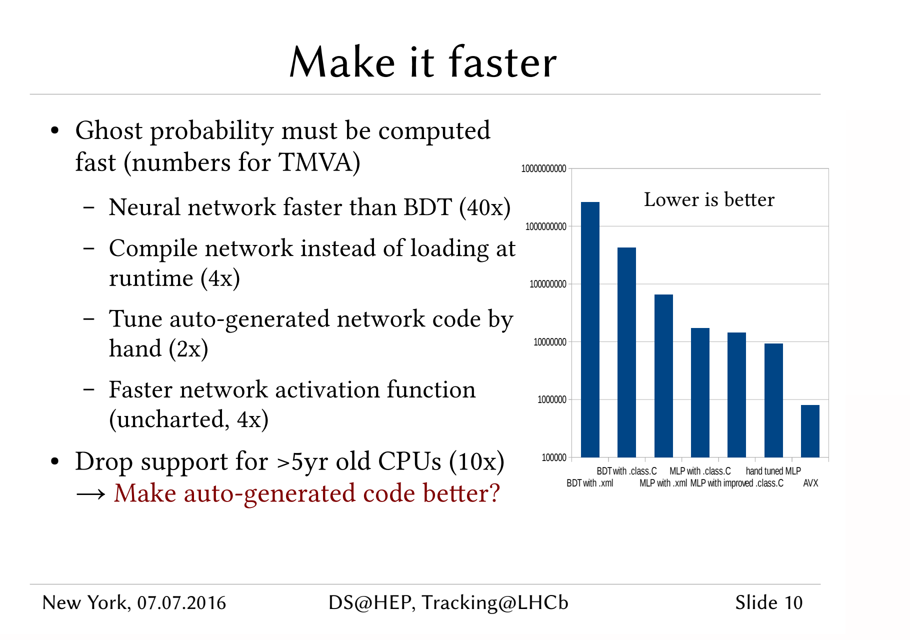
# we need a fast TMVA

Paul Seyfert

INFN Milano Bicocca

7th October 2016



# Make it faster

- Ghost probability must be computed fast (numbers for TMVA)
  - Neural network faster than BDT (40x)
  - Compile network instead of loading at runtime (4x)
  - Tune auto-generated network code by hand (2x)
  - Faster network activation function (uncharted, 4x)
- Drop support for >5yr old CPUs (10x)
  ⟶ Make auto-generated code better?

Lower is better

New York, 07.07.2016 DS@HEP, Tracking@LHCb Slide 10

Sascha Stahl, PS at DS@HEP workshop 2016

# disclaimer I

## what i have to offer

- This is somewhat a review of TMVA evaluation speed observations I made at some point in the preparation of 2015 data taking for LHCb (only MLP)
- some things already went into TMVA
- some are ready as pull request (though review never harms)
- some will be hard to put into TMVA
- maybe we don't even want to:
  - more recent developments anyhow better?
  - future computing models?
    (single core was a MUST. I do one evaluation at a time, batch evaluation might be better ultimately)
  - some tuning steps might become nightmares in general
- code and stuff at
  - blog post
  - github pseyfert/tmva-mlp (the one network I optimised, go through the commit history!)
  - Our reconstruction code (. . . if you really want to read the original)
  - what I managed to pack into a pull request
  - what's merged already

# disclaimer II

## why I did what I did

- LHCb track reconstruction uses a NN to distinguish fake tracks from real tracks
- was deployed in the software trigger for 2015 (every bit of timing counts)
- the NN from 2012 was way too slow due to computation of the input variables
- so I was profiling the entire algorithm
- eventually I reached a point where the NN evaluation was the bottleneck (probably I could've stopped here)
- trigger runs several processes on each CPU (→ no multithreading for the individual process)
- I only looked at evaluations, no training.

## how I did it

- code profiling in LHCb reconstruction is done with callgrind
  (I sometimes wonder why, because all computer scientists outside of HEP I know use different tools with different behaviour)
- I also experimented with the printout of gcc when enabeling autovectorisation
- google paper on NN evaluation speed optimisation

# which method

- just ran TMVAClassificationApplication with one method at a time (same events, same number of executions)
- minor adjustments to the code to run from the standalone .class.C

| | |
|---|---|
| BDT with .xml | 2 627 196 471 |
| BDT with .class.C | 427 128 646 |
| MLP with .xml | 65 365 395 |
| MLP with .class.C | 17 391 466 |

## finding

- expected MLP to be faster than BDT (simple float point math vs. many ifs and branch misses)
- did not expect the .xml vs .class.C difference to be so large! (for MLP I think it makes sense in hindsight, for BDT I have no explanation)
- to be honest: went for .class.C because I didn't know how to tell our build framework (CMT) back in 2012 how to change link flags. .class.C was almost trivial in include.

