AGH
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Piotr Nowak

kierunek studiów: **fizyka techniczna**

Trenowanie sieci neuronowej do klasyfikacji śladów w eksprymencie LHCb przy wykorzystaniu kart graficznych

Opiekun: **dr hab. inż. Tomasz Szumlak**

Kraków, styczeń 2018

Oświadczam, świadomy(-a) odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

.....
(czytelny podpis)

Na kolejnych dwóch stronach proszę dołączyć kolejno recenzje pracy popełnione przez Opiekuna oraz Recenzenta (wydrukowane z systemu MISIO i podpisane przez odpowiednio Opiekuna i Recenzenta pracy). Papierową wersję pracy (zawierającą podpisane recenzje) proszę złożyć w dziekanacie celem rejestracji.

Streszczenie

W fizyce wysokich energii ogromne zastosowanie mają metody uczenia maszynowego, w tym sztuczne sieci neuronowe. Obliczenia równoległe przeprowadzane na procesorach kart graficznych posiadają doskonałe wydajności i wydają się być świetnym rozwiązaniem problemu ogromnej ilości danych generowanych podczas eksperymentów. W poniższej pracy znajduje się przykład zastosowania sieci neuronowych w klasyfikatorze na eksperymencie LHCb. Stworzono w tym celu program w języku C++ oraz jego modyfikację wykorzystującą obliczenia równoległe w architekturze CUDA, a następnie dokonano analizy działania obu programów.

Spis treści

1 Wstęp

W ostatnich latach dużą popularność zdobywa wykorzystanie kart graficznych do obliczeń naukowych. Na rynku pojawiły się dedykowane urządzenia do wykonywania obliczeń na procesorach kart graficznych (GPGPU). Karty graficzne pozwalają na zwiększenie szybkości działania algorytmów poprzez zastosowanie równoległych obliczeń. Im więcej pojedyńczych niezależnych obliczeń tym lepsze wyniki daje zastosowanie programowania równoległego. Jednym z najpopularniejszych dedykowanych rozwiązań jest technologia CUDA wraz z językiem C for CUDA stworzona przez firmę Nvidia. [1]

Sztuczna sieć neuronowa to jedna z najpopularniejszych metod uczenia maszynowego. Wraz z rozwojem techniki obliczeń równoległych rozwiązania te są stosowane coraz częściej w działaniu sieci neuronowych. Budowanych jest wiele bibliotek programistycznych wspierających wykorzystanie kart graficznych do przyspieszenia działania obliczeń w sieciach neuronowych.

W eksperymentach w ośrodku CERN pod Genewą generuje się mnóstwo danych fizycznych wymagających wielkich nakładów pracy do analizy. Zastosowanie metod uczenia maszynowego nie jest nowością w fizyce wysokich energii, używa się ich od dawna m.in. przy poszukiwaniu rzadkich rozpadów. Jednych z głównych eksperymentów w ośrodku pod Genewą jest eksperyment LHCb, który z powodu bardzo dużej liczby danych wymaga wydajnego systemu selekcji przypadków do archiwizacji.

W niniejszej pracy zbudowano sztuczną sieć neuronową w języku C++, a następnie zaimplementowano wersję programu w architekturze CUDA. Celem było stworzenie wydajnej i poprawnie działającej sieci neuronowej bez wykorzystania gotowych bibliotek do klasyfikatora przypadków w eksperymencie LHCb. Na danych z symulacji Monte Carlo dokonano analizy działania sieci neuronowej w zależności od zastosowanych rozwiązań.

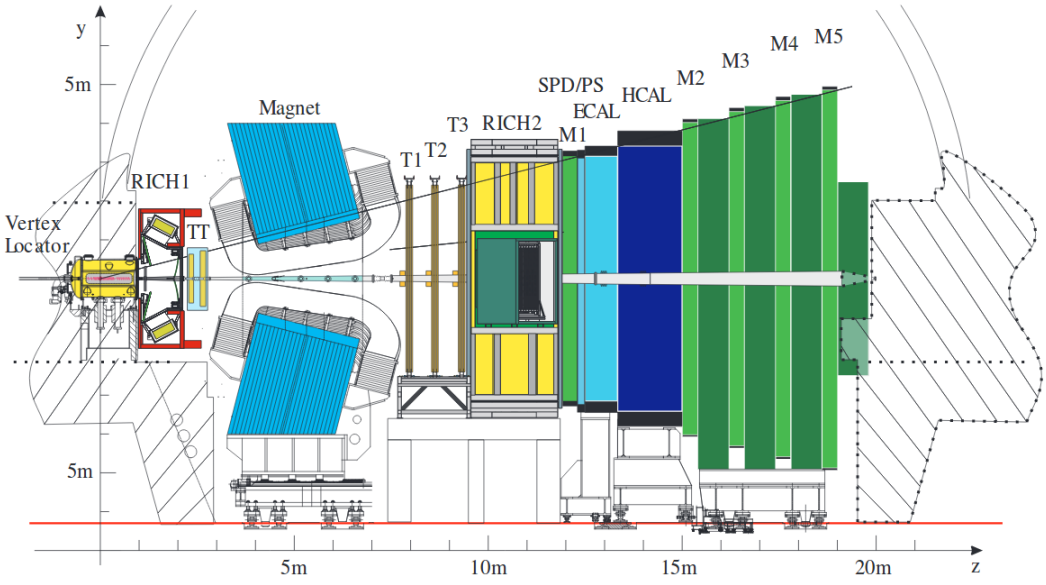
2 Wstęp teoretyczny

2.1 LHC

Large Hadron Collider (ang. Wielki Zderzacz Hadronów) to największy i najpotężniejszy akcelerator cząstek na świecie. Znajduje się w Europejskim Ośrodku Badań Jądrowych CERN w pobliżu Genewy. Akcelerator znajduje się ponad 100m pod ziemią w 27km tunelu. Przy użyciu akceleratora zderzane są wiązki protonów. Podczas zderzeń powstają nowe, często nietrwałe cząstki, których pomiarem zajmują się detektory usytuowane w miejscu przecięcia wiązek.

2.1.1 Eksperyment LHCb

LHCb (ang. Large Hadron Collider beauty) to jeden z czterech głównych eksperymentów LHC. Jego celem jest badanie łamania symetrii ładunkowo-przestrzennej CP oraz wytłumaczenie dominacji materii nad antymaterią we Wrzechświecie. W skład eksperymentu wchodzi kilka detektorów m.in VELO, detektory Czerenkowa, kalorymetry. Na rysunku 2.1 przedstawiono przekrój poprzeczny detektora LHCb.



Rysunek 2.1: Przekrój poprzeczny detektora LHCb [2]

2.1.2 VELO

Detektor VELO (ang. Vertex Locator) jest to mikropaskowy detektor krzemowy, mierzący położenie cząstek naładowanych w pobliżu punktu oddziaływań wiązek protonowych. Jego głównym celem jest rekonstrukcja wierzchołków pierwotnych i wierzchołków wtórnych oraz pomiar geometrycznego parametru zderzenia, która to wielkość jest kluczowa dla systemu wyzwalania (ang. trigger) detektora LHCb.

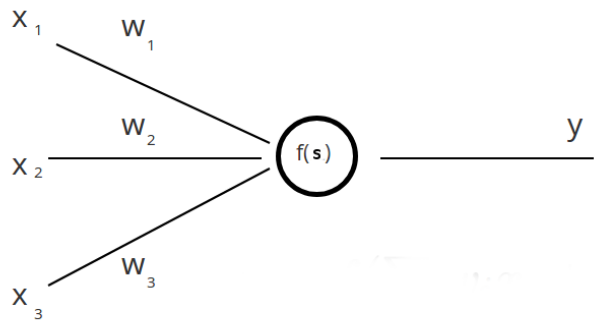
2.1.3 Selekcja danych

Podczas pomiarów wiązki przecinają się z częstotliwością ok. 40 MHz generując mnóstwo danych, niemożliwych do archiwizacji w rzeczywistym czasie. Aby rozwiązać ten problem stosuje się redukcję strumienia danych poprzez wstępny wybór interesujących nas przypadków. Pierwszy poziom (L0) to zaimplementowany w elektronice system, a drugi (HLT - ang. "High Level Trigger") to program komputerowy zaimplementowany w języku C++, na podstawie którego wybierane są przypadki do archiwizacji.
W związku z planowaną roku modernizacją detektora [7] zaistniała potrzeba stworzenia wydajniejszego systemu klasyfikacji śladów. Jednym z projektów [5] jest stworzenie klasyfikatora do wstępnej selekcji przypadków przy użyciu modeli uczenia maszynowego. Proponowaną metodą są sztuczne sieci neuronowe. W związku dużą liczbą danych potrzebny jest wydajny system trenowania sieci. Jednym z rozwiązań tego problemu jest zastosowanie obliczeń równoległych na kartach graficznych.

2.2 Sztuczne sieci neuronowe

Sztuczne sieci neuronowe (SSN) były pierwotnie próbą symulacji działania ludzkiego mózgu. Jednak wraz z rozwojem technologii i powstaniem algorytmu wstecznej propagacji [2] sieci neuronowe stały się osobnym narzędziem niemającym wiele wspólnego z początkowymi założeniami zapożyczając jedynie koncepcję działania.

2.2.1 Model sztucznego neuronu



Rysunek 2.2: Schemat sztucznego neuronu

Podstawowym elementem sieci neuronowej w mózgu jest neuron, tak samo jest w przypadku sztucznych sieci neuronowych [16]. W neuronie sygnały wejściowe x_i są mnożone przez wartości wag w_i, a następnie sumowane i przekazywane do funkcji aktywacji f jako argument, której wynik jest wyprowadzany na wyjście neuronu y (rysunek 2.2). Dodatkowo często dodaje się do sumy iloczynu sygnałów wejściowych i wag pewną wartość nazywaną przesunięciem w_0 (ang.

bias). Wyjście neuronu można zapisać jako:

