The Drone NN: a method to fit Parton Distribution Functions

Luigi del Debbio, Michael Wilson, David Sola Gil

University of Edinburgh

September 29, 2019

Abstract

Neural Networks have been proven to be a crucial tool for data processing in High Energy Physics (HEP) and could even be the key to new discoveries, as they are able to learn about patterns and features of data that might not have been considered before[1]. In this project, we explore a newborn Neural Network: the Drone Neural Network (DNN), which could contribute to handle the unmanageable amount of data from particle physics experiments, whilst providing us a way to assess the complexity of certain functions: the Parton Distribution Functions. A description of the implementation process, the major difficulties that arose and an evaluation of the final DNN are provided.

1 Introduction

Data collection and real-time analysis of such data is an issue of growing importance in HEP experiments: there is a massive amount of information to process and the current methods will become obsolete in a few years due the improvement of detectors and other measuring devices. Nowadays, software triggers are preferred over Machine Learning (ML) tools, as they are faster and easier to implement in the current frameworks (C++/Python). However, it has been proven in several LHC experiments that ML classifiers enable more powerful data reduction[2].

The main aim of this project is to implement a Drone Neural Network (DNN), a type of Neural Network (NN) devised to learn from a previously trained ML classifier. Provided that the training has been done correctly, the DNN will be behave just as the classifier but will be lighter and thus easier to evaluate and implement, paving the way to the substitution of triggers in HEP experiments.

All the files have been coded in Python, using the Pytorch library to design and update the NN. These and their resulting plots can be found in the dronepdf GitHub repository.

2 The Drone Neural Network

2.1 Initial structure

The DNN is initialised as a NN with a unique hidden layer of 5 neurons, using a `Sigmoid`[3] activation function and a linear model to relate layers. Since we want to learn as much as we can from the original classifier, the loss function $\mathcal{L}$ is defined as:

$$\mathcal{L} = \sum_i \left(F(\mathbf{x_i}) - G(\mathbf{x_i})\right)^2, \tag{1}$$

where $F(\mathbf{x_i})$ and $G(\mathbf{x_i})$ are the outputs of the original and drone models on the *ith* data point of the batch, respectively. Stochastic gradient descent (SGD) is the optimizer chosen for the implementation, applied after each batch processing. Besides, the program keeps track of the total loss per epoch during training for later study.

2.2 Training of the Drone Neural Network

What characterizes the DNN is the possibility of modifying its hidden layer during the training process when certain conditions are satisfied. To keep the model as simple and light as possible, for the DNN to trigger its extension in the *jth* epoch, letting δ_j be the relative loss improvement, κ the learning threshold, σ the minimun required improvement and $\hat{\mathcal{L}}$ the value of loss when the hidden layer was last extended, we must have that:

$$\delta_j = \frac{|\mathcal{L}_j - \mathcal{L}_{j-1}|}{\mathcal{L}_j} < \kappa, \text{ and} \tag{2}$$

$$\mathcal{L}_j < \hat{\mathcal{L}} - \sigma_j, \text{ where} \tag{3}$$

$$\sigma_j = m(1 - e^{-b(\hat{t}+n)})\delta_j \mathcal{L}_j. \tag{4}$$

The precise values of $\kappa, n, m,\ \hat{t}$ (epoch number) and b are not of particular importance, but the conditions described by equations (2) and (3) are vital. The equation (2) will be satisfied if the relative loss is below a certain threshold: we are not learning as much as we expected. For the equation (3) to hold, we must have that the loss has improved since we added the last neuron. As equation (4) shows, σ_j increases with epoch, with the aim of scaling the expected decrease in δ_j and minimising the chances of getting stuck in an isolated local minima. To achieve such increase, σ_j is initialised from a minimum value determined by n and scales up until reaching a maximum at m, with the steepness of this increase (b) chosen so that on average the transition from minimum to maximum takes 50 epochs. Finally, bear in mind that associated neurons are introduced with their weight set to zero to preserve a continuous loss function.

Now that we understand how the DNN works and its purpose, we can move on to how our Drone evolved as we tried to fit increasingly complicated functions.

3 Method - constructing our DNN

3.1 Toy model

As our starting point, we used the single layer NN provided in the Pytorch webpage for the skeleton for the Drone, connecting the layers with a linear model. Our aim was to fit very simple functions whose behaviour was well-known to us: the quadratic polynomial $f : \mathbb{R} \to \mathbb{R}$ given by $f(x) = x^2$, and the Chebyshev polynomials[4].

3.1.1 Modification of the hidden layer

At the initial stage, the main challenge was to implement the mutation of the hidden layer: how to add neurons whilst the training was going on.

The solution we came up with consists in using a mask tensor for the output layer. Once we have chosen a maximum amount of neurons (`max_N_neurons`), the first weight tensor (`self.w1`) is initialised with the first 5 weights taken from a normal distribution of 0 mean and 0.01 standard deviation. The rest of the weights (`max_N_neurons` $- 5$) are initialised as 0. Then, in every forward pass in the NN, we multiply the weights tensor by our mask (`self.mask`), a $1 \times$ `max_N_neurons` tensor containing $1's$ in the weights corresponding to the neurons that have been "switched on" and $0's$ everywhere else. After this multiplication, Pytorch understands that those weights multiplied by 0 do not need an update, and only considers the weights multiplied by 1 in the back-propagation process, just as we intend.

3.1.2 Activation function - dead neurons

Despite the fact that the original DNN takes `Sigmoid` as the activation function, we initially opted for `ReLU`[3] as it is the state-of-the-art in deep learning and was giving more accurate fits.

Nonetheless, while we were running some tests to check that the morphing of the hidden layer was working correctly, we noticed that some of the additional neurons were not being updated. These were neurons that had been successfully added to the Drone yet whose value had not been changed from 0, what we call "dead neurons". This issue was being caused by the activation function we had chosen: `ReLU`. As all the additional neurons were initialised with a value of 0, any tiny change in the negative direction would set the corresponding weight in the constant part of the image, hence giving a 0 gradient in that point and cancelling out the possible update of the weight in every step. Changing the activation function from `ReLU` to `Leaky ReLU`[3], which substitutes the constant part of `ReLU` with a straight line of smaller slope, corrected the problem whilst producing a decrease in the amount of added neurons during training, as every neuron was contributing to the fit then.

3.1.3 Validation of the toy model

Once the toy model was constructed, including the conditions to morph the hidden layer as described in Sect.2.2, we tried to fit both the quadratic and the Chebyshev polynomial of degree 6. For both estimations, we picked 200 evenly spaced x values in the interval $[-1, 1]$ and gave these and their respective $f(x)$ values as input to the toy model. The values of the parameters that condition the expansion of the hidden layer were set to match those of the DNN paper, i.e. $\kappa = 0.02$, $b = 0.04$, $n = 2.05 * 10^{-3}$ and $m = 50$.

As we can observe in the plots from Fig.1, the toy model was definitely decreasing its loss as the epochs increased. The epoch-by-epoch addition of neurons was contributing to the training process and the DNN was beginning to learn the overall features of even more complex functions, which ensured to us that we were walking in the right direction.

3.2 Parton Distribution Functions - DNN

The groundwork had been laid and the our toy model was working reasonably well. Taking it as the starting point for our final DNN, the next step was to test whether it could fulfill its main purpose: fitting a function created from another NN, in our case the Parton Distribution Functions (PDFs) for 8 different data sets (8 different flavours in the fitting basis). The validphys API and its main functionalities[5] were used in the implementation of the `getdata.py` file to obtain the files containing the y-values of the PDFs. Precisely, we chose the `NNPDF31_nlo_as_0118` set of PDFs from LHAPDF6[6].

3.2.1 Changes on the Optimizer

Initially, the fits were quite similar to the fit of the Chebyshev polynomial (Fig.1c): the DNN was only learning the overall behaviour but was not grasping the shape of PDFs properly, as we expected for functions with greater curvature changes.

After several unsuccessful approaches to improve the learning process: reducing the starting learning rate, tweaking the parameters that control the modification of the hidden layer and even testing different activation functions; we made an unexpected breakthrough. Changing the optimizer from the standard SGD to ADAM[7] led to a massive reduction in the loss as well as the number of neurons required (Fig.2). Moreover, as we were also trying different activation functions, we discovered that `Softmax`[3] gave us the best fit on average, possibly due to its curvilinear nature since it is based on exponential functions. This latter improvement came at a cost: the amount of additional neurons was greater than in the `Leaky ReLU` case, so perhaps the `Leaky ReLU` can arrive at the same degree of accuracy if we train it for long enough. Thus, we decided to stick to it. Both facts can be observed by comparing Fig.2a to Fig.2b and Fig.2c to Fig.2d.

3.2.2 Imposing a stopping criteria - The L infinity norm

As a means to assess the amount of accuracy we wanted in our fits, we decided to follow the L-infinity norm with a factor of 0.01, i.e. the maximum relative difference (m.r.d.) between the real PDF and our estimation must not exceed 0.01, so our fits were wrong by at most 1% in every point. Nonetheless, this norm turned out to be too ambitious for our training of the DNN. After hundreds of epochs, it could only fit the PDFs up to a factor of ~ 5 in the best cases, way above our limit (Fig.3).

For our initial guess, we considered the modification of our learning rate scheduler, as ADAM does the decay of the learning rate automatically[7]. Switching off the `ReduceLRoPlateau` scheduler improved the results, now with the m.r.d. stagnating around 10. However, the paper that gave birth to ADAM recommends the use of an exponentially decaying learning rate[7], so after some trials with different parameters, we re-implemented the `ReduceLRonPlateau` module, this time with an immense `patience` (~ 7000) epochs. Then, the learning rate was not reduced by the scheduler unless it was strictly necessary, leading to a general decrease in the m.r.d. which reached values close to 2 in the best case (Fig.4).

3.2.3 Pre-processing of the input data

The critical improvement came with the pre-processing of our data. After having taken a closer look at which point the m.r.d. occurred, we realised that it tended to be near the tails, where the value of the PDF is very close to 0, hence making any tiny difference immensely huge. We knew that the PDFs are generated multiplying the output of a NN by two exponential factors:

$$PDF(x) = x^{\alpha} * (1-x)^{\beta} * p(x), \tag{5}$$

where $p(x)$ is the output of the NN, x^{α} describes the behaviour of the PDF as $x \to 0$ and $(1-x)^{\beta}$ describes the behaviour as $x \to 1$. Hence, we took the approach of pre-processing the dataset we were trying to fit so that the Drone NN would fit $p(x)$ instead, i.e.

$$DroneNN(x) \sim \frac{PDF(x)}{x^{\alpha} * (1-x)^{\beta}} = p(x), \tag{6}$$

where α and β are 2 parameters estimated from the actual PDFs.

