POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka
SPECJALNOŚĆ: Systemy informatyki w medycynie (IMT)

PRACA DYPLOMOWA MAGISTERSKA

Badanie efektywności metody zamrażania w zastosowaniu do uczenia sieci głębokich.

Testing the effectiveness of the freezing method in the application for deep network learning.

AUTOR:
Michał Banach

PROWADZĄCY PRACĘ:

Prof. dr hab. inż. Ewa Skubalska-Rafajłowicz, K-9

OCENA PRACY:

WROCŁAW 2019

Spis treści

Streszczenie

W niniejszej pracy analizie poddano metody regularyzacji używane podczas uczenia głębokich sieci neuronowych. Badanie przeprowadzono z wykorzystaniem metody wymrażania [14] i dropout'u [16] oraz odniesiono te wyniki do wyników uzyskanych dla sieci neuronowych nauczonych bez regularyzacji. W celu uzyskania wyników, które można odnieść do innych zastosowań, badanie przeprowadzono dla 3 baz danych, używając 3 algorytmów optymalizacji oraz 2 architektów sieci neuronowych dla każdej z baz. Uzyskane wyniki wskazują na wyższość metody wymrażania w stosunku do braku regularyzacji, ale najlepszą jakość uzyskano dla algorytmu dropout. Ze względu na wysoką wariancję uzyskanych wyników, w celu potwierdzenie tych wniosków potrzebne jest przeprowadzenie dalszych badań.

Rozdział 1

Wstęp

W ostatnich latach bardzo popularną dziedziną uczenia maszynowego jest tak zwane uczenie głębokie tzn. uczenie sieci neuronowych o wielu warstwach ukrytych. Jest to zasługa dynamicznego rozwoju kognitywnego przetwarzania danych. To technologia związana z tym, w jaki sposób komputer może zrozumieć sygnały wejściowe, takie jak obrazy czy mowa. Rozwiązania takie są stosowane coraz częściej, a najbardziej znanymi spośród nich są autonomiczne samochody, które przy pomocy m.in. kamer i lidarów rozpoznają obiekty i przeszkody na drodze oraz inteligentni asystenci, którzy wykorzystują algorytmy rozpoznawania i syntezy języka naturalnego w celu rozmawiania z ludźmi i pomaganiu im w codziennych czynnościach, jak robienie zakupów czy umawianie się do fryzjera.

Poważnym problemem w uczeniu sieci neuronowych jest to, że głębokie sieci są podatne na przeuczenie. Terminem tym nazywane jest zjawisko, kiedy sieć traci swoje zdolności generalizacji. Dzieje się tak w ogólności dlatego, że zastosowany do danego zadania model matematyczny jest zbyt skomplikowany, a w przypadku sieci neuronowych jest to najczęściej wynik zbyt wielu wag w sieci oraz zbyt dużej liczby epok uczących. Model poprawia się do pewnej chwili, a następnie nadmiernie dopasowuje się do obecnych w danych szumów tzn. części sygnałów o charakterze przypadkowym. Istnieją metody rozpoznawania tego momentu oraz zapobiegania przeuczeniu. Wśród nich znajdują się:

1. wczesne zatrzymanie (ang. early stopping) [12],
2. regularyzacja (L1, L2) [6],
3. dropout [16],
4. ograniczanie wielkośći wag np. ograniczenia max norm [16],
5. sposoby dodawania szumu do wag sieci neuronowej.

Nowym algorytmem, który ma takie zadanie jest algorytm wymrażania [14]. Zapobiega on uczeniu się losowego zbioru wag w danych przebiegu propagacji wstecznej. Dokładniej ta metoda została opisana w dalszej części pracy.

1.1 Cele pracy

Celem niniejszej pracy jest zbadanie w jaki sposób różne metody zapobiegania przeuczeniu wpływają na uczenie się sieci neuronowej oraz na jej skuteczność.

Metody, które będą testowane to:

1. dropout,
2. wymrażanie.

Wpływ powyższych metod będzie badany w zależności od:

1. architektury sieci neuronowej,
   (a) ilości warstw ukrytych,
   (b) zastosowania warstw konwolucyjnych [6],
2. algorytmów optymalizacji,
   (a) algorytm gradientu stochastycznego [6],
   (b) algorytm AdaGrad [3],
   (c) algorytm Adam [8].
3. zadania,
   (a) zadanie klasyfikacji,
   (b) zadanie predykcji.

Pierwszą częścią badań było porównanie skuteczności uczenia sieci dla normalnego przypadku, a w części drugiej sprawdzono w jaki sposób zachowa się sieć neuronowa, dla której bez zastosowania metod zapobiegania przeuczeniu występuje nadmierne dopasowanie.

1.2 Zakres pracy

W rozdziale 2. wyjaśniono budowę głębokich sieci neuronowych uczonych w ramach badań. Opisane zostało też działanie warstw, z których się składał, to jest:

1. max pooling [6],
2. warstwa konwolucyjna [6],
3. warstwa gęsta [6].

Użyte algorytmy optymalizacji oraz sposób ich działania zostały przedstawione w rozdziale 3. Wśród nich znajdują się:

1. metoda gradientu stochastycznego,
2. algorytm Adagrad,
3. algorytm Adam z momentum Nesterova [2].

W rozdziale 4. opisano sposób działania metod zapobiegania przeuczeniu, które zbadano:

1. dropout,
2. wymrażanie,

Implementację metody wymrażania, która została stworzona na potrzeby niniejszej pracy oraz środowisko programistyczne przedstawiono w rozdziale 5.

W rozdziale 6 opisany został sposób w jaki przeprowadzono eksperymenty. Wyszczególnione zostały użyte bazy danych, architektury sieci użyte dla każdej z nich podobnie jak algorytmy optymalizacyjne i metody przeciwdziałania przeuczeniu. Przedstawiono w nim również wyniki badań. Pierwsza część tego rozdziału skupia się skuteczności sieci neuronowych w normalnych warunkach, podczas gdy w drugiej przedstawiono wpływ zastosowania dropout'u oraz wymrażania na sieć, w której bez regularyzacji występuje przeuczenie.

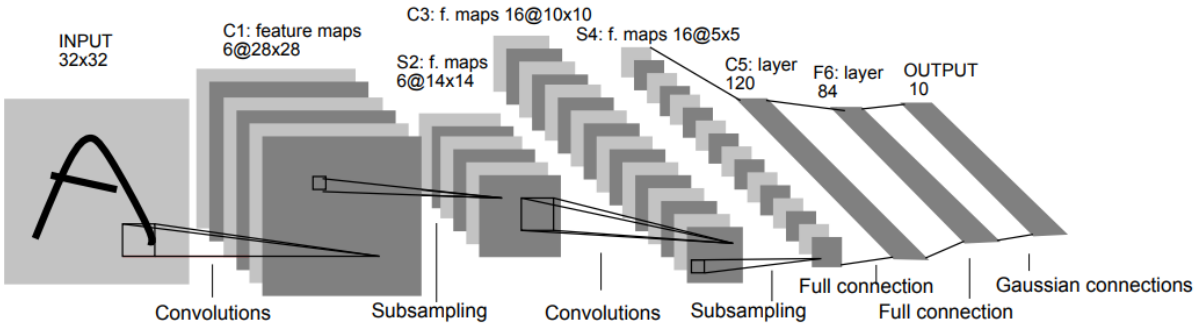
W rozdziale 7. podsumowano wyniki badań oraz wypełnione cele pracy.

Rozdział 2

Architektura głębokich sieci neuronowych

W niniejszej pracy wykorzystano sieci gęste tzw. MLP (ang. multilayer perceptron, pl. perceptron wielowarstwowy) [6] oraz architekturę konwolucyjnej sieci neuronowej na podstawie sieci LeNet-5 [10], która została przedstawiona na rysunku 2.1. Składała się ona z następujących 7 warstw:

1. warstwa konwolucyjna $6 \times 5 \times 5$,
2. warstwa poolingowa 2×2,
3. warstwa konwolucyjna $16 \times 5 \times 5$,
4. warstwa poolingowa 2×2,
5. warstwa gęsta o 120 neuronach,
6. warstwa gęsta o 84 neuronach,
7. warstwa gęsta o 10 neuronach.



Rysunek 2.1: Architektura konwolucyjnej sieci neuronowej LeNet-5 stworzonej do rozpoznawania cyfr [10].

Konwolucyjne sieci neuronowe CNN są dedykowane do zadań przetwarzania obrazów w takim zastosowaniu mają poważne przewagi nad standardowymi sieciami gęstymi MLP.

Interpretacja geometrycznych zależności - główna siła sieci CNN polega na tym, że mogą wyciągać informacje z wartości położenia piksela względem wartości pikseli z nim sąsiadujących. Tym sposobem można sprawdzić czy na obrazie widoczna jest np. pozioma linia. Zachowanie takie jest osiągnięte poprzez zastosowanie filtrów o trzech wymiarach, które są udoskonalane w trakcie uczenia. Dwa z tych wymiarów można określi i są to szerokość i wysokość filtru, a trzecim jest ilość kolorów np. 3 dla kolorów RGB. Dodatkowo pozwala to na uniezależnienie się od przesunięć na zdjęciu, co jest niemożliwe do osiągnięcia w sieciach MLP.

Dzielenie wag - wyżej wspomniane filtry, są przesuwane po całym obrazie w celu znalezienia zależności. W porównaniu do strategii przypisania jednego neuronu do jednej pozycji na zdjęciu, pozwala to na znaczne ograniczenie ilości wag w całej sieci tzn. uwspólnienie znaczącej ich części. Takie sieci szybciej się uczą, ze względu na mniejszą ilość wymiarów hiperpłaszczyzny, której minimum jest szukane. Dodatkowo są mniej podatne na przeuczenie.

Mimo, że konwolucyjne sieci neuronowe są przeznaczone przede wszystkim do pracy z obrazem, to w ostatnim czasie coraz więcej pojawia się prac, które w kreatywny sposób obchodzą to ograniczenie, np. przedstawiając dany sygnał jako obraz 2D [13].

W kolejnych podrozdziałach zostały opisane warstwy sieci neuronowych, z których składają się modele w tej pracy.

2.1 Max pooling

Działanie warstwy max poolingowej [6] polega na podpróbkowaniu (ang. subsampling) obrazu wejściowego w celu uzyskania reprezentacji wyjściowej posiadającej mniej wymiarów. Korzyściami płynącymi z takiego przetworzenia obrazu są:

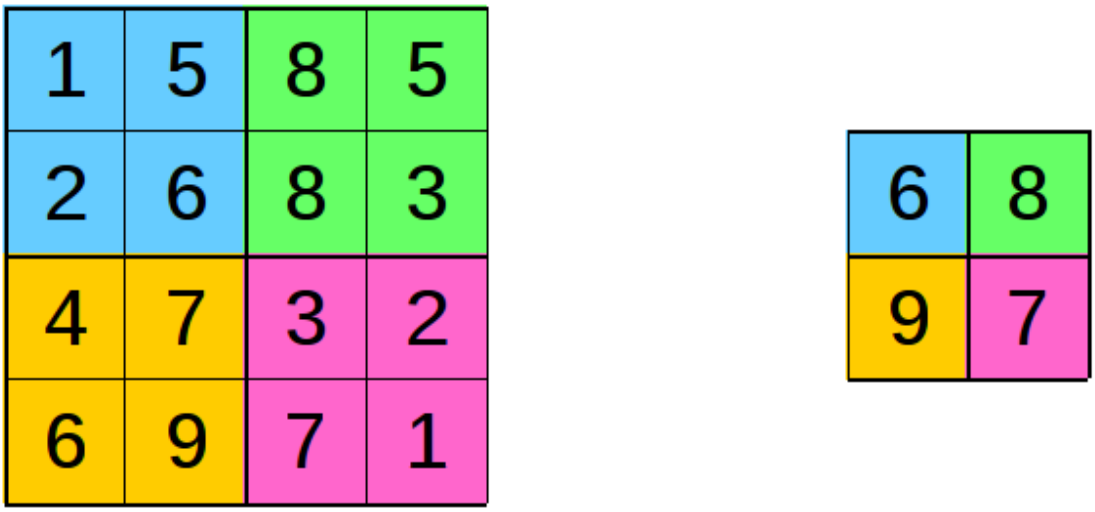
1. zmniejszenie podatności sieci na przeuczenie,
2. przyspieszenie procesu uczenia,
3. niewrażliwość sieci na lekkie rotacje i translacje obrazu,
4. zwiększenie zdolności generalizowania.

Podczas max poolingu obraz jest dzielony na n obszarów o rozmiarach $h \times w$, z których następnie wybierana jest wartość maksymalna znajdująca się w tym obszarze.

$$O(i,j) = max(I(k,l), I(k+1,l), ..., I(k+h-1,l), I(k,l+1), ..., I(k+h-1,l+w-1)), \quad (2.1)$$

gdzie $O(i,j)$ - wartość komórki obrazu wyjściowego w i-tym wierszu i j-tej kolumnie,
$I(k,l)$ - wartość komórki obrazu wejściowego w k-tym wierszu i l-tej kolumnie.

Kosztem wyżej wspomnianych zalet jest stracenie pewnej ilości informacji, dlatego należy być świadomym, że zbyt częsty pooling na zbyt małym obszarze może spowodować, że model będzie miał zbyt duże obciążenie (ang. bias) tj. sieć pozostanie niedouczona albo będzie potrzebowała większej ilości epok w procesie uczenia.



Rysunek 2.2: Wizualizacja działania warstwy max pooling [1].

2.2 Warstwa konwolucyjna

Warstwa konwolucyjna [6] powstała przy założeniu, że w danych wejściowych istnieją zależności między pewnymi wartościami i zależności te mają charakter geometryczny tj. jeśli potraktujemy dane wejściowe jako obraz, to widoczny jest na nim kształt. Założenia takie sugerują, aby do budowy sieci neuronowej wykorzystać techniki z dziedziny przetwarzania obrazów. Jedną z nich jest zastosowanie specjalnych masek lub jąder przekształcenia do wyciągnięcia ze zdjęcia takich informacji jak np. krawędzie lub usunięcie szumu. Sieci konwolucyjne biorą swoją nazwę z faktu, że maski nakłada się na obraz przy pomocy dyskretnej dwuwymiarowej operacji splotu:

$$O(i,j) = (I * H)(i,j) = \sum_{k=1}^{h} \sum_{l=1}^{w} H(k,l) I(i-k, j-l), \tag{2.2}$$

gdzie $O(i,j)$ - wartość komórki obrazu wyjściowego w i-tym wierszu i j-tej kolumnie,
$I(i,j)$ - wartość komórki obrazu wejściowego w i-tym wierszu i j-tej kolumnie,
H - maska,
h - wysokość maski,
w - szerokość maski.

Każda warstwa konwolucyjna składa się z zestawu jąder przekształcenia, które jednak nie są standardowymi maskami jak filtr Sobela [5], ale raczej odpowiadają neuronom w warstwie gęstej tj. są zmieniane podczas procesu uczenia, tak jak są zmieniane wagi przypisane konkretnym połączeniom. Z tego właśnie wynika największa siła sieci konwolucyjnych: potrafią one się nauczyć własnych filtrów, które są najlepiej dopasowane do danych.

Warstwa konwolucyjna ma wymiary $h \times w \times d \times n$, gdzie
h - wysokość maski,
w - szerokość maski,
d - głębokość maski,
n - ilość filtrów w warstwie.

Wartość parametru d musi być dopasowania do kształtu danych przychodzących do danej warstwy konwolucyjnej. Jeśli przetwarzany jest obraz w kolorach RGB filtr w pierwszej warstwie ma rozmiar $h \times w \times 3$. Jeśli tych filtrów jest 32, to filtry w warstwie drugiej będą miały kształt $h \times w \times 32$, aby móc jednocześnie przetwarzać wszystkie 32 obrazy utworzone przez poprzednią warstwę.

