

Uniwersytet Jagielloński w Krakowie
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Wojciech Ozimek

Nr albumu: 1124802

Zastosowanie sieci neuronowych do analizy rzutów kością do gry

Praca licencjacka
na kierunku Informatyka

Praca wykonana pod kierunkiem
prof. dr hab. Piotra Białasa
FAIS UJ

Kraków 2018

Spis treści

1	Cel pracy	3
2	Wstęp	4
2.1	Sztuczny neuron	4
2.2	Historia	4
3	Technologie	5
3.1	Język programowania i środowisko	5
3.2	Biblioteki	5
3.3	Technologie wspomagające	6
4	Słownik pojęć	8
4.1	Zbiór danych	8
4.2	Budowa sieci neuronowej	8
4.3	Propagacja wsteczna	10
4.4	Konwolucyjna sieć neuronowa	10
4.5	Warstwy sieci neuronowej	11
4.6	Funkcje aktywacji	12
4.7	Optymalizator	14
4.8	Przetrenowanie	15
4.9	Procesy	15
4.10	Inne	16
5	Tworzenie zbiorów danych	17
5.1	Uzyskiwanie zbiorów ze zdjęć	17
5.2	Zbiór obrazów kwadratowych	17
5.3	Zbiór obrazów prostokątnych	19
6	Sieć rozpoznająca ilość oczek	22
6.1	Pierwsze eksperymenty	22
6.2	Analiza różnych optymalizatorów	27
6.3	Wykorzystanie sieci AlexNet	29
6.4	Porównanie dla obrazów kolorowych i czarno-białych	31
6.5	Próby wykorzystania prostokątnych obrazów	32
6.6	Najskuteczniejszy model z prostokątnymi obrazami	36

6.7	Doskonalenie modelu z prostokątnymi obrazami	38
7	Podsumowanie	39
	Bibliografia	41
8	Dodatek A - zbiory danych	43
9	Dodatek B - modele sieci neuronowych	44
9.1	Obrazy kwadratowe 64x64	44
9.2	Obrazy prostokątne	45
9.3	Wytrenowane modele	46
9.4	Programy weryfikujące modele	46

1. Cel pracy

Celem pracy jest zastosowanie sieci neuronowych do rozpoznawania obrazów. Zadanie polega na rozpoznaniu ilości oczek wyrzuconych na kostce do gry. Sieć powinna rozpoznawać kostki o dowolnym zestawieniu kolorystycznym ścian i oczek oraz działać na rzeczywistym obrazie przesyłanym z kamery.

Dodatkowym aspektem poruszonym w pracy jest porównanie modeli w zależności od architektury sieci oraz zbiorów danych.

2. Wstęp

Sieci neuronowe składają się ze sztucznych neuronów, które są uproszczonymi modelami ich biologicznych odpowiedników. Pierwsze prace nad tym zagadnieniem zaczęły się już w latach 50. XX wieku.

2.1 Sztuczny neuron

Najprostszym modelem sztucznego neuronu jest tzw. neuron McCullocha-Pittsa, nazywany również neuronem binarnym. Taki neuron posiada wiele wejść, z których każde ma przypisaną wagę w postaci liczby rzeczywistej. Suma wartości wejściowych mnożona jest przez odpowiadające im wagi i podawana jako argument do funkcji aktywacji, wyjaśnionej w późniejszych rozdziałach. Wartość tej funkcji jest wyjściem neuronu i staje się wejściem innych neuronów, tak jak ma to miejsce w przypadku biologicznego neuronu [1].

Neurony najczęściej łączone są w struktury nazywane sztucznymi sieciami neuronowymi, ponieważ wykorzystanie pojedynczego neuronu nie pozwala na rozwiązywanie skomplikowanych zadań. Przykładowo próba realizacji prostych funkcji logicznych AND, OR, NOT i XOR okazuje się możliwa jedynie dla trzech pierwszych podanych funkcji [2]. Problem wiąże się z brakiem możliwości uzyskania poprawnych rezultatów dla zbiorów które nie są liniowo separowalne, czego przykładem jest właśnie funkcja XOR. Tworzenie rozbudowanych struktur z pojedynczych neuronów znacząco zwiększa możliwości adaptacyjne dla poszczególnych problemów, niejednokrotnie zaskakując osiąganymi wynikami.

2.2 Historia

Rozpoczęcie prac nad sztucznym neuronem można datować na rok 1943 kiedy Warren McCulloch i Walter Pitts przedstawili jego model [3]. Pierwszymi zastosowaniami sieci neuronowych były operacji bitowe oraz przewidywanie kolejnych wystąpień bitów w ciągu. Mimo licznych prób zastosowania ich do realnych problemów nie zyskały one popularności. W początkowym okresie rozwoju sieci neuronowych przyjmowano także wiele błędnych założeń, które wraz z ograniczonymi możliwościami obliczeniowymi komputerów skutecznie zniechęcały naukowców do prac nad tym zagadnieniem.

3. Technologie

Rozwój informatyki i wzrost zainteresowania sieciami neuronowymi spowodowały pojawienie się wielu, ułatwiających pracę, narzędzi. Wykorzystane w tej pracy technologie pozwoliły na skupienie się wokół faktycznego problemu i pominięcie wielu technicznych aspektów.

3.1 Język programowania i środowisko

Python

Kod stworzony na potrzeby tej pracy został napisany w języku programowania Python, który obecnie jest bardzo popularnym narzędziem wykorzystywanym w pracy naukowej. Powodem tego wyboru jest bardzo czytelna i zwięzła składnia. Python jest także domyślnym językiem używany w bibliotekach wykorzystywanych do tworzenia modeli sieci neuronowych.

Jupyter Notebook

Do łatwiejszego tworzenia i analizowania modeli sieci neuronowych wykorzystano aplikację Jupyter Notebook, pozwalającą uruchamiać w przeglądarce pliki, składające się z wielu bloków. W blokach może znajdować się wykonywalny kod programu m.in. w języku Python, a w pozostałych można umieszczać teksty, wykresy bądź tabele, co zdecydowanie ułatwia pracę.

3.2 Biblioteki

OpenCV

Praca na mniejszych niż otrzymywanych bezpośrednio z kamery obrazów bardzo przyspiesza naukę sieci neuronowych. Do obróbki obrazów wykorzystano bibliotekę OpenCV. Najczęściej wykorzystywana jest w językach C++ oraz Python, pozwalając m.in. na skalowanie, kadrowanie i obracanie obrazów.

TensorFlow

TensorFlow jest biblioteką ułatwiającą tworzenie sieci neuronowych. Jej ogromną zaletą jest implementacja w języku C++ umożliwiającą wykorzystywanie procesorów oraz kart graficznych. Z uwagi na charakter wykonywanych obliczeń praca przy użyciu kart graficznych jest kilkunastokrotnie szybsza niż na procesorze, co znacząco przyspiesza proces uczenia. Biblioteka najlepiej wspiera języki programowania C++ oraz Python.

Keras

Używaną w tej pracy biblioteką do tworzenia modeli sieci neuronowych jest Keras, wykorzystujący bardziej niskopoziomowe biblioteki jak TensorFlow. Zaletami Kerasa są zarówno przystępny interfejs pozwalający w krótkim czasie stworzyć model sieci oraz możliwość tworzenia zaawansowanych modeli przy średnim pogorszeniu czasu uczenia się sieci o 3-4% w stosunku do bibliotek, na których bazuje.

W sytuacji kiedy interfejs oraz dokumentacja do TensorFlow mogą być dla początkującej osoby niezrozumiałe, jest to świetna alternatywa do wdrożenia się w to zagadnienie.

Numpy

Wykonywanie operacji na dużych zbiorach danych dostarczanych do sieci zostało przyspieszone dzięki wykorzystaniu modułu Numpy. Moduł umożliwia wykonywanie zaawansowanych operacji na macierzach oraz wektorach, wspierający liczne funkcje matematyczne. Ponadto Numpy wprowadza własne typy danych oraz funkcje niedostępne w standardowej instalacji Pythona.

Matplotlib

Wykresy potrzebne do analizy sieci neuronowych tworzone były przy użyciu narzędzia Matplotlib. Posiada bardzo duże możliwości pozostając jednocześnie prostym w użyciu co sprawia, że jest bardzo popularny. Dodatkowo zawiera moduł pyplot, który w założeniu ma maksymalnie przypominać interfejs w programie MATLAB.

LaTeX

Oprogramowanie do składu tekstu, nie będące edytorem tekstowy typu WYSIWYG. LaTeX bazuje na TeX, który jest systemem składu drukarskiego. Ogromną zaletą jest możliwość tworzenia w tekście zaawansowanych wzorów matematycznych. Tekst niniejszej pracy napisany został przy pomocy tego narzędzia.

3.3 Technologie wspomagające

Kilkukrotne przyspieszenie tworzenia modeli do tej pracy możliwe było dzięki wykorzystaniu możliwości oferowanych przez szybkie karty graficzne.

NVIDIA CUDA

Technologia obecna w kartach firmy NVIDIA to równoległa architektura obliczeniowa, pozwalająca na wielokrotne przyspieszenie obliczeń podczas uczenia się sieci neuronowych. Dzięki bibliotekom TensorFlow i Keras, które wspierają obliczenia na kartach graficznych, czas uczenia sieci drastycznie maleje. Podczas tej pracy wykorzystana została karta NVIDIA TESLA K80 12GB GPU dostępna na Amazon AWS oraz Google Compute Engine.

Amazon AWS EC2

Szybkie karty graficzne wykorzystane w tej pracy były dostępne na platformie z wirtualnymi maszynami, które można dostosować do potrzeb klienta. Wykorzystano maszyny zoptymalizowane do obliczeń na kartach graficznych i uczenia maszynowego. Dzięki wykorzystaniu Amazon AWS, czas uczenia sieci zmniejszył się średnio 6-10 krotnie w stosunku do pracy na komputerze wykorzystującym procesor.

4. Słownik pojęć

Z sieciami neuronowymi wiąże się wiele pojęć, na które napotyka się podczas pracy z nimi. Poniżej znajduje się opis kluczowych elementów architektury sieci i zagadnień niezbędnych do realizacji tej pracy.

4.1 Zbiór danych

Do realizacji pracy konieczne było stworzenie zbioru danych składającego się z obrazów z kośćmi do gry. Zdjęcia zostały zrobione kamerą o rozdzielczości 1600x1200 pikseli, a następnie zmniejszone w celu zaoszczędzenia pamięci i ilości danych, która ma kluczowe znaczenie w przypadku zastosowań sieci neuronowych. Oprócz zmniejszenia obrazu były również obracane, deformowane (*ang. warping*) oraz kadrowane. Tak powstałe, liczne zbiory były wykorzystywane w trybie RGB oraz w odcieniach skali szarości, co pozwoliło na przyspieszenie pracy przez zmniejszenie liczby kanałów do jednego.

W pracy używano zbiorów z obrazami o rozmiarach 64x64 oraz 106x79 pikseli. Każde zdjęcie w zbiorze miało przypisaną wartość liczbową informującą o faktycznej ilości oczek wyrzuconych na przedstawionej kostce. Wartość ta zwana jest odpowiedzią i wykorzystywana jest w procesie uczenia sieci jako docelowa informacja, którą sieć ma zwrócić po weryfikacji danego obrazu.

Zbiór treningowy

Część zbioru danych, która wykorzystywana jest w procesie uczenia sieci, określana jest jako zbiór treningowy lub zbiór uczący. Jego liczebność to zazwyczaj 60-80% całego zbioru danych. W tej pracy zawsze przed rozpoczęciem uczenia, dane znajdujące się w tym zbiorze poddawane są losowej permutacji.

Zbiór testowy

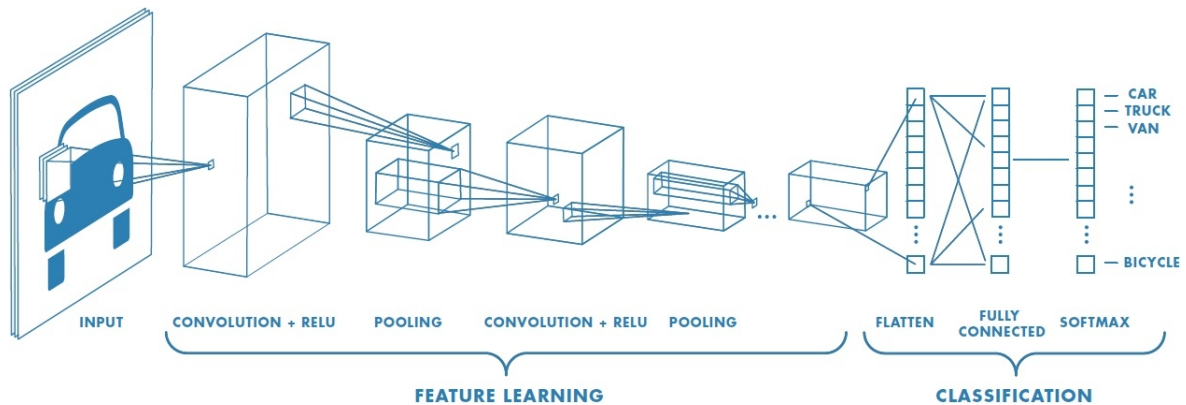
Do oceny zdolności sieci neuronowej do rozpoznawania danych służy zbiór testowy lub walidacyjny. Zbiór jest oddzielony od treningowego, by podczas weryfikacji sieci operowano na nieznanym jej wcześniej danych.

