UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ

W LUBLINIE

Wydział Matematyki, Fizyki i Informatyki

Kierunek: **Informatyka**
Specjalność: -

Damian Tomczyszyn
nr albumu: 296585

Zastosowanie sztucznych sieci neuronowych do rozpoznawania pisma

Application of artificial neural networks for text recognition

Praca licencjacka
napisana w Katedrze Oprogramowania Systemów Informatycznych
pod kierunkiem dr Marcina Denkowskiego

Lublin 2022

Spis treści

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. 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 neuronowych trenowanych przy użyciu syntetycznie wygenerowanych danych uczących na danych rzeczywistych. Dodatkowym celem pracy jest dokładniejsze wytrenowanie sieci poprzez poprawienie hiperparametrów sieci i zmodyfikowanie generatora danych, oraz zaimplementowanie funkcji generatora wsadowego do sieci neuronowej odpowiedzialnej za rozpoznawanie 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 został 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.

Rozdział 1. Wprowadzenie do optycznego rozpoznawania znaków

1.1. Definicja

Optyczne rozpoznawanie znaków (ang. *Optical character recognition*) [1] to dziedzina badań w zakresie rozpoznawania wzorców, sztucznej inteligencji i widzenia komputerowego zajmująca się konwersją tekstu. Proces konwersji polega na przetwarzaniu obrazu z tekstem przez maszynę lub system, w celu uzyskania nieprzetworzonego tekstu, bądź innego ustrukturyzowanego wydruku. Stosowana jest dla różnych typów dokumentów, takich jak pliki PDF, zeskanowane dokumenty papierowe lub zdjęcia wykonane aparatem cyfrowym. Dokumenty te mogą być zapisane tekstem pisanym odręcznie lub zakodowanym maszynowo.

OCR jest szeroko wykorzystywane jako forma wprowadzania znaków z drukowanych dokumentów papierowych takich jak paszporty, faktury, wyciągi bankowe, paragony, dokumenty pocztowe, wydruki danych statycznych lub innych dokumentacji. Jest to powszechna metoda digitalizacji papierowych dokumentów. Cyfrowe dokumenty mają przewagę nad swoim papierowym odpowiednikiem, ponieważ można je elektronicznie edytować, przesyłać, przeszukiwać, przechowywać w bardziej zwarty sposób i wykorzystywać w procesach przetwarzania maszynowego, takich jak automatyczne tłumaczenie maszynowe, zamiana tekstu na mowę, znacznie szybsze wyszukanie kluczowych dane i eksploracja tekstu.

1.2. Problematyka

Systemy OCR działają coraz lepiej, chociaż nadal nie działają zbyt dobrze na zdjęciach obejmujących rzeczywiste scenariusze [2]. 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 dla obrazów ze świata rzeczywistego skutkuje brakiem informacji zwrotnych do systemu OCR w celu poprawy wykrywania lub

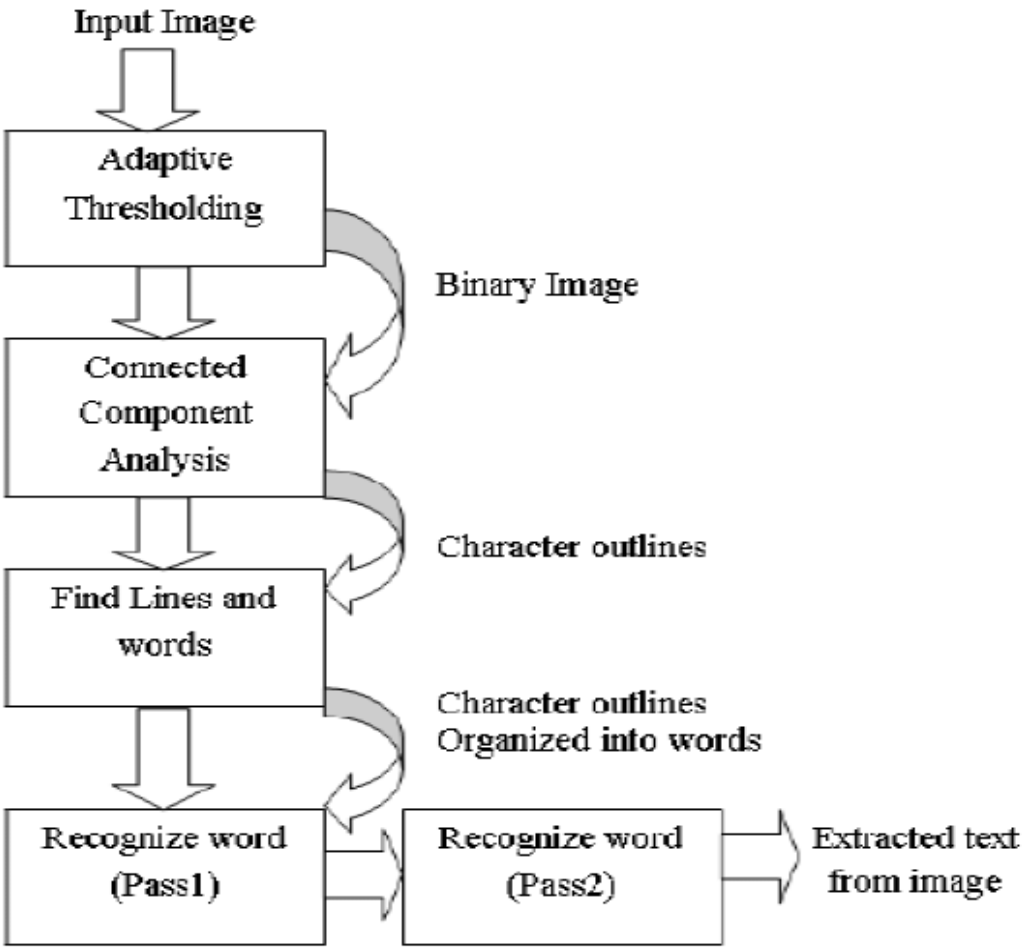
ekstrakcji, w oparciu o błędy w dalszej aplikacji. Jest to powodem braku kompleksowego wyszkolenia sieci, co przekłada się na gorszą skuteczność silnika.

Idealnym rozwiązaniem tego problemu są generatory danych syntetycznych, które zapełniają luki w zbiorach danych. Generowane dane od razu zostają opatrzone odpowiednimi oznaczeniami. Dzięki temu nie ma potrzeby wykonywania adnotacji ręcznie. Przekłada się to na ogromną oszczędność czasu oraz środków. Problemem jest odwzorowanie danych rzeczywistych w danych generowanych syntetycznie. Jeśli dane wygenerowane syntetycznie nie będą odwzorowywać danych rzeczywistych to sieć nauczy się rozpoznawać dane syntetyczne, ale podczas wykonywania rozpoznania danych rzeczywistych nie sprawdzi się. 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.