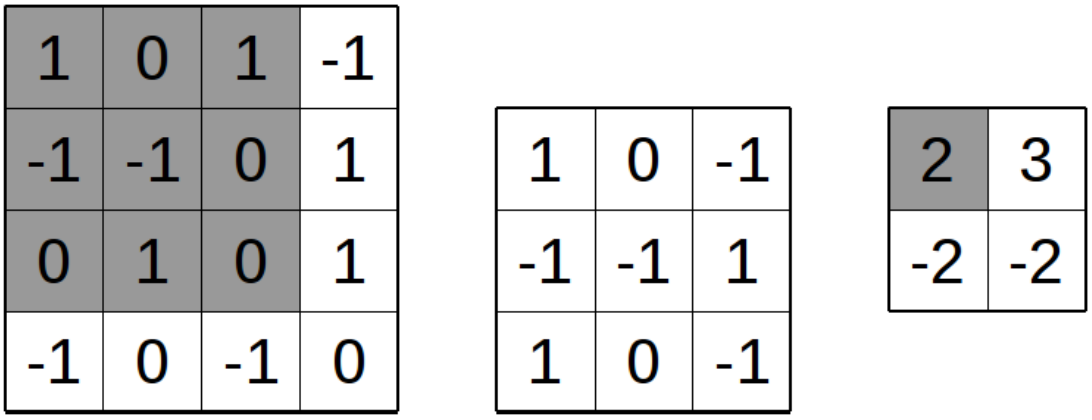
Podobnie jak w tradycyjnej dziedzinie przetwarzania obrazów jest jeszcze kilka parametrów startowych algorytmu (ang. hiperparameter), które trzeba wybrać. Pierwszym z nich jest krok filtru, czyli wartość mówiąca o ile kolumn lub wierszy filtr będzie przesunięty między kolejnymi operacjami splotu. Standardowa wartość to 1.

Drugim zagadnieniem jest sposób radzenia sobie z krawędziami. Zwykle przy zastosowaniu splotu obraz wyjściowy będzie pomniejszony w stosunku do wejściowego przy każdej krawędzi zgodnie z

$$L = floor(\frac{a}{2}), \tag{2.3}$$

gdzie L - strata przy jednej krawędzi w pikselach,
a - wysokość maski dla krawędzi dolnej i górnej oraz szerokość dla prawej i lewej.

Z powyższego wzoru wynika, że jeśli zostanie zastosowany filtr o wymiarach 3×3, to obraz wyjściowy będzie w pionie i poziomie mniejszy o $2L = floor(\frac{3}{2}) = 2$ piksele. Żeby zapobiec tej stracie można zastosować tzw. padding [5], czyli potraktowanie pikseli poza obrazem jakby miały jakąś wartość. Standardowymi rozwiązaniami jest przypisanie im wartości 0 lub odbicie lustrzane obrazu względem danej krawędzi.

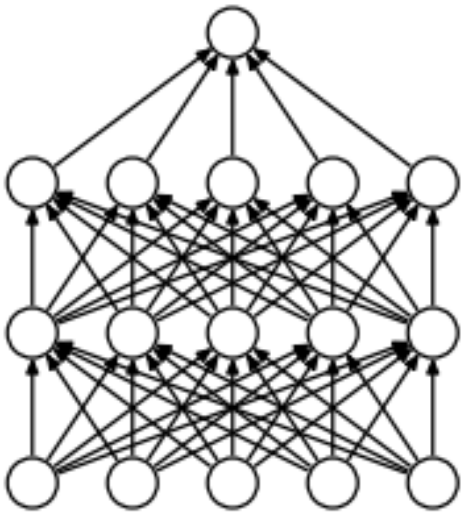


Rysunek 2.3: Schemat działania jednej maski w warstwie konwolucyjnej: obraz wejściowy, maska, obraz wyjściowy [1].

2.3 Warstwa gęsta

Warstwa gęsta to zwykła warstwa, z których składają się sieci MLP [6]. Każdy neuron znajdujący się w jednej warstwie jest połączony ze wszystkich neuronami znajdującymi się w kolejnej warstwie. Tak zbudowane sieci są w stanie modelować bardzo skomplikowane funkcje i mają różne zastosowanie. Można używać je np. w zadaniach regresji albo jako klasyfikatory. Ta uniwersalność jest jednak też ich wadą, ponieważ nie mają takich zalet w wąskiej dziedzinie jak sieci konwolucyjne w przetwarzaniu obrazów. Dodatkowo bardzo duża ilość połączeń między neuronami powoduje, że problem optymalizacji ma bardzo dużo wymiarów, przez co uczenie jest wolniejsze. Duża ilość wag też powoduje, że takie sieci są podatne na przeuczenie.

W sieciach konwolucyjnych warstwy gęste stosuje się dopiero na końcu sieci. Mają one za zadanie zaklasyfikować obraz na podstawie wszystkich cech, które wcześniej zostały wyekstrahowane z obrazu przez warstwy konwolucyjne. Często w ostatniej warstwie stosuje się softmax [6] jako funkcję aktywacji.



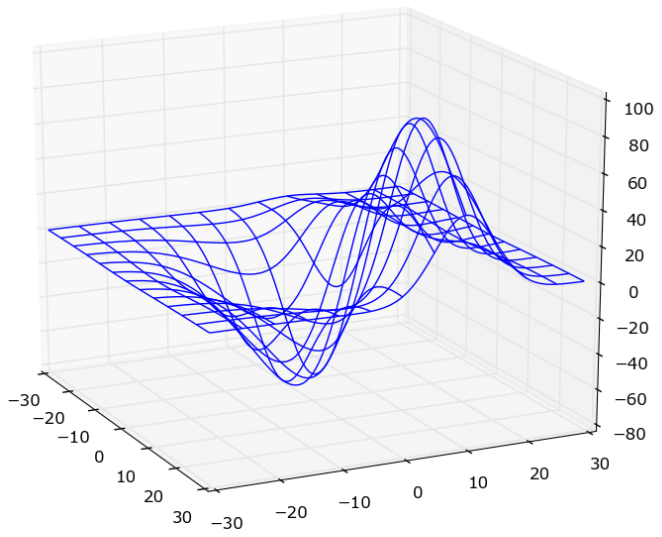
(a) Standard Neural Net

Rysunek 2.4: Sieć neuronowa składająca się z warstw gęstych [16].

Rozdział 3

Algorytmy optymalizacji

Proces uczenia sieci neuronowych tak naprawdę jest procesem optymalizacji, w którym celem jest znalezienie optimum funkcji celu. Funkcja ta nie jest podana wprost, ale to dane w zbiorze treningowym oraz uczony model wyznaczają jej kształt. W prostych przykładach można wyznaczyć minimum funkcji strat, w sposób analityczny, obliczając na podstawie ciągu uczącego dokładne wartości, dla których sieć popełnia na tym ciągu najmniejszy błąd. Jednak należy pamiętać, że ilość wymiarów przestrzeni, w której szukamy optimum zależy od ilości wag w sieci użytych w sieci neuronowej. Każda z nich rozpina kolejny wymiar, co sprawia, że nawet dla niewielkich sieci potrzeba bardzo dużej mocy obliczeniowej, aby wynik znaleźć w sposób analityczny. Z tego względu najpopularniejsze dziś są algorytmy iteracyjne pierwszego rzędu jak algorytm gradientu prostego. Dostępne są również metody np. drugiego rzędu, które potrafią znaleźć optimum w mniejszej liczbie iteracji, ale metody sprawiają, że każda z tych iteracji jest bardziej kosztowna obliczeniowo i w ostatecznym rozrachunku działają wolniej.



Rysunek 3.1: Przykładowa funkcja celu w przestrzeni trójwymiarowej.

W kolejnych podrozdziałach opisano trzy algorytmy, które zostały użyte w niniejszej pracy. Wszystkie są metodami pierwszego rzędu stworzonymi na podstawie algorytmu gradientu prostego.

3.1 Algorytm gradientu stochastycznego

Algorytm gradientu prostego [6], nazywany również algorytmem największego spadku, jest podstawowym algorytmem iteracyjnym używanym w problemach optymalizacji. Polega on na tym, że model może być przedstawiony w przestrzeni jako punkt na hiperpłaszczyźnie, która reprezentuje błąd sieci neuronowej w funkcji wartości wszystkich wag. W tym punkcie obliczane są pochodne cząstkowe funkcji strat względem każdej wagi, aby określić, w którym kierunku strata spada najszybciej. Następnie model wykonuje w tym kierunku "krok" tzn. wagi sieci neuronowej są zmieniane, aby zmniejszyć błąd.

Często używanym sposobem obliczania gradientu jest wzór:

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h}, \tag{3.1}$$

gdzie h - promień obszaru, w którym gradient jest obliczany.
Metoda ta jest kosztowniejsza obliczeniowo niż branie pod uwagę tylko otoczenia w jedną stronę, ale pozwala obliczyć gradient z większą dokładnością.

Algorytm gradientu prostego dokonuje zmian wag zgodnie ze wzorem:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t), \tag{3.2}$$

gdzie θ_t - macierz wag sieci połączeń w sieci neuronowej,
η - hiperparametr prędkości uczenia zadany na początku procesu nauczania,
$\nabla_\theta J(\theta_t))$ - gradient funkcji strat względem wag sieci.

Dostosowanie wag modelu może mieć miejsce, co epokę w procesie uczenia tzn. sieć neuronowa najpierw musi "zobaczyć" wszystkie dane w zbiorze treningowym i dopiero wtedy algorytm wstecznej propagacji koryguje wagi. Rozwiązanie takie ma tę wadę, że w uczeniu głębokim wykorzystuje się najczęściej bardzo duże zbiory danych, a, co za tym idzie, jedna epoka trwa przez długi czas. Dlatego robienie jednego "kroku" na epokę nie jest preferowanym sposobem rozwiązania tego problemu. Znacznie lepszą metodą jest korygowanie wag sieci, po każdej paczce (ang. batch) danych, licząc na to, że obliczone w ten sposób gradienty, po dodaniu do siebie i wyeliminowaniu w ten sposób szumu, doprowadzą model do optimum funkcji strat. Algorytm ten nazywa się metodą gradientu stochastycznego i to on został zastosowany w niniejszej pracy.

Algorytm spadku prostego zapewnia znalezienie lokalnego minimum. Może to być problemem, ale dowody empiryczne pokazują, że dla dużych sieci neuronowych różnica w wartości funkcji strat między minimum globalnym, a lokalnym jest mała [6]. Celem uczenia sieci nie jest zatem znalezienie globalnego optimum, ale lokalnego minimum, które jest wystarczająco do niego zbliżone.

3.2 Algorytm AdaGrad

Algorytm AdaGrad [3] ten jest modyfikacją algorytmu gradientu prostego. Zmiana polega na dodaniu parametru odpowiadającego skumulowanej sumie kwadratów gradientów

dla każdego wymiaru przestrzeni problemu. Na początku jest to wektor zer, a po każdej iteracji wartości zwiększane są o wartość gradientu odpowiadającego danej wadze. Następnie parametr prędkości uczenia sieci jest dzielony przez pierwiastek sumy kwadratów wag. Dzięki temu zabiegowi, algorytm potrafi dostosować swoją prędkość uczenia do przestrzeni, w której się znajduje. W kierunkach, w których funkcja strat zmienia się szybko gradient dzielony jest przez dużo wartość, a w kierunkach wolnozmiennych gradient dzielony jest przez małą wartość. W ten sposób długość "kroku", który jest wykonywany podczas uczenia jest w pewnym sensie normalizowany.

Wadą tego algorytmu jest to, że skumulowana suma kwadratów gradientów, może w pewnym momencie urosnąć tak bardzo, że prędkość uczenia będzie bliska zeru, a algorytm efektywnie przestanie się uczyć.

Poniżej przedstawiono wzory według, zgodnie z którymi działa algorytm. Najpierw obliczana jest nowa skumulowana suma kwadratów gradientów

$$S_t = S_{t-1} + (\nabla_\theta J(\theta_t))^2, \tag{3.3}$$

gdzie S_t - suma kwadratów gradientów w chwili t.
Następnie przy pomocy powyższej sumy wyznaczane są nowe wagi w sieci neuronowej.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{S_t + \epsilon}} \nabla_\theta J(\theta_t), \tag{3.4}$$

gdzie ϵ - wartość 10^{-8}, która zapobiega dzieleniu przez 0.

3.3 Algorytm Adam

Algorytm Adam [8] jest połączeniem algorytmu Adagrad z momentum oraz efektem zapominania skumulowanych kwadratów gradientów.

Momentum sprawia, że algorytm jest bardziej odporny na szum podczas uczenia tj. na sytuacje, w których gradient wskazuje kierunek, który nie jest optymalny globalnie. Ma to miejsce szczególnie, kiedy wagi aktualizowane są po każdej paczce danych, a nie całej epoce. Dodatkowo momentum sprawia, że algorytm "rozpędza się" w kierunku średnio wskazywanym przez gradienty i może to spowodować szybsze odnalezienie minimum funkcji strat.

Efekt zapominania jest remedium na wadą algorytmu Adagrad tj. na sytuację, w której zbyt duża suma kwadratów gradientów sprawia, że efektywna prędkość uczenia spada do zera. Rozwiązaniem tego problemu jest zastosowanie cieknącej sumy. Dzięki temu stare wartości mają mniejszy wpływ na wartość skumulowanej sumy i algorytm nie zatrzymuje się.

Poniżej przedstawiono wzory według, zgodnie z którymi działa algorytm. Najpierw obliczana jest nowa wartość momentum oraz nowa skumulowana suma kwadratów gradientów

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_\theta J(\theta_t), \tag{3.5}$$

gdzie m_t - wartość momentum w chwili t,
β_1 - hiperparametr odpowiedzialny za efekt zapominania momentum.

$$S_t = \beta_2 S_{t-1} + (1 - \beta_2)(\nabla_\theta J(\theta_t))^2, \tag{3.6}$$

gdzie β_2 - hiperparametr odpowiedzialny za efekt zapominania skumulowanej sumy kwadratów gradientów.

Następnie obliczane są nowe parametry sieci

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{S_t + \epsilon}} m_t \tag{3.7}$$

Wadą tego algorytmu jest potrzeba ręcznego ustawiania parametrów β_1 i β_2, które odpowiadają odpowiednio za efekt zapominania momentum i efekt zapominania skumulowane sumy kwadratów gradientów. Autorzy algorytmu proponują domyślne wartości tych parametrów jako $\beta_1 = 0.9$ i $beta_2 = 0.999$ [8]. Jednak, aby uzyskać wartości najlepsze dla danego problemu, wskazane jest sprawdzenie różnych wartości w procesie walidacji krzyżowej (ang. cross validation).

3.4 Momentum Nesterowa

Momentum Nesterowa [11] jest modyfikacją metody momentum, która w pewnych warunkach wykazuje lepszą zdolność do zbiegania do optimum niż oryginalny algorytm [17].

Zwykła metoda momentum działa przez dodanie wektora prędkości do gradientu z zadaną wagą i obliczenie w ten sposób nowym wag. Działanie to przedstawia się wzorem:

$$m_t = \mu m_{t-1} + \eta \nabla_\theta J(\theta_t), \tag{3.8}$$

$$\theta_{t+1} = \theta_t - m_t, \tag{3.9}$$

gdzie $\mu \in [0, 1]$ - współczynnik momentum.