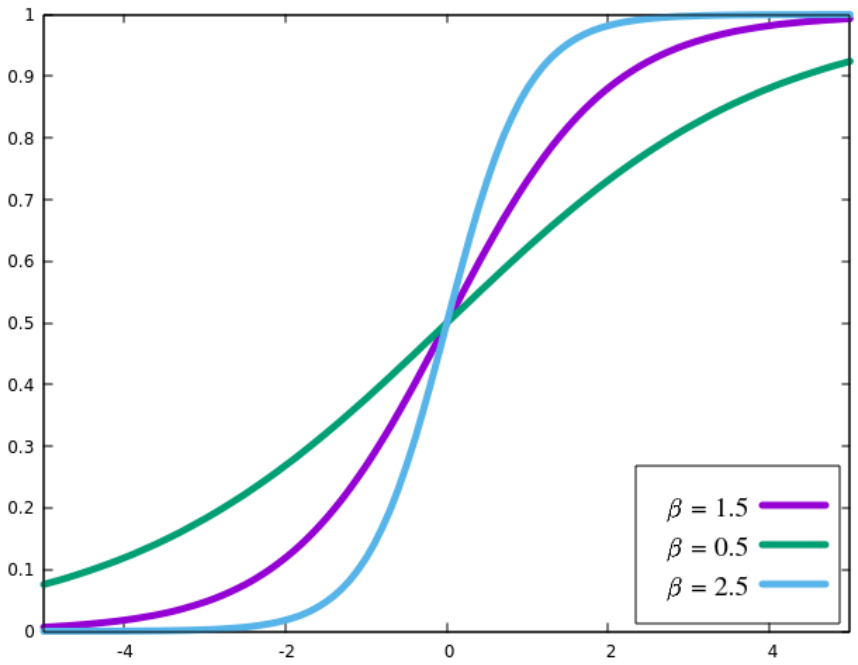
$$s = \sum_{i=1}^{n} w_i x_i + w_0 \tag{2.1}$$

$$y = f(\sum_{i=1}^{n} w_i x_i + w_0) \tag{2.2}$$

2.2.2 Funkcja aktywacji

W przyrodzie neuron może być albo aktywny albo nie, stąd funkcja aktywacji ma charakter skokowy. Jednak z powodu omówionego w rozdziale 2.2.4 przyjmujemy, że funkcja musi być ciągła i różniczkowalna. Najpopularniejszą funkcją aktywacji jest funkcja logistyczna (sigmoidalna) przedstawiona na rysunku 2.3 mająca postać:

$$f(s) = \frac{1}{1 + e^{-\beta s}} \tag{2.3}$$



Rysunek 2.3: Wykres zależności funkcji sigmoidalnej od parametru β

Używana w algorytmie wstecznej propagacji pochodna funkcji aktywacji dla funkcji logistycznej wynosi

$$f'(s) = \beta f(s)(1 - f(s)) \tag{2.4}$$

Inną użyteczną funkcją aktywacji jest funkcja ReLU (ang. rectified linear unit), która ma postać:

$$f(s) = \begin{cases} s & \text{gdy } s > 0 \\ 0 & \text{gdy } s <= 0 \end{cases} \tag{2.5}$$

Pochodna funkcji ReLU ma postać:

$$f'(s) = \begin{cases} 1 & \text{gdy } s > 0 \\ 0 & \text{gdy } s <= 0 \end{cases} \tag{2.6}$$

Obecnie coraz częściej odchodzi się od funkcji logistycznej na rzecz ReLU, głównie z powodu szybszego obliczenia wartości funkcji. [9] Jednak dużym problemem jest "śmierć" neuronu w przypadku, gdy jego wynikiem funkcji będzie zero. Rozwiązaniem tego problemu jest modyfikacja algorytmu o nazwie Leaky ReLU, która ma postać:

$$f(s) = \begin{cases} s & \text{gdy } s > 0 \\ \alpha s & \text{gdy } s <= 0 \end{cases} \tag{2.7}$$

Jej pochodna wygląda analogicznie do pochodnej ReLU

$$f'(s) = \begin{cases} 1 & \text{gdy } s > 0 \\ \alpha & \text{gdy } s <= 0 \end{cases} \tag{2.8}$$

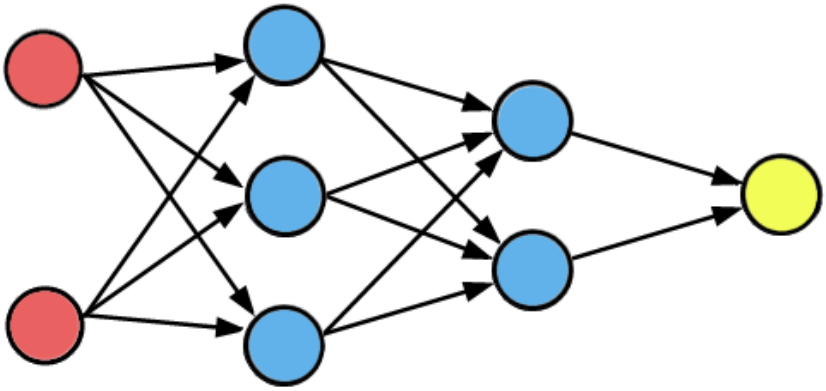
Parametrowi najczęściej nadaje się wartość $\alpha = 0.01$. Dzięki temu funkcja dla $s < 0$ posiada niewielki dodatni gradient co pozwala jej na ewentualne wydostanie się z nieodwracalnego dla ReLU stanu.
Jeszcze jedną bardzo użyteczną funkcją, zwłaszcza dla ostatniej warstwy jest funkcja Softmax, czyli inaczej znormalizowana funkcja wykładnicza.

$$f(s)_i = \frac{e^{s_i}}{\sum_{k=1}^{n} e^{s_k}} \tag{2.9}$$

Stosowana jest w problemach klasyfikacji, gdyż jej wynik można interpretować jako prawdopodobieństwo.

2.2.3 Struktura sieci neuronowej

Najpopularniejszym rodzajem sieci neuronowej jest sieć neuronowa jednokierunkowa wielowarstowowa. W warstwie wejściowej znajdują się dane przekazywane sieci. Każde wejście połączone jest z neuronami z warstwy ukrytej, a te dalej z neuronami kolejnej warstwy ukrytej lub warstwy końcowej. Warstwa wyjściowa reprezentuje wynik sieci. Rysunek 2.4 przedstawia poglądowy schemat sieci jednokierunkowej.
Istnieje wiele różnych modeli sieci neuronowych, w tej pracy użyta będzie jednokierunkowa sieć wielowarstwowa.



Rysunek 2.4: Przykładowy schemat sieci neuronowej
jednokierunkowej wielowarstwowej
(czerwone - warstwa wejściowa, niebieskie - warstwy ukryta,
żółte - warstwa wyjściowa)

2.2.4 Uczenie sieci neuronowej

Sieć neuronowa do poprawnego rozwiązania danego zagadnienia potrzebuje wcześniejszego treningu. Najpopularniejszym rodzajem uczenia jest uczenie nadzorowane. Podczas trenowania sieci neuronowej do pakietu N danych wejściowy x dołączony jest oczekiwany wynik działania sieci t. Po zwróceniu przez sieć wyniku y (ang. forward-pass) następuje sprawdzenie zgodności z oczekiwanym wynikiem. Reprezentacją różnicy jest funkcja kosztu (ang. loss function) [10]. Można ją obliczyć stosując średnią entropie krzyżową (ang. cross-entropy), a w przypadku problemu klasyfikacji jej wersję binarną (ang. binary cross-entropy).

$$J(\Theta) = -\frac{1}{N}\sum_{n=1}^{N}(t_n log y_n + (1-t_n)log(1-y_n)) \tag{2.10}$$

Θ jest oznaczeniem wartości wszystkich wag w sieci.

Wsteczna propagacja (ang. back propagation) to metoda zmiany wartości wag pomiędzy neuronami. Dla każdej wagi połączeń między neuronami obliczana jest pochodna cząstkowa $\frac{\partial J(\Theta)}{\partial w_{ij}}$, którą dla całej sieci można oznaczyć jako $\nabla_\Theta J(\Theta)$. [8]

$$\frac{\partial J(\Theta)}{\partial w_{ij}} = \frac{\partial J(\Theta)}{\partial y_i}\frac{\partial y_i}{\partial s_i}\frac{\partial s_i}{\partial w_{ij}} \tag{2.11}$$

Pomijając obliczenia [14], wynik dla wartości wag dla neuronów warstwy wyjściowej wynosi

$$\frac{\partial J(\Theta)}{\partial w_{ji}} = (y_i - t_i)y_j \tag{2.12}$$

W przypadku kolejnych warstw oblicza się δ, czyli błąd poszczególnych neuronów. Dla neuronów warstwy wyjściowej błąd wynosi:

$$\delta_i = y_i - t_i \tag{2.13}$$

Dla neuronów warstw ukrytej

$$\delta_{ji} = \sum_{i}^{N} \delta_{j+1,i} w_{j+1,i} \tag{2.14}$$

Wtedy dla wartości wag dla neuronów warstw ukrytej pochodna cząstkowa wynosi

$$\frac{\partial J(\Theta)}{\partial w_{ij}} = \delta_j y_i f'(s_j) \tag{2.15}$$

W obliczeniu gradientu potrzebna jest wartość pochodnej funkcji aktywacji, stąd wymóg, by funkcja aktywacji była różniczkowalna.

Epoka - jest to przeprowadzenie uczenia dla całego zbioru danych treningowych (*forward-pass* oraz *back propagation*).
Batch (ang. paczka) - jest to część zbioru danych, dla którego przeprowadzamy pojedyncze obliczenie funkcji kosztu i wsteczną propagacje.
Iteracja - jest to ilość paczek *batch* potrzebnych do przeprowadzenia jednej epoki. [8]

Wartości wag w następnej iteracji są obliczane ze wzoru

$$w_{ij_{t+1}} = w_{ij_t} - \eta \frac{\partial J(\Theta_t)}{\partial w_{ij}} \tag{2.16}$$

Parametr η nazywamy współczynikiem uczenia i wpływa on na szybkość zmiany wag w procesie uczenia. Istnieje kilka możliwości obliczenia funkcji kosztu i gradientu.
Batch gradient descent - dla całego zbioru na raz i obliczenie średniej wartości kosztu oraz gradientu.
Mini-batch gradient descent - dla każdego pojedynczego pakietu będącego częścią zbioru danych i obliczenie średniej wartości kosztu oraz gradientu.
Stochastic gradient descent (SGD) - dla każdego pojedynczego przypadku ze zbioru danych i obliczenie wartości kosztu oraz gradientu. [8]

Istnieją również algorytmy modyfikujące aktualizacje wag (2.16). Głównym problemem jest ustalenie wartości współczynnika uczenia η, powolny proces uczenia w przypadku *Batch gradient descent* oraz oscylacje wokół optymalnego rozwiązania w *Stochastic gradient descent*.
Momentum - algorytm uwzględnia poprzednią zmianę wag z mnożnikiem γ. [8]

$$v_t = \gamma v_{t-1} + \eta \nabla_\Theta J(\Theta_t) \tag{2.17}$$

$$\Theta_{t+1} = \Theta_t - v_t \tag{2.18}$$

Adagrad (ang. Adaptive Gradient) - algorytm oblicza dla każdej wagi osobno współczynnik uczenia na podstawie sumy wcześniejszych zmian wag. Dodatkowo pojawia się parametr ϵ

mający zapobiegać błędom numerycznym. Wartość parametru ustawia się najczęściej na $\epsilon = 10^{-8}$. [11]

$$g_t = \nabla_\Theta J(\Theta_t) \tag{2.19}$$

$$G_t = \sum_{j=1}^{t} g_j \tag{2.20}$$

$$\Theta_{t+1} = \Theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t \tag{2.21}$$

RMSprop - algorytm podobnie jak Adagrad wylicza dla każdej wagi osobny współczynnik uczenia co iterację na podstawie wcześniejszych zmian wag. [17]

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \tag{2.22}$$

$$\Theta_{t+1} = \Theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{2.23}$$

Adam (ang. Adaptive Moment Estimation) - algorytm podobnie jak Adagrad i RMSprop wylicza dla każdej wagi osobny współczynnik uczenia oraz podobnie jak Momentum uwzględnia poprzednie zmiany wag. Parametry najczęściej przyjmują wartość $\beta_1 = 0.9$ i $\beta_2 = 0.999$. [12]

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.24}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{2.25}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1} \tag{2.26}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2} \tag{2.27}$$

$$\Theta_{t+1} = \Theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t \tag{2.28}$$

2.2.5 Trenowanie sieci neuronowej

Zazwyczaj podczas trenowania sieci neuronowych dzielimy pakiet danych wejściowych na część treningową i testową. Dla części treningowej stosujemy proces uczenia sieci neuronowej (*forward-pass* oraz *back propagation*). Natomiast część testowa służy do sprawdzania wyników sieci dla przypadków, na których nie przeprowadzała wstecznej propagacji. [8]

Niedotrenowanie (ang underfiting) - występuje, gdy model nie daje wystarczająco dobrych wyników dla zbioru treningowego, jak i testowego. Należy wtedy zwiększyć czas trenowania lub zmienić parametry i budowę sieci.