3.2.4 Estimation of the exponents - linear method

To estimate these exponents, we opted for a linear regression method, linearizing both terms as:

$$\alpha \log(PDF(x)) = \log(x) \text{ as } x \to 0, \text{ and} \tag{7}$$

$$\beta \log(PDF(x)) = \log(1-x) \text{ as } x \to 1. \tag{8}$$

Since we did not have a precise definition on how close to 0 or 1 the value of x had to be for a valid estimation and we had a list of discrete values, we began by choosing some sensible bounds: 10^{-3} as the upper bound for α and $4*10^{-1}$ as the lower bound for β. Then, we would try a linear fit for α (or β, respectively) with all the values of x going from the smallest value to such upper bound (or with all the values ranging from the lower bound to the maximum value of x) and compute its corresponding coefficient of determination (R^2). After that fit, the point that was closest to the bound was removed in both cases and another fit was produced, again with its associated R^2. This process was repeated until we ended up with 2 points in either side of the tail. Finally, we would compare the values of R^2 for each fit and select the parameters that gave a result that was the closest to 1.

The pre-processing using these exponents, together with taking the natural logarithm of the x-values before introducing these in the neural network, led to a great improvement. At the expense of enormous amounts of additional neurons, the fits of almost every PDF were almost perfect, and the m.r.d. was reduced to values below 1 (Fig.5), being the ”c” flavour the only exception (Fig.6b). We suspect that this has to do with the greater fluctuation in the values of the PDF (going from 0 to 14), as the fit itself is quite accurate (Fig.5b).

Despite the better results, bearing in mind that the neural network that creates these PDFs only depends on about 37 parameters, we certainly were adding more neurons than required. Moreover, the addition of neuron was not being efficient at all in some cases. For instance, in Fig.6a, the DNN was just becoming heavier and less efficient but could not overcome the lower bound of ~ 3.335.

3.3 Scanning of the m values

To cope with the unnecessarily massive amount of additional neurons, we decided to impose more restrictive parameters and explore how this change affects the training process.

One of the major problems of reducing the learning rate with any kind of the scheduler is that, since we were training for 100000 epochs, at some point the learning rate would become so small that the equation for κ will always be satisfied(2). Hence, we needed to greatly reduce κ, to force the DNN to reach a plateau before adding a neuron. After several blind guesses, $\kappa = 0.0005$ with `patience` $= 200$ seemed to be yielding the best results overall. Then we focused on finding the best value for m which seemed to us the most important after κ.

As we had no intuition on how to vary m, we made a list of predesignated values of m and trained the DNN for 2000 epochs with every value. To our surprise, the value of m does make a key difference not only in the amount of additional neurons, but also in how fast the loss converges to an asymptote and what the minimum value of m.r.d. is.

Looking carefully at the plots for c data (Fig.7a and Fig.7b), we can observe that the value of m vastly conditions how fast the loss converges to a value of ~ 1, despite the fact that, leaving the $m = 0.02$ case aside, all the values yield the same final m.r.d. value. On the other hand, on the plots for t8 data, the value of m barely conditions how fast the loss converges to a minimum value, but there is a huge discrepancy on the final value of the m.r.d. depending on the m. This told us that the parameters had to be scanned and optimized for every data set, as they are highly dependent on the PDF we are trying to fit.

3.4 Validation of the DNN

Our final version of the DNN: `drone_alphabetalin.py` is a single layer NN that begins with 5 neurons and is able to expand its hidden layer up to 2000 neurons during a training of 10000 epochs. Before starting the training, the 200 pairs of points $(x, PDF(x))$ corresponding to each data set are pre-processed as described in Subsect.3.2.2 and Subsect.3.2.4. Then training is carried out in batches with `batch_size` $= 20$, using a linear model to connect the layers and the activation function `Leaky ReLU` from the input to the hidden layer. The chosen optimizer is `ADAM` and we combined it with the `ReducedLRonPlateau` scheduler, with `patience` set to 200.

With regards to the parameters that control the expansion of the hidden layer, we set $\kappa = 0.005$, $b = 0.04$ and $n = 2.05 * 10^{-3}$. Finally, after some trials, we manually picked the optimal value for m for each data set, trying to minimize the size of the hidden layer while forcing the loss and m.r.d. to converge to its minimal value as quickly as possible.

As we can observe in Fig.8, the loss function, independently of the data set chosen, is converging to a value close to 0, without adding an alarming amount of neurons to the fit, just as we intended. Besides, the fits (Fig.9a and Fig.10a) seem accurate and match the overall behaviour of the PDFs, diverging from these in the regions with major fluctuations but still providing a sensible approximation.

4 Final results and discussion

Below there is a table with the values of m chosen for each data set and the final results obtained(Table.1).

data file	m	m.r.d.	final loss	alpha	beta	additional neurons
c_data.csv	5000	1198.770509	2.091183	0.269369	4.722114	22
g_data.csv	3000	0.211193	0.446010	-0.091950	-3.541796	16
sigma_data.csv	3000	0.035999	0.0433015	-0.130582	2.318618	17
t3_data.csv	3000	0.567048	1.306733	0.204448	2.696448	15
t8_data.csv	3000	0.046759	0.0494184	0.039584	2.428226	20
v_data.csv	5000	0.145842	0.087336	0.372357	2.129262	14
v3_data.csv	1500	0.121791	0.083395	0.528474	2.274550	25
v8_data.csv	1500	0.479662	0.086299	0.348145	2.664449	25

Table 1: Chosen m and final results for each data set.

Even though none of the m.r.d. is below our goal and the results for t3 and c data sets seem to be even worse (compare Fig.6 to Fig.9b and Fig.10b), do bear in mind that we have significantly reduced the amount of epochs we trained the DNN for and the amount of additional neurons, thus being these results a more precise representation of what the DNN can and cannot achieve. In fact, the results for c data are not concerning, as we know that the PDF for such quark is quite flawed.

In regards to the rest of the data sets, the addition of neurons seems to be enhancing the learning process quite efficiently, adding a neuron only when it will have a positive impact and leading to unexpectedly accurate fits. Overall, the size of the DNN was significantly reduced, and even for the cases where the m.r.d. is well above 10%, the DNN has managed to grasp the monotony and some changes of curvature of the function. Perhaps training for longer with minor tweaks in the other parameters could give us the fit we are looking, as in some cases the m.r.d. is still decreasing when we stop the training (Fig.11d and Fig.11e).

Apart from just fitting functions, the DNN could shed some light in other fields of mathematics and artificial intelligence. The additional amount of neurons required for each data set could be used, to some extent, to assess the complexity of different functions and create some sort of ranking criteria. Moreover, the NN we are trying to learn from uses a multi-layered structure with 37 parameters (weights + biases). Is this telling us that several layers, despite having less neurons overall, can behave in a more complex way?

5 Further studies and potential pitfalls

Even just in the function fitting affair, the DNN does require further investigation. Just tweaking the value of m led to a great improvement on the quality of the fits and how faithful these are, but there is no intuitive reason on why the value of m would have such an impact, nor do we know how to obtain the best value for each data set, as there is no obvious pattern. Hence, several approaches could be tested in future studies:

- Changing the activation function from `Leaky ReLU` to `Softmax` or `Arctan`. The DNN struggles the most to learn the sudden changes of curvature, but surprisingly enough, spikes (lines that meet at a point but with different slopes) are fitted almost perfectly. This could do with the activation function we are using: `Leaky ReLU`, that is constructed with 2 lines. We should definitely look into curvilinear activation functions, due to the nature of the PDFs.

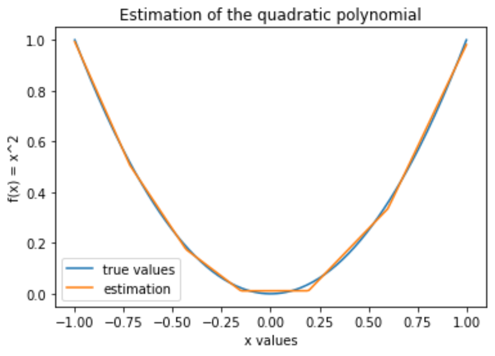
- Training the DNN for longer, testing loads of different groups of parameters, as we have just seen the tip of the iceberg. There are optimal values of m for the κ we have blindly chosen, but there may be even better settings of parameters that have not been discovered yet.
- Enlarging the data files with more points from the PDF and reduce the amount of epochs we train for. In the original DNN paper, training is carried out for less epochs (300), so testing whether this change makes the precise values of the parameters unimportant could shed light on how these work.
- Changing the m.r.d. criteria to another metric of accuracy. Some of the PDFs tend to have values very close to 0 (c data for example), resulting in every difference, despite how small it is, being shoot up to immense values when we consider the relative difference. Moreover, when the PDFs are generated, the error is not the same on every point (see GitHub plots) and the actual PDF values fluctuate a lot in certain regions, so we could try to give some weighting to the errors depending on the region where these are located.

6 Conclusion

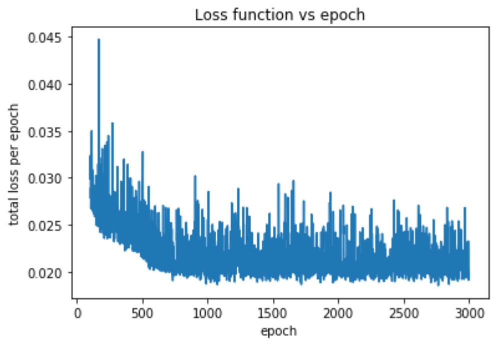
In the light of above, we have provided some examples where the DNN can actually learn from the PDFs and reproduce their shape to almost the degree of accuracy that we imposed, while keeping a less intricate and easier to implement structure. Hence, they could be the longed-for substitutes of the widespread classifiers for HEP experiments in a near future, provided that these match the accuracy requirements. In fact, in the previous exploration of the DNN no information on how accurate the resulting fits are is given, so we might have imposed an even stricter criteria.

Even though there is scope for improvement and many questions are left unanswered, we are taking steps towards the solution of the data management issue that motivated both the DNN paper and this project.