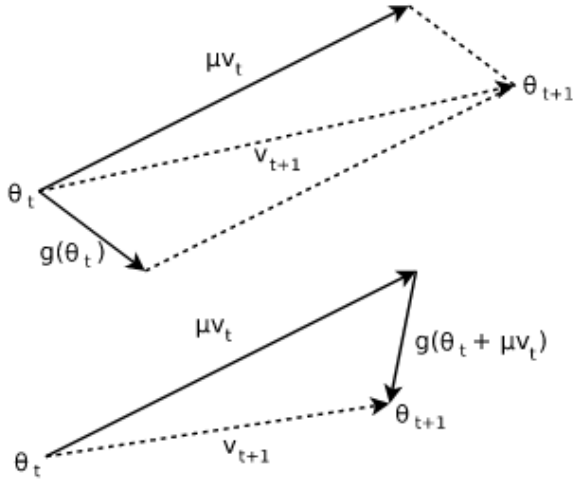
Modyfikacja zastosowana w momentum Nesterowa polega na tym, że gradientu do aktualizacji wag nie oblicza się w punkcie, w którym znajduje się algorytm w chwili t, ale jest on liczony w punkcie, w którym znalazłby się algorytm, gdyby poprawka polegała na wzięciu pod uwagę tylko wartości momentum.

$$m_t = \mu m_{t-1} + \eta \nabla_\theta J(\theta_t - \eta m_{t-1}) \tag{3.10}$$

$$\theta_{t+1} = \theta_t - m_t \tag{3.11}$$

Wadą tego rozwiązania jest to, że jest bardziej kosztowna obliczeniowo, co jest istotnym czynnikiem, kiedy proces uczenia trwa wiele iteracji.

W niniejszej pracy wykorzystano algorytm NAdam [2] tj. modyfikację algorytmu Adam, w której zwykłe momentum zostało zastąpione przez momentum Nesterova.

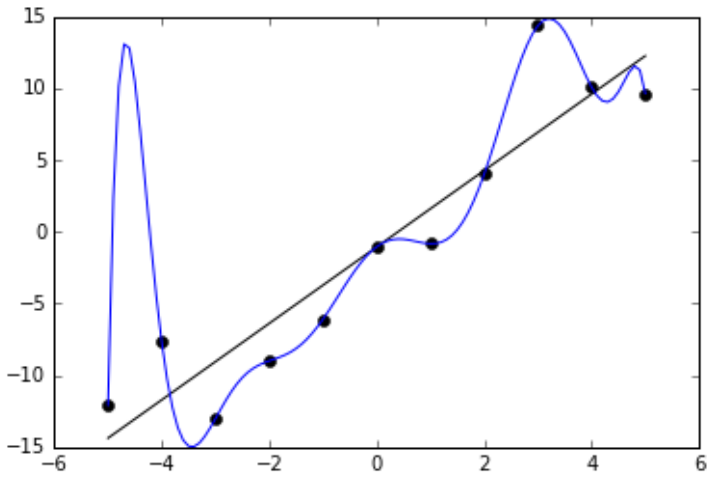


Rysunek 3.2: Działanie metody momentum **(góra)** oraz momentum Nesterowa **(dół)** [17].

Rozdział 4

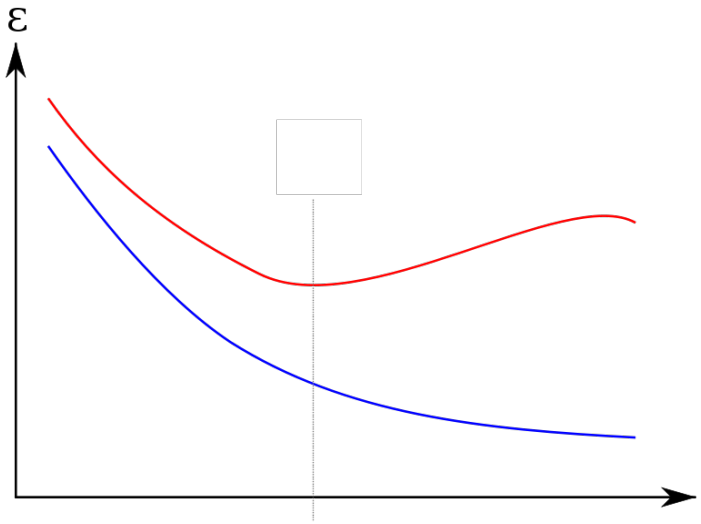
Metody zapobiegania przeuczeniu

Przeuczenie (ang. overfitting) w uczeniu maszynowym to zjawisko kiedy model matematyczny zbyt wiernie odwzorowuje dane treningowe i traci zdolność generalizacji. W takich przypadkach model zaczyna dopasowywać się do szumu, który jest w danych, zamiast do ogólnego trendu. Najczęstszą przyczyną tego problemu jest zbyt duża ilość parametrów, które są zmieniane podczas uczenia w stosunku to liczebności zbioru treningowego. Przeuczenie często występuje podczas uczenia sieci neuronowych, ponieważ w sieciach MLP ilość wag rośnie bardzo szybko, kiedy dokładane są kolejne neurony lub kiedy dodawane są kolejne warstwy. Z tego względu należy zachować szczególną ostrożność, gdy stosuje się sieci neuronowe do problemów, w których jest mało danych uczących.



Rysunek 4.1: Porównanie modelu rzeczywistego (czarny) i przeuczonego (niebieski) do danych treningowych [18].

Jednym ze sposobów na ocenienie czy sieć neuronowa została przeuczona jest wydzielenie zbioru treningowego, który będzie użyty do uczenia modelu, oraz testowego, dla którego zostanie obliczony błąd popełniany przez sieć po każdej epoce uczenia. Dysponując wykresami błędu na zbiorze treningowym i błędu na zbiorze testowym w funkcji ilości epok, można wyznaczyć miejsce, w którym błąd testowy osiąga swoje minimum i od tego miejsca już rośnie, podczas gdy błąd treningowy dalej spada. Jest to dowód na to, że większa ilość epok prowadzi do przeuczenia i model nie powinien być tak długo uczony. Celem jest tutaj osiągnięcie kompromisu między obciążeniem klasyfikatora, a jego wariancją.



Rysunek 4.2: Wykres wartości błędu treningowego (niebieski) oraz testowego (czerwony) w funkcji ilości epok [19].

Skrajnym przypadkiem przeuczenia jest model, który był trenowany tak długo, aż błąd, który popełnia na zbiorze uczącym jest zerowy. Sytuacja taka może się wydawać pożądaną, ale prawdopodobnie model ten popełni duży błąd na rzeczywistych danych. Nie jest to jednak regułą. Od specyfiki problemu, który jest rozwiązywany, zależy czy przeuczenie będzie miało znaczny wpływ na wynik np. klasyfikacji. Możliwe jest, że dla danego problemu nawet dla dużych sieci i małych zbiorów danych nie będzie można mówić o przeuczeniu ze względu na jego znikomy wpływ na skuteczność.

4.1 Dropout

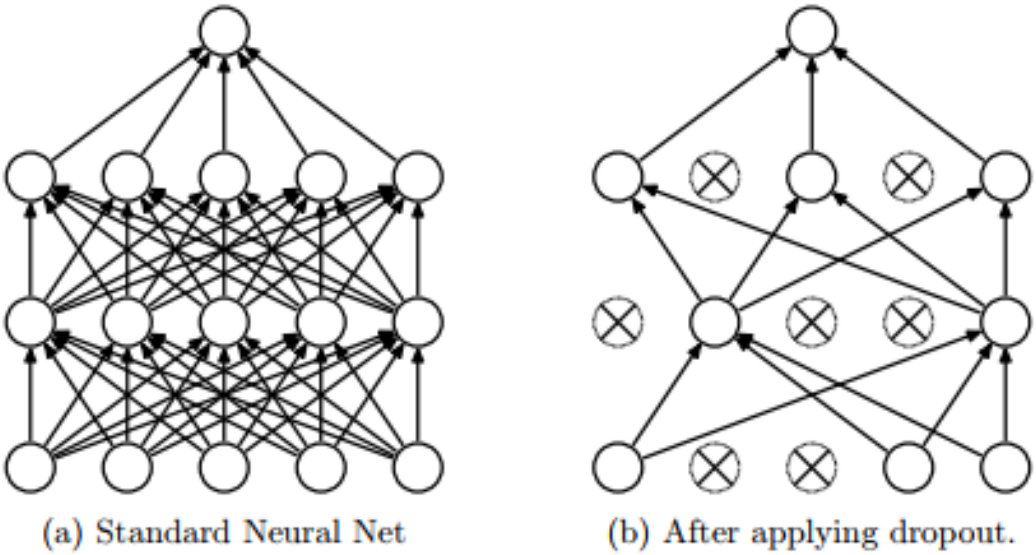
Dropout [16] jest to jedna z metod regularyzacji tj. przeciwdziałania przeuczeniu i zwiększania zdolności generalizacji sieci. Używa się go w procesie uczenia sieci neuronowej. Dla każdej warstwy sieci neuronowej należy wybrać, jaka ich część nie będzie aktywna w danej epoce lub dla danej paczki danych (ang. batch). Nieaktywne neurony na swoim wyjściu zawsze podają wartość równą 0. Dzięki temu nie mają wpływu na decyzje sieci dla danego przykładu uczącego i, w związku z tym, ich wagi nie są zmieniane podczas propagacji wstecznej. Nieaktywne neurony są efektywnie usunięte z sieci neuronowej.

Intuicyjnie dropout można zrozumieć kilka sposobów. Poniżej przedstawione są dwa z nich.

Zespół klasyfikatorów - za każdym razem kiedy część neuronów jest ”wyrzucana” z sieci, prowadzi to efektywnie do powstania innej sieci neuronowej o innej architekturze, ale która mimo wszystko współdzieli wagi z innymi sieciami powstałymi w ten sposób. Wszystkie stworzone tak sieci głosują, a ich siły wsparcia są uśredniane i na tej podstawie obliczana jest końcowa decyzja. Uwspólnianie wag jest samo w sobie metodą regularyzacji, a zastosowanie zespołu klasyfikatorów to metoda, która często pozwala otrzymać lepsze efekty niż przy zastosowaniu pojedynczego klasyfikatora.

Nie poleganie na jednej cesze - podczas uczenia sieci neuronowej może zdarzyć się, że dużą część danych uczących da się odpowiednio zaklasyfikować polegając przede wszystkim na tylko jednej z cech zawartych w tym przykładzie. W takim przypadku

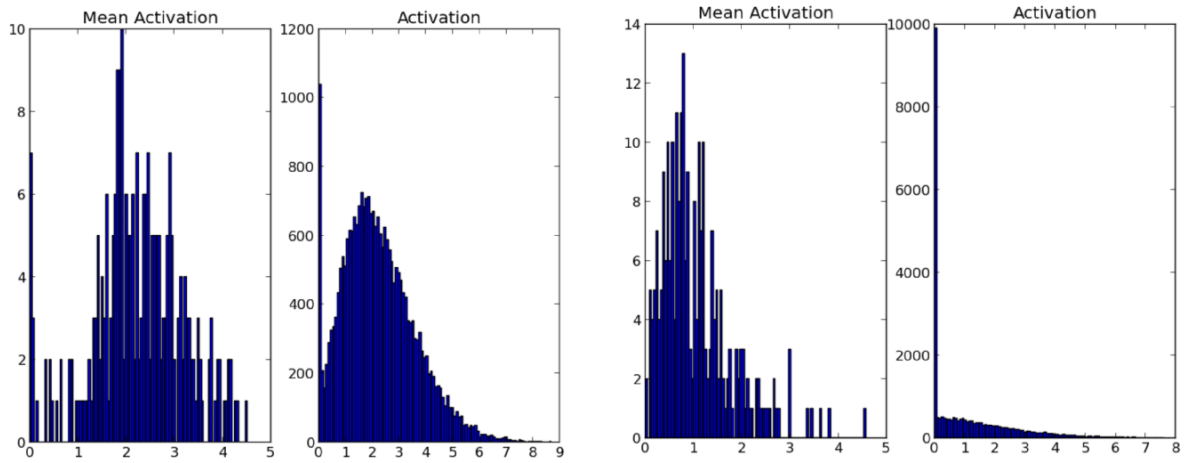
istnieje ścieżka od wejścia do wyjścia sieci, która składa się z neuronów wyspecjalizowanych w wyciąganiu wniosków właśnie na tej podstawie. Może to być problemem, kiedy w danych rzeczywistych ta cecha czasami się nie powtarza. Podczas uczenia z użyciem dropout'u taka sytuacja nie powinna wystąpić, ponieważ każdy z neuronów z danej ścieżki może zostać dezaktywowany w danej części procesu. Sprawia to, że sieć neuronowa wykształca inne możliwości rozpoznawania danych wejściowych, które opierają się też na innych cechach. Cechy te są redundantne i dzięki temu nie każda z nich musi zostać wykryta w danych wejściowych, żeby klasyfikacja była poprawna.



Rysunek 4.3: Porównanie sieci neuronowej i sieci z zastosowanym dropout'em **a)** Standardowa sieć neuronowa z dwiema warstwami ukrytymi. **b)** Ta sama sieć po zastosowaniu dropout'u. Przekreślone neurony zostały odrzucone [16].

Ważnym aspektem dropout'u jest fakt, że po fazie treningowej, kiedy rozpatrywana jest cała sieć ze wszystkimi neuronami aktywnymi, wsparcia zwracane przez tę sieć będą większe niż podczas treningu, ze względu na większą liczbę neuronów biorących w procesie. Należy pamiętać o tym, aby przeskalować wsparcia, które daje sieć. W przypadku, w którym w procesie uczenie 50% neuronów było nieaktywnych, wartości wyjściowe neuronów będą dwukrotnie większe niż powinny, ponieważ dociera do nich dwa razy więcej sygnałów wejściowych. Przy użyciu tej sieci należy podzielić wartości wsparć przez 2.

Interesującym efektem ubocznym stosowania dropout'u jest rzadkość aktywacji neuronów. Neurony zdają się specjalizować w rozpoznawaniu pewnych cech, pozostając nieczułymi na inne, co można odczytać jako bardziej dogłębne nauczenie sieci. Na rysunku 4.4 na pierwszym i trzecim histogramie przedstawiono średnie wartości aktywacji, a na drugim i czwartym ilość neuronów, które osiągnęły daną wartość aktywacji. Można zauważyć, że średnia wartość aktywacji jest mniejsza dla sieci z zastosowanym dropout'em oraz, że w takiej sieci bardzo dużo neuronów słabo reaguje na dane wejściowe, a jedynie wyspecjalizowane neurony mają wysokie wartości aktywacji.

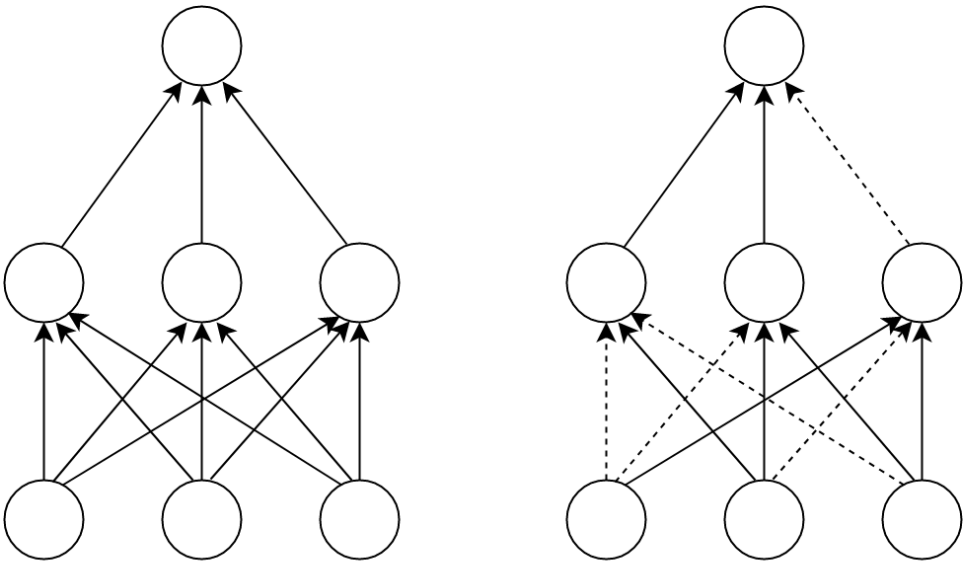


(a) Without dropout (b) Dropout with $p = 0.5$.

