AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Fizyki i Informatyki Stosowanej

Praca Inżynierska

Kamil Hałat

kierunek studiów: **informatyka stosowana**

Algorytm do rekonstrukcji śladów cząstek długożyciowych w eksperymencie LHCb oparty o sztuczne sieci neuronowe

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

Kraków, luty 2020

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.”, oświadczam, że niniejszą pracę dyplomowąwykonałem(-am) osobiście i samodzielnie i nie korzystałem(- am) ze źródeł innych niż wymienione w pracy. Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. — Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

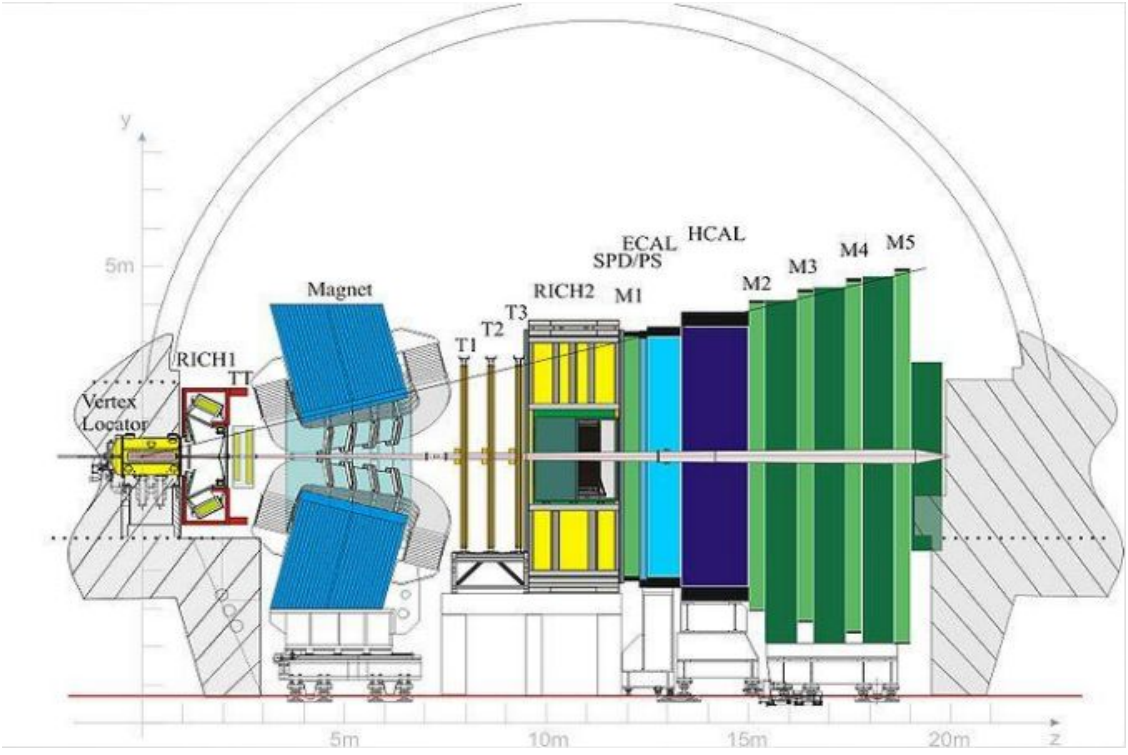
..... (czytelny podpis)

1. Wprowadzenie

Eksperyment LHCb (Large Hadron Collider beauty experiment) [1] - jest jednym z 8 eksperymentów prowadzonych w europejskim ośrodku badań jądrowych CERN, specjalizującym się w fizyce cząstek elementarnych. Eksperyment polega na badaniu produktów powstałych na skutek zderzeń cząstek rozpędzonych w akceleratorze LHC. Są to przede wszystkim kwarki niskie zwane również kwarkami pięknymi — stąd "beauty" w nazwie. Detektor przystosowany jest do pomiaru cząstek łamiących symetrię parzystości oraz ładunku.

1.1. Budowa detektora

Detektor wykorzystuje fakt że dwa b-hadrony powstają przeważnie skierowane w tą samą stronę. Elementy detektora ułożone są w kształt stożka co przedstawiono na rysunku 1.1, zwiększa prawdopodobieństwo zaobserwowania pary cząstek. Głównymi elementami detektora istotnymi dla tej pracy są trzy detektory: Vortex Locator (VELO), TT oraz detektory T1, T2, oraz T3 tworzące stację T. Istotny jest także magnes odchylający cząstki posiadające ładunki.

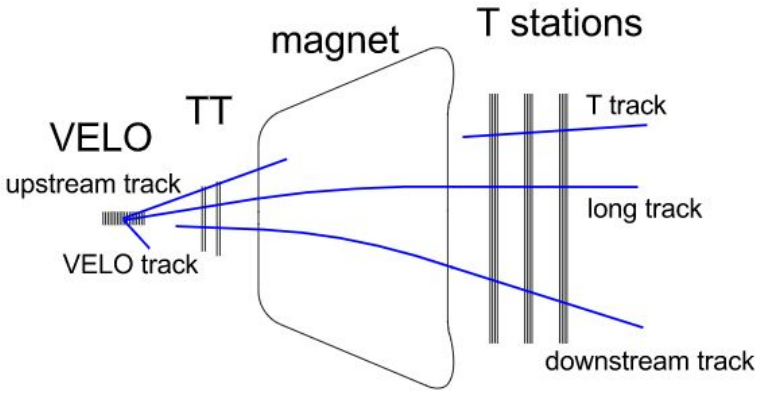


Rysunek 1.1: schemat detektora LHCb

1.2. Typy śladów

Ślady dzielone są względem tego w jakich detektorach zostają zrekonstruowane [2]. W tej pracy wzięto pod uwagę ślady oznaczone na rysunku 1.2 jako downstream track i stworzono klasyfikator binarny którego zadaniem jest odrzucenie wszystkich śladów posiadających trafienia w detektorze VELO, klasyfikujących się jako downstream tracks od tych które takich trafień nie posiadają.

.



Rysunek 1.2: Podział śladów rekonstruowanych przez detektor

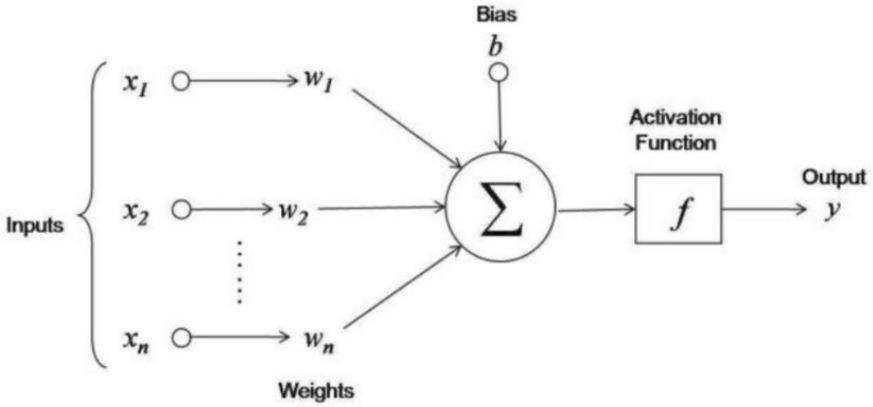
1.3. Cele pracy

Głównym celem pracy było stworzenie narzędzi, pozwalających na szybkie dobranie modelu uczenia maszynowego w zależności od posiadanych danych i założonego rezultatu oraz prostego i bezbolesnego sposobu na znalezienie optymalnych hiperparametrów. Założeniem pracy było wykorzystanie w tym celu biblioteki Pytorch [3] oraz przedstawienie korzyści jakie niesie ze sobą ta biblioteka, a także uzyskanie wydajnego modelu służącego do klasyfikacji śladów.

2. Wstęp teoretyczny

2.1. Problem klasyfikacji i sztuczna sieć neuronowa

Klasyfikacja polega no podzieleniu zbioru danych na dwie lub więcej klas na podstawie parametrów poszczególnych rekordów. Problem ten można rozwiązać wykorzystując nadzorowane algorytmy uczenia maszynowego. Jednym z takich algorytmów jest perceptron [4]. Jest on oparty o prosty model składający się z jednego sztucznego neuronu:

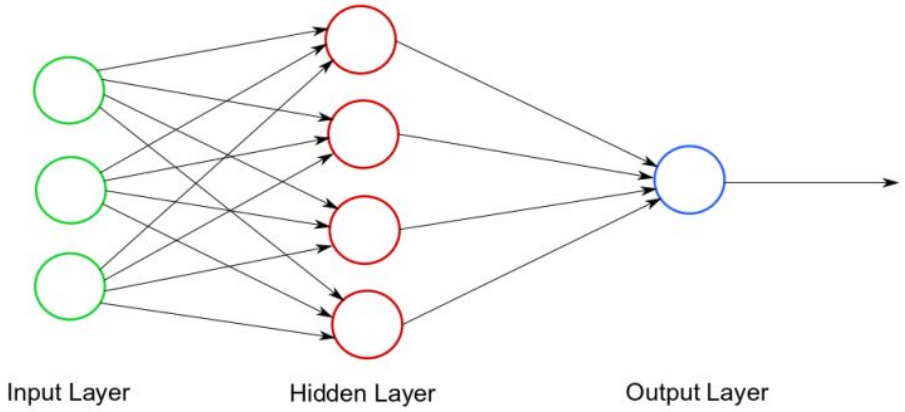


Rysunek 2.1: schemat klasyfikatora [2]

Takie neurony posiadają wiele wejść oraz jedno wyjście. Każde z wejść mnożone jest przez jego wagę i sumowane razem ze sobą oraz z wartością biasu, a wynik wprowadzany jest do funkcji aktywacyjnej.

$$y(x) = f(\sum_{i=1}^{n} x_n \cdot w_n + b) \; gdzie \; x = \{x_1 ... \; x_n\} \tag{1}$$

Wykorzystując perceptron jako podstawowy budulec możemy w bardzo prosty sposób uzyskać wielowarstwową sieć neuronową [5] gdzie każdy z neuronów przekazuje swoje wyjście do każdego neuronu z następnej warstwy. Warstwy pomiędzy warstwą wejściową a wyjściową nazywamy ukrytymi. Każdy neuron wyjściowy odpowiada jednej klasie obiektów.
Dla klasyfikacji binarnej wystarczy jeden neuron wyjściowy.



Rysunek 2.2: Schemat sieci neuronowej z jedną ukrytą warstwą

Wzorem na wynik wielowarstwowej sieci neuronowej jest zagnieżdżona funkcja z wzoru (1) gdzie wejściami do każdego neuronu są wyjścia wszystkich neuronów z warstwy poprzedniej.

$$s = y^{(L)}(y^{(L-1)}(...y^{(1)}(x))) \tag{2}$$

gdzie $y^{(l)}$ jest tensorem wyników l-tej warstwy

Używając odpowiednich (innych niż liniowe) funkcji aktywacyjnych oraz dobierając odpowiednie wagi, jesteśmy w stanie odwzorować dowolną funkcję matematyczną, z dokładnością zależną od ilości ukrytych warstw i liczby neuronów w warstwach i złożoności funkcji aktywacyjnych.

