A Machine Learning Method for calorimeter signal processing in sPHENIX

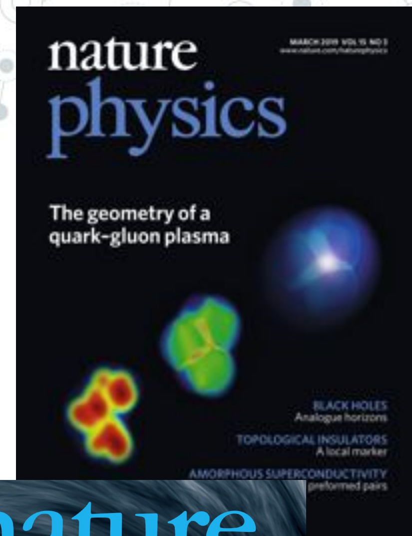
M.Potekhin for sPHENIX Collaboration

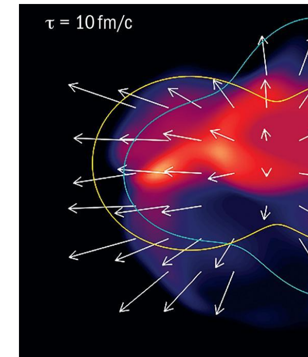
Brookhaven National Laboratory

ACAT 2022

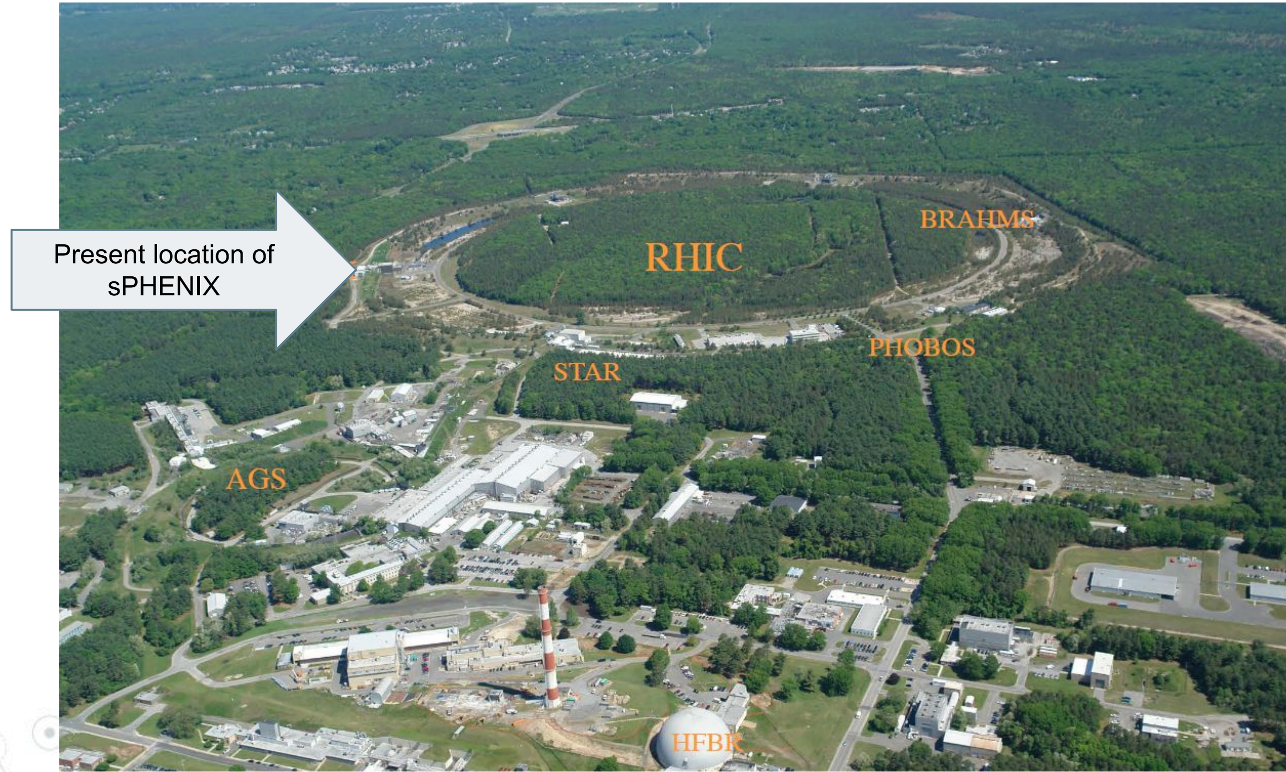
sPHENIX is an experiment at RHIC

- RHIC at Brookhaven National Laboratory – “Relativistic Heavy Ion Collider” – is one of only two operating heavy-ion colliders, built for the purpose of study if nuclear matter at extremely high temperature and/or density
- The most important discovery made at RHIC is that of the Quark-Gluon Plasma (QGP) – a “perfect fluid” of quarks and gluons
- sPHENIX aims to answer some of important questions about the behavior of QGP

