



UMCS

**UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
W LUBLINIE**
Wydział Matematyki, Fizyki i Informatyki

Kierunek: **informatyka**

Specjalność: –

Rafał Hrabia

nr albumu: 296583

**Rozpoznawanie liter języka migowego z
zastosowaniem technik uczenia
maszynowego**

Sign Language Letter Recognition using Deep Learning

Praca licencjacka
napisana w Katedrze Cyberbezpieczeństwa
pod kierunkiem dr hab. inż. Michała Wydry

Spis treści

Wstęp	4
Cel i zakres pracy	5
1 Język migowy American Sign Language	6
1.1 Historia	6
1.2 Populacja	7
1.3 American Manual Alphabet	8
2 Konwolucyjne sieci neuronowe i rozpoznawanie obrazów	10
2.1 Zasada działania sieci konwolucyjnych (CNN)	10
2.2 Architektura CNN	10
2.2.1 Warstwy konwolucyjne	10
2.2.2 Warstwy Pooling	13
2.2.3 Warstwa Flatten	14
2.2.4 Warstwy Dense	15
2.3 Zastosowania	15
2.3.1 MNIST	16
3 Implementacja aplikacji do rozpoznawania języka migowego ASL	20
3.1 Stworzenie zbioru obrazów	20
3.2 Budowa modelu	22
3.2.1 Generatory i argumentacja danych	22
3.2.2 Architektura modelu	24
3.2.3 Trening modelu	26
3.3 Aplikacja do rozpoznawania języka migowego w czasie rzeczywistym . . .	28
3.4 Analiza błędów	30
4 Podsumowanie	32
Spis tabel	33

Spis rysunków	34
Spis listingów	35
Bibliografia	35

Wstęp

Sieci neuronowe mają za zadanie naśladowanie zachowań sieci neuronów znajdujących się w mózgu człowieka. Zostały stworzone do rozwiązywania zadań trudnych lub prawie niemożliwych do opisania za pomocą reguł, wyrażeń logicznych i innych narzędzi programistycznych. Wraz z rozwojem sieci neuronowych powstało wiele wariantów, które ze względu na swoją budowę lepiej lub gorzej sprawdzają się w różnych problemach. W przypadku rozpoznawania obrazów w postaci dwu-wymiarowej macierzy dla danych monochromatycznych lub trój-wymiarowej macierzy dla zdjęć kolorowych jednym z najlepszych wyborów będą sieci konwolucyjne. Sieci te rozpoznają wzorce, poczynając od linii horyzontalnych i wertykalnych, a w dalszych warstwach kończąc na skomplikowanych strukturach. Budowa i działanie sieci konwolucyjnych daje wielki potencjał do klasyfikacji obrazów. Niniejsza praca licencjacka przedstawia przykład wieloklasowej klasyfikacji tj. rozpoznawania liter alfabetu ASL przy wykorzystaniu sieci konwolucyjnych.

Cel i zakres pracy

Celem pracy jest stworzenie programu wyposażonego w wytrenowany model do rozpoznawania obrazu, który w czasie rzeczywistym używając kamery internetowej będzie w stanie odczytać i wyświetlić na ekranie transkrypcję znaków języka migowego pokazywanych przez osobę znajdującą się w polu widzenia kamery. Dodatkowo, w celu osiągnięcia celu pracy, zrealizowano następujące zadania: utworzenie zbioru danych składającego się z około 50000 zdjęć zawierających wszystkie litery alfabetu ASL, utworzenie modelu z warstw konwolucyjnych i gęstych oraz wytrenowanie modelu i tuning parametrów. W teoretycznej części pracy przybliżono temat języka migowego American Sign Language, jak również temat konwolucyjnych sieci neuronowych i sposobu działania modeli.

Rozdział 1

Język migowy American Sign Language

American Sign Language (Amerykański Język Migowy, ASL) to złożony język wizualno-przestrzenny używany przez osoby niesłyszące w Stanach Zjednoczonych Ameryki oraz anglojęzycznych częściach Kanady. Jest to w pełni kompletny język naturalny. ASL jest językiem natywnym dla wielu mężczyzn, kobiet i dzieci, a także niektórych słyszących dzieci w rodzinach, gdzie opiekunowie prawni są niesłyszący [1].

1.1 Historia

Pochodzenie dzisiejszej społeczności osób niesłyszących w Stanach Zjednoczonych jest powszechnie utożsamiane z założeniem pierwszej szkoły dla niesłyszących - American School for the Deaf (ASD), założonej w 1817 roku w Hartford, Connecticut. Przed założeniem szkoły ASD na terenie USA działało wiele niezależnych społeczności, poczynając od małych grup o wielkości pojedynczej rodziny do większych - całych wsi. W małych społecznościach uformowały się niezależne znaki i systemy języka migowego, które są obecne po dziś dzień w tych środowiskach. Istnieją dowody na to, że niesłyszące dzieci kształtowały swoje własne systemy języków migowych, które były o wiele bardziej wyrafinowane od tych, używanych w społeczności, w których się znajdowały [2].

Istnieją również doniesienia o innym niezależnie uformowanym systemie języka migowego Martha's Vineyard Sign Language (MVSL), który istniał przed założeniem American School for the Deaf. Język ten był głównie używany w wsi Chilmark na wybrzeżu Massachusetts. Powstanie MVSL zapoczątkował fakt, że wspomniana społeczność Chilmark miała wysoki odsetek mutacji genetycznych prowadzących do głuchoty. W skali miasteczka około 4 procent mieszkańców było niesłyszących. Wynikało to z wysokiego odsetka mieszanych małżeństw od wielu pokoleń, począwszy od hrabstwa Kent w Anglii, zanim wieś Chilmark została założona w roku 1690 [3]. Mieszkańcy, którzy nie byli

niesłyszący również posługiwali się językiem MVSL, wówczas gdy znajdowali się w towarzystwie osób z niepełnosprawnością ale również, gdy w gronie rozmówców nie było osoby niesłyszącej. Język był używany do czasu założenia szkoły dla osób niesłyszących ASD w Hartford. Dzieci z wsi Chilmark zaczęto wysyłać do szkoły American School for the Deaf we wczesnych latach dwudziestych XIX wieku. Skutkowało to zatarciem się języków MVSL i nowego języka migowego ewoluującego w dzisiejszy język ASL [2].

Kiedy szkoła dla niesłyszących została założona, stała się miejscem, gdzie wiele pomniejszych systemów migowych stykało i mieszało się ze sobą przez 175 lat. Z mieszkańców tych języków powstał dzisiejszy język ASL. Niesłyszący nauczyciel - Laurent Clerc, pochodzący z Francji, był pierwszym nauczycielem w wspomnianej placówce. Z kraju swojego pochodzenia przywiózł wiedzę o Francuskim Języku Migowym, którego znaków nauczał w amerykańskiej szkole. Ta sytuacja spowodowała bardzo mocne doprawienie wówczas powstającego języka ASL o aspekty zaciągnięte z Francuskiego Języka Migowego. Na dzień dzisiejszy około 60 procent współczesnego systemu ASL opiera się na starym migowym języku Francuskim [2].

1.2 Populacja

Informacje dostępne w internecie nie dają jednoznacznej odpowiedzi na pytanie - ile osób używa języka ASL. Wyszukane rezultaty są bardzo niejednoznaczne [4]. Znalezione szacunkowe (Tabela 1.1) liczby użytkowników języka ASL wahają się począwszy od 100 tysięcy do nawet 15 milionów. Warto zwrócić uwagę, że prawdopodobnie szacunki Aetna InteliHealth zostałyomyłkowo utożsamione z użytkownikami języka ASL, a tak naprawdę mogą przedstawiać liczbę osób ze znacznym i całkowitym ubytkiem słuchu [4].

Starając się uogólnić wszystkie szacunki płynące ze źródeł internetowych, kształtują się dwa główne twierdzenia:

- Jest mniej niż dwa miliony użytkowników ASL, ale bardziej prawdopodobne, że ta liczba jest mniejsza i wynosi mniej niż półtora miliona ludzi w Stanach Zjednoczonych Ameryki;
- ASL może być trzecim najczęściej występującym językiem w Stanach Zjednoczonych [4].

Zaprezentowane w tabeli 1.1 dane pochodzą głównie z okresu lat dwutysięcznych. W trakcie pisania niniejszej pracy prawdopodobnie nie istniały bardziej aktualne i ogólnodostępne źródła.

