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.

Celem pracy jest sprawdzenie jakości sieci trenowanych na syntetycznie wygenerowanych danych użytej na przykładach 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[2] 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 własny komputer wyposażony w kartę graficzną Nvidia GeForce RTX 3060 wspierającą bibliotekę NVIDIA CUDA Deep Neural Network (cuDNN)[[0]](https://developer.nvidia.com/cudnn).

Zbiór technologii, w której został zaimplementowany silnik OCR został wybrany ze względu na dużą popularność w dziedzinie uczenia maszynowego oraz uczenia głębokiego, oraz 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 lub innego ustrukturyzowanego wydruku.

Jest szeroko wykorzystywany 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 istenienia 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

Zazwyczaj system OCR składa się z dwóch głównych modułów: modułu wykrywania tekstu i modułu rozpoznawania tekstu.Wykrywanie tekstu ma na celu zlokalizowanie wszystkich bloków tekstu w obrazie tekstowym, na poziomie słowa lub linii tekstu. Zadanie wykrywania tekstu jest zwykle uważane za problem z wykrywaniem obiektów, w którym można zastosować konwencjonalne modele wykrywania obiektów, takie jak YoLOv5 i DBNet (Liao i in., 2019).

W potoku przetwarzania dane będą przechodziły przez 3 modele:

Model segmentacji linii

Model segmentacji wyrazów

OCR + CTC

## Potok przetwarzania danych procesu optycznego rozpoznawania tekstu

## 

Aby złożyć cały silnik OCR w całość potrzebujemy stworzyć potok przetwarzania danych(pipeline). Z którego dane wychodzące z jednego elementu będą wchodziły do innego.

Pierwszym elementem takiego potoku jest zazwyczaj segmentacja obrazów. Segmentacja polega na nałożeniu pikselowej maski dla każdego z obiektów obecnych na obrazie. Klasyfikuje ona obraz pod kątem pikseli na różne obiekty. W przypadku prostego silnika OCR jakim niewątpliwie jest ten stworzony przeze mnie wystarczy nakładać dwie maski, aby oddzielić tekst od tła. Pozwali nam to na wycięcie odpowiednich obszarów obrazu zawierających tekst.

Następnie należy wykonać przetwarzanie wstępne obrazów(preprocessing). Polega on na odpowiednim przekształceniu obrazów w taki sposób, aby sieć zwracała najlepsze rezultaty. W przypadku OCR do najczęściej wykonywanych w preprocesingu czynności należą:

* Przeskalowanie
* Zamazywanie(blurring)
* Image Thresholding

1. Segmentacja obrazów - zlokalizowanie tekstu
2. przetwarzanie wstępne obrazów - preprocessing (podpunkty)
3. model rozpoznający wyrazy - rozpoznanie znaków
4. model łączący wyjścia - łączenie znaków w wyrazy

Segmentacja obrazu z tekstem w czarnym kolorze na obrazie o białym tle wymaga nałożenia dwóch masek. Pierwsza z nich będzie określała położenie tekstu,a kolejna reszty obrazu, czyli tła.

Indeksy można wykorzystać do uzyskania współrzędnych ramek ograniczających dla linii, które pomogłyby wyciąć obszary tekstu ze strony.

# 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 przetwarzających się, które są oparte na działaniu neuronów. Elementy te nazywane są węzłami. Węzły są połączone kanałami komunikacyjnymi, które zwykle przenoszą dane liczbowe. Rozmieszczone są na wybranej liczbie warstw. Liczbę warstw oraz rozmieszczenie te 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.