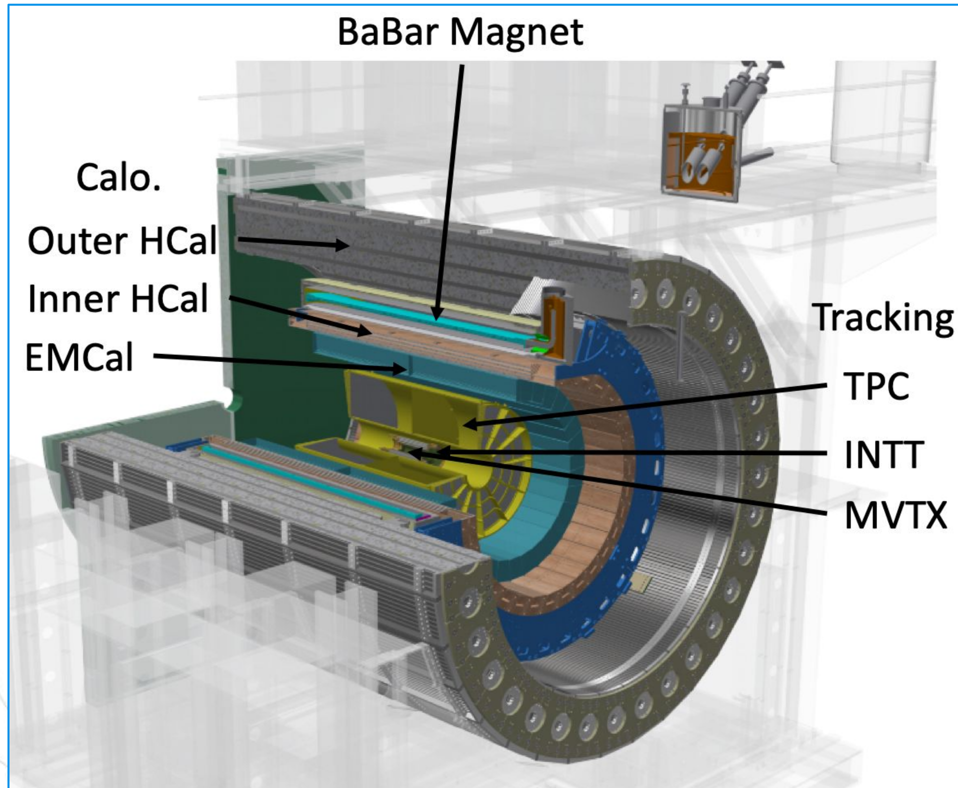




sPHENIX at RHIC



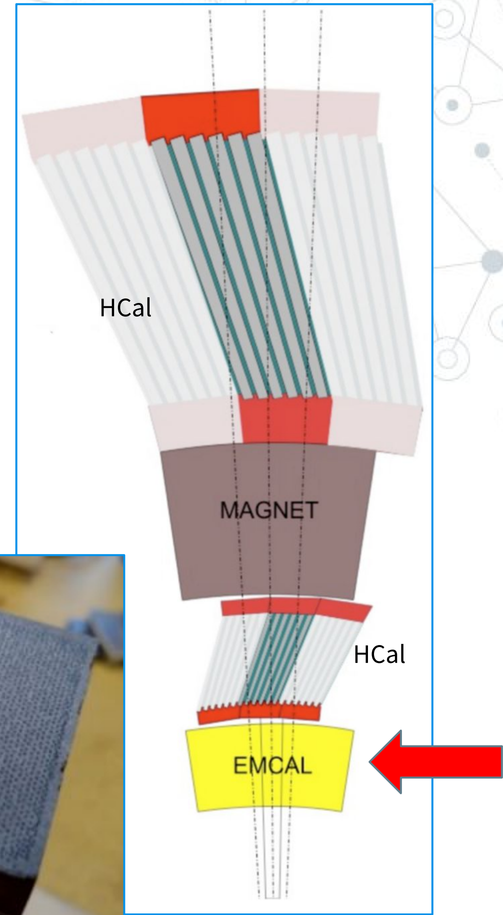
The sPHENIX Detector



First run year	2023
$\sqrt{s_{NN}}$ [GeV]	200
Trigger Rate [kHz]	15
Magnetic Field [T]	1.4
First active point [cm]	2.5
Outer radius [cm]	270
$\|\eta\|$	$\leqslant$1.1
$\|z_{vtx}\|$ [cm]	10
N(AuAu) collisions*	1.43×10^{11}

* In 3 years of running

sPHENIX: the EM Calorimeter (EMCAL)



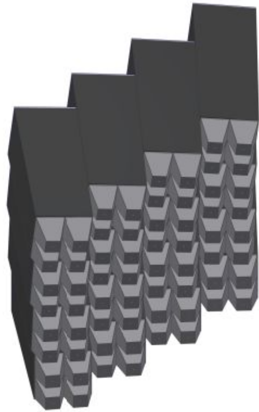
- Tungsten, epoxy & fiber, 2D projective design
- ~$20X_0$, |η| < 1.1, 2π azimuthal acceptance
- 24576 towers
- ~13%/√E resolution

