Master Project: Softmax-based Error Scaling for RRN-based Observation

Moritz Gmeiner and Simon Bachhuber
Department of Artificial Intelligence in Biomedical Engineering
Friedrich-Alexander-Universität Erlangen-Nürnberg, 91058 Erlangen, Germany
Email: {moritz.gmeiner,simon.bachhuber}@fau.de

Abstract—Motion tracking of kinematic chains using inertial measurement units (IMUs) has uses in a wide variety of applications. While both magnetometer-free as well as sparse pose estimation are well-researched problems on their own, their combination is still challenging. A recent approach attempts to solve the observability question constructively, providing an estimation of the pose itself at the same time, by using an recurrent neural network (RNN) trained on synthetically generated motion data. In this paper that approach is extended by scaling the error distribution over time using a parametrised softmax function, with the goal of getting a better control over the error distribution over time. Both an unnormalised and a normalised approach are evaluated against the original method. While in its current form, no direct advantage over the original method could be realised, refinements of this method could still hold the potential for further investigation.

I. INTRODUCTION

IMUs refer to systems combining certain kinds of electronic sensors with the goal of estimating both the position and the orientation (sometimes in combination called the pose) of an object. Most commenly an IMU will have 9 degrees of measurement: 3 for linear acceleration, 3 for angular velocity, and 3 for magnetic field strength, one in the direction of every coordinate axis for each. Particularly important in recent years have become IMUs based on the microelectromechanical system (MEMS) technology, becoming small and cheap enough to embed them in a wide variety of use cases. Furthermore, while for simple rigid bodies without additional internal degrees of freedom (e.g. phones) a single IMU is sufficient, more complicated setups might require more IMUs to fully capture not only the global pose of the object relative to its environment, but also the relative poses of different parts of the object to each other. To fully capture the pose of a human including their arms, legs, as well as the head for example would require 10 IMUs, which can drive cost of material and computational complexity of the evaluation, especially for more complex objects and more fine-grained captures. Furthermore there is always the possibility of the failure of individual IMUs, which leads to incomplete measurement signals, forcing a repetition of experiments, which drives costs even more.

Generally we consider the case of measuring the pose of kinematic chains, which consist of a set of rigid links connected by joints, either revolut or prismatic. In a dense measurement setup, each link would carry an IMU, while in sparse inertial motion tracking (IMT), IMU measurement data are available only for a subset of the links. To complicate the problem even further, while IMUs generally provide 9 degrees of measurement (linear acceleration, angular velocity, and magnetic field strength), in certain applications, in particular indoors or in close proximity to significant amounts of metal, the magnetometer readings are unusable due to magnetic disturbances, leaving us with only linear acceleration and angular velocity data. In combination with a sparse measurement setup, this leaves us with significantly less information than in the dense 9D case.

The first problem to consider in these sparse IMT situations is that of observability: whether or not it is even possible to recover the complete internal pose of the measured kinematic chains from the sparse 6D IMU data and if so, for which configurations. The next step is then to actually estimate the internal pose, usually by estimating the state in the joint space. One novel approach, presented in [1], is to leverage the recent advances is the field of deep learning by using a RNN, implemented in JAX [2] and haiku [3] and trained on synthetically generated motion data to estimate the pose in joint space from the measurement data.

For this problem not only the mean error is of interest, but also how the error is distributed over time. An estimation with a certain mean error e over time might have an error of e consistently over the entire timespan, or it might have a perfect accuracy over 99% of the time and an error of $100e$ in the remaining 1%, which might make a significant different in certain application, in particular safety-critical ones. To this end we want to manipulate the error distribution before reducing it by multiplying the error distribution by a factor derived from a softmax of the error distribution itself, putting more weight on higher errors, with a tunable hyperparameter β which determines the strength of this scaling.

II. PROBLEM FORMULATION

The fundamental problem is magnetometer-free sparse IMT: a collection of rigid links is connected by joints (revolut or prismatic) and to some links, but not all, a 6D IMU is attached, giving (usually noise) measurements of linear acceleration and angular velocity. The goal is then to reconstruct the pose of the kinematic chain, that is the position and orientation of each link. In particular, we want to recover the kinematic chain's state in joint space, which describes the relative

pose of the kinematic chain. The method introduced in [1] uses a RNN-based observer (RNNo) [5], which is a RNN trained on synthetic data to map the IMU data $\mathbf{y} \in \mathbb{R}^{6n}$, where $\mathbf{y}$ is the combined measurement signal of all IMU accelerometer and gyroscope measurements and n is the number of attached IMUs, to the joint space $\mathbf{q} \in \mathbb{R}^k$, where k is the number of joints.

