Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Wojciech Ozimek

Nr albumu: 1124802

Wykorzystanie sztucznych sieci neuronowych do analizy obrazów na przykładzie kostki 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

Celem pracy jest zastosowanie sieci neuronowych do rozpoznawania obrazów. Zadanie polega na rozpoznaniu ilość 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, a pierwsze prace nad nimi zaczęły sie już w latach 50. XX wieku.

2.1 Sztuczny neuron

Najprostszym modelem 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].
Tak zbudowany, sztuczny model neuronu jest podstawowym składnikiem sztucznych sieci neuronowych.
Neurony najczęściej łączone są w struktury nazywane sieciami neuronowymi, ponieważ wykorzystanie pojedynczego neuronu nie pozwala na rozwiązywanie skomplikowanych zadań. Przykładowo próba realizacji prostych funkcji logicznych ANR, OR, NOT i XOR okazuje się możliwa jedynie dla trzech pierwszych podanych funkcji [14]. 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 wspomnianym wyżej sztucznym neuronem można datować na rok 1943 kiedy Warren McCulloch i Walter Pitts przedstawili pierwszy jego model. [4]. Pierwsze sieci neuronowe używane były do operacji bitowych oraz przewidywania 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 jezyku programowania Python, który obecnie jest bardzo popularnym narzędziem wykorzystywanym w pracy naukowej. Podowem jego tego wyboru jest bardzo czytelna i zwięzła składnia, która w pełni pozwala skupić się na danym problemie. Python jest on 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, nazywane notebookami, które składają się z wielu bloków. W blokach może znajdować się wykonywalny kod programu lub jego fragment, a w pozostałych można prezentować m.in. teksty, wykresy bądź tabele, co zdecydowanie ułatwia pracę.

3.2 Biblioteki

OpenCV

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

TensorFlow

TensorFlow jest biblioteką ułatwiającą tworzenie sieci neuronowych. Jej ogromną zaletą jest implementacja w języku C++ umożliwiająca wykorzystywanie procesorów oraz kart graficznych. Z uwagi na charakter wykonywanych obliczeń praca przy użyciu kart graficznych jest

kilkukrotnie szybsza niż na procesorze, co znacząco przyśpiesza proces uczenia. Biblioteka najlepiej wspiera języki programowania C++ oraz Python.

Keras

Wykorzystywaną 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 przyśpieszone dzięki wykorzystaniu modułu Numpy. Moduł umożliwia wykonywanie zaawansowanych operacji na macierzach oraz wektorach, wspierający liczne funkcje matematyczne. Jest bardzo rozpowszechniony i wykorzystywany w wielu, głównie naukowych zastosowaniach. 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 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 do prezentacji w formie graficznej. Ogromną zaletą jest możliwość tworzenia w tekście zaawansowanych wzorów matematycznych. Tekst niniejszej pracy napisany został przy pomocy tego narzędzia.

3.3 Technologie wspomagające

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

NVIDIA CUDA

Technologia obecna w kartach firmy NVIDIA to równoległa architektura obliczeniowa, pozwalająca na wielokrotne przyśpieszneie 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 wykorzytane w tej pracy były dostępne na plaftormie 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

4.1 Zbiór danych

Do realizacji pracy konieczny 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 osiągnięcia żądanego rozmiaru, który ma kluczowe znaczenie w przypadku zastosowań sieci neuronowych. Oprócz zmniejszenia obrazy były również obracane, deformowane *(warping)* oraz kadrowane. Tak powstałe, liczne zbiory były wykorzystywane w trybie RGB oraz w odcieniach skali szarości, co pozwoliło na oszczędzenie pamięci 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 jest wykorzystywana w procesie uczenia sieci jako docelowa informacja, którą ma zwrócić sieć po weryfikacji danego obrazu.

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

Zbiór testowy
Do oceny zdolności sieci neuronowej do rozpoznawania danych służy zbiór testowy lub walidacyjny. Celem rozdzielenia tego zbioru od danych testowych jest weryfikacja sieci na danych, które nie zostały przez nią przetworzone podczas treningu.

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 [10, 16]. 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.

Neuron

Neuron jest najmniejszym elementem sieci neuronowej, posiadającym wiele wejść i jedno wyjście [3, 4, 5]. Sygnały z neuronów w poprzednich warstwach docierają na każde z wejść neuronu z przypisaną mu wagą. W neuronie obliczana jest suma ważona wejść, która porównywana jest zwartością progową, inaczej biasem [6]. Jeśli przekroczy ona wartość progową to neuron jest uaktywniany, a suma przekazywana jest do funkcji aktywacji. Wartość funkcji przekazywana jest na wyjście neuronu.

Poniższy wzór 4.1 przedstawia sumę ważoną wejść neuronu wraz z dodatkową wagą.

$$f(x_i) = f\Big(\sum_i w_i x_i + b\Big) \tag{4.1}$$

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

Wzór 4.1: Suma ważona w neuronie wraz z dodatkową wagą b

Wagi neuronu

Jak opisano już wcześniej, neurony połączone są między sobą dzięki licznym wejściom. Każde takie połączenie ma przypisaną wagę, która zmienia się podczas treningu, dostosowując sieć neuronową do danych treningowych. Efektem tego jest osiąganie coraz lepszych wyników w miarę trwania procesu uczenia.

Warstwa

Neurony w sieci zorganizowana są w warstwach. Komunikacja odbywa się tylko między kolejnymi warstwami, neurony w danej warstwie nie są ze sobą połączone [1, 7]. Istnieje wiele rodzajów warstw, a sygnał przechodzący przez całą sieć zaczyna się w tzw. warstwie wejściowej oraz kończy w tzw. warstwie wyjściowej.

Funkcja kosztu

Do uzyskiwania informacji o wartości błedu pomiędzy wartościami przewidzianymi przez sieć a wartościami docelowymi wykorzystywana jest funkcja kosztu lub funkcja błędu. Jest ona niezbędna do prawidłowego przeprowadzenie 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 4.2. W tej pracy z uwagi na posiadanie 6 możliwych do uzyskania wyników na wyjściu sieci zastosowano wielowymiarową entropię krzyżową (*ang. Categorical cross-entropy*) 4.3, która jest preferowana, po-

nieważ uwypukla aż tak bardzo nieprawidłowych wartości [31]. Jest również sugerowana przez bibliotekę Keras oraz wykorzystywana w wielu przykładowych modelach sieci neuronowych.

$$J(\eta) = \frac{1}{2}\sum_{i}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \tag{4.2}$$

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

Wzór 4.2: Błąd średniokwadratowy

$$L_i = -\sum_{j} t_{i,j} log(p_{i,j}) \tag{4.3}$$

gdzie: p - predykcje, t - cele, i - wartość, j - klasa

Wzór 4.3: Wielowymiarowa entropia krzyżowa

4.3 Propagacja wsteczna

Propagacja wsteczna lub wsteczna propagacja błędów *(ang. Backpropagation)* jest algorytmem uczenia sieci neuronowych [2, 8]. 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 wejściowe, czemu algorytm zawdzięcza swoją nazwę.

4.4 Konwolucyjna sieć neuronowa

Przy wykorzystywaniu sieci neuronowych do operowania na zdjęciach, pojawia się problem dużą ilością parametrów odpowiadających wartościom każdego z pikseli. W celu ich zmniejszenia stosuje się konwolucyjne sieci neuronowe *(ang. CNN - Convolutuinal Neural Network)*. Do kolejnych warstw zamiast przekazywania informacje o wszystkich pikselach znajdujących się na obrazach, sieć analizuje obraz przy użyciu filtrów konwolucyjnych i przesyła dalej informacje o zaobserwowanych cechach.
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 splot 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 splotu, w zależności od przypadku, pozwala na rozmycie, wyostrzanie lub wydobycie głębi z danego obrazu [30]. Poniższy wzór 4.4 przedstawia sposób obliczenia splotu dwóch funkcji.

$$h[m,n] = (f * g)[m,n] = \sum_{j}^{m} \sum_{k}^{n} f[j,k] * g[m-j, n-k] \tag{4.4}$$

Wzór 4.4: Splot funkcji. Funkcja f to dwuwymiarowa macierz z pikselami obrazu. Funkcja g to filtr. Funkcja h to nowo otrzymany obraz.

