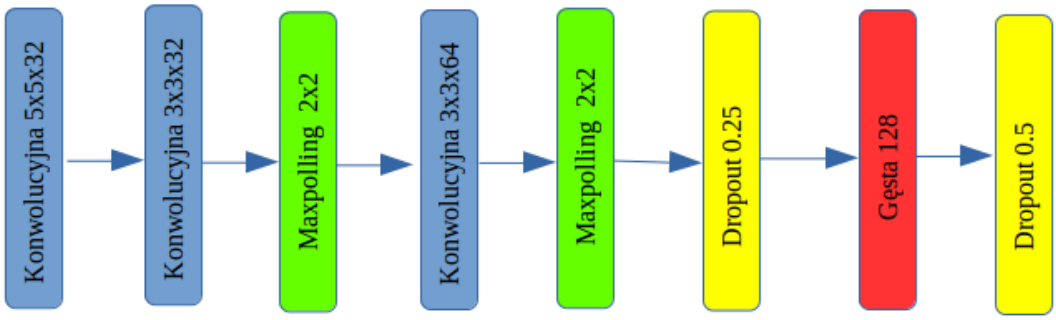
W celu określenia najlepszej architektury sieci na potrzeby projektu przetestowano 5 propozycji, które w sposób schematyczny przedstawiono na poniższych diagramach. Każda z poniższych architektur kończy się warstwą gęstą składającą się z 62 dwóch neuronów tak, aby liczba wyjść sieci zgadzała się z docelową liczbą klas.



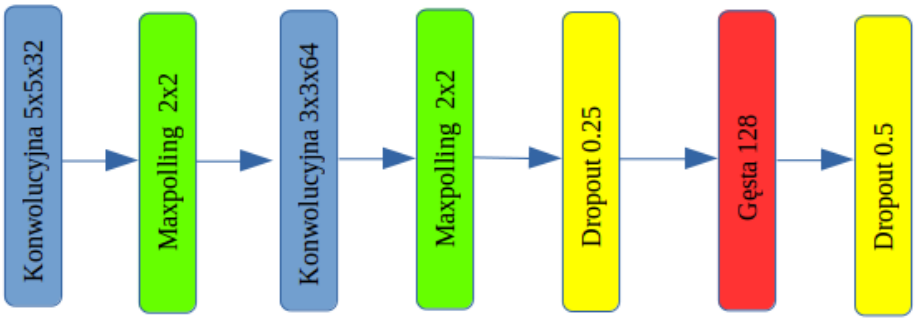
Rysunek 7.2: Architektura 1



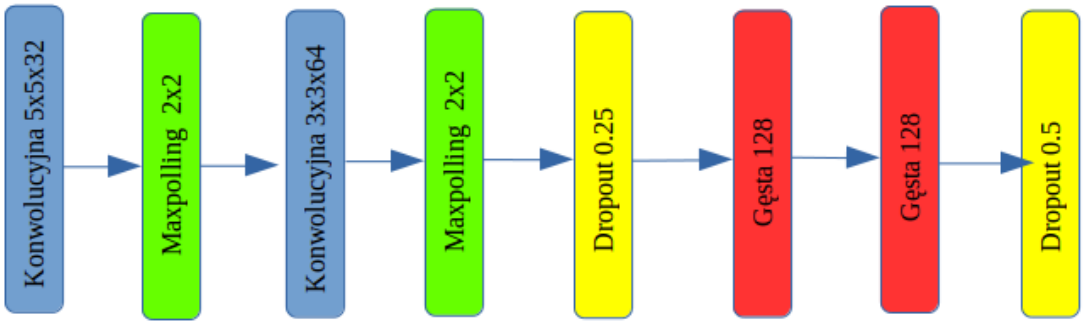
Rysunek 7.3: Architektura 2



Rysunek 7.4: Architektura 3



Rysunek 7.5: Architektura 4



Rysunek 7.6: Architektura 5

	Architektura 1		Architektura 2		Architektura 3		Architektura 4		Architektura 5	
	Skuteczność[%]		Skuteczność[%]		Skuteczność[%]		Skuteczność[%]		Skuteczność[%]	
liczba epok	nauka	walidacja	nauka	walidacja	nauka	walidacja	nauka	walidacja	nauka	walidacja
1	59,58	81,01	50,54	77,83	61,14	81,45	61,75	81,45	61,85	81,25
2	75,76	82,46	72,66	80,38	76,64	83,37	76,68	83,61	78,52	83,48
3	78,27	84,50	75,82	82,28	79,01	84,93	78,91	84,11	81,42	84,11
4	79,48	84,64	77,33	83,18	80,71	85,90	80,08	85,29	81,65	84,74
5	80,80	85,01	78,65	82,97	81,49	84,46	80,90	85,81	82,23	86,03

Tabela 7.1: Skuteczność rozpoznawania poszczególnych architektur.

