Computing Methods for Experimental Physics and Data Analysis - Description of the project

Luca Callisti, Marco Carotta, Igor Di Tota

1 Introduction

The main purpose of this project is to implement a lossy compression, using flow-based generative models.

The realization is obtained using Affine Autoregressive Flows, belonging to the family of Normalizing Flows. The workflow was as follows: we trained the model to map the data into Gaussian distributions, and from these we obtained the associated uniform distributions by applying the erf-function. The compressed distribution was obtained by rescaling the uniform distributions by the compression factor 2^N and then casting the result to integer. Applying the reverse procedure, the new data, the decompressed ones, were obtained from the inverse flow. As an application of Normalizing Flows, it was also shown how new data can be generated from the Gaussian distributions into which the original data are mapped: the sampling procedure. We used the implementation of normalizing flows provided by the nflows package.
This work was inspired by the Baler tool development, where a lossy compression is realized through an autoencoder. For this reason the datasets used were the same, so that a comparison could be made.

2 Basics and Definitions

The main idea of flow-based modeling is to express $\mathbf{x}$, the D-dimensional vector data, as a transformation F of a real vector $\mathbf{z}$, also of D-dimension, sampled from $p_z(\mathbf{z})$, the base distribution. The defining property of flow models is that the transformation F must be invertible and both F and F^{-1} must be differentiable, so must be a diffeomorphism.
A normalizing flow generally consists of a sequence of invertible transformation functions f_i. Exploiting the change of variables, defined by:

$$p_x(\mathbf{x})d\mathbf{x} = p_z(\mathbf{z})d\mathbf{z} \tag{1}$$

the density of $\mathbf{x}$ can be obtained as:

$$p_x(\mathbf{x}) = p_z(\mathbf{z}) \det \left| \frac{d\mathbf{z}}{d\mathbf{x}} \right| \quad \text{where} \quad \mathbf{z} = F^{-1}(\mathbf{x}) \tag{2}$$

so:

$$p_x(\mathbf{x}) = p_z(F^{-1}(\mathbf{x})) \det \left| \frac{d\mathbf{z}}{d\mathbf{x}} \right| \tag{3}$$

Taking the logarithm, the loss function can be express as:

$$\mathcal{L}(\phi) = -\mathbb{E}_{p_x^*(\mathbf{x})}[\log\big(p_z(F^{-1}(\mathbf{x};\phi))\big) + \log(\det \mathbb{J}_{F^{-1}}(\mathbf{x};\phi))] \tag{4}$$

An illustrative diagram of the model is shown in Fig 1.
Our use of flow, however, will be in reverse: taking advantage of the invertibility of the transformation, the data distribution $p_x(\mathbf{x})$ will be mapped to the base distribution, which in our case is a normal distribution $\mathcal{N}(0,1)$. The Gaussian will be the starting point for data compression.

3 Datasets

The data we used were taken from CMS collaboration (2017). JetHT primary dataset in AOD format from Run of 2012, from the CERN Open Data Portal. Events stored are a collection of proton-proton collisions, in which multiple jets may be present for each event. `CERN Virtual Machine` was used to download the data, using the release `CMSSW_5_3_32`. Several datasets were created for training, validation and test, with each entry corresponding to a jet. Files were uploaded to Dropbox and not to GitHub because of their excessive size; `wget` is used to get them.
The event is characterized by 24 features. On the `.ipynb` files all their names are given, although, for memory issues and to make training more sustainable, we used only the first four. These four features constitute the components of the 4-vector *jet*:

$$j^\mu = (p_T, \eta, \phi, m) \tag{5}$$

where: p_T is the momentum of the jet perpendicular to the direction of the colliding proton beams, η is the spatial coordinate describing the angle of a particle relative to the beam axis, ϕ is the azimuthal angle measured around the beam axis and m is the mass of the jet.
The number of entries for the training dataset is set to 236413. This value was also chosen for the validation dataset.

4 Preprocessing

To facilitate training, the data has been preprocessed: transformations are applied to bring the data into a form that is easier to learn. The Preprocessor class was created for this purpose; the implementation of which is shown in Listing 1. We apply the logarithm on p_T and on $mass$, after shifting the data to ensure that they were far from divergence. Then we bring all data to zero mean and unit variance with `StandardScaler`. Finally we apply the `QuantileTransformer` to bring η and ϕ into a normal distribution, this last transformation is particularly suitable for training of angular distributions. The `forward` method performs these operations, while the `backward` method does the reverse transformation. The `preprocessor_settings` dictionary is created to contain this information.

The original distributions are shown in Fig. 2, while the preprocessed data is shown in the Fig 3.

5 Model training

The model is created with the `model_definer` function, contained in the `model.py` file. The body of the function is reported in Listing 2. The first three arguments are used to define the model: number of features, the number of f_i (`num_iterations`) and the number of hidden features, a parameter used to construct a suitable neural network to generate F. The other parameters are used to better manage the training; in fact, they are involved in adjusting the learning rate through the definition of a scheduler with `ReduceLROnPlateau`. `model_definer` returns the flow, the optimizer, for which `Adam` from Pytorch was used, and the scheduler.

Before the actual training, different values of the hyperparameters were used to train the flow for 30 epochs. The number of iterations for model optimization was `range_iterations=([2, 4, 6, 8])`, while the number of hidden features used was `range_hidden_features = ([16, 32, 64, 128])`. The Fig. 4 shows the results. All these tests for various models are done by training the flow in batches. The batch size for the training data was set as `batch_size = 1024`, while for the validation data `val_batch_size = 10000`.

Two models were chosen for training, the Table 1 summarises their properties, while Table 2 gives the details and results of the training. Furthermore, the plots of the losses of the two models are shown respectively in Figure 5 and in Figure 6.

Name	Num iterations	Hidden features	Patience	Factor	Min lr	Initial lr	Parameters
model1	6	64	10	0.5	10^{-6}	0.001	104880
model2	4	32	10	0.5	10^{-6}	0.004	18592

Table 1 – Parameters used to define models.

Name	Epochs	Batch size	Validation batch size	Loss	Validation Loss	Time [min]
model1	1000	10000	10000	2.857	2.899	37
model2	300	32	10000	2.768	2.946	311

Table 2 – Training parameters and results.

In both trainings the scheduler changed the learning rate, arriving at the minimum value in ten steps for `model1`, while instead it stops at a value of 6.25×10^{-5} in six steps for `model2`. In addition, an early stopping procedure was implemented: a counter was increased by one unit each time the two losses differed more than $\Delta = 3$. In either case, the condition for arrest was not met.

6 Compression

Before addressing compression, we now report an example application of Normalizing Flows: sampling. After sending our data into a normal distribution we can use the flow to generate new data. The Fig. 7 and Fig. 8 show the original and sampled distributions compared, for `model1` and `model2`.

To achieve lossy compression we created the `Compressor` class (reported in Listing 3) that has two methods: `compress` and `decompress`. Through the learned flow, the original data are molded into gaussian distributions, subsequently we can obtain uniform distributions using the error-function. Compression consists of casting as integer after rescaling the unit interval of the uniform distribution by a factor of 2^N, where N is the number of compression bits used. To make sure that the inverse error function does not return infinite numbers we used Scikit-learn's `MaxAbsScaler` in the `compress` method, to rescale each feature so that the distribution is within $[-3, +3]$. In the `decompress` method we apply the inverse transformation.

The expected normal and uniform distributions obtained from the flow for both models are now presented: Fig. 9 and Fig. 10 for `model1` and Fig. 11 and Fig. 12 for `model2`. Two values of compression bits N were chosen: N=13, in order to have a good compromise between compression ratio R (6) and relative error (7), and N=20 to have about the same R as in Baler's article, about $R = 1.7$. Fig. 13 shows the distributions of the variables after the compression performed with `model1` and $N = 13$, while Fig. 14 shows the distributions obtained with `model2` and with $N = 20$.

Normality tests were performed on these distributions, using normaltest from Scipy, the results are shown in Table 3. Having fixed the significance level $\alpha = 0.05$, it can be seen that all distributions do not respect the null hypothesis.

Name	Distribution	Statistic				p-value			
		p_T	η	ϕ	m	p_T	η	ϕ	m
`model1`	Before	7	37	128	61	2.79×10^{-2}	1.16×10^{-8}	1.41×10^{-28}	5.11×10^{-14}
	After ($N = 13$)	86	39	244	112	2.05×10^{-19}	3.88×10^{-9}	1.30×10^{-53}	5.26×10^{-25}
	After ($N = 20$)	136	62	316	166	2.74×10^{-30}	3.28×10^{-14}	2.34×10^{-69}	8.29×10^{-37}
`model2`	Before	85	383	129	439	3.98×10^{-19}	5.25×10^{-84}	9.28×10^{-29}	5.51×10^{-96}
	After ($N = 13$)	155	208	76	711	2.29×10^{-34}	8.61×10^{-46}	2.91×10^{-17}	4.87×10^{-155}
	After ($N = 20$)	208	378	49	845	7.19×10^{-46}	9.78×10^{-83}	2.45×10^{-11}	3.63×10^{-184}

Table 3 – Results of normality tests for the two models. *Before* refers to the distributions before compression, *After* refers to after compression.

Results on compression are now presented. The original data, used for the training, and decompressed data were compared: the first row of the following figures shows their distributions, the second represents the ratio between the original and decompressed data, and the last contains the differences of the two datasets. Fig. 15 and Fig. 16 refer to `model1`, respectively for $N = 13$ and $N = 20$; similarly Fig. 17 and Fig. 18 for `model2`.

In reference to Baler's article, there were two metrics used: the residuals, which in our case we called difference, and the response, defined as the residuals divided by the original data. The plots for this second metric are now presented: Fig. 19 and Fig. 20 for `model1` and Fig. 21 and Fig. 22 for `model2`. Note that in the legends, in addition to the average, the percentage of `NaN` and infinities obtained from the division and subtracted is also reported.

