UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ   
W LUBLINIE

Wydział Matematyki, Fizyki i Informatyki

**Spis treści**

[**Wstęp**](#_heading=h.gjdgxs) **5**

[**Rozdział 1. Wprowadzenie do optycznego rozpoznawania znaków**](#_heading=h.30j0zll)

**Rozdział 2. Sieci neuronowe 11**

2.1 Sieci neuronowe

2.2 Architektura Unet

2.3 sieci rekurencyjne(LSTM i architektura crnn) [11](#_heading=h.26in1rg)

**Rozdział 3. Implementacja/trening sieci do segmentacji 12**

[3.1. Zbiór danych 12](#_heading=h.35nkun2)

[3.2. Wstępne przetwarzanie - preprocessing 12](#_heading=h.1ksv4uv)

[3.3.](#_heading=h.44sinio) Szczegóły implementacyjne

**Rozdział 4. Implementacja/trening crnn i lstm 12**

4[.1.](#_heading=h.35nkun2)

**Rozdział 5. Eksperymenty, testy, rezultaty, wnioski 12**

5[.1.](#_heading=h.35nkun2)  [12](#_heading=h.44sinio)

**Podsumowanie 13**

**Bibliografia 14**

# Wstęp

Wraz z rozwojem technologicznym przyspieszają procesy cyfryzacji i automatyzacji. Szybka wymiana informacji oraz przechowywanie dokumentów w sposób elektroniczny są ogólnie powszechne na lotniskach, w bibliotekach, w firmach. Codziennie wykonuje się skany lub zdjęcia dokumentów, książek oraz innych tekstów w celu ich utrwalenia w pamięci. Dokumenty utrwalone w pamięci mogą być kopiowane oraz przesyłane, co daje im znaczną przewagę nad ich papierową formą. Automatycznie wyodrębnianie tekstu z grafiki podczas dalszego przetwarzania obrazu umożliwia automatyczne uzupełnianie formularzy wyodrębnionym tekstem, oraz kompresję zapisanych danych w lepszej jakości poprzez zmianę formatu dokumentu graficznego na tekstowy. Takie rozwiązanie zapewnia oszczędność czasu oraz nakładów finansowych. Jednym z możliwych rozwiązań automatycznego wyodrębniania tekstu jest zastosowanie sztucznej inteligencji. Największą trudnością w zastosowaniu takiego rozwiązania jest trudność w uzyskaniu odpowiedniego zbioru danych do nauki sieci.

skuteczności modeli rozpoznawania tekstu trenowanych na syntetycznych danych

Celem pracy jest sprawdzenie jakości sieci neuronowych trenowanych na syntetycznie wygenerowanych danych uczących w odniesieniu do danych rzeczywistych, oraz implementacja silnika OCR za pomocą własnoręcznie wytrenowanych sieci służących do segmentacji oraz klasyfikacji znaków.

Skrypty zostały zaimplementowane w języku Python przy pomocy biblioteki Tensorflow i nakładki Keras napisanej w Pythonie. Temat został wybrany, ponieważ obecnie sztuczna inteligencja jest ciągle rozwijającą się nauką, która może znacznie przyczynić się do postępu technologicznego. Jako bazę sprzętową wykorzystałem komputer wyposażony w kartę graficzną Nvidia GeForce RTX 3060 wspierającą bibliotekę NVIDIA CUDA Deep Neural Network (cuDNN).

Zbiór technologii, w której został zaimplementowany silnik OCR został wybrany ze względu na dużą popularność w dziedzinie uczenia maszynowego oraz uczenia głębokiego, ogólną dostępność i łatwą przenośność.

Python jest popularnym językiem skryptowym w sztucznej inteligencji. Posiada wiele gotowych do zaimportowania bibliotek oraz przykładów i znaczną dokumentację.

W pierwszym rozdziale zawarte jest wprowadzenie do optycznego rozpoznawania znaków. Drugi rozdział poświęcony sieciom neuronowym oraz wykorzystanym architekturom. Trzecia część pracy opisuje treningi sieci wykorzystanych do segmentacji obrazu. Czwarty część pracy zawiera opis treningu sieci wykorzystanej do rozpoznawania znaków. Ostatni rozdział przedstawia uzyskane wyniki i porównanie ich z silnikiem open source Tesseract.

# Wprowadzenie do optycznego rozpoznawania znaków

## Definicja

Optyczne rozpoznawanie znaków(Optical character recognition) to dziedzina badań w zakresie rozpoznawania wzorców, sztucznej inteligencji i widzenia komputerowego. Jest to proces polegający na przetwarzaniu obrazu przez maszynę lub system w celu uzyskania nieprzetworzonego tekstu bądź innego ustrukturyzowanego wydruku.

Jest szeroko wykorzystywana jako forma wprowadzania znaków z drukowanych dokumentów papierowych takich jak paszporty, faktury, wyciągi bankowe, paragony, dokumenty pocztowe, wydruków danych statycznych lub jakiejkolwiek odpowiedniej dokumentacji. jest to powszechna metoda digitalizacji wydrukowanych tekstów, aby można je było elektronicznie edytować, przeszukiwać, przechowywać w bardziej zwarty sposób, wyświetlać online i wykorzystywać w procesach maszynowych, takich jak tłumaczenie maszynowe, zamiana tekstu na mowę, kluczowe dane i eksploracja tekstu.

Systemy OCR działają coraz lepiej, chociaż nadal nie działają dobrze na zdjęciach obejmujących rzeczywiste scenariusze. Należy wziąć pod uwagę brak dużych zestawów danych OCR z tekstem opatrzonym adnotacjami. Tekst wyodrębniony przez systemy OCR sam w sobie nic nie znaczy, dopóki nie zostanie użyty do rozwiązania zadania, które polega na użyciu tekstu sceny.

Niedostępność adnotacji tekstowych z adnotacjami dla obrazów ze świata rzeczywistego i brak informacji zwrotnych do systemu OCR w celu poprawy wykrywania lub ekstrakcji w oparciu o błędy w dalszej aplikacji, tj. brak kompleksowego szkolenia.

W rezultacie modele OCR wytrenowane na tych zestawach danych zwykle nie radzą sobie dobrze z zadaniami wykonywanymi w innych typach scen. Co więcej, istniejące zbiory danych zwykle zawierają małą liczbę słów na obraz, co czyni je mniej gęstymi, zróżnicowanymi i idealnymi do trenowania modeli OCR dla zadań zwykle charakteryzujących się dużą gęstością tekstu.