4.2 Budowa sieci neuronowej

Sieć neuronowa

Sieć neuronowa (*ang. ANN Artificial Neural Network*) to struktura matematyczna, składająca

się z neuronów połączonych w warstwy, mająca odzwierciedlać działanie biologicznych sieci neuronowych, a w szczególności mózgu [4]. Sieci mają szerokie zastosowanie w bardzo wielu dziedzinach. Wymagają jednak kosztownego obliczeniowo i czasowo procesu uczenia, podczas którego dostosowują się do danego problemu. Użycie wytrenowanej sieci, nie wymaga powtarzania uczenia, co pozwala na jej natychmiastowe wykorzystanie. Przykładowa ilustracja sieci neuronowej przedstawiona jest poniżej (zob. rys. 4.1)



<https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html>

Rysunek 4.1: Przykładowa sieć neuronowa.

Neuron

Neuron jest najmniejszym elementem sieci neuronowej, posiadającym wiele wejść i jedno wyjście [3, 5]. Sygnały z neuronów w poprzednich warstwach docierają na każde z wejść neuronu z przypisaną mu wagą. W neuronie obliczana jest suma ważona wejść, od której odejmowana jest z wartość progowa inaczej bias [6]. Przekroczenie wartości progowej uaktywnia neuron, a wartość sumy przekazywana jest do funkcji aktywacji. Wartość funkcji przekazywana jest na wyjście neuronu. Poniższy wzór 4.2 przedstawia sumę ważoną wejść neuronu.

$$f(x_i) = f\left(\sum_i w_i x_i - b\right)$$

gdzie: w - wagi wejść neuronu, x - wartość sygnału na wejściu, b - bias

Wzór 4.2: Suma ważona w neuronie wraz biasem b

Wagi neuronu

Jak napisano wcześniej, każde połączenie między pojedynczymi neuronami ma przypisaną wagę, która zmienia się podczas treningu, dostosowując sieć neuronową do danych treningowych. Efektem tego jest osiągnięcie coraz lepszych wyników w miarę trwania procesu uczenia.

Warstwa

Neurony w sieci zorganizowane są w warstwach. Komunikacja odbywa się tylko między ko-

lejnymi warstwami, neurony w danej warstwie nie są ze sobą połączone [1, 7]. Istnieje wiele rodzajów warstw, które opisane będą w dalszej części tego rozdziału.

Funkcja kosztu

Informacje o wartości błędu pomiędzy wartościami przewidzianymi przez sieć a wartościami docelowymi można uzyskiwać dzięki funkcji kosztu lub funkcji błędu. Jest ona niezbędna do prawidłowego przeprowadzenia procesu uczenia. Funkcja dostarcza informacje o różnicy między obecnym stanem sieci a optymalnym rozwiązaniem dla danych treningowych. Algorytm uczenia oblicza wartość funkcji kosztu w kolejnych krokach w celu jej zminimalizowania.

Najczęściej wykorzystywaną funkcją kosztu jest błąd średniokwadratowy 4.3. W tej pracy z uwagi na posiadanie 6 możliwych do uzyskania wyników na wyjściu sieci zastosowano wielowymiarową entropię krzyżową (*ang. Categorical cross-entropy*) 4.4, która zapobiega uwypukleniu nieprawidłowych wartości [8]. Jest również sugerowana przez bibliotekę Keras oraz wykorzystywana w wielu przykładowych modelach sieci neuronowych.

$$J(\eta) = \frac{1}{2} \sum_i^m \left(h_{\theta}(x^{(i)}) - y^{(i)} \right)^2$$

gdzie: $h_{\theta}(x)$ - wartość przewidziana przez sieć, y - wartość oczekiwana

Wzór 4.3: Błąd średniokwadratowy

$$L(p, q) = - \sum_i p_i \log(q_i)$$

gdzie: p - rozkład prawdopodobieństwa dla prawdziwych wartości, q - rozkład prawdopodobieństwa dla predykcji

Wzór 4.4: Wielowymiarowa entropia krzyżowa

4.3 Propagacja wsteczna

Propagacja wsteczna lub wsteczna propagacja błędów (*ang. Backpropagation*) jest algorytmem uczenia sieci neuronowych [9, 10]. Służy do wyliczenia gradientu funkcji kosztu, który informuje o szybkości spadku wartości tej funkcji z uwzględnieniem wag neuronów. Obliczenie gradientu w sieci propagowane jest od warstwy wyjściowej do wejściowej, czemu algorytm zawdzięcza swoją nazwę.

4.4 Konwolucyjna sieć neuronowa

Korzystanie z sieci neuronowych do operowania na zdjęciach sprawia problemy z dużą ilością parametrów odpowiadających wartościom każdego z pikseli. Zapobiega temu stosowanie konwolucyjnych sieci neuronowych (*ang. CNN - Convolutuinal Neural Network*) [11, 12]. Zamiast

przekazywać informacje o wszystkich pikselach na obrazie, sieć analizuje je przy użyciu filtrów konwolucyjnych. Te wartości przesyłane są dalej do następnej warstwy.

Obecnie ten typ sieci odnosi największe osiągnięcia w dziedzinie rozpoznawania obrazów, w wielu przypadkach dorównując lub nawet pokonując ludzkie wyniki.

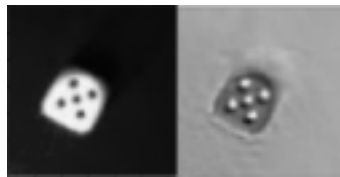
Konwolucja

Wymieniona wyżej konwolucja, inaczej spłot, polega na złożeniu dwóch funkcji. W przypadku obrazów jedna z tych funkcji to obraz, który ma rozmiary większe niż druga funkcja określana mianem filtra konwolucyjnego. Zastosowanie spłotu, w zależności od przypadku, pozwala na rozmycie, wyostanie lub wydobycie głębi z danego obrazu [13]. Poniższy wzór 4.5 przedstawia sposób obliczenia spłotu dwóch funkcji.

$$h[m, n] = (f * g)[m, n] = \sum_j^m \sum_k^n f[j, k] * g[m - j, n - k]$$

Wzór 4.5: Spłot funkcji. Funkcja f to dwuwymiarowa macierz z pikselami obrazu. Funkcja g to filtr. Funkcja h to nowo otrzymany obraz.

Poniżej przedstawiono przykładowy obraz na wyjściu filtra konwolucyjnego (zob. rys. 4.6):



Rysunek 4.6: Po lewej oryginalny obraz, po prawej na wyjściu filtra konwolucyjnego.

4.5 Warstwy sieci neuronowej

W sieciach neuronowych wyróżniamy kilka rodzajów warstw, które poza wejściową i wyjściową, można dobierać zależnie od sposobu rozwiązania danego problemu i typu posiadanych danych.

Wejściowa

W sytuacji, gdy danymi jest zbiór obrazów, warstwa wejściowa ma rozmiar identyczny z wymiarami obrazu.

Wyjściowa

Rozmiar warstwy wyjściowej odpowiada ilości klas, do jakiej dane były rozdzielane. W tej pracy, gdzie oczekiwanym wyjściem była liczba oczek możliwych do wyrzucenia na kostce, odpowiada on 6 klasom. Wyjściem takiej sieci jest więc wektor o wymiarach 6x1.

Konwolucyjna

Warstwa konwolucyjna służy do przetworzenia danych z poprzedniej warstwy przy użyciu filtrów konwolucyjnych [1]. Filtry mają określone wymiary i służą do znajdowania cech na obrazach lub ich fragmentach. Zastosowanie wielu warstw konwolucyjnych umożliwia filtrom analizowanie bardziej złożonych zależności na obrazach.

W pełni połączona

Najpopularniejszy typ warstwy to w pełni połączony (*ang. Fully Connected, Dense*). Każdy neuron łączy się ze wszystkimi neuronami następnej warstwy, co skutkuje dużą ilością potrzebnych do wykonania obliczeń. W sieciach konwolucyjnych umieszczane są zwykle po warstwach konwolucyjnych i służą do powiązania nieliniowych kombinacji.

Flatten

Warstwa spłaszczająca (*ang. Flatten*) stosowana jest w celu połączenia warstw konwolucyjnych z warstwami w pełni połączonymi. Realizowane jest to poprzez przekształcenie warstwy wejściowej do jednowymiarowego wektora, który następnie służy za wejście do kolejnych warstw.

Odrzucająca

Użycie warstwy odrzucającej (*ang. Dropout*) jest jednym z najlepszych sposobów na poradzenie sobie ze zjawiskiem przetrenowania, opisanym na końcu tego rozdziału [14]. Warstwa odrzucająca nie wykorzystuje wyjść pewnych neuronów, co skutkuje rozpoznawaniem wyraźniejszych cech. W warstwie tej określamy prawdopodobieństwo, z jakim neurony zostaną zachowane. Najczęściej stosuje się ją po warstwach w pełni połączonych, zarówno przy przechodzeniu w przód, jak i tył [15].

Pooling

Warstwa ta wykorzystywana jest do zmniejszenia rozmiaru pamięci oraz ilości obliczeń w sieci neuronowej. Operacja polega na wybraniu jednego piksela z danego obszaru i przekazaniu go do następnej warstwy. Najczęściej wykorzystywany jest tzw. MaxPooling, wybierający piksel o największej wartości. Pooling nie zachowuje informacji o położeniu piksela przekazanego na wyjście warstwy, co może objawiać się błędnymi interpretacjami przez sieci.

4.6 Funkcje aktywacji

Pomnożona suma wartości wejściowych i ich wag zostaje przekazana jako argument do funkcji aktywacji, a obliczona wartości staje się wyjściem neuronu. Wybór funkcji jest jednym z kluczowych parametrów danej sieci, ponieważ niewłaściwy wybór może prowadzić do problemów podczas uczenia sieci [16, 17].

Sigmoid

Sigmoid jest popularną funkcją aktywacji. Problemem z nią związanym jest ryzyko zaniknięcia gradientu, co może prowadzić do problemu tzw. umierającego neuronu. Występuje ono, gdy dla danej funkcji aktywacji, gradient staje się bardzo mały, co jest równoznaczne z zaprzestaniem procesu uczenia. W sigmoid gradient może zanikać obustronnie. Wzór funkcji sigmoidalnej pokazany jest poniżej 4.7.

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

Wzór 4.7: Funkcja sigmoidalna

ReLU

ReLU (*and. Rectified linear unit*) jest najpopularniejszą funkcją aktywacji wykorzystywaną w sieciach neuronowych [4, 18]. Zasadą tego jest szybki czas uczenia sieci bez znaczącego kosztu w postaci generalizacji dokładności. Problem z zanikającym gradientem jest mniejszy niż w przypadku funkcji sigmoidalnej, ponieważ występuje on tylko z jednej strony. Wzór 4.8 na obliczenie ReLU przedstawiony jest poniżej.

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

Wzór 4.8: ReLU - Rectified linear unit

LeakyReLU

LeakyReLU jest ulepszeniem funkcji ReLU [4] dzięki zastosowaniu niewielkiego gradientu w sytuacji, dla której ReLU jest nieaktywne. Zmiana ta pozwala na uniknięcie problemu znikającego gradientu. Jej wzór 4.9 przedstawiony jest poniżej.

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{if } x < 0 \end{cases}$$

Wzór 4.9: LeakyReLU

Softmax

Softmax jest funkcją obliczającą rozkład prawdopodobieństwa wystąpienia danego zdarzenia. Wykorzystywana jest w sytuacjach wymagających przyporządkowania elementu do jednej z wielu klas.

W tej pracy wykorzystywana jest w ostatniej warstwie do przypisania ilości rozpoznanych oczek do każdego ze zdjęć. Wzór funkcji 4.10 zaprezentowany jest poniżej.

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=0}^k (e^{x_j})}$$

Wzór 4.10: Funkcja softmax

4.7 Optymalizator

Algorytmy optymalizacyjne wykorzystywane są do minimalizacji funkcji błędu poprzez znajdowanie nowych wartości wag neuronów w sieci. Mają kluczowe znaczenie podczas procesu uczenia sieci w kwestii czasu oraz skuteczności [19].

