

SCHOOL OF DATA ANALYSIS



# Trigger system in HEP

Tatiana Likhomanenko

Lund, MLHEP 2016

# Introduction



# Intro notes

- › two tracks:
  - › introductory course
  - › advanced track (this one): Mon, Tue, Wed, then two tracks are merged
- › two lectures and two practice seminars on each day (today's seminars with the introductory track)
- › kaggle challenges
  - › 'Triggers' - only for advanced track, lasts for 3 days
  - › 'Higgs' - for both tracks, lasts for 7 days

# Trigger system at CERN

- › The goal is selecting interesting events (proton-proton collisions) based on detailed online analysis of its physics observables.
- › Hardware stage:
  - › the task is to reduce the rate of visible interaction to the rate at which a detector can be readout (from 40MHz to 1MHz or 100 kHz)
  - › ML in electron PID (CMS), muon trigger (CMS), energy reconstruction (CMS)

# Trigger system at CERN

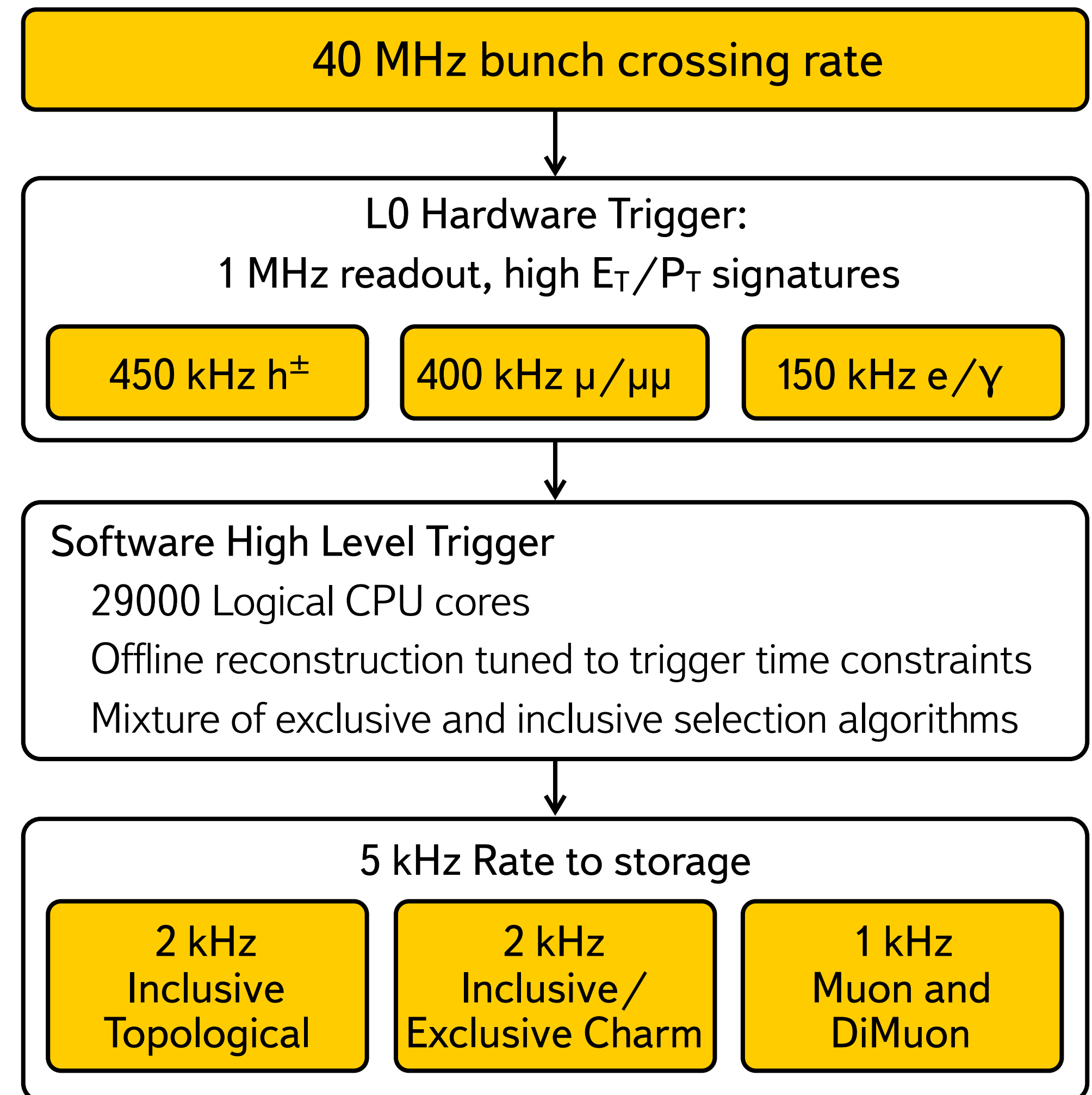
- › Software stage (High Level Trigger, HLT):
  - › performs a full reconstruction of events and writes selected data out to permanent storage
  - › ensure a large acceptance for physics signals
  - › rate is reduced from 1MHz to several thousands Hz or several hundred Hz
  - › ML in isolation algorithms (CMS), final decision rule in topological trigger (LHCb)

# LHCb topological trigger



# LHCb trigger system

- › Select events to store them for offline processing
- › Should efficiently select interesting events
- › An event is interesting if it contains at least one interesting secondary vertex(SV)
- › Output rate for trigger system is limited

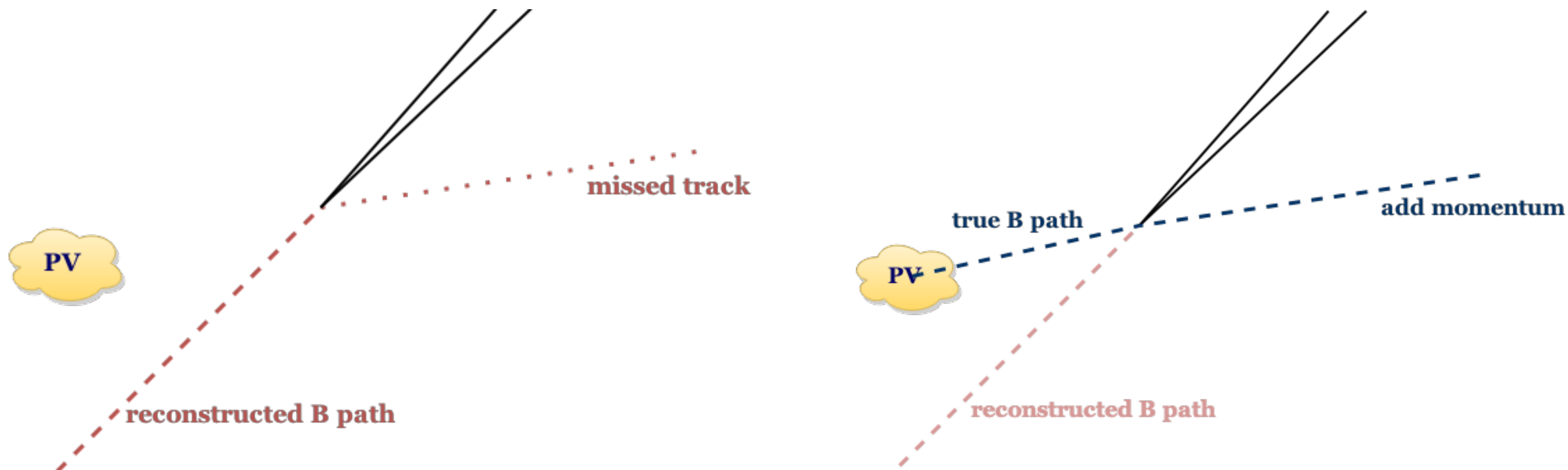


# LHCb topological trigger

- › Generic trigger for decays of beauty and charm hadrons
- › It designed to be inclusive trigger line to efficiently select any B decay with at least 2 charged daughters
- › Look for 2, 3, 4 track combinations in a wide mass range
- › Designed to efficiently select decays with missing particles
- › Use fast-track fit to improve signal efficiency and minbias rejection



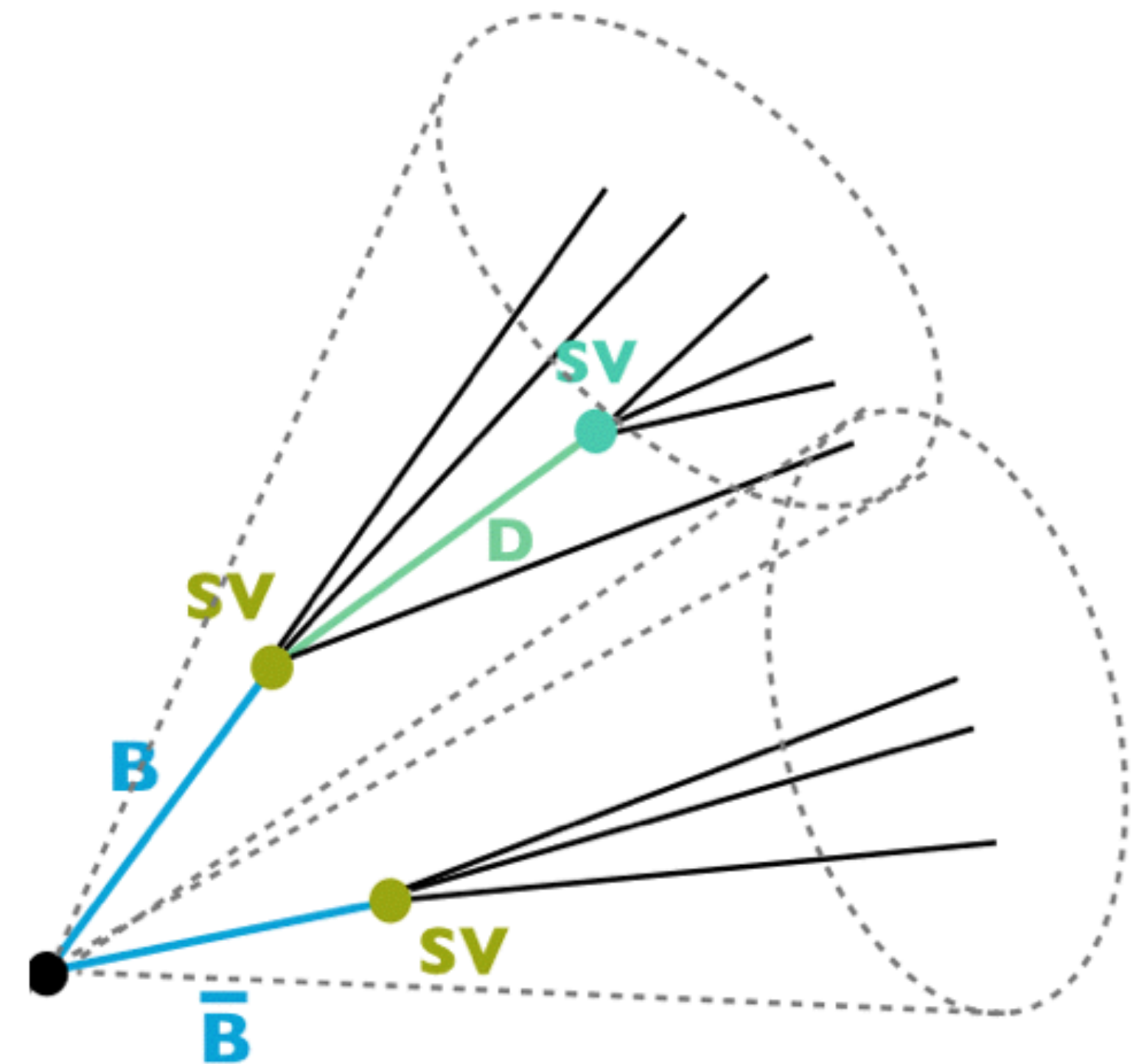
# LHCb topological trigger: omission of daughters



$$m_{corrected} = \sqrt{m^2 + \left| p'_{Tmissing} \right|^2} + \left| p'_{Tmissing} \right|$$

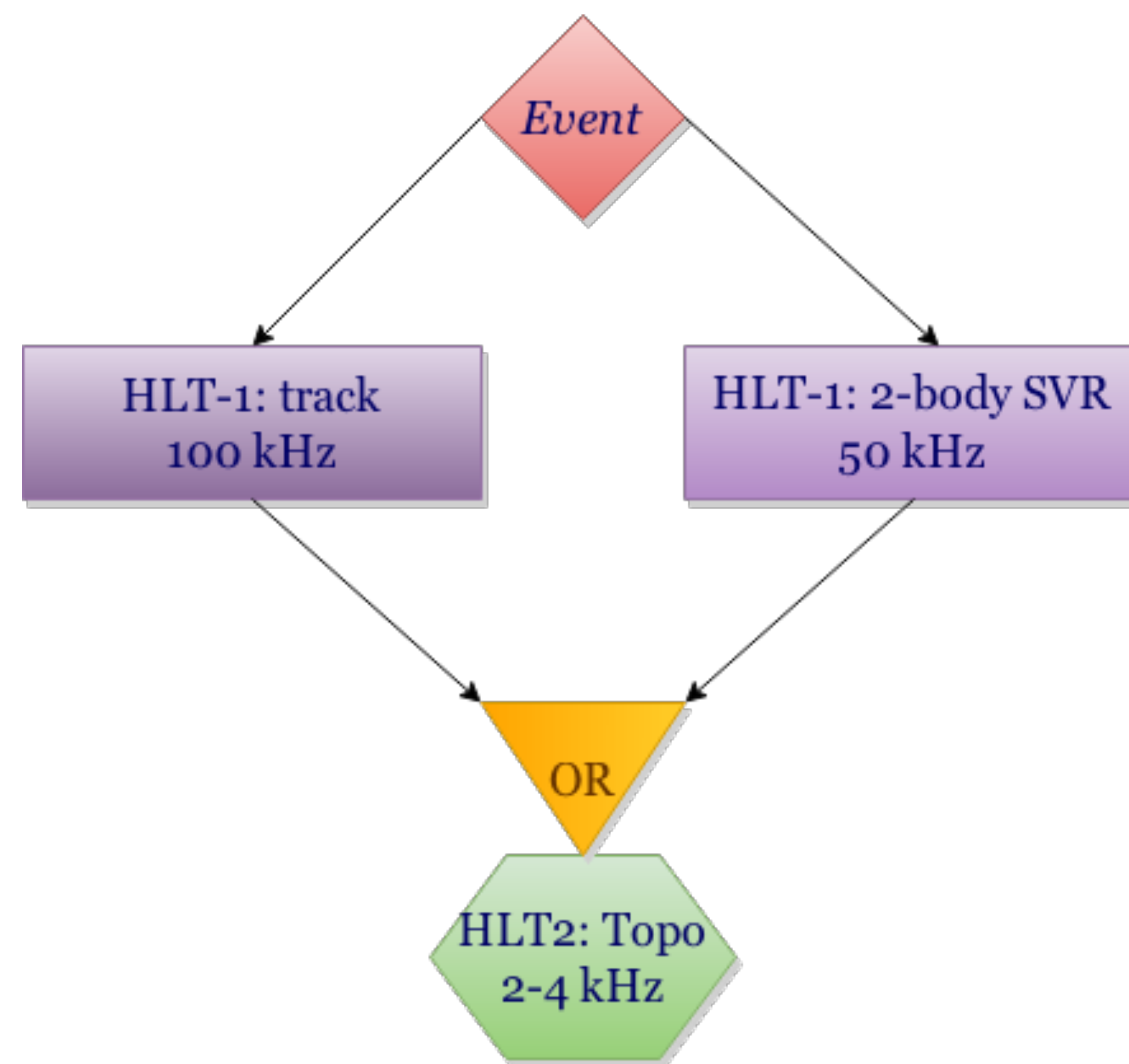
# LHC Event

- › Sample: one proton-proton bunches collision
- › Event consists of:
  - tracks (track description)
  - secondary vertices (SV description)
- › Questions:
  - How to describe event in ML terms?
  - How to train model on such samples?



