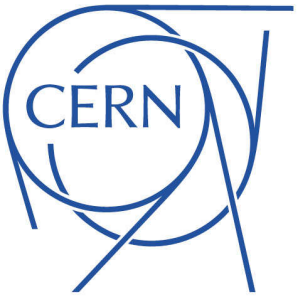
# Deep learning in TMVA

## Benchmarking Benchmarking TMVA DNN
## Integration of a Deep Autoencoder

Marc Huwiler
marc.huwiler@windowslive.com
Supervised by Lorenzo Moneta and Sergei Gleyzer

September 1, 2017




**Abstract**

The TMVA library in ROOT is dedicated to multivariate analysis, and in particular offers numerous machine learning algorithms in a standardized framework. It is widely used in High Energy Physics for data analysis, mainly to perform regression and classification. To keep up to date with the state of the art in deep learning, a new deep learning module was being developed this summer, offering deep neural network, convolutional neural network, and autoencoder. TMVA did not have yet any autoencoder method, and the present project consists in implementing the TMVA autoencoder class based on the deep learning module. It also includes some benchmarking performed on the actual deep neural network implementation, in comparison to the Keras framework with Tensorflow and Theano backend.

# Contents

# 1 Introduction

The aim of this report is to cover the work done during my Summer Student programme at CERN. It focuses on deep learning, which is a field of machine learning that use stacked layers of non-linear processing units to learn features or representations of the data. In particular, this project focuses on *Deep Neural Networks* (DNN) and *Deep Autoencoders* (DAE), where the processing units are artificial neurons, introduced in sections 4.1 and 4.2.

# 2 ROOT and TMVA

ROOT is the data analysis framework for High Energy Physics (HEP) developed at CERN, and used by most of the experiments to store and process the data. It is written in c++, and offers the possibility to run scripts fully interpreted. TMVA (Toolkit for Multivariate Analysis) is one of ROOT's libraries, dedicated to multivariate analysis, and offering numerous machine learning methods in a standardized framework. The main tasks it is dedicated to are classification and regression. Its paradigm is to book the methods thorough a *Factory*, that takes care of the data loading and sampling as well as feeding it properly to the given method. Several methods can be instantiated through the factory for a given task, trained , tested and evaluated at once. To keep up to date with the state of the art, a group of *Google Summer of Code* students has been implementing a new deep learning module for TMVA, offering a series of deep learning algorithms, including a new DNN as well as a DAE implementation.

# 3 Project details and goals

The first part of this project consists in Benchmarking the actual deep neural network implementation against the one of Keras library, with both Theano and Tensorflow backends. In a second time, a study of the possible integration of an autoencoder is presented, and eventually one of them was implemented and the results are shown.

# 4 Deep learning in brief

## 4.1 Neural networks

A neural network is a conceptual object made of artificial neurons, arranged into layers. On figure 1, each circle represents an artificial neuron. The first layer is called the *input layer*, and simply consists of the inputs (from the dataset). The last layer is called the *output layer*, and usually consists of one neuron in case of binary classification and $n$ neurons in case of n-class classification. The intermediate layers are called *hidden layers*, and a network having one or several of them is called a *deep neural network* (DNN).

*Deep learning* is the field of machine learning focusing on deep neural networks and deep autoencoders.

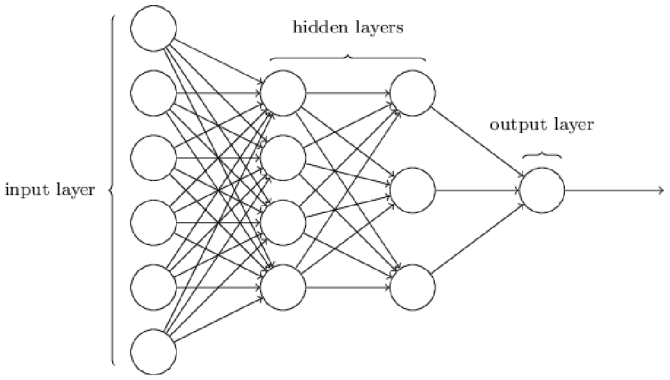



Figure 1: Scheme of a neural network.