4.5 Warstwy sieci neuronowej

W sieciach neuronowych wyróżniamy kilka rodzajów warstw, które poza wejściową i wyjściową, można dopierać 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. Pierwsza warstwa charakteryzuje się brakiem wejść.

Wyjściowa
Rozmiar warstwy wyjściowej odpowiada ilości klas, do jakiej dane były klasyfikowane. W tej pracy, gdzie czekiwanym wyjściem była liczba oczek możliwych do wyrzucenia na kostce, odpowiada to 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
Najbardziej rozpowszechnionymi typami warstw są w pełni połączone *(ang. Fully Connected, Dense)*. Każdy neuron łączy się ze wszystkimi neuronami następnej warstwy. Skutkuje to dużą ilością potrzebnych do wykonania obliczeń, choć są stosunkowo proste do zaaplikowania. W sieciach konwolucyjnych umieszczane są po lub między warstwami konwolucyjnymi i służą do powiązania nieliniowych kombinacji, które zostały wygenerowane przez warstwy konwolucyjne oraz ich sklasyfikowania.

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 [26]. Warstwa odrzucająca nie wykorzystuje wyjść pewnych neuronów, co zapobiega rozległym zależnościom między neuronami. 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 [25].

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 wykorzystywaną warstwą poolingową jest MaxPooling, wybierający piksel o największej wartości. Pooling jest krytykowany za brak zachowywania informacji o położeniu piksela przekazanego na wyjście warstwy, co może objawiać się błędnymi interpretacjami danych 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.

Sigmoid

Sigmoid popularną funkcją aktywacji. Problemem z nią związanym jest ryzyko zaniknięcia gradientu, co może prowadzić do problemu tzw. umierającego neuronu [12]. Występuje ono, gdy dla danej funkcji aktywacji, gradient staję się bardzo mały, co jest równoznaczne z zaprzestaniem procesu uczenia. W sigmoid gradient może zanikać obustronnie. Wzór funkcji sigmoidalnej pokazany jest poniżej 4.5.

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.5}$$

gdzie: x - wartość wyjścia

Wzór 4.5: Funkcja sigmoidalna

Tangens hiperboliczny

Tangens hiperboliczny lub tanh jest w istocie przekształconą funkcją sigmoidalną [12, 13]. Wykorzystanie jej powoduje większe wahania gradientu. Wzór na obliczenie tangensu hiperbolicznego zaprezentowany jest poniżej 4.6.

$$f(x) = tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})} = \frac{2}{1 + e^{-2x}} - 1 = 2sigmoid(2x) - 1 \tag{4.6}$$

gdzie: x - wartość wyjścia

Wzór 4.6: Tangens hiperboliczny

ReLU

ReLU *(and. Rectified linear unit)* jest najpopularniejszą funkcją aktywacji wykorzystywaną w sieciach neuronowych [3, 17]. Zasługą 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 4.7 na obliczenie ReLU przedstawiony jest poniżej.

$$f(x) = max(0, x) \tag{4.7}$$

gdzie: x - wartość wyjścia

Wzór 4.7: ReLU - Rectified linear unit

LeakyReLU

LeakyReLU jest ulepszeniem funkcji ReLU [3] dzięki zastosowaniu niewielkiego gradientu w sytuacji, dla której ReLU jest nieaktywne. Zmiana ta pozwala na uniknięcie problemu znikającego gradientu. Jej wzór 4.8 przedstawiony jest poniżej.

$$f(x) = \begin{cases} x & \text{if } x \geqslant 0 \\ 0.01x & \text{if } x < 0 \end{cases} \tag{4.8}$$

gdzie: x - wartość wyjścia

Wzór 4.8: LeakyReLU

Softmax

Softmax jest funkcją obliczającą rozkład prawdopodobieństwa wystąpienia danego zdarzenia ze wszystkich możliwych. Wykorzystywana jest w zadagnieniach wymagających przyporządkowania elementu do jednej z wielu klas. Wzór funkcji 4.9 zaprezentowany jest poniżej.

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{k}(e^{x_j})} \tag{4.9}$$

gdzie: x - wartość wyjścia

Wzór 4.9: Funkcja softmax

4.7 Optymalizator

Algorytmy optymalizacyjne wykorzystywane są do minimalizacji funkcji błędu poprzez znajdowanie nowych wartości wag neuronów w sieci neuronowej. Ma kluczowe znaczenie podczas procesu uczenia sieci w kwestii czasu oraz skuteczności. Z tego powodu to zagadnienie jest obiektem wielu obecnych badań i rozwoju sieci neuronowych [9].

Współczynnik uczenia

Do ustalenia z jakąś szybkością sieć neuronowa będzie dostosowywać się do danych, wykorzystujemy współczynnik lub wskaźnik uczenia *(ang. lerning 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 4.10 znajduje się poniżej.

$$\theta = \theta - \eta * \nabla J(\theta) \tag{4.10}$$

gdzie: η- współczynnik uczenia, J - funkcja błedu

Wzór 4.10: Metoda największego spadku

Stochastic gradient descent

Stochastic gradient descent *(skrót. SGD)* jest rozwinięciem metody gradientu prostego, bardzo często wykorzystywana w praktyce [28]. Ulepszenie polega na obliczaniu gradientu dla jednego lub niewielkiej ilości przykładów treningowych. Najczęściej korzysta się z więcej niż jednego przykładu, co zapewnia lepszą stabilność oraz wykorzystuje zrównoleglenie obliczeń. SGD pozwala na lepsze dopasowanie niż metoda gradientu prostego, ale wiąże się z koniecznością zastosowania mniejszego współczynnika uczenia. Sposób obliczania SGD 4.11 przedstawiony jest poniżej.

Adam

Adam to skrót od angielskiej nazwy *Adaptive Moment Estimation* i jest rozwinięciem metody

$$\theta = \theta - \eta * \nabla J(\theta; x_i; y_i) \tag{4.11}$$

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

Wzór 4.11: Stochastic gradient descent

Stochastic Gradient Descent [29, 28]. Metoda ta zapewnia dostosowanie współczynnika uczenia z osobna do każdego z parametrów, 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 4.12 4.13. Obliczone momenty podstawiane są do wzoru

$$\begin{aligned} \hat{m_t} &= \frac{m_t}{1 - \beta_1^t}, \\ \hat{v_t} &= \frac{v_t}{1 - \beta_2^t}, \end{aligned} \tag{4.12}$$

Wzór 4.12: Wzór na obliczanie pierwszego *(m)* i drugiego (v) momentu centralnego

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v_t}} + \epsilon} \hat{m_t} \tag{4.13}$$

gdzie najczęściej $\beta^t{}_1 = 0.9$ oraz $\beta^t{}_2 = 0.99$ a $\epsilon = 10^{-8}$

Wzór 4.13: Optymalizator Adam

4.8 Procesy

Pracy z sieciami neuronowymi towarzyszy kilka procesów, które są niezbędne do tworzenia oraz korzystania z nich.

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 ilośći oczek wyrzuconych na kostce. Trening jest bardzo kosztowny obliczeniowo, ponieważ operacje dodawania i mnożenia wektorów oraz macierzy wykonywane są miliony razy.

Epoka

Podczas uczenia się, przez sieć przechodzą wszystkie elementy z treningowego zbioru danych. Jednorazowe przejście wszystkich tych elementów 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.

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.

Przetrenowanie

Przetrenowanie *(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. Jedną z najlepszych technik na uniknięcie przetrenowania jest zastosowanie warstw odrzucających [26].

4.9 Inne

ILSVRC

ImageNet Large Scale Visual Recognition Competition [21] to coroczny konkurs organizowany od 2010 roku, w którym naukowcy walczą o najlepszy wynik w dziedzinie rozpoznawania obrazów przez skonstruowane przez siebie algorytmy.