## Struktura silnika OCR

Złożenie silnik OCR w całość wymaga stworzenia potoku przetwarzania danych(pipeline). Dane wychodzące z jednego elementu będą przesłane na wejście następnego. Struktura potoku przetwarzania danych silnika OCR zazwyczaj składa się z dwóch głównych modułów. Pierwszym z nich jest **moduł wykrywania tekstu**. Jest on odpowiedzialny za segmentację obrazu. Segmentacja polega na nałożeniu pikselowej maski dla każdego z obiektów obecnych na obrazie. Klasyfikuje ona obraz pod kątem pikseli na różne obiekty. To umożliwia wycięcie odpowiednich obszarów obrazu zawierających tekst. Wycięte obszary tekstu poddaje się przetwarzaniu wstępnemu. Najpopularniejszymi procesami przetwarzania wstępnego w OCR są:

1. Skalowanie obrazu(rescaling)
2. Binaryzacja(binarization)
3. Odkrzywianie tekstu (skew correction)
4. Usuwanie szumów (noise removal)
5. Pogrubianie i zdrabnianie (Thinning and Skeletonization)

**Skalowanie obrazu** jest procesem polegającym na zmianie rozmiaru obrazu. W przypadku problemu jakim jest OCR przeskalowywanie wykonuje się proporcjonalnie, aby znaki nie utraciły swoich kształtów. Jeśli w obrazie istotne jest tło należy zastosować (padding).

**Binaryzacja** polega na konwersji kolorowego obrazu na obraz składający się wyłącznie z czarnych i białych pikseli. Zazwyczaj wykorzystywana jest wartość progowa(threshold) która decyduje o kolorze danego piksela. Jeśli wartość piksela jest mniejsza niż próg to piksel oznaczany jest jako czarny, w przeciwnym razie jako biały.

**Odkrzywianie** polega na rotacji pochylonego tekstu w sposób, aby był jak najbardziej czytelny, czyli linia tekstu tworzyła z dołem kartki 90 stopni.

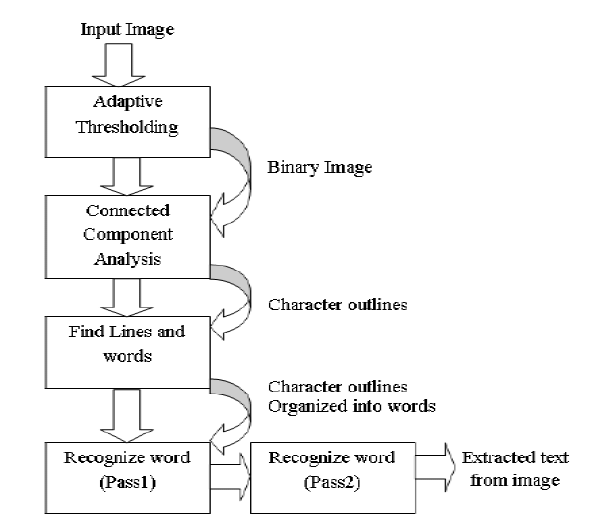
**Szum** jest losową zmianą koloru lub jasności między pikselami obrazu. Zmniejsza ona czytelność tekstu z obrazu.  **Usuwanie szumu** polega na filtrowaniu obrazu w celu poprawienia koloru lub jasności pikseli.

**Pogrubianie i zdrabnianie tekstu** polega na odpowiednio zwiększeniu lub zmniejszeniu grubości tekstu.

Następnie dane przekształcone w tensory przekazywane są do **modułu rozpoznawania tekstu**. Obecnie najpopularniejszym modułem rozpoznawania tekstu jest sieć neuronowa rekurencyjno-konwolucyjna(CRNN) wykorzustująca komórki LSTM. Otrzymanie odpowiednich wyników danych przetworzonych przez sieć należy je zdekodować.

Dodatkowo można wykonać postprocessing danych. Przetwarzanie danych finalnych umożliwiające zwiększenie dokładności poprzez wyłapanie błędów, albo umożliwiające zwrócenie danych w wymaganej formie.

Przetwarzanie końcowe optycznego rozpoznawania znaków (OCR) obejmuje etapy czyszczenia danych w przypadku dokumentów, które zostały zdigitalizowane, takich jak książka lub artykuł w gazecie. Jednym z etapów tego procesu jest identyfikacja i korekta błędów ortograficznych i gramatycznych powstałych w wyniku wad systemu OCR

[](https://www.researchgate.net/publication/277142272_Training_TESSERACT_Tool_for_Amazigh_OCR)

# Sieci neuronowe

## Wprowadzenie do sieci neuronowych

Sztuczna inteligencja jest dziedziną badającą inteligencję demonstrowaną przez maszyny. Odnosi się do idei dającej maszynom lub oprogramowaniu możliwość podejmowania własnych decyzji o predefiniowane reguły lub modele rozpoznawania wzorców. Idea rozpoznawania wzorców prowadzi do modeli **uczenia maszynowego**. Sieć neuronowa jest jednym z rodzajów uczenia maszynowego wykorzystującym uczenie głębokie.

Sieci neuronowe, poprawnie nazywane sztucznymi sieciami neuronowymi(artificial neural network) są systemem złożonym z wielu elementów powarzających się, które są oparte na działaniu neuronów. Elementy te nazywane są węzłami bądź komórkami(cell), gdzie każdy z nich odpowiada za proste obliczenia. Węzły są połączone kanałami komunikacyjnymi, które zwykle przenoszą dane liczbowe. Rozmieszczone są na wybranej liczbie warstw. Liczbę warstw oraz rozmieszczenie węzłów określa struktura sieci. Do każdego węzła przypisana jest waga. Odpowiada ona za

Sztuczne sieci neuronowe uczą się podczas procesu nazwanego treningiem. Trening polega na pokazywaniu sieci ogromnej liczby przykładów uczących w celu dopasowania wag.