1.3. Struktura silnika OCR

Złożenie silnika OCR w całość wymaga stworzenia potoku przetwarzania danych (ang. *pipeline*). Dane wychodzące z jednego elementu będą przesłane na wejście następnego. Opisując przykładową strukturę wzorowałem się na Tesseract OCR [3]. Struktura potoku przetwarzania danych silnika OCR zazwyczaj składa się z dwóch głównych modułów i wykorzystanych w nich podmodułów. 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 (ang. *prep-processing*). Tesseract wykonuje różne operacje przetwarzania wstępnego przy użyciu biblioteki Leptonica przed wykonaniem właściwego OCR. Przykładowymi i popularnymi operacjami przetwarzania wstępnego danych w procesie OCR są:

1) skalowanie obrazu (ang. *rescalin*g),
2) binaryzacja (ang. *binarization*),
3) odkrzywianie tekstu (ang. *skew correction*),
4) usuwanie szumów (ang. *noise removal*),
5) usuwanie granic (ang. *border removal*).



Rysunek 1.1. Architektura Tesseract OCR [4]

Następnie dane są przekształcane w tensory i przekazane do **modułu rozpoznawania tekstu**. Obecnie jako skuteczny moduł rozpoznawania tekstu wykorzystywana jest architektura sieci neuronowej *convolutional recurrent neural network* [5].

Proces rozpoznawania znaków można zakończyć wykonując przetwarzanie końcowe danych (and. *post-processing*). Polega on na przetwarzaniu danych finalnych umożliwiającym zwiększenie dokładności (ang. *accuracy*) poprzez wyłapanie i korekcję błędów błędów ortograficznych i gramatycznych powstałych w wyniku wad systemu OCR.

Rozdział 2. Sieci neuronowe

2.1. Wprowadzenie do sieci neuronowych

Sztuczna inteligencja (ang. *artificial intelligence*) jest nauką badającą inteligencję demonstrowaną przez maszyny. Odnosi się do idei nadającej maszynom lub oprogramowaniu możliwość podejmowania własnych decyzji o predefiniowane reguły lub modele rozpoznawania wzorców. Idea rozpoznawania wzorców prowadzi do modeli **uczenia maszynowego** (ang. *machine learning*). 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 powtarzających 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 (ang. *layers*). Liczbę warstw oraz rozmieszczenie węzłów określa struktura sieci. Jeśli struktura sieci obejmuje wiele dodatkowych warstw pomiędzy warstwą wejściową (ang. *input layer*) a warstwą wyjściową (ang. *output layer*) to taka sieć nazwana jest **głęboką siecią neuronową** (ang. *deep neural network*). Każdy węzeł ma przypisaną wagę (ang. *weight*), oraz może być wyposażony w *bias*, czyli stałą dodatkową wartość. Wartość wyjściowa węzła składa się z sumy wartości z warstw poprzednich pomnożonej przez wagi oraz nałożonej na tę sumę funkcji aktywacji. Funkcja aktywacji jest przypisana do warstwy, gdzie każdy węzeł w tej warstwie jest aktywowany za pomocą tej funkcji.

Sztuczne sieci neuronowe uczą się podczas procesu nazwanego **treningiem** (ang. training). Trening polega na pokazywaniu sieci ogromnej liczby przykładów uczących w celu dopasowania wartości wag. Sieć należy trenować do momentu kiedy przestanie zmniejszać wartość walidacyjnej funkcji strat (ang. *validation loss function*). Jeśli przerwie się naukę sieci wcześniej, to taka sieć będzie niedouczona (ang. *underfitting*). Odwrotnym problemem do niedouczenia jest przeuczenie (ang. *overfitting*). Jest to sytuacja, gdy sieć zwiększa swoją dokładność prognozy na danych treningowych, ale nie zwiększa dokładności przewidywania na danych walidacyjnych. Powstało wiele rozwiązań, które mają pomóc zapobiec przeuczeniu sieci. Najpopularniejsze rozwiązania to:

1) zmniejszenie pojemności sieci poprzez usuwanie warstw lub zmniejszanie liczby komórek w warstwach ukrytych,
2) zastosowanie regularyzacji (ang. *regularization*) za pomocą dodania kosztu do funkcji straty dla dużych ciężarów,
3) dodanie do struktury sieci *dropout layers* [6].

Przyspieszenie treningu sieci zapewnia warstwa (ang. *batch normalization***)** [7]. Jest to 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.

Optymalizator (ang. optimizer) jest jednym z dwóch argumentów wymaganych do kompilacji modelu Keras [8]. Jest to algorytm lub metoda używana do zmiany atrybutów sieci neuronowej, takich jak wagi i szybkość uczenia się w celu zmniejszenia strat.

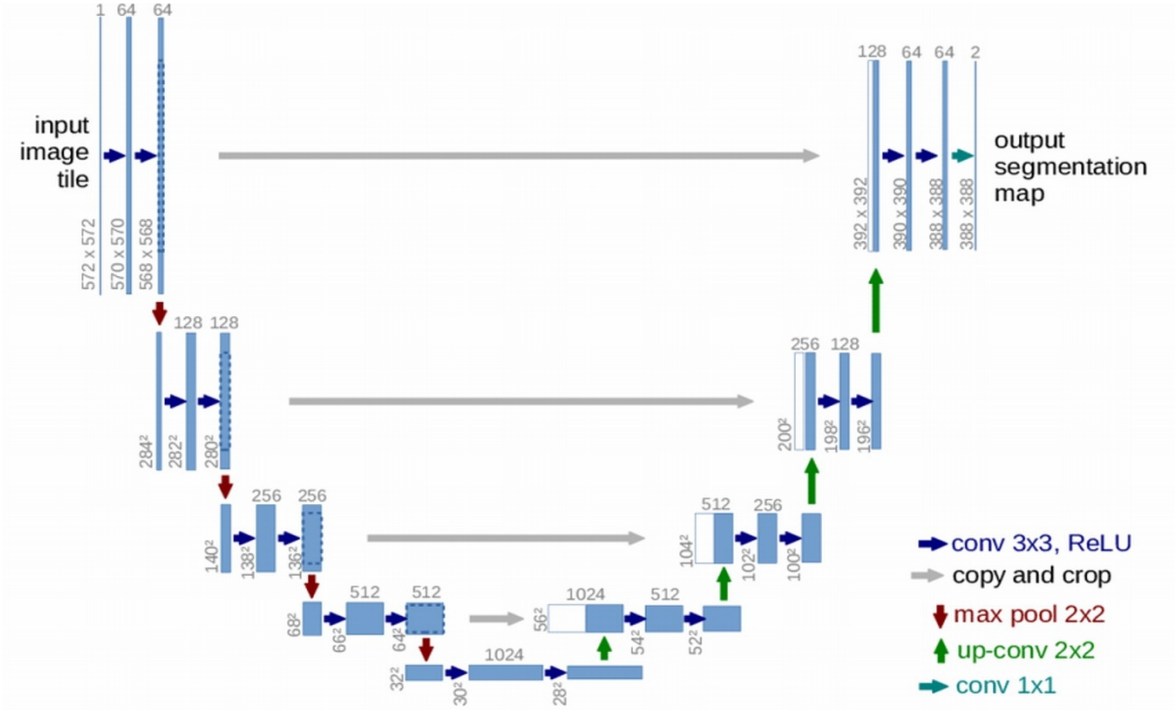
2.2. Architektura U-net i głębokie sieci konwolucyjne

Sieci konwolucyjne (ang. convolutional neural network) [9] są odpowiedzialne za wyodrębnianie sekwencji cech z każdego obrazu wejściowego. Zostały zaprojektowane z myślą o zadaniach związanych z rozpoznawaniem obrazu. Tym co je odróżnia od zwykłych sztucznych sieci neuronowych jest posiadanie trzech głównych warstw. Są to **warstwa konwolucji** (ang. *convolution layer*), **warstwa puli** (ang. *pooling layer*) i warstwa w **pełni połączona** (ang. *fully connected*) [10]. W przypadku rzeczywistych danych obrazu CNN działają lepiej niż wielowarstwowe perceptrony (ang. *multilayer perceptron*). 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.