Przetrenowanie (overfiting) - dużo częstszym problemem jest gdy model daje bardzo dobre wyniki dla zbioru treningowego, lecz niezadowalające dla zbioru testowego. Oznacza to, że sieć nauczyła się dawać w wyniku oczekiwaną wartość dla przypadków treningowych, ale nie nauczyła się rozwiązywać problemu dla wszystkich przypadków.

Dropout - jest techniką zapobiegającą ewentualnemu przetrenowaniu poprzez losowe omijanie neuronów z warstwy ukrytej i wejściowej z pewnym prawdopodobieństwiem p. Dzięki temu wynik sieci nie jest zdominowany przez jeden neuron. W praktyce omijanie neuronu polega na zerowaniu jego wyjścia. [15]

Wczesne zatrzymanie - w praktyce przy dużych zbiorach trenujących najpopularniejszą metodą jest zakończenie trenowania sieci w momencie, gdy wynik dla zbioru treningowego i testowego jest zadowalający oraz funkcja kosztu dla zbioru treningowego nie jest dużo mniejsza niż dla testowego.

Max-norm - jest to technika zapobiegająca błędom numerycznym w trakcie działania sieci. Jej działanie opiera się na ograniczeniu do pewnego zakresu λ wartości wag $w_{ij} \in (-\lambda, \lambda)$.

2.3 Klasyfikacja binarna

Modele klasyfikujące binarnie posiadają pewne parametry opisu stosowane w tej pracy.

Czas uczenia - czas trenowania klasyfikatora do rozwiązania danego problemu.

Szybkość działania - czas działania klasyfikatora, po krórym otrzymujemy wynik dla danego przypadku.

Macierz konfuzji - ocena jakości klasyfikacji binarnej, na którą składają się przypadki:

- **Prawdziwie pozytywna TP** (ang. true positive) poprawnie sklasyfikowane jako prawdziwe
- **Fałszywie pozytywna FP** (ang. false positive) błędnie sklasyfikowane jako prawdziwe
- **Fałszywie negatywna FN** (ang. false negative) błędnie sklasyfikowane jako fałszywe
- **Prawdziwie negatywna TN** (ang. true negative) poprawnie sklasyfikowane jako fałszywe

Precyzja - procent prawdziwych przypadków wśród przypadków sklasyfikowanych jako prawdziwe

$$\frac{TP}{TP + FP} \tag{2.29}$$

Dokładność - procent poprawnie sklasyfikowanych przypadków wśród wszystkich przypadków

$$\frac{TP + TN}{TP + FP + TN + FN} \tag{2.30}$$

Czułość - procent poprawnie sklasyfikowanych przypadków wśród wszystkich prawdziwych przypadków

$$\frac{TP}{TP + FN} \tag{2.31}$$

Specyficzność - procent poprawnie sklasyfikowanych przypadków wśród wszystkich fałszywych przypadków

$$\frac{TN}{FP + TN} \tag{2.32}$$

Krzywa ROC (ang. Receiver Operating Characteristic) - graficzna prezentacja określająca efektywność klasyfikatora binarnego. Na osi odciętych znajduje się 1-specyficzność, a na osi rzędnych czułość. Dodatkowo pole pod krzywą ROC określa się jako AUC (ang. Area Under the Curve), przyjmuje on wartość z przedziału <0,1> i im większą ma wartość tym lepszy jest nasz model predykcyjny.
Threshold (ang. próg) - prawdopodobieństwo, powyżej którego model klasyfikuje przypadek jako prawdziwy.

2.4 Wykorzystanie kart graficznych

GPGPU (ang. general-purpose computing on graphics processing units) - jest to technika wykonywania obliczeń na procesorach kart graficznych GPU (ang. graphics processing unit) zamiast na klasycznych procesorach CPU (ang. central processing unit). Została stworzona na potrzeby programowania równoległego, które polega na wykonywaniu niezależnych obliczeń w tym samym czasie.

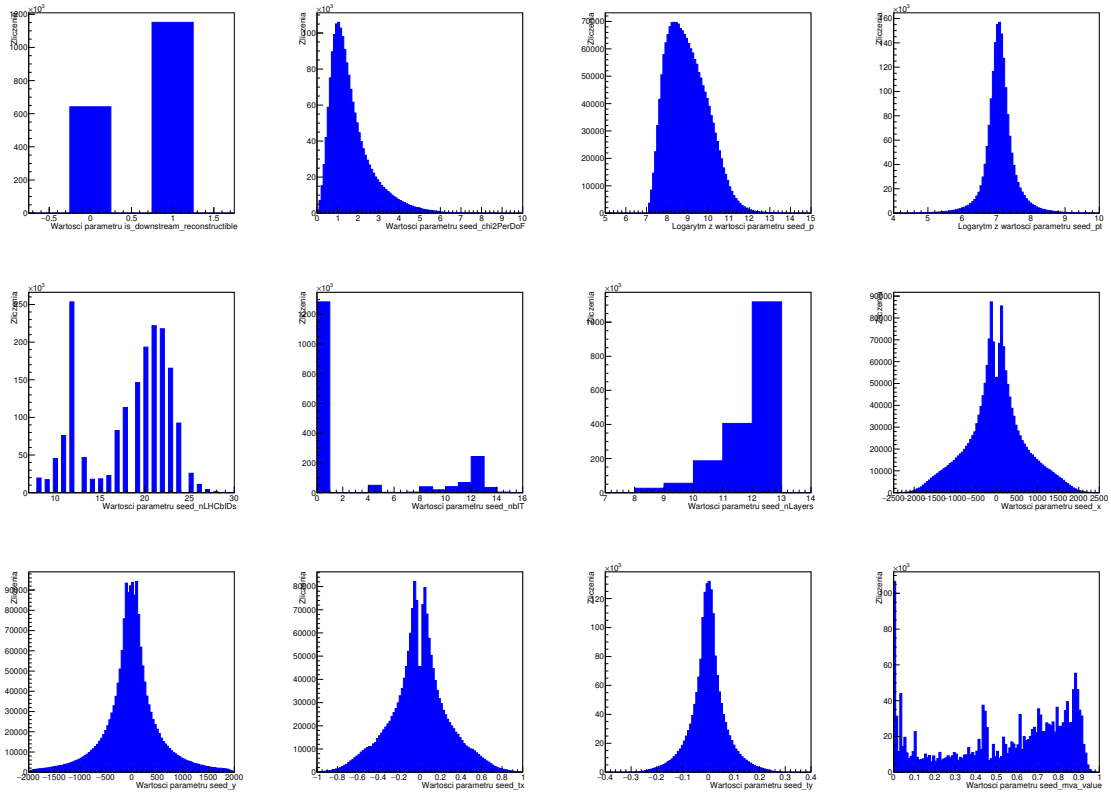
Procesory w kartach graficznych posiadają wiele rdzeni mogących działać niezależnie, dlatego wykorzystanie GPU w obliczeniach równoległych daje bardzo dobre wyniki.

CUDA (ang. Compute Unified Device Architecture) jest to architektura procesorów wielordzeniowych pozwalająca na wykonywanie obliczeń równoległych stworzona przez firmę NVidia. Do korzystania z architektury CUDA został stworzony język C for CUDA. Program napisany w architekturze CUDA wykonywany jest przez CPU, a poszczególne części programu, które mogą być wykonane równolegle na GPU. [1]

3 Analiza działania programu

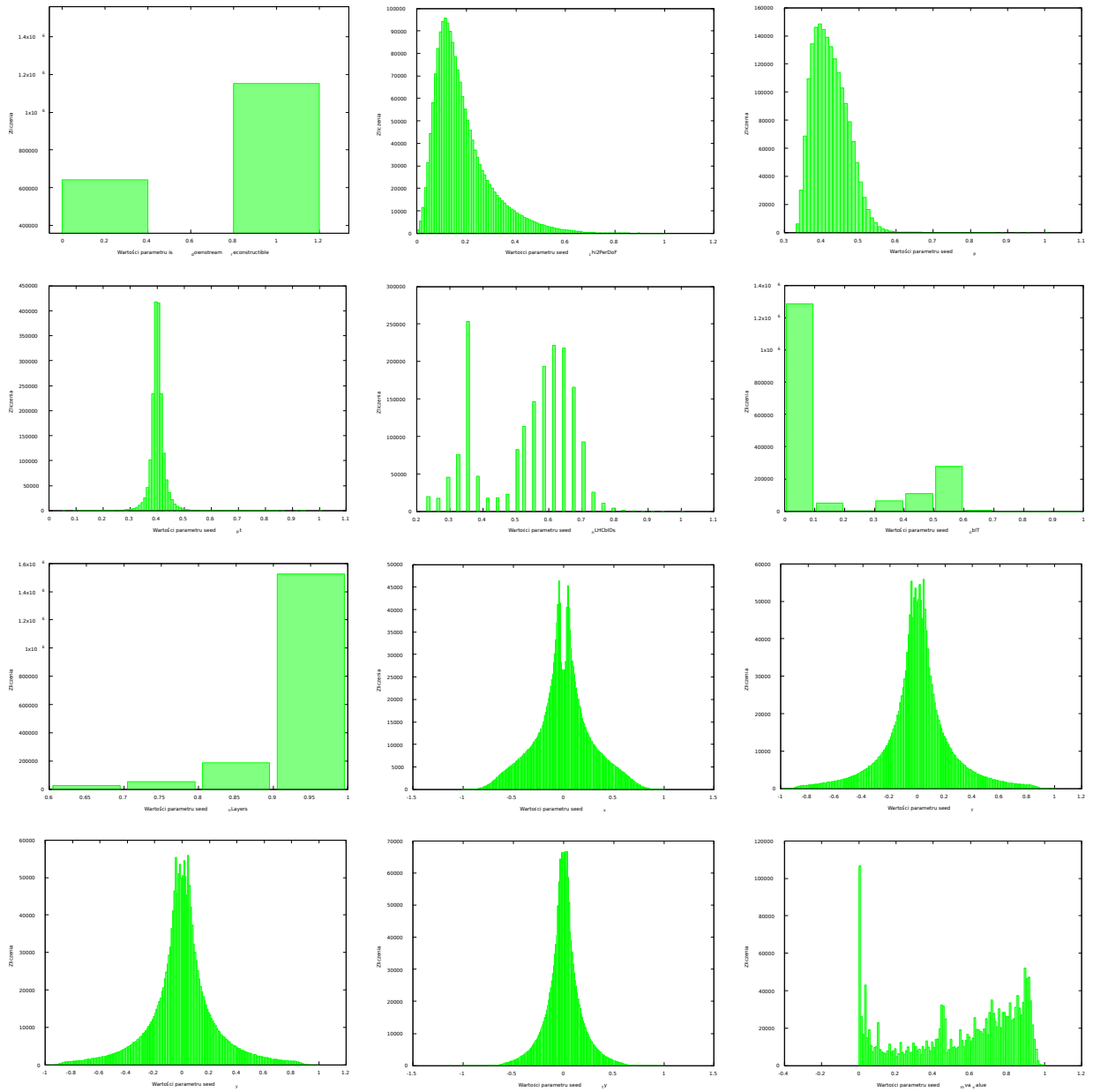
3.1 Analizowane dane

Dane przygotowane do treningu klasyfikatora zostały stworzone w symulacji Monte Carlo i zapisane w pliku ROOT. Zawierają 11 zmiennych, które można zastosować jako dane wejściowe do klasyfikatora oraz dodatkową etykietę poprawności identyfikacji będącą oczekiwaną wartością klasyfikatora. Surowe dane należy przygotować zanim będzie można wykorzystać je do trenowania sieci neuronowej. Wartości danych wejściowych powinny być z podobnego zakresu, inaczej czas trenowania się wydłuży i sieć może nie znaleść optymalnego rozwiązania. Rysunek 3.1 przedstawia rozkład wszystkich zmiennych dla wszystkich przypadków (prawdziwych i fałszywych).