2.2. Funkcje aktywacji

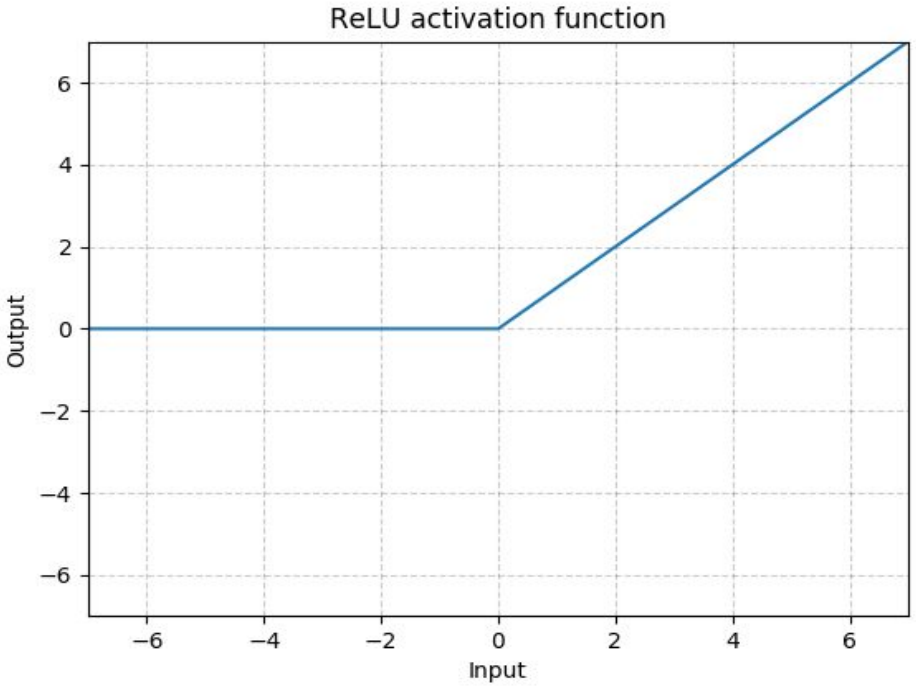
Do rozwiązania problemu klasyfikacji wykorzystuje się głównie 4 funkcje aktywacji w zależności od potrzeb. Funkcje te dostępne są zazwyczaj w bibliotekach udostępniających narzędzia do tworzenia modeli uczenia maszynowego ale w przypadku, tak jak w przypadku biblioteki Pytorch [6].

2.2.1. ReLU

Jest to bardzo prosta funkcja podobna do funkcji liniowej jednak równa zeru dla wartości ujemnych. Sprawia to że jej pochodna jest bardzo tania w obliczeniu a w przeciwieństwie do funkcji liniowej użyta w

modelu sieci neuronowej umożliwia mu aproksymacje złożonych funkcji. Wykorzystywana głównie w ukrytych warstwach modelu. Jest jedną z najpopularniejszych funkcji aktywacji i praktycznie całkowicie wyparła z użytku funkcję Sigmoid z ukrytych warstw.

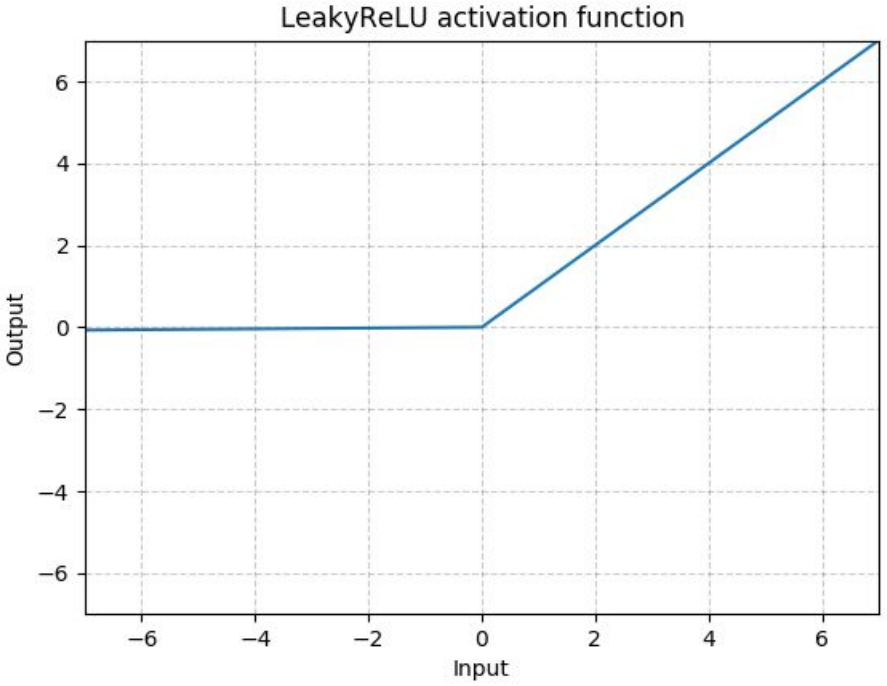
$$ReLU(x) = max(0, x) \tag{3}$$



2.2.2. LeakyReLU - (ang rectified linear unit)

Podobna do funkcji ReLU jednak dla wartości ujemnych zachowuje się jak funkcja liniowa o bardzo małym ujemnym nachyleniu. Pomaga to w przypadku kiedy gradienty modelu zbyt szybko się zerują i model przestaje efektywnie się uczyć.

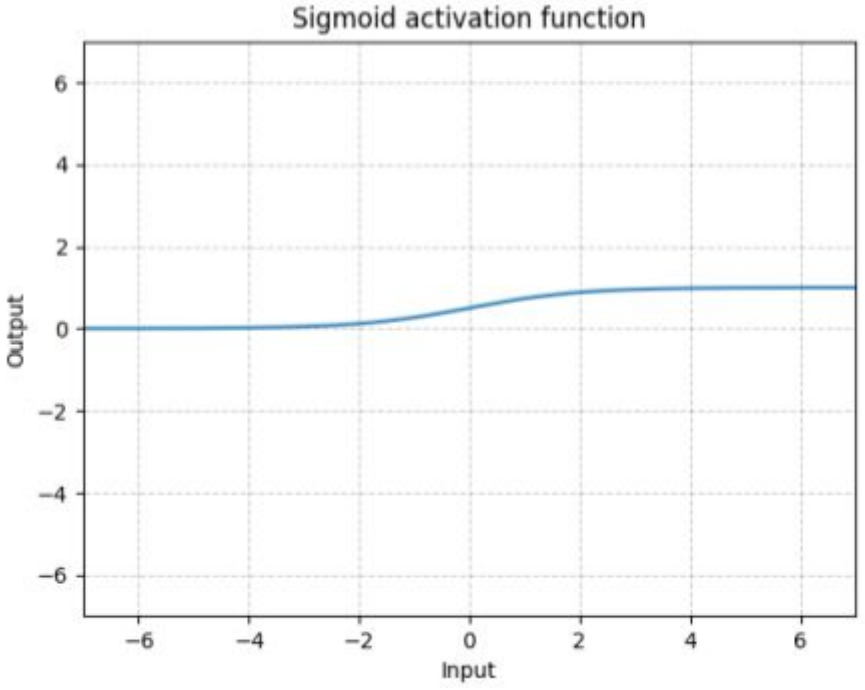
$$LeakyReLU(x) = max(0, x) + negative\_slope \cdot min(0, x) \tag{4}$$



2.2.3. Sigmoid

Funkcja używana głównie do modeli rozwiązujących problemy klasyfikacji. W dzisiejszych czasach jako funkcja warstwy wyjściowej.

$$Sigmoid(x) = \frac{1}{1+exp(-x)} \tag{5}$$



2.2.4. Softmax

Funkcja wykorzystywana do klasyfikacji wieloklasowej manipuluje tensorem tak aby elementy n-wymiarowego wyjścia sumowały się do wartości 1 i leżały w przedziale [0, 1].

$$Softmax(x_i) = \frac{exp(x_i)}{_j exp(x_j)} \quad (6)$$

2.3. Trenowanie modelu

Aby model dobrze klasyfikował obiekty musimy ustawić odpowiednie wagi dla każdego neuronu. Dobranie ich ręcznie byłoby bardzo czasochłonne i całkowicie pozbawione sensu. Dlatego wykorzystuje się w tym celu algorytmy optymalizujące które robią to za nas poprzez propagację wsteczną (z ang. backpropagation) [5]. Polegają one na obliczeniu błędu inaczej kosztu naszego modelu po wprowadzeniu do niego danych wejściowych i porównaniu wyjścia z oczekiwanymi odpowiedziami.
Najprostszą funkcją błędu jest:

$$C(x,\ y) = |x - y| \tag{7}$$

jednak dla problemu klasyfikacji binarnej można wykorzystać funkcję BCELoss czyli Binary Cross Entropy Loss:

$$C(x, y) = y \cdot log\ x\ +\ (1 - y) \cdot log(1 - x) \tag{8}$$

Algorytm oblicza błąd modelu i na jego podstawie aktualizuje kolejne wagi zaczynając od ostatniej warstwy posuwając się wstecz, minimalizując błąd.

2.3.1. Propagacja

Najpierw dla każdego neuronu każdej warstwy wyznaczany jest wektor a korzystając z wzoru (2).
Po wykonaniu tego zabiegu dla ostatniej warstwy otrzymamy także predykcje modelu i na tej podstawie obliczamy błąd korzystając ze wzoru (7).

2.3.2. Propagacja wsteczna

Po obliczeniu wartości funkcji straty obliczamy pochodne cząstkowe dla każdej wagi i biasu zaczynając od warstwy wyjściowej:

$$\delta^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \tag{9}$$

co pozwala nam obliczyć błędy dla pozostałych warstw:

$$\delta^l = ((w^{l+1})^T \cdot \delta^{l+1}) \odot f^{l\prime}(z^l) \tag{10}$$

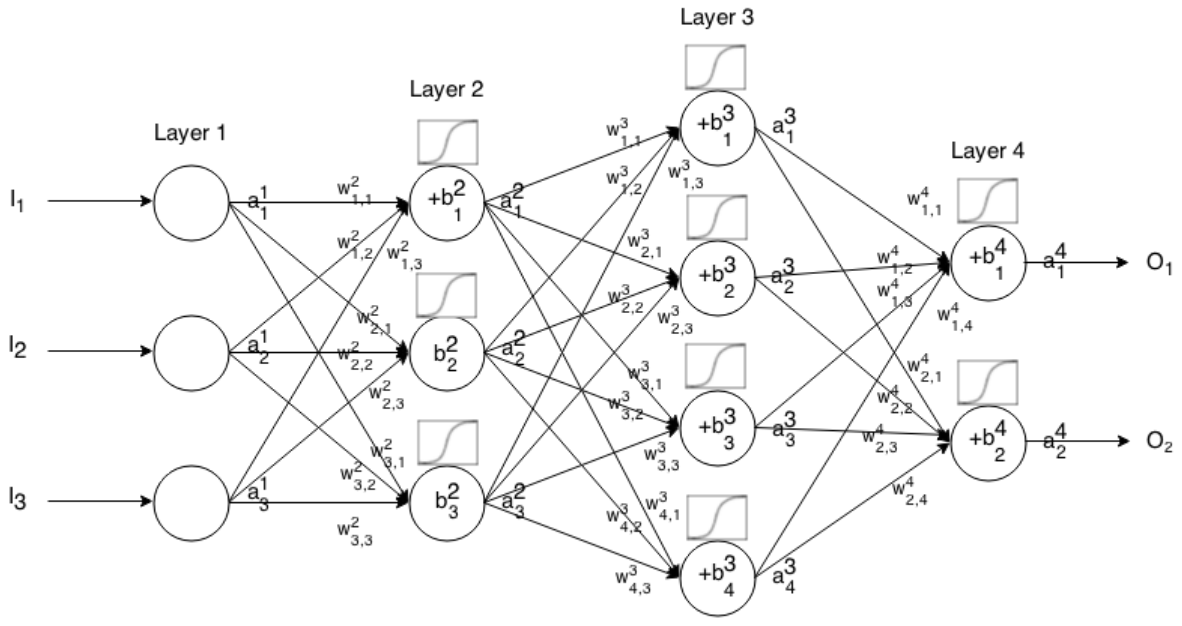
Mając obliczone błędy dla każdej warstwy możemy w końcu zaktualizować wagi (10) i biasy (11)

$$w_{ik}^{l+} = w_{ik}^{l} - \eta \cdot \delta_i^l \cdot s_k^{l-1} \tag{11}$$

$$b_i^{l+} = b_i^l - \eta \cdot \delta_l^l \cdot s_k^{l-1} \tag{12}$$

Gdzie i oraz k wskazują na poszczególne wagi i biasy a l ukryte warstwy co pokazuje rysunek 2.3. + w wyrażeniu: w_{ik}^{l+} wskazuje że jest to nowa wartość dla wagi.