Współczynnik uczenia

Do ustalenia z jakąś szybkością sieć neuronowa będzie dostosowywać się do danych, wykorzystujemy współczynnik lub wskaźnik uczenia (*ang. learning rate*). Wybranie zbyt małego współczynnika wydłuży proces uczenia, a zbyt duża jego wartość może spowodować problemy ze znalezieniem optymalnego rozwiązania. W niektórych algorytmach współczynnik uczenia zmniejsza się w czasie, by lepiej dostosować się do danych i zniwelować oba wspomniane problemy.

Metoda największego spadku

Metoda największego spadku (*ang. Gradient Descent*) jest podstawowym algorytmem służącym do zmiany wartości wag podczas procesu uczenia sieci. Wadą tej metody jest przeprowadzanie jednorazowej aktualizacji po wyliczeniu gradientu dla całego zestawu danych. Spowalnia to uczenie i może powodować problem z ilością zajmowanego miejsca w pamięci. Sporą wadą jest możliwość doprowadzenia do stagnacji w jednym z lokalnych minimów funkcji. Wzór 4.11 znajduje się poniżej.

$$\theta = \theta - \eta * \nabla J(\theta)$$

gdzie: η - współczynnik uczenia, J - funkcja błędu

Wzór 4.11: Metoda największego spadku

Stochastic gradient descent

Stochastic gradient descent (*skrót. SGD*) jest rozwinięciem metody największego spadku, bardzo często wykorzystywaną w praktyce [20]. Ulepszenie polega na obliczaniu gradientu dla jednego lub niewielkiej ilości przykładów treningowych. SGD pozwala na lepsze dopasowanie niż metoda gradientu prostego, ale wiąże się z koniecznością zastosowania mniejszego współczynnika uczenia. Sposób obliczania SGD przedstawiony jest poniżej 4.12.

$$\theta = \theta - \eta * \nabla J(\theta; x_i; y_i)$$

gdzie η - współczynnik uczenia, J - funkcja błędu, x , y - wejściowe i wyjściowe dane treningowe dla określonej iteracji

Wzór 4.12: Stochastic gradient descent

Adam

Adam to skrót od angielskiej nazwy *Adaptive Moment Estimation* i jest rozwinięciem metody Stochastic Gradient Descent [20, 21]. Metoda ta zapewnia dostosowanie współczynnika uczenia do każdego z parametrów z osobna, korzystając przy tym z pierwszego i drugiego momentu centralnego, czyli odpowiednio ze średniej oraz wariancji. Adam obecnie jest bardzo popularny z uwagi na szybkość działania i osiąganie bardzo dobrych wyników. Poniższe wzory przedstawiają sposób na jego obliczenie 4.13, 4.14. Obliczone momenty podstawiane są do wzoru

$$\begin{aligned}\hat{m}_t &= \frac{m_t}{1 - \beta_1^t}, \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t},\end{aligned}$$

Wzór 4.13: Wzór na obliczanie pierwszego (m) i drugiego (v) momentu centralnego

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

gdzie najczęściej $\beta_1^t = 0.9$ oraz $\beta_2^t = 0.99$ a $\epsilon = 10^{-8}$

Wzór 4.14: Optymalizator Adam

4.8 Przetrenowanie

Zjawisko przetrenowania (*ang. Overfitting*) polega na błędnym rozpoznawaniu nowych danych, kiedy podczas treningu sieć uczyła się poprawnie. Może być ono bardzo trudne do wykrycia, co powoduje, że jest jednym z największych problemów podczas pracy z sieciami neuronowymi. Istnieje wiele sposobów na ograniczenie przetrenowania, a jedną z najlepszych technik jest zastosowanie warstw odrzucających [14].

4.9 Procesy

Uczenie

Uczenie bądź trening sieci jest procesem, w którym wagi sieci dostosowują się do dostarczonych danych. Nauczona sieć powinna sprawnie realizować zamierzone zadanie, na przykład poprawnego rozpoznawania ilości oczek wyrzuconych na kostce. Trening jest bardzo kosztowny

obliczeniowo, ponieważ operacje dodawania i mnożenia wektorów oraz macierzy wykonywane są miliony razy.

Testowanie

Testowanie polega na porównaniu wyników dla zbioru testowego z oczekiwanymi wartościami. Istotne jest, aby zbiór służący do testowania nie był wcześniej użyty do treningu sieci. Nauczony model powinien być w stanie rozpoznawać nowe, nieużyte podczas procesu uczenia dane i poprawnie je klasyfikować.

Predykcja

Wartości zwrócone przez sieć po umieszczeniu w niej określonych danych nazywane są predykcją. Pozwala to na wykorzystanie nauczonego modelu w praktycznym zastosowaniu.

Epoka

Epoka nie zalicza się do procesów, ale jest bezpośrednio związana z procesem uczenia. Jednorazowe przejście wszystkich elementów z treningowego zbioru danych nazywane jest epoką. W praktycznym zastosowaniu ilość epok ustala się na co najmniej kilkanaście, chociaż w wielu pracach naukowych przedstawiane są wyniki po 100 epokach treningu.

4.10 Inne

Poniżej przedstawiono dwa najbardziej rozpowszechnione zbiory danych do rozpoznawania obrazów. Architektura pierwszej sieci bazowała na modelach służących do rozpoznawania tych zbiorów.

MNIST

Baza danych MNIST [22] to zbiór 60000 treningowych i 10000 testowych czarno-białych obrazów w rozmiarze 28x28x1, zawierających ręcznie napisane cyfry 0-9. Jest jednym z najpopularniejszych zbiorów służących do rozpoczęcia nauki sztucznych sieci neuronowych i uczenia maszynowego.

CIFAR-10

Zbiór CIFAR-10 [23] składa się z 50000 treningowych i 10000 testowych kolorowych obrazów w rozmiarze 32x32x2. Jest podzielony na 10 klas: samolot, samochód, ptak, kot, jeleń, pies, żaba, koń, statek, ciężarówka; gdzie każdej z nich przypada po 6000 obrazów. Jest to podobnie jak MNIST jeden z najbardziej popularnych zbiorów do uczenia maszynowego i sieci neuronowych.

5. Tworzenie zbiorów danych

Aby sieć neuronowa mogła rozpoznawać obrazy, konieczne jest stworzenie danych do jej nauki i testowania. Priorytetem przy ich tworzeniu było umożliwienie rozwoju różnych modeli sieci neuronowych bez konieczności każdorazowej zmiany w zbiorach lub dopasowywania ich do konkretnej sieci.

Wszystkie zbiory dostępne są pod linkami w części Dodatek A [8.]

5.1 Uzyskiwanie zbiorów ze zdjęć

Ilość zdjęć wykonanych ręcznie z wykorzystaniem kamery była zbyt mała by sieć bezpośrednio z nich korzystała. Oprócz tego rozmiar zdjęć 1600x1200 był na tyle duży, że bardzo wydłużyłby proces uczenia sieci.

Rozwiązaniem obu problemów było zwiększenie ilości zdjęć przy jednoczesnym zmniejszeniu ich rozmiarów. Do wszystkich obrazów wykorzystywanych w tej pracy, zastosowano następujące kroki:

- Skalowanie obrazów
- Przekształcanie (*ang. warping*) na kilka różnych sposobów, by uzyskać lekko zdeformowane kostki
- Rotacja o określony kąt by umożliwić rozpoznawanie kości niezależnie od jej obrotu
- Kadrowanie obrazu do pożądanej wielkości

Wszystkie zbiory wykorzystane w tej pracy były podzielone na części treningową i testową w stosunku 4:1, niezależnie od ich liczebności.

5.2 Zbiór obrazów kwadratowych

Pierwszy stworzony zbiór składał się z obrazów kwadratowych. Było to podyktowane faktem, że praktycznie wszystkie przykłady użycia sieci neuronowych do rozpoznawania obrazów korzystały z kwadratowych zdjęć.

Najważniejszym założeniem przy tworzeniu zbioru było uzyskanie zdjęć kości położonej na środku obrazu z kamery, na obszarze kwadratu o boku długości około 3 krotnie większej niż długość ściany samej kości. Pozwalało to uzyskać niewielkie rozmiary obrazów 64x64 piksele, gdzie kość stanowiła około 14%, a jedno oczko 0,2% powierzchni.

Kolejną kwestią było przystosowanie sieci do rozpoznawania kości o różnych kolorach ścian, oczek i samego tła. Zdjęcia podzielono na zestawy, z których każdy miał ustaloną barwę tła, kości oraz oczek. Każda ścianka sfotografowana była od 3 do 30 razy, dając liczebność każdego z zestawów między 18 a 180 zdjęć. Niektóre zdjęcia poddawane były działaniu ostrego, punkтового światła w celu wygenerowania trudniejszych do rozpoznania obrazów. Zestawienia kolorystyczne wypisane są w tabeli poniżej 5.1.

Kolor			Ilość zdjęć	
kości	oczek	tła	ściany	kostki
biały	czarny	czerwony	30	180
biały	czarny	granatowy	3	18
biały	czarny	czarny	8	48
beżowy	czarny	czerwony	3	18
czarny	biały	czarny	10	60
czarny	biały	czerwony	10	60
czerwony	biały	czerwony	5	30
czerwony	czarny	czerwony	3	18
granatowy	złoty	biały	4	24
granatowy	złoty	niebieski	6	36
zielony	biały	zielony	8	48
zielony	biały	biały	5	30
różowoczerwony	czarny	biały	5	30
<i>Łączna ilość zdjęć:</i>			<i>100</i>	<i>600</i>

Tablica 5.1: Zestawienie kolorystyczne obrazów 1

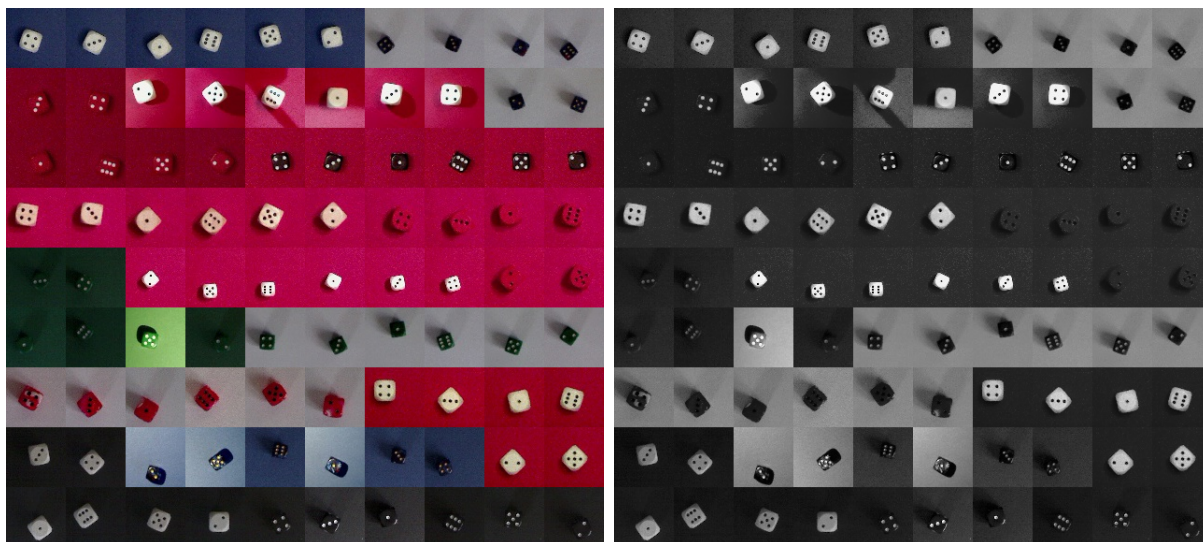
Po wykonaniu zdjęć każde z nich zostało poddane obróbce w celu zmniejszenia i uzyskania licznych, zróżnicowanych zbiorów. W przypadku zbioru kwadratowego wszystkie zdjęcia zostały przeskalowane do 18,66% początkowego rozmiaru (stosunek 5,36:1). Następnie poddano je 6 przekształceniom, które zaprezentowano poniżej (zob. rys. 5.1):



Rysunek 5.1: Przekształcenia obrazów. Oryginalny obraz znajduje się po lewej.

Kolejnym krokiem był obrót o kąt 15° lub 30° , zależnie czy dany zbiór wykorzystywany był w mniej lub bardziej rozbudowanych sieciach. Ostatnią operacją było kadrowanie zdjęć do rozmiaru 64x64 piksele.

Ostatecznie uzyskano 84 lub 168 zdjęć o rozmiarze 64x64 z każdego z początkowych obrazów o rozmiarze 1600x1200 pikseli. Pełne zbiory liczyły 50400 i 100800 obrazów, jednakowo w wersjach kolorowych RGB oraz w odcieniach skali szarości. Wybrane zdjęcia ze zbioru znajdują się (zob. rys. 5.2) poniżej:



Rysunek 5.2: Obrazy o rozdzielczościach 64x64

5.3 Zbiór obrazów prostokątnych

Po wytrenowaniu kilku sieci na zbiorach kwadratowych, zdecydowano się na wykorzystanie większych obrazów, które zachowują kształt prostokątny i stosunek szerokości do wysokości identyczny z obrazem otrzymywanym z kamery. Miało to ułatwić detekcję kości na obrazie, ponieważ obszar 64x64 pikseli był na tyle mały, że ciężko byłoby w niego trafić podczas rzutu kostką.