MNIST

Baza danych MNIST [19] 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 [20] 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

Priorytetem było stworzenie zestawu danych umożliwiającego rozwój różnych modeli sieci neuronowych bez konieczności każdorazowego ingerowania w zbiór lub dopasowywania go specjalnie do konkretnej sieci.
Wszystkie zbiory dostępne są pod linkami w części Dodatek A [8]

5.1 Uzyskiwanie zbiorów ze zdjeć

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, dając odpowiednio 80640 oraz 20160 zdjęć w każdej z nich.

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 zdjeć.
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% powierzchnii.
Kolejną kwestią było przystosowanie sieci do rozpoznawania kości o różnych kolorach ścian,

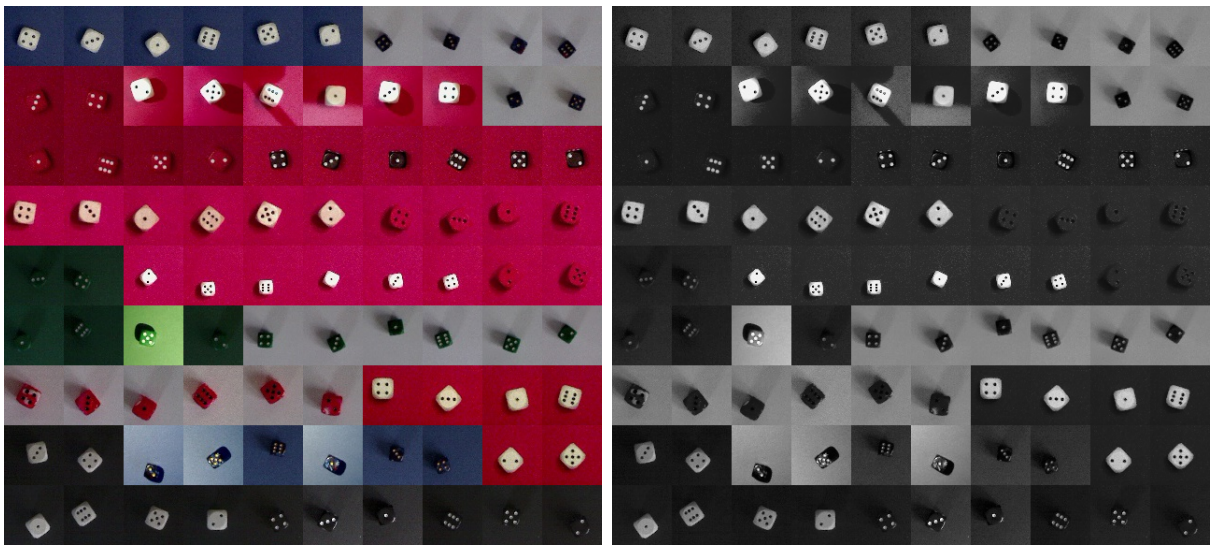
oczek i samego tła. W związku z tym zdjęcia podzielono na zestawy, z których każdy miał ustaloną barwę tła, kości oraz oczek. Każda ścianka sfotografowana była od 3 do 30 razy, dając liczebność każdego z zestaów miedzy 18 a 180 zdjęć. Niektóre zdjęcia poddawane były działaniu ostrego, punktowego światła 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
Łączna ilość zdjęć:			*100*	*600*

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 0,1866% początkowego rozmiaru (stosunek 5,36:1). Następnie poddano je 6 przekształceniom, które rozciągały lub zwężały obrazy o 1-5% z różnych strony. Kolejnym krokiem był obrót o kąt 15^o lub 30^o, zależnie czy dany zbiór wykorzystywany był w mniej lub bardziej rozbudowanych sieciach. Ostatnią operacją było kadrowanie zdjęć do rozmairu 64x64 piksele.

Powyższe działania spowodowały uzyskanie 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 na rysunku 5.1 poniżej:



Rysunek 5.1: 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 jak w obrazie otrzymywanym z kamery. Miało to ułatwić detekcję kości na obrazie, ponieważ obszar 64x64 pikseli był na tyle niewielki, że ciężko w niego trafić podczas rzutu kostką.

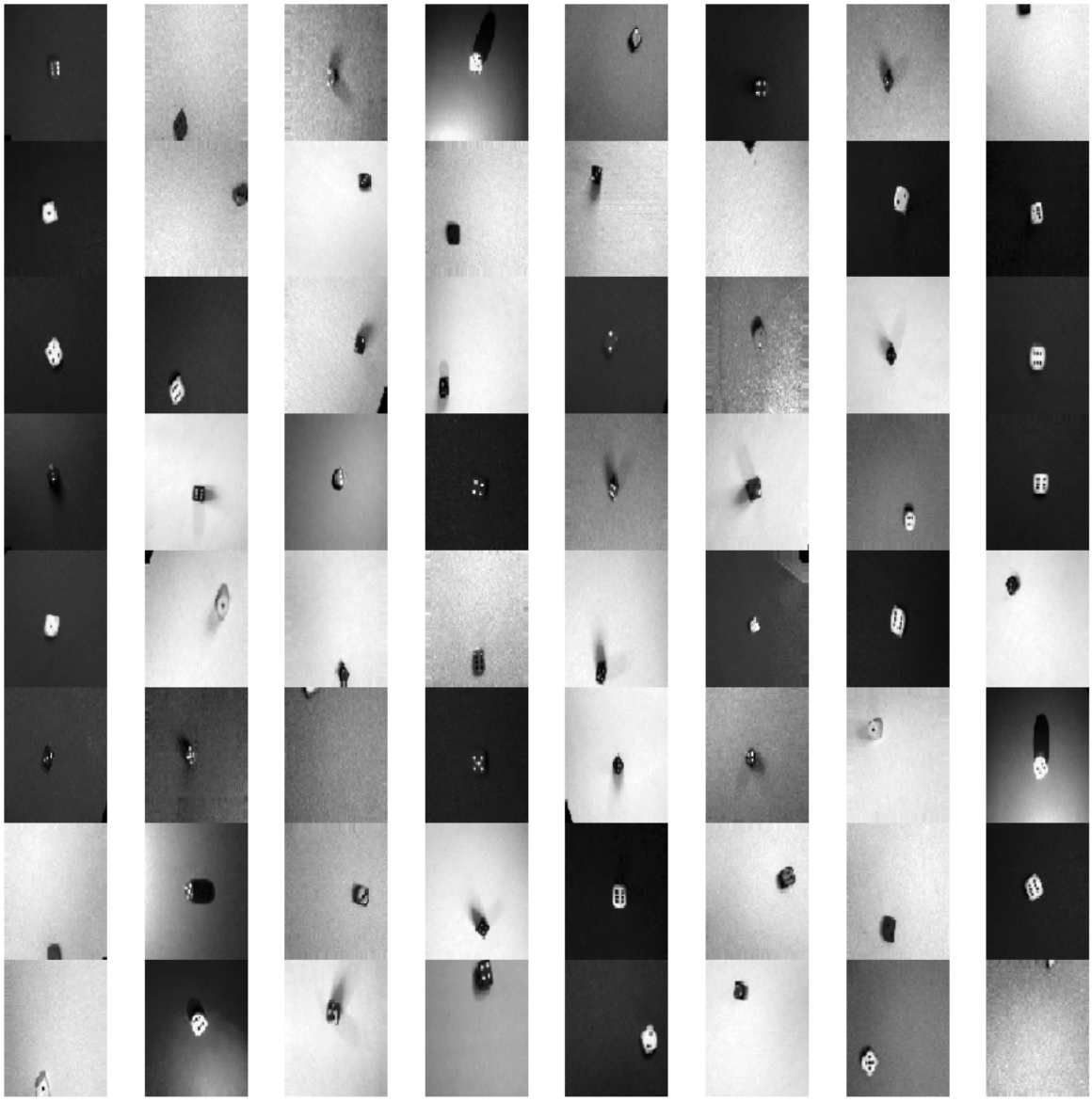
Bardzo istotną rzeczą w tym zbiorze był fakt, że pomimo większego rozmiaru zdjęć, stosunek rozmiaru kości do całego zdjęcia był zdecydowanie mniejszy. W zdjęciach kwadratowych kostka zajmowała 14%, natomiast w prostokątnych jedynie 2% powierzchnii obrazu, a dodatkowo pojedyncze oczko to tylko 0,08% powierzchnii. 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
Łączna ilość zdjęć:			*164*	*984*

Tablica 5.2: Zestawienie kolorystyczne obrazów 2

Tak jak poprzednio, zdjęcia zostały poddane obróbce. Początkowo wszystkie zostały przeskalowane do 0,165% początkowego rozmiaru (stosunek 6:1). Następnie poddano je 4, analogicznym jak poprzednio przekształceniom. Dokonano również rotacji o kąt 30^o i skadrowanie zdjęć do rozmairu 106x79 pikseli.

