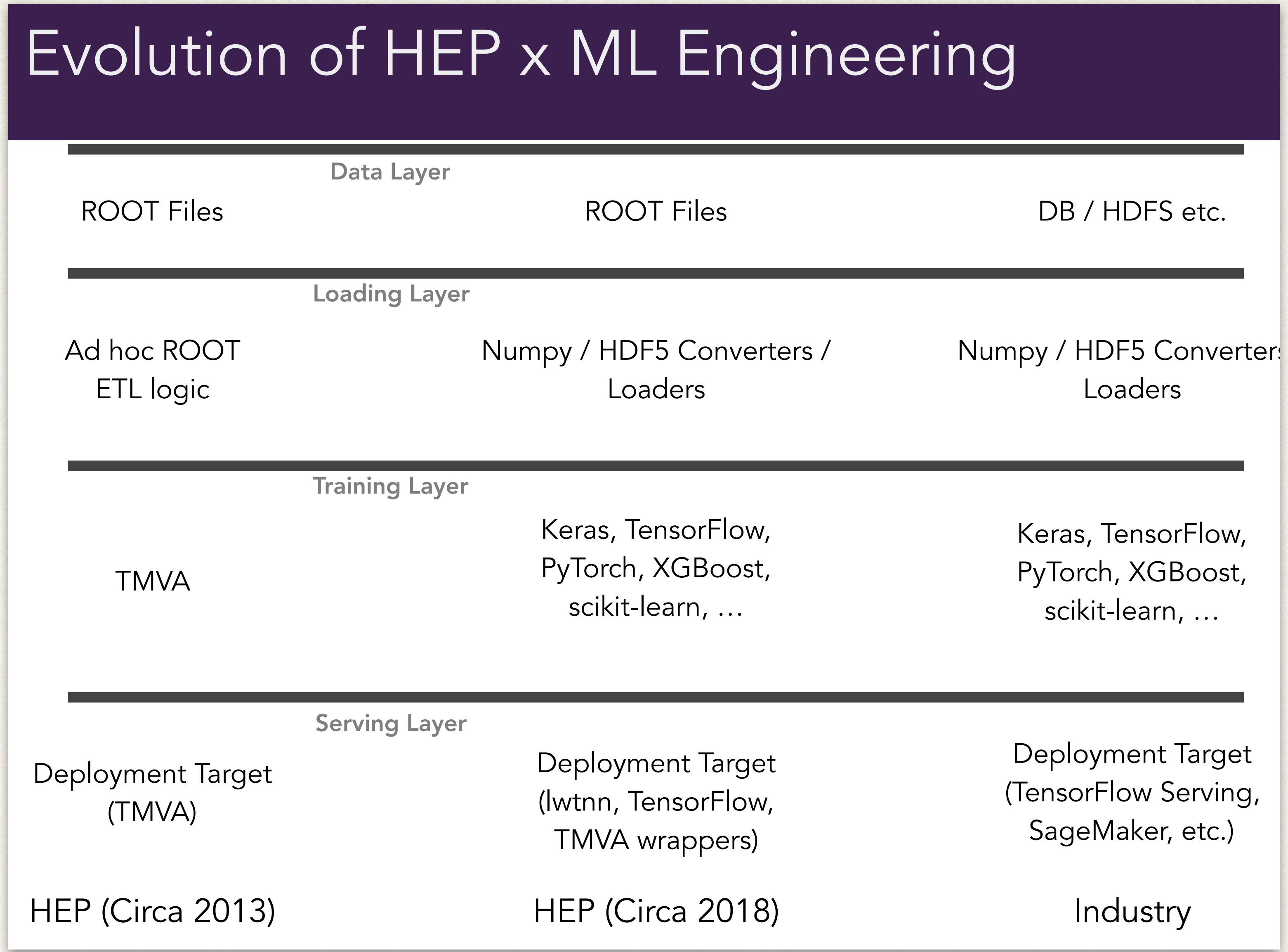
MLaaS for HEP @ PB scale

Valentin Kuznetsov, Cornell University

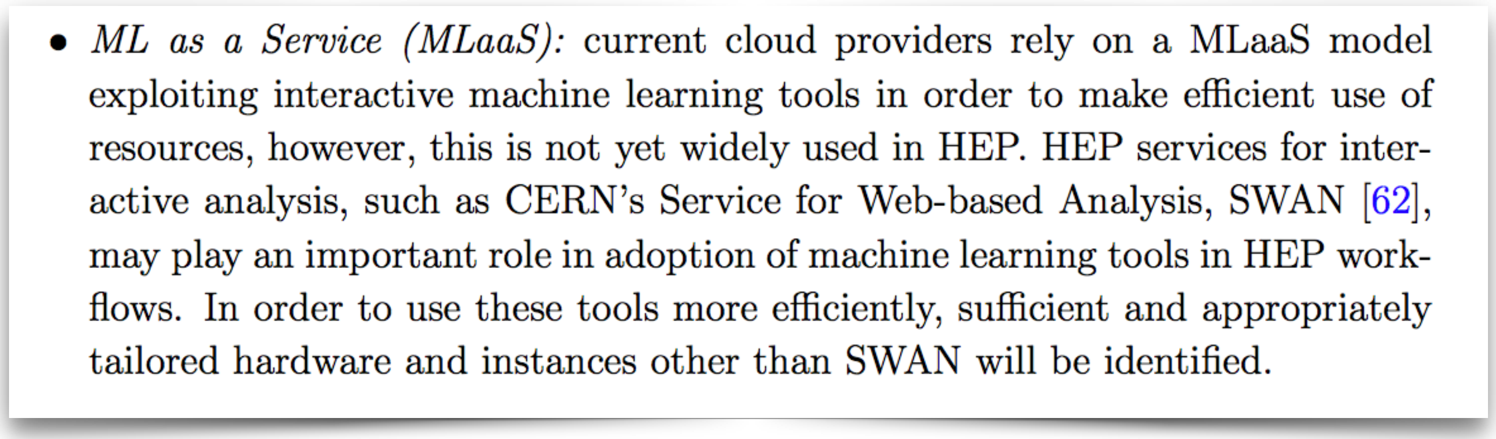
CMS R&D

Luke De Oliveira: IML workshop @ CERN



Evolution of HEP x ML Engineering

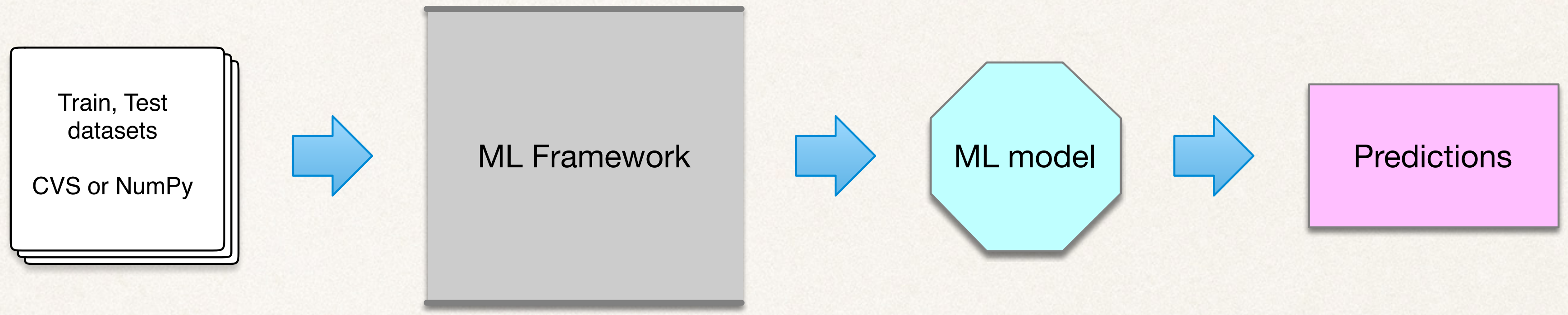
	HEP (Circa 2013)	HEP (Circa 2018)	Industry
Data Layer	ROOT Files	ROOT Files	DB / HDFS etc.
Loading Layer	Ad hoc ROOT ETL logic	Numpy / HDF5 Converters / Loaders	Numpy / HDF5 Converter Loaders
Training Layer	TMVA	Keras, TensorFlow, PyTorch, XGBoost, scikit-learn, …	Keras, TensorFlow, PyTorch, XGBoost, scikit-learn, …
Serving Layer	Deployment Target (TMVA)	Deployment Target (lwtnn, TensorFlow, TMVA wrappers)	Deployment Target (TensorFlow Serving, SageMaker, etc.)



- *ML as a Service (MLaaS):* current cloud providers rely on a MLaaS model exploiting interactive machine learning tools in order to make efficient use of resources, however, this is not yet widely used in HEP. HEP services for interactive analysis, such as CERN's Service for Web-based Analysis, SWAN [62], may play an important role in adoption of machine learning tools in HEP workflows. In order to use these tools more efficiently, sufficient and appropriately tailored hardware and instances other than SWAN will be identified.