Sieć należy uczyć do momentu kiedy przestanie zmniejszać swoje val\_loss. Jeśli przerwie się naukę sieci wcześniej to taka sieć będzie niedouczona(underfitting). Odwrotnym problemem do niedouczenia jest przeuczenie(overfitting). Jest to sytuacja, gdy sieć zwiększa swoją dokładność na danych treningowych, ale nie zwiększa na danych walidacyjnych. Powstało wiele rozwiązań, które mają pomóc zapobiec przeuczeniu sieci. Najpopularniejsze rozwiązania to :

* Zmniejszenie pojemności sieci - usuwając warstwy lub zmniejszając liczbę elementów w warstwach ukrytych
* Zastosowanie regularyzacji(regularization) - co sprowadza się do dodania kosztu do funkcji straty dla dużych ciężarów
* Używanie warstw dropout - losowo usunął niektóre funkcje, ustawiając je na zero

Przyspieszenie treningu sieci zapewnia warstwa **batch normalization.** Technika uczenia bardzo głębokich sieci neuronowych, która standaryzuje dane wejściowe do warstwy dla każdej partii. Wpływa to na stabilizację procesu uczenia się i radykalne zmniejszenie liczby epok treningowych wymaganych do trenowania głębokich sieci.

**Callback** to obiekt, który może wykonywać akcje na różnych etapach uczenia (np. na początku lub na końcu epoki, przed lub po pojedynczej partii itp.).

**Optymalizatory** jest jednym z dwóch argumentów wymaganych do kompilacji modelu Keras. Można albo utworzyć instancję optymalizatora przed przekazaniem go do model.compile()programu , jak w powyższym przykładzie, albo przekazać go przez jego identyfikator ciągu. W tym drugim przypadku zostaną użyte domyślne parametry optymalizatora. Optymalizatorem wykorzystywanych w sieciach których użyłem jest Adam. Optymalizacja Adama to stochastyczna metoda opadania gradientu oparta na adaptacyjnej estymacji momentów pierwszego i drugiego rzędu.

**Kompilacja modelu** compile()

**trenowanie modelu** fit()

## Architektura U-net i sieci konwolucyjne

Do przeprowadzenia procesu segmentacji

**Głębokie konwolucyjne sieci neuronowe** są odpowiedzialne za wyodrębnianie sekwencji cech z każdego obrazu wejściowego. Zostały zaprojektowane z myślą o zadaniach związanych z rozpoznawaniem obrazu. Pierwotnie były stosowane do rozwiązania problemów związanych z rozpoznawaniem odręcznych cyfr. Sieci CNN są dość podobne do zwykłych sztucznych sieci neuronowych:

Składają się z neuronów o wagach, których wartości można nauczyć na podstawie danych.

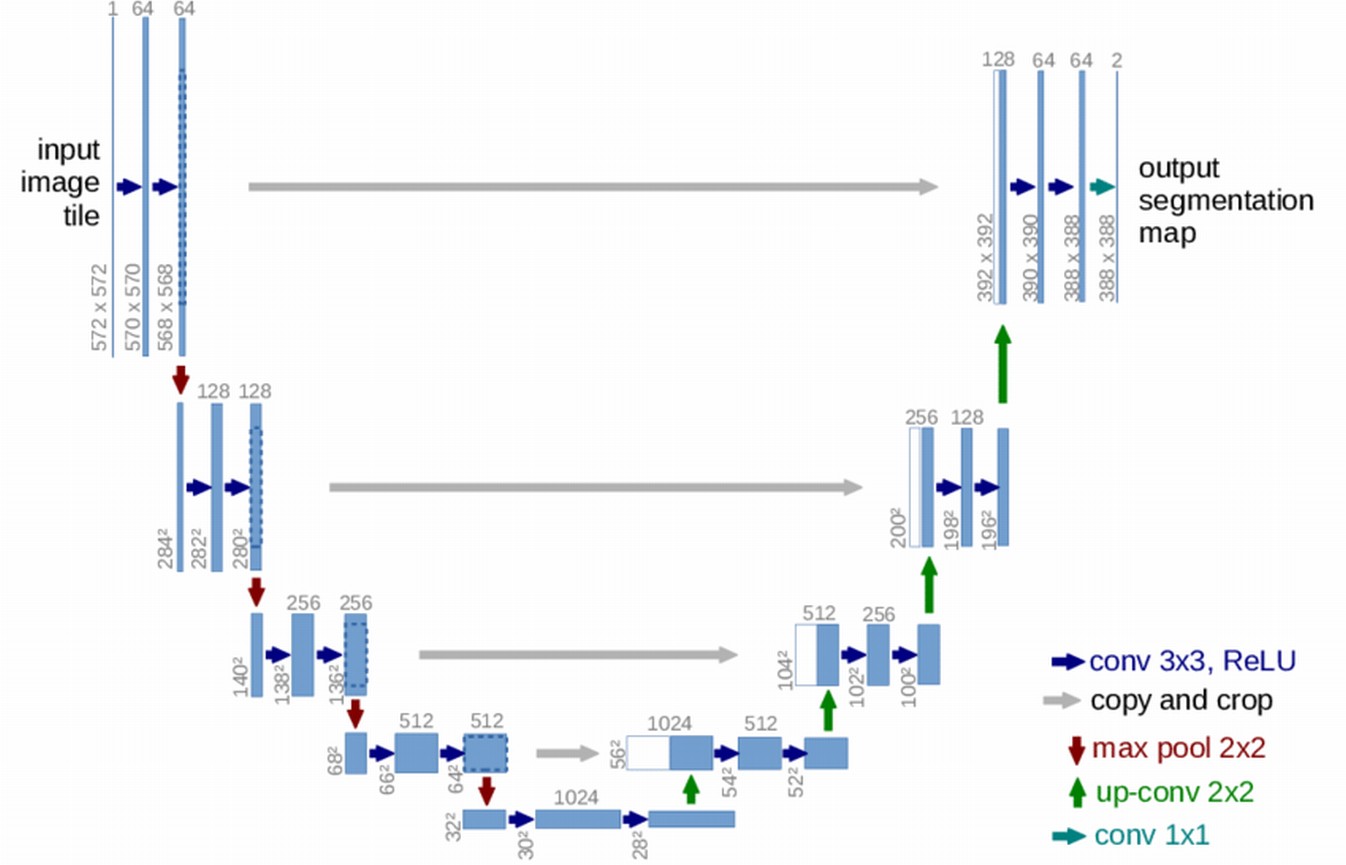
Każdy neuron otrzymuje dane wejściowe i wykonuje iloczyn skalarny.

Posiadają funkcję straty na ostatniej w pełni połączonej warstwie.

Tym co je odróżnia od zwykłych ANN jest posiadanie trzech głównych warstw. Są to **warstwa konwolucji**(convolution layer), **warstwa puli**(pooling layer) i warstwa w **pełni połączona**(FC).