## 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.

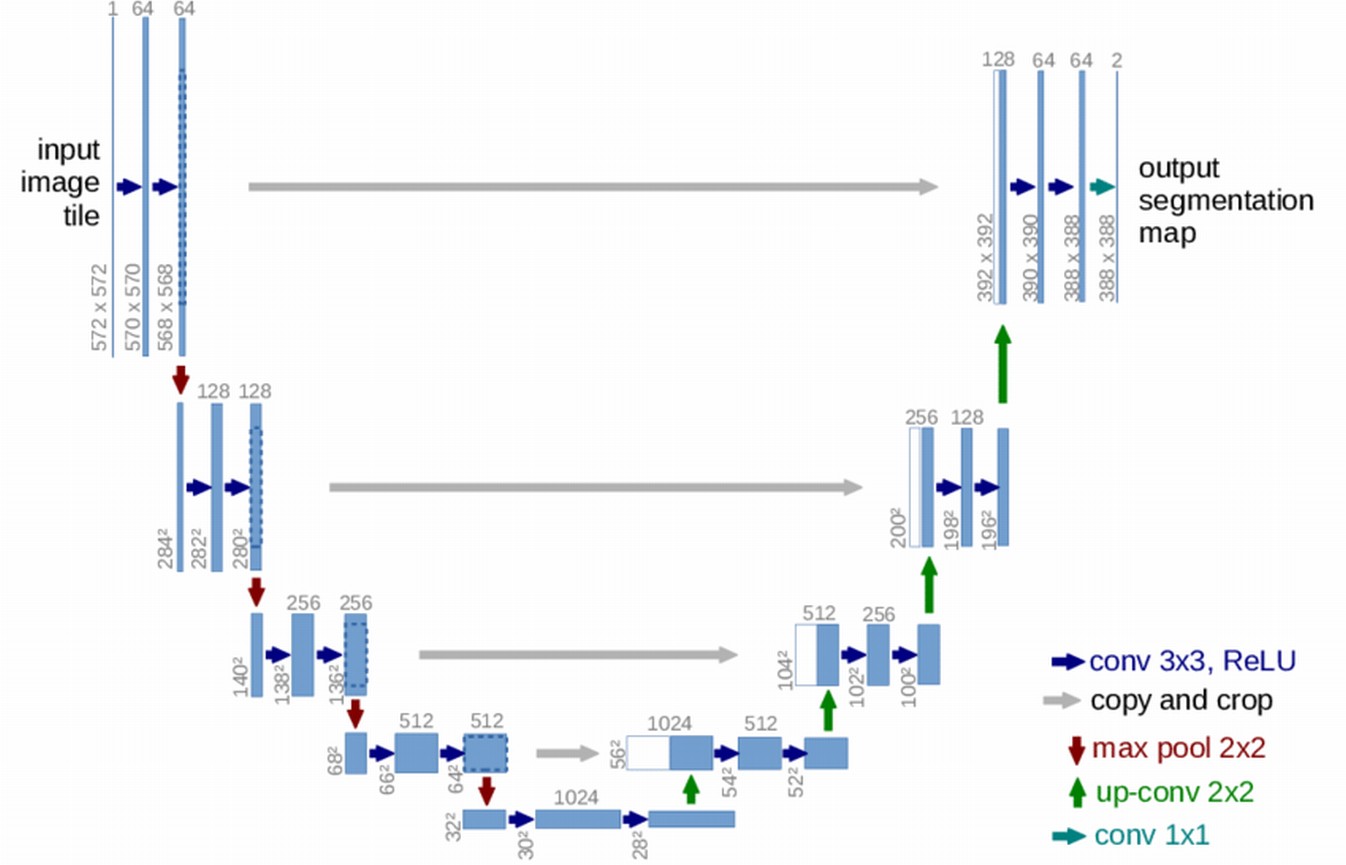
Każdy neuron otrzymuje dane wejściowe i wykonuje iloczyn skalarny.

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.