HSF Whitepaper (ArXiv:1712.06982)

Traditional ML workflow



- 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
  - use ML framework to train the model which can be used for inference
- Input datasets are usually small, O(GB) and should fit into RAM of the training node

In HEP our data are not stored in tabular format and potentially a dataset may spawn a PB scale

Can we read a PB dataset and train on it?

ML and HEP (CMS use case)

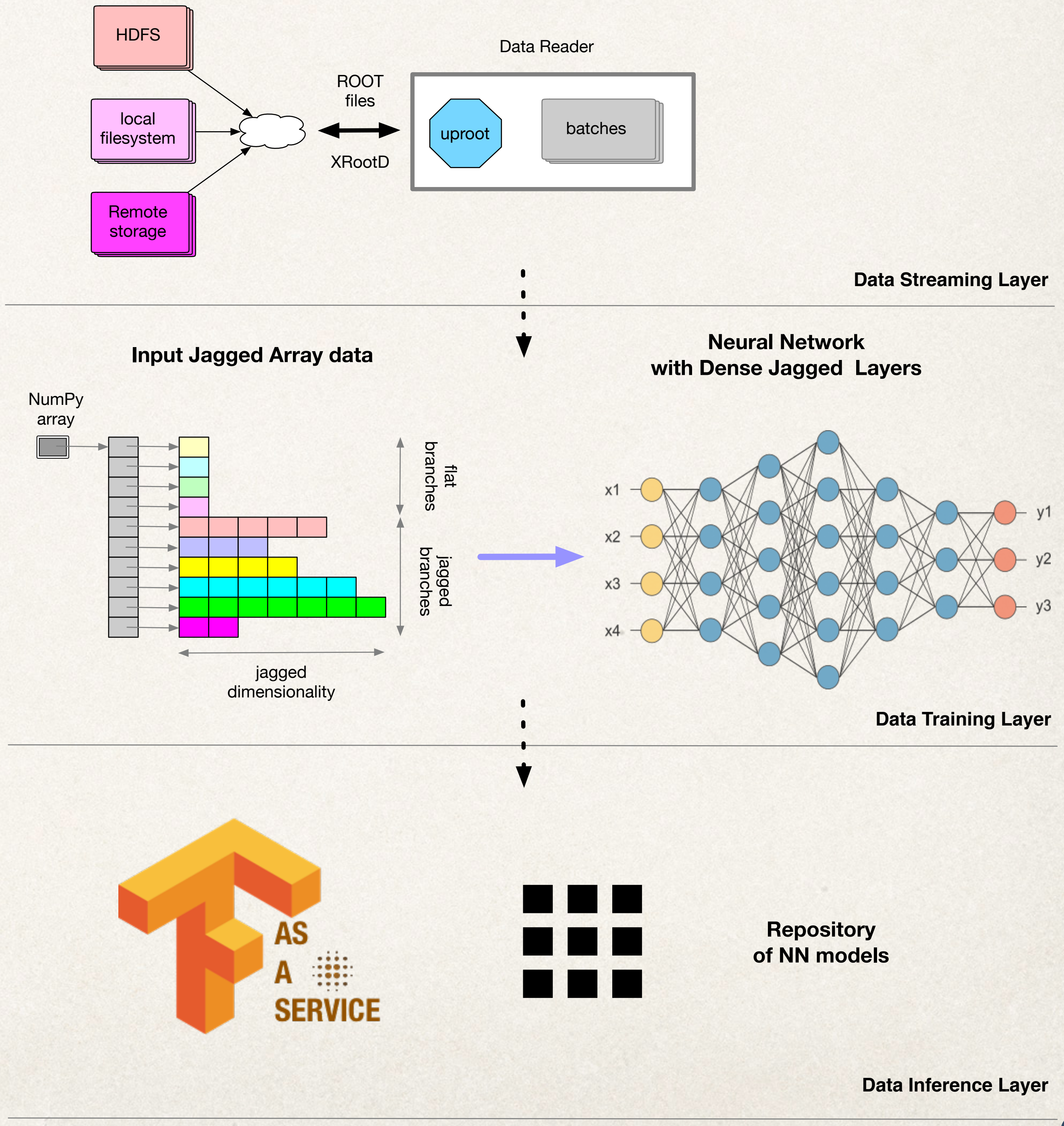
- In CMS we use integrated approach
  - we train our ML models elsewhere using datasets composed in CSV data-format
    - in most cases we use Python+Keras stack
    - we transform our data from ROOT data-format to CSV/NumPy for training purposes
  - we access trained ML models in CMSSW via TensorFlow library for inference purposes, see CMSSW-DNN library

Step towards MLaaS

- Machine Learning as a Service should provide the following:
  - data streaming layer to natively read HEP data (ROOT files)
    - be able to read data from remote, distributed data-sources (AAA model)
  - utilize heterogeneous resources, local CPU, GPUs, farms, cloud resources, etc.
  - be able to work with variety of ML frameworks (TensorFlow, PyTorch, Keras, whatever-will-come-next)
  - be able to serve trained models without touching existing infrastructure or code-base, i.e. you should be able to use your trained model in any language (Python, C++, …), in any framework (CMSSW or plain python code)
  - serve pre-trained HEP models
- Recent development within DIANA-HEP project open-up a possibility to develop MLaaS prototype

MLaaS for HEP

- **Data Streaming Layer** is responsible for remote data access of HEP ROOT files
    - most of R&D was done by DIANA-HEP, see uproot package
- **Data Training Layer** is responsible for feeding HEP ROOT data into existing ML frameworks
    - How to deal with Jagged Array data-representation as input for ML
- **Data Inference Layer** provides access to pre-trained HEP model for HEP users
    - the inference layer can be completely decoupled from existing infrastructure and be implemented “as a Service” solution, see TFaaS implementation (next slides)
- All three layers are independent from each other and allow separation of resources and their locality



MLaaS vs integration approach

MLaaS

- Separation of layers, code maintenance and framework independent
- Decoupling of hardware resources, i.e. your run-time resources are not constrained by ML tasks
- ML training and inference can be separated from run time environment
- Provide data-streaming layer, no data-conversion is required
- Easy to maintain multiple or frequently changed ML models, centrally managed repository of ML models
- Works well in distributed environment, e.g. clients talk to central service
- High throughput achievable via horizontal scaling

Integrated approach

- Close integration of ML within existing framework: CMSSW-DNN or LWTNN (ATLAS)
- Either work within the same language or extra wrapper/translation are required
- Job, storage resources are limited, should be controlled
- ML models should be reachable at run time environment
- Locally managed ML models and *assume* that ML models will not change often
- Harder to maintain and work with distributed environment
- High throughput within a job since ML is integrated within framework