An artificial neuron is a logical unit (see scheme 2) whose output is given by a function of its inputs. Each neuron of a given layer takes as inputs the outputs of every neuron in the previous layer. Each input ($x_i$) is multiplied by a weight ($\omega_i$) and added up together. Eventually, a bias is added to the sum. The output is given by a function $\alpha$ of the sum of inputs multiplied by their respective weight and the bias:

$$output = \alpha\big(\vec{\omega} \cdot \vec{x} + b\big) \tag{1}$$

The function alpha can typically be linear, tanh, sigmoid, softmax, and some other function that is odd around zero.

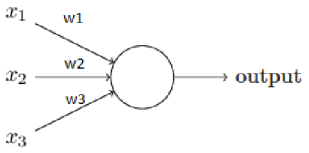


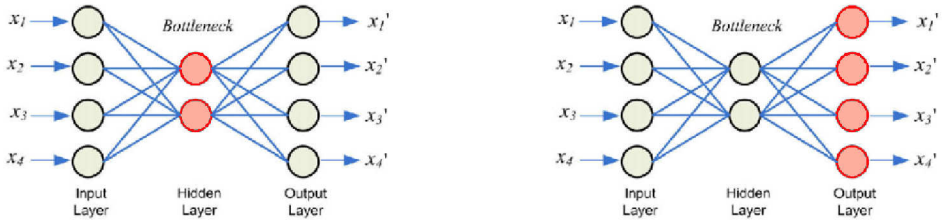

Figure 2: Scheme of an artificial neuron.

## 4.2 Autoencoder

An autoencoder (AE) is a special shaped neural network with a bottleneck, and an equal number of *decoded outputs* (neurons in the last layer) as inputs. It is trained to reproduce decoded outputs as close as possible to the inputs. This ensures the layer in the bottleneck called *encoded outputs* holds a compressed representation of the data, according to the patterns and correlations the network was able to find in the data.

## 4.3 Neural network training

The neural networks and autoencoders are algorithms that undergo a "training", where they learn patterns inherent to the data. This training phase is performed upon so called




(a) The layer in the bottleneck holds a compressed representation of the data, according to the pattern the network learned from it.

(b) The decoded outputs are aimed to be as close as possible to the inputs, forcing the compressed outputs to learn an accurate representation of the data.

Figure 3: Scheme of an autoencoder. It is a neural network with a bottleneck (compressed outputs), and the last layer (decoded outputs) has an equal number of neurons to the inputs.

*training data*. Then, the networks can be used to predict outcomes (i.e. classification, regression or dimensionality reduction) on new data. The learning is done by adjusting the weights and biases while iterating over the training data. An error function is defined (deviation from targets/class in case of supervised learning and between the decoded outputs and the inputs in case of AE). An algorithm called *optimizer* adjusts the weights and biases of each neuron in order to minimise the error function. The training can end either when it converges, or after reaching the upper bound in training epochs.

# 5 Benchmarking TMVA DNN against PyKeras

## 5.1 Methodology

In order to run representative benchmarking and have a meaningful comparison, it is essential to establish a common basis between the two frameworks that are confronted. This was partly eased by the existence in TMVA of the PyKeras method. It is a wrapper for the Keras interface, that is run from within TMVA in the same way as any other method. Thus, the data sampling and preprocessing undergoes the same process for both frameworks. Only the parameters for the neural network layout and for the training had to be taken care of in view of similar run conditions.

