PRINCETON UNIVERSITY

Training RNNs with half-precision floats

Alexey Svyatkovskiy[1], Julian Kates-Harbeck[2], William Tang[1]

[1]Princeton University

[2]Harvard University

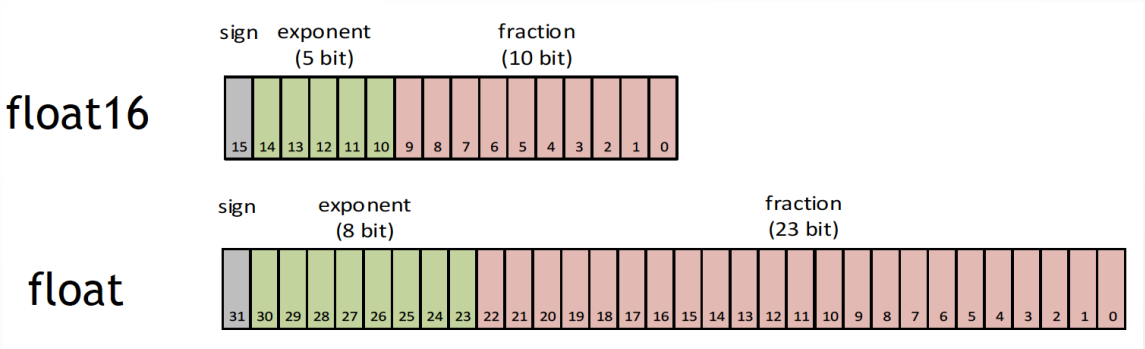
Outline

- Introduction
  - Half-precision float training: motivation and challenges
- The physics problem we are trying to solve
  - Disruption forecasting in tokamak fusion plasmas
- Fusion Recurrent Neural Net (FRNN) framework
  - Distributed data-parallel synchronous SGD
  - Network architecture
- Enabling FP16 training
  - Master copy of weights, Loss scaling
  - Hardware specifications
  - FP16 performance comparisons for FRNN
- FRNN strong scaling
  - Results on OLCF Titan
  - FP16 results on Princeton University Tiger cluster
- Conclusions

Half-precision training: motivation and challenges (I)

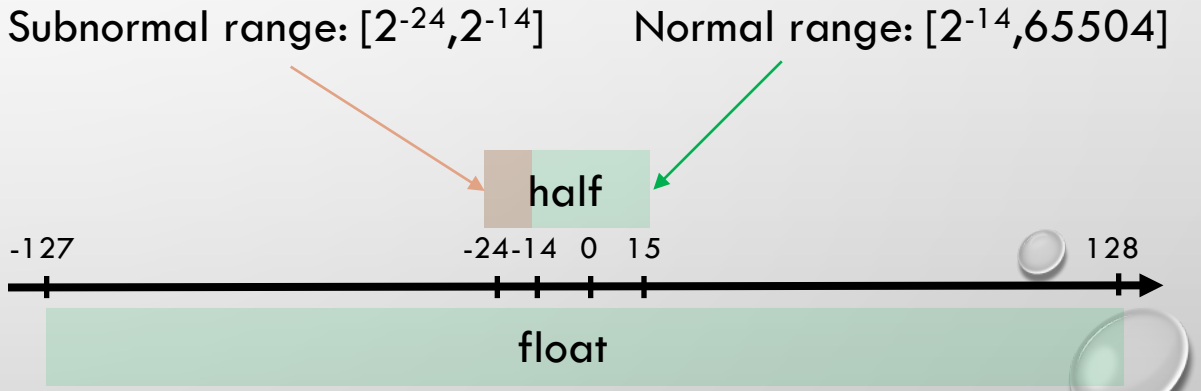
Motivation

- Optimize memory bandwidth
  - Ability to train models with 2x parameters
- Optimize network bandwidth
- Faster training
  - Improve maximum computational throughput
  - Ability to train with larger mini-batch size



Challenges

- IEEE 754 FP16 has a narrow numerical range
  - Overflow problem: Inf/NaN cause irreversible damage to training (*quite rare*)
  - "Vanishing gradient" (*quite common*)
    - Small valued gradients
    - Large ratio of weight to weight gradient



Physics problem we are trying to solve

- Fusion energy: a clean source of energy that can be produced on Earth in a tokamak-style fusion reactor
- Tokamak: a type of design for a nuclear fusion device. It is a torus-shaped vacuum chamber surrounded by magnetic coils
  - Existing tokamak-style fusion reactors: JET, DIII-D, NSTX
- Plasma disruptions are large macroscopic instabilities resulting in:
  - Loss of confinement – ends fusion reaction
  - Intense radiation – damaging concentration in small areas
  - Current quench – produces high magnetic forces

- The prediction and avoidance of disruptions has been proven to be unavoidable aspect of tokamak operation, especially in high performance regime

- ITER is the key experimental step between today's fusion research machines and tomorrow's fusion power plants
  - **ITER cannot tolerate disruptions at maximum current**

Datasets: JET

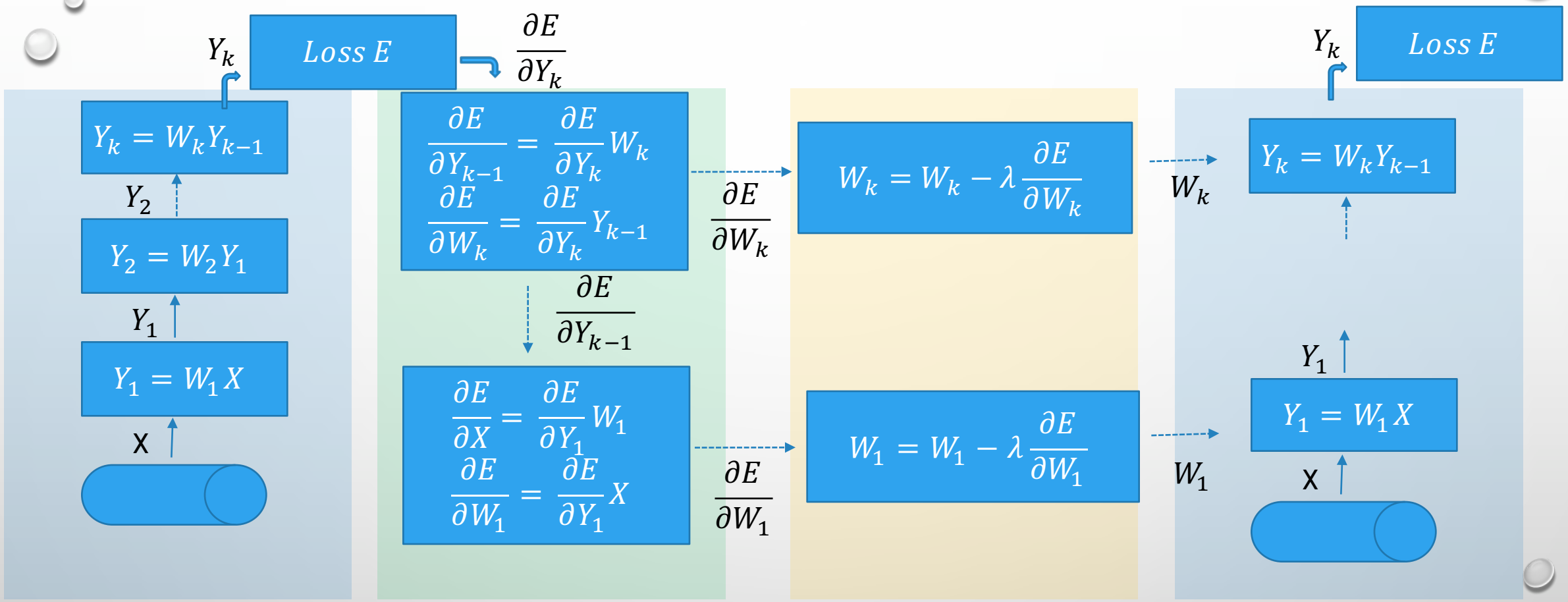
- **Test on a real scientific dataset from Joint European Torus (JET) experiment**
- The JET dataset consists of time series containing multi-modal sensory measurements including scalar and 1D arrays, collected with a sampling rate of 1 ms
  - 10% of shots end with disruption
- JET experiment produces order of Terabyte of data per day
  - 55 GB per shot
  - Over 350 TB with multi-dimensional time traces to be analyzed

# Shots	Disruptive	Non-disruptive	Totals
Carbon Wall	324	4029	4353
Beryllium Wall (ITER-like wall)	185	1036	1221
Totals	509	5065	5574

- Repeat test on the Large Movie Review Dataset (IMDB), which is a benchmark dataset in machine learning community

Sample scalar time series	
Plasma Current	I_p
Locked mode amplitude	LM
Plasma Density	n
Radiated Power	P_{rad}
Total Input Power	P_{in}
d/dt Stored Diamagnetic Energy	$\frac{\partial E_{int}}{\partial t}$
Plasma Internal Inductance	L_i
Safety factor	q_{95}
Stored Diamagnetic Energy	E_{int}

Training flow (I)