W przypadku rzeczywistych danych obrazu CNN działają lepiej niż wielowarstwowe perceptrony (MLP).

Jako wejście przyjmują macierz, a nie prosty wektor numeryczny bez



struktury przestrzennej. Nie tracą informacji o przestrzennym rozmieszczeniu liczb macierzy. Pozwala to na zachowanie wzorców danych wielowymiarowych. ConvNet rozmieszcza swoje neurony w trzech wymiarach: szerokości, wysokości i głębokości. Każda warstwa przekształca swoją wejściową objętość 3D w wyjściową objętość 3D neuronów za pomocą funkcji aktywacji.

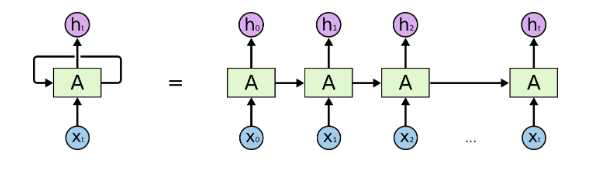
## Architektura CRNN i komórki LSTM

Model użyty do rozpoznawania przyciętych obrazów słownych nosi nazwę Convolutional Recurrent Neural Network (CRNN), ponieważ, ponieważ łączy głębokie konwolucyjne sieci neuronowe (DCNN) i rekurencyjne sieci neuronowe(RNN). Architektura tego modelu została przedstawiona na rysunku 2.

Składa się z trzech elementów, w tym warstw splotowych, warstw rekurencyjnych i warstwy transkrypcyjnej, w kolejności od dołu do góry.

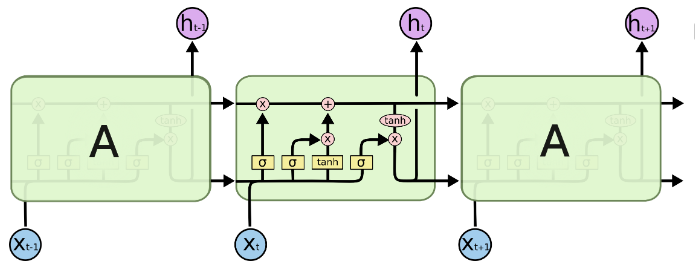
Tradycyjne sieci neuronowe nie potrafią zapamiętywać danych, przez co za każdym razem uczą się od nowa. **Sieci rekurencyjne** potrafią zapamiętać poprzednie dane, dzięki czemu mogą odnieść się do poprzedniego kontekstu w czasie nauki nowego. Umożliwia im to pętla, która sprawia, że sieć neuronowa wygląda jak wiele kopii tej samej sieci, z których każda przekazuje wiadomość do następcy.

Ta pętla umożliwia udostępnianie danych różnym węzłom i predykcje zgodnie z zebranymi informacjami. Ten proces można nazwać pamięcią. Struktura pętli umożliwia sieci neuronowej przyjmowanie sekwencji danych wejściowych. RNN przekształca zmienną niezależną w zmienną zależną dla swojej następnej warstwy.



Zwykłe sieci rekurencyjne posiadają jednak wadę jaką jest **Problem długoterminowych zależności(The Problem of Long-Term Dependencies)**,nie s w stanie poradzić sobie ze zwiększającą się odległością zależności od poprzedniego kontekstu.

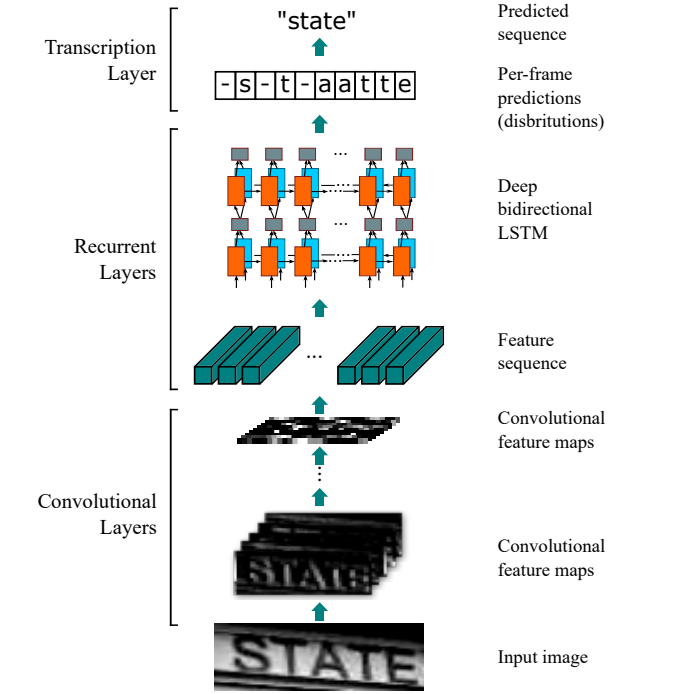
Problem ten rozwiązuje **komórki długoterminowej pamięci krótkoterminowej (LSTM)**. Zostały one zaprojektowane, aby uniknąć tego problemu. Zapamiętywanie informacji przez długi czas jest ich domyślnym zachowaniem. Wprowadzają stan komórki, oraz sygnały kontrolne. Sygnały kontrolne nazywane są bramkami i odpowiadają za sterowanie stanem komórki.



W wykorzystanej architekturze CRNN do przewidywania tekstu na obrazach wykorzystano głęboką dwukierunkową sieć LSTM(deep bidirectional LSTM).

Danymi wyjściowymi z sieci LSTM są sekwencje.