η - (ang. learning rate) - hiper parametr definiujący jak szybko model ma się uczyć, powinien być mniejszy od 0.01.



Rysunek 2.3: Schemat sieci neuronowej z opisanymi wagami

2.3.3. Epoki, metryki i terminologia

Jeden cykl propagacji i propagacji wstecznej nazywamy epoką.

Po każdej epoki dobrze wyznaczyć metryki [7] modelu takie jak:

dokładność - (ang. accuracy) ACC

$$ACC = \frac{TP + TN}{P+N} \tag{13}$$

precyzja - (ang. precision) PPV

$$PPV = \frac{TP}{TP+FP} \tag{14}$$

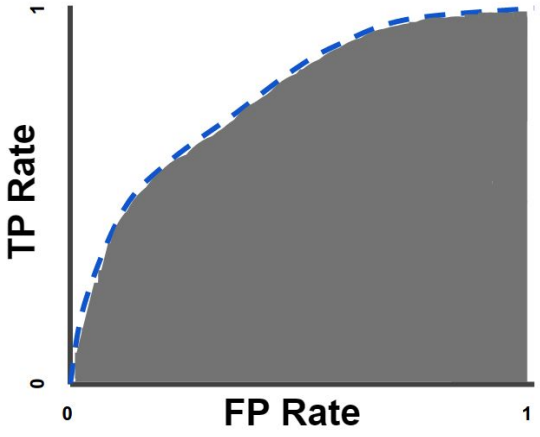
czułość - (ang. recall) TPR

$$TPR = \frac{TP}{TP+FN} \tag{15}$$

gdzie TP - (true positive) poprawnie zakwalifikowany jako klasa,
TN - (true negative) poprawnie odrzucony,
FP - (false positive) fałszywy alarm,
FN - (false negative) niepoprawnie zakwalifikowany jako klasa

Krzywa ROC:

Jest to krzywa pokazująca w jaki sposób predykcje modelu klasyfikują dane jako TP i FP w zależności od wybranego progu klasyfikacji.



(ang: Area under Curve):

Metryka z przedziału [0, 1], wyznaczona na podstawie krzywej ROC, jest polem pod tą krzywą. Im bliżej wartości 1 tym lepsza klasyfikacja modelu. Modele o AUC niższym niż 0.5 radzą sobie gorzej niż wylosowanie klas lub klasyfikują na odwrót.

Podczas trenowania modelu powinno się podzielić dane na treningowe i testowe. W proporcjach 0.1 do 0.2 w zależności od wielkości zbioru danych. Jak wskazują nazwy dane treningowe wykorzystujemy podczas uczenia modelu a dane testowe służą tylko do wyznaczenia metryk.
Dobrze jest także podzielić zbiór danych na mniejsze fragmenty tak zwane batch'e (z ang. partie) ma to kluczowy wpływ na to jak szybko uczy się nasza funkcja.
Porównując metryki wyznaczone dla danych testowych oraz dla danych treningowych jesteśmy w stanie stwierdzić czy nie dochodzi do nadmiernego dopasowania naszego modelu do danych treningowych tak zwanego overfitting'u (z ang. nadmierne dopasowanie). Jest to zjawisko wynikające z przeuczenia i powoduje ono słabszą dokładność w styczności z danymi spoza zbioru testowego.

2.4. Optymalizacja hiperparametrów

Hiperparametry są to różne parametry naszego modelu zależne głównie od użytkownika w przeciwieństwie do wag i biasów.
Są to na przykład learning rate, ilość ukrytych warstw, ilość neuronów w warstwach czy też funkcje aktywacji.
Istnieje wiele metod na znalezienie optymalnych lub suboptymalnych hiperparametrów:

2.4.1. Metoda prób i błędów

polega na ręcznym dobieraniu hiperparametrów i próbie znalezienia optymalnego ich ustawienia. Niepraktyczne dla modeli z dużą ilością hiperparametrów jednak jeżeli mamy do czynienia z prostym modelem który dobrze znamy i wystarczy nam suboptymalne rozwiązanie może być całkowicie wystarczająca.

2.4.2. Wyszukiwanie w siatce (ang. Grid Search)

Polega na stworzeniu n-wymiarowej siatki gdzie n to liczba hiperparametrów i wybraniu wartości granicznych dla każdego z nich. Algorytm Grid Search tworzy wtedy model dla każdego punktu siatki i

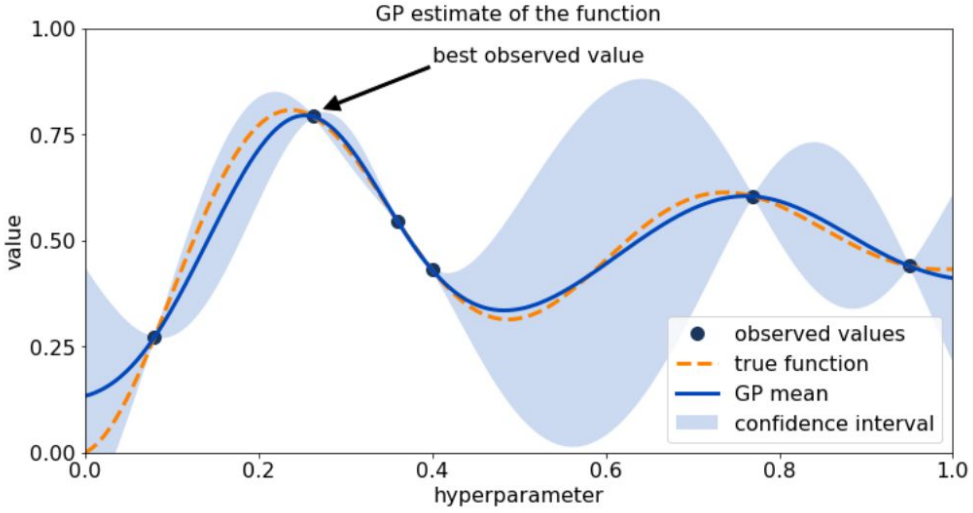
sprawdza ich wyniki znajdując w ten sposób model o optymalnych hiperparametrach.
Metoda bardzo czasochłonna nie praktyczna dla modeli potrzebujących dużo czasu na trening.

2.4.3. Losowe Wyszukiwanie w siatce (ang. Random Search)

Bardzo podobne do poprzedniej metody jednak w tym przypadku algorytm losuje zadaną ilość punktów z siatki dla których trenuje model.
Może być bardzo efektywne w połączeniu z metodą prób i błędów ponieważ dzięki niej możemy zawęzić poszukiwania.

2.4.4. Optymalizacja Bayesowska (ang. Bayesian optimization) [8]

Bardzo wydajna metoda znajdowania optymalnych hiperparametrów. Pierwszym krokiem jest wyznaczenie wyniku dla kilku losowych zestawień hiperparametrów podobnie jak w przypadku metody Random Search. Następnie na ich podstawie można wyznaczyć funkcję zastępczą przechodzącą przez wszystkie wcześniej znalezione punkty i wybraniu nowego punktu w maksimum wyznaczonej funkcji i obliczamy kolejną funkcję zastępczą uwzględniając nowo znaleziony punkt.



Rysunek 2.5: przedstawia metodę optymalizacji Bayesowskiej

3. Wykorzystane technologie

Poniżej zostały przedstawione rozwiązania technologiczne wybrane przez autora podczas pisania pracy.

3.1. Język Python - Wysoko poziomowy, interpretowany język programowania zorientowany obiektowo. Posiada rozbudowaną społeczność uczenia maszynowego co daje ułatwiony dostęp do gotowych rozwiązań i poradników z tego zagadnienia oraz ogromną liczbę bibliotek i ekosystemów pozwalających tworzyć zaawansowane modele uczenia maszynowego w prosty sposób co czyni go idealnym środowiskiem pracy w tym temacie.

3.1.1. PyTorch - biblioteka oparta o operacje na tensorach podobnie jak numpay, jednak z naciskiem na silną akceleracją GPU, oraz rozbudowaną architekturę pozwalającą tworzyć zaawansowane modele głębokiego uczenia maszynowego.
Jej głównym atutem jest obliczanie gradientów funkcji przeprowadzanych na tensorach poprzez metodę backward co jest dużym atutem podczas przeprowadzania algorytmu propagacji wstecznej. Do tego korzysta on z metod i funkcji zaimplementowanych w języku C++ co sprawia że jest bardzo wydajna.

3.1.2. sklearn - łatwa w użyciu biblioteka pozwalająca na tworzenie prostych predefiniowanych modeli uczenia maszynowego, w tym przypadku użyta głównie do wstępnego przetworzenia danych oraz wyznaczenia metryk modelu.

3.1.3. bayesian-optimization - wykorzystany do zoptymalizowania hiperparametrów modelu przy pomocy optymalizacji bayesowskiej.

3.2. Platforma Jupyter notebook - jest to aplikacja webowa pozwalająca na tworzenie projektów w postaci dokumentów zawierających kod, obliczone wyniki i wizualizację danych. Pozwala na uruchomienie lokalnego serwera i pracę nad kodem zdalnie z innej maszyny. Dzięki temu mamy możliwość zamknięcia danej sesji, a po ponownym nawiązaniu z nią kontaktu możemy bez problemu uzyskać obliczone w trakcie przerwy wyniki. Przydaje się to zwłaszcza w sytuacji kiedy okres oczekiwania na wynik algorytmu wynosi więcej niż 24h.

3.3. Root - modularna naukowa platforma służąca do pracy z big data, wykorzystana w projekcie do odczytania i przetwarzania otrzymanych danych danych w formacie .root.

Platforma ta posiada api pozwalające na korzystanie z niej w języku Python, PyRoot.

3.3.1. RootPy - jest wraperem ułatwiającym pracę z biblioteką pyroot udostępniającym metodę csv użytą w tym projekcie.

4. Przebieg Projektu

4.1. Konwersja i analiza danych

4.1.1. Konwersja danych do pliku csv

W projekcie użyto danych dostarczonych w postaci pliku .root zawierającego ponad 30 milionów rekordów.
Po odczytaniu danych z pliku i wpisaniu nazw kolumn przy pomocy poniższego skryptu:

```
from rootpy.io import root_open

myfile = root_open('some_file.root')
mytree = myfile.Particle_Data
labels = mytree.branchnames
print(labels)
```

Otrzymaliśmy:

```
['UT_cos', 'UT_cosT', 'UT_dxDy', 'UT_lhcbID', 'UT_planeCode', 'UT_sinT',
'UT_tanT', 'UT_weight', 'UT_xAtYEq0', 'UT_xAtYMid', 'UT_xT', 'UT_zAtYEq0',
'track_charge', 'track_chi2', 'track_chi2PerDoF', 'track_nLHCbIDs',
'track_p', 'track_phi', 'track_position_phi', 'track_position_r',
'track_position_x', 'track_position_y', 'track_position_z', 'track_pt',
'track_tx', 'track_ty', 'track_pseudoRapidity', 'track_ghostProbability',
'eventID', 'particle_key', 'particle_eta', 'particle_fullInfo',
'particle_hasScifi', 'particle_hasUT', 'particle_hasVelo',
'particle_isDown', 'particle_isDown_noVelo', 'particle_isLong',
'particle_isLong_andUT', 'particle_ovtx_x', 'particle_ovtx_y',
'particle_ovtx_z', 'particle_evtx_x', 'particle_evtx_y',
'particle_evtx_z', 'particle_p', 'particle_phi', 'particle_pid',
'particle_pt', 'particle_px', 'particle_py', 'particle_pz']
```

Z powyższych kolumn wybrane zostały tylko te zaczynające się od track_ oraz kolumny takie jak: particle_hasVelo, particle_isDown, particle_isDown_noVelo, particle_isLong, 'particle_isLong_andUT na których podstawie podzielimy zbiór danych testowych na klasy.

