

**WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ**

Katedra Informatyki i Automatyki

Piotr Maślanka

Uczenie głębokie - podstawy
teoretyczne i implementacja
programowa

Praca dyplomowa magisterska

Opiekun pracy:
prof. dr hab. inż. Jacek Kluska

Rzeszów 2016

Spis treści

1. Wstęp	4
2. Uczenie głębokie	9
2.1. Przykłady uczenia głębokiego w przemyśle	10
2.2. Obecny stan bibliotek uczenia głębokiego	11
2.3. Rola GPU w uczeniu głębokim	13
2.4. Wybrane architektury uczenia głębokiego	14
2.4.1. Głębokie sieci neuronowe	14
2.4.2. Splotowe sieci neuronowe	15
2.5. Przegląd komponentów uczenia głębokiego	15
2.5.1. Regularyzacja typu dropout	16
2.5.2. Funkcja aktywacji ReLU	18
2.5.3. Warstwa splotowa	19
2.5.4. Max-pooling	21
2.6. Zastosowane narzędzia obliczeniowe	22
2.6.1. Python	22
2.6.2. Theano	23
2.6.3. nnetsys	23
2.7. Zbiory danych	24
2.7.1. MNIST	24
2.7.2. CIFAR-10	25
3. Eksperymenty	26
3.1. Test stabilności numerycznej algorytmu	26
3.2. Opis biblioteki	27
3.3. Kwestia inicjalizacji wag	32
3.4. Wpływ współczynnika uczenia	34
3.5. Rozmiar wsadu, momentum - sieci głębokie a klasyczne	37
3.6. Porównanie jakości sieci	40
3.6.1. MNIST	41
3.6.2. CIFAR-10	43

3.7. ReLU i tanh - głębia	44
3.8. Architektura sieci splotowych	47
3.9. Sztuczny resampling danych treningowych	49
3.10 Porównanie sieci splotowej i dużej sieci klasycznej	50
3.11 Czego uczy się sieć splotowa	54
4. Zastosowanie istniejących bibliotek uczenia głębokiego	59
4.1. Instalacja TensorFlow	59
4.2. Głębokie sieci neuronowe	60
4.3. Sieci splotowe	64
5. Konfiguracja środowiska Python i Theano	72
5.1. Instalacja środowiska	72
5.1.1. Instalacja dla GPU	73
5.1.2. Instalacja kompilatora C i Theano	73
5.2. Uruchamianie eksperymentów	74
5.2.1. test	75
5.2.2. dnn_mnist	76
5.2.3. cnn_cifar	76
6. Podsumowanie	77
Spis rysunków	79
Spis tabel	80
Bibliografia	81
Dodatek	84

1. Wstęp

Technologie uczenia maszynowego są wykorzystywane w wielu dziedzinach współczesnego życia, m.in. w systemach wspomagania decyzji w technice, medycynie (głównie w diagnostyce), wojsku (w systemach rozpoznawania czy wczesnego ostrzegania), biznesie (do prognozowania zachowań klientów czy w transakcjach bankowych), wyszukiwarkach internetowych, do projektowania zaawansowanych aplikacji dla smartfonów i do wielu innych zastosowań. Klasyczne systemy uczenia maszynowego mają jednak liczne ograniczenia. Zwykle ich zastosowanie wymaga zaangażowania i uważnej pracy wielu ekspertów dziedzinowych, którzy powinni wskazać w jaki sposób zredukować dane w sensie liczby rekordów i liczby cech (atrybutów), np. jak uzyskać specyficzne cechy charakteryzujące obraz złożony z pikseli. Wymagana jest więc analiza danych wejściowych, często nawet z użyciem metod heurystycznych, aby na tej podstawie system realizujący algorytm (czy zespół algorytmów) uczenia, był w stanie wygenerować interesujące nas konkluzje. Systemy klasyczne nie są w stanie poprawnie rozpoznawać bardzo skomplikowanych wzorców, które zwykle mają olbrzymie rozmiary, co więcej napływają one masowo w czasie rzeczywistym. Od współczesnych systemów wymaga się poprawnego funkcjonowania w środowisku zmieniającym się w czasie, co jest dodatkowym, niezwykłym wyzwaniem dla informatyków i badaczy z wielu różnych dyscyplin.

Rozwiązaniu problemu w pewnym stopniu przyczynił się wzrost możliwości obliczeniowych komputerów. Systemy obliczeń GPU, takie jak CUDA, pozwalają na prowadzenie obliczeń na kartach graficznych. Okazuje się, że są one bardzo dobrze przystosowane do obliczeń prowadzonych na potrzeby mechanizmów uczenia maszynowego, zwłaszcza obliczeń na macierzach, które pozwalają uczyć duże sieci na relatywnie tanim sprzęcie, co sprzyja popularyzacji tej gałęzi nauki.

Przeszkoda w postaci konieczności wyboru specjalnych cech przez ekspertów w sieciach głębokich nie występuje. Ideą “głębokości” takiej sieci

jest zdolność do automatycznego wyciągnięcia ważnych cech z zbioru danych. Sieć taka składa się z wielu warstw (tj. jest głęboka), a każda warstwa przekształca dane w ich reprezentację na wyższym poziomie abstrakcji. Dostateczna ilość takich przekształceń pozwala sieci na nauczenie się bardzo złożonych funkcji. Kluczową różnicą jest to, że warstwa przekształceń cech nie jest projektowana przez odpowiedniego eksperta; powstaje ona samoczynnie poprzez ekspozycję sieci na dane [26].

W 2006 zainteresowanie sieciami głębokimi odżyło dzięki grupie badaczy z Canadian Institute for Advanced Research (CIFAR). Postanowili oni wstępnie nauczyć sieć nieoznaczonymi danymi (tj. wykorzystywania uczenia nienadzorowanego). Pozwalało to później uczyć sieć danymi oznaczonymi i uzyskiwać lepsze wyniki, gdyż sieć była w stanie wcześniej odkryć strukturę w istniejących danych. Później usprawniono to podejście, a obecnie systemy uczenia głębokiego zazwyczaj nie wymagają wstępnego uczenia. Na specjalne miejsce zasłużyła sobie pewna klasa sieci głębokich niewymagającego wstępnego uczenia. Sieci splotowe uzyskały wiele praktycznych sukcesów w zakresie rozpoznawania dźwięków, obrazów i wideo, dzięki czemu na stałe zdominowały się w wizji komputerowej. Obecnie wiodące korporacje IT prowadzą szeroko zakrojone badania nad uczeniem głębokim. Powstał szereg bibliotek, umożliwiających pracę nad systemami uczenia głębokiego. W niektórych przypadkach to zadaniem programisty jest podanie sposobu przeprowadzenia operacji w postaci grafu działań oraz zdefiniowania danych i parametrów (np. macierzy wag sieci). Systemy te są w stanie samoczynnie liczyć pochodne z zadanych operacji, dzięki czemu sieci te mogą się uczyć. Kompilatory tych bibliotek, przetwarzając taki graf generują kod wykonywalny na CPU lub GPU, w zależności od dostępnego sprzętu obliczeniowego. Takie biblioteki poprawniej technicznie byłoby nazwać kompilatorami wyrażeń symbolicznych, w odróżnieniu od innych bibliotek, gdzie programista może zażyczyć sobie wykorzystania konkretnych warstw obliczeniowych (np. zdefiniowanie warstwy perceptronu, zamiast *explicite* definiowania macierzy wag i przesunięć oraz mnożenia, dodawania i aktywacji). Znakomita część tych bibliotek jest typu

open-source, dostępne są do nich również kursy i przykłady.

Sposoby uczenia głębokiego siłą rzeczy muszą różnić się od klasycznych sieci neuronowych. Mimo powierzchownego podobieństwa, różnice sięgają dość głęboko, a osoba która zamierza podjąć się zadania opanowania tej technologii musi zdawać sobie sprawę z tego, że działają one nieco inaczej. **Celem pracy jest** zbadanie podstawowych różnic między uczeniem klasycznym, a wybranymi architekturami uczenia głębokiego. Architekturami tymi są głębokie sieci neuronowe (DNN, ang. deep neural network) oraz sieci splotowe (CNN, ang. convolutional neural network). Wymaga to przeprowadzenia szeregu eksperymentów, sprawdzających drobne elementy konstrukcyjne, takie jak np. dobór hiperparametrów. Same eksperymenty wymagają również zbudowania odpowiednich architektur, co wymaga wykorzystania odpowiednich bibliotek oraz zbudowania programów. Programem takim będzie, między innymi, zbudowana na potrzeby tych eksperymentów autorska biblioteka uczenia głębokiego *nnet-sys*, wykorzystująca kompilator wyrażeń symbolicznych Theano. Oprócz tej biblioteki zamierzono przedstawić również budowę sieci głębokich przy użyciu biblioteki TensorFlow, pierwotnie autorstwa Google. Spróbowano podjąć się również częściowo przedstawienia sposobu działania sieci splotowej.

Przeprowadzenie eksperymentu ma na celu sporządzenie szeregu wskazówek, które mogłyby być przydatne dla osoby zamierzającej podjąć się pracy z systemami uczenia głębokiego. Zwłaszcza nacisk zostanie położony na elementy, które są w porównaniu do systemów uczenia klasycznego różne.

W ramach pracy wykonano następujące zadania częściowe:

- wykonano bibliotekę uczenia maszynowego w oparciu o kompilator wyrażeń symbolicznych Theano, wspierającą głębokie sieci neuronowe (DNN) oraz sieci splotowe (CNN) wraz z algorytmami wspierającymi
- zaplanowanie, stworzenie skryptów przeprowadzających odpowiednie eksperymentów związanych z celem pracy, uruchomienie ich,

zebranie i opracowanie wyników

- wykonano wprowadzenia do budowy architektur głębokich w bibliotece TensorFlow

Omawiane w pracy zagadnienia podzielono na trzy części, z czego pierwsza dotyczy ogólnego opisu systemów oraz wspierających je algorytmów, druga i trzecia stanowią opisu eksperymentów i demonstracji pracy w środowisku TensorFlow.

W rozdziale 2 opisano zasadniczą różnicę stanowiącą o uczeniu głębokim. Wspomniano o kilku zastosowaniach tych algorytmów w przemyśle, oraz wspomniano o istniejących bibliotekach uczenia głębokiego. Opisano również wybrane architektury uczenia głębokiego, wraz z kluczowymi algorytmami wspierającymi. Zaznaczono również zasadnicze różnice od klasycznych sieci neuronowych. Na końcu opisano środowisko obliczeniowe, które posłużyło do przeprowadzenia eksperymentów.

W rozdziale 3 przedstawiono przeprowadzone eksperymenty, ich warunki początkowe, wyniki oraz wnioski. Eksperymenty przeprowadzono w oparciu o graficzne zbiory danych. Na początku uzasadniono celowość prowadzenia obliczeń na GPU. Później przeprowadzono szereg eksperymentów mających porównać dobrą niektórych hiperparametrów, takich jak współczynnik uczenia czy wartość momentum w uczeniu głębokich sieci neuronowych (DNN), a sieci klasycznych. Kolejnym etapem byłoby przeprowadzenie porównania uczenia klasycznego i sieci splotowych (CNN), kończące się porównaniem trzech rodzajów systemów (CNN, DNN, sieci klasyczne) na relatywnie trudnym zbiorze danych CIFAR-10. Rozdział kończy uproszczona analiza sposobu działania sieci splotowej i graficzna interpretacja jej wag.

W rozdziale 4 zamieszczono wprowadzenie do konstruowania sieci uczenia głębokiego oraz sieci splotowych przy użyciu biblioteki TensorFlow. Przedstawiono instrukcję instalacji biblioteki na systemie Linux. Wyjaśniono co generują odpowiednie komendy, przedstawiono sposób pracy TensorFlow, oraz praktycznie przedstawiono zasady określania rozmiaru filtrów w sieciach splotowych.

W rozdziale 5 zamieszczono instrukcję konfiguracji środowiska Theano oraz Python do współpracy z zarówno GPU jak i CPU. Dodatkowo umieszczone tam instrukcję uruchomienia załączonych eksperymentów.

2. Uczenie głębokie

Ostatnimi laty uczenie głębokie zdobywa sobie coraz większą popularność w środowisku uczenia maszynowego. Pozwala ono przetwarzać dużo bardziej skomplikowane zbiory danych, niż klasyczne uczenie maszynowe, ze względu na podstawową różnicę.

Klasyczne uczenie maszynowe wymaga wstępnego przetworzenia danych. Spośród znacznej ilości danych wejściowych należy wybrać jedynie te istotne. Wymaga to pracy eksperta domenowego, który wybierze (lub powie jak wyliczyć) wzorce (ang. *features*) które będą dobrą dyskryminantą klas. Przykładem takich wzorców mogą być momenty Hu [20] w analizie obrazów. Dzięki takiemu podejściu sieci, które inaczej musiałyby radzić sobie z bardzo dużymi wektorami wejściowymi (obraz 1024x768 ma ok. 786 tys. wymiarów), mogą analizować problem na mniejszej liczbie bardzo specyficznych wartości.

Podejście takie staje się jednak problemem, gdy nie wiadomo, w jaki sposób wyznaczyć charakterystyczne wartości. Jednym z rozwiązań tego problemu jest analiza głównych składowych (PCA). Są jednak sytuacje kiedy PCA z pewnych względów (np. struktura danych) nie może być zastosowane. Nasuwa się więc pomysł, aby sieć sama nauczyła się tych wzorców. Będzie to wymagało architektury z większą ilością warstw (stąd uczenie **głębokie**), jednak w wykonaniu okazuje się niezwykle skuteczne. Sieć sama jest w stanie nauczyć się charakterystycznych cech. Z tego też powodu wydaje się, że porównywanie rozwiązań klasycznych, takich jak perceptron wielowarstwowy czy SVM z sieciami głębokimi na danych gdzie nie są dostępne wektory źródłowe, a jedynie już przetworzone w postaci wektorów cech, wydaje się mijać z celem.

Cechą wspólną systemów uczenia głębokiego jest wykorzystanie wielu warstw przekształceń nieliniowych, gdzie każda warstwa jako wejście przyjmuje wyjście warstwy poprzedzającej. Systemy takie formułują więc hierarchiczną reprezentację danych wejściowych, gdzie wzorce niższych warstw są rozpoznawane przez neurony w niższych warstwach, a wzorce

wysokopoziomowe w warstwach bliższych warstwie wyjściowej. Dzięki takiemu podejściu sieć jest w stanie skutecznie uczyć się istotnych wzorców, pomocnych w klasyfikacji. Mogą być one trenowane w uczeniu nadzorowanym (np. DNN) bądź nienadzorowanym (np. DBN, autoenkodery).

2.1. Przykłady uczenia głębokiego w przemyśle

Uczenie głębokie pozostaje ciągle technologią opracowywaną, a kwestia jej stosowalności w przemyśle jest ciężką do prześledzenia. Korporacje zajmujące się tą tematyką są niechętne do publikowania artykułów o swoich zastosowaniach uczenia głębokiego. Niemniej jednak, dostępnych jest na tyle danych aby krótko opisać temat obecnego zastosowania tej technologii w przemyśle.

Istotnym problemem z którym zmagała się firma Google przy budowie systemu StreetView było odczytywanie z tablic numerów domów. Ze względu na niską jakość zdjęć i różnorodność tablic problem ten był trudny do zrealizowania konwencjonalnymi algorytmami uczenia maszynowego. Przez pewien czas Google korzystało z systemu reCAPTCHA [17] do digitalizowania numerów domów. System reCAPTCHA służy do odróżnienia ludzi od botów na stronach internetowych w takich sytuacjach jak rejestrowanie nowego użytkownika czy publikowanie komentarzy publicznych w portalach. Dzięki temu numery budynków były tak naprawdę rozpoznawane przez ludzi.

W 2013 Google z powodzeniem zastosowało splotową sieć neuronową do rozpoznawania numerów budynków z tablic z dokładnością 96% [13]. System ten na testach reCAPTCHA uzyskiwał dokładność 99,8%. Skuteczność tego systemu była równa skuteczności ludzkiej, w niektórych warunkach nawet ją przekraczając.

Innym zastosowaniem sieci głębokich do przetwarzania obrazów była poprawa tagowania obrazów w usłudze Google Image Search [16]. Mimo że technologia istniała już dłuższy czas, prawdziwym przełomem który sprawił że poprawione wyszukiwanie zdjęć jest możliwe był wzrost możliwości obliczeniowych. System ten popełnia racjonalne błędy (np. pomylenie psa

z kozą, w kontrakście do pomylenia psa z drzewem), ma dobrą dokładność w obrazach specyficznych domenowo (np. gatunki kwiatów), czy obsługuje koncepty mające wiele modalności (np. pojęcie **samochód** odnosi się do zdjęcia samochodu z zewnątrz, jak i wnętrza pojazdu).

Dodatkowo przetwarzanie głosu za pomocą uczenia głębokiego pozwoliło Google wyprzedzić funkcjonalnością analogicznie rozwiązanie Apple Siri [19]. O ile rozpoznawanie głosu realizowane było wcześniej przez gaussowskie modele mieszane, to teraz wykorzystywane są systemy uczenia głębokiego. Zmiana ta pozwoliła na redukcję błędów w rozpoznawaniu mowy o 25%.

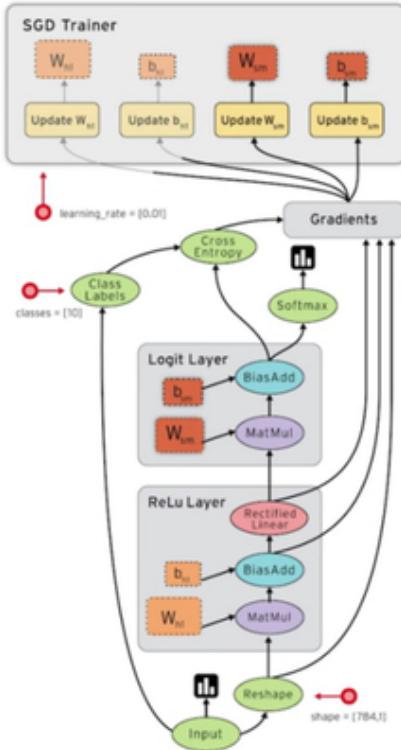
Firma IBM umożliwiła klientom korzystającym z systemu IBM Watson dostęp do API uczenia głębokiego. System Watson zasłynął z pokonania dwóch mistrzów teleturnieju Jeopardy! [21], a obecnie jest oferowany jako produkt do automatycznej interakcji z ludźmi czy tłumaczenia.

Zastosowanie systemów uczenia głębokiego nie jest tak powszechnne w przemyśle jak tych opartych o klasyczne metody, jak MLP czy SVM. Wiodące korporacje IT dostrzegają jednak potencjał systemów uczenia głębokiego. Obecnie są to projekty ograniczone do wdrożeń pilotażowych prowadzonych przez działy R&D, jednak w najbliższych latach możemy spodziewać się seryjnych wdrożeń rozwiązań opartych o uczenie głębokie.

2.2. Obecny stan bibliotek uczenia głębokiego

Uczenie głębokie jako technologia wkracza do przemysłu. Nie dziwi więc fakt powstawania szeregu rozwiązań, które mają być przydatne dla praktyków. Takimi rozwiązaniami są biblioteki pozwalające w prosty sposób wykorzystywać algorytmy uczenia głębokiego, bez konieczności implementacji ich od podstaw. W tej sekcji przedstawiono kilka rozwiązań tego typu.

TensorFlow [4], opracowana w ramach projektu Google Deep Mind, pozwala uruchamiać szereg algorytmów uczenia głębokiego na jednym lub wielu procesorach tudzież jednostkach GPU. Wspiera języki Python i C++. Obecnie wspiera sieci głębokie, sieci splotowe i sieci rekurencyjne. Charak-



Rysunek 1: Wygenerowany graf obliczeniowy dwuwarstwowej sieci neuronowej skonstruowanej za pomocą TensorFlow.

terystyczną cechą TensorFlow jest możliwość zdefiniowania sieci w języku graficznym, zachowując przy tym pełną rozszerzalność wynikającą z możliwości pisania własnych rodzajów warstw i algorytmów uczących. Czyni to TensorFlow dobrą biblioteką dla początkujących, a jednocześnie potężnym narzędziem do zastosowań przemysłowych. Bibliotekę tą charakteryzuje możliwość przedstawiania obliczeń w formie graficznej. Przykładowy zrzut zaprezentowano na rysunku 1.

Biblioteka Torch [7], wykorzystywana m.in. przez Facebook, Google i IBM, pozwala budować głębokie sieci w języku Lua. Wspiera bezpośrednio kompilator JIT tego języka, oraz komplikację do GPU.

Caffe [22], opracowana na Uniwersytecie Berkeley, wykorzystującą język C++. Czyni ją to nieco bardziej skomplikowaną od pozostałych bibliotek, korzystających z wymagających niewielkiego narzutu języków skryptowych.

cuDNN jest zestawem procedur uczenia głębokiego firmy nVidia. Jest o tyle istotna, że od postaw napisana jest do wykorzystywania technologii CUDA, pozwalającej wykonywać obliczenia na jednostkach GPU tejże firmy.

Zazwyczaj nie pisze się w niej sieci bezpośrednio, wspiera ona jednak szereg innych bibliotek mogących kompilować algorytmy do CUDA, takich jak Theano, Caffe czy Torch.

Deeplearning4j [10] wśród pozostałych bibliotek wyróżnia fakt, że jest ona przeznaczona przede wszystkim dla przemysłu, a nie badaczy. Jest zintegrowana z uznanymi systemami przetwarzania Big Data takimi jak Hadoop czy Spark. Wspiera większość dostępnych algorytmów - wśród ciekawszych należy wymienić sieci głębokich przekonań czy rekursywne autoenkodery.

Theano jest biblioteką Pythona implementującą funkcjonalność kompilatora wyrażeń matematycznych na kod wykonywalny przez CPU i GPU (aktualnie wspierane są tylko karty nVidia). Dostępne w niej funkcje czynią ją szczególnie przydatną przy uczeniu głębokim. Theano szerzej opisano w rozdziale 2.7.2.

Istniejące rozwiązania określić można jako dojrzałe. Programista nie będzie miał trudności z zbudowaniem w nich systemu uczenia głębokiego, tak więc z punktu widzenia inżynierii oprogramowania zastosowanie ich nie nastręcza żadnych trudności.

2.3. Rola GPU w uczeniu głębokim

Istotnym problemem uniemożliwiającym uczenie głębokich sieci neuronowych jest wymagana duża moc systemu liczącego. Obliczenia te mają równoległy charakter, zaś wykonywanie ich na sekwencyjnych CPU nie wykorzystuje tej zależności.

Obliczenia prowadzone na procesorach graficznych (GPU), składających się z setek niezależnych, prostych procesorów zmiennoprzecinkowych w znaczący sposób poprawiają prędkość obliczeń. Wiodące jednostki badawcze, opracowując rozwiązania uczenia głębokiego znacząco koncentrują się na obliczeniach GPU, gdyż poprawia to zauważalnie możliwości uczenia sieci [32].

Zasadniczą cechą powodującą przyśpieszenie obliczeń jest fakt, że większość operacji głębokich sieci neuronowych to albo mnożenie dużych

macierzy, albo splot (wykonywany zazwyczaj za pomocą transformacji Fouriera). Karty graficzne są bardzo dobrze zoptymalizowane do przeprowadzania właśnie takich operacji, co skutkuje przyśpieszeniem obliczeń.

2.4. Wybrane architektury uczenia głębokiego

Każda sieć neuronowa charakteryzuje się pewną architekturą. Architektura ta determinuje jej przeznaczenie, możliwości a także wymiarowość przetwarzanych danych.

W pracy tej opisano dwa rodzaje algorytmów uczenia głębokiego. Krótko opisano je w tym dziale.

2.4.1. Głębokie sieci neuronowe

Głębokie sieci neuronowe (DNN, deep neural networks) są rozwinięciem koncepcji perceptronu wielowarstwowego. Budowane są one z dużej ilości warstw perceptronowych. Niewiele różniłoby je od perceptronu wielowarstwowego, gdyby nie różnice w szczegółach. Przede wszystkim, aby umożliwić uczenie takich głębokich sieci, stosuje się nieco inną funkcję aktywacji, taką jak ReLU. ReLU to funkcja aktywacji postaci

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

Jej właściwości szerzej opisano w 2.5.2. Ciekawą cechą tej funkcji jest stała pochodna niezależna od jej wartości, więc nie obserwuje się efektu “zaniku gradientu”, co umożliwia uczenie praktycznie dowolnie głębokich sieci. Ze względu na prostotę obliczenia, uczenie głębszych sieci staje się możliwe obliczeniowo. Dodatkowo, stosuje się nieco inną regularyzację, która wykazuje się dużo lepszymi parametrami uczenia, czyli dropout. Dropout to technika regularyzacji polegająca na stochastycznym “zerowaniu” wyników niektórych neuronów wcześniejszej warstwy i uczeniu jedynie pozostałych, mająca w zamierzeniu zwalczać koadaptację neuronów.

