ML as a Service for HEP

Valentin Kuznetsov, Cornell University
Machine Learning as a Service

MLaaS is a set of tools and services service providers offer to clients to perform various ML tasks, e.g. classification, regression, DeepLearning, etc.

Tools and services include: data visualization, pre-processing, model training and evaluation, serving predictions, etc.

Major tech companies (Goolge, Amazon, Microsoft, IBM, etc.) and plethora of start-ups provide different types of MLaaS services.

Can we use existing MLaaS in HEP?
Traditional ML workflow consists of the following components:

- Obtain train, test, validation datasets in tabular (row-wise) data-format, most of the cases ML deal with either CSV or NumPy arrays representing tabular data.
- Train ML model and for inference (separation leads towards MLaaS concept).
- Input datasets are usually small, < O(GB) and should fit into RAM of the training node.
- HEP datasets may be quite large, at PB scale, they are stored in ROOT data-format, and can be distributed across the GRID nodes.
- Traditional ML frameworks can’t read HEP data in ROOT data-format.
- It creates a gap between CS, ML and HEP communities.
ML in HEP

Training phase:

- we transform our data from ROOT data-format to CSV/NumPy for training purposes
  - other pre-processing steps can be done at this phase
- we train our ML models using available ML frameworks, e.g. Python+Keras, TF, PyTorch
  - we don’t use ROOT data directly in ML frameworks

Inference phase:

- we access trained ML models via external libraries integrated into our frameworks, e.g. CMS CMSSW-DNN, ATLAS LTNN libraries, etc.
  - R&D for specialized solutions to speed-up inference on FPGAs, e.g. HLS4ML, SonicCMS, etc.
  - Resource utilization constrained by run-time, can’t be used outside framework language (C++)

Does these approaches sufficient?
Step towards MLaaS for HEP

MLaaS for HEP should provide the following:

- **natively read HEP data**, e.g. be able to read ROOT files from local or remote distributed data-sources
- **utilize heterogeneous resources**, local CPU, GPUs, farms, cloud resources, etc.
- **use different ML frameworks** (TF, PyTorch, etc.)
- **minimize infrastructure changes**, should be able to use it in different frameworks, inside or outside framework boundaries
- **serve pre-trained HEP models**, à la model repository, and access it easily from any place, any code, any framework
ML as a Service for HEP R&D

- Data streaming and training tools: [github.com/vkuznet/MLaaS4HEP](https://github.com/vkuznet/MLaaS4HEP)

- Data inference tools: [github.com/vkuznet/TFaaS](https://github.com/vkuznet/TFaaS)

**Goal:**
- be able to read arbitrary size dataset(s) from ROOT files
- be able to plug ROOT data into existing ML frameworks
- be able to access pre-trained models anywhere
MLaaS for HEP

- **Data Streaming Layer** is responsible for local or remote data access of HEP ROOT files.

- **Data Training Layer** is responsible for feeding HEP ROOT data into existing ML frameworks.

- **Data Inference Layer** provides access to pre-trained HEP model for HEP users.

- All three layers are independent from each other and allow independent resource allocation.

---

V. Kuznetsov, CMS R&D
Data Streaming Layer

- Recent development of DIANA-HEP uproot ROOT I/O library provides ability to read ROOT data in Python, access them as NumPy arrays, and implements XrootD access.

- Now we’re able to access ROOT files via XrootD protocol in C++, Python and Go.

- MLaaS4HEP extends uproot library and provide APIs to read local and remote distributed ROOT files and feed them into existing ML frameworks.
  - the DataReader and DataGenerator wrappers were created to read local or remote ROOT files and deliver them upstream as batches.
    - random reads from multiple files are also supported (data shuffle mode).
  - the ROOT data are read and represented as Jagged Arrays.
    - we explored both vector and matrix representations, see next slides.
We can read ROOT files via uproot

Each event is a composition of flat and jagged arrays

Usually flat arrays size is less then jagged ones

Such data representation is not directly suitable for ML (dynamic dimension of jagged arrays across events) and should be flatten to fixed size inputs

To feed these data into ML two-step procedure is required:

- obtain dimensionality of jagged arrays
- flatten jagged array into fixed size array
Data transformation (vector wise)

Transform jagged NumPy array into flat one

V. Kuznetsov, CMS R&D
Data transformation (matrix wise)

NumPy array

Transform jagged NumPy matrix form (eta-phi phase)

Transform matrix form into vector

flat branches

branch vector w/ 0's

branch vector

rest of branch vectors

jagged branches representation as fixed size branch vectors in some (eta-phi) space
ML and Jagged Arrays

