Przygotowanie narzędzi do uczenia maszynowego opartego na danych z eksperymentu CMS.

Paweł Czajka Mateusz Fila Rafał Masełek

Opiekun projektu: dr hab. A. Kalinowski

Cel projektu

Celem projektu było przygotowanie narzędzi do efektywnego wykorzystania metod uczenia maszynowego (ang. *Machine Learning* – ML) do analiz fizycznych w eksperymencie Compact Muon Solenoid (CMS) przy Wielkim Zderzaczu Hadronów (LHC) w ośrodku badawczym CERN pod Genewą. Bazowym założeniem było wykorzystanie istniejącej analizy dedykowanej rozpadom bozonu Higgsa na parę taonów ($H \rightarrow \tau\tau$), wyekstraktowanie z niej interesujących obserwabli do plików w formacie czytelnym z poziomu języka programowania Python, a następnie wykorzystanie zaawansowanych bibliotek dedykowanych do uczenia maszynowego w celu poprawienia wydajności analizy.

1 Wprowadzenie

1.1 Badanie własności bozonu Higgsa

Najważniejszym celem do którego mogą zostać zastosowane narzędzia powstałe w naszym projekcie jest detekcja rozpadów bozonu higgsa na parę taonów $H \rightarrow \tau\tau$. Taony przed detekcją rozpadają się na inne cząstki. Rozpad taki może zajść na kilka sposobów. Tauon może na przykład rozpaść się na hadrony oznaczane τ_h oraz neutrino. Jedną z innych możliwości rozpadu tauonu jest powstanie mionu μ oraz neutrin. Szczególnie interesują nas przypadki typu $H \rightarrow \tau_h\mu$, to znaczy takie w których bozon Higgsa rozpadł się na dwa tauony, z których jeden przeszedł hadronowy rozpad, natomiast z drugiego powstał mion (neutrina nie są mierzone więc są pomijane w zapisie).

W przypadku rozpadów $H \rightarrow \tau\tau$ istnieją metody analizy danych zarejestrowanych przez CMS mające na celu częściowe oddzielenie tych rozpadów od tła podobnie wyglądających rozpadów [1]. Polega ona na podziale zdarzeń na kilka kategorii (rozłącznych). Następnie w każdej kategorii wybierane są 2 zmienne które najlepiej nadają się do odróżniania zdarzeń $H \rightarrow \tau\tau$ od tła. Dane na podstawie których skonstruowano ten model pochodzą z symulacji rozpadów $H \rightarrow \tau\tau$ oraz różnych procesów tła.

Chcielibyśmy mieć analogiczny model odróżniający rozpady $H \rightarrow \tau_h\mu$ od procesów tła (rozpadów produkujących podobne cząstki). Chcielibyśmy jednak użyć uczenia maszynowego (głębokich sieci neuronowych) by uzyskać prosty model w którym nie ma już dzielenia zdarzeń na klasy. Taki model powinien odróżniać rozpady $H \rightarrow \tau_h\mu$ od tła na podstawie pewnego zbioru zmiennych mierzonych w CMS (lub odtwarzanych przy pomocy wyspecjalizowanych algorytmów na podstawie zmiennych mierzonych w CMS). Wśród tych zmiennych powinny znajdować się zmienne które okazały się istotne w analizie przedstawionej w artykule [1]. Do tego należy dołączyć inne standardowe zmienne które mogą okazać się istotne.

Zdecydowaliśmy się na ekstrakcję między innymi następujących zmiennych:

- **jets** - czteropędy dżetów hadronowych zaobserwowanych w zderzeniu.
- **legs** - czteropędy mionu (let[0]) oraz τ_h (leg[1]).
- **visMass** - masa niezmiennicza obliczona dla układu τ_h i mionu μ.
- **decayMode** - kanał rozpadu τ_h (z jedną lub trzema cząstkami naładowanymi).