The compression ratio is defined as

$$R = \frac{\text{size(input file)}}{\text{size(decompressed file)}} \tag{6}$$

To calculate it, new `.txt` files are created to contain only the data actually used, so we take into account the reductions made to the number of entries and the number of features. To be consistent with Baler, the `.npz` format is also used to create new files. The compression results are reported in the Table 4. Two things must be noted: the size of the generated file does not depend on the model, so the ratio R does not depend on it either; also, the size of the input and output files does not depend on the number of compression bits N used.

File		size(`.txt`) [MB]	size(`.npz`) [MB]
input		12.90	5.36
output		16.97	6.69
compressed	$N = 13$	4.83	2.34
	$N = 20$	6.69	3.18
R	$N = 13$	2.67	2.29
	$N = 20$	1.93	1.68

Table 4 – Size of input, output and compressed data files. Sizes are given in MB.

Before performing the test, some quantities were studied as the compression bits N changed. In particular, we focused on the compression ratio, again computed on the training data, the mean of the module of the residuals, defined as residuals = original - decompressed, and the relative error:

$$\textit{Relative error} = \sqrt{\left(\frac{\textit{original} - \textit{decompressed}}{\textit{std(original)}}\right)^2} \tag{7}$$

Note that the average within the root is done features by features, with `axis=0`, while the second average returns only one value, averaging over the four variables. In this case, although the last two quantities depend on the model, the plots are very similar. We therefore report the plots for `model1` in Fig. 23. Instead, the numerical values for our N compression bits are reported in Table 5:

`model1`	Mean of module of residual	Relative error	`model2`	Mean of module of residual	Relative error
$N = 13$	3.06×10^{-3}	1.44×10^{-3}	$N = 13$	2.44×10^{-3}	7.80×10^{-4}
$N = 20$	2.60×10^{-5}	1.60×10^{-5}	$N = 20$	2.09×10^{-5}	1.13×10^{-5}

Table 5 – Mean of module of residual and relative error for `model1` on the left and `model2` on the right.

7 Test

Now the testing part is performed, so compressing new data with the previously trained flow. Five subsets of 200000 entries each were extracted from the test dataset. Results concerning residuals and response for both models, with fixed $N = 13$,

are presented. Of the five tests, only one is shown. Fig. 24 and Fig. 25 refers to `model1`, Fig. 26 and Fig. 27 to `model2`. Of the five calculated compression ratios R, the averages with standard deviations, associated with the two models, are shown here in Table 6.

	`.txt`	`.npz`
`model1`	2.682 ± 0.011	2.309 ± 0.048
`model2`	2.702 ± 0.007	2.308 ± 0.004

Table 6 – Averages of the compression ratio R obtained from the five test datasets. N is set at 13.

8 Conclusions

This project showed how it was possible to use Normalizing Flows to achieve lossy compression of HEP data. The results show that compression is effective, there is a reduction in file size despite the fact that the distributions originated from the flow are not perfectly Gaussian and uniform. The results obtained with the compression models with $N = 20$ are comparable to those of Baler, with reference to the response metric. The main limitations encountered during the implementation were mainly due to Google Colab, both due to GPU memory issues and the time limit on usage, so much so that a subscription to Colab Pro had to be purchased.

Images section

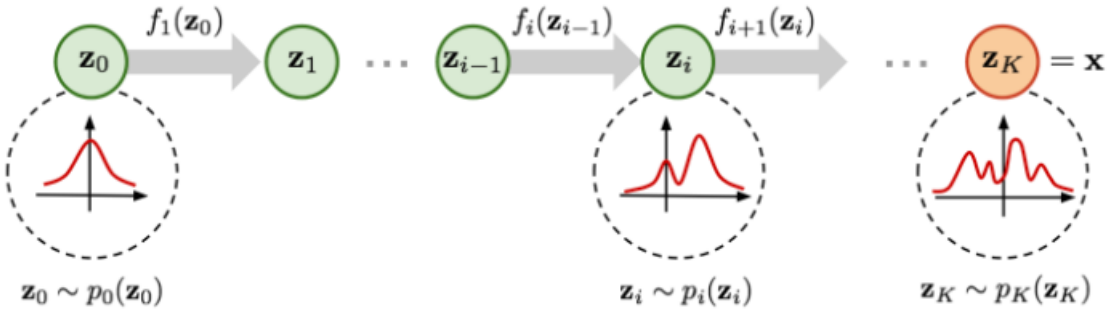

Figure 1 – Illustration of a normalizing flow model, taken from Lilian Weng's blog.

```python
class Preprocessor:
    def __init__(self, settings):
        self.index_log = settings['index_log']
        self.range_quantile = settings['range_quantile']
        self.n_quantiles = settings['n_quantiles']
        self.standard_scaler = preprocessing.StandardScaler()
        self.quantile_scaler = preprocessing.QuantileTransformer(n_quantiles=self.n_quantiles,
    output_distribution='normal')

    def forward(self,data):
        y = np.copy(data)

        for i in self.index_log:
            y[:,i] = np.log(10 + y[:,i])

        self.standard_scaler = preprocessing.StandardScaler().fit(y)
        y = self.standard_scaler.transform(y)

        if self.range_quantile:
            self.quantile_scaler = preprocessing.QuantileTransformer(n_quantiles=self.n_quantiles,
    output_distribution='normal').fit(y[:,min(self.range_quantile):max(self.range_quantile)+1])
            y[:,min(self.range_quantile):max(self.range_quantile)+1] = self.quantile_scaler.transform(y[:,
    min(self.range_quantile):max(self.range_quantile)+1])

        data_preprocessed = y
        return data_preprocessed

    def backward(self, data_reconstructed_preprocessed):
        data_reconstructed = np.copy(data_reconstructed_preprocessed)
        if self.range_quantile:
            data_reconstructed[:,min(self.range_quantile):max(self.range_quantile)+1] = self.
    quantile_scaler.inverse_transform(data_reconstructed[:,min(self.range_quantile):max(self.
    range_quantile)+1])

        data_reconstructed = self.standard_scaler.inverse_transform(data_reconstructed)
        for i in self.index_log:
            data_reconstructed[:,i] = np.exp(data_reconstructed[:,i]) - 10
        return data_reconstructed
```

Listing 1 – Definition of *Preprocessor* class.

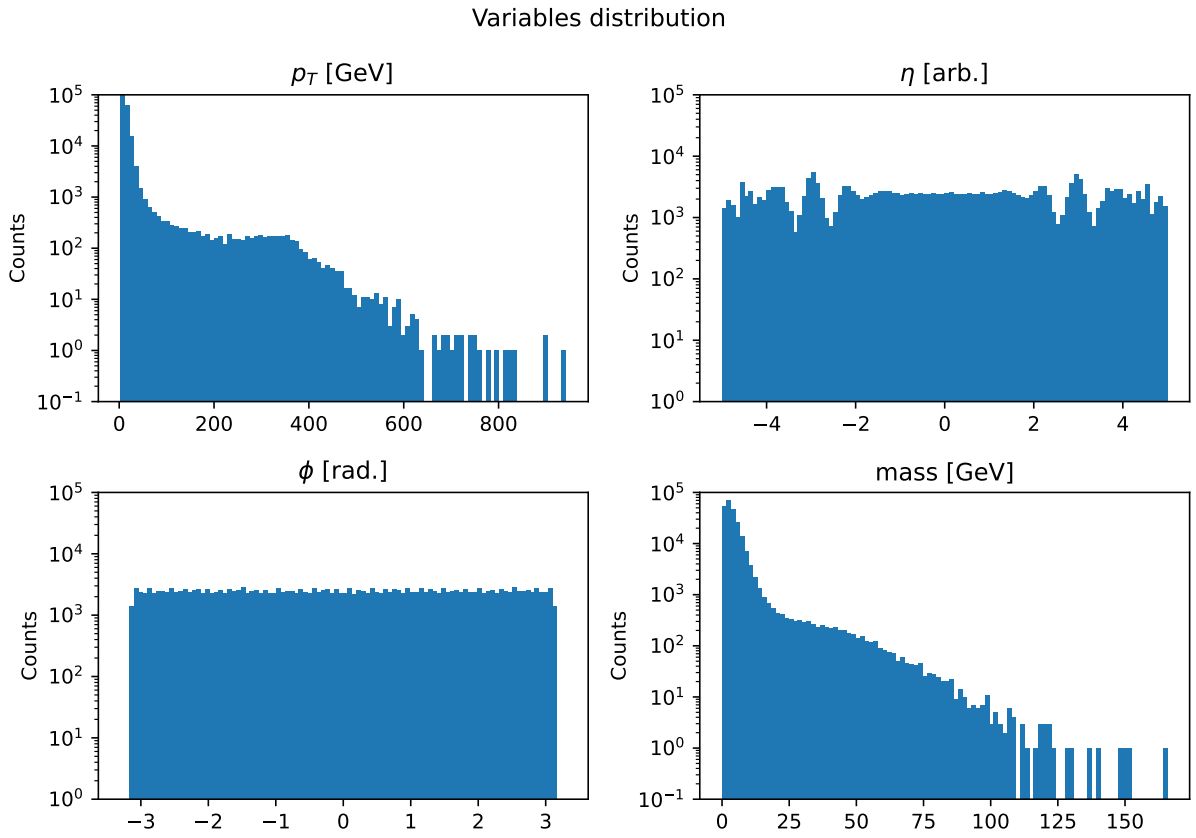

Figure 2 – Distributions of original features.