Do przeprowadzenia procesu segmentacji wykorzystano architekturę U-net [11]. U-net to architektura opracowana w 2015 roku na Uniwersytecie we Freiburgu w Niemczech.

Jest to obecnie jedna z najpopularniejszych architektur stosowanych w zadaniach segmentacji semantycznej. Została zaprojektowana do uczenia się z mniejszej liczby próbek. Architektura ta jest ulepszeniem sieci w pełni konwolucyjnej opracowanej przez Jonathana Longa w 2014 roku [10].

Kształt architektury U-net przypomina literę U, która składa się z czterech bloków kodera i czterech bloków dekodera połączonych mostem. Sieć enkodera (ścieżka kontraktująca) dzieli na pół wymiary przestrzenne i podwaja liczbę filtrów w każdym bloku enkodera. Odwrotny proces następuje w dekoderze. Sieć podwaja liczbę wymiarów przestrzennych i redukuje o połowę liczbę kanałów funkcji w każdym bloku. Mosty łączą bloki kodera i dekodera oraz uzupełniają przepływ informacji. Mosty składają się z dwóch filtrów konwolucyjnych o rozmiarach 3x3. Po każdej konwolucji wywoływana jest funkcja aktywacji ReLu.

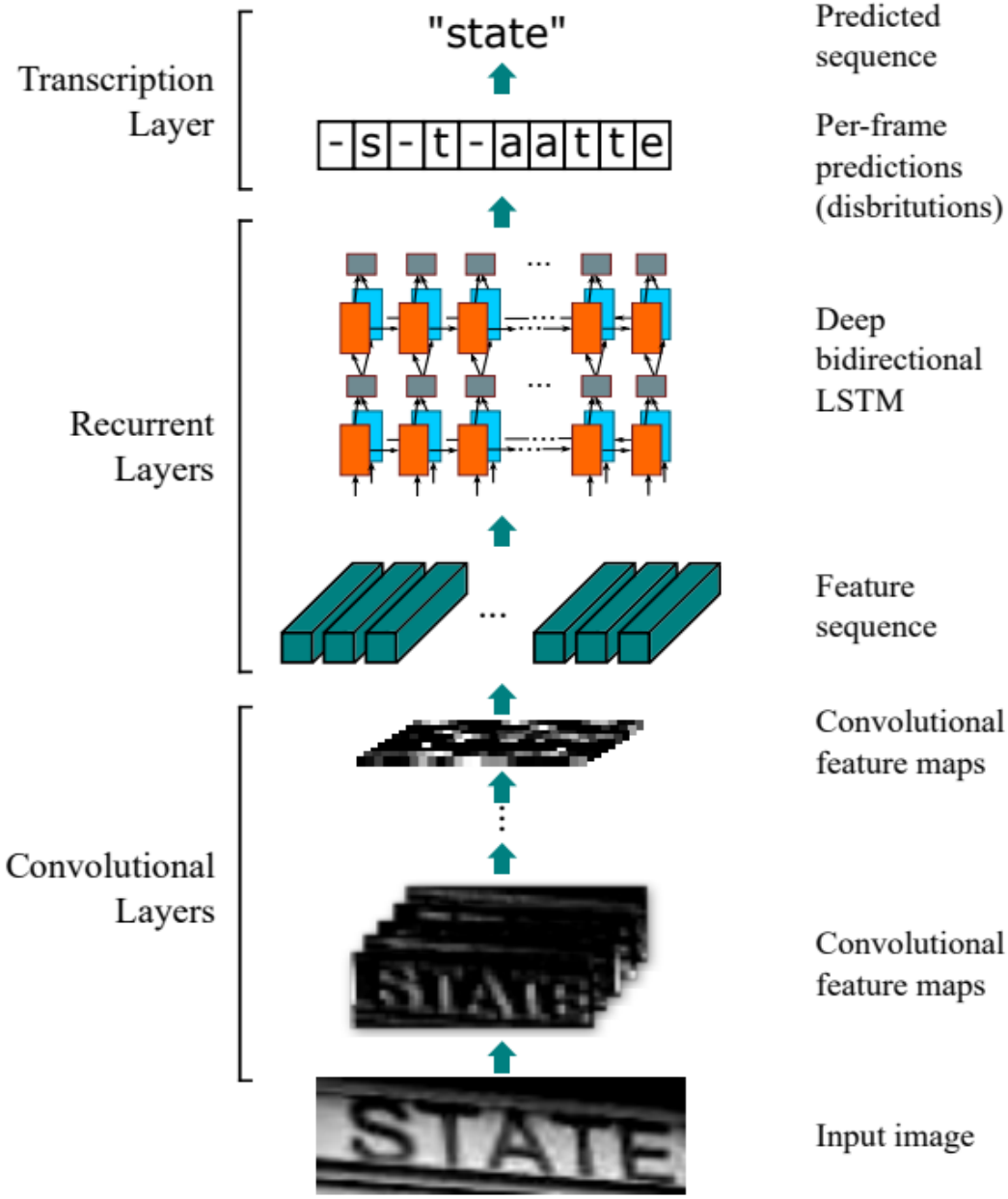


Rysunek 2.2. Architektura sieci U-net [15]

2.3. Architektura CRNN i komórki LSTM

Model użyty do rozpoznawania przyciętych obrazów słownych nosi nazwę *Convolutional Recurrent Neural Network* (CRNN) [5]. Łączy on głębokie konwolucyjne sieci neuronowe (DCNN) i rekurencyjne sieci neuronowe(ang. *recurrent neural network*). Architektura tego modelu została przedstawiona na rysunku 2.3.

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

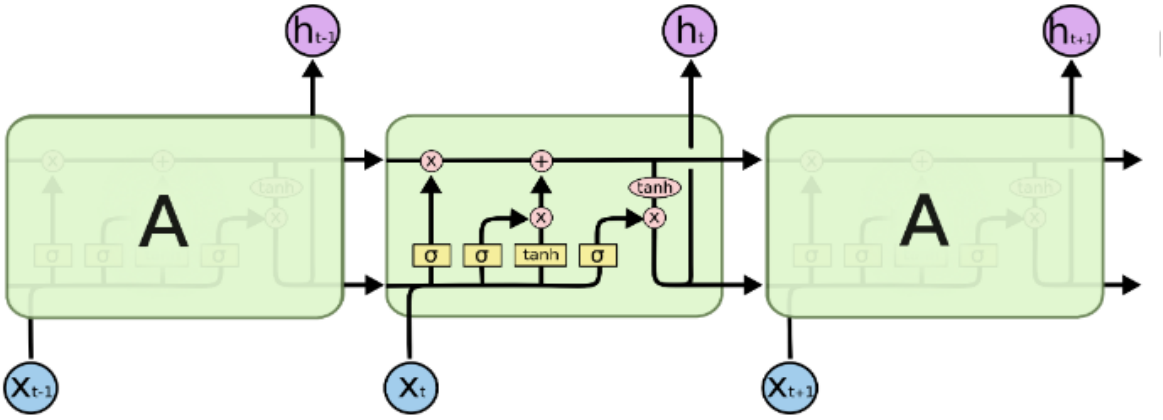


Rysunek 2.3. Architektura *convolutional recurrent neural network* [5].