Aby zapisać dane do pliku csv wykorzystano metodę:
rootpy.tree.Tree.csv():

```
labels = ['track_charge', 'track_chi2',
'track_chi2PerDoF', 'track_nLHCbIDs', 'track_p',
'track_phi', 'track_position_phi', 'track_position_r',
'track_position_x', 'track_position_y',
'track_position_z', 'track_pt', 'track_tx', 'track_ty',
'track_pseudoRapidity', 'particle_hasVelo',
'particle_isDown', 'particle_isDown_noVelo',
'particle_isLong', 'particle_isLong_andUT']
out_file = open("data.csv", "w")
mytree.csv(branches=labels, stream=out_file)
```

4.1.2. Podział danych na klasy

Po wczytaniu danych do struktury *dataframe* dostarczonej przez bibliotekę pandas dane zostały podzielone na klasy *True* i *False* wykorzystując warunki:
particle_isDown == 1 & particle_hasVelo == 0 & particle_isLong_andUT == 0
(particle_isDown == 0 | particle_hasVelo == 1) & particle_isLong_andUT == 0
Co podzieliło dane na: 1769906 True i 3549788 False

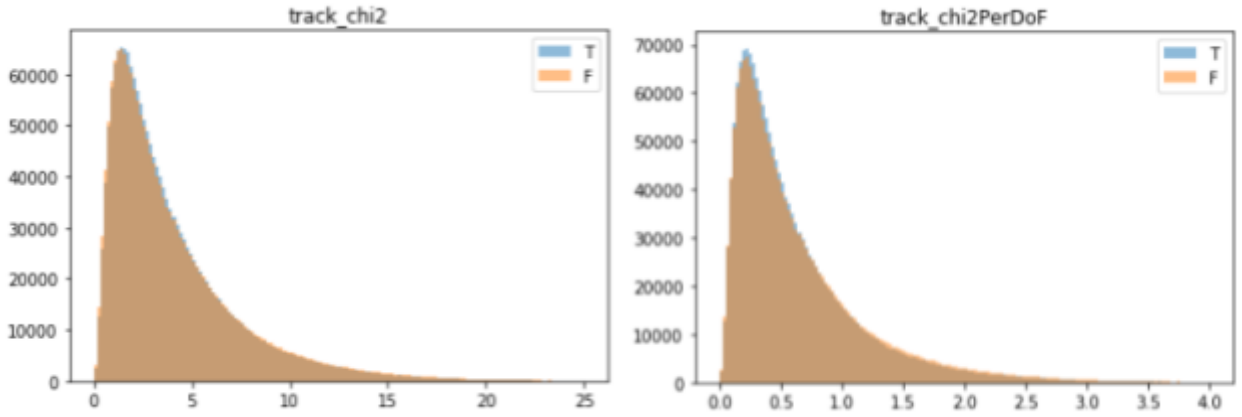
Następnie z klasy zawierającej większą ilość rekordów odrzucono losowe wartości tak aby zrównać obie klasy co pozwala utworzyć histogramy wyodrębnionych klas, macierze korelacji oraz wykresy rozproszenia (z ang. scatter plot). Wizualizacje te pozwalają na wytypowanie ostatecznych cech które zostaną wykorzystane podczas uczenia modelu.

4.1.3. Analiza histogramów

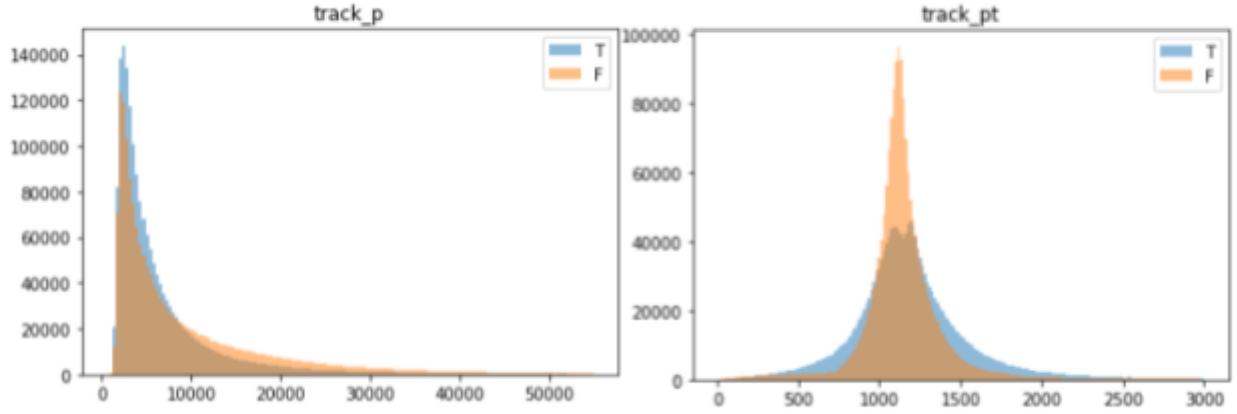
Skorelowane cechy mają negatywny wpływ na uczenie się naszego modelu oraz wydłużają czas treningu nie poprawiając jego wyników. Może to prowadzić do zmniejszenia czułości modelu na inne cechy zbioru danych.

Dzięki zestawieniu ze sobą różnych klas w histogramach łatwo można zobaczyć które z cech znaczącą rozróżniają nasz zbiór. Jeżeli histogram jednej cechy praktycznie się nie różnią nie oznacza to od razu że ta klasa powinna zostać usunięta, ale pozwala to wybrać kandydata w przypadku głębszych korelacji:

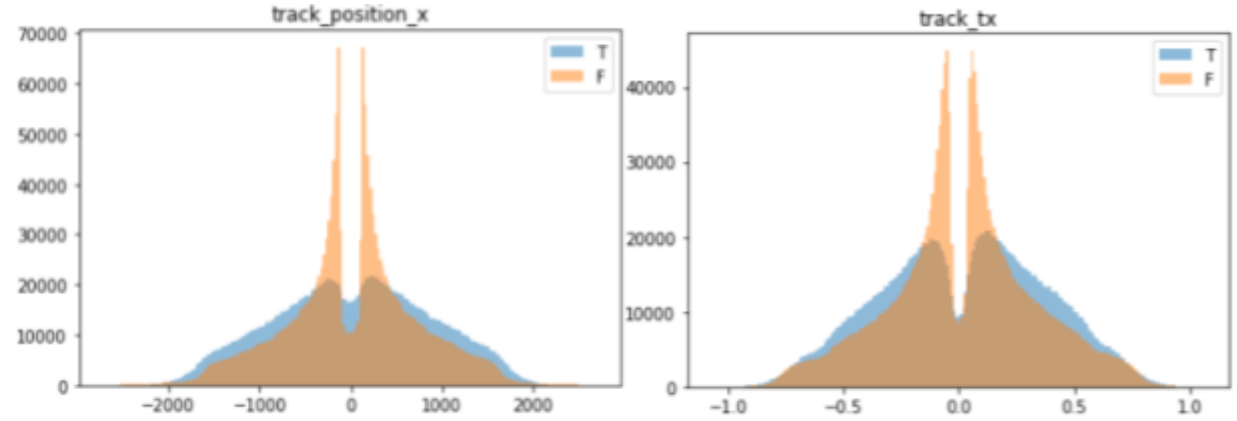
Z dostarczonych danych uzyskano następujące histogramy:



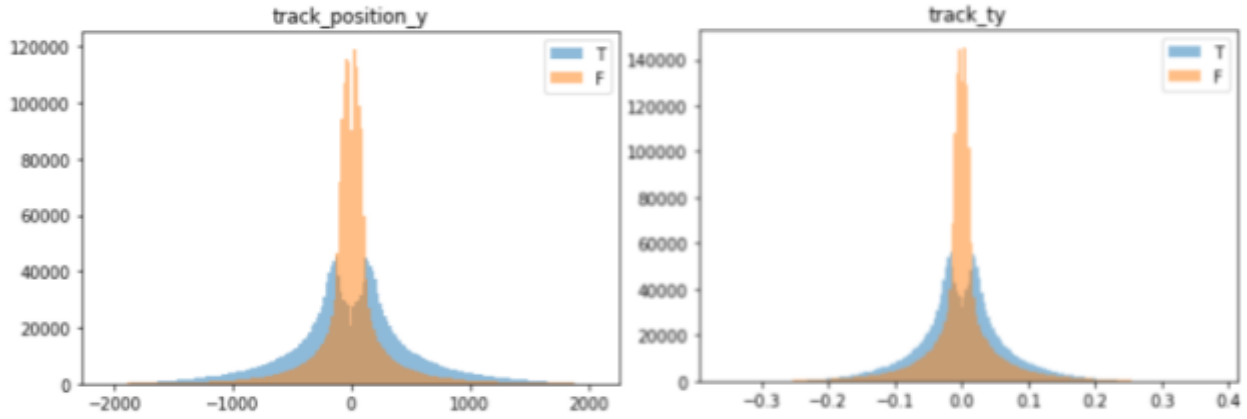
Rysunek 4.1: histogramy klas T i F dla cech chi2 oraz chi2PerDoF



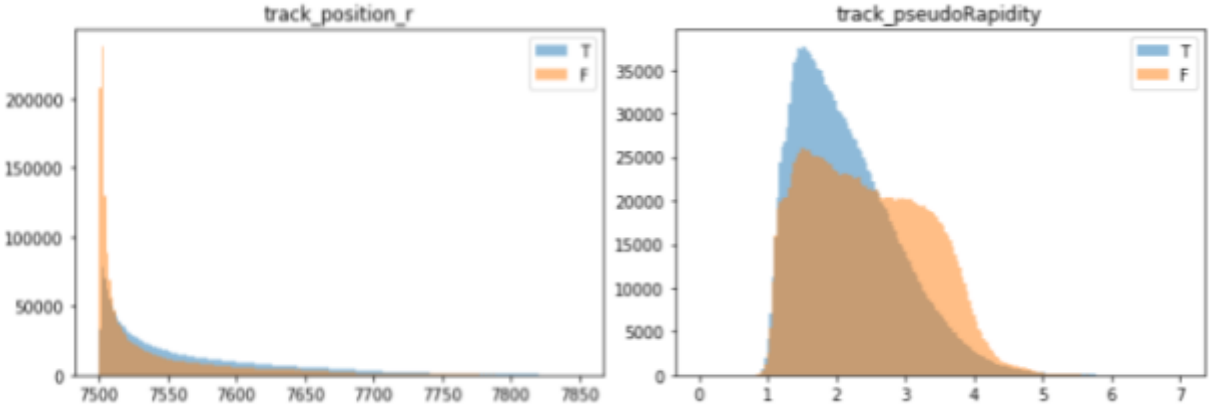
Rysunek 4.2: histogramy klas T i F dla cech p oraz pt



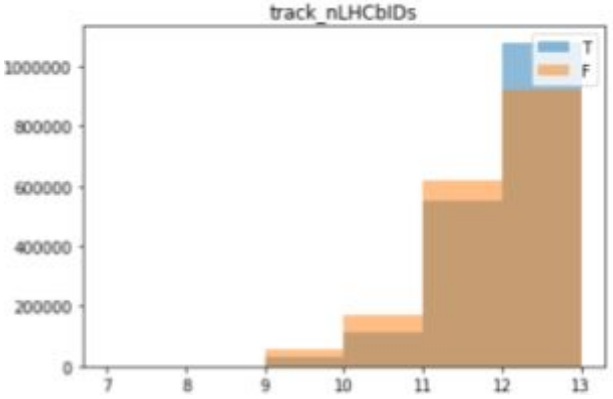
Rysunek 4.3: histogramy klas T i F dla cech position_x oraz tx