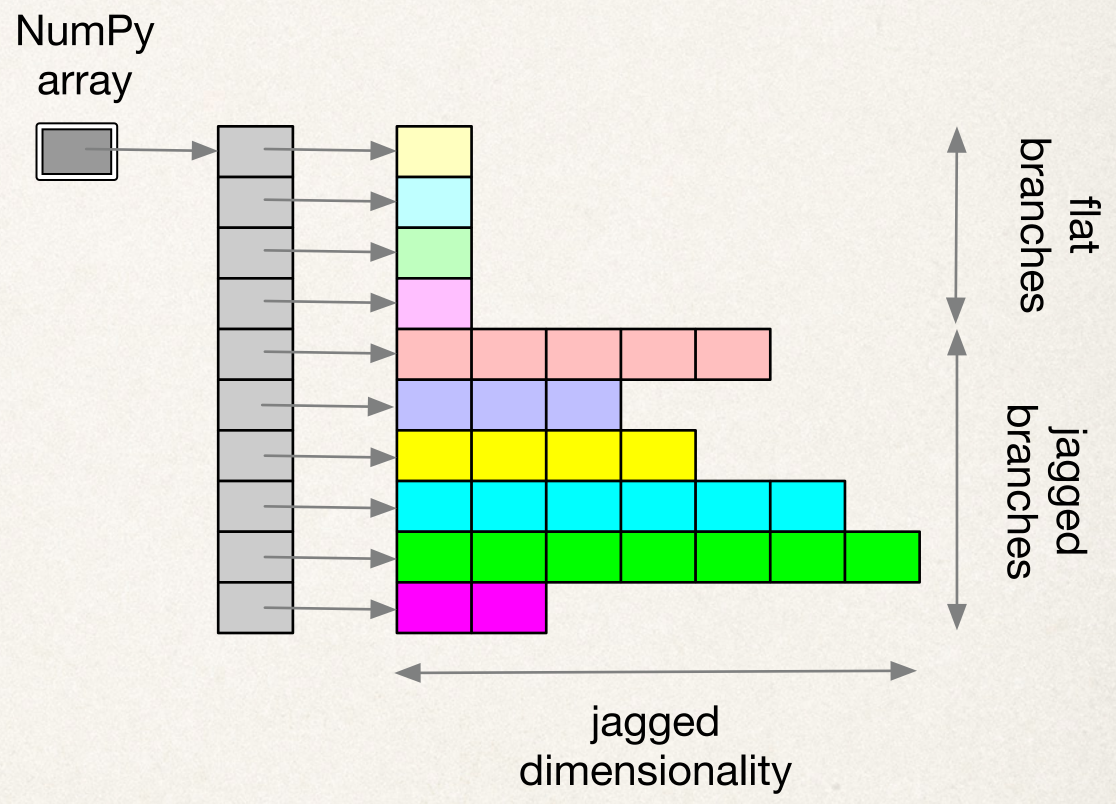
Data Streaming Layer

BETA

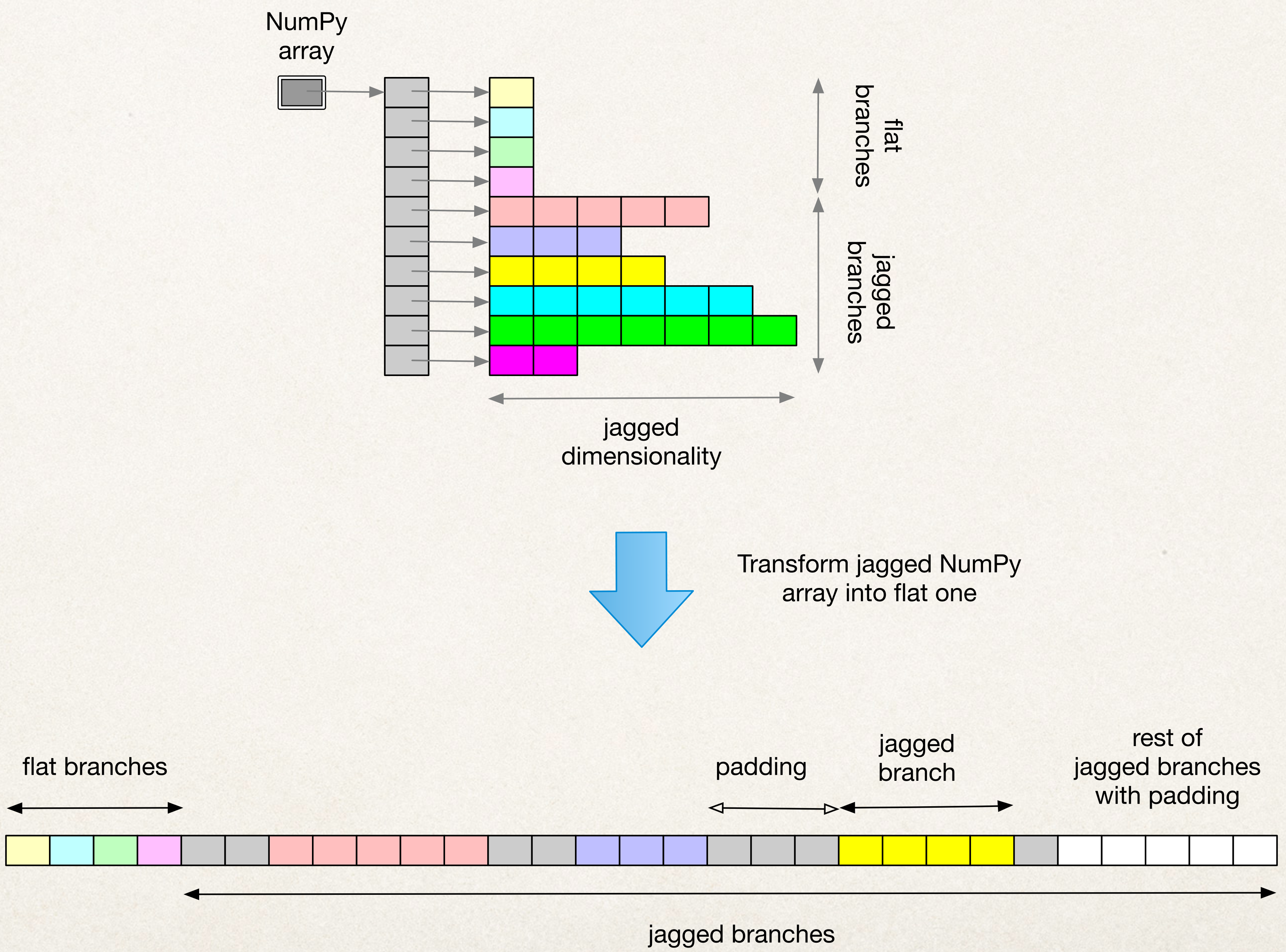
- For years we're able to access ROOT files via XrootD protocol. The AAA project provides middleware to resolve this issue (C++, Python and Go libraries).
  - reading ROOT files was mostly done in our legacy application (C++ frameworks) but Go implementation of ROOT I/O lead to development of uproot Python library
- Recent development of DIANA-HEP provides uproot ROOT I/O library to access ROOT data in Python and moreover access it as NumPy arrays
  - it also implements XrootD access to files
- Therefore it was obvious to extend uproot and provide a Data Streaming wrapper to read remote distributed ROOT files
  - the code is available as a part of TFaaS project, the low-level API is available in reader.py
  - the DataGenerator wrapper was created to serve as a generator of data between remote files and upstream code
  - the ROOT data are read and represented as Jagged Arrays

Data Training Layer, part I **R&D**

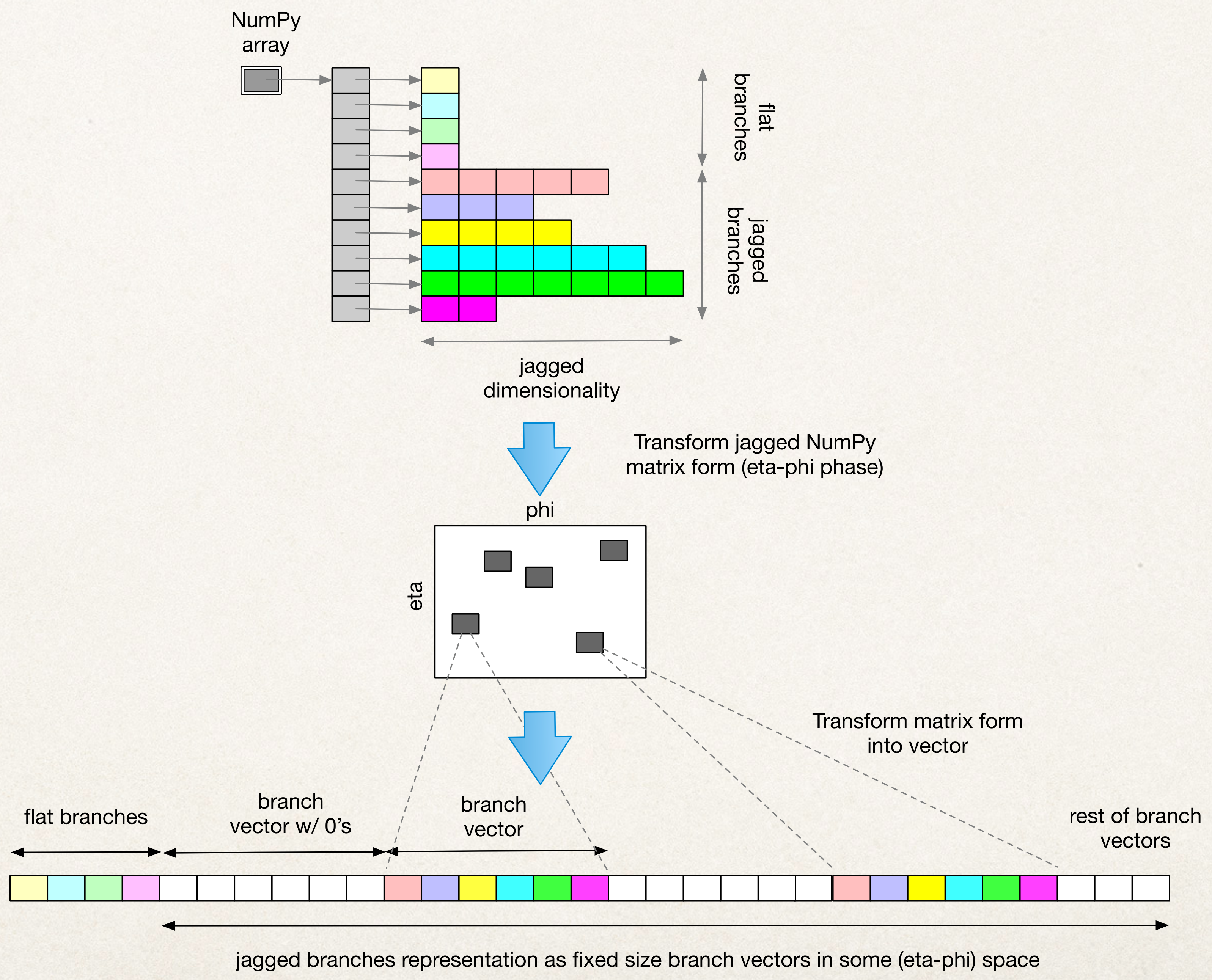
- We can read ROOT files via uproot
- Each event is a composition of flat and jagged arrays
- Usually flat arrays size is less then jagged ones
- Such data representation is not directly suitable for ML (dynamic dimension of jagged arrays across events) and should be flatten to fixed size inputs
- Two-step processing is required:
  - obtain dimensionality of jagged arrays
  - flatten jagged array into fixed size one