Rysunek 3.1: Rozkład wartości dla poszczególnych zmiennych zbioru danych

Wszystkie zmienne zostały znormalizowane do wartości z przedziału $(-1, 1)$ lub $(0, 1)$, przy czym kilka etykiet wymagało z powodu swojego rozkładu wcześniejszego zlogarytmowania. Rozkład wartości zmiennych już po normalizacji przedstawia rysunek 3.2



Rysunek 3.2: Rozkład wartości dla poszczególnych zmiennych zbioru danych po normalizacji

Problem stanowi też dysproporcja dla etykiety sygnalizującej poprawność zrekonstruowanego sygnału. Ilość przypadków prawdziwych jest niemalże dwa razy większa od ilości przypadków fałszywych. Do treningu sieci należy używać możliwie równej ilości przypadków poprawnych jak i błędnych.

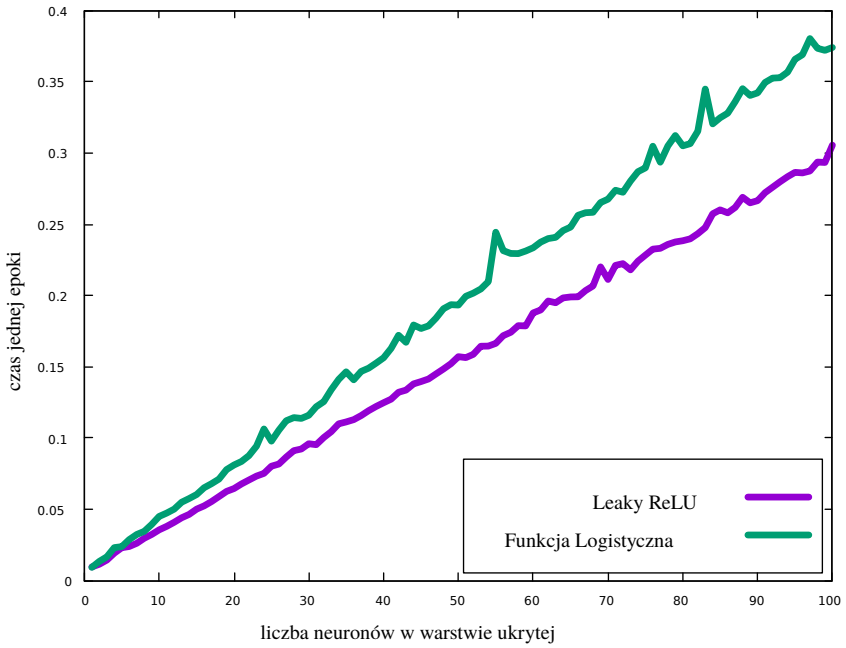
3.2 Działanie programu

Z uwagi na specyficzne środowisko programistyczne stosowane przez eksperyment LHCb autor zbudował sieć neuronową jednokierunkową w języku C++ bez użycia bibliotek do uczenia maszynowego. Napisano program do uruchamiania sieci w architekturze CUDA wraz ze zmodyfikowanym kodem do jak najwydajnieszego działania [6]. Modyfikacja polegała głównie na

zrównolegleniu możliwie jak największej liczby obliczeń. Warstwa wyjściowa składa się z dwóch neuronów, z których pierwszy określa prawdopodobieństwo otrzymania prawdziwego sygnału, a drugi określa prawdopodobieństwo otrzymania fałszywego. Liczba warstw ukrytych i neuronów w warstwie ukrytej oraz wejściowej mogą być zmieniane według potrzeb. Wartość przesunięcia *bias* zaimplementowano tylko dla pierwszej warstwy ukrytej. Aby ułatwić obliczenia i działanie programu, do warstwy wejściowej dodano dodatkowe wejście ustawione zawsze na wartość 1. Dzięki temu otrzymano parametr bias bez tworzenia dodatkowych funkcji. Według najnowszych badań brak parametru przesunięcia w dalszych warstwach ukrytych nie wpłynie w znaczącym stopniu na działanie sieci. [13]
Porównano działanie sieci neuronowej w zależności od m.in liczby warstw ukrytych i neuronów, funkcji aktywacji, metod obliczania gradientu i aktualizacji wag.

3.2.1 Analiza w zależności od funkcji aktywacji

Z powodu dużego zbioru danych użytego do treningu bardzo ważna jest szybkość uczenia. Dla sieci o jednej warstwie ukrytej, 11 wejściach i zmiennej liczbie neuronów w warstwie ukrytej dokonano analizy czasu cyklu uczenia jednej epoki 10^4 przypadków dla dwóch funkcji aktywacji, funkcji logistycznej i Leaky ReLU. Funkcją aktywacji neuronów warstwy wyjściowej w obu przypadkach była funkcja Softmax. Wyniki przedstawiono na rysunku 3.3.

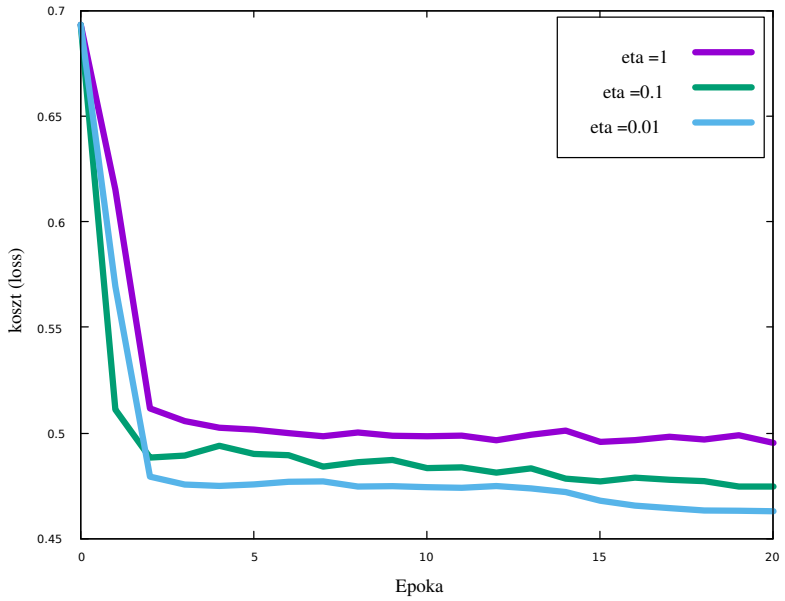


Rysunek 3.3: Porównanie wykonywania jednej epoki w zależności od ilości neuronów w warstwie ukrytej dla różnych funkcji aktywacji

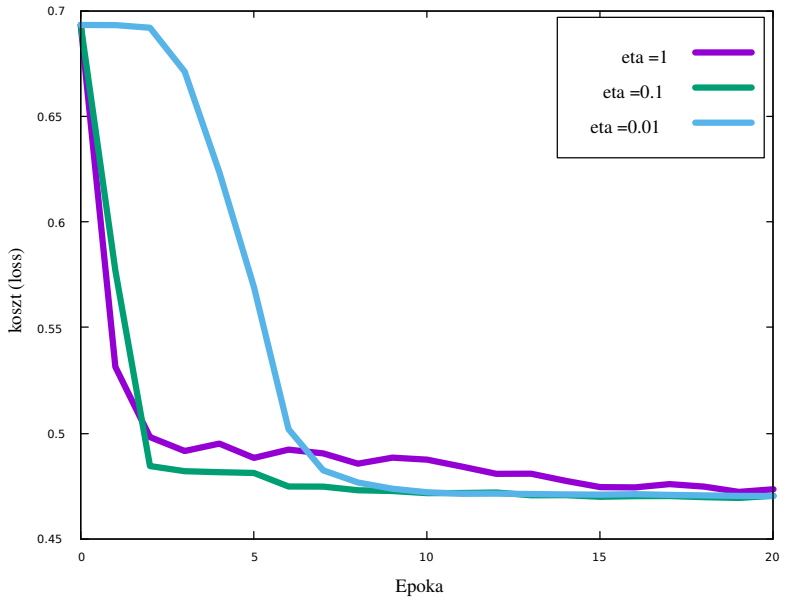
Jak widać sieć z funkcją aktywacji Leaky ReLU wykonuje się dużo szybciej co nie powinno dziwić, gdyż wyrażenie matematyczne na tę funkcję jest dużo mniej skomplikowane.

3.2.2 Analiza w zależności od wartości współczynnika uczenia

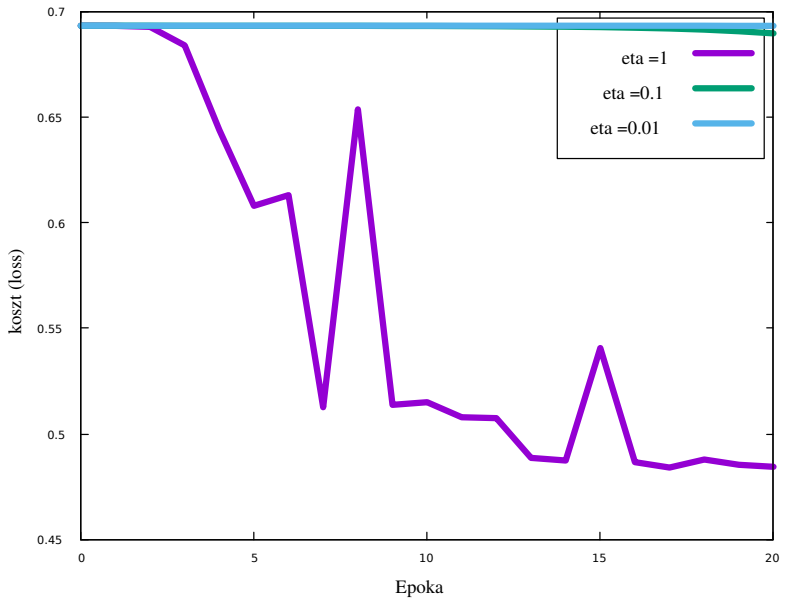
Ustalenie odpowieniego współczynnika szybkości uczenia w przypadku metod obliczania gradientu o stałej wartości η jest bardzo ważne. Dla różnych rozmiarów paczek sprawdzono wpływ współczynnika η na uczenie. Użyto sieci o jednej warstwie ukrytej z funkcją aktywacji Leaky ReLU i 100 neuronami w warstwie ukrytej, dla 10^4 przypadków w epoce. Wyniki zobrazowano na rysunkach 3.4, 3.5 i 3.6.