Rysunek 4.4: histogramy klas T i F dla cech position_y oraz ty



Rysunek 4.5: histogramy klas T i F dla cech position_r oraz pseudoRapidity

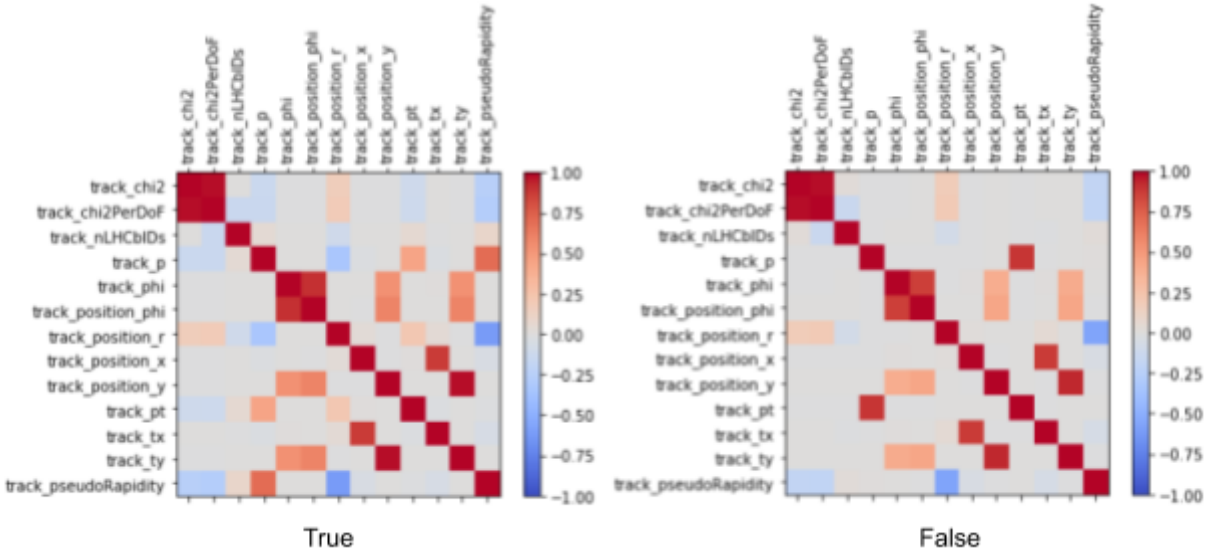


Rysunek 4.6: histogramy klas T i F dla cechy nLHCbIDs

Można zauważyć na rysunku 4.1 że dla chi2 oraz chi2PerDoF histogramy wartości True i False praktycznie się pokrywają.

4.1.4. Macierze Korelacji

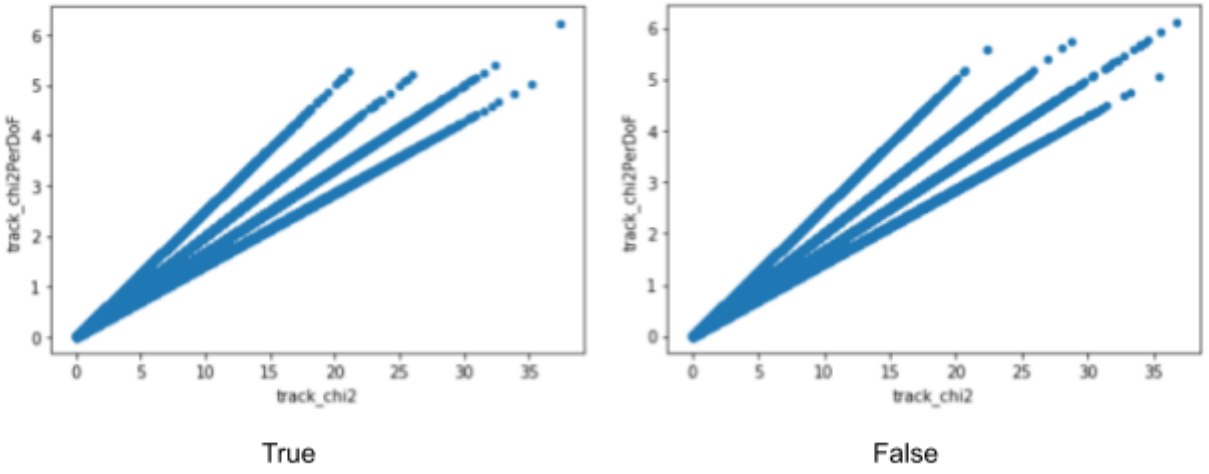
Następnie stworzone zostały macierze korelacji dla obu klas, przedstawione na rysunku 4.7, na których widać silną korelację między cechami chi2 oraz chi2PerDoF.
Jest tak dlatego ponieważ wartości chi2PerDoF są wartościami chi2 podzielonymi przez stopnie swobody, dlatego pierwszym kandydatem do odrzucenia jest chi2.



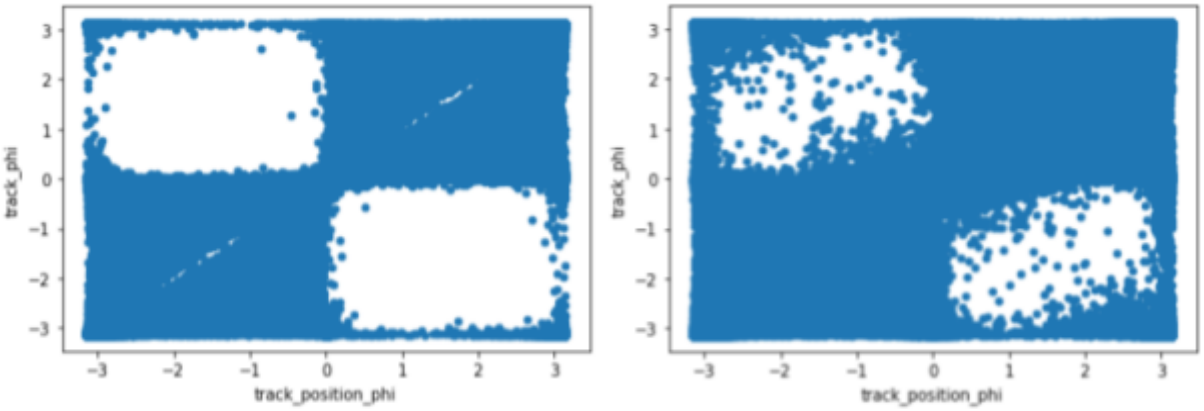
Rysunek: 4.7: macierze korelacji dla obu klas.

4.1.5. Wykresy rozproszenia

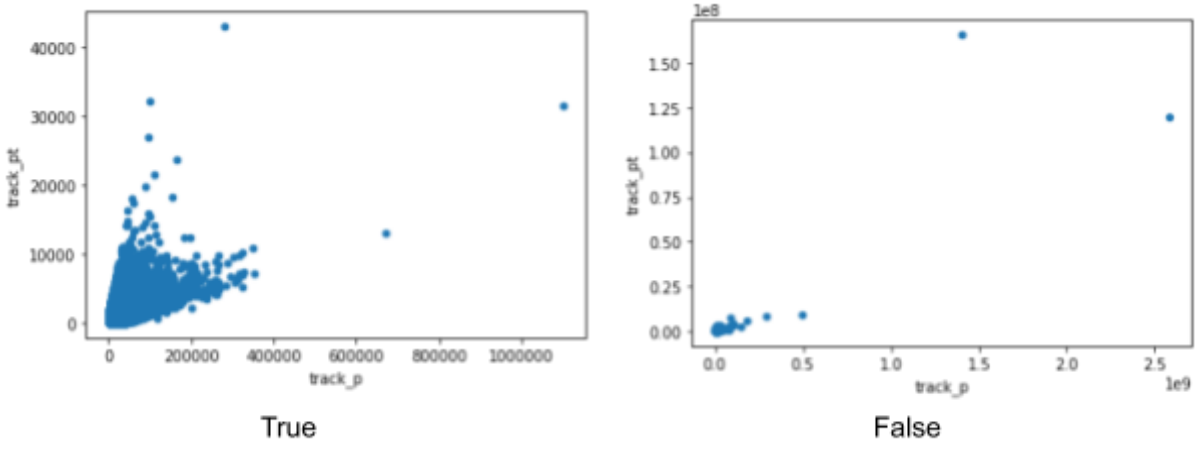
Dla pozostałych klas dla których zauważyć można mniejsze lub większe korelacje wyznaczono wykresy rozproszenia:



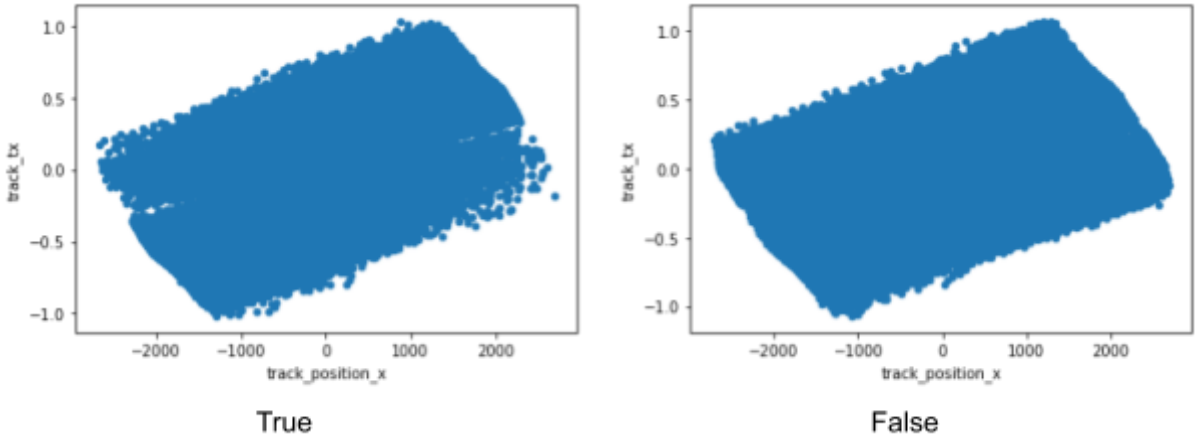
Rysunek 4.8: Wykresy rozproszenia klas T i F dla cech chi2 oraz chi2PerDoF



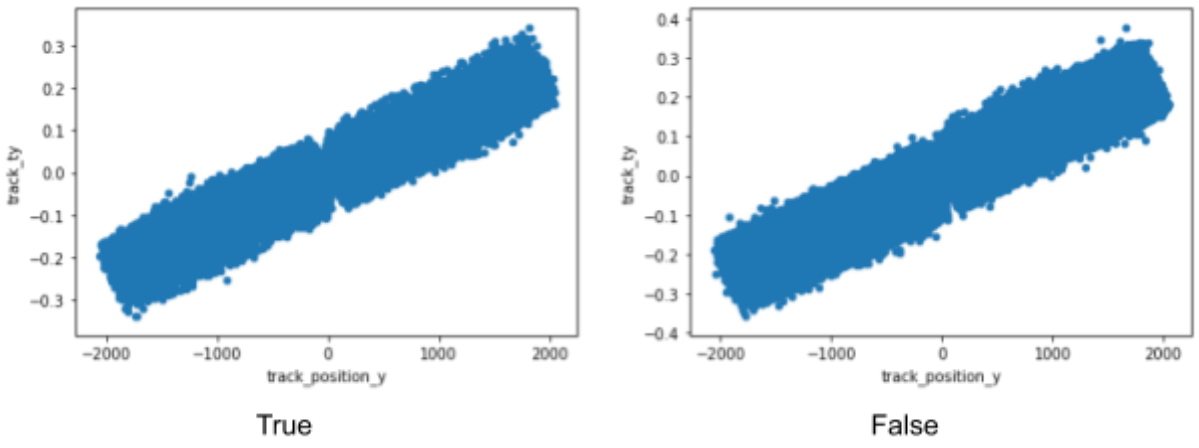
Rysunek 4.9: Wykresy rozproszenia klas T i F dla cech phi oraz position_phi