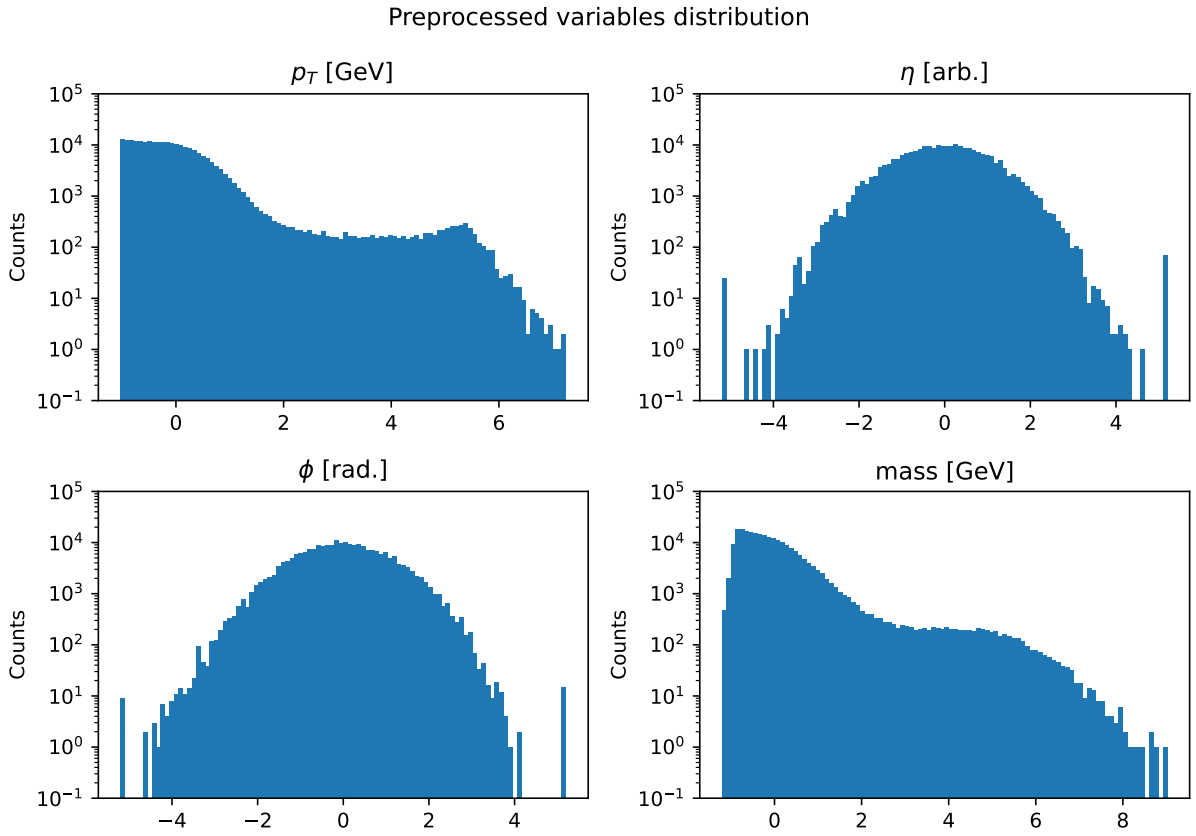

Figure 3 – Distributions of preprocessed features.

```
def model_definer(num_features, num_iterations, hidden_features, patience, factor, min_lr, initial_lr):

    base_dist = StandardNormal(shape=[num_features])

    transforms = []
    for _ in range(num_iterations):
        transforms.append(RandomPermutation(features=num_features))
        transforms.append(MaskedAffineAutoregressiveTransform(features=num_features,hidden_features=
    hidden_features))

    transform = CompositeTransform(transforms)

    flow = Flow(transform, base_dist)
    num_parameters = sum(p.numel() for p in flow.parameters() if p.requires_grad)
    print('Num. iterations = {}, Num. hidden_features = {}, Num. trainable parameters = {}'.format(
    num_iterations,hidden_features,num_parameters))
    optimizer = optim.Adam(flow.parameters(), lr=initial_lr)

    flow = flow.to('cuda')

    scheduler = ReduceLROnPlateau(optimizer, patience=patience, factor=factor, min_lr=min_lr)

    return flow, optimizer, scheduler
```

Listing 2 – *model_definer* function.

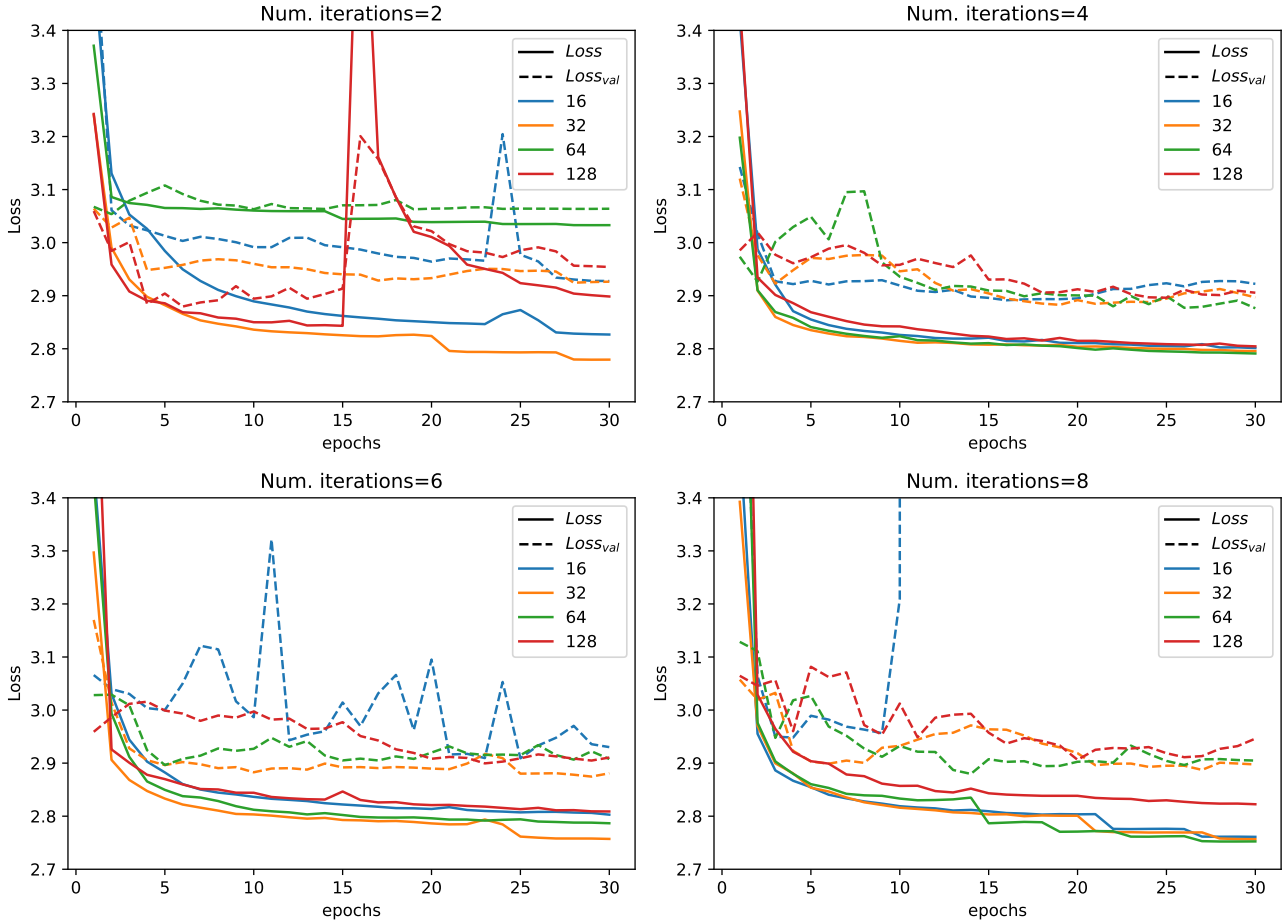

Figure 4 – Training and validation loss functions for different hyper-parameters configurations. Different colors correspond to a different number of hidden features.

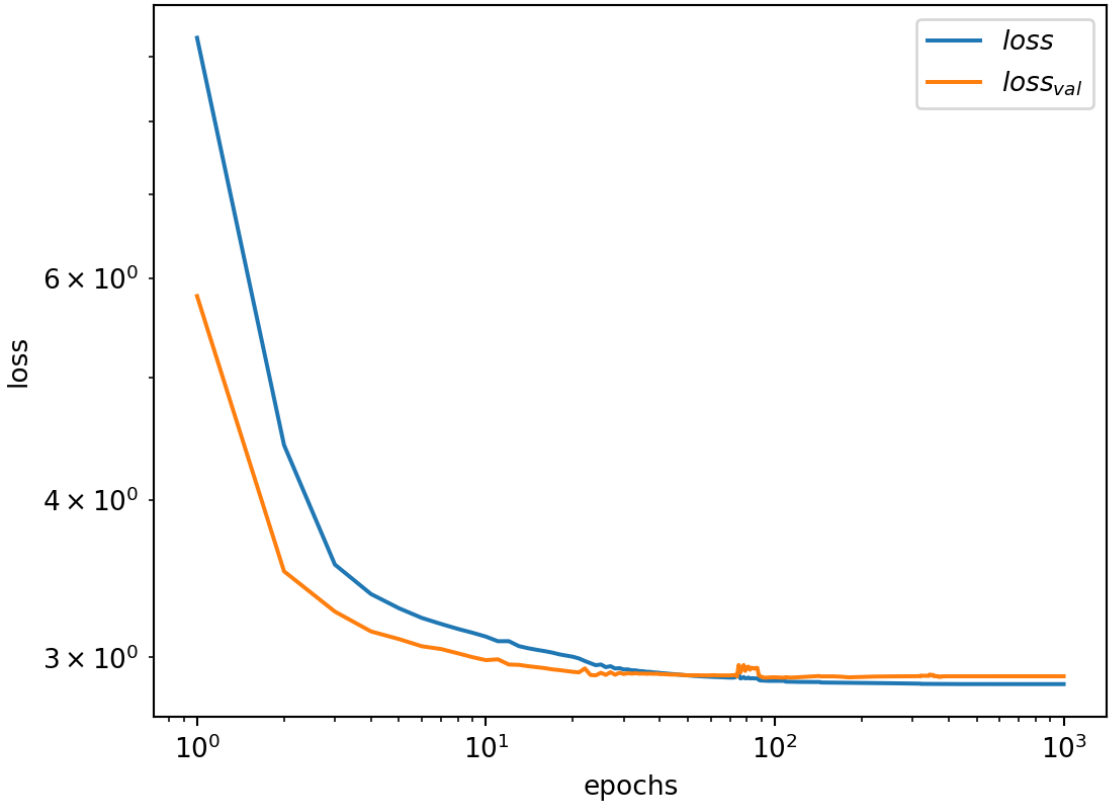

Figure 5 – Training and validation loss for `model1`. Results are reported in Table 2.