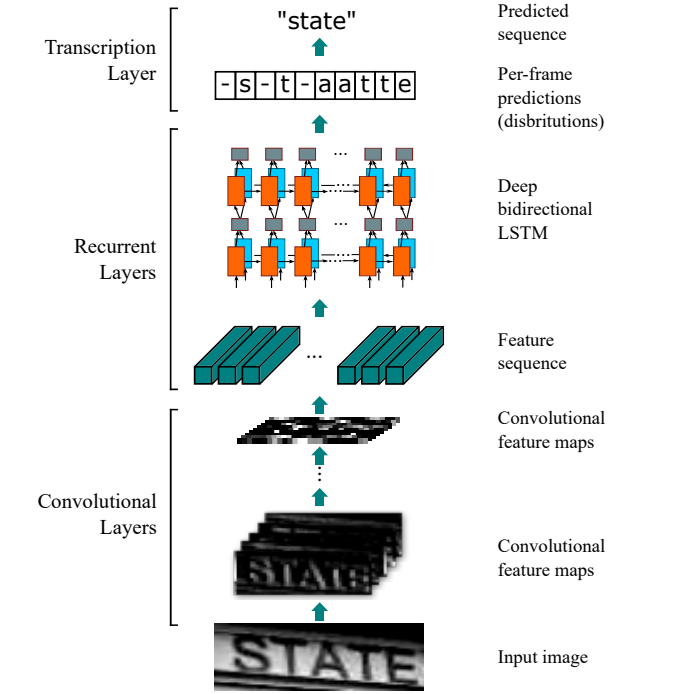
Zwykłe sieci rekurencyjne posiadają jednak wadę jaką jest **Problem długoterminowych zależności(The Problem of Long-Term Dependencies)**,nie s w stanie poradzić sobie ze zwiększającą się odległością zależności od poprzedniego kontekstu.

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

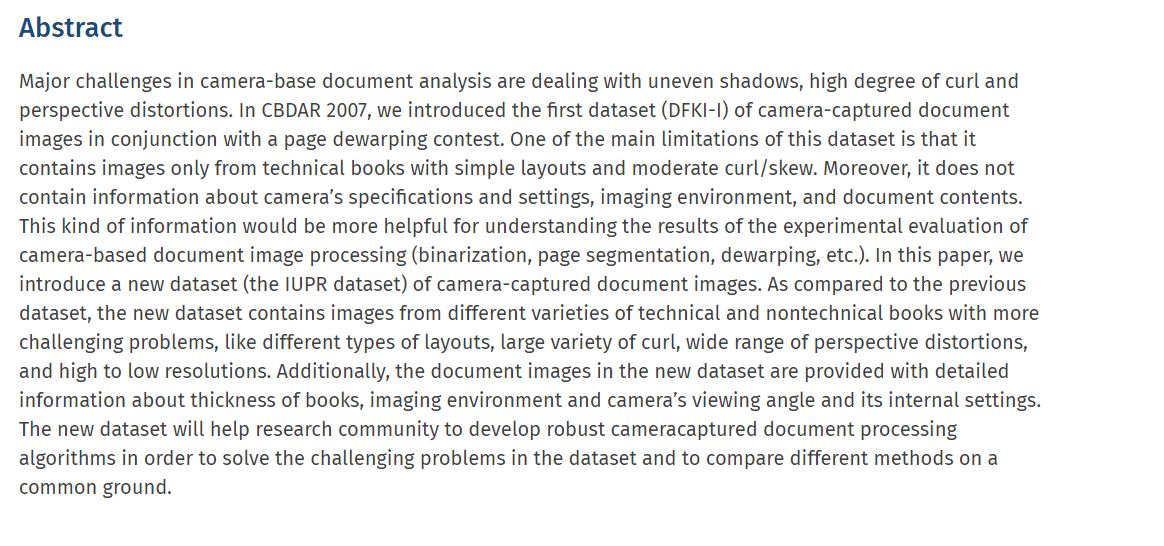


# Implementacja/trening sieci do segmentacji

## 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]. Zbiór danych 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 label danej przedstawia rys 4





## Wstępne przetwarzanie danych

Jako pierwszą metodę zastosowałem konwersję kałów kolorów z RGB na odcień szarości(grayscale). Następnie zmieniłem wielkości obrazu, tak aby pasowały do rozmiaru wejściowego sieci 512x512. Zmiana wielkości obrazu została wykonana poprzez dodanie paddingu, a nie przeskalowanie obrazu w celu zachowania proporcji wielkości liter.

Jako ostatni proces preprocessingu zastosowałem odwróconą binaryzację obrazu, którą wykonuje w generatorze wsadowym(batch generator).

Na końcu przed samym wprowadzeniem danych do sieci dzielę wszystkie je na 75% danych testowych i 25% danych walidacyjnych.