### 5.1.1 Common input parameters

Table 1 shows a list of parameters in TMVA DNN with their counterpart in PyKeras. These have the same effect, except the possible values might differ between the two frameworks. There is also a series of parameters that have no counterpart in the other framework, and listed in table 2. For these, the value was set to the hardcoded one of the other framework or tuned to a value to produce a behaviour as close as possible to the other framework.

| **TMVA DNN** | **PyKeras** |
|---|---|
| network layout | network layout |
| WeightInitialization | initializer |
| ErrorStrategy | loss |
| LearningRate | lr |
| Momentum | momentum |
| ConvergenceSteps | TriesEarlyStopping |
| DropConfing | Dropout |

Table 1

| **TMVA DNN** |
|---|
| Repetitions |
| TestRepetitons |
| WeightDecay |
| regularizations |
| several different trainings |

| **PyKeras** |
|---|
| NumEpochs |
| LearningRateSchedule |

Table 2

For instance, Keras offers the option to set an upper bound to the number of epochs, which is not available in TMVA DNN; thus this number was set very high, in order not to be reached and have *TriesEarlyStopping* terminating the training.

### 5.1.2 Workflow

To provide the most flexible interface (without depending on a specific programming language or environment), the whole process of benchmarking was chosen to be run through a batch script. The workflow is presented in figure 4.

Two macros, one for TMVA DNN and one for PyKeras were written in C++ respectively in Python. They take an input file as argument where the common parameters are set. The framework dependent parameters are hardcoded in the macros. If needed, they can be changed. The batch script generates the input file and feeds it into both macros. The two macros perform the training, testing and evaluation, compute the benchmarks, and write them into a common output file, which path is defined in the input file. The output file uses CSV format with space as separator, and four columns: the first is the name of the method (TMVAdnncpu or pykeras), the second contains the ROC (receiver operating characteristic) curve integral value, the third the CPU time for training and the last the real training time of the method. They can then be plotted using any software able to read simple CSV files.

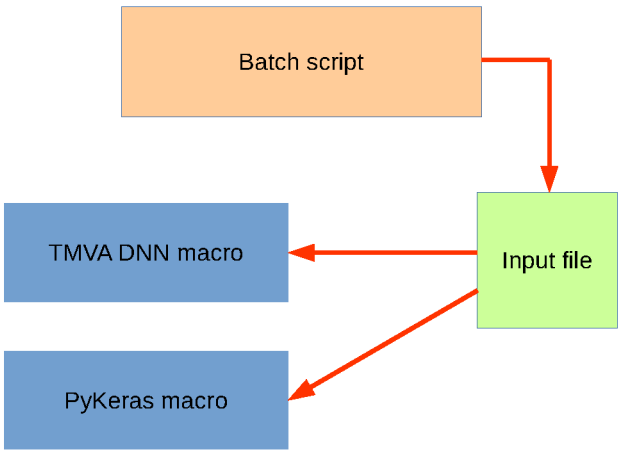



Figure 4: Scheme of the benchmarking workflow: a batch script generates an input file with the given parameters, and feeds it to the two macros for both TMVA DNN and PyKeras methods. The macros then write the ROC curve integral, CPU time and real time in a common output file. These steps can be run iteratively by the batch file, to study the effect of a parameter of interest.

## 5.2 Results and discussion

Benchmarking was performed to study the effect of the number of neurons per layer, the number of convergence steps, the batch size and the dropout probability, and comparing the behaviour of the two frameworks while iterating over these parameters. Table 3 shows the input parameters used for the benchmarking runs, and the parameters which were iterated are shown in red. Only one parameter was studied at a time. The values over which the parameter was iterated are on the $x$ axis of the following graphs, and the other parameters were set as in table 3. The study lead to a huge amount of results, not all are shown here since their relevance is not noticeable.

### Study of the number of neurons

As a first study, the number of neurons per layer was changed. A network with 3 layers, and in each layer N neurons was built and evaluated. This was the straightest guess in order to influence strongly on the computation time. Indeed, as shown in figures 6 and 7, the time goes up exponentially for the TMVA DNN method. On the other hand, Keras with Tesorflow backend seems to be performing better with a larger number of neurons than with a smaller. This result is very surprising, and not yet fully understood. It could be that Tensorflow does a better parallelization with a larger number of neurons. It must also be added that the Theano interface was not running in multithread.

| Input parameters | |
|---|---|
| nNeurons | 100 |
| nLayers | 3 |
| Activation | 2 |
| Lastactivation | 3 |
| Initializer | 1 |
| Lossfunction | 0 |
| Transformations | ”N,D” |
| Factorystring | !V:!Silent:Color:DrawProgressBar:Transformations=I:AnalysisType=Classification |
| Learningrate | 0.1 |
| Momentum | 0.0 |
| Batchsize | 128 |
| Convergencesteps | 100 |
| Dropout | 0.0 |
| Ntrainsignal | 50000 |
| Ntrainbackground | 50000 |
| Ntestsignal | 100000 |
| Ntestbackground | 100000 |

Table 3: values of the input parameters.