One particular issue arising in this setup is the distribution of the error over time $e(t) = \|\hat{\mathbf{q}}(t) - \mathbf{q}(t)\| \in \mathbb{R}^k$. Neural networks are usually trained on the mean of the errors $\frac{1}{T}\sum_t \|\mathbf{e}(t)\|$ as their cost function, which will minimise the mean of the error. For the IMT problem this means the RNNo will prefer e.g. an estimation that is very accurate at most times, but at some instances has a very high error, to an estimation that is somewhat less accurate, but has a consistent error over all times. In many applications, rather than minimising the mean of the errors, we would prefer to minimise a certain percentile. The p-th percentile of the error distribution e_p, for some p between 0 and 100, will be the smallest number such that at least p percent of errors are smaller than e_p. This mean by choosing e.g. $p = 99$ we can train our network to produce good estimations in at least 99% of time frames.

III. METHODS

Unfortunately, while appealing from a theoretical perspective, in practice percentiles are very unsuitable for training neural networks. Neural networks are trained by propagating error gradients backward through the network using the first derivative of the forward functions. However, numerically percentiles are calculated by first sorting the errors $e_1 < e_2 < \cdots < e_N$ and then selecting $e_p = \frac{1}{2}(e_i + e_{i+1})$ where $i = \lfloor pN \rfloor$, which means the propagated error gradient will only depend on the error at merely 2 of all in our case 1000 timesteps. This means most of the information contained in the error distribution will be simply thrown away, which in turn will make training extremely slow.

A. Unnormalised softmax scaling

An alternative approach is to instead significantly punish outliers in the error distribution. One way this can be accomplished is by scaling errors using the softmax function

$$\operatorname{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{e^{x_1} + \cdots + e^{x_N}} \tag{1}$$

where e is Euler's number. The exponential will turn additive differences $y = x + a$ into multiplicative scalings $e^y = e^a e^x$, making differences in the input much more pronounced. The degree to which these differences get amplified can be controlled by introducing a pseudo-emperature β, turning (1) into

$$\operatorname{softmax}(\beta\,\mathbf{x}) = \frac{e^{\beta x_i}}{e^{\beta x_1} + \cdots + e^{\beta x_N}} \tag{2}$$

While β is not exactly a temperature as in physics, it has a similar effect: the higher β the more the differences between the values will be amplified. This is particularly useful for colour-coding: in the following, lower values of β will generally be displayed as bluer ("colder") and higher values of β will be displayed as redder ("redder").

The softmax function can be used to scale the error before reducing it using a mean:

$$\mathbf{e}_{\text{sm}} = N \cdot \operatorname{softmax}(\beta\,\mathbf{e}) \odot \mathbf{e} \tag{3}$$

where N is the length of $\mathbf{e}$ and $\odot$ is the Hadamard product (component-wise multiplication). The multiplication with N is to renormalise $\mathbf{e}_{\text{sm}}$, as the output of the softmax function is a probability distribution. The cost function will then be

$$C = \sum_i \operatorname{softmax}(\beta\,\mathbf{e})_i e_i = \operatorname{softmax}(\beta\,\mathbf{e}) \cdot \mathbf{e} \tag{4}$$

which is of the same order as the cost function without scaling. Particularly for $\beta = 0$ we recover our unscaled cost function, since the softmax of a constant vector of length N will be $(\frac{1}{N}, \frac{1}{N}, \ldots, \frac{1}{N})$. On the other hand, for $\beta \to \infty$ the softmax will converge to a regular max, i.e. only the largest error will influence the training.

B. Normalised softmax scaling

While the approach shown in III-A works, it has a property that might be undesirable: during training the errors will generally decrease in magnitude, which means that the softmax scaling, which depends on the absolute differences between the different values, will have less and less impact as training progresses. This is somewhat equivalent to decreasing the temperature over time, which, while in some cases desirable, in other cases it might be not. To this end the errors can be normalised before the softmax function, changing equation (3) to

$$\mathbf{e}_{\text{sm}} = N \cdot \operatorname{softmax}\left(\beta \frac{\mathbf{e}}{\operatorname{mean}(\mathbf{e})}\right) \odot \mathbf{e} \tag{5}$$

This has the effect that even when the mean error decreases as training progresses, the strength of the scaling effect remains constant.