Rysunek 4.4: Porównanie histogramów aktywacji neuronów w **a)** zwykłej sieci neuronowej i **b)** sieci z zastosowanym dropout'em z dezaktywacją 50% neuronów [16].

4.2 Wymrażanie

Wymrażanie [14] jest to metody regularyzacji, która polega na wybraniu podzbioru wag sieci, które w danej iteracji będą zmieniane tj. w przeciwieństwie do normalnego procesu uczenia nie wszystkie wagi są aktualizowane zgodnie z obliczonymi przez algorytm optymalizacji poprawkami, ale część z nich jest "zamrożona" podczas danej epoki. Losowe wybieranie części parametrów, które pozostają stałe sprawia, że do sieci neuronowej wprowadzone jest szum, który zapobiega zjawisku przeuczenia oraz zwiększa zdolności generalizacji.



a) Standardowa sieć neuronowa b) Sieć neuronowa po zastosowaniu wymrażania

Rysunek 4.5: Porównanie sieci neuronowej i sieci z zastosowanym wymrażaniem **a)** Standardowa sieć neuronowa z jedną warstwą ukrytą. **b)** Ta sama sieć po zastosowaniu wymrażania. Połączenia zaznaczone przerywaną linią nie ulegają aktualizacji.

Można zauważyć pewne podobieństwa między wymrażaniem, dropout'em. W obu do sieci wprowadzana jest losowość podczas procesu uczenia i w obu tylko pewien podzbiór wag jest zmieniany w danej iteracji. Poniżej opisano dwie różnice.

1. W dropoucie istnieje bardzo konkretny sposób wyboru wag, które nie będą aktualizowane w danej części treningu. Kiedy neuron zostaje dezaktywowany to wszystkie wagi wchodzące tj. łączące ten neuron z neuronami poprzedniej warstwy, oraz wszystkie wagi wychodzące tj. łączące ten neuron z neuronami warstwy następnej, pozostają stałe. Zostają one wybierane konkretnymi grupami. W metodzie wymrażania podzbiór zamrożonych wag jest wybierany losowo i dowolnie ze wszystkich dostępnych i, tak jak pokazano na rysunku 4.5, może się zdarzyć, że tylko część połączeń wchodząc i wychodzących z danego neuronu będzie zauktualizowanych. Pod tym względem można uznać wymrażanie za uogólnienie dropout'u.

2. Wartości wyjściowe sieci w trakcie uczenia są różne w obu algorytmach, ze względu na to, że każdy przykład uczący jest rozpoznawany przez inny zbiór wag. W dropoucie przy zadanej ilości odrzucanych neuronów (np. 20%) za każdym razem dane treningowe rozpoznawane są wyłącznie przez pozostałe neurony. W metodzie wymrażania zawsze cała sieć bierze udział w podejmowaniu decyzji, ale wyłącznie jej część (np. 80% wag) jest aktualizowanych zgodnie z wyliczonym błędem.

Algorytm wymrażania oferuje też możliwość optymalizacji ilości obliczeń, a co za tym idzie przyspieszenia uczenia. Ponieważ znacząca część wag nie jest aktualizowana za każdym razem, obliczanie poprawek dla nich jest zbędne i można je pominąć.

W dalszej części pracy zostały przedstawione wyniki badań, które porównują wpływ dropout'u i wymrażania na jakość klasyfikacji, predykcji oraz jak skutecznie zapobiegają przeuczeniu.

Rozdział 5

Implementacja

Część implementacyjna niniejszej pracy została wykonana w języku Python 3.7. Do zbudowania i nauczenia sieci neuronowej posłużyła biblioteka Keras, która udostępnia metody uczenia głębokiego. Keras wykorzystuje skompilowaną bibliotekę Tensorflow m.in. w celu optymalizacji operacji na macierzach.

Biblioteka Keras udostępnia klasę *Sequential*, która pozwala w prosty sposób budować sieci neuronowe typu feedforward tj. takie, w których dane są przesyłane wyłącznie z warstwy poprzedniej do kolejnej w kierunku od wejścia do wyjścia sieci.

```
1  model = Sequential()
2  model.add(Conv2D(32, kernel_size=(3, 3),
3                   activation='relu',
4                   input_shape=input_shape))
5  model.add(MaxPooling2D(pool_size=(2,2)))
6  model.add(Dropout(0.25))
7  model.add(Conv2D(32, (3, 3), activation='relu'))
8  model.add(MaxPooling2D(pool_size=(2, 2)))
9  model.add(Dropout(0.25))
10 model.add(Flatten())
11 model.add(Dense(num_classes, activation='softmax'))
12
13 model.compile(loss=keras.losses.categorical_crossentropy,
14               optimizer=keras.optimizers.Nadam(),
15               metrics=['accuracy'])
```

Listing 1: Budowa sieci neuronowej przy użyciu klasy *Sequential* z pakietu Keras.

Na listingu 1 przedstawiono, jak zbudować przykładową sieć neuronową, używając klasę *Sequential*. Warto zwrócić uwagę na łatwość dodawania kolejnych warstw.

Wersja algorytmu dropout zaimplementowana w bibliotece Keras ma postać warstwy sieci neuronowej, która przepuszcza lub blokuje dane, które chcą przez nią przejść. Zostało to pokazane w liniach 6. i 9. listingu 1.

Algorytm optymalizacji, który zostanie użyty podczas uczenia sieci neuronowej jest wybierany w funkcji *model.compile*() po sprecyzowaniu architektury sieci. W tej funkcji zostały również określone metryka i funkcja strat.

5.1 Klasa FreezingSequential

Wymrażanie nie ma swojej dostępnej implementacji, dlatego, w odróżnieniu od pozostałych algorytmów, została zaimplementowana w ramach niniejszej pracy. Na listingu 2 przedstawiono implementację tej metody, a na listingu 3 znajduje się kod budujący sieć neuronową podobną do tej na listingu 1, ale używającą wymrażania 25% w miejsce dropou't 25%.

Wymrażanie zostało zaimplemetowane jako algorytm działający na wszystkich wagach w sieci jednakowo. Jest to rozbieżność z implementacją dropout'u, którego używa się na wzór warstw sieci. Zamrażanie wag tylko w jednej warstwie byłoby niezgodne z algorytmem jak został przedstawiony w [14], ale mogłoby sprawić, że szum byłby rozkładany bardziej regularnie w całej sieci.

Klasa *FreezingSequential* jako parametr konstruktora przyjmuje wartość odpowiadającą procentowi wag, które mają być każdorazowo zamrożone. Odpowiada to parametrowi dla dropout'u przekazywanemu da każdej warstwy.

Wagi, które są zamrożone są zmieniane dla każdej kolejnej paczki danych, a w razie niepodania rozmiaru paczki dla każdej epoki.

```
class FreezingSequential(Sequential):

def __init__(self, percent_of_frozen_neurons):
  super().__init__()
  self.percent_of_frozen_neurons = percent_of_frozen_neurons

def fit(self, x=None, y=None, batch_size=32, epochs=1,
    verbose=1, callbacks=None, validation_split=0.0,
    validation_data=None, shuffle=True, class_weight=None,
    sample_weight=None, initial_epoch=0, **kwargs):
  size = len(y)
  idxs = np.random.permutation(size)
  if batch_size == None:
    batch_size = size
  for e in range(epochs):
    x,y = x[idxs], y[idxs]
    batch_epochs = int(np.ceil(size / batch_size))

    for be in range(batch_epochs):
      batch_xs = x[be*batch_size : (be+1)*batch_size]
      batch_ys = y[be*batch_size : (be+1)*batch_size]
      old_weights, masks = self.pick_old_weights_and_masks(self.get_weights())
      super().fit(batch_xs, batch_ys, batch_size, 1, 0, callbacks,
                validation_split, validation_data, shuffle,
                class_weight, sample_weight, initial_epoch)
      self.set_weights(self.get_new_weights(self.get_weights(), old_weights, masks))

def pick_old_weights_and_masks(self, weights):
  old_weights = []
  masks = []
  for matrix in weights:
    mask = np.random.rand(*matrix.shape) < self.percent_of_frozen_neurons
    old_weights.append(matrix[mask])
    masks.append(mask)
  return old_weights, masks

def get_new_weights(self, current_weights, old_weights, masks):
  new_weights = []
  for current, old, mask in zip(current_weights, old_weights, masks):
    current[mask] = old
    new_weights.append(current)
  return new_weights
```

Listing 2: Klasy *FreezingSequential*, która implemetuje metodę wymrażania. W celu zwiększenia czytelności część kodu odpowiadająca za logowanie postępów uczenia została usunięta.

```
model = FreezingSequential(.25) # zamrażanie 0.25
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Nadam(),
              metrics=['accuracy'])
```

Listing 3: Modyfikacja sieci z listingu 1 używająca wymrażania zamiast dropout'u.

Rozdział 6

Badania i wyniki

Badania w niniejszej pracy zostały przeprowadzone dla 3 baz danych wybranych tak, aby zadanie było inne:

1. MNIST - klasyfikacja, sieć MLP,
2. CIFAR-10 - klasyfikacja, sieć konwolucyjna,
3. Boston Housing Prices - regresja, sieć MLP.

Dla każdej bazy zostały przygotowanie 2 architektury sieci neuronowych: prosta oraz złożona. Zrobiono tak, aby sprawdzić skuteczność metod przeciwdziałania przeuczeniu.

Metryką na podstawie, której oceniona została jakość klasyfikacji jest skuteczność (ang. accuracy). Została ona obliczona po każdej epoce uczenia tzn. po tym, jak sieć "zobaczyła" wszystkie przykłady z ciągu uczącego. Każda epoka składała się z paczek po 128 przykładów, które zostały wylosowane bez zwracania z ciągu uczącego.

Algorytmy optymalizacji wykorzystane w badaniach to:

1. Adagrad,
2. Nadam,
3. SGD.

Dla każdej kombinacji architektury i algorytmu optymalizacji wykonano badania dla każdej z poniższych konfiguracji:

1. sieć bez regularyzacji,
2. wymrażania 10%,
3. wymrażania 20%,
4. wymrażania 30%,
5. wymrażania 50%,
6. dropout 10%,
7. dropout 20%,
8. dropout 30%,

9. dropout 50%.

W celu zapewnienia tego, że uzyskane wyniki są reprezentatywne zastosowano 5-krotną walidację krzyżową. Została ona wybrana zamiast 10-krotnej walidacji, która jest zwykle używana, ze względu na zbyt duży koszt obliczeniowy takiego rozwiązania.

Sieci neuronowe zaprezentowane w kolejnych podrozdziałach używają ReLu jako funkcji aktywacji w warstwach ukrytych oraz funkcji softmax w warstwie wyjściowej dla zadania klasyfikacji. W nawiasach podano rozmiar warstwy.

W tabelach, w których porównane zostały wyniki uczenia sieci neuronowych odchylenie standardowe obliczone zgodnie ze wzorem

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}, \tag{6.1}$$

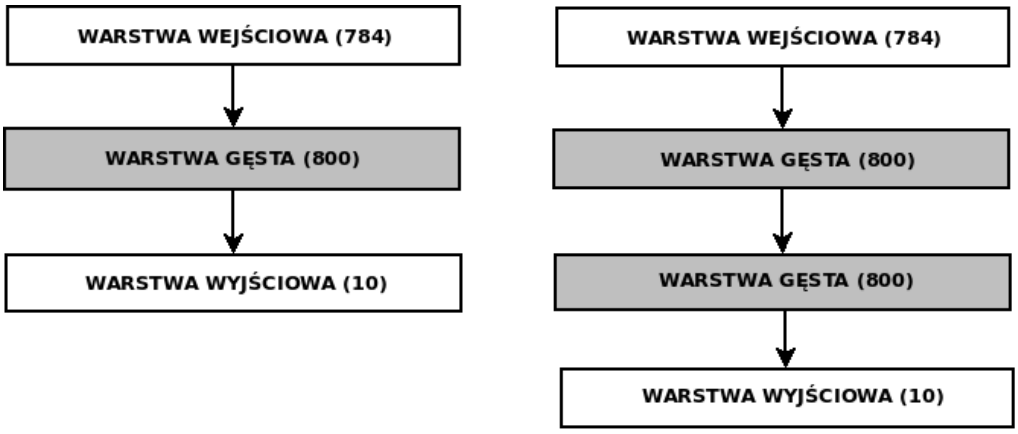
gdzie N - ilość powtórzeń podczas walidacji krzyżowej,
x_i - wynik w i-tym powtórzeniu,
$\bar{x}$ - średnia N wyników.

Jest to istotne ze względu na małe N, które w tym wypadku wynosi 5, a zastosowanie dzielenia przez N zamiast przez N-1 mocno zmieniłoby wynik.

6.1 MNIST

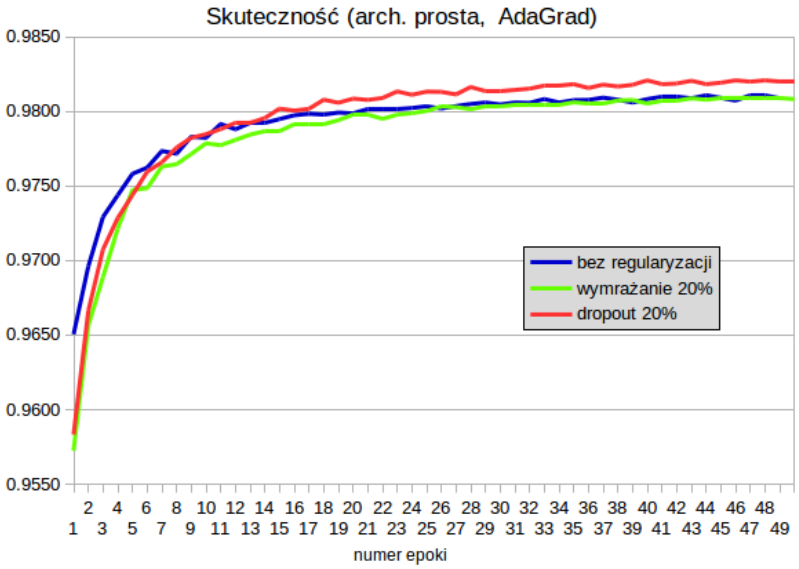
Baza MNIST [10] zawiera 70 tys. czarno-białych obrazów o rozmiarach 28×28, które przedstawiają ręcznie pisane cyfry. Zadaniem jest zaklasyfikowanie obrazu to jednej z 10 kategorii.

Prostą wersję architektury zaczerpnięto z pracy [15].



Rysunek 6.1: Architektura sieci neuronowych wykorzystanych w eksperymentach na bazie MNIST. Od lewej odpowiednio prosta i złożona.

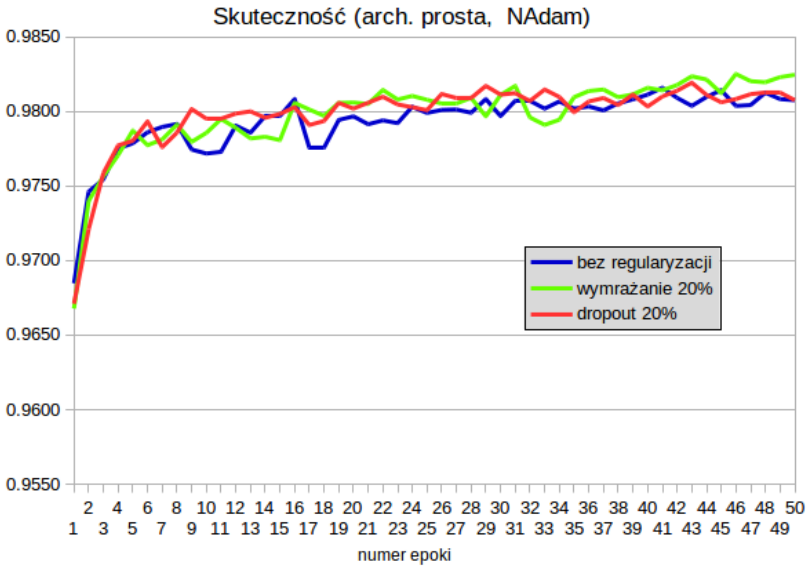
6.1.1 Architektura prosta



Rysunek 6.2: Porównanie skuteczności klasyfikacji dla architektury prostej i algorytmu AdaGrad. Dla sieci bez regularyzacji, z wymrażaniem 20% oraz z dropout'em 20%.