Data transformation (vector wise)



Data transformation (matrix wise)

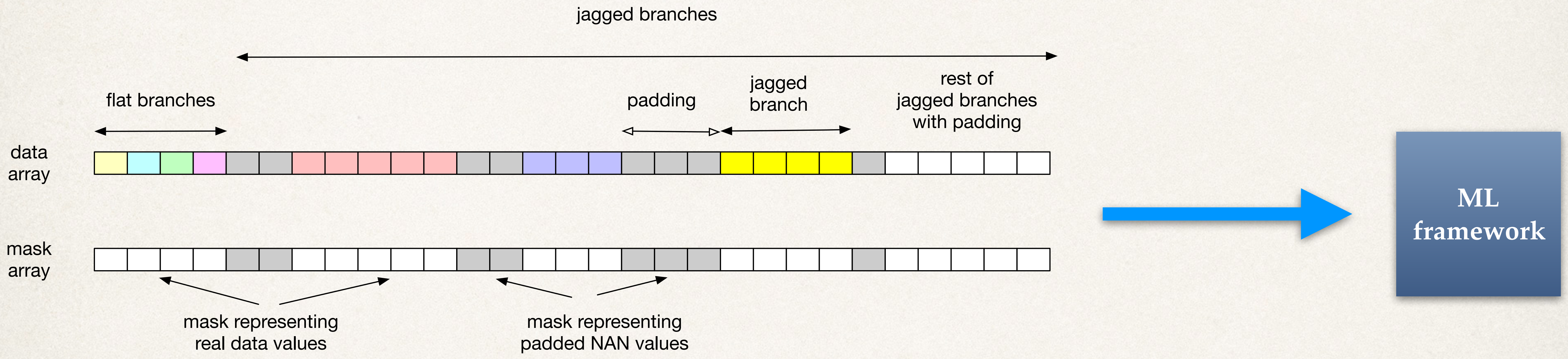


NumPy array
flat branches
jagged branches
jagged dimensionality
Transform jagged NumPy matrix form (eta-phi phase)
phi
eta
Transform matrix form into vector
flat branches
branch vector w/ 0's
branch vector
rest of branch vectors
jagged branches representation as fixed size branch vectors in some (eta-phi) space

NeuralNets and Jagged Array

- In order to use NN 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)
- In NN we perform lots of multiplications (**W**x**X**) and therefore we can replace NANs with zeros to exclude them from minimization of loss function but in order to train network (weights) we need lots of data to cover padding cells

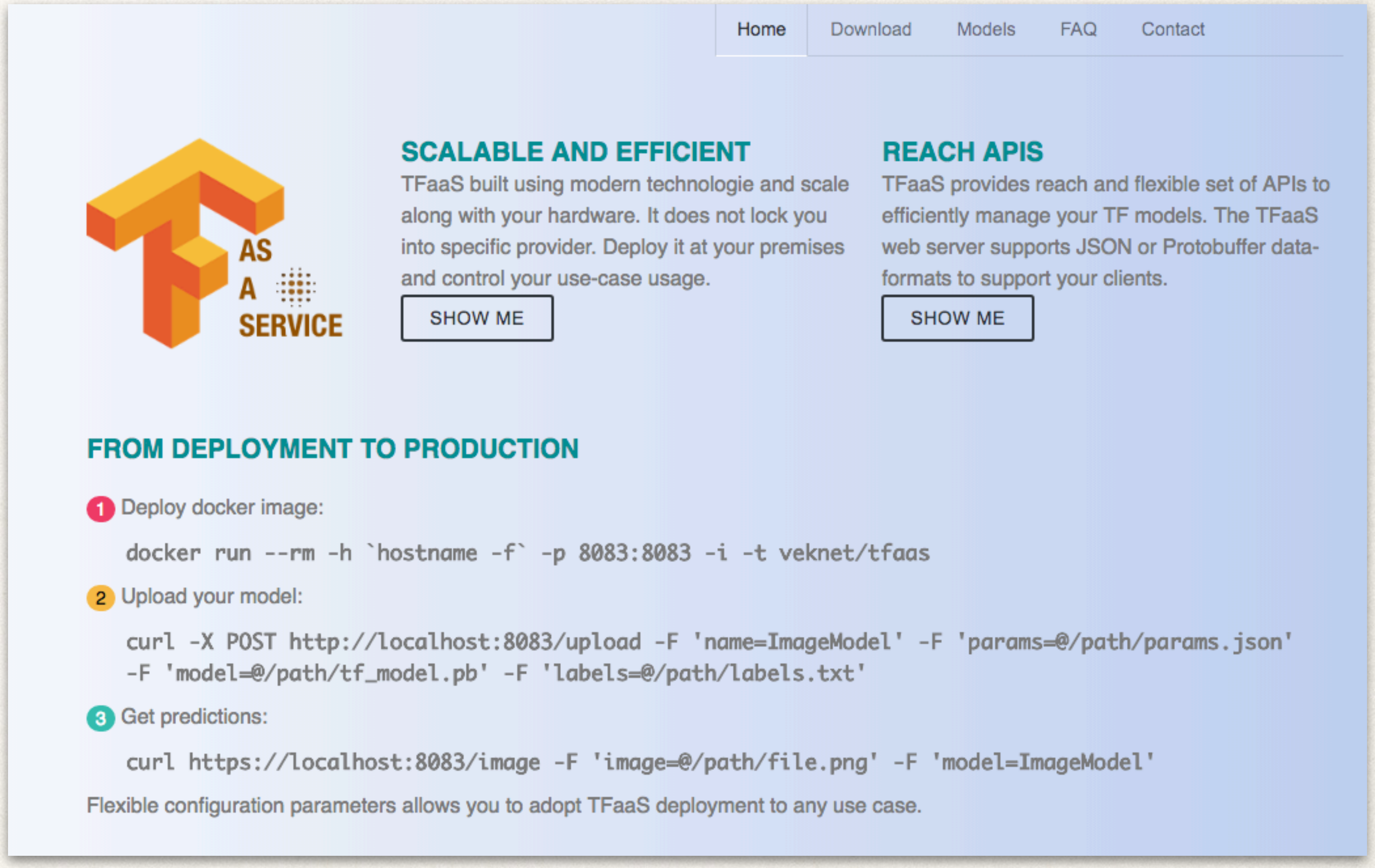
Data Training Layer, part II *R&D*


jagged branches
flat branches
padding
jagged branch
rest of jagged branches with padding
data array
mask array
mask representing real data values
mask representing padded NAN values
ML framework

- Once data are flatten from Jagged Array to fixed size one we need to decide what to do with padding values
    - One solution is to create two arrays: data array with paddings of NaNs and mask array to cast real values of data array from padding ones
    - Existing ML framework may require adjustment to deal with this data representation or we need to represent our data in suitable flat row-wise data-format
- Data Training Layer become independent from Data Streaming one and can reside at facility with suitable hardware (GPU/TPU node, farm)

Data Inference Layer

- Data Inference Layer is implemented as "aaS" data as a Service
- Capable of serving any TensorFlow models
  - model obtained by other ML frameworks can be converted using onnx.ai tool
- Can be used as global repository of pre-trained HEP models
- Can be deployed at different location/sites or centrally
  - tfaas.web.cern.ch is served by Kubernetes image
  - tfaas is available as part of DODAS (Dynamic On Demand Analysis Service)



SCALABLE AND EFFICIENT

TFaaS built using modern technologie and scale along with your hardware. It does not lock you into specific provider. Deploy it at your premises and control your use-case usage.

SHOW ME

REACH APIS

TFaaS provides reach and flexible set of APIs to efficiently manage your TF models. The TFaaS web server supports JSON or Protobuffer data-formats to support your clients.