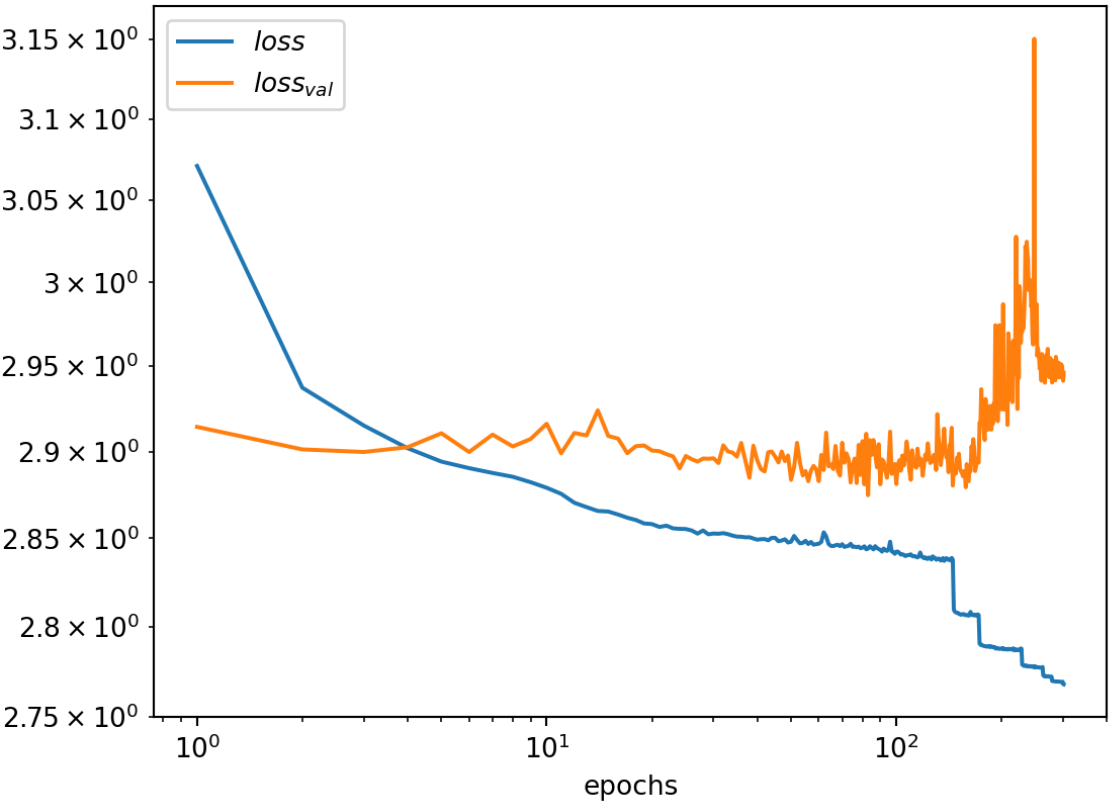

Figure 6 – Training and validation loss for `model2`. Results are reported in Table 2.

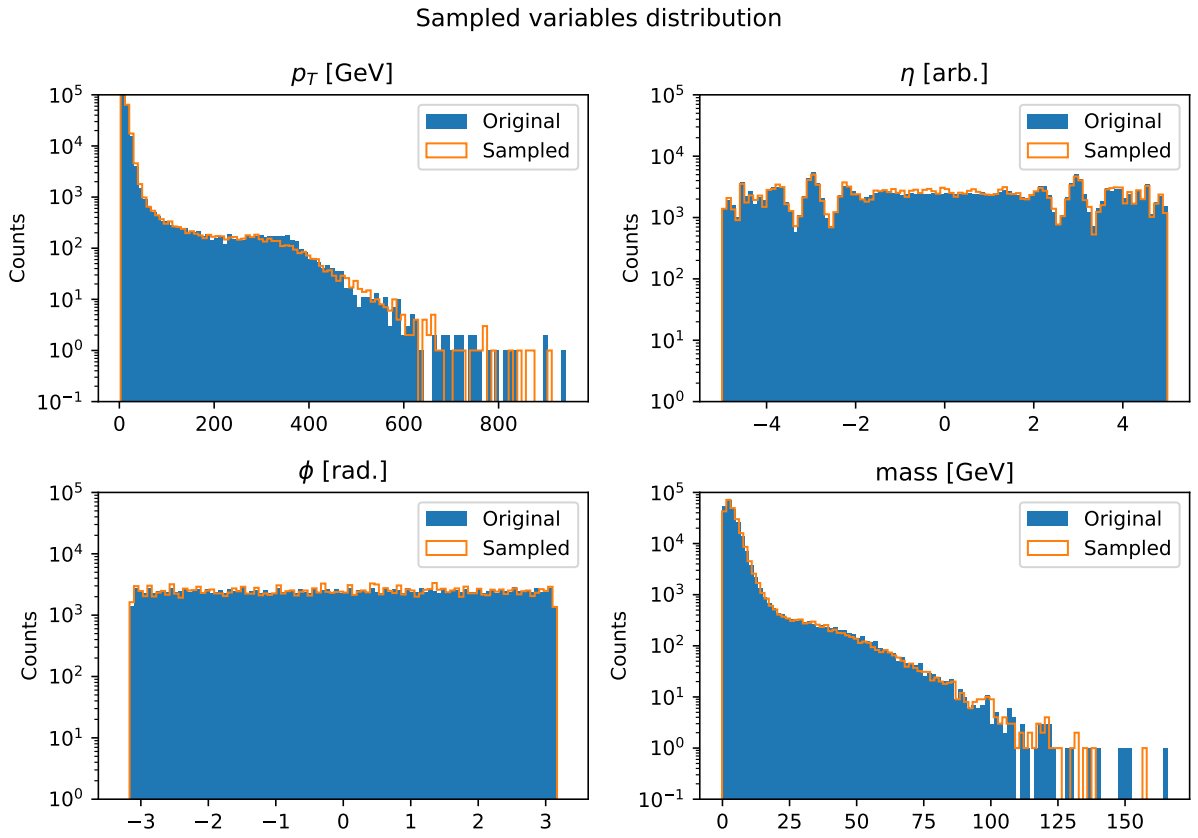

Figure 7 – Original and sampled variables distributions for `model1`.

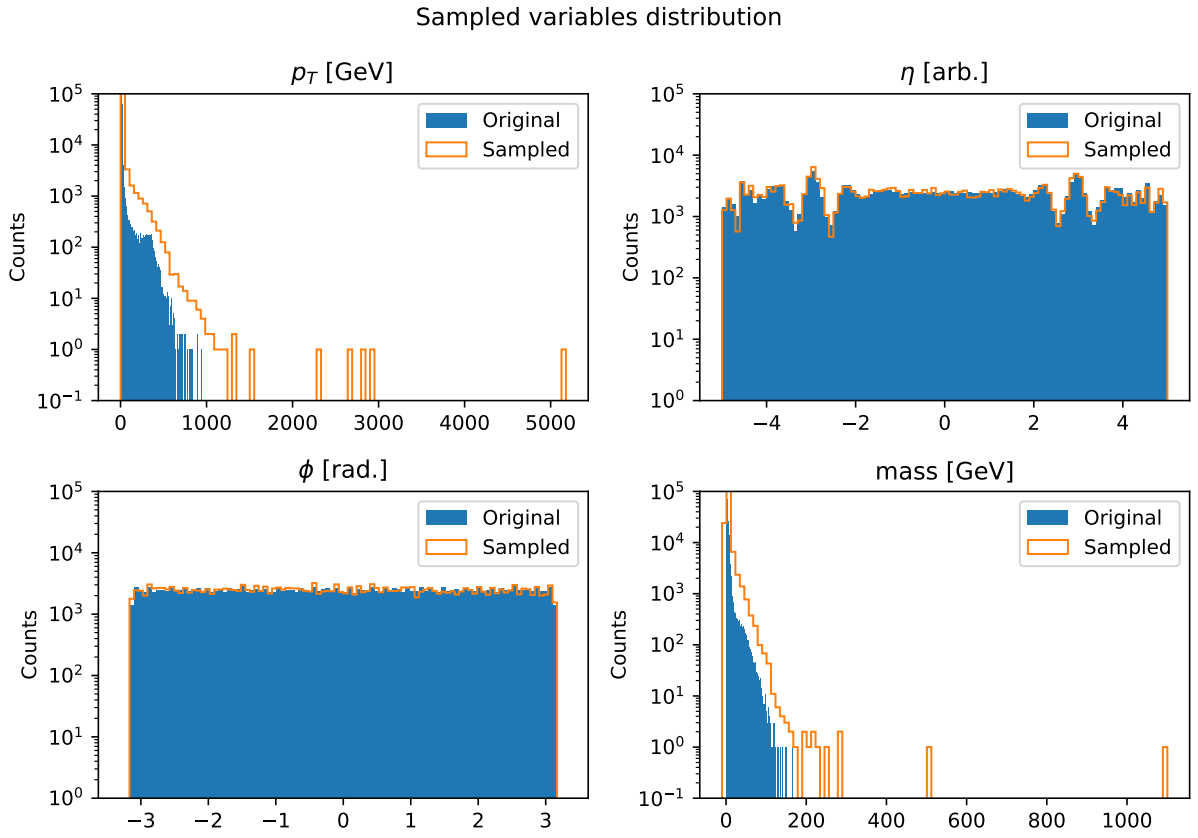

Figure 8 – Original and sampled variables distributions for `model2`.

```
class Compressor:
    def __init__(self, flow, N, limit):
        self.flow = flow
        self.N = N
        self.limit = limit
        self.maxabsscaler = preprocessing.MinMaxScaler()

    def compress(self,data):
        if isinstance(data, np.ndarray):
            data = torch.tensor(data).to('cuda').float()

        gaus_tensor = self.flow.transform_to_noise(data)
        gaus = (gaus_tensor.cpu().detach().numpy())

        self.maxabsscaler = preprocessing.MaxAbsScaler().fit(gaus)
        gaus_prep = self.maxabsscaler.transform(gaus)
        gaus_prep = self.limit*gaus_prep

        unif = scipy.special.erf(gaus_prep)
        unif_prep = unif * 2**self.N
        data_compressed = unif_prep.astype(int)

        return data_compressed, gaus, unif

    def decompress(self,data_compressed):
        unif = data_compressed/2**self.N
        gaus = scipy.special.erfinv(unif)

        gaus_post = self.maxabsscaler.inverse_transform(gaus)
        gaus_post = gaus_post/self.limit

        gaus_tensor = torch.tensor(gaus_post).to('cuda').float()
        data_tensor_decompressed, _ = self.flow._transform.inverse(gaus_tensor)
        data_decompressed = data_tensor_decompressed.cpu().detach().numpy()

        return data_decompressed, gaus
```

Listing 3 – Definition of *Compressor* class.