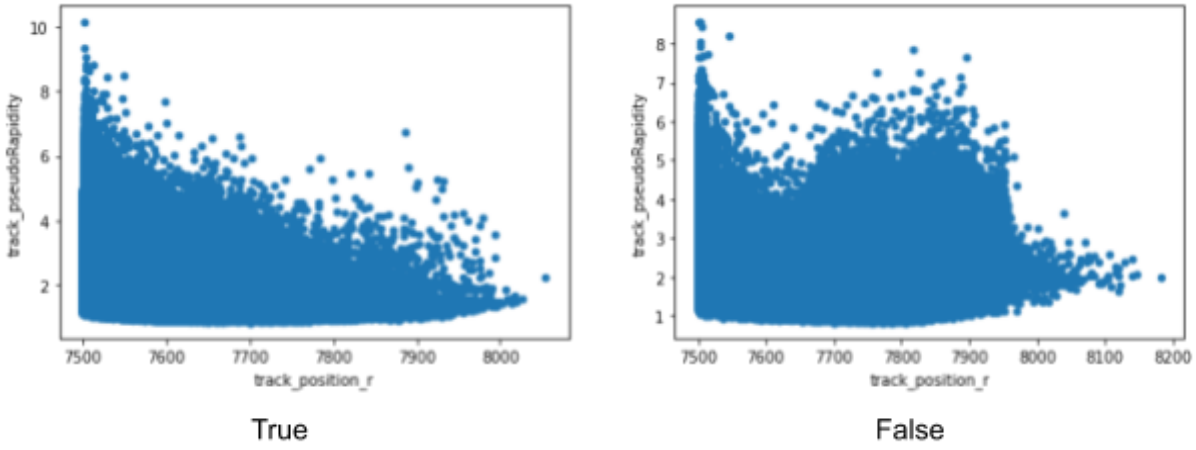
Rysunek 4.10: Wykresy rozproszenia klas T i F dla cech p oraz pt [6]



Rysunek 4.11: Wykresy rozproszenia klas T i F dla cech position_x oraz tx [6]



Rysunek 4.12: Wykresy rozproszenia klas T i F dla cech position_y oraz ty [6]



Rysunek 4.13: Wykresy rozproszenia klas T i F dla cech pseudoRapidity oraz position_r [6]

Na podstawie powyższych wyników z cech przeznaczonych do treningu odrzucona zosta cecha chi2.

4.2. Przetwarzanie wstępne i podział danych:

W zależności od modelu wprowadzane dane powinny zostać wstępnie przetworzone. W przypadku sieci neuronowej powinno się przeprowadzić standaryzację danych polegającą na przekształceniu każdej wartości zgodnie ze wzorem (9) znacznie poprawia to działanie modelu zmniejszając znaczenie przypadków krańcowych oraz eliminuje problemy wynikające z rozbieżnościami między cechami wynikającymi z natury problemu. Przykładem problemem mogłaby być rozbieżność między np. cenami obiektu a jego wagą. Waga wahałaby się od 0 do 1 kg a cena od 100 zł do 1000zł. Spowodowałoby to całkowite przykrycie znaczenia wagi przez cenę i nauczanie modelu aby stał się na nią wrażliwy byłoby bardzo czasochłonne.

$$z = \frac{x-u}{s} \tag{9}$$

gdzie u to średnia wszystkich wartości a s to odchylenie standardowe.

W tym projekcie do tego celu wykorzystano metodę udostępnioną przez bibliotekę sklearn *sklearn.preprocessing.**StandardScaler***

```
scaler = StandardScaler()
scaler.fit(features)
X = scaler.transform(features)
```

rozwiązanie to wymaga jednak aby każde dane wprowadzone do modelu po treningu były poddane tej operacji, co za tym idzie jeżeli nauczony model miałby zostać umieszczony w innym projekcie razem z nim powinniśmy także przekazać obiekt **scaler** np. przy pomoc biblioteki **Pickle.**

Po ustandaryzowaniu danych zostały one podzielone na dane testowe oraz treningowe używając funkcji *sklearn.model_selection.**train_test_split***

```
X_train, X_test, y_train, y_test = train_test_split(X,
targets, test_size = 0.2)
```

4.3. Tworzenie modelu

Tworzenie modelu uczenia maszynowego opartego o sieci neuronowe wykorzystując bibliotekę PyTorch jesty rozległe. Biblioteka oferuje nam gotowe rozwiązania ale także pozwala w prosty sposób modyfikować lub tworzyć własne modele. W tej pracy został przedstawiony model regresji liniowej którego głównym atutem jest możliwość ustawienia “kształtu” i ilości neuronów. Pozwala on stworzyć proste modele kwadratowe, piramidy, klepsydry oraz odwróconej klepsydry.
Hiperparametrami: n_hidden (ilość ukrytych warst), first_hidden (ilość neuronów pierwszej ukrytej warstwy) center_hidden (ilość neuronów środkowej ukrytej warstwy) i last_hidden (ilość neuronów ostatniej warstwy) możemy z dużą dowolnością manipulować kształtem modelu.

Model został stworzony tworząc klasę dziedziczącą po ***torch.nn.Module***.
w konstruktorze klasy definiowane są poszczególne warstwy neuronów.

W przypadku tego projektu warstwy tworzone są przy pomocy poniższego kodu:

```
for i in range(int(n_hidden)+1):
            self.leyer_list.append(nn.Linear(layer_w, hidden_dim))
            if i < (n_hidden/2)-1:
                layer_w = hidden_dim
                hidden_dim =
(int)((i+1)*(2*(center_hidden-first_hidden)/n_hidden)+first_hidden)
            else:
                layer_w = hidden_dim
                hidden_dim =
(int)((i-n_hidden/2+1)*(2*(last_hidden-center_hidden)/n_hidden)+center
_hidden)
        self.leyer_list.append(nn.Linear(layer_w, 1))
```

ilości neuronów w kolejnych warstwach wyprowadzane są na podstawie dwóch funkcji liniowych, przechodzących przez kolejno **first_hidden** i **center_hidden** dla pierwszej połowy warstw oraz **center_hidden** i **last_hidden** dla drugiej połowy warstw.

utworzenie modelu z parametrami:

model = LogisticRegressionModel(input_dim=10, n_hiden=11, first_hiden=15, center_hiden=4, last_hiden=2)

utworzy model o postaci:

```
 (leyer_list): ModuleList(
   (0): Linear(in_features=10, out_features=15, bias=True)
   (1): Linear(in_features=15, out_features=13, bias=True)
   (2): Linear(in_features=13, out_features=11, bias=True)
   (3): Linear(in_features=11, out_features=9, bias=True)
   (4): Linear(in_features=9, out_features=7, bias=True)
   (5): Linear(in_features=7, out_features=5, bias=True)
   (6): Linear(in_features=5, out_features=3, bias=True)
   (7): Linear(in_features=3, out_features=3, bias=True)
   (8): Linear(in_features=3, out_features=3, bias=True)
   (9): Linear(in_features=3, out_features=2, bias=True)
   (10): Linear(in_features=2, out_features=2, bias=True)
   (11): Linear(in_features=2, out_features=2, bias=True)
   (12): Linear(in_features=2, out_features=1, bias=True)
 )
```

Następnie zdefiniowana jest metoda forward w której definiujemy przepływ danych przez model. To w tym miejscu definiujemy funkcje aktywacji dla każdej warstwy.

```
def forward(self, x):
      Relu = nn.LeakyReLU()
      Sigm = nn.Sigmoid()
      for layer in self.leyer_list[:-1]:
            x = Relu(layer(x))
      y = Sigm(self.leyer_list[-1](x))
      return y
```

4.4. Trenowanie modelu

Model trenowany jest przez pętle trenującą składającą się z 4 głównych elementów:
Propagacji, czyli obliczenie predykcji modelu: pred = model(batch_features)
Obliczenie funkcji straty: loss_val = loss_fun(pred, batch_targets)
Propagacja wsteczna: loss_val.backward()
Optymalizacja wag: optimizer.step()
Po tych krokach należy jeszcze wyzerować gradienty obliczone automatycznie podczas propagacji metodą: optimizer.zero_grad()

Poniżej przedstawiono funkcję trenującą użytą w projekcia:
Argument verbose odpowiada za wypisywane przez funkcję informacje. verbose = 0 spowoduje wypisanie metryk tylko po zakończeniu treningu, a dla wartości 1 przedstawi je co epokę.

```
def train(model, training_set, test_data, stat_dict, epochs, lr_rate, batch_size, verbose = 0):
  input_dim = X_train.shape[1]
  X_test, y_test = test_data
  loss_fun = torch.nn.BCELoss().to(device)
  optimizer = torch.optim.Adam(model.parameters(), lr_rate)
  model.to(device)
  n_epoch = len(stat_dict["train_acc"])
  scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=50, gamma=0.2)
  f = FloatProgress(min=n_epoch, max=epochs)
  display(f)
  for epoch in range(n_epoch, epochs):
    f.value = epoch
    training_generator = utils.data.DataLoader(training_set, batch_size = batch_size, shuffle
= True)
    train_acc, train_loss, train_precision, train_recall, _= acc(model, loss_fun, X_train,
```

```
y_train)
        test_acc, test_loss, test_precision, test_recall, elapsed_time = acc(model, loss_fun,
X_test, y_test)
        for batch_features, batch_targets in training_generator:
            batch_features, batch_targets = batch_features.to(device).view(-1,input_dim),
batch_targets.to(device).view(-1,1)
            def closure(batch_features, batch_targets):
                optimizer.zero_grad()
                pred = model(batch_features)
                loss_val = loss_fun(pred, batch_targets)
                loss_val.backward()
            optimizer.step(closure(batch_features, batch_targets))
        scheduler.step()
        stat_dict["test_loss"].append(test_loss)
        stat_dict["test_acc"].append(test_acc)
        stat_dict["test_precision"].append(test_precision)
        stat_dict["test_recall"].append(test_recall)
        stat_dict["train_loss"].append(train_loss)
        stat_dict["train_acc"].append(train_acc)
        stat_dict["train_precision"].append(train_precision)
        stat_dict["train_recall"].append(train_recall)
        stat_dict["elapsed_time"].append(elapsed_time)
        if verbose == 1:
            print("epoch {0}".format(epoch+1))
            print("train: loss {0:.4f}, acc = {1:.4f}, precision = {2:.4f}, recall =
{3:.4f}".format(train_loss, train_acc, train_precision, train_recall))
            print("test:  loss {0:.4f}, acc = {1:.4f}, precision = {2:.4f}, recall = {3:.4f}, time=
{4}".format( test_loss, test_acc, test_precision, test_recall, elapsed_time))
            if epoch!=0 and epoch%100==0:
                plot_training_classification(stat_dict)
                auc_val = plot_roc(X_test, y_test, model)
    f.value = epochs
    if verbose == 0:
        print("train: loss {0:.4f}, acc = {1:.4f}, precision= {2:.4f}, recall = {3:.4f}".format(train_loss,
train_acc, train_precision, train_recall))
        print("test:  loss {0:.4f}, acc = {1:.4f}, precision = {2:.4f}, recall = {3:.4f}".format( test_loss,
test_acc, test_precision, test_recall))
        plot_training_classification(stat_dict)
        auc_val = plot_roc(X_test, y_test, model)
```

PyTorch udostępnie narzędzie jakim jest ***lr_scheduler*** pozwalającą na zdefiniowanie jak tempo nauki ma zmieniać się w trakcie treningu. W projekcie użyty został ***lr_scheduler .step_size*** który umożliwia stworzenie dynamicznego hiperparametru zmieniającego po wykonaniu odpowiedniej ilości epok.
scheduler = StepLR(optimizer, step_size=50, gamma=0.2) sprawi że learning_rate będzie mnożono o wartość 0.2 co 50 epok. Należy pamiętać o wywołaniu metody ***scheduler.step()*** w każdej epoce.