SHOW ME

FROM DEPLOYMENT TO PRODUCTION

1 Deploy docker image:

```
docker run --rm -h `hostname -f` -p 8083:8083 -i -t veknet/tfaas
```

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'
```

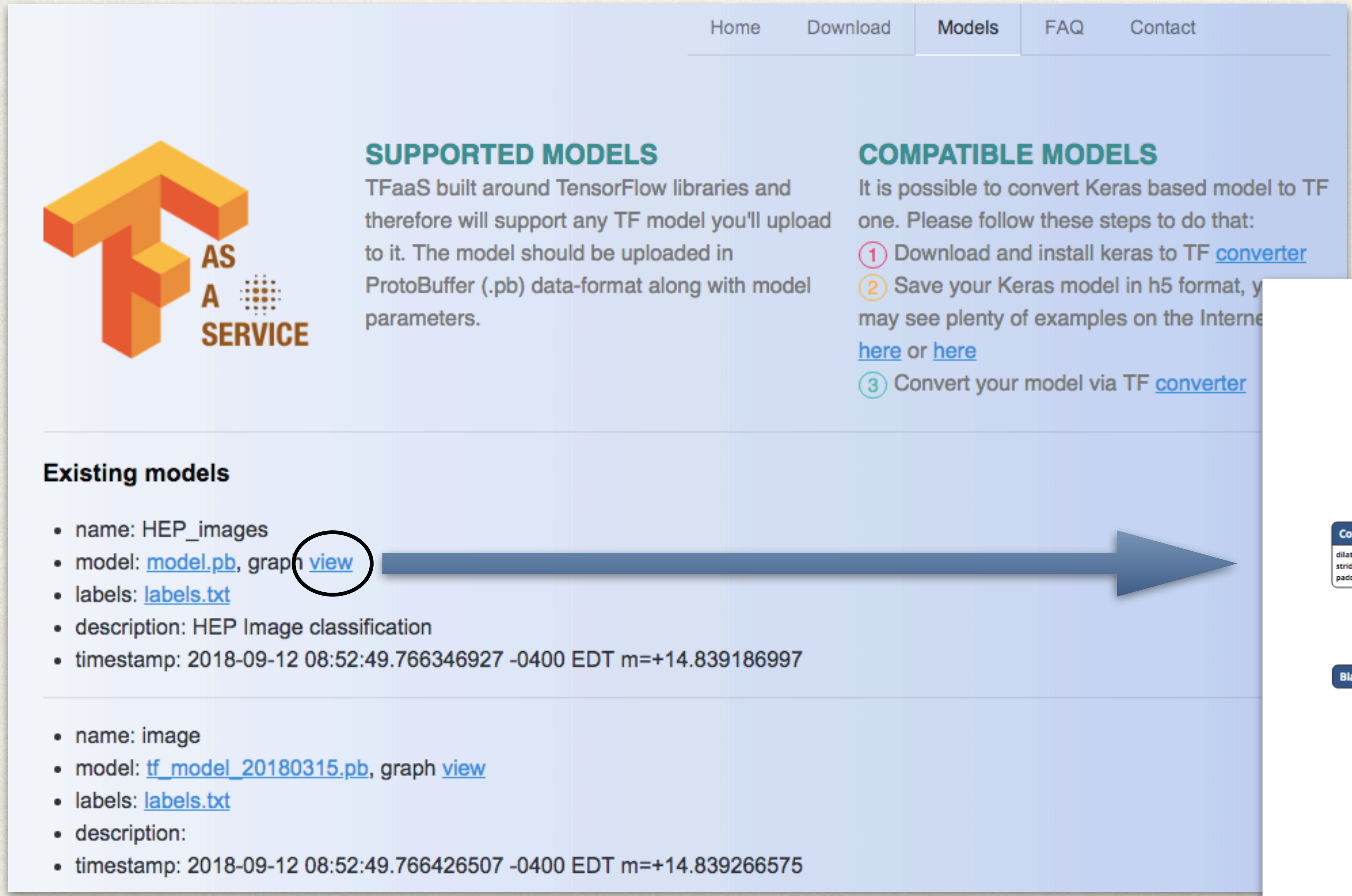
Flexible configuration parameters allows you to adopt TFaaS deployment to any use case.

http://tfaas.web.cern.ch

TFaaS features

- HTTP server (written in Go) to serve **any** TF models
  - users may upload **any number** of TF models, models are stored on local filesystem and cached within TFaaS server
  - TFaaS provides instructions/tools to convert Keras models to TF
  - TFaaS supports JSON and ProtoBuffer data-formats
- Separate model training from inference, train your model on your GPUs locally or remotely, upload them to TFaaS server and serve the predictions via TFaaS APIs
- Clients only required to support HTTP protocol, e.g. curl, C++ (via curl library or TFaaS C++ client), Python can talk to TFaaS via HTTP APIs
  - C++ client library talks to TFaaS via ProtoBuffer data-format, all other clients uses JSON
- Benchmarked with 200 concurrent calls, observed throughput **500 requests/second** for tested TF model
  - performance are similar to JSON and ProtoBuffer clients

Model inspection



SUPPORTED MODELS

TFaaS built around TensorFlow libraries and therefore will support any TF model you'll upload to it. The model should be uploaded in ProtoBuffer (.pb) data-format along with model parameters.

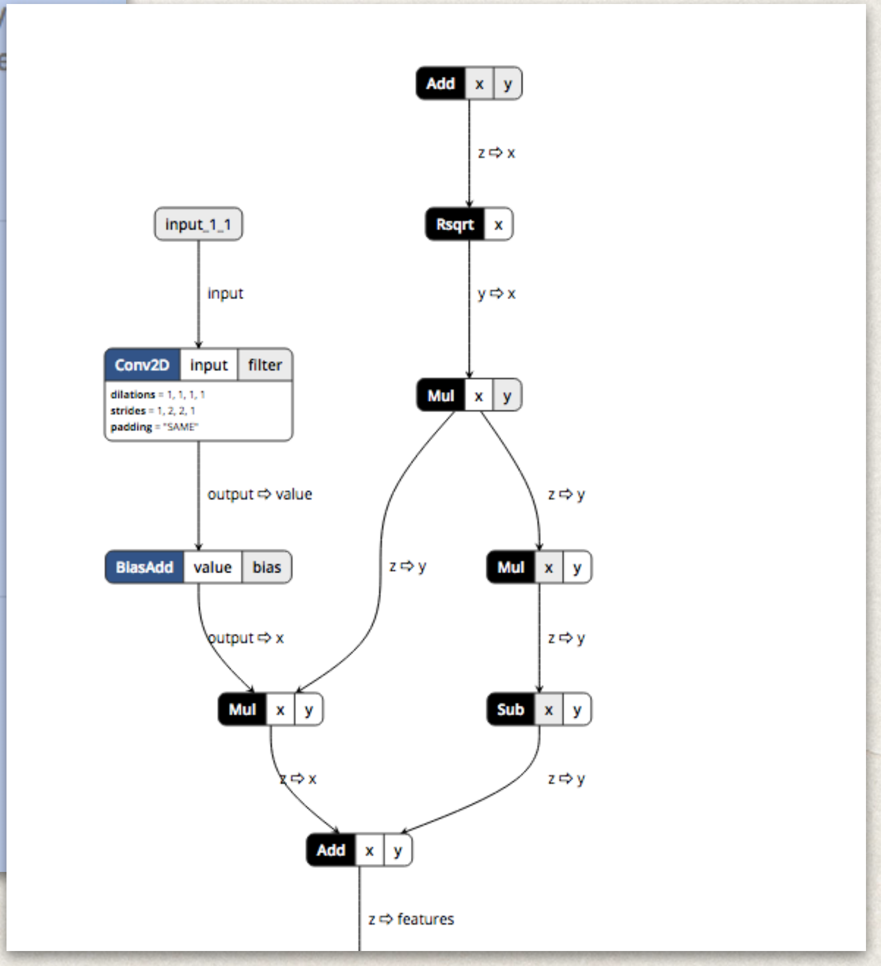
COMPATIBLE MODELS

It is possible to convert Keras based model to TF one. Please follow these steps to do that:
1. Download and install keras to TF converter
2. Save your Keras model in h5 format, y may see plenty of examples on the Interne here or here
3. Convert your model via TF converter

Existing models

- name: HEP_images
- model: model.pb, graph view
- labels: labels.txt
- description: HEP Image classification
- timestamp: 2018-09-12 08:52:49.766346927 -0400 EDT m=+14.839186997

- name: image
- model: tf_model_20180315.pb, graph view
- labels: labels.txt
- description:
- timestamp: 2018-09-12 08:52:49.766426507 -0400 EDT m=+14.839266575



Python client: github repository

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

```
tfaas_client.py --url=url --upload=upload.json
```

- **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

```
tfaas_client.py --url=url --image=/path/file.png --model=HEPImageModel
```

C++ client

CMS BuildFile.xml

```
<use name="protobuf"/>
<lib name="curl"/>
<lib name=“protobuf"/>
```

```
#include <iostream>
#include <vector>
#include <sstream>
#include “TFClient.h”                                   // include TFClient header

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

    // fill out our data
    for(int i=0; i<42; i++) {                           // the model I tested had 42 parameters
        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
    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 APIs: available 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