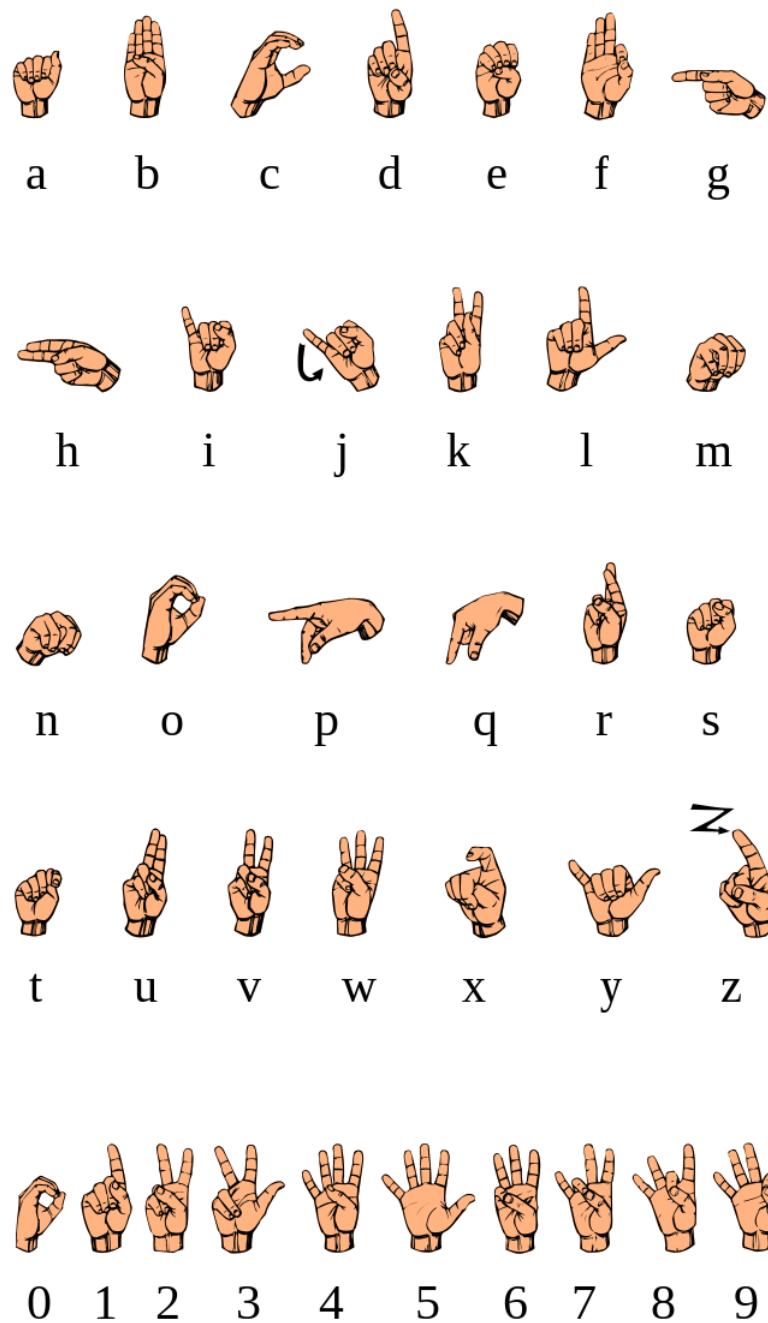
Tabela 1.1: Szacowane rankingi użytkowników ASL według różnych źródeł

Oszacowany ranking	Źródło
100,000 – 500,000	ERIC Digests (Wilcox & Peyton, 1999) MSN Encarta (Wilcox, 2004) Ethnologue.com (Ethnologue, 2004)
250,000 – 500,000	American Sign Language Program @ The University of Iowa (Department of Speech Pathology and Audiology, 2004) ASLTA (NC ASLTA and NCAD Ad Hoc Committee, 2004) Colorado Department of Human Services (Colorado Commission for the Deaf and Hard of Hearing, n.d.)
300,000 – 500,000	Barnes&Noble.com (Costello, 1994) SignWriting.org (Rosenberg, 1999)
500,000	American Academy of Family Physicians (CDGAP, 1997) ASLinfo.com (ASLinfo.com, n.d.) DEAF C.A.N.! (Deaf Community Advocacy Network, n.d.)
500,000 – 2,000,000	Brenda Schick, Ph.D. (Schick, 1998) DawnSignPress (DawnSignPress, 2003) Gallaudet University Library (Harrington, 2004)
15,000,000	Aetna InteliHealth (Gordon, 2001)
3rd most used language in the U.S.	HandSpeak (HandSpeak.com, n.d.) Health Literacy Consulting (Osborne, 2003) Missouri Office of State Courts Administrator (Office of State Courts Administrator, n.d.)
4th most used language in the U.S.	The ASHA Leader Online (Scott & Lee, 2003) Deaf Resource Library (Nakamura, 2002) NIDCD (National Institute on Deafness and Other Communication Disorders, 2000)
3rd to 10th most used language in the U.S.	Wikipedia (Wikimedia, 2003)

1.3 American Manual Alphabet

American Manual Alphabet to zbiór różnych gestów dlonią reprezentujący wszystkie litery angielskiego alfabetu. Są one ważnym elementem komunikacji i uzupełniają American Sign Language o zestaw znaków pozwalających na wyrażenie słów, dla których rozmówca nie zna konkretnego znaku lub nazw własnych, takich jak imiona czy nazwy firm. Często używane są przez użytkowników ASL dla upewnienia się, że idea tego co chcieli powiedzieć została dokładnie i zrozumiale zakomunikowana drugiej stronie rozmowy. Początkujący użytkownicy języka ASL, z racji ograniczonego zbioru znaków często posługują się alfabetem migowym do komunikowania słów, dla których nie znają

odpowiadających znaków [5].



Rysunek 1.1: American Manual Alphabet [6]

W alfabetie znajduje się 26 znaków odpowiadających literom języka angielskiego. Na odpowiedniki liter składa się 19 ułożień dłoni. Niektóre ułożenia występują kilkukrotnie tak jak litery "i" i "j" oraz "p" i "k". Różnica pomiędzy tymi znakami polega na ruchu dłonią (przy literze "j" występuje ruch w kształcie półkola) lub pozycji względem ciała (przy literze "k" dłoń jest skierowana palcami do góry, a przy literze "p" w dół lub w bok) [5].

Rozdział 2

Konwolucyjne sieci neuronowe i rozpoznawanie obrazów

Konwolucyjne sieci neuronowe pozwoliły na przełomowe osiągnięcia ostatniej dekady w dziedzinie rozpoznawania wzorców, zaczynając od obrazów a kończąc na rozpoznawaniu głosu. Jednym z wyróżników sieci CNN jest znaczna redukcja ilości parametrów w dalszej części modelu. Mniejsza ilość parametrów pozwoliła na budowanie większych sieci, które mogą rozwiązywać bardziej skomplikowane problemy niż dotychczasowe modele z użyciem zwykłych sieci neuronowych [7].

2.1 Zasada działania sieci konwolucyjnych (CNN)

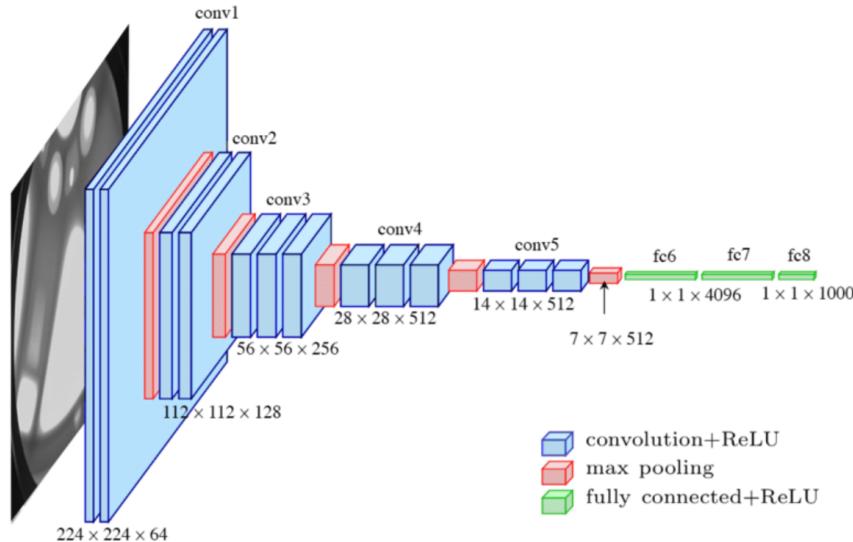
Model CNN na swoim wejściu posiada najczęściej naprzemiennie ustawione warstwy konwolucyjne i Pooling. Pozwala to na automatyczne wykrywanie cech charakterystycznych i bardzo szybkie zmniejszenie objętości przetwarzanych danych. Przetworzone dane obrazu wejściowego są spłaszczane do jednego wymiaru przez warstwę Flatten. Spłaszczone dane przyjmują kolejne warstwy gęsto połączone i dokonują ostatecznej klasyfikacji danych wejściowych do danej kategorii [8].