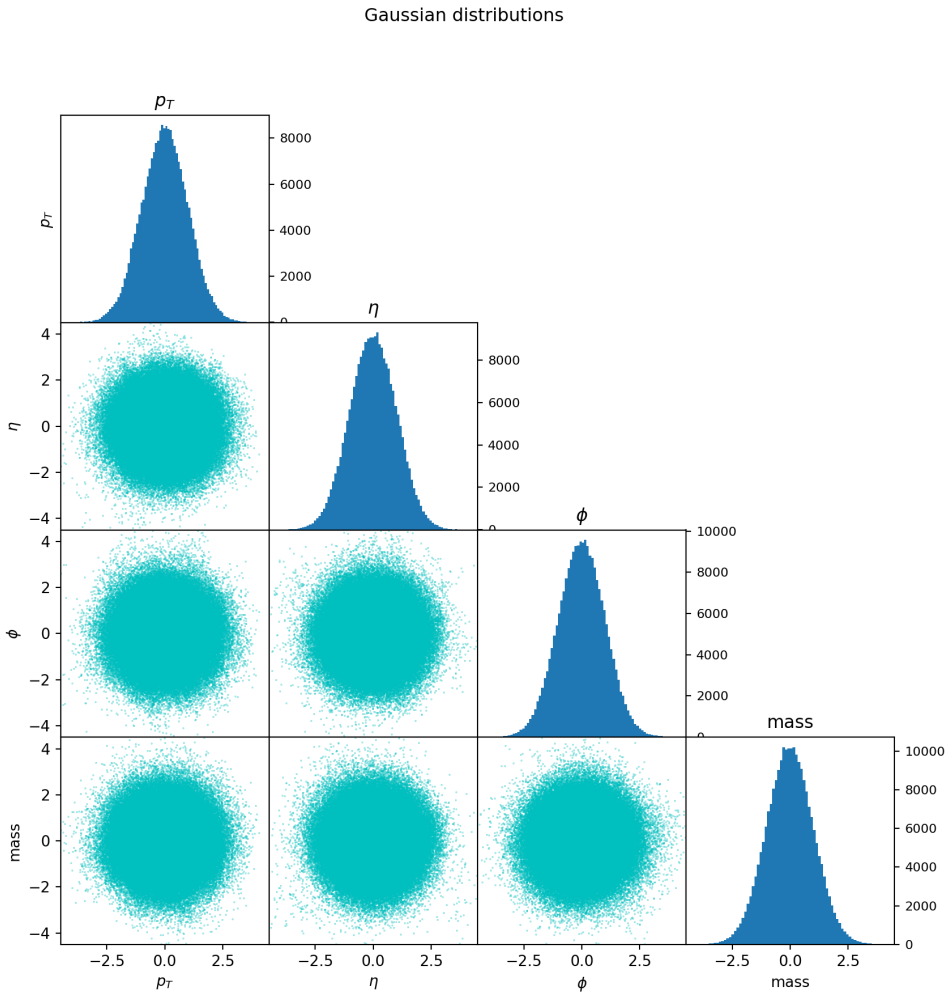

Figure 9 – Variables distributions on the diagonal and scatter plots of each pair of variables for `model1`.

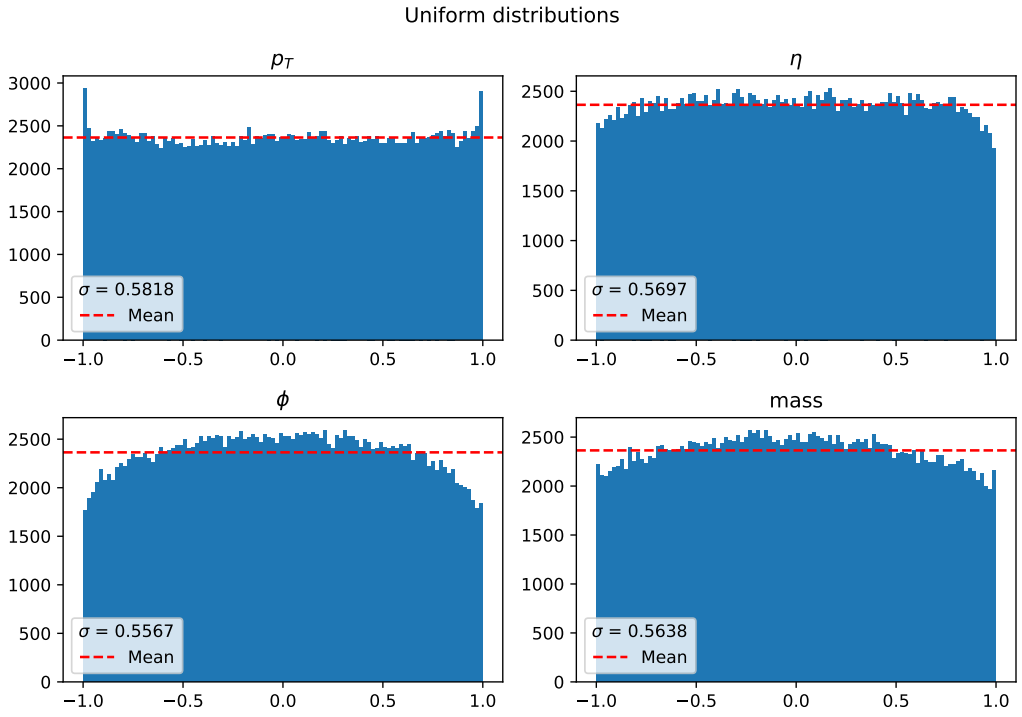

Figure 10 – Variables distributions after applying the error function for `model1` - expected uniform distribution. Mean and std are also shown: a standard deviation of $1/\sqrt{3} \approx 0.577$ is expected for a uniform distribution.

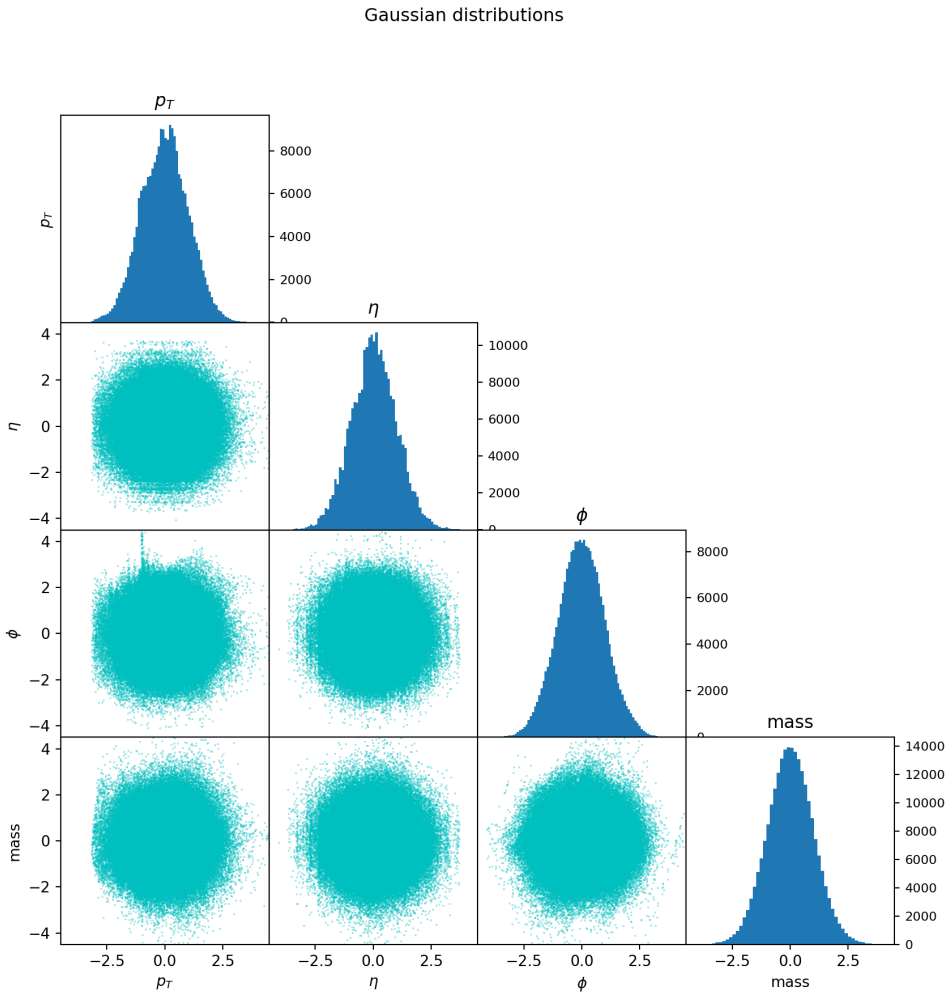

Figure 11 – Variables distributions on the diagonal and scatter plots of each pair of variables for `model2`.