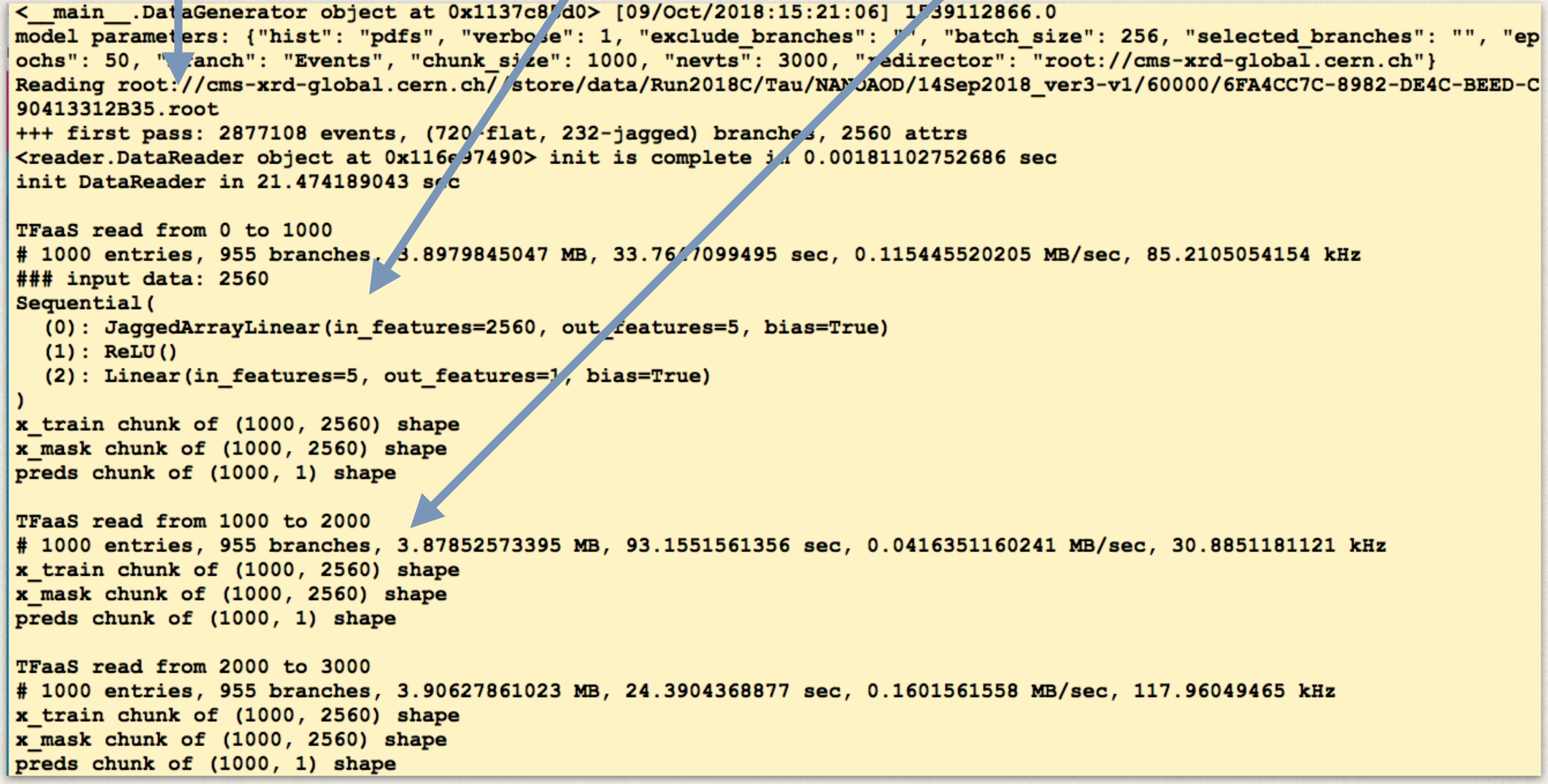
Use cases

- TFaaS provides framework independent access to your TF models
  - easy to integrate into your workflow, either Python or C++
  - C++ client library can be used to integrate within CMSSW as well
- Rapid development or continuous training of TF models and their verification
  - clients can test multiple TF models at the same time
- TFaaS can be used as repository of TF models
- Integration of ML into existing infrastructure without extra development and maintenance effort to support ML infrastructure
- 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)
- TFaaS fits well in distributed environment where clients can connect to central TFaaS server(s) via HTTP APIs

Proof of concept

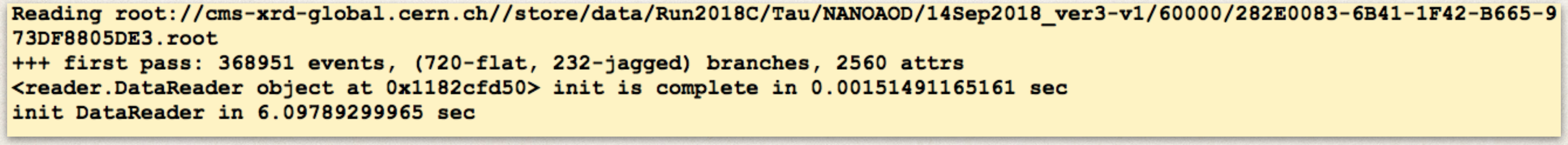
- TFaaS repository contains proof-of-concept test code for training PyTorch and TensorFlow models (via Keras)
- I was able to train a toy NN with CMS NANOAOD (distributed) files
  - discover files with dasgoclient for a pattern dataset
  - pre-process data to identify dimensionality of jagged array branches
  - train toy models in PyTorch and TF (via Keras)
    - used 3 different remote ROOT files, read 3K events from each file
  - run code locally on laptop and on remote GPU node
  - serve model in TFaaS deployed on CERN k8s cluster

Read remote ROOT file   Init ML model   Perform train cycle



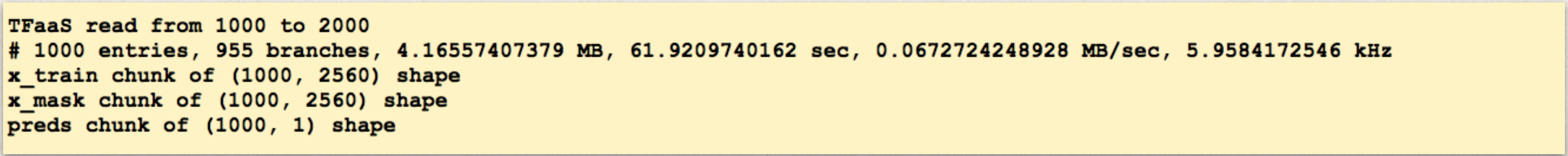
```
<__main__.DataGenerator object at 0x1137c85d0> [09/Oct/2018:15:21:06] 1539112866.0
model parameters: {"hist": "pdfs", "verbose": 1, "exclude_branches": "", "batch_size": 256, "selected_branches": "", "ep
ochs": 50, "branch": "Events", "chunk_size": 1000, "nevts": 3000, "redirector": "root://cms-xrd-global.cern.ch"}
Reading root://cms-xrd-global.cern.ch//store/data/Run2018C/Tau/NANOAOD/14Sep2018_ver3-v1/60000/6FA4CC7C-8982-DE4C-BEED-C
90413312B35.root
+++ first pass: 2877108 events, (720-flat, 232-jagged) branches, 2560 attrs
<reader.DataReader object at 0x116e97490> init is complete in 0.00181102752686 sec
init DataReader in 21.474189043 sec

TFaaS read from 0 to 1000
# 1000 entries, 955 branches, 3.8979845047 MB, 33.7647099495 sec, 0.115445520205 MB/sec, 85.2105054154 kHz
### input data: 2560
Sequential(
   (0): JaggedArrayLinear(in_features=2560, out_features=5, bias=True)
   (1): ReLU()
   (2): Linear(in_features=5, out_features=1, bias=True)
)
x_train chunk of (1000, 2560) shape
x_mask chunk of (1000, 2560) shape
preds chunk of (1000, 1) shape

TFaaS read from 1000 to 2000
# 1000 entries, 955 branches, 3.87852573395 MB, 93.1551561356 sec, 0.0416351160241 MB/sec, 30.8851181121 kHz
x_train chunk of (1000, 2560) shape
x_mask chunk of (1000, 2560) shape
preds chunk of (1000, 1) shape

TFaaS read from 2000 to 3000
# 1000 entries, 955 branches, 3.90627861023 MB, 24.3904368877 sec, 0.1601561558 MB/sec, 117.96049465 kHz
x_train chunk of (1000, 2560) shape
x_mask chunk of (1000, 2560) shape
preds chunk of (1000, 1) shape
```

Read another ROOT file



```
Reading root://cms-xrd-global.cern.ch//store/data/Run2018C/Tau/NANOAOD/14Sep2018_ver3-v1/60000/282E0083-6B41-1F42-B665-9
73DF8805DE3.root
+++ first pass: 368951 events, (720-flat, 232-jagged) branches, 2560 attrs
<reader.DataReader object at 0x1182cfd50> init is complete in 0.00151491165161 sec
init DataReader in 6.09789299965 sec
```