Przykładowy model został przedstawiony na rysunku 2.1.

2.2 Architektura CNN

2.2.1 Warstwy konwolucyjne

Warstwa konwolucyjna wykonuje prostą operację nałożenia filtra zachowującego się na wzór funkcji aktywacji (rys. 2.2). Przeprowadzenie tej operacji piksel po pikselu przez wiele filtrów tworzy mapy cech, które są później wykorzystywane w kolejnych warstwach modelu [10]. Warstwy konwolucyjne zamiast pełnego połączenia, gdzie każdy piksel ob-



Rysunek 2.1: Model CNN [9]

razu wejściowego jest połączony z każdym pikselem obrazu wyjściowego, stosuje lokalne pola receptive zmniejszając drastycznie liczbę połączeń (rys. 2.3).

Warstwy konwolucyjne pozwalają na jeszcze większe zmniejszenie ilości parametrów. Mowa o opcji ustawienia skoku pomiędzy polami receptivenymi - *stride*, które domyślnie zachodzą na siebie przy wartości parametru = 1. Po konfiguracji *stride* możliwe jest częściowe lub całkowite rozdzielenie poszczególnych węzłów (rys. 2.4) [7].

Jedną z wad warstw konwolucyjnych jest utrata informacji, która pojawia się na obrzeżach obrazu. Istnieje bardzo prosta i efektywna metoda do rozwiązania tego problemu - "zero-padding". Zaletą takiego rozwiązania, jest również zachowanie rozmiaru obrazu na wyjściu warstwy. Dla przykładu (rys. 2.5) z wartościami $N=7$, $F=3$ i parametrem $stride=1$, rozmiar danych wyjściowych wyniesie 5×5 , gdzie oryginalny rozmiar danych wejściowych wynosił 7×7 [7].

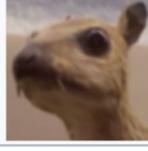
Wzór na obliczenie rozmiaru wyjściowego obrazu O dla danych wejściowych $N \times N$ i rozmiarze filtra $F \times F$, jest następujący:

$$O = 1 + \frac{N - F}{S} \quad (2.1)$$

Gdzie:

- N - rozmiar danych wejściowych,
- F - rozmiar filtra,
- S - wartość parametru *stride*.

Natomiast, dodając wypełnienie zerami rozmiar danych wyjściowych będzie wynosił 7×7 , taki sam jak danych wejściowych (rys. 2.6) [7].

Operation	Kernel ω	Image result $g(x,y)$
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Ridge detection	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur 3×3 (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Rysunek 2.2: Przykładowe działanie filtrów 3×3 na obrazach [11]

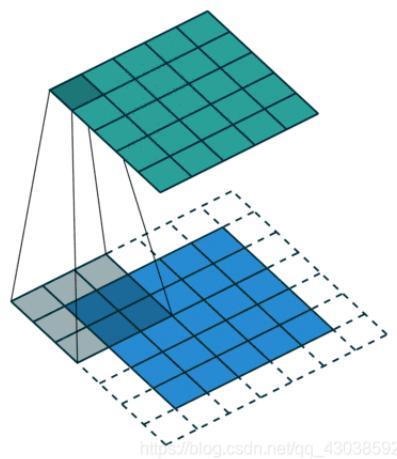
W przypadku uzupełniania zerami następuje lekka modyfikacja wzoru 2.1:

$$O = 1 + \frac{N + 2P - F}{S} \quad (2.2)$$

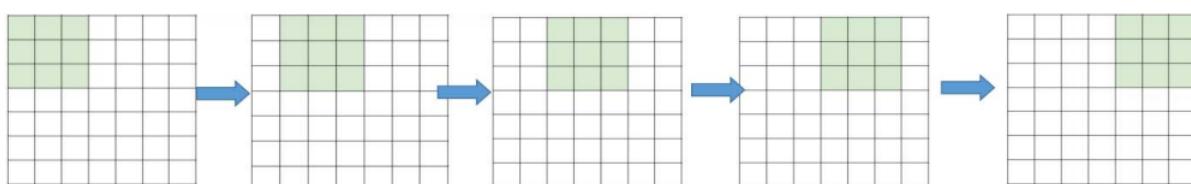
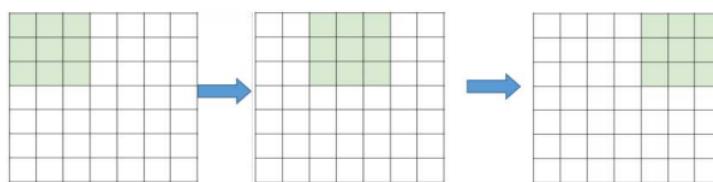
gdzie:

- N - rozmiar danych wejściowych,
- F - rozmiar filtra,
- S - wartość parametru *stride*,
- P - liczba warstw *zero-padding*.

Idea uzupełniania brzegów obrazu zerami pozwala na zachowanie rozmiaru obrazu w niezmienionej formie i zatracać informacji. Dzięki temu możliwe jest zastosowanie jakiejkolwiek liczby warstw konwolucyjnych [7].



Rysunek 2.3: Warstwa konwolucyjna - zasada działania [12]

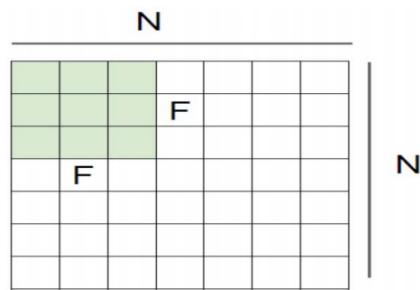
Stride = 1**Stride = 2**Rysunek 2.4: Różnica w odstępach pomiędzy polami recepcyjnymi w zależności od parametru *stride* [7]

2.2.2 Warstwy Pooling

Główną ideą działania warstwy *Pooling* jest redukcja wymiaru danych dla następnych warstw, starając się zachować jak najwięcej informacji zawartych w danych. W domenie przetwarzania obrazów może to być porównane do zmiany rozdzielczości obrazu, na którym dokonujemy przekształcenia. Warstwa ta nie wpływa na liczbę map cech wygenerowanych przez poprzednią warstwę konwolucyjną, tylko na ich rozmiar [7].

Max-pooling to jedna z najczęściej używanych metod. Dzieli ona obraz na regiony i zwraca maksymalne wartości występujące w danym regionie. Bardzo popularną wielkością regionów jest rozmiar 2x2 z parametrem *stride* = 2. Powoduje to, że regiony na siebie nie nachodzą i wyjściowy obraz o rozmiarze N (gdzie N jest liczbą parzystą) po dokonaniu na nim operacji będzie miał rozmiar $\frac{N}{2}$ [7].

Zasadę działania warstwy *max pooling* opisaną w poprzednim akapicie przedstawia

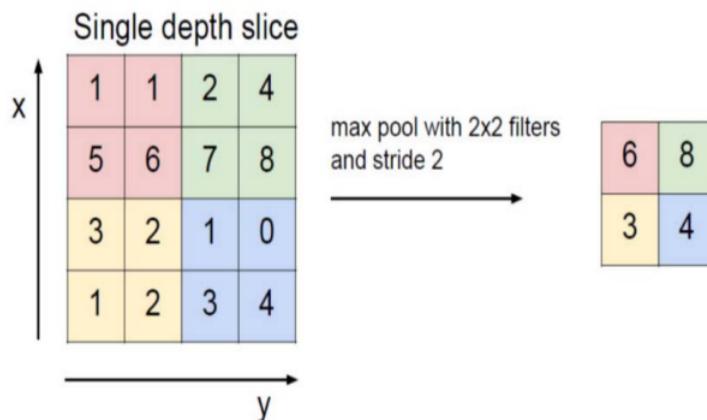


Rysunek 2.5: Dane bez zero-padding [7]

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Rysunek 2.6: Dane z zero-padding [7]

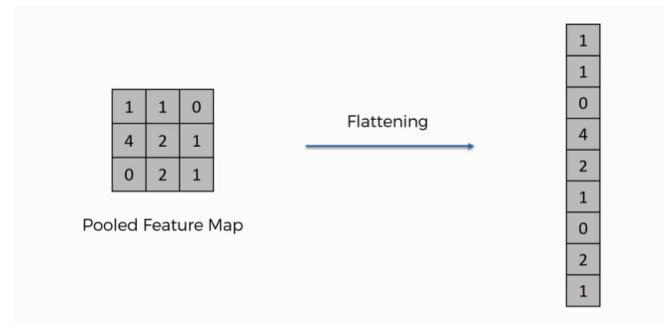
rys. 2.7.