- In order to use ML we need to resolve how to treat Jagged Array input
  - as array with padding values via vector-wise transformation
    - need to know up-front dimensionality of every jagged array attribute (pre-processing step)
    - padding values should be assigned as NANs since all other numerical values can represent attribute spectrum
  - as a large sparse array via matrix-wise transformation
    - need to choose granularity of matrix cells
    - need to choose a view transformation (X-Y, eta-phi, etc.)
  - it is possible to have collisions in a cell from different jagged array attributes which happen to have the same cell coordinate (can be resolve via multi-dimensional matrix representation, e.g. combining X-Y, Y-Z and Z-X views)
Keep data array with paddings (NaNs) and mask array with their locations

- mask array may be useful when training AutoEncoder type models where we can use mask array to cast padded values after the decoding the data

- Either adjust ML framework or handle data accordingly for existing ML frameworks

- write a wrapper for existing ML framework to deal with two input arrays, e.g. for training NN models we can assign NaNs to zeros to handle **WxX** multiplications
Data Inference Layer

- Data Inference Layer is implemented as TensorFlow as a Service (TFaaS)
- Capable of serving any TensorFlow models
- Can be used as global repository of pre-trained HEP models
- Can be deployed everywhere
  - Docker image and Kubernetes files are provided
  - TFaaS is available as part of DODAS (Dynamic On Demand Analysis Service)

FROM DEPLOYMENT TO PRODUCTION

1. Deploy docker image:
   ```
   docker run --rm -h 'hostname' -f -p 8083:8083 -i -t veknet/terraform
   ```
2. Upload your model:
   ```
   curl -X POST http://localhost:8083/upload -F 'name=ImageModel' -F 'params=@/path/params.json' -F 'model=@/path/tf_model.pb' -F 'labels=@/path/labels.txt'
   ```
3. Get predictions:
   ```
   curl https://localhost:8083/image -F 'image=@/path/file.png' -F 'model=ImageModel'
   ```

Flexible configuration parameters allows you to adopt TFaaS deployment to any use case.
TFaaS features

- HTTP server written in Go to serve arbitrary TF model(s) using Google Go TF APIs
  - users may upload any number of TF models, models are stored on local filesystem and cached within TFaaS server
  - TFaaS repository provides instructions/tools to convert Keras models to TF
  - TFaaS supports JSON and ProtoBuffer data-formats
- Any client supporting HTTP protocol, e.g. curl, C++ (via curl library or TFaaS C++ client), Python, can talk to TFaaS via HTTP end-points
  - C++ client library talks to TFaaS using ProtoBuffer data-format, all others use JSON
- Benchmarks: 200 concurrent calls, throughput 500 req/sec for TF model with 1024x1024 hidden layers
  - performance are similar to JSON and ProtoBuffer clients
Model inspection

Integrated **Netron** model viewer
TFaaS HTTP end-points

- **/json**: handles data send to TFaaS in JSON data-format, e.g. you’ll use this API to fetch predictions for your vector of parameters presented in JSON data-format (used by Python client)

- **/proto**: handlers data send to TFaaS in ProtoBuffer data-format (user by C++ client)

- **/image**: handles images (png and jpg) and yields predictions for given image and model name

- **/upload**: upload your TF model to TFaaS

- **/delete**: delete TF model on TFaaS server

- **/models**: return list of existing TF models on TFaaS

- **/params**: return list of parameters of TF model

- **/status**: return status of TFaaS server
Python client

- **upload** API lets you upload your model and model parameter files to TFaaS

  ```
tfaas_client.py --url=url --upload=upload.json  # upload.json contains model parameters
  ```

- **models** API lets you list existing models on TFaaS server

  ```
tfaas_client.py --url=url --models
  ```

- **delete** API lets you delete given model on TFaaS server

  ```
tfaas_client.py --url=url --delete=ModelName
  ```

- **predict** API lets you get prediction from your model and your given set of input parameters

  ```
  # input.json: {"keys":["attr1", "attr2", ...], "values": [1,2,...]}

tfaas_client.py --url=url --predict=input.json
  ```

- **image** API provides predictions for image classification (supports jpg or png data-formats)

  ```
tfaas_client.py --url=url --image=/path/file.png --model=HEPImageModel
  ```
#include <iostream>
#include <vector>
#include <sstream>
#include "TFClient.h"

// part of TFaaS repository

int main() {
    std::vector<std::string> attrs;
    std::vector<float> values;
    auto url = "http://localhost:8083/proto"; // define your TFaaS server URL
    auto model = "MyModel"; // name of your model in TFaaS

    // fill out your data
    for(int i=0; i<42; i++) {
        values.push_back(i); // create your vector values
        std::ostringstream oss;
        oss << i;
        attrs.push_back(oss.str()); // create your vector headers
    }

    // make prediction call
    auto res = predict(url, model, attrs, values); // get predictions from TFaaS server
    for(int i=0; i<res.prediction_size(); i++) {
        auto p = res.prediction(i); // fetch and print model predictions
        std::cout << "class: " << p.label() << " probability: " << p.probability() << std::endl;
    }
}
TFaaS: use cases