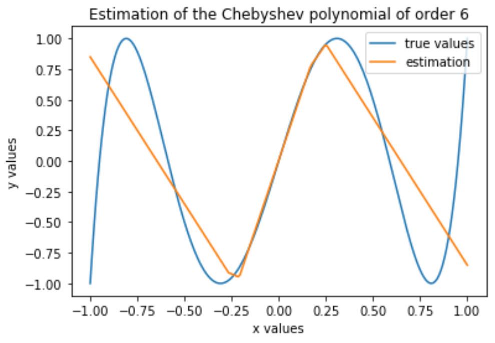
7 Plots



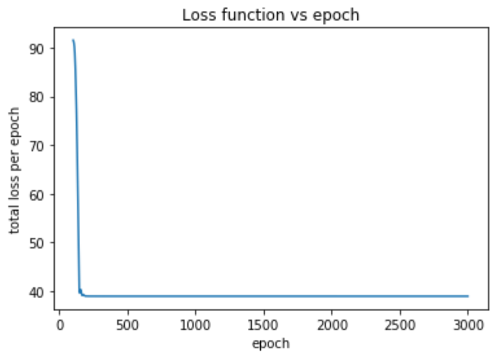
(a)



(b)

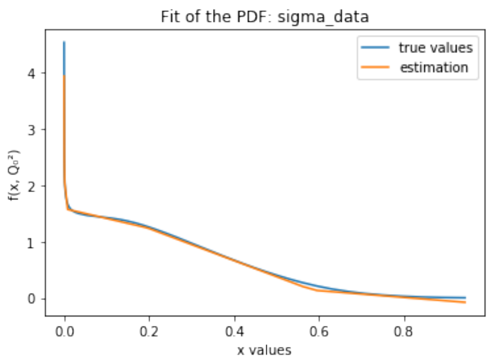


(c)

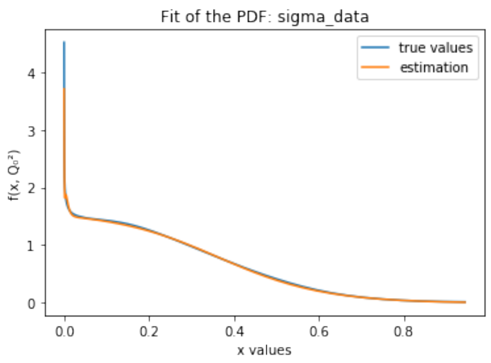


(d)

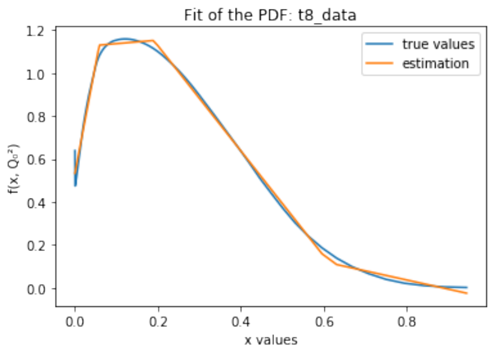
Figure 1: Validation plots of the toy model. Fig.1a is the fit of a quadratic polynomial which resulted in 71 neurons being added and a loss per epoch as shown in Fig.1b. Fig.1c is the fit of a Chebyshev polynomial of order 6 which resulted in 186 neurons being added and a loss per epoch as shown in Fig.1d.



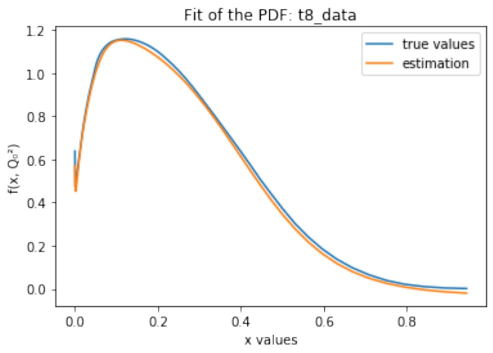
(a)



(b)



(c)



(d)

Figure 2: Fits of some PDFs with ADAM and different activation functions. Fig.2a is a fit of Sigma data set using `Leaky ReLU`, that resulted in 49 additional neurons; whilst Fig.2b the same fit using `Softmax` instead, resulting in 92 additional neurons. Fig.2c is a fit of t8 data set using `Leaky ReLU`, that resulted in 20 additional neurons; whilst Fig.2d the same fit using `Softmax` instead, resulting in 129 additional neurons.

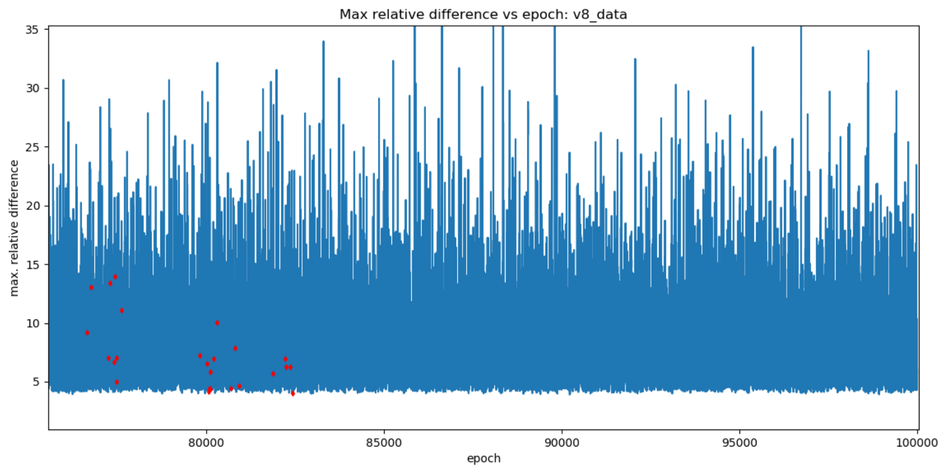


Figure 3: Tail of the m.r.d. plot for the v8 flavour fit using ADAM and `ReducedLRonPlateau` with `patience` = 400. It is clearly bounded below by a value ~ 2.35 and the addition of neurons (red diamonds on the graph) does not seem to enhance our Drone at all.

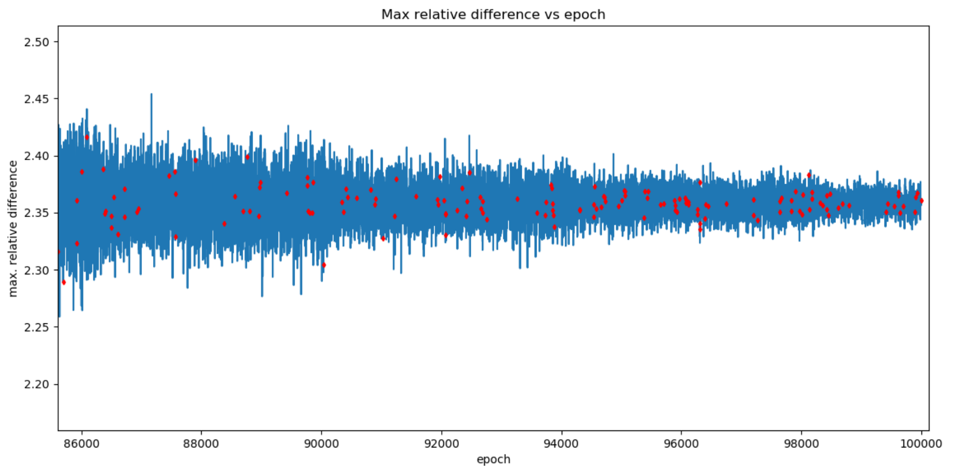
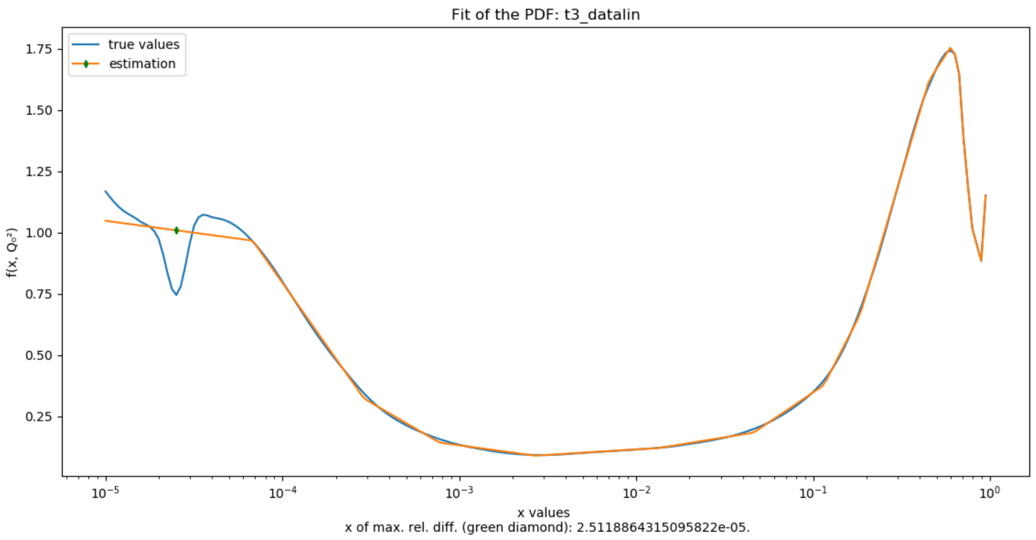
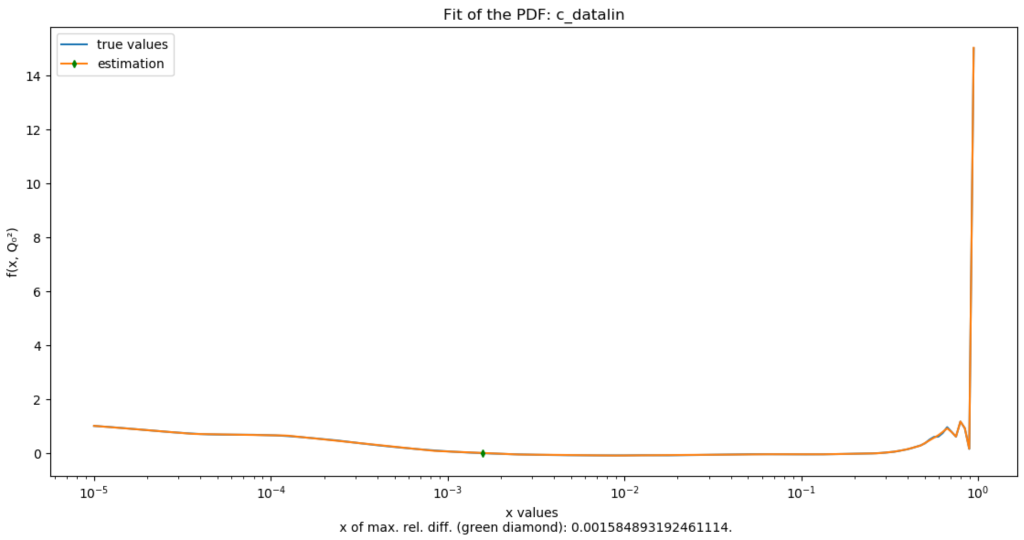


Figure 4: Tail of the m.r.d. plot for the v8 flavour fit using ADAM and `ReducedLRonPlateau` with `patience` = 7000. As we can observe, the addition of neurons is only reducing the fluctuations of the m.r.d., whose value tends asymptotically to ~ 2.35.