Jak widać wśród zmiennych są zarówno zmienne ciągłe jak (np. visMass) jak i zmienne kategoryczne (np. leg_2_decaymode)

1.2 Elementy uczenia maszynowego

Uczenie maszynowe to zdolność komputerów do uczenia się bez programowania nowych umiejętności wprost. Warto przytoczyć następujące definicje związane z uczeniem maszynowym:

- **dane uczące** (ang. learning dataset) jest to zbiór danych używanych do uczenia. Jest to zbiór instancji.

- **instancja** (ang. example) w przypadku danych z eksperymentu CMS instancją są dane zebrane przez detektor w wyniku jednego zderzenia (oraz wielkości policzone na podstawie tych danych przy pomocy wyspecjalizowanych algorytmów) uzupełnione o informacje o tym do jakiego procesu doszło. Ogólnie jest to opis pojedynczego obiektu w danych uczących. Instancja składa się z atrybutów oraz etykiety (tej pracy zajmujemy się wyłącznie etykietowanymi danymi uczącymi).

- **etykieta** (ang. label) jest to cecha instancji którą chcemy nauczyć się przewidywać na podstawie pozostałych cech instancji.

- **atrybut** (ang. feature) jest to cecha instancji na podstawie której chcemy przewidzieć etykietę. Atrybuty dzielimy na numeryczne oraz kategoryczne.

- **atrybut numeryczny** jest to cecha instancji która jest ciągła (to znaczy jest charakteryzowana wektorem liczb rzeczywistych). W przypadku eksperymentu CMS takim atrybutem może być na przykład visMass.

- **atrybut kategoryczny** jest to cecha instancji która jest charakteryzowana przez liczbę całkowitą z pewnego skończonego zakresu. W przypadku eksperymentu CMS może to być na przykład leg_2_decayMode. Atrybut kategoryczny można zamienić na atrybut numeryczny przy pomocy procedury kod 1 z n.

- **kod 1 z n** (ang. one hot encoding) jest to zmiana atrybutu kategorycznego na numeryczny w taki sposób, że jeśli mamy atrybut kategoryczny k (ze zbioru wartości $1 \dots n$) to zamieniamy go na atrybut numeryczny w postaci wektora $v^i = \delta^i_k$ gdzie δ to delta Kroneckera.

- **problem klasyfikacji binarnej** (ang. binary classification) jest to problem w którym etykiety mogą przyjmować tylko 2 wartości (standardowo 0 lub 1). Naszym celem jest przewidzieć wartość tej etykiety

na podstawie atrybutów. W przypadku eksperymentu CMS naszym celem jest rozpoznanie czy uzyskane cząstki pochodziły z rozpadu bozonu Higgsa.

- **problem regresji** (ang. regression) jest to problem w którym etykieta to liczba rzeczywista.

- **model** (ang. model) jest to (dla naszych potrzeb) program będący rozwiązaniem problemu klasyfikacji lub regresji. Jest wiele rodzajów modeli na przykład drzewa decyzyjne oraz algorytmy bayesowskie. W naszym programie skupimy się na głębokich sieciach neuronowych. Gdy chcemy utworzyć model wybieramy na początek jego architekturę (np w przypadku głębokich sieci neuronowych liczbę oraz rozmieszczenie neuronów), a następnie w procesie uczenia zostają dobrane pozostałe parametry modelu zwane wagami (można powiedzieć, że model to architektura plus wagi modelu).

- **wagi modelu** (ang. model weights) wszystkie parametry modelu.

- **głęboka sieć neuronowa** (ang. deep neural networks) jest to model w którym na wejściu znajdują się atrybuty numeryczne (w postaci wektora liczb rzeczywistych), te atrybuty numeryczne są przekształcane aplikowanymi na zmianę przekształceniami liniowymi oraz nieliniowymi (polegającymi na aplikacji do każdego elementu wektora funkcji aktywacji). Na końcu uzyskuje się pożądany wynik.

- **uczenie** (ang. training) jest to proces ustalania wag modelu. W tym procesie używane są dane uczące.