Uzyskano w ten sposób 60 zdjęć o rozmiarze 106x79 z każdego poczatkowego 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 rysunku 5.2 poniżej



Rysunek 5.2: Obrazy o rozdzielczościach 106x79

6. Sieć rozpoznająca ilość oczek

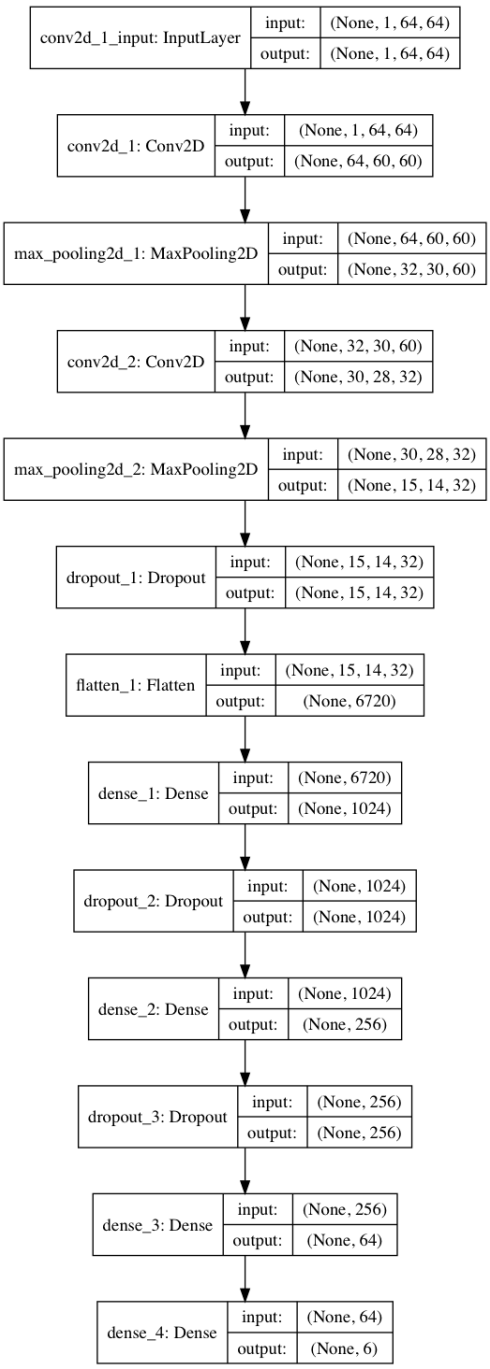
W momencie zrobienia zbioru danych składających się z kwadratowych obrazów, przystąpiono do próby stworzenia i nauczenia modelu sieci neuronowej rozpoznającego ilość oczek wyrzuconych na kostce do gry.
Odnośniki do wszystkich modeli dostępne są pod linkami w części Dodatek B [9]

6.1 Obawy przed rozpoczęciem uczenia

Przed przystąpieniem do treningu sieci pojawiły się wątpliwości w kwestii możliwości jakiegokolwiek rozpoznania ilości oczek przez sieć. Obawy te wiązały się z faktem, że dla przykładu w zbiorze MNIST, położenie cyfr na obrazie było stałe. W przypadku kiedy na obrazach kolor biały był obszarem w kształcie przypominającym pionową kreskę, z dużym prawdopodobieństwem można było założyć, że jest to cyfra 1 lub 7, a analogiczna sytuacja zachodziła dla innych cyfr. W rozpoznawaniu oczek na kostce, zarówno sama kostka mogła być umieszczona w różnych miejscach na obszarze całego obrazu, jak i dopuszczalny był jej obrót o dowolny kąt. Kolejną obawą była niewielka ilość pikseli, które przedstawiały dane oczko w stosunku do rozmiaru całego obrazu. Oznaczało to, że nawet niewielki szum lub zniekształcenia mogą utrudnić prawidłowe rozpoznanie kości.

6.2 Pierwsze eksperymenty

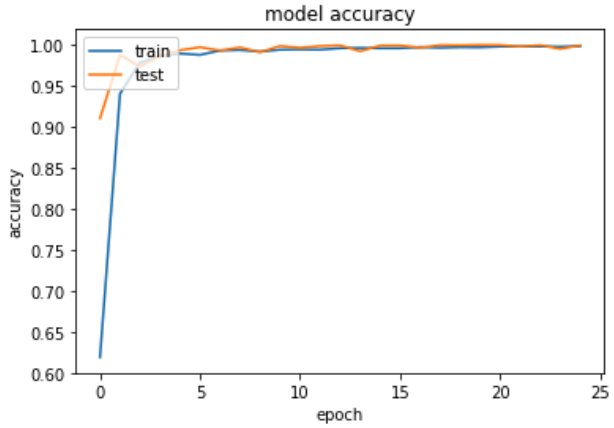
Pierwszy model
Pierwsza próba stworzenia sieci, mając na uwadze wyżej wspomniane wątpliwości, miała na celu wytrenowanie możliwie prostego, nigdy później niewykorzystywanego zbioru obrazów. W tym celu wykorzystano jedynie obrazy z czerwonym tłem, białą kością o czarnych oczkami. Kąt rotacji zmniejszono 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 podzielony na dwie tury, 25 epok w pierwszej oraz 10 epok w drugiej. Nigdzie wcześniej nie zauważono tego typu praktyki, aby rozdzielać epoki uczenia. Zrobiono to dlatego, aby umożliwić wcześniejsze zakończenie całego procesu w sytuacji, gdyby nie zauważono żadnych postępów. Model sieci prezentował się następująco 6.1:



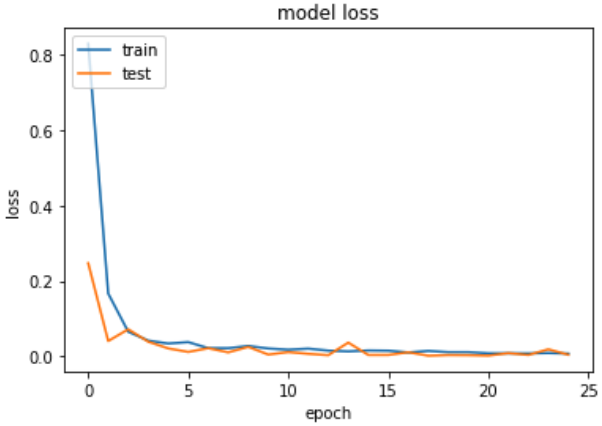
Rysunek 6.1: Pierwszy model sieci

Rezultaty osiągnięte przez ten model były zdumiewające. Po pierwszej epoce sieć uzyskała 61,98% skuteczności ostatecznie osiągając wynik 99,88% po 25 epokach. Kolejne 10 epok praktycznie nie poprawiło już tego rezultatu.