Najlepsza okazała się architektura nr 5. Jako funkcja aktywacji została wykorzystana funkcja ReLU, która jest dana wzorem

$$f(x) = max(0, x).$$

Zaletą tej funkcji jest bardzo łatwe obliczanie jej pochodnej, która jest przedstawiona wzorem

$$f'(x) = \begin{cases} 1, & dla\ x > 0 \\ 0, & dla\ x \leqslant 0 \end{cases}.$$

Funkcja ReLU została wybrana ze względu na tempo uczenia, a w warstwach gęstych wykorzystana jest funkcja softmax, żeby można było interpretować ostateczny wynik jako prawdopodobieństwo, że obraz wejściowy przedstawia dany znak.

7.2 Wybór algorytmu optymalizacji

W celu określenia najlepszego algorytmu optymalizacji na potrzeby niniejszej pracy przetestowano trzy algorytmy:

1. Adagrad,
2. Adam,
3. Adam z momentum Nesterova.

Przetestowane zostały tylko metody pierwszego rzędu ze względu na to, że metody drugiego rzędu są mało skuteczne przy dużych zestawach danych. Dużych ciągów uczących

	NAdam		Adam		Adagrad	
	Skuteczność[%]		Skuteczność[%]		Skuteczność[%]	
liczba epok	nauka	walidacja	nauka	walidacja	nauka	walidacja
1	61,44	80,56	51,56	79,55	54,09	79,62
2	76,43	83,51	73,26	82,05	72,19	81,78
3	78,58	84,27	76,36	83,70	75,19	83,43
4	80,34	85,14	77,99	83,66	76,54	82,80
5	81,05	85,96	79,30	85,28	77,12	83,27

Tabela 7.2: Wyniki dla poszczególnych algorytmów.

nie da się zmieścić do pamięci RAM komputera, więc stosuje się uczenie porcjami (ang. "batch learning") tzn. wczytywana jest tylko część danych i na nich liczony jest błąd dla sieci, a następnie wczytuje się inną część danych i powtarza te kroki. Ze względu na to, że sieć nie ma dostępu do całości danych na raz, metody drugiego rzędu popełniają duże błędy na poszczególnych porcjach, co spowalnia uczenie zamiast je przyspieszać.

Wszystkie pomiary przedstawione w tabeli 7.2 zostały wykonane na jednakowo podzielonych danych, żeby wyeliminować losowość związaną z rozmieszczeniem przykładów w zestawie uczącym i testującym.

Ze względu na najszybsze uczenie został wybrany algorytm Adam z momentum Nesterova (NAdam).

7.3 Działanie sieci

Konwolucyjna sieć neuronowa była uczona przez 10 epok. Po tym czasie skuteczność na zestawie walidacyjnym wyniosła 87,23%. Jest to niska skuteczność, jak dla problemu rozpoznawania czarno-białych liter, dlatego przeprowadzono badania mające na celu ustalić przyczynę tak niskiego wyniku. Test polegał na przygotowaniu zestawu testowego składającego się z 10 tysięcy przykładów wybranych z ciągu uczącego, tak aby każdy znak pojawiał się w tym zestawie podobną ilość razy. Następnie obrazy te zostały podane na wejście wytrenowanej konwolucyjnej sieci neuronowej oraz rozpoznane. Wektor decyzji dla każdego obrazu składał się z 62-óch liczb, z których każda określa prawdopodobieństwo, że rozpoznany znak należy do danej kategorii. Za decyzję uznano kategorię, której zostało przyporządkowane największe prawdopodobieństwo. Następnie wektor wyjść został porównany z żądanym wyjściem. Każde wystąpienie znaku w wektorze żądanych wyjść zostało zliczone, tak samo jak każdy raz kiedy sieć neuronowa błędnie rozpoznała obraz. Wynikiem tego testu jest częstotliwość błędnej klasyfikacji dla każdego znaku, która została zaprezentowana na rysunku 7.7