Tablica 6.1: Porównanie ostatecznej skuteczności dla architektury prostej i algorytmu AdaGrad.

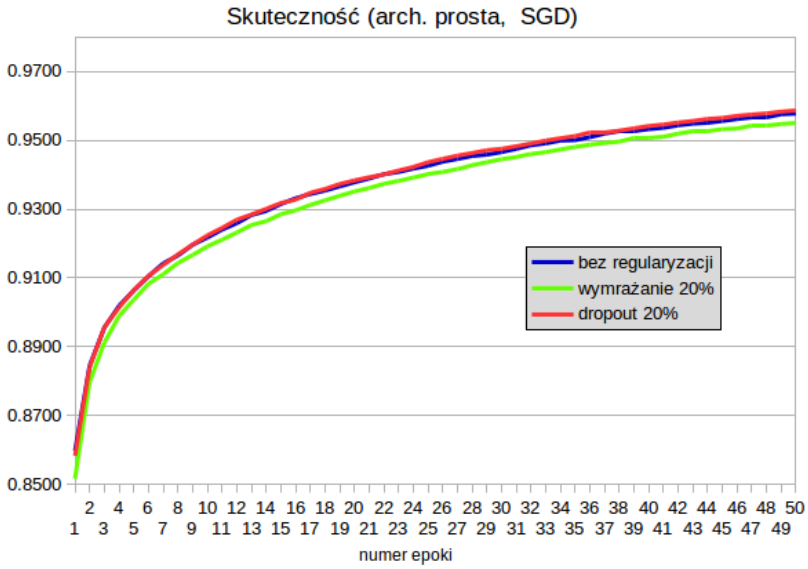
	bez regularyzacji	wymrażanie 20%	dropout 20%
średnia	0,9809	0,9803	0,9819
odchylenie standardowe	0,0019	0,0018	0,0018



Rysunek 6.3: Porównanie skuteczności klasyfikacji dla architektury prostej i algorytmu NAdam. Dla sieci bez regularyzacji, z wymrażaniem 20% oraz z dropout'em 20%.

Tablica 6.2: Porównanie ostatecznej skuteczności dla architektury prostej i algorytmu NAdam.

	bez regularyzacji	wymrażanie 20%	dropout 20%
średnia	0,9808	0,9818	0,9817
odchylenie standardowe	0,0026	0,0031	0,0025



Rysunek 6.4: Porównanie skuteczności klasyfikacji dla architektury prostej i algorytmu SGD. Dla sieci bez regularyzacji, z wymrażaniem 20% oraz z dropout'em 20%.

Tablica 6.3: Porównanie ostatecznej skuteczności dla architektury prostej i algorytmu SGD.

	bez regularyzacji	wymrażanie 20%	dropout 20%
średnia	0,9578	0,9535	0,9594
odchylenie standardowe	0,0040	0,0043	0.,0046

Powyższe badania wykonano dla bazy MNIST i sieci o architekturze prostej przedstawionej na rysunku 6.1. Można z nich wyciągnąć następujące wnioski i spostrzeżenia:

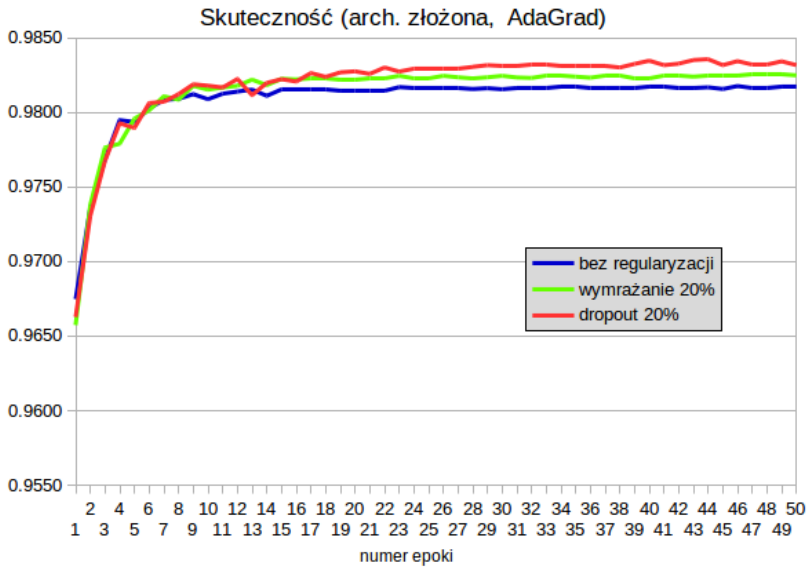
1. 50 epok jest zbyt krótkim czasem uczenia, dla algorytmu SGD i dla tego problemu, ponieważ błąd testowy cały czas miał mocną tendencję spadkową. Jest to zgodne z oczekiwaniami, że SGD będzie najwolniejszym z testowanych algorytmów.

2. Średnie wyników dla wszystkich metod są podobne, a odchylenie standardowe jest niskie.

3. Skuteczność sieci jest bardzo wysoka i sugeruje to, że ten stopień złożoności architektury jest wystarczający dla tego problemu.

4. Najlepszy wynik dla bazy MNIST został uzyskany przez sieć neuronową o architekturze prostej, z algorytmem NAdam i wymrażaniem 50%. Uzyskany poziom błędu to 1,6%, co jest na podobnym poziom co 1,25% [16] osiągnięte w literaturze.

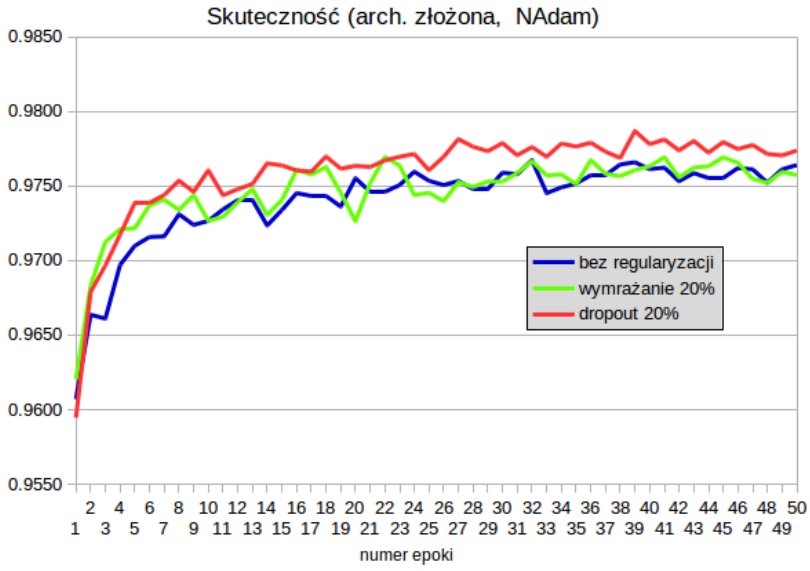
6.1.2 Architektura złożona



Rysunek 6.5: Porównanie skuteczności klasyfikacji dla architektury złożonej i algorytmu AdaGrad. Dla sieci bez regularyzacji, z wymrażaniem 20% oraz z dropout'em 20%.

Tablica 6.4: Porównanie ostatecznej skuteczności dla architektury złożonej i algorytmu AdaGrad.

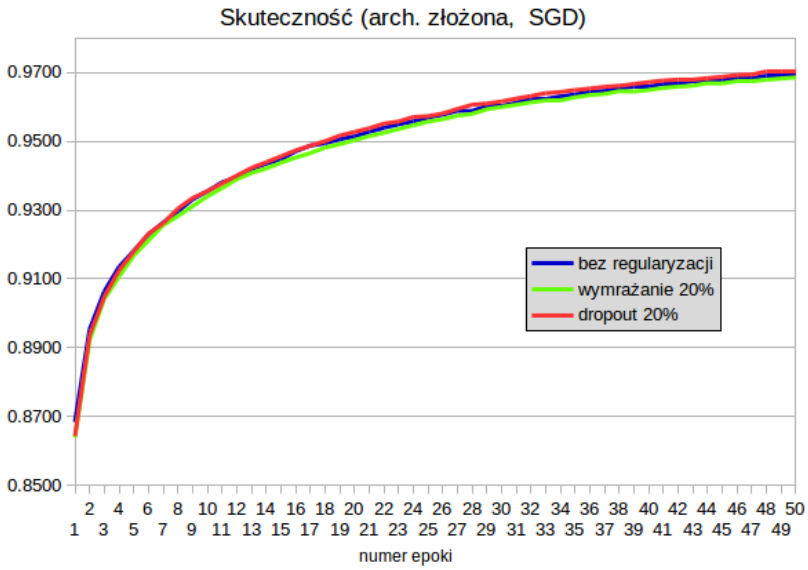
	bez regularyzacji	wymrażanie 20%	dropout 20%
średnia	0,9818	0,9803	0,9840
odchylenie standardowe	0,0015	0,0023	0,0019



Rysunek 6.6: Porównanie skuteczności klasyfikacji dla architektury złożonej i algorytmu NAdam. Dla sieci bez regularyzacji, z wymrażaniem 20% oraz z dropout'em 20%.

Tablica 6.5: Porównanie ostatecznej skuteczności dla architektury złożonej i algorytmu NAdam.

	bez regularyzacji	wymrażanie 20%	dropout 20%
średnia	0,9764	0,9775	0,9778
odchylenie standardowe	0,0030	0,0036	0,0017



Rysunek 6.7: Porównanie skuteczności klasyfikacji dla architektury złożonej i algorytmu SGD. Dla sieci bez regularyzacji, z wymrażaniem 20% oraz z dropout'em 20%.

Tablica 6.6: Porównanie ostatecznej skuteczności dla architektury złożonej i algorytmu SGD.

	bez regularyzacji	wymrażanie 20%	dropout 20%
średnia	0,9696	0,9668	0,9700
odchylenie standardowe	0,0033	0,0031	0,0036

Powyższe badania wykonano dla bazy MNIST i sieci o architekturze złożonej przedstawionej na rysunku 6.1. Można z nich wyciągnąć następujące wnioski i spostrzeżenia:

1. Średnie ostatecznych wyników dla wymrażanie, dropout'u i braku regularyzacji są bardzo podobne, a odchylenie standardowe jest niskie.

2. Algorytm gradientu stochastycznego jest najwolniejszy z zaprezentowanych i 50 epok jest zbyt krótkim czasem dla pełnego nauczenia sieci.

3. Sieci o architekturze złożonej są lepsze od tych o architekturze prostej tj. uczą się szybciej, a w przypadku SGD uzyskują dzięki temu lepszą skuteczność ostateczną.

4. Fakt, że najlepszą ze wszystkich sieci była sieć o architekturze prostej sugeruje, że zastosowana architektura złożona jest nadmiarowo skomplikowana w stosunku do rozwiązywanego problemu.

6.1.3 Porównanie szybkości uczenia

W poniższych tabelach uśredniono ilość epok potrzebną do osiągnięcia 99% końcowej skuteczności dla 5-krotnej walidacji, architektury prostej i złożonej oraz dla różnych ustawień wymrażania i dropout'u tj. 10%, 20%, 30% i 50%.

Tablica 6.7: Porównanie średniej ilości epok, które potrzebne były aby osiągnąć 99% końcowej skuteczności dla algorytmu AdaGrad.

	bez regularyzacji	wymrażanie	dropout
średnia	3,00	3,50	4,13
odchylenie standardowe	1,41	1,85	1,25

Tablica 6.8: Porównanie średniej ilości epok, które potrzebne były aby osiągnąć 99% końcowej skuteczności dla algorytmu NAdam.

	bez regularyzacji	wymrażanie	dropout
średnia	2,50	2,00	2,75
odchylenie standardowe	0,71	0,00	1,04

Tablica 6.9: Porównanie średniej ilości epok, które potrzebne były aby osiągnąć 99% końcowej skuteczności dla algorytmu SGD.

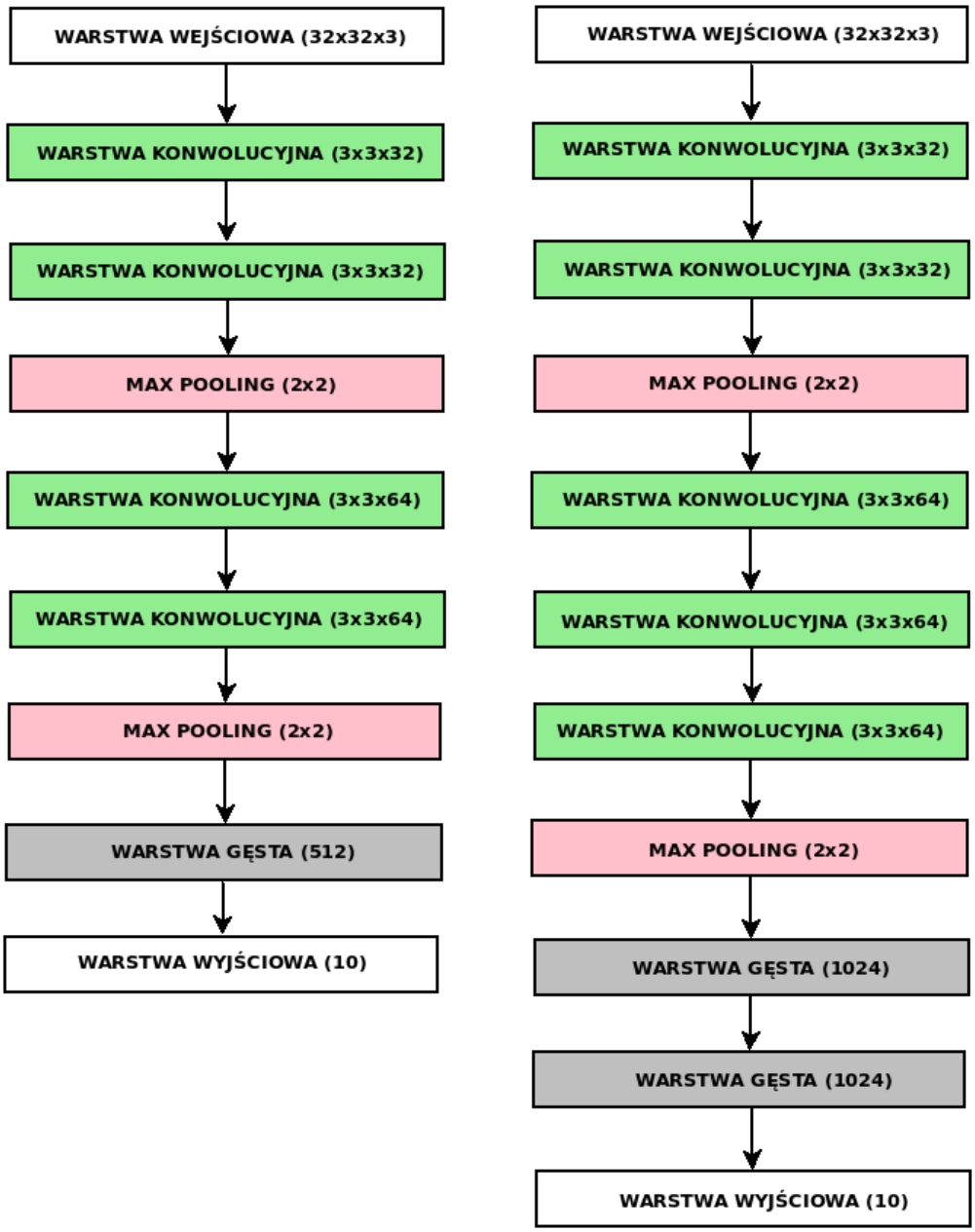
	bez regularyzacji	wymrażanie	dropout
średnia	30,50	31,88	30,75
odchylenie standardowe	2,12	1,25	1,39

Wyniki dla zmierzonej szybkość algorytmów optymalizacji są zgodne z oczekiwaniami. Najszybszym algorytmem jest NAdam, drugim AdaGrad, a najwolniejszym SGD. Wpłynęło to też na końcową skuteczność, ponieważ algorytm SGD nie zdążył osiągnąć maksymalnego wyniku w ciągu 50 epok.