Bardzo istotną rzeczą w tym zbiorze był fakt, że pomimo większego rozmiaru zdjęć, stosunek rozmiaru kości do całego zdjęcia był zdecydowanie mniejszy. W zdjęciach kwadratowych kostka zajmowała 14%, natomiast w prostokątnych jedynie 2% powierzchni obrazu, a dodatkowo pojedyncze oczko to tylko 0,08% powierzchni. Dla zwiększenia trudności kości były rozmieszczane w bardziej zróżnicowany sposób, nie skupiały się jedynie wokół środka obrazu. Do wykonanych na potrzeby poprzedniego zbioru zdjęć dodano nowe i tak utworzono kilkanaście zestawień kolorystycznych, przedstawionych w poniższej tabeli 5.2

Kolor			Ilość zdjęć	
kości	oczek	tła	ściany	kostki
biały	czarny	czerwony	30	180
biały	czarny	granatowy	3	18
biały	czarny	czarny	8	48
beżowy	czarny	czerwony	3	18
czarny	biały	czarny	10	60
czarny	biały	czerwony	10	60
czerwony	biały	czerwony	5	30
czerwony	czarny	czerwony	3	18
granatowy	złoty	biały	4	24
granatowy	złoty	niebieski	6	36
zielony	biały	zielony	8	48
zielony	biały	biały	5	30
różowoczerwony	czarny	biały	5	30
biały	czarny	zielony	6	36
czerwony	biały	zielony	6	36
czerwony	czarny	różowy	7	42
czarny	biały	szary	6	36
czarny	biały	niebieski	6	36
beżowy	czarny	szary	6	36
beżowy	czarny	niebieski	6	36
granatowy	złoty	biały	8	48
zielony	biały	żółty	7	42
różowoczerwony	czarny	pomarańczowy	6	36
<i>Łączna ilość zdjęć:</i>			<i>164</i>	<i>984</i>

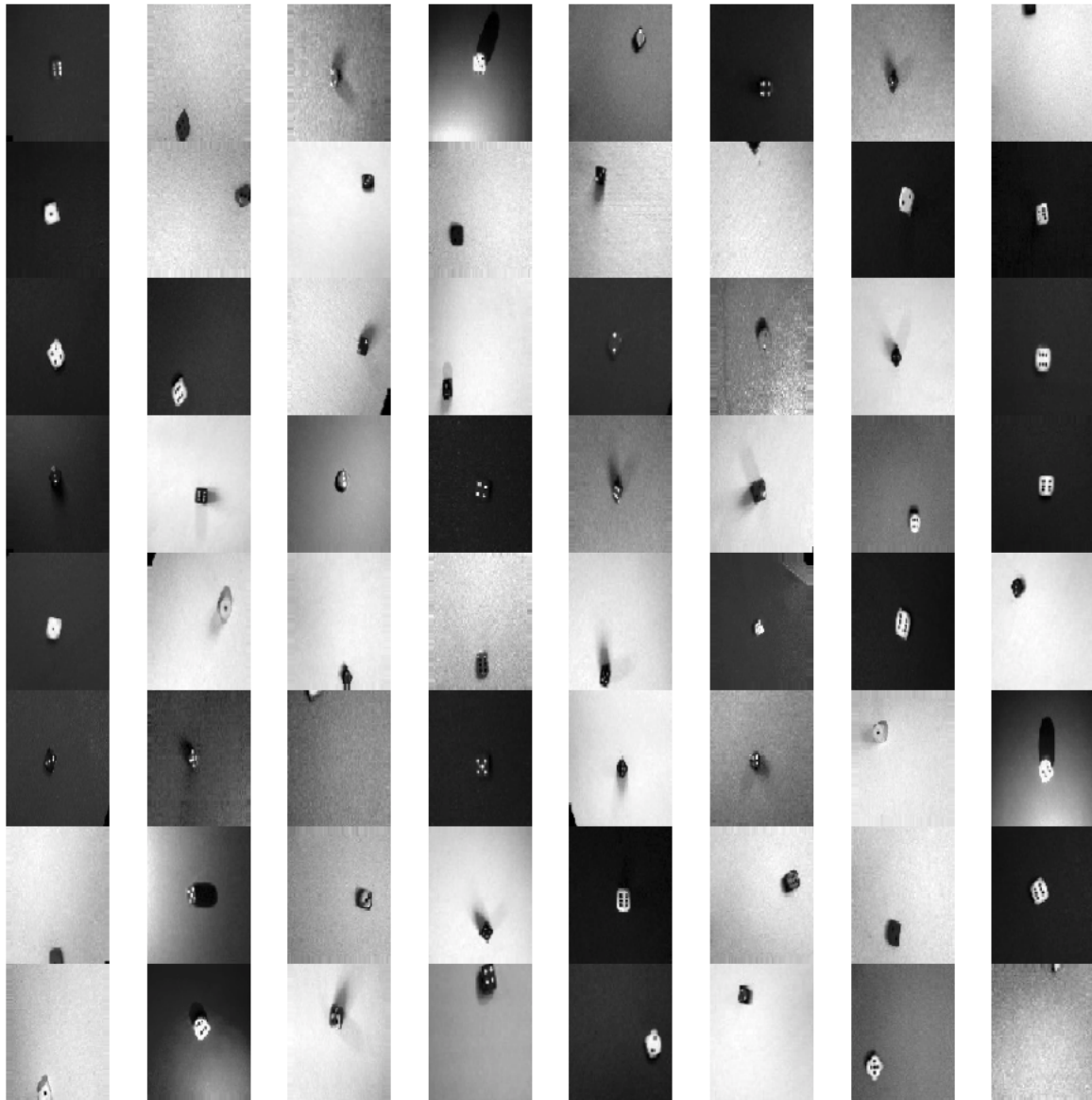
Tablica 5.2: Zestawienie kolorystyczne obrazów 2

Tak jak poprzednio, zdjęcia zostały poddane obróbce. Początkowo wszystkie zostały przeskalowane do 16,5% początkowego rozmiaru (stosunek 6:1). Następnie poddano je 4, analogicznym jak poprzednio przekształceniom, które pokazano poniżej (zob. rys. 5.3):



Rysunek 5.3: Przekształcenia obrazów. Oryginalny obraz znajduje się po lewej.

Dokonano również rotacji o kąt 30° i kadrowanie zdjęć do rozmiaru 106x79 pikseli. Uzyskano w ten sposób 60 zdjęć o rozmiarze 106x79 z każdego początkowego zdjęcia. Obrazy zastosowane w sieciach występowały jedynie w odcieniach skali szarości. Całkowita ilość obrazów w pełnym zbiorze danych wynosiła 59040, co przełożyło się na 47232 obrazów treningowych i 11808 testowych. Przykładowe zdjęcia ze zbioru znajdują się na (zob. rys. 5.4) poniżej:



Rysunek 5.4: Obrazy o rozdzielczościach 106x79

6. Sieć rozpoznająca ilość oczek

Po zrobieniu zbioru danych składających się z kwadratowych obrazów, przystąpiono do tworzenia sieci neuronowych rozpoznających ilość oczek wyrzuconych na kostce do gry.

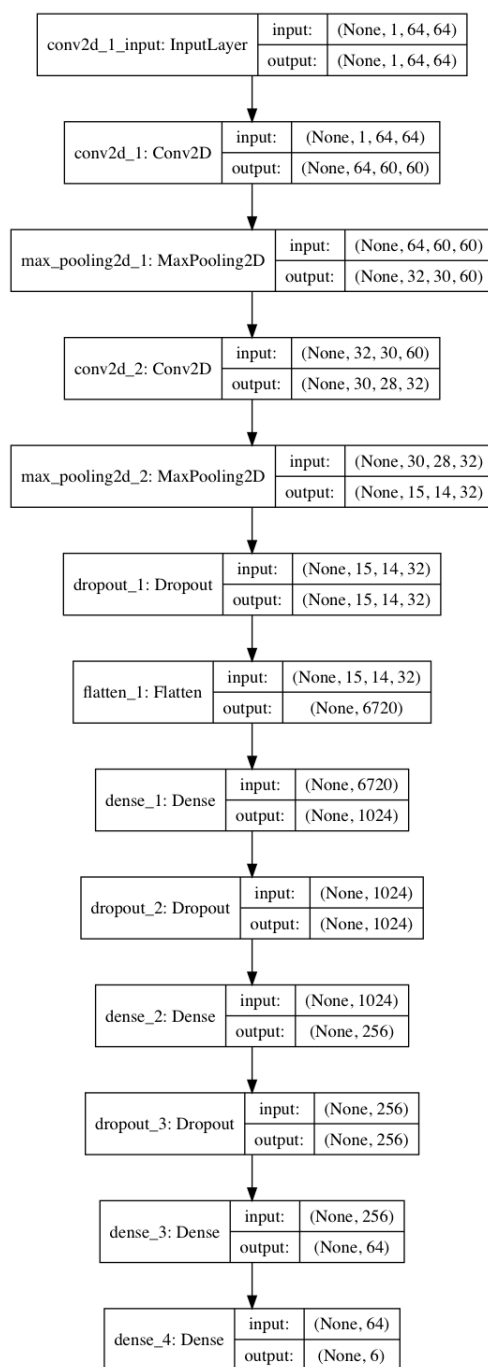
Odnosiniki do wszystkich modeli dostępne są pod linkami w części Dodatek B [9.]

6.1 Pierwsze eksperymenty

Pierwszy model

Pierwsza próba stworzenia sieci miała na celu wytrenowanie sieci na możliwie prostym, nigdy później niewykorzystywanym zbiorze obrazów. Wykorzystano jedynie obrazy z czerwonym tłem, białą kością i czarnych oczkami. Kąt rotacji zmniejszono do 5° , uzyskując 60480 zdjęć ze 120 oryginalnych.

Architektura tej sieci była dobierana bez większego wdrażania się w szczegóły i bazowała na modelach sieci udostępnionych na stronach Keras oraz TensorFlow, wykorzystywanych do analizy zbiorów MNIST oraz CIFAR10. Za optymalizator został wybrany Adam, a sam trening został przeprowadzony przez 25 epok. Model sieci prezentował się następująco (zob. rys. 6.1):



Rysunek 6.1: Pierwszy model sieci

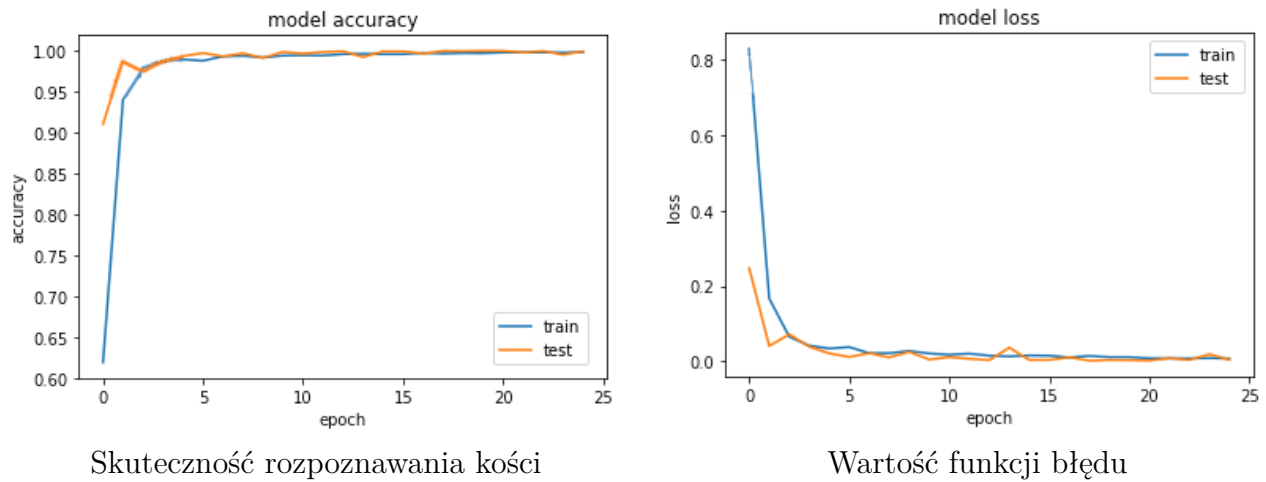
Ta oraz wszystkie kolejne sieci korzystają z ReLU jako funkcji aktywacji, co pozwoliło na zmniejszenie ilości różnic między modelami.

Na końcu każdej z sieci zastosowana jest funkcja softmax, która oblicza prawdopodobieństwo wyrzucenia danej ilości oczek na kostce.