IV. RESULTS

A. Unnormalised softmax

The effect of using softmax scaling on the final training loss is displayed in Fig. 1. As can be seen quite clearly, for small β the final loss is very similar to the training loss without any scaling, as is expected since a small β means the effect of the scaling is small. After the $\beta = 10^{-2}$ point the final loss begins to rise steadily with increasing β until about $\beta = 10^3$, after which it starts to plateau off.

One caveat is that an increased β simply needs higher training times until the loss starts to converge to its final value. While the strongest proof that this is not the case would be given by retraining the higher β values for a larger number of epochs, Fig. 2 shows while for higher values of β the training performance degrades, the curve flattens off at the end of training, suggesting the observed final loss is indeed close to the best achievable training result.

This suggests that using softmax scaling can not improve the final training loss and in fact for β values

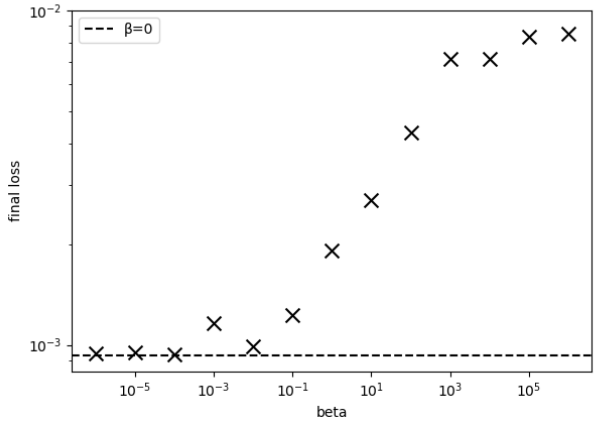

Figure 1
final loss over β with $\beta = 0$ baseline, for softmax scaling without normalisation. Final loss is the average loss over the last 100 of 1500 epochs.

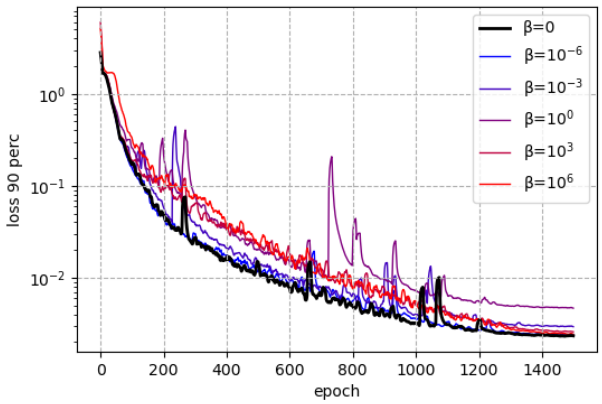

Figure 3
90th percentile of training loss over epochs for different values of β, without normalisation. Values have been smoothed using a linear Savitzky-Golay (`scipy.signal.savgol_filter` [6]) filter with window width 11

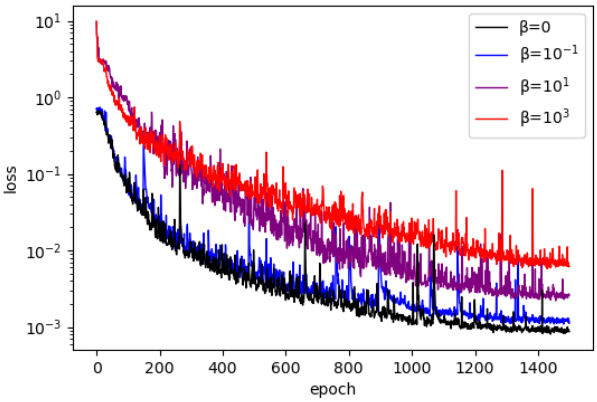

Figure 2
loss over epochs for the baseline $\beta = 0$ and different values of β in the unnormalised case

larger than 10^{-1} the final training loss starts to deteriorate significantly.

One possibility is that while the mean error does get worse, the percentiles for the error actually improve. The comparison of the 90th percentile for different β values to the baseline can be seen in Fig. 3. This shows rather clearly that for the 90th percentile across all values of β there is no improvement over the baseline, and in fact higher values of β tend to do worse again. The situation is very similar for the 50th, the 95th, and the 99th percentile, as can be seen in Fig. 8, Fig. 9, and Fig. 10 in the appendix respectively.