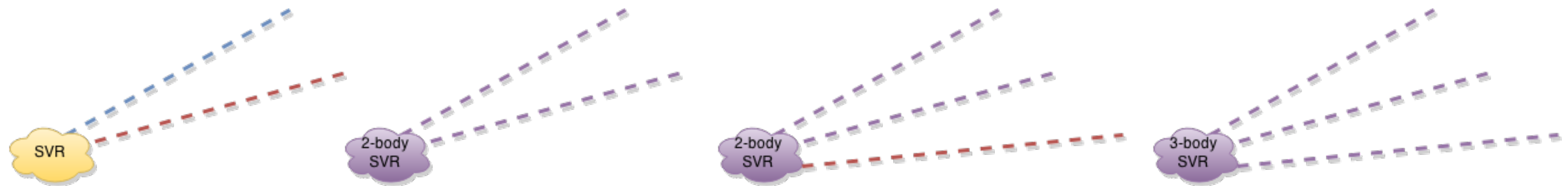
# Run-II LHCb topological trigger

- › HLT-1 track is looking for either one super high PT or high displacement track
- › HLT-1 2-body SV classifier is looking for two tracks making a vertex
- › HLT-2 improved topo classifier uses full reconstructed event to look for 2, 3, 4 and more tracks making a vertex



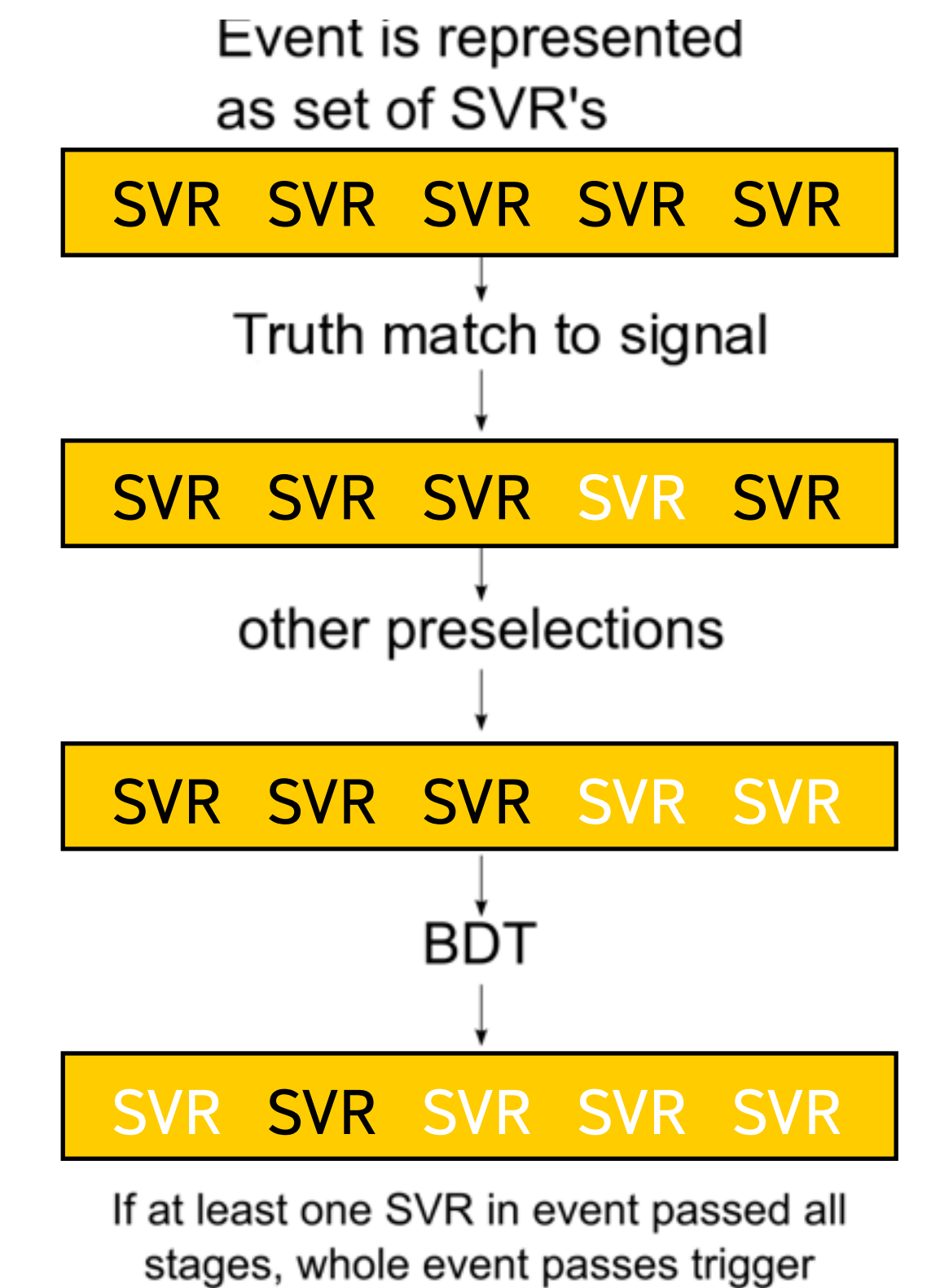
# N body tracks

- › Two, three or four tracks are combined to form a SV
- › Each secondary vertex in Monte Carlo data is preselected in such way, that all tracks must be matched to particles from the signal decay (true match preselection)



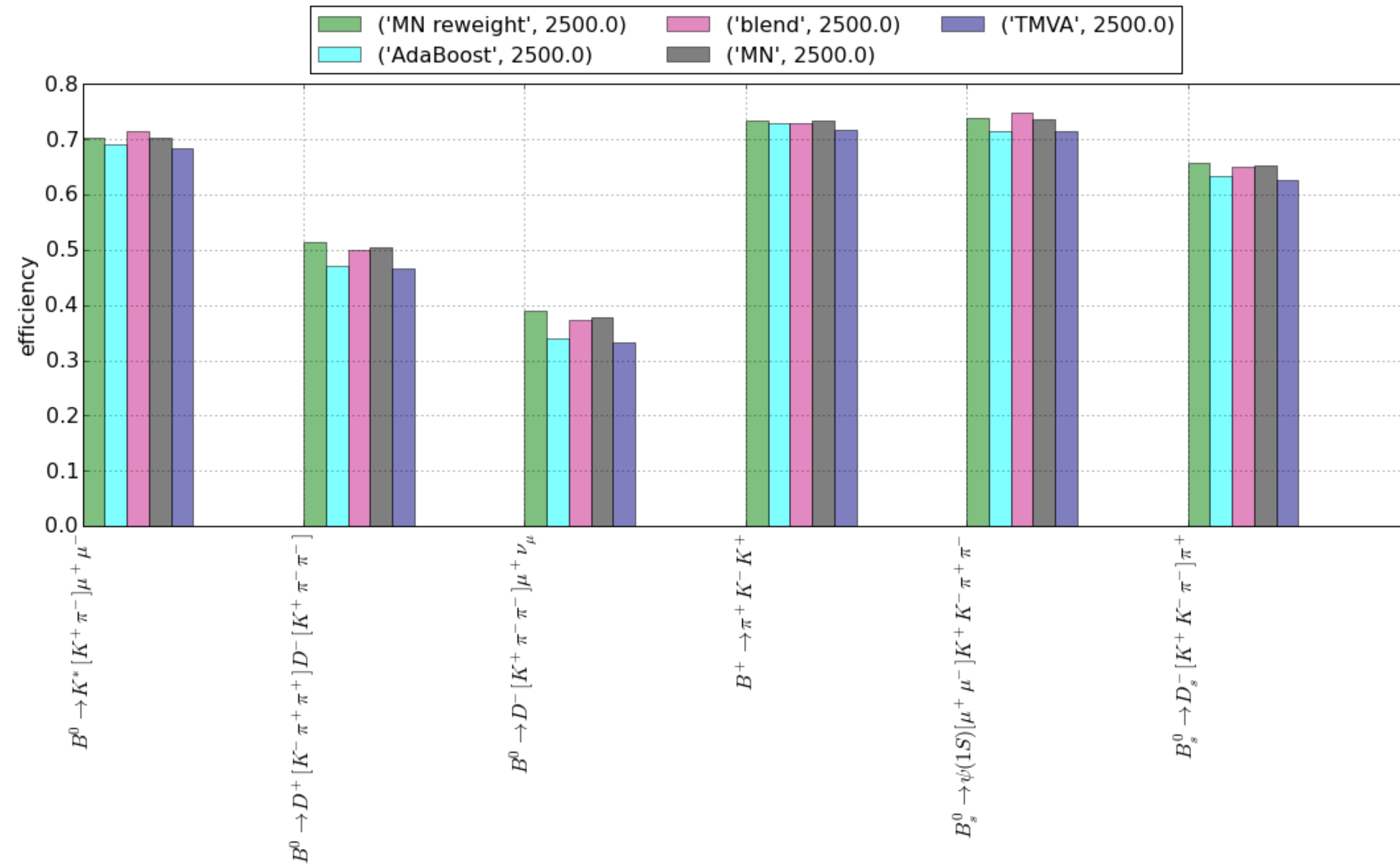
# Data

- › Monte Carlo samples (used as signal-like) are simulated 13-TeV B decays of various topologies
- › Generic Pythia 13-TeV proton-proton collisions are used as background-like sample
- › Training data are a set of SVs for all events
- › Most events have many secondary vertices (not all events have them)
- › Goal is to improve efficiency for each type of signal events along fixed efficiency for background



Event representation

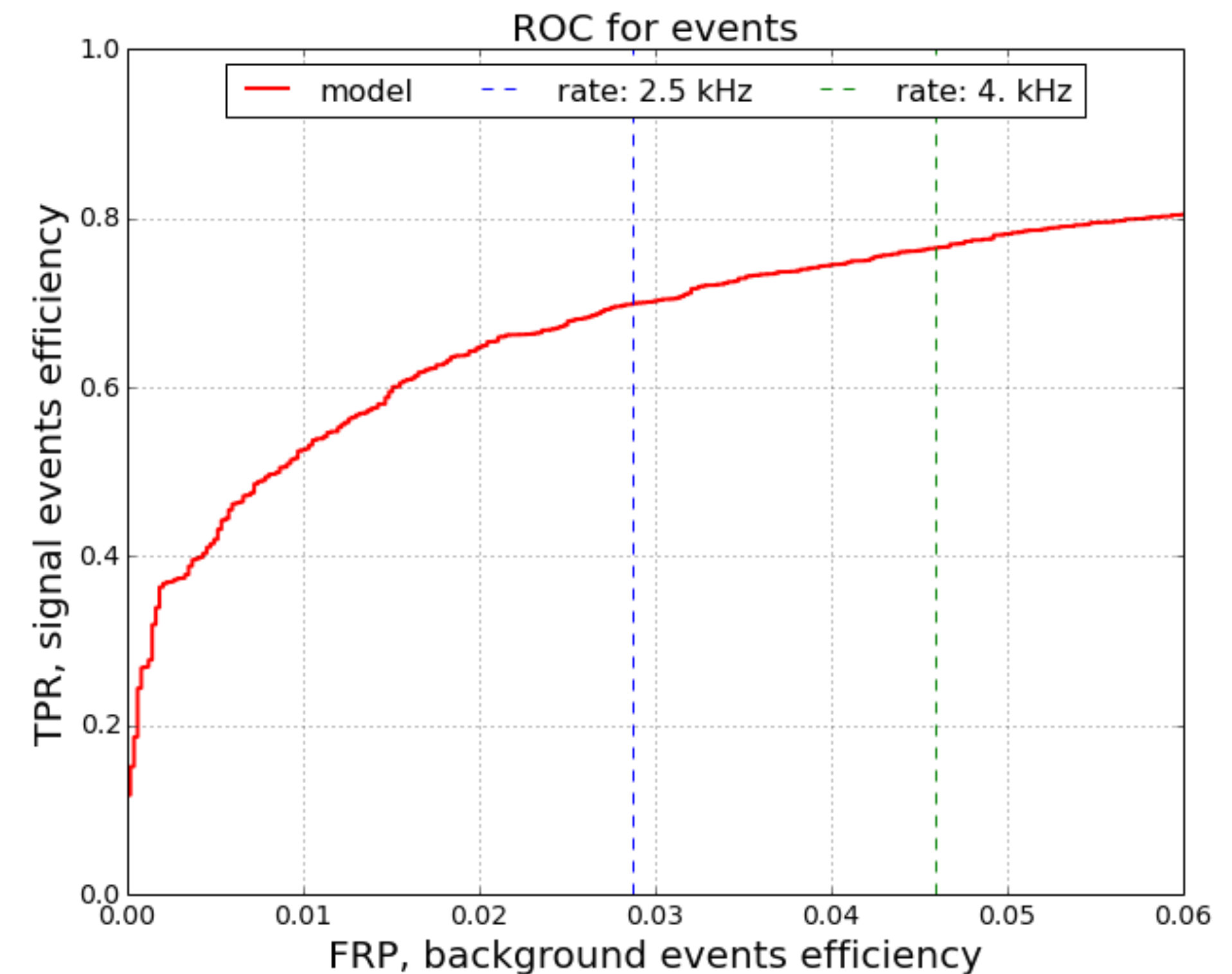
# How to measure quality?





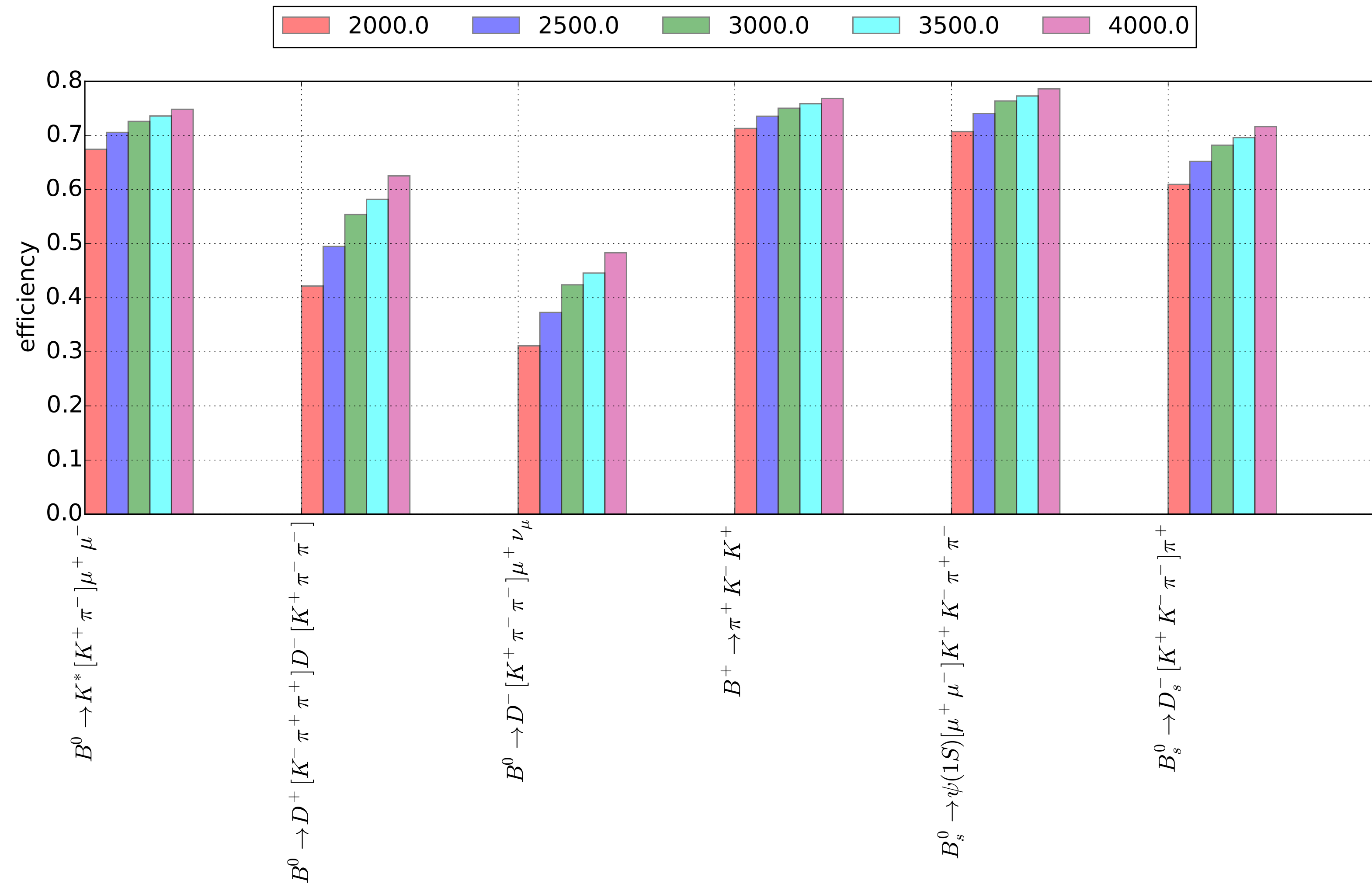
# ROC curve, computed for events

- › Output rate = false positive rate (FPR) for events
- › Optimize true positive rate (TPR) for fixed FPR for events
- › Weight signal events in such way that channels have the same sum of weights.
- › Optimize ROC curve in a small FPR region