Rezultaty osiągnięte przez ten model były zdumiewające. Po pierwszej epoce sieć uzyskała 61,98% skuteczności ostatecznie osiągając wynik 99,88% po 25 epokach.

Po analizie wyniku okazało się, że wykorzystanie wielu bardzo podobnych obrazów o identycznym układzie barw, spowodowało występowanie niemal identycznych z treningowymi, zdjęć

testowych. Przy bardziej zróżnicowanych zdjęciach, wynik okazałby się zdecydowanie gorszy. Występuje tu zjawisko przetrenowania, które pomimo bardzo dobrych wyników na zbiorze testowym, w praktyce osiągałoby niezadowalające rezultaty. Poniżej znajdują się wykresy (zob. rys. 6.2) dla sieci w kolejnych epokach:

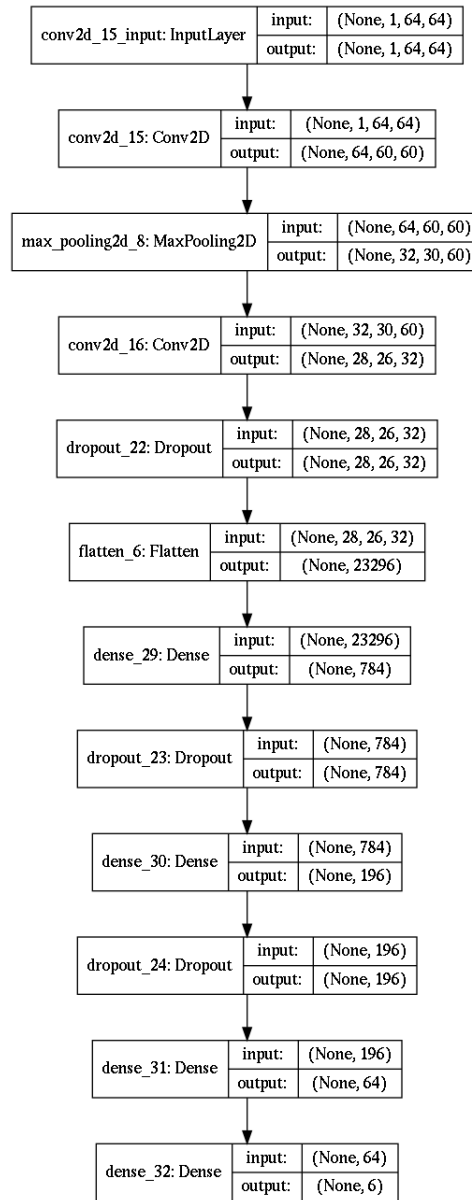


Rysunek 6.2: Wykresy dla pierwszego modelu

Model ze zróżnicowanymi zdjęciami

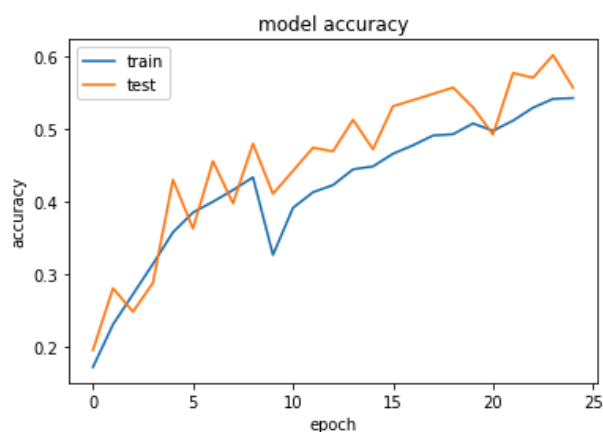
Po stworzeniu modelu, który wykazał, że zadanie stworzenia dobrze działającej sieci dla tego problemu jest wykonalne, zabrano się do kolejnego etapu prac. W tym celu wykorzystano opisany w poprzednim rozdziale zbiór obrazów kwadratowych złożony ze 100800 zdjęć podzielonych na 80640 i 20160 w części treningowej i testowej.

W modelu wykorzystano architekturę nieznacznie zmienioną w stosunku do pierwszego modelu. Poniżej znajduje się opisywany model (zob. rys. 6.3):

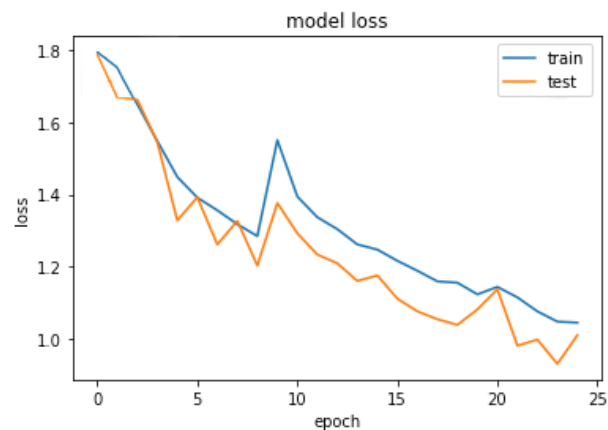


Rysunek 6.3: Model bazujący na optymalizatorze Adam

Tak jak poprzednio sieć uczona była przez 25 epok. Po pierwszych 25 epokach uzyskano dokładność 55,64% co potwierdziło przypuszczenie z poprzedniej sieci o przetrenowaniu i konieczności zróżnicowania danych. Poniżej znajdują się wykresy (zob. rys. 6.4) uczenia się modelu:



Skuteczność rozpoznawania kości

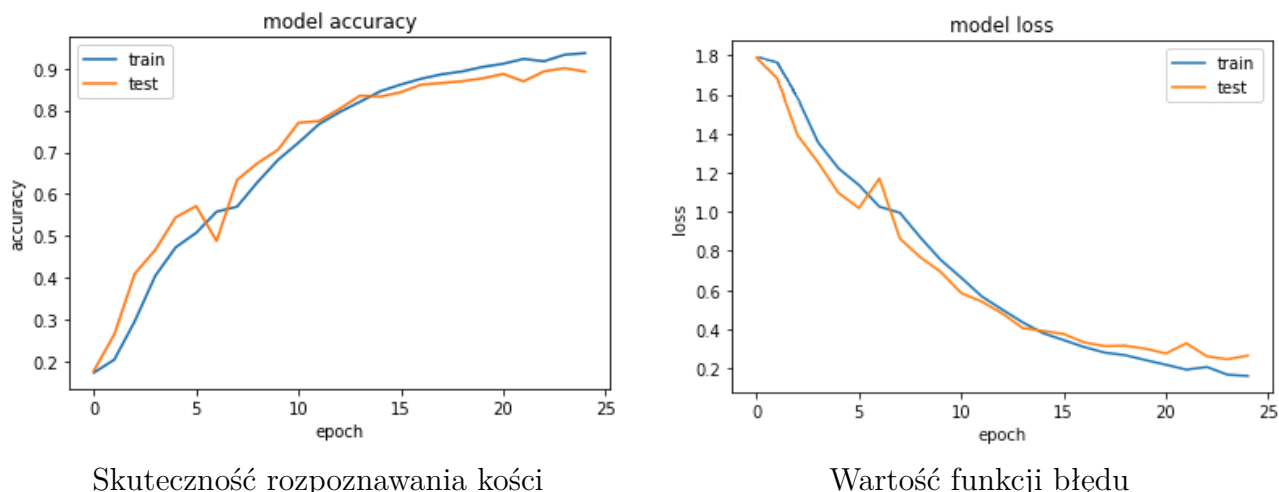


Wartość funkcji błędu

Rysunek 6.4: Wykresy dla modelu z optymalizatorem Adam

Model z wybranymi zdjęciami

Kolejnym punktem sprawdzenia jak zróżnicowanie danych wpływa na wyniki sieci, było wykorzystanie tylko niektórych zestawów danych z całego zbioru 100800 kwadratowych obrazów. Zestawy były dobrane tak, by kontrast między tłem a kością był wyraźny. Architektura sieci była identyczna z powyższym przykładem. 25 epok wystarczyło do uzyskania dokładności na poziomie 89,23% co jest rezultatem zdecydowanie lepszym niż uzyskane wcześniej 55,64%. Przebieg uczenia sieci widoczny jest poniżej (zob. rys. 6.5):



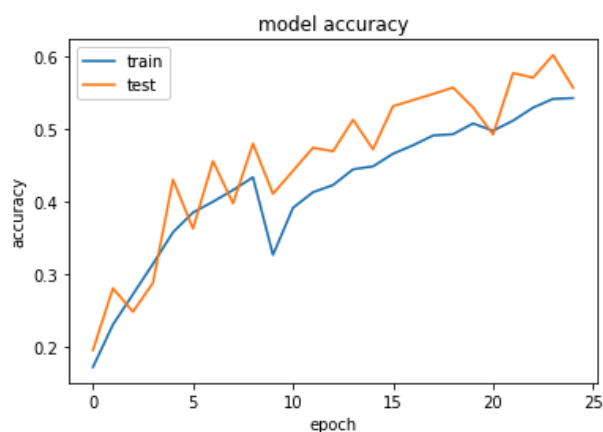
Rysunek 6.5: Wykresy dla modelu korzystającego z wybranych zestawów kolorystycznych

6.2 Analiza różnych optymalizatorów

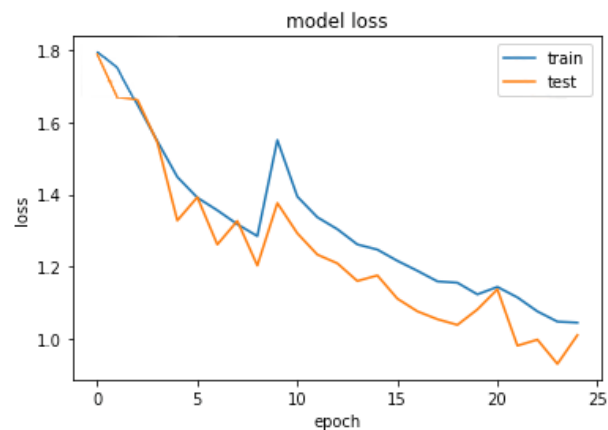
Następnym krokiem analizy sieci neuronowych było przetestowaniu kilku z dostępnych w bibliotece Keras optymalizatorów dla identycznych sieci. Zbiorem danych był opisany w poprzednim rozdziale zbiór złożony z 100800 kwadratowych obrazów. W tym celu postanowiono użyć optymalizatorów RMSprop oraz SGD.

Oba modele z optymalizatorami RMSprop oraz SGD zostały, podobnie jak wcześniej opisany model z Adam, poddane uczeniu przez 25 epok różnorodnych zbiorów obrazów. Wynik RMSprop był zdecydowanie wyróżniający się i wyniósł 84,34% skuteczności. Najgorszy rezultat w zestawieniu uzyskał model korzystający z SGD, zdobywając jedynie 36,92% poprawności. Pośrednim okazał się Adam, który jak opisane wyżej, uzyskał 55,64% po sesji uczenia przez 25 epok.