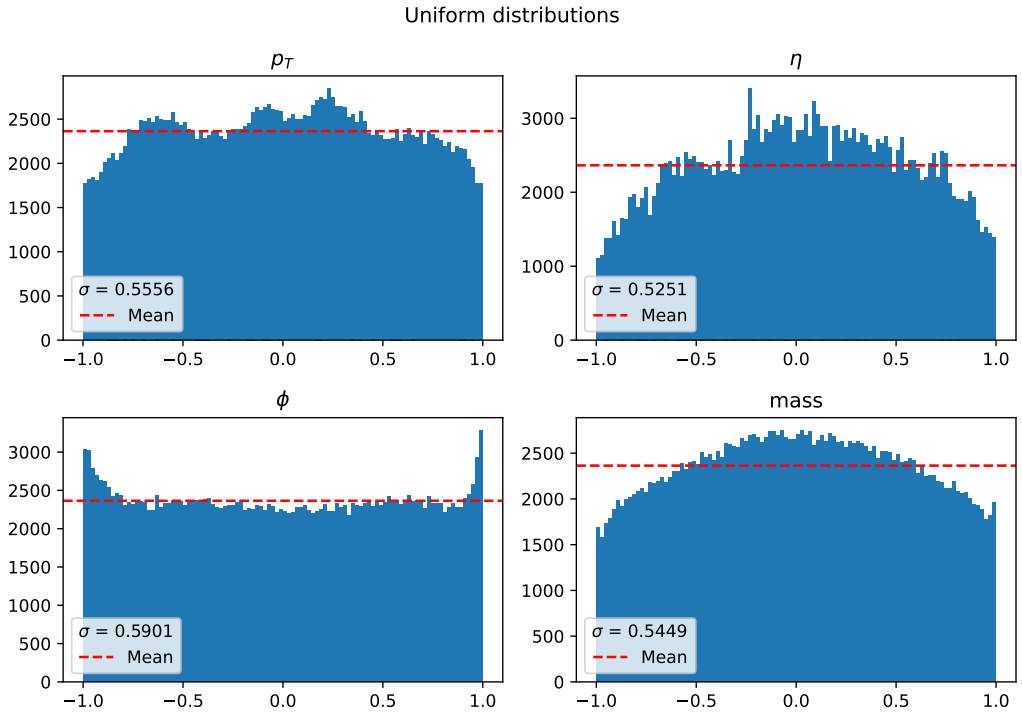

Figure 12 – Variables distributions after applying the error function for `model2` - expected uniform distribution. Mean and std are also shown: a standard deviation of $1/\sqrt{3} \approx 0.577$ is expected for a uniform distribution.

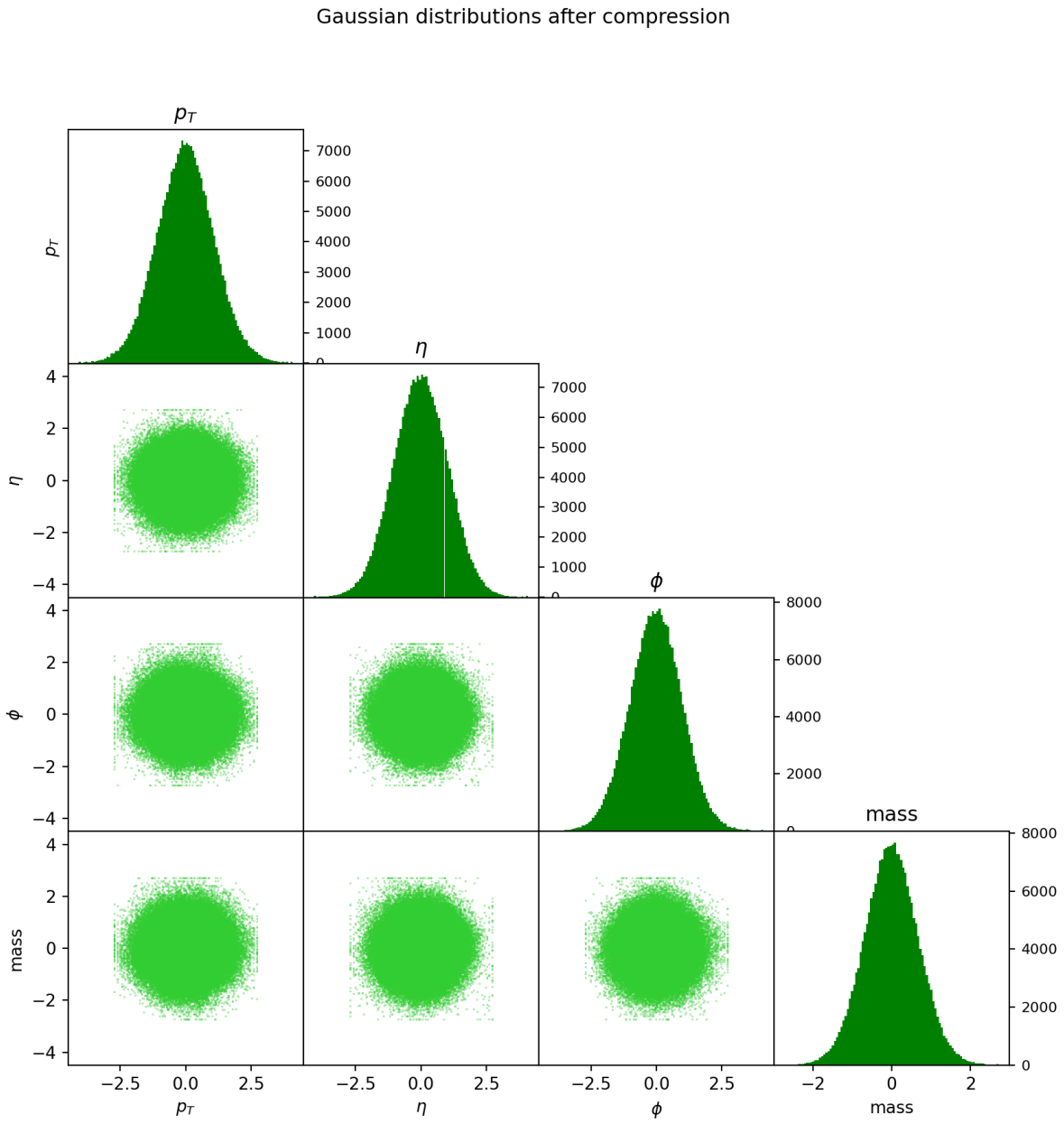

Figure 13 – Variables distributions after decompression on the diagonal and scatter plots of each pair of variables for `model1` and $N = 13$.

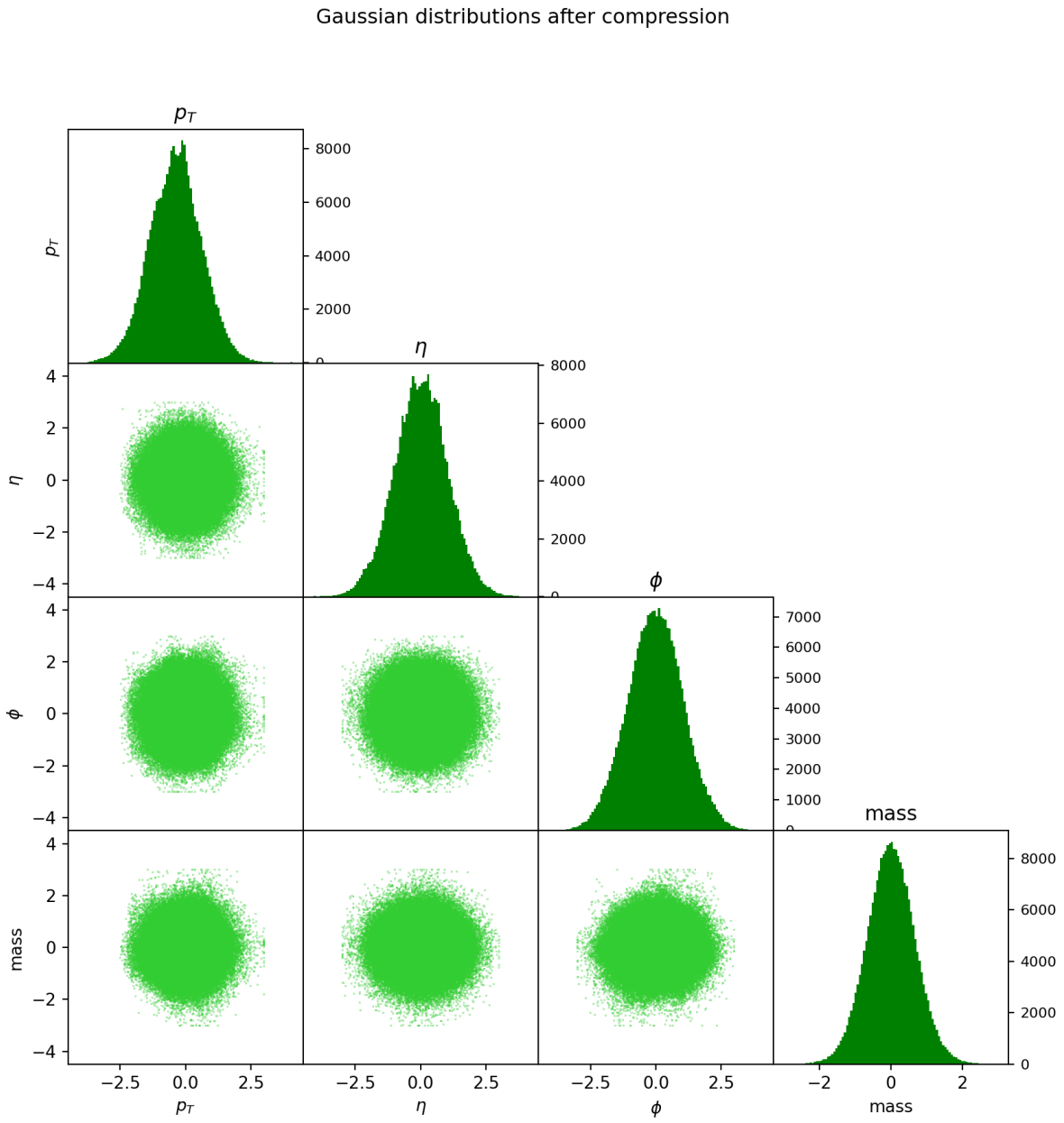

Figure 14 – Variables distributions after decompression on the diagonal and scatter plots of each pair of variables for `model2` and $N = 20$.