Architekturnie są one tożsame z perceptronem wielowarstwowym, za wyjątkiem możliwości dysponowania wieloma warstwami, zdecydowanie powyżej trzech klasycznych dla perceptronu.

2.4.2. Splotowe sieci neuronowe

Splotowe sieci neuronowe inspirowane są działaniem biologicznych systemów rozpoznających obrazy, chociażby takich jak kora wzrokowa ssaków. W pierwszej warstwie kory wzrokowej, dokąd docierają (u człowieka) z jądra kolankowatego bocznego dane po wstępnej obróbce z siatkówki, rozpoznawane są proste elementy, takie jak krawędzie, w różnych lokalizacjach pola widzenia. W warstwach wyższych, na ich podstawie, rozpoznawane są coraz bardziej złożone elementy [28]. Obszar obrazu na który odpowiada pojedynczy neuron nazywamy **polem receptivejnym** - z natury jest on skończony i dotyczy stosunkowo niewielkiej liczby "pikseli".

W przetwarzaniu wielowymiarowych, ustrukturyzowanych danych, takich jak obrazy, głębokie sieci neuronowe napotykają szereg przeszkód. Przede wszystkim okazuje się, że nieekonomiczne staje się konstruowanie bardzo głębokich sieci neuronowych dla tego typu danych. Ze względu na lokalny charakter danych nie ma sensu korelacja odległych pikseli ze sobą przy rozpoznawaniu prostych, lokalnych elementów. Takie łączenie ma ze sobą sens dopiero na wyższych poziomach, i będzie dotyczyło już przetworzonych danych z niższych warstw.

Pozwala to równocześnie ograniczyć liczbę parametrów, dzięki czemu sieci splotowe mogą być głębsze od równoważnych im głębokich sieci neuronowych. Typową architekturą sieci splotowej jest pewna ilość warstw składających się z warstwy splotowej i max-poolingowych, oraz, na samym końcu, z (w pełni połączonego) perceptronu wielowarstwowego. Ideą takiej architektury jest możliwość wykrywania coraz bardziej złożonych elementów wizualnych w coraz wyższych warstwach sieci splotowej, a następnie podjęcie decyzji w oparciu o szereg takich elementów w perceptronie wielowarstwowym.

2.5. Przegląd komponentów uczenia głębokiego

Uczenie głębokie, jako istotna poprawka do obecnych systemów sieci neuronowych, dysponuje również zestawem własnych technik, różniących

się od tych stosowanych w uczeniu klasycznym. W tym dziale przedstawiono przegląd takich technik. Znajomość ich jest niezwykle ważna do zrozumienia całokształtu uczenia głębokiego.

2.5.1. Regularyzacja typu dropout

Regularyzacją w sieciach neuronowych nazywa się działanie mające na celu zmniejszenie przetrenowania sieci. Przetrenowaniem nazywamy zjawisko w którym sieć dopasowuje się do zbioru danych treningowych w taki sposób, że nie skutkuje to lepszą dokładnością klasyfikacji na zbiorze weryfikacyjnym.

Klasyczne metody regularyzacji, takie jak L1 i L2, bazują na penalizacji sieci za wysokie wartości współczynników. Polegają one na modyfikacji funkcji błędu poprzez dodanie sumy (lub kwadratu sumy, jak w L2) wartości współczynników. Dzięki temu sieć podczas nauki będzie “preferować” niskie wartości parametrów, co powinno zniwelować problem przetrenowania. Jeśli Θ_n to n-ty parametr, to dodatkowe składniki funkcji błędu będą wyrażać się:

$$L1 : R(\Theta) = L_1 \sum_{i=1}^n |\Theta_i|$$
$$L2 : R(\Theta) = L_2 \sum_{i=1}^n \Theta_i^2$$

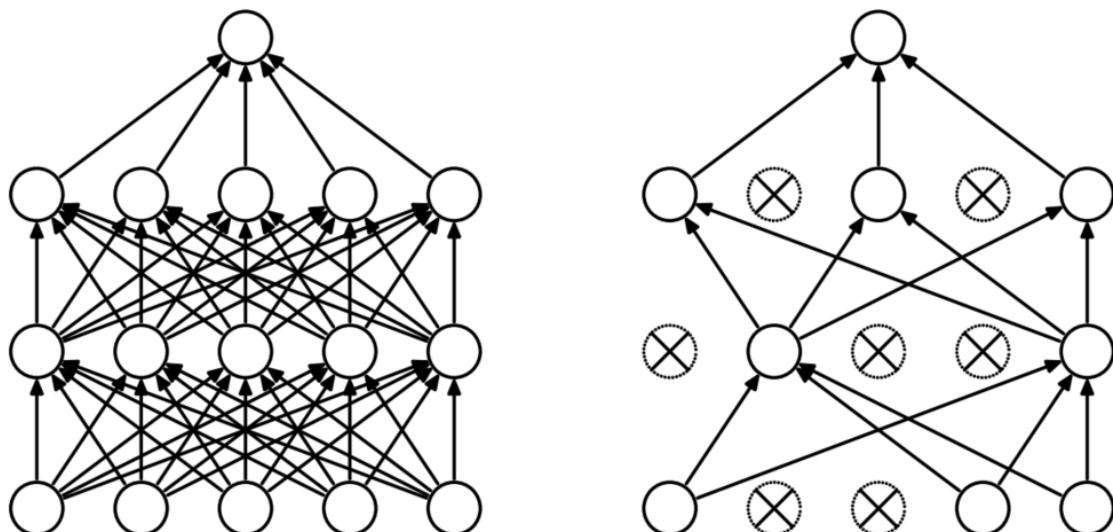
Gdzie L_1 i L_2 są stałymi dobieranymi przez twórcę sieci. Widać więc, że L2 bardziej penalizuje większe wartości współczynników niż L1. Stosować można je również równocześnie. Dropout [31] jest zupełnie inną metodą regularyzacji. Nie opiera się ona na dokładaniu nowych składników do funkcji błędu, a polega na pewnej obserwacji.

W uczeniu maszynowym istnieje pojęcie komitetu klasyfikatorów. Mowa tu o pewnym zbiorze różnych klasyfikatorów, z których każdy dokonuje pewnej klasyfikacji, a wynik jest uśredniany. Taki uśredniony wynik jest wtedy klasyfikacją komitetu. Zazwyczaj jest on lepszy od wyniku pojedynczego klasyfikatora. Ponieważ w tej sytuacji mamy do czynienia z sieciami neuronowymi, więc komitet taki składałby się z wielu sieci neuronowych

o identycznej architekturze, ale różnych parametrach. Niestety, podczas uczenia dużych struktur głębokich rzadko kiedy jest czas na uczenie grupy sieci.

Podczas gdy uczenie sieci za pomocą stochastycznej zejścia po gradiencie powoli zbliża się do poszukiwanego celu, o tyle dropout ze względu na losowe pomijanie neuronów będzie przeszukiwał również pobliskie rozwiązania. Wraz z zmniejszaniem współczynnika uczenia, rozrzut ten będzie odpowiednio mniejszy. Wymaga to ustawienia większego początkowego współczynnika uczenia niż przy podobnych sieciach nie korzystających z dropout, aby można było tą właściwość wykorzystać.

Rozwiązaniem tego problemu jest dropout. Polega on na pomijaniu przy epoce uczenia losowo wybranych neuronów (innych w każdej epoce), a następnie uśrednieniu wyniku. W ten sposób otrzymujemy niejako komitet klasyfikatorów w jednym klasyfikatorze. Dzięki takiemu zabiegowi neurony nie adaptują się wzajemnie i znacznie wzrasta dokładność klasyfikacji. Ideę przedstawiono na rysunku 2.



Rysunek 2: Po lewej standardowa sieć z 2 warstwami ukrytymi, po prawo przeredzona sieć powstała przez zastosowanie dropoutu na sieci po lewej. Przekreślone neurony zostały pominięte. Reprodukcja z [31]

Każdemu neuronowi przypisywane jest prawdopodobieństwo p z jakim przy każdej epoce zostanie on zachowany. Następnie, na początku epoki, generowana jest podsieć, która będzie uczona. Należy zaznaczyć że po-

mijamy (zerujemy) jedynie wagi - wartości progowej (bias) nigdy nie jest pomijamy.

Przy weryfikacji sieć działa nieco inaczej. Każdy neuron zostaje pozostawiony, natomiast wagi neuronu mnożone są przez p . Operacja ta reprezentuje uśrednianie podsieci. Wartości progowej nie przemnażamy, gdyż nie była ona nigdy pomijana.

Regularyzacja dropout charakteryzuje się bardzo dobrymi właściwościami obliczeniowymi, gdyż dodatkowe składniki funkcji błędu nie komplikują pochodnych. Dzięki temu, w uczeniu głębokim regularyzacje L1 i L2 nie są stosowane.

2.5.2. Funkcja aktywacji ReLU

Podstawowym problemem w budowie głębokich struktur przy użyciu klasycznych funkcji aktywacji (takich jak funkcja sigmoidalna czy gaussa) jest zjawisko zanikania gradientu. Ze względu na to że funkcje te przyjmują w pewnych obszarach bardzo niewielkie wartości, to ich pochodna w tym miejscu jest również niewielka, co skutkuje nikłą poprawką do sieci wynikającą z propagacji wstecznej. Skutkiem tego pierwsze warstwy sieci uczą się bardzo powoli, albo w ogóle.

Pewnym rozwiązaniem tego problemu jest funkcja aktywacji ReLU. Prezentuje się ona następująco:

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

Jej pochodna nie istnieje w zerze (zazwyczaj podstawa się wtedy 0) i przyjmuje tylko dwie wartości - 0 lub 1, dzięki czemu jest bardzo szybka do wyliczenia. Zeroowy gradient dodatkowo sprawia że sieć jest odpowiednio rzadka (choć zazwyczaj rzadkość sieci wspomaga się regularyzacją). Aby zapobiec zbyt szybkiemu "wypadaniu" neuronów ReLU, należy rozpoczynać od niskich wartości współczynnika uczenia. Efekt "wypadania" czy też "umierania" (ang. *dying ReLUs*) stanowi do pewnego poziomu problem w uczeniu sieci głębokich, jest to problem niejasny. Pewne wyniki wskazują na to, że nasycenia w zerze w praktyce wspomagają proces uczenia się [12].

Z drugiej strony, przy dużych współczynnikach uczenia się możliwe że znaczna część neuronów “wypadnie”, co zredukuje pojemność modelu. “Wypadanie” objawia się tu przyjęciem wagi 0, co ze względu na zerowy gradient takiego neuronu nie może być później poprawione w drodze uczenia. O ile przy uczeniu za pomocą zejścia po gradiencie problem ten dotyczy ok. 5% neuronów, to o tyle przy wykorzystaniu dropout wartość ta dochodzić może do 60% [14]. Istnieje szereg generalizacji neuronów typu ReLU, eliminujących ten problem. Podstawowym rozwiązaniem tego typu jest “leaky ReLU”, wyrażone wzorem:

$$f(x) = \max(0, x)(x > 0) + (\alpha x)(x < 0)$$

Funkcja taka daje niewielki, regulowany (parametrem α gradient). Zalety takich rozwiązań nad typowymi neuronami ReLU pozostają w obszarze badań, natomiast pewne inne rozwiązania tego samego problemu dają bardzo dobre wyniki [18].

Do zalet tej funkcji wobec funkcji klasycznych należą niewielka złożoność obliczeniowa oraz brak problemu zanikającego gradientu ze względu na stałą wartością pochodnej. Brak zanikania gradienty jest cechą kluczową, bez których budowanie sieci **głębokich** nie jest możliwe, gdyż nie dałoby się później nauczyć takiej sieci. Cechą negatywną ReLU (choć nie jest to do końca jasne) jest eliminacja neuronów przy trenowaniu, co może redukować pojemność modelu.

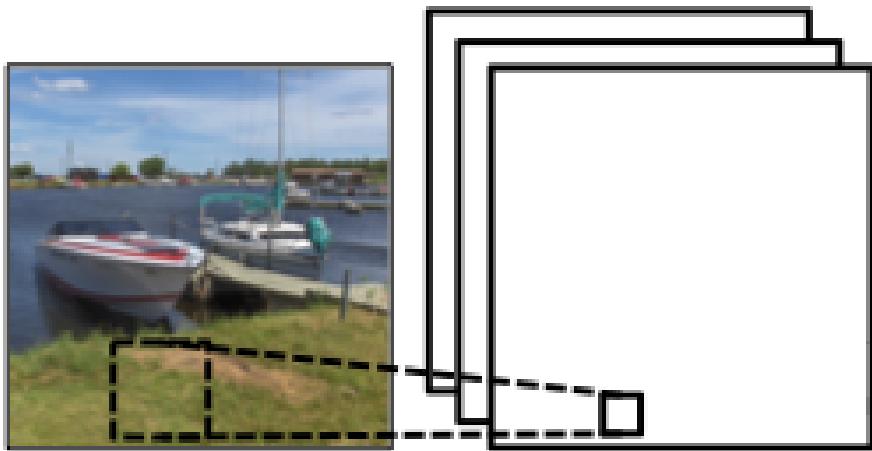
Dzięki wymienionym zaletom, funkcje aktywacji ReLU są szeroko stosowane w uczeniu głębokim.

2.5.3. Warstwa splotowa

Warstwa splotowa jest podstawowym elementem konstrukcyjnym splotowych sieci neuronowych. Na wejściu przyjmuje tensor 3D w postaci obrazu - jego wymiarami są wysokość, szerokość i kanał. Kanały w przypadku obrazów kolorowych mogą być 3, lub 1 w przypadku obrazów czarno-białych. Obraz również jest wyjściem sieci, ale może mieć on odpowiednio więcej lub mniej kanałów, wartość ta zależy od projektanta sieci.

Większa ilość kanałów będzie odpowiadać wzrostowi wymiarów danych wejściowych.

Jeden piksel warstwy wyższej będzie powstawał w wyniku splotu wyckinka obrazu wejściowego i filtra warstwy splotowej (jądra splotu), po czym zaaplikowana zostanie nieliniowa funkcja aktywacji. Idea takiego przetworzenia zaprezentowana jest na obrazie nr 1. Operacja taka zmniejsza rozmiar obrazu w warstwie wyjściowej - jeśli obraz ma szerokość n , a filtr szerokość m , to szerokość obrazu wyjściowego będzie wynosić $n - m + 1$. Główną zasadą jest to, że pojedynczy filtr nakładany jest na cały element obrazu i jego parametry są współdzielone - "uczy" się więc on z całego obrazu. Pojedyncza warstwa splotowa dysponuje zestawem jąder splotu, równemu liczbie warstw wyjściowych. Ilość takich wyjściowych kanałów ogólnie odpowiada to ilości neuronów w warstwie ukrytej DNN.



Rysunek 3: Jeden piksel wyjściowy warstwy splotowej jest generowany na podstawie zbioru pikseli obrazu wejściowego lub niższej warstwy

Jak przedstawiono na obrazie 1, takich filtrów (kanałów) może być więcej. Na wartość każdej z nich różne filtry zdefiniowane w bieżącej warstwie będą składać się w różny sposób. Tak więc wagę filtrów w warstwie sieci splotowej określa tensor 4D, gdzie wymiarami są:

- współrzędna x elementu filtru
- współrzędna y elementu filtru

- numer warstwy wyjściowej (filtru)
- numer warstwy wejściowej

Dla uproszczenia, gdybyśmy zakładali że obraz w warstwie wejściowej ma tylko jeden kanał (obraz monochromatyczny), wage warstwy splotowej opisywałby tensor 3D, jednak ponowne wprowadzenie czterowymiarowości byłoby niezbędne, gdybyśmy chcieli wykrywać więcej niż jedna cechę obrazu i potrzebowali kilku filtrów w wyższych warstwach splotowych.

Należy zauważyc pewną zależność. Gdyby zamiast obrazu analizować szereg czasowy (którego elementy są wektorami), to wagi takiej warstwy opisywałby tensor 3D (jeden wymiar mniej). Gdyby natomiast, zamiast szeregu czasowego, analizować zwykły wektor, degenerując taką sieć do zwyczajnej sieci głębokiej, byłby to tensor 2D - tak, jak w zwykłych sieciach neuronowych. Analogicznie, gdyby budować splotową sieć mającą analizować strumienie wideo, byłby to tensor 5D.

Zakładając, że wagi k-tego kanału warstwy wyjściowej oznaczymy jako W^k a próg jako b^k , to wartość wyjściową w danym miejscu uzyskujemy jako:

$$h_{i,j}^k = f((W^k \star x)_{i,j} + b_k)$$

gdzie $f(x)$ jest funkcją aktywacji, a \star jest operatorem splotu. Podobnie jak w sieciach neuronowych zwykłego rodzaju, można stosować jako funkcję aktywacji przekształcenia sigmoidalne (jak $tanh$) albo $ReLU$.

2.5.4. Max-pooling

Warstwy max-pooling są specjalnymi warstwami w splotowych sieciach neuronowych (CNN). Służą dwojakiemu celowi. Pierwszym celem jest zapewnienie pewnego stopnia niezależności od translacji (przesunięć) obrazu wejściowego, zaś drugim - redukcja liczby przetwarzanych danych. Zamieniają one kanał obrazu na mniejszy (pod względem wysokości i szerokości) dzieląc go na nienakładające się prostokąty i z każdego prostokątu podając maksimum.

Zapewnia to niezależność od translacji w ten sposób, że nie jest dokładnie istotne gdzie wystąpiła dana cecha - ważne że wystąpiła. Zazwyczaj w taki sposób funkcjonują też organizmy biologiczne. Ważniejsze wtedy staje się wykrycie cechy zamiast dokładnego jej umiejscowienia. Dzięki izolacji ważnej cechy redukują też szum. Redukcja liczby przetwarzanych danych wydaje się oczywista, przy odrzuceniu części z nich.

Warstwy max-pooling stosowane są zazwyczaj bezpośrednio po warstwach splotowych.

2.6. Zastosowane narzędzia obliczeniowe

W dziale tym przedstawiono zastosowane narzędzia obliczeniowe, wykorzystywane przy realizacji pracy. Należą do nich język programowania Python, biblioteka Theano oraz autorski pakiet *nnetsys*.

2.6.1. Python

Python to szeroko stosowany język skryptowy wysokiego poziomu ogólnego przeznaczenia. Wspiera on szereg popularnych paradymatów programowania, takich jak programowanie imperatywne, obiektowe czy funkcyjne. Cechuje go wielozadaniowość - jest on stosowany w szerokim spektrum zastosowań, takich jak budowa narzędzi skryptowych, backendów stron internetowych czy modułów systemów operacyjnych. Dzięki bibliotekom takim jak *numpy* czy *scipy* stanowi on coraz popularniejszą alternatywę dla płatnych narzędzi takich jak Matlab (Python jest oprogramowaniem open-source).

Python w zasadzie istnieje w dwóch wersjach. Są one wzajemnie niezgodne, jednak do pewnego stopnia można tworzyć kod który zostanie uruchomiony w obydwu wersjach. Python 3 to uporządkowana wersja języka, jednak ze względu na to, że powstała później, jest mniej popularna. Szerzej stosowany jest Python 2.7.

2.6.2. Theano

Theano [6] jest biblioteką Pythona implementującą funkcjonalność kompilatora wyrażeń matematycznych na kod wykonywalny przez CPU i GPU (aktualnie wspierane są tylko karty nVidia). Pozwala on zapisywać wyrażenia w przejrzysty sposób z wykorzystaniem języka Python, a obliczenia przeprowadzać przy użyciu zoptymalizowanego kodu generowanego kompilatorem C na procesorze lub karcie graficznej. Funkcja ta skraca czas obliczeń w porównaniu do typowych bibliotek obliczeń numerycznych, jakich jak numpy. Theano jest zgodny zarówno z Pythonem 2 jak i 3.

Należy zaznaczyć, że mimo częstej klasyfikacji Theano jako systemu uczenia głębokiego, nie jest on takim systemem. Zawiera on szereg pomocnych operacji elementarnych, jednak to na programiście spoczywa obowiązek odpowiedniej definicji operacji. Jak piszą o Theano samo autorzy:

"Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently" [3]

Obliczenia na potrzeby tej pracy zostały przeprowadzone na komputerze Pracowni Informatyki Medycznej Wydziału Elektrotechniki i Informatyki Politechniki Rzeszowskiej. Do obliczeń została wykorzystana technologia CUDA zrealizowana na karcie nVidia Quadro 6000 [29].

2.6.3. nnetsys

nnetsys jest autorską biblioteką open-source pozwalającą na budowę sieci neuronowych uczonych za pomocą Theano. Napisana w języku Python i zgodna jedynie z wersją 2 Pythona, pozwala ona na budowę:

- wielowarstwowych perceptronów, z funkcjami aktywacji tanh, sigmoidalnej, ReLU i softmax
- warstw splotowych i warstw max-pooling
- stosowanie regularyzacji L1, L2 i dropout

- uczenie metodą z momentum
- uzenie sieci metodą stochastycznego spadku wzdłuż gradientu (SGD, ang. *stochastic gradient descent*)

nnetssys jest otwartym oprogramowaniem i można pobrać je ze strony
<https://github.com/piotrmaslanka/nnetssys>

Przy użyciu tej biblioteki wykonane zostały obliczenia na potrzeby tej pracy. Ponieważ biblioteka została zbudowana jako istotna część pracy, jej szerszemu omówieniu poświęcono podrozdział 3.2.

2.7. Zbiory danych

W tej pracy ograniczono się do dwóch zestawów danych. Decyzja ta była podyktowana chęcią uproszczenia porównań między różnymi architekturami systemów uczących się. Przy wielu różnych zbiorach danych porównanie takie wymagałoby dużo większej liczby eksperymentów, a wnioski byłyby bardziej skomplikowane do interpretacji. Wykorzystywanie prostych zbiorów o niewielkiej liczbie atrybutów czy próbek nie wydaje się być dobrym rozwiązaniem dla sieci głębokich. Należy pogodzić się z tym, że przy takich zbiorach lepsze będą metody prostsze, zgodnie zresztą z *no free lunch theorem*, mówiącym że nie istnieje najlepszy klasyfikator do każdego rodzaju problemu [33]. Dobrano więc zbiory danych, które możliwe są do analizy zarówno klasycznymi sieciami neuronowymi, jak i sieciami głębokimi, co umożliwia ich dokładne porównanie.

2.7.1. MNIST

MNIST to duża baza danych odręcznie napisanych cyfr. Jest ona szeroko stosowana do trenowania różnych systemów przetwarzania obrazów. Zawiera ona 60000 obrazów treningowych i 10000 obrazów testowych. Są to obrazy w skali szarości (pikselowi przypisuje się wartość rzeczywistą z przedziału $< 0; 1 >$), a rozdzielcości 28x28. Przypisuje się im jedną z 10 klas - numer cyfry.

Dostępna jest strona internetowa, z której można pobrać te dane [27]. Prowadzi ona również rejestr najlepszych metod, którymi te dane były przetwarzane, oraz ich wyników. Aktualnie (czerwiec, 2016) najlepszym sposobem klasyfikacji był komitet 35 sieci splotowych (błąd 0.35%).

2.7.2. CIFAR-10

W sytuacjach gdy MNIST wydawać może się zbyt prostym zbiorem, zdecydowano się zastosować zbiór CIFAR-10 [24]. Jest to zbiór obrazów o 10 klasach (np. owoce i warzywa, kwiaty, duzi roślinożercy i wszystkożercy), po 60000 z każdej klasy.

CIFAR-10 jest uzupełnionym w klasy podzbiorem zbioru 80 milionów małych obrazów. Są to obrazy o rozdzielczości 32x32, kolorowe (RGB). Instancja ma więc 3072 ciągły atrybutów i 10 możliwych klas. Powinien on być wystarczająco trudny dla celów porównawczych sieci klasycznych i głębokich.