Wyniki te dowiodły, że optymalizator jest kluczowym parametrem (*ang. hyperparameter*) dla odpowiednio skutecznego uczenia. Uzyskany wynik dla RMSprop jest sporym zaskoczeniem, ponieważ w licznych tekstach naukowych to Adam uznawany jest za jeden z najlepszych optymalizatorów, ciesząc się ogromną popularnością. Również z tego powodu, pomimo gorszego niż RMSprop wyniku, Adam będzie wykorzystywany w następnych modelach. Poniżej znajdują się wyniki dla każdego z analizowanych optymalizatorów (zob. rys. 6.6, 6.7, 6.8):

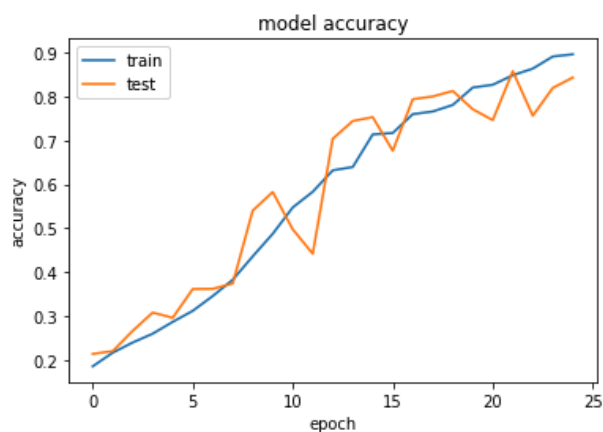


Skuteczność rozpoznawania kości

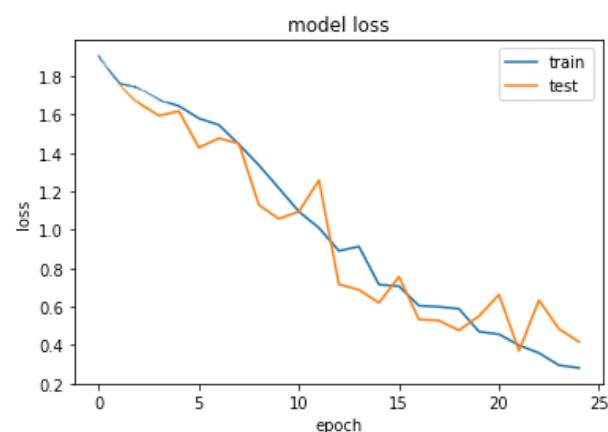


Wartość funkcji błędu

Rysunek 6.6: Wykresy dla modelu korzystającego z optymalizatora Adam

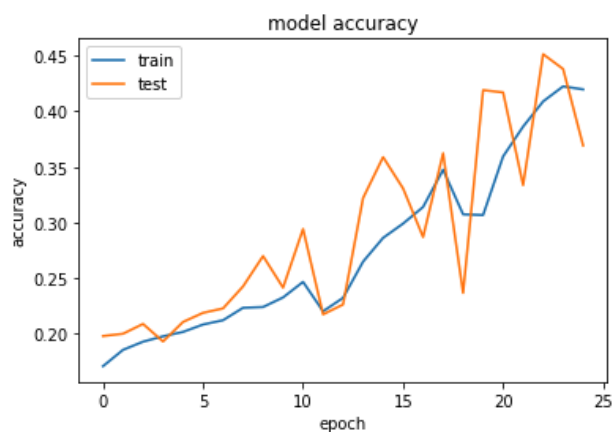


Skuteczność rozpoznawania kości

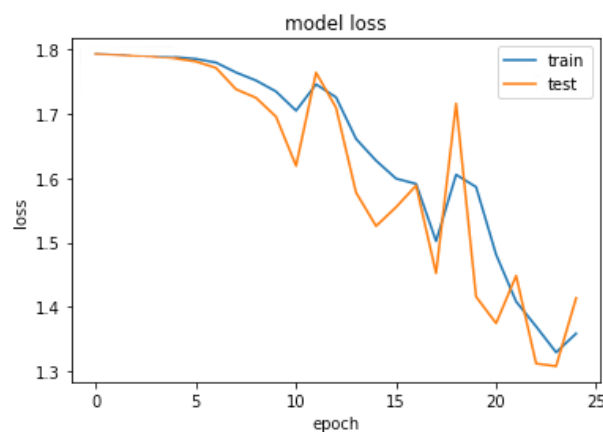


Wartość funkcji błędu

Rysunek 6.7: Wykresy dla modelu korzystającego z optymalizatora RMSprop



Skuteczność rozpoznawania kości

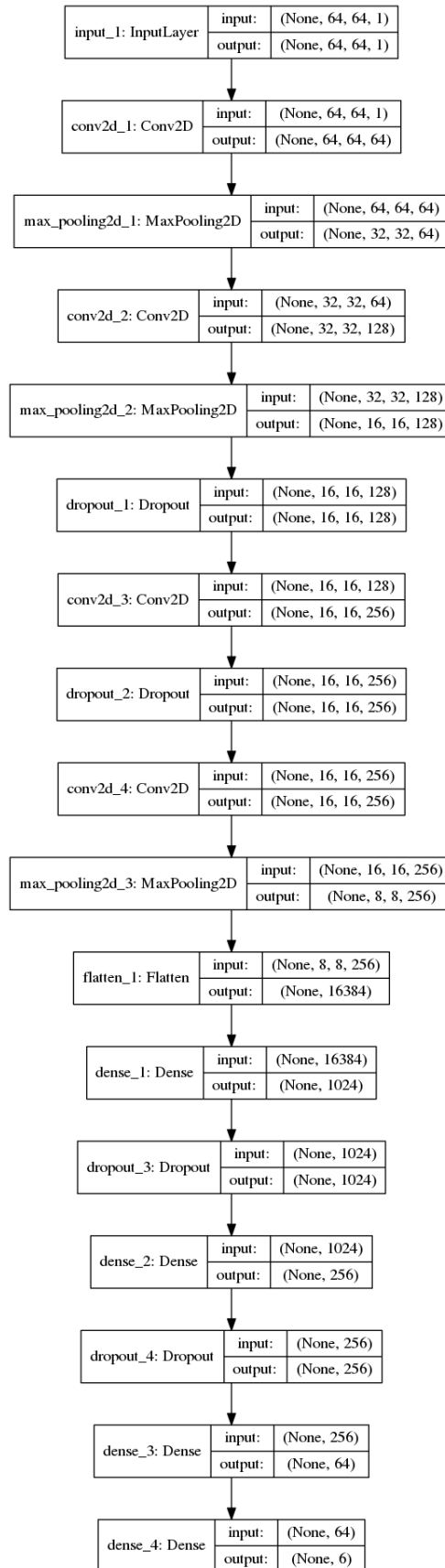


Wartość funkcji błędu

Rysunek 6.8: Wykresy dla modelu korzystającego z optymalizatora SGD

6.3 Wykorzystanie sieci AlexNet

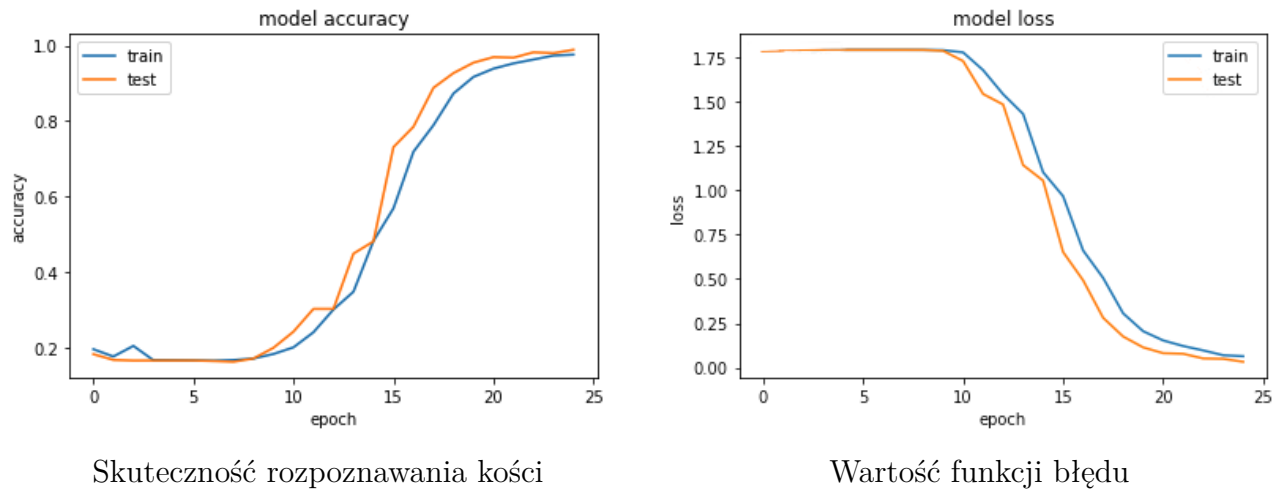
Sieć o nazwie AlexNet [24, 25] została zaprezentowana przez Alexa Krizhevskyego, Geoffreya Hintoną oraz Ilya Sutskevera w 2012 roku. Jej ideą jest zastosowanie większej ilości warstw konwolucyjnych, gdzie początkowe dwie warstwy mają filtry rozmiarów odpowiednio 11x11 oraz 5x5. Następnie umieszczone są trzy warstwy konwolucyjne z filtrami 3x3, które w przeciwieństwie do pierwszych dwóch warstw nie są rozdzielone warstwami z maxpoolingiem. AlexNet zdeklasowała rywali w konkursie ILSVRC 2012 na rozpoznawanie obrazów. Dzięki świetnym wynikom, zdecydowano się na realizację modelu przypominającego jej budowę. Zastosowano model bazujący na idei AlexNet, z uproszczoną architekturą, która przedstawiona jest poniżej (zob. rys. 6.9):



Rysunek 6.9: Model bazujący na Alexnet

Na zbiorze 100800 kwadratowych obrazów, po 25 epokach osiągnięto poprawność wynoszącą aż 98.88%. Na wykresach uczenia (zob. rys. 6.10) można zaobserwować, że przez pierwsze 10

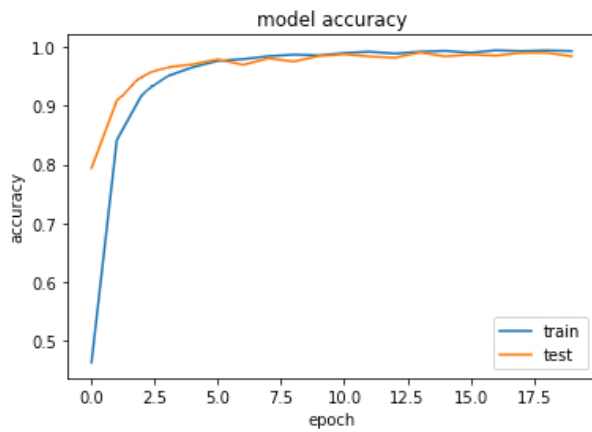
epok precyzja przewidywań i wartość funkcji kosztu sieci praktycznie się nie zmieniały. Obserwacja sugeruje, że sieci głębsze mogą potrzebować większej ilości epok do rozpoczęcia procesu prawidłowego rozpoznawania obrazów.



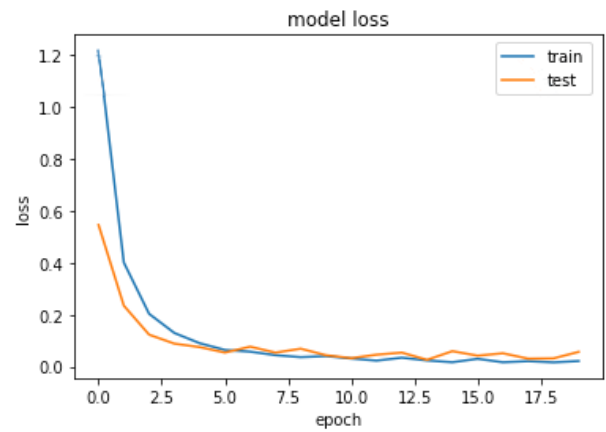
Rysunek 6.10: Wykresy dla sieci opartej o AlexNet

6.4 Porównanie dla obrazów kolorowych i czarno-białych

Obraz w skali szarości posiada jedynie jeden kanał odpowiadający jasności. Obrazy kolorowe RGB posiadają trzy kanały informujące o nasyceniu odpowiednio czerwonego, zielonego i niebieskiego koloru. Wzrost ilości informacji wiąże się z większym obciążeniem pamięci i większą ilością parametrów. W celu weryfikacji różnicy między uczeniem obu rodzajów obrazów podjęto próbę porównania wyników uczenia dla dwóch jednakowych sieci. Pierwsza z sieci korzystała z kolorowej wersji zbioru obrazów kwadratowych, dla których zastosowano kąt obrotu 30° , uzyskując 50400 obrazów. Sieć korzystająca z czarno-białych obrazów miała te same zdjęcia z jednym zamiast trzech kanałów informujących o kolorze. Do tego eksperymentu wykorzystano narzędzie generatora dostępne w Keras, które przekazuje obrazy do sieci bez konieczności wcześniejszego ładowania całego zbioru do pamięci. Architektura obu sieci była uproszczona, by zniwelować długi czas uczenia. Wykresy (zob. rys. 6.11) przedstawiają proces uczenia na obrazach kolorowych, a wykresy (zob. rys. 6.12) na obrazach w odcieniach skali szarości:

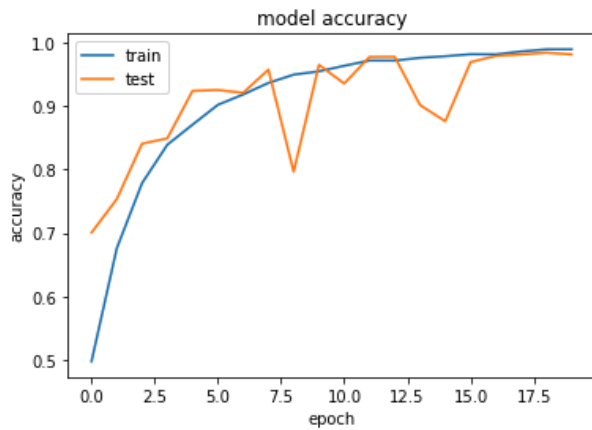


Skuteczność rozpoznawania kości

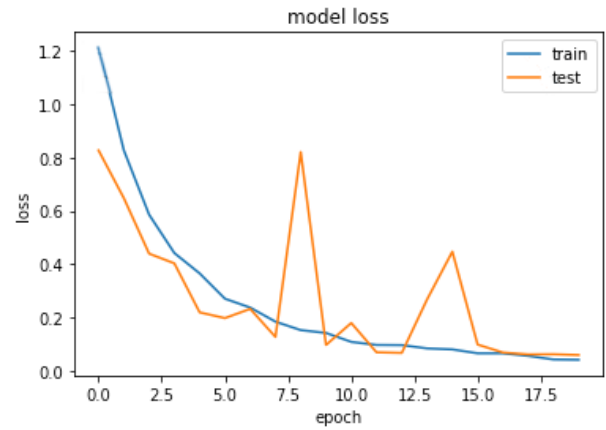


Wartość funkcji błędu

Rysunek 6.11: Wykresy dla sieci z obrazami kolorowymi



Skuteczność rozpoznawania kości



Wartość funkcji błędu

Rysunek 6.12: Wykresy dla sieci z obrazami w skali szarości

Jak można zauważyć, oba modele po 20 epokach wykazały się bardzo zbliżonymi wynikami 98,47% oraz 98,12% na korzyść sieci z kolorowymi obrazami. Model z lepszym wynikiem nauczył się bardzo szybko do poziomu 95%, osiągając go już po 4 epoce, gdzie drugi osiągnął ten wynik po 10 epokach. Obserwacja ta może sugerować, że większa różnica w wartościach dla kolorowych obrazów może przyspieszać uczenie, ale wymaga większej ilości pamięci i spowalnia każdą epokę o 15%.