- **sprawdzenie** (ang. validation) jest to proces sprawdzenia czy model dobrze rozwiązuje problem klasyfikacji binarnej lub regresji. Do tego testu należy użyć danych których model nie używał do uczenia.

- **uczące atrybuty** jest to pojęcie które wprowadzamy na potrzeby tej pracy. Są to atrybuty wykorzystywane do uczenia.

- **ignorowany atrybut** jest pojęcie które wprowadzamy na potrzeby tej pracy. Są to atrybuty które nie są wykorzystywane podczas uczenia modelu, ale są przechowywane razem z atrybutami uczącymi by móc później przeprowadzić analizę (na przykład korelacji tych ignorowanych atrybutów z przewidywaniami modelu).

2 Upublicznienie kodu projektu

Dla projektu stworzono dedykowane repozytorium na serwisie GitHub. Znajduje się ono pod adresem: `https://github.com/Rav2/HEPMachineLearning/tree/v1.0`. Początkowa część projektu polegała na modyfikacji analizy do rozpadów $H \rightarrow \tau\tau$, dostępnej pod adresem `https://github.com/akalinow/RootAnalysis`.

3 Modyfikacja istniejącej analizy

Pierwszym krokiem w budowie zintegrowanego narzędzia do analiz fizycznych z użyciem ML była modyfikacja istniejącej analizy dra Kalinowskiego. Analiza została napisana w języku C++ z użyciem zorientowanego obiektowo środowiska programistycznego ROOT[2]. Wykorzystanie obiektowego formatu danych ROOT w bibliotekach języka Python dedykowanych uczeniu maszynowemu byłoby trudne i prawdopodobnie nieefektywne, dlatego zdecydowano o następującej architekturze projektu:

1. Umozliwienie użytkownikowi wybrania danych z analizy C++ przeznaczonych do dalszej obróbki ML.

2. Zapisanie danych z obiektowo zorientowanego środowiska ROOT do plików binarnych typu *.root*, rezygnacja z obiektowości na rzecz typów prostych.

3. Wczytanie pliku .root przez skrypt napisany w języku Python. Nadanie danym struktury wymaganej przez poszczególne narzędzia do uczenia maszynowego.

3.1 Instalacja

Pliki C++ stworzone w ramach projektu, zostały zintegrowane z analizą dra Kalinowskiego i są dostępne pod adresem `https://github.com/akalinow/RootAnalysis`. Instrukcja kompilacji i użycia znajduje się w pliku README.md w katalogu głównym repozytorium. Po skompilowaniu analizy nie jest potrzebna żadna dodatkowa instalacja modułu ML. Warto nadmienić, iż analiza wymaga oprogramowania takiego jak: Pythia8[3], ROOT[2], Python[4] 2.7.x oraz biblioteki Boost[5] dla języka C++.

3.2 Założenia

Przyjęto następujące wytyczne:

- Kod powinien być zgodny ze standardem C++11.
- Modyfikacja powinna mieć formę modułu, nie zakłócać ani nie spowalniać normalnego działania analizy. Powinna istnieć możliwość wyłączenia działania modułu.
- Kod powinien być na tyle elastyczny, aby mógł być potencjalnie użyty do różnych analiz. Powinna istnieć możliwość wyboru parametrów zapisywanych do pliku.
- Sterowanie parametrami programu powinno odbywać się bez potrzeby ponownej kompilacji kodu, powinno dać się w razie potrzeby zautomatyzować. Zdecydowano się na użycie zewnętrznego pliku tekstowego z ustawieniami.
- Kod powinien być udokumentowany i napisany zgodnie z tzw. zasadami dobrego programowania.
- Należy stworzyć skrypt czytający powstałe pliki rootowe do języka Python.