Z analizy wykresu 7.7 widać, że często błędnie klasyfikowanie były następujące znaki:

$$0, C, S, V, W, X, Z, l, o, s, v, w, x, z.$$

Można tutaj zauważyć oczywistą właściwość tj. sieć ma problemy z poprawnym rozpoznaniem znaków, które są podobne do innych. W przypadku takich grup jak '0' (zero) oraz 'O' nie da się tego poprawić i jest to zwyczajne zachowanie, ponieważ nawet ludzie

mają problemy z odróżnieniem tych dwóch znaków bez znajomości kontekstu, a konwolucyjne sieci neuronowe budowane są, aby imitować ludzki sposób rozpoznawania liter, więc logicznym jest, że dziedziczą po nas pewne ograniczenia. W innych grupach jednak, biorąc pod uwagę konkretne zastosowanie sieci w niniejszej pracy, można wyróżnić grupy, w których przedstawione błędy są iluzoryczne. Ze względu na to, że w następnym etapie, tj. etapie korekty słownikowej, wszystkie wielkie litery zmieniane są na małe, nie ma potrzeby precyzyjnego rozróżniania grup liter:

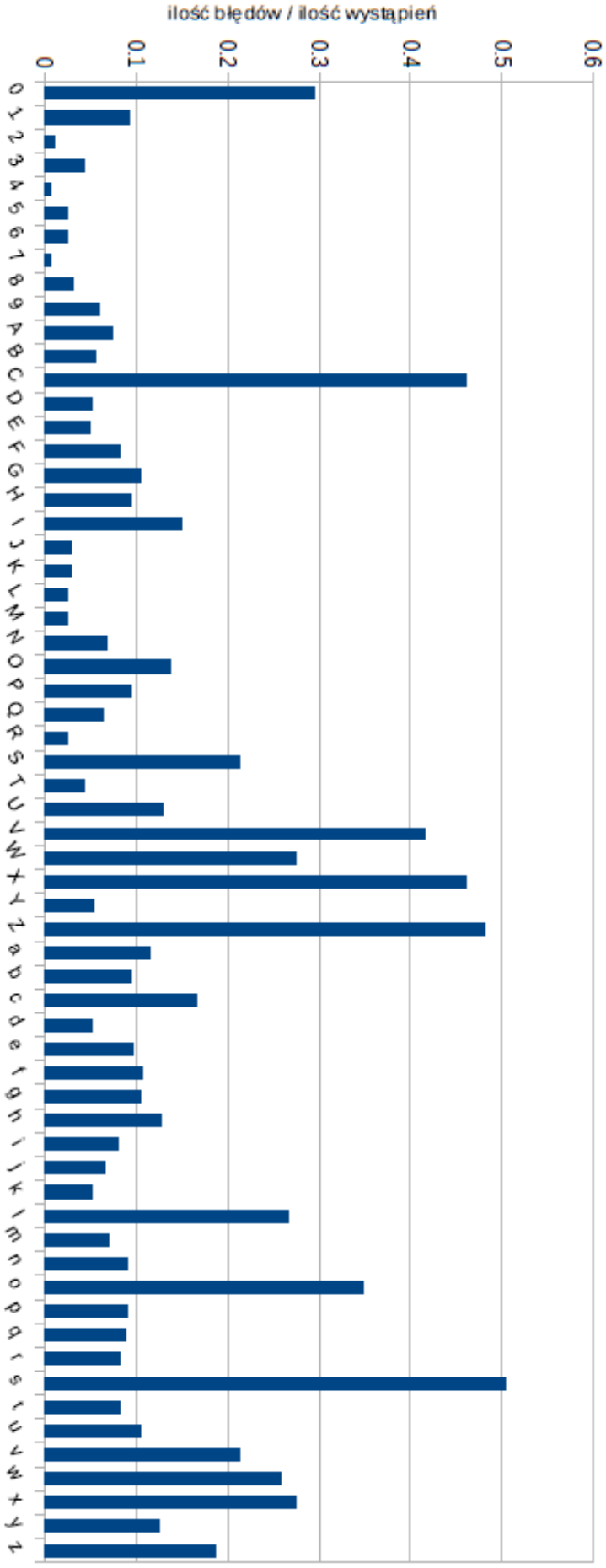
- c, C,
- o, O,
- s, S,
- v, V,
- w, W,
- x, X,
- z, Z.