Po analizie wyniku okazało się, że wykorzystanie wielu bardzo podobnych obrazów o identycznym układzie barw, spowodowało występowanie niemal identycznych z testowymi, zdjęć treningowych. W sytuacji, gdyby zdjęcia były bardziej zróżnicowane, 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 6.2 dla sieć w kolejnych epokach:



Skuteczność rozpoznawania kości



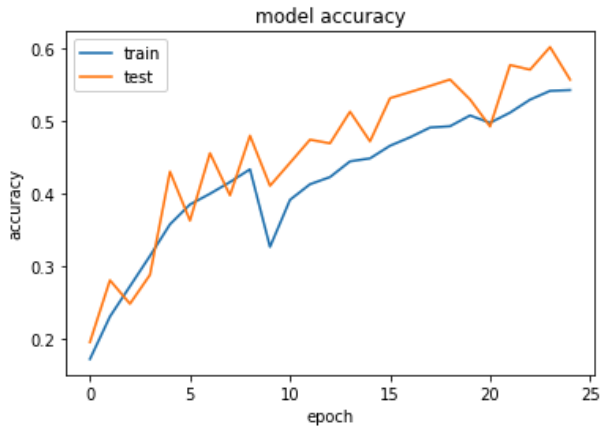
Wartość funkcji błędu

Rysunek 6.2: Wykresy dla pierwszego modelu

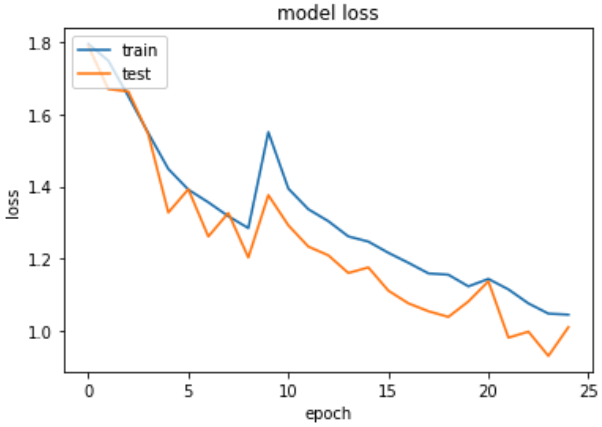
Model ze zróżnicowanymi zdjęciami

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 pierwszego modelu. Poniżej znajdują się wykresy 6.3 uczenia się modelu:



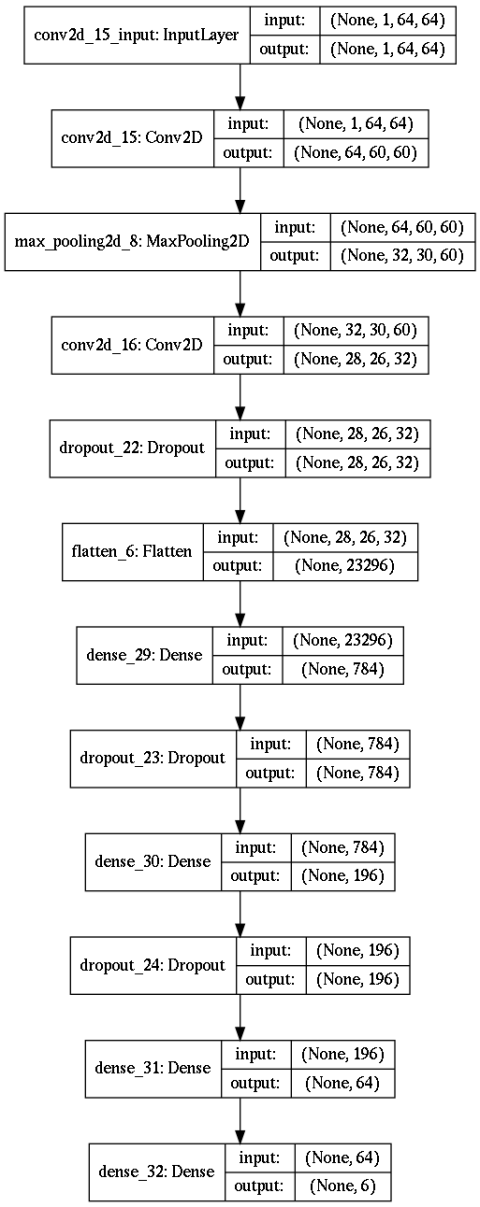
Skuteczność rozpoznawania kości



Wartość funkcji błędu

Rysunek 6.3: Wykresy dla modelu z optymalizatorem Adam

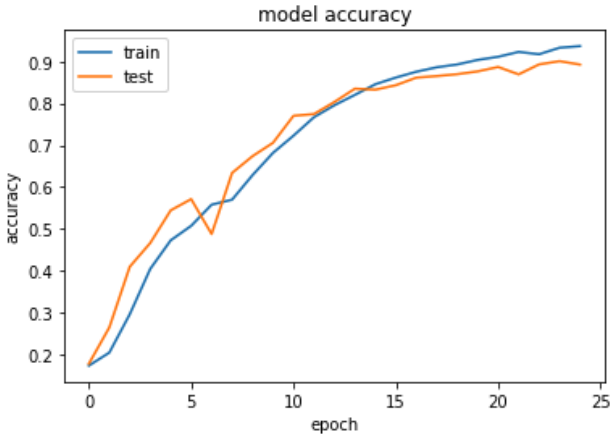
Tak jak poprzednio sieć uczona była przez 25, a następnie przez 10 epok. Po pierwszych 25 epokach uzyskano dokładność 55,64% co potwierdziło przypuszczenie z poprzedniej sieci o przetrenowaniu i konieczności zróżnicowania danych. Kolejne 10 epok poprawiło wynik sieci do 65,30%. Poniżej znajduje się opisywany model 6.4:



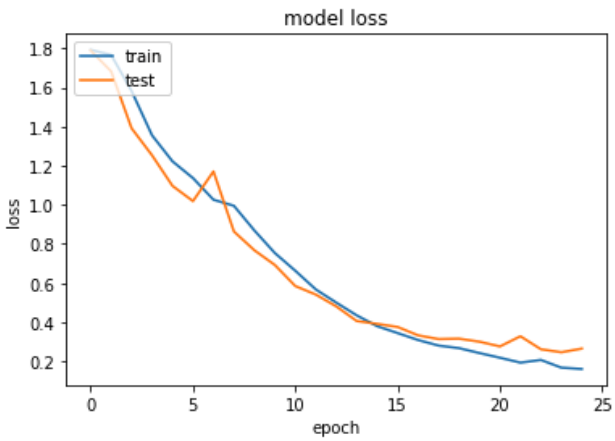
Rysunek 6.4: Model bazujący na optymalizatorze Adam

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 przykładach, chociaż pominięto tu już, tak jak w każdym następnym przykładzie, uczenie z podziałem na dwie tury. 25 epok wystarczyło by uzyskać dokładność na poziomie 89,23% co jest rezultatem zdecydowanie lepszym niż uzyskane wcześniej 55,64%. Przebieg uczenia sieci widoczny jest poniżej 6.5:



Skuteczność rozpoznawania kości



Wartość funkcji błędu

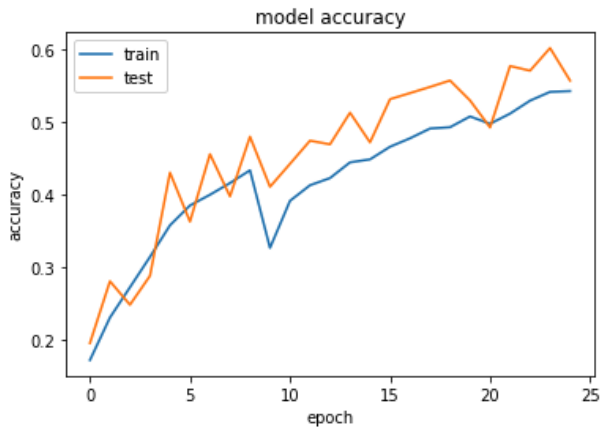
Rysunek 6.5: Wykresy dla modelu korzystającego z wybranych zestawów kolorystycznych

6.3 Analiza różnych optymalizatorów

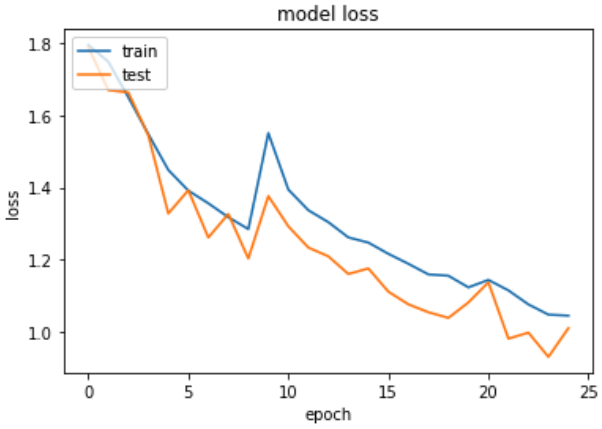
Następnym krokiem analizy sieci neuronowych było przetestowaniu kilku z dostępnych w bibliotece Keras optymalizatorów dla identycznych sieci. Zbiorem danych był opisany w poprzednim rozdziale zbiór złożony z 100800 kwadratowych obrazów. W tym celu postanowiono użyć optymalizatorów RMSprop oraz SGD.