6.5 Próby wykorzystania prostokątnych obrazów

Dotychczasowe modele korzystały ze zbiorów o obrazach w kształcie kwadratów z kośćmi umieszczonymi w ich środkowej części. Drugi rodzaj przygotowanych zbiorów danych zwiększał trudność zadania, dzięki zastosowaniu obrazów prostokątnych z proporcjonalnie mniejszym rozmiarem kości w stosunku do pierwszego rodzaju obrazów.

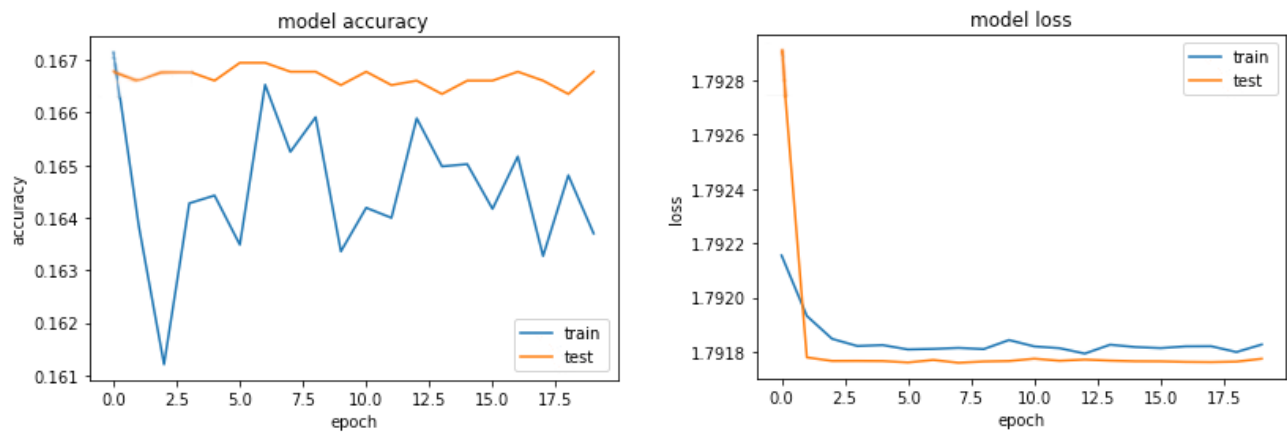
Po uzyskaniu wielu bardzo dobrych wyników powyżej 90% na obrazach o rozmiarach 64x64 przystąpiono do prób ze znacznie zwiększonymi obrazami prostokątnymi.

Pierwszą próbą było wykorzystanie obrazów 320x240 zarówno w wersjach kolorowych, jak i w skali szarości. Jeden z modeli bazował na architekturze AlexNet, drugi bezpośrednio ją kopiował, ale pomimo świetnego wyniku na mniejszych obrazach, w obu przypadkach uczenie zakończyło się całkowitą klęską. Przez wszystkie 25 epok skuteczność rozpoznawania nie poprawiała się w żadnym stopniu. Warto wspomnieć, że czas potrzebny na jedną epokę był około 15-krotnie większy niż w przypadku obrazów 64x64.

Ostania próba z obrazami w tym rozmiarze zakładała użycie uproszczonej architektury, podobnie jak przy porównaniu uczenia obrazów RGB i w skali szarości. Finalnie, tak jak wcześniej, pomimo 20 epok sieć nie wykazała żadnego postępu w rozpoznawaniu kości.

Niepowodzenia spowodowały konieczność zmniejszenia ilości danych do przetworzenia przez ograniczenie rozmiarów obrazów. Kluczowa przy tym była chęć uniknięcia problemów z brakiem ostrości oczek na kostce co mogłoby uniemożliwić skuteczną naukę. Zdecydowano się na dwukrotne zmniejszenie rozmiarów obrazów, licząc, że pozwoli to na zaobserwowanie, chociaż niewielkich postępów.

Sieć bazująca na zdjęciach 160x120 była pierwszą próbą, gdzie zamiast jak dotychczas kwadratowych, użyto prostokątnych filtrów konwolucyjnych. Proces uczenia po 20 epokach niestety również zakończył się niepowodzeniem. Wykresy (zob. rys. 6.13) uczenia się przedstawione są poniżej:



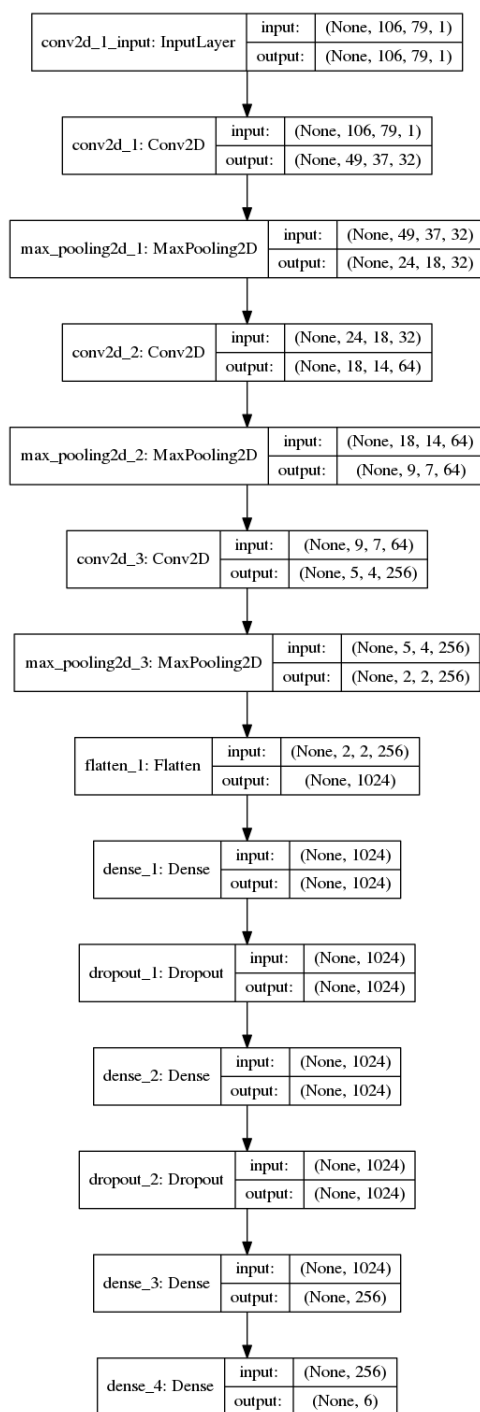
Skuteczność rozpoznawania kości

Wartość funkcji błędu

Rysunek 6.13: Wykresy dla nienauczonej sieci z obrazami 160x120

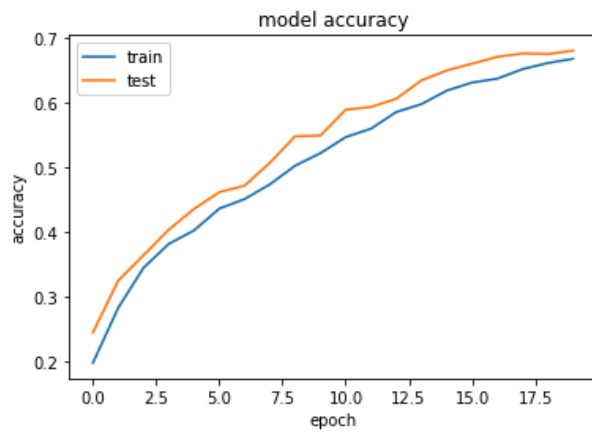
Wytrenowany prostokątny model

Powyższe porażki oraz wcześniejsze sukcesy na kwadratowych obrazach sugerowały, że obrazy mogą mieć za duży rozmiar. To przypuszczenie rozpoczęło dobieranie odpowiedniej rozdzielczości zdjęć tak by rozmiar był mały, a jednocześnie nie powodował zanikania informacji o oczkach na kostce. Najlepszym wyborem okazały się zdjęcia w rozmiarze 106x79, zachowujące proporcje identyczne z obrazami 160x120, ale posiadające o 50% mniejszą liczbę parametrów. Poniżej przedstawiony jest model tej sieci (zob. rys. 6.14):

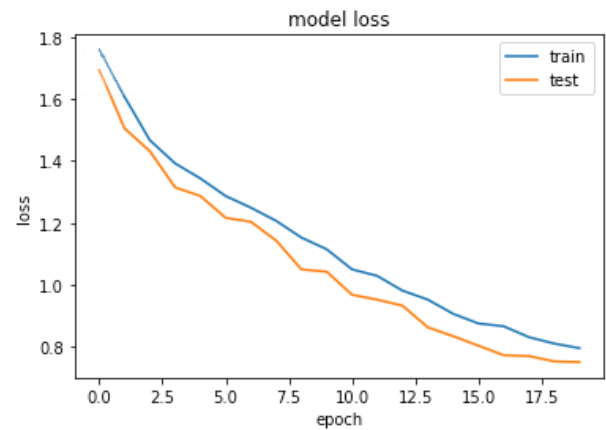


Rysunek 6.14: Model bazujący na Alexnet

Pierwsza próba przeprowadzona przez 20 epok z filtrami konwolucyjnymi o prostokątnych kształtach wreszcie zakończyła się sukcesem. Sieć osiągnęła wynik 68,03% co nie było świetnym rezultatem, ale sugerowało, że dalsza nauka jest możliwa. Wykres (zob. rys. 6.15) uczenia znajduje się poniżej.



Skuteczność rozpoznawania kości



Wartość funkcji błędów

Rysunek 6.15: Wykresy dla nauczanej sieci z obrazami 106x79

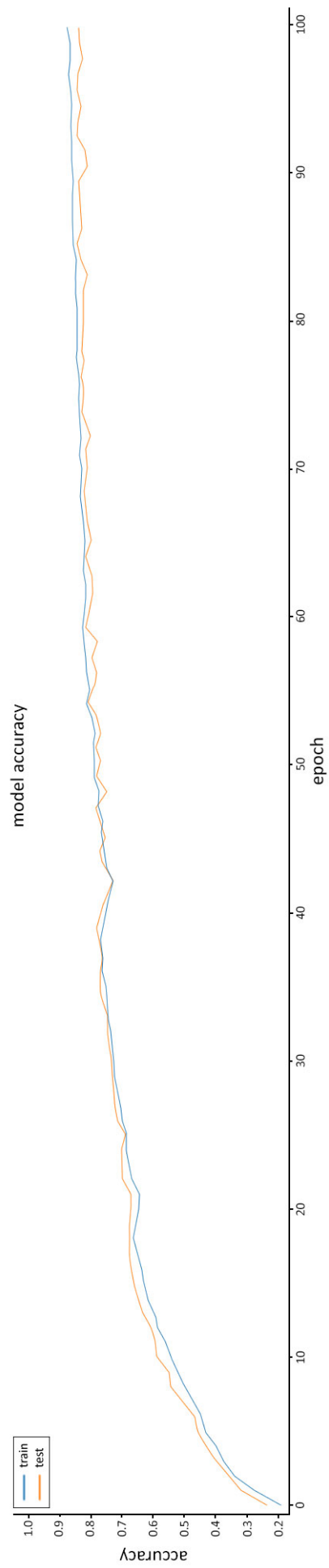
6.6 Najskuteczniejszy model z prostokątnymi obrazami

Wyżej przedstawiony wykres uczenia kształtem przypominał funkcję logarytmiczną, co wskazywało na możliwość dalszej nauki sieci z obrazami 106x79 pikseli. Nauka była kontynuowana do kolejno 40, 60, 80 i 100 epok.