ROC curve interpretation

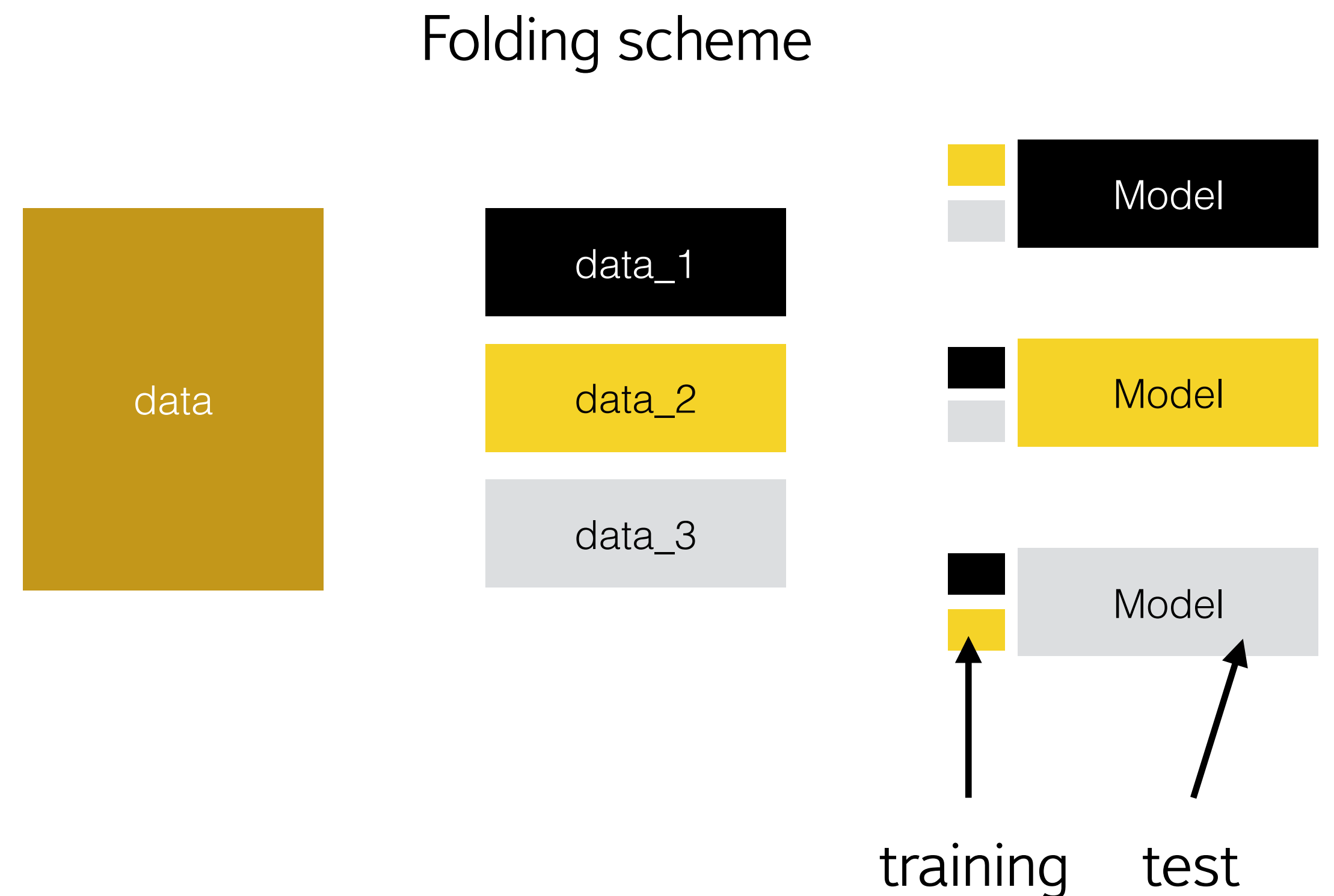
# Dependence on output rate





# Hierarchical training

- › Train separate models for:
  - each channel
  - each n-body type: 2, 3, 4
- › Use them as additional features later
- › Use folding scheme to train additional features or to apply additional classifier selections



# Folding scheme (rep library)

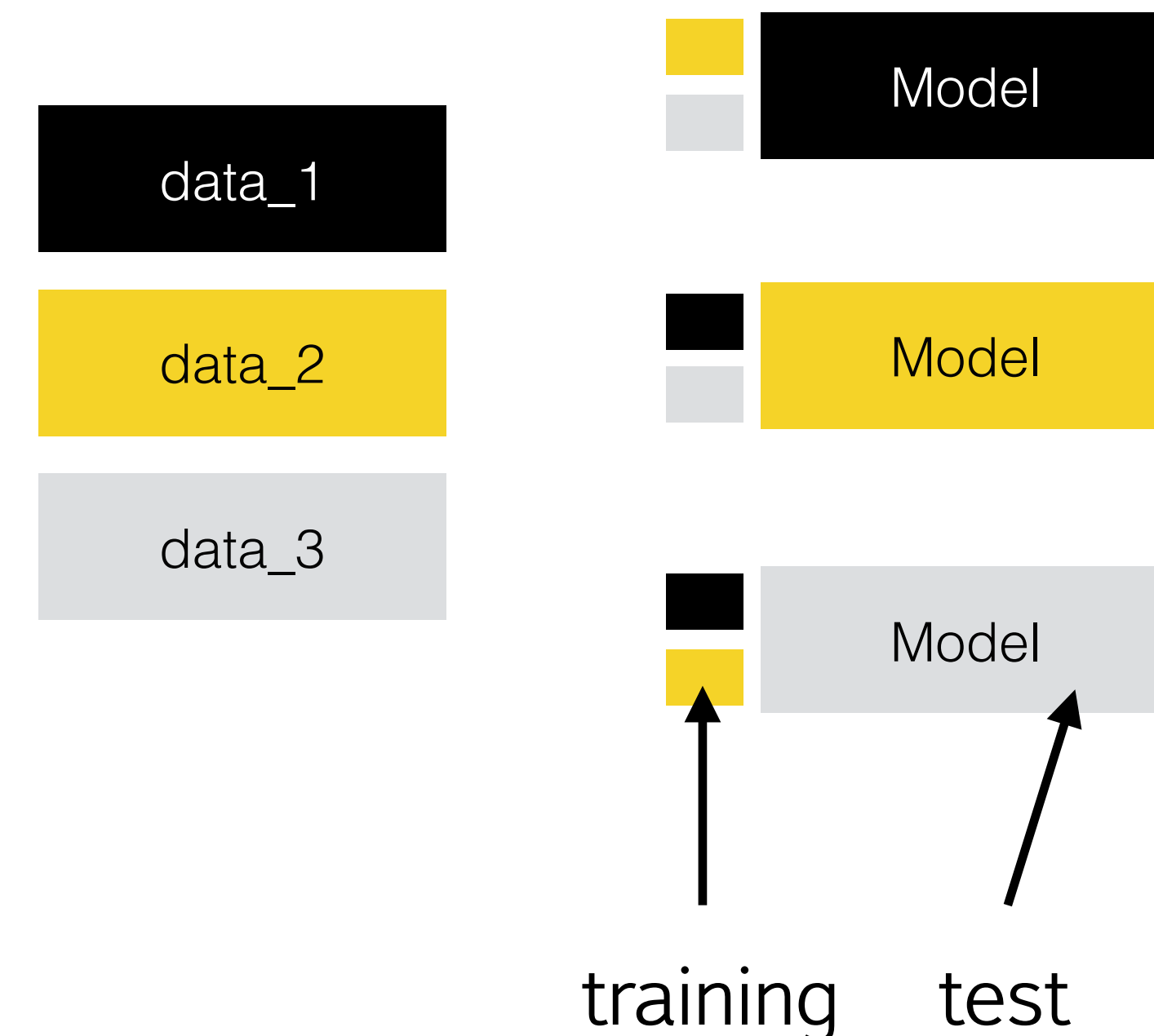
```
from rep.metaml import FoldingClassifier


# define number of folds
n_folds = 4
# define folding scheme classifier with base estimator GB
folder = FoldingClassifier(GradientBoostingClassifier(n_estimators=30)
                          n_folds=n_folds,
                          parallel_profile='threads-4')
# FoldingClassifier has sklearn interface
folder.fit(train_data, train_labels)

folder.predict_proba(test_data)
```



Folding scheme



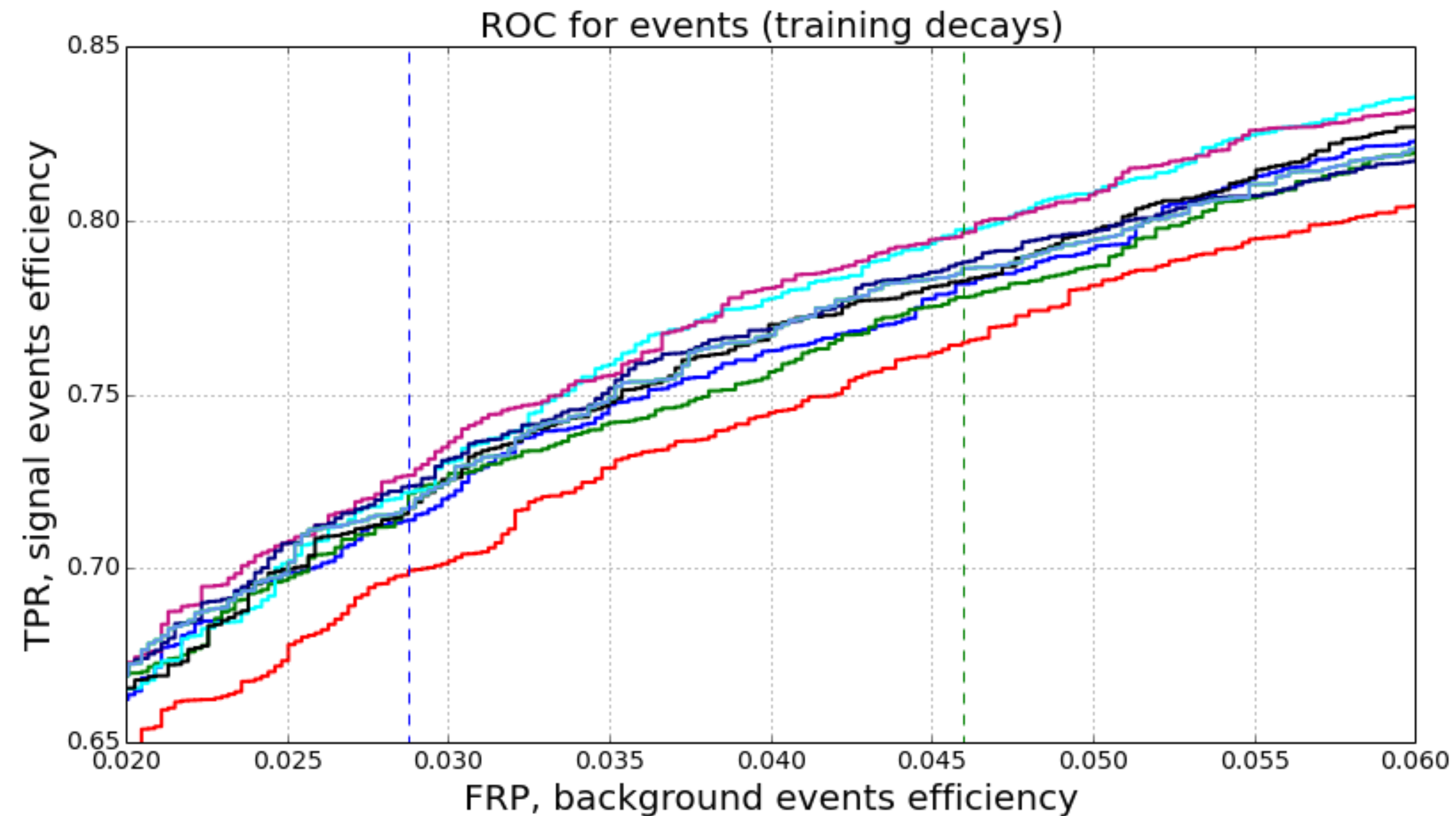
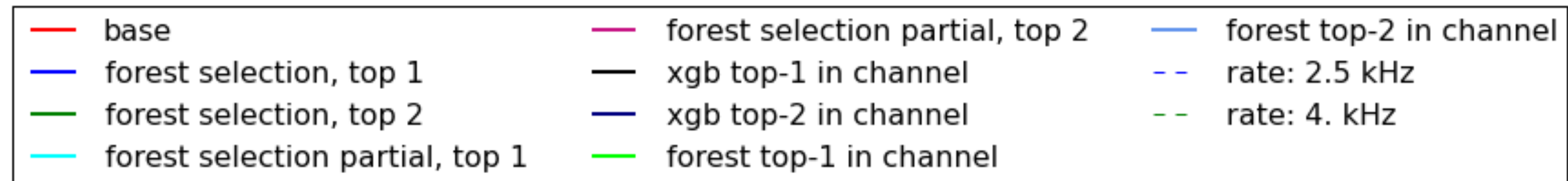


Simulated signal event  
contains at least one  
interesting *SV*, but not each  
*SV* should be interesting

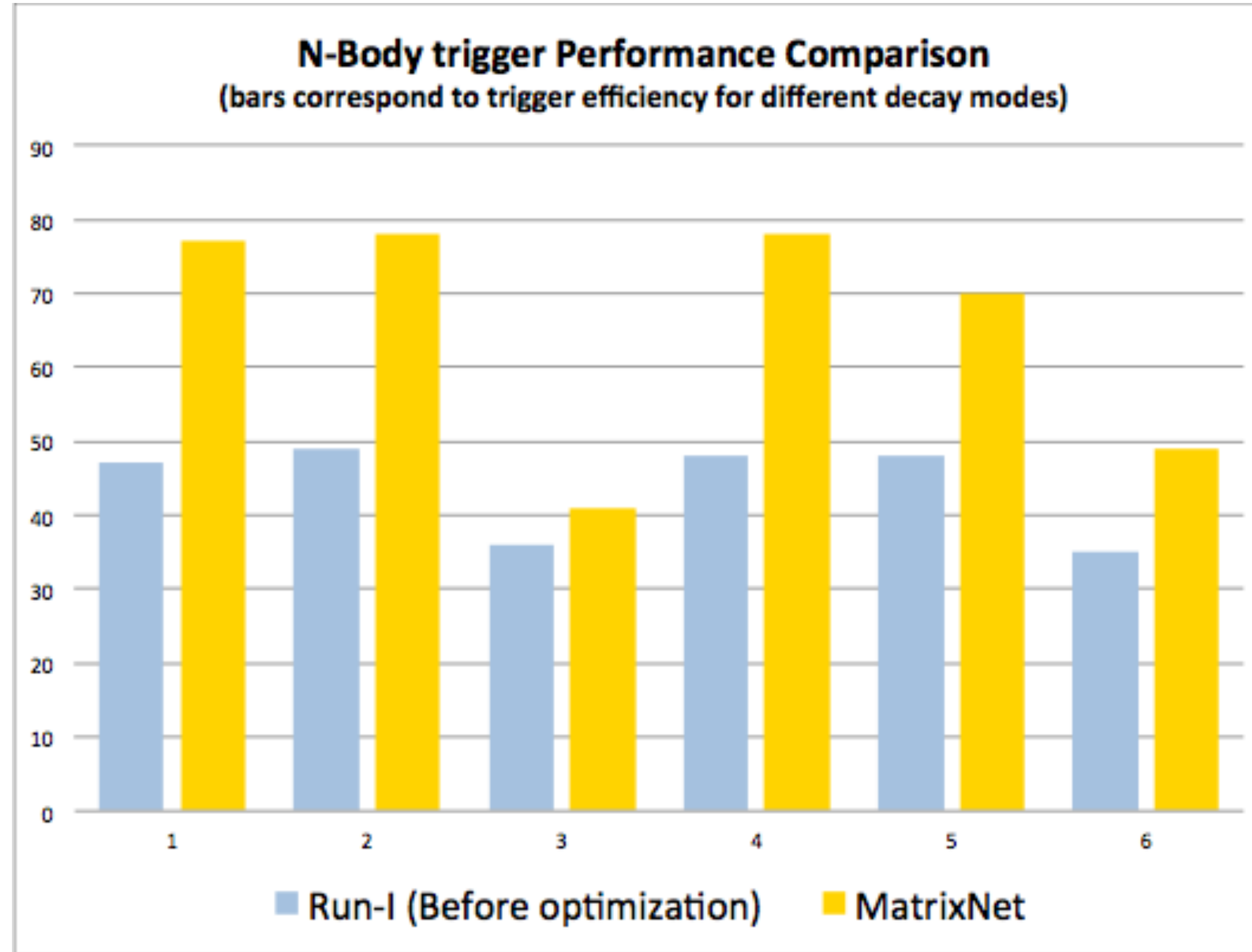
# Random forest (RF) for SVs selection

- › Train random forest (RF) on SVs using folding scheme
  - RF is stable to noise in data
  - RF doesn't penalize in case of misclassification (can find noisy samples)
- › Select top-1, top-2 SVs by RF predictions for each signal event
- › Train classifier on selected SVs
- › Try another algorithm instead of RF, maybe it will work!