4 Przepływ danych

1. Działanie programu ropoczyna się od wczytania pliku konfiguracyjnego, np. *HTauTau/Analysis/config/htt_MuTau.ini*. Znajdują się tam podstawowe parametry programu, takie jak: liczba wątków, folder na wyniki, ścieżka do plików wejściowych z danymi z eksperymentu CMS oraz ich nazwy.
2. Funkcja main() programu C++ (analizy $H \rightarrow \tau\tau$) znajduje się w pliku *HTauTau/Analysis/src/HTTAnalysis.cc*. Wykonywane są w niej kolejno:
   (a) Wczytanie pliku konfiguracyjnego (np. *htt_MuTau.ini*) oraz parsowanie parametrów.
   (b) Utworzenie użytecznych obiektów:
   - **std::vector<Analyzer*> myAnalyzers** – kontener na obiekty dziedziczące po klasie *Analyzer*, tj. obiekty analizy implementujące selekcję danych i obliczenia. W dalszej części dokumentu będziemy nazywać takie obiekty po prostu *analizami*.

- **EventProxyHTT *myEvent** – obiekt zawierający dane podlegające analizie.
- **ObjectMessenger* OMess** – obiekt służący do przekazywania wyników obliczeń pomiędzy różnymi analizami.

(c) Obiekty analiz są tworzone i umieszczane w kontenerze *myAnalyzers*. Jako ostatnia jest dodawana analiza przygotowująca dane do obróbki algorytmami ML[1]. Klasa za to odpowiedzialna nazywa się *MLAnalyzer* i dziedziczy po klasie *Analyzer*. Wymaga do pracy obiektu klasy *MLObjectMessenger* dziedziczącego po klasie *ObjectMessenger*.

(d) Następnie tworzony jest obiekt **TreeAnalyzer *tree**, który będzie sterował wykonywaniem analiz. Obiekt ma dostęp do kontenera z analizami, obiektu typu *ObjectMessenger* oraz pliku konfiguracyjnego.

(e) W końcowym etapie działania funkcji następuje wypisanie statystyk na stdout oraz zwolnienie pamięci.

3. W obiekcie klasy TreeAnalyzer tworzone są histogramy wspólne dla wszystkich analiz. W metodzie *int TreeAnalyzer::loop()* znajduje się pętla iterująca po wszystkich *zdarzeniach* (ang. event). Dla każdego zdarzenia wykonywane są analizy z kontenera *myAnalyzers* w kolejności w jakiej zostały tam dodane. Wykorzystywana jest do tego metoda *bool TreeAnalyzer::analyze(const EventProxyBase&)*.

4. ObjectMessenger jest klasą zaopatrzoną w kontenery do przechowywania danych w wybranych typach podstawowych. Dane są umieszczane i pobierane z kontenerów za pomocą odpowiednich metod. Ze względu na swoją elastyczność, funkcjonalność klasy ObjectMessenger może zostać poszerzona o nowe typy i dedykowane kontenery. Jedną z takich klas jest klasa MLObjectMessenger, która gromadzi wyniki pracy analiz aby dostarczyć je do analizy ML.

5. Analiza ML wykonuje następujące czynności:

   (a) W trakcie tworzenia obiektu wczytywany jest plik *HTauTau/Analysis/config/ml_Properties.ini*, w którym użytkownik może wybrać, które wielkości powinny znaleźć się w pliku wyjściowym.

[1]Głównym zadaniem analizy ML jest przygotowanie i zapisanie danych do pliku .root. Ponieważ na obecnym etapie rozwoju analizy wspomniany typ plików jest wykorzystywany jedynie w trybie pracy jednowątkowej, dlatego analiza ML nie zostanie utworzona, jeśli program zostanie uruchomiony w trybie wielowątkowym.

(b) Dane pobrane z obiektu MLObjectMessenger są przygotowywane do zapisu. W pliku wyjściowym tworzona jest wewnętrzna struktura (określona w pliku ml_Properties.ini).

(c) Czteropędy zrekonstruowanych cząstek i jetów są zapisywane do pliku wyjściowego.

(d) Pozostałe parametry są zapisywane.