3. Eksperymenty

3.1. Test stabilności numerycznej algorytmu

Dokonując obliczeń na karcie graficznej istotną sprawą staje się stabilność numeryczna zastosowanego algorytmu. Biblioteka theano w przypadku wykonywania obliczeń na GPU wymaga zastosowania liczb zmiennoprzecinkowych o 32-bitowej precyzji. Licząc na CPU, wykorzystywane są liczby o 64-bitowej precyzji. Należy więc ustalić, czy i jaki wpływ na stabilność numeryczną obliczeń mają zastosowane typy.

Sprawdzenia tego zdecydowano się dokonać wykonując identyczny test przesiewowy na CPU (64-bitowa precyzja) i GPU nVidia Quadro 6000 (32-bitowa precyzja). Test ten polegał na uczeniu struktury głębokiej przez 80 epok przy zmiennym współczynniku uczenia (bez momentum). Zastosowano sieć z następującymi warstwami:

- 768 neuronów wejściowych, bez dropoutu
- 900 neuronów w warstwie ukrytej, f. akt. ReLU, dropout p=0.5
- 900 neuronów w warstwie ukrytej, f. akt. ReLU, dropout p=0.5
- 300 neuronów w warstwie ukrytej, f. akt. ReLU, dropout p=0.5
- 50 neuronów w warstwie ukrytej, f. akt. ReLU, dropout p=0.5
- 10 neuronów w warstwie wyjściowej, f. akt. softmax, bez dropoutu

Sieć tą uczono na bazie danych MNIST z uczeniem typu stochastyczne zejście wzdłuż gradientu. Wykonano test przesiewowy dla zmiennej wartości współczynnika uczenia od 0.3 do 0.9 z krokiem 0.1, oraz dla zmiennej liczności obserwacji, od 10 do 70 z krokiem 5. Spodziewa się, że jeśli dokładność klasyfikatora zbudowanego na CPU i GPU nie będzie istotnie różna, to algorytm jest stabilny numerycznie i nic nie stoi na przeszkodzie by do dalszych testów stosować procesor graficzny.

Kod źródłowy przedstawionego testu, oraz pełne wyniki znajdują się w załączniku w katalogu *przesiew_bs_alpha*.

Tabela 1: Wyniki testu stabilności numerycznej algorytmu

GPU			CPU		
Rozm. batch	Wsp. ucz.	Dokł.	Rozm. batch	Wsp. ucz.	Dokł.
10	0,3	96,919996%	10	0,3	96,92%
65	0,4	93,709999%	65	0,4	93,71%
30	0,6	96,529996%	30	0,6	96,53%
60	0,7	95,419997%	60	0,7	95,42

Celem testu było sprawdzenie stabilności numerycznej algorytmu uczącego. Wybrane wiersze wyników przedstawiono w tabeli nr 1.

Rozmiar batch określał rozmiar *minibatch* stosowanego przy uczeniu metodą SGD, a współczynnik uczenia o procencie poprawki jaki był przyjmowany na wagi. Dokładność to procentowa ilość poprawnie zidentyfikowanych cyfr ze zbioru weryfikującego.

Co prawda wartości z CPU były w stanie być lepiej zaokraglone (64-bitowa wartość), jednak po zaokragleniu są one identyczne z tymi z GPU. Prowadzi to do wniosku że algorytm jest stabilny numerycznie, a dalsze obliczenia na potrzeby tej pracy można kontynuować za pomocą karty graficznej. Pozwoli to zredukować czas niezbędny do wyćwiczenia sieci, co przyśpieszy badanie bez dodatkowych kosztów w jakości sieci.

3.2. Opis biblioteki

nnetsys, jako biblioteka obiektowego języka programowania, również jest obiektowa. Zawiera następujące obiekty:

- **MaxpoolingLayer** - warstwa typu max-pooling. Jedyne, co określa użytkownika, to rozmiar “prostokątu” z którego wybrane będzie maksimum.
- **ConvolutionalLayer** - warstwa splotowa. Użytkownik określa ilość jąder splotu, ich rozmiary, oraz ilość kanałów obrazu wejściowego. Określić może również rodzaj funkcji aktywacji.
- **ReshapeLayer** - różne warstwy spodziewają się danych w różnym

formacie. Warstwa perceptronowa spodziewać będzie się macierzy, a splotowa czy max-poolingowa - w formacie tensora 4D. Ta warstwa implementuje zmianę charakteru danych pomiędzy warstwami.

- **PerceptronLayer** - warstwa wielowarstwowego perceptronu. Użytkownik określa ilość neuronów warstwy, rodzaj funkcji aktywacji oraz opcjonalny dropout czy dropconnect. Można więc za jej pomocą zbudować zarówno klasyczny perceptron wielowarstwowy, jak i sieci głębokie.
- **Network** - wirtualna “wastwa” umożliwiająca “spięcie” pozostałych warstw w jedną klasę.
- **Classifier** - klasa, mająca daną sieć, pozwalającą sklasyfikować jeden przykład
- **Validator** - klasa obliczająca statystyki jakości sieci (dokładność, macierz błędów) na podstawie zbioru weryfacyjnego czy testowego
- **MinibatchSGDTeacher** - klasa implementująca uczenie sieci metodą stochastycznego zejścia gradientu przy użyciu “minibatch”. Można zadać jej regularyzacje L1/L2, współczynnik uczenia i współczynnik momentum.
- **affine** - umożliwia przeprowadzanie operacji na danych takich jak transformacje afiniczne, zmiana kontrastu i jasności czy translacje. Są one stosowane do sztucznego powiększania zbioru danych uczących.

Klasy te są implementacjami powyższych algorytmów przy użyciu funkcji Theano. O ile Theano zapewnia część funkcji algebraicznych (splot, funkcja sigmoidalna, max-pooling), o tyle dokładniejsze “spięcie” tych funkcji, aby działały jako sieć neuronowa, realizuje już *nnetssys*.

Dzięki tej bibliotece można znacznie uprościć budowanie i tworzenie sieci neuronowych. Aby zbudować i nauczyć sieć splotową rozpoznającą obrazy MNIST, wystarczy już kod podany na listingu 1.

Listing 1: Budowa splotowej sieci neuronowej za pomocą biblioteki nnetsys

```
nnet = Network(  
    ReshapeLayer((-1, 1, 28, 28)),  
    ConvolutionalLayer(5, 5, 1, 20),  
    MaxpoolingLayer(2, 2),  
    ConvolutionalLayer(5, 5, 20, 50),  
    MaxpoolingLayer(2, 2),  
    ReshapeLayer((-1, 50*4*4)),  
    Perceptron(50*4*4, 500, activation='tanh'),  
    Perceptron(500, 10, activation='softmax')  
)  
  
teacher = MinibatchSGDTeacher(nnet, data[0], batch_size=500,  
                               learning_rate=0.1)  
  
for i in xrange(0, 100):  
    teacher.train_epoch()
```

Ciekawa w Theano jest sama konstrukcja wyrażeń matematycznych odpowiedzialnych za budowę sieci neuronowych. Aby zbudować warstwę regresji liniowej, mając już zainicjalizowane warstwy (nnetsys inicjalizuje warstwy metodą Glorot, zaś warstwę regresji zerami). Najpierw należy zbudować zmienne współdzielone dla wag i biasu, co informuje Theano że mogą być one załadowane na pamięć układu liczącego (np. karty graficznej):

```
W = theano.shared(w)  
B = theano.shared(b)
```

Następnie definiujemy charakter parametrów wejściowych, wyjściowych i współczynnika uczenia. Określenie takiego charakteru jest niezbędne, aby kompilator wiedział z jakim rodzajem danych ma do czynienia:

```
x = T.matrix('x')  
y = T.ivector('y')  
a = theano.shared(0.2)
```

Tak więc wejście do macierz liczb rzeczywistych (n próbek razy m atry-

butów), a wyjście to wektor liczb całkowitych (n numerów klas). Następnie można zbudować funkcję która liczy wyjście warstwy regresji liniowej:

```
fun = T.nnet.softmax(T.dot(x,W)+B)
```

Warstwa propagacji wstecznej, podobnie jak inne warstwy, uczona będzie metoda propagacji wstecznej. Aby jednak dało się zastosować tą metodę, niezbędne jest zdefiniowanie funkcji kosztu. Taką funkcją dla warstwy liniowej jest funkcja *negative log-likelihood*. Ponieważ mamy wiele przykładów, gdyż trenujemy wsadowo, interesować nas będzie średnia *negative log-likelihood*. W przeciwnieństwie do sumy uniezależnia nas ona od rozmiaru zbioru uczącego. Po zapisie, funkcja ta będzie wyglądać następująco:

```
cost_f = -T.mean(
    T.log(
        self.ff_net.get_learning_passthrough(self.x)
    )
    [T.arange(self.y.shape[0]), self.y]
)
```

Funkcja ta może być wykorzystywana osobno, jednak w obecnej chwili najbardziej interesująca jest funkcja uczenia warstwy regresji:

```
train_model = theano.function(
    inputs=[x, y],
    updates=[
        (W, W - T.grad(cost_f, W) * a),
        (B, B - T.grad(cost_f, W) * b),
    ]
)
```

W tym momencie wywołanie funkcji `train_model(dane, klasy)` realizuje jedną rundę uczenia wsadowego. Zauważać można, że pochodna wykorzystywana do propagacji wstecznej liczona jest przez bibliotekę, co umożliwia programiście definiowanie nawet skomplikowanych funkcji kosztu bez konieczności podania ich pochodnych. nnetsys pozwala również na rapor-

towanie informacji jakościowych na temat sieci. W tym przypadku, aby po uczeniu wypisać takie informacje, można skorzystać z klasy *Validator* w następujący sposób:

```
validator = Validator(nnet, data[1])  
print 'Jakosc sieci: %s wartosc funkcji celu: %s' % (  
    validator.validate(),  
    validator.calculate_loss())  
print 'Macierz bledu: %s' % \  
(validator.calculate_confusion_matrix(), )
```

Gdzie *data[1]* to zbiór weryfikacyjny. Można również podać *data[0]* aby weryfikować wyniki na zbiorze treningowym. Kod ten wypisuje jakość sieci na zbiorze testowym, wartość funkcji celu, oraz macierz błędu (ang. confusion matrix). Wykorzystany model można również wykorzystać do klasyfikacji, korzystając z klasy *Classifier*. Poniższy kod prezentuje sposób sklasyfikowania całego zbioru testowego. Wyniki takiej klasyfikacji można następnie porównać z prawdziwymi etykietami (w tym wypadku zachowanymi w *data[1][1]*).

```
print 'Klasyfikacja zbioru treningowego: %s' % (  
    classifier.classify(data[1][0]))
```

Opisany kod jest przez Theano kompilowany i wykonywany na CPU lub GPU, w zależności od ustawień. Oczywiście, deklarowanie tych funkcji dla każdej warstwy sieci neuronowej mija się z celem w przypadku uczenia sieci głębokich, składających się z wielu warstw. Podobne fragmenty kodu zostały zebrane w przedstawione biblioteki nnetsys zawierającej implementacje wybranych algorytmów uczenia głębokiego przedstawionych w tym rozdziale, wraz z dodatkowymi funkcjami. Kod jest bardziej skomplikowany od przedstawionego tu, jednak idea jest w zasadzie identyczna.

Celowość implementacji własnej biblioteki uczenia głębokiego można poddać pod dyskusję, jednak ze względu na możliwość kontroli sieci na każdym etapie pozwala ona prowadzić dokładniejsze badania i poczynić potencjalnie więcej spostrzeżeń na temat ich działania.

3.3. Kwestia inicjalizacji wag

Głębokie sieci neuronowe korzystające z funkcji aktywacji ReLU wymagają innego sposobu inicjalizacji wag niż sieci oparte o *tanh* czy *sigmoid*. W sekcji tej przedstawiono metody inicjalizacji wag i przedstawiono eksperyment, który służył porównaniu różnych sposobów.

Rozsądne wydaje się nie inicjalizowanie wag wartościami ujemnymi przy inicjalizacji bias na 0. Wydawać mogłoby się, że przy normalizacji danych, które mają być wykorzystywane w sieciach opartych o ReLU, należy minimum atrybutu ustalić na wartość $0 + \epsilon$, gdyż bias sieci zwyczajowo inicjalizuje się na 0, natomiast zarówno gradient jak i wartość ReLU przy wejściu 0 wynosi 0, co właściwie wyłącza neuron warstwy wyjściowej z działania. Eksperymentalnie okazuje się jednak, że blokowanie wstępnej propagacji gradientu na niektórych warstwach jest zaletą. Cytując [12]

One may hypothesize that the hard saturation at 0 may hurt optimization by blocking gradient back-propagation. (...) However, experimental results tend to contradict that hypothesis, suggesting that hard zeros can actually help supervised training. We hypothesize that the hard non-linearities do not hurt so long as the gradient can propagate along some paths, i.e., that some of the hidden units in each layer are non-zero. With the credit and blame assigned to these ON units rather than distributed more evenly, we hypothesize that optimization is easier."

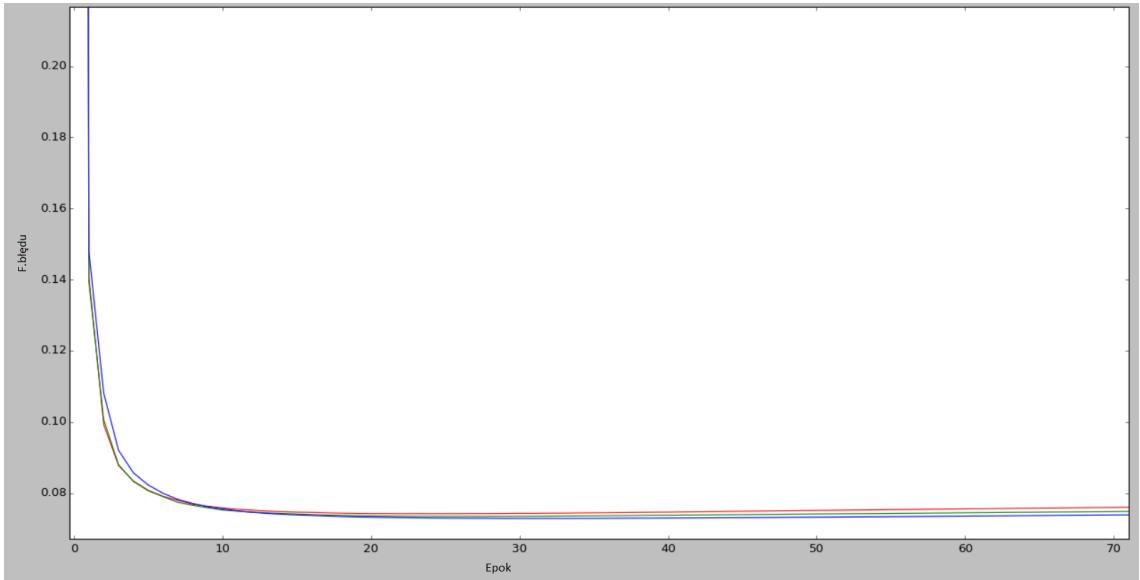
Szybki przegląd literatury ujawnia kilka możliwych podejść do tego problemu:

1. Inicjalizacja wag rozkładem Gaussa o średniej 0 i odchyleniu standardowym 0.01. Inicjalizacja biasów w warstwach nieparzystych 0 i parzystych 1 [25].
2. Inicjalizacja wag rozkładem Gaussa o średniej 0 i odchyleniu standardowym $\sqrt{\frac{2}{n_1}}$, gdzie n_1 jest liczbą neuronów w pierwszej warstwie. Inicjalizacja biasów wartością 0 [18].

3. Inicjalizacja wag rozkładem Gaussa o średniej 0 i odchyleniu standardowym 0.01, inicjalizacja biasów wartością 0.01 [30].
4. Inicjalizacja wag rozkładem jednolitym $[-\sqrt{\frac{6}{n_j+n_{j+1}}}, \sqrt{\frac{6}{n_j+n_{j+1}}}]$ gdzie n_j jest rozmiarem wektora wchodzącego do warstwy a n_{j+1} rozmiarem wychodzącego, popularna przy funkcjach aktywacji *tanh* i *sigmoid*. Inicjalizacja biasów wartościami 0 [11].

Wspólna dla tych opcji wydaje się być inicjalizacja ReLU niewielkimi wartościami, bliskimi zeru. W przybliżeniu równa liczba tych wartości będzie dodatnia, co ujemna. Porównano więc inicjalizację sieci za pomocą Var= $\sqrt{\frac{2}{n_1}}$ i Bias=0.1 a metodą Var= $\sqrt{\frac{2}{n_1}}$ i Bias=0 oraz inicjalizacją Glorot i Bias=0. Współczynnik uczenia ustalany był wzorem $\alpha = \frac{0.1}{n_{repoki}}$.

Przeprowadzono to poprzez porównanie wartości średnich wartości funkcji błędu podczas uczenia sieci przez 100 epok. Kod źródłowy oraz wartości funkcji znajdują się w katalogu *porow_inicjalizacji*. Metody ze stałą wariancją nie wypróbowano, gdyż wydaje się, że jej jakość będzie zależna od rozmiaru sieci (pozostałe metody przy obliczaniu wariancję biorą pod uwagę rozmiar warstwy).



Rysunek 4: Czerwony pasek obrazuje inicjalizację $\sqrt{\frac{2}{n_1}}$, zielony $\sqrt{\frac{2}{n_1}}$ z bias=0.1, a niebieski inicjalizację Glorot.

Wizualizacja wyników, którą zaprezentowano na rysunku 4 prowadzi do dość ciekawych wniosków. Metoda Glorot, opracowana dla klasycznych

funkcji aktywacji, w przypadku ReLU nie jest zupełnie nieuzasadniona, zapewniając początkowo gorsze warunki, jednak na dłuższą metę będąc lepsza od pozostałych funkcji. Drugą z kolei jest funkcja pierwiastkowa z biasem 0.1. Wydaje się, że dodatkowa wartość dodatnia sprawia że neurony ReLU rzadziej wypadają poza 0, dzięki czemu większa ich ilość bierze udział w obliczeniach. Reasumując, widać tutaj jednak duże podobieństwo, które nie powinno być brzemienne w skutkach dla obliczeń realizowanych na potrzeby tej pracy.

3.4. Wpływ współczynnika uczenia

W trakcie uczenia sieci opartych o dropout pojawił się pewien problem. Mianowicie, sieć głęboka z użyciem dropout miała niezwykle powolną (żeby nie powiedzieć zerową) zbieżność przy niskim początkowym współczynniku uczenia. Przy wartości 0.01 sieć nie mogła wyjść z dużej wartości funkcji straty, a uczenie ze względu na to kończyło się. Problem naprawiło ustalenie początkowego współczynnika uczenia na 0.1. Sieć płytka nie miała takich problemów. Wyłączenie dropout pozwalało sieci głębokiej zbiegać się nawet przy wartościach LR rzędu 0.01.

Wniosek płynący ze spostrzeżenia nakazuje wiązać funkcjonowanie dropout z odpowiednią wartością współczynnika uczenia. Istotnie, praktycy [31] zalecają stosowanie wysokiego początkowego współczynnika uczenia podczas korzystania z dropoutu.

Aby szerzej zbadać ten problem, zdecydowano się na przeprowadzenie eksperymentu. Miały one na celu zbadać końcową jakość sieci przy różnym początkowym współczynniku uczenia oraz bardzo krótkim uczeniu. Pozwoli to sprawdzić, czy przy danym współczynniku uczenia zachodzi proces uczenia się sieci. Jako zbiór danych przyjęto MNIST. Sieci *tanh* i ReLU nie miały porównywalnej architektury, więc z eksperymentu można zauważyć tylko jak kształtuje się jakość sieci w funkcji współczynnika uczenia.. Wartość momentum wynosiła 0, funkcją aktywacji było ReLU, dropout 50%. Sieć uczono przez 5 iteracji (aby mieć porównanie wpływu startowego współczynnika uczenia), rozmiar batch 150. Poprzez ogranicze-

Tabela 2: Wyniki testu sieci dropout przy różnych współczynnikach uczenia

Wsp. ucz.	0.005	0.01	0.08	0.1	0.2	0.4
Jakość	74.55%	85.35%	95.75%	96.31%	97.3%	97.69%
Wsp. ucz.	0.5	0.8	0.9	1		
Jakość	97.55%	97.39%	97.33%	96.78%		

Tabela 3: Wyniki testu sieci tanh przy różnych współczynnikach uczenia

Wsp. ucz.	0.005	0.01	0.08	0.1	0.2	0.4
Jakość	88.24%	89.95%	92.85%	93.11%	94.35%	95.68%
Wsp. ucz.	0.5	0.8	0.9	1		
Jakość	96.06%	91.49%	86.23%	73.91%		

nie liczby iteracji można zilustrować jak przebiega proces uczenia się sieci dla konkretnych wartości startowych, nie jest jednak możliwe ustalenie końcowej wartości sieci. Jest to jednak dobry eksperyment, by ustalić z jakich wartości początkowych należy zaczynać uczenie sieci. Dla każdej architektury uczono 3 sieci, po czym wybierano tą z najlepszym wynikiem. Uczyniono to w celu zminimalizowania wpływu losowości wynikającej z inicjalizacji sieci. Współczynnik uczenia wyrażono:

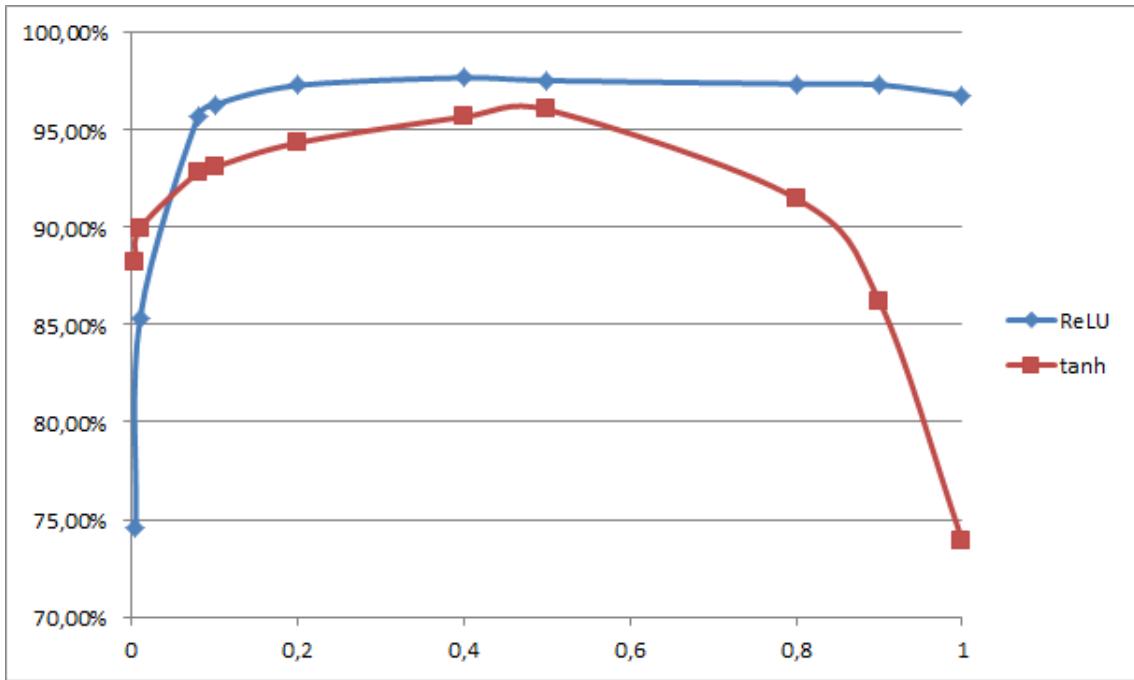
$$\alpha_{iteracja} = 0.9^i \alpha$$

gdzie i to numer iteracji. Oprócz samej jakości końcowej, weryfikowanej na zbiorze testowym, rejestrzano również wartość funkcji celu, obliczanej na zbiorze weryfikacyjnym. Wyniki przedstawiono na w tabeli 2. Skrypt dokonujący obliczeń znajduje się w katalogu *jakosc_dropout*.

Porównano zmiany funkcji celu w kolejnych iteracjach. Są one różne (różna jest dokładność), ale proporcjonalne do siebie, więc z punktu analizy zjawiska nie są interesujące. Jako grupę kontrolną przeprowadzono również analogiczny eksperyment na sieciach z funkcją aktywacji tanh bez regularyzacji dropout. Wyniki przedstawiono w tabeli 3.

Wyniki porównano w postaci wykresu na rysunku 5.