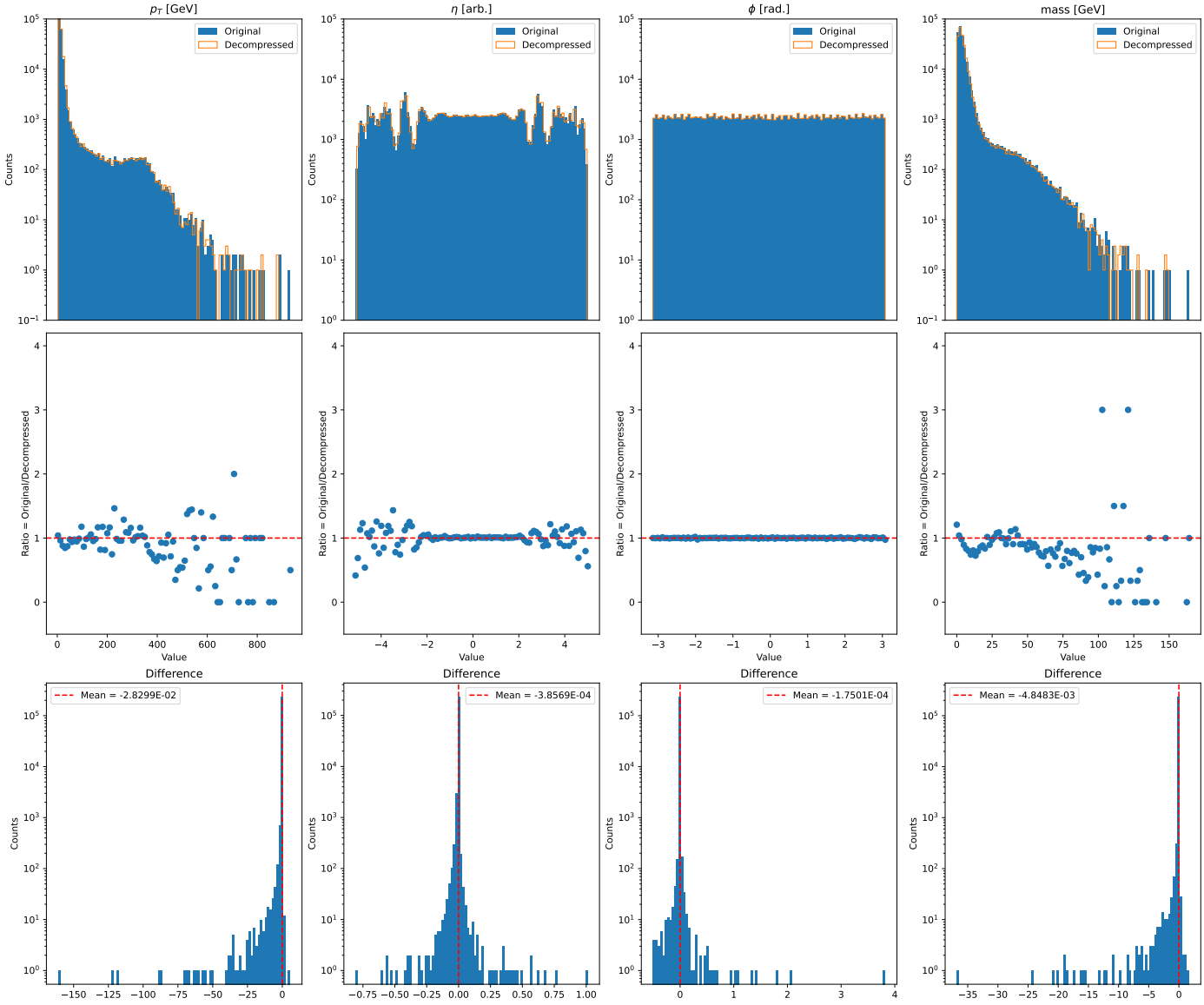

Figure 15 – Distributions, ratio and differences between original and decompressed training data for `model1` and $N = 13$.

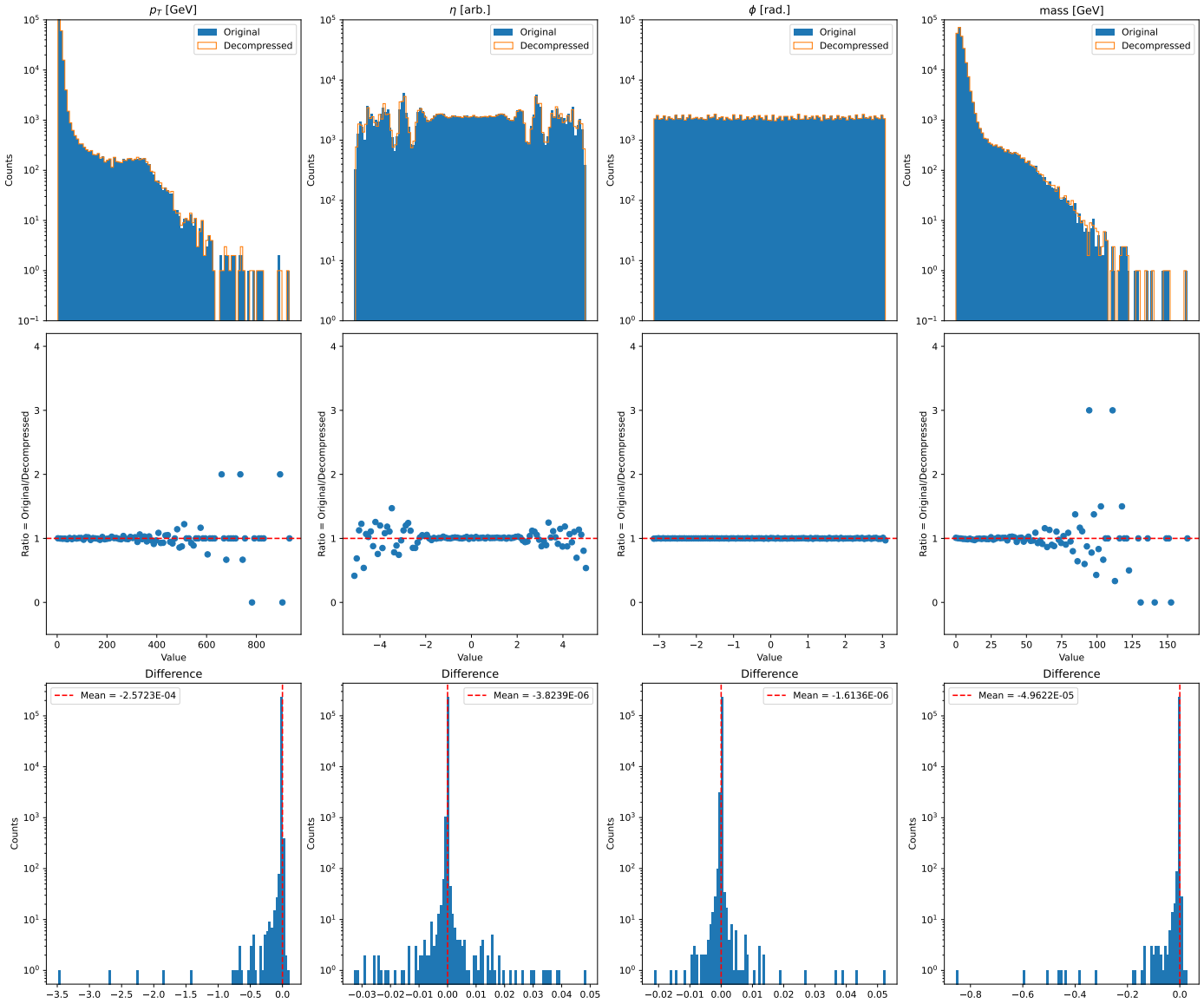

Figure 16 – Distributions, ratio and differences between original and decompressed training data for `model1` and $N = 20$.