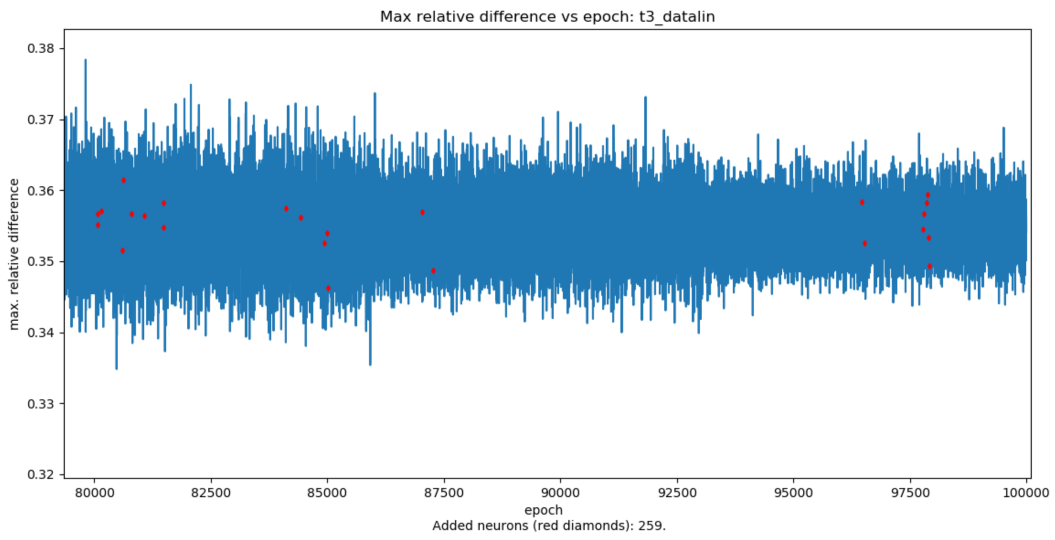


(a)

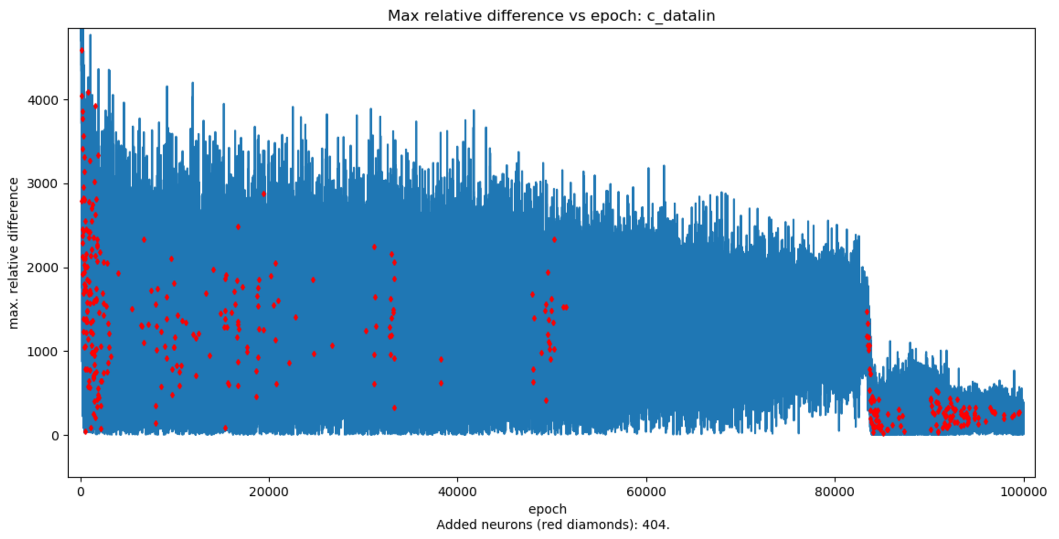


(b)

Figure 5: Fits of the t3 and c data sets after the alpha-beta pre-processing.

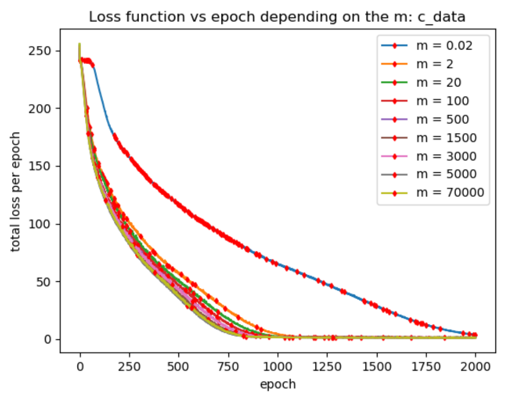


(a)

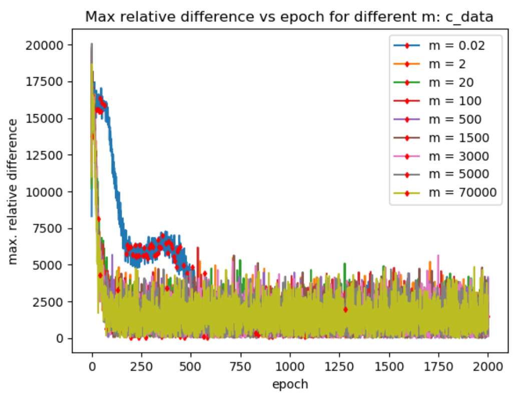


(b)

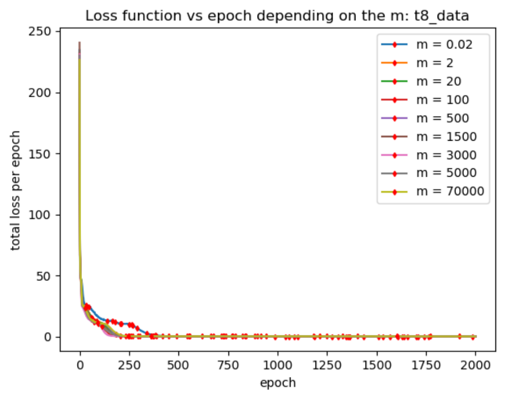
Figure 6: Evolution of the m.r.d. against epoch for t3 and c data sets after the alpha-beta pre-processing.



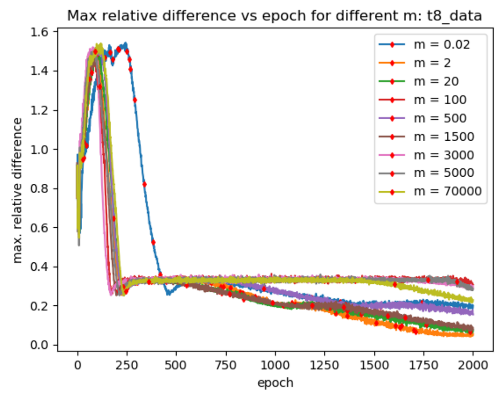
(a)



(b)



(c)



(d)

Figure 7: Plots of the loss and the m.r.d. vs epoch for different data sets and values of m. Fig.7a and Fig. 7b correspond to c data, while Fig.7c and Fig.7d correspond to t3 data. The red diamonds indicate the epochs when a neuron was added.

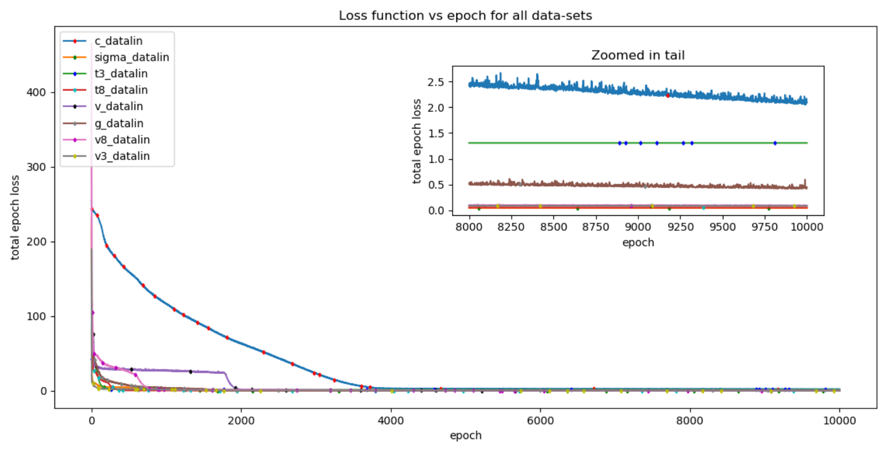
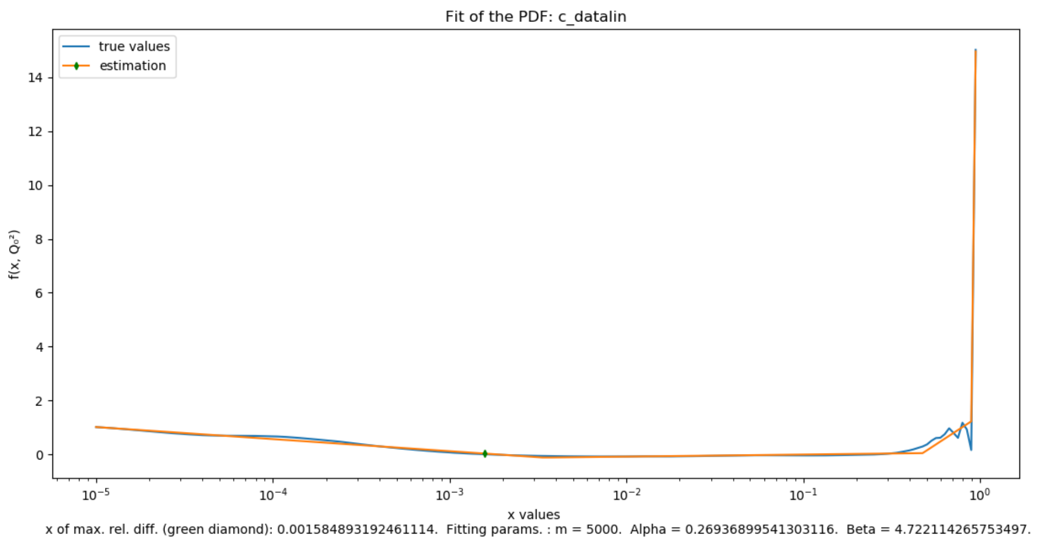
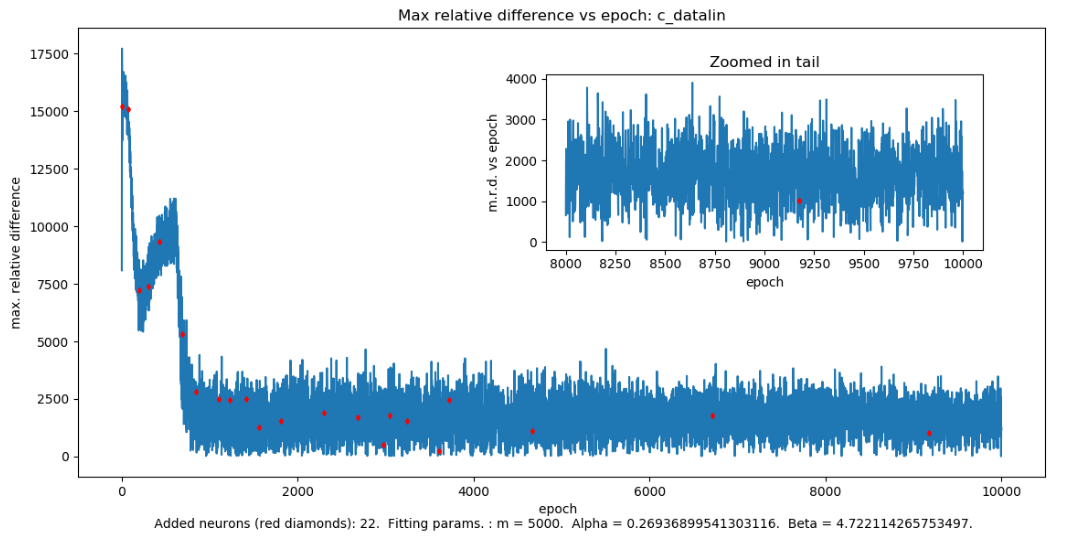


Figure 8: Loss vs epoch during training of the final DNN for every data set, also including a zoom of the tail.