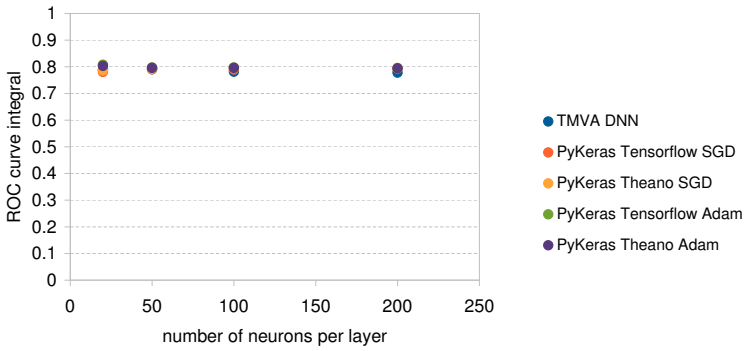



Figure 5: Comparison of the ROC curve integral values as function of the number of neurons per layer.

## Statistical validity of the results

Because of time reasons, not all the benchmarkings could be run several times. In order to check the statistical validity of the results, a series of five runs in TMVA DNN with identical parameters is plotted in the graphs 8, 9 and 10.

The graph for the ROC curve integral (graph 8) shows small variations, the error bars are not even visible. The Standard deviation (error bars) becomes more relevant when it comes to execution time (graphs 9 and 10). This not surprising, since the program has not full control over the processes running at the same time on the machine. This can however not be held responsible for the big difference between TMVA DNN execution time and PyKeras shown in graph 6.

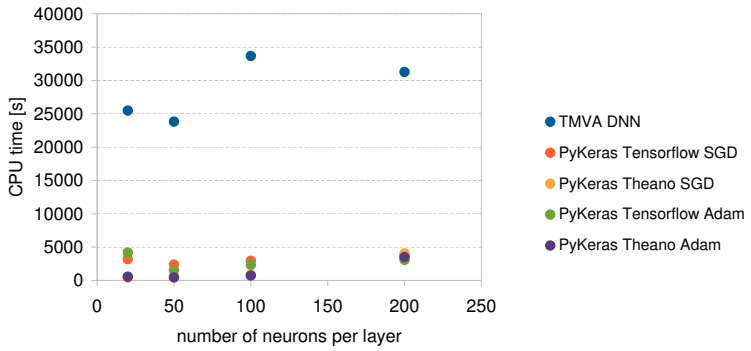



Figure 6: Comparison between the different frameworks of the CPU time for training as function of the number of neurons per layer .

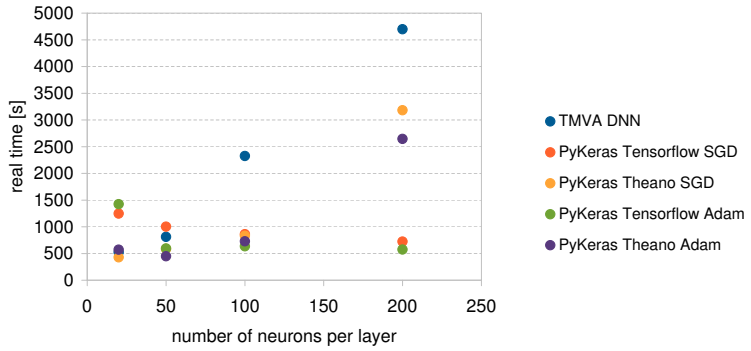



Figure 7: Comparison between the different frameworks of the real time for training as function of the number of neurons per layer.