Metody regularyzacji lekko spowalniają proces uczenia, ale jest to efekt bardzo mały, szczególnie jeśli wziąć pod uwagę różnice między średnimi w stosunku do odchylenia standardowego.

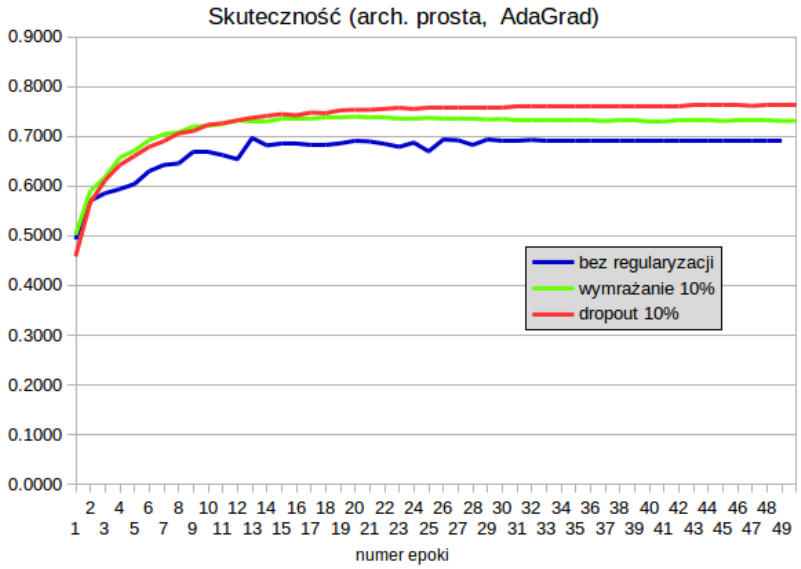
6.2 CIFAR-10

Baza CIFAR-10 [9] zawiera 60 tys. obrazów RGB o rozmiarach 32×32, które przedstawiają m.in. koty, psy czy samochody. Zadaniem jest zaklasyfikowanie obrazu to jednej z 10 kategorii.



Rysunek 6.8: Architektura sieci neuronowych wykorzystanych w eksperymentach na bazie CIFAR-10. Od lewej odpowiednio prosta i złożona.

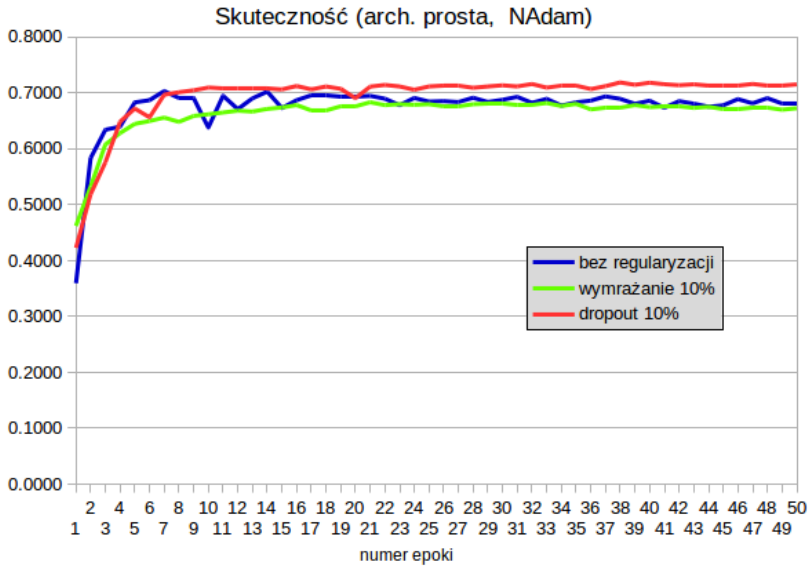
6.2.1 Architektura prosta



Rysunek 6.9: Porównanie skuteczności klasyfikacji dla architektury prostej i algorytmu AdaGrad. Dla sieci bez regularyzacji, z wymrażaniem 10% oraz z dropout'em 10%.

Tablica 6.10: Porównanie ostatecznej skuteczności dla architektury prostej i algorytmu AdaGrad.

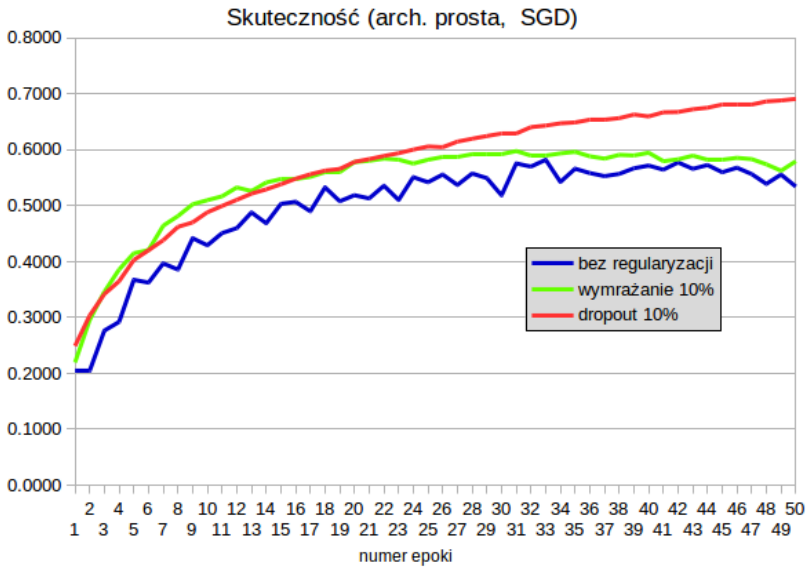
	bez regularyzacji	wymrażanie 10%	dropout 10%
średnia	0,6919	0,7317	0,7629
odchylenie standardowe	0,0073	0,0140	0,0060



Rysunek 6.10: Porównanie skuteczności klasyfikacji dla architektury prostej i algorytmu NAdam. Dla sieci bez regularyzacji, z wymrażaniem 10% oraz z dropout'em 10%.

Tablica 6.11: Porównanie ostatecznej skuteczności dla architektury prostej i algorytmu NAdam.

	bez regularyzacji	wymrażanie 10%	dropout 10%
średnia	0,6807	0,6728	0,7154
odchylenie standardowe	0,0559	0,0465	0,0207



Rysunek 6.11: Porównanie skuteczności klasyfikacji dla architektury prostej i algorytmu SGD. Dla sieci bez regularyzacji, z wymrażaniem 10% oraz z dropout'em 10%.

Tablica 6.12: Porównanie ostatecznej skuteczności architektury prostej i algorytmu SGD.

	bez regularyzacji	wymrażanie 10%	dropout 10%
średnia	0,5343	0,5795	0,6913
odchylenie standardowe	0,0277	0,0426	0,0050

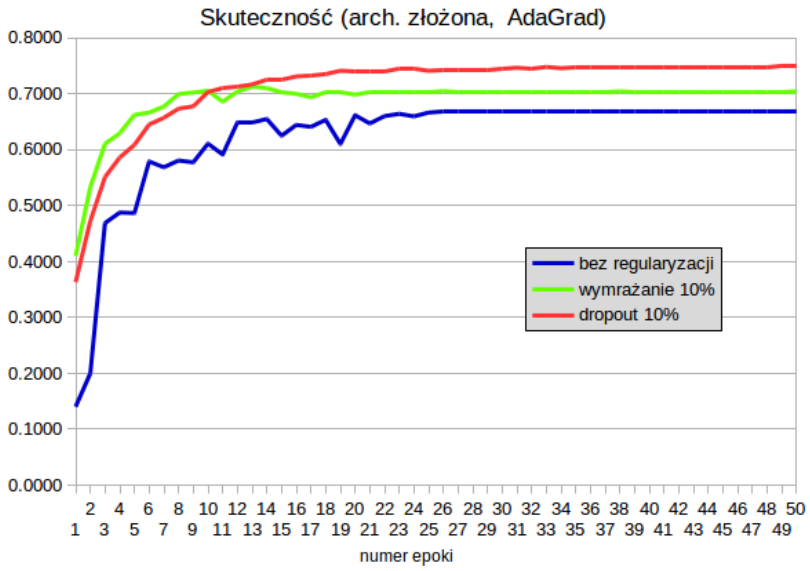
Powyższe badania wykonano dla bazy CIFAR-10 i sieci o architekturze prostej przedstawionej na rysunku 6.8. Można z nich wyciągnąć następujące wnioski i spostrzeżenia:

1. Z eksperymentów wynika, że najlepszą skuteczność mają sieci z zastosowanym dropout'em, ale ze względu na wysokie odchylenie standardowe nie można uznać takiej hipotezy na potwierdzoną. W większości przypadków średnie leżą od siebie w odległości mniejszej niż 2σ, zatem nie jest to wynik statystycznie istotny.

2. Problem klasyfikacji danych z bazy CIFAR-10 wydaję się być dużo bardziej złożony niż z bazy MNIST. Przemawiają za tym niższe końcowe skuteczności oraz wysoka wariancja.

3. Wyniki uzyskane w niniejszej pracy dla bazy CIFAR-10 są dużo gorsze niż te uzyskane w ostatnich latach. Najlepsze konfiguracja sieci uzyskała tj. sieć neuronowa o architekturze prostej, z algorytmem Adagrad i dropout'em 20% osiągnęła błąd na poziomie 22,82%, podczas gdy w literaturze jest to 15,60% [16].

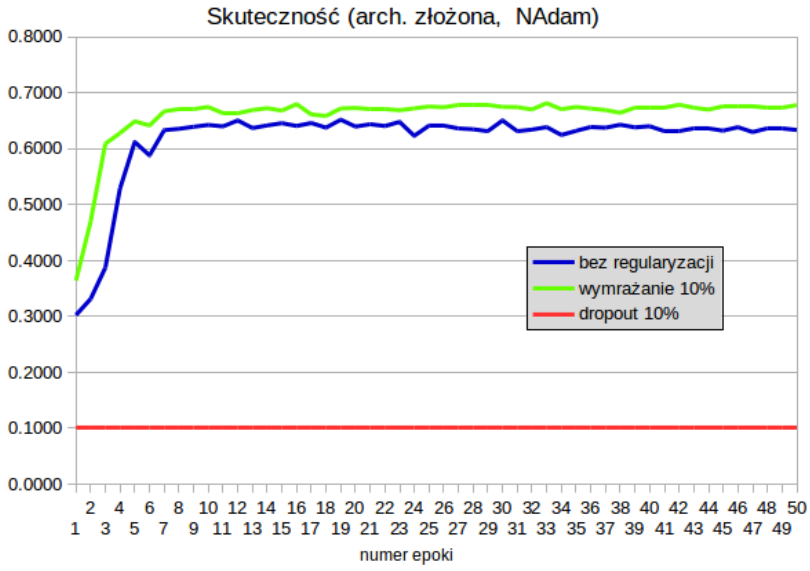
6.2.2 Architektura złożona



Rysunek 6.12: Porównanie skuteczności klasyfikacji dla architektury złożonej i algorytmu AdaGrad. Dla sieci bez regularyzacji, z wymrażaniem 10% oraz z dropout'em 10%.

Tablica 6.13: Porównanie ostatecznej skuteczności dla architektury złożonej i algorytmu AdaGrad.

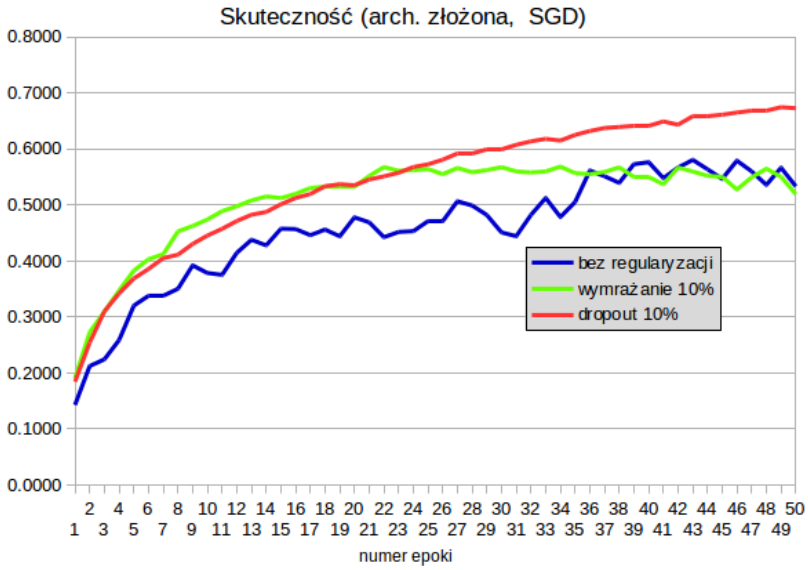
	bez regularyzacji	wymrażanie 10%	dropout 10%
średnia	0,6681	0,7048	0,7491
odchylenie standardowe	0,0087	0,0060	0,0095



Rysunek 6.13: Porównanie skuteczności klasyfikacji dla architektury złożonej i algorytmu NAdam. Dla sieci bez regularyzacji, z wymrażaniem 10% oraz z dropout'em 10%.

Tablica 6.14: Porównanie ostatecznej skuteczności dla architektury złożonej i algorytmu NAdam.

	bez regularyzacji	wymrażanie 10%	dropout 10%
średnia	0,6337	0,6784	0,1000
odchylenie standardowe	0,0419	0,0349	0,0000



Rysunek 6.14: Porównanie skuteczności klasyfikacji dla architektury złożonej i algorytmu SGD. Dla sieci bez regularyzacji, z wymrażaniem 10% oraz z dropout'em 10%.

Tablica 6.15: Porównanie ostatecznej skuteczności dla architektury złożonej i algorytmu SGD.

	bez regularyzacji	wymrażanie 10%	dropout 10%
średnia	0,5337	0,5196	0,6732
odchylenie standardowe	0,0285	0,0768	0,0113

Powyższe badania wykonano dla bazy CIFAR-10 i sieci o architekturze złożonej przedstawionej na rysunku 6.8. Można z nich wyciągnąć następujące wnioski i spostrzeżenia:

1. Średnie skuteczności wskazują na to, że sieci bez regularyzacji są najgorsze, a sieci z droout'em najlepsze, ale ze względu na wysokie odchylenie standardowe nie można uznać takiej hipotezy na potwierdzoną.

2. Sieci z wymrażaniem w początkowym stadium uczenia mają lepszą skuteczność niż sieci z dropout'em. Wydaje się, że te sieci mają niższy maksymalny potencjał, ale szybciej go osiągają.

3. Na rysunku 6.13 można zauważyć, że sieć z zastosowanym dropout'em w ogóle się nie uczyła, a jej skuteczność pozostała na poziomie 10%. Jest to prawdopodobnie spowodowane błędem w bibliotece Keras. Nie jest to odosobniony przypadek. Dla wielu przetestowanych sieci na bazie CIFAR-10 z algorytmem NAdam pojawią się ten problem.

6.2.3 Porównanie szybkości uczenia

W poniższych tabelach uśredniono ilość epok potrzebną do osiągnięcia 99% końcowej skuteczności dla 5-krotnej walidacji, architektury prostej i złożonej oraz dla różnych ustawień wymrażania i dropout'u tj. 10%, 20%, 30% i 50%.