Rysunek 3.4: Porównanie treningu sieci dla różnych wartości η dla paczki (batch) o rozmiarze 1



Rysunek 3.5: Porównanie treningu sieci dla różnych wartości η dla paczki (batch) o rozmiarze 10

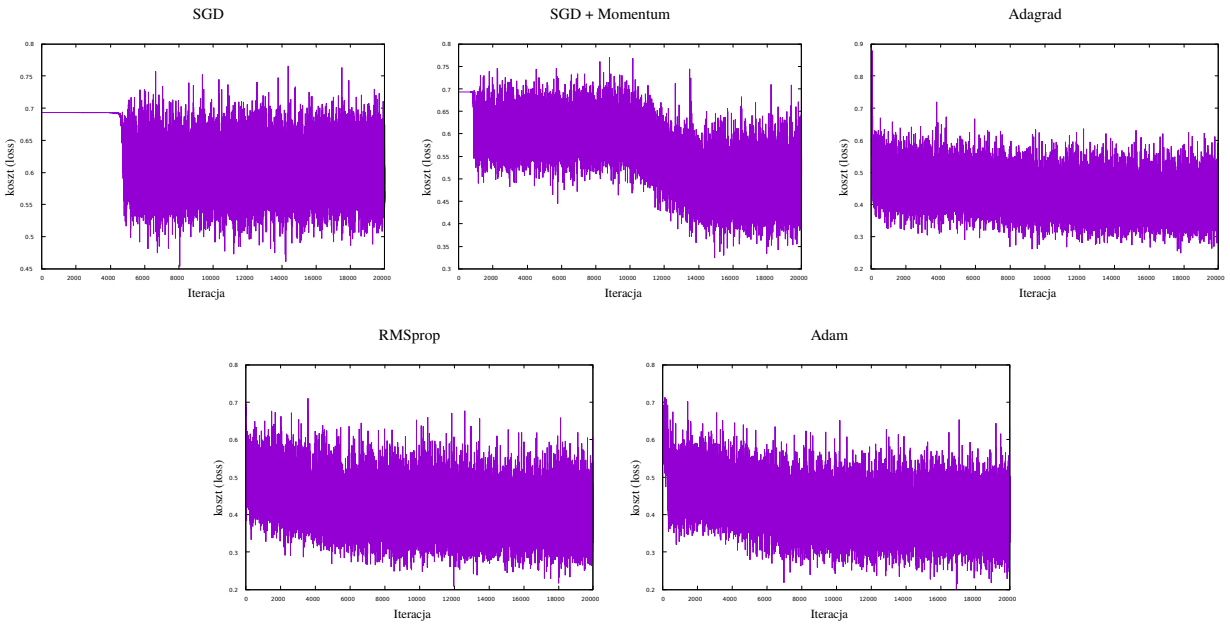


Rysunek 3.6: Porównanie treningu sieci dla różnych wartości η dla paczki (batch) o rozmiarze 1000

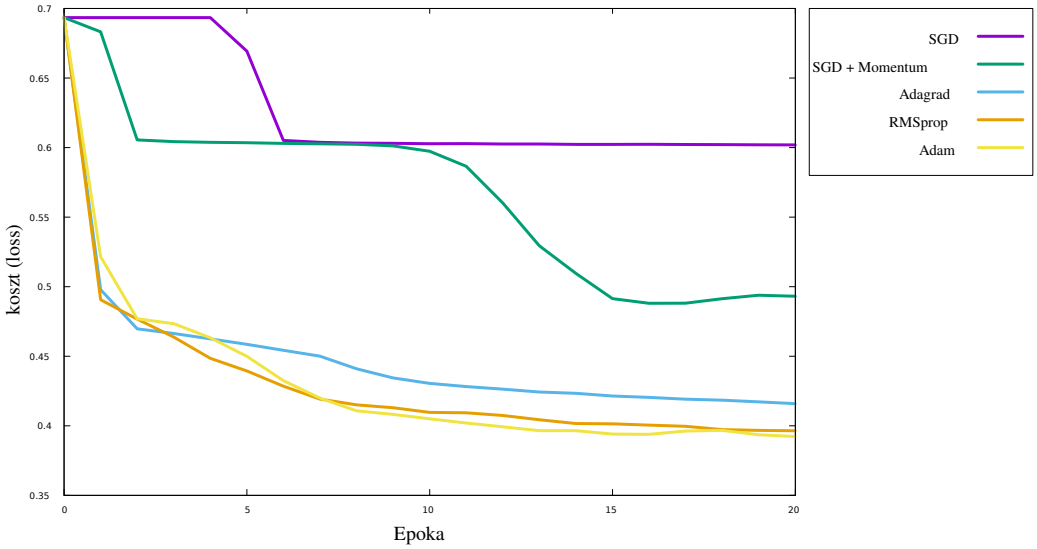
Dla paczki o rozmiarze 1 widać, że zależnie od wartości η sieć osiąga stabilne rozwiązanie wokół innej wartości i wynik bardzo zależy od współczynnika uczenia. Można to tłumaczyć tym, iż im większy współczynnik uczenia tym bardziej sieć oscyluje wokół poprawnego rozwiązania. Dla paczki o rozmiarze 10 widać, iż wybór współczynnika η ma niewielki wpływ na wynik końcowy, ale niebagatelny na czas trenowania potrzebny do osiągnięcia minimum. Dla przykładu $\eta = 0.01$ widzimy charakterystyczny płaski początek i nagły spadek po kilku epokach.
Dla paczki o rozmiarze 1000 widzimy wydłużony czas potrzebny na znalezienie minimum, a dla małych wartości współczynnika η obserwujemy charakterystyczny płaski początek ciągnący się przez ponad 30 epok.

3.2.3 Analiza w zależności od metody obliczania gradientu

Jak widać wybór rozmiaru paczki (*batch*) oraz współczynnika η ma znaczny wpływ na wynik i czas uczenia sieci. Przy większych zbiorach danych może być to kłopotliwe i czasochłonne, dlatego stosuje się modyfikacje sposobu aktualizacji wag opisane w rozdziale (2.2.4). Zgodnie z normami przyjętymi w literaturze z powodu dużego zbioru treningowego nazwa SGD będzie stosowana również dla małej liczby paczek.
Dla metod opisanych w rozdziale (2.2.4) sprawdzono wpływ wyboru metody na uczenie. Użyto sieci o dwóch warstwach ukrytych z funkcją aktywacji Leaky ReLU i 100 neuronami w warstwie ukrytej dla 10^5 przypadków w epoce i paczki (*batch*) o rozmiarze 100. Porównanie kosztu dla różnych metod przedstawiono na rysunkach 3.7 i 3.8.



Rysunek 3.7: Porównanie kosztu (*loss*) w iteracji treningu sieci dla różnych metod aktualizacji wag



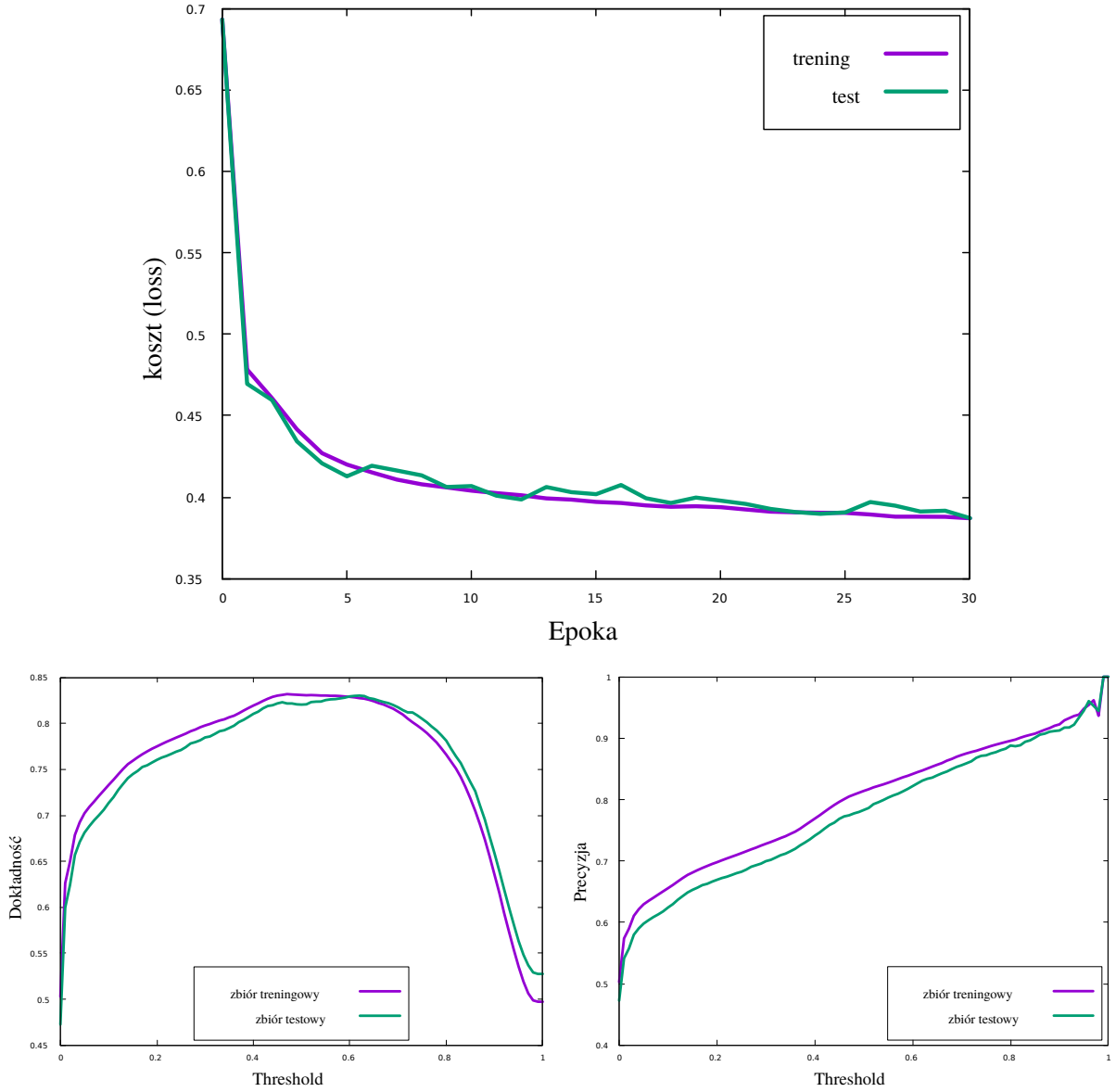
Rysunek 3.8: Porównanie kosztu (*loss*) w epoce treningu sieci dla różnych metod aktualizacji wag