Podejście to okazało się bardzo skuteczne, ponieważ po każdych 20 epokach sieć odnosiła coraz lepsze rezultaty, które prezentowały się następująco:

- 20 epok: 68,03%
- 40 epok: 78,21%
- 60 epok: 81,59%
- 80 epok: 82,39%
- 100 epok: 84,67%

Wyniki uświadamiają, że prawdopodobnie nawet nieudane próby z większymi zdjęciami mogłyby się powieść, konieczne byłoby jedynie zwiększenie ilości epok. Wiąże się to z ogromnym nakładem czasu, ponieważ prawdopodobnie sensowne rezultaty można by osiągnąć dopiero po 100 epokach. Wykres (zob. rys. 6.16) dokładności sieci znajduje się na następnej stronie.



Rysunek 6.16: Wykres uczenia przez 100 epok

6.7 Doskonalenie modelu z prostokątnymi obrazami

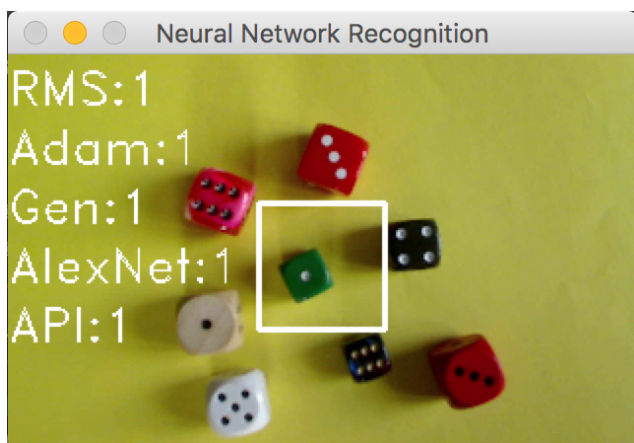
Po sukcesie sieci z obrazami prostokątnymi podjęto decyzję o jej ulepszeniu. Zaczęto od próby zmniejszenia ilości parametrów przez zamianę większych filtrów konwolucyjnych, większą ilością mniejszych. Zabieg ten znacząco zmniejszał ilość parametrów sieci potrzebną do nauczania [7]. Po zaaplikowaniu ulepszeń i nauce sieci okazało się, że model nie poprawił się w żadnym stopniu. Prawdopodobnym powodem była zmiana architektury sieci wraz z powtórным zastosowaniem kwadratowych filtrów.

Drugim pomysłem na ulepszenie sieci było zastosowanie zastępowania większych filtrów kilkoma mniejszymi. Dodatkowo w celu uniknięcia problemów z zanikającym neuronem zmieniono z ReLU na LeakyReLU. Ulepszenie jednak podobnie jak wcześniejsze nie sprawdziło się w ogóle i nie umożliwiło poprawy dokładności sieci.

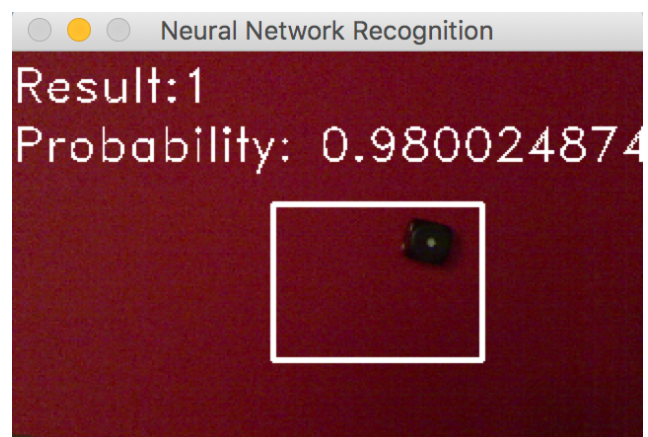
7. Podsumowanie

Zwieńczeniem całej pracy i stworzonych modeli są dwa skrypty rozpoznające ilość oczek wyrzuconych na kości w czasie rzeczywistym. Programy korzystają z wytrenowanych modeli załadowanych do pamięci, które analizują obraz dostarczany z kamery podłączonej do komputera. Modele są przystosowane do rozpoznawania kości na obrazach o wielkościach 64x64 oraz 106x79 pikseli.

Obraz 1600x1200 dostarczany z kamery jest wielokrotnie większy, co wymusiło wyznaczenie jego części, która jest analizowana przez zaaplikowane sieci neuronowe. Ta część obrazu została oznaczona białymi krawędziami dla łatwiejszego zlokalizowania miejsca, w którym powinna być umieszczona kość pod kamerą (zob. rys. 7.1) :



Rozpoznawanie obrazów 64x64



Rozpoznawanie obrazów 106x79

Rysunek 7.1: Przykłady działania sieci neuronowych

Dla obrazów 64x64 wykonano kilka modeli sieci, które osiągnęły powyżej 90% skuteczności. Program umożliwia porównanie wyników dzięki zaaplikowaniu ich do analizy tego samego obszaru na zdjęciu. Pozwala to zauważyć między innymi lepsze rozpoznawanie pewnych kombinacji kolorystycznych niektórych modeli lub trudności z rozróżnieniem wyrzuconych czterech oczek od sześciu, z uwagi na efekt zlewania się ich podczas skalowania.

Dla obrazów 106x79 zaaplikowany jest jedynie jeden model, wytrenowany przez 100 epok.

Wyniki zaobserwowane podczas prac związanych z tworzeniem sieci neuronowych przyniosły wiele niespodziewanych sytuacji. Wielokrotnie otrzymywane wyniki były zdumiewające, biorąc pod uwagę, jak drobne różnice w budowie sieci wpływały na końcowe rezultaty.

Wszystkie eksperymenty pokazały, że nawet proste zadanie rozpoznawania oczek na kostce wymaga dużej ilości danych i mocy obliczeniowej, by uzyskać satysfakcjonujące wyniki. Przykład

ten pokazuje także jak wiele zastosowań może mieć zyskujące na popularności uczenie maszynowe, które wraz z rozwojem technologii będzie mieć coraz więcej możliwości na dostosowywania algorytmów do określonych danych.

Bibliografia

- [1] Stanford University course CS231n
<http://cs231n.github.io/convolutional-networks/>
- [2] Aditya V. D
<https://becominghuman.ai/neural-network-xor-application-and-fundamentals-6b1d539941ed>
- [3] Eric Roberts, Stanford University
<https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/>
- [4] Stanford University course CS231n
<http://cs231n.github.io/neural-networks-1/>
- [5] Tuples Edu
<https://becominghuman.ai/what-is-an-artificial-neuron-8b2e421ce42e>
- [6] Jaron Collis
<https://medium.com/deeper-learning/glossary-of-deep-learning-bias-cf49d9c895e2>
- [7] Leonardo Araujo dos Santos
https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/convolutional_neural_networks.html
- [8] James D. McCaffrey
<https://jamesmccaffrey.wordpress.com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/>
- [9] Stanford University course CS231n
<http://cs231n.github.io/optimization-2/>
- [10] Leonardo Araujo dos Santos
<https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/backpropagation.html>
- [11] Ujjesh Karn
<https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>

- [12] https://en.wikipedia.org/wiki/Convolutional_neural_network
- [13] Krzysztof Sopyła
<https://ksopyla.com/python/operacja-splotu-przetwarzanie-obrazow/>
- [14] Jason Morrison
<https://medium.com/paper-club/paper-review-dropout-a-simple-way-to-prevent-neural-networks-from-overfitting-4f25e8f2283a>
- [15] Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, Rob Fergus
<https://cs.nyu.edu/~wanli/dropc/>
- [16] Avinash Sharma V
<https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0>
- [17] Sagar Sharma
<https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>
- [18] [https://en.wikipedia.org/wiki/Rectifier_\(neural_networks\)](https://en.wikipedia.org/wiki/Rectifier_(neural_networks))
- [19] Anish Singh Walia
<https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f>
- [20] Sebatian Ruder
<http://ruder.io/optimizing-gradient-descent/>
- [21] Diederik P. Kingma, Jimmy Ba
<https://arxiv.org/abs/1412.6980>
- [22] Yann LeCun, Corinna Cortes, Christopher J.C. Burges
<http://yann.lecun.com/exdb/mnist/>
- [23] Alex Krizhevsky
<https://www.cs.toronto.edu/~kriz/cifar.html>
- [24] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton
<https://www.nvidia.cn/content/tesla/pdf/machine-learning/imagenet-classification-with-deep-convolutional-nn.pdf>
- [25] Tugce Tasci, Kyunghee Kim
http://vision.stanford.edu/teaching/cs231b_spring1415/slides/alexnet_tugce_kyunghee.pdf

8. Dodatek A - zbiory danych

Zbiory danych wykorzystane do uczenia przedstawionych modeli sieci neuronowych znajdują się w linku dostępnym poniżej:

https://drive.google.com/open?id=1SfoUCMWL6tiGv2biTL_ZN4DeJfU3Z-Q_

- *photos_64x64x1.zip* - Zbiór 64x64x1, obrazy w skali szarości, opisany w części 5.1
- *photos_64x64x3_gen.zip* - Zbiór 64x64x3, obrazy kolorowe, przygotowane do generatora, zbiór opisany w części 5.1
- *photos_64x64x1_gen.zip* Zbiór 64x64x1, obrazy w skali szarości, przygotowane do generatora, zbiór opisany w części 5.1
- *photos_106x79_gen.zip* - Zbiór 106x79x1, obrazy w skali szarości, przygotowane do generatora, zbiór opisany w części 5.2

9. Dodatek B - modele sieci neuronowych

Wszystkie modele sieci neuronowych przedstawione w pracy dostępne są pod linkami podanymi poniżej:

9.1 Obrazy kwadratowe 64x64

Pierwszy model z obrazami w jednej kolorystyce

https://github.com/oziomek1/NNLearning/blob/master/dices_cnn/dice_cnn.ipynb

Pierwszy model ze zróżnicowanymi zdjęciami

https://github.com/oziomek1/neural_network_dice/blob/master/dice_cnn_adam.ipynb

Model z wybranymi zdjęciami

https://github.com/oziomek1/neural_network_dice/blob/master/dice_cnn_adam_wybrane.ipynb

Modele do analizy optymalizatorów

Optymalizator Adam:

https://github.com/oziomek1/neural_network_dice/blob/master/dice_cnn_adam.ipynb

Optymalizator RMSprop:

https://github.com/oziomek1/neural_network_dice/blob/master/dice_cnn_rmsprop.ipynb

Optymalizator SGD:

https://github.com/oziomek1/neural_network_dice/blob/master/dice_cnn_sgd.ipynb

Model oparty o AlexNet

https://github.com/oziomek1/neural_network_dice/blob/master/dice_AlexNet.ipynb

Porównanie modeli dla obrazów kolorowych i w skali szarości

Obrazy kolorowe:

https://github.com/oziomek1/neural_network_dice/blob/master/generator_comparison.ipynb

Obrazy w skali szarości:

https://github.com/oziomek1/neural_network_dice/blob/master/generator_comparison_gray.ipynb

9.2 Obrazy prostokątne

Trzy nienauczone modele z prostokątnymi obrazami 320x240

https://github.com/oziomek1/neural_network_dice/blob/master/generator_320x240_AlexNet.ipynb

https://github.com/oziomek1/neural_network_dice/blob/master/generator2_320x240_AlexNet.ipynb

https://github.com/oziomek1/neural_network_dice/blob/master/generator3_320x240.ipynb

Model ze prostokątnymi obrazami 160x120

https://github.com/oziomek1/neural_network_dice/blob/master/simple_NN_160x120.ipynb

Nauczony model z prostokątnymi obrazami 106x79

https://github.com/oziomek1/neural_network_dice/blob/master/simple_NN_106x79.ipynb

Ulepszanie modelu przez redukcję dużych konwolucji

https://github.com/oziomek1/neural_network_dice/blob/master/substituting_106x79.ipynb

Ulepszanie modelu przez LeakyReLU

https://github.com/oziomek1/neural_network_dice/blob/master/subst_LReLU_106x79.ipynb

Kontynuacja uczenia jedyne go nauczonego modelu 20-60 epok

https://github.com/oziomek1/neural_network_dice/blob/master/simple_NN_106x79_continue.ipynb

Kontynuacja uczenia jedyne go nauczonego modelu 60-100 epok

https://github.com/oziomek1/neural_network_dice/blob/master/simple_NN_106x79_continue_80_100.ipynb

9.3 Wytrenowane modele

Wytrenowane modele do weryfikacji działania na obrazie z kamery

<https://drive.google.com/drive/folders/1H1JJ22AM9Jps96LIBpt-Mwf9tkqnvwtY>

9.4 Programy weryfikujące modele

Program z obrazami 64x64x1

https://github.com/oziomek1/neural_network_dice/blob/master/camera_with_neural_network_test.ipynb

Program z obrazami 106x79x1

https://github.com/oziomek1/neural_network_dice/blob/master/camera_CNN_106x79.py