Następnie przeprowadzono test aby sprawdzić jaka jest skuteczność sieci, zakładając, że powyższe grupy liter nie będą rozróżniane. Sieć została od początku nauczona na tym samym zestawie uczącym, ale ze zmienionym wektorem żądanych wyjść. Została ona wytrenowana, aby rozpoznawać tylko 55 kategorii, ponieważ ze standardowych 62 kategorii zostały te, które odpowiadały następującym małym literom:

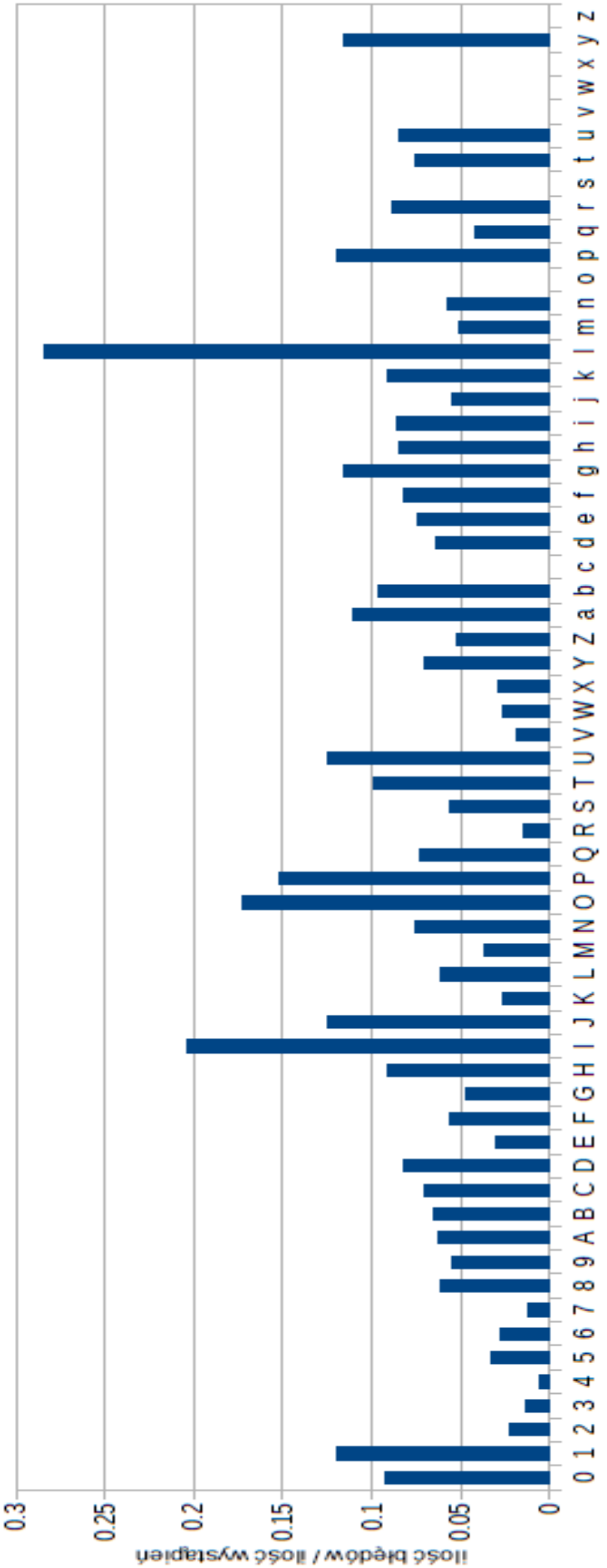
$$c, o, s, v, w, x, z.$$

Na rysunku 7.8 zostały przedstawione częstotliwości błędnej klasyfikacji dla poszczególnych znaków z wyłączeniem powyższych 7 kategorii.

Z wykresu 7.8 wynika, że wcześniej podana skuteczność klasyfikowania znaków przez konwolucyjną sieć neuronową jest zaniżona, jeśli spojrzeć na nią przez pryzmat wymagań jej stawianych w tym przypadku. Bliższym prawdy jest wynik 95,34%, który został otrzymany przy testowaniu sieci z 55 kategoriami. Skuteczność ta nie jest bardzo duża biorąc pod uwagę stopień skomplikowania problemu, ale jeśli spojrzeć na zakłócenia wynikające z etapów, które dane muszą przejść przed dotarciem do etapu rozpoznawania przez sieć neuronową, to taki wynik jest zadowalający. Już kiedy użytkownik robi zdjęcie tekstowi, który ma być rozpoznany, wszystkie odchylenia tekstu od właściwej pozycji mogą uniemożliwić segmentację tekstu na odrębne litery, czego potem już się nie da skorygować. Również czcionka, kolor tekstu i tła mają znaczenie, gdyż na etapie wstępnego przetwarzania obrazów mogą uniemożliwić prawidłowe wykrycie i oddzielenie od siebie znaków. W związku z powyższym aktualna skuteczność klasyfikowania przez sieć została uznana za wystarczającą, ponieważ ewentualny wysiłek włożony w jej polepszenie byłby lepiej spożytkowany na rozwijaniu innych etapów przetwarzania.