- TFaaS provides access to TF models independently from framework and infrastructure
  - easy to integrate into existing workflow, e.g. Python or C++ or any other (via HTTP protocol)
  - C++ client library can be used to integrate within C++ framework, based on curl and protobuf libraries

- Rapid development or continuous training of TF models and their validation
  - clients can test multiple TF models at the same time

- TFaaS can be used as repository of pre-trained HEP TF models

- TFaaS deployment is trivial (via docker) and you can setup your TFaaS server at your premises, e.g. on your local hardware or at a cloud provider (tested with DODAS) or as k8s deployment

- Can be used in distributed environment, i.e. clients can connect to TFaaS server(s) via HTTP
MLaaS for HEP proof-of-concept

- Train toy models in PyTorch and TF (via Keras) using CMS NANOAOD data
  - run code locally on laptop, lxplus and GPU node
  - access data from local or remote ROOT files
- Serve model in TFaaS server deployed at CERN k8s cluster
MLaaS workflow w/ user models

PyTorch example

```python
from jarray.pytorch import JaggedArrayLinear
import torch

def model(idim):
    "Simple PyTorch model for testing purposes"
    model = torch.nn.Sequential(
        JaggedArrayLinear(idim, 5),
        torch.nn.ReLU(),
        torch.nn.Linear(5, 1),
    )
    return model
```

Keras/TF example

```python
from keras.models import Sequential
from keras.layers import Dense, Activation

def model(idim):
    "Simple Keras model for testing purposes"
    model = Sequential([
        Dense(32, input_shape=(idim,)),
        Activation('relu'),
        Dense(2),
        Activation('softmax'),
    ])  
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

```bash
./workflow.py -files=files.txt -model=<model.py> -params=params.json
```
Read remote ROOT file

Init ML model

Perform train cycle

Read another ROOT file

Perform another train cycle
Benefits of MLaaS approach

- Clear separation of streaming, training and inference layers
  - dynamically and independently scale resources for training and inference layers
- Hide complexity of data transformation from ROOT I/O to ML
  - user data-transformation can be dynamically loaded using user based functions
- Ability to use ML framework of your choice
  - R&D work towards model transformation from one framework to another
- Inference results can accessible via HTTP protocol
  - new models can be uploaded and used immediately without changes of existing infrastructure
  - can be used as a global repository of HEP pre-trained models shared across experiment boundaries
Summary: MLaaS training

- MLaaS training layer is capable of reading remote ROOT files
- Data transformation from Jagged Array representation to vector form
- Random reads from multiple files (data shuffle mode)
- Customization: total number of events to read; data read chunk size; select or exclude branches to read, choice of XrootD redirector
- Dynamically load user based models

Planning: dynamically load user based pre-processing functions; visual inspection of ROOT file content (via go-hep/groot); possible graphical UI to build full workflow pipeline (via go-hep/groot); perform tests as part of DODAS infrastructure
Summary: TFaaS inference

- TFaaS server natively supports concurrency, it organizes TF models in hierarchical structure on local file system, and it uses cache to serve TF models to end-users
- no integration is required to include TFaaS into your infrastructure, i.e. clients talks to TFaaS server via HTTP protocol (python and C++ clients are available)
- allow separation TF models from experiment framework, do not use experiment framework run-time resources, dedicated resources can be used to scale TFaaS
- can be used as model repository, TFaaS architecture allows to implement model versioning, tagging, …
- TFaaS server was tested with concurrent clients, we obtained 500 req/sec throughput for mid-size model inference (subject of TF model complexity)
- TFaaS docker image and k8s deployment files are available
Summary

- MLaaS for HEP is a feasible and we demonstrated fully functional proof-of-concept workflow

- **MLaaS4HEP** repository:
  - **Data Streaming Layer** responsible for remote access to distributed ROOT files and capable of streaming ROOT data via uproot ROOT I/O to upstream layers
  - **Data Training Layer** provides necessary data transformation and batch streaming to existing ML frameworks. The main problem is understanding how to deal with Jagged Array in context of ML framework

- **TFaaS** repository (non HEP specific):
  - **Data Inference Layer**: serves TF models via Go HTTP server and Google TF Go APIs
    - ready to use (docker or k8s), provides basic TF model repository functionality
R&D topics

- Model conversion: PyTorch, fast.ai, etc. to TensorFlow
- Model repository: implement persistent model storage, look-up, versioning, tagging, etc.
- MLaaS/TFaaS scalability: explore Kubernetes, auto-scaling, resource provisioning (FPGAs, GPUs, TPUs, etc.)
- Real model training model with distributed data (MLaaS with DODAS)

Collaboration is welcome

https://arxiv.org/abs/1811.04492