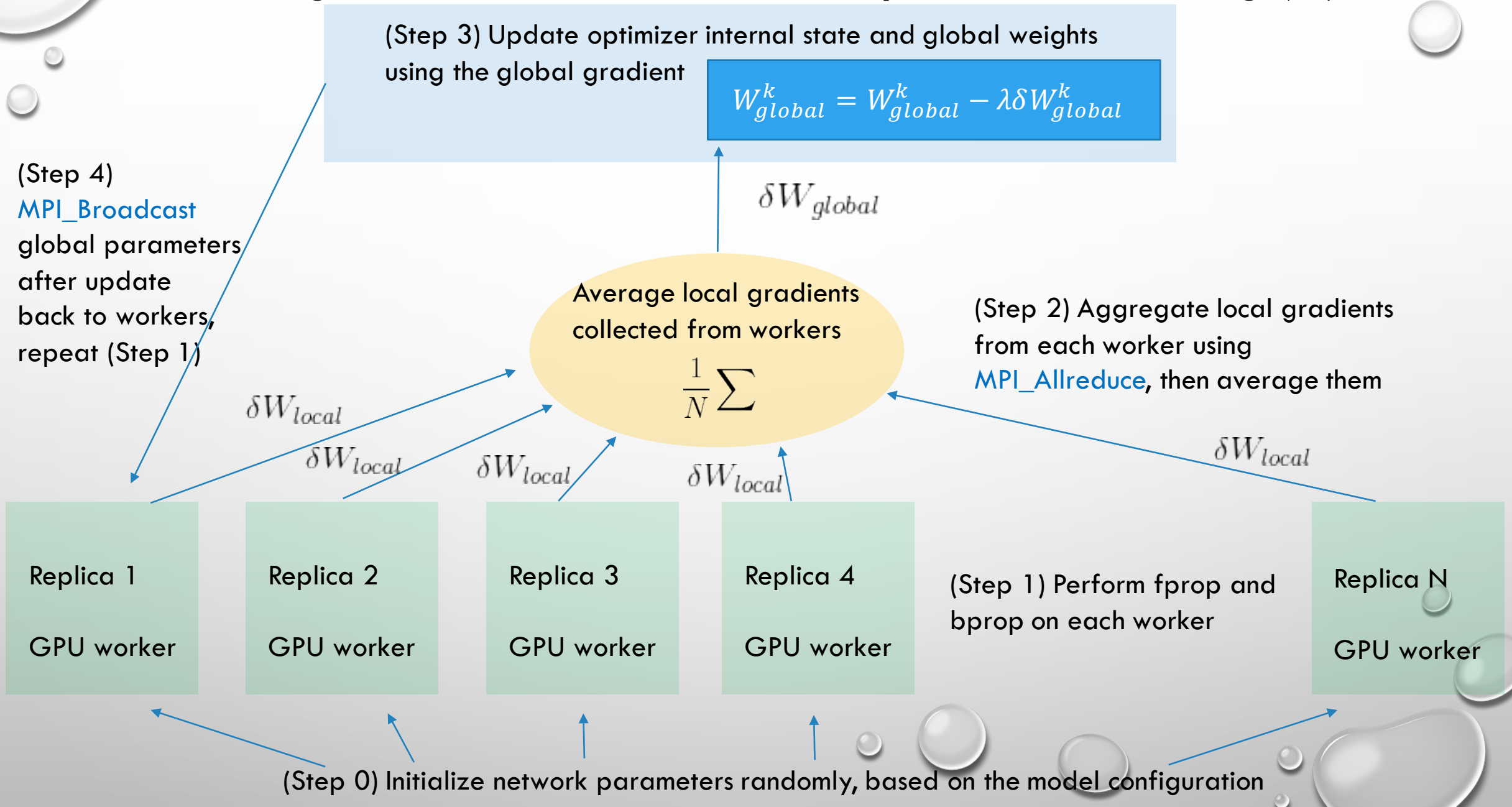
Forward pass: mostly matrix and element-wise multiplications

Backprop: chain-rule differentiation, element-wise multiplications

Optimizer weight update: multiply gradients by learning rate (e.g. SGD)

Forward pass on the next mini-batch of data…

Training flow: distributed data parallel training (II)



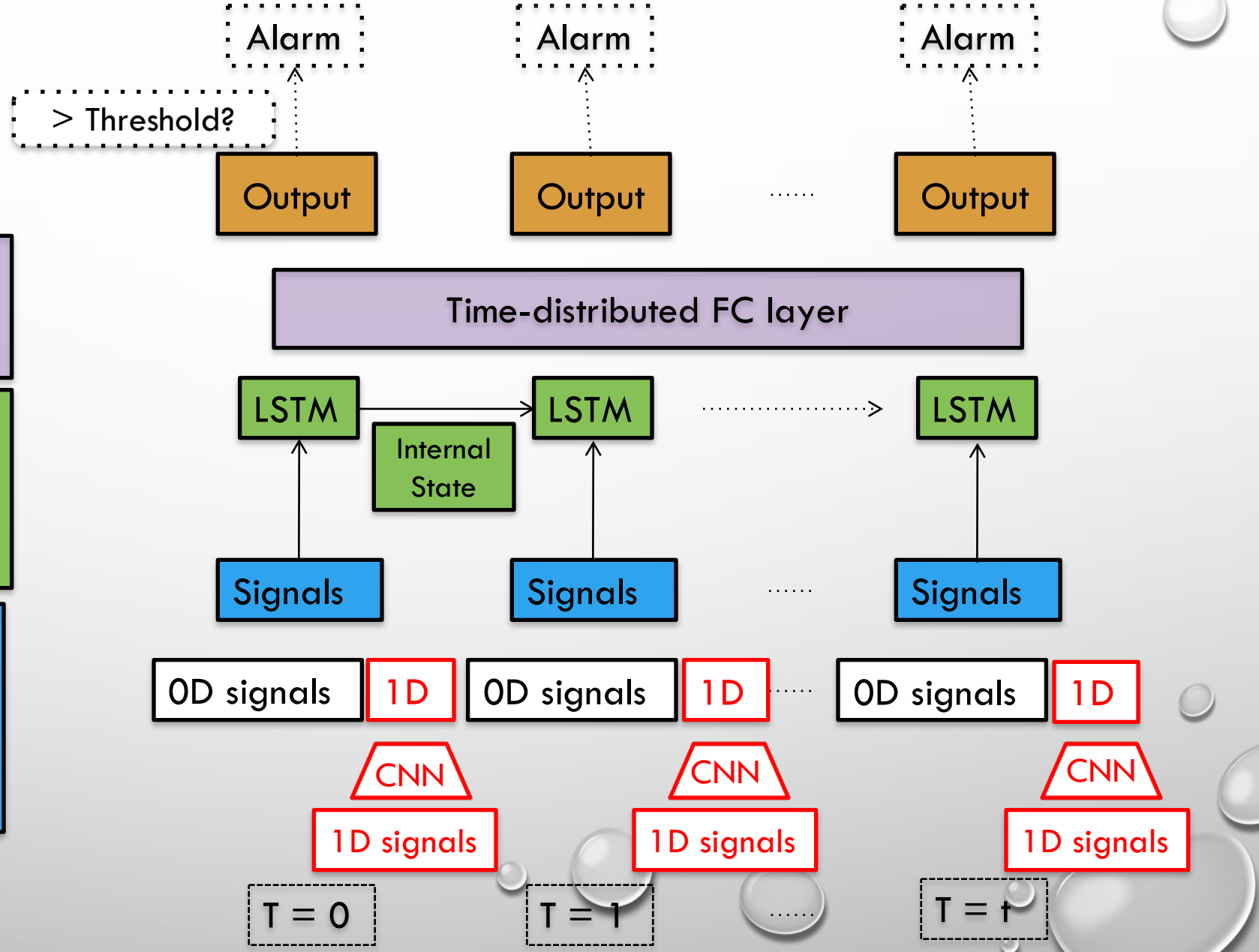
Fusion Recurrent Neural Net (FRNN) schematic

Output: Disruption coming?

Time-distributed FC layer
- apply to every temporal slice on LSTM output

RNN Architecture:
- LSTM, 3 layers
- 300 hidden units per cell
- Stateful, returns sequences

CNN architecture:
- Number of convolutional filters: 10
- Size of convolutional filters: 3
- Number of convolutional layers: 2
- Pool size: 2



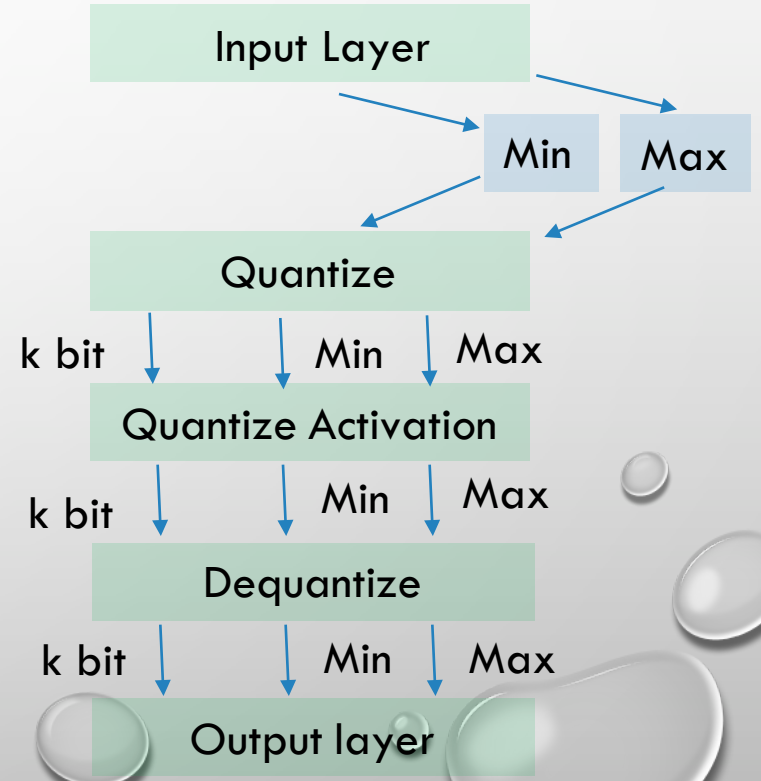
Enabling Float16 training (I)