Widać więc, że o ile uczenie sieci za pomocą metod klasycznych daje podobne wyniki niezależnie od wyboru startowego współczynnika uczenia,



Rysunek 5: Porównanie jakości sieci współczynników uczenia dla dwóch różnych rodzajów sieci.

o tyle sieć oparta o dropout i ReLU koniecznie musi startować z wysokich współczynników uczenia. Zjawisko to zostało opisane w literaturze [31]:

"Dropout introduces a significant amount of noise in the gradients compared to standard stochastic gradient descent. Therefore, a lot of gradients tend to cancel each other. In order to make up for this, a dropout net should typically use 10-100 times the learning rate that was optimal for a standard neural net."

Zjawiskiem spodziewanym jest, że przy bardzo wysokich współczynnikach uczenia jakość sieci tanh będzie dramatycznie spadać. Taki właśnie wynik uzyskano. Możliwe jest, że w dalszych iteracjach wagi sieci są przeregulowane w sposób który powoduje wysycenie gradientów i zanik procesu uczenia sieci, jednak sprawdzenie tej tezy wymagałoby dalszych badań. Zjawisku temu nie podlega natomiast sieć oparta o ReLU, która jest odporna jest na wysycenie gradientu ze względu na liniową pochodną (zawsze 0 lub 1, niezależnie od wag). Ze względu na użyty sposób regularyzacji, sieć dodatkowo zyskuje w ten sposób na jakości. W tym przypadku nie jest

to wartość optymalna (bo taką okazało się 0,4), jednak przy skrajnie wysokich wartościach jakość sieci nie ulega znacznemu pogorszeniu. Stanowi to istotną różnicę w stosunku do sieci klasycznych.

Innym sposobem rozwiązania tego problemu może być ustawienie odpowiednio wysokiej wartości momentum, która w dużym stopniu zniweluje szum wynikający z zastosowania dropout. Niezależnie od sposobu rozwiązania tego problemu, trenując sieci neuronowe korzystające z regularyzacji dropout warto pamiętać o tym zjawisku i wykorzystywać odpowiednio wysokie współczynniki uczenia, zazwyczaj daleko przewyższające typowe współczynniki. Można również przeprowadzić przeszukiwanie różnych wartości współczynników, przyjmując zakres przeszukiwania o górnej granicy równej 1, czego nie stosuje się w klasycznych sieciach.

3.5. Rozmiar wsadu, momentum - sieci głębokie a klasyczne

Proces uczenia sieci neuronowych charakteryzuje się doborem tzw. hiperparametrów. Są to “nastawy” dotyczące procesu uczenia, takie jak sposób zmiany współczynnika uczenia w czasie, wartość momentum czy rozmiar wsadu (ang. batch). Najczęściej precyzyjne wartości hiperparametrów otrzymuje się w wyniku eksperymentów, istnieją również zalecenia co do tego, jakich wartości należy się spodziewać przy różnych typach sieci. Zazwyczaj hiperparametrów poszukuje się na skali logarytmicznej, poszerzając zakres jeśli wartość wypada na granicy przedziału poszukiwania.

Ten eksperiment miał na celu ustalenie czym różnią się typowe wartości hiperparametrów (jeśli w ogóle się różnią) między sieciami klasycznymi i głębokimi. Można się spodziewać, że przy stosowaniu diametralnie różnych funkcji aktywacji i sposobów regularyzacji rzeczywiście wystąpią jakieś zmiany. Aby to ustalić, wykonano serię testów, w których:

- dla sieci głębokich badano wpływ parametru momentum i rozmiaru wsadu
- dla sieci klasycznych badano wpływ parametru momentum, rozmiaru

wsadu i parametru regularyzacji L2

W celu badań posłużyły się sieciami o architekturze opisanej w 3.6.1 i zestawem danych MNIST. Wykonano 48, eksperymentów, dla możliwych wartości:

- momentum: 0.02, 0.1, 0.5, 0.9
- rozmiaru wsadu: 10, 50, 100, 200
- (dla sieci klasycznych) wartości regularyzacji L2: $2 \cdot 10^{-6}$, $2 \cdot 10^{-5}$, $2 \cdot 10^{-4}$

Każda sieć uczona była przez 13 minut. Warto wspomnieć jeszcze, że w tym eksperymencie sprawdzano uczenie metodą zejścia po gradiencie z momentum. Istnieje szereg innych metod, które nie były przedmiotem tego eksperymentu. Wyniki przedstawiono w tabeli 4, skrypt dostępny jest w katalogu *hiperparametry*.

Pierwsze spostrzeżenie, które można poczynić, to jakość generowanych sieci ReLU była w dużym stopniu uzależniona od rozmiaru wsadu. Większe rozmiary wsadów konsekwentnie skutkowały lepszymi sieciami. Spostrzeżenie to nie zachodzi w przypadku sieci opartych o funkcję tanh. Przy małych rozmiarach wsadu i dużych wartościach momentum sieci ReLU uczenie nie było zbieżne. Problem ten w mniejszym stopniu dotyczył funkcji *tanh*, choć można dostrzec, że przy słabej regularyzacji L2 zjawisko to pojawiało się również tam.

Można próbować wyjaśnić to zjawisko, powołując się na sposób działania regularizacji dropout zastosowany w uczonych sieciach ReLU. Stochastyczne wyłączanie z działania neuronów zwiększa szum w sieci. Taki sam wpływ ma również stosowanie małych minibatchów, gdyż im mniejszy ich rozmiar, tym mniej reprezentatywne są dane w stosunku do całego zbioru danych. Przy pewnej granicznej wartości szumu sieć uczy się powoli. Jeśli jeszcze dodać dużą wartość momentum to wektor parametrów przestaje poruszać się w kierunku minimum, przekraczając granicę przy której sieć uczy się powoli, a przestaje uczyć się w ogóle. Tłumaczy to również słabe wyniki sieci ReLU przy niskich wartości momentum.

Tabela 4: Jakości sieci w funkcji różnych wartości wsadu, regularyzacji i momentum

ReLU, dropout=0.5				
M/Batch size	10	50	100	200
0.02	96.46%	97.34%	97.5%	96.58%
0.1	95.92%	97.07%	97.52%	96.89%
0.5	9.8%	97.19%	97.4%	97.59%
0.9	9.91%	9.8%	96.55%	97.4%
Tanh, L2=2E-6				
M/Batch size	10	50	100	200
0.02	95.64%	98.06%	97.76%	97.38%
0.1	94.72%	97.92%	97.86%	97.47%
0.5	9.8%	97.7%	98%	97.81%
0.9	9.82%	8.92%	94.8%	96.6%
Tanh, L2=2E-5				
M/Batch size	10	50	100	200
0.02	95.87%	98.05%	97.95%	97.37%
0.1	95.15%	97.99%	97.88%	97.44%
0.5	89.98%	97.86%	98.01%	97.91%
0.9	84.68%	96.93%	95.16%	96.53%
Tanh, L2=2E-4				
M/Batch size	10	50	100	200
0.02	97.57%	97.92%	97.66%	97.24%
0.1	97.44%	97.96%	97.73%	97.28%
0.5	97.22%	98.02%	97.97%	97.76%
0.9	88.96%	98.15%	98.14%	98.04%

Na podstawie eksperymentu nie da się zaobserwować istotnego wpływu wartości momentum na jakość uczenia się sieci ReLU oraz tanh. Da się to wytlumaczyć stosowaniem stałej wartości momentum przez cały cykl uczenia, podczas gdy w praktyce wartość ta jest zmienna w funkcji epoki. Temat badania wpływu momentum na zachowanie sieci wykracza poza zakres tej pracy, ale stanowi ciekawy temat badawczy.

Brak większej ilości jasnych korelacji budzi wątpliwości co do zastosowanej metody badawczej. Możliwe jest, że sposób którym uczyono sieci (hiperboliczny zanik współczynnika uczenia i stała wartość momentum) jest metodą dalece nieoptymalną. Zarzut ten wymaga dalszych badań, jasne jest jednak że w przypadku uczenia głębokiego, zwłaszcza wykorzystującego regularyzację dropout, kluczowe jest stosowanie dużych rozmiarów minibatch.

Należy wspomnieć jednak, że przy tym eksperymencie uczenie ograniczone było czasem rzeczywistym, nie zaś liczbą epok. Przeiterowanie przez wiele podzbiorów (tym więcej im mniejszy minibatch) zajmowało więcej czasu i tak, sieć ReLU z minibatch 10 wykonała 19 iteracji, zaś ta sama sieć przy rozmiarze minibatch 200 wykonała 102 iteracje. Wnioskować można, że skoro sieć przez 19 iteracji nie poczyniła żadnych postępów ponad sieć zainicjonowaną losowo, to takich postępów już nie poczyni, ale aby wyeliminować możliwość pomyłki, przeprowadzono dodatkowy test. Obie sieci uczyono przy tych samych hiperparametrach, za wyjątkiem rozmiaru minibatch. Parametr momentum ustawiono na 0.8, uczyono sieć przy rozmiarze 10 i 200, obie przez 50 iteracji. Uczenie sieci z mniejszym rozmiarem wsadu zajęło ok. 5 razy więcej czasu. Zgodnie z oczekiwaniami nie zaobserwowano zbieżności (dokładność 9,79%) zaś sieć z rozmiarem wsadu 200 miała dokładność 97,11%. Potwierdza to przypuszczenie na temat wpływu dużej ilości szumu na uczenie się sieci ReLU.

3.6. Porównanie jakości sieci

Kolejnym eksperymentem było porównanie sieci klasycznej i sieci uczenia głębokiego. Rozsądny oczekiwaniem jest to, że nie można porówny-

Tabela 5: Ogólne wzory sieci podlegających testom jakości dla zbioru MNIST

Atrybut	Sieć klasyczna	Sieć głęboka
Funkcja aktywacji	tanh	ReLU
Regularizacja	L2=0.0005	Dropout=0.5
Startowy wsp. ucz.	0.2	1
Startowy momentum	0.4	
Aktualizacja wsp. ucz.	$wsp = wsp \cdot 0.5$	
Aktualizacja momentum	$mom = \max(mom + 0.06, 0.99)$	
Rozmiar wsadu	100	
Architektura	768-W1-W2-10	768-W1-W2-W3-10

wać sieci neuronowych i głębokich pod względem liczby neuronów. Ciężko jest bowiem ustalić jak będzie wyglądać “odpowiadająca” sieci klasycznej sieć głęboka. Z tego powodu należy przyjąć inne kryterium porównania sieci. Po analizie możliwych rozwiązań zdecydowano, że będzie to kryterium czasu uczenia sieci.

Po upływie wyznaczonego czasu, uczenie zarówno sieci klasycznej jak i głębokiej, zostanie zatrzymane, a sieci zostaną ocenione. Parametry sieci zdecydowano się dopasować na podstawie rozważań z wcześniejszych części tego rozdziału.

3.6.1. MNIST

Do celu porównania sieci klasycznej i głębokiej na zbiorze danych MNIST zbudowano szereg sieci typu klasycznego i głębokiego. Różniły się one jedynie liczbą neuronów w konkretnych warstwach. Skrypt realizujący ten eksperyment znajduje się w katalogu *dnn_mnist*. Opis porównywanych sieci zamieszczono w 5.

Po każdej epoce aktualizowano momentum i współczynnik uczenia. Nauczono łącznie 450 sieci o różnych ilościach neuronów. Wszystkie sieci uczone przez 3 minuty, co skutkowało typowo 16-28 iteracji algorytmu uczącego. Wartości W1, W2 i W3 zawierały się w zbiorze 800, 1000, 1200, 1400, 1600 . Praktyczne eksperymenty wykazały, że czas ten jest

Tabela 6: Wybrane wycinki rankingu sieci MNIST

Pozycja	Architektura	Jakość
1	ReLU, 1600-1200-1400	97.64%
2	ReLU, 1400-1000-1600	97.62%
3	ReLU, 1600-1400-800	97.58%
4	ReLU, 1600-1000-1400	97.56%
5	ReLU, 1600-1200-1200	97.53%
...		
120	Tanh, 1200-800	96.41%
121	ReLU, 800-1000-1400	96.52%

wystarczający do osiągnięcia zbieżności przez oba rodzaje sieci. Każda architektura uczona była trzy razy, więc jej wynikiem jest wynik najlepszy. Istniało więc 150 różnych architektur sieci. W tabeli 6 poniżej zestawiono wybrane wycinki rankingu sieci.

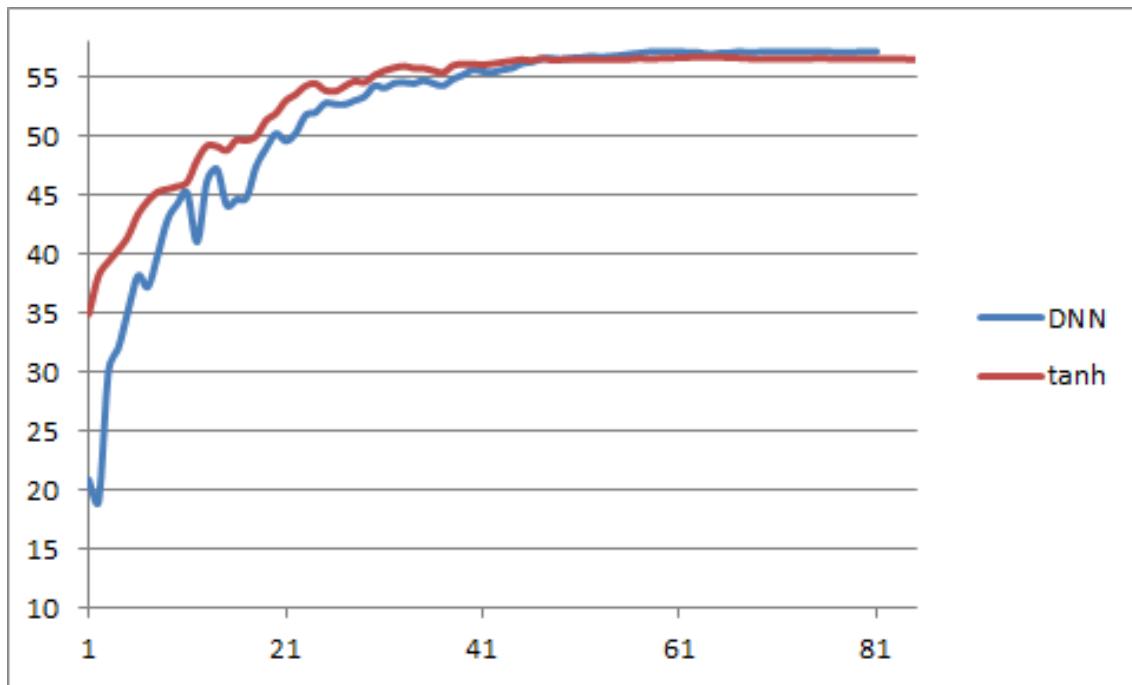
Pozycja 120 to pierwsza pozycja na której w rankingu pojawia się sieć oparta o tanh. Dowodzi to, że w tej sytuacji sieci ReLU uczą się wynikowo dużo lepiej, nawet o 1,2%. Zysk zależy oczywiście od charakteru danych, jednak widać prawidłowość, że sieci ReLU, gdzie w pierwszej warstwie jest relatywnie mało neuronów zachowują się gorzej. Kilka architektur ReLU nie było w stanie nauczyć się, skutkując sieciami z dokładnością rzędu 9%. Wszystkim sieciom tanh udało się nauczyć.

Eksperyment ten pokazuje, że głębokie sieci z 3 warstwami są lepsze od analogicznych dwuwarstwowych sieci, jeśli chodzi o spożytkowanie czasu na uczenie się. Można jednak co do tego eksperymentu postawić zarzut że nie były to sieci o jednakowej architekturze. Z definicji niemalże mająca mniej parametrów sieć klasyczna powinna uczyć się gorzej, co nie wykazuje zasadniczej przewagi uczenia głębokiego nad uczeniem klasycznym, jeśli zastosowana jest duża liczba parametrów.

3.6.2. CIFAR-10

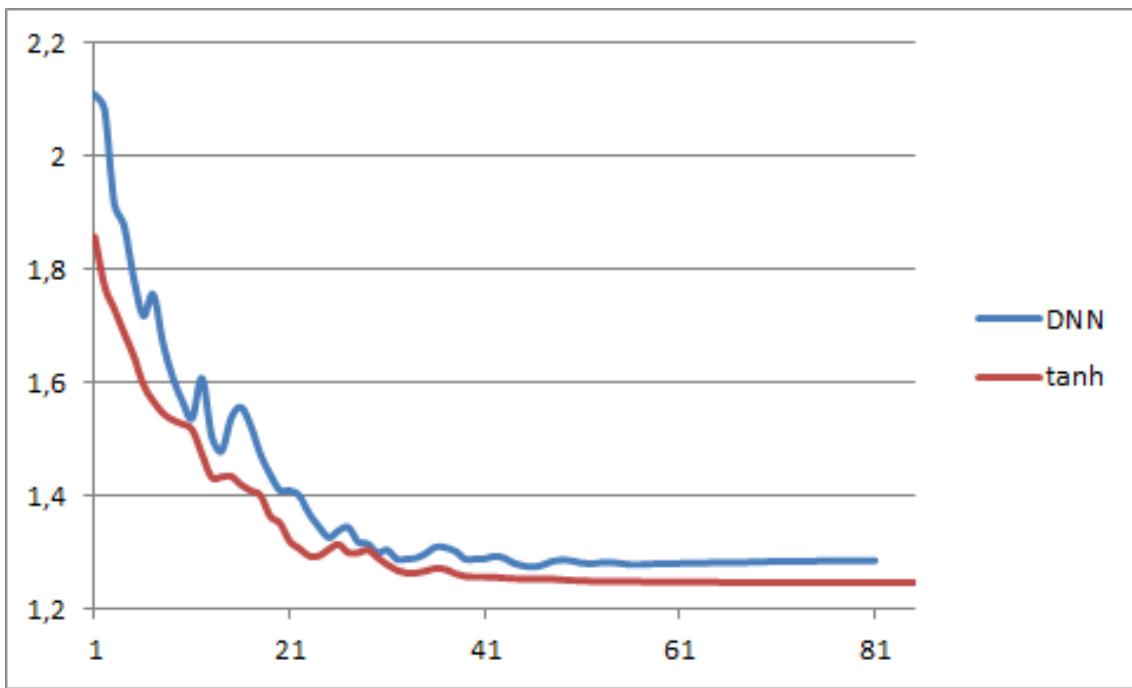
Podobną analizę co dla zbioru MNIST, wykonano również dla zbioru CIFAR-10. Zbiór CIFAR-10 to zbiór 60000 kolorowych obrazów o rozdzielczości 32x32 przedstawiające jeden z 10 rodzajów obiektów, takich jak samolot, samochód czy pies [23].

Analogicznie jak dla MNIST wykonano eksperyment również dla tego zbioru. Celem eksperymentu była identyfikacja różnic w uczeniu się sieci oraz ich wynikowej jakości na tym zbiorze. Nie jest to prosty zbiór danych, w praktyce nie wykorzystuje się go do badań nad perceptronami wielowarstwowymi i sieciami głębokimi DNN. Skrypt realizujący ten eksperyment znajduje się w katalogu *dnn_cifar*. Porównano więc dwie architektury - jedną uczoną za pomocą *tanh* i regularyzacji L2 oraz głęboką sieć DNN.



Rysunek 6: Jakość uczenia się sieci na problemie CIFAR-10 przez sieć DNN i klasyczną tanh

Sieć *tanh* miała architekturę 3072-3000-2000-1500-10. Sieć DNN miała architekturę 3072-3000-2000-2000-15000-10. Sporządzono wykresy pokazujące jakość (wykres 6) i wartość funkcji celu (wykres 7) w funkcji numeru epoki uczenia.



Rysunek 7: Wartość funkcji celu sieci na problemie CIFAR-10 przez sieć DNN i klasyczną tanh

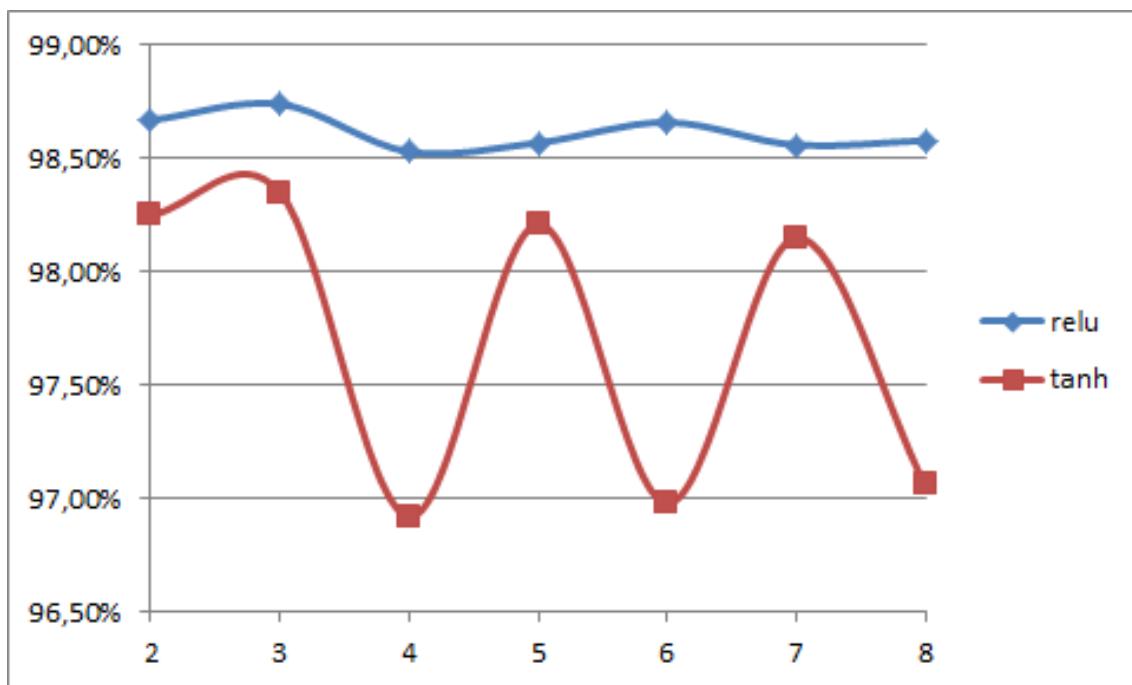
Sieć głęboka przejawia większą chaotyczność w uczeniu się. Zarówno jakość, jak i wartość funkcji celu podlega fluktuacyjnym zmianom, które dla porównania w sieci *tanh* są minimalne. Nie jest temu winny dropout, ponieważ nie został on zastosowany w sieci głębokiej tym razem, musi więc istnieć inny czynnik warunkujący takie zachowanie, taki jak ogólna trudność problemu. Jakość sieci głębokiej (57,21%) jest minimalnie wyższa od jakości sieci *tanh* (56,75%). W tym przypadku możliwe że zadanie jest zbyt trudne, aby sieć w pełni połączona była się go w stanie nauczyć przy użyciu zastosowanych metod uczenia (SGD + momentum). Zadanie samo z siebie wydaje się być trudne dla w pełni połączonych sieci, ponieważ inne klasy sieci głębokich potrafią osiągać nawet 20% lepsze wyniki od zaprezentowanych w tej części (więcej o tym w 3.10).

3.7. ReLU i tanh - głębia

Poprzedni eksperyment (3.6) wykazał że głębokie sieci z regularyzacją ReLU uczą się lepiej od płytowych sieci klasycznych. Wniosek ten nie po-

winięń dziwić, wszak głęboka sieć będzie w stanie wykorzystać większą ilość parametrów. Aby zademonstrować potrzebę korzystania z metod uczenia głębokiego zdecydowano się więc przeprowadzić eksperyment, który porównałby identyczne architekturowo sieci uczenia głębokiego i klasycznego, aby ustalić przy jakiej głębokości sieci należy korzystać z uczenia głębokiego.

Wytrenowany zostanie szereg coraz głębszych sieci o identycznej architekturze, zarówno uczenia klasycznego ($tanh$, $L2=0,0002$) jak i ReLU (dropout, $p=0,5$). Jakość tych sieci zostanie porównana. Jeśli teza, że metody uczenia głębokiego lepiej sprawują się przy sieciach o wielu warstwach od sieci klasycznych, jakość sieci klasycznej powinna spadać w funkcji głębokości sieci. Każda architektura po 3 razy. Przy głębszych sieciach nieznacznie modyfikowano współczynniki uczenia, tak aby uzyskać zbieżność. Przy porównywalnych sieciach, wyniki dokładności w funkcji głębokości sieci przedstawia wykres 8.