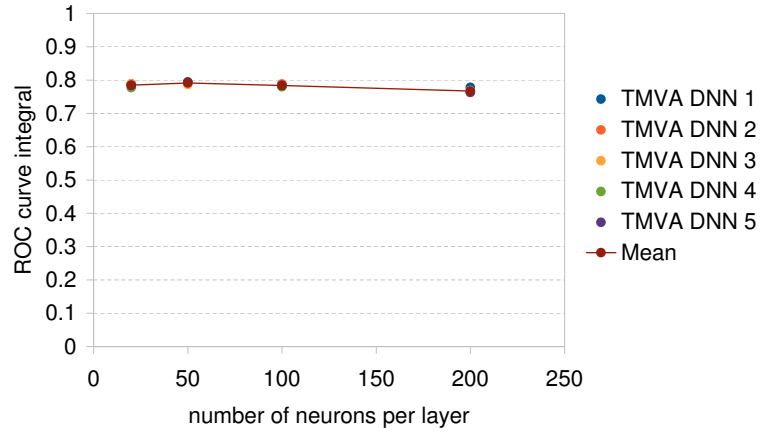



Figure 8: Statistical fluctuations of the ROC curve integral over 5 runs with TMVA DNN.

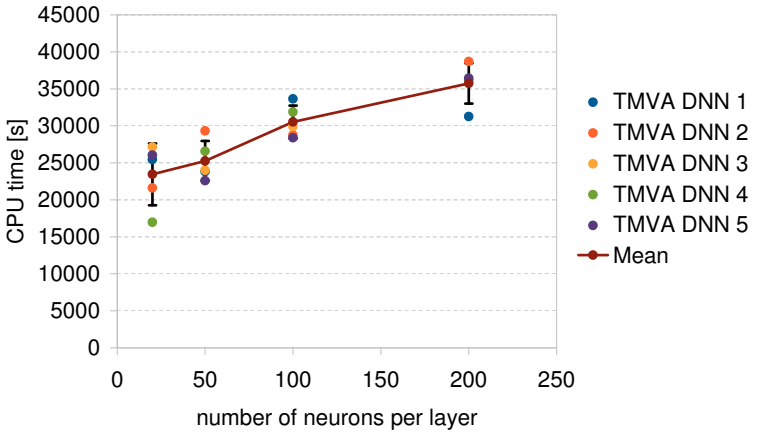



Figure 9: Statistical fluctuations of the CPU time over 5 runs with TMVA DNN.

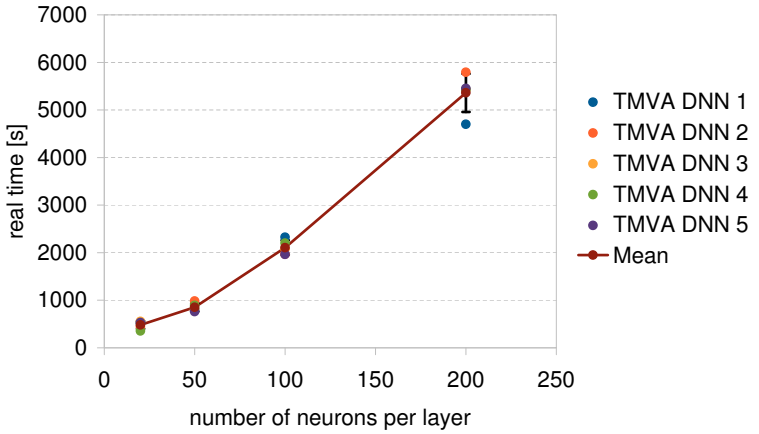



Figure 10: Statistical fluctuations of the real time over 5 runs with TMVA DNN.