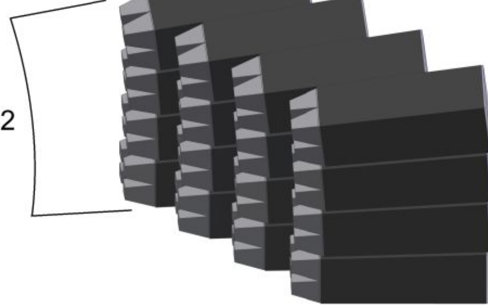
sPHENIX readout

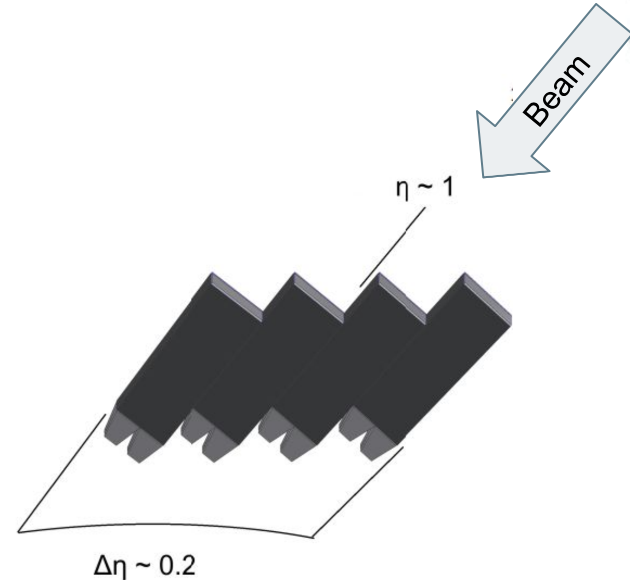
- sPHENIX is the first large experiment which does not have online event building – data streams from its components are combined offline
- FEE is interfaced with FELIX boards
- The nominal data rate is 135Gbit/s
- Data volume: ~70PB of data annually, for the first two years of running

Test Beam Experiment (conducted at FNAL)

- Prototype: 4 blocks, each comprising a 2×2 array of towers
- Geometry corresponds to the nominal η~1, coverage: Δη×Δϕ = 0.2×0.2
- 60MHz digitizer