Master copy of weights

- Store weights, activations and gradients in FP16, perform weight update in FP32
  - Solves "vanishing gradient" problem during
  - Disadvantage: conversion between FP32 and FP16 may be slow; extra memory to keep an extra copy of weights in FP32

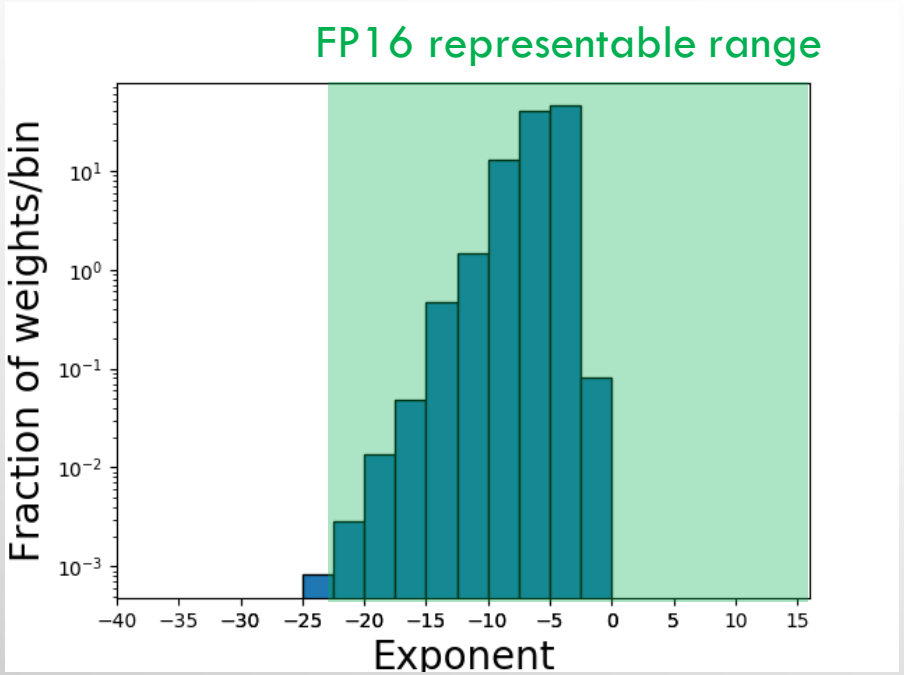
Weight quantization

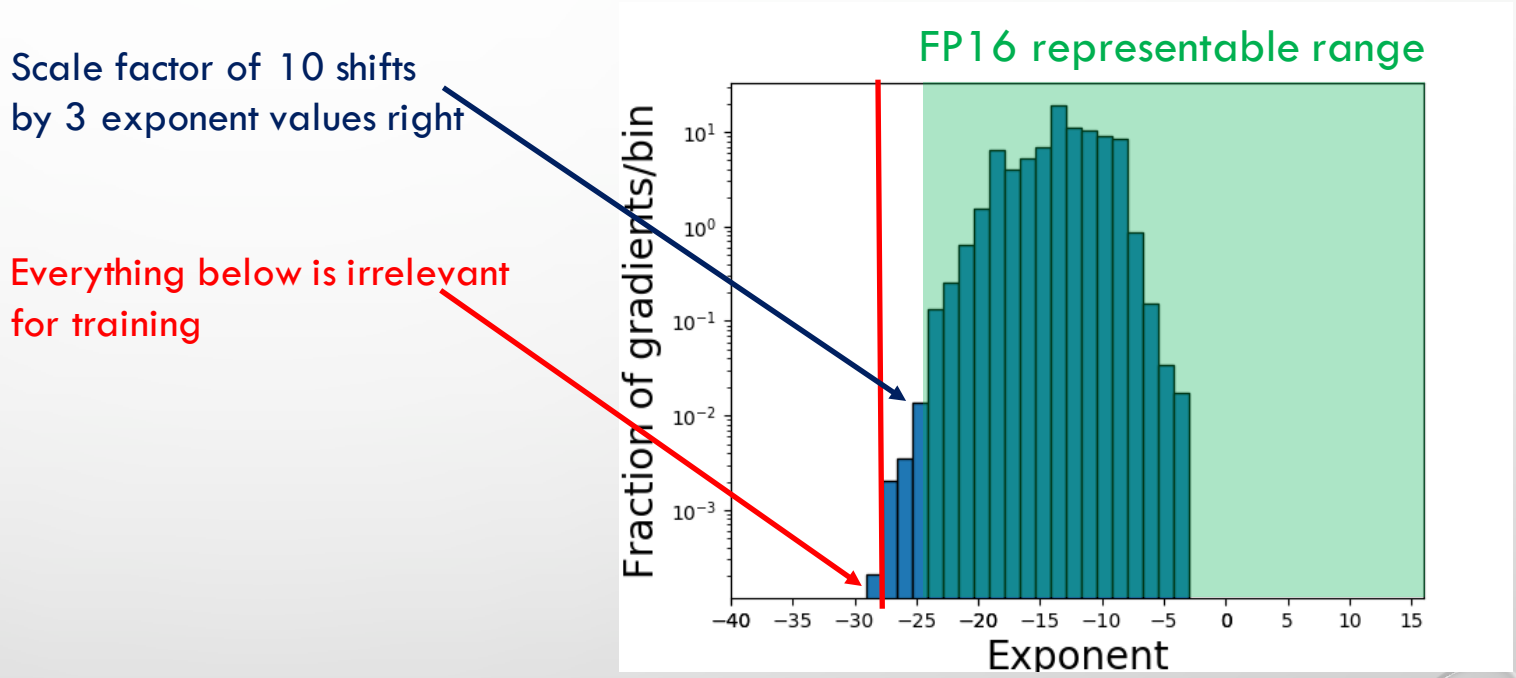
- Quantization is an umbrella term that refers to a set of data compression techniques
  - E.g.: binarization: encode data in k=2 bits
- Ex. quantization compression approach for a layer:
  store the min and max for each layer, then compress each float value to a k-bit (k < 32 bits) integer representing the closest real number in a linear set of 2^k within the range
  - All of these approaches leave the gradients unmodified in single-precision and therefore the computation cost during bprop is unchanged
  - Can't use regular optimizer as derivatives are often zero



Enabling Float16 training (II)

- **Loss scaling:** shift gradients to occupy higher bins of FP16 representable range, which are not used
  - Apply a scalar factor to the loss function before backpropagation step
    - Unscale weight gradients before the update step to maintain the magnitude of updates the same as for FP32; alternatively re-tune the hyperparameters

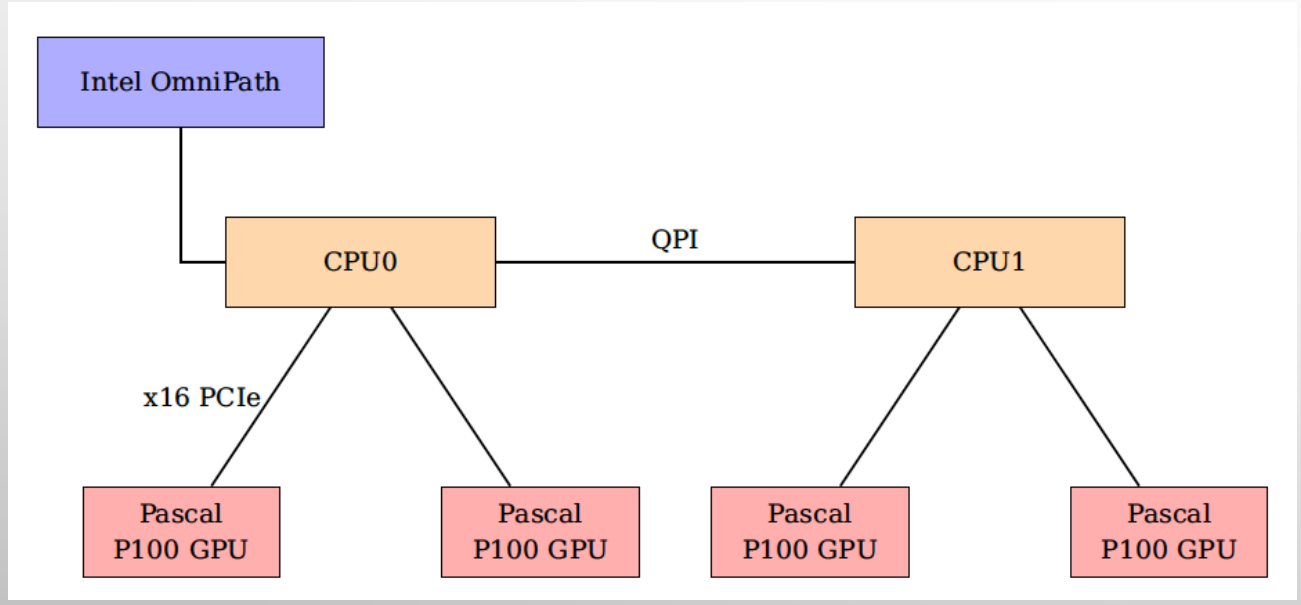




- Scaling factors will depend on the neural network/dataset in hand
  - FRNN training on JET dataset required loss scale factor of 10 (i.e 3 exponent values to the right)