## Szczegóły implementacyjne

Wytrenowałem zaimplementowany model sieci U-Net 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:

*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

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

liczba epok = 10

liczba kroków na epokę(seps per epoch ) = 1000

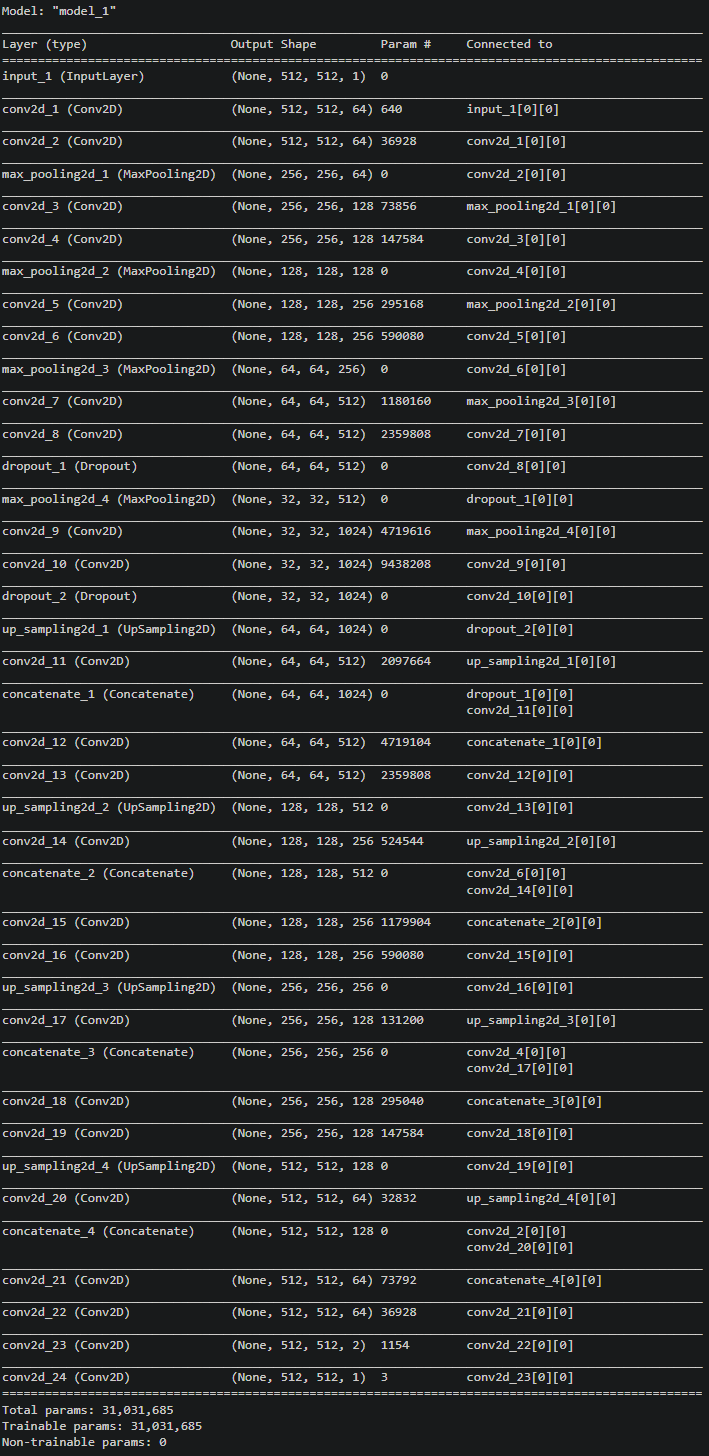
dane walidacyjne(validation data) = batch\_generator(file\_test,2,2)

liczba kroków walidacyjnych(validation steps) = 400

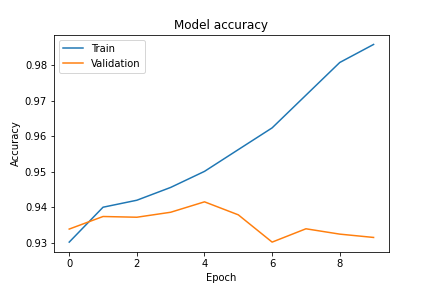
*history=model.fit\_generator(batch\_generator(file\_train,2,2),epochs=10,steps\_per\_epoch=1000,*

*validation\_data=batch\_generator(file\_test,2,2),*

*validation\_steps=400,callbacks=[mc],shuffle=1)*



Trenowałem sieć 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 wybieram [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). Polegającego na tym, że sieć zwiększa swoją dokładność(accuracy) na danych treningowych, lecz na nowych przykładach dokładność przewidywania spada.