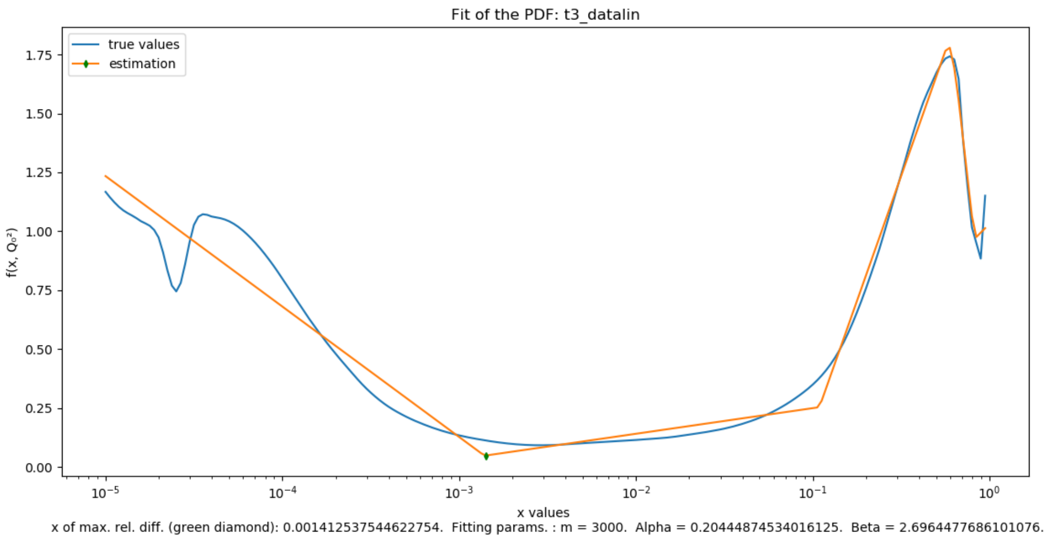


(a)

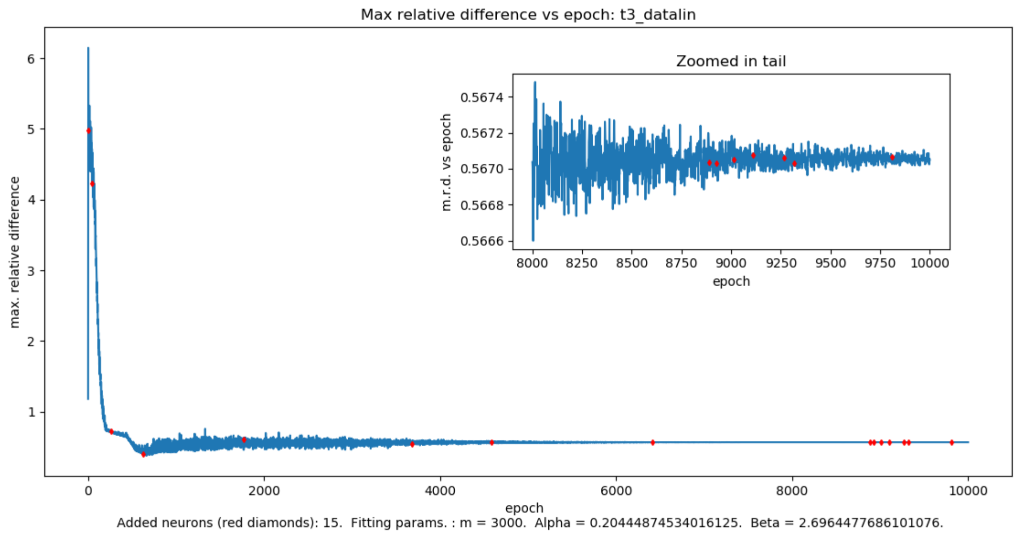


(b)

Figure 9: Plots of the final fit for the c data. Fig.9a depicts the estimation from the DNN vs the actual PDF and Fig.9b its corresponding m.r.d. vs epoch, with the zoomed in tail of the same graph to observe how values fluctuate at the end.

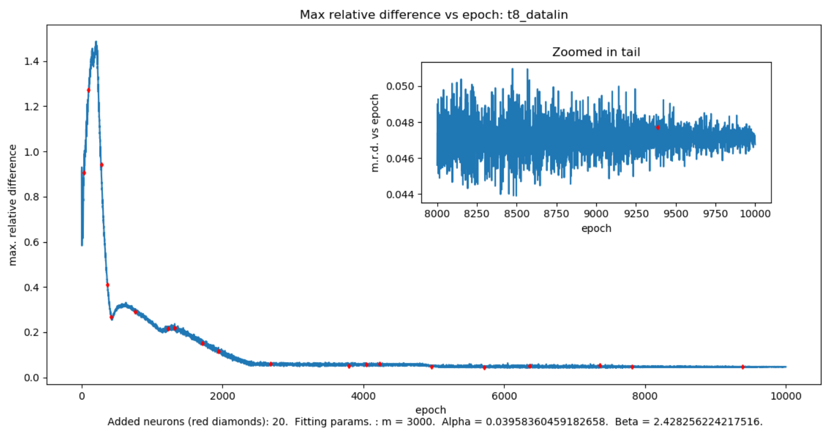


(a)

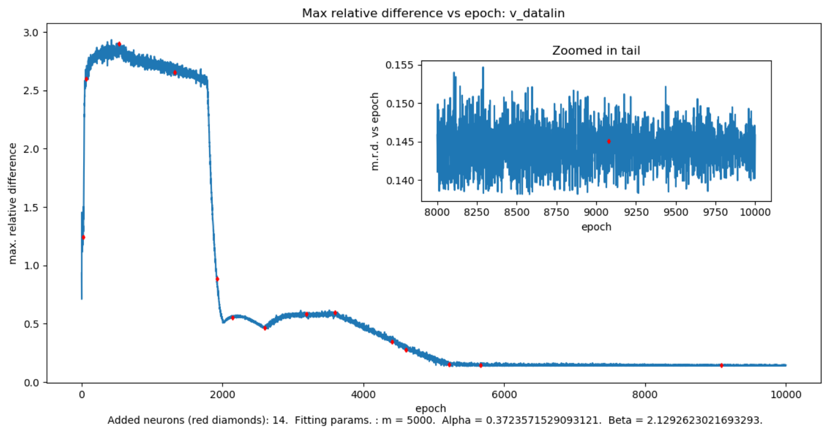


(b)

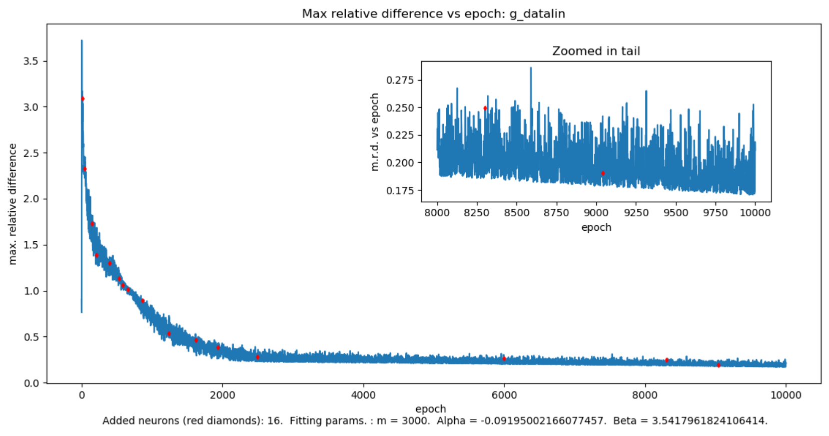
Figure 10: Plots of the final fit for the c data. Fig.10a depicts the estimation from the DNN vs the actual PDF and Fig.10b its corresponding m.r.d. vs epoch, with the zoomed in tail of the same graph to observe how values fluctuate at the end.



(a)

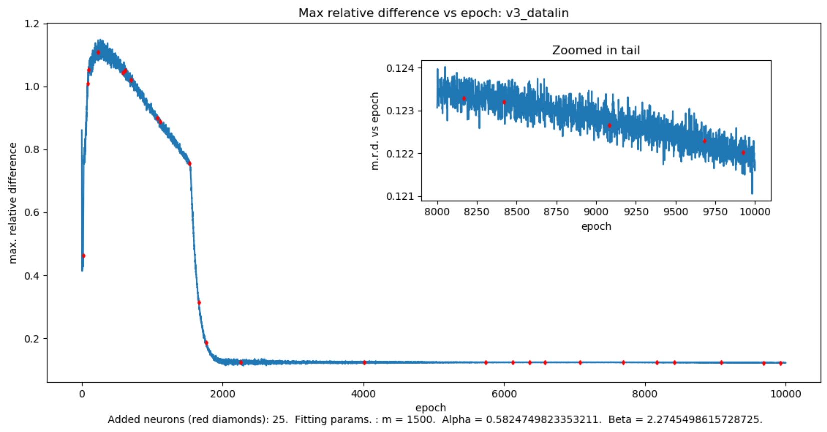


(b)

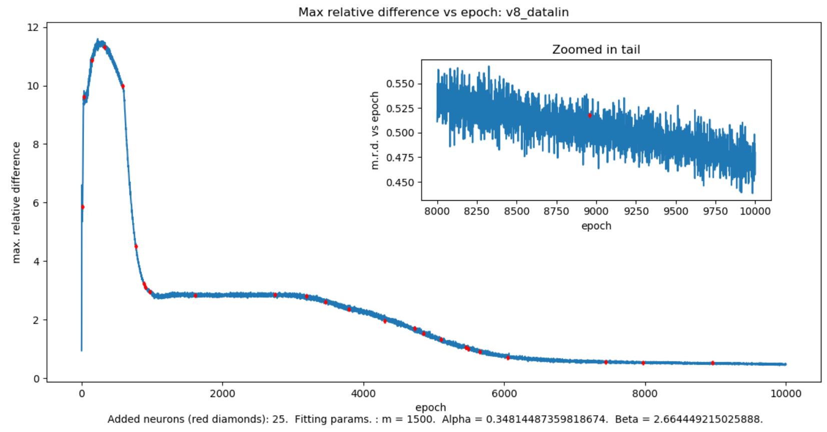


(c)

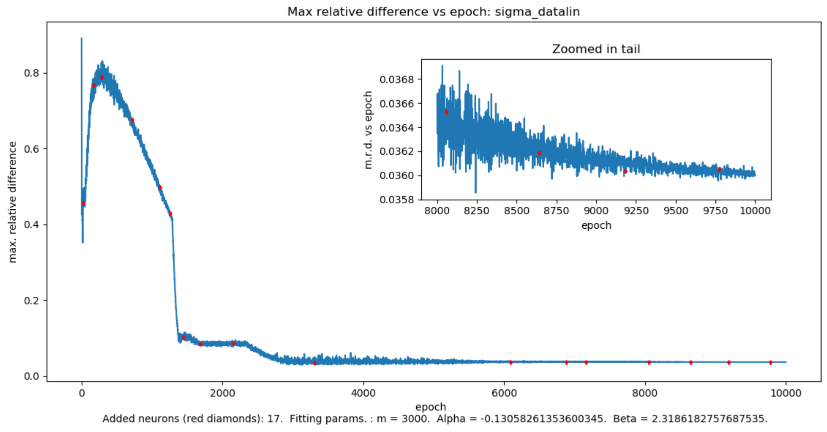
Figure 11: Plots of the m.r.d. vs epoch in the final fit of the PDFs. Fig.11a corresponds to the t8 data set, Fig.11b corresponds to the v data set and Fig.11c corresponds to g data set.