# 6 Implementing Autoencoder classes for TMVA

## 6.1 Concepts

The integration of the autoencoder into TMVA turned out to be more involved, due to the fact that autoencoders are in the category of unsupervised learning. On the contrary, TMVA performs classification and regression (which are the common HEP applications), that are both supervised learning. This transpires through the whole data flow in TMVA: targets for regression or classes must be provided while loading the data. Thus, without substantial changes, it was not possible to load the data for unsupervised learning. Moreover, TMVA has data evaluation features that can compare different classification or regression methods. Adding unsupervised learning would have implied thinking of a new performance evaluation. For autoencoders, one could imagine the difference between the reconstructed output and the input. For other future unsupervised algorithms, the question remains.

## 6.2 Use cases of autoencoders in HEP

To avoid too heavy changes in the data loading process, the other option was to think of applications for autoencoders that suit better into the TMVA framework and that meet HEP applications. A review of the ideas that came up is made below, and eventually the first method was implemented.

### 6.2.1 Variable transformation

The autoencoder could be used as a variable pretransformation, namely as dimensionality reduction. The encoded (or compressed) representation of the data stored in the hidden layer would be passed to the further methods. It can be seen as non-linear PCA (Principal Component Analysis), since an autoencoder has the ability to figure out complex and non-linear patterns among the variable.

### 6.2.2 Weight initialisation of DNN layers

Also called Greedy layerwise approach, this procedure initialises consecutively the layers of a deep neural net by making them the hidden layer of an autoencoder. The initialised layer becomes the input layer for the next layer which in turn is the hidden layer of an autoencoder. This ensures a better initialisation than random, and leads to faster and better convergence in the fine tuning steps where the whole network is trained at once.

### 6.2.3 Anomaly detection

Anomaly detection is a use case that would be very useful in HEP data analysis, since it could detect incorrect data. It could be used to point out a malfunctioning detector, or an event of a new type. Concretely, an autoencodre would be trained on normal data. Then, the data to be analysed is passed through the autoencoder and the decoded output is compared to the input. A significant mismatch would suggest that the given event does not resemble the data the autoencoder was trained on, and this could be an anomaly.

### 6.2.4 Classification

The last use case would be to build a neural net by adding a classification layer on top of an autoencoder. The "autoencoder part" would make sure some correlations or patterns within the data are extracted and the classification is made upon this compressed dataset. This is very similar to using the *variable transformation* before feeding the data to a neural net, except that here the whole net could still have the ability to be fine tuned at once.[1] It should also show better performance than using two methods.

[1] It could be seen as a classification Greedy layerwise initialised net with decreasing number of neurons in the layers.

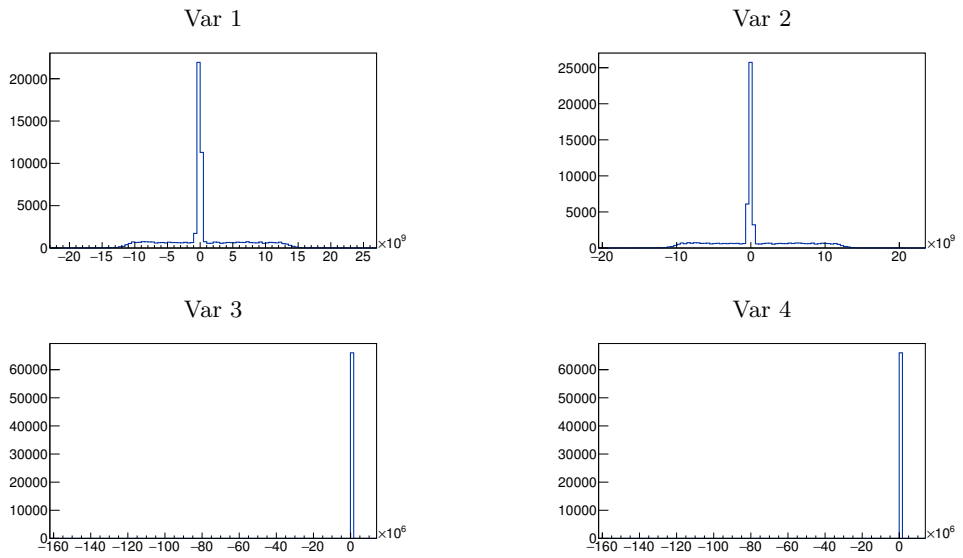



Figure 11: Measure of the training accuracy of the autoencoder. The histograms show the distribution of the difference $\Delta = x_i - x'_i$ between the decoded outputs and the inputs over the whole dataset.