Rysunek 8: Zależność jakości sieci od ilości warstw

Problemem było ustalenie warunków zakończenia sieci. Intuicyjne wydaje się, że większa sieć będzie potrzebowała większej ilości iteracji, aby nauczyć się najlepiej jak potrafi. Z tego powodu też, do kontroli procesu

uczenia wykorzystano heurystykę uczenia się. Przerywała ona uczenie, gdy jakość sieci na zbiorze danych walidacyjnym nie poprawiała się przez 5 iteracji pod rząd.

Wyniki nie przedstawiają w jasny sposób przewagi sieci głębokich nad sieciami. Wydaje się, że rozwiązywany problem jest na tyle prosty, aby nawet nieskomplikowany klasyfikator dał sobie z nim radę w zadowalający sposób (co zresztą widać po wynikach klasyfikatorów [27]), uwidacznia jednak pewną cechę sieci klasycznych. Dodatkowe warstwy nie wspomagają uczenia się sieci, a są nawet je w stanie zaburzyć. Pozwala to poszerzać możliwości sieci głębokich poprzez dodawanie kolejnych warstw (co na problemie MNIST, ze względu na jego prostotę, nie polepszało wyników). Sieci klasyczne ustępują tu sieciom głębokim.

Nie da się nie dostrzec własności oscylacyjnej jakości sieci klasycznej na tym przykładzie. Aby rozwinać ten temat, dokonano również obliczeń dla klasycznej sieci o 9 i 10 warstwach. Wyniki zaprezentowano na wykresie 9 i wydają się one potwierdzać spostrzeżenie świadczące o tym, że dodawanie kolejnych warstw do sieci klasycznej nie zwiększa jej dokładności, a może być źródłem problemów. Geneza tego zjawiska jest nieznana.



Rysunek 9: Zależność jakości sieci tanh od ilości warstw

Tabela 7: Wyniki wybranych architektur sieci splotowych

Warstwa splotowa	Jakość sieci
32 filtry 5x5 + 2x2 MP + 50 filtrów 5x5 + 2x2 MP	99.01%
32 filtry 7x7 + 2x2 MP + 50 filtrów 4x4 + 2x2 MP	98.92%
32 filtry 9x9 + 2x2 MP + 50 filtrów 5x5 + 2x2 MP	99.01%
32 filtry 3x3 + 2x2 MP + 50 filtrów 4x4 + 2x2 MP	98.85%
60 filtrów 3x3 + 2x2 MP + 50 filtrów 4x4 + 2x2 MP	98.93%
60 filtrów 5x5 + 2x2 MP + 50 filtrów 5x5 + 2x2 MP	98.89%
50 filtrów 5x5 + 2x2 MP + 50 filtrów 5x5 + 2x2 MP	99.09%

3.8. Architektura sieci splotowych

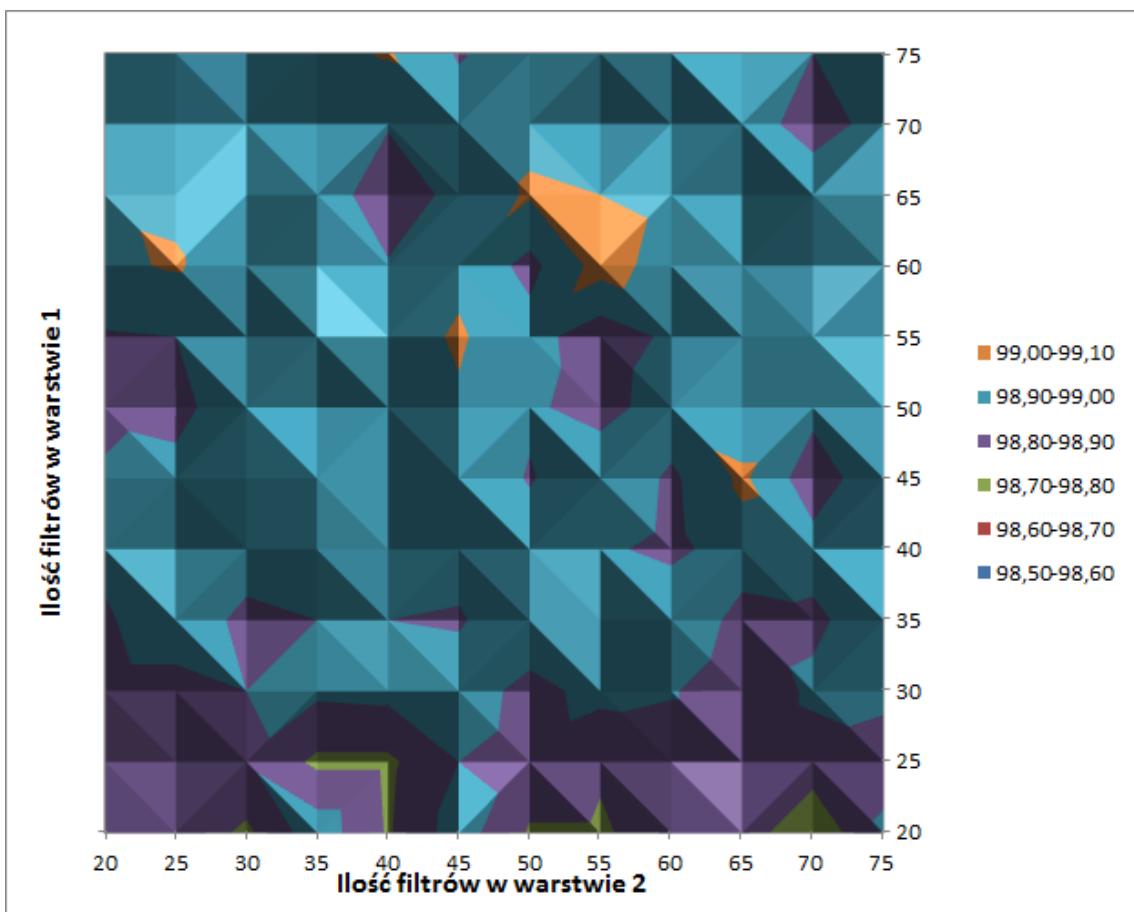
W przypadku sieci splotowej architektura ma dwojaki charakter. Pierwszą częścią sieci splotowych są jądra, inaczej nazywane też filtrami, splotowymi. Jest ich określona ilość, ale przede wszystkim mają one określony i skończony rozmiar. Dobór zarówno ilości jak i rozmiaru filtrów jest krytyczny dla jakości sieci splotowej. Ponadto, dobierać można jeszcze regularyzację, funkcje aktywacji, itp. Liczba hiperparametrów w porównaniu do zwykłej sieci głębokiej DNN jest więc nieco większa, należy więc przeznaczyć czas na zastanowienie się jakimi hiperparametrami należy sieć uczyć.

Ze względu na długi czas nauki pojedynczej sieci splotowej na zbiorze danych CIFAR-10 (rzędu kilku godzin na GPU) do celów porównania sieci użyto zbioru MNIST. Zbudowano kilka architektur sieci splotowych, różniących się tylko rozmiarami filtrów warstw splotowych. Analogiczną parametryzację przeprowadzono dla warstwy wyższej, wybierając jako warstwę niższą najlepszą z wynikowych. Parametry warstw, wraz z wynikami zestawiono w tabeli 7. Skrypty realizujące eksperyment znajdują się w katalogu *cnn/architektura*. W warstwie wyższej zastosowano 500 neuronów *tanh* i 10 neuronów *softmax*.

Nie zauważono znacznych różnic między filtrami różnych rozmiarów, natomiast dość dobrym predyktorem jakości sieci była liczba filtrów. Róż-

nica polegająca na zastosowaniu 50 filtrów w pierwszej warstwie zamiast 32 poskutkowała podniesieniem jakości sieci do 99,09%.

Zdecydowano się przeszukać dwuwymiarową przestrzeń rozmiarów filtrów w pierwszej i drugiej warstwie splotowej, aby odszukać sieć o najlepszej jakości. Ustawiono więc eksperyment, generujący sieci o różnej liczbie filtrów z pewnego przedziału. W warstwach pełni połączonej ustaloną 800 neuronów *tanh* i 10 neuronów wyjściowych *softmax*. Wyniki przedstawiono na wykresie 10. Eksperyment dla każdej liczby warstw powtórzony był 3 razy, co dało ok. 11 dni obliczeń.



Rysunek 10: Wykres jakości sieci od ilości filtrów splotowych w warstwie 1 i 2

Oprócz klastra "lepiej niż 99%" przy 55/45, 65/50 i 60/55 (warstwa 1/warstwa 2) ilość filtrów wydaje się nie mieć bezpośredniego wpływu na jakość sieci, nawet przy dość małej ilości filtrów. Stanowi to argument na rzecz poszukiwania lepszych sieci splotowych w inny sposób niż manipulowanie ilością filtrów czy ich rozmiarem. Można to zrobić poprzez

zmianę innych hiperparametrów czy metod uczenia się, gdyż zastosowana tu metoda - zejście po gradiencie z momentum - jest metodą uznawaną za leciwą i coraz rzadziej stosowaną na rzecz wydajniejszych, jak RMSProp czy ADAM.

3.9. Sztuczny resampling danych treningowych

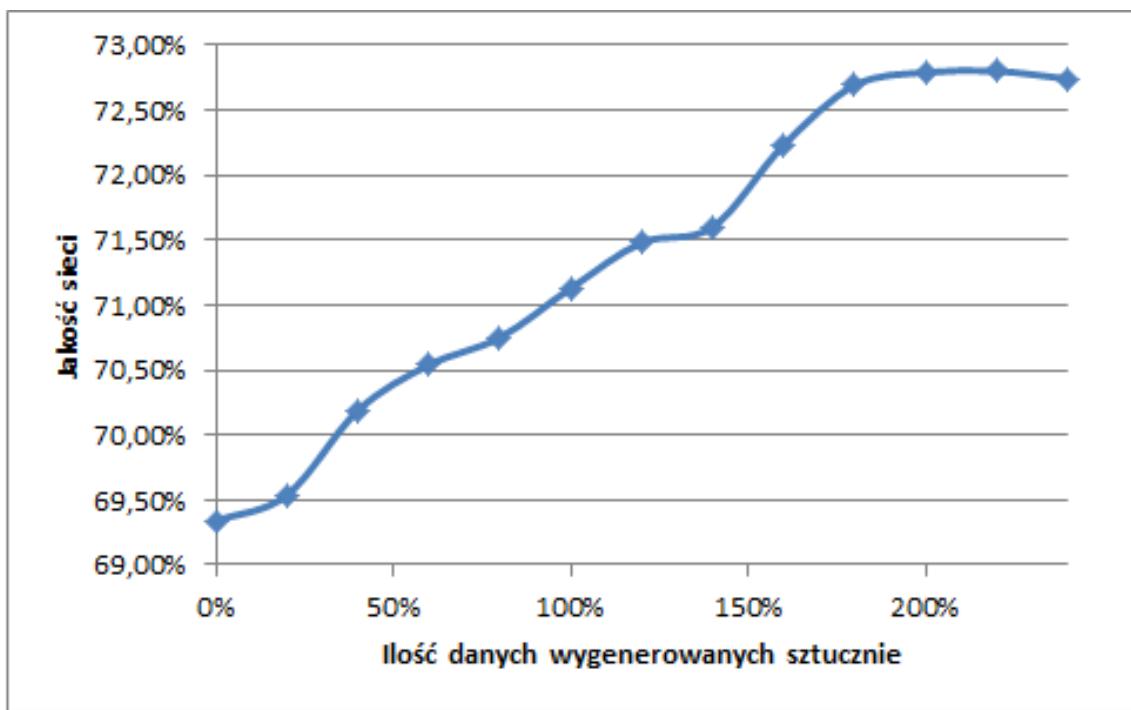
Większość sieci charakteryzujących się najlepszą jakością dla rozpatrywanych tu problemów (tj. MNIST, CIFAR-10) stosuje sztuczne poszerzanie zbioru uczącego poprzez translacje czy transformacje afinczne [5]. Polega ono na ładowaniu do zbioru uczącego również przekształconych wersji danych. W ten sposób, znany w literaturze jako data augmentation, zwiększa się liczebność zbioru uczącego. W sekcji tej zbadano wpływ sztucznego powiększenia zbioru danych treningowych na jakość wynikowej sieci na zbiorze CIFAR-10. Skrypt realizujący eksperyment znajduje się w *cnn/afine.py*.

Zbiór danych treningowych (wynoszący 50000 przykładów) powiększono o różną liczbę przykładów, losowo wybierając jeden przykład z zbioru danych treningowego i (operacja losowa):

- przesuwając go o losowy wektor, max. 4 piksele w pionie i poziomie
- odbijając go w poziomie

Sporządzono wykres ilustrujący jakość sieci w funkcji tego, o ile powiększono zbiór uczący w odniesieniu do ilości dostarczonej (50000). W tej sytuacji 100% danych wygenerowanych sztucznie będzie oznaczać zbiór treningowy o łącznej liczności 100000. Wyniki przedstawiono na wykresie 11. Każdą próbę powtórzono 3 razy i wybrano najlepszy wynik.

Najlepszy uzyskany wynik to 72,80%, przy wygenerowaniu dodatkowych 220% sztucznych danych, co oznaczało zbiór uczący o liczbie 160000 zamiast bazowych 50000, przy których wynik wyniósł 69,34%. Wyniki wskazują na to, że zastosowanie sztucznego próbkowania zbioru uczącego może przynieść dobre efekty, nie mniej jednak istnieje pewna granica do której klasyfikator można w ten sposób poprawić. Zjawisko to



Rysunek 11: Wykres jakości sieci splotowej od udziału sztucznych danych

jest wykorzystywane w praktyce, zostało udokumentowane chociażby przy uczeniu sieci głębokich rozpoznających dźwięk [8].

Zjawisko to nasuwa pewien ciekawy wniosek. Mianowicie istnieje pewna granica do której pomóc mogą lepsze algorytmy klasyfikujące. Wydaje się jednak, że nic nie jest w stanie pokonać zwyczajnego uczenia klasyfikatora na większej ilości danych treningowych. Samo zautomatyzowane pozyskiwanie danych treningowych jest zagadnieniem ciekawym, jednak nie mieszczącym się w zakresie tej pracy.

3.10. Porównanie sieci splotowej i dużej sieci klasycznej

Porównano jak sieć splotowa radzi sobie z zbiorem danych MNIST w porównaniu do klasycznej sieci neuronowej opartej o nielinowość *tanh*. Wykonano również wykres jakości sieci w funkcji czasu (licząc czas w sekundach oraz epokach, po jednym wykresie). Analogiczny eksperyment przeprowadzono dla zbioru CIFAR-10. Skrypt realizujący eksperyment znajduje się w katalogu *splot_klasyk*.

Pierwszy eksperyment przeprowadzono na zbiorze MNIST. Uczenie każ-

dej sieci powtarzano 3 razy. Sieć tanh o architekturze 768-1200-1400-1200-10 osiągnęła jakość 97,8% po 224 sekundach uczenia. Sieć splotowa o architekturze:

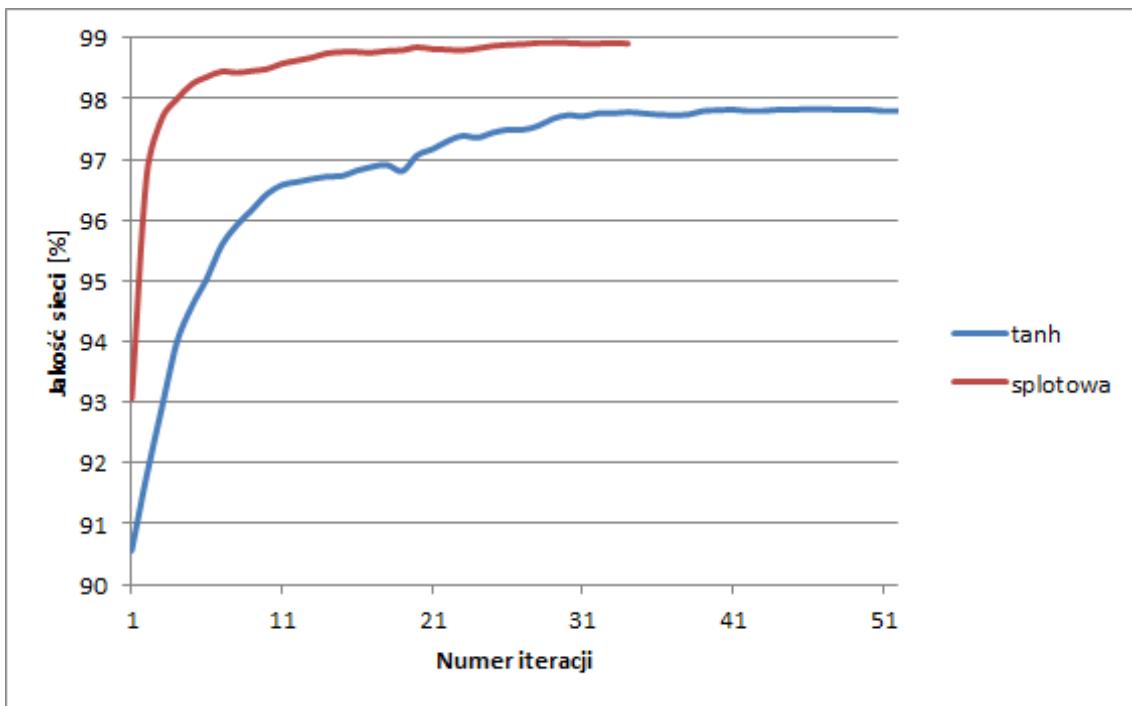
1. 50 filtrów 5x5
2. Max-pooling 2x2
3. 50 filtrów 5x5
4. Max-pooling 2x2
5. 900 neuronów *tanh*
6. 10 neuronów *softmax*

osiągnęła jakość 98,92% po 1814 sekundach nauki. Sporządzono dwa wykresy. Wykres 12 prezentuje jakość sieci w funkcji epoki uczenia, zaś wykres 13 prezentuje jakość sieci w funkcji czasu. Ze względu na duże dysproporcje czasu (uczenie sieci splotowej trwało ok. 10 razy dłużej) zdecydowano się, aby na wykresie Y przedstawić czas na osi logarytmicznej.

MNIST jest stosunkowo prostym zbiorem danych. Nie dziwi więc, że sieć splotowa była w stanie od pierwszej iteracji nauczyć się go lepiej. Czyni to sieć splotową uniwersalnie lepszą w tych warunkach, mimo około dziesięciokrotnie czasu dłuższego uczenia sieci splotowej. Prosta na początku wykresu jakości sieci splotowej w funkcji czasu spowodowany jest długim czasem epoki - aż 40 sekund, w porównaniu do długości epoki sieci tanh - ok. 9 sekund. Gdyby zadaniem było nauczyć sieć lepiej w czasu krótszym niż 1 epoka sieci splotowej, to opłacalne byłoby uczenie sieci tanh. Dla porównania, sieć *tanh* miała 4293,6 tys. parametrów, a sieć splotowa - 792,75 tys. parametrów. To aż 5,4 raza mniej się w sieci *tanh*.

W przypadku eksperymentu CIFAR zastosowano sieć neuronową tanh o architekturze 3072-2000-2200-1200-10 oraz sieć splotową o architekturze:

1. 60 filtrów 5x5

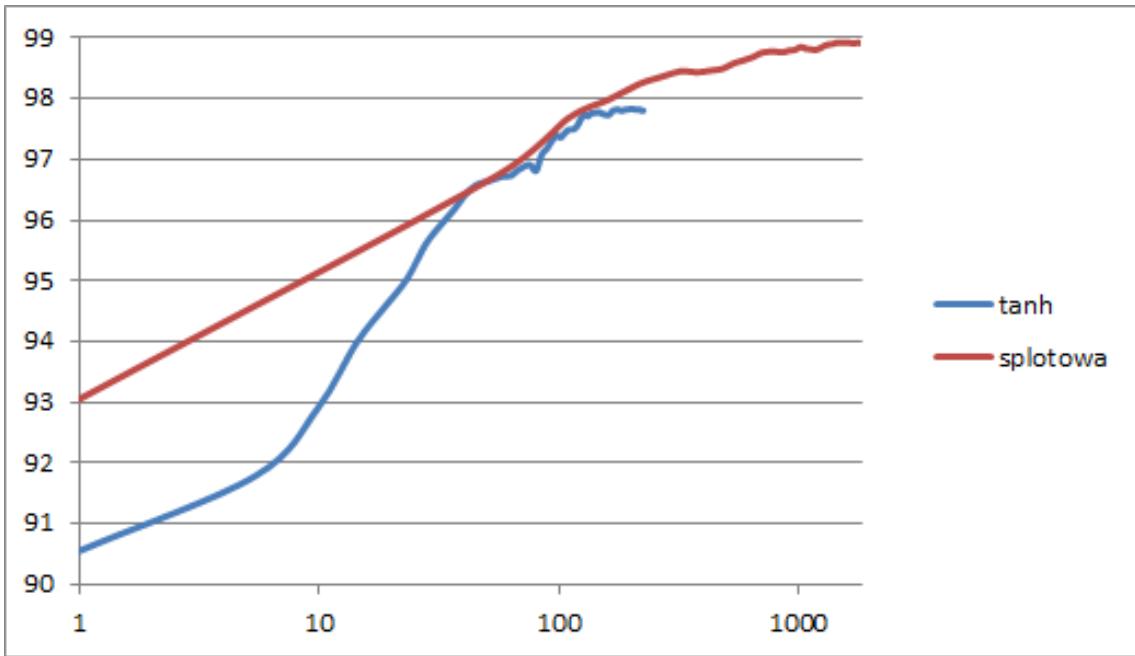


Rysunek 12: Porównanie jakości sieci tanh i splotowej na problemie MNIST w funkcji numeru epoki

2. Max-pooling 2x2
3. 60 filtrów 5x5
4. Max-pooling 2x2
5. 1200 neuronów *tanh*
6. 10 neuronów *softmax*

Ze względu na stopień skomplikowania problemu CIFAR dodatkowo przedstawiono w tym problemie również sieć neuronową DNN o architekturze 3072-2000-2200-1200-10 z funkcją aktywacji ReLU, w celach porównawczych (architektura taka jak *tanh*).

Uczenie każdej sieci powtórzono 3 razy. Sieć typu tanh osiągnęła najlepszy wynik 50,42% po 63 epokach. Sieć splotowa osiągnęła najlepszy wynik 69,34% po 70 epokach. Sieć głęboka DNN osiągnęła wynik 56,49% po 68 epokach. Wykres 14 prezentuje jakości sieci w funkcji epoki, zaś wykres YB prezentuje jakości sieci w funkcji czasu.

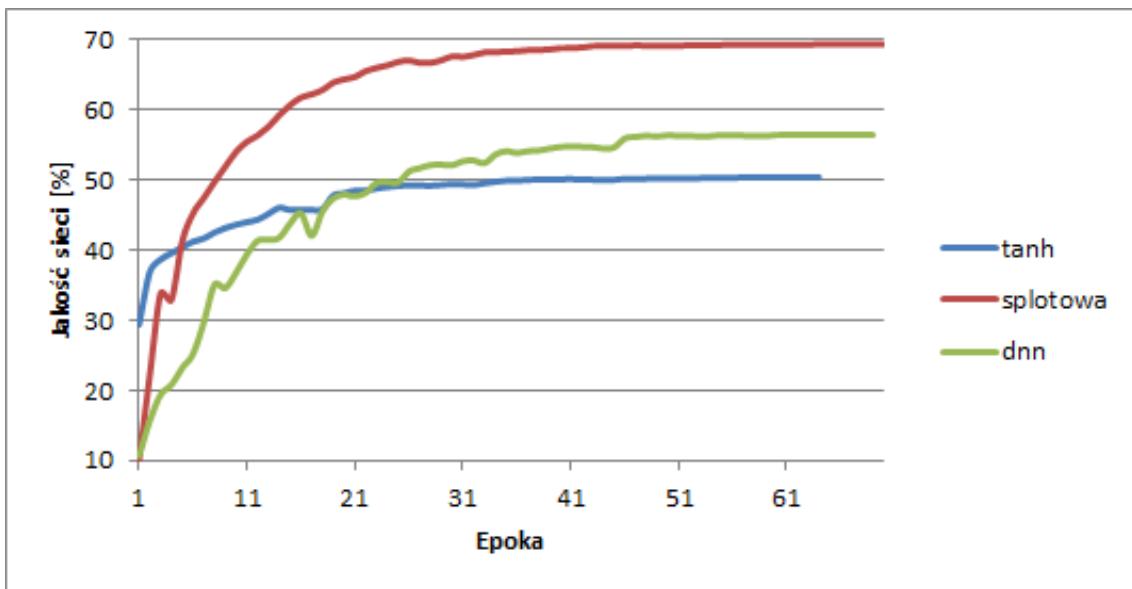


Rysunek 13: Porównanie jakości sieci tanh i splotowej na problemie MNIST w funkcji upływu czasu

Można by twierdzić, że w przypadku trudniejszych zestawów danych w krótkiej perspektywie czasowej uczenia lepsze są względnie płytkie sieci klasyczne. W przypadku prostego MNIST nie zostało to zaobserwowane, ale widoczne jest w sieci CIFAR.