(d)



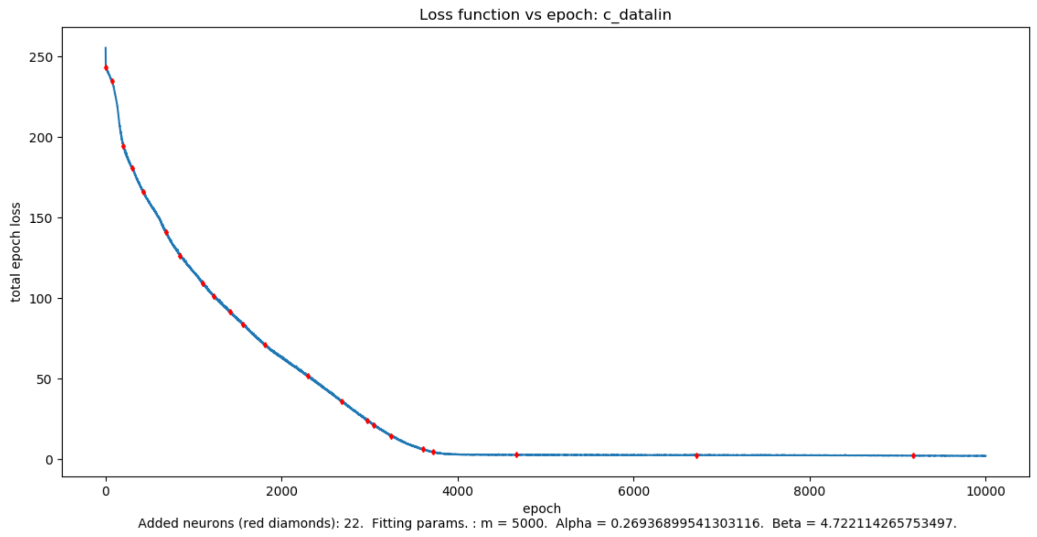
(e)



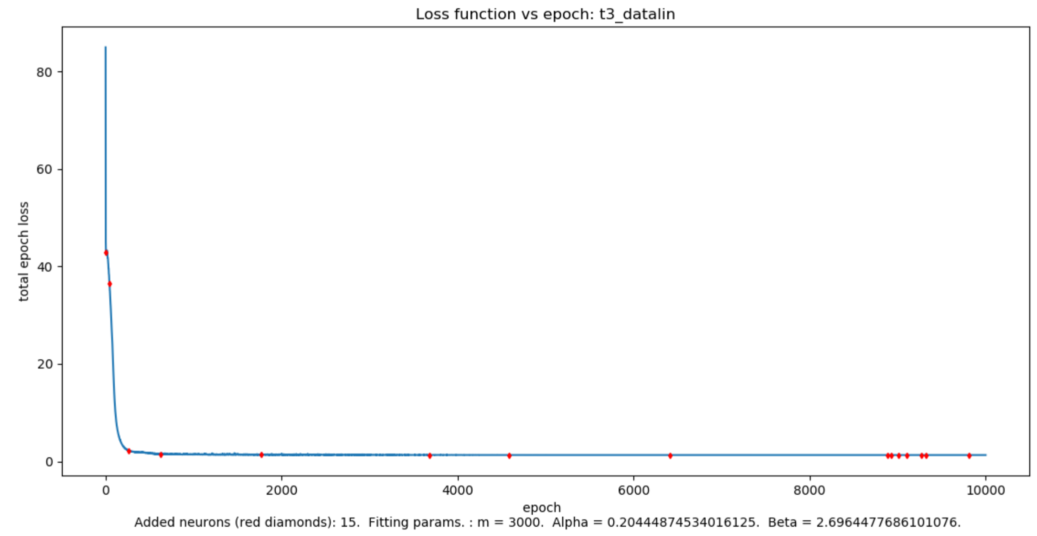
(f)

Figure 11: Plots of the m.r.d. vs epoch in the final fit of the PDFs. Fig.11d corresponds to the v3 data set, Fig.11e corresponds to the v8 data set and Fig.11f corresponds to sigma data set.

A Remaining plots of the final fits

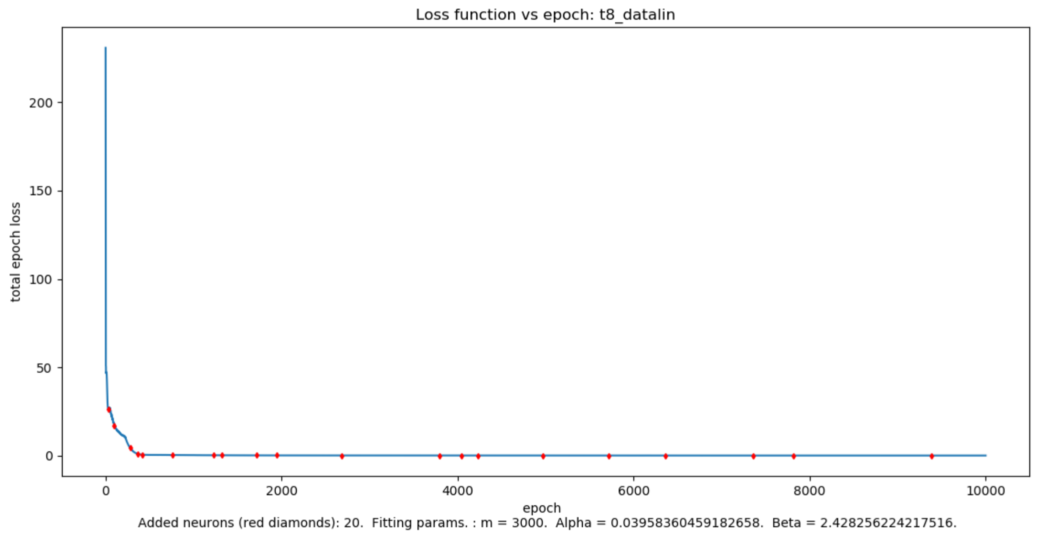


(a)

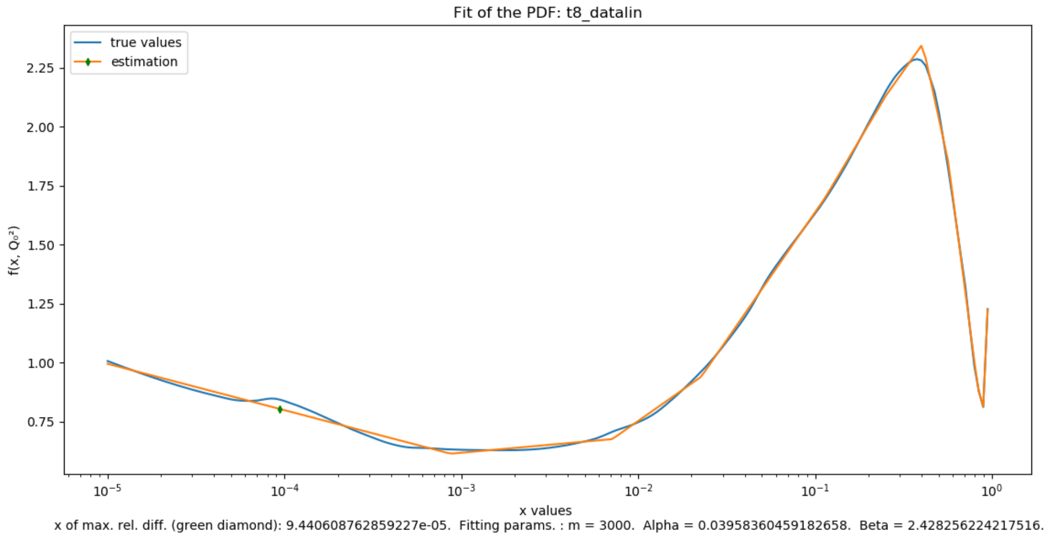


(b)

Figure 12: Plots of the loss vs epoch for the final fits corresponding to the c data set (Fig.12a) and the t3 data set (Fig.12b). For the m.r.d. vs epoch and the estimation of the PDF check Fig.9 and Fig.10 respectively.