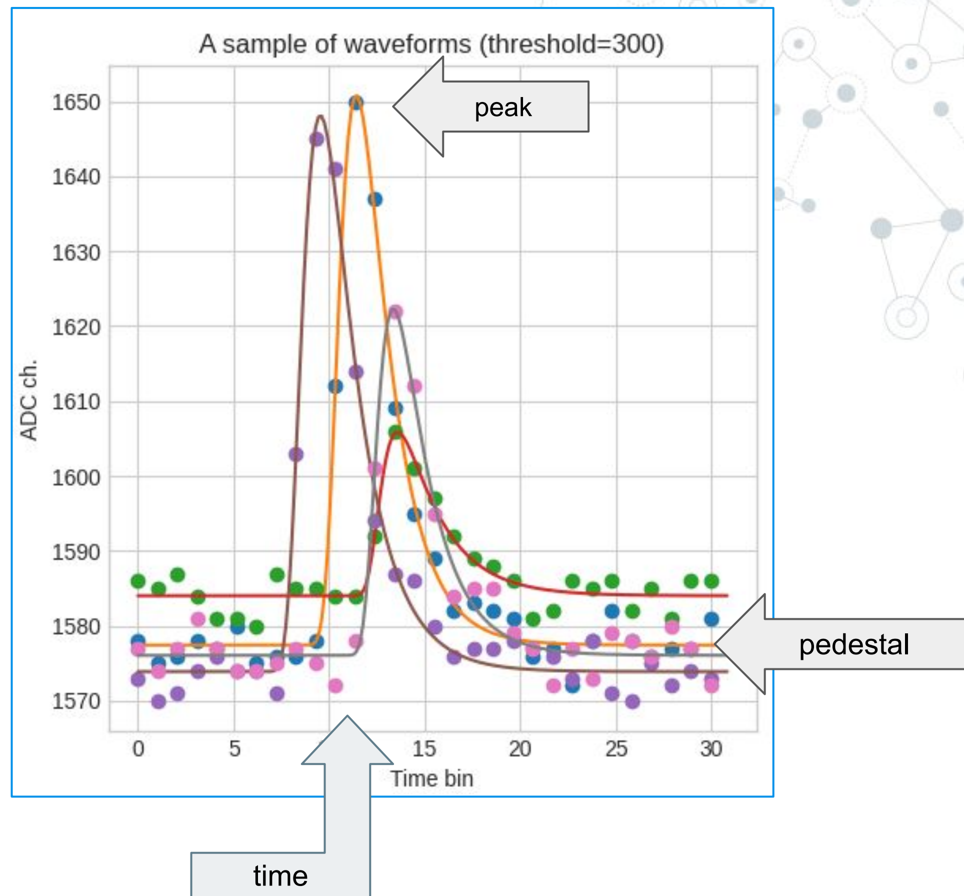






EMCAL: SiPM readout

- In the test beam experiment, the signal was digitized in 32 time bins in the test beam experiment, will be reduced to 16 bins in the final detector
- Three signal features need to be extracted from the **discrete** ADC data points:
  - Top signal amplitude
  - Time of the signal maximum
  - Pedestal
- This is typically done by fitting the data points with a suitable function
- The amplitude (above the pedestal) is the measure of the energy
- Timing information is necessary for building clusters



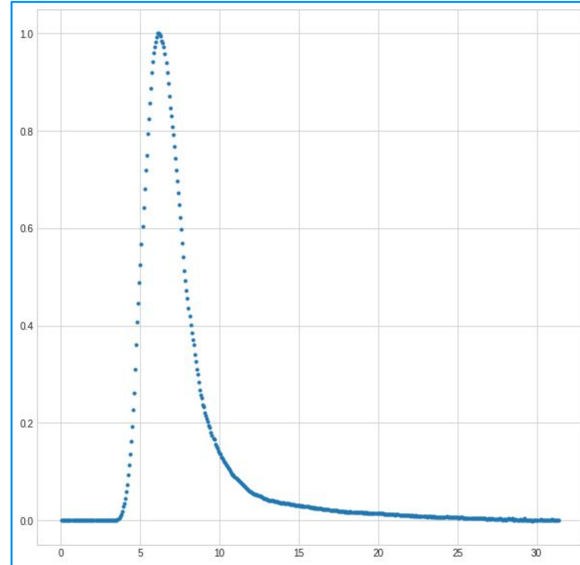
Feature extraction by fitting the signal

Method: estimate the waveform parameters by applying an appropriate fit.