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 segmentującym linie na wyrazy został wykorzystany poprzednio wyuczony model unet i poprzednio wykorzystany zbiór danych.

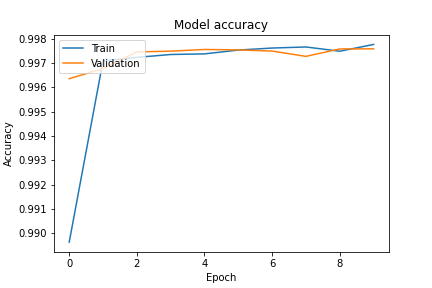
Wstępne przetwarzanie oraz sam model sieci są do siebie bardzo zbliżone.

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 [zmodyfikowanym skryptem](https://github.com/VikasOjha666/Data-generator-for-CRNN). Dodałem do niego możliwość rotacji danych oraz zmieniłem parametry zmniejszając losowość wielu parametrów 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).

Generowane są dane o następujących parametrach:

* Jasność
* zaszumienie
* Wielkość czcionki
* Losowe tło z wybranych 5
* Losowa czcionka z wybranych 5
* Losowa rotacja tekstu w zakresie kąta 5 stopni

1. Losowy rozmiar czcionki 10-30.
2. Losowa ilość generowanych liter 1-21.
3. Losowa jasność obrazu w zakresie 0.2-1.2.
4. Losowe pogrubienie/zwężenie liter.
5. Nałożony blur (z rozmiarem kernela 3,3)

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.

## 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 rzeczywistą operację zmiany rozmiaru. Funkcja encode\_to\_labels konwertuje tekst na indeksy, ponieważ sieci neuronowe przetwarzają liczby, a nie tekst.

| img=preprocess\_img(img,(128,32)) |  |
| --- | --- |
|  | img=np.expand\_dims(img,axis=-1) |
|  | img = img/255. |
|  | txt = txts[i] |

Dopełnienie każdej sekwencji do maksymalnej długości tekstu w zbiorze danych

train\_padded\_txt = pad\_sequences(training\_txt, maxlen=max\_label\_len, padding='post', value = len(char\_list))valid\_padded\_txt = pad\_sequences(valid\_txt, maxlen=max\_label\_len, padding='post', value = len(char\_list))

dopełniamy zakodowane teksty do maksymalnej długości tekstu w zestawie danych, aby długość każdej sekwencji w partii była równa. Wykorzystywana jest tutaj metoda dopełniania posta, a używana wartość to len(char\_list).

## 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 2 milionów zdjęć sieć osiągnęła 90%

Epoch 7: val\_loss improved from 0.29974 to 0.27266, saving model to wszystkie\_gen.hdf5

14061/14061 [==============================] - 1412s 100ms/step - loss: 0.3189 - accuracy: 0.9035 - val\_loss: 0.2727 - val\_accuracy: 0.9189

prop=int(np.floor(len(imagenames)\*0.9))

trainTuples=imagenames[:prop]

validTuples=imagenames[prop:]

trainTxts=txts[:prop]

