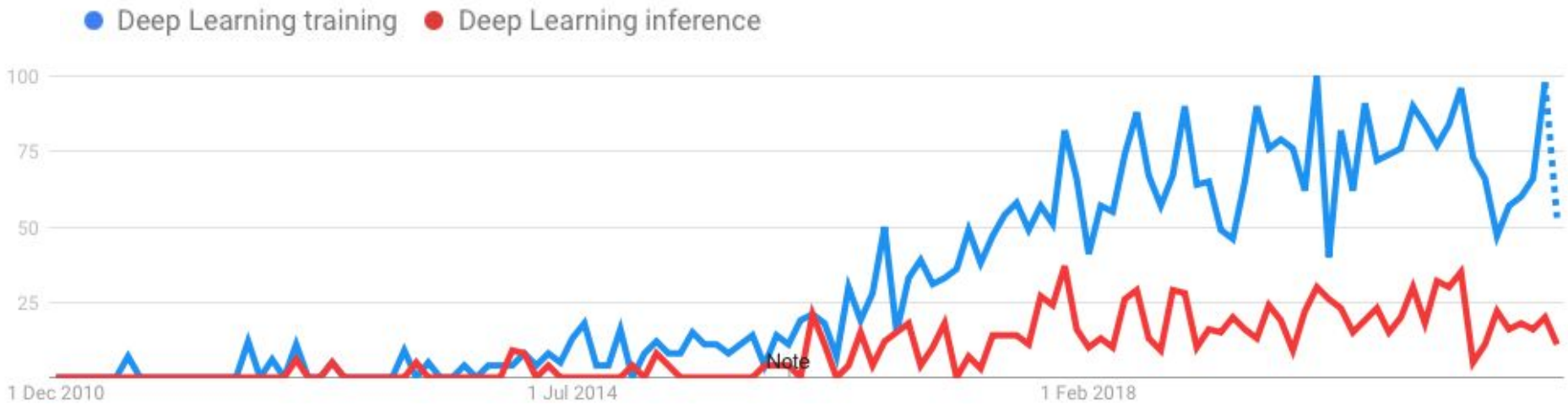
# C++ Code Generation for Fast Inference of Deep Learning Models in ROOT/TMVA

Sitong An for the ROOT team
s.an@cern.ch

ROOT
Data Analysis Framework
https://root.cern

# Overview of the ML Ecosystem

- Heavy focus on algorithm development/training
- Deployment/inference in production environment often neglected
  - “Inference” = application of trained model on actual data



# How to do inference

- Tensorflow/PyTorch offers inference functionalities, but…
  - Can only run inference on its own models respectively
  - Heavy dependencies
  - Usage in production environment in C++ is cumbersome

- ONNX (“Open Neural Network Exchange”)
  - Aims to define a common standard for describing DL models

ONNX

# ONNX Runtime

- ONNX Runtime project (since early 2019)
- Pro: Powerful, efficient inference engine backed by Microsoft
- Con:
  - Relatively large dependency
  - C++ API still in experimental stage
  - Difficult to integrate into HEP ecosystem (Much effort spent in CMSSW ONNXRuntime integration)

# Idea for Inference Code Generation

- An inference engine that…
  - Input: trained ONNX model file
    - Supported by PyTorch natively
    - Converters available for Tensorflow and Keras
  - Output: Generated C++ code that hard-codes the inference function
    - Easily invokable directly from other C++ project (plug-and-use)
    - minimal dependency (on BLAS/eigen only)
- …as part of the modernisation for TMVA

# Code Example

- Transpose Operator
- Example: Transpose a tensor with shape [1,2,3,4] by permutation [3,2,1,0]
  - I.e. the output tensor will have a shape of [4,3,2,1]
- Allows the compiler to do automatic loop unrolling
- Automatic generated code:

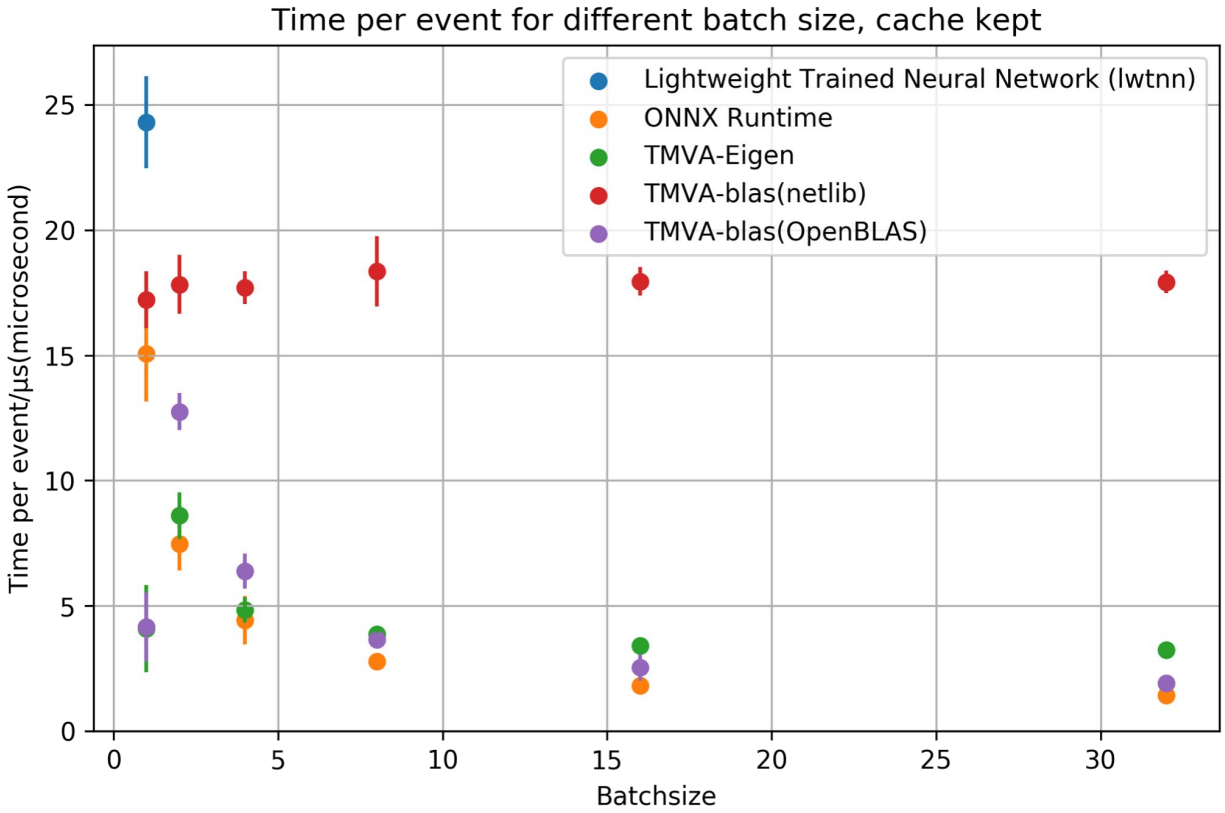
```
for (int id = 0; id < 24 ; id++){
        tensor_2[id / 24 + id / 12 % id / 4 % 3 * 2 + id % 4 * 6] = tensor_1[id];
}
```