- A variety of analytical and parameterized approximations
- Fitting with a "signal template" i.e. the average **normalized** discrete signal shape, with the three varying parameters used in the fit: (scale, time, pedestal)

The second option works well and was chosen as the baseline method. The initial benchmark time needed to fit all 24576 channels using this method was estimated as 1.4s at nominal detector occupancy.

An example of the signal template



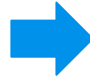
Motivation and approach

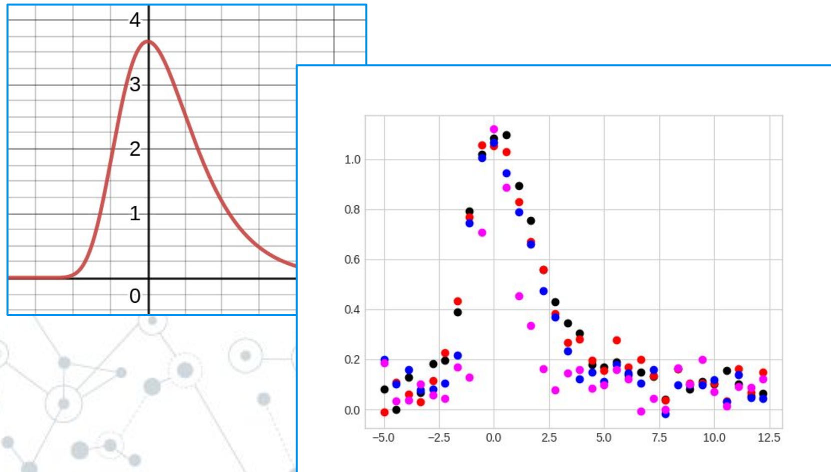
- The main motivation is speed, as the nominal 1.4s per event benchmark applies to the very first and basic step in event processing, and merits an effort to minimize that number
- In modern HEP experiments, use of Machine Learning techniques in calorimetry e.g. clustering is increasingly popular (cf. ATLAS, CMS)
- The ML application presented here aims to significantly speed up the process of signal feature extraction
- In the current study, the digitized waveform (ADC counts in each time bin) is fed directly into an artificial Neural Network as a vector
- Three outputs describing the signal: (scale, time, pedestal)

Three phases of the ML study

Simulated Signals (“MC Truth”)

- Estimate viability of the approach in controlled conditions, with variable noise, shape distortion and time jitter
- Gauge the initial performance gain vis a vis traditional fitting methods





Test Beam Data

- Train the ML model to reproduce the result of the conventional fit, using the data collected in the test beam experiment, with the actual hardware elements of the calorimeter
- Check precision and performance

Deployment and Integration

- Evaluate different methods of integrating the ML inference runtime into the sPHENIX software stack
- Consider: Python vs C++

The platform

- Keras/TensorFlow was chosen due to its user-friendly interface and learning curve, power and flexibility
- A suite of Python scripts, plus Jupyter notebooks for exploring data
- Separate training and validation data samples

dense_input: InputLayer	float32	input:	[(None, 32)]
		output:	[(None, 32)]

dense: Dense	float32	input:	(None, 32)
		output:	(None, 32)

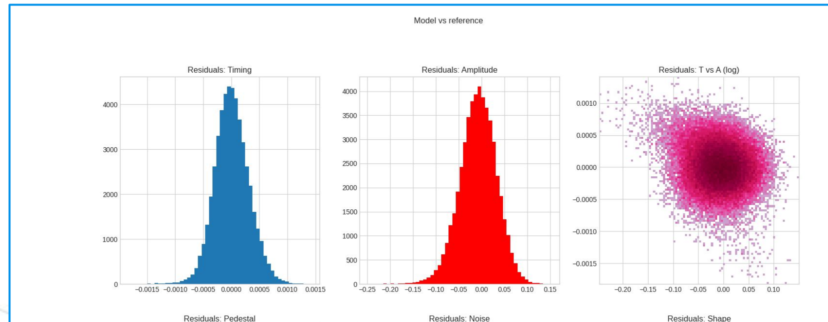
dense_1: Dense	float32	input:	(None, 32)
		output:	(None, 3)

```
# The (simple) model, dense NN
model = Sequential()
model.add(Dense(32, input_dim=L, activation='relu'))
model.add(Dense(3, activation='linear'))
model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])

# Inference requires just one function call
answer = model.predict(X, batch_size=batch)
```