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 6.6 6.7 6.8:

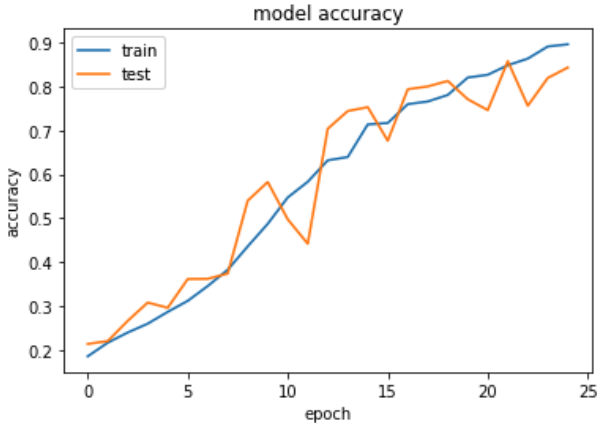


Skuteczność rozpoznawania kości

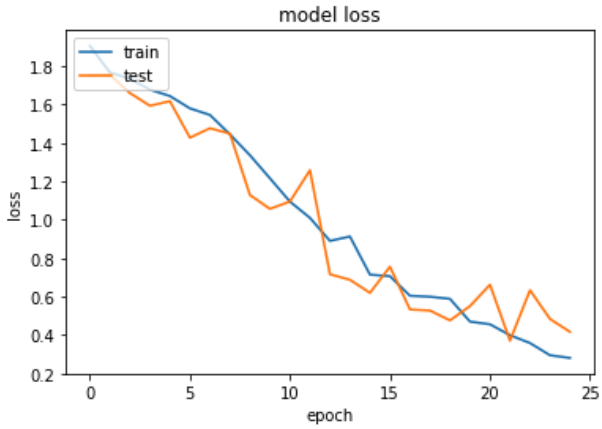


Wartość funkcji błędu

Rysunek 6.6: Wykresy dla modelu korzystającego z optymalizatora Adam

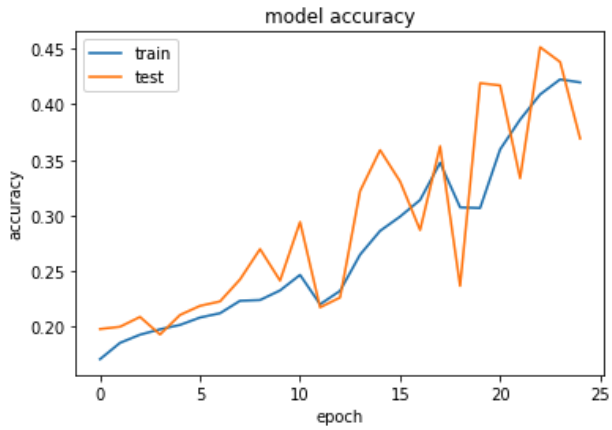


Skuteczność rozpoznawania kości

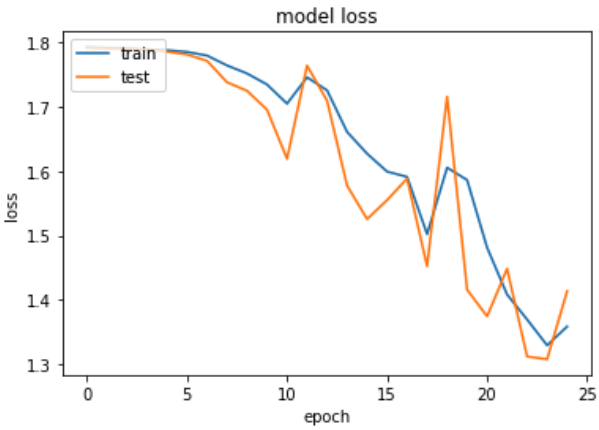


Wartość funkcji błędu

Rysunek 6.7: Wykresy dla modelu korzystającego z optymalizatora RMSprop



Skuteczność rozpoznawania kości

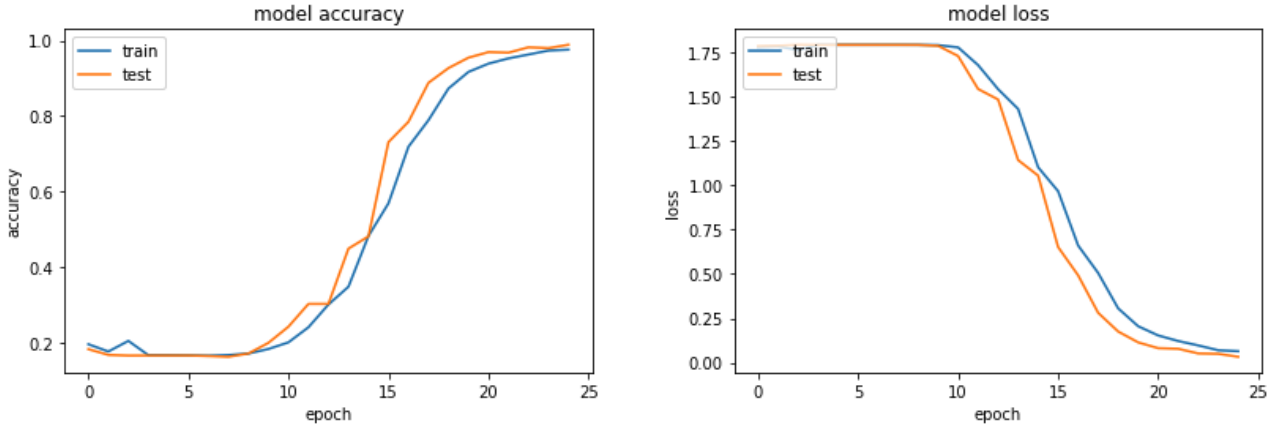


Wartość funkcji błędu

Rysunek 6.8: Wykresy dla modelu korzystającego z optymalizatora SGD

6.4 Wykorzystanie sieci AlexNet

Sieć o nazwie AlexNet [23, 22, 24] została zaprezentowana przez Alexa Krizhevskyego, Geoffreya Hintona 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 i swoimi rezultatami zszokowała wielu naukowców. Z racji tak świetnych rekomendacji, zdecydowano się na realizację modelu przypominającego jej budowę.
Zastosowano uproszczoną architekturę, bazującą na idei sieci AlexNet, na zbiorze 100800 kwadratowych obrazów. Po 25 epokach pozwoliło to na osiągnięcie poprawności wynoszącej aż 98.88%. Na wykresach uczenia 6.9 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.