5 Lista plików C++

Poniżej znajduje się lista plików C++ utworzonych bądź znacząco zmodyfikowanych w trakcie realizacji pierwszej części projektu:

- **MLAnalyzer.h** – plik nagłówkowy dla klasy MLAnalyzer dziedziczącej po klasie Analyzer. Zadaniem klasy jest wczytanie ustawień z pliku *ml_Properties.ini*, utworzenie struktury drzewa w pliku .root, zapisanie odpowiednich danych do tego pliku.

- **MLAnalyzer.cc** – plik źródłowy dla klasy MLAnalyzer.

- **ObjectMessenger.h** – plik nagłówkowy dla klasy *ObjectMessenger*. Obiekty tej klasy są kontenerami, w których przechowywane są dane przesyłane pomiędzy różnymi analizami. Klasa posiada interfejsy do modyfikacji i odczytu danych dla najczęściej używanych typów podstawowych.

- **ObjectMessenger.cc** – plik źródłowy dla klasy *ObjectMessenger*.

- **MLObjectMessenger.h** – plik nagłówkowy dla klasy *MLObjectMessenger* dziedziczącej po klasie *ObjectMessenger*. W stosunku do klasy bazowej jest wzbogacona o obsługę typów *HTTParticle* oraz *HTTAnalysis::sysEffects*.

- **ml_Properties.ini** – plik tekstowy umożliwiający wybór predefiniowanych parametrów, które zostaną zapisane do drzewa w pliku .root. Dane z pliku są wczytywane w metodzie *MLAnalyzer::ParseCfg(const std::string&)*

5.1 Konwersja danych z pliku ROOT do Pythona

Po wyekstraktowaniu danych z analizy do pliku .root należy wczytać je do części systemu napisanej w języku programowania Python. Służy do

tego skrypt umieszczony w głównym repozytorium projektu, w podfolderze *scripts*. Skrypt *read_ tree.py* przyjmuje dwa parametry: ścieżkę do pliku, który ma być wczytany oraz nazwę drzewa (domyślnie "Summary"). Wczytane dane są zapisywane do pythonowych list, które są zwracane przez funkcję *read_ tree()*. Można te listy później wykorzystać (decydując się jednocześnie na to, które dane są kategoryczne) implementując funkcję get_data 6.3.1.

6 Uczenie maszynowe w Python

Skorzystaliśmy z pakietów Tensorflow (dalej nazywany TF) [6], Numpy (dalej nazywany NP) [7] oraz json [8].

6.1 Założenia programów z tej części projektu

- Jako cel postawiono sobie osiągnięcie ogólności kodu. Powinien on działać zarówno dla problemów binarnej klasyfikacji jak i regresji. W obecnej wersji można użyć dowolnych atrybutów numerycznych oraz atrybutów kategorycznych (które zostaną zamienione przy pomocy kodu 1 z n automatycznie zamienione na atrybuty numeryczne).

- Użytkownik implementuje tylko jedną funkcję w języku Python, która łączy się z resztą systemu (patrz 6.3.1). Funkcja ta odpowiada za zebranie danych, ich podział na etykiety oraz atrybuty, podział atrybutów na kategoryczne oraz numeryczne oraz podział atrybutów na uczące i ignorowane.

- Podejście funkcyjne. Jedynymi efektami ubocznymi napisanych funkcji są powstawanie plików oraz zmiany wag modelu. Dzięki temu kod można łatwiej zrozumieć oraz modyfikować.

- Wszystkie ważne użyte funkcje mają dokumentację umieszczoną w komentarzach.

- Staraliśmy się zastosować do wytycznych dotyczących wydajności kodu ze strony `https://www.tensorflow.org/guide/performance/datasets`. Dzięki temu uczenie zachodzi szybko.

- W programie użyte są standardowe (opisane w dokumentacji Tensorflow) estymatory używające głębokich sieci neuronowych tf.estimator.DNNClassifier oraz tf.estimator.DNNClassifier. Dzięki temu

łatwo można zmodyfikować kod tak, by korzystać z innych estymatorów TF (być może własnoręcznie napisanych) gdyż wszystkie estymatory mają podobne funkcje train, evaluate oraz predict (patrz `https://www.tensorflow.org/guide/estimators`).