Perform another train cycle



```
TFaaS read from 1000 to 2000
# 1000 entries, 955 branches, 4.16557407379 MB, 61.9209740162 sec, 0.0672724248928 MB/sec, 5.9584172546 kHz
x_train chunk of (1000, 2560) shape
x_mask chunk of (1000, 2560) shape
preds chunk of (1000, 1) shape
```

Summary

- MLaaS for HEP @ PB scale is a feasible. To do that we need three layers
- **Data Streaming Layer** provides remote access to distributed ROOT files and capable of streaming ROOT data via uproot ROOT I/O to upstream layers (clients, e.g. Data Training Layer)
- **Data Training Layer** should provide 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
    - we explored vector and matrix Jagged Array transformation and build ANN with vector representation in PyTorch and Keras
    - a pre-processing step is required to identify dimensionality of Jagged Arrays
- **Data Inference Layer** is in production state and developed as data service (TFaaS)
    - implementation is done using Go server (build-in concurrency) and using Google TensorFlow Go APIs
    - TFaaS is capable of serving any number of TF models and can be used as repository of pre-trained models
    - TFaaS is available as a source code, docker image and kubernetes files. It is also deployed at CERN k8s cluster and DODAS infrastructure

Summary, con’t

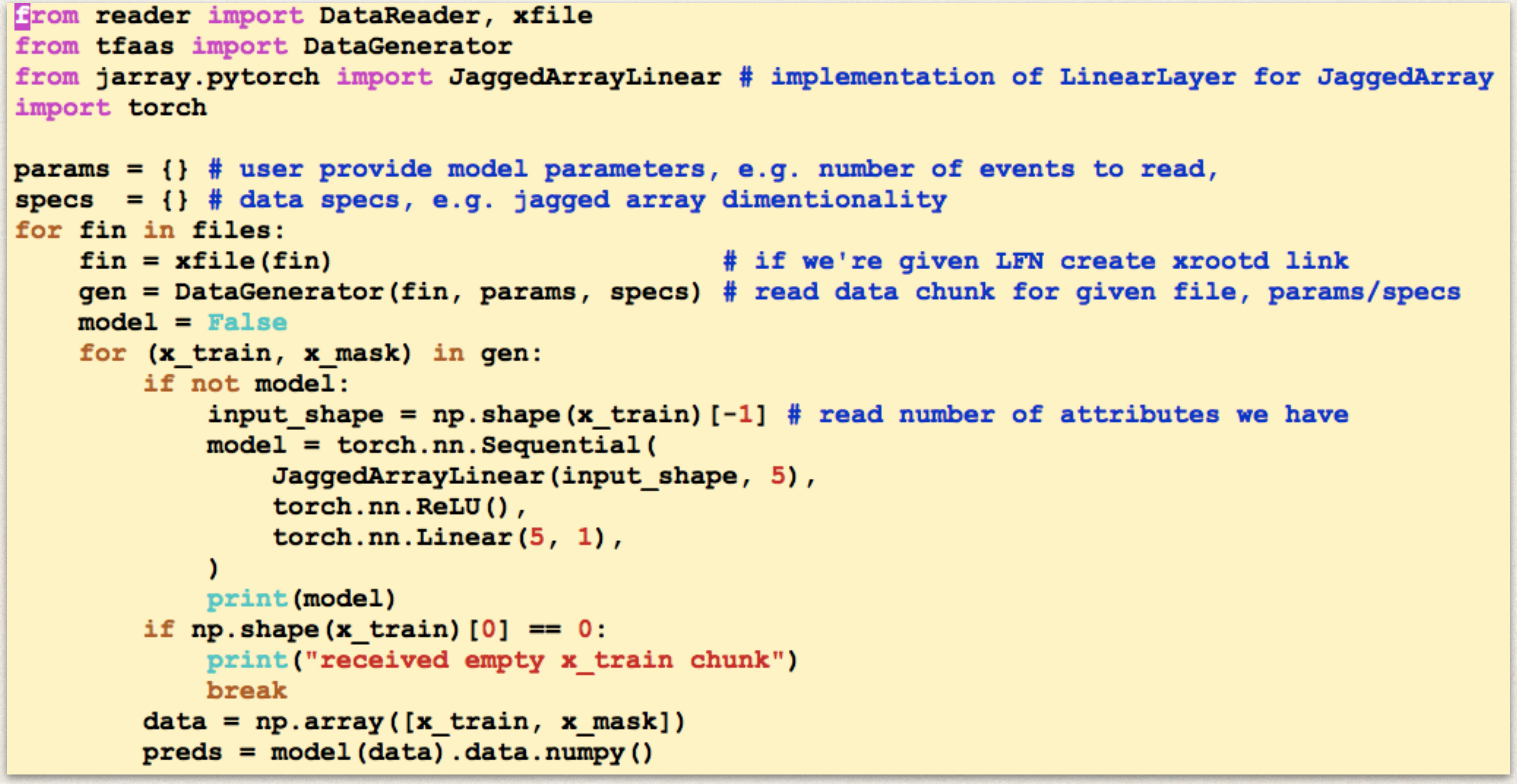
- TFaaS server natively supports concurrency, it organizes TF models in hierarchical structure on local file system, and it uses cache to serve TF models to end-users
    - no integration is required to include TFaaS into your infrastructure, i.e. clients talks to TFaaS server via HTTP protocol (python and C++ clients are available)
    - allow separation TF models from CMSSW framework, do not use CMSSW run-time resources, dedicated resources can be used to serve any number of TF models we may need at run-time
    - can be used as model repository, TFaaS architecture allows to implement model versioning, tagging, …
- TFaaS server was tested with heavy loaded concurrent clients and delivery 500 req/sec throughput (subject of TF model complexity)
- TFaaS is available at http://tfaas.web.cern.ch as a testbed for end-users and we plan to put it to production (cmsweb). Feel free to upload your model and test it (open GitHub issue ticket if necessary).

R&D topics

- Model conversion: PyTorch/fast.ai to TensorFlow
- Model repository: implement persistent model storage, versioning, tagging, etc.
- MLaaS/TFaaS clustering: explore kubernetes and auto-scaling
- Real model training model with distributed data