Tradycyjne sieci neuronowe nie potrafią zapamiętywać danych, przez co za

każdym razem uczą się od nowa. **Sieci rekurencyjne** [12] zapamiętują 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** (ang. *problem of long-term dependencies*), nie są w stanie poradzić sobie ze zwiększającą się odległością zależności od poprzedniego kontekstu. Problem ten rozwiązują **komórki długoterminowej pamięci krótkoterminowej** (ang. *long short-term memory*) [13]. Zostały one zaprojektowane, aby uniknąć tego problemu. Zapamiętywanie informacji przez długi czas jest ich domyślnym zachowaniem. Wprowadzają stan komórki i sygnały kontrolne. Sygnały kontrolne nazywane są bramkami i odpowiadają za sterowanie stanem komórki. Komórka LSTM posiada trzy bramki: bramkę zapomnienia, bramkę wejściową i bramkę wyjściową.



Rysunek 2.4. Struktura komórki LSTM [14].

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

Danymi wyjściowymi z sieci LSTM są sekwencje. Sekwencje należy wysłać do funkcji dekodującej. W tym celu potrzebujemy warstwy transkrypcyjnej. W omawianej architekturze została wykorzystana funkcja connectionist temporal classification (CTC) [15]. Przewiduje ona podany tekst na obrazie dla każdej ramki na podstawie

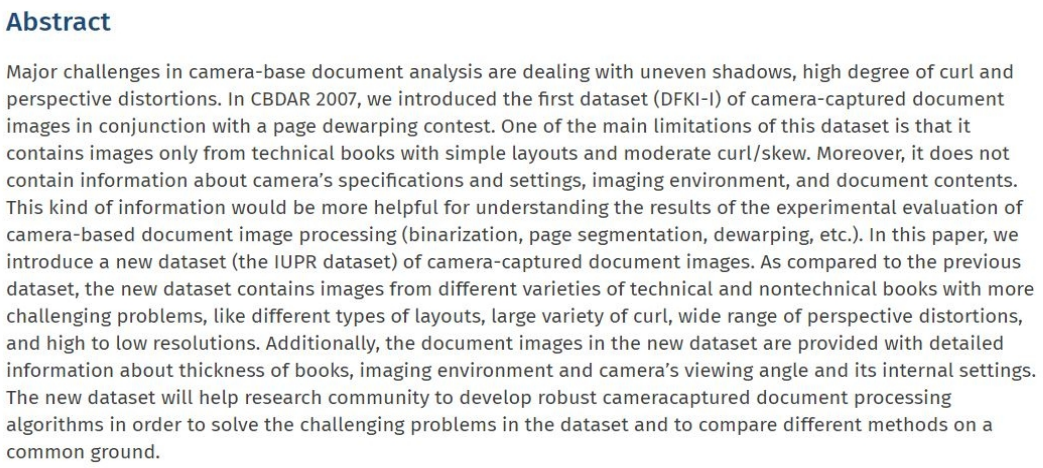
prawdopodobieństwa. Pozwala ona na obejście braku znajomości wyrównania między wejściem a wyjściem.

Rozdział 3. Implementacja/trening sieci do segmentacji

3.1. Segmentacja Linii

3.1.1. Zbiór danych

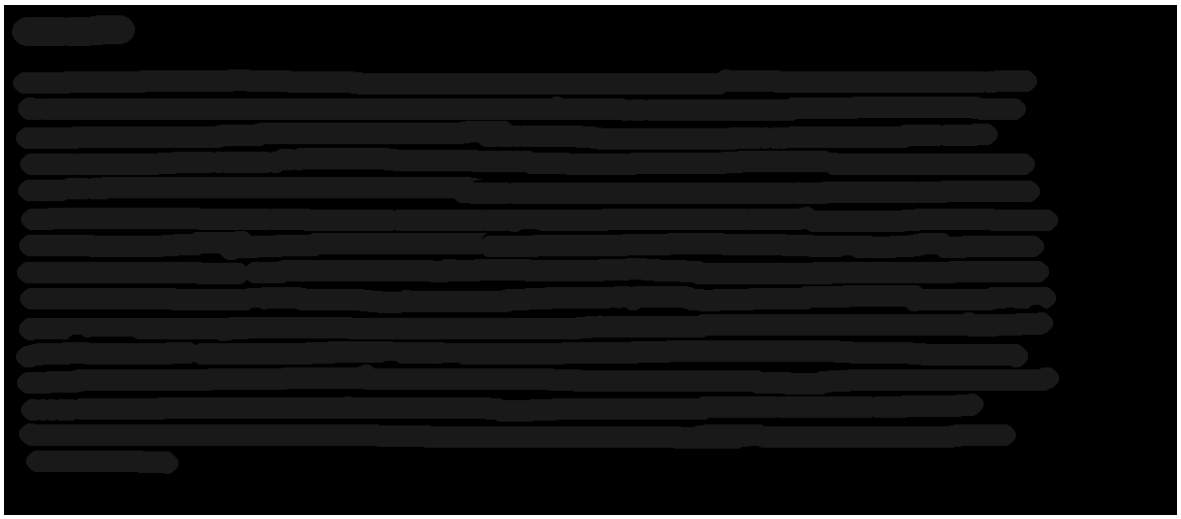
Zbiór danych (ang. *dataset*) zawiera 300 obrazów i został przygotowany ręcznie przez autora historii [16]. Powstał poprzez wykonanie wielu zrzutów ekranu z różnych stron dokumentów, a następnie oznaczenie tekstu poprzez nałożenie maski. Do tego zadania zostało wykorzystane narzędzie do adnotacji pikseli. Piksele tekstu oraz tła otrzymały maski w innych kolorach, co pozwoliło na oddzielenie konturów tekstu od tła. Kształty wielkości danych oscylują wokół rozmiaru strony A5. Przykładowy zrzut ekranu przedstawia rysunek 3.1, a etykietę danej przedstawia rysunek 3.2.



Abstract

Major challenges in camera-base document analysis are dealing with uneven shadows, high degree of curl and perspective distortions. In CBDAR 2007, we introduced the first dataset (DFKI-I) of camera-captured document images in conjunction with a page dewarping contest. One of the main limitations of this dataset is that it contains images only from technical books with simple layouts and moderate curl/skew. Moreover, it does not contain information about camera's specifications and settings, imaging environment, and document contents. This kind of information would be more helpful for understanding the results of the experimental evaluation of camera-based document image processing (binarization, page segmentation, dewarping, etc.). In this paper, we introduce a new dataset (the IUPR dataset) of camera-captured document images. As compared to the previous dataset, the new dataset contains images from different varieties of technical and nontechnical books with more challenging problems, like different types of layouts, large variety of curl, wide range of perspective distortions, and high to low resolutions. Additionally, the document images in the new dataset are provided with detailed information about thickness of books, imaging environment and camera's viewing angle and its internal settings. The new dataset will help research community to develop robust cameracaptured document processing algorithms in order to solve the challenging problems in the dataset and to compare different methods on a common ground.