# Random forest for SVs selection



# Topological trigger results (without RF trick)



# Trigger system competition



# Competition

- › link to the competition
- › timescale: from now till 22 June 11:59:00 PM UTC
- › ROC AUC as a metric
- › teams with 2 members, only one of which could be from the physics department, are allowed. It is forbidden to have two physicists in a team.
- › notebook with a baseline solution is in the github
- › use gitter for competition discussion and questions



# Data

- › Monte Carlo samples. Six decays of various topology, which are studied by physicists, are marked as «interesting». As "uninteresting" events generic simulated proton-proton collisions are provided.
- › Each event consists of the set of reconstructed secondary vertices (SV).
- › Number of secondary vertices varies for each event.
- › Dataset is a set of all reconstructed secondary vertices for all events (sample=secondary vertex from an event).
- › Training file contains weight

# List of available features

- › **EventID** - id for event (it will be the same for secondary vertices come from the same event)
- › **Label** - "interesting" event (label 1) or "uninteresting" (label 0)
- › **Mass** - mass of the SV
- › **Corrected\_mass** - "corrected" mass of the SV ([details](#), page 7)
- › **Pt** - transverse momentum
- › **Pt\_sum** - summation of transverse momentums (pt) for all tracks in the SV
- › **Pt\_min** - the minimum of tracks pt in the SV
- › **IP\_chi2** - impact parameter chi2 of the SV
- › **IP\_chi2\_sum** - sum of ipchi2 for all tracks in the SV.
- › **Flight\_distance** - flight distance chi2 of the SV from the proton-proton collision
- › **Pseudorapidity** - pseudo rapidity
- › **Track\_number\_PV** - number of preselected tracks in the primary vertex
- › **Tracks\_number** - number of tracks in the SV
- › **Tracks\_number\_passed** - number of preselected tracks
- › **Vertex\_chi2** - vertex chi2 of the SV

# Challenge summary

- › try different models and libraries to improve the quality (XGBoost, TMVA, sklearn, Keras, ...)
- › play with decision rule for event probability definition from the SVs probabilities
- › hierarchical training: different topologies, different number of tracks, ...
- › different approaches to preselect signal samples for training
- › your own ideas

Triggers: meta algorithm



# Why?

Typical physics trigger:

- › the set of classifiers (small decision trees)
- › each uses a subset of the raw observables
- › the most costly operation at test time is not the evaluation of the tree but the construction of the features
- › the cost of classifier depends on:
  - › the raw features it uses
  - › features that have already been constructed in previously evaluated trees

# Markov decision process (MDP)

MDDAG is a post-processing method that takes the output of a trained classifiers and “sparsifies” it (selection should be data-dependent).

- › classifiers are sorted in order of “importance” or performance of the base classifiers.

*$N$  base classifiers  $\mathcal{H} = (\mathbf{h}_1, \dots, \mathbf{h}_N)$ .*

$$\mathbf{f}(\mathbf{x}) = \sum_{j=1}^N \mathbf{h}_j(\mathbf{x})$$

- › for each classifier we choose action:

- › evaluate

- › skip

- › quit

$$b_j(\mathbf{x}) = 1 - \mathbb{I} \{a_j = \text{SKIP OR } \exists j' < j : a_{j'} = \text{QUIT}\}$$

$$\mathbf{f}^{(N)}(\mathbf{x}) = \sum_{j=1}^N b_j(\mathbf{x}) \mathbf{h}_j(\mathbf{x}).$$

# Markov decision process (MDP)

- › the decision action  $a_j$  will be made based on the index of the classifier  $j$  and the output vector of the final classifier:

$$\mathbf{f}^{(j)}(\mathbf{x}) = \sum_{j'=1}^j b_{j'}(\mathbf{x}) \mathbf{h}_{j'}(\mathbf{x}).$$

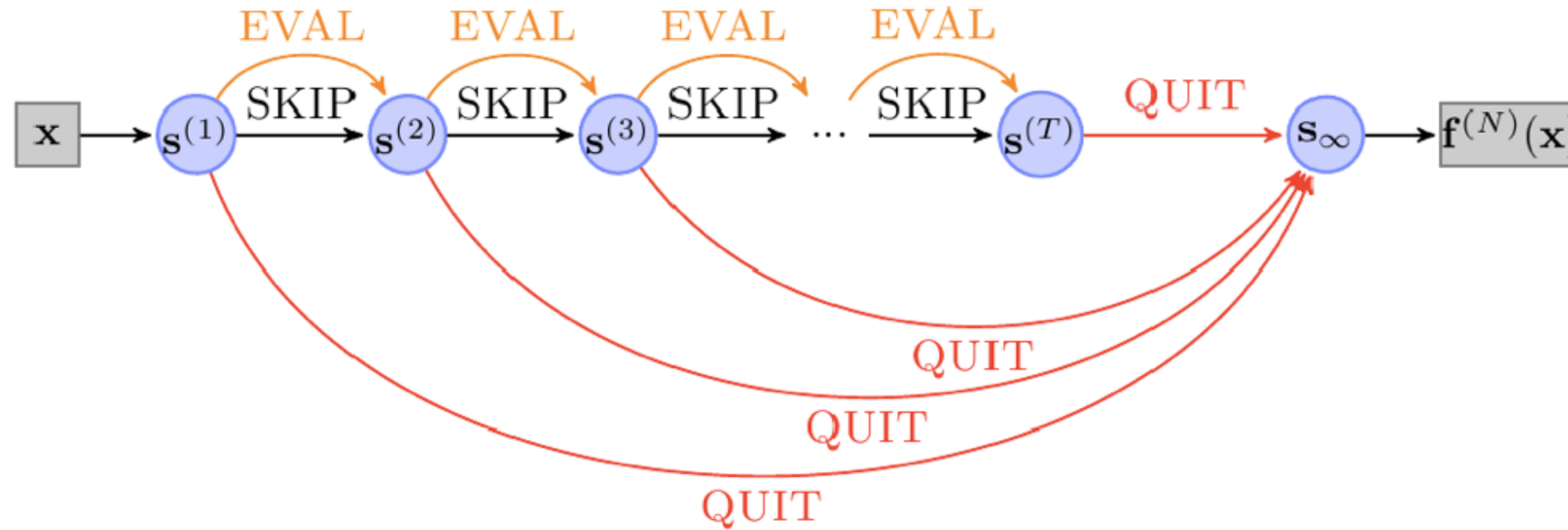
Formally,  $a_j = \pi(\mathbf{s}_j(\mathbf{x}))$ , where

$$\mathbf{s}_j(\mathbf{x}) = (f_1^{(j-1)}(\mathbf{x}), \dots, f_K^{(j-1)}(\mathbf{x}), j-1) \in \mathbb{R}^K \times \mathbb{N}^+$$

- ›  $\mathbf{s}_j$  is the state where we are before visiting  $\mathbf{h}_j$ , and is a policy that determines the action in state  $\mathbf{s}_j$ . The initial state  $\mathbf{s}_1$  is the zero vector with  $K+1$  elements.



# Markov decision process (MDP)



This setup formally defines a Markov decision process (MDP). An MDP is a 4-tuple  $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ , where  $\mathcal{S}$  is the (possibly infinite) state space and  $\mathcal{A}$  is the countable set of actions.  $\mathcal{P} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$  is the transition probability kernel which defines the random transitions  $s^{(t+1)} \sim \mathcal{P}(\cdot | s^{(t)}, a^{(t)})$  from a state  $s^{(t)}$  applying the action  $a^{(t)}$ , and  $\mathcal{R} : \mathbb{R} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$  defines the distribution  $\mathcal{R}(\cdot | s^{(t)}, a^{(t)})$  of the *immediate reward*  $r^{(t)}$  for each state-action pair. A *deterministic policy*  $\pi$  assigns an action to each state  $\pi : \mathcal{S} \rightarrow \mathcal{A}$ . We will only use *undiscounted* and *episodic* MDPs where the policy  $\pi$  is evaluated using the *expected sum of rewards*

$$\varrho = \mathbb{E} \left\{ \sum_{t=1}^T r^{(t)} \right\} \quad (5)$$

with a finite horizon  $T$ . In the episodic setup we also have an *initial* state ( $s_1$  in our case) and a *terminal* state  $s_{\infty}$  which is impossible to leave.



# MDP: reward

› multiclass margin ( $l$  - *true class*)  $\rho^{(t)}(\mathbf{x}, \ell) = f_{\ell}^{(t)}(\mathbf{x}) - \max_{\ell' \neq \ell} f_{\ell'}^{(t)}(\mathbf{x})$ .

› then reward for the right action:

$$r_{\mathbb{I}}^{(t)}(\mathbf{x}, \ell) = \mathbb{I} \left\{ \rho^{(t)}(\mathbf{x}, \ell) > 0 \right\}.$$

› use convex upper bound for the reward:

$$r_{\text{EXP}}^{(t)}(\mathbf{x}, \ell) = \exp \left( \rho^{(t)}(\mathbf{x}, \ell) \right).$$

› introduce hyperparameter accuracy/speed trade-off  $\beta$ :

› the global goal is to learn a policy which maximizes:

$$\mathcal{Q} = \mathbb{E}_{(\mathbf{x}, \ell) \sim \mathcal{D}} \left\{ r(\mathbf{x}, \ell) - \beta \sum_{j=1}^N b_j(\mathbf{x}) \right\}$$

› this optimization problem can be solved by reinforcement learning (RL) with Q-learning approach, for example

# Speed-up



# Issues

- › boosted decision trees (real-time trigger systems)
  - › running times
- › state-of-the-art performance of deep neural networks in many domains of large-scale ML -> millions of learnable parameters (portable devices, phones)
  - › expensive hardware -> memory demands (89%-100% for weights of layers)
  - › running times

Boosted decision trees  
speed-up



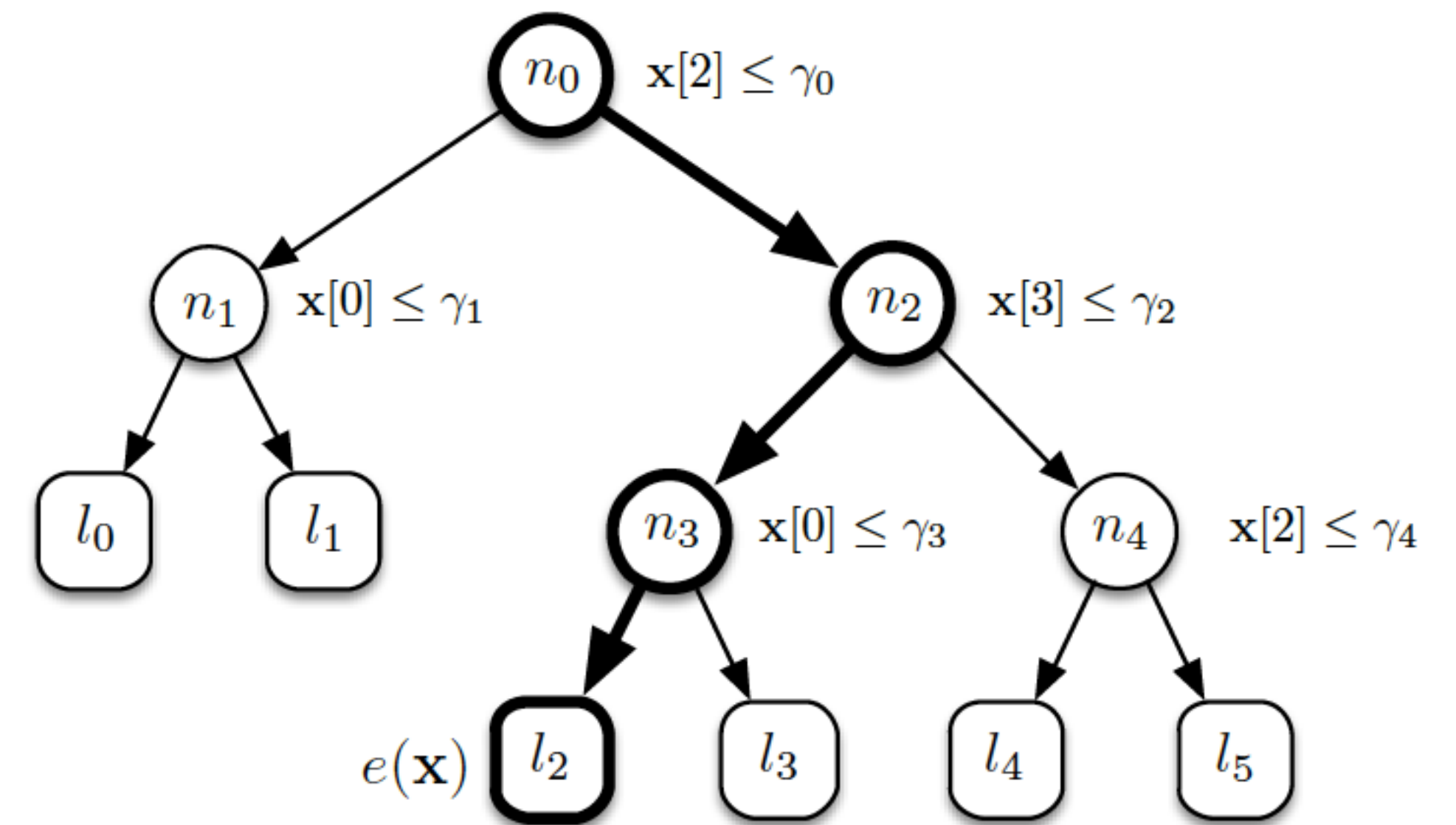
# Information retrieval (IR)

- › ranking query
- › the most effective rankers: GB regression trees (GBRT) and Lambda-MART
- › thousands of trees -> the most expensive part of pipeline in terms of computational time
- › speed-up ranking without losing the quality is the urgent research topic in Web search

# QuickScorer: overview

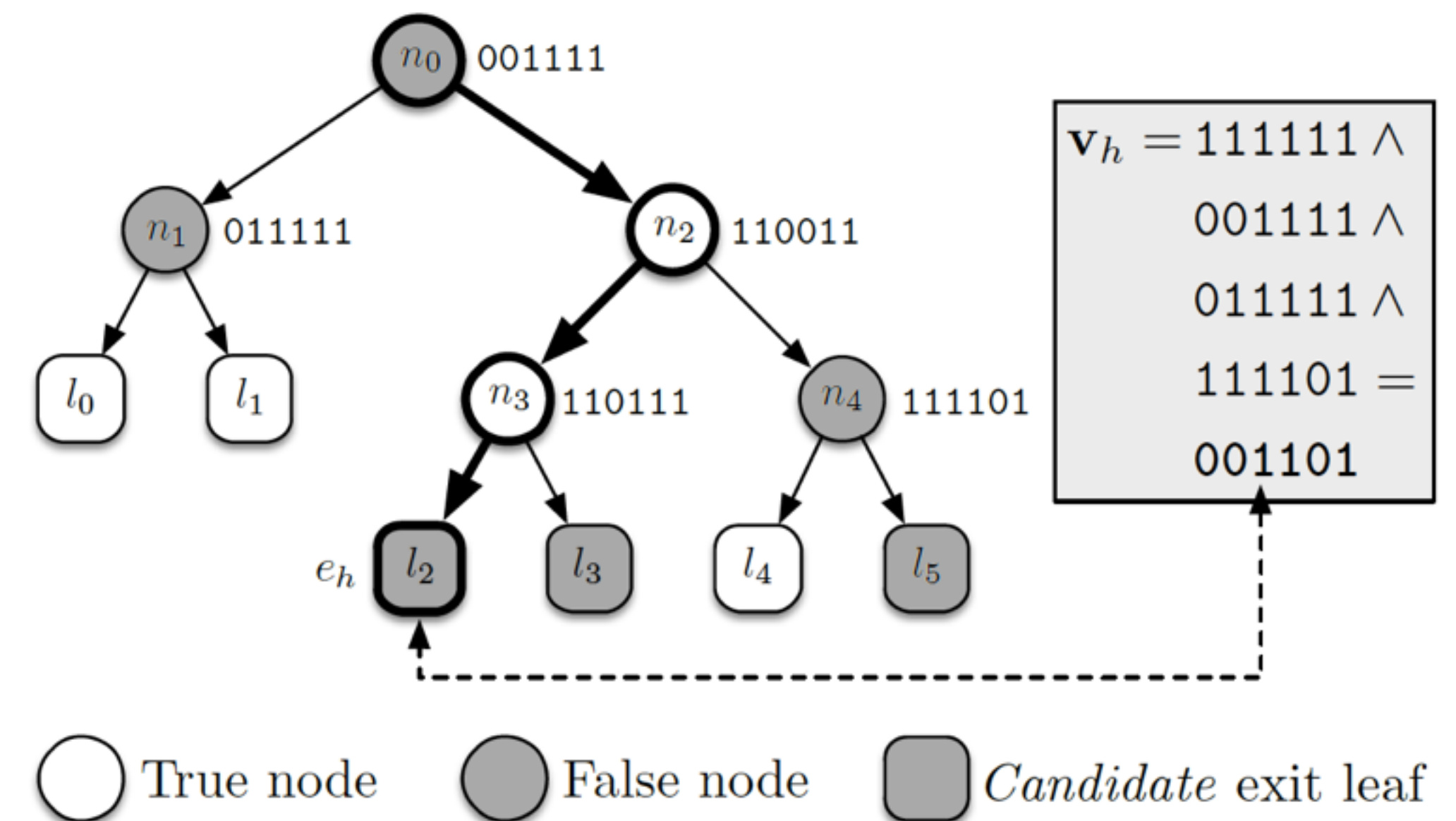
- › fast find all the false nodes
- › define a set of candidates exit leaves: fast operations on compact bitvectors
- › exit leaf is always the one associated with the smallest identifier

Order leaves from left to right.