Hardware specifications

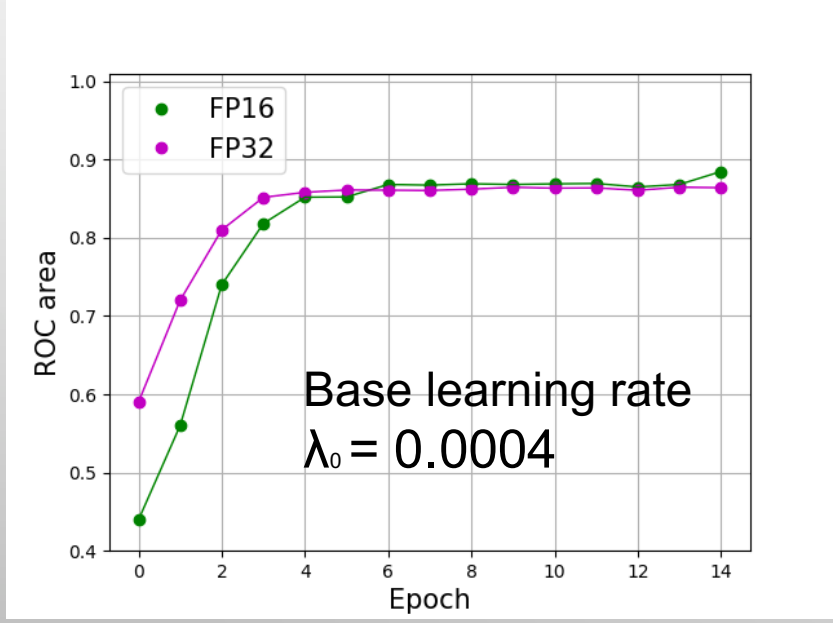
- Scaling tests are performed on Princeton University “Tiger” cluster
  - Theoretical peak performance of 27 petaflops
  - 320 NVIDIA Tesla P100 GPUs PCIe across 80 Intel Broadwell nodes
  - Nodes on the cluster are interconnected by Intel OmniPath high-speed interconnect
  - GPU Direct, PSM2 10.3.3
  - SLURM scheduler

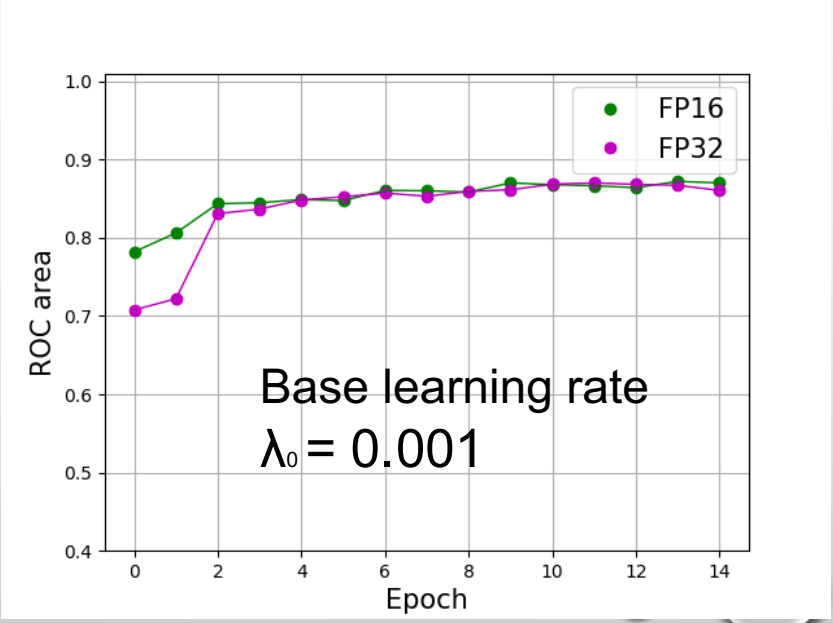


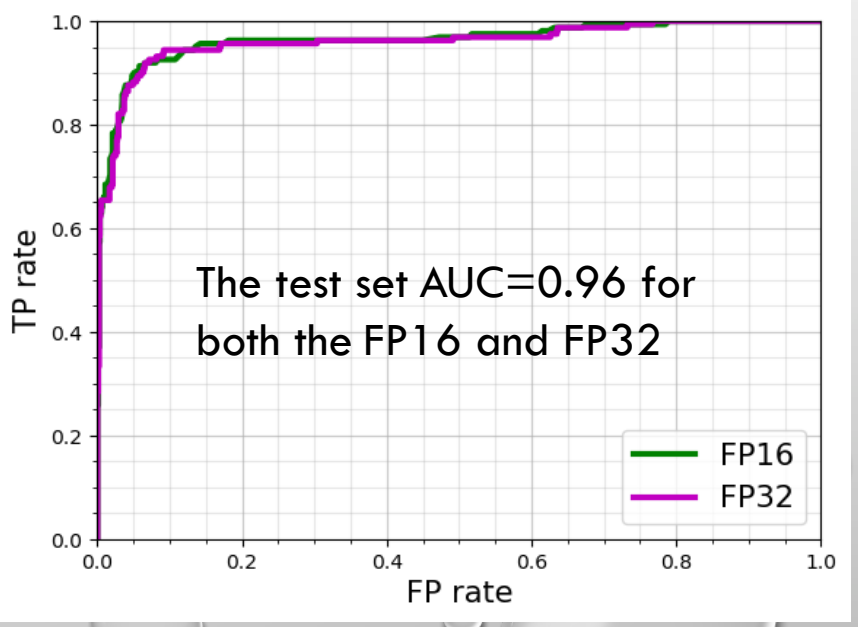
- NVIDIA GPUs with compute capability 5.3 and later support FP16 math, arithmetics and comparisons
- The latest generation of Intel CPUs (e.g. Haswell, Broadwell) provide a capability for converting between single and half in hardware by means of the F16C instruction set

Performance comparison

- Use validation level area under the ROC and AUC (area under curve) to characterize the quality of FRNN and applicability of the half-precision training
- Validation level AUC as a function of epoch show similar shapes for both FP16 and FP32
  - Reaching the plateau at around AUC=0.87 by the epoch 6
  - SGD optimizer with momentum, loss scaling factor=10







Data parallel training: time per epoch

$$T \sim \frac{1}{N}\left(A + B\ log(N)\right) = O\left(\frac{log(N)}{N}\right)$$

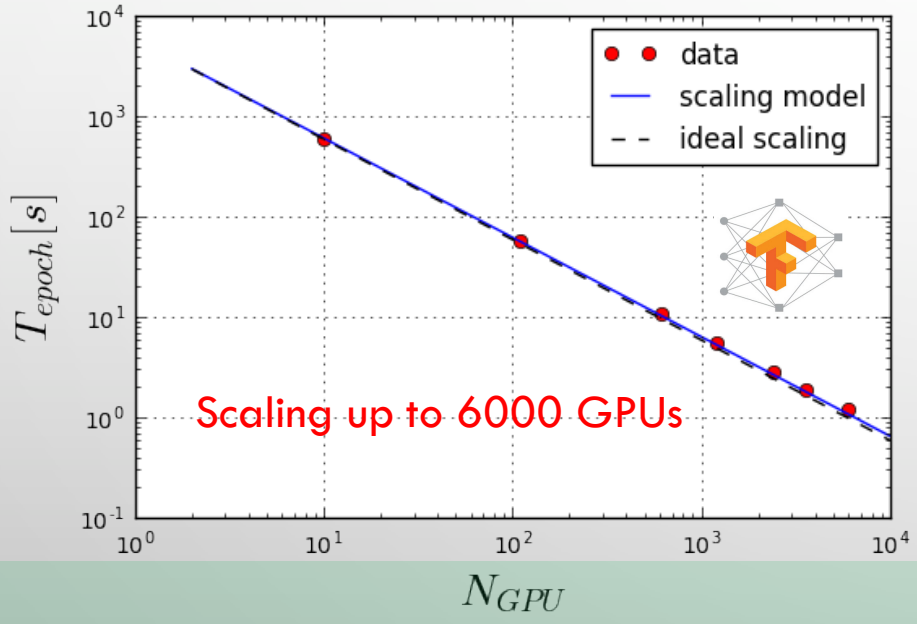
Number of batches

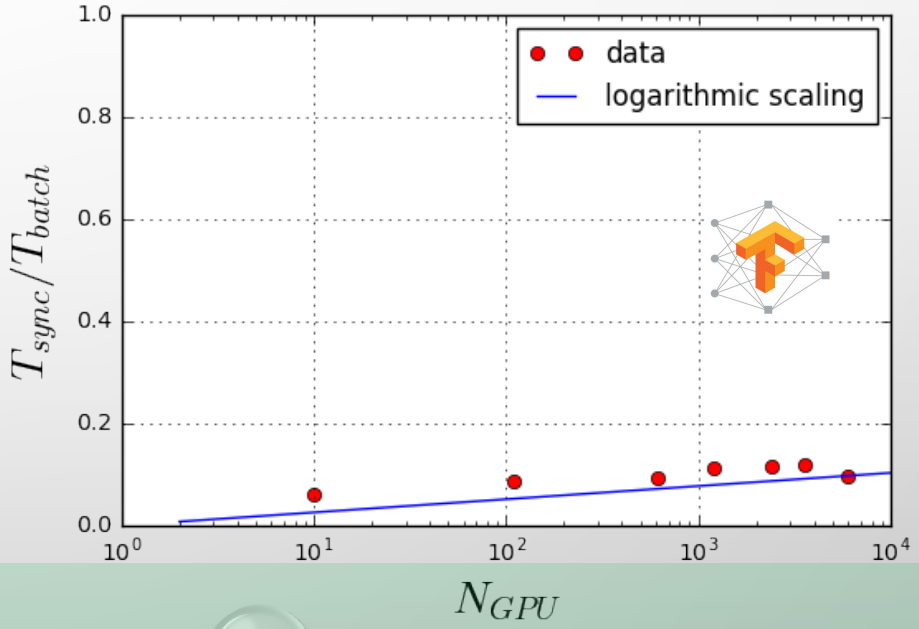
Compute time per epoch is constant

MPI communication per mini-batch is logarithmic

FRNN scaling (I)