Rysunek 3.1. Przykładowy obrazek ze zbioru danych.



Rysunek 3.2. Adnotacja rysunku 3.1.

3.1.2. Wstępne przetwarzanie danych

Dane podawane są do modelu poprzez funkcję generatora wsadowego (ang. *batch generator*), wobec czego cały proces przetwarzania wstępnego obrazu odbywa się w środku tej funkcji. Jako pierwszą metodę już podczas samego wczytywania obrazu za pomocą biblioteki OpenCV [17] zastosowano konwersję kanałów koloru z RGB na **odcienie szarości** (ang. *grayscale*). Następnie zastosowano odwróconą binaryzację obrazu przedstawioną na listingu 3.1.

Listing 3.1. Binaryzacja obrazu z wykorzystaniem biblioteki OpenCV.

```
ret,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. Później wykonano wstępne przetwarzanie etykiet. Do tego celu napisano funkcję, która wykonuje cały proces i zwraca gotowy tensor. W środku funkcji następuje przeskalowanie maski w taki sam sposób jak obrazu, oraz usunięcie wymiarów. Potem etykiety uzupełniane są zerami w pustych miejscach, aby zapewnić wymagany rozmiar. Na końcu następuje podział zbioru danych na 75% danych testowych i 25% danych walidacyjnych.

3.1.3. Szczegóły implementacyjne

Wykorzystałem model architektury U-Net zaimplementowany w repozytorium [18].

Model podczas treningu wykorzystuje ModelCheckpoint z biblioteki Keras. Ustawiony jest w sposób zaprezentowany na listingu 3.2.

Listing 3.2. Ustawienia obiektu klasy Model Checkpoint.

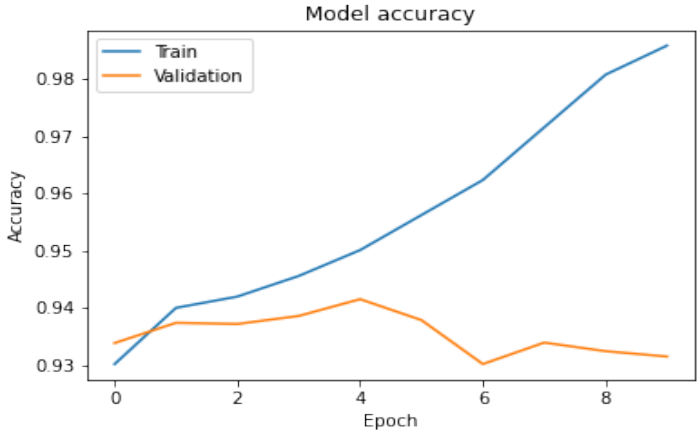
```
ModelCheckpoint('weights{epoch:08d}.h5', save_weights_only=True, period=1)
```

Zapewnia to zapisywanie wag sieci po zakończeniu każdej epoki pod nazwą weights{numer epoki}.h5. Do skompilowania modelu użyłem optymalizatora Adam z ustawioną szybkością uczenia (ang. *learning rate*) na wartość 0.0001, oraz funkcją straty binarnej entropii krzyżowej. Jako śledzoną metrykę wybrałem dokładność. Podsumowanie modelu po kompilacji widnieje na rysunku 3.3. Następnie trenowałem model poprzez wywołanie funkcji fit przez 10 epok (ang. *epoch*). Każda epoka miała ustawione wykonanie 1000 kroków na epokę (ang. *steps per epoch*) dla danych treningowych oraz 400 kroków na epokę dla danych walidacyjnych. Dane treningowe jak i walidacyjne były podawane poprzez zaimplementowany generator wsadowy.