Jak widać na rysunku 3.8 największym problemem w SGD jest dopasowanie współczynnika uczenia. Duża wartość spowoduje szybsze wyjście z rejonu minimum lokalnego, jednak pojawią się duże oscylacje w pobliżu minimum globalnego. SGD + Momentum również cierpi na problem dopasowania współczynnika η jednak szybciej niż SGD wychodzi z rejonu minimum lokalnego. Metody dynamicznej zmiany współczynnika uczenia osiągają optymalny wynik w nieporównywanie krótszym czasie. Pewien problem z osiągnięciem optimum posiada algorytm Adagrad z powodu stale malejącego współczynnika uczenia. Problem ten rozwiązują algorytmy

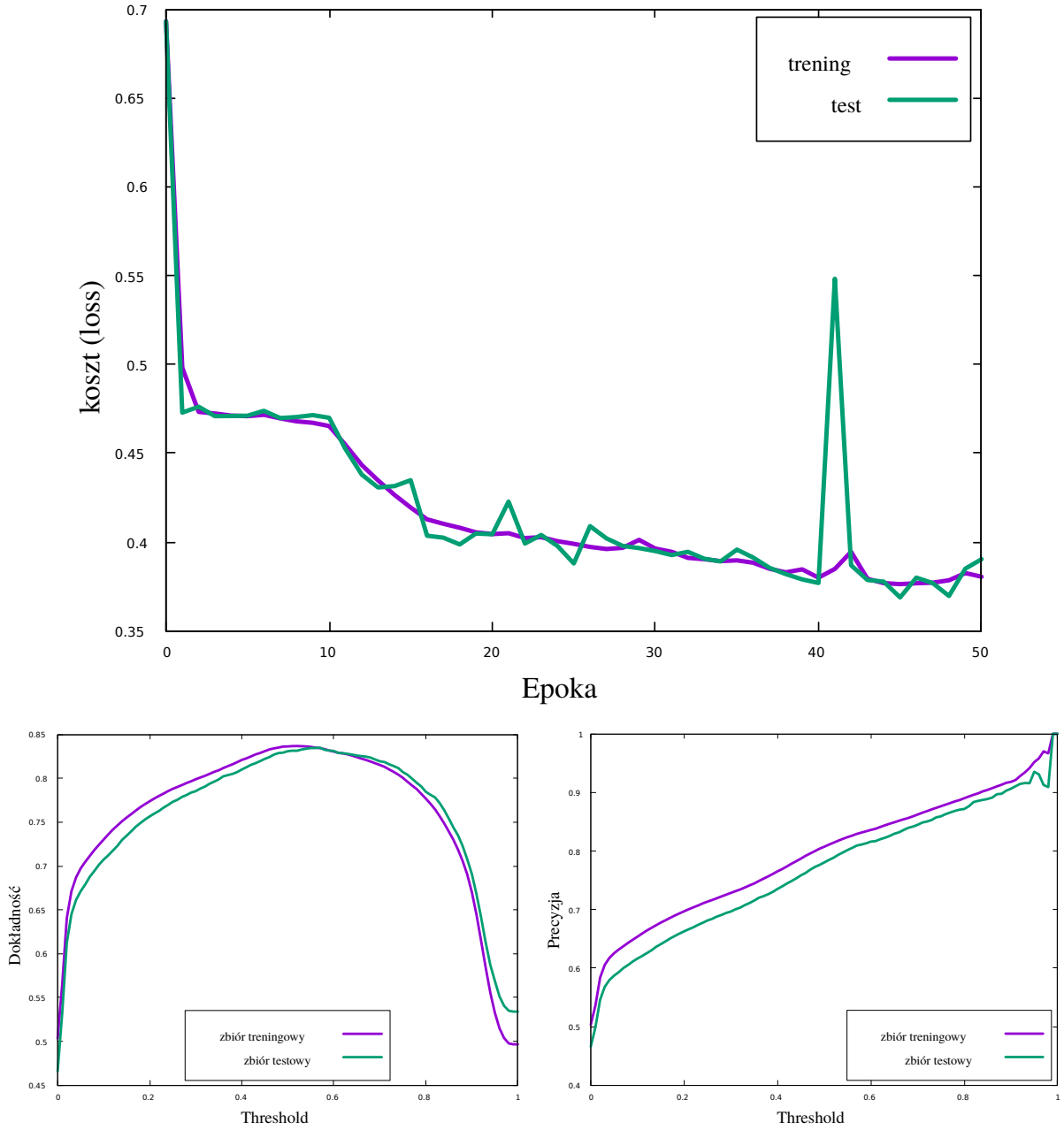
RMSprop oraz Adam. Najlepszy wynik osiąga algorytm Adam co wynika z jego rozbudowanej formy łączącej najlepsze cechy algorytmów Adagrad, RMSprop i Momentum.

3.2.4 Analiza w zależności od rozmiaru sieci

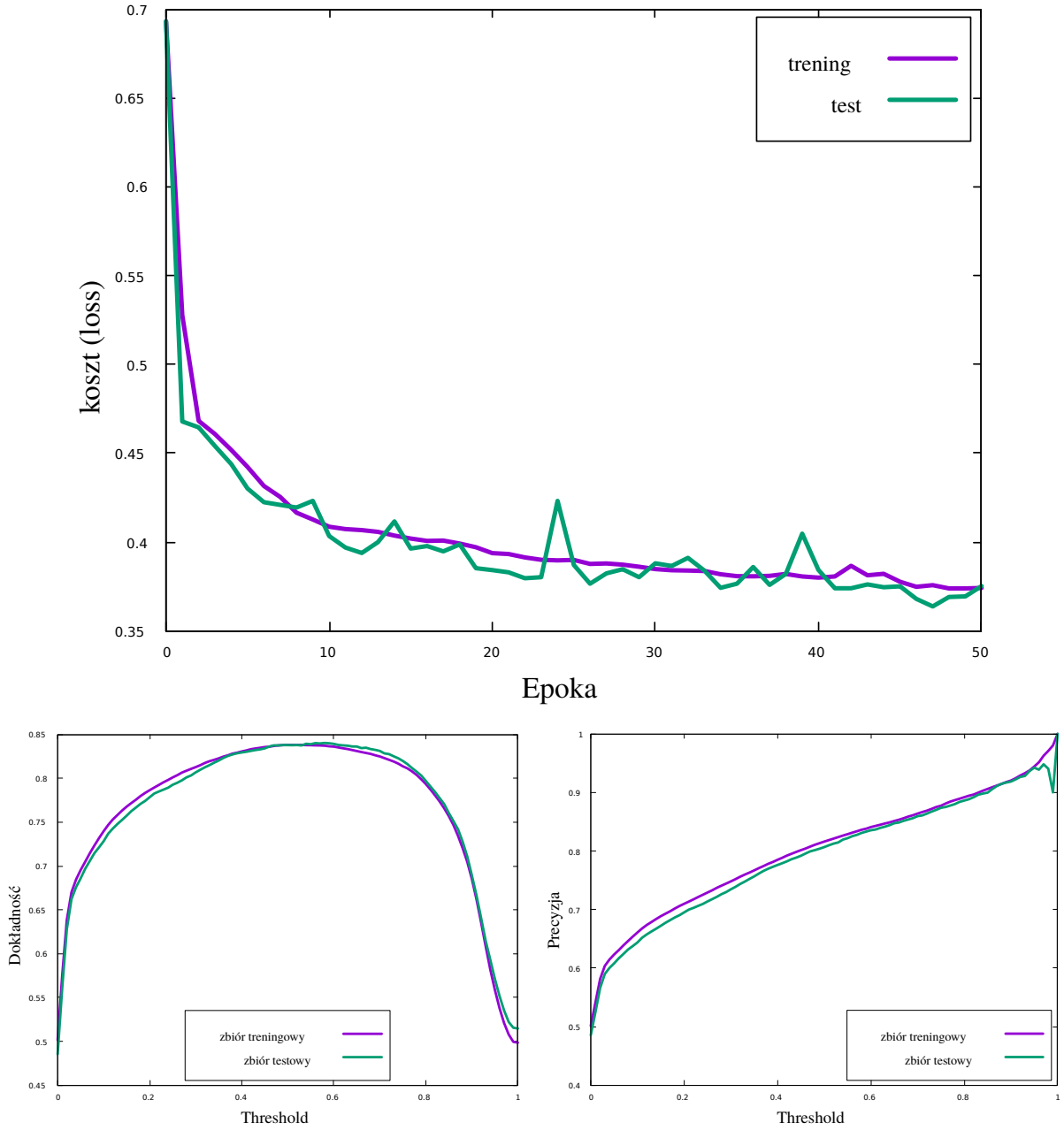
Dla 10^5 przypadków w zbiorze treningowym i 10^4 w zbiorze testowym dokonano sprawdzenia umiejętności rozwiązania problemu klasyfikacji dla zmiennej liczby warstw ukrytych i neuronów w tych warstwach. Używając algorytmu Adam do uczenia, porównano koszty (*loss*) dla zbioru treningowego i testowego. Dodatkowo sprawdzono wpływ ustawienia progu (*Threshold*) na dokładność oraz precyzje klasyfikacji. Wyniki przedstwiono na rysunkach 3.9, 3.10, 3.11 i 3.12.