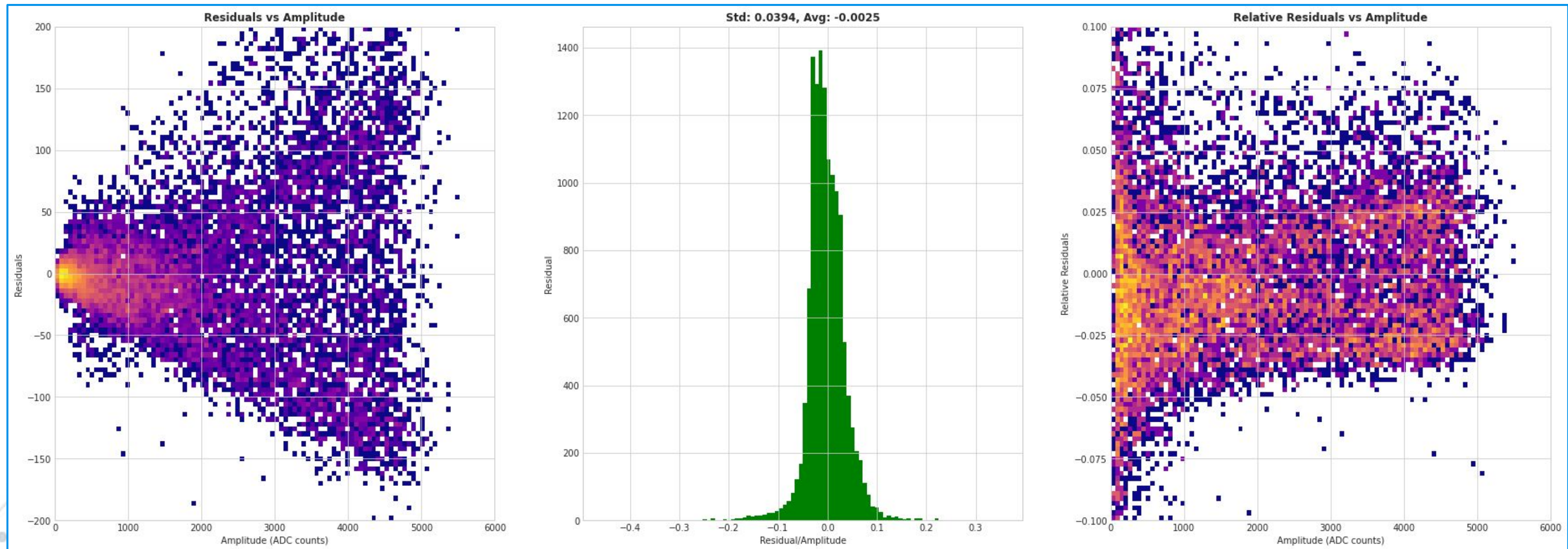
Initial results

- First results (with simulated signal shapes), performance when large batches: 3 μs inference time – per input vector – was achieved
- This is roughly 50 times faster than the conventional fit – of course this needed to be validated with real data
- Initial results on the accuracy of inference with respect to “MC truth” were also encouraging (at the percent level)
- This provided the motivation to apply ML to the existing test beam data



Moving on to real data (test beam)

- Using data collected at FNAL with a test electron beam in the energy range of (2..28) GeV. Focus on the residuals wrt to the conventional fit – one working example presented below:

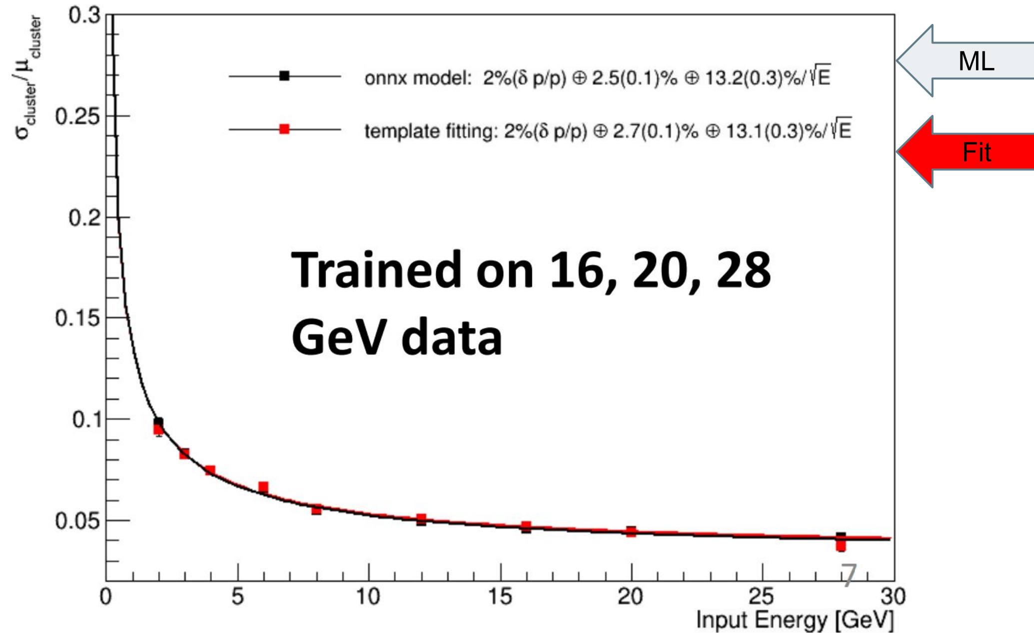


Lessons learned

- Normalization of inputs matters. It is helpful to have input values normalized to the same scale at least within an order of magnitude – e.g. having input tuples (0.0001, 10000.0) will not be optimal. This is a standard advice in the ML industry (while perhaps not entirely intuitive)
- Composition of the training sample matters, with respect to the dynamic range. Even when the problem is largely a linear one, the learning process appears to be sensitive to the ratios of the peak to pedestal and peak to noise. Covering all of the dynamic range has been shown to be important.

Precision/resolution wrt conventional fit

- By properly tuning the learning sample, it is possible to replicate the energy resolution achieved by the conventional fit



Deployment and Integration

The challenge

- Like most experiments, sPHENIX relies on a complex ROOT-based framework for reconstruction and analysis; in this study (with Keras) we relied on the TensorFlow backend
- Building TensorFlow for the C++ environment requires a specific tool (Bazel) and results in fairly large libraries

Option 1: serverize

- 1a: inference server
- 1b: microservices running on worker nodes
- Advantage: complete decoupling from the main software stack, flexibility
- Disadvantage: more moving parts

Option 2: use a compatibility layer

- Use a portable inference runtime, compatible with Keras or Pytorch-derived models
- Create an appropriate interface, evaluate ease of use and impact on software

1a, 1b: Inference service prototypes created and tested

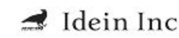
- 1a – NGINX/Django service:
  - problem-agnostic – given an appropriate model, can perform any type of inference based on the data received, not just signal fitting
  - Overall simplicity and low volume of the code
  - Performance comparable with the standalone Python applications, with network performance having a (negative) impact
- 1b – a Gunicorn/WSGI microservice
  - Tested on an actual HTCondor cluster at SDCC (BNL)
  - Overall simplicity and low volume of the code
  - Managing the population of running instances on a farm is an additional task, it's manageable but admittedly adds moving parts
- Scalability of either version would need to be addressed

Technology downselect

- After consideration, a decision was made to explore a more direct way of integrating ML inference into the sPHENIX software stack
- ONNX (Open Neural Network Exchange) is a good candidate, and has been used in HEP
- From the website:

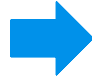
"ONNX is an open format built to represent machine learning models. ONNX defines a common set of operators - the building blocks of machine learning and deep learning models - and a common file format to enable AI developers to use models with a variety of frameworks, tools, runtimes, and compilers."