Overall it should be concluded that a positive effect of the softmax scaling on could not be established for any choice of β.

B. Normalised softmax

As noted in III-B, one major drawback of the previous approach was the fact that as the loss decreased during training, the size of the softmax scaling effect decreased significantly. Fig. 2 shows that the baseline loss decreases during training by about 3 orders of magnitude, which has the same effect as changing the base in the softmax from $e \approx 2.218$ to merely $e^{10^{-3}} \approx 1.001$. One possible way to counter this problem is, as laid out in III-B, to normalised the error to mean 1 before inputting it into the softmax function. Then the softmax scaling effect should be upheld over the entire training period, rather than decaying over time. The resulting final loss over the values of β can be seen in Fig. 4. It is immediately visible that the trend seen in Fig. 1 can be observed here again: for lower values of β the final loss hovers around the baseline final loss, until around $\beta = 10^{-3}$ the final loss starts to rapidly increase with increasing beta, in fact even stronger than in the unnormalised case.

The loss over epochs shown in Fig. 5 paints an even more extreme picture: while in the $\beta = 10^4$ case the loss decreases somewhat regularly, but merely 1 order of magnitude opposed to the 3 orders of magnitude in the baseline, for $\beta = 10^2$ the loss quickly decreases over the first 400 epochs, but then becomes unstable and begins jumping back up multiple times before converging to what looks to be its final value (that, in fact, is worse than the loss attained around the 400th epoch right before the phase of instability!). Overall, the degradation of training is even more pronounced compared to Fig. 2.

For sake of completeness, the 90th loss percentile for the normalised case can be observed in Fig. 6. It can again be seen that for small values of β ($\beta = 10^{-8}$), the curve closely follow the baseline curve, while for increasing values of β the 90th loss percentile also begins to deteriorate and become unstable.

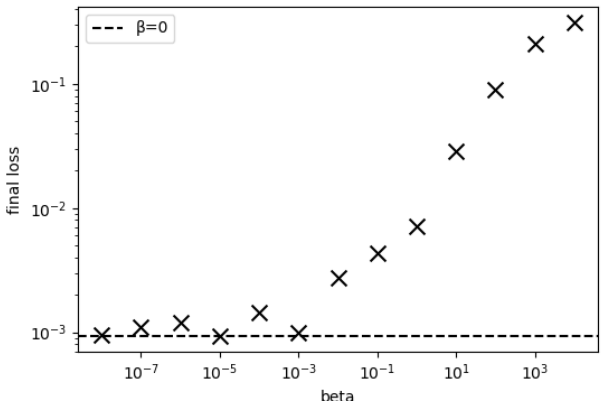

Figure 4
final loss over β with $\beta = 0$ baseline, for softmax scaling with normalisation. Final loss is the average loss over the last 100 of 1500 epochs.

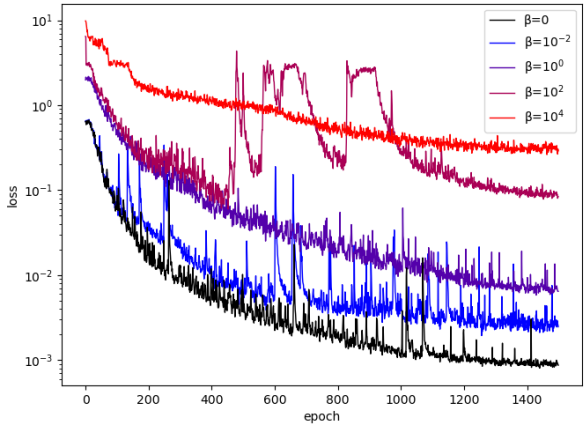

Figure 5
loss over epochs for the baseline $\beta = 0$ and different values of β in the normalised case

V. CONCLUSION

A method was developed to prescale errors in a RNNo to accomplish a better training performance with a particular focus on improving higher percentiles of the loss by putting more weight on larger errors. This was done by calculating a scaling factor using the softmax function on the errors, both with and without normalising the input to the softmax function. A temperature parameter β in the softmax was used to control the strength of the scaling effect. For both cases the RNNo was trained on a wide range of β values and training results were compared to a baseline network trained without any scaling.