Rysunek 2.7: Przykładowe działanie max pooling [7]

2.2.3 Warstwa Flatten

Warstwa *flatten* to warstwa buforowa pomiędzy wielowymiarowymi danymi pochodzącyymi z warstw konwolucyjnych i *pooling*, a warstwami klasycznych sieci neuronowych *Dense*. Jak nazwa opisywanej warstwy wskazuje - dokonuje ona spłaszczenia wielowymiarowych danych do jednego długiego wektora, który może zostać przekazany do następnych warstw, które na podstawie danych zawartych w tym wektorze sklasyfikują ostateczny wynik [13].

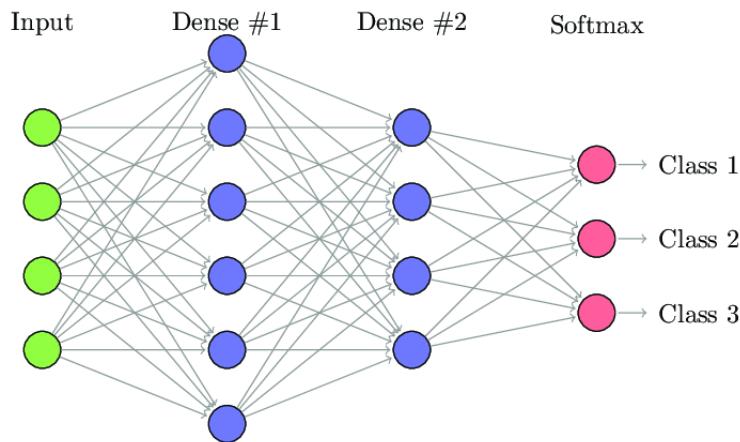


Rysunek 2.8: Przykładowe działanie warstwy flatten [13]

Przykład działania warstwy *flatten* został przedstawiony na rysunku 2.8.

2.2.4 Warstwy Dense

Warstwy *fully-connected* są podobne do tych znajdujących się w zwykłych sieciach neuronowych. Każdy węzeł jest bezpośrednio połączony z każdym węzłem z poprzedniej i następnej warstwy (rys. 2.9) [7].



Rysunek 2.9: Przykład warstw *fully-connected* [14]

Przy przetwarzaniu obrazów wprowadzane do sieci są bardzo duże ilości danych. Dzięki sekcji warstw konwolucyjnych i *pooling* następuje redukcja rozmiaru danych jakie trafiają do warstw *fully-connected*. Bez tego przetwarzanie obrazów byłoby niemożliwe lub bardzo wolne z powodu dużego zapotrzebowania warstw *fully-connected* na moc obliczeniową z racji na obecność wielu równoległych obliczeń w tych warstwach [7].

2.3 Zastosowania

Sieci *CNN* są używane w wielu istniejących aplikacjach. Można ich użyć wszędzie tam gdzie istnieje potrzeba analizy wzorców występujących w tekscie, dźwięku czy oczywiście

w obrazach. Najczęstsze zastosowania między innymi to:

- rozpoznawanie twarzy,
- rozpoznawanie emocji twarzy,
- rozpoznawanie obiektów,
- autonomiczne pojazdy,
- translacja języków,
- predykcja następnego słowa na podstawie kontekstu wpisywanego zdania,
- rozpoznawanie pisma odręcznego,
- analiza obrazów medycznych takich jak zdjęcia rentgenowskie,
- wykrywanie nowotworów,
- przetwarzanie zapytań w języku naturalnym,
- opisywanie obrazów,
- uwierzytelnianie na podstawie parametrów biometrycznych,
- klasyfikacja dokumentów,
- segmentacja trójwymiarowych obrazów medycznych [15].

Na potrzeby tej pracy w rozdziale 2.3.1 zostanie przedstawiony przykład klasyfikacji wieloklasowej.

2.3.1 MNIST

Zbiór *MNIST* zawiera w sobie 70000 obrazów odręcznie pisanych cyfr z zakresu 0-9. Problem klasyfikacji jest aktualnie przedstawiany jako trywialne i podstawowe zagadnienie w dziedzinie *AI*. Niemniej jednak jest on używany do celów nauki, jak również przez analityków jako próba spekulacji na temat obliczeń sztucznych sieci neuronowych [10].

Głównym wyzwaniem w zbiorze *MNIST* jest zróżnicowanie odręcznie pisanych cyfr pod względem wielkości, grubości, orientacji i innych czynników. Wynika to z tego, że cyfry były komponowane przez wiele osób z różnymi sposobami pisania [10]. Na rysunku 2.10 zostały przedstawione przykładowe cyfry ze zbioru *MNIST*. Czynnością, od której należy zacząć chcąc pracować z zbiorzem *MNIST* jest skorzystanie z interfejsu *Keras API* w celu pobrania zbioru danych. Zbiór *MNIST* jest podzielony na kilka części - część przeznaczoną do treningu modelu i część do ewaluacji stworzonej sieci. Wewnątrz zbiorów

Rysunek 2.10: Przykłady cyfr ze zbioru *MNIST* [16]

treninguowych i testowych dane rozdzielone są na obrazy o rozmiarze 28x28 i etykiety, które zawierają numer z zakresu 0-9 odpowiadający cyfrze, do której jest przypisany [17]. Poszczególne instrukcje zostały przedstawione na listingu 2.1.

```
1 import tensorflow as tf
2 (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.
3     load_data()
```

Listing 2.1: Pobieranie zbioru *MNIST* [17]

W celu użycia zbioru danych w modelu niezbędna jest zmiana wymiarów tablic, w których znajdują się obrazy w celu dodatnia jednego dodatkowego wymiaru. Jest to wymóg użytego API, gdzie ostatni wymiar jest przeznaczony na jedną lub więcej cech danych. Następnie wartości dla poszczególnych pikseli obrazów, które domyślnie znajdują się w zakresie od 0 do 255 muszą zostać znormalizowane do zakresu 0.0-1.0. Najprościej osiągnąć to poprzez operację dzielenia przez 255 wszystkich komórek (listing 2.2) [17].

```
1 x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
2 x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
3 input_shape = (28, 28, 1)
4
5 x_train = x_train.astype('float32')
6 x_test = x_test.astype('float32')
7
8 x_train /= 255
9 x_test /= 255
```

Listing 2.2: Zmiana rozmiarów tablic i normalizacja [17]

Model do klasyfikacji cyfr na wejściu składa się z jednej warstwy konwolucyjnej posiadającej 28 filtrów i warstwy *MaxPooling*, która zmniejszy rozmiar danych dwukrotnie.