Uczenie sieci splotowej trwało tylko 6 razy dłużej niż sieci tanh. Tym razem uczenie sieci splotowej zajęło większą liczbę epok. Być może im bardziej skomplikowany model, tym prostsze staje się zrobienie prostej sieci płytkiej, ale większy zwrot w czasie uczenia daje wykonanie sieci splotowej. Na tym zestawie widać też, że sieć splotowa jest dużo potężniejszym klasyfikatorem, uzyskując niemalże 20% poprawy w jakości przy 10-klasowym zbiorze CIFAR. Dla porównania, sieć *tanh* i DNN miały 13196 tys. parametrów, a sieć splotowa - 1906,5 tys. parametrów. To aż 6,9 raza mniej się w sieci *tanh*. Z całą pewnością jest tak między innymi dlatego, że sieć splotowa nie jest w pełni połączona. Sieć splotowa uczyła się na początku podobnie jak sieć splotowa, jednak z upływem czasu zbliżyła się do jakości sieci *tanh*, a nawet ją przerosła o ok. 6,5%.

Widac, że w obu przypadkach sieć splotowa zapewniała lepszą jakość



Rysunek 14: Jakość uczenia sieci CIFAR w funkcji numeru epoki

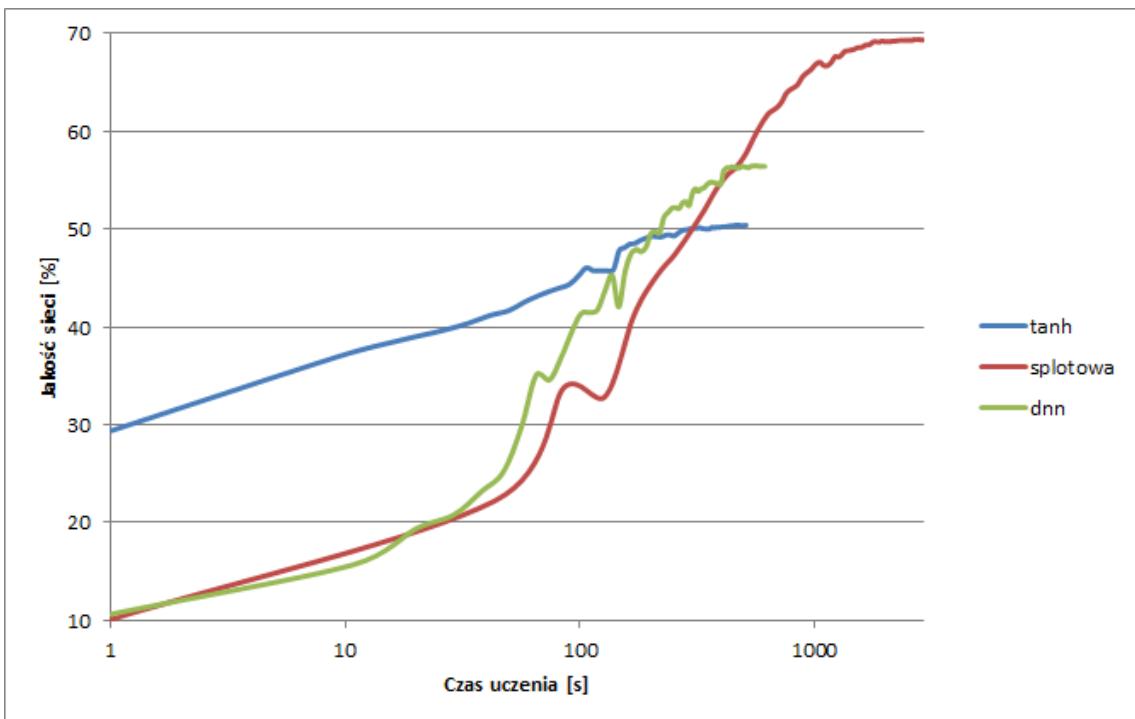
klasyfikacji przy mniejszym modelu. Jest to cenne ze względu na minimalizację zużycia pamięci w urządzeniach na których rozwiązanie będzie wdrażane. Sieć uczona byłaby na dużych klastrach obliczeniowych, zaś wdrożona byłaby na mniejszych urządzeniach (np. telefony komórkowe).

3.11. Czego uczy się sieć splotowa

Ponieważ pierwsza warstwa sieci splotowej (i w ogóle jakiekolwiek innej sieci neuronowej) aplikowana jest bezpośrednio na analizowane dane, w tym przypadku na obraz, powinny mieć one zrozumiałą interpretację graficzną. Filtr taki jest liczony dla każdego fragmentu obrazu, rozsądnie byłoby więc wnioskować że reaguje on na fragment obrazu mający charakter kodowany przez obraz. Rozsądnie więc byłoby nauczyć sieć splotową pewnego zbioru danych, a następnie przejrzeć filtry pierwszej warstwy sieci.

Do analizy zbioru MNIST zaprojektowano następującą sieć splotową:

1. Warstwa splotowa, 32 filtry o rozdzielczości 7x7
2. Warstwa max-pooling 2x2
3. Warstwa splotowa, 50 filtrów o rozdzielczości 5x5



Rysunek 15: Jakość sieci CIFAR w funkcji czasu ich uczenia

4. Warstwa perceptronu, 300 jednostek ukrytych

5. Warstwa perceptronu 10 neuronów *softmax*

Filtry w pierwszej warstwie dobrano ku maksymalnej wielkości tak, aby można było skutecznie zinterpretować ich wizualizację. Temu kryterium podporządkowano projekt sieci - nie ma ona mieć maksymalnej dokładności, a jedynie ma pozwolić odkryć, czego taka sieć właściwie się uczy.

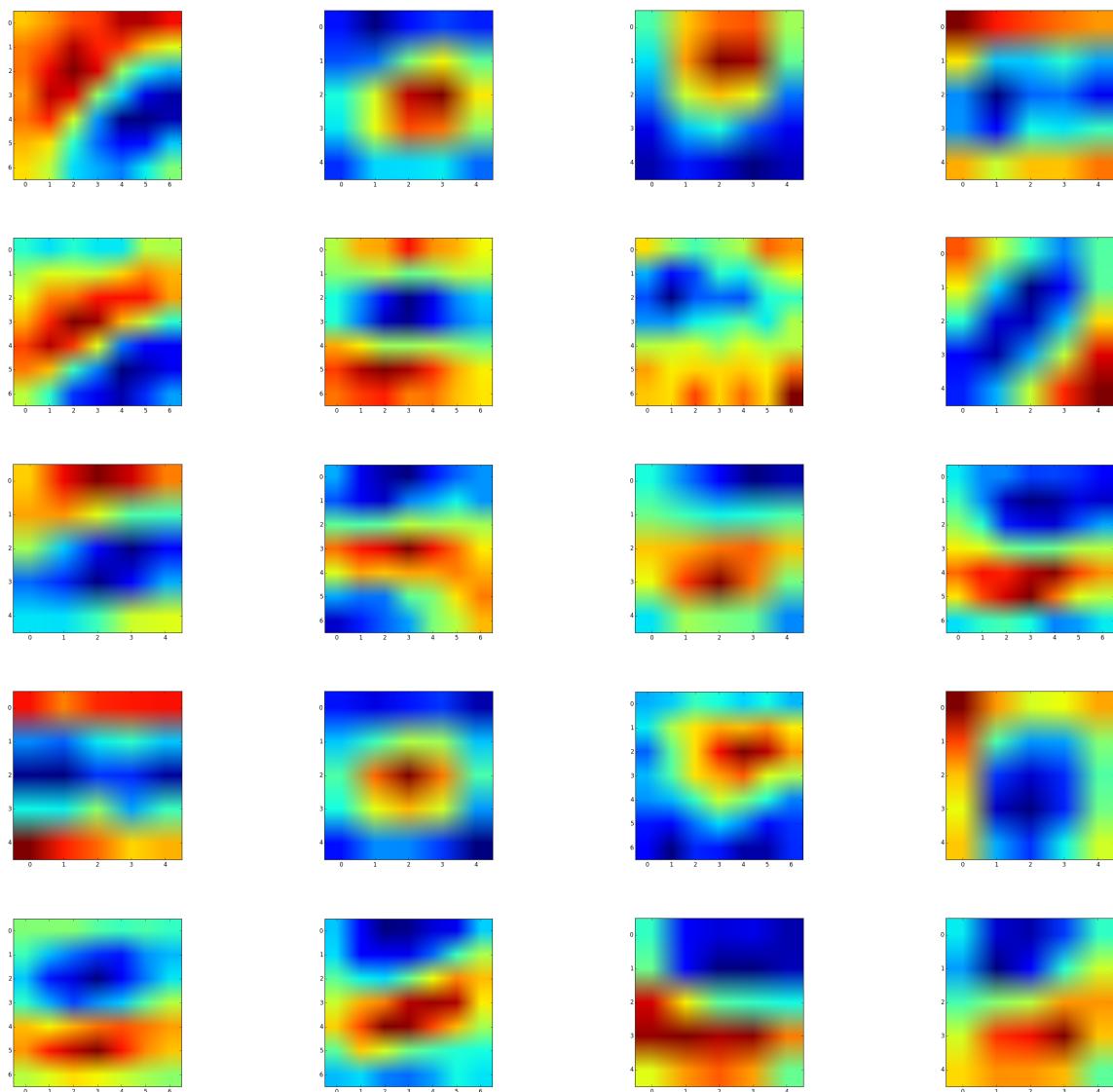
Po szeregu iteracji, które doprowadziły sieć do współczynnika dokładności rzędu 98.5% pobrano i zwizualizowano filtry pierwszej warstwy. Zostały one przedstawione w tabeli 8.

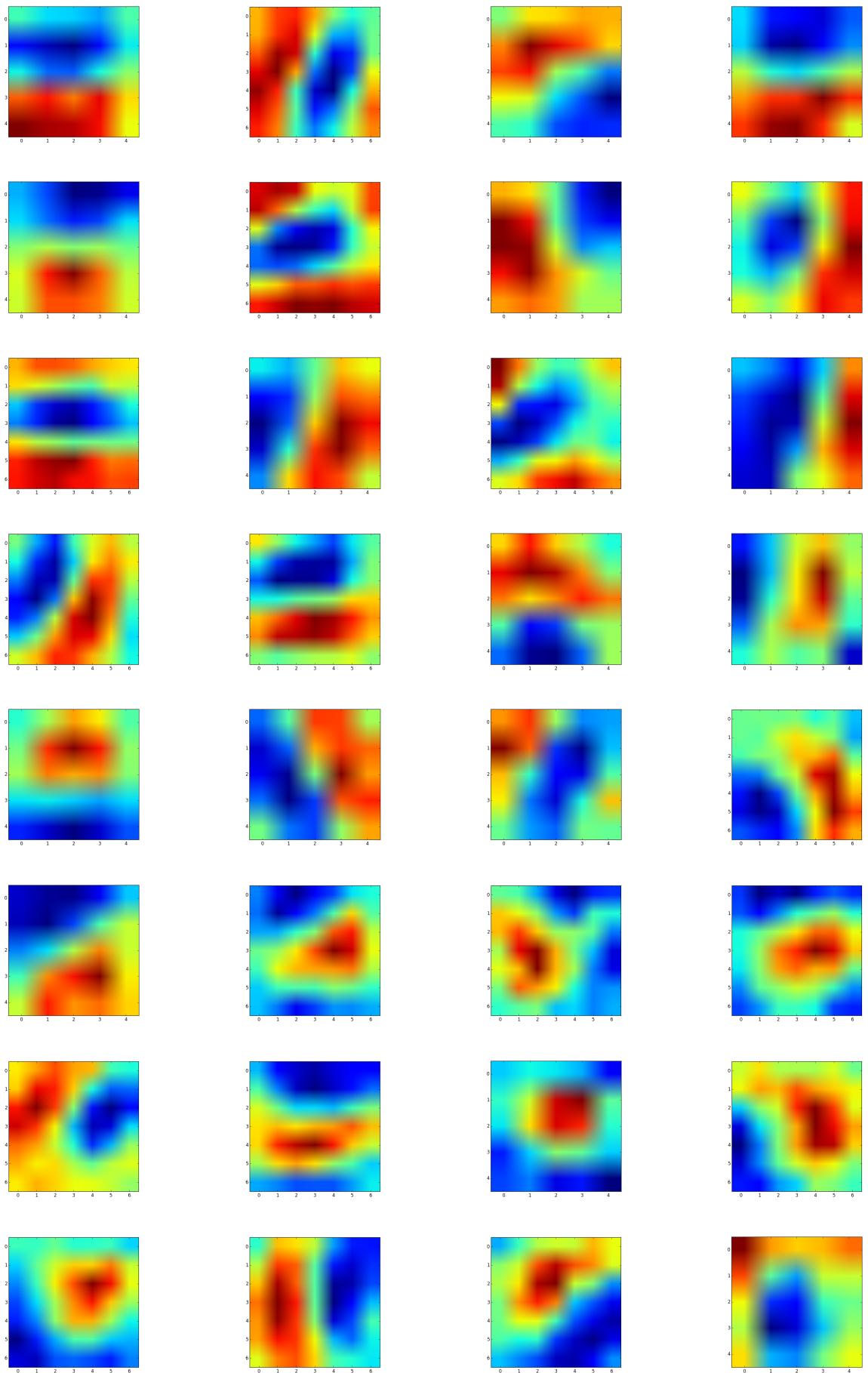
Obrazy uzyskiwane tutaj, w kontekście wiedzy czego uczą się neurony zwierzęce, dość łatwo zinterpretować. Są to po prostu fragmenty obrazów - krzywe, załamania, krawędzie. Wiedząc, co rozpoznawane jest w pierwszej warstwie, można poczynić pewne hipotezy na temat tego, co może rozpoznawać druga warstwa.

Jeśli pierwsza rozpoznaje charakterystyczne fragmenty takie, jak usta-

wione pod odpowiednim kątem krzywe, pętle, czy linie, druga warstwa zapewne sprawdza, czy są one ułożone w odpowiednich miejscach. Taki neuron sprawdzający liczbę zero mógłby reagować gdy linie i krzywe byłyby w odpowiednich miejscach, pasujących tej cyfrze. Być może tak właśnie wygląda postrzeganie obrazów u zwierząt - każda wyższa warstwa rozpoznaje coraz bardziej globalne koncepty, idąc "od szczegółu do ogólnego". Taka właśnie teoria [9] stanowi obecny konsensus środowisk naukowych. Zgodnie z nią okazuje się więc, że rozwój postrzegania obrazów u zwierząt jest zjawiskiem o charakterze emergencyjnym, odbywającym się w reakcji na środowisko.

Niestety, interpretacja graficzna filtrów drugiej warstwy nie jest już tak oczywista i wykracza poza ramy tej pracy.





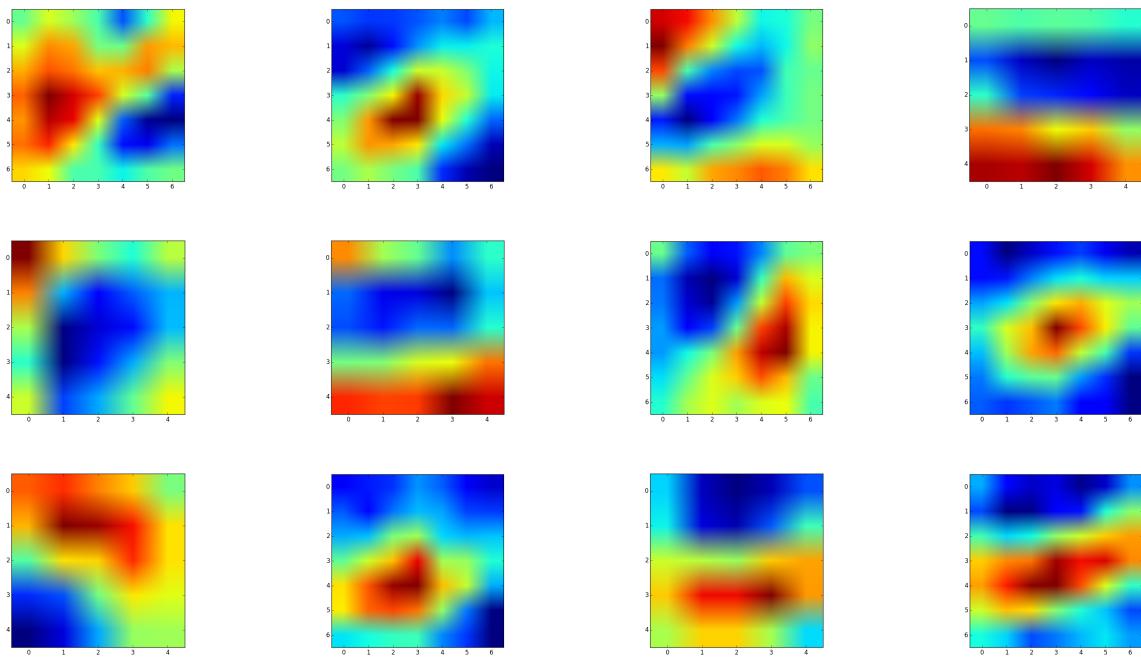


Tabela 8: Wizualizacje filtrów pierwszej warstwy sieci splotowej uczonej na zbiorze MNIST

4. Zastosowanie istniejących bibliotek uczenia głębokiego

Istnieje szereg bibliotek wspomagających uczenie głębokie, umożliwiających prowadzenie obliczeń na GPU i CPU. Z praktycznego punktu widzenia wybranie popularnej biblioteki jest dobrym rozwiązaniem, minimalizuje bowiem szansę wystąpienia błędów i umożliwia dostęp do większej liczby typowych narzędzi. Skorzystanie z gotowej implementacji zwalnia również programistę z obowiązku implementowania rozwiązań, co w przypadku złożonych technologii, jak CUDA, jest kłopotliwe.

Przykładem, który zostanie zaprezentowany w tym rozdziale, jest TensorFlow. Napisany został pierwotnie przez naukowców i inżynierów pracujących w zespole Google Brain, obecnie jest projektem open source. Zostanie on tu zaprezentowany ze względu na popularność, oraz dostępność dużej liczby poradników i tutoriali w sieci Internet, dzięki czemu może stanowić dobry punkt wyjścia dla początkujących.

4.1. Instalacja TensorFlow

Do komplikacji TensorFlow wykorzystuje system komplikacji Bazel autorstwa Google. Bazel jest narzędziem o rozległych korzeniach uniksowych, tak więc niemożliwa jest jego użycie na systemie Windows, choć trwają prace nad jego adaptacją [15]. Do instalacji można użyć maszyny wirtualnej z 64-bitowym systemem Linux. Ten przykład będzie korzystał z maszyny wirtualnej Debian GNU/Linux 8, tak więc dalsza część tego rozdziału zakłada obecność zainstalowanego systemu Debian GNU/Linux 64-bit, ponieważ instalacja tego systemu wykracza poza zakres tej pracy.

Aby zainstalować TensorFlow, należy dysponować zainstalowanym Pythonem 2.7. Dodatkowo, należy dysponować również nagłówkami C Pythona, menedżerem pakietów Pythona oraz kompilatorem C. Można je zainstalować, wydając polecenie:

```
sudo apt-get install python python-dev gcc python-setuptools
```

Następnie należy zaktualizować menedżer pakietów Pythona, wydając polecenie:

```
sudo easy_install pip
```

Następnie można już zainstalować TensorFlow

```
sudo pip install -upgrade
```

```
https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.8.0-  
cp27-none-linux_x86_64.whl
```

Polecenie to pobierze i zainstaluje bibliotekę TensorFlow w wersji 0.8. Biblioteka powinna być teraz dostępna z poziomu interpretera Pythona.

4.2. Głębokie sieci neuronowe

W części tej przedstawione zostanie działanie TensorFlow na przykładzie zbioru danych MNIST i głębokiej sieci neuronowej (DNN).

Pierwszym krokiem po uruchomieniu interpretera Pythona jest pobranie danych treningowych, walidacyjnych i weryfikacyjnych. TensorFlow umożliwia pobieranie typowych zbiorów danych w prosty sposób. Aby pobrać zbiór danych MNIST, należy wykonać komendy Pythona:

```
from tensorflow.examples.tutorials.mnist import input_data  
data = input_data.read_data_sets('MNIST_data', one_hot=True)
```

Komenda ta pobiera z Internetu zbiór danych MNIST. Może on zostać użyty w uczeniu, ale zanim TensorFlow zacznie obliczenia, należy go zainicjować. TensorFlow do właściwych wykorzystuje bibliotekę C++, z której połączenie nosi nazwę *sesji*. Z założenia wymagane jest określenie całej struktury sieci przed uruchomieniem, jednak dostępna jest przydatna badaczom funkcja *sesji interaktywnej*, która pozwala mieszać rozkazy definiujące obliczenia i przeprowadzające ją. Aby uruchomić interaktywną sesję TensorFlow, należy wpisać:

```
import tensorflow as tf  
session = tf.InteractiveSession()
```

Kolejnym etapem jest utworzenie symbolicznych zmiennych, które będą przechowywać zbiór wektorów wejściowych (w przypadku MNIST 784-

wymiarowy), oraz wyjściowych (10-wymiarowe jako prawdopodobieństwo dla każdej cyfry). Definiuje się je za pomocą komend:

```
_x = tf.placeholder(tf.float32, shape=[None, 784])  
_y = tf.placeholder(tf.float32, shape=[None, 10])
```

Wyrażenie **None** w parametrze kształtu oznacza, że ten wymiar ma pozostać nieokreślony. **shape=[None, 784]** definiuje macierz o nieokreślonej ilości wierszy oraz 784 kolumnach. Konkretna ilość wierszy zależeć będzie od rozmiaru wsadu, ilości weryfikowanych zmiennych, itd. Nie może więc być określona z góry. Argument **shape** jest tutaj opcjonalny, jednak zastosowanie go pozwoli TensorFlow łapać proste pomyłki już na tym etapie.

W przykładzie tym pokazane zostanie, jak zbudować dwuwarstwową sieć głęboką. Idea budowania głębszych sieci jest niezwykle podobna, sprowadza się do odpowiedniego powtórzenia komend dodawania warstw. Powiedzmy że będzie to sieć o architekturze 768-1000-10. Na samym początku należy zdefiniować wagi i biasy pierwszej warstwy:

```
W1 = tf.Variable(tf.truncated_normal([784, 1000]))  
B1 = tf.Variable(tf.zeros([1000]))
```

W pierwszej warstwie - warstwie wejściowej - odwzorowujemy 784 wejść na 1000 neuronów w kolejnej warstwie, stąd odpowiednie rozmiary macierzy przechowujących parametry tej warstwy. Komenda **truncated_normal** inicjuje wagi wartościami próbkowanymi z rozkładu normalnego, z tym że jeśli zostaną wybrane wartości spoza przedziału $[-1; 1]$, są wybierane ponownie. Tak naprawdę zdefiniować musimy szereg zmiennych, które - wraz z kodem niezbędnym do ich zdefiniowania - opisano w tabeli 9.