Rysunek 7.7: Częstotliwość błędnej klasyfikacji poszczególnych znaków dla 62 kategorii.



Rysunek 7.8: Częstotliwość błędnej klasyfikacji poszczególnych znaków dla 55 kategorii.

Rozdział 8

Korekta słownikowa

Część projektu odpowiedzialna za poprawianie tekstu metodą słownikową została opracowana na podstawie pomysły Peter'a Norvig'a na licencji MIT. Poniżej, w formie pseudokodu, przedstawiona została logika odpowiedzialna za tworzenie wszystkich pojedynczych korekcji dla danego słowa.

$words \leftarrow replaceDigitsWithSimilarLetters(inputWord)$
for all word in words **do**
  $proposals \leftarrow proposals + allInserts(word)$
  $proposals \leftarrow proposals + allTranspositions(word)$
  $proposals \leftarrow proposals + allDeletes(word)$
  $proposals \leftarrow proposals + allReplaces(word)$
end for
return propozycje

Funkcja **replaceDigitsWithSimilarLetters()** zwraca listę słów stworzonych przez zastąpienie w słowie wejściowym cyfr podobnymi do nich literami. Dzieje się to przede wszystkim w celu poprawiania błędów popełnianych przez konwolucyjną sieć neuronową podczas rozpoznawania tekstu. Ze względu na podobieństwo znaków sieć może np. błędnie rozpoznać literę 'g' jako cyfrę '9'. Podczas działania programu mogą mieć miejsce następujące zastąpienia:

- 2 na z,
- 3 na b,
- 4 na a,
- 5 na s,
- 7 na j,
- 8 na b,
- 1 na i,
- 1 na l,
- 6 na b,
- 6 na h,
- 9 na g,

- 9 na q,
- 0 na o,
- 0 na q.

Następnie dla każdego ze słów zwróconych przez funkcję **replaceDigitsWithSimilarLetters()** tworzony jest zestaw propozycji obejmujących następujące zmiany:

- Dodawanie - wynikiem jest zestaw propozycji, w których każda z liter występujących w angielskim alfabecie tj.

  $$a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z$$

  jest wstawiana na początku i na końcu wyrazu wejściowego, także między każdymi dwiema jego literami. W jednej propozycji tylko jedna litera jest dodawana w jednym miejscu.

- Transpozycja - wynikiem jest zestaw propozycji, w których każde dwie sąsiadujące ze sobą litery w słowie wejściowym są zamieniane miejscami tak, że transpozycje dla słowa "kot" to "okt" oraz "kto".

- Usuwanie - wynikiem jest zestaw propozycji, w których pojedyncze litery zostały usunięte ze słowa wejściowego (np. usunięcia dla słowa "kot" to "ot", "kt" oraz "ko").

- Zastępowanie - wynikiem jest zestaw propozycji, w których pojedyncze litery zostały zastąpione literami z alfabetu angielskiego. W jednej propozycji tylko jedna litera została zastąpiona przez jedną inną (np. dla słowa kot zostaną stworzone zastąpienia "aot", "bot", "cot" itd.).

Ze względu na to, że dopuszczane są dwie zmiany w słowie, dla wszystkich korekcji zwróconych dla słowa wejściowego jest tworzony kolejny zestaw propozycji. Następnie decyzją algorytmu zostaje propozycja, która najwięcej razy pojawia się w pliku wykorzystywanym jako baza danych.

Wykorzystany algorytm jest dość prosty i naiwny, dlatego zwraca jako wynik bardzo dużo propozycji. Ich zestaw odległych od słowa początkowego jedną zmianę ma rozmiar $54n + 25$ dla słowa o rozmiarze n, które nie zawiera żadnych cyfr. W tym:

- n dodań,
- $n - 1$ transpozycji,
- $26n + 26$ usunięć,
- $26n$ zastąpień.

Nie jest również brane pod uwagę prawdopodobieństwo wystąpienia danego wyrazu w kontekście słów sąsiadujących z nim, co zmniejsza skuteczność algorytmu, który w związku z tym osiąga 68% trafności [6].

Baza używana przez program została stworzona z elementów książek uzyskanych z projektu Gutenberg [7] oraz lista najczęściej występujących wyrazów w języku angielskim z serwisu Wiktionary [8] oraz British National Corpus [4]. Z tego względu program ten powinien być wykorzystywany do rozpoznawania tekstu napisanego w języku angielskim.