4.5. Przeprowadzenie treningu

Analizując pokazaną wcześniej funkcję train można zauważyć że metrykami zwracanymi przez funkcje do słownika ***stat_dict*** są: loss - wynik funkcji błędu, acc - dokładność, precision - precyzja i recall - czułość dla danych testowych oraz dla danych treningowych, a także czas jaki zajęło obliczenie predykcji dla danych testowych w sekundach.

Dla ułatwienia przygotowano także funkcję ***fit_with*** przygotowującą model, słownik metryk oraz dane.
Argument funkcji return_model dla wartości True zwróci model razem ze słownikiem metryk w przypadku false zwraca wynik modelu, wykorzystany do znalezienia optymalnych hiperparametrów.

```
def fit_with(epochs, n_hiden, first_hiden, center_hiden, last_hiden,
             lr, batch_size, return_model=True, verbose = 0):
    params = [int(n_hiden), int(first_hiden),
              int(center_hiden), int(last_hiden)]
    input_dim = X_train.shape[1]
    output_dim = 1
    model = LogisticRegressionModel(input_dim, *params)
    stat_dict = {
        "train_loss" : [],
        "test_loss" : [],
        "train_acc" : [],
        "test_acc" : [],
        "train_precision" : [],
        "test_precision" : [],
        "train_recal" : [],
        "test_recal" : [],
        "elapsed_time": []
    }
    training_set = MyDataset(X_train, y_train)
```

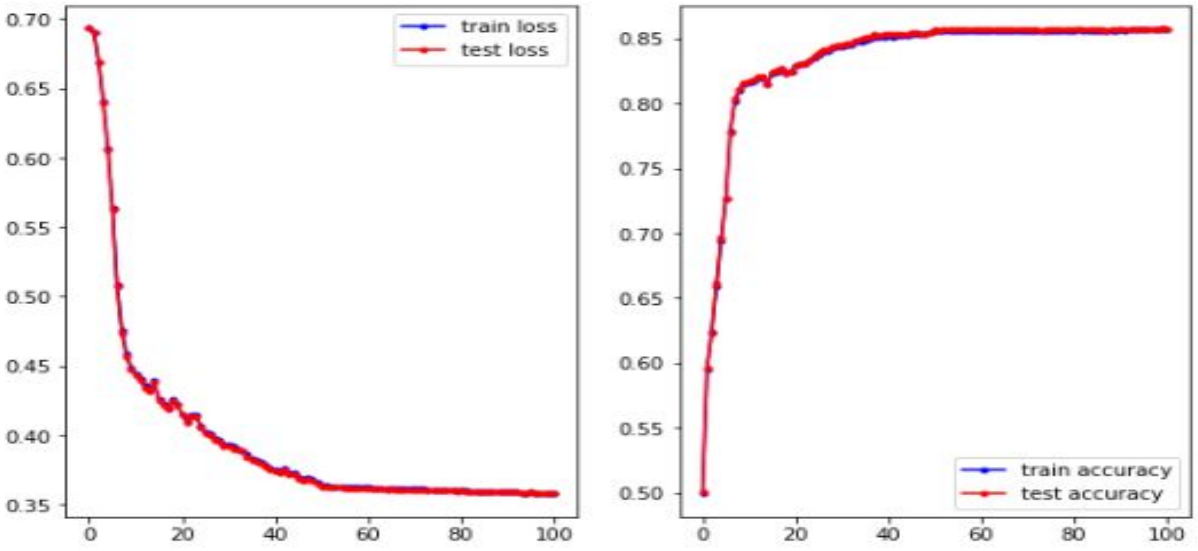
```
    test_data = (X_test, y_test)
    train(model, training_set, test_data, stat_dict,
          epochs, lr, int(batch_size), verbose)
    elapsed_time = mean(stat_dict["elapsed_time"][-5:])
    if verbose == 0:
        print("time: {}, batch_size: {}".format(
            elapsed_time, batch_size))
    score = Score(stat_dict["test_loss"][-1].item(),
                  stat_dict["test_acc"][-1].item(),
                  stat_dict["test_recal"][-1].item(),
                  elapsed_time)
    if return_model:
        return model, stat_dict
    else:
        return score
```

Wykonanie metody z poniższymi parametrami:

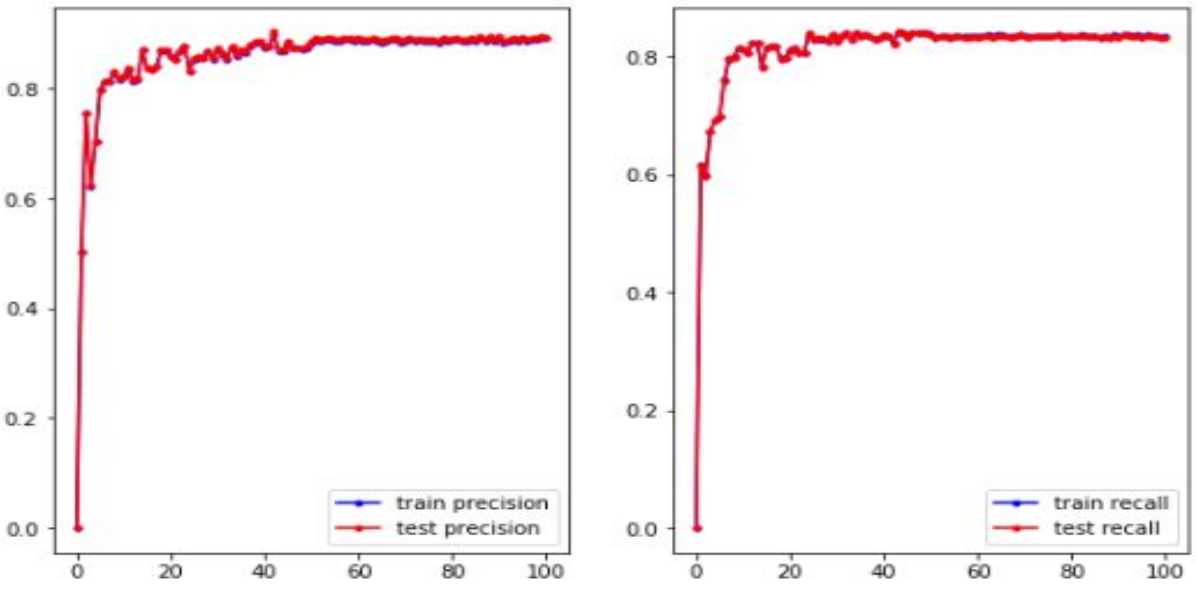
fit_with(epochs=200, n_hiden=6, first_hiden=12, center_hiden=7, last_hiden=3, lr=0.01, batch_size=30000, return_model=True, verbose=1)

Stworzy model zbudowany z 42 ukrytych neuronów cechujący się następującymi wynikami:

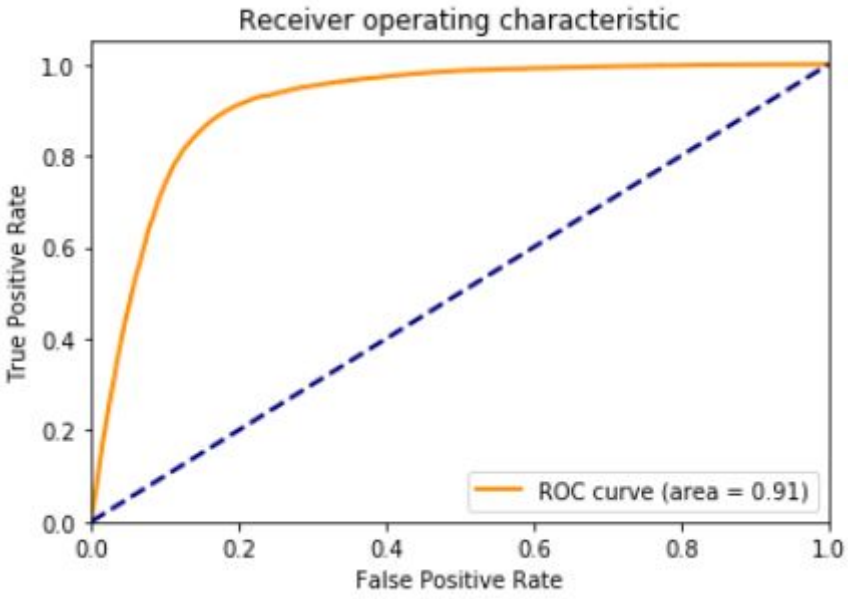
train: loss 0.3761, acc = 0.8511, precision= 0.8832, recall = 0.8300
test: loss 0.3762, acc = 0.8502, precision = 0.8827, recall = 0.8285
time= 0.001512765884399414



Rysunek 4.14: wykres funkcji błędu i precyzji



Rysunek 4.15: wykres funkcji błędu i precyzji



Rysunek 4.16: wykres krzywej ROC

Wybierając hiperparametry ręcznie udało się uzyskać całkiem dobrą celność oraz dokładność.

4.6. Optymalizacja hiperparametrów

W projekcie do optymalizacji hiperparametrów wykorzystano metodę Bayesowska opisaną w punkcie 2.3.4.

Do jej implementacji wykożystano bibliotekę bayesian-optimization:

```
from bayes_opt import BayesianOptimization

optimizer = BayesianOptimization(
    f=fit_with_partial,
    pbounds=pbounds,
    verbose=2,
    random_state=1,
)

optimizer.maximize(init_points=15, n_iter=50)
```

gdzie ***pbounds*** to słownik zawierający granice poszczególnych hiperparametrów:

```
pbounds = {'n_hiden': (4, 15), "first_hiden": (5, 15),
        "center_hiden": (5, 15), "last_hiden": (5, 15),
        'lr': (1e-3, 1e-2), "batch_size": (5e+3, 5e+4)}
```

parametr ***f*** to funkcja której ma zostać zoptymalizowana.
W przypadku tej pracy była to funkcja ***fit_with*** opakowana funkcją ***fit_with_partial = functools.partial(fit_with, epochs)***
Pozwala to na ustalenie hiperparametrów przez użytkownika użytych w każdej iteracji.

Jako wynik którego maximum jest szukane przez funkcje optymalizacyjna wybrane zostało równanie (15), bierze ono pod uwagę wydajność modelu a nie tylko jego skuteczność.

$$score = ACC - model\_cost \tag{15}$$

W pierwszych próbach jako model_cost autor pracy próbował wykożystać czas w jakim dane tostowe zostają przetworzone jednak model jest na tyle mały że jego wielkość nie miała znacznego wpływu na szybkość przetwarzania danych dla tak małych próbek.