Następnie trzeba użyć funkcji dekodującej dane wyjściowe RNN. W tym celu potrzebujemy warstwy transkrypcyjnej. W omawianej architekturze została wykorzystana funkcja koneksjonistycznej klasyfikacji czasowej (CTC). Przewiduje ona podany tekst na obrazie dla każdej ramki na podstawie prawdopodobieństwa. Pozwala ona na obejście braku znajomości wyrównania między wejściem a wyjściem. Wykorzystuje ona praw

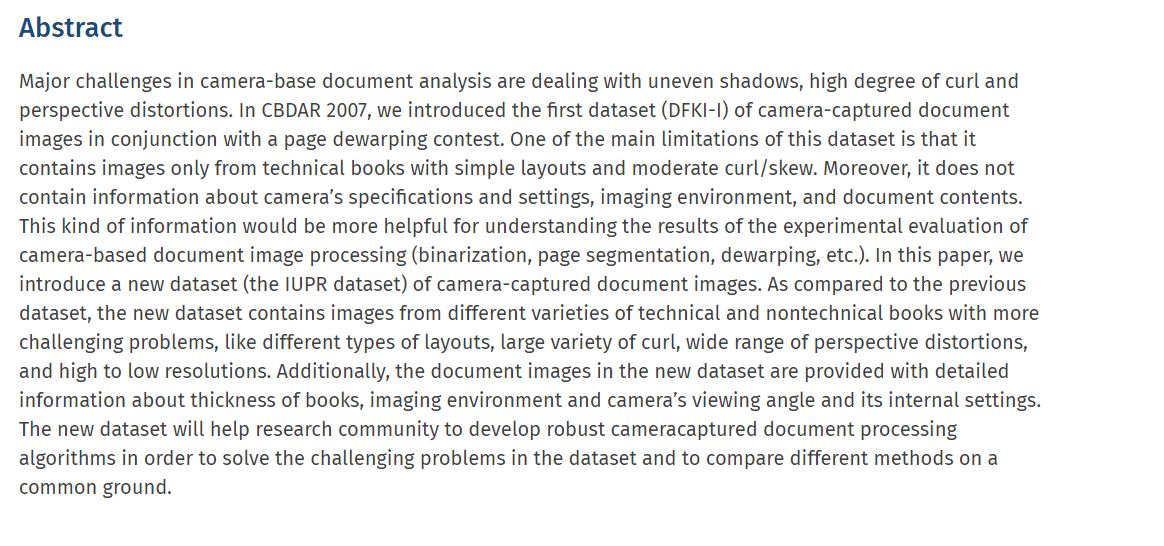


# Implementacja/trening sieci do segmentacji

## Segmentacja Linii

### Zbiór danych

[Zbiór danych](https://drive.google.com/uc?id=1ixh-tv-IKMdNjpoINbILJKGRNwcfH5Tj)(Dataset)[3] zawiera 300 obrazów i został przygotowany ręcznie przez autora [historii](https://medium.com/geekculture/detecting-text-lines-in-a-document-image-using-deep-learning-5a21b480bc4c)[4]. Powstał poprzez wykonanie wielu zrzutów ekranu z różnych stron dokumentów, a następnie oznaczenie tekstu poprzez nałożenie maski. Do tego zadania zostało wykorzystane narzędzie do adnotacji pikseli. Piksele tekstu oraz tła otrzymały maski w innych kolorach, co pozwoliło na oddzielenie konturów tekstu od tła. Przykładowy zrzut ekranu przedstawia rys 3, a etykietę(label) danej przedstawia rys 4

[](https://link.springer.com/chapter/10.1007/978-3-642-29364-1_13#:~)



### Wstępne przetwarzanie danych

Dane podawane są do modelu poprzez funkcję generatora wsadowego(batch generator), wobec czego cały preprocessing odbywa się w środku tej funkcji.

Jako pierwszą metodę już podczas samego wczytywania obrazu za pomocą biblioteki [OpenCV](https://opencv.org) zastosowano konwersję kanałów kolorów z RGB na **odcienie szarości**(grayscale). Następnie zastosowano odwróconą binaryzację obrazu:

**1ret,img=cv2.threshold(img,150,255,cv2.THRESH\_BINARY\_INV)**

Następnym etapem wstępnego przetwarzania danych było **zmienienie wielkości** obrazów, tak aby pasowały do rozmiaru wejściowego sieci 512 na 512 pikseli.

Kolejnym etapem było usunięcie wymiaru i znormalizowanie obrazu;

Następnie następuje preprocessing labeli. Funkcja get\_segmented\_img wykonuje całe zadanie i zwraca gotowy tensor. W środku funkcji następuje przeskalowanie maski w taki sam sposób jak obrazu x oraz usunięcie wymiarów. Potem labele uzupełniane są zerami w pustych miejscach, aby zapewnić wymagany rozmiar.

Na końcu dzielę zbiór danych na 75% danych testowych i 25% danych walidacyjnych.

### Szczegóły implementacyjne

Wytrenowałem model architektury U-Net zaimplementowany w [repozytorium](https://github.com/VikasOjha666/Textline-Segmentation-using-UNet)[5].

Model podczas treningu wykorzystuje ModelCheckpoint z biblioteki keras.callbacks. Ustawiony jest w następujący sposób:

Listing 1.1. Przykładowy listing.

mc = ModelCheckpoint('weights{epoch:08d}.h5',

save\_weights\_only=True, period=1)

Zapewnia to zapisywanie wag sieci po zakończeniu każdej epoki pod nazwą weights{numer}.h5

Listing 1.1. Przykładowy listing.

model.compile(optimizer = tf.keras.optimizers.Adam(lr = 1e-4),

loss = 'binary\_crossentropy', metrics = ['accuracy'])

Użytym otymalizatorem jest Adam, a szybkość uczenia się sieci(learnin rate) została ustawiona na 1e-4. Jako funkę straty ustawiono binary\_crossentropy, a wyświetlaną metryką jest accuracy

Trenuję model za pomocą metody model.fit\_generator() z następującymi parametrami:

**liczba epok = 10**

**liczba kroków na epokę(seps per epoch ) = 1000**

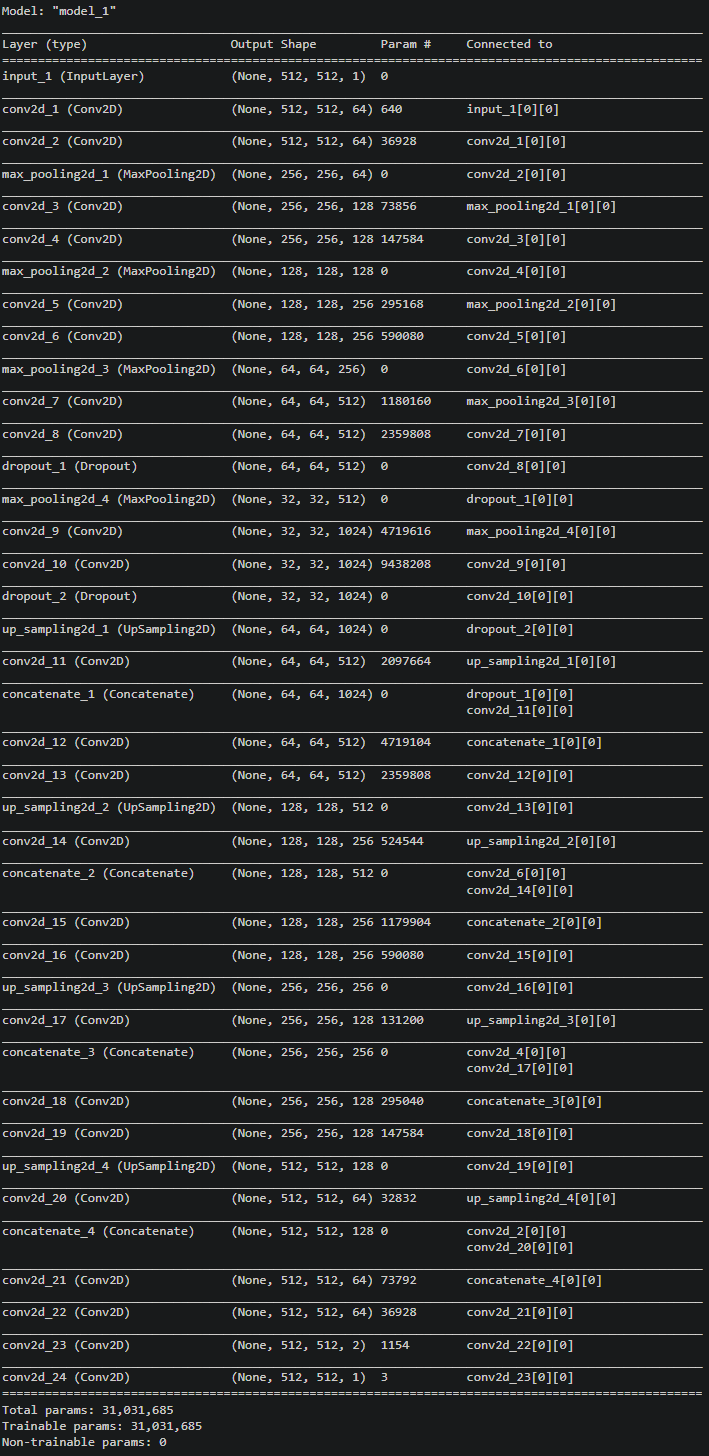
**dane wejściowe(train data) = batch\_generator(file\_train,2,2)**

**dane walidacyjne(validation data) = batch\_generator(file\_test,2,2)**

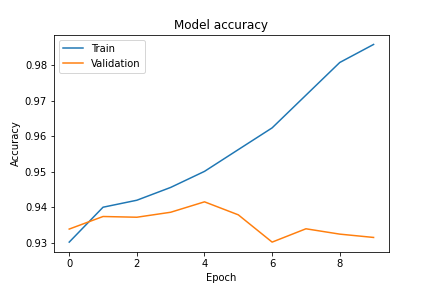
**liczba kroków walidacyjnych(validation steps) = 400**

**callbacks=mc - wcześniej opisany ModelCheckpoint**

**shuffle=1**



Trenowałem sieć przez 10 epok. Studiując proces nauki i wykres przedstawiony na rysunku nr 3 łatwo zauważyć, że najlepsze wyniki sieć osiągnęła po 5 epoce. Wobec tego jako domyślne wybrałem [wagi zapisane po zakończeniu treningu 5 epoki](https://drive.google.com/file/d/1a5F8m5xtbwq7_QkAc_H5YbfphB_Sc9Ze/view?usp=sharing). Wszystkie następne epoki są książkowym przykładem przeuczenia sieci(overfitting).



Następnie stworzony został skrypt ze zdefiniowaną funkcją o nazwie segment\_into\_lines , która przyjmuje adres do obrazu strony. Ta funkcja odczytuje plik w formacie skali szarości, a następnie dokonuje binaryzacji obrazu. Po binaryzacji zmienia rozmiar obrazu do 512x512, ponieważ jest to rozmiar wejściowy naszego modelu segmentacji linii. Następnie rozszerza wymiar w poprzek osi 0, aby uwzględnić wymiar wsadowy. Rozszerza również domyślnie wymiar wzdłuż osi kanału, ponieważ gdy obraz jest otwierany w formacie skali szarości, nie ma kanału koloru. Następnie przekazał go do modelu w celu przewidzenia maski segmentacji linii.

Ponownie usuwa wymiar wsadowy i wymiar kanału, ponieważ ponownie musimy wykonać binaryzację maski segmentacji, aby wykonać na niej detekcję konturu. Po wykonaniu binaryzacji wykonujemy detekcję konturu na obrazie za pomocą funkcji findCountours . Po uzyskaniu konturów przekazujemy je do funkcji boundingRect w OpenCV, która przybliża prostokątne pola dla tych konturów. Skalujemy również te obwiednie w stosunku do rozmiaru oryginalnego obrazu, ponieważ oryginalny obraz jest większy i bardziej wyraźny. Na koniec używamy tych obwiedni do przycinania obrazów linii z obrazu i zwracania ich dodanych na liście.

Poniżej znajduje się wynik po uruchomieniu wywołania tej funkcji na stronie.

## 

## Segmentacja wyrazów

Do wygenerowania zbioru danych w modelu dzielącym linie na wyrazy został wykorzystany poprzednio wyuczony model unet i poprzednio wykorzystany zbiór danych.

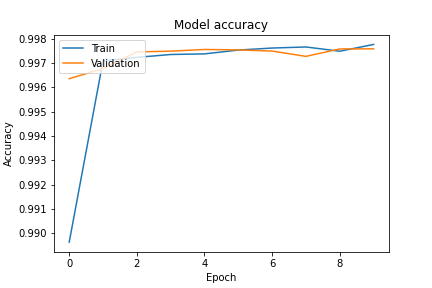
Wstępne przetwarzanie oraz sam model sieci są do siebie bardzo zbliżone.

Nauka drugiego modelu przebiegała następująco jak na rys 5.

Sieć była już nauczona w trzeciej epoce, a każda kolejna była zbędna, więc do tworzenia całego silnika wykorzystałem wagi zapisane po 3 epoce.

377s 377ms/step - loss: 0.0204 - accuracy: 0.9972 - val\_loss: 0.0202 - val\_accuracy: 0.9975

Następnym krokiem jest wykorzystanie skryptu napisanego w języku Python SegmentLine.py z zaimplementowaną funkcją segment\_into\_words, która zwraca wycięte wyrazy z obrazu wejściowego.