It could generally be observed that for small values of β the effect was very small and the training performance closely mirrored the baseline, as was expected. As β was increased the training performance quickly

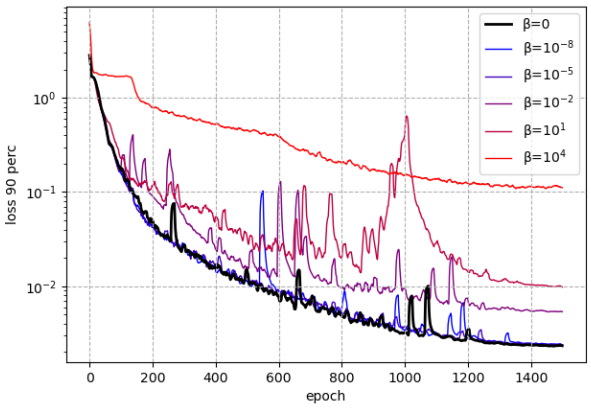

Figure 6
90th percentile of training loss over epochs for different values of β, with normalisation. Values have been smoothed using a linear Savitzky-Golay (`scipy.signal.savgol_filter` [6]) filter with window width 11

deteriorated significantly in both cases.

Possibilities for further research could be repeating the training with a finer selection of β values. While steps of 1 in the logspace are a quite common strategy for parameter tuning and give a good picture of the general behaviour of the β parameter, i.e. in this case small values are similar to the baseline while high values show deteriorating performance, it is entirely possible that for certain values in a smaller range certain benefits could be observed. In particular the transition area from no effect to worsening performance, as seen around $\beta = 1e-2$ in Fig. 1 and around $\beta = 1e-3$ in Fig. 4, could be a target for further investigation at a finer scale.

Another possible approach could be to make the scaling factor not depend on the size of the errors themselves, but rather of the position of the error in the sorted array, e.g. by first sorting the errors $e_1 < e_2 < \cdots < e_N$ and then using the normalised index as the input to the softmax function

$$s_i = \frac{e^{\beta \frac{i}{N}}}{\sum_{j=1}^{N} e^{\beta \frac{j}{N}}} \tag{6}$$

which would also achieve the desired effect of putting more weight on the higher errors, but in more consistent fashion, along the lines of a "soft percentile". The result can be seen in Fig. 7, showing that a small β will produce a nearly uniform distribution while a large β will produce a distribution heavily biased towards the larger error values.

Ultimately, while there is some potential for further research, the possibility has to be considered that the desired effect of improving the training quality of the RNNo is simply not being achieved and future energy might be better directed into different ways of improving training quality, like regularisation for example.

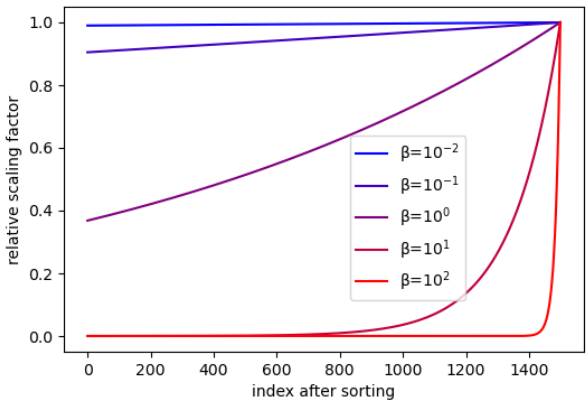

Figure 7
scaling factors softmax$(\beta \cdot (1, 2, \ldots, N)/N)$ over index after sorting. Normalised by dividing by the largest factor, for easier comparison.

References

[1] S. Bachhuber, D. Weber, I. Weygers, and T. Seel, "Rnn-based observability analysis for magnetometer-free sparse inertial motion tracking," in *2022 25th International Conference on Information Fusion (FUSION)*, 2022, pp. 1–8.

[2] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs. [Online]. Available: http://github.com/google/jax

[3] T. Hennigan, T. Cai, T. Norman, L. Martens, and I. Babuschkin. (2020) Haiku: Sonnet for JAX. [Online]. Available: http://github.com/deepmind/dm-haiku

[4] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," in *Nature*, vol. 585. Springer Science and Business Media LLC, Sep. 2020, pp. 357–362.