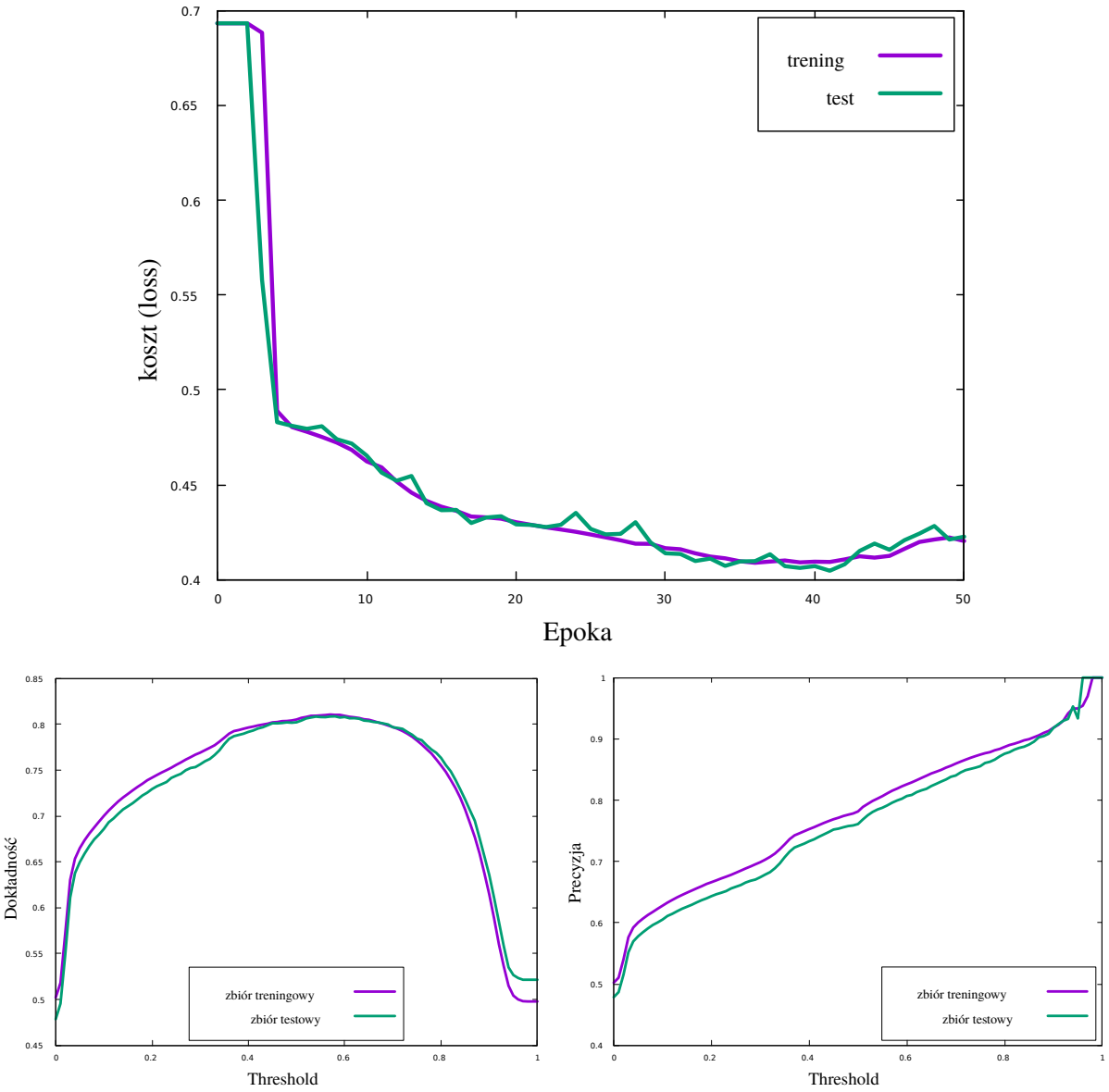
Rysunek 3.9: Koszt (*loss*) w treningu oraz wpływ ustawienia progu (*Threshold*) na dokładność oraz precyzję dla sieci o jednej warstwie ukrytej zawierającej 100 neuronów.



Rysunek 3.10: Koszt (*loss*) w treningu oraz wpływ ustawienia progu (*Threshold*) na dokładność oraz precyzję dla sieci o dwóch warstwach ukrytych zawierających po 100 neuronów.



Rysunek 3.11: Koszt (*loss*) w treningu oraz wpływ ustawienia progu (*Threshold*) na dokładność oraz precyzję dla sieci o trzech warstwach ukrytych zawierających po 50 neuronów.



Rysunek 3.12: Koszt (*loss*) w treningu oraz wpływ ustawienia progu (*Threshold*) na dokładność oraz precyzję dla sieci o czterech warstwach ukrytych zawierających po 50 neuronów.

Sieć z jedną warstwą ukrytą podczas treningu funkcja kosztu osiągneła minimum po ok. 20 epokach. Jako, że zbiór treningowy jest bardzo duży, w trakcie treningu widać chwilowe przetrenowania, które natychmiast znikają. (Rysunek 3.9)

Sieć z dwoma warstwami ukrytymi osiągneła minimum funkcji kosztu dopiero po ok. 45 epokach. Koszt dla zbioru testowego podobnie jak dla sieci z jedną warstwą ukrytą chwilami się zwiększa, sygnalizując przetrenowanie jednak po kilku epokach znów oscyluje wokół kosztu dla zbioru treningowego. Bardzo spektakularnie wygląda wybuch wartości kosztu w okolicach 40 epoki jednak natychmiast wraca do poziomu kosztu zbioru treningowego. (Rysunek 3.10)

Sieć z trzema warstwami ukrytymi osiągneła minimum funkcji kosztu po ok. 40 epokach. Oscylacje kosztu dla zbioru testowego wokół kosztu dla zbioru treningowego mają większą amplitudę w porównaniu do poprzednich sieci. Dla tej sieci osiągamy najlepszy wynik wytrenowania sieci w funkcji kosztu. Dodatkowo wykres dokładności i precyzji dla zbioru treningowego jak i testowego jest niemal identyczny. (Rysunek 3.11)

Sieć z czterema warstwami ukrytymi osiągneła minimum funkcji kosztu po ok. 40 epokach, jednak następnie koszt dla zbioru testowego oraz treningowego zwiększył się co świadczy, że było to minimum lokalne i sieć jest jeszcze niedotrenowana. (Rysunek 3.12)

Dla każdego przykładu wybrano próg (Threshold) o największej dokładności i przedstawiono wyniki w tablicach 1 i 2.

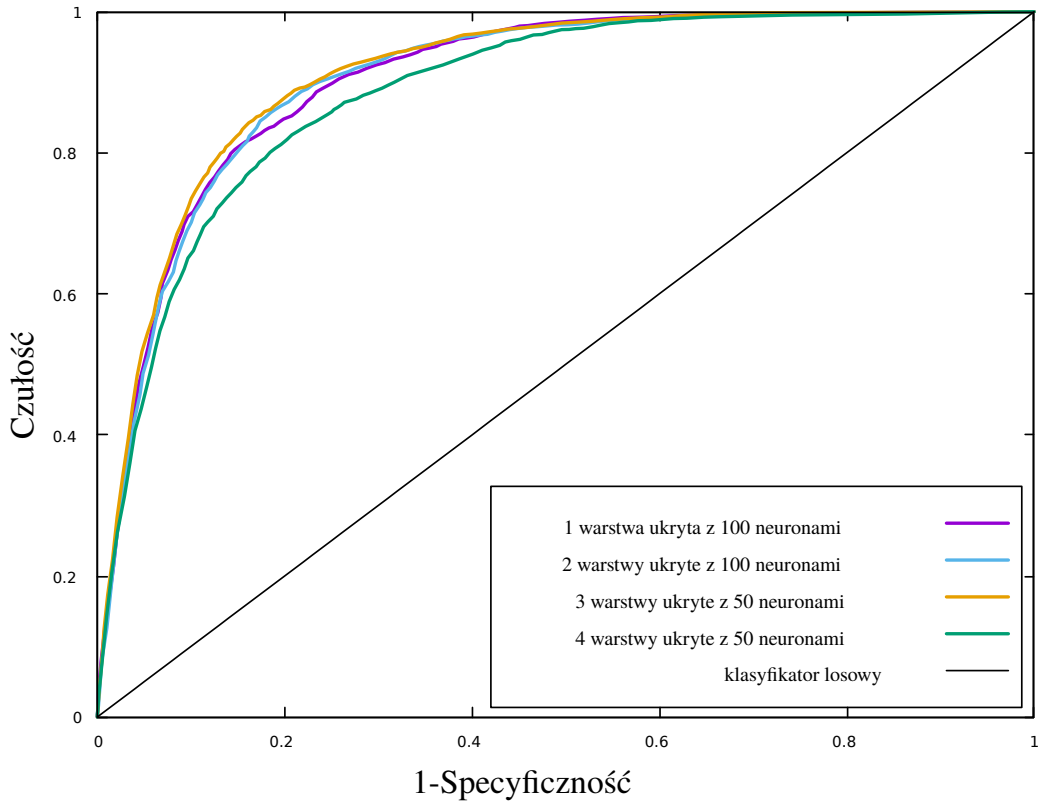
Sieć	Threshold	Funkcja kosztu	Dokładność	Precyzja
1 warstwa ukryta z 100 neuronami	0.60	0.3869	0.8289	0.8416
2 warstwy ukryte z 100 neuronami	0.56	0.3825	0.8350	0.8261
3 warstwy ukryte z 50 neuronami	0.58	0.3741	0.8371	0.8361
4 warstwy ukryte z 50 neuronami	0.58	0.4202	0.8101	0.8189

Tablica 1: Resultaty sieci dla zbioru treningowego

Sieć	Threshold	Funkcja kosztu	Dokładność	Precyzja
1 warstwa ukryta z 100 neuronami	0.60	0.3870	0.8294	0.8219
2 warstwy ukryte z 100 neuronami	0.56	0.3849	0.8345	0.8052
3 warstwy ukryte z 50 neuronami	0.58	0.3752	0.8403	0.8307
4 warstwy ukryte z 50 neuronami	0.58	0.4225	0.8085	0.7994

Tablica 2: Resultaty sieci dla zbioru testowego

Dla każdego przykładu sporządzono krzywą ROC dla zbioru testowego i przedstawiono na rysunku 3.13. W tabeli 3 zestawiono wyniki AUC dla zbioru testowego.

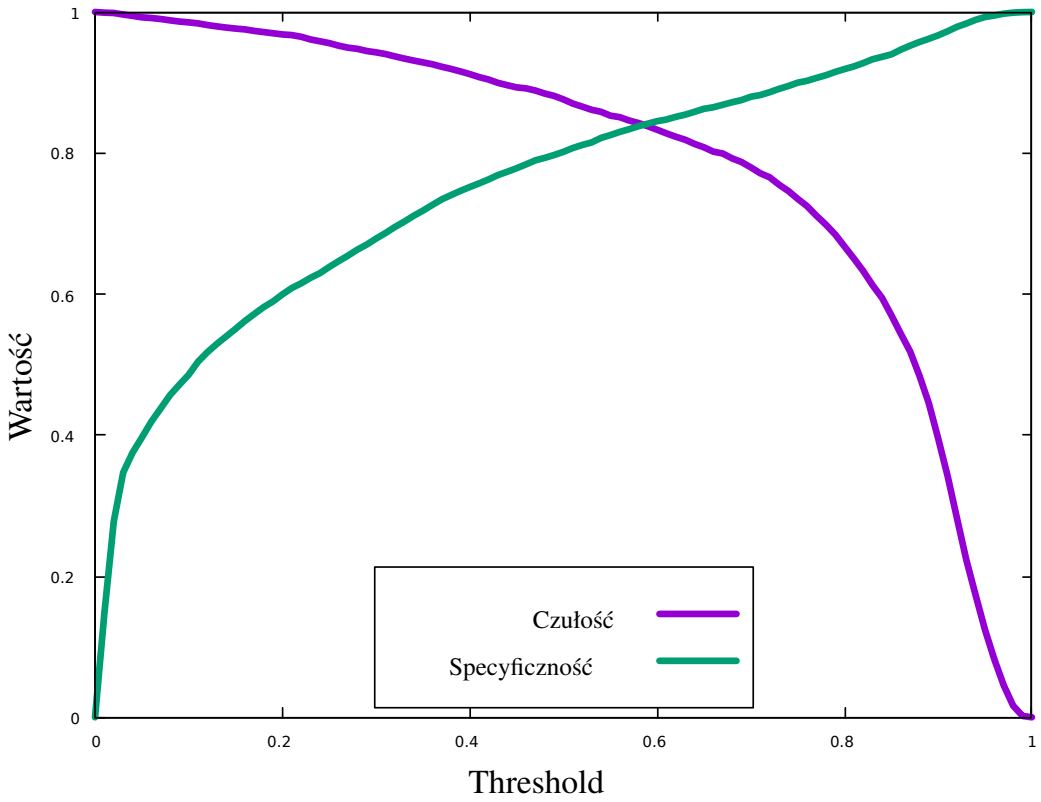


Rysunek 3.13: Porównanie krzywej ROC dla badanych sieci dla zbioru testowego

Sieć	AUC
1 warstwa ukryta z 100 neuronami	0.90496
2 warstwy ukryte z 100 neuronami	0.90518
3 warstwy ukryte z 50 neuronami	0.91120
4 warstwy ukryte z 50 neuronami	0.88677

Tablica 3: Wartości AUC dla badanych sieci dla zbioru testowego

Dla sieci o 3 warstwach ukrytych z 50 neuronami ukrytymi otrzymaliśmy najniższą wartości kosztu oraz największą wartość AUC, dlatego wybrano tą sieć jako przykład w klasyfikatorze w naszym eksperymencie. Parametrem ważniejszym od dokładności w naszym przypadku jest jak największa czułość i jak najmniejsza specyficzność. Wykres 3.14 przedstawia te dwie wartości w zależności od ustalonego progu.