Zanim zmienne zostaną wykorzystane, muszą zostać zainicjowane. Zmienne TensorFlow zdefiniowane w interpreterze do tej pory można zainicjować komendą:

```
session.run(tf.initialize_all_variables())
```

Kolejnym etapem będzie opisanie zależności matematycznych funkcjonujących w sieci. Niezbędne będzie:

Tabela 9: Zmienne sieci głębskiej dla problemu MNIST w TensorFlow

Kod	Zmienna
W1 = tf.Variable(tf.truncated_normal([784, 1000]))	Wagi pierwszej warstwy
B1 = tf.Variable(tf.zeros([1000]))	Bias pierwszej warstwy
W2 = tf.Variable(tf.truncated_normal([1000, 10]))	Wagi drugiej warstwy
B2 = tf.Variable(tf.zeros([10]))	Bias drugiej warstwy
P_1 = tf.placeholder(tf.float32)	Parametr dropout pierwszej warstwy
_x = tf.placeholder(tf.float32, shape=[None, 768])	Zmienne przechowujące wejście sieci
_y = tf.placeholder(tf.float32, shape=[None, 10])	Zmienne przechowujące wyjście sieci
LR = tf.placeholder(tf.float32)	Wartość współczynnika uczenia

1. Opisanie sposobu działania pierwszej warstwy
2. Opisanie regularyzacji dropout pierwszej warstwy
3. Opisanie sposobu działania drugiej warstwy, jednocześnie opisując predykcję sieci
4. Opisanie funkcji celu sieci

Wykonać można to następująco:

1. $O1 = \text{tf.nn.relu}(\text{tf.matmul}(_x, W1) + B1)$
2. $DO1 = \text{tf.nn.dropout}(O1, P_1)$
3. $O2 = \text{tf.nn.softmax}(\text{tf.matmul}(DO1, W2) + B2)$
4. $\text{cost} = \text{tf.reduce_mean}(-\text{tf.reduce_sum}(_y * \text{tf.log}(O2 + 1E-12), \text{reduction_indices}=[1]))$

Operację dodawania 1E-12 zawarto, aby system nie musiał nigdy liczyć logarytmu z wartości 0¹. Jest to wartość nieokreślona, tak więc algorytm nie zadziałałby poprawnie. Ponieważ opisano już działanie sieci, TensorFlow może automatycznie policzyć pochodne w celu odnalezienia gradientu. Biblioteka zawiera szereg wbudowanych algorytmów uczących.

¹Nie ma potrzeby martwić się ujemnymi wartościami O2, albowiem takich funkcja softmax nigdy nie generuje.

W tym przypadku zastosujemy proste stochastyczne zejście po gradiencie, definiowane:

```
train_step = tf.train.GradientDescentOptimizer(LR).minimize(cost)
```

Samo uczenie można zrealizować za pomocą pętli:

```
for i in xrange(600):
    batch = mnist.train.next_batch(200)
    train_step.run(feed_dict={_x: batch[0],
                             _y: batch[1],
                             P_1: 0.5,
                             LR: 0.5})
```

Po wydaniu tych komend uczenie zostanie rozpoczęte. Każda iteracja ładuje po 50 wektorów treningowych, zaś parametr **feed_dict** określa, jakie zmienne zostaną dostarczone sieci do obliczeń. Parametr **P_1** wymusza 50% dropout, zaś **LR** współczynnik uczenia na poziomie 0.5. Ponieważ zestaw ćwiczeniowy MNIST zawiera 60000 przykładów, powyższa pętla wykona 2 pełne epoki.

Aby wnioskować coś o jakości takiego systemu, należy sprawdzić model. Ponieważ wyjściem sieci jest 10 neuronów, zawierających prawdopodobieństwo że wektor wejściowy jest daną cyfrą, należy zastosować funkcję, która znajdzie maksymalne prawdopodobieństwo. Można do tego zastosować funkcję **tf.argmax**, którą można zastosować tak:

```
correct = tf.equal(tf.argmax(O2, 1), tf.argmax(_y, 1))
```

Funkcja ta zwróci listę wartości typu boolean, mówiących czy klasyfikacja była poprawna. Aby policzyć jakość sieci, można odwzorować je na liczby i policzyć średnią:

```
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

Następnie można ją wypisać:

```
print accuracy.eval(feed_dict={_x: mnist.test.images,
                                 _y: mnist.test.labels,
                                 P_1: 1.0})
```

Po dwóch iteracjach uczenia takiej prostej sieci można spodziewać się jakości rzędu 45%-64%. Ciekawić może ustawienie parametru P_1 na 1.0. Oznacza to po prostu, że w obliczeniach wezmą udział wszystkie neurony. Nie jest to już bowiem uczenie sieci, a jej walidacja.

TensorFlow, za pomocą modułu TensorBoard, pozwala również zbierać dane na temat działania sieci, a nawet wizualizować obliczenia wchodzące w jej udział. Rejestrowanie parametrów sieci, a także opisanie jej w sposób który umożliwi generację przyjaznego wizualnie grafu jest zadaniem dość skomplikowanym. Graf wykonany w tym rozdziale zaprezentowano dokładnie na rysunku 16. Umożliwia on również w prosty sposób badanie zmian wartości zmiennych w czasie - wykresy średniej wartości funkcji celu przedstawiono na wykresie 18, a dokładność na wykresie 17.

Odpowiednik tej sieci (tylko z inną architekturą) wykonany za pomocą biblioteki nnetsys przedstawiono w 5.2.2.

4.3. Sieci splotowe

Tu pokazane zostanie, w taki sposób za pomocą TensorFlow stworzyć, nauczyć, dokonać walidacji i wykreślić graf splotowej sieci neuronowej klasyfikującej zbiór danych CIFAR-10.

TensorFlow, sam z siebie, nie posiada funkcji potrafiącej pobrać zbiór CIFAR-10. Należy to zrobić ręcznie (można pobrać pliki *pickle* Pythona z [23]). Zbiór ten należy rozpakować (założymy że do katalogu *cifar10-batches-py*) i wydać następujące komendy:

```
import pickle
import numpy as np
from tensorflow.contrib.learn.python.learn.datasets.mnist \
    import dense_to_one_hot

train_data, train_labels, test_data, test_labels = [], [], [], []
for i in xrange(1, 6):
    d = pickle.load(
        open('cifar-10-batches-py/data_batch_%s' % (i, ), 'rb'))
```

Tabela 10: Plan architektury dla sieci splotowej CIFAR-10

Warstwa i architektura	Wym. wejściowy	Wym. wyjściowy
Splotowa, 32 filtry 5x5	32x32x3	28x28x32
Max-pool, 2x2	28x28x32	14x14x32
Splotowa, 32 filtry 5x5	14x14x32	10x10x32
Max-pool, 2x2	10x10x32	5x5x32
ReLU, 1000 neuronów	800	1000
ReLU, 500 neuronów	1000	500
Softmax, 10 neuronów	500	10

```
train_data.extend(d['data'] / 255.0)
train_labels.extend(dense_to_one_hot(
    np.array(d['labels']))))

d = pickle.load(
    open('cifar-10-batches-py/test_batch', 'rb'))
)

test_data = d['data'] / 255.0
test_labels = dense_to_one_hot(np.array(d['labels']))
```

Etykiety są cyframi z numerem klasy, natomiast do celów obliczenia funkcji celu niezbędny jest tzw. wektor one-hot. Przykładowo wektorem one-hot dla klasy 3 byłby [0, 0, 1, 0, ..., 0]. Funkcja TensorFlow **dense_to_one_hot** dokonuje tej transformacji. Wymaga ona argumentu w postaci wektora biblioteki NumPy, stąd konwersja za pomocą np.array. Dzielenie danych wejściowych przez 255 służy ich normalizacji - są one bowiem podane w formacie RGB. Komendy te znajdują się rozproszony po 5 plikach zbiór danych uczących oraz danych testowych, wraz z etykietami.

Teraz należy rozplanować architekturę sieci splotowej. Obrazy mają rozmiar 32x32 i są kolorowe (3 kanały). Warstwy splotowe zostaną pozostawione bez regularyzacji dropout. Można więc zastosować architekturę przedstawioną w tabeli 10.

Aby zbudować taką sieć, należy zdefiniować w TensorFlow szereg zmien-

Tabela 11: Zmienne Python dla sieci splotowej CIFAR-10

Komenda	Co zawiera zmienna
BATCH_SIZE = 200	Rozmiar batch
W1 = tf.Variable(tf.truncated_normal([5,5,3,32]))	Jądra transformacji pierwszej warstwy splotowej
B1 = tf.Variable(tf.zeros([32]))	Biasy pierwszej warstwy
W2 = tf.Variable(tf.truncated_normal([5,5,32,32]))	Jądra transformacji drugiej warstwy splotowej
B2 = tf.Variable(tf.zeros([32]))	Biasy drugiej warstwy
W3 = tf.Variable(tf.truncated_normal([800,1000]))	Wagi trzeciej warstwy
B3 = tf.Variable(tf.zeros([1000]))	Biasy trzeciej warstwy
W4 = tf.Variable(tf.truncated_normal([1000,500]))	Wagi czwartej warstwy
B4 = tf.Variable(tf.zeros([500]))	Biasy czwartej warstwy
W5 = tf.Variable(tf.truncated_normal([500,10]))	Wagi piątej warstwy
B5 = tf.Variable(tf.zeros([10]))	Biasy piątej warstwy
P = tf.placeholder(tf.float32)	Parametr dropout
_x = tf.placeholder(tf.float32, shape=[None, 3072])	Wejście sieci
_y = tf.placeholder(tf.float32, shape=[None, 10])	Wyjście sieci
LR = tf.placeholder(tf.float32)	Wartość współczynnika uczenia

nych, przedstawionych w tabeli 11.

Kolejnym krokiem jest zdefiniowanie relacji matematycznych panujących w sieci. Wiadomo, że zmiana wymiarowości tensora będzie niezbędna przynajmniej w dwóch krokach. Warstwa splotowa spodziewa się tensora 4D postaci (indeks, szerokość, wysokość, kanał RGB), natomiast próbki są w postaci macierzy (indeks, dane). Kolejna transformacja nastąpi przechodząc z warstw splotowych do warstw w pełni połączonych (DNN). Uwzględniając to, oraz dodając odpowiednią regularyzację dropout w każdej warstwie, komendy (z dodanymi komentarzami) będą wyglądały następująco:

```
# Przejście z macierzy na tensor 4D
IN = tf.reshape(_x, [-1, 32, 32, 3])

# Pierwsza splotowa
O1 = tf.nn.conv2d(IN, W1, [1,1,1,1], padding='VALID')
O1 = tf.nn.bias_add(O1, B1)
O1 = tf.nn.relu(O1)
```

```

# Pierwsza max-pool
O1 = tf.nn.max_pool(O1, ksize=[1,2,2,1],
                     strides=[1,2,2,1], padding='VALID')

# Druga splotowa
O2 = tf.nn.conv2d(O1, W2, [1,1,1,1],
                  padding='VALID')
O2 = tf.nn.bias_add(O2, B2)
O2 = tf.nn.relu(O2)

# Druga max-pool i przejscie z tensora 4D na macierz
O2 = tf.nn.max_pool(O2, ksize=[1,2,2,1],
                     strides=[1,2,2,1], padding='VALID')
O2 = tf.nn.reshape(O2, [-1, 800])

# Pierwsza w pelni polaczona
O3 = tf.nn.relu(tf.matmul(O2, W3) + B3))
O3 = tf.nn.dropout(O3, P)

# Druga w pelni polaczona
O4 = tf.nn.relu(tf.matmul(O3, W4) + B4))
O4 = tf.nn.dropout(O4, P)

# Warstwa wyjsciowa softmax
O5 = tf.nn.softmax(tf.matmul(O4, W5) + B5)

```

Operacje obliczenia funkcji celu, dokładności i uczenia systemu definiuje się analogicznie jak w przypadku sieci DNN:

```

cost = -tf.reduce_sum(_y*tf.log(O5+1e-12), reduction_indices=[1])
cost = tf.reduce_mean(cost, name='cross_entropy')
train = tf.train.GradientDescentOptimizer(LR).minimize(cost)
correct = tf.equal(tf.argmax(O5, 1), tf.argmax(_y, 1))

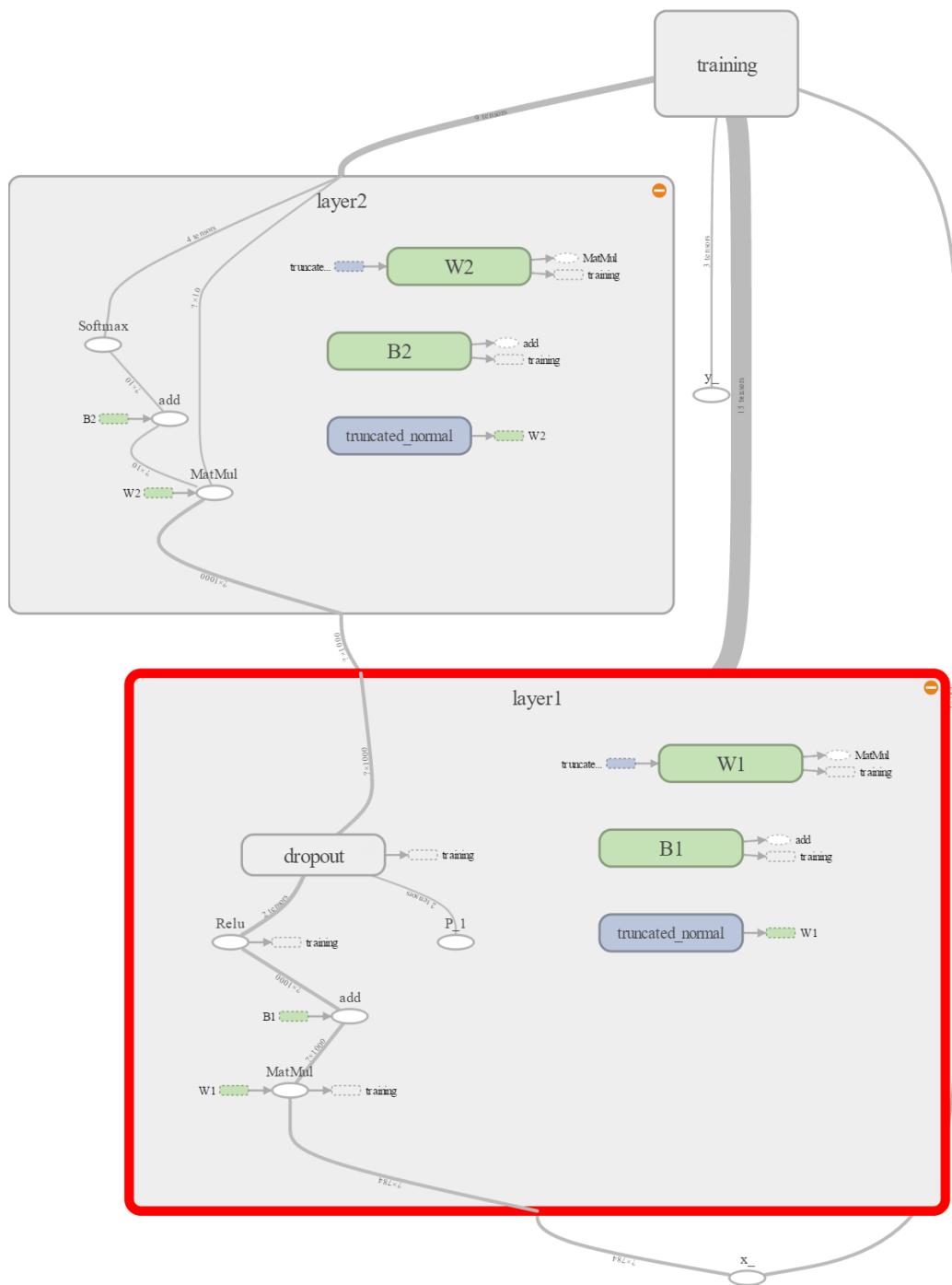
```

```
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

Dane zostały załadowane ręcznie, tak więc niemożliwe jest wykorzystanie automatycznego generowania iteracji przez TensorFlow. Problem należy rozwiązać ręczną iteracją po każdym wsadzie dla epoki. Jedna epoka wyrażać się więc będzie kodem:

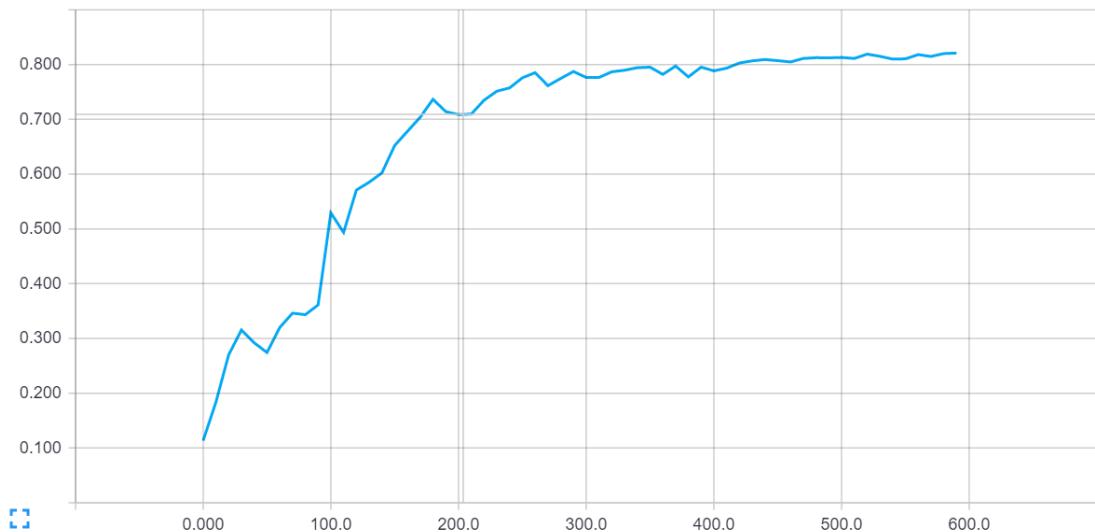
```
for i in xrange(len(train_data)/BATCH_SIZE):
    session.run([train], feed_dict={
        _x: train_data[i*BATCH_SIZE:(i+1)*BATCH_SIZE],
        _y: train_labels[i*BATCH_SIZE:(i+1)*BATCH_SIZE],
        LR: 0.5,
        P: 0.5
    })
```

Testowanie sieci realizuje się analogicznie jak w przypadku sieci DNN. Odpowiednik tej sieci (tylko z inną architekturą) wykonany za pomocą biblioteki nnetsys przedstawiono w 5.2.3. Graf obliczeniowy wygenerowany dla tej sieci przedstawiono na rysunku 19.



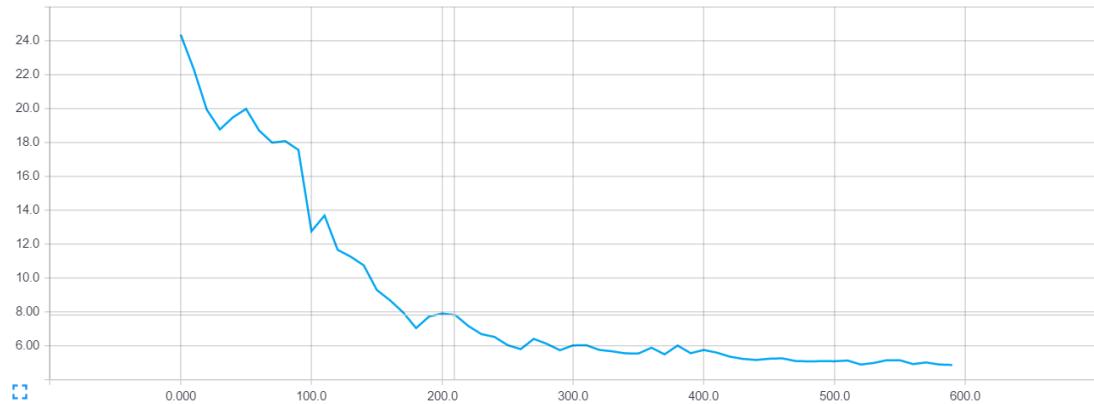
Rysunek 16: Graf obliczeniowy sieci DNN narysowany przy użyciu TensorFlow

Dokladnosc

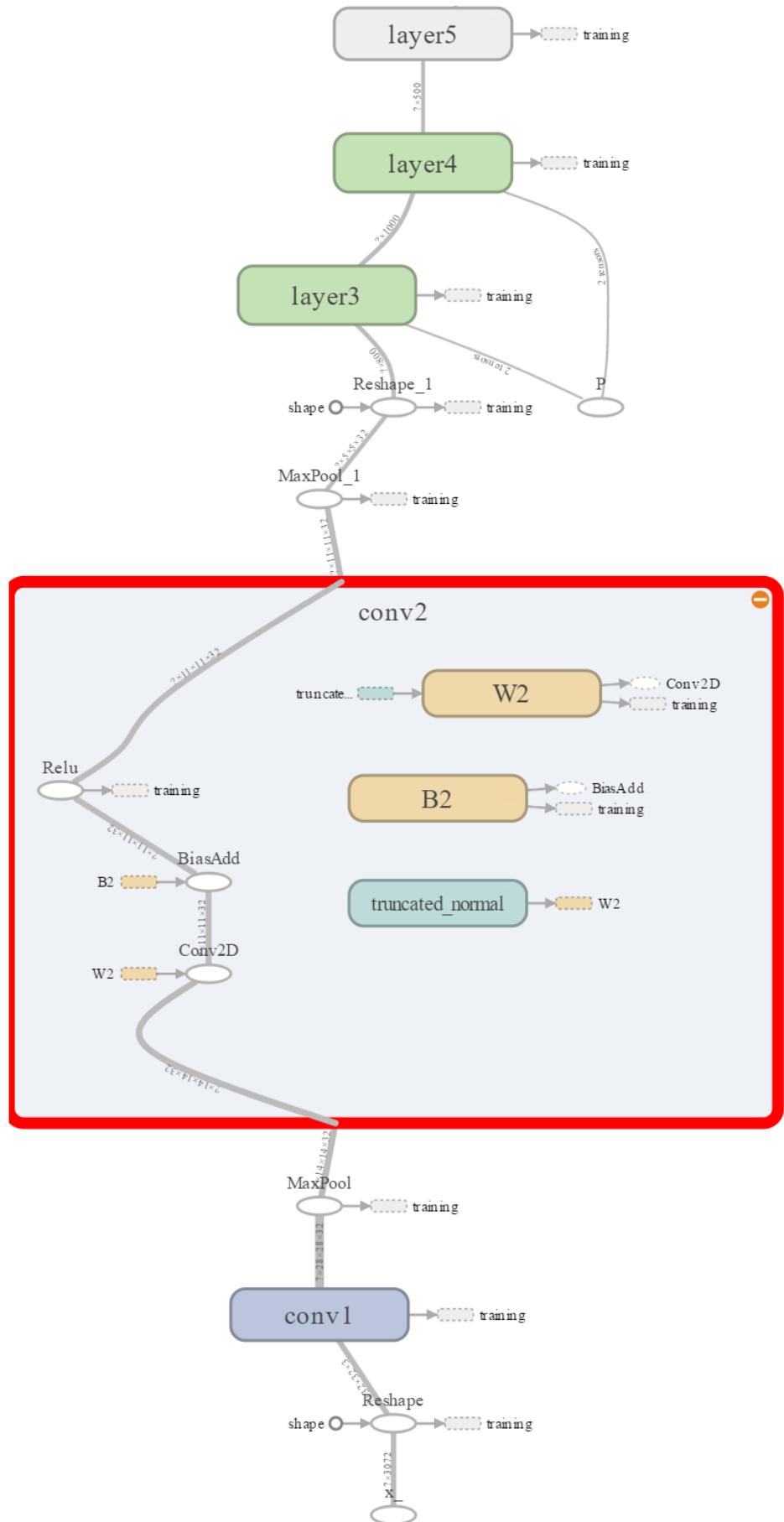


Rysunek 17: Jakość sieci w funkcji epoki, wykres z TensorFlow.

Funkcja celu



Rysunek 18: Wartość funkcji celu w funkcji czasu, wykres z TensorFlow.



Rysunek 19: Graf obliczeniowy sieci splotowej, wygenerowany przez Tensor-Flow.
71

5. Konfiguracja środowiska Python i Theano

W tym rozdziale zamieszczono instrukcję w jaki sposób ustawić środowisko Python współpracujące z Theano, zdolne do obliczeń zarówno za CPU jak i GPU firmy nVidia (technologia CUDA). Będzie się to wiązało z instalacją pakietów Python, Theano oraz CUDA SDK. Bezpośrednie polecenia przedstawione zostaną czcionką **wytłuszczona**. Instrukcja ta powstała na podstawie instrukcji napisanej przez autorów Theano, dostępnej pod adresem [1].

5.1. Instalacja środowiska

Przy instalacji z oficjalnej dystrybucji Pythona należy ręcznie skonfigurować szereg pakietów, co bywa czasochłonne. Powstał więc szereg dystrybucji Pythona, w tym skierowanych do odbiorców naukowych. Jedną z takich dystrybucji jest WinPython [2]. Zawiera on większość pakietów wykorzystywanych w celach naukowych oraz liczne edytory i środowiska programistyczne. Jednym z wygodniejszych edytorów dostępnych w ramach WinPython jest *spyder*, dostępne do uruchomienia z katalogu instalacyjnego WinPython. Konfiguruje on ponadto odpowiednio środowisko uruchomieniowe Pythona. Cechą ta będzie niezwykle ważna przy konfiguracji Pythona do współpracy z kompilatorem C i kartą graficzną.