```
Model: "model_1"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #   Connected to
=================================================================================================
input_1 (InputLayer)         (None, 512, 512, 1)       0
_________________________________________________________________________________________________
conv2d_1 (Conv2D)            (None, 512, 512, 64)      640       input_1[0][0]
_________________________________________________________________________________________________
conv2d_2 (Conv2D)            (None, 512, 512, 64)      36928     conv2d_1[0][0]
_________________________________________________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, 256, 256, 64)   0         conv2d_2[0][0]
_________________________________________________________________________________________________
conv2d_3 (Conv2D)            (None, 256, 256, 128 73856     max_pooling2d_1[0][0]
_________________________________________________________________________________________________
conv2d_4 (Conv2D)            (None, 256, 256, 128 147584    conv2d_3[0][0]
_________________________________________________________________________________________________
max_pooling2d_2 (MaxPooling2D)  (None, 128, 128, 128 0         conv2d_4[0][0]
_________________________________________________________________________________________________
conv2d_5 (Conv2D)            (None, 128, 128, 256 295168    max_pooling2d_2[0][0]
_________________________________________________________________________________________________
conv2d_6 (Conv2D)            (None, 128, 128, 256 590080    conv2d_5[0][0]
_________________________________________________________________________________________________
max_pooling2d_3 (MaxPooling2D)  (None, 64, 64, 256)    0         conv2d_6[0][0]
_________________________________________________________________________________________________
conv2d_7 (Conv2D)            (None, 64, 64, 512)       1180160   max_pooling2d_3[0][0]
_________________________________________________________________________________________________
conv2d_8 (Conv2D)            (None, 64, 64, 512)       2359808   conv2d_7[0][0]
_________________________________________________________________________________________________
dropout_1 (Dropout)          (None, 64, 64, 512)       0         conv2d_8[0][0]
_________________________________________________________________________________________________
max_pooling2d_4 (MaxPooling2D)  (None, 32, 32, 512)    0         dropout_1[0][0]
_________________________________________________________________________________________________
conv2d_9 (Conv2D)            (None, 32, 32, 1024)      4719616   max_pooling2d_4[0][0]
_________________________________________________________________________________________________
conv2d_10 (Conv2D)           (None, 32, 32, 1024)      9438208   conv2d_9[0][0]
_________________________________________________________________________________________________
dropout_2 (Dropout)          (None, 32, 32, 1024)      0         conv2d_10[0][0]
_________________________________________________________________________________________________
up_sampling2d_1 (UpSampling2D)  (None, 64, 64, 1024)   0         dropout_2[0][0]
_________________________________________________________________________________________________
conv2d_11 (Conv2D)           (None, 64, 64, 512)       2097664   up_sampling2d_1[0][0]
_________________________________________________________________________________________________
concatenate_1 (Concatenate)  (None, 64, 64, 1024)      0         dropout_1[0][0]
                                                                 conv2d_11[0][0]
_________________________________________________________________________________________________
conv2d_12 (Conv2D)           (None, 64, 64, 512)       4719104   concatenate_1[0][0]
_________________________________________________________________________________________________
conv2d_13 (Conv2D)           (None, 64, 64, 512)       2359808   conv2d_12[0][0]
_________________________________________________________________________________________________
up_sampling2d_2 (UpSampling2D)  (None, 128, 128, 512   0         conv2d_13[0][0]
_________________________________________________________________________________________________
conv2d_14 (Conv2D)           (None, 128, 128, 256 524544    up_sampling2d_2[0][0]
_________________________________________________________________________________________________
concatenate_2 (Concatenate)  (None, 128, 128, 512 0         conv2d_6[0][0]
                                                            conv2d_14[0][0]
_________________________________________________________________________________________________
conv2d_15 (Conv2D)           (None, 128, 128, 256 1179904   concatenate_2[0][0]
_________________________________________________________________________________________________
conv2d_16 (Conv2D)           (None, 128, 128, 256 590080    conv2d_15[0][0]
_________________________________________________________________________________________________
up_sampling2d_3 (UpSampling2D)  (None, 256, 256, 256   0         conv2d_16[0][0]
_________________________________________________________________________________________________
conv2d_17 (Conv2D)           (None, 256, 256, 128 131200    up_sampling2d_3[0][0]
_________________________________________________________________________________________________
concatenate_3 (Concatenate)  (None, 256, 256, 256   0         conv2d_4[0][0]
                                                            conv2d_17[0][0]
_________________________________________________________________________________________________
conv2d_18 (Conv2D)           (None, 256, 256, 128 295040    concatenate_3[0][0]
_________________________________________________________________________________________________
conv2d_19 (Conv2D)           (None, 256, 256, 128 147584    conv2d_18[0][0]
_________________________________________________________________________________________________
up_sampling2d_4 (UpSampling2D)  (None, 512, 512, 128   0         conv2d_19[0][0]
_________________________________________________________________________________________________
conv2d_20 (Conv2D)           (None, 512, 512, 64) 32832     up_sampling2d_4[0][0]
_________________________________________________________________________________________________
concatenate_4 (Concatenate)  (None, 512, 512, 128   0         conv2d_2[0][0]
                                                            conv2d_20[0][0]
_________________________________________________________________________________________________
conv2d_21 (Conv2D)           (None, 512, 512, 64) 73792     concatenate_4[0][0]
_________________________________________________________________________________________________
conv2d_22 (Conv2D)           (None, 512, 512, 64) 36928     conv2d_21[0][0]
_________________________________________________________________________________________________
conv2d_23 (Conv2D)           (None, 512, 512, 2)  1154      conv2d_22[0][0]
_________________________________________________________________________________________________
conv2d_24 (Conv2D)           (None, 512, 512, 1)  3         conv2d_23[0][0]
=================================================================================================
Total params: 31,031,685
Trainable params: 31,031,685
Non-trainable params: 0
```

Rysunek 3.3. Podsumowanie skomplikowanego modelu U-net [19].

Studiując proces nauki i wykres przedstawiony na rysunku o numerze 3.4 łatwo wybrać epokę 5 jako najskuteczniejszą dla danych walidacyjnych. Wobec tego jako domyślne wybrałem wagi zapisane po zakończeniu treningu 5 epoki. Wszystkie następne epoki są książkowym przykładem przeuczenia sieci.



Rysunek 3.4. Wykres przedstawiający proces nauki modelu U-net.

Później wykorzystałem stworzony skrypt ze zdefiniowaną funkcją dzielącą tekst na linie przy pomocy wytrenowanego modelu. 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, 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. Usuwany jest 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 przekazane są do funkcji *boundingRect*, która przybliża prostokątne pola dla tych konturów. Następuje skalowanie tych obwiedni w stosunku do rozmiaru oryginalnego obrazu, ponieważ oryginalny obraz jest większy i bardziej wyraźny. Na koniec wykorzystano obwiednie do wycięcia linii tekstu z obrazu.

3.2. Segmentacja wyrazów

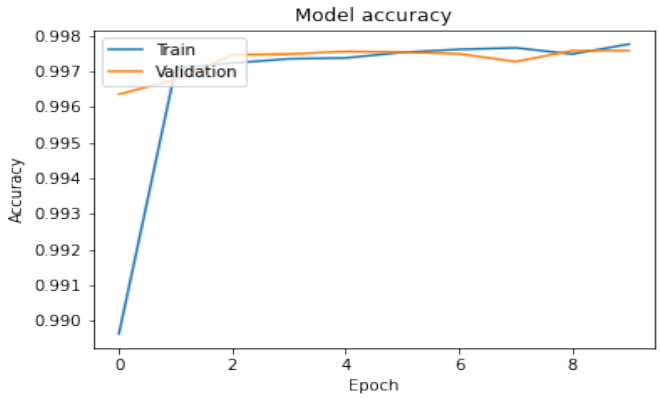
Do wygenerowania zbioru danych w modelu dzielącym linie na wyrazy został wykorzystany poprzednio wyuczony model unet i poprzednio wykorzystany zbiór danych.

Wstępne przetwarzanie oraz sam model sieci są do siebie bardzo zbliżone. Nauka drugiego modelu przebiegała w sposób przedstawiony na rysunku nr 3.5. Sieć była już nauczona w trzeciej epoce, a każda kolejna nie przynosiła dużej korzyści w dokładności, więc do tworzenia silnika wykorzystałem wagi zapisane po 3 epoce. Dokładniejsze wartości przedstawia listing 3.3.

Listing 3.3. Informacja zwrotna modelu po zakończeniu treningu w trzeciej epoce.

```
377s 377ms/step -loss: 0.020  - accuracy: 0.997 - val_loss: 0.020 - val_accuracy: 0.997
```

Następnym krokiem było wykorzystanie skryptu napisanego w języku Python z zaimplementowaną funkcją, która zwraca wycięte wyrazy z obrazu wejściowego.



Rysunek 3.5. Wykres przedstawiający proces nauki modelu U-net segmentującego linie na wyrazy.

Rozdział 4. Implementacja/trening crnn i lstm

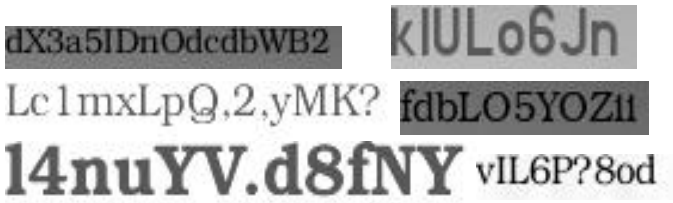
4.1. Zbiór danych

Zbiór danych wygenerowałem syntetycznie przy pomocy pobranego skryptu [20], który lekko zmodyfikowałem. Dodałem do niego możliwość rotacji danych, z której ostatecznie nie skorzystałem oraz zmieniłem ustawienia kilku parametrów zmniejszając losowość, aby wygenerowany obraz był mniej zniekształcony. Generator tworzy obrazki z tekstem i zapisuje je w nowym folderze a adnotacje zapisuje w pliku tekstowym. Na początku losowany jest ciąg znaków o długości w przedziale od 1 do 20 znaków. Do napisania tekstu użyta jest jedna z sześciu losowych czcionek. Rozmiar czcionki jest losowany w przedziale od 10 do 29 pikseli. W następnej kolejności tworzone jest tło w rozmiarze wylosowanego wyrazu w oknie powiększonym względem rozmiaru wyrazu o 10 pikseli. Później tekst jest wklejany w środek tego tła. Na końcu wykonywane są losowe transformacje obrazu i zapisanie obrazu oraz etykiet.

Losowe transformacje obejmują:

1) nałożenie odcieni szarości na obraz przy pomocy przygotowanej w tym celu grafiki szarego tła,
2) rozjaśnienie obrazka,
3) pogrubienie obrazka co przekłada się na pogrubienie samej czcionki,
4) erozja obrazu.

Wygenerowane dane obejmują znaki takie jak małe i duże litery łacińskie,cyfry arabskie i kilka znaków interpunkcyjnych.



Rysunek 4.1. Przykładowo wygenerowane dane testowe.

4.2. Wstępne przetwarzanie danych

Do pobrania listy znaków użyłem modułu *string*. W procesie wstępnego przetwarzania danych wykorzystałem dwie funkcje. 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. Kolor tych pikseli jest kolorem tła. 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.

Wykorzystana funkcja zapewnia dopasowanie nasz obrazu wejściowy do rozmiaru 128x32 i pozwala na uniknięcie zmiany rozmiaru, ponieważ zmiana rozmiaru obrazu ma wpływ na dokładność. W dalszej kolejności wykorzystałem funkcję konwertują tekst na indeksy, ponieważ sieci neuronowe przetwarzają liczby, a nie tekst.

Listing 4.1. Implementacja funkcji konwertującej tekst na indeksy.