# QuickScorer: bitvectors

- › present a set of candidate exit leaves  $C_h$  as a bitmask, where each bit corresponds to a distinct leaf
- › every internal node  $n$  is associated with a node bitvector, a bitmask that encodes the set of leaves to be removed from  $C_h$  whenever node is a false node.
- › bitwise logical AND between the bitvector and the false node bitvector
- › more operations but no code branching



# QuickScorer: false nodes

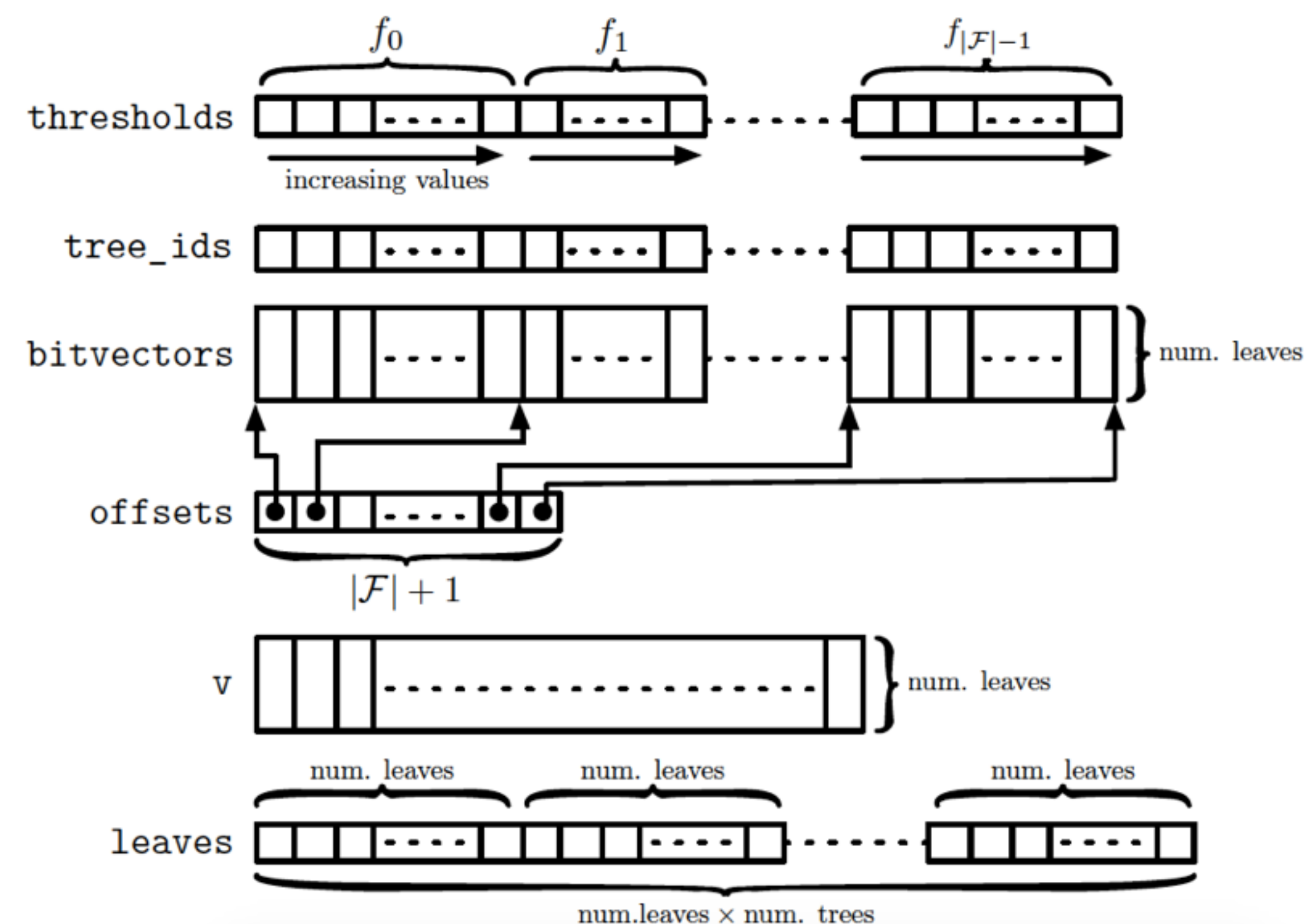
- › perform a global visit of the whole tree ensemble (identify efficiently the false nodes of all the tree ensemble by exploiting an interleaved evaluation of all the trees in the ensemble).
- › incrementally discover for each feature the false nodes involving this feature in any tree of the ensemble (we are able to operate in a cache-aware fashion with a small number of Boolean comparisons and branch mis-predictions)
- › the bitvector of a certain tree is updated as soon as a false node for that tree is identified (once the algorithm has processed all the features each bitvector is guaranteed to encode the exit leaf in the corresponding tree)



# QuickScorer: false nodes

Concentrate on the processing of a feature  $f_k$ :

- › present node with involving feature  $f_k$  as a triplet (feature threshold, tree id, the node bitvector)  
store them as three separate arrays from memory aspect
- › sort these triples in ascending order of their feature thresholds (sorting is crucial)
- › feature value splits sorted triplets in two sublists: False and True sublists of nodes

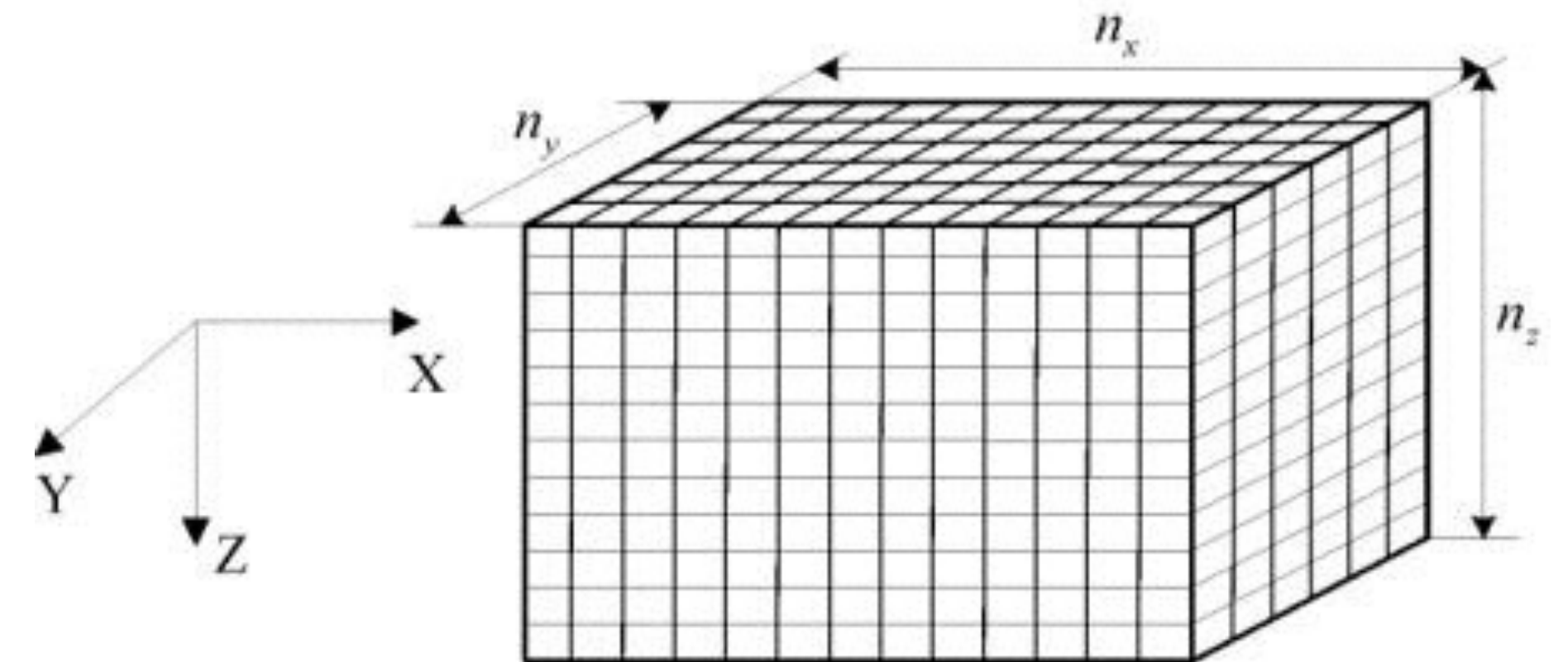


# QuickScorer: summary

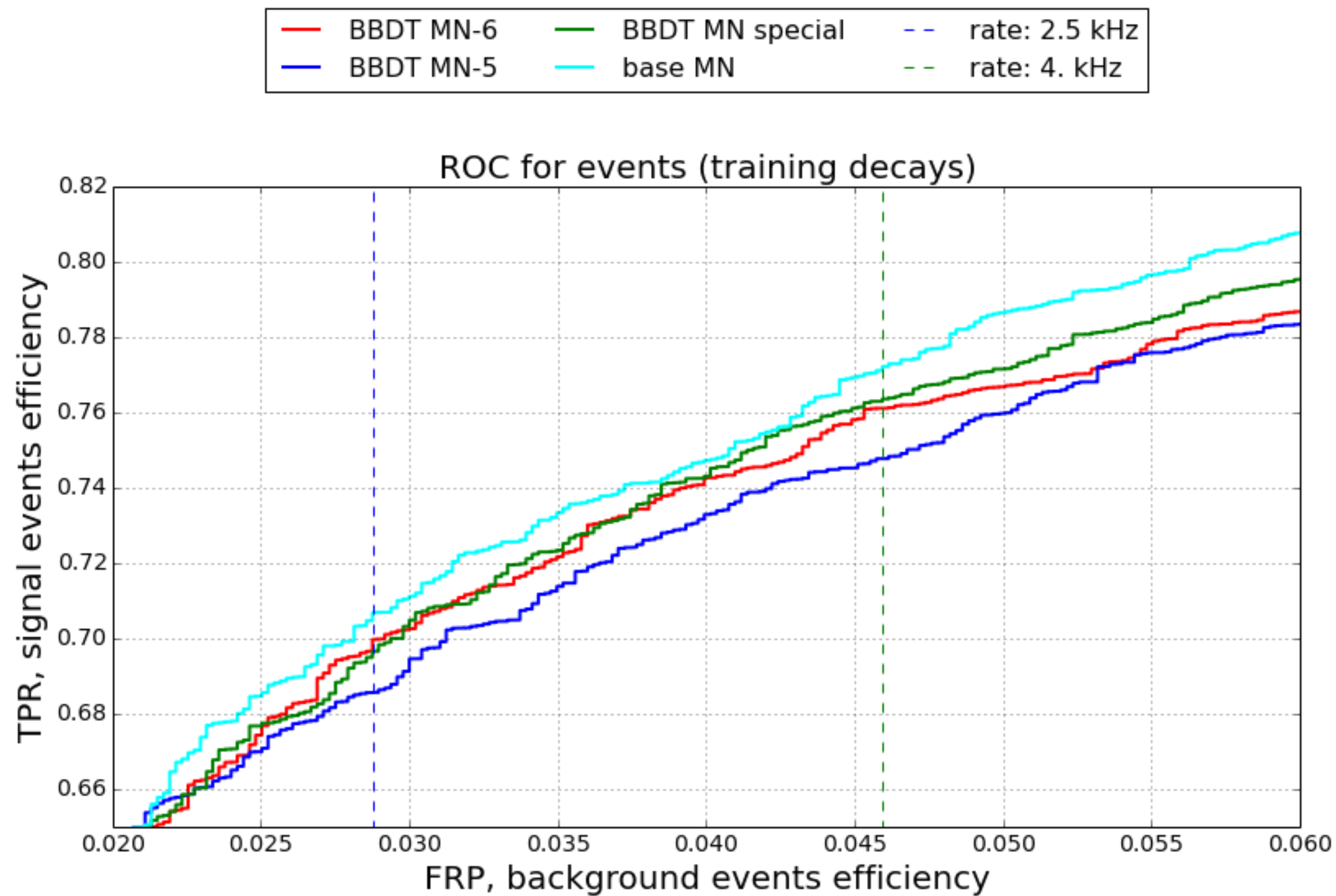
- › usually maximum number of leaves is kept small:  $\leq 64$ ; the length of bitvectors is equal to (or less than) a typical machine word of modern CPUs (64 bits)  $\rightarrow$  can be realized very efficiently, because they involve machine words
- › the cache usage can greatly benefit from the layout and access modes of proposed data structures
- › sorted triplets  $\rightarrow$  test only part of all conditions
- › the speedups are up to **6.5x** over the best state-of-the-art algorithm

# Bonsai BDT format (BBDT)

- › Features hashing using bins before training
- › Converting decision trees to n-dimensional table (lookup table)
- › Table size is limited in RAM (1Gb), thus count of bins for each features should be small (5 bins for each of 12 features)
- › Discretization reduces the quality
- › Prediction operation takes one reading from the table

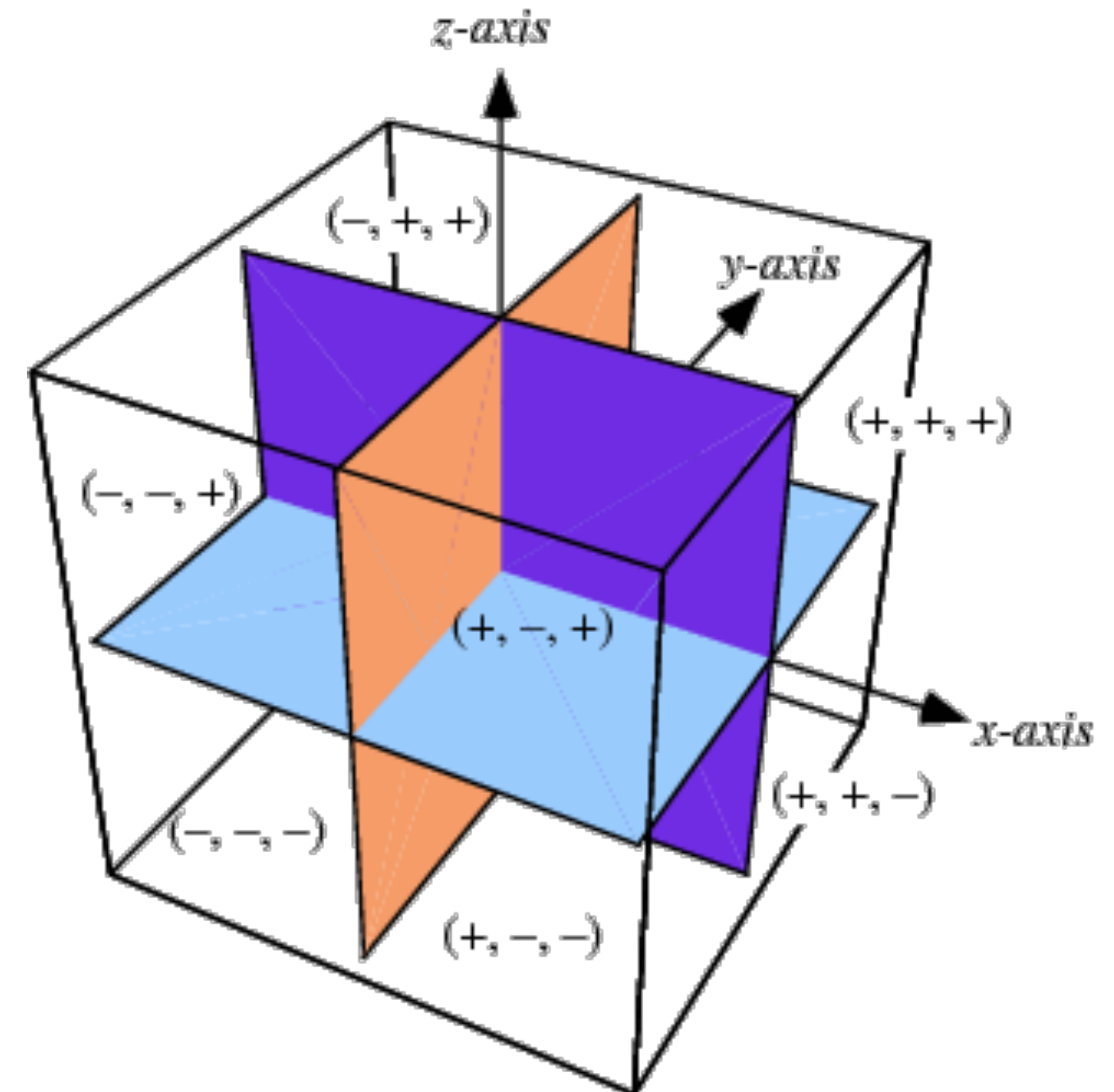


# BBDT, results



# Oblivious decision tree

- › split the space with orthogonal cuts
- › If-Then-Else clauses are absent
- › bit operations can be used in code
- › significantly less degrees of freedom (compared to the arbitrary tree)