Theano jest zgodne z Pythonem 3, jednak ze względu na to, że instalacje Python 2 są lepiej wspierane, zalecana jest domyślna instalacja WinPythona wersji 2.7 w zupełności wystarcza do pracy z systemem. nnetsys nie współpracuje z Pythonem 3, co wymusza instalację wersji 2.7. Do pracy z dużymi zbiorami danych wykorzystywana będzie 64-bitowa wersja Pythona - wersja 32-bitowa nie będzie w stanie przetworzyć zbiorów większych niż 2 GB, a tyle pamięci bez problemu jest w stanie zająć zbiór CIFAR-10 z siecią splotową. **Należy więc pobrać i zainstalować 64-bitową wersję WinPython 2.7.**

Jeśli zależy nam na prowadzeniu obliczeń za pomocą GPU, należy teraz kontynuować instalację od punktu 5.1.1. Jeśli nie, można przejść do

punktu 5.1.2.

5.1.1. Instalacja dla GPU

Jeśli chcemy korzystać z wsparcia GPU do prowadzenia obliczeń, należy w tym momencie wykonać kilka czynności przed dalszą instalacją Theano. Przede wszystkim jest to instalacja pakietu CUDA. Pakiet CUDA do instalacji wymaga kompilatora Visual Studio. **Należy więc pobrać i zainstalować oprogramowanie Microsoft Visual Studio 2010 Express.** Jest to dość stara wersja oprogramowania, możliwa jest instalacja nowszej wersji na własną odpowiedzialność. W takim przypadku należy ustalić odpowiednią wersję CUDA. Ponieważ instalowana jest wersja 64-bitowa, należy doinstalować również 64-bitowy kompilator C. Visual Studio 2010 nie posiada natywnie takiego kompilatora - należy doinstalować pakiet Microsoft Windows SDK. **Instalujemy Microsoft Windows SDK w wersji 7.1.**

Następnie możliwa jest instalacja CUDA. Zaleca się wersję 5.5, gdyż współpracuje ona z kompilatorem C które zostanie zainstalowany na końcu. W przypadku instalacji nowszej wersji Visual Studio należy określić jaka wersja CUDA współpracuje z tym produktem - konkretne wersje CUDA są dość wymagające w stosunku do wersji kompilatora C. **Należy pobrać i zainstalować odpowiednią wersję nVidia CUDA SDK (zalecana wersja to 5.5).**

Po instalacji CUDA SDK Visual Studio 2010 nie jest już potrzebny. Może zostać odinstalowany. **Należy odinstalować Visual Studio 2010. Należy kontynuować instrukcję od punktu 5.1.2.**

5.1.2. Instalacja kompilatora C i Theano

Python nie może kompilować modułów bez swoich nagłówków C. Nagłówki te dostarcza kompilator Microsoft na potrzeby Pythona - należy go więc również zainstalować. Jest on dostępny pod adresem

<https://www.microsoft.com/en-us/download/details.aspx?id=44266>.

Należy pobrać i zainstalować Microsoft Visual C++ Compiler for Py-

thon 2.7. Do katalogu VC\include należy pobrać plik z

<http://msinttypes.googlecode.com/svn/trunk/stdint.h>

W katalogu settings instalacji WinPython **należy ponadto utworzyć plik .theanorc.txt**, o następującej treści:

```
[global]
device = gpu
floatX = float32
```

Informuje on Theano o tym, że do obliczeń numerycznych wykorzystywana będzie karta graficzna, a domyślny rozmiar wartości zmiennoprzecinkowych to 32 bity (64 bity na GPU nie są na tą chwilę obsługiwane).

Jeśli nie chcemy korzystać z GPU, pomijamy krok tworzenia pliku .theanorc.txt.

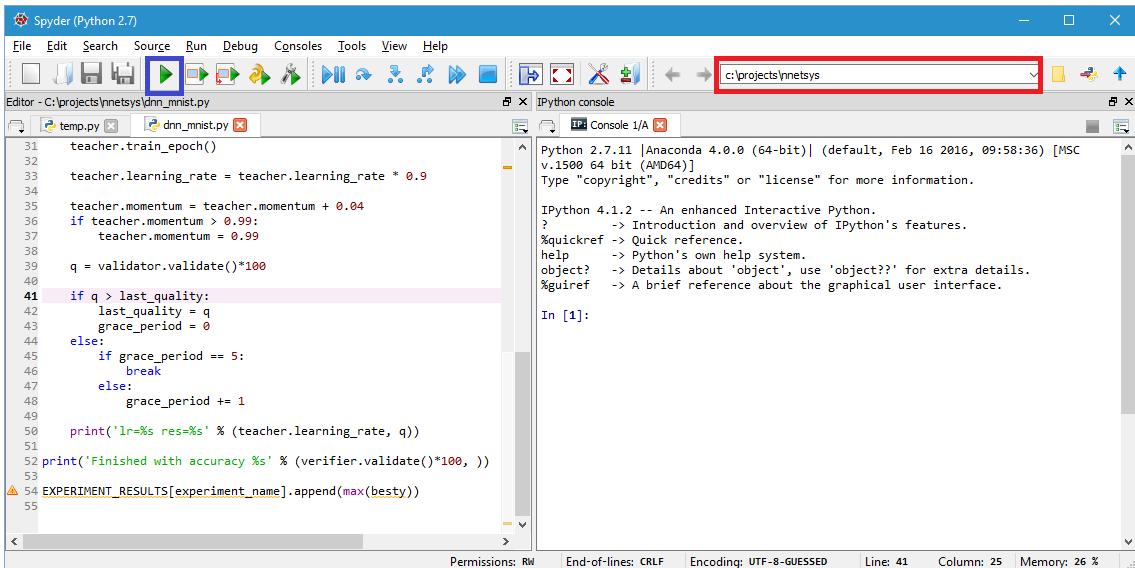
Obliczenia numeryczne wykonywane na GPU na potrzeby uczenia głębokiego są stabilne numerycznie, co ustalono w rozdziale 3.1.

5.2. Uruchamianie eksperymentów

Na załączonej do pracy płycie CD umieszczono katalog *nnetsys*. Zawiera on samą bibliotekę *nnetsys* (nie wymagającej osobnej instalacji), kod źródłowy eksperymentów wykonanych w rozdziale 3, oraz trzy przykładowe skrypty w formie gotowej do uruchomienia.

test.py jest skryptem testującym poprawność instalacji środowiska. Potrafi on także stwierdzić czy instalacja Theano została skonfigurowana do pracy z CPU czy GPU. **dnn_mnist.py** jest skryptem uczącym sieć DNN zbioru danych MNIST. Prezentuje ona komplet wyników z uczenia sieci. **cnn_cifar.py** jest skryptem uczącym sieć CNN zbioru danych CIFAR-10.

Zakładając zainstalowany WinPython, otwieramy narzędzie *spyder*. Znajduje się ono w Menu Start lub w katalogu, w którym narzędzie zostało zainstalowane. Z menu File>Open otwieramy eksperiment, który chcemy otworzyć. Należy upewnić się, że w polu z katalogiem roboczym wybrany jest katalog zawierający ten plik. Na rysunku 20 pole to zostało zaznaczone czerwonym kolorem.



Rysunek 20: Zrzut ekranu środowiska spyder. Kolorem czerwonym zaznaczono pole katalogu roboczego, niebieskim - guzik uruchomienia skryptu.

Następnie należy kliknąć raz w konsoli (część po prawej stronie), a potem kliknąć ikonkę Run (zaznaczona niebieską ramką na rysunku 20). Bez jednego kliknięcia w konsoli uruchomienie skryptu zostanie przerwane z błędem *No Python console is currently selected to run*. W kolejnych rozdziałach omówione zostaną 3 eksperymenty, ich kod źródłowy, oraz uruchomienie i wykonanie.

5.2.1. test

test jest skryptem o dwóch działaniach. Po pierwsze, sprawdza czy wymagane moduły są zainstalowane, a jeśli tak to informuje o tym. W przypadku błędu identyfikuje moduły które nie zostały zainstalowane, lub są niedostępne z innego powodu.

Skrypt wykrywa również, czy instalacja Theano została skonfigurowana do pracy z CPU i GPU. Wykorzystuje do tego celu różnice w rozkazach emitowanych przez kompilator w przypadku kompilacji na CPU oraz GPU. W przypadku GPU funkcja wykładnicza operuje na wektorach, a nie na pojedynczych elementach.

5.2.2. dnn_mnist

dnn_mnist buduje głęboką sieć neuronową (DNN) za pomocą biblioteki *nnetsys*. Jest to sieć o architekturze 768-1600-1600-1300-1300-1400-10 z dropout 20% w pierwszej warstwie o 50% w pozostałych warstwach. Sieć ta jest uczona za pomocą SGD z momentum. Sieć uczona jest zbioru danych MNIST, do momentu w którym wyniki przestają się poprawiać na zbiorze walidującym. Na końcu wypisywana jest jakość sieci na zbiorze testowym. Sieć powinna osiągnąć jakość ok. 98%.

5.2.3. cnn_cifar

cnn_cifar buduje splotową sieć neuronową (CNN) za pomocą biblioteki *nnetsys*. Jest to sieć o architekturze 80 filtrów 5x5 w warstwie pierwszej, następnie max-pool 2x2, następnie 60 filtrów 5x5, następnie warstwa max-pool 2x2, następnie 800 neuronów tanh i 10 neuronów softmax. Sieć uczona jest na zbiorze CIFAR-10 metodą SGD z momentum. Uczenie przerywane jest gdy jakość sieci przestanie się poprawiać. Na końcu wypisywana jest jakość wynikowej sieci na zbiorze testowym. Sieć powinna osiągnąć dokładność ok. 67%.

6. Podsumowanie

Uczenie głębokie stanowi rozwojową gałąź uczenia maszynowego. Osiaga ono niezwykle dobre efekty, pozwalając wykorzystać duże wolumeny danych na niespotykaną do tej pory skalę, przybliżając się do ludzkich wyników w sytuacjach, w których do tej pory komputer radził sobie znacznie gorzej. Stanowi ono praktyczne podstawy uczenia nienadzorowanego, opierającego się na determinowaniu struktury danych poprzez obserwację. Wszak uczenie ludzkie i zwierzęce jest w dużej mierze nienadzorowane. Wzorowanie się na naturze wydaje się być uzasadnione, zważywszy na zasadnicze podobieństwo pracy pierwszej warstwy kory wzrokowej zwierząt i sieci splotowych.

Co ważniejsze, ze względu na powszechną dostępność sprzętu o wysokiej mocy obliczeniowej, zwłaszcza systemów GPU, uczenie głębokie może być popularyzowane. Tworzenie nowych systemów i uczenie sieci nie wymaga złożonego klastra (choć w przypadku bardzo dużych zbiorów jest to dalej wymagane), a biblioteki są dostatecznie proste w obsłudze aby korzystać z nich bez formalnego szkolenia, posługując się jedynie zasobami dostępnymi w Internecie.

Uczenie głębokie stanowi rozszerzenie istniejącej już koncepcji sieci neuronowych. Mimo prostoty, różni się ono jednak w kilku miejscach. Świadomość tych różnic jest niezbędna, jeśli zamiarem jest budowanie klasyfikatorów wysokiej jakości, czy prowadzenie badań nad tymi sieciami. W pracy tej przedstawiono najważniejsze algorytmy, stojące za działaniem sieci głębokich. Opisano sposób ich działania i przedstawiono wpływ na proces uczenia i działanie sieci głębokiej. Zademonstrowano również sposób konfiguracji środowiska Theano na popularnej platformie Windows w sposób który pozwoli korzystać z GPU. Dla użytkowników tego systemu ta biblioteka stanowi dobry punkt wyjścia do zainteresowania się uczeniem głębokim, podczas gdy użytkownicy Linuksa mogą skorzystać z biblioteki TensorFlow, opisanej w tej pracy w rozdziale 4.

Najważniejsza część pracy demonstrowała kluczowe różnice w uczeniu

głębokim i klasycznych sieciach neuronowych. Wykazano podstawowe różnice w doborze hiperparametrów takich jak współczynnik uczenia czy momentum. Wskazano dlaczego właściwie niezbędna jest zmiana zasadniczych algorytmów, by sieć mogła skorzystać z dużej ilości warstw i wykazywać się “głębokością”. Przedstawiono również zachowanie sieci splotowej, demonstrując jej przewagę na zadaniach rozpoznawania obrazów. Rozdział 3, poświęcony eksperymentom, zamyka próba wyjaśnienia sposobu działania sieci splotowej poprzez analizę graficzną wag, czyli parametrów których uczy się sieć.

Praca ta stanowi więc praktyczny poradnik, wprowadzający w najważniejsze pojęcia dotyczące nadzorowanego uczenia głębokiego i pozwala na zapoznanie się z algorytmami oraz poradami dotyczącymi projektowania i uczenia dwóch klas tego typu systemów - głębokich sieci neuronowych (DNN) i sieci splotowych (CNN). Rozwiązania te długo pozostaną jeszcze w obszarze zainteresowania specjalistów od uczenia maszynowego oraz zastosowań przemysłowych.

Autor za wkład własny pracy uważa:

- konfiguracja i sporządzenie opisu konfiguracji środowiska programowego do przeprowadzenia eksperymentów uczenia głębokiego,
- zaplanowania, napisania skryptów realizujących, przeprowadzenia i opracowania danych z eksperymentów mających na celu porównanie klasycznych systemów uczenia i systemów głębokich,
- napisania autorskiej biblioteki uczenia głębokiego, implementującej struktury głębokich sieci neuronowych oraz splotowych sieci neuronowych, oraz przygotowanie dwóch eksperymentów w formie “gotowej do uruchomienia”,
- napisania autorskiej biblioteki uczenia głębokiego, implementującej struktury głębokich sieci neuronowych oraz splotowych sieci neuronowych, oraz przygotowanie dwóch eksperymentów w formie “gotowej do uruchomienia”,
- sporządzenie wprowadzenia do biblioteki TensorFlow.

Spis rysunków

1.	Render architektury sieci w TensorFlow	12
2.	Model sieci neuronowej	17
3.	Demonstracja działania filtrów sieci splotowej	20
4.	Wynik porównań funkcji inicjalizacji	33
5.	Wynik porównań wartości wsp. uczenia	36
6.	Jakość sieci <i>tanh</i> i DNN w funkcji epoki na zbiorze CIFAR-10	43
7.	Wartość funkcji celu <i>tanh</i> i DNN w funkcji epoki na zbiorze CIFAR-10	44
8.	Zależność jakości sieci od ilości warstw	45
9.	Zależność jakości sieci <i>tanh</i> od ilości warstw	46
10.	Wykres jakości sieci od ilości filtrów splotowych w warstwie 1 i 2	48
11.	Wykres jakości sieci splotowej od udziału sztucznych danych	50
12.	Porównanie jakości sieci <i>tanh</i> i splotowej na problemie MNIST w funkcji numeru epoki	52
13.	Porównanie jakości sieci <i>tanh</i> i splotowej na problemie MNIST w funkcji upływu czasu	53
14.	Jakość uczenia sieci CIFAR w funkcji numeru epoki	54
15.	Jakość sieci CIFAR w funkcji czasu ich uczenia	55
16.	Graf obliczeniowy sieci DNN w renderze TensorFlow	69
17.	Jakość sieci w funkcji epoki, wykres z TensorFlow	70
18.	Wartość funkcji celu w funkcji czasu, wykres z TensorFlow .	70
19.	Graf obliczeniowy sieci splotowej, wygenerowany przez TensorFlow	71
20.	Zrzut ekranu środowiska spyder	75

Spis tabel

1.	Wyniki testu stabilności numerycznej algorytmu	27
2.	Wyniki testu sieci dropout przy różnych współczynnikach uczenia	35
3.	Wyniki testu sieci tanh przy różnych współczynnikach uczenia	35
4.	Jakości sieci w funkcji różnych wartości wsadu, regularyzacji i momentum	39
5.	Ogólne wzory sieci podlegających testom jakości dla zbioru MNIST	41
6.	Wybrane wycinki rankingu sieci MNIST	42
7.	Wyniki wybranych architektur sieci splotowych	47
8.	Wizualizacje filtrów pierwszej warstwy sieci splotowej uczonej na zbiorze MNIST	58
9.	Zmienne sieci głębszej dla problemu MNIST w TensorFlow .	62
10.	Plan architektury dla sieci splotowej CIFAR-10	65
11.	Zmienne Python dla sieci splotowej CIFAR-10	66

Bibliografia

- [1] *Installation of Theano on Windows.* http://deeplearning.net/software/theano/install_windows.html.
- [2] *WinPython.* <http://winpython.sourceforge.net/>.
- [3] *Theano.* <http://deeplearning.net/software/theano/>, dostęp 28 czerwca 2016.
- [4] Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo i Zhi-feng Chen: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, 2015. Software available from tensorflow.org.
- [5] Benesov, Rodrigo: *What is the class of this image? Discover the current state of the art in objects classification.* http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html, dostęp 28 czerwca 2016.
- [6] Bergstra, James, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley i Yoshua Bengio: *Theano: a CPU and GPU Math Expression Compiler.* W *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Czerwiec 2010.
- [7] Collobert, Ronan, Samy Bengio i Johnny Marithoz: *Torch: A Modular Machine Learning Software Library*, 2002.
- [8] Cui, Xiaodong, Vaibhava Goel i Brian Kingsbury: *Data Augmentation for Deep Neural Network Acoustic Modeling.* IEEE/ACM Trans. Audio, Speech & Language Processing, 23(9):1469–1477, 2015.
- [9] DiCarlo, James J i David D Cox: *Untangling invariant object recognition.* Trends in Cognitive Sciences, 11(8):333–341, aug 2007, ISSN 1364-6613.

- [10] Gibson, Adam, Chris Nicholson i Josh Patterson: *Deeplearning4j: Open-source, distributed, deep-learning library for the JVM*. <http://deeplearning4j.org/>, dostęp 28 czerwca 2016.
- [11] Glorot, Xavier i Yoshua Bengio: *Understanding the difficulty of training deep feedforward neural networks*. W In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10)*. Society for Artificial Intelligence and Statistics, 2010.
- [12] Glorot, Xavier, Antoine Bordes i Yoshua Bengio: *Deep Sparse Rectifier Neural Networks*. W Gordon, Geoffrey J. i David B. Dunson (redaktorzy): *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, volume 15, strony 315–323. Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011.
- [13] Goodfellow, Ian J., Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud i Vinay D. Shet: *Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks*. CoRR, abs/1312.6082, 2013.
- [14] Goodfellow, Ian J., David Warde-farley, Mehdi Mirza, Aaron Courville i Yoshua Bengio: *Maxout networks*. W In *ICML*, 2013.
- [15] Google: *Building Bazel on Windows*. <http://bazel.io/docs/windows.html>, dostęp 28 czerwca 2016.
- [16] Google: *Improving Photo Search: A Step Across the Semantic Gap*. <http://googleresearch.blogspot.ca/2013/06/improving-photo-search-step-across.html>, dostęp 28 czerwca 2016.
- [17] Google: *reCAPTCHA*. <https://www.google.com/recaptcha/intro/index.html>, dostęp 28 czerwca 2016.
- [18] He, Kaiming, Xiangyu Zhang, Shaoqing Ren i Jian Sun: *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. CoRR, abs/1502.01852, 2015.

- [19] Hof, Robert: *Meet the guy who helped Google beat Apple's Siri*. <http://www.forbes.com/sites/roberthof/2013/05/01/meet-the-guy-who-helped-google-beat-apples-siri/>, dostęp 28 czerwca 2016.
- [20] Hu, Ming Kuei: *Visual pattern recognition by moment invariants*. Information Theory, IRE Transactions on, 8(2):179–187, February 1962, ISSN 0096-1000.
- [21] IBM: *The Jeopardy! Quiz Show*. http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=2158.
- [22] Jia, Yangqing, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama i Trevor Darrell: *Caffe: Convolutional Architecture for Fast Feature Embedding*. arXiv preprint arXiv:1408.5093, 2014.
- [23] Krizhevsky, Alex: *The CIFAR-10 dataset*. <https://www.cs.toronto.edu/~kriz/cifar.html>.
- [24] Krizhevsky, Alex: *Learning multiple layers of features from tiny images*. Raport techniczny, 2009.
- [25] Krizhevsky, Alex, Ilya Sutskever i Geoffrey E. Hinton: *Imagenet classification with deep convolutional neural networks*. W *Advances in Neural Information Processing Systems*, strona 2012.
- [26] LeCun, Y., Y. Bengio i G.E. Hinton: *Deep Learning*. Nature, 521:436–444.
- [27] LeCun, Yann, Corinna Cortes i Christopher J.C. Burges: *THE MNIST DATABASE of handwritten digits*. <http://yann.lecun.com/exdb/mnist/>, dostęp 28 czerwca 2016.
- [28] Longstaff, Alan: *Krótkie wykłady - neurobiologia*, strony 199–201. PWN, ISBN 978-83-01-13805-9.
- [29] nVidia: *Nvidia Quadro 6000*. https://www.nvidia.com/docs/IO/40049/NV_DS_QUADRO_6000_Oct10_US_LR.pdf, dostęp 28 czerwca 2016.

- [30] Simonyan, Karen i Andrew Zisserman: *Very Deep Convolutional Networks for Large-Scale Image Recognition*. CoRR, abs/1409.1556, 2014.
- [31] Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever i Ruslan Salakhutdinov: *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. J. Mach. Learn. Res., 15(1):1929–1958, Styczeń 2014, ISSN 1532-4435.
- [32] Vasilache, Nicolas, Jeff Johnson, Michaël Mathieu, Soumith Chintala, Serkan Piantino i Yann LeCun: *Fast Convolutional Nets With fbfft: A GPU Performance Evaluation*. CoRR, abs/1412.7580, 2014.
- [33] Wolpert, D. H. i W. G. Macready: *No free lunch theorems for optimization*. IEEE Transactions on Evolutionary Computation, 1(1):67–82, Apr 1997, ISSN 1089-778X.

Dodatek

Na załączonej płycie CD umieszczone:

- tekst pracy w formacie PDF oraz \LaTeX ,
- kod źródłowy pracy,
- pliki zawierające wyniki eksperymentów oraz ich kody źródłowe.

Sygnatura:

POLITECHNIKA RZESZOWSKA im. I. Łukasiewicza
Wydział Elektrotechniki i Informatyki
Katedra Informatyki i Automatyki

Rzeszów, 2016

STRESZCZENIE PRACY DYPLOMOWEJ MAGISTERSKIEJ

UCZENIE GŁĘBOKIE - PODSTAWY TEORETYCZNE I IMPLEMENTACJA PROGRAMOWA

Autor: Piotr Maślanka, nr albumu: EF/AA-DU-127172

Opiekun: prof. dr hab. inż. Jacek Kluska

Słowa kluczowe: uczenie głębokie, uczenie maszynowe, sztuczna inteligencja,
rozpoznawanie obrazów

Praca stanowi praktyczne wprowadzenie do sieci głębokich dla osób, które do tej pory zajmowały się sieciami neuronowymi. Przedstawia terminy i algorytmy wykorzystywane w uczeniu głębokim. Pokazuje jak za pomocą dwóch wiodących bibliotek uczenia głębokiego skonstruować ićwiczyć wybrane sieci uczenia głębokiego. Przedstawia najczęstsze problemy i istotne spostrzeżenia na temat uczenia tych sieci. Prezentuje uproszczoną analizę sposobu działania sieci splotowych.

RZESZOW UNIVERSITY OF TECHNOLOGY

Rzeszow, 2016

Faculty of Electrical and Computer Engineering

Department of Control and Computer Engineering

DIPLOMA THESIS (MS) ABSTRACT

DEEP LEARNING - BASIC PRINCIPLES AND A SOFTWARE IMPLEMENTATION

Author: Piotr Maślanka, code: EF/AA-DU-127172

Supervisor: Prof. Jacek Kluska, DSc, PhD, Eng.

Key words: deep learning, machine learning, artificial intelligence, image recognition

This thesis is a practical introduction to deep learning for those, who have experience in artificial neural networks. It presents the terms and algorithms used in deep learning. It shows how to construct and train chosen deep learning architectures. It presents common problems and important insights into training those networks. It conducts a simplified analysis of the way convolutional networks work.