Zmienienie samej bazy danych na inny język nie przeniesie całej funkcjonalności niniejszej pracy z powodu różnych reguł odmian słów w różnych językach np. deklinacja w języku polskim. Do takich zastosowań niezbędne byłoby zmodyfikowanie programu przez dodanie do niego logiki obsługującej te zasady i być może również sprawdzającej prawdopodobieństwo wystąpienia wyrazu nie dla danego wyrazu, ale dla jego nieodmienionej formy np. prawdopodobieństwo wystąpienia słowa "kotów" będzie określane jako prawdopodobieństwo wystąpienia wyrazu "kot".

Rozdział 9

Opis implementacji

Program został napisany przy użyciu języka skryptowego Python w wersji 3.4 i został podzielony na pięć modułów:

1. main.py - moduł główny wywołujący pozostałe w odpowiedniej kolejności i zapewniający komunikację między nimi,
2. letter_segmentation.py - moduł odpowiedzialny za wstępne przetworzenie danych oraz podział obrazu na segmenty zawierające po jednym znaku,
3. learning.py - moduł tworzący oraz uczący konwolucyjną sieć neuronową,
4. classification.py - moduł używający nauczonej konwolucyjnej sieci neuronowej do rozpoznawania segmentów,
5. spellingCorrector.py - moduł odpowiedzialny za ewentualne poprawienie rozpoznanego tekstu.

W dalszej części rozdziału została omówiona implementacja tych modułów.

9.1 Moduł main.py

Implementacja tego modułu jest stosunkowo prosta i jej główna część znajduje się na poniższym listingu.

```
import letter_segmentation as ls
import classification
import spellingCorrector as sc

def main(image_path, folder_path):
    ls.segment(image_path, folder_path)
    letters = classification.run(folder_path)
    print('Word before correction: ', letters)
    word = sc.correct(letters)
    print('Word after correction: ', word)
```

Funkcja **main()** przyjmuje 2 argumenty:

- image_path - ścieżka do zdjęcia, które ma zostać przetworzone,
- folder_path - ścieżka do katalogu, w którym będą zapisane segmenty, na które zostanie podzielone zdjęcie.

Poniżej przedstawione jest przykładowe wywołanie modułu main.py oraz efekty jego działania.

```
$ python main.py './words/parameter.png' './cut_letters'
Using Theano backend.
Images read.
Word before correction:  paTameteT
Word after correction:  parameter
```

Jak widać na powyższym listingu, podczas działania programu został użyty moduł Theano, który został opisany w podrozdziale 9.3.

9.2 Moduł letter_segmentation.py

Manipulacja zdjęciem została przeprowadzona z użyciem dwóch bibliotek:

1. Python Imaging Library (PIL),
2. SciPy.

Pierwsza z nich zapewnia funkcje odpowiedzialne za m.in. skalowanie obrazów, podczas gdy druga została wykorzystana przy filtrowaniu oraz progowaniu zdjęcia. Dzięki modułowi SciPy można też wczytać obraz binarny jako dwuwymiarową tablicę liczb, gdzie dana o współrzędnych (0, 0) to prawy górny piksel.

Główną funkcją tego modułu jest funkcja **segment()**, która przyjmuje te same argumenty i w tej samej kolejności, co funkcja **main()** w module main.py. Na poniższym listingu przedstawiona została funkcja **find_cut_lines()**, która jest odpowiedzialna za wyznaczanie granic między znakami.

```
max_steps_in_direction = 4

def find_cut_lines(image_array):
    height, width = image_array.shape
    borders = []

    for w in range(width):
        current_w = w
        steps_right = 0
        steps_left = 0
        h = 1
        while h < (height + 1): # for minus indexing
            if image_array[-h, current_w] == 0:
                break
            if h == height:
                borders.append((w + current_w) / 2.0)
```

```
                break
            elif ((image_array[-(h + 1), current_w] == 0) and
                current_w < width - 1 and
                image_array[-h, current_w + 1] != 0 and
                steps_right < max_steps_in_direction):
                # If step right is possible, do one.
                current_w += 1
                steps_right += 1
            elif ((image_array[-(h + 1), current_w] == 0) and
                current_w >= 1 and
                image_array[-h, current_w - 1] != 0 and
                steps_left < max_steps_in_direction + steps_right):
                # If step left is possible, do one.
                current_w -= 1
                steps_left += 1
            else:
                h += 1

    return borders_to_cut_lines(borders)
```

Funkcja **borders_to_cut_lines()** wylicza linie przecięcia na podstawie wyznaczonych granic, tak jak to omówiono w rozdziale 6.