## 6.3 Results and discussion

### 6.3.1 Variable transformation

The results obtained with the variable transformation are very recent, and not yet fully understood. The *tmva_class_example.root* dataset was used and the variable transformation was performed in front of a BDT (Boosted Decision Tree) classifier.

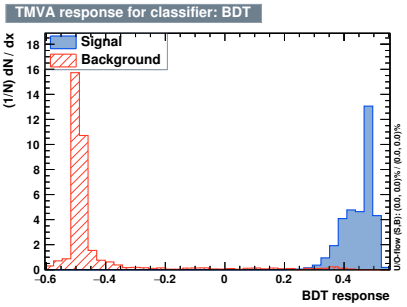
### 6.3.2 Training accuracy

The graphs in figure 11 provide a measure for the autoencoder training accuracy. The autoencoder was trained with the whole set and evaluated with the same set. A total number of 15′000 epochs of training were performed, with a learning rate of 0.1 a corruption level of 0.2 and no dropout. The layout comprised two hidden layers, of size 4 and 2 respectively, and with *sigmoid* activation function. The difference between the inputs inputs $\{x_i\}$ and the decoded outputs $\{x'_i\}$ was computed:

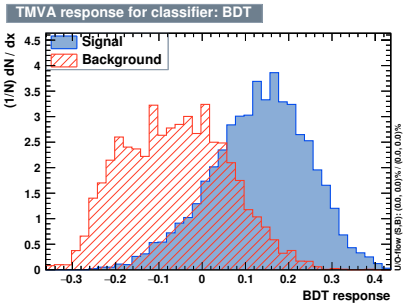
$$\Delta = x_i - x'_i \tag{2}$$

The histograms in figure 11 show the distribution of this error for the events from the whole dataset.

Overall, the histograms of figure 11 show a very good training accuracy, even though the evaluation was done on the same dataset as the testing. This is due to the way pretransformations are trained in TMVA : since it is unsupervised learning and should




(a) BDT classifier response with DAE pretransformation




(b) BDT classifier response without pretransformation

Figure 12: Classifier response (cheated) of the BDT method on *tmva_class_example.root* dataset.

pretransform the data for another method, it is not split into testing and training sets, in order to not waste data for the next method.

### 6.3.3 Classifiction response

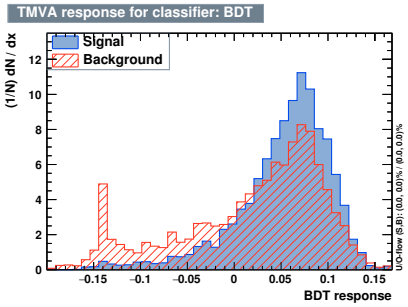
The graphs 13 show the response of the BDT classifier with and without DAE pretransformation. The first graph in figure 12 is "cheated" for the following reason. During the pretransformation, two separate autoencoder were trained for signal respectively background. Then, during the classification process, each event was passed to the according autoencoder knowing its class, before being fed to the BDT. In short, the DAE knows the class during pretransformation, the BDT does not but gets events that were transformed according to their class. This does not reflect what would happen in reality, since the classes are not known (hence the classification).

A more realistic scenario was performed for the graphs in figure 13, where an autoencoder was trained on all events, including signal and background. This seemingly has the effect to entangle signal and background and to give worse classifier response than a bare BDT without pretransformation. The dataset might not be a good example for autoencoder pretransformation, or the number of hidden neurons could be too low.