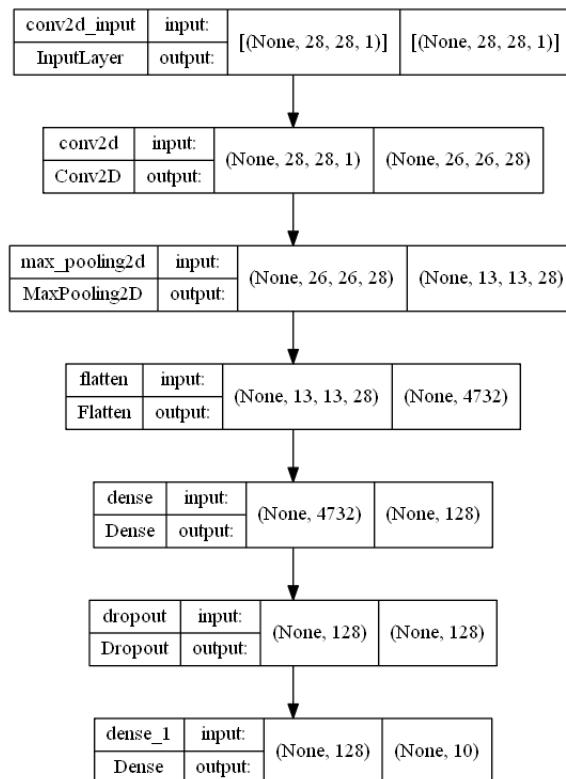
Następnie po spłaszczeniu danych do postaci jednowymiarowego wektora, dane przekazywane są do warstw *Dense*, kolejno do warstwy posiadającej 128 węzłów (neuronów) a następnie do warstwy wyjściowej. Warstwa wyjściowa posiada 10 węzłów czyli dokładnie tyle co klas, które model ma przypisywać. Funkcja aktywacji *softmax* pozwoli na odczyt wyników w formie rozkładu prawdopodobieństwa (listing 2.3) [17].

```

1  from tensorflow.keras.models import Sequential
2  from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten,
   MaxPooling2D
3  model = Sequential()
4  model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))
5  model.add(MaxPooling2D(pool_size=(2, 2)))
6  model.add(Flatten())
7  model.add(Dense(128, activation=tf.nn.relu))
8  model.add(Dropout(0.2))
9  model.add(Dense(10, activation=tf.nn.softmax))

```

Listing 2.3: Budowa modelu [17]



Rysunek 2.11: Wizualizacja modelu

Model stworzony w listingu 2.3 musi zostać skompilowany. Zostaną do niego dodane optymalizator, funkcja straty i metryki. Tak skompilowany model może zostać przedstawiony do treningu (listing 2.4) [17].

```

1  model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
   , metrics=['accuracy'])

```

```
2 model.fit(x=x_train,y=y_train, epochs=10)
```

Listing 2.4: Kompilacja i trening modelu [17]

Wytrenowany model może zostać poddany ewaluacji na testowych danych (listing 2.5). Model osiąga wyniki w okolicach 98%-99% poprawności klasyfikacji [17].

```
1 model.evaluate(x_test, y_test)
```

Listing 2.5: Ewaluacja modelu [17]

Rozdział 3

Implementacja aplikacji do rozpoznawania języka migowego ASL

Celem niniejszej pracy jest stworzenie aplikacji, która pozwoli na rozpoznawanie części języka ASL - *ASL Manual Alphabet* w czasie rzeczywistym. Program będzie rozpoznawał 26 znaków odpowiadających pojedynczym literom. Aplikacja jako źródło obrazu wykorzystywać będzie kamerę internetową znajdująca się na ekranie monitora. Rozpoznawane znaki będą akceptowane automatycznie przez program i wyświetlane na ekranie po określonej ilości czasu.

3.1 Stworzenie zbioru obrazów

Z racji słabej jakości zbiorów danych dostępnych w ogólnodostępnych źródłach, takich jak portal *Kaggle.com*, na potrzeby tej pracy został stworzony autorski zbiór około 51821 zdjęć przedstawiających 26 znaków z języka *American Sign Language*.

Przeprowadzone testy na zbiorach danych pobranych ze strony z *Kaggle.com* [18][19] nie spełniały oczekiwania. Stworzony model miał problemy z treningiem i stabilnością. Było to spowodowane bardzo złą jakością większości zdjęć ze zbioru. Oświetlenie gra kluczową rolę w dziedzinie rozpoznawania obrazu. Wspomniane zdjęcia były w ogólnym pojęciu bardzo ciemne ze względu na prawdopodobny brak jakiegokolwiek źródła światła, a czasem wręcz zdjęcie zostało zrobione ”pod” światło (rys. 3.1).

Dobra jakość zdjęcia jest niezbędna przy próbie wykrywania znaków języków migowych. Często znaki te różnią się pozycją jednego palca. Model nie był w stanie zauważać takich zmian nie widząc wyraźnych krawędzi pomiędzy poszczególnymi częściami dloni użytkownika.

Ważnym aspektem było też zachowanie realizmu przy tworzeniu zbioru danych. Zdjęcia zawierają nie tylko dłoń na w miarę jednolitym tle, ale również częściowo postać użytkownika i fragmenty niejednolitego otoczenia za nim. Dłoń użytkownika również w



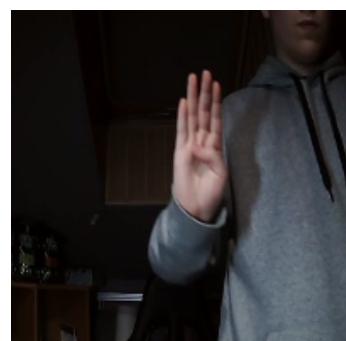
Rysunek 3.1: Słabej jakości zdjęcie znaku ASL ze zbioru danych opublikowanych na *Kaggle.com* [18]

realistycznych warunkach będzie znajdowała się w różnych odległościach, pozycjach na ekranie i pod różnymi kontami względem kamery. W takich warunkach model będzie eksploatowany w stworzonej aplikacji więc racjonalnym podejściem było wprowadzić te zmiany już na etapie treningu.

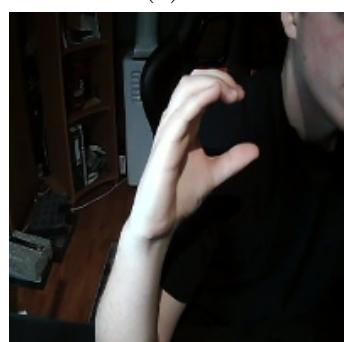
Przykładowe zdjęcia ze stworzonego zbioru danych znajdują się na rys. 3.2.



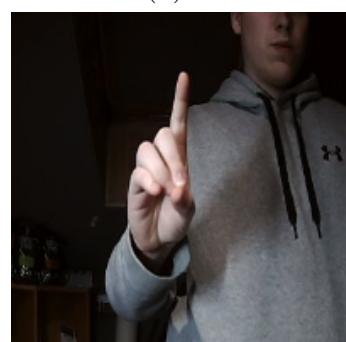
(a) A



(b) B



(c) C



(d) D

Rysunek 3.2: Zestawienie przykładowych obrazów z autorskiego zbioru liter języka ASL

3.2 Budowa modelu

Pierwszą czynnością prowadzącą do budowy aplikacji jest przygotowanie danych oraz zbudowanie i wytrenowanie modelu, który będzie wykorzystywany w aplikacji.

Wszystkie wykorzystywane biblioteki do kolejnych podrozdziałów zostały zawarte na listingu 3.1.

```
1 # %% Imports
2 from pathlib import Path
3
4 import cv2
5 import matplotlib.pyplot as plt
6 import numpy as np
7 import tensorflow as tf
8 from keras_preprocessing.image import ImageDataGenerator
```

Listing 3.1: Importowane biblioteki

Budowa modelu CNN będzie odbywać się z wykorzystaniem biblioteki *Keras*, która znajduje się w bibliotece *tensorflow*. Zawiera ona narzędzia niezbędne do tworzenia, trenowania i wykorzystywania modeli uczenia maszynowego. Dane, których będziemy używać w celu treningu modelu są w formacie zdjęć, więc rozsądny jest użycie klasy *ImageDataGenerator* z biblioteki *keras-preprocessing* w celu wygodnego wczytywania obrazów podczas uczenia.

3.2.1 Generatory i argumentacja danych

Generatory to bardzo popularna metoda wczytywania danych prosto z folderów zawierających zbiór zdjęć. Generator z *Keras API* automatycznie dzieli wczytane obrazy na klasy jeśli są umieszczone w określonej strukturze folderów na dysku [20]. Pozwalają również na bardzo prostą augmentację danych poprzez różne przekształcenia obrazu, które są losowo aplikowane na wczytywane obrazy. Jest to bardzo dogodne podejście, znacznie wygodniejsze niż wczytywanie wszystkich obrazów do pamięci operacyjnej na raz.