# Implementacja/trening crnn i lstm

## Zbiór danych

Zbiór danych wygenerowałem syntetycznie pobranym [skryptem](https://github.com/VikasOjha666/Data-generator-for-CRNN), który lekko zmodyfikowałem. Dodałem do niego możliwość rotacji danych oraz zmieniłem ustawienia kilku parametrów zmniejszając losowość, aby wygenerowany obraz miał mniej zaszumień(noises) i bardziej przypominał czysto zeskanowany dokument niż zdjęcie niskiej jakości. Kod zmienionego generatora można pobrać z mojego [repozytorium na githubie](https://github.com/damiantomczyszyn/Praca-licencjacka-OCR-CRNN).

Generator posiada następujące funkcje:

Generowanie tekstu na tłach podanych w folderze

Generowanie tekstu czcionkami podanymi w folderze

Nakładanie szarości

Dane treningowe zostały wygenerowane przy użyciu następujących ustawień generatora:

1. Długość wyrazu w przedziale od 1 do 20 znaków
2. Wielkość znaków w przedziale 10 do 29 w pikselach
3. Losowana wartość jasności obrazu w przedziale 0.6 do 1.1 od czysto wygenerowanego obrazu. Nakładana jest poprzez wykorzystanie funkcji Multiply() biblioteki imgaug
4. Powiększenie poprzez wykorzystanie funkcji dilate z biblioteki OpenCV z wielkością kernel2=np.ones((1,1),np.uint8)
5. dodanie erozji poprzez wykorzystanie funkcji erode z biblioteki OpenCV z kernelem kernel=np.ones((2,2),np.uint8)

Podczas generowania danych do pliku annotation.txt zapisywane są nazwy zapisanych obrazów oraz ich label. Jeden element jest zapisany w jednej linii, a nazwę pliku od labela oddziela niewykorzystywany znak separacji.

Wygenerowane dane obejmują znaki takie jak małe i duże litery łacińskie, oraz cyfry arabskie. Dodałem jeszcze kilka znaków interpunkcyjnych punclist='.?,()":'

## Wstępne przetwarzanie danych

Do pobrania listy znaków użyłem modułu string.

W procesie wstępnego przetwarzania danych wykorzystałem dwie funkcje.

1.Pierwsza z nich znajduje kolor tła obrazu. Na każdym obrazie liczba pikseli tła zawsze będzie większa niż liczba pikseli koloru tekstu, dlatego piksele mające największe występowanie na obrazie to piksele związane z tłem. A zatem kolor tych pikseli jest kolorem tła.

2.W następnej funkcji odpowiadającej za całe wstępne przetwarzanie wykonuje po kolei:

1. Sprawdzenie czy obraz nie jest pusty,
2. Przeskalowanie obrazu na wymiar oczekiwany przez wejście sieci,
3. Wywołanie funkcji znajdującej kolor tła obrazu.

Stworzyliśmy tę funkcję, abyśmy mogli dopasować nasz obraz wejściowy do 128x32 i uniknąć zmiany rozmiaru, ponieważ zmiana rozmiaru ma wpływ na dokładność. Funkcja preprocess\_img wykonuje operację zmiany rozmiaru. Funkcja encode\_to\_labels konwertuje tekst na indeksy, ponieważ sieci neuronowe przetwarzają liczby, a nie tekst.

Listing 1.1. Przykładowy listing.

**1 def encode\_to\_labels(txt):**

**2 dig\_lst = []**

**3 for index, char in enumerate(txt):**

**4 try:**

**5 dig\_lst.append(char\_list.index(char))**

**6 except:**

**7 print(char)**

**8 return dig\_lst**

Listing 1.1. Przykładowy listing.

**1 def find\_dominant\_color(image):**

**2 width, height = 150,150**

**3 image = image.resize((width, height),resample = 0)**

**4 pixels = image.getcolors(width \* height)**

**5 sorted\_pixels = sorted(pixels, key=lambda t: t[0])**

**6 dominant\_color = sorted\_pixels[-1][1]**

**7 return dominant\_color**

Listing 1.1. Przykładowy listing.

**1 def preprocess\_img(img, imgSize):**

**2 if img is None:**

**3 img = np.zeros([imgSize[1], imgSize[0]])**

**4 print("Image None!")**

**5 (wt, ht) = imgSize**

**6 (h, w) = img.shape**

**7 fx = w / wt**

**8 fy = h / ht**

**9 f = max(fx, fy)**

**10 newSize = (max(min(wt, int(w / f)), 1),**

**11 max(min(ht, int(h / f)), 1))**

**12 img = cv2.resize(img, newSize, interpolation=cv2.INTER\_CUBIC)**

**13 most\_freq\_pixel=find\_dominant\_color(Image.fromarray(img))**

**14 target = np.ones([ht, wt]) \* most\_freq\_pixel**

**15 target[0:newSize[1], 0:newSize[0]] = img**

**16 img = target**

**17 return img**

Dopełnienie labeli każdej sekwencji do maksymalnej długości tekstu w zbiorze danych. Do tego celu wykorzystana została funkcja pad\_sequences z biblioteki tf.keras.utils.

parametr padding został ustawiony na post, aby uzupełniano koniec każdej sekwencji, a nie początek.

parametr value będący wartością dopełniania został ustawiony na len(char\_list)

teraz długość każdej sekwencji w partii jest równa

prop=int(np.floor(len(imagenames)\*0.95))# split generated data

trainTuples=imagenames[:prop]

validTuples=imagenames[prop:]

trainTxts=txts[:prop]

validTxts=txts[prop:]

podział danych na 95% treningowych i 5% testowych

## Szczegóły implementacyjne

Model wzorowany na <https://arxiv.org/pdf/1507.05717.pdf>

Wykorzystany model CRNN przyjmuje na wejściu(input) obraz o rozmiarze 128 na 32 piksele. Dodatkowo dołączamy 31 do training\_label\_length i valid\_input\_length, ponieważ maksymalny znak, jaki nasz model może rozpoznać, to 32, ponieważ RNN ma w naszym modelu 32 kroki czasowe. Jeśli chcemy rozpoznać więcej, możemy zwiększyć wymiar obrazu.

Przy wygenerowaniu 4 milionów zdjęć sieć w 8 epoce osiągnęła następujące poziomy metryk:

* val\_loss 0.02566
* loss 0.324
* accuracy 0.9867
* val\_accuracy 0.9892

Epoka trwała 3312s 56ms/step przy wykorzystywanym hardware.