Skuteczność rozpoznawania kości

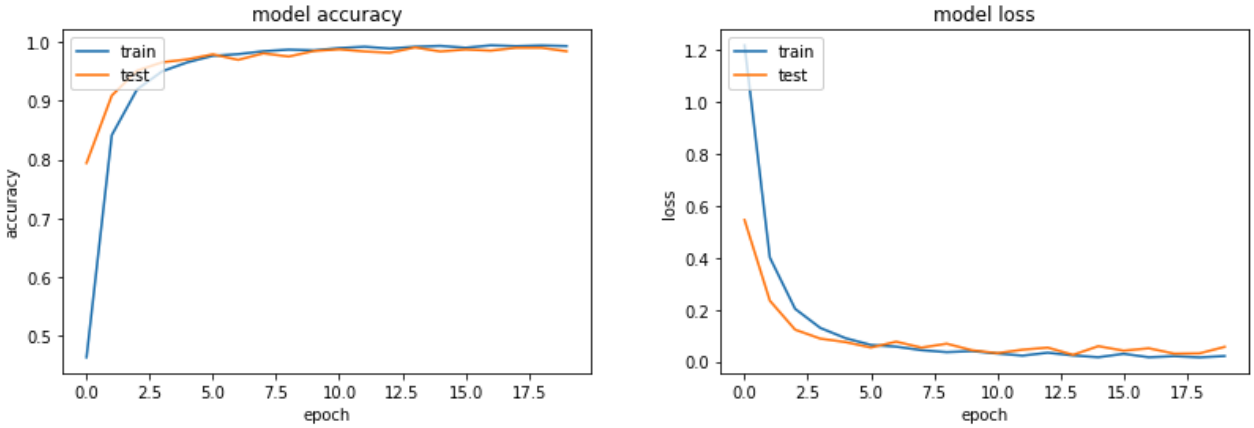
Wartość funkcji błędu

Rysunek 6.9: Wykresy dla sieci opartej o AlexNet

6.5 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

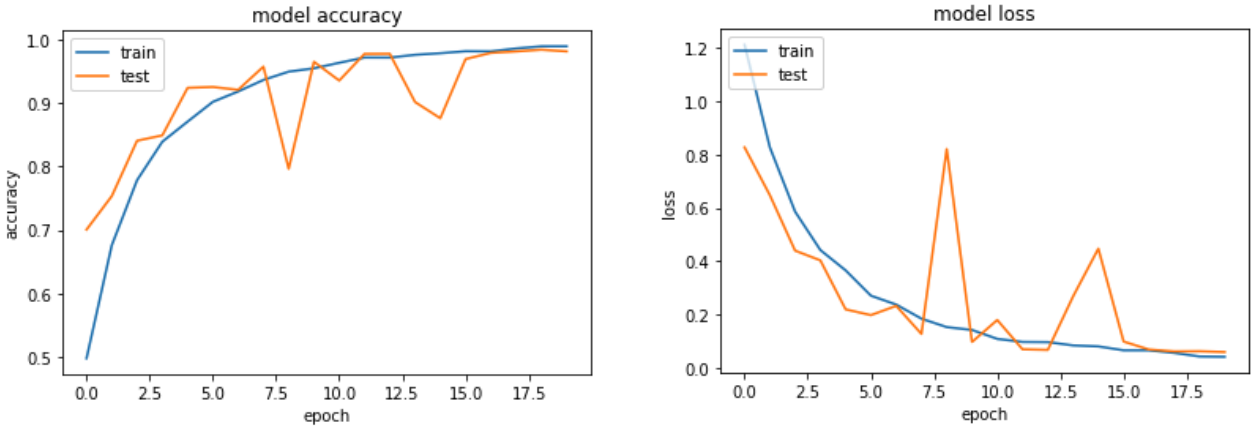
kolorowej wersji zbioru obrazów kwadratowych, dla których zastosowano kąt obrotu 30°, uzyskując 50400 obrazów. Sieć korzystająca z czarno-białych obrazów miała te same zdjęcia z jednym zamiast trzech kanałów informujących o kolorze. Do tego eksperymentu wykorzystano narzędzie generatora dostępne w Keras, które przekazuje obrazy do sieci bez konieczności wcześniejszego ładowania całego zbioru do pamięci. Architektura obu sieci była uproszczona, by zniwelować długi czas uczenia. Pierwsze wykresy 6.10 przedstawiają proces uczenia na obrazach kolorowych:



Skuteczność rozpoznawania kości Wartość funkcji błędu

Rysunek 6.10: Wykresy dla sieci z obrazami kolorowymi

Wykresy 6.11 przedstawiają proces uczenia na obrazach w odcieniach skali szarości:



Skuteczność rozpoznawania kości Wartość funkcji błędu

Rysunek 6.11: Wykresy dla sieci z obrazami w skali szarości

Jak widać na wykresach, oba modele po 20 epokach wykazały się bardzo zbliżonymi wynikami 98,47% oraz 98,12% na korzyść sieci z kolorowymi obrazami. Lepiej radzący sobie model, dodatkowo nauczył się bardzo szybko do poziomu 95%, osiągając go już po 4 epoce, gdzie operujący na obrazach w skali szarości osiągnął ten wynik po 10 epokach. Obserwacja ta może sugerować, że większa różnica w wartościach dla kolorowych obrazów może przyśpieszać uczenie, ale wymaga większej ilości pamięci i spowalnia każdą epokę o 15%.

6.6 Próby wykorzystania prostokątnych obrazów

Doświadczenia zakończone niepowodzeniem

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 pierwsze rodzaju obrazów.

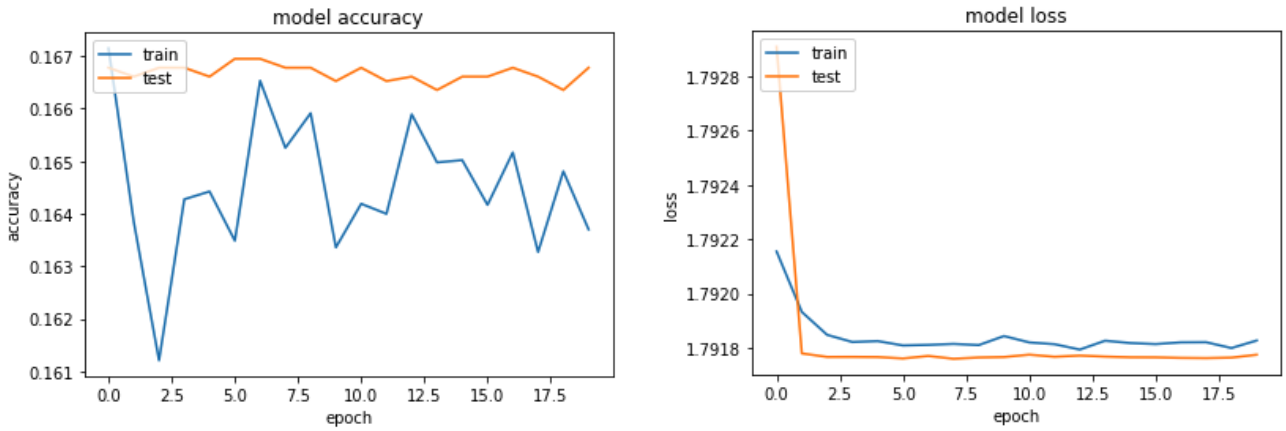
Po uzyskaniu wielu bardzo dobrych wyników powyżej 90% na obrazach o rozmiarach 64x64 przystąpiono do prób znacznego zwiększenia obrazów prostokątnych.

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. 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 zbiorów przez ograniczenie rozmiarów obrazów. Problemem podczas zmniejszania 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 6.12 uczenia się przedstawione sa poniżej:

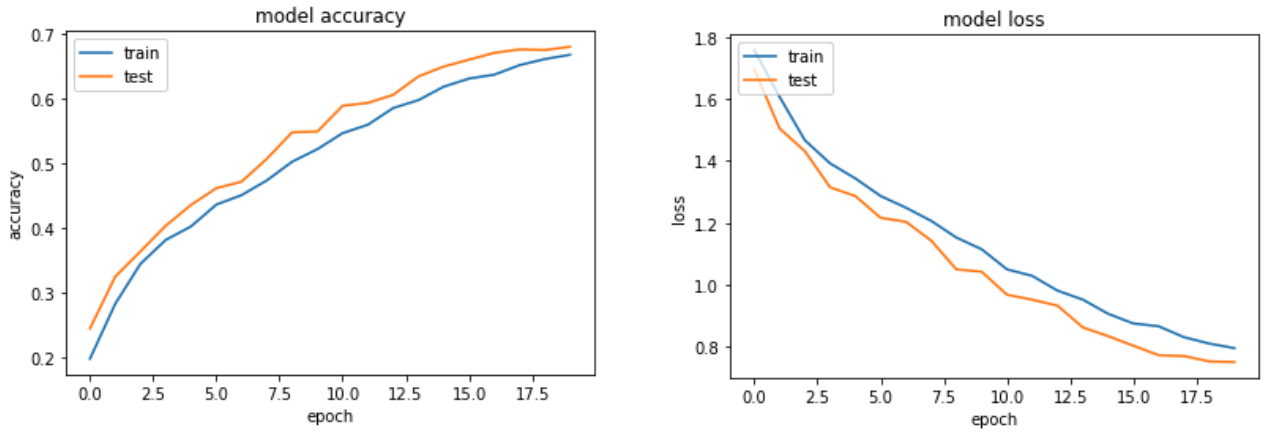


Skuteczność rozpoznawania kości Wartość funkcji błędu

Rysunek 6.12: 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. Pierwsza próba przeprowadzona przez 20 epok z filtrami konwolucjnymi 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 6.13 uczenia znajduje się poniżej.