- Our goal is to develop portable ML software for plasma disruption forecasting that would allow for rapid training on GPU clusters
- Tests on OLCF Titan CRAY supercomputer
  - Strong linear runtime scaling and logarithmic communication time
  - **OLCF Director's Discretionary Award**: Scaling Studies on Titan
  - Thousands of Tesla K20 GPUs

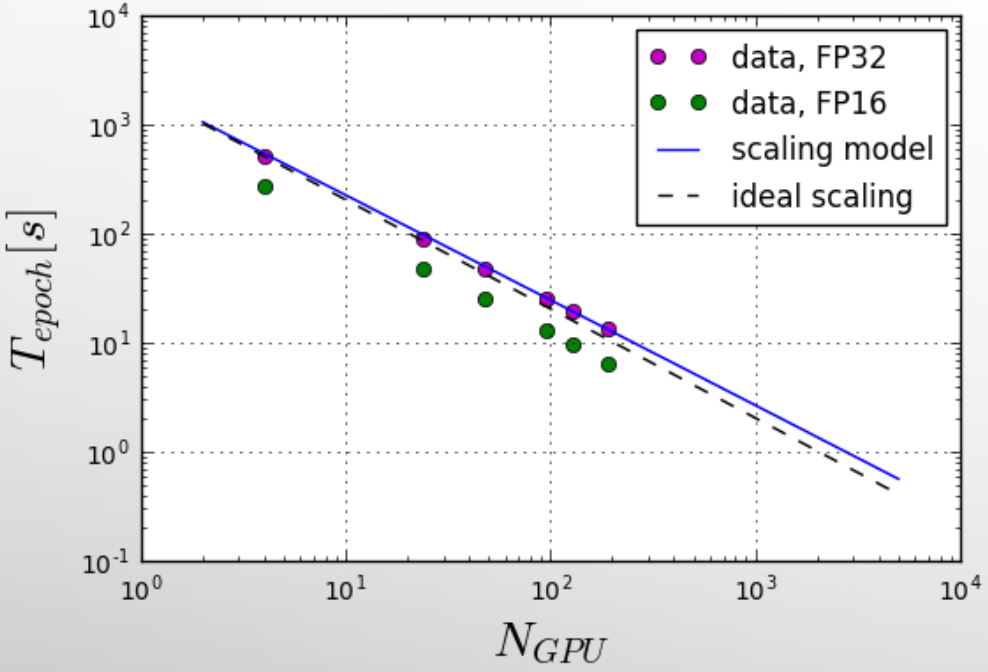




Manuscript in preparation: **"Disruption Forecasting in Tokamak Fusion Plasmas using Deep Recurrent Neural Networks", J. Kates-Harbeck, A. Svyatkovskiy, K. Felker, E. Feibush, W. Tang, *2017***

FRNN scaling (II)

- Strong linear runtime scaling and logarithmic communication complexity hold for F16 training
- Possibility for training with larger mini-batches with FP16: double mini batch size keeping for the same memory bandwidth the same



Roughly 2x faster processing times per epoch with FP16 due to increased mini-batch size

Precision	Npar (million parameters)	Nlayers	Batch size
FP64	4.6	15	256
FP32	9.2	29	256
FP16	18.2	58	256
FP64	18.2	58	64
FP32	36.3	118	64
FP16	72.1	234	64

Maximum number of trainable parameters, batch size, and equivalent model depths fitting in Tesla P100 GPU device memory in half-precision, single and double floating point precision

Summary and Next steps

- Developed a deep learning framework integrating TensorFlow with custom parameter averaging and global weight update routines implemented with CUDA-aware MPI intended for disruption forecasting in tokamaks
  - Distributed data-parallel synchronous SGD approach with strong nearly linear runtime scaling on GPU clusters
  - Learning rate scheduling approach to facilitate model convergence when training on HPC clusters with O(100) worker GPUs
- Evaluated training of deep RNNs with half-precision floats on that framework
  - Use scientific JET plasma disruption time series dataset
  - Cross check with benchmark IMDB dataset
  - Use loss scaling approach to facilitate model convergence at FP16
- Obtained comparable validation level performances at the end of each epoch for half and single floating point precisions
- FP16 optimizes memory and network bandwidths, allowing the training of models with over 70 million trainable parameters
  - Ability to use large mini-batch sizes
- Look forward for opportunities to deploy FRNN on Volta GPU to take advantage of Tensor Cores

Backup

Half-precision training (I)

Half-precision training:

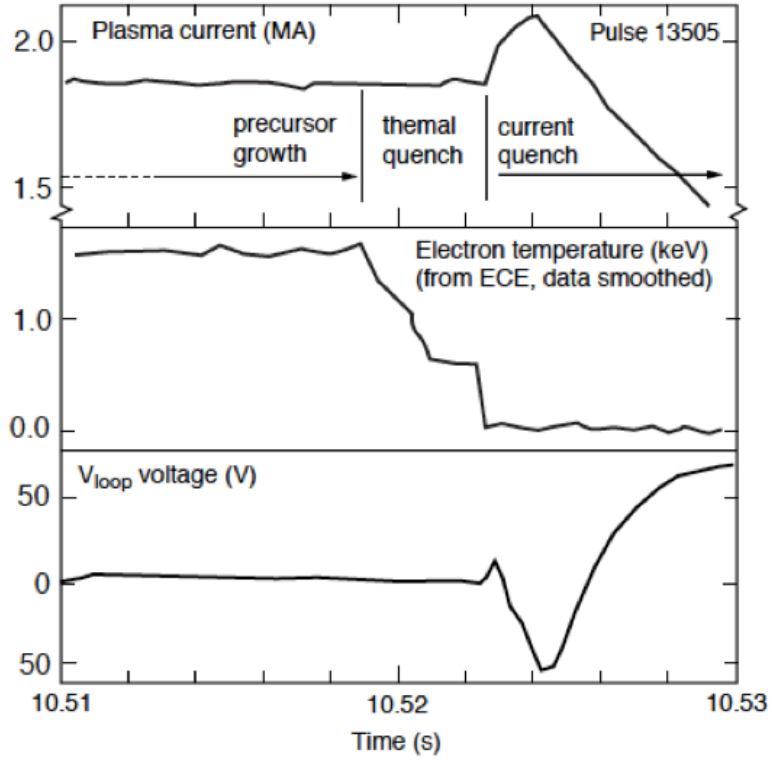
- Store weights, activations and gradients in FP16
- Perform matrix and element-wise multiplications, reduction ops in FP16 (including during fprop, bprop and weight update)

- You already know it is possible to perform FP16 math, arithmetics and comparisons on NVIDIA GPUs with compute capability 5.3 and greater
  - CUDA half intrinsics (round to nearest even): http://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__HALF.html#group__CUDA__MATH__INTRINSIC__HALF
  - TensorFlow has DT_HALF registered type: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/types.cc
  - We add a custom MPI datatype of 2 contiguous bytes: https://github.com/PPPLDeepLearning/plasma-python/blob/master/plasma/primitives/ops.py

- **This talk:**
  - Review techniques to enable FP16 and mixed precision training
  - Introduce FRNN framework for disruption forecasting in tokamak fusion plasmas , discuss strong scaling
  - Evaluate FP16 training on FRNN framework and a real scientific dataset from JET experiment
  - Evaluate FP16 training on the benchmark IMDB dataset

Plasma disruption characteristics

- Plasma disruptions are large macroscopic instabilities resulting in:
  - Loss of confinement – ends fusion reaction
  - Intense radiation – damaging concentration in small areas
  - Current quench – produces high magnetic forces
- Time scale: milliseconds
  - Need at least 30 ms warning to mitigate – rapid forecasting is necessary
- Consequences: more severe with higher volume-to-surface area ratio



- The prediction and avoidance of disruptions has been proven to be unavoidable aspect of tokamak operation, especially in high performance regime

FRNN package structure

- Python deep learning code for disruption prediction in fusion (tokamak) experiments
  - https://github.com/PPPLDeepLearning/plasma-python

Primitives

Preprocessing

Models

Utils

Abstractions specific to the domain
Shots, ***Machines*** and ***Signals***

Preprocessing and normalization classes, including the methods necessary to prepare physical data for stateful LSTM training

Python classes necessary to build, train and optimize deep NN models. Including a distributed data-parallel synchronous implementation of optimizer. FRNN integrates Tensorflow with CUDA-aware MPI for communication, enabling the use of high-speed interconnects on the cluster (OmniPath) and the GPUDirect technology

Auxiliary functions for preprocessing, performance evaluation and learning curves analysis.