# BDT post-pruning

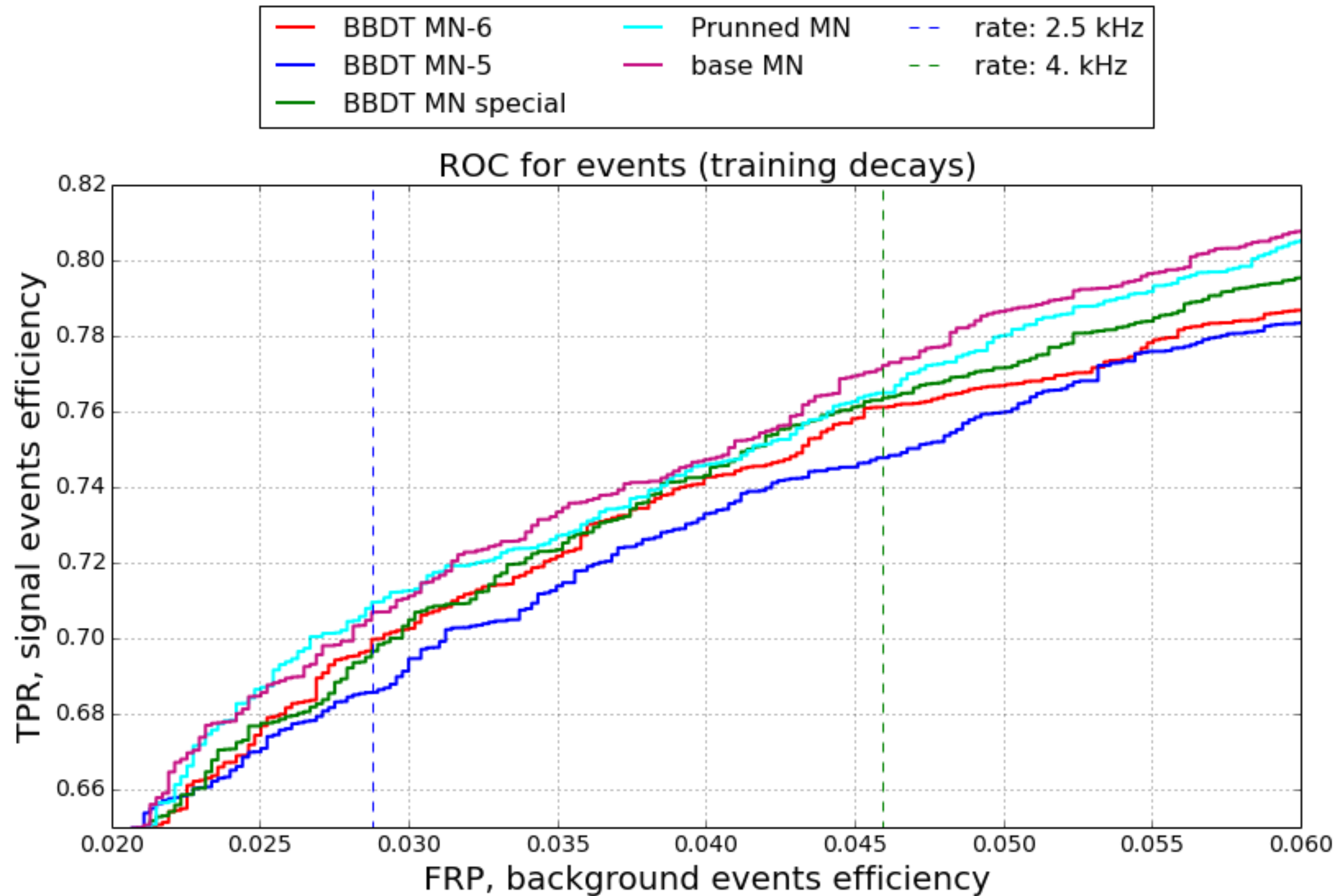
- › Train GB over **oblivious** trees with several thousands trees
- › Reduce this amount of trees to a hundred
- › Greedily choose trees in a sequence from the initial ensemble to minimize a modified loss function:

$$\sum_{\text{signal}} \log \left( 1 + e^{-F(x)} \right) + \sum_{\text{background}} e^{F(x)}$$

- › At the same time change values in leaves (tree structure is preserved)



# Post-pruning, results



# FPGA and trigger system

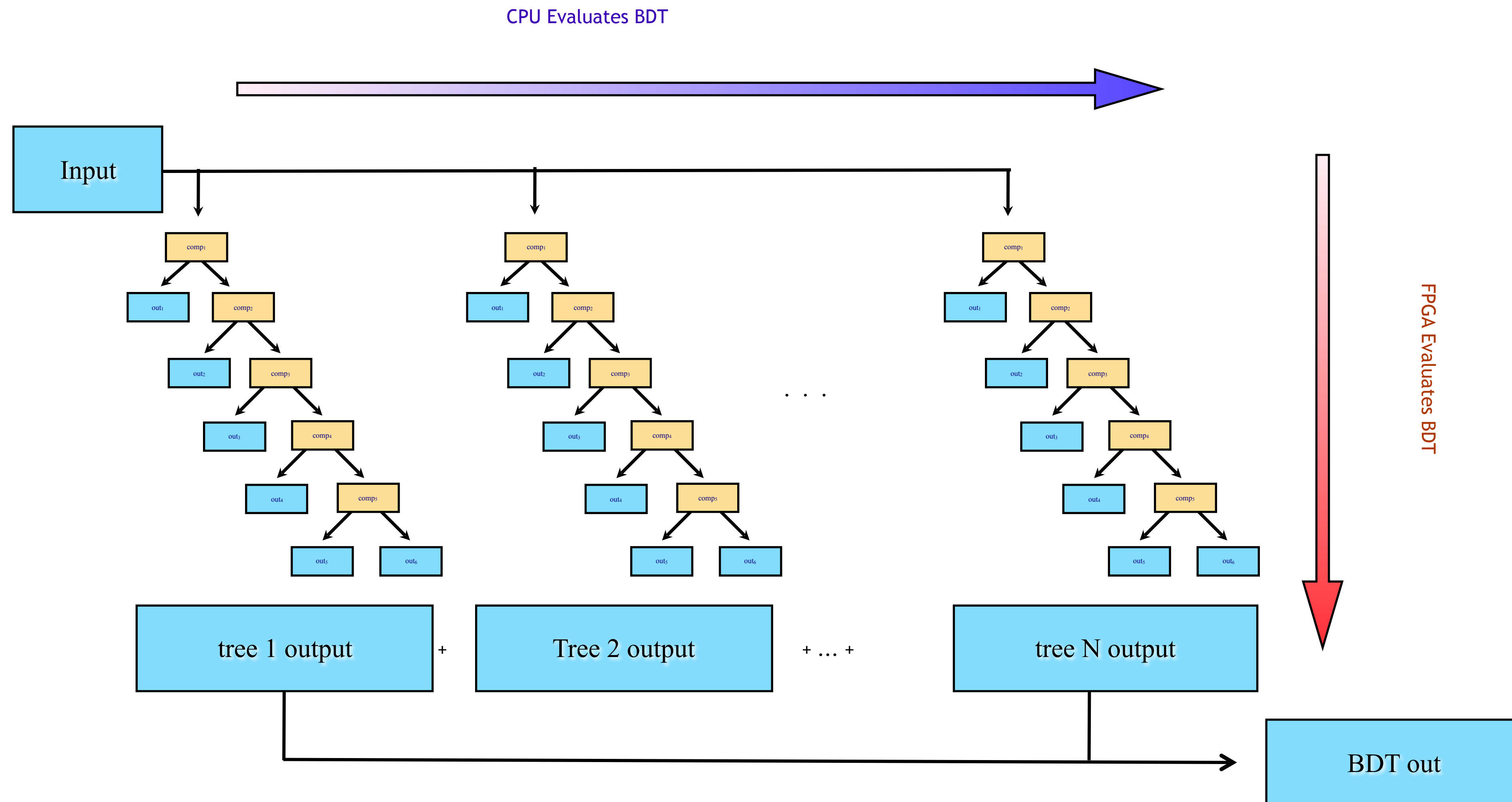
- › FPGAs have been around in trigger systems for a while
- › Latest large FPGAs give a huge amount of flexibility and are used in the LHC experiments
- › Revolutionized trigger systems since the logic (algorithms) do not need to be fixed when the board is produced
- › Can change the algorithms running in hardware, in light of better detector understanding, even physics discoveries



# BDTs in FPGA

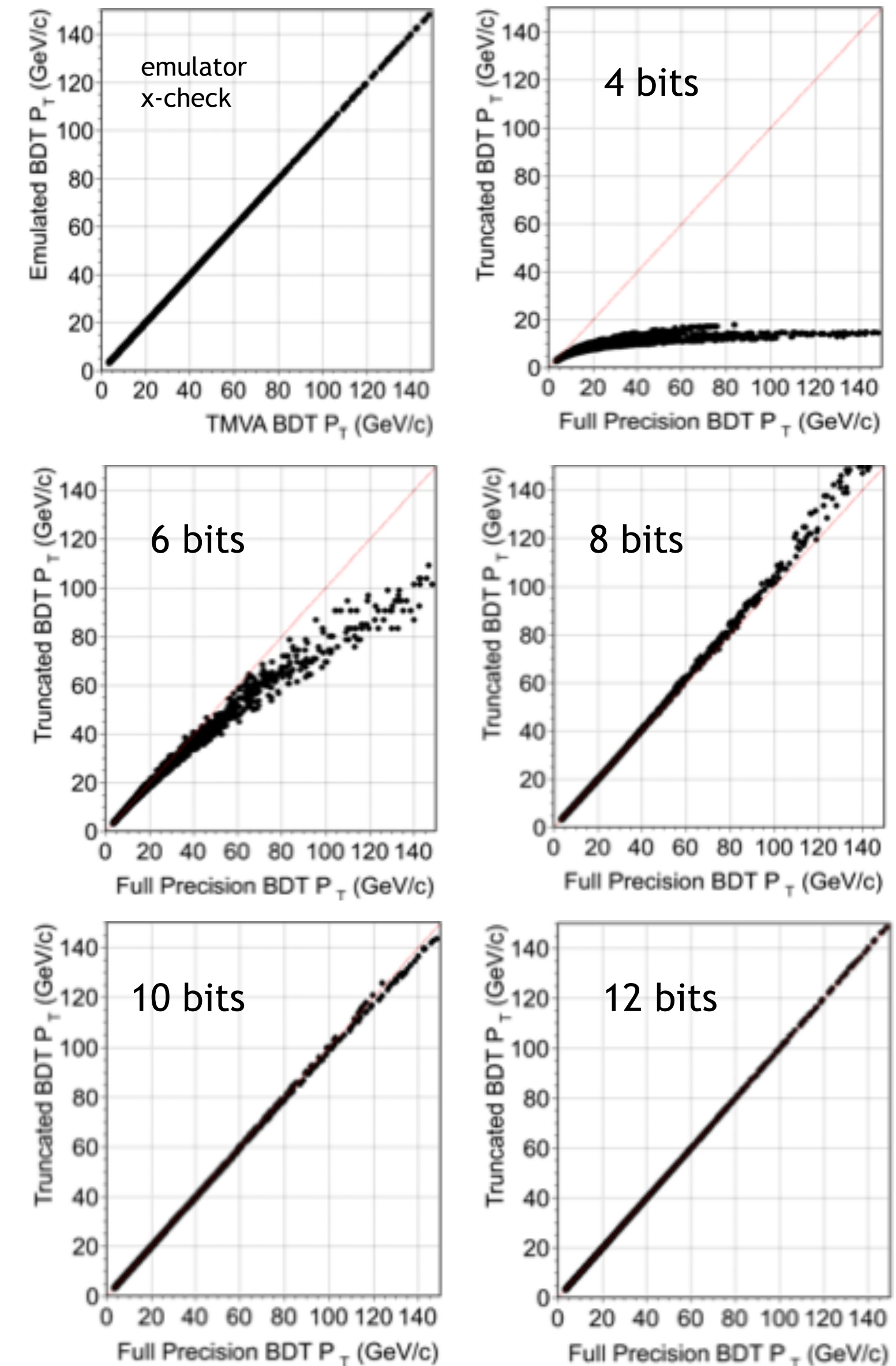
- › studied BDTs expecting good algorithms to generate complex trees
- › design usage for regression is exactly the opposite:
  - › complex trees tend to latch onto details
  - › use simple trees, but lots of them in BDT
  - › example TMVA “default”: approximately 20 nodes, 500 trees
- › basically: lots of very simple, fast evaluations (comparisons)
- › same input values -> all trees evaluated in parallel
- › closely matches the paradigm of FPGA computation
- › can we possibly evaluate our BDTs online at L1?

# BDTs in FPGA



# CMS BDTs in FPGA

- › implement BDTs in FPGA for  $p_t$ -assignment
- › reverse engineered for implementation in FPGA logic:
  - › parallel evaluation of all trees in forest
  - › inputs, outputs discretized:  
discretization of BDT output with 10+ bits yields  $p_T$  values almost indistinguishable from floating point computed values
- › “FPGA ready” BDT:
  - › 256 trees, 10 nodes/tree, output discretized to 10 bits
  - › bitwise reproduced by firmware emulator
  - › reproduces TDR to within 2% in relevant  $p_t$  range



# Neural networks speed-up



# Deep NN vs shallow NN

- › 1 million labeled points
- › shallow NN with one fully connected feed-forward hidden layer has 86% accuracy
- › deep NN (with pooling, convolutional layers) has 91% accuracy
- › it was proved that a network with a large enough single hidden layer of sigmoid units can approximate any decision boundary ([link](#))
- › what is the source of the quality difference?
- › Ba and Caruana: shallow NN are capable of learning the same function as deep NN

# Training shallow NN to mimic deep NN

Ba and Caruana propose the following idea:

- › train a state-of-the-art deep model
- › train a shallow model to mimic a deep model
- › the mimic model is trained using the model compression method
- › a shallow model with the same number of parameters as a deep net learn to mimic the last one with high fidelity -> deep net does not really have to be deep

# Model compression

Idea: train a compact model to approximate the function learned by a larger, more complex model

- › pass unlabeled data through the large, accurate model
- › collect the scores produced by that model -> synthetically labeled data
- › train the smaller mimic model on the synthetic labeled data

Thus, in principle, small NN could learn the more accurate function. The complexity of a learned model, and the size and architecture of the representation best used to learn that model, are different things.

# Mimic model training

- › directly take log probabilities values, called logits, from the deep model
- › logits make learning easier for mimic model:  
 $p = (2*10^{-9}, 5*10^{-5}, 0.9999)$  - cross-entropy minimization focuses on the third target)
- › logits are able to learn without suffering from the information loss (when go to probability space).
- › train regression model with standard error back-propagation and stochastic gradient descent with momentum by minimizing L<sub>2</sub> loss function:

$$\mathcal{L}(W, \beta) = \frac{1}{2T} \sum_t ||g(x^{(t)}; W, \beta) - z^{(t)}||_2^2,$$