W celu ułatwienia odwoływanego się do poszczególnych ścieżek na dysku dobrą praktyką jest zdefiniowanie zmiennych przechowujących te ścieżki za pomocą biblioteki *pathlib* (listing 3.2).

```
1 # %% var
2 data_path = Path('data/raw')
3 train_path = data_path.joinpath('asl-own')
4 models_path = Path('models')
```

Listing 3.2: Definicja ścieżek do danych i modeli

Obiekty generatorów pozwalają na proste podzielenie danych na podzbiory treningowe i walidacyjne. Zbiór walidacyjny posłuży do lepszej oceny sprawności modelu w

trakcie uczenia. Oba generatory normalizują wartości pikseli we wczytywanych obrazach do zakresu od 0 do 1 dzięki parametrowi *rescale*. Na danych treningowych w momencie wczytywania zostanie przeprowadzona prosta argumentacja obrazów składająca się z odbicia zdjęć w płaszczyźnie horyzontalnej i losowego przybliżania lub oddalania zdjęcia, w skutek czego sieć neuronowa za każdym razem będzie otrzymywała lekko inny obraz (listing 3.3).

```
1 # %% generators
2
3 data_generator = ImageDataGenerator(
4     horizontal_flip=True,
5     fill_mode='nearest',
6     rescale=1 / 255.0,
7     zoom_range=0.2,
8     validation_split=0.1
9 )
10 valid_generator = ImageDataGenerator(
11     rescale=1 / 255.0,
12     validation_split=0.1
13 )
14
15 train_gen = data_generator.flow_from_directory(
16     directory=train_path,
17     target_size=(200, 200),
18     color_mode="rgb",
19     batch_size=64,
20     class_mode="sparse",
21     seed=2022,
22     subset="training"
23 )
24 valid_gen = valid_generator.flow_from_directory(
25     directory=train_path,
26     target_size=(200, 200),
27     color_mode="rgb",
28     batch_size=64,
29     class_mode="sparse",
30     seed=2022,
31     subset="validation"
32 )
33 classes = list(train_gen.class_indices.keys())
34 print(classes)
35
```

Listing 3.3: Definicja generatorów

Generatory można przetestować ręcznie i wyświetlić wczytywane obrazy w sposób przedstawiony na listingu 3.4 za pomocą biblioteki *OpenCV* [21].

```
1 # %% test generators
2 def decode_class(cls, one_hot):
3     return cls[np.argmax(one_hot)]
4
5 for i in range(10):
6     image, label = next(train_gen)
7     print(decode_class(classes, label))
8     image = image[0] * 255.0
9     image = image.astype('uint8')
10    image = np.squeeze(image)
11    cv2.imshow('i', image)
12    cv2.waitKey(0)
13    cv2.destroyAllWindows()
```

Listing 3.4: Wczytanie i wyświetlenie pierwszych 10 zdjęć za pomocą ImageDataGenerator

Na tym etapie generatory są gotowe do użycia w sieci neuronowej. Użycie klasy *ImageDataGenerator* pozwala na bardzo wygodną augmentację danych i doczytywanie zdjęć w trakcie treningu modelu, a nie wczytania całości zbioru danych do pamięci operacyjnej przed treningiem, co w przypadku większych zbiorów zdjęć byłoby zadaniem niemożliwym z racji ograniczonych zasobów pamięci RAM.

3.2.2 Architektura modelu

Przedstawiony model w listingu 3.5 i jego struktura została wyłoniona jako najlepsza z testowanych konfiguracji sieci w procesie tworzenia aplikacji. Zastosowane zostały warstwy *Conv2D* występujące w parach, pomiędzy którymi znajduje się warstwa *MaxPooling2D*. Warstwy konwolucyjne posiadają standardowy rozmiar jądra 3x3 i funkcję aktywacji *relu*. W kolejnych warstwach zastosowano rosnącą dwukrotnie, w stosunku do poprzedniej pary, liczbę filtrów.

```
1 # %% Build model
2 model = tf.keras.models.Sequential()
3 model.add(tf.keras.layers.Conv2D(32, (3, 3), activation="relu",
4     padding="same", input_shape=(200, 200, 3)))
5 model.add(tf.keras.layers.Conv2D(32, (3, 3), activation="relu",
6     padding="same"))
7 model.add(tf.keras.layers.MaxPooling2D(3, 3))
8
9 model.add(tf.keras.layers.Conv2D(64, (3, 3), activation="relu",
10    padding="same"))
11 model.add(tf.keras.layers.Conv2D(64, (3, 3), activation="relu",
12    padding="same"))
13 model.add(tf.keras.layers.MaxPooling2D(3, 3))
```

```
11 model.add(tf.keras.layers.Conv2D(128, (3, 3), activation="relu",
12     padding="same"))
13 model.add(tf.keras.layers.Conv2D(128, (3, 3), activation="relu",
14     padding="same"))
15 model.add(tf.keras.layers.MaxPooling2D(3, 3))
16
17 model.add(tf.keras.layers.Conv2D(256, (3, 3), activation="relu",
18     padding="same"))
19 model.add(tf.keras.layers.Conv2D(256, (3, 3), activation="relu",
20     padding="same"))
21
22 model.add(tf.keras.layers.Flatten())
23
24 model.add(tf.keras.layers.Dense(1500, activation="relu"))
25 model.add(tf.keras.layers.Dropout(0.5))
26
27 model.add(tf.keras.layers.Dense(26, activation="softmax"))

opt = tf.keras.optimizers.Adam(learning_rate=0.0001)
model.compile(optimizer=opt, loss="sparse_categorical_crossentropy",
metrics=['accuracy'])
model.summary()
```

Listing 3.5: Definicja modelu

Stworzony model charakteryzuje się wysoką liczbą (20 028 782) trenowalnych parametrów, głównie z powodu dużej liczby połączeń w przedostatniej warstwie (listing 3.6). Pomimo znacznej liczby parametrów model jest wystarczająco szybki aby przetwarzać w czasie rzeczywistym obraz z kamery.

```
1 Model: "sequential"
2 -----
3 Layer (type)          Output Shape         Param #
4 =====
5 conv2d (Conv2D)        (None, 200, 200, 32)    896
6
7 conv2d_1 (Conv2D)      (None, 200, 200, 32)    9248
8
9 max_pooling2d (MaxPooling2D) (None, 66, 66, 32)    0
10 )
11
12 conv2d_2 (Conv2D)      (None, 66, 66, 64)    18496
13
14 conv2d_3 (Conv2D)      (None, 66, 66, 64)    36928
15
16 max_pooling2d_1 (MaxPooling2D) (None, 22, 22, 64)    0
17
18
```

```

19 conv2d_4 (Conv2D)           (None, 22, 22, 128)    73856
20
21 conv2d_5 (Conv2D)           (None, 22, 22, 128)    147584
22
23 max_pooling2d_2 (MaxPooling2D) (None, 7, 7, 128)    0
24
25 conv2d_6 (Conv2D)           (None, 7, 7, 256)     295168
26
27 conv2d_7 (Conv2D)           (None, 7, 7, 256)     590080
28
29 flatten (Flatten)          (None, 12544)        0
30
31 dense (Dense)              (None, 1500)         18817500
32
33 dropout (Dropout)          (None, 1500)         0
34
35 dense_1 (Dense)            (None, 26)           39026
36
37 =====
38 Total params: 20,028,782
39 Trainable params: 20,028,782
40 Non-trainable params: 0
41
42 -----
43

```

Listing 3.6: Podsumowanie modelu

3.2.3 Trening modelu

Trening modelu przebiega w oparciu o zatrzymywanie procesu uczenia na podstawie wyników poszczególnych epok. W wypadku, gdy model zaczyna się przetrenowywać uczenie modelu jest zatrzymywane. Innym równie ważnym czynnikiem jest mechanizm *ModelCheckpoint*. Pozwala on na zapisywanie obiektu modelu po każdej epoce treningu w celu późniejszego wczytania najlepszej wersji.

```

1 # %% Fit
2 model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
3     # tekst formatujacy nazwe pliku do ktorego zapisywany jest model
4     filepath=models_path.joinpath('{epoch:02d}-{val_loss:.2f}.hdf5'),
5     # zapisz caly model
6     save_weights_only=False,
7     monitor='val_accuracy',
8     mode='max',
9     # zapisz kazdy model
10    save_best_only=False)

```

```
11
12     history = model.fit(train_gen,
13         epochs=999,
14         validation_data=valid_gen,
15         # dodanie wczesniej utworzonego mechanizmu ModelCheckpoint do
16         # zapisywania modelu i EarlyStopping do zatrzymywania treningu po
17         # spadku precyzyji
18         callbacks=[model_checkpoint_callback, tf.keras.callbacks.
19                     EarlyStopping(patience=2)],
20         # liczba procesow uzywanych do wczytywania obrazow
21         workers=20)
```