Wykorzystana została więc ilość ukrytych neuronów modelu a koszt obliczony wzorem (16), gdzie neuron_max oznacza maksymalną liczbę neuronów dla przeprowadzanej optymalizacji a cost_ratio jest współczynnikiem regulującym wpływu kosztu na ostateczny wynik.

$$model\_cost = \frac{number\_of\_neurons}{neuron\_max} \cdot cost\_ratio \tag{16}$$

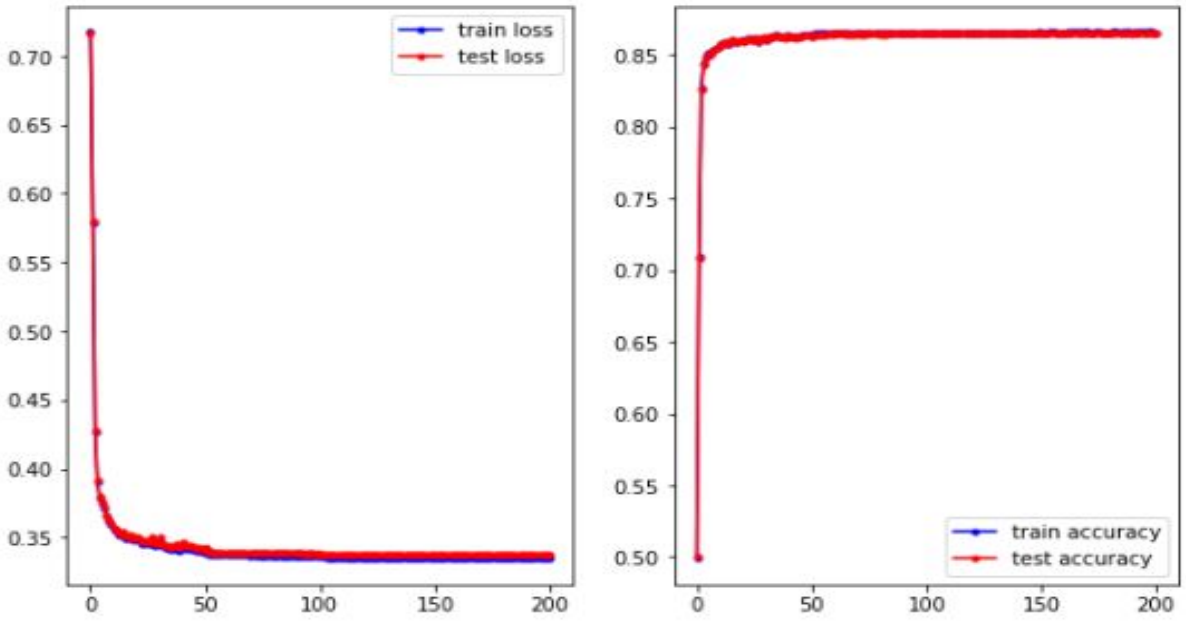
4.6.1. Wyniki optymalizacji Bayesowskiej

Po 65 iteracjach najlepszym uzyskanym rezultatem był:

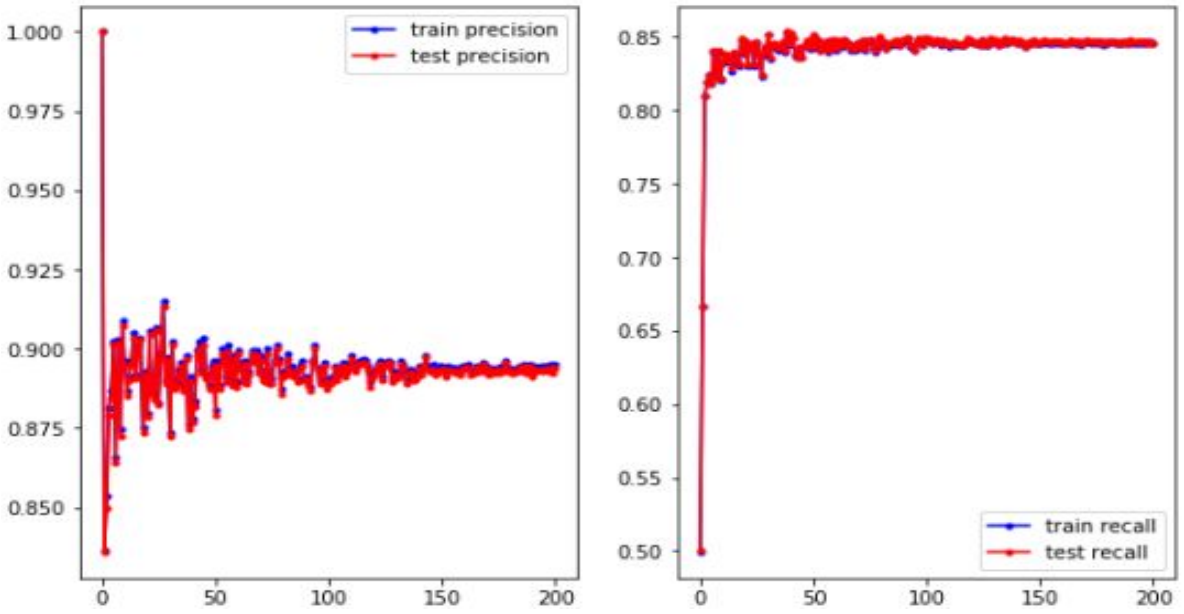
target: 0.8361 dla modelu zbudowanego z 20 ukrytych neuronów.

train: loss 0.3349, acc = 0.8658, precision = 0.8951, recall = 0.8453
test: loss 0.3369, acc = 0.8655, precision = 0.8943, recall = 0.8459
time= 0.00079[s]

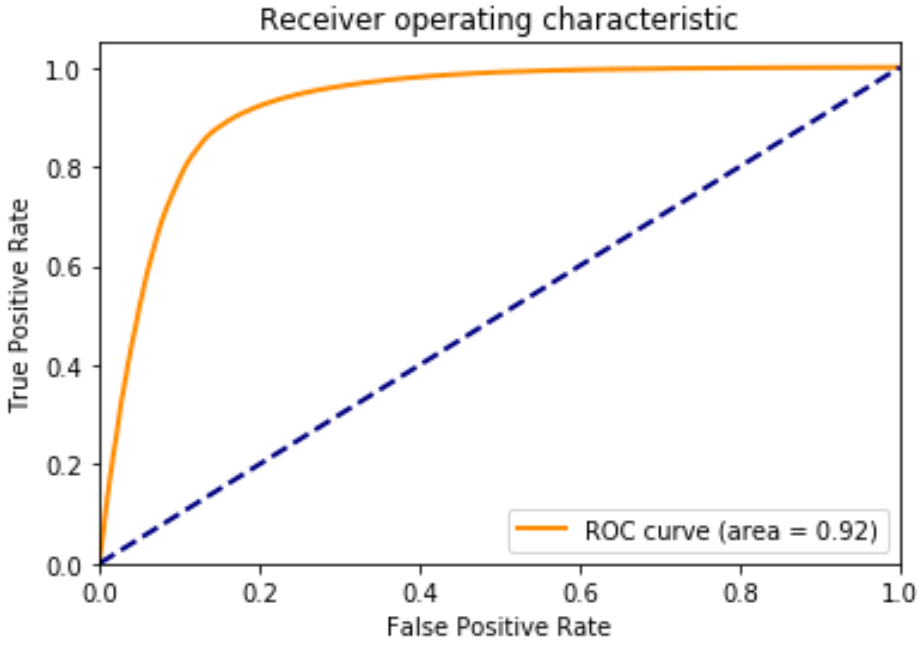
dla parametrów:
n_hiden: 4, center_hiden: 14, first_hiden: 13, last_hiden: 7
lr: 0.004, batch_size: 6704



Rysunek 4.17: wykres funkcji błędu i precyzji dla modelu o zoptymalizowanych hiperparametrach

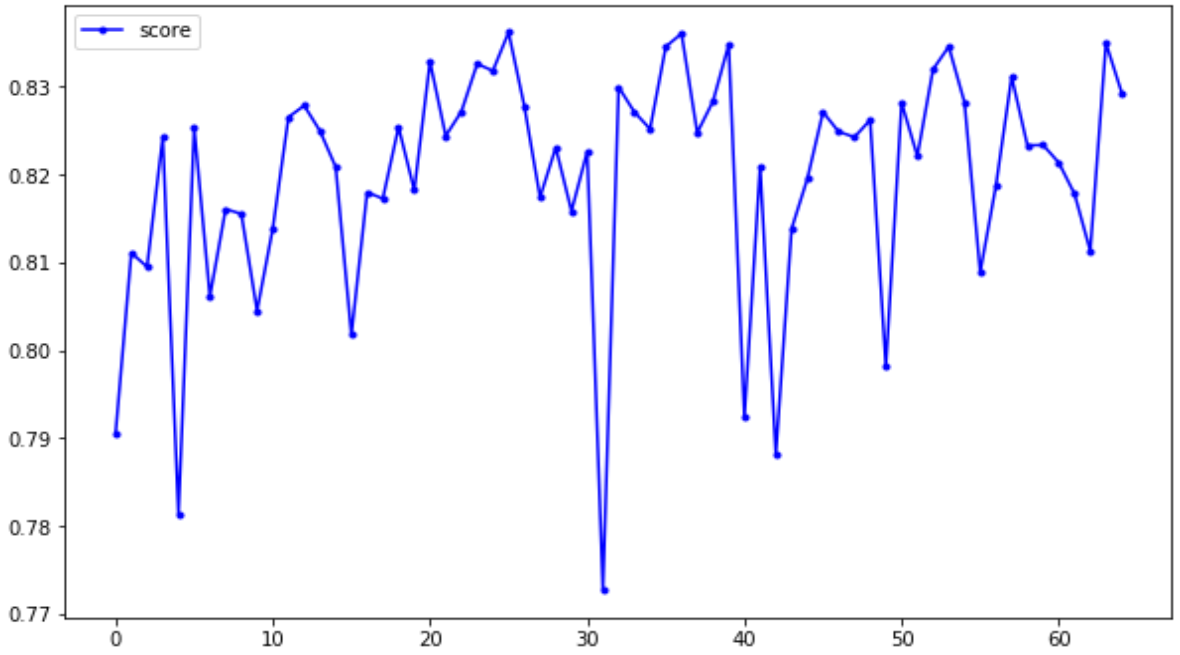


Rysunek 4.18: wykres funkcji błędu i precyzji dla modelu o zoptymalizowanych hiperparametrach



Rysunek 4.19: wykres krzywej ROC

Po przeprowadzeniu optymalizacji celność modelu podniosła się o 1.5% a ilość ukrytych neuronów została zredukowana z 42 do 20.



Rysunek 4.17: wykres przedstawiający wynik modelu dla każdej iteracji [7]

5.Podsumowanie:

Celem Projektu było stworzenie prostego modelu uczenia maszynowego opartego o sztuczne sieci neuronowe do rozwiązania problemu klasyfikacji danych uzyskanych w eksperymencie LHCb. Model stworzony został wykorzystując bibliotekę Pytorch dla języka programowania Python.

Zaprezentowane narzędzia bardzo ułatwiają tworzenie prostych i wydajnych modeli uczenia maszynowego, ale mogą być także użyte do znacznie bardziej złożonych projektów Model pokazany w punkcie 4.3 można bardzo łatwo przebudować na model klasyfikacji obrazów dodając do niego warstwy konwolucyjne lub dodać inne warstwy poprawiające skuteczność takie jak np. warstwy dropout czy warstwy normalizacyjne.

Optymalizacja bayesowska pokazana w punkcie 4.6. może zostać wykorzystana do innych problemów optymalizacji niekoniecznie optymalizowania hiperparametrów modelu.

Przedstawiony w pracy model nie osiągną jednak satysfakcjonującego wyniku jakim mogłaby być celność na poziomie 90%, jednak może wynikać to z bardzo pokrywających się ze sobą klas które należało rozpoznać co widać na histogramach przedstawionych w punkcie 4.1.3.

Wyniki pracy umieszczone zostały w serwisie Github pod linkiem:
https://github.com/Barnabasz/RECONSTRUCTION

6. Bibliografia:

[1] Eksperyment LHCb - https://home.cern/science/experiments/lhcb
[2] Opis cząstek długożyciowych - https://cernbox.cern.ch/index.php/s/8F9Uc26f7aBMMTS
[3] Biblioteka PyTorch - https://pytorch.org/
[4] Perceptron - https://pl.wikipedia.org/wiki/Neuron_McCullocha-Pittsa
[5] Proces uczenia sieci neuronowej - http://neuralnetworksanddeeplearning.com/chap2.html
[6] Funkcje aktywacji - https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity
[7] Metryki - https://brain.fuw.edu.pl/edu/index.php/Uczenie_maszynowe_i_sztuczne_sieci_neuronowe/Wyk%C5%82ad_Ocena_jako%C5%9Bci_klasyfikacji
[8] Optymalizacja Bayesianowska - http://cds.cern.ch/record/2702355/files/1911.02501.pdf