# Code Example

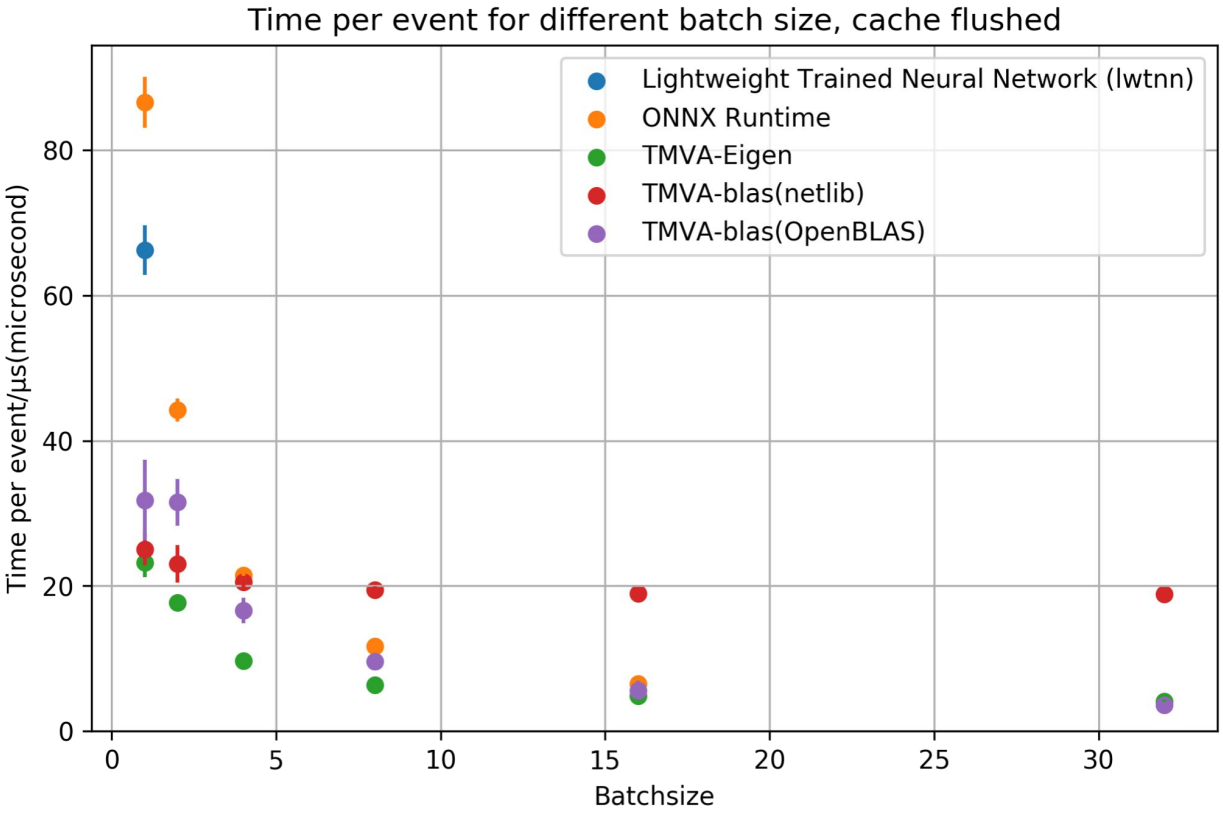
- Gemm (matrix multiplication) Operator
- Essential for dense layer and convolutional operations
- Example: Gemm Operator
- Automatic generated code for netlib BLAS:

```
BLAS::sgemm_(&op_1_transB, &op_1_transA, &op_1_n, &op_1_m, &op_1_k, &op_1_alpha, tensor_4,
&op_1_ldb, tensor_3, &op_1_beta, tensor_5, &op_1_n);
```

# Benchmark


Time per event for different batch size, cache kept
Lightweight Trained Neural Network (lwtnn)
ONNX Runtime
TMVA-Eigen
TMVA-blas(netlib)
TMVA-blas(OpenBLAS)
Time per event/µs(microsecond)
Batchsize
0
5
10
15
20
25
30

# Benchmark


Time per event for different batch size, cache flushed
Lightweight Trained Neural Network (lwtnn)
ONNX Runtime
TMVA-Eigen
TMVA-blas(netlib)
TMVA-blas(OpenBLAS)
Time per event/µs(microsecond)
Batchsize
0
20
40
60
80
0
5
10
15
20
25
30

# Next Steps

- Further code optimisation for event-loop inference and large batchsize
- Support for conv operator implemented, currently validating
- Proposed work on RNN operators (LSTM, GRU) and popular regularization operators (BatchNorm) this summer
- Expected to support inference of most of the popular DL architectures this summer

- Further integration with ROOT
  - interactively-run inference via JITting with Cling
  - Integration with ROOT RDataframe

# Conclusion

- Link to development prototype
- Link to benchmark sample code
- Link to current ROOT PR


The presenter gratefully acknowledges the support of the Marie Skłodowska-Curie Innovative Training Network Fellowship of the European Commission Horizon 2020 Programme, under contract number 765710 INSIGHTS.

# Benchmark Details

- A sequential network of 10 Dense layers of width 50, activation functions ReLU
- All inferences are instructed to be single-threaded only
- Each data point is produced from 1000 inference runs
- To flush the cache, a large vector (~42 Mb) of random numbers on the stack is written to after each inference run

- Benchmarked on CERN Openlab Machine running CentOS 8
- CPU: Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz
- Cache size: 35 Mb

# Interface

- Currently in namespace TMVA::Experimental::Sofie
- Expected to be ready for experimental release soon
- Proposed interface:

```
RModelParser_ONNX parser;                //dependency on protobuf only on the parser
RModel model = parser.Parse("./LinearNN.onnx");          //decoupled from the parser.
// Rmodel is an Intermediate representation that can be serialised into a ROOT file
model.Generate("./LinearNN.hxx");
```