[5] S. Bachhuber. Rnn-based observer (RNNo). [Online]. Available: https://github.com/SimiPixel/neural_networks

[6] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," in *Nature Methods*, vol. 17, 2020, pp. 261–272.

[7] S. Bachhuber. x_xy: A tiny kinematic tree simulator. [Online]. Available: https://github.com/SimiPixel/x_xy_v2

[8] L. Campagnola. Vispy: interactive scientific visualization in python. [Online]. Available: https://doi.org/10.5281/zenodo.592490

Appendix A
ADDITIONAL WORK

Additionally to the main task, a side task was performed to add more geometry to the `x_xy` [7] package, which implements the underlying kinematics for the RNNo. Specifically, additionally to the already implemented `Box`, `Sphere`, and `Cylinder` geometries a new `Capsule` geometry was added, which represents a cylinder with a semi-sphere added to each base. That meant in particular adding a class which has the capsules radius and length as members, as well as a method which provides its moment of inertia. As there is (currently) no collision mechanics implemented, its boundary in space was not required, although it would of course possible to calculate this based on the length and radius if required.

Furthermore, the possibility to render kinematic trees using vispy [8] is also included in the `x_xy` package, but at that point only very simplistic rendering for the `Box` geometry was implemented. Rendering for `Sphere`, `Cylinder`, and `Capsule` geometries was implemented as well as the rendering system was somewhat refactored to prepare for additional rendering backends to be implemented.

Appendix B
ADDITIONAL FIGURES

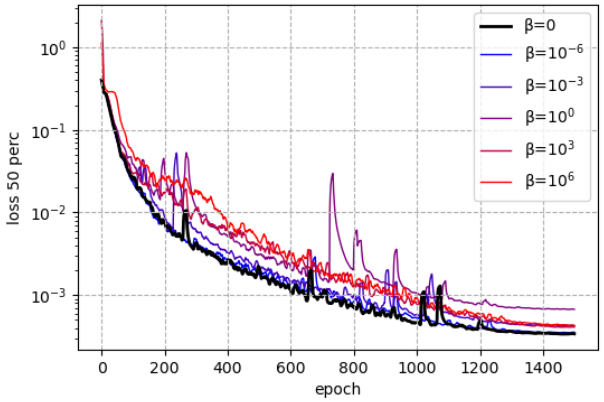

Figure 8
50th percentile of training loss over epochs for different values of β, without normalisation. Values have been smoothed using a linear Savitzky-Golay filter with window width 11

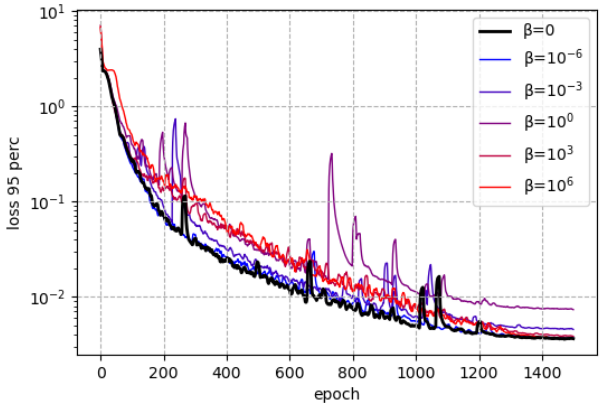

Figure 9
95th percentile of training loss over epochs for different values of β, without normalisation. Values have been smoothed using a linear Savitzky-Golay filter with window width 11

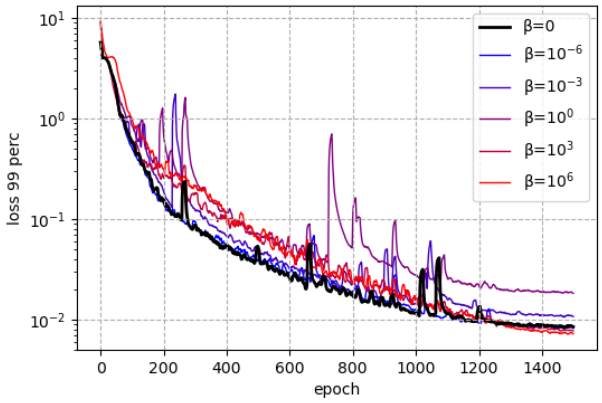

Figure 10
99th percentile of training loss over epochs for different values of β, without normalisation. Values have been smoothed using a linear Savitzky-Golay filter with window width 11