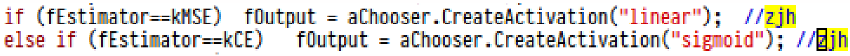
# Activation functions I

- After tuning long enough, calls to tanh appeared in the profiles
  - tanh known as super slow function in LHCb reconstruction (should be exterminated by now)
  - friend in human brain project confirmed, where they have CPU neurons, they cannot use anything related to exp
- naive me looked up sigmoid functions and went for $\frac{x}{\sqrt{1+x^2}}$ (sqrt is absurdely fast and available in SIMD units)
- Helge then added ReLU to TMVA (even faster)

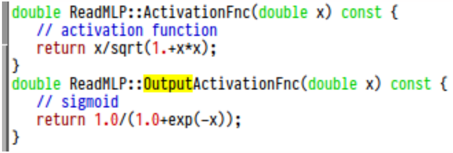
| function | default compiler options | AVX vectorisation by hand |
|---|---:|---:|
| tanh | 19,355,124,355 | n/a |
| $\frac{1}{1+e^{-x}}$ | 21,140,125,632 | n/a |
| $\frac{x}{\sqrt{1+x^2}}$(*) | 415,121,741 | 195,121,939 |
| $\frac{x}{1+\|x\|}$ | 395,121,798 | 195,104,759 |
| $\max(0,x)$ | 155,095,875 | 115,095,891 |

# Activation functions II

- output layer activation function depends on estimator type
  ⇒ remained something with exp for cross entropy
- Fine for training, but in application I'm spending CPU on a monotoneous transformation of the response (we do our own rarity transformation anyhow afterwards)
- changed by hand . . .



```
if (fEstimator==kMSE)  fOutput = aChooser.CreateActivation("linear");  //zjh
else if (fEstimator==kCE)   fOutput = aChooser.CreateActivation("sigmoid"); //zjh
```



```
double ReadMLP::ActivationFnc(double x) const {
   // activation function
   return x/sqrt(1.+x*x);
}
double ReadMLP::OutputActivationFnc(double x) const {
   // sigmoid
   return 1.0/(1.0+exp(-x));
}
```

- discussing at data science workshop with data scientists, output layer activation functions not considered too expensive
- changing the output layer activation function between training and application might cause more trouble than it's worth

# step by step

## double → float

float precision faster than double (mostly . . . friends in HBP reported there are exceptions)

⇒ converted all doubles to floats

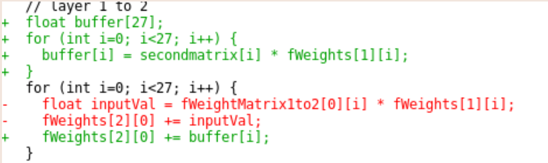
⇒ 4 % speedup (wrt. initial version)

## remove `fLayerSize[i]`

anyhow constant, hard code them

⇒ 8 % speedup (wrt. initial version)

## reduce output layer weight matrix to vector and reorder loops



```
    // layer 1 to 2
+   float buffer[27];
+   for (int i=0; i<27; i++) {
+     buffer[i] = secondmatrix[i] * fWeights[1][i];
+   }
    for (int i=0; i<27; i++) {
-     float inputVal = fWeightMatrix1to2[0][i] * fWeights[1][i];
-     fWeights[2][0] += inputVal;
+     fWeights[2][0] += buffer[i];
    }
```

(hint by vectorisation messages from compiler)

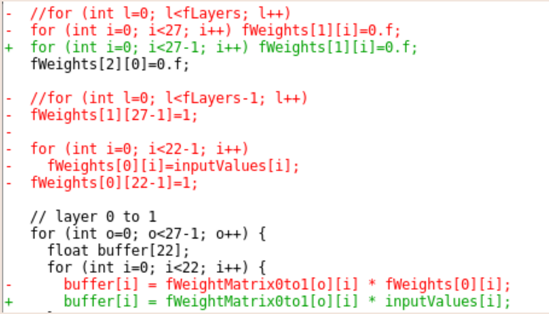
⇒ 14 % speedup (wrt. initial version)