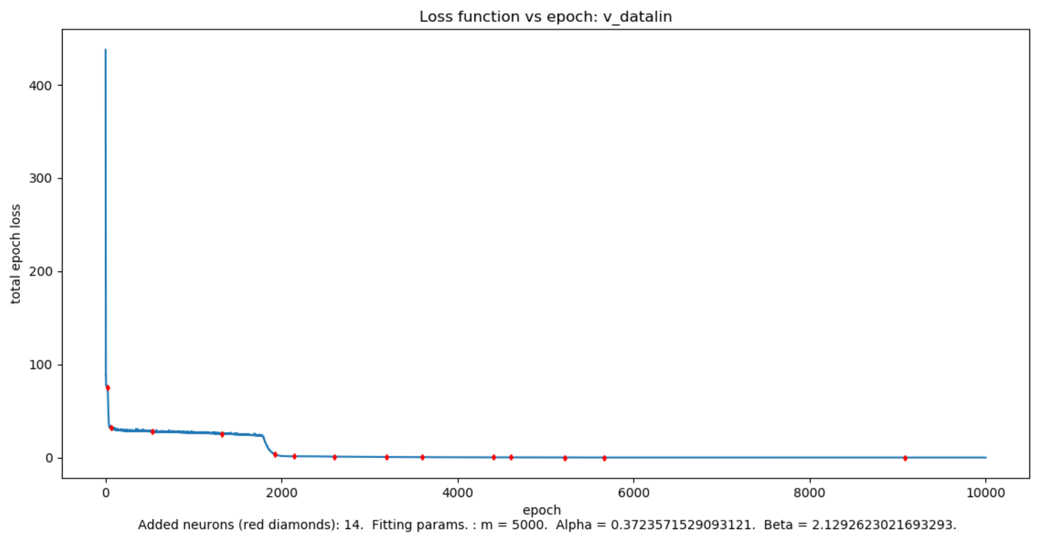


(a)

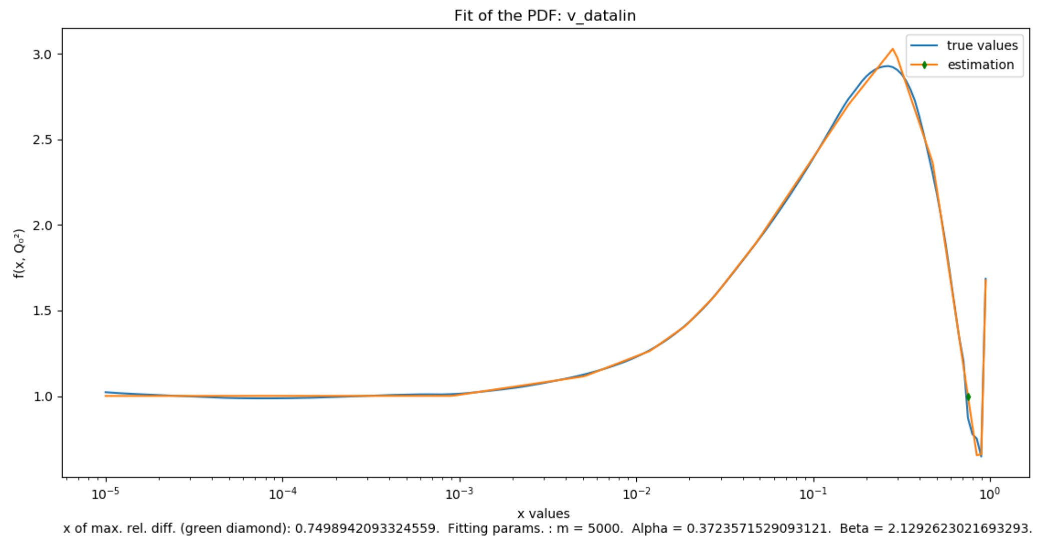


(b)

Figure 13: Plots of the loss vs epoch and the estimation of the PDF for the final fit corresponding to the t8 data set. For the m.r.d. vs epoch and the estimation of the PDF check Fig.11a.

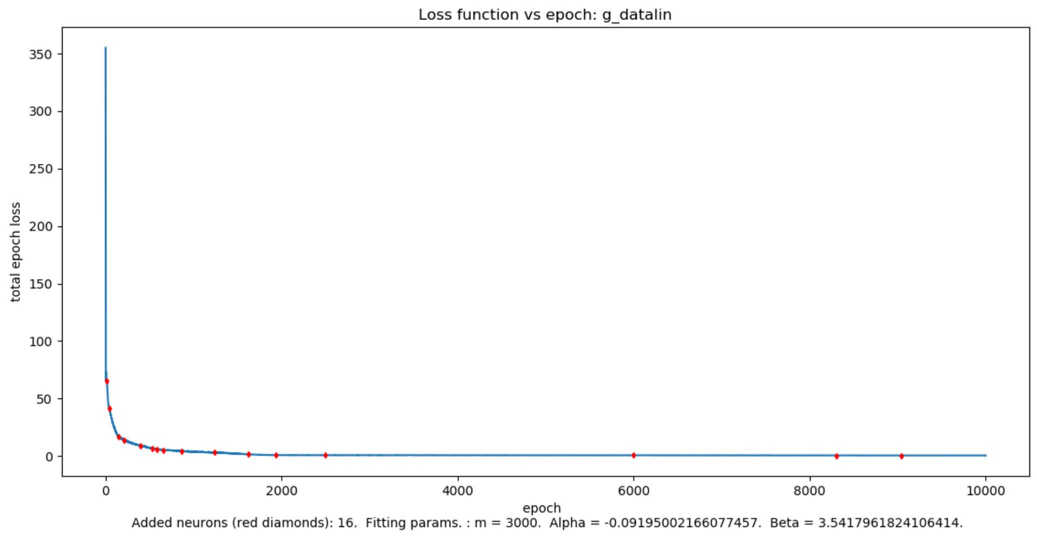


(a)

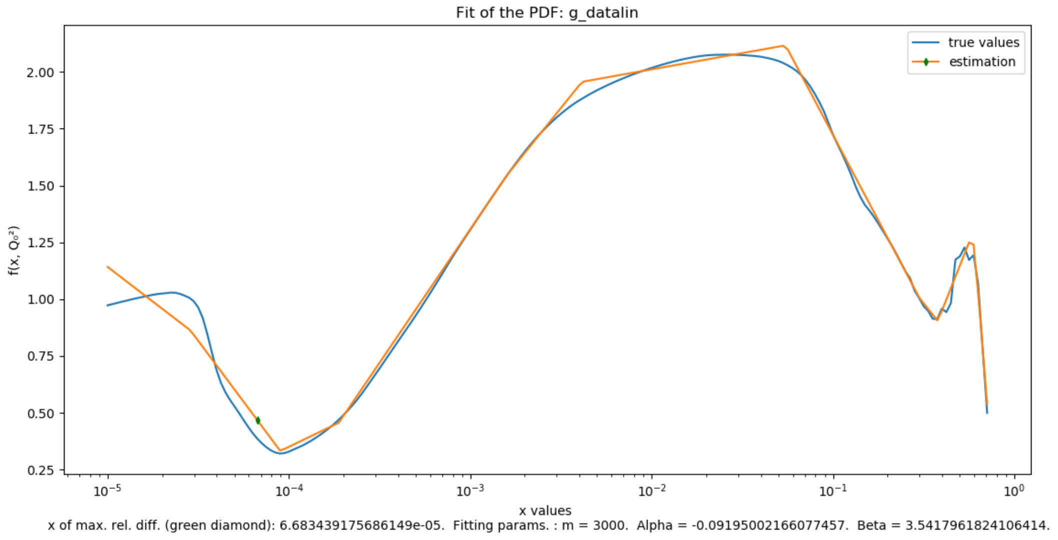


(b)

Figure 14: Plots of the loss vs epoch and the estimation of the PDF for the final fit corresponding to the v data set. For the m.r.d. vs epoch and the estimation of the PDF check Fig.11b.

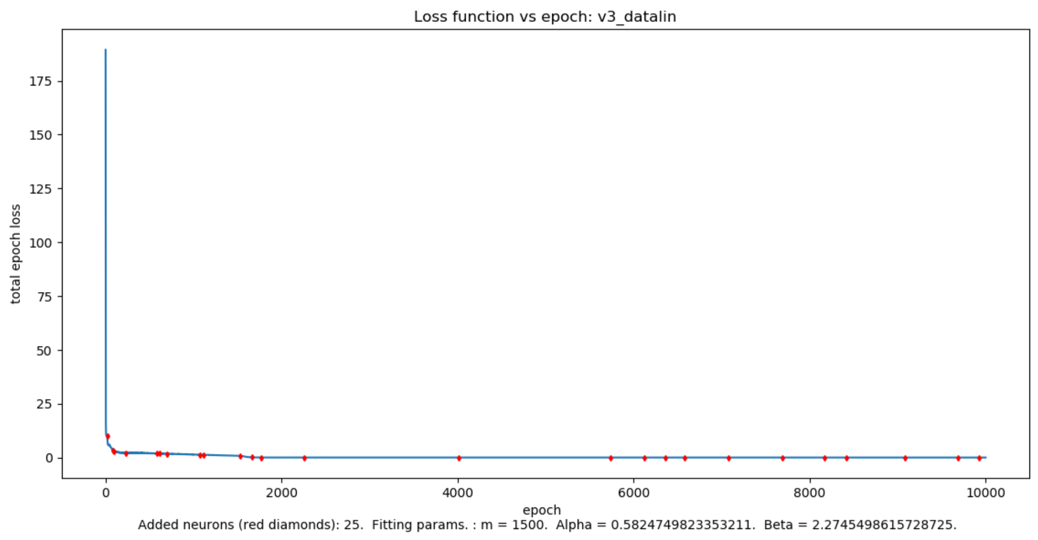


(a)

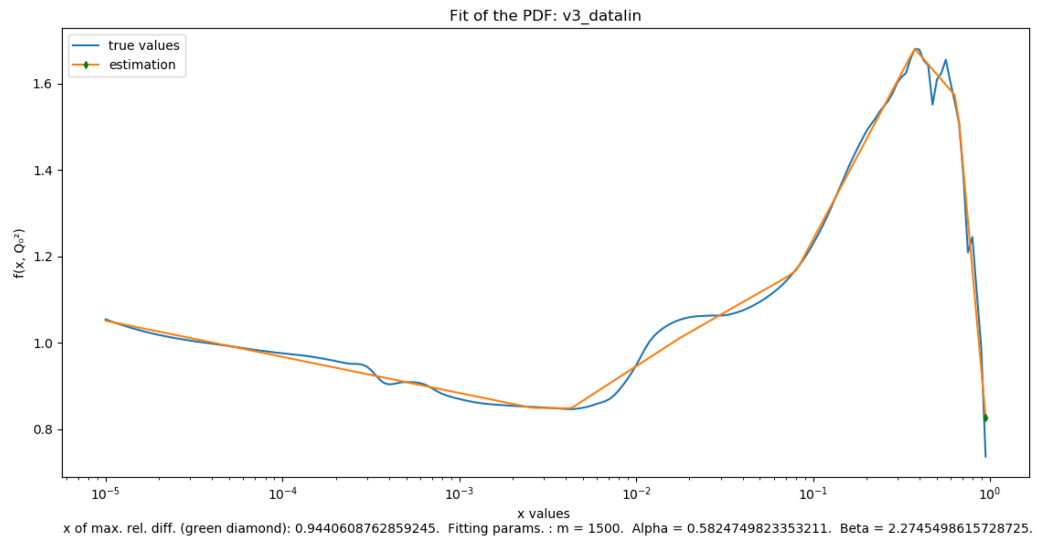


(b)

Figure 15: Plots of the loss vs epoch and the estimation of the PDF for the final fit corresponding to the g data set. For the m.r.d. vs epoch and the estimation of the PDF check Fig.11c.