•Dependencies:

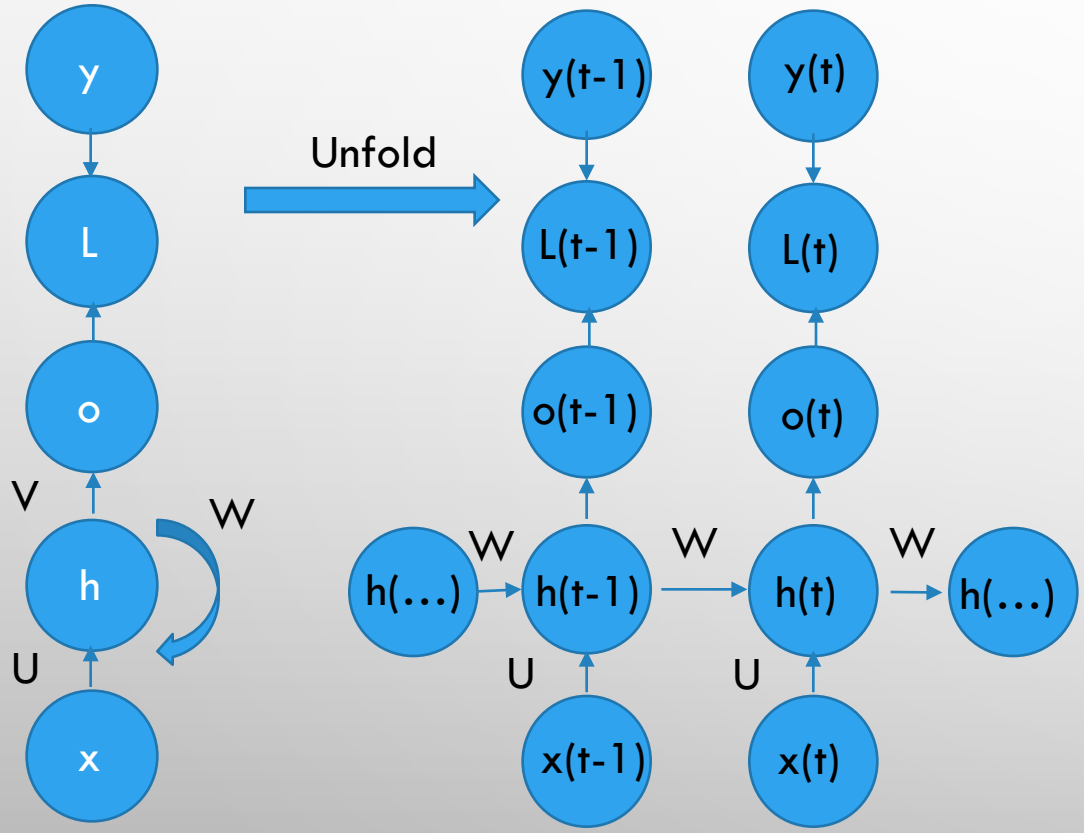
–Tensorflow 1.3, Keras 2.0.6, OpenMPI 2.1.0, CUDA 8, CuDNN 6

–Pathos

Recurrent Neural Nets: basic description

- RNNs are a family of neural networks to process sequential data
- Feed forward equations are recurrent:

$$a(t) = b + Wh(t-1) + Ux(t)$$
$$h(t) = \tanh\big(a(t)\big)$$
$$o(t) = c + Vh(t)$$



Notations:

x – input sequence,
U – is the input to hidden weight matrix,
W - hidden to hidden,
V – hidden to output weights
b,c are the biases
tanh() is the activation function (non-linearity)
o – output sequence
Loss **L** and target values are denoted as **y**

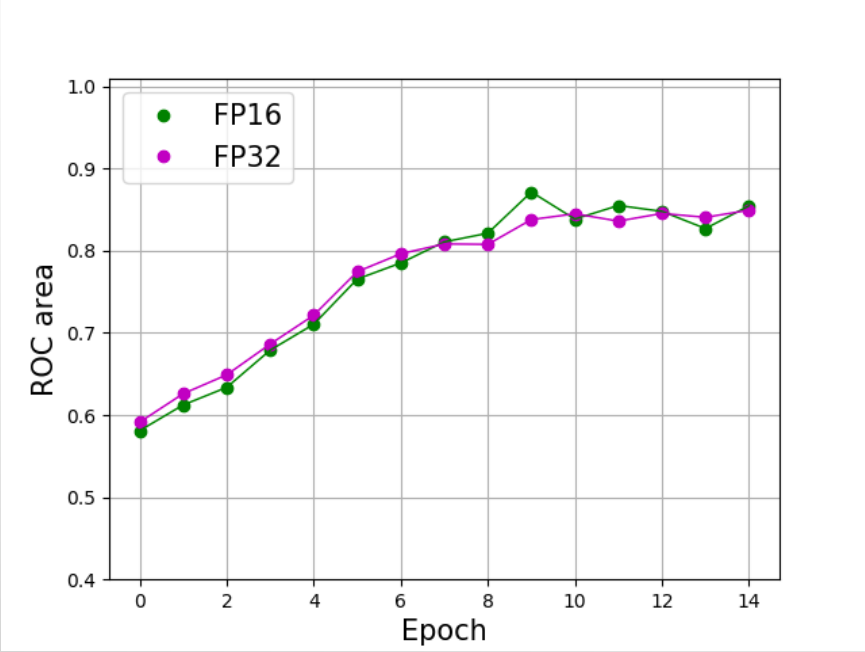
Gated units, LSTM cell

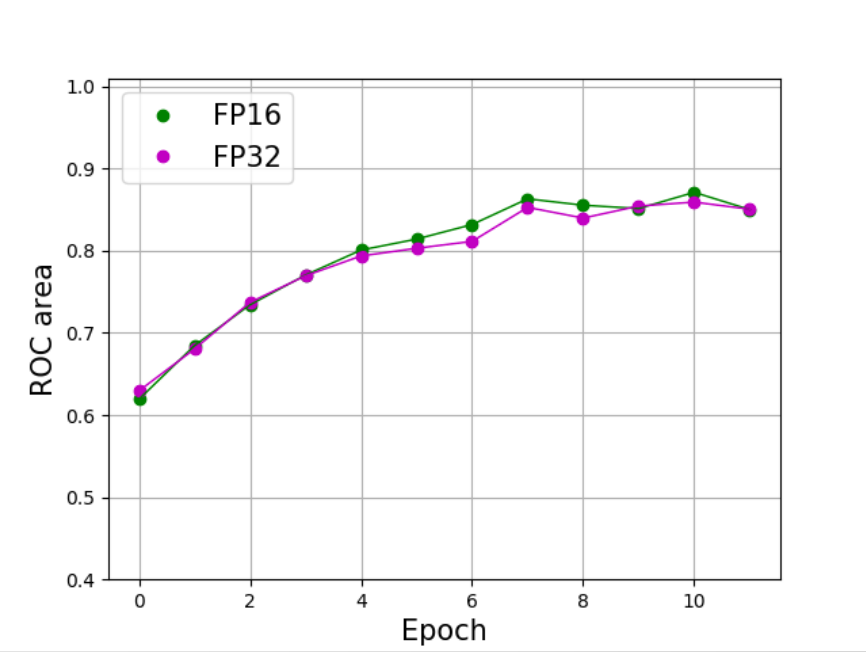
- LSTM is a gated RNN
- LSTM introduces a self-loop – an internal recurrence, in addition to the outer recurrence of the RNN
- The weight of this self-loop is controlled by a forget gate – a notion of memory as input sequence is fed to the model, some information is accumulated in the internal memory
- LSTMs are stateful, as opposed to feedforward neural networks
- $f_i(t) = \sigma(b_i^f + \sum_j U_{ij}^f x_j(t) + \sum_j W_{ij}^f h_j(t-1))$
- $s_i(t) = f_i(t) s_i(t-1) + g_i(t)\sigma(b_i + \sum_j U_{ij} x_j(t) + \sum_j W_{ij} h_j(t-1))$
- $g_i(t) = \sigma(b_i^g + \sum_j U_{ij}^g x_j(t) + \sum_j W_{ij}^g h_j(t-1))$
- $h_i(t) = \tanh(s_i(t)) q_i(t)$
- $q_i(t) = \sigma(b_i^o + \sum_j U_{ij}^o x_j(t) + \sum_j W_{ij}^o h_j(t-1))$

Notations:
x – input sequence,
U – is the input to hidden weight matrix,
W - hidden to hidden,
V – hidden to output weights
b,c are the biases
tanh() is the activation function (non-linearity)
s – state unit
f- forget gate unit
g-external input gate unit
q-output gate unit

Benchmark with IMDB dataset

- IMDB: the Large Movie Review Dataset
  - Contains movie reviews along with their associated binary sentiment polarity labels.
  - Comprises 50000 reviews split evenly between train and test





Validation level AUC per epoch calculated for FP16 and FP32 precisions for the Large Movie Review Dataset. Left: base learning rate $\lambda_0 = 0.02$, right: base learning rate $\lambda_0 = 0.05$.