# step by step

## output layer activation function (discussed before)

⇒ 16 % speedup (wrt. initial version)

## reduce resetting and copying variables



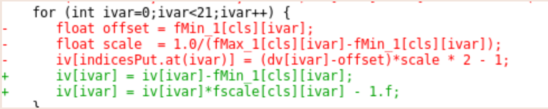
```
-  //for (int l=0; l<fLayers; l++)
-  for (int i=0; i<27; i++) fWeights[1][i]=0.f;
+  for (int i=0; i<27-1; i++) fWeights[1][i]=0.f;
   fWeights[2][0]=0.f;

-  //for (int l=0; l<fLayers-1; l++)
-  fWeights[1][27-1]=1;
-
-  for (int i=0; i<22-1; i++)
-    fWeights[0][i]=inputValues[i];
-  fWeights[0][22-1]=1;

   // layer 0 to 1
   for (int o=0; o<27-1; o++) {
     float buffer[22];
     for (int i=0; i<22; i++) {
-      buffer[i] = fWeightMatrix0to1[o][i] * fWeights[0][i];
+      buffer[i] = fWeightMatrix0to1[o][i] * inputValues[i];
```

⇒ 16 % speedup (wrt. initial version)

## rarrange input variable normalisation



```
    for (int ivar=0;ivar<21;ivar++) {
-       float offset = fMin_1[cls][ivar];
-       float scale  = 1.0/(fMax_1[cls][ivar]-fMin_1[cls][ivar]);
-       iv[indicesPut.at(ivar)] = (dv[ivar]-offset)*scale * 2 - 1;
+       iv[ivar] = iv[ivar]-fMin_1[cls][ivar];
+       iv[ivar] = iv[ivar]*fscale[cls][ivar] - 1.f;
    }
```

also avoid copying from one vector to another (`indicesGet/Put`)

also overwrite input vector !!! (changed interface, no const)

also remove check of vector length !!!

⇒ 67 % speedup (wrt. initial version)

## more rarrangement of linear transformations

⇒ 67 % speedup (wrt. initial version)

# challenge accepted

**writing SSE3 intrinsics code**

⇒ 93 % speedup (wrt. initial version)
I am still amazed that the compiler couldn't do that

**writing AVX intrinsics code**

⇒ 95 % speedup (wrt. initial version)

# some words of caution

- SSE 3 and AVX code didn't go into production (didn't want to write machine dependent code and introduce overhead code to determine the architecture)
- makes the code very dependent on number of neurons/variables
  - what's the remainder of nodes divided by four or eight
  - what's $\log_2$ of it (for "horizontal adding")
- fun to do it for one network (challenging to do as much as possible in `__m128` variables)
- write vectorised versions of activation function
- make the most use of each `_mm_hadd_ps` call

### but to be serious

This should happen in some math library. I'm surprised Eigen didn't beat my code, though might be inefficient use of interfaces (esp. for activation function)

# Conclusion

- only dealt with good old MLP
- parallel implementations might be the future, but single core SIMD is the NOW
- modular input transformations a bit of a barrier for making all tweaks generic
- fixed size arrays instead of vectors for interface, removed const-ness of input variables not trivial either
- scalability of `.class.C` networks??? (I'm not sure if I want to have many hard coded networks in my compiled code. reading from `.xml` seems more maintainable on the long run)

something completely different

# automatised tmva response adding to ttrees

- github pseyfert/tmva-branch-adder (advertisement on roottalk)
- TMVA reader asks user to spell out order of input variables
  ✓ good sanity check
- but over the years it became anoying to write loops over ntuples by hand, in which branch variables get handed over to TMVA just to fill one more branch.
  ? but wait ... if the reader knows the name of the input variables ... it can also just get them itself

→ copy&pasted the variable-name checking code

→ feed variables into TTreeFormulas (such that also formulas get parsed)

⇒ add response to tree as new branch

## future of this tool?

- I use it already
- so far a few limitations (aimed for command line, no good documented c++ interface, python experimental, cannot evaluate more than one MVA at a time)
- paranoia tests in place: never overwrite or update files (don't want to be responsible for files getting corrupted)