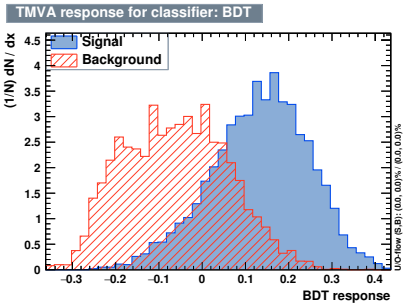
Further testing should be done on a bigger dataset, and with a lower compression factor than 2. The parameters of the autoencoder can probably also be better tuned.

## 7 Further possibilities

This project was a first step towards integrating autoencoder methods in TMVA, and leaves behind a couple of future development opportunities. First of all, in the variable transformation method, it would be useful to have the ability to pass parameters. This feature could be extended to the other transformations as well. There are also the other




(a) BDT classifier response with DAE pretransformation




(b) BDT classifier response without pretransformation

Figure 13: More realistic classifier response of the BDT method on *tmva_class_example.root* dataset.

use cases for AE discussed, that might be implemented. Finally, if proven useful, TMVA could be adapted to support unsupervised learning.

# 8 Conclusion

In this project, a benchmarking study was performed between the DNN implementation of TMVA and Keras with Tensorflow and Theano backend. The results show that the TMVA DNN implementation is substantially slower. An implementation of the Adam optimiser could be envisaged. even thoug it dit not make a huge difference, should it be for Tensorflow or Theano. The ability to set an upper limit to the number of training epochs would also beneficial. Indeed, with Keras after a certain number of epochs the results are already close to the ones obtained after fulfilling the convergence criterion. The second part of the project did undergo a slightly more troublesome process. Indeed, since autoencoders are unsupervised learning, the integration into TMVA was not straightforward. A couple of applications for autoencoders that make sense in HEP were identified, and the variable transformation was implemented. Thus, TMVA offers now the ability to compress the data by an autoencoder before being passed to a further method. There are still a bunch of further possibilities left regarding autoencoders with the new deep learning module. Improvements to the variable transformation method are also possible, especially the ability to set parameters such as the number of training epochs, the learning rate, and the corruption level.

This project allowed me to take an insight in the root development, and especially in TMVA, the machine learning library. I learned a lot about ROOT, data analysis, and deep learning which was e new field of machine learning for me. I also improved my software engineering skills, using git, cmake, and working on big and complex projects. This experience will be very valuable for my future as a physicist, and widened my knowledge in computing. I also enjoyed the social experience, meeting people from all over the world, and CERN's friendly atmosphere.

## 9 Acknowledgements

The realisation of this project would not have been possible without the help of some people.

In this perspective, I would like to thank Dr Lorenzo Moneta and Dr Sergei Gleyzer, who gave me the opportunity to work at CERN and supported me during the project.

I would also like to thank Akshay Vashistha, who did a tremendous work for the AE in the new deep learning module, and with whom it was friendly and enjoyable to work.

Last but not least, I would like to thank Kim Albertsson, who was always available for questions or discussions whenever I was stuck on something.

And I would also like to thank the SFT group for having been so nice colleagues during this Summer Student school.

## References

[1] "TMVA Users Guide", A.Hoecker & Al., CERN, March 2017

[2] "Integrating Deep AutoEncoders in TMVA", https://indico.cern.ch/event/662490/, Akshay Vashistha, August 2017

[3] "Using neural nets to recognize handwritten digits", Michael Nielsen, May 2017

[4] https://people.cs.pitt.edu/~xianeizhang/notes/NN/NN.html consulted in August 2017

[5] https://prateekvjoshi.com/2014/10/18/autoencoders-in-machine-learning/ consulted in August 2017

[6] http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders consulted in August 2017

[7] https://pgaleone.eu/neural-networks/2016/11/24/convolutional-autoencoders/ consulted in July 2017

[8] http://blackecho.github.io/blog/machine-learning/2016/02/29/denoising-autoencoder-tensorflow.html consulted in July 2017

[9] https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/ consulted in June 2017

[10] https://blog.keras.io/building-autoencoders-in-keras.html consulted in July 2017

[11] "A Tutorial on Deep Learning, Part 2: Autoencoders, Convolutional Neural Networks and Recurrent Neural Networks", Quoc V. Le, October 20, 2015