Skuteczność rozpoznawania kości Wartość funkcji błędu

Rysunek 6.13: Wykresy dla nauczonej sieci z obrazami 106x79

6.7 Doskonalenie modelu z prostokątnymi obrazami

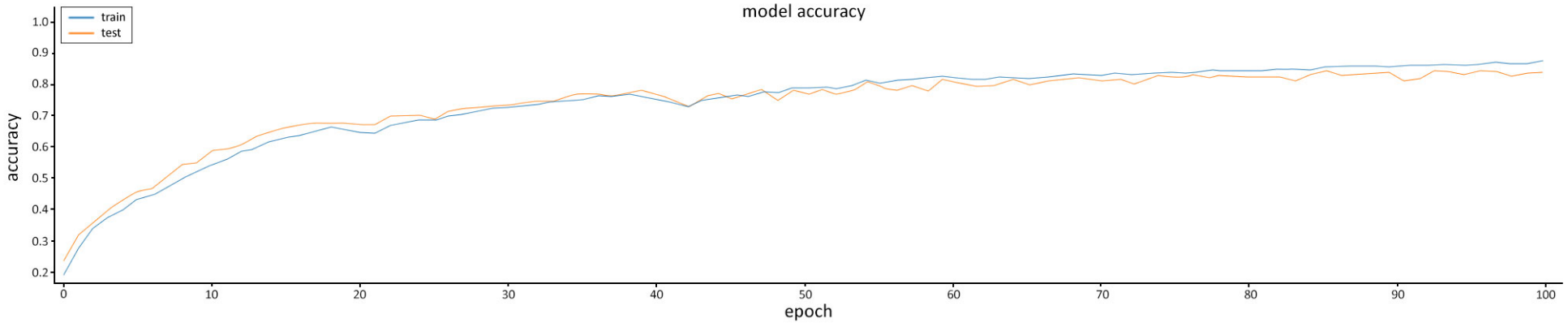
Po pierwszym sukcesie sieci z obrazami prostokątnymi, podjęto decyzję o jej ulepszeniu. Zaczęto od próby zmniejszenia ilości parametrów przez zamianę większych filtrów konwolucyjnych, większą ilością mniejszych, co znacząco zmniejszało ilość parametrów sieci potrzebną do nauczenia [7]. Po zaaplikowaniu ulepszeń i nauce sieci okazało się, że sieć nie poprawiła się w żadnym stopniu. Prawdopodobnym powodem była zmiana architektury sieci wraz z powtórnym zastosowaniem kwadratowych filtrów.
Drugim pomysłem na ulepszenie sieci było zastosowanie zastępowania większych filtrów kilkoma mniejszymi połączonego ze zamianą funkcji aktywacji ReLU na LeakyReLU w celu uniknięcia możliwych do wystąpienia problemów z zanikającym neuronem. Ulepszenie jednak podobnie jak wcześniejsze nie sprawdziło się w ogóle i nie umożliwiło poprawy dokładności sieci.

6.8 Najskuteczniejszy model z prostokątnymi obrazami

Wykres procesu uczenia sieci z obrazami 106x79 kształtem przypominał funkcję logarytmiczną, co nasunęło pomysł ze zwiększeniem ilości epok. Podjęto decyzję o kontynuacji uczenia do kolejno 40, 60, 80 i 100 epok.
Podejście to okazało się bardzo skuteczne, ponieważ po każdych 20 epokach sieć odnosiła 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 jednak z ogromnym nakładem czasu, ponieważ prawdopodobnie sensowne rezultaty można by osiągnąć dopiero po 100 epokach. Wykres 6.14 dokładności sieci znajduje się na następnej stronie.



Rysunek 6.14: Wykres uczenia przez 100 epok

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 dostarczany z kamery jest wielokrotnie większy, co wymusiło wyznaczenie jego części, która jest analizowana przez zaaplikowane sieci neuronowe. Ta część obrazu oznaczona jest białymi krawędziami dla łatwiejszego zlokalizowania miejsca, w którym powinna być umieszczona kość pod kamerą. 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. Często 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] Stanford University course CS231n
http://cs231n.github.io/optimization-2/

[3] Stanford University course CS231n
http://cs231n.github.io/neural-networks-1/

[4] Eric Roberts, Stanford University
https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/

[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] Leonardo Araujo dos Santos
https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/backpropagation.html

[9] Anish Singh Walia
https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f

[10] Ujjesl Karn
https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

[11] Eugenio Culurciello
https://towardsdatascience.com/neural-network-architectures-156e5bad51ba

[12] Avinash Sharma V
https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

[13] Sagar Sharma
https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

[14] Aditya V. D
https://becominghuman.ai/neural-network-xor-application-and-fundamentals-6b1d539941ed

[15] https://en.wikipedia.org/wiki/Feedforward_neural_network

[16] https://en.wikipedia.org/wiki/Convolutional_neural_network

[17] https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

[18] https://en.wikipedia.org/wiki/Computer_vision#Recognition

[19] Yann LeCun, Corinna Cortes, Christoper J.C. Burges
http://yann.lecun.com/exdb/mnist/

[20] Alex Krizhevsky
https://www.cs.toronto.edu/~kriz/cifar.html

[21] ImageNet Large Scale Visual Recognition Competition
http://www.image-net.org/challenges/LSVRC/

[22] Adit Deshpande
https://adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html

[23] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton
https://www.nvidia.cn/content/tesla/pdf/machine-learning/imagenet-classification-with-deep-convolutional-nn.pdf

[24] Tugce Tasci, Kyunghee Kim
http://vision.stanford.edu/teaching/cs231b_spring1415/slides/alexnet_tugce_kyunghee.pdf

[25] Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, Rob Fergus
https://cs.nyu.edu/~wanli/dropc/

[26] Jason Morrison
https://medium.com/paper-club/paper-review-dropout-a-simple-way-to-prevent-neural-networks-from-overfitting-4f25e8f2283a

[27] Matthew D. Zeiler, Rob Fergus
https://arxiv.org/abs/1311.2901

[28] Sebatian Ruder
http://ruder.io/optimizing-gradient-descent/

[29] Diederik P.Kingma, Jimmy Ba
https://arxiv.org/abs/1412.6980

[30] Krzysztof Sopyła
https://ksopyla.com/python/operacja-splotu-przetwarzanie-obrazow/

[31] 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/

8. Dodatek A - zbiory danych

Zbiory danych wykorzystane do uczenia przedstawionych modeli sieci neuronowych znajdują się w linkach dostępnych poniżej:

Zbiór 64x64x1, obrazy w skali szarości, opisany w częsci 5.1
`https://drive.google.com/open?id=1-qCVGTql6A3QHLxS0aQvhFxSz5wUY7uG`

Zbiór 64x64x3, obrazy kolorowe, opisany w częsci 5.1
`https://drive.google.com/open?id=1oWa8t5pCWCGyC3Mbdn_XJ2hVdf0yqrMu`

Zbiór 106x79x1, obrazy w skali szarości, opisany w części 5.2
`https://drive.google.com/open?id=1JS_C_pJqU2TmBEVvMTULTXue7VZJKONH`

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` OptymalizatorSGD:
`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 protokątnymi 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 redukcje 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 jedynego nauczonego modelu 20-60 epok

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

Kontynuacja uczenia jedynego 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