# Speeding-up learning

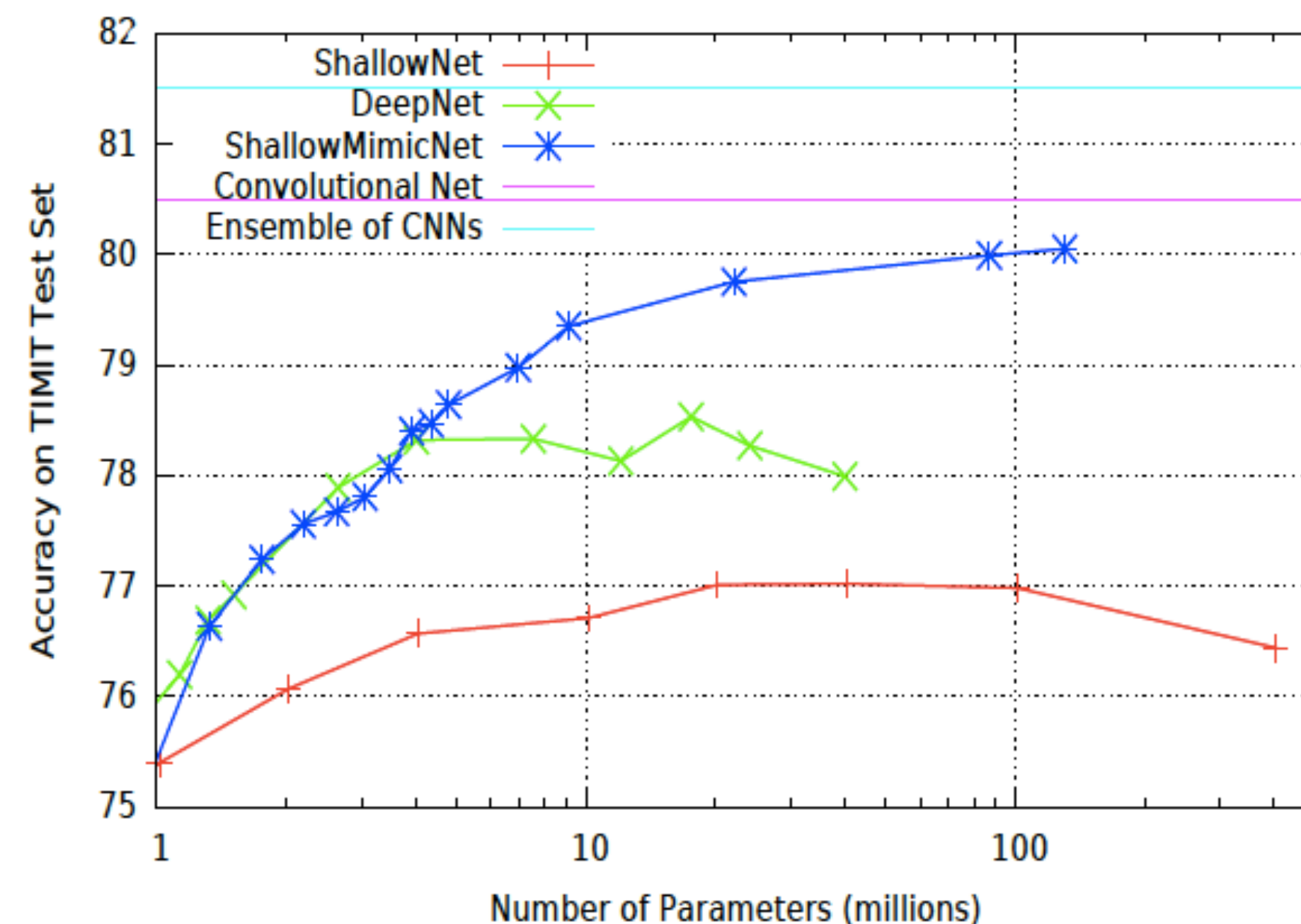
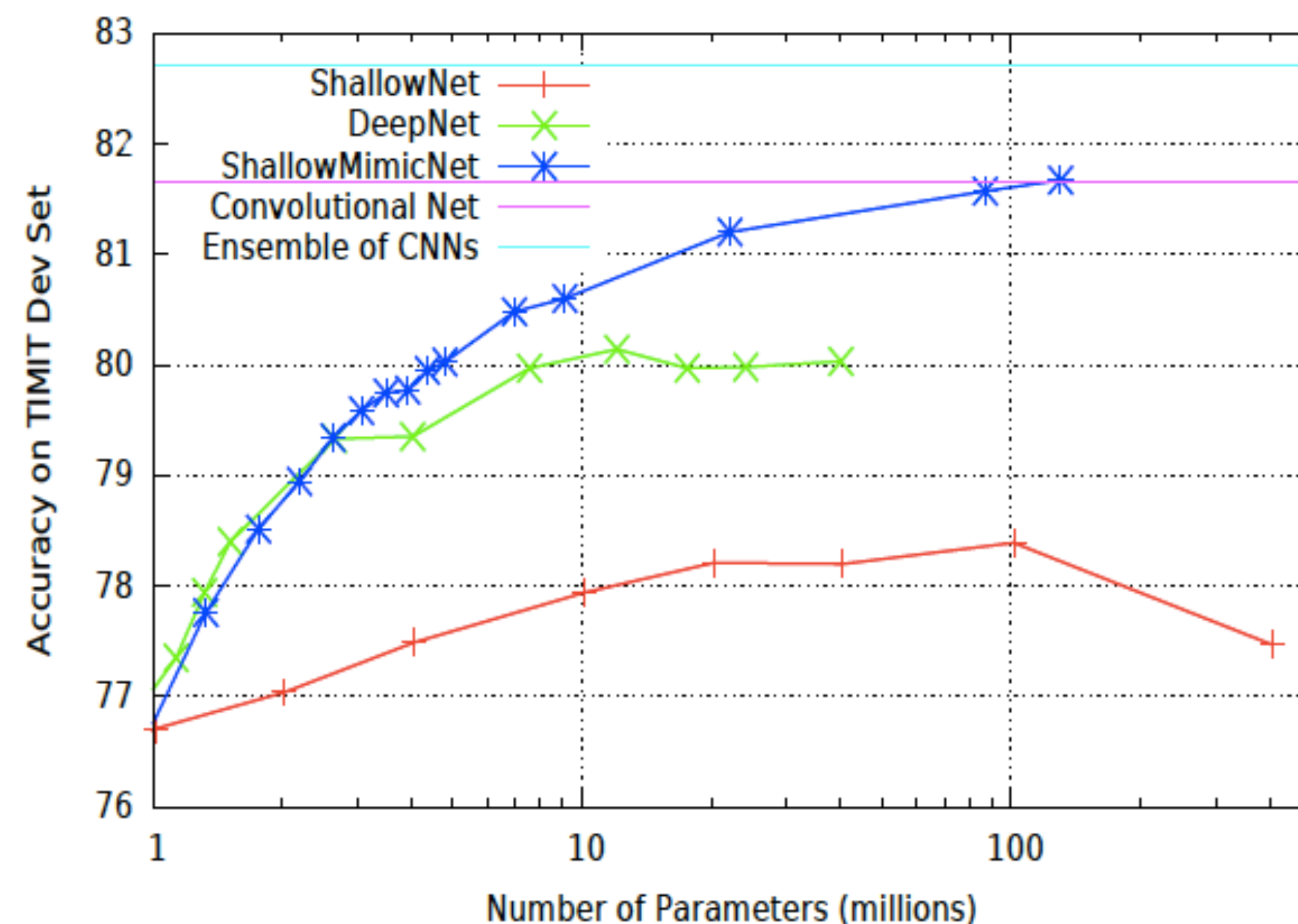
- › shallow NN has to have many neurons in a single layer -> large weight matrix
- › it is very slow to learn (multiple weeks) even with a GPU:
  - › highly correlated parameters -> gradient descent converges slowly
  - › matrix multiplication is very cost (the main part of computations)
- › introducing a bottleneck linear layer:

$$\mathcal{L}(U, V, \beta) = \frac{1}{2T} \sum_t ||\beta f(UVx^{(t)}) - z^{(t)}||_2^2$$

- › increase the convergence rate
- › memory  $O(k(N + M))$  instead of  $O(NM)$

# Experiments

- › model compression needs
  - › very large unlabeled set
  - › unlabeled samples do not fall on train points (teacher is overfit on them)
- › form an ensemble of several deep models trained on different training set



# Distillation approach

Hinton et al. proposed the following idea:

- › transfer the knowledge from the cumbersome model to a small model that is more suitable for deployment
- › approach similar to the mimic model and generalizing it
- › the relative probabilities of incorrect answers tell us a lot about how the cumbersome model tends to generalize
- › use the class probabilities produced by the cumbersome model as “soft targets” for training the small model
- › for ensembles as cumbersome model we can use an arithmetic or geometric mean of their individual predictive distributions as the soft targets

# Distillation approach

- › idea is to raise the temperature of the final softmax until the cumbersome model produces a suitably soft set of targets
- › use the same high temperature when training the small model to match these soft targets
- › using the original training set works well

$$q_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

- › take high temperature, convert the logits to probabilities, train the small model with the same temperature, after training it uses temperature one

# Distillation approach: objective functions

It was found to use weighted average of two different objective functions:

- › the cross entropy with the soft targets (is computed using the same high temperature in the softmax of the distilled model as was used for generating the soft targets from the cumbersome model)
- › the cross entropy with the correct labels (use the same logits in softmax of the distilled model but at a temperature of 1)

Use considerably lower weight for the second objective

# Matching logits

Each case in the transfer set contributes the gradient of cross-entropy:

$$H(p, q) = - \sum_x p(x) \log q(x).$$

If the cumbersome model has logits  $v_i$ , which produce soft target probabilities  $p_i$  and the transfer training is done at a temperature of  $T$ , this gradient is given by:

$$\frac{\partial C}{\partial z_i} = \frac{1}{T} (q_i - p_i) = \frac{1}{T} \left( \frac{e^{z_i/T}}{\sum_j e^{z_j/T}} - \frac{e^{v_i/T}}{\sum_j e^{v_j/T}} \right)$$



# Matching logits

If the temperature is high compared with the magnitude of the logits, we can approximate:

$$\frac{\partial C}{\partial z_i} \approx \frac{1}{T} \left( \frac{1 + z_i/T}{N + \sum_j z_j/T} - \frac{1 + v_i/T}{N + \sum_j v_j/T} \right)$$

If we now assume that the logits have been zero-meaned separately for each transfer case so that

$$\sum_j z_j = \sum_j v_j = 0$$

then

$$\frac{\partial C}{\partial z_i} \approx \frac{1}{NT^2} (z_i - v_i)$$

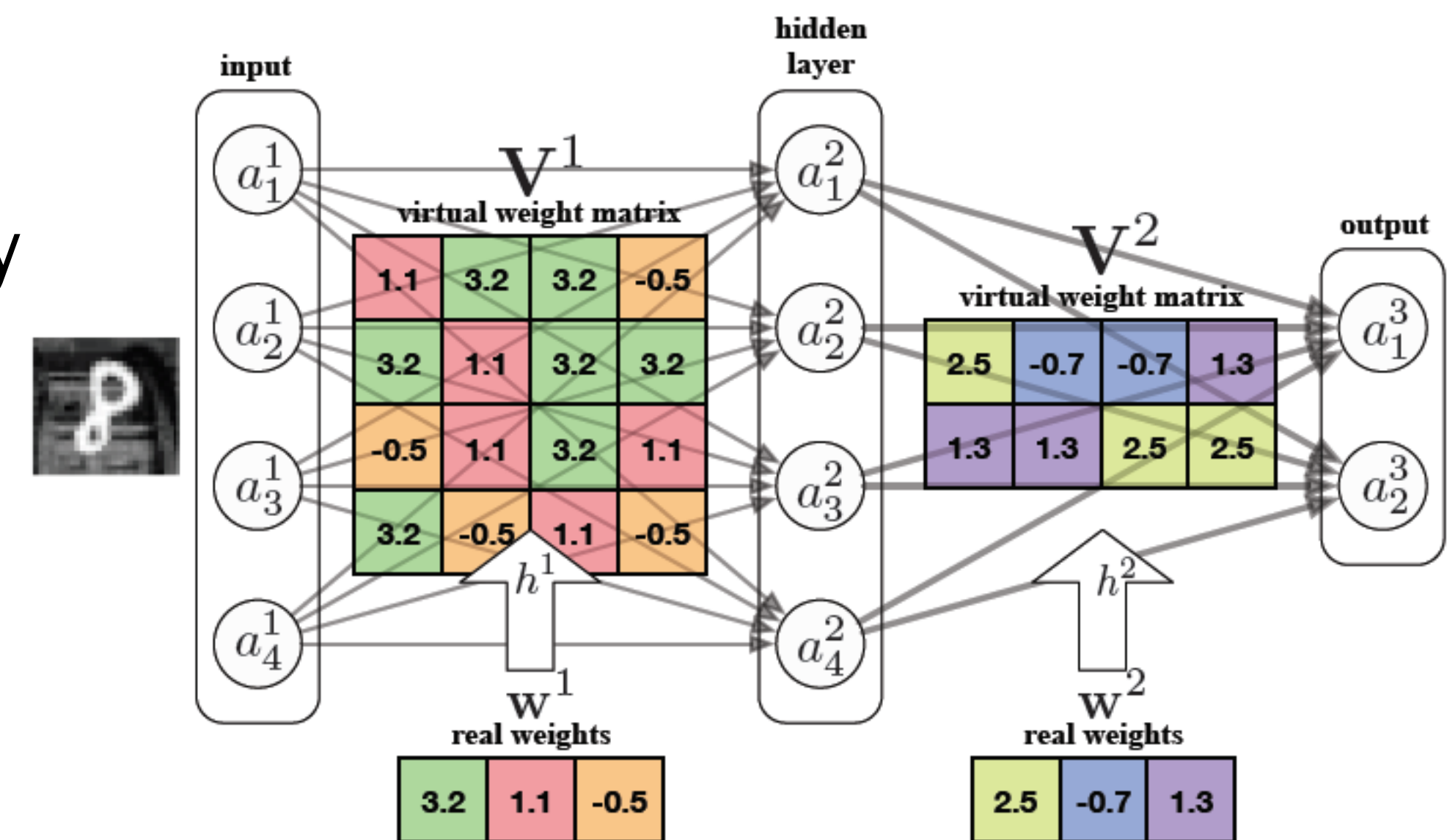
So in the high temperature limit, distillation is equivalent to minimizing MSE (mimic model) provided the logits are zero-meaned separately for each transfer case.

# HashedNets

Tie random subsets of weights using special  
hashing techniques (HashedNets)

(compression factor 8 for 2-layered NN on the MNIST)

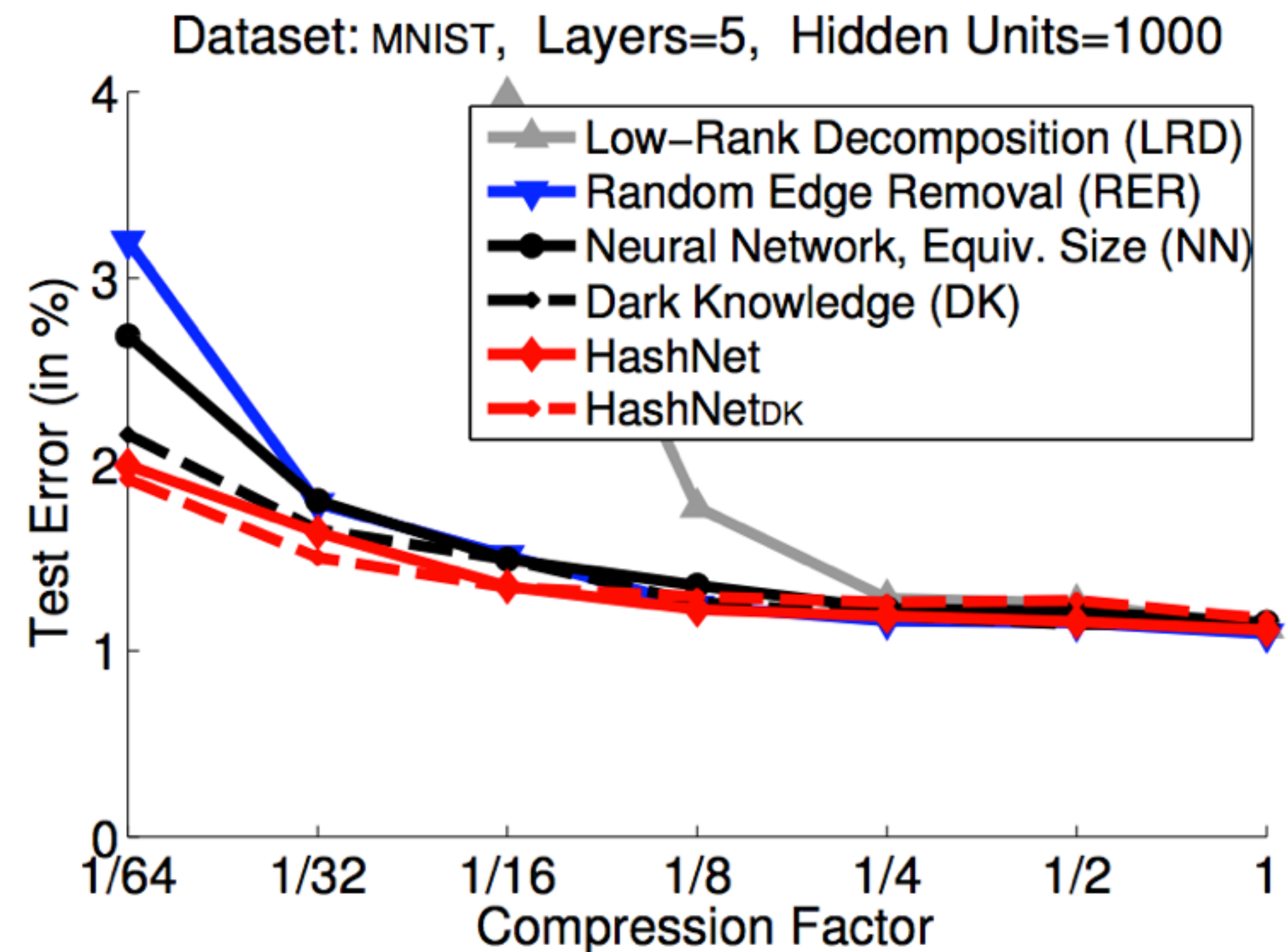
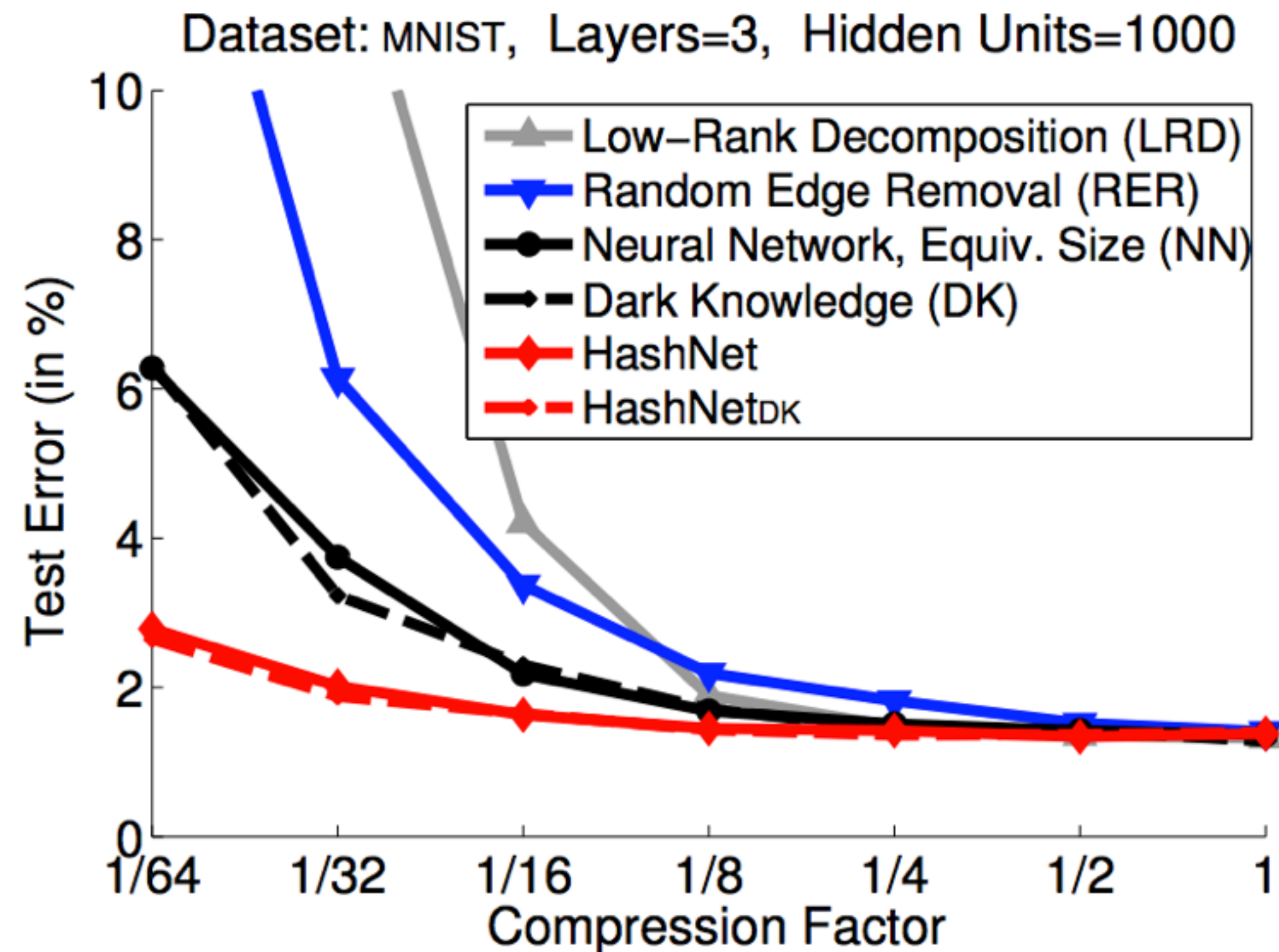
- › the shared weight of each connection is determined by a hash function
- › sparse feature vectors  $\rightarrow$  the number of hash collisions is minimized  $\rightarrow$  ReLU activation to minimize collisions
- › additional gradient ties the virtual weight matrix to the actual weights through the hashed map