The ONNX Community



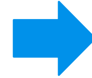
How this works

Train



Develop and train models using familiar tools (e.g. Keras or PyTorch)

Convert

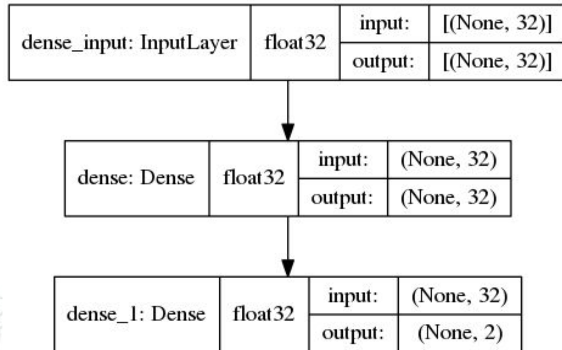


Convert the model to ONNX format

Run

Use any of the available runtime libraries/platforms compatible with ONNX – which are many

The logical choice is to use the ONNX runtime (NB. Used in ATLAS)



ONNX runtime binaries and source – availability

Optimize Inferencing	Optimize Training

Platform	Windows	Linux	Mac	Android	iOS	Web Browser (Preview)		
API	Python	C++	C#	C	Java	JS	Obj-C	WinRT
Architecture	X64	X86	ARM64	ARM32	IBM Power			
Hardware Acceleration	Default CPU	CoreML	CUDA	DirectML	oneDNN			
	OpenVINO	TensorRT	NNAPI	ACL (Preview)	ArmNN (Preview)			
	MIGraphX (Preview)	TVM (Preview)	Rockchip NPU (Preview)	Vitis AI (Preview)				
Installation Instructions	Please select a combination of resources							

Getting and interfacing the ONNX runtime

- C++ runtime is available in pre-built form (binaries) for download, 15MB total, trivial to build against (just need to define CPLUS_INCLUDE_PATH and LD_LIBRARY_PATH)
- Can also be built from source locally with minimal effort (tested)
  - Apparently pre-build/downloaded libraries perform better, which needs further investigation
- Because of the low-level nature of the ONNX framework (cf. compatibility) some C++ scaffolding and wrappers need to be created to have a functional interface – this adds a modest amount of code

ONNX C++ interface example: code snippets

```
// Transform the ROOT input data:
for (int i=0; i<N; i++) {
      Int_t m = branch->GetEntry(i);
      std::vector<int> inp;
      inp.insert(inp.begin(), std::begin(waveform[channel]), std::end(waveform[channel]));
      std::transform(inp.begin(), inp.end()-1, w31.begin(), [](int x)
      {return ((float)x)/1000.0;}); // Conversion to float and scaling
      input.insert(input.end(), w31.begin(), w31.end());
}

// Initialize input and output tensors (note multiple inputs)
inputTensors.push_back (Ort::Value::CreateTensor<float>(oS->_memoryInfo, input.data(),
N*31,   inputDimsN.data(),  inputDimsN.size()));
outputTensors.push_back(Ort::Value::CreateTensor<float>(oS->_memoryInfo,
outputTensorValuesN.data(), N*3,    outputDimsN.data(), outputDimsN.size()));
// Run inference
oS->session->Run(Ort::RunOptions{nullptr}, inputNames.data(), inputTensors.data(), 1,
outputNames.data(), outputTensors.data(), 1);
// NB. Boilerplate contained in the header files
```

ONNX runtime: precision and performance

- Native Keras, ONNX Python runtime and ONNC C++ runtime all produce same numerical results. Precision of the ONNX inference has been demonstrated with the test beam data
- Similar to the initial Keras experimentation, batching the data for inference leads to increase in speed per input vector — inference time well under 1 μs was achieved
- Reason for that is memory allocation and other fixed costs
- …so one possible scenario includes processing data from all the channels in the calorimeter in one function call and getting processing done on a millisecond scale

Status and Plans

- Integration of the ML-enabled software into the overall calorimeter data processing workflow is work in progress, close to completion
- In a test with realistic data, and using the sPHENIX software framework, the estimated gain in signal processing performance when using ML inference is at least 2 orders of magnitude, compared with the fastest signal fitting method previously used: estimated at 12 ms/event for ALL of the 24576 calorimeter channels
- An initial study is underway to explore the possibility of applying ML to other aspects of the calorimeter data analysis, such as discriminating π^0 vs γ based on the shower profile