Tablica 6.16: Porównanie średniej ilości epok, które potrzebne były aby osiągnąć 99% końcowej skuteczności dla algorytmu AdaGrad.

	bez regularyzacji	wymrażanie	dropout
średnia	18,00	14,25	36,63
odchylenie standardowe	2,83	6,76	10,42

Tablica 6.17: Porównanie średniej ilości epok, które potrzebne były aby osiągnąć 99% końcowej skuteczności dla algorytmu NAdam.

	bez regularyzacji	wymrażanie	dropout
średnia	6,50	9,50	25,50
odchylenie standardowe	0,71	1,73	21,92

Tablica 6.18: Porównanie średniej ilości epok, które potrzebne były aby osiągnąć 99% końcowej skuteczności dla algorytmu SGD.

	bez regularyzacji	wymrażanie	dropout
średnia	27,00	27,50	46,67
odchylenie standardowe	12,73	12,86	2,34

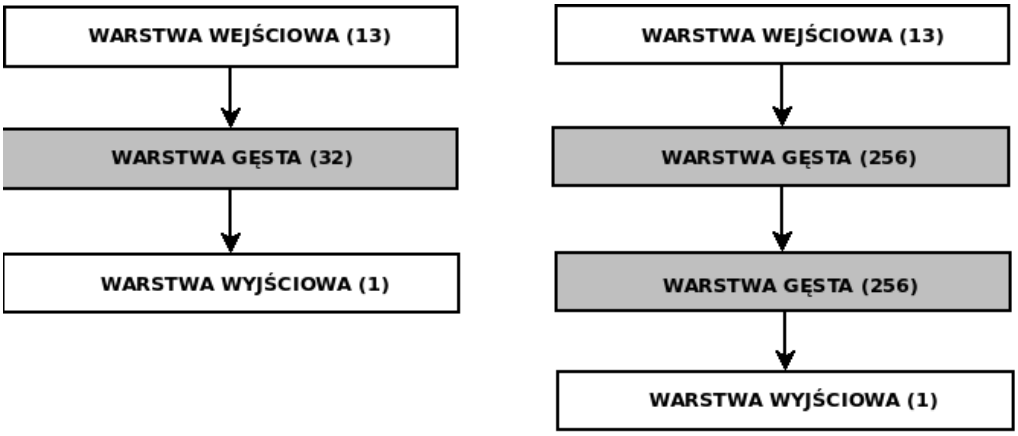
Duża wariancja w szybkości uczenia dla algorytmu NAdam z zastosowanym dropout'em spowodowana jest przez problem opisany wcześniej polegający na tym, że dla tego algorytmu sieć często się nie uczy, a jej skuteczność pozostaje na poziomie 10%. Te wyniki zostały odrzucone.

Zgodnie z oczekiwaniami najszybszym algorytmem był NAdam, następne AdaGrad, a najwolniejszym SGD.

Metoda wymrażania wydaję się nie mieć wpływu na szybkość uczenia sieci, podczas gdy dropout mocno zmniejsza szybkość uczenia. Jest to istotna wada, ale jest ceną za większą skuteczność uzyskiwaną dla tej metody regularyzacji.

6.3 Boston Housing Prices

Baza Bostoun Housing Prices [7] zawiera dane na temat ceny 506 domów oraz 14 zmierzonych atrybutów. Celem jest predykcja ceny domu na podstawie tychże atrybutów.



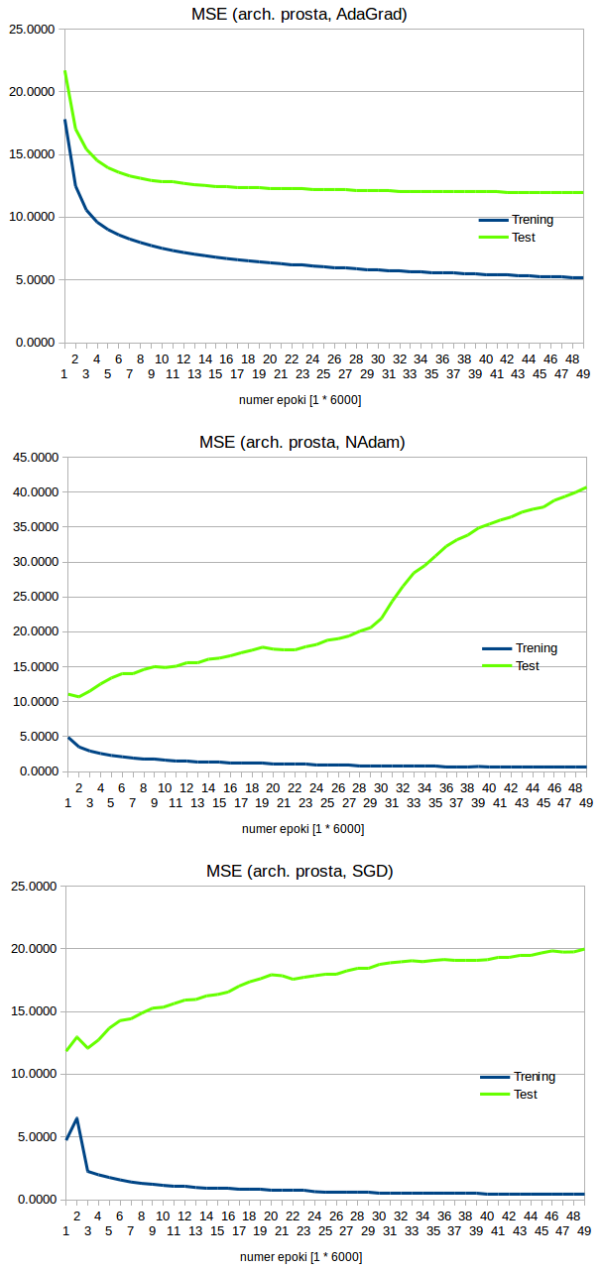
Rysunek 6.15: Architektura sieci neuronowych wykorzystanych w eksperymentach na bazie Boston Housing Prices. Od lewej odpowiednio prosta i złożona.

Rozmieszczenie wykresów w tym podrozdziale jest inne niż w dwóch poprzednich, ze względu na to, że wystąpił efekt przeuczenia. Najpierw przedstawione zostały wyniki uczenia sieci bez regularyzacji, a następnie, jak te wyniki się zmieniają pod wpływem wymrażania i dropout'u.

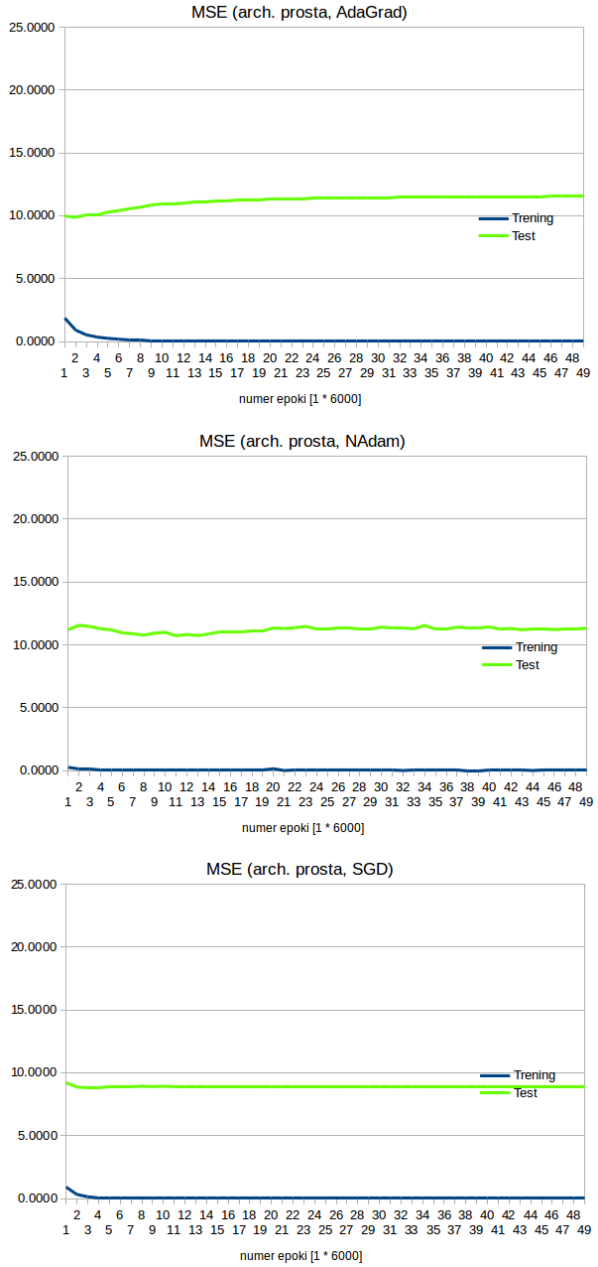
6.3.1 Wyniki bez regularyzacji

Poniżej przedstawiono wykresy błędów treningowego oraz testowego w funkcji ilości epok. Istotnym jest aby pamiętać o tym, że oś X jest w skali 1:6000, a całe uczenie trwało 300 000 epok. Dodatkowo wynik po pierwszych 6000 epok nie został zawarty na wykresach, ponieważ wymagałoby to zmiany skali, co spowodowałoby zmniejszenie czytelności wyników dla kolejnych epok.

Zachowanie sieci przez pierwsze 6000 epok było normalnym uczeniem sieci, które zostało już przedstawione dla baz MNIST oraz CIFAR-10.



Rysunek 6.16: Porównanie MSE dla sieci bez regularyzacji o architekturze prostej. Kolejno: AdaGrad, NAdam, SGD. Niebieski - faza uczenia, zielony - faza testowa.

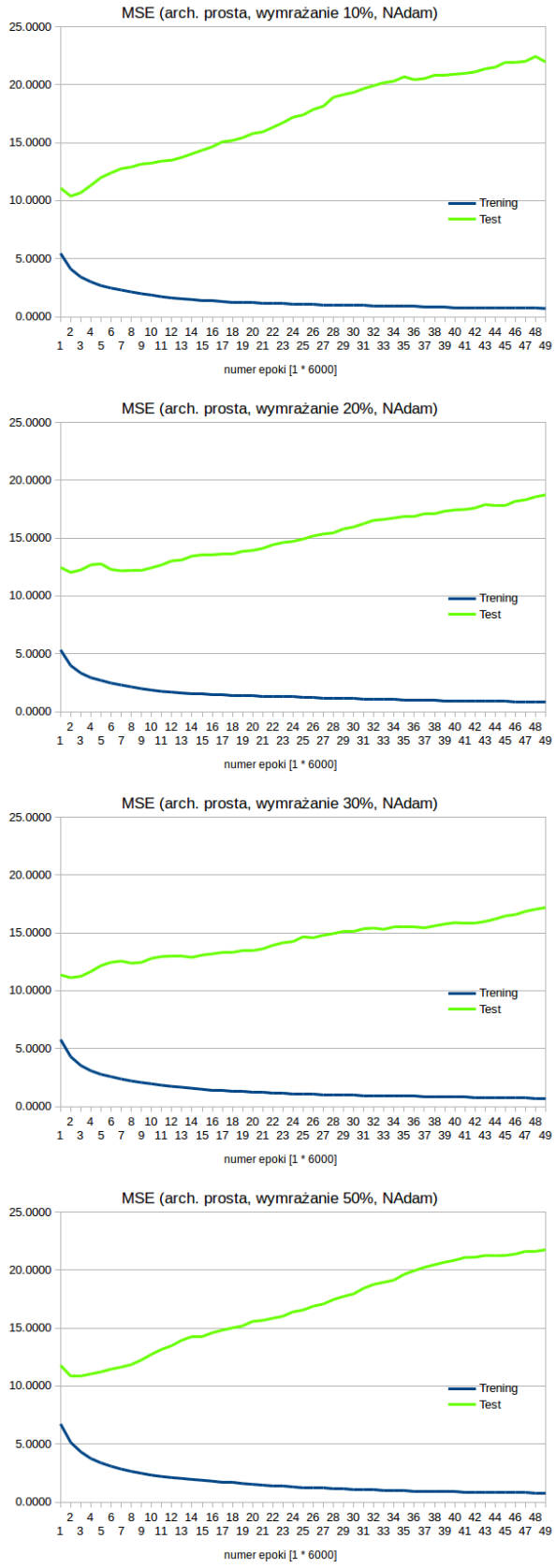


Rysunek 6.17: Porównanie MSE dla sieci bez regularyzacji o architekturze złożonej. Kolejno: AdaGrad, NAdam, SGD. Niebieski - faza uczenia, zielony - faza testowa.

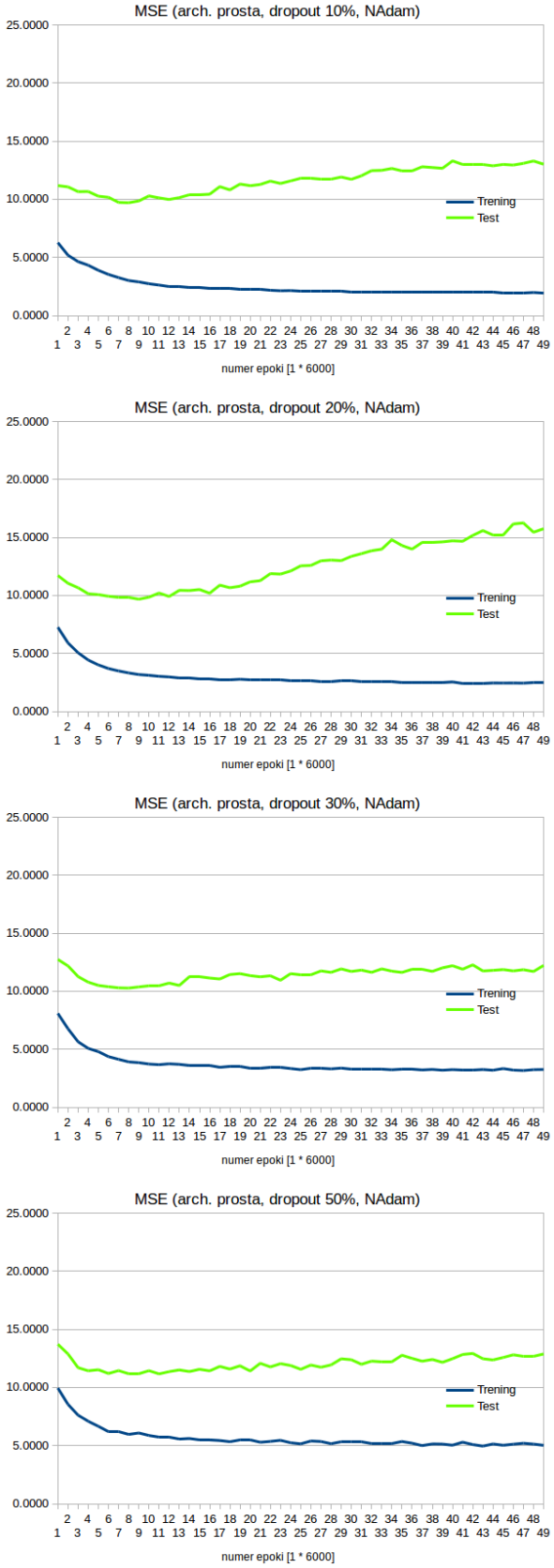
Z rysunków 6.16 oraz 6.17 można zauważyć, że znaczące przeuczenie pojawia się tylko dla sieci o architekturze prostej z algorytmem NAdam oraz dla sieci o architekturze prostej z algorytmem SGD. W kolejnych rozdziałach przedstawiono, jak zmieniają się błąd treningowy i testowy dla powyższych sieci, jeśli zastosować do nich wymrażanie i dropout z różnymi nastawami.

Sieci z architekturami złożonymi są mniej podatne na przeuczenie. Możliwe, że ze względu na to, że ze wzrostem liczby wymiarów minima lokalne zaczynają mieć podobne wartości.

6.3.2 Wyniki dla regularyzacji sieci z architekturą prostą i algorytmem NAdam



Rysunek 6.18: Porównanie MSE dla sieci o architekturze prostej z wymrażaniem kolejno: 10%, 20%, 30%, 50%. Niebieski - faza uczenia, zielony - faza testowa.