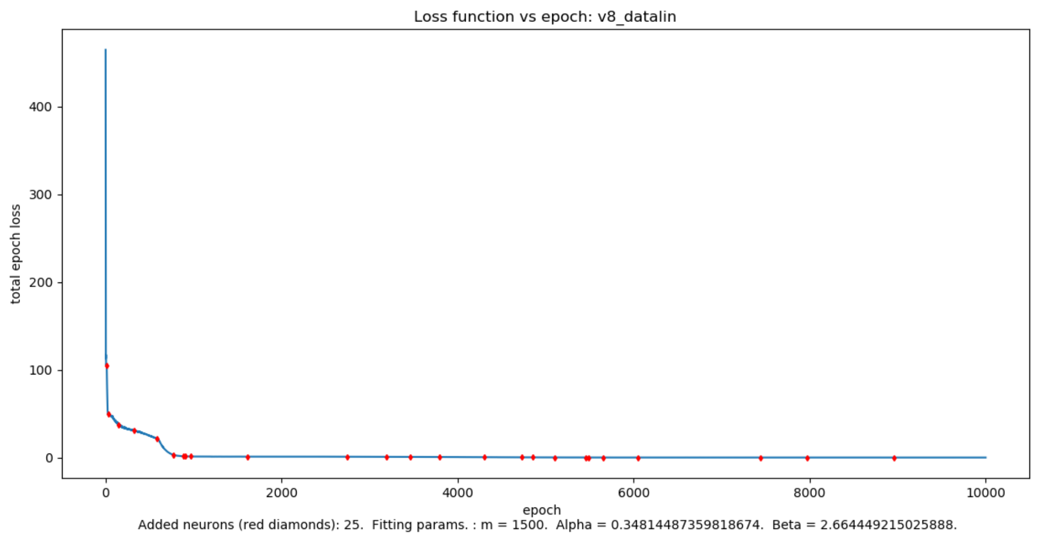


(a)

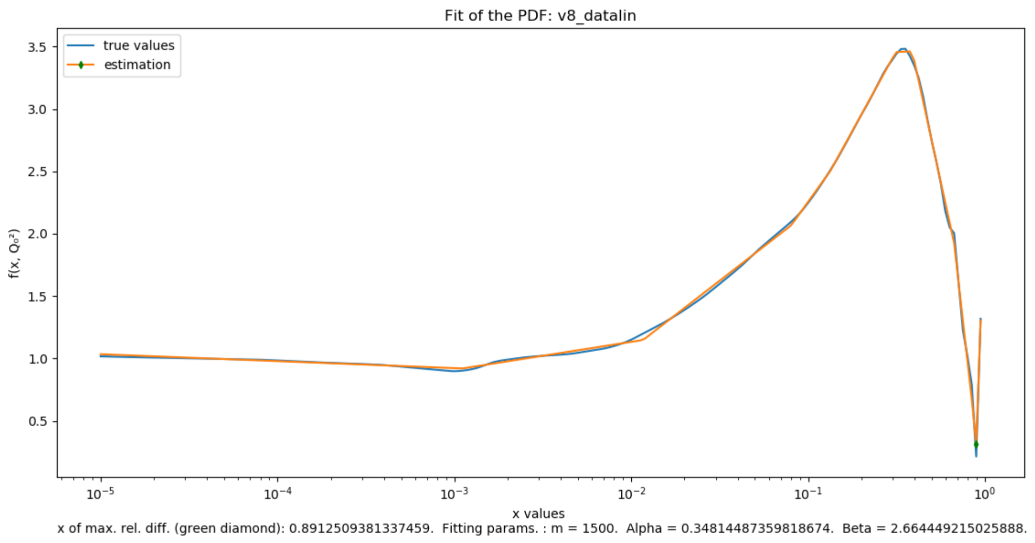


(b)

Figure 16: Plots of the loss vs epoch and the estimation of the PDF for the final fit corresponding to the v3 data set. For the m.r.d. vs epoch and the estimation of the PDF check Fig.11d.

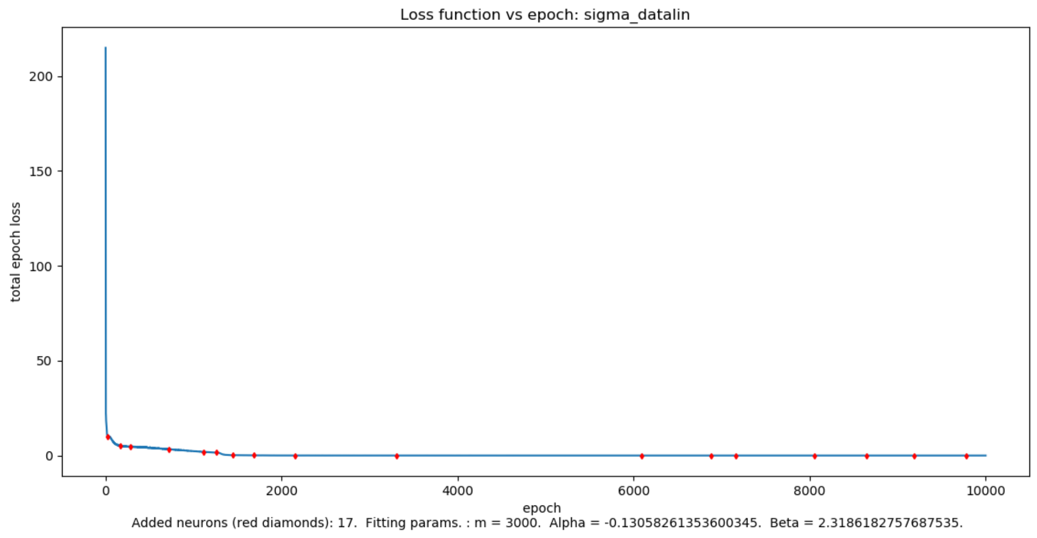


(a)

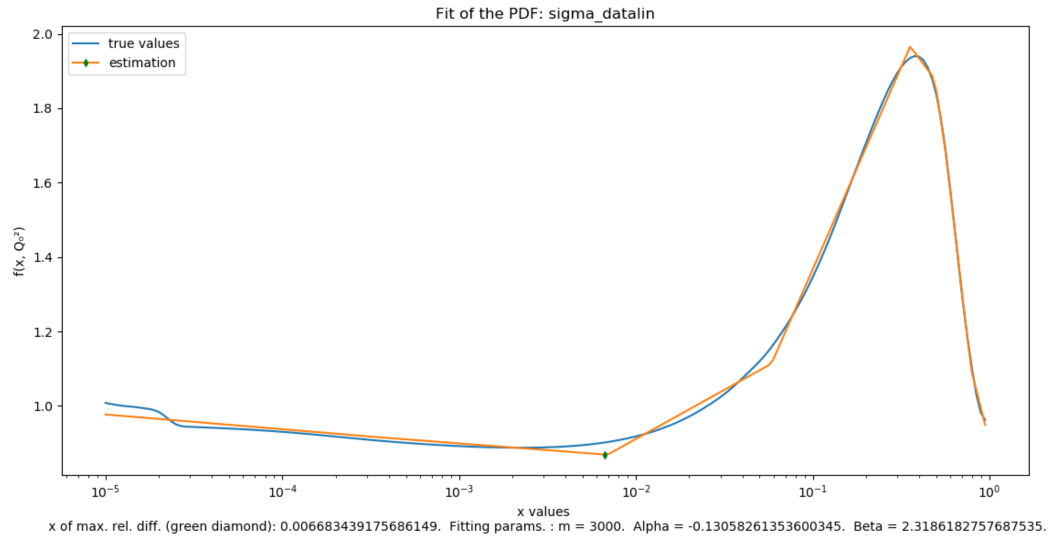


(b)

Figure 17: Plots of the loss vs epoch and the estimation of the PDF for the final fit corresponding to the v8 data set. For the m.r.d. vs epoch and the estimation of the PDF check Fig.11e.



(a)



(b)

Figure 18: Plots of the loss vs epoch and the estimation of the PDF for the final fit corresponding to the sigma data set. For the m.r.d. vs epoch and the estimation of the PDF check Fig.11f.

B Acknowledgements

This project has been possible thanks to the funding provided by the University of Edinburgh School of Physics and Astronomy Career Development Summer Scholarships. Also special thanks to Luigi Del Debbio for his supervision and to the PhD student Michael Wilson, for his guidance with the Pytorch library and the uncountable hours of technical support.

C Personal statement

The project was entirely based on Machine Learning, a completely wonderful yet unknown field of computer sciences for me. In its development, I have managed to apply some of my mathematical tools, as well as to learn many other concepts and definitions in data processing. Besides, I have greatly enhanced my coding abilities in Python and I have obtained superficial knowledge of a very powerful Python library: Pytorch. With respect to generic skills, being involved in a scientific project for the first time ever has taught me how to deal with frustration and dead ends, how to break down a massive problem into more manageable pieces and above all, how important is to question all your previous steps and to think from a different perspective. Having a sceptical view of what is going on is essential to contribute in the meetings and make breakthroughs.

D Lay Summary

High Energy Physics experiments, such as collisions in the LHC, produce massive amounts of information to process. However, the current classifiers used to separate the essential data from the bulk will become obsolete in few years, with the improvement of detectors and other measuring devices. Machine learning classifiers could be the solution, but they are not fast enough and hard to implement once they are trained. Here is where the Drone Neural Network comes into play, a recent kind of neural network whose main functionality is to learn from these previously trained machine learning classifiers and supposedly to mirror their behaviour, whilst keeping a lighter and simpler structure. The main aim of this project is to understand how the Drone works, to build your own from scratch using Pytorch: a powerful Python library for machine learning, and finally to try to assess how good a Drone is at learning from these classifiers.

References

[1] Duarte, J., S. Han, P. Harris, E. Jindariani, B. Kreinar, M. Kreis, R. Ngadiuba, N. Pierini, Z. Rivera, J. Tran, and S. Wu. *Fast Inference of Deep Neural Networks in FPGAs for Particle Physics.* Journal Of Instrumentation 13, no. 7 (2018): 1-28.

[2] Sean Benson[a], Konstantin Gizdov[b]. *NNDrone: a toolkit for the mass application of machine learning in High Energy Physics.* [a]Nikhef National Institute for Subatomic Physics, Amsterdam, The Netherlands. [b]School of Physics and Astronomy, University of Edinburgh, Edinburgh, United Kingdom.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* Adaptive Computation and Machine Learning. Cambridge, Massachusetts: MIT Press, 2016.

[4] Mason, J. C., and D. C. Handscomb. *Chebyshev Polynomials.* Boca Raton, Fla. ; London: Chapman and Hall/CRC, 2003. Print.

[5] Zahari Kassabov. (2019, February 18). *Reportengine: A framework for declarative data analysis (Version v0.27).* Zenodo. http://doi.org/10.5281/zenodo.2571601

[6] Buckley, Andy, et al. *LHAPDF6: Parton Density Access in the LHC Precision Era.* The European Physical Journal C, vol. 75, no. 3, 2015, pp. 1–20.

[7] Diederik P. Kingma[a], Jimmy Lei Ba[b]. *ADAM: A Method for Stochastic Optimization.* [a]University of Amsterdam, OpenAI. [b]University of Toronto.