Listing 3.7: Trening modelu i mechanizmy *Callback*

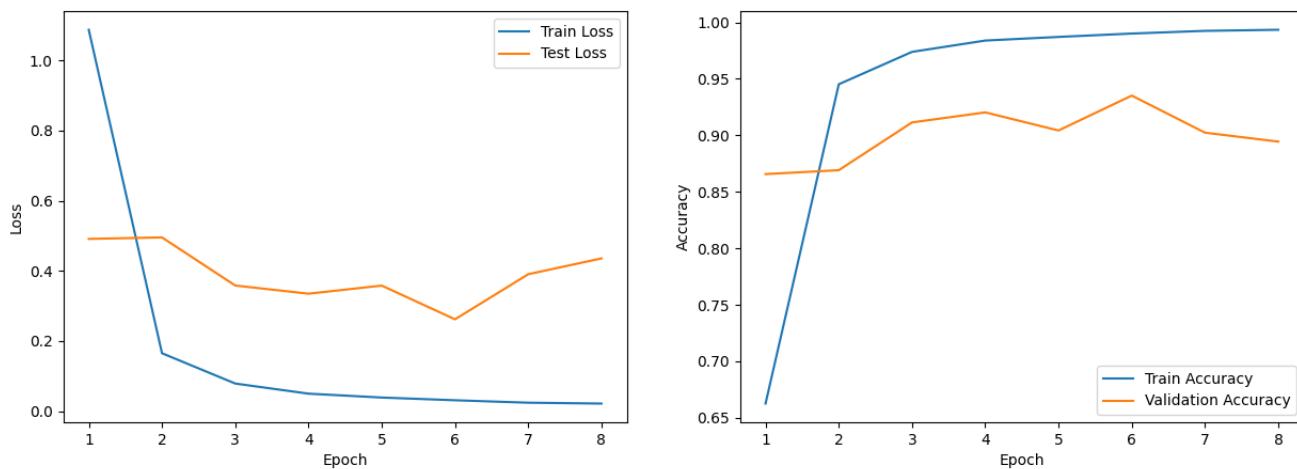
Model bardzo szybko osiąga dosyć wysoki wynik powyżej 90%. Po czwartej epoce model zaczyna się przetrenowywać i dokładność na zbiorze walidacyjnym spada. Z danej sesji treningowej najlepszym modelem będzie model zapisany po 4 epoce.

```
1  729/729 [=====] - 108s 138ms/step - loss:
2    1.0876 - accuracy: 0.6626 - val_loss: 0.4911 - val_accuracy: 0.8657
3  Epoch 2/999
4  729/729 [=====] - 133s 179ms/step - loss:
5    0.1650 - accuracy: 0.9452 - val_loss: 0.4953 - val_accuracy: 0.8691
6  Epoch 3/999
7  729/729 [=====] - 126s 171ms/step - loss:
8    0.0786 - accuracy: 0.9738 - val_loss: 0.3581 - val_accuracy: 0.9114
9  Epoch 4/999
10 729/729 [=====] - 124s 168ms/step - loss:
11   0.0497 - accuracy: 0.9839 - val_loss: 0.3348 - val_accuracy: 0.9203
12 Epoch 5/999
13 729/729 [=====] - 108s 145ms/step - loss:
14   0.0386 - accuracy: 0.9871 - val_loss: 0.3579 - val_accuracy: 0.9043
15 ...
16 ...
```

Listing 3.8: Przykładowy trening modelu

Na wykresie 3.3 został zwizualizowany proces treningu i towarzyszące mu parametry dokładności i straty dla danych treningowych i walidacyjnych.

Średni czas potrzebny na trening modelu wynosił od 6 do 10 minut. Model był trenowany na komputerze posiadającym procesor Intel Core i9 10850K na fabrycznym tak-towaniu rdzeni oraz wykorzystywał układ GPU - Nvidia RTX 3080. W celu trenowania modelu z wykorzystaniem karty graficznej zainstalowane zostało oprogramowanie *Nvidia Cuda*, które jest wykorzystywane w celu przeprowadzania obliczeń na procesorach graficznych Nvidia [22].



Rysunek 3.3: Wizualizacja treningu

3.3 Aplikacja do rozpoznawania języka migowego w czasie rzeczywistym

W ramach niniejszej pracy licencjackiej powstała aplikacja rozpoznająca znaki języka ASL w czasie rzeczywistym, stanowiąca niejako studium wykonalności. Aplikacja z dobrze wytrenowanym modelem jest w stanie w czasie rzeczywistym z dużą skutecznością rozpoznawać znaki pokazywane przez użytkownika.

W celu implementacji skorzystano z bibliotek *OpenCV*, *Tensorflow*, *numpy* i *time*. Przed wystąpieniem głównej pętli programowej inicjalizowane są zmienne odpowiadające za między innymi interfejs kamery i wczytany model. Metoda *decode_class* odpowiada za translację wektora zwracanego przez model na konkretną literę alfabetu (listing 3.9).

```

1 import cv2
2 import numpy as np
3 import tensorflow as tf
4 import time
5
6 vid = cv2.VideoCapture(0)
7 vid.set(cv2.CAP_PROP_FRAME_WIDTH, 200)
8 vid.set(cv2.CAP_PROP_FRAME_HEIGHT, 200)
9
10 model = tf.keras.models.load_model('models/04-0.34.hdf5')
11
12 def decode_class(one_hot):
13     cls = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
14           'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
15     return cls[np.argmax(one_hot)]
```

```
15  
16     last_char = 'None'  
17     text = ''  
18     char_time_start = time.time()
```

Listing 3.9: Wykorzystywane biblioteki i inicjalizacja zmiennych

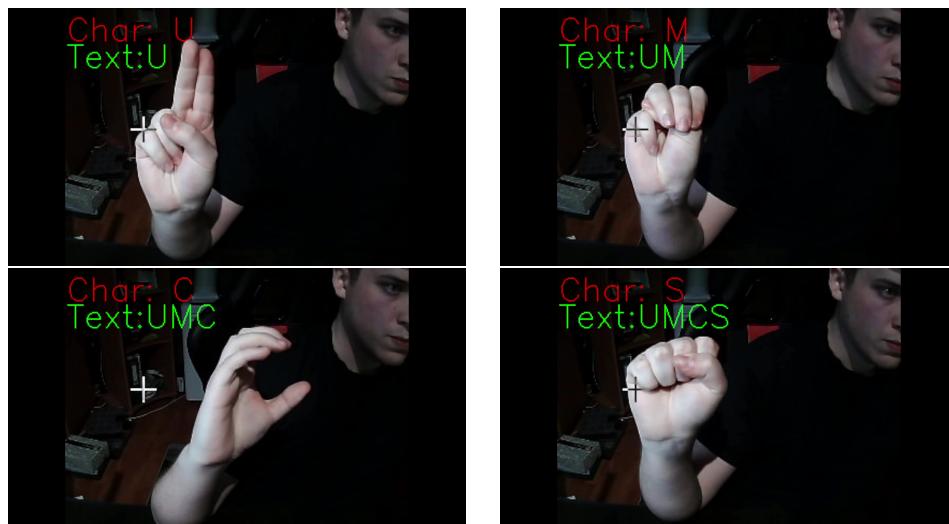
Główna pętla programowa zajmuje się pozyskiwaniem klatek z kamery za pomocą metody *vid.read()*. Tak pozyskana klatka jest zmniejszana do rozmiaru 200x200 pikseli i normalizowana w celu klasyfikacji przez model. Następnie wynik klasyfikacji jest dekodowany wcześniej wspomnianą metodą *decode_class* dla otrzymania konkretnej litery. Litera jest akceptowana i wyświetlana na ekranie jeśli występuje niezmiennie przez określoną ilość czasu.

```
1   while True:  
2       ret, frame = vid.read()  
3       batch = cv2.resize(frame, (200, 200))  
4       batch = np.reshape(batch, (1, 200, 200, 3))  
5       batch = batch * (1 / 255.)  
6       pred = model.predict(batch, verbose=0)  
7       char = decode_class(pred)  
8       if char != last_char:  
9           last_char = char  
10          char_time_start = time.time()  
11          if (time.time() - char_time_start) > 2.0:  
12              text += char  
13              char_time_start = time.time()  
14          frame = cv2.putText(frame,  
15              f'Char: {char}',  
16              org=(1, 30),  
17              fontFace=cv2.FONT_HERSHEY_SIMPLEX,  
18              fontScale=1,  
19              thickness=1,  
20              color=(0, 0, 255))  
21          frame = cv2.putText(frame,  
22              f'Text:{text if len(text) < 10 else text[-10:]}',  
23              org=(1, 55),  
24              fontFace=cv2.FONT_HERSHEY_SIMPLEX,  
25              fontScale=1,  
26              thickness=1,  
27              color=(0, 255, 0))  
28          cv2.imshow('frame', frame)  
29          key = cv2.waitKey(1) & 0xFF  
30          if key == ord('a'):  
31              text += char  
32              char_time_start = time.time()  
33          if key == ord('s'):
```

```
34     text = text[:-1]
35     if key == ord('q'):
36         break
37     vid.release()
38     cv2.destroyAllWindows()
```

Listing 3.10: Pętla programowa

Praktyczne działanie programu przedstawia rys. 3.4. W danym przypadku udało się pomyślnie wypisać na ekranie skrót nazwy Uniwersytetu Marii Curie-Słodowskiej.