```
def encode_to_labels(txt):
    dig_lst = []
    for index, char in enumerate(txt):
        try:
            dig_lst.append(char_list.index(char))
        except:
            print(char)
    return dig_lst
```

Listing 4.2. Implementacja funkcji znajdującej kolor tła.

```
def find_dominant_color(image):
    width, height = 150,150
    image = image.resize((width, height),resample = 0)
    pixels = image.getcolors(width * height)
    sorted_pixels = sorted(pixels, key=lambda t: t[0])
    dominant_color = sorted_pixels[-1][1]
return dominant_color
```

Model wymaga dopełnienia etykiet każdej sekwencji do maksymalnej długości tekstu w zbiorze danych. Do tego celu wykorzystana została funkcja *pad_sequences* z biblioteki Keras. Parametr *padding* został ustawiony na *post*, aby uzupełniano koniec każdej sekwencji, a nie początek. Parametr *value* będący wartością dopełniania został ustawiony na niewykorzystywany znak. Na końcu dane zostają podzielone na zbiór 95% danych treningowych i 5% danych testowych z wygenerowanych 4.000.000 danych.

4.3. Szczegóły implementacyjne

Wykorzystany model CRNN był wzorowany na przedstawionym w artykule [5]. Podsumowanie skompilowanego modelu przedstawia rysunek 4.2. Przyjmuje on na wejściu obraz o rozmiarze 128 na 32 piksele. RNN w użytym modelu ma 32 kroki czasowe, a więc największa długość wyrazu jaki może przyjąć wynosi 31 znaków. Sieć była trenowana przez 10 epok. Przy użyciu wcześniej wygenerowanego zbioru danych sieć w 8 epoce osiągnęła najmniejszy błąd walidacyjny. Przy wykorzystaniu danej bazy sprzętowej czas trenowania jednej epoki trwał około 3310s co przekładało się na 56ms/krok. Więcej wartości parametrów przedstawia tabela 4.1.

Tabela 4.1. Parametry sieci po 8 epokach szkolenia.

Parametr	Wartość
Wartość funkcji straty walidacyjnej	0.02566
Wartość funkcji straty treningowej	0.324
Dokładność treningowa	0.9867
Dokładność walidacyjna	0.9892

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 32, 128, 1)]      0

conv2d (Conv2D)              (None, 32, 128, 64)       640

max_pooling2d (MaxPooling2D  (None, 16, 64, 64)        0
)

conv2d_1 (Conv2D)            (None, 16, 64, 128)       73856

max_pooling2d_1 (MaxPooling  (None, 8, 32, 128)        0
2D)

conv2d_2 (Conv2D)            (None, 8, 32, 256)        295168

conv2d_3 (Conv2D)            (None, 8, 32, 256)        590080

max_pooling2d_2 (MaxPooling  (None, 4, 32, 256)        0
2D)

conv2d_4 (Conv2D)            (None, 4, 32, 512)        1180160

batch_normalization (BatchN  (None, 4, 32, 512)        2048
ormalization)

conv2d_5 (Conv2D)            (None, 4, 32, 512)        2359808

batch_normalization_1 (Batc  (None, 4, 32, 512)        2048
hNormalization)

max_pooling2d_3 (MaxPooling  (None, 2, 32, 512)        0
2D)

conv2d_6 (Conv2D)            (None, 1, 31, 512)        1049088

lambda (Lambda)              (None, 31, 512)           0

bidirectional (Bidirectiona  (None, 31, 256)           656384
l)

bidirectional_1 (Bidirectio  (None, 31, 256)           394240
nal)

dense (Dense)                (None, 31, 66)            16962

=================================================================
Total params: 6,620,482
Trainable params: 6,618,434
Non-trainable params: 2,048
_________________________________________________________________
```

Rysunek 4.2. Podsumowanie skompilowanego modelu CRNN.

Do skompilowania modelu jako funkcję strat wykorzystano CTC [15], a jako optymalizator wybrałem Adam z prędkością uczenia ustawioną na 0.0001. Śledzoną metryką podczas treningu pozostała dokładność. Jako że jedna epoka trwała średnio prawie godzinę do treningu dodałem parametr pozwalający na wcześniejsze zatrzymanie procesu nauki (ang. *early stopping*). Ustawiony on został na monitorowanie wartości funkcji straty walidacyjnej w trybie minimum oraz cierpliwości (patience) 5. Takie ustawienie powodowało zakończenie treningu, w razie wypadku, gdy sieć poprzez 5 epok nie poczyniła postępu poprzez poprawienie minimalnej wartości straty walidacyjnej. Został wykorzystany również parametr checkpoint, który został ustawiony w taki sposób, aby za każdym razem gdy sieć osiągnie mniejszą stratę walidacyjną zapisywał wagi modelu do pliku. Sieć uczę przy pomocy własnoręcznie napisanej klasy generatora wsadowego. Umożliwia mi to uczenie sieci 4 milionami danych, które normalnie przy wczytaniu nie zmieściłyby się w niemalże żadnej pamięci karty graficznej.

Rozdział 5. Eksperymenty, testy, rezultaty, wnioski.

5.1. Dane testowe

Zbiór czystych danych testowych przygotowałem poprzez odpowiednią obróbkę plików w formacie pdf. Za pomocą darmowych narzędzi internetowych przyciąłem marginesy do tekstu, a następnie każdą stronę A4 podzieliłem na pół i ponownie po przycinałem marginesy. W taki sposób przygotowałem cały zbiór testowy do procesu OCR. Do tak przygotowanych obrazów wykonałem adnotacje.

Zbiór zeskanowanych danych testowych przygotowałem poprzez wykonanie zrzutów ekranu fragmentów skanów książki i wykonanie adnotacji do zrobionych skanów. Dane testowe zostały przygotowane w następujący sposób ponieważ sieć wykorzystana do segmentacji na linie była szkolona na obrazach o kształcie przypominającym stronę A5.

Przy porównaniu wynikowych prognoz z tekstami wykorzystałem znormalizowane wartości. Szczegółowe dane o stworzonych zbiorach danych zawiera tabela 5.1. W procesie normalizacji z wyników otrzymanych w procesie OCR oraz etykiet zostały usunięte puste linie oraz nadmiarowe białe znaki.

Tabela 5.1. Statystyki stworzonych zbiorów danych.

Testowy zbiór danych	Liczba znaków	Liczba wyrazów	Liczba wierszy	Przykładowy wycinek danych
PDF	9474	2044	203	St Bertrand de Comminges is
Skany	5829	1353	145	"That will bring somebody."