Model convergence at large N

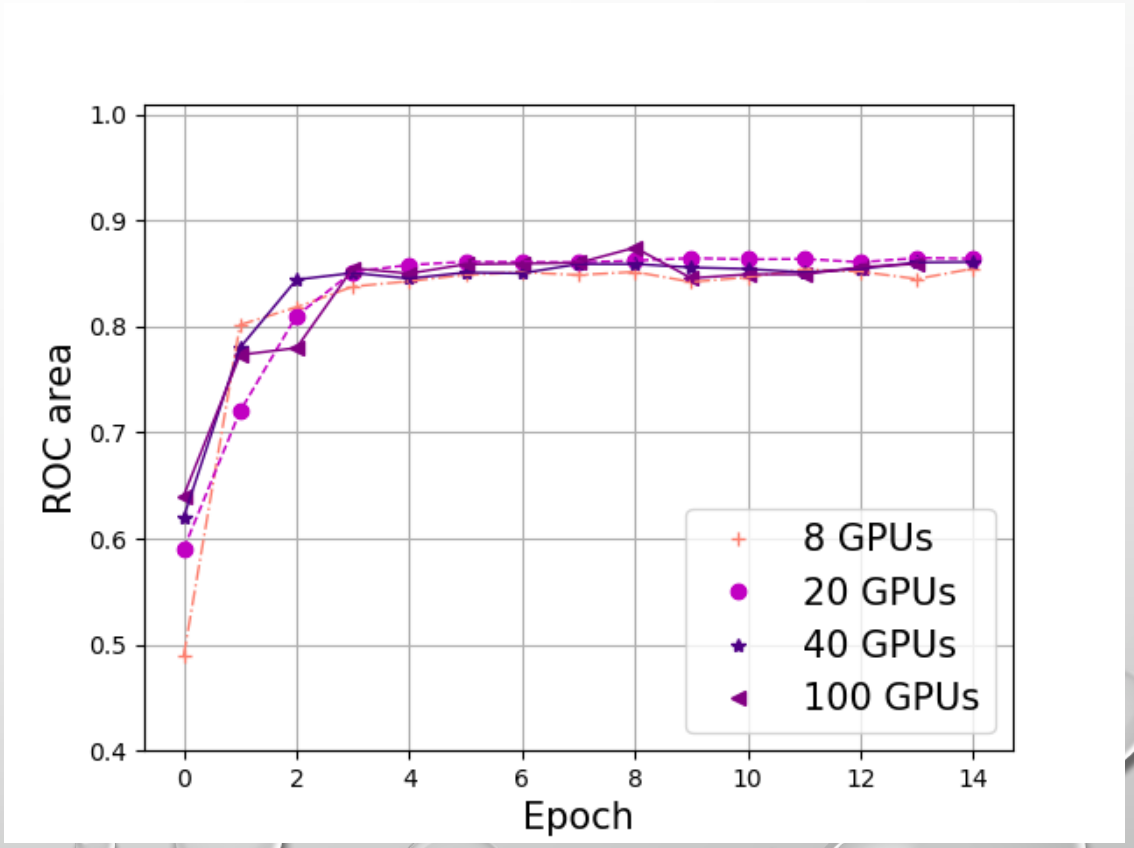
- Learning rate schedule is crucial to facilitate model convergence during distributed training when the effective mini-batch size is large
- We use exponential learning rate decay with adjustable base learning rate

$$\lambda_i = \lambda_0 \gamma^i$$

- Effective batch size is multiplied by the number of workers
- In a distributed regime, the base learning rate is reduced as the number of workers N is increased

$$\lambda_0(N, n) = \frac{\lambda_0}{1.0 + N/n}$$

- Learning rate is clipped



Performance comparison

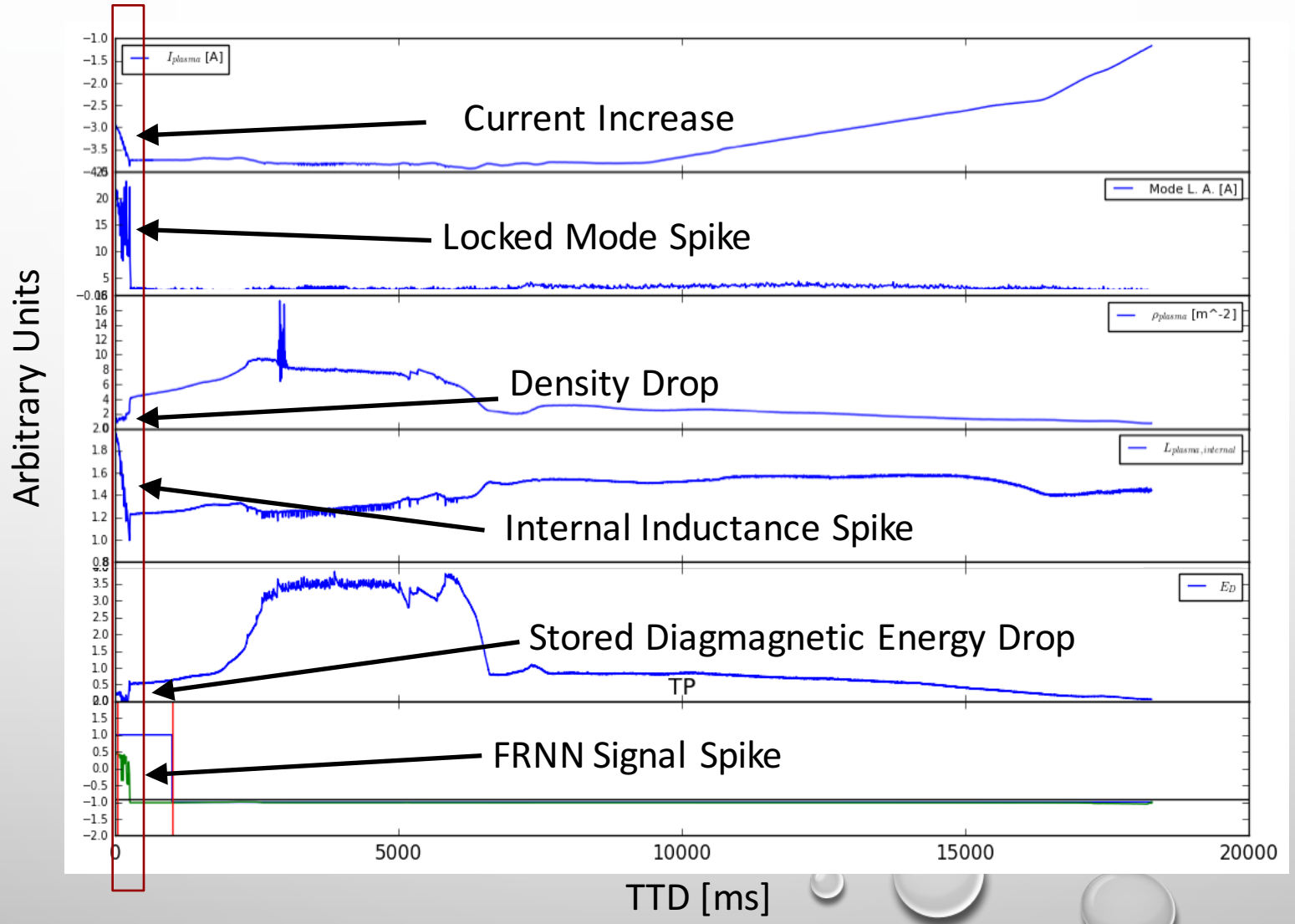
- Overall insights
  - Deep learning appears competitive in raw performance and more suited to generalization
  - Deep RNN can use 1D profiles efficiently
  - On smaller, 0D, and more uniform datasets, "shallow" approaches are still competitive

ROC area @30 ms before disruption

Train set	Test set	Deep RNN	Random Forest
JET (Carbon wall)	JET (ITER-like wall)	0.96	0.88
DIII-D (+1D)	DIII-D (+1D)	0.88	-
DIII-D	DIII-D	0.84	0.85
DIII-D	JET (ITER-like wall)	0.82	0.51
JET (Carbon wall)	DIII-D	0.67	0.62