Rysunek 3.4: Praktyczne działanie programu

3.4 Analiza błędów

Aplikacja pomimo odpowiednio przygotowanych zdjęć i dobrego oświetlenia ma problemy z rozróżnianiem znaków języka migowego odpowiadającym literom "m", "n", "s", "t" i czasami "a". Wynika to z podobieństwa układów dloni i palców reprezentujących te litery. We wszystkich przypadkach jest to zaciśnięta pięść, gdzie czynnikiem determinującym konkretną literę jest pozycja kciuka - na zewnątrz pięści, wewnętrz lub pomiędzy którymi palcami się on znajduje, przykładowo tak jak w znakach odpowiadającym literom "m", "n" i "t".

Rozwiążanie tego problemu może mieć wiele składowych. Przede wszystkim zwiększenie ilości danych w zbiorze zdjęć mogłoby mieć znaczący wpływ na izolację wykrywania znaków języka migowego od czynników otoczenia występujących w polu widzenia kamery. Zbiór danych powinien zostać rozszerzony o nowe otoczenia i aktorów. Również duże znaczenie mogłoby wnieść zmiana rozdzielczości obrazów jakie sieć przyjmuje z 200x200 pikseli na wartości wyższe (400x400, 600x600 i tym podobne). Zmiana rozdzielczości obrazu powiększyłaby liczbę szczegółów widocznych na obrazie co ułatwiłoby klasyfikację

modelu.

Rozdział 4

Podsumowanie

Sieci konwolucyjne to bez wątpienia bardzo dobre narzędzie do analizy i klasyfikacji obrazów. Sposób w jaki sieci konwolucyjne są budowane znacznie zmniejsza zapotrzebowanie na moc obliczeniową w porównaniu do zwykłych sieci neuronowych. Pozwala to stosować modele w aplikacjach wymagających czasu reakcji niemalże w czasie rzeczywistym. Rozpoznawanie języka migowego jest bardzo wymagającym zadaniem, nie jest jednak zadaniem niemożliwym. Przy dzisiejszym postępie technologicznym w dziedzinie sztucznych sieci neuronowych możliwe jest niewielkim nakładem pracy stworzenie modelu oraz aplikacji, które w zadowalający sposób poradzą sobie z problemem klasyfikacji znaków języka migowego.

Spis tabel

1.1 Szacowane rankingi użytkowników ASL według różnych źródeł	8
---	---

Spis rysunków

1.1	American Manual Alphabet [6]	9
2.1	Model CNN [9]	11
2.2	Przykładowe działanie filtrów 3x3 na obrazach [11]	12
2.3	Warstwa konwolucyjna - zasada działania [12]	13
2.4	Różnica w odstępach pomiędzy polami receptivejnymi w zależności od parametru <i>stride</i> [7]	13
2.5	Dane bez zero-padding [7]	14
2.6	Dane z zero-padding [7]	14
2.7	Przykładowe działanie max pooling [7]	14
2.8	Przykładowe działanie warstwy flatten [13]	15
2.9	Przykład warstw <i>fully-connected</i> [14]	15
2.10	Przykłady cyfr ze zbioru <i>MNIST</i> [16]	17
2.11	Wizualizacja modelu	18
3.1	Słabej jakości zdjęcie znaku ASL ze zbioru danych opublikowanych na <i>Kaggle.com</i> [18]	21
3.2	Zestawienie przykładowych obrazów z autorskiego zbioru liter języka ASL	21
3.3	Wizualizacja treningu	28
3.4	Praktyczne działanie programu	30

Listings

2.1	Pobieranie zbioru <i>MNIST</i> [17]	17
2.2	Zmiana rozmiarów tablic i normalizacja [17]	17
2.3	Budowa modelu [17]	18
2.4	Kompilacja i trening modelu [17]	18
2.5	Ewaluacja modelu [17]	19
3.1	Importowane biblioteki	22
3.2	Definicja ścieżek do danych i modeli	22
3.3	Definicja generatorów	23
3.4	Wczytanie i wyświetlenie pierwszych 10 zdjęć za pomocą <code>ImageDataGenerator</code>	24
3.5	Definicja modelu	24
3.6	Podsumowanie modelu	25
3.7	Trening modelu i mechanizmy <i>Callback</i>	26
3.8	Przykładowy trening modelu	27
3.9	Wykorzystywane biblioteki i inicjalizacja zmiennych	28
3.10	Pętla programowa	29

Bibliografia

- [1] K. Nakamura, “About american sign language,” *Deaf Research Library*, 1995.
- [2] B. Bahan, *Non-Manual Realization of Agreement in American Sign Language*. PhD thesis, Boston University, 1996.
- [3] N. E. Groce, *Everyone Here Spoke Sign Language: Hereditary Deafness on Martha’s Vineyard*. Harvard University Press Revised ed., 1988.
- [4] R. E. Mitchell, T. A. Young, B. Bachleda, and M. A. Karchmer, “How many people use asl in the united states?: Why estimates need updating,” *Sign Language Studies*, 2006.
- [5] E. Costello, *American Sign Language Dictionary*. Random House, 2012.
- [6] “American sign language.” https://en.wikipedia.org/wiki/American_Sign_Language. Dostęp: 2022-04-02.
- [7] S. Albawi, T. A. Mohammed, and S. Al-Zawi, “Understanding of a convolutional neural network,” in *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6, 2017.
- [8] Y. Lee, “Image classification with artificial intelligence: Cats vs dogs,” in *2021 2nd International Conference on Computing and Data Science (CDS)*, pp. 437–441, 2021.
- [9] “Different types of cnn architectures explained: Examples.” <https://vitalflux.com/different-types-of-cnn-architectures-explained-examples/>. Dostęp: 2022-06-20.
- [10] M. Jain, G. Kaur, M. P. Quamar, and H. Gupta, “Handwritten digit recognition using cnn,” in *2021 International Conference on Innovative Practices in Technology and Management (ICIPTM)*, pp. 211–215, 2021.
- [11] “Kernel (image processing).” [https://en.wikipedia.org/wiki/Kernel_\(image_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing)). Dostęp: 2022-04-25.

- [12] "How to customize convolution kernel weight parameters by pytorch." <https://developpaper.com/how-to-customize-convolution-kernel-weight-parameters-by-pytorch/>. Dostęp: 2022-04-25.
- [13] "Convolutional neural networks (cnn): Step 3 - flattening." <https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-3-flattening>. Dostęp: 2022-05-04.
- [14] "Example of fully-connected neural network.." https://www.researchgate.net/figure/Example-of-fully-connected-neural-network_fig2_331525817. Dostęp: 2022-05-04.
- [15] "Real-world applications of convolutional neural networks." <https://vitalflux.com/real-world-applications-of-convolutional-neural-networks/>. Dostęp: 2022-05-04.
- [16] "Mnist database." https://en.wikipedia.org/wiki/MNIST_database. Dostęp: 2022-05-05.
- [17] "Image classification in 10 minutes with mnist dataset." <https://towardsdatascience.com/image-classification-in-10-minutes-with-mnist-dataset-54c35b77a38d>. Dostęp: 2022-05-04.
- [18] "Asl alphabet." <https://www.kaggle.com/datasets/grassknotted/asl-alphabet>. Dostęp: 2022-03-15.
- [19] "Asl alphabet test." <https://www.kaggle.com/datasets/danrasband/asl-alphabet-test>. Dostęp: 2022-03-15.
- [20] "Imagedatagenerator." https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/imagedatagenerator. Dostęp: 2022-06-20.
- [21] "Opencv." <https://opencv.org/>. Dostęp: 2022-06-20.
- [22] "Nvidia cuda." <https://developer.nvidia.com/cuda-toolkit>. Dostęp: 2022-06-20.