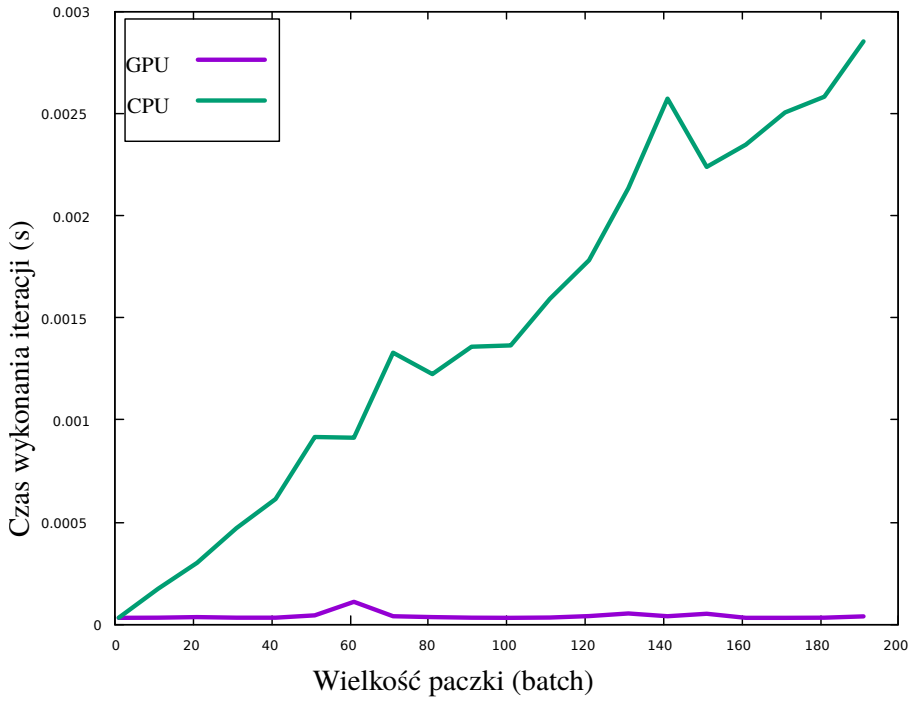


Rysunek 3.14: Wykres zależności czułości i swoistości od progu (Threshold) dla sieci z 3 warstwami ukrytymi z 50 neuronami dla zbioru testowego

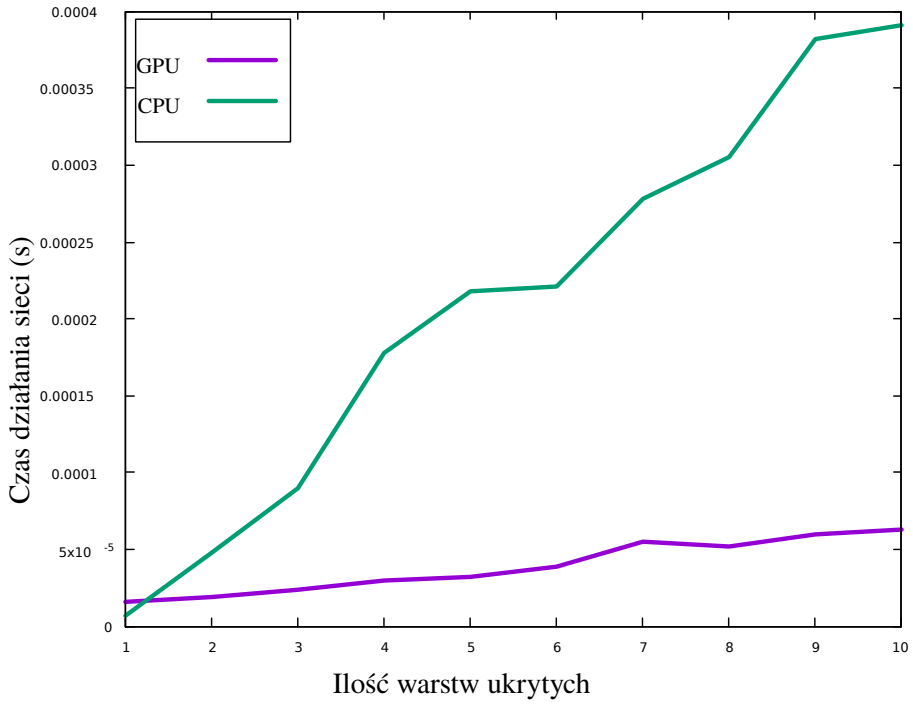
Jak widać na wykresie ustalenie wartości 3.14 progu o wartości 0.1 sprawi, że tracąc ok. 1.5% prawdziwych przypadków zlikwidujemy ok. 50% fałszywych sygnałów. Ustalenie progu o warości 0.4 pozwoli na likwidację ponad 75% fałszywych sygnałów przy stracie niecałych 10% prawdziwych przypadków.

3.2.5 Analiza czasów wykonywania CPU i GPGPU

Porównano czas uczenia sieci dla obliczeń wykonywanych przez program w C++ wykorzystujący procesor CPU i program napisany w języku C for CUDA wykorzystującym obliczenia na karcie graficznej GPU. Do testów wykorzystano procesor Intel$^{(R)}$ Core$^{(TM)}$ i7-4720HQ oraz kartę graficzną GeForce GTX 950M. Wyniki zestawiono na rysunkach 3.15 i 3.16.



Rysunek 3.15: Porównanie czasu wykonania jednej iteracji w zależności od wielkości paczki (*batch*)



Rysunek 3.16: Porównanie czasu działania sieci (forward-pass) w zależności od ilości warstw ukrytych

Jak widać wpływ rozmiaru paczki (*batch*) ma główny wpływ na lepszą wydajność obliczeń na GPU. Spowodowane jest to niezależnością obliczeń pomiędzy paczkami podczas trenowania sieci, dzięki czemu można rozdzielić obliczenia na kilka wątków. Struktura programu była stwo-

rzona pod jak najszybszy trening dla paczki (*batch*) o rozmiarach powyżej kilku przypadków. Mimo to porównano czas potrzebny do obliczenia wartości sieci (forward-pass) dla zmiennej liczby warstw ukrytych. Jak widać już dla dwóch warstw ukrytych obliczenia wykorzystujące GPU są bardziej wydajne. Autor chce nadmienić, że modyfikacja programu i zastosowanie zaawansowanych technik programowania równoległego w architekturze CUDA zgodnie z jego wiedzą powinny spowodować dużo większą wydajność podczas działania na GPU.

4 Podsumowanie

W poniższej pracy stworzono oraz dokonano analizy działania sztucznej sieci neuronowej. Zbudowana sieć miała rozwiązywać problem klasyfikacji przypadków w algorytmie Downstream tracking stosowanym w eksperymencie LHCb do rekonstrukcji śladów cząstek długożyciowych. Cel pracy można uznać za osiągnięty.

Napisana od podstaw jednokierunkowa sieć neuronowa dla trzech warstw ukrytych i 50 neuronów w każdej warstwie ukrytej osiągnęła dokładność klasyfikacji na poziomie 84%. Dodatkowo przy odpowiednio dobranym progu klasyfikacji likwidowane jest ponad 50% fałszywych sygnałów przy jednoczesnej stracie prawdziwych przypadków poniżej 2%.

Autor pragnie zauważyć, że jest to wynik bardzo dobry, możliwy dzięki zastosowaniu najnowszych algorytmów i rozwiązań z dziedziny uczenia maszynowego. Zrozumienie i zaimplementowanie algorytmów uczenia sieci było najbardziej czasochłonnym procesem. Z powodu rezygnacji z użycia gotowych bibliotek do uczenia maszynowego pojawiało się mnóstwo problemów z działaniem programu, które autor musiał rozwiązać. Całość programu została zaadaptowana do technologii CUDA, co po optymalizacji działania kodu wybitnie zwiększyło wydajność.

Doświadczenie zdobyte podczas tworzenia i testowania kodu jest dla autora dodatkowym atutem, gdyż będzie niezwykle przydatne w przyszłych wyzwaniach. Autor pragnie zauważyć, że możliwa jest dalsza optymalizacja programu w architekturze CUDA [3] aby osiągnąć wydajniejsze czasowo rozwiązania. Praca może potencjalnie zostać użyta przy badaniach nad nowymi algorytmami dla zmodernizowanego detektora.

Klasyfikator został udostępniony do użycia [4].

Literatura

[1] http://www.nvidia.pl/object/cuda-parallel-computing-pl.html.

[2] http://cds.cern.ch/record/1461250/files/LHCb-PUB-2012-010.pdf?version=1.

[3] http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

[4] https://github.com/PiotrWNowak/praca-inzynierska.

[5] Tracking strategies used in lhcb. https://twiki.cern.ch/twiki/bin/view/LHCb/LHCbTrackingStrategies.

[6] NVIDIA Corporation. Nvidia cuda c programming guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[7] R Forty. The lhcb upgrade, 24 Jul 2012. http://cds.cern.ch/record/1461250/.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. Microsoft Research https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper.pdf.

[10] Katarzyna Janocha and Wojciech Marian Czarnecki. On loss functions for deep neural networks in classification. https://arxiv.org/pdf/1702.05659.pdf.

[11] Yoram Singer John Duchi, Elad Hazan. Adaptive subgradient methods for online learning and stochastic optimization. http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[12] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. https://arxiv.org/pdf/1412.6980.pdf.

[13] Dubravko Majetic, Danko Brezak, Branko Novakovic, and Josip Kasac. Neural network without bias neuron for hidden layer. pages 239–240, 01 2005.

[14] Peter Sadowski. Notes on backpropagation. https://www.ics.uci.edu/~pjsadows/notes.pdf.

[15] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. Department of Computer Science University of Toronto http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf.

[16] Liliana Teodorescu. Artficial neural networks in high-energy physics. https://cds.cern.ch/record/1100521/files/p13.pdf.

[17] Tijmen Tieleman. Overview of mini-batch gradient descent. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.