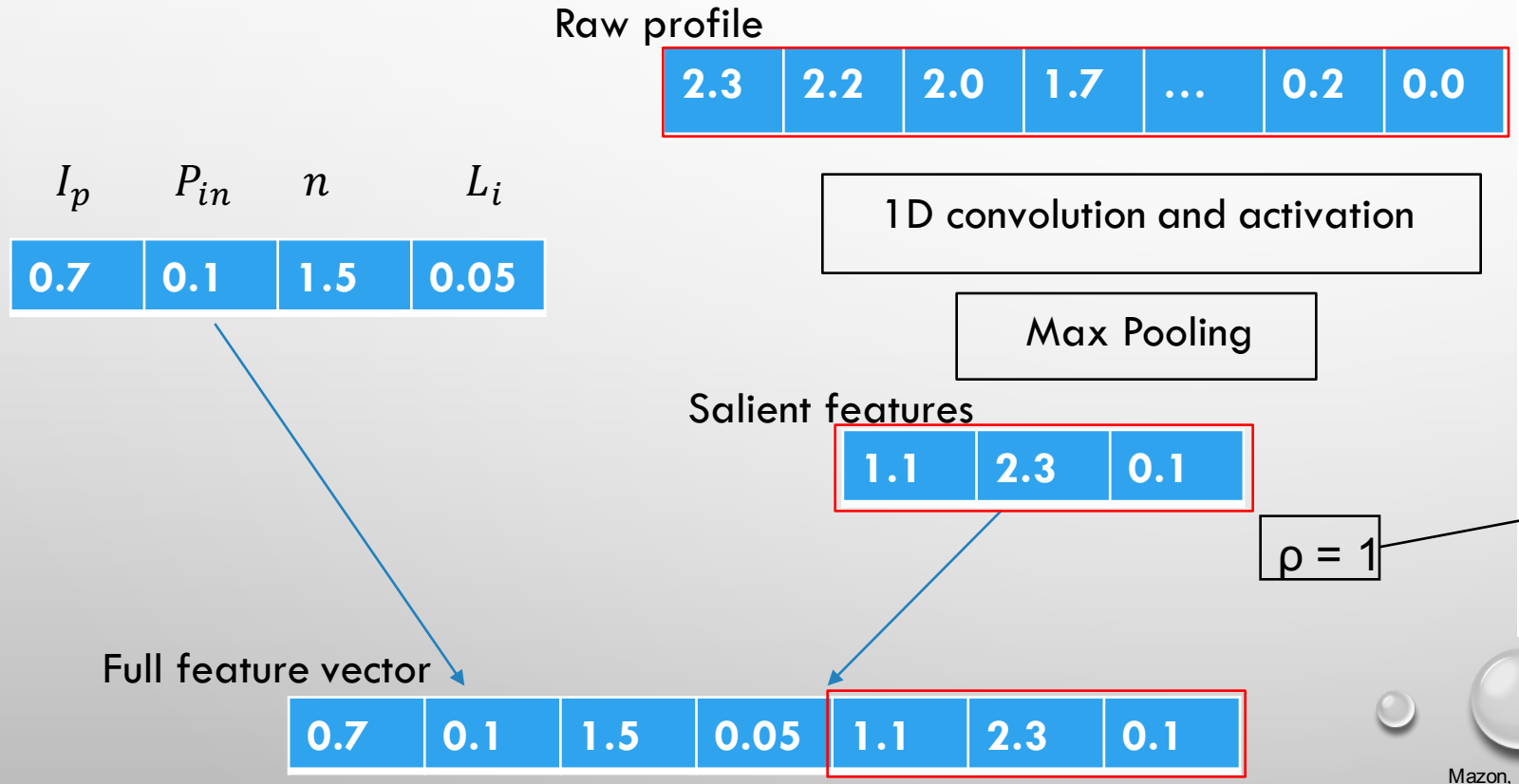
RNN Predictions

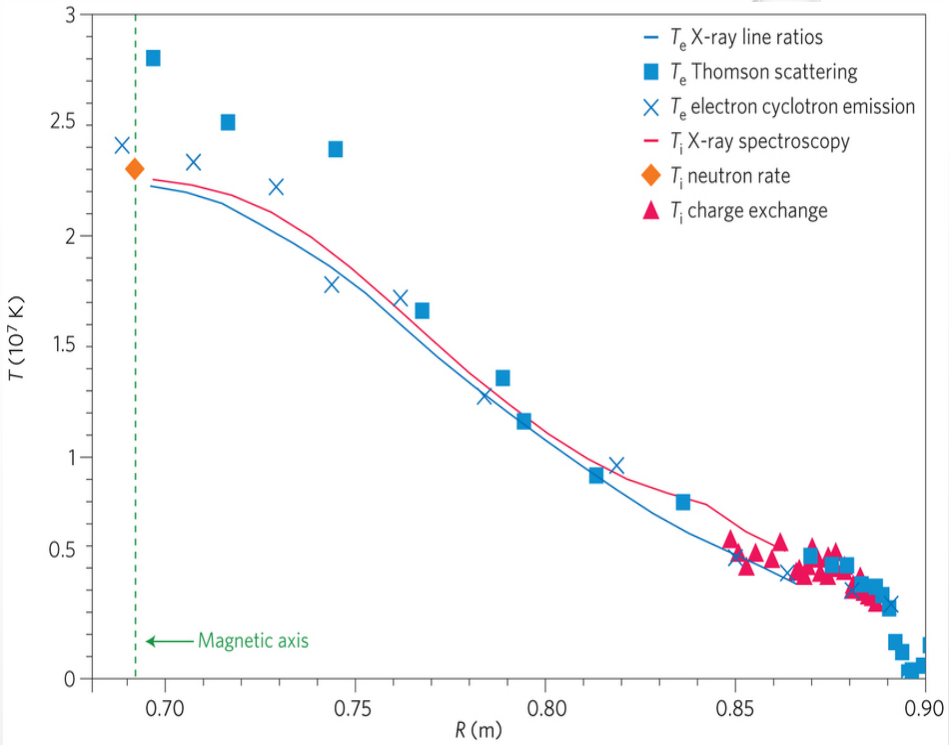
True positive example

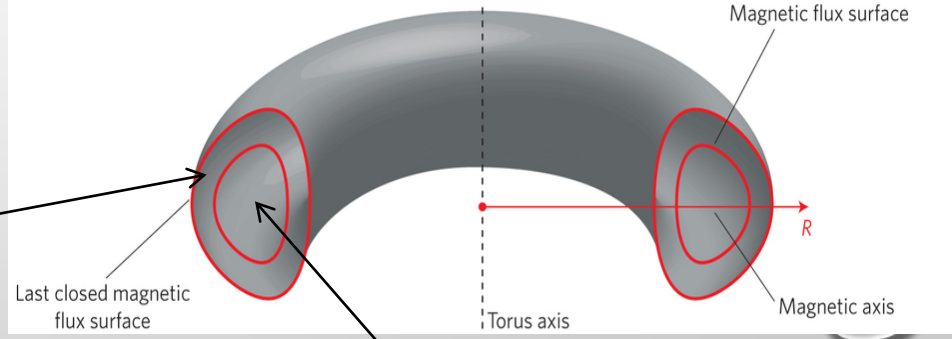


Handling high-dimensional data: DIII-D

- At each timestep: arrays instead of scalars
- All as a function of ρ (normalized flux surface)
- Examples:
  - 1D Current profiles
  - 1D Electron temperature profiles
  - 1D Radiation profiles



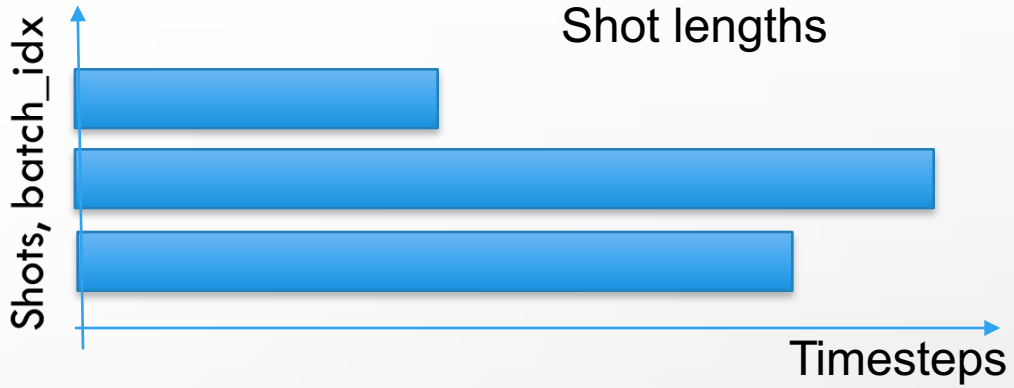




Mazon, Didier, Christel Fenzi, and Roland Sabot. "As hot as it gets." Nature Physics 12.1 (2016): 14-17.

Challenges of stateful LSTM training, sequences of variable length

- Lengths of shots in e.g. JET data vary by orders of magnitude:
  - Minimum length: 1400
  - Mean length: ~27,000
  - Max: ~40,000 time-steps



- Zero-padding to the max length is not the best option with such spread in sequence lengths
- For a model to converge, the best approach is to feed subsequences of shot smaller length and do not reset states after each mini-batch
  - Training is stateful when the last state for each sample at a timestep ***i*** in a mini-batch will be used as initial state for the sample of timestep ***i*** in the following mini-batch
  - Reset states in the end of shot, individually
- The challenges is to implement a custom batch generator which would do that (see next slide)

Challenges of stateful LSTM training, sequences of variable lengths

- Implement a custom batch generator:
  - Takes a list of shots (for instance 2800 shots, each shot a time series of 1400-40000 timesteps). 9 scalar measurements at each time point
- Create Xbuff and Ybuff tensors each holding **batch_size** shots
  - Xbuff shape: **(batch_size, Maximum shot length, dimension of data)**
  - Ybuff shape: **(batch_size, Maximum shot length, 1)**
- For each shot adjust the length to be a multiplier of the LSTM model length, e.g:
  - Model length: 128 (hyper parameter, but generally << **shot length**)
  - Shot length: 25000 timesteps, adjusted shot length: **(Shot length//model length)*model length**
- Fill an array **end_indices**: which contains lengths of shots
- Create a **reset_batches** boolean array containing indicating whether a model states need to be reset (if current shot just ended)
- Each time batch generator yields a tensor of shape **(batch_size, model length, dim of data)**, re-adjusts the Xbuff and Ybuff shifting to the beginning of array by **model length**, decrements **end_indices** by **model length** and checks whether any of **end_indices** are less than zero (meaning we have hit end of shot for a shots at **batch_idx**)
- Once we hit the end of a shot, we do a partial batch reset, then fill in new shot at a **batch_idx**