Rysunek 6.19: Porównanie MSE dla sieci o architekturze prostej z dropout'em kolejno: 10%, 20%, 30%, 50%. Niebieski - faza uczenia, zielony - faza testowa.

Tablica 6.19: Porównanie ostatecznego MSE na zbiorze testowym dla sieci z architekturą prostą i algorytmem NAdam.

	średnia	odchylenie standardowe
bez regularyzacji	40,7149	40,6471
wymrażanie 10%	21,9662	5,6951
wymrażanie 20%	18,7388	4,6598
wymrażanie 30%	17,1903	1,8997
wymrażanie 50%	21,7619	6,6209
dropout 10%	13,0226	3,4211
dropout 20%	15,7574	8,9997
dropout 30%	12,2257	6,3014
dropout 50%	12,8875	4,3403

Tablica 6.20: Porównanie najlepszego MSE na zbiorze testowym dla sieci z architekturą prostą i algorytmem NAdam.

	średnia	odchylenie standardowe
bez regularyzacji	10,6863	3,2701
wymrażanie 10%	10,3884	2,5968
wymrażanie 20%	12,0315	4,9027
wymrażanie 30%	11,1264	3,7372
wymrażanie 50%	10,8510	3,7922
dropout 10%	9,7023	3,6763
dropout 20%	9,6781	2,9925
dropout 30%	10,2690	4,2296
dropout 50%	11,1670	4,1830

Powyższe badania wykonano dla bazy Bostoun Housing Prices i sieci o architekturze prostej przedstawionej na rysunku 6.15 z algorytmem NAdam. Można z nich wyciągnąć następujące wnioski i spostrzeżenia:

1. Z powyższych wykresów można wywnioskować, że metoda wymrażania i dropout zapobiegają przeuczeniu, ale zwiększenie liczby neuronów, które są poddawane działaniu tych algorytmów, nie zawsze skutkuje zmniejszaniem overfittingu.

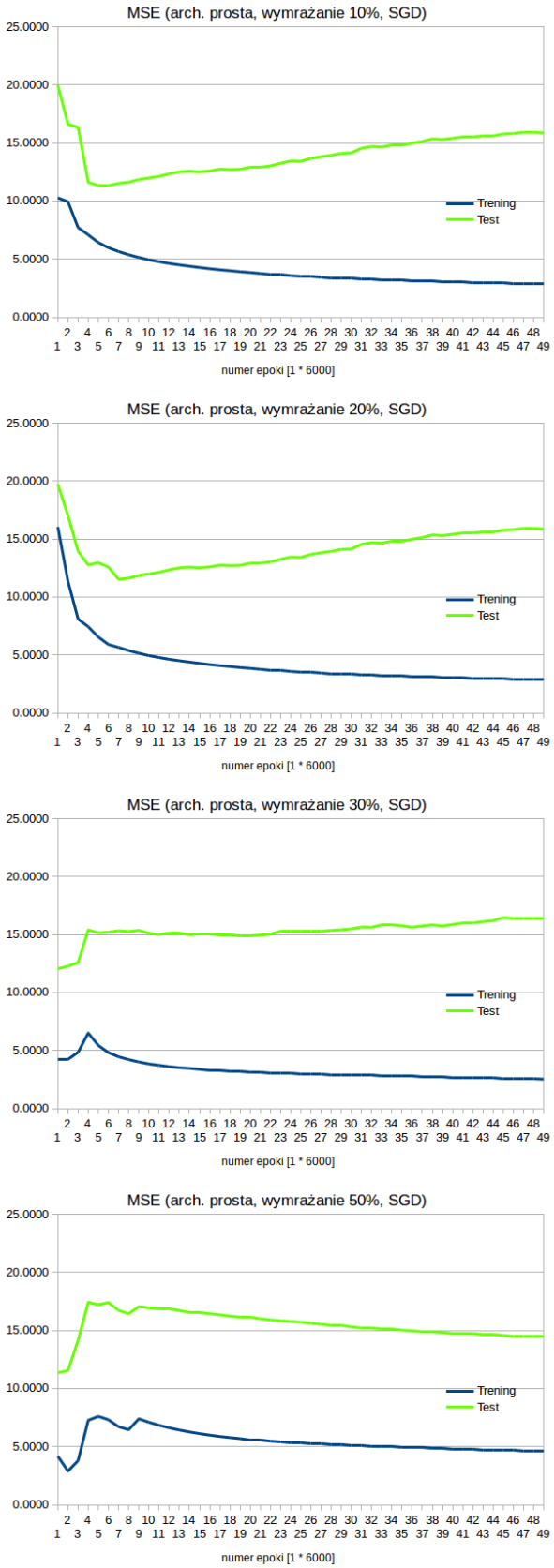
2. Dropout jest skuteczniejszy w walce z przeuczeniem niż wymrażanie i to dla niego z nastawą 20% został osiągnięty najlepszy wynik dla badanej sieci.

3. Powyższy wniosek bierze pod uwagę wyłącznie średnie. Ze względu na bardzo duże odchylenie standardowe nie jest on statystycznie znaczący, a większość średnich znajduje się od siebie w odległości mniejszej niż 2σ.

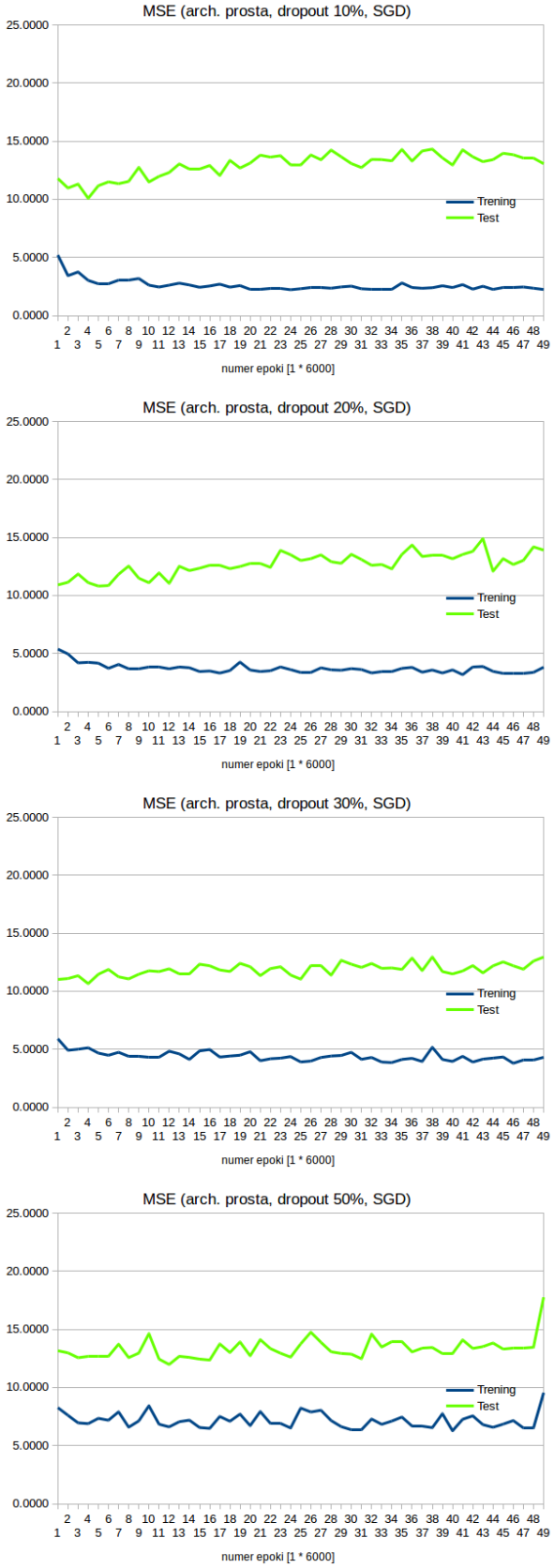
4. Wyniki dla sieci z architekturą prostą i algorytmem NAdam mają bardzo wysoką średnią i odchylenie standardowe. Jest to spowodowane przez to, że w jednym z eksperymentów została osiągnięta wartość aż 112,7200. Gdy się ją usunie to średnia spada do 22,7137, a odchylenie standardowe do 6,5275.

5. Ze względu na małe rozmiary bazy, wybrany rozmiar paczki danych (tj. 128) mógł okazać się za duży i mogło to zmniejszyć skuteczność metod regularyzacji. Wymaga do dalszych badań.

6.3.3 Wyniki dla regularyzacji sieci z architekturą prostą i algorytmem SGD



Rysunek 6.20: Porównanie MSE dla sieci o architekturze prostej z wymrażaniem kolejno: 10%, 20%, 30%, 50%. Niebieski - faza uczenia, zielony - faza testowa.



Rysunek 6.21: Porównanie MSE dla sieci o architekturze prostej z dropout'em kolejno: 10%, 20%, 30%, 50%. Niebieski - faza uczenia, zielony - faza testowa.

Tablica 6.21: Porównanie ostatecznego MSE na zbiorze testowym z architekturą prostą i algorytmem SGD.

	średnia	odchylenie standardowe
bez regularyzacji	19,9891	7,8497
wymrażanie 10%	15,8723	8,4479
wymrażanie 20%	15,8363	8,1125
wymrażanie 30%	16,3614	1,6942
wymrażanie 50%	14,4836	3,7063
dropout 10%	13,0608	2,7093
dropout 20%	13,9145	5,1764
dropout 30%	12,9481	4,3080
dropout 50%	17,7673	7,6693

Tablica 6.22: Porównanie najlepszego MSE na zbiorze testowym dla sieci z architekturą prostą i algorytmem SGD.

	średnia	odchylenie standardowe
bez regularyzacji	11,8396	2,6110
wymrażanie 10%	11,3645	3,2375
wymrażanie 20%	11,5272	3,6755
wymrażanie 30%	12,0482	2,6996
wymrażanie 50%	11,3620	2,8730
dropout 10%	10,0825	1,3848
dropout 20%	10,8066	4,1407
dropout 30%	10,6539	4,0101
dropout 50%	11,9833	4,0244

Powyższe badania wykonano dla bazy Bostoun Housing Prices i sieci o architekturze prostej przedstawionej na rysunku 6.15 z algorytmem SGD. Można z nich wyciągnąć następujące wnioski i spostrzeżenia:

1. Najlepszy wynik dla badanej sieci został osiągnięty algorytmu dropout z dezaktywacją 20% neuronów.

2. Dropout bardzo osłabił wyeliminował efekt przeuczenia, ale na wykresach widoczny jest znaczący szum.

3. Dropout jest skuteczniejszy w walce z przeuczeniem niż wymrażanie, ale ze względu na duże odchylenie standardowe ten wniosek nie jest statystycznie znaczący. Większość średnich znajduje się od siebie w odległości mniejszej niż 2σ.

4. Wyniki uzyskane dla bazy Boston Housing Prices są nieznacznie lepsze od wyników publikowanych w artykułach naukowych. W niniejszej pracy sieć neuronowe o architekturze złożonej z algorytmem SGD oraz dropout'em 30% osiągnęła RMSE na poziomie 2,75, podczas gdy w literaturze jest to 2,97 [4].

5. Dla bazy Boston Housing Prices nie porównano szybkości uczenia się sieci. Wyniki nie byłyby wiarygodne ze względu na małą rozdzielczość tj. wynik co 6000 epok.

Rozdział 7

Podsumowanie

Celem niniejszej pracy było porównanie wpływu algorytmów regularyzacji na skuteczność uczenia sieci neuronowych oraz przeciwdziałanie przeuczeniu.

Badanie wykonano z użyciem metody wymrażania [14] i dropout'u [16] oraz porównano je z sieciami neuronowymi nauczonymi bez stosowania metod przeciwdziałania overfittingowi. Zostały one też wykonane dla różnych algorytmów optymalizacji, dla różnych architektur oraz dla różnych baz danych, aby uzyskane wyniki były jak najbardziej obiektywne.

Stosowanie metody wymrażania, dało lepsze wyniki niż sieci neuronowe nauczone bez regularyzacji, a jeszcze lepsze wyniki uzyskano dla algorytmu dropout. Wyniki te jednak mają duża wariancję, co sprawia, że takie porównanie nie jest wiarygodne. Poszczególne wnioski z badań można przeczytać w rozdziale 6.

W badaniach porównane zostały metoda wymrażania i dropout dla takich samych nastaw np. 20%. Może nie być to najlepsze wyjście, ponieważ są to różne algorytmy.

W celu rozwinięcia i pogłębienia tematyki tej pracy można zastosować poniższe pomysły:

1. zbadanie jak wielkość paczki danych wpływa na działanie metody wymrażanie,
2. porównanie tej metody z regularyzają L1 i L2,
3. zbadanie metod hybrydowych np. wymrażanie + dropout lub wymrażanie + L2,
4. zmienienie implementacji tak, aby zawsze wybierany zostawał dokładnie dany procent neuronów,
5. zbadanie w jaki sposób dobór parametru momentum wpłynie na jakość uczenia.

Bibliografia

[1] Banach, Michał. *Rozpoznawanie tekstu przy pomocy konwolucyjnych sieci neuronowych i korekty słownikowej*, (2017).

[2] Dozat, Timothy. *Incorporating Nesterov momentum into adam.* (2016).

[3] Duchi, John, Elad Hazan, Yoram Singer. *Adaptive subgradient methods for online learning and stochastic optimization.* Journal of Machine Learning Research 12.07. (2011): 2121-2159.

[4] Gal, Yarin, Zoubin Ghahramani. *Dropout as a bayesian approximation: Representing model uncertainty in deep learning.* International Conference on Machine Learning. (2016).

[5] Gonzalez, Rafael C., and Richard E. Woods. *Digital image processing, 2/e* Pearson Education, (2012).

[6] Goodfellow, Ian, Yoshua Bengio, Aaron Courville. *Deep learning.* MIT press, (2016).

[7] Harrison Jr, David, Daniel L. Rubinfeld. *Hedonic housing prices and the demand for clean air.* Journal of environmental economics and management 5.1 (1978): 81-102.

[8] Kingma, Diederik P., Jimmy Ba. *Adam: A method for stochastic optimization.* arXiv:1412.6980 (2014).

[9] Krizhevsky, Alex, Geoffrey Hinton. *Learning multiple layers of features from tiny images.* Tom 1. Nr 4. Technical report, Uniwersytet Toronto, (2009).

[10] LeCun, Yann, et al. *Gradient-based learning applied to document recognition.* Proc. IEEE 86.11 (1998): 2278-2324.

[11] Nesterov, Yu. *A method of solving a convex programming problem with convergence rate* $O(1/k^2)$ *O (1/k2).* Sov. Math. Dokl. Tom 27. Nr 2.

[12] Prechelt, Lutz. *Early stopping-but when?* Neural Networks: Tricks of the trade. Springer, Berlin, Heidelberg, 1998. 55-69.

[13] Rad, Nastaran Mohammadian, et al. *Convolutional neural network for stereotypical motor movement detection in autism.* arXiv:1511.01865 (2015).

[14] Rutkowski, Leszek et al. (Eds.): ICAISC 2016, Part I, LNAI 9692, pp. 148–157, (2016).

[15] Simard, Patrice Y., David Steinkraus, John C. Platt. *Best practices for convolutional neural networks applied to visual document analysis.* Icdar. Tom 3. Nr 2003.

[16] Srivastava, Nitish, et al. *Dropout: a simple way to prevent neural networks from overfitting.* The Journal of Machine Learning Research 15.1 (2014): 1929-1958.

[17] Sutskever, Ilya, et al. *On the importance of initialization and momentum in deep learning.* International conference on machine learning. (2013).

[18] https://en.wikipedia.org/wiki/File:Overfitted_Data.png [dostęp 10.05.2019r.]

[19] https://en.wikipedia.org/wiki/File:Overfitting_svg.svg [dostęp 10.05.2019r.]

Spis rysunków