Kompilacja modelu

model.compile(loss={'ctc': lambda y\_true, y\_pred: y\_pred}, optimizer=tf.keras.optimizers.Adam(learning\_rate=0.0001),metrics=['accuracy'])

es = EarlyStopping(monitor='val\_loss', mode='min', verbose=1, patience=5)

filepath='CRNN\_model.hdf5'

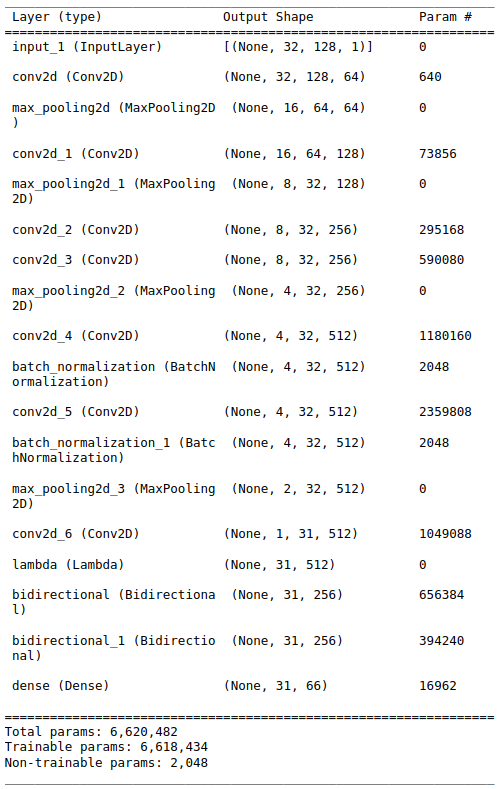
checkpoint = ModelCheckpoint(filepath=filepath, monitor='val\_loss', verbose=1, save\_best\_only=True, mode='auto')

callbacks\_list = [checkpoint,es]

epochs = 10

history = model.fit(train\_generator,validation\_data = valid\_generator,epochs = epochs,verbose = 1, callbacks = callbacks\_list)

Sieć uczę przy pomocy generatora wsadowego. 4 miliony danych nie zmieszczą się w niemalże żadnej pamięci karty graficznej.



Jako funkcje straty wykorzystuję CTC. Umieszczam ją przy pomocy wyrażenia lambda przed dwukierunkowymi sieciami LSTM

Funkcja straty CTC, która jest warstwą transkrypcyjną używaną do przewidywania danych wyjściowych dla każdego kroku czasowego.

Wysokości 32 i szerokości 128

warstwy splotu z rozmiarem jądra (3,3)

A liczba filtrów została większona z 64 do 512 warstwa po wastwie.

Dodanawane są dwie warsty puli o rozmiarze (2,2)

# Eksperymenty, testy, rezultaty, wnioski.

## Dane testowe

Dane testowe przygotowałem poprzez odpowiednią obróbkę pdf. Za pomocą darmowych narzędzi internetowych po przycinałem marginesy do tekstu , a następnie każdą stronę A4 podzieliłem na pół i ponownie po przycinałem marginesy. W taki sposób przygotowałem całego pdfa do procesu OCR.

## Test na Tesseract[[6]](https://tesseract-ocr.github.io).

Do przeprowadzenia porównania skuteczności sieci z popularnym silnikiem OCR wybrałem [Tesseract OCR](https://tesseract-ocr.github.io). Jest to darmowy silnik OCR wspierany przez Google, który osiąga jedne z najwyższych wyników dokładności. Jako interfejs wykorzystałem [pytesseract](https://pypi.org/project/pytesseract/). Pozwala mi on na wykorzystywanie zainstalowanego Tesseract OCR w wersji 5.1.0 w skryptach języka Python.

W celu sprawdzenia wydajności mojej sieci te same testowe obrazy załadowałem do silnika Tesseract. Napisałem prosty skrypt w języku Python który ładuje obraz, a następnie zmienia kodowanie koloru obrazu i zapisuje wynik procesu optycznego rozpoznawania znaków. W ustawieniach ustawiłem listę dostępnych znaków na taką samą jaką wyuczyłem swoją sieć. Wynikiem jest tekst uzyskany poprzez wywołanie funkcji image\_to\_string() z biblioteki pytesseract.

Podobny skrypt stworzyłem do swojego silnika OCR. Następnie wywołałem pojedynczo oba skrypty i porównałem wyniki.

Tutaj jakieś wyniki porównawcze 2 grafiki obok siebie i tabelka z wynikami

# Podsumowanie

Celem tej pracy było sprawdzenie skuteczności modeli rozpoznawania tekstu trenowanych na syntetycznych danych oraz przeuczenie sieci zaimplementowanego silnika OCR.

W pracy zostały poprawnie wytrenowane trzy sztuczne sieci neuronowe. Dwie o architekturze U-net i jedna o architekturze CRNN.

Ogromne modele U-net nie są najlepszym rozwiązaniem problemu segmentacji obrazu. W celu poprawy wydajności można zastosować lżejsze modele detekcji tekstu jakim jest na przykład Differential Binarization. Ten model jest znacznie szybszy i został specjalnie zaprojektowany do rozpoznawania tekstu.

Można zastosować lepszy generator danych niż wykorzystany. Można łatwo odnaleźć wiele generatorów danych nadających się do rozwiązania do tego problemu dostępnych na GitHub.

Można zmieniać parametry sieci w celu odnalezienia ustawienia poprawiającego dokładność sieci. Zmniejszenie learning rate prawdopodobnie niewiele poprawiłoby skuteczność modelu.

As mentioned earlier we have used greed search to decode our prediction. Hence Beam search can be used instead to improve the accuracy.

The CRNN model can be trained to perform recognition on line level instead of word level.

Detections of the words can be done once instead of first segmenting into lines and then into words. This can be achieved by simply doing annotation on word level in the page but that would be time-consuming.

Dodanie większej ilości czcionek oraz grafik z tłem i zaszumieniami, dodanie większej liczby znaków do nauczenia.

Dodanie uczenia w locie, a nie na wcześniej wygenerowanych danych.

augmentacja danych do unetu, albo stworzenie większego datasetu.

# Bibliografia

1. Informacje o API Keras <https://keras.io/about/>, (dostęp 06.2022)
2. sd
3. sd
4. sd
5. Dokumentacja Tesseract https://tesseract-ocr.github.io
6. sd

github original codes