# HashedNets

- › it is applicable to feed forward NN, recurrent NN and others
- › with the help of feature hashing, Vowpal Wabbit , a large-scale learning system, is able to scale to terafeature datasets
- › about GPU: specific challenge is to avoid non-coalesced memory accesses due to the pseudo-random hash functions



# Low-rank approaches

Construct low-rank representation of the weights matrices:

- › matrix factorization
- › singular value decomposition
- › tensor train decomposition (TensorNets)  
compression factor 7

# TensorNets

- › treat weight matrix as a multi-dimensional tensor
- › apply the Tensor Train (TT) decomposition algorithm:

A  $d$ -dimensional array (tensor)  $\mathcal{A}$  is said to be represented in the *TT-format* [17] if for each dimension  $k = 1, \dots, d$  and for each possible value of the  $k$ -th dimension index  $j_k = 1, \dots, n_k$  there exists a matrix  $\mathbf{G}_k[j_k]$  such that all the elements of  $\mathcal{A}$  can be computed as the following matrix product:

$$\mathcal{A}(j_1, \dots, j_d) = \mathbf{G}_1[j_1] \mathbf{G}_2[j_2] \cdots \mathbf{G}_d[j_d]. \quad (1)$$

- › construct bijection from matrix/vector into  $d$ -dimensional tensor, present it in TT-format -> TT-matrix/TT-vector

- › memory: matrix representation  $\prod_{k=1}^d n_k$  ; tensor representation  $\sum_{k=1}^d n_k r_{k-1} r_k$

# TensorNets: TT-layer

Fully-connected layers apply a linear transformation to the input vector  $\mathbf{x}$ :

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b},$$

- › the TT-layer is a fully-connected layer with the weight matrix stored in the TT-format
- › TT-ranks allow to control the number of parameters
- › TT-layer transforms a  $d$ -dimensional tensor (TT-vector) to the  $d$ -dimensional tensor (TT-vector):

$$\mathcal{Y}(i_1, \dots, i_d) = \sum_{j_1, \dots, j_d} \mathbf{G}_1[i_1, j_1] \dots \mathbf{G}_d[i_d, j_d] \mathcal{X}(j_1, \dots, j_d) + \mathcal{B}(i_1, \dots, i_d).$$

- › time:  $O(dr^2m \max \{M, N\})$

# TensorNets: Learning

- › gradient computation and then converting it to the TT-format for weight matrix update are not a way
- › directly compute the gradient of the loss function w.r.t. the cores of the TT-representation of weight matrix
- › dynamic programming is used to compute gradients w.r.t. the cores



# TensorNets: time and memory

Operation	Time	Memory
FC forward pass	$O(MN)$	$O(MN)$
TT forward pass	$O(dr^2 m \max\{M, N\})$	$O(r \max\{M, N\})$
FC backward pass	$O(MN)$	$O(MN)$
TT backward pass	$O(d^2 r^4 m \max\{M, N\})$	$O(r^3 \max\{M, N\})$

Table 1: Comparison of the asymptotic complexity and memory usage of an  $M \times N$  TT-layer and an  $M \times N$  fully-connected layer (FC). The input and output tensor shapes are  $m_1 \times \dots \times m_d$  and  $n_1 \times \dots \times n_d$  respectively ( $m = \max_{k=1\dots d} m_k$ ) and  $r$  is the maximal TT-rank.

Type	1 im. time (ms)	100 im. time (ms)
CPU fully-connected layer	16.1	97.2
CPU TT-layer	1.2	94.7
GPU fully-connected layer	2.7	33
GPU TT-layer	1.9	12.9

Table 3: Inference time for a  $25088 \times 4096$  fully-connected layer and its corresponding TT-layer with all the TT-ranks equal 4. The memory usage for feeding forward one image is 392MB for the fully-connected layer and 0.766MB for the TT-layer.

# TensorNets: perspective

- › Restricting the TT-ranks of the weight matrix (in contrast to the matrix rank) allows to use much wider layers potentially leading to the greater expressive power of the model.
- › Compared to the Tucker format and the canonical format, the TT-format is immune to the curse of dimensionality and its algorithms are robust.
- › Compared to the HashedNets with the same architecture and with the compression factor 64 (12720 parameters) TensorNets have 1.6% test error (all ranks are set 8), while HashedNets have 2.79%.

# Binarized neural nets

- › binary weights: advantage from a hardware perspective
- › binary activations
- › bit-wise operations:  $a_1 + = \text{popcount}(\text{xnor}(a_0^{32b}, w_1^{32b}))$
- › binary convolutional NN can lead to binary convolution kernel repetition (dedicated hardware could reduce the time complexity by 60%)



# BNNs: binarization

Binarization functions:

› deterministic:

$$x^b = \text{Sign}(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{otherwise,} \end{cases}$$

› stochastic (harder implement: requires the hardware to generate random bits when quantizing):

$$x^b = \begin{cases} +1 & \text{with probability } p = \sigma(x), \\ -1 & \text{with probability } 1 - p, \end{cases}$$

where  $\sigma$  is the “*hard sigmoid*” function:

$$\sigma(x) = \text{clip}\left(\frac{x+1}{2}, 0, 1\right) = \max\left(0, \min\left(1, \frac{x+1}{2}\right)\right).$$

# BNNs: gradients

- › real-valued gradients of the weights are accumulated in real-valued variables (they are likely required for Stochastic Gradient Descent to work at all)
- › it is important to keep sufficient resolution for accumulators
- › BNNs training can be seen as a variant of dropout (form of regularization)
- › use straight-through estimator for the gradient:

$$q = \text{Sign}(r),$$

and assume that an estimator  $g_q$  of the gradient  $\frac{\partial C}{\partial q}$  has been obtained (with the straight-through estimator when needed). Then, our straight-through estimator of  $\frac{\partial C}{\partial r}$  is simply

$$g_r = g_q 1_{|r| \leq 1}. \quad (4)$$

- › constrain each real-valued weight between -1 and 1 by clipping
- › when using a weight, quantize it using deterministic binarization function

# BNNs: optimization

Replace multiplications by shift operations (accuracy is the same):

- › batch normalization -> shift batch normalization
- › ADAM learning rule -> shift-based ADAM rule

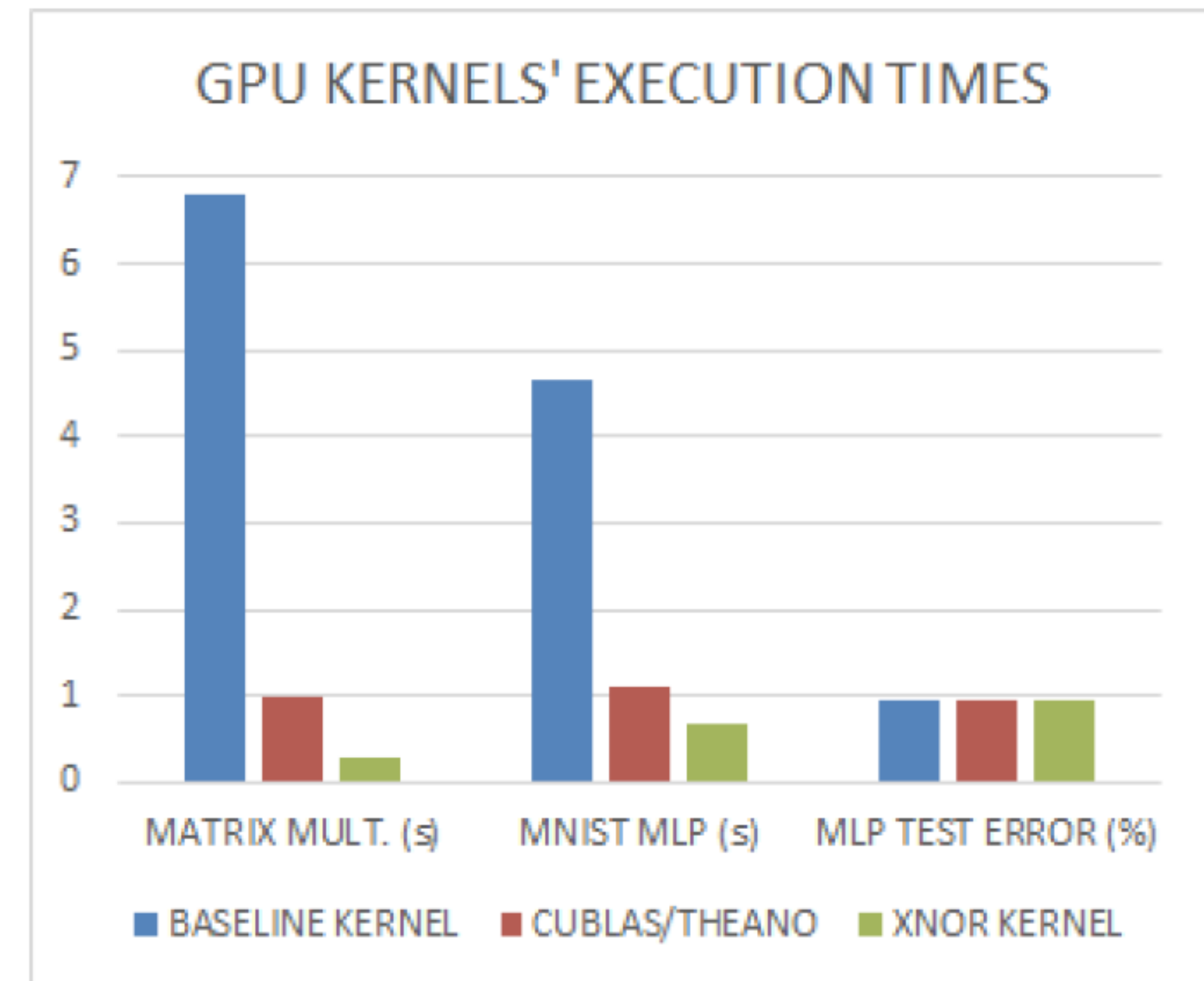
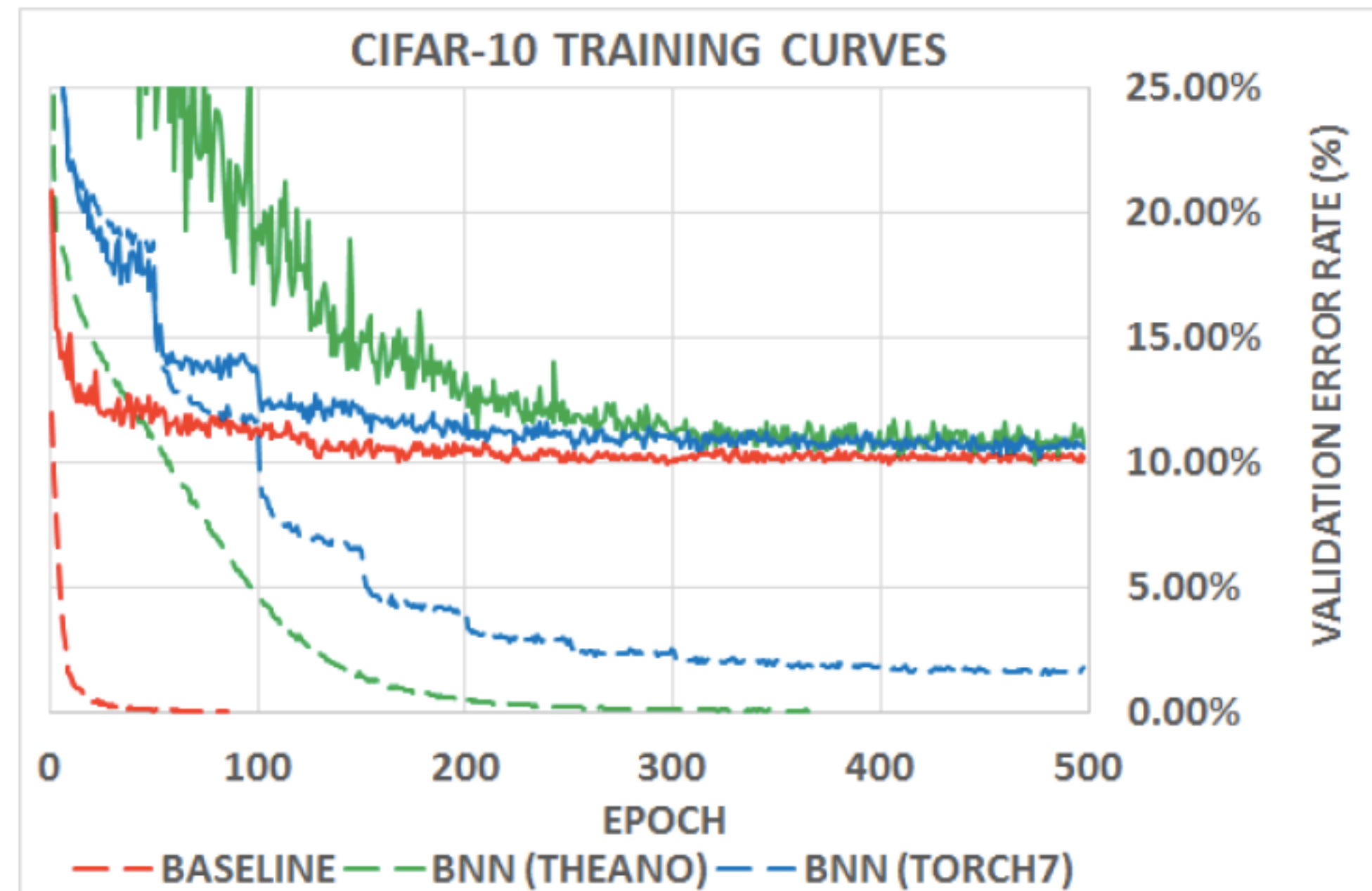
All the layers input are binary except first one:

- › handle continuous-valued inputs as fixed point numbers, with ***m***-bits of precision (for example, 8-bit fixed point inputs):

$$s = x \cdot w^b$$
$$s = \sum_{n=1}^8 2^{n-1} (x^n \cdot w^b),$$

- › use this trick during binary matrix-by-vector multiplication (most of the 32-bit floating point multiply-accumulations are replaced by 1-bit XNOR-count operations)

# BNNs: experiments



Seven Times Faster on GPU at Run-Time

# Summary

- › Nowadays trigger system uses ML
- › Trigger system can be considered as Markov decision process
- › Speeding-up of prediction is necessary in HEP, Web search, etc.
- › Discussed different approaches to speed-up BDTs and ANNs

Thanks for attention

# Contacts

Likhomanenko Tatiana  
researcher-developer



[antares@yandex-team.ru](mailto:antares@yandex-team.ru), [tatiana.likhomanenko@cern.ch](mailto:tatiana.likhomanenko@cern.ch)