5.2. Porównanie wyników z Tesseract OCR.

Do przeprowadzenia porównania skuteczności sieci z popularnym silnikiem OCR wybrałem Tesseract OCR [3] w najnowszej na ten czas wersji 5.1.0. Jest to darmowy silnik OCR wspierany przez Google, który osiąga jedne z najwyższych wyników dokładności.

Jako interfejs wykorzystałem pytesseract. Pozwala mi on na wykorzystywanie zainstalowanego Tesseract OCR w skryptach języka Python. W celu sprawdzenia wydajności załadowałem obrazy testowe do obu silników. Dla każdego silnika napisałem prosty skrypt, który wczytuje obraz, a następnie przeprowadza optyczne rozpoznawanie znaków i zapisuje wynik procesu optycznego rozpoznawania znaków w pliku tekstowym.

Tesseract osiąga zdecydowanie wyższe wartości niż przeszkolony przeze mnie silnik. W przypadku zbioru danych przygotowanego z PDF Tesseract osiąga 99,79% dokładności ,a stworzony silnik 90,73% dokładności. Zdecydowanie gorzej silnik wypada na zbiorze danych przygotowanym ze skanów. Tutaj różnica dokładności wynosi ponad 45% przy 98,38% silnika wspieranego przez Google i 51,70% wyuczonego silnika. Opisane dane przedstawia tabela 5.2

Tabela 5.2. Wyniki testu obu silników.

Silnik	Zbiór danych	Suma niewytrenowanych znaków w zbiorze danych	Dokładność
Tesseract	PDF	Brak	99,79%
	skany	Brak	98,38%
CRNN	PDF	81	90,73%
	skany	138	51,70%

5.3. Wnioski

Sieć CRNN wyszkoloną na danych syntetycznych można skutecznie zastosować na danych rzeczywistych. Przyczyną słabych wyników sieci jest słaba skuteczność sieci odpowiedzialnych za segmentację oraz brak uwzględnienia podczas treningu wszystkich znaków użytych w zbiorze testowym. Pomimo tego sieć osiągnęła ponad 50% dokładności na skanach książki wydrukowanej w roku 1916, której tekst jest wydrukowany w nie wyuczonej przez sieć oraz dość zniekształconej czcionce.

Na zbiorze danym powstałym z pliku pdf silnik osiąga dużo lepsze wyniki. Dane

bardziej przypominają te, na których sieć była trenowana. Zdecydowanie mniejszą część znaków stanowią te niewyszkolone. W tym teście ponownie największą wadą były sieci odpowiedzialne za segmentacje. W ten sposób najwięcej niewłaściwych wyników zostało źle sklasyfikowanych.

Podsumowanie

Wszystkie cele pracy jakimi było sprawdzenie skuteczności modeli rozpoznawania tekstu trenowanych na syntetycznych danych oraz przeuczenie sieci zaimplementowanego silnika OCR zostały zrealizowane. Wytrenowałem trzy sztuczne sieci neuronowe. Dwie o architekturze U-net i jedna o architekturze CRNN, którą wytrenowałem przy wykorzystaniu napisanej klasy generatora wsadowego. Następnie połączyłem je w całość przy pomocy gotowych skryptów w działający silnik, który przetestowałem na własnoręcznie przygotowanych danych.

Otrzymane wyniki pokazują niedoskonałość wykorzystanego przeze mnie rozwiązania. W celu poprawienia działania silnika należy poprawić moduł odpowiedzialny za segmentację. Lepszym rozwiązaniem zamiast sieci U-net byłoby zastosowanie lżejszego modelu detekcji tekstu jakim jest na przykład *differential binarization*. Kolejnym słabym elementem tego silnika jest generator. Wykorzystałem bardzo prosty generator danych, który sam zmodyfikowałem. Zdecydowanie lepszym pomysłem byłoby zastosowanie jednego z wielu dostępnych zaawansowanych generatorów na GitHub. Inną propozycją na zwiększenie rezultatu jest znalezienie lepszych hiperparametrów ustawionych podczas treningu sieci. Wzrost wydajności powinno zapewnić zastosowanie CRNN do rozpoznawania całych linii tekstu, a nie pojedynczych wyrazów.

Bibliografia

[1] Rosebrock A., Thanki A., Paul S., Haase J., (2020), PyImageSearch, OCR with OpenCV, Tesseract, and Python.

[2] TextOCR: Towards large-scale end-to-end reasoning for arbitrary-shaped scene text https://arxiv.org/pdf/2105.05486v1.pdf, (czas dostępu 06.2022).

[3] User manual Tesseract OCR https://tesseract-ocr.github.io/tessdoc/, (czas dostępu 06.2022).

[4] K El G Mohammed and C Amazighe 2015 Training TESSERACT Tool for Amazigh https://www.researchgate.net/publication/277142272_Training_TESSERACT_Tool_for_Amazigh_OCR, (czas dostępu 06.2022).

[5] An End-to-End Trainable Neural Network for Image-based Sequence Recognition and Its Application to Scene Text Recognition https://arxiv.org/pdf/1507.05717.pdf, (czas dostępu 06.2022).

[6] Dropout: A Simple Way to Prevent Neural Networks from Overfitting, https://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf, (czas dostępu 06.2022).

[7] Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift, https://arxiv.org/pdf/1502.03167.pdf, (czas dostępu 06.2022)

[8] Dokumentacja biblioteki Keras https://keras.io, (czas dostępu 06.2022).

[9] Sewak M., Karim Md. Rezaul., Pujari P., (2018), Packt Publishing, Practical Convolutional Neural.

[10] Fully Convolutional Networks for Semantic Segmentation, https://arxiv.org/pdf/1411.4038.pdf, (czas dostępu 06.2022).

[11] U-Net: Convolutional Networks for Biomedical Image Segmentation, https://arxiv.org/pdf/1505.04597.pdf, (czas dostępu 06.2022).

[12] Salem M. F., (2021), Springer, Recurrent Neural Networks.

[13] Long Short-term Memory, https://www.researchgate.net/publication/13853244_Long_Short-term_Memory, (czas dostępu 06.2022).

[14] Understanding LSTM Networks, https://colah.github.io/posts/2015-08-Understanding-LSTMs/,(czas dostępu 06.2022).

[15] Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks, https://www.cs.toronto.edu/~graves/icml_2006.pdf, (czas dostępu 06.2022).

[16] Detecting Text-lines in a Document Image Using Deep Learning, https://medium.com/geekculture/detecting-text-lines-in-a-document-image-using-deep-learning-5a21b480bc4c, (czas dostępu 06.2022).

[17] Oficjalna strona biblioteki OpenCV https://opencv.org, (czas dostępu 06.2022).

[18] Repozytorium zawierajace implementację wykorzystanego modelu U-net, https://github.com/VikasOjha666/Textline-Segmentation-using-UNet, (czas dostępu 06.2022).

[19] Źródło wykorzystanej grafiki modelu CRNN, https://medium.com/geekculture/building-a-complete-ocr-engine-from-scratch-in-python-be1fd184753b, (czas dostępu 06.2022).

[20] Repozytorium zawierające wykorzystany skrypty generujący obrazy z tekstem, https://github.com/VikasOjha666/Data-generator-for-CRNN, (czas dostępu 06.2022).