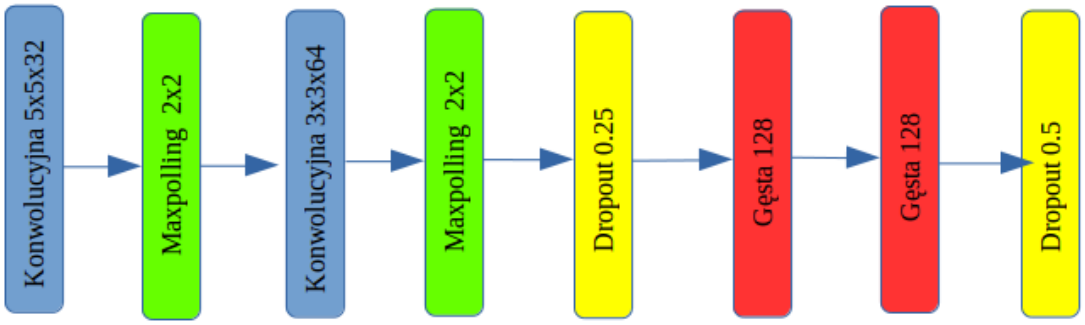
9.3 Moduł learning.py

Do stworzenia i nauczenia konwolucyjnej sieci neuronowej została użyta biblioteka Keras. Pozwala ona w prosty sposób składać głębokie sieci neuronowe z następujących po sobie warstw. Udostępnia również konfigurowalne algorytmy ich uczenia. Do efektywnego przetwarzania obliczeń Keras wykorzystuje bibliotekę Theano. Pozwala ona na szybkie przeprowadzanie działań na dużych macierzach. Umożliwia również wykorzystanie karty graficznej GPU, żeby zrównoleglić obliczenia i przyspieszyć uczenie sieci.

Na poniższym listingu został przedstawiony etap budowy sieci.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu'
                 ,input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.mean_squared_error,
              optimizer=keras.optimizers.Nadam(),
```



Rysunek 9.1: Architektura 5

```
metrics=['accuracy'])
```

Sieć została stworzona zgodnie z rysunkiem 9.1.

Jako algorytm optymalizujący zostosowany został algorytm Adam z momentum Nesterova, który jest opisany w rozdziale 5. Kryterium, według którego oceniana jest jakość danego rozwiązania, została średniokwadratowa funkcja strat, dana wzorem

$$L_i = (f(x) - y_i)^2,$$

gdzie L_i - wektor wartości funkcji strat dla i-tego przykładu,
f(x) - wektor odpowiedzi, które zwróciła sieć neuronowa,
y_i - wektor żądanych odpowiedzi.
Moduł learning.py został użyty przed innymi modułami, a jako wynik daje plik, w którym jest zapisana architektura sieci oraz jej wagi. Etapy stworzenia i nauczenia konwolucyjnej sieci neuronowej zostały opisane w rozdziale 7.

9.4 Moduł classification.py

Moduł ten wczytuje nauczoną konwolucyjną sieć neuronową z pliku wygenerowanego po zakończonym etapie uczenia przez moduł learning.py. Następnie używa jej, aby obliczyć wektor wyjściowy, czyli wektor prawdopodobieństw, które określają to, jak bardzo algorytm jest przekonany, że dany obraz to znak odpowiadający danej kategorii. Jako decyzja sieci jest wybierana kategoria, dla której prawdopodobieństwo jest największe. Etap klasyfikacji jest wykonywany od razu dla wszystkich segmentów, ponieważ funkcja klasyfikująca przyjmuje ich tablicę.

W następnym kroku litera odpowiadająca wybranej kategorii jest dodawana do łańcucha znaków, który następnie jest zwracany jako wynik działania funkcji **run()** modułu classification.py.

```
model = load_model('cnn.h5')
y_predicted = model.predict(images)
label_dict = read_label_dict()
output_letters = ""
for y_pred in y_predicted:
    category_pred = np.argmax(y_pred)
    output_letters = output_letters + label_dict[category_pred]
return output_letters
```

Powyżej znajduje się listing, na którym przedstawiona została klasyfikacja oraz budowanie wyjściowego łańcucha.

9.5 Moduł spellingCorrector.py

Główną funkcją tego modułu jest funkcja **correct()**, która przyjmuje łańcuch znaków do poprawy, a zwraca łańcuch z najbardziej prawdopodobną poprawną korekcją. Na poniższym listingu została przedstawiona logika odpowiadająca za tworzenie wszystkich wariacji słowa wejściowego, które się od niego różnią o dokładnie jedną zmianę.