6.2 Lista plików Python

Powstały następujące pliki pythonowe, które można znaleźć w folderze machine_learning_python na repozytorium:
`https://github.com/Rav2/HEPMachineLearning/tree/v1.0`.
Znajduje się tam także plik typu jupyter notebook o nazwie szybkosc_test.ipynb, który prezentuje działanie systemu.

- **io_TFRecords** - moduł umożliwiający zapisywanie danych w postaci plików binarnych w tensorflowowym formacie TFRecords, który jest wspomniany w dokumentacji tensorflow. Jego opis znajduje się także na stronie:
  `https://www.tensorflow.org/tutorials/load_data/tf_records`.
  Ten binarny format danych pozwala modelom TF uczyć się na dużych zbiorach danych w szybki sposób. Obsługa tego formatu jest jednakże dość skomplikowana, gdyż trzeba pamiętać dokładnie jakiego typu dane były zapisane w pliku binarnym by można było go odczytać. Z tego powodu moduł ten tworzy foldery w których oprócz binarnych danych w formacie TFRecords znajdują się pliki z informacją o rodzajach zapisanych danych, które umożliwiają wygodne odczytanie danych. W plikach można zapisywać numeryczne atrybuty, które są listami floatów (ustalonej długości) oraz atrybuty kategoryczne w postaci intów (wówczas dostarczamy klasie także możliwe wartości naszych kategorycznych danych). Przewidywana etykieta może być zarówno binarną informacją typu tło-sygnał jak również może być przewidywaną liczbą rzeczywista. Dodatkowo wraz z danymi używanymi do trenowania można zapisać dane, które będą przez estymatory ignorowane. Dzięki temu można później sprawdzać korelacje tych nieużywanych atrybutów z uzyskanymi wynikami.

- **model_dnn** - moduł zawierający obudowane standardowe estymatory TF w taki sposób, by mogły one łatwo uczyć się na zbiorach danych zapisanych przy pomocy modułu io_TFRecords. W zależności od tego, czy problem polega na binarnej klasyfikacji czy regresji użyty jest estymator tf.estimator.DNNClassifier lub tf.estimator.DNNRegressor. Ta klasa zajmuje się także normalizacją danych wchodzących do estymatora oraz przeprowadzaniem kodu 1 z n.

- **create_tf_records_folder** - moduł korzystający z modułu io_TFRecords by zapisać dane w postaci formatu TFRecords. Korzysta on także z pliku user_defined_function.py, w którym znajduje się jedyna zależna od problemu funkcja, którą musi zaimplementować użytkownik. Funkcja ta nazywa się get_data() i zwraca dane, które mają zostać zapisane w formacie TFRecords 6.3.1.

6.3 Użycie plików Python

6.3.1 user_defined_function.py

W tym pliku należy zaimplementować funkcję get_data(). Funkcja ta ma za zadanie przekazać dane atrybuty numeryczne oraz kategoryczne. W tym miejscu użytkownik decyduje się, które atrybuty mają zostać użyte do uczenia, a które mają występować tylko w celu analizy wyników. Tutaj również należy określić, czy powstający dataset ma służyć do binarnej klasyfikacji czy do regresji. Szczegółowa dokumentacja tej funkcji znajduje się w komentarzu na początku pliku create_tf_records_folder.py. Przy implementacji tej funkcji warto użyć funkcji read_tree opisanej wcześniej w 5.1.

6.3.2 create_tf_records_folder.py

W tym pliku zaimplementowana jest funkcja create_tfr_record która tworzy przy pomocy modułu io_TFRecords plik z danymi zwróconymi przez funkcję user_defined_function.get_data(). Moduł ten zajmuje się wstępnym tasowaniem tych danych oraz automatycznie rozponaje jakie są zbiory możliwych wartości atrybutów kategorycznych.