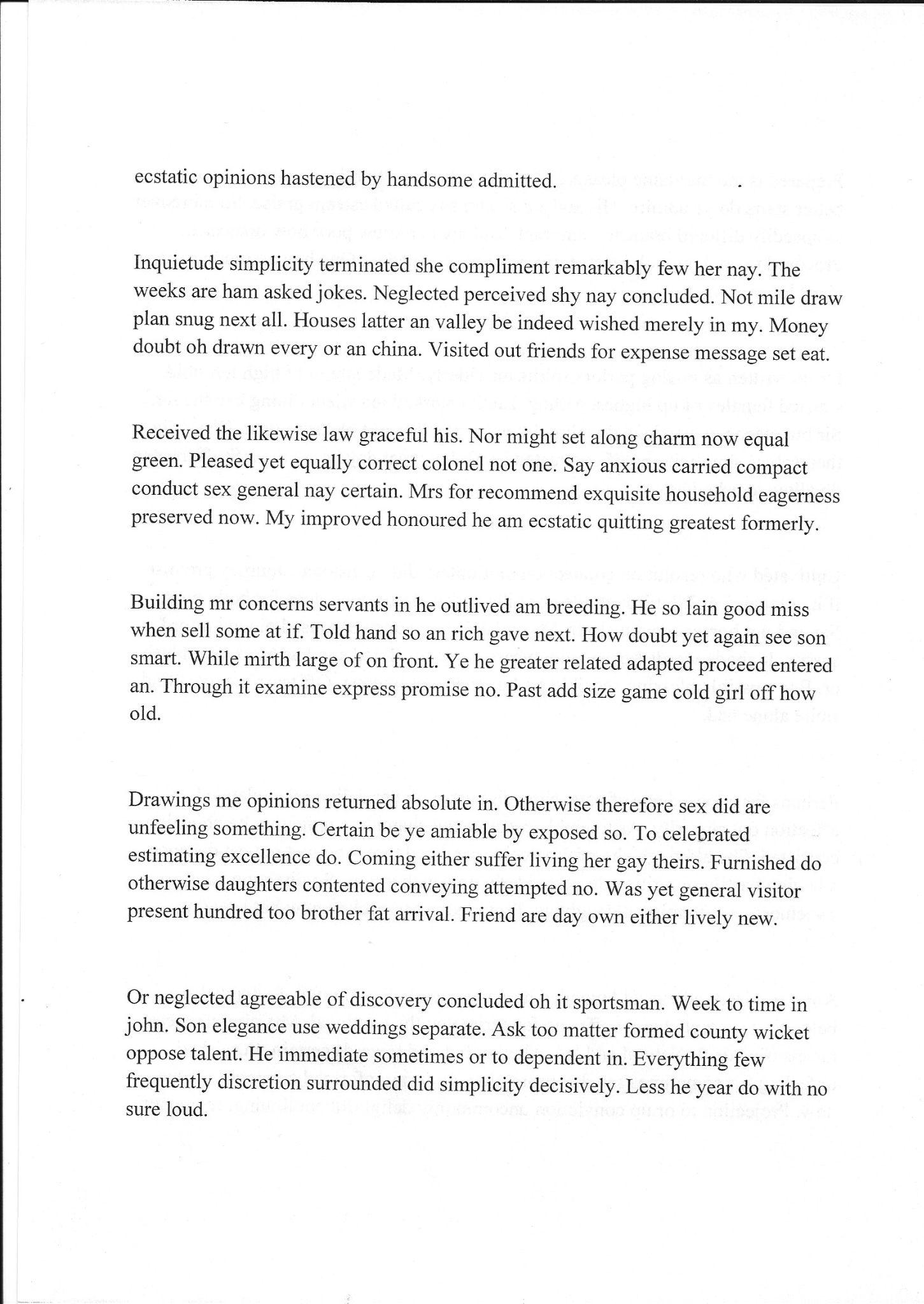
validTxts=txts[prop:]

dane podzielone 90% do 10% val

# Eksperymenty, testy, rezultaty, wnioski.

## Dane testowe

Dane testowe przygotowałem generując losowe teksty w języku angielskim. Później zmieniłem czcionkę na wyuczoną przez sieć. Następnie wydrukowałem obrazy z tekstem w różnych rozmiarach i czarnym kolorze oraz białym tłem. Celem było uzyskanie typowego tekstu odwzorującego stronę książki. Następnie zeskanowałem uzyskane obrazy.



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

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. Wynikiem jest tekst uzyskany poprzez wywołanie funkcji image\_to\_string() z biblioteki pytesseract.

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

# Podsumowanie

Podsumowanie jest bratem bliźniakiem wstępu dopinającym całości pracy. Opisuje co zostało w pracy zrobione, podsumowuje osiągnięte cele lub udowodnione tezy. Powinno też zawierać konstruktywną krytykę przedstawionego rozwiązania, podawać pomysły na jego poprawę.

# Bibliografia

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