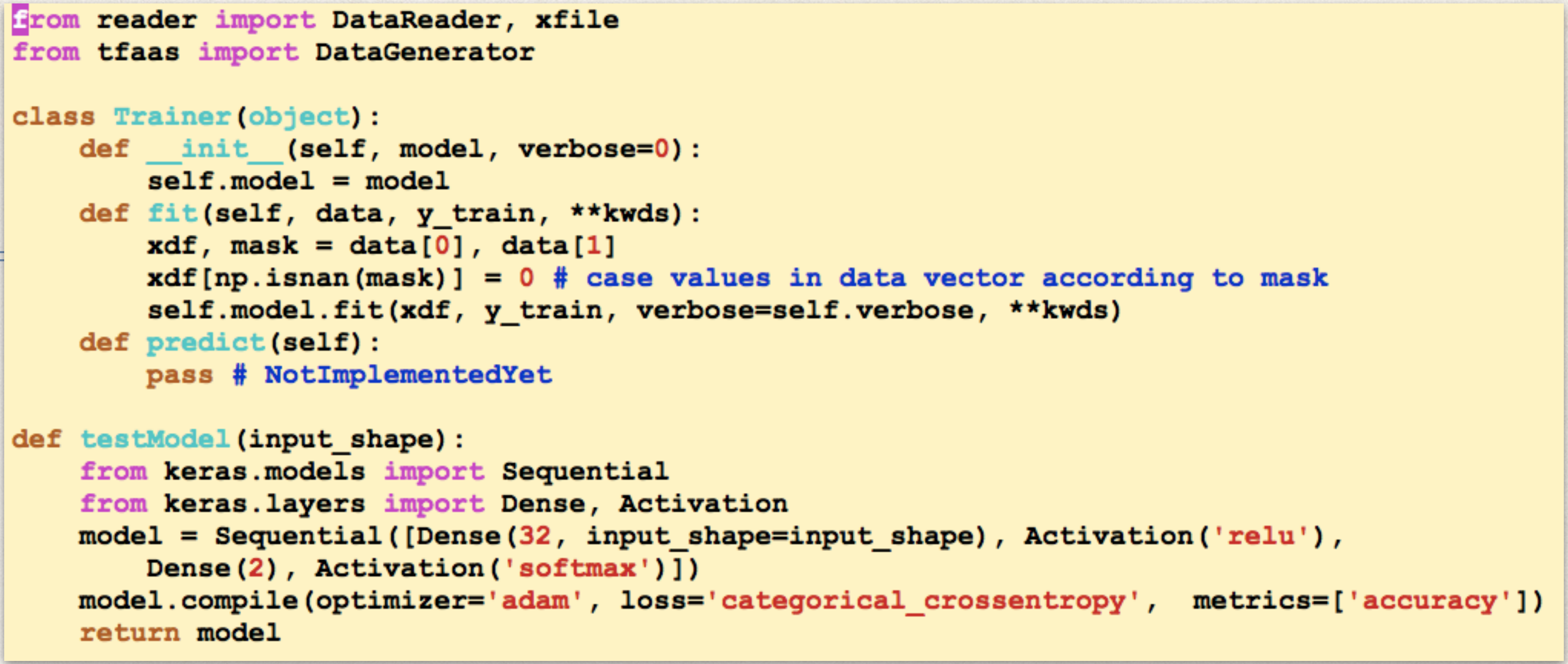
Back-up slides

Torch code example

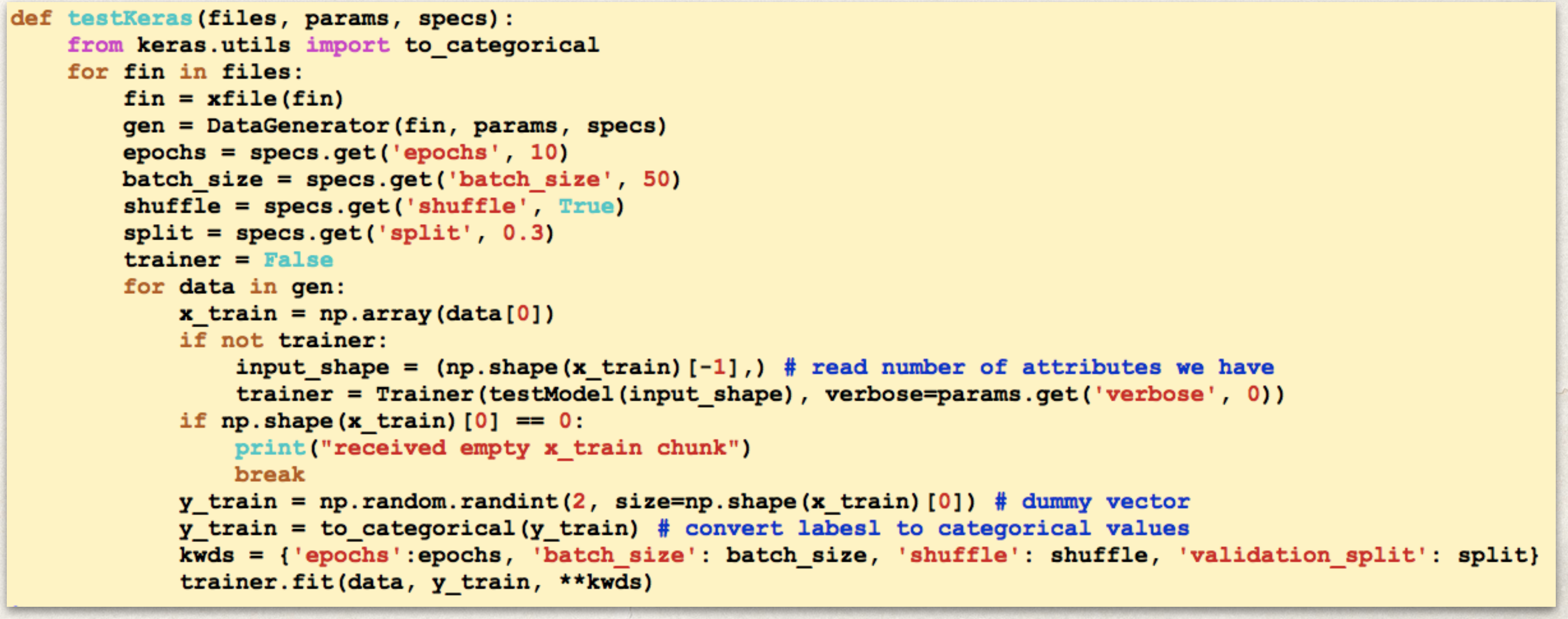


```python
from reader import DataReader, xfile
from tfaas import DataGenerator
from jarray.pytorch import JaggedArrayLinear # implementation of LinearLayer for JaggedArray
import torch

params = {} # user provide model parameters, e.g. number of events to read,
specs  = {} # data specs, e.g. jagged array dimentionality
for fin in files:
    fin = xfile(fin)                        # if we're given LFN create xrootd link
    gen = DataGenerator(fin, params, specs) # read data chunk for given file, params/specs
    model = False
    for (x_train, x_mask) in gen:
        if not model:
            input_shape = np.shape(x_train)[-1] # read number of attributes we have
            model = torch.nn.Sequential(
                JaggedArrayLinear(input_shape, 5),
                torch.nn.ReLU(),
                torch.nn.Linear(5, 1),
            )
            print(model)
        if np.shape(x_train)[0] == 0:
            print("received empty x_train chunk")
            break
        data = np.array([x_train, x_mask])
        preds = model(data).data.numpy()
```

Keras code example



```python
from reader import DataReader, xfile
from tfaas import DataGenerator

class Trainer(object):
    def __init__(self, model, verbose=0):
        self.model = model
    def fit(self, data, y_train, **kwds):
        xdf, mask = data[0], data[1]
        xdf[np.isnan(mask)] = 0 # case values in data vector according to mask
        self.model.fit(xdf, y_train, verbose=self.verbose, **kwds)
    def predict(self):
        pass # NotImplementedYet

def testModel(input_shape):
    from keras.models import Sequential
    from keras.layers import Dense, Activation
    model = Sequential([Dense(32, input_shape=input_shape), Activation('relu'),
        Dense(2), Activation('softmax')])
    model.compile(optimizer='adam', loss='categorical_crossentropy',  metrics=['accuracy'])
    return model
```



```python
def testKeras(files, params, specs):
    from keras.utils import to_categorical
    for fin in files:
        fin = xfile(fin)
        gen = DataGenerator(fin, params, specs)
        epochs = specs.get('epochs', 10)
        batch_size = specs.get('batch_size', 50)
        shuffle = specs.get('shuffle', True)
        split = specs.get('split', 0.3)
        trainer = False
        for data in gen:
            x_train = np.array(data[0])
            if not trainer:
                input_shape = (np.shape(x_train)[-1],) # read number of attributes we have
                trainer = Trainer(testModel(input_shape), verbose=params.get('verbose', 0))
            if np.shape(x_train)[0] == 0:
                print("received empty x_train chunk")
                break
            y_train = np.random.randint(2, size=np.shape(x_train)[0]) # dummy vector
            y_train = to_categorical(y_train) # convert labesl to categorical values
            kwds = {'epochs':epochs, 'batch_size': batch_size, 'shuffle': shuffle, 'validation_split': split}
            trainer.fit(data, y_train, **kwds)
```

TFaaS demos

How to build and run from source code

https://drive.google.com/file/d/1Ipwt9dOJCCb9EN3lmiYKhExel6dd4baO/view

Client-server interaction

curl client: https://www.youtube.com/watch?v=ZGjnM8wk8eA

python client: https://youtu.be/ZhD2jEqc0Fw

Example of model input for TFaaS

upload.json: describe input model

```
{
     "model": "/opt/cms/models/vk/model.pb",
     "labels": "/opt/cms/models/vk/labels.txt",
     "name": "vk",
     "params":"/opt/cms/models/vk/params.json"
}
```

params.json: describe model parameters

```
{
     "name": "vk",
     "model": "model.pb",
     "description": "vk test model",
     "labels": "labels.txt",
     "inputNode": "dense_1_input",
     "outputNode": "output_node0"
}
```

Example of data input for TFaaS

```
{
"keys": ["nJets", "nLeptons", "jetEta_0", "jetEta_1", "jetEta_2",
"jetEta_3", "jetEta_4", "jetMass_0","jetMass_1", "jetMass_2",
"jetMass_3", "jetMass_4", "jetMassSoftDrop_0",
"jetMassSoftDrop_1","jetMassSoftDrop_2", "jetMassSoftDrop_3",
"jetMassSoftDrop_4", "jetPhi_0", "jetPhi_1", "jetPhi_2",
"jetPhi_3","jetPhi_4", "jetPt_0", "jetPt_1", "jetPt_2", "jetPt_3",
"jetPt_4", "jetTau1_0", "jetTau1_1", "jetTau1_2","jetTau1_3",
"jetTau1_4", "jetTau2_0", "jetTau2_1", "jetTau2_2", "jetTau2_3",
"jetTau2_4", "jetTau3_0","jetTau3_1", "jetTau3_2", "jetTau3_3",
“jetTau3_4"],
"values": [2.0, 0.0, 0.9228423833849999,-1.1428750753399999, 0.0, 0.0,
0.0, 155.239425659, 142.709609985, 0.0, 0.0, 0.0, 83.5365982056,
120.549507141,0.0, 0.0, 0.0, 1.9305502176299998, -1.17742347717, 0.0,
0.0, 0.0, 481.419799805, 449.04394531199995, 0.0, 0.0,0.0,
0.296700358391, 0.286615312099, 0.0, 0.0, 0.0, 0.164555206895,
0.19625715911400002, 0.0, 0.0, 0.0,0.117722302675, 0.155229091644, 0.0,
0.0, 0.0],
"model":"vk"}
```