```
def editOnce(word):
    "All edits that are one edit away from `word`."
    allProposals = set()
    for oneWord in replaceDigitsWithSimilarLetters(word):
        splits = getSplits(oneWord)

        allProposals.update(getDeletes(splits)
                            + getTransposes(splits)
                            + getReplaces(splits)
                            + getInserts(splits))
    return allProposals
```

Dopuszczane są dwie zmiany w słowie, dlatego funkcja **editOnce()** z powyższego listingu jest wywoływana raz na słowie wejściowym, a następnie kolejny raz na każdym ze słów, które zostały zwrócone w pierwszym kroku. Jako decyzja algorytmu zostaje przedstawiona ta propozycja, która najczęściej pojawia się w bazie danych.

Działanie korekcji słownikowej zostało omówione w rozdziale 8.

Rozdział 10

Podsumowanie

Celem niniejszej pracy inżynierskiej było stworzenie programu zdolnego rozpoznać tekst widoczny na zdjęciu za pomocą konwolucyjnej sieci neuronowej oraz wykorzystanie korekcji słownikowej w celu poprawienia skuteczności.

Wstępne przetwarzanie danych, tj. odfiltrowanie szumów oraz konwersja danych wejściowych na obraz binarny, zostało osiągnięte za pomocą filtru Gaussa oraz progowania z progiem globalnym. Jak zdjęcie wejściowe jest zgodne z założeniami opisanymi w rozdziale szóstym, tak długo metody te są wystarczające, żeby osiągnąć założone cele.

Algorytm segmentujący został zaprojektowany oraz zaimplementowany, jednak, ze względu na swoją prostotę, jest bardzo wrażliwy na sytuacje, kiedy sąsiadujące ze sobą znaki są w jakiś sposób połączone. Może to być skutek zakłóceń albo wyglądu czcionki. Jeśli tekst jest przekreślony lub podkreślony, to uniemożliwia to odpowiednie podzielenie znaków. Do takich samych problemów może dojść, jeśli tekst na obrazie wejściowym jest napisany czcionką pochyłą, ponieważ algorytm nie znajdzie w takim przypadku pionowej linii rozdzielające dane dwa znaki. Sposobem na poprawne rozpoznawaniu liter, które często są ze sobą łączone np. "VW" mogłoby być dodanie do sieci neuronowej takiej kategorii, która odpowiadałaby dwóm literom.

Rozpoznawanie znaków jest przeprowadzane przy użyciu konwolucyjnej sieci neuronowej na podstawie sieci LeNet-5 [3]. Została ona nauczona wykorzystując dane z bazy "The Chars74K dataset", który zawiera m.in. obrazy przedstawiające litery alfabetu angielskiego. Ostateczna skuteczność klasyfikacji tej sieci to ok. 95%

Algorytm słownikowej korekty tekstu został napisany na podstawie implementacji Peter'a Norvig'a na licencji MIT. Dodatkową możliwością tego algorytmu jest zamienianie cyfr na podobne do nich litery w celu poprawienia jakości rozpoznawania. Ostateczna jakość korekcji tej implementacji to ok. 68%. Możliwym sposobem na poprawienie tej skuteczności mogłoby być przyporządkowywanie większego prawdopodobieństwa tym korekcją, które mają w sobie mniej zmian względem słowa początkowego.

Bibliografia

[1] T. E. de Campos, B. R. Babu, M. Varma *"Character recognition in natural images"* 2009.

[2] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov *"Dropout: a simple way to prevent neural networks from overfitting"* 2014.

[3] Yann LeCun, Leon Bottou, Yoshua Bengio, Patrick Haffner *"Gradient-based learning applied to document recognition"*. 1998.

[4] *"British National Corpus database and word frequency lists"*. http://www.kilgarriff.co.uk/bnc-readme.html (dostęp 7.12.2017r.)

[5] Andrej Karpathy *"CS231n Convolutional Neural Networks for Visual Recognition"*. https://cs231n.github.io/neural-networks-3/#sgd (dostęp 19.11.2017r.)

[6] Peter Norvig *"How to Write a Spelling Corrector"*. http://norvig.com/spell-correct.html (dostęp 19.11.2017r.)

[7] *"Projekt Gutenberg"*. http://www.gutenberg.org/wiki/Main_Page (dostęp 7.12.2017r7r.)

[8] *"Wiktionary: Frequency lists"*. https://en.wiktionary.org/wiki/Wiktionary:Frequency_lists (dostęp 7.12.2017rr.)

Spis rysunków