6.3.3 model_dnn.py

Użytkownik chcący wytrenować model będzie korzystał z następujących funkcji (mają one dokładną dokumentację w postaci komentarzy w kodzie)

- **create_model_folder**, która pozwala utworzyć model TF oraz wybrać jego architekturę.

- **train** pozwalająca uczyć model na folderze danych uczących utworzonym przez klasę io_TFRecords. Trenowanie zaczyna się w momencie w którym skończyło się poprzednie trenowanie.

- **evaluate** pozwalająca sprawdzić (ewaluować) działanie modelu. Ta funkcja używa również z danych zapisanych przy pomocy klasy io_TFRecords

- **predict** pozwala liczyć predykcje modelu. Zwraca pythonowy generator predykcji. Ta funkcja również używa danych zapisanych przy pomocy klasy io_TFRecords, predykcje są zwracane dla danych w kolejności w jakiej są zapisane w tym folderze.

- **load_model** jest istotna, jeśli chcemy samodzielnie zaimplementować jakąś funkcjonalność. Wczytuje ona model TF zapisany w określonym folderze (jeśli model był już wytrenowany to zostanie wczytany wytrenowany model).

- **input_fn** jest również istotna dla osób modyfikujących działanie tego modułu. Wszystkie estymatory tensorflowowe muszą przyjmować dane w postaci odpowiedniej funkcji zwanej input function. Można tą funkcję zmodyfikować do swoich potrzeb. Ta funkcja to dość ogólna input funkcja czytająca folder utworzony przez moduł io_TFRecords.

6.3.4 io_TFRecords

Z tego modułu użytkownik nie musi bezpośredni korzystać, jednakże warto przytoczyć funcjonalność w celu ułatwienia ewentualnej modyfikacji.

- **create_empty_data_folder** tworzy pusty folder danych. Należy jej powiedzieć jakiego typu danych się spodziewamy w tym folderze (to znaczy podajemy rozmiary atrybutów numerycznych oraz możliwe wartości atrybutów kategorycznych)

- **write_one_example** zapisuje pojedynczą instancję do folderu

- parse_individualy_not_ignored pozwala odczytać tf.dataset zawierający atrybuty uczące. Z tej funkcji korzysta model_dnn.

- **parse_batch_all** pozwala odczytać zarówno atrybuty ignorowane jak i uczące. Jest ona użyteczna jeśli ktoś chce porównać wartość tych ignorowanych wielkości z predykcjami modelu.

7 Podsumowanie

W wyniku projektu powstało oprogramowanie które można zastosować do odróżniania rozpadów $H \to \tau_h \mu$ od tła. Oprogramowanie składa się z 2 dość niezależnych części. Pierwsza z nich pozwala na odczytanie wysokopoziomowych zmiennych charakteryzujących zderzenie, których można użyć do

trenowania głębokiej sieci neuronowej. Druga zaś część umożliwia trenowanie głębokiej sieci neuronowej w wygodny sposób na dość dowolnych danych. Obydwie części są zapisane w ogólny sposób więc mogą być dostosowane do konkretnych potrzeb.

Literatura

[1] The CMS Collaboration. Observation of the Higgs boson decay to a pair of leptons with the CMS detector. *Physics Letters B*, 779:283–316, 2018.

[2] R. Brun and F. Rademakers. ROOT: An object oriented data analysis framework. *Nucl. Instrum. Meth.*, A389:81–86, 1997.

[3] Torbjorn Sjostrand, Stephen Mrenna, and Peter Z. Skands. A Brief Introduction to PYTHIA 8.1. *Comput. Phys. Commun.*, 178:852–867, 2008.

[4] Python org. webpage. `https://www.python.org`. Accessed: 2019-01-21.

[5] Boost library webpage. `https://www.boost.org`. Accessed: 2019-01-21.

[6] Tensorflow library webpage. `https://www.tensorflow.org`. Accessed: 2019-01-21.

[7] Numpy library webpage. `http://www.numpy.org`. Accessed: 2019-01-21.

[8] json library webpage. `https://docs.python.org/2/library/json.html`. Accessed: 2019-01-21.