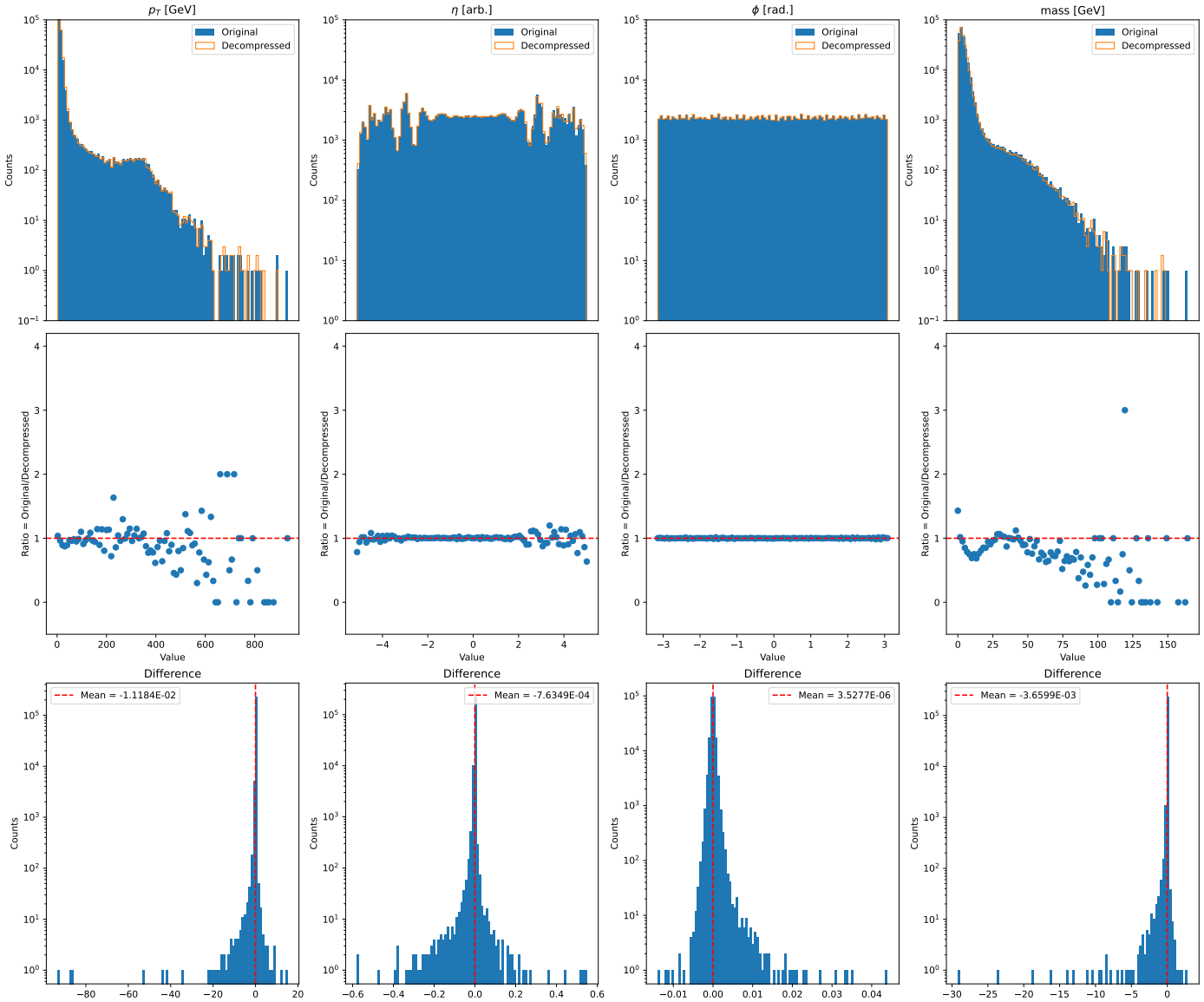

Figure 17 – Distributions, ratio and differences between original and decompressed training data for `model2` and $N = 13$.

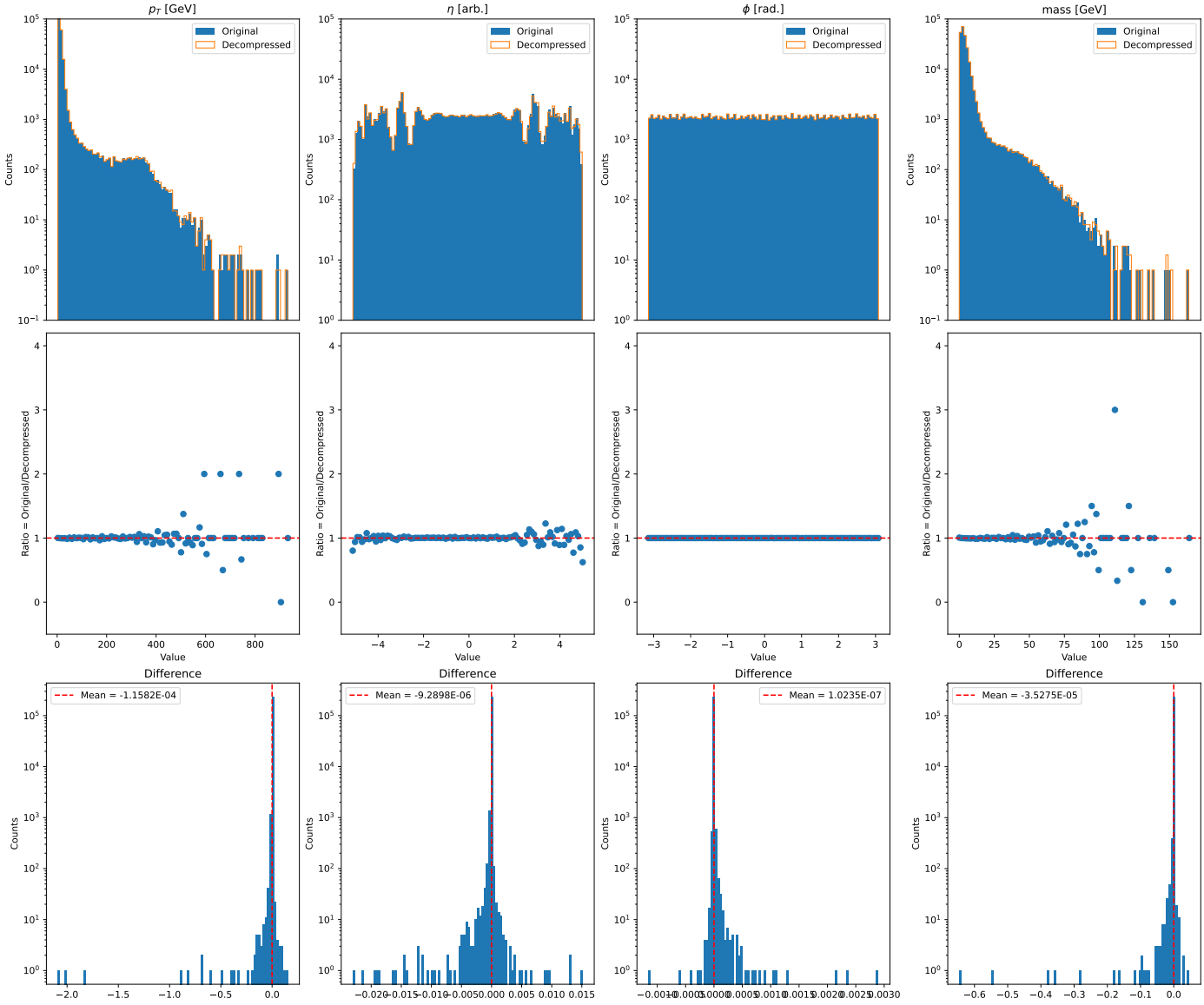

Figure 18 – Distributions, ratio and differences between original and decompressed training data for `model2` and $N = 20$.

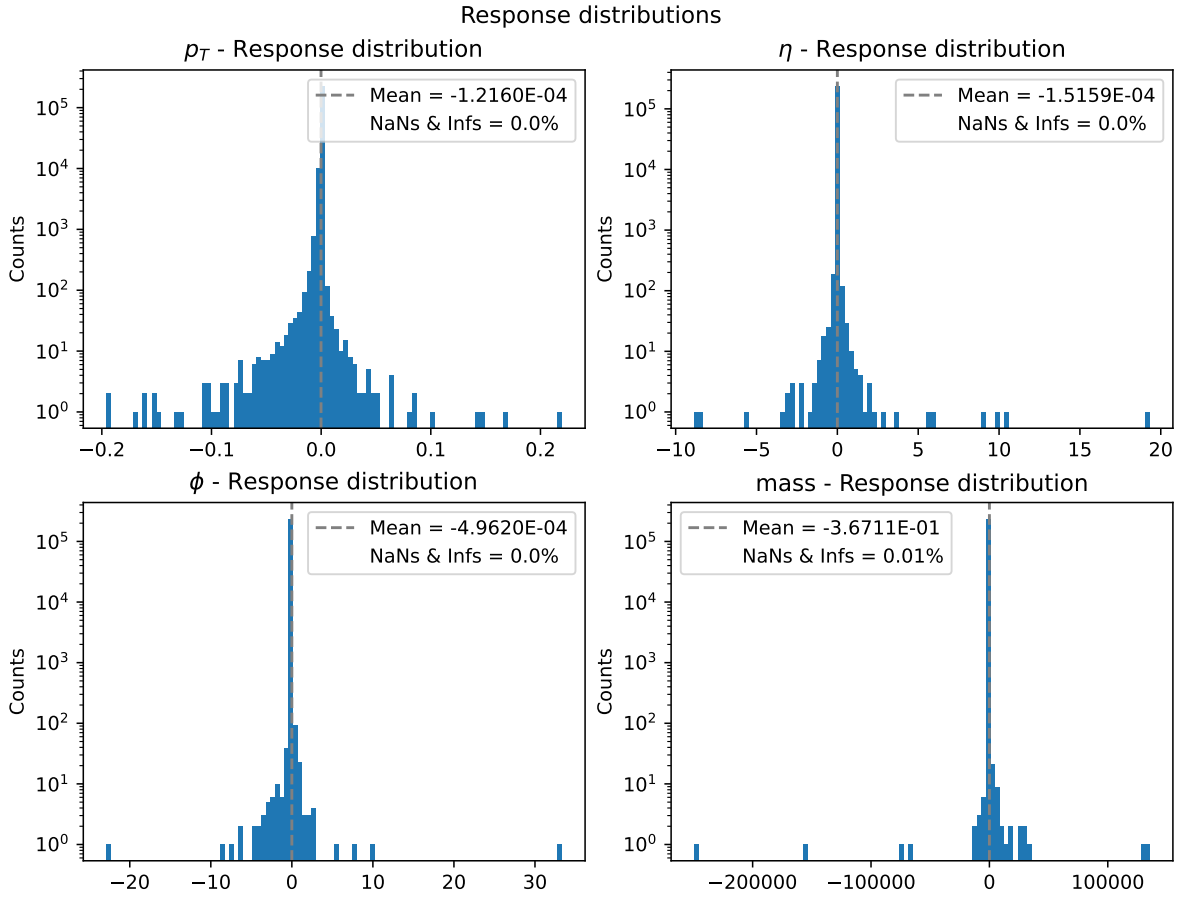

Figure 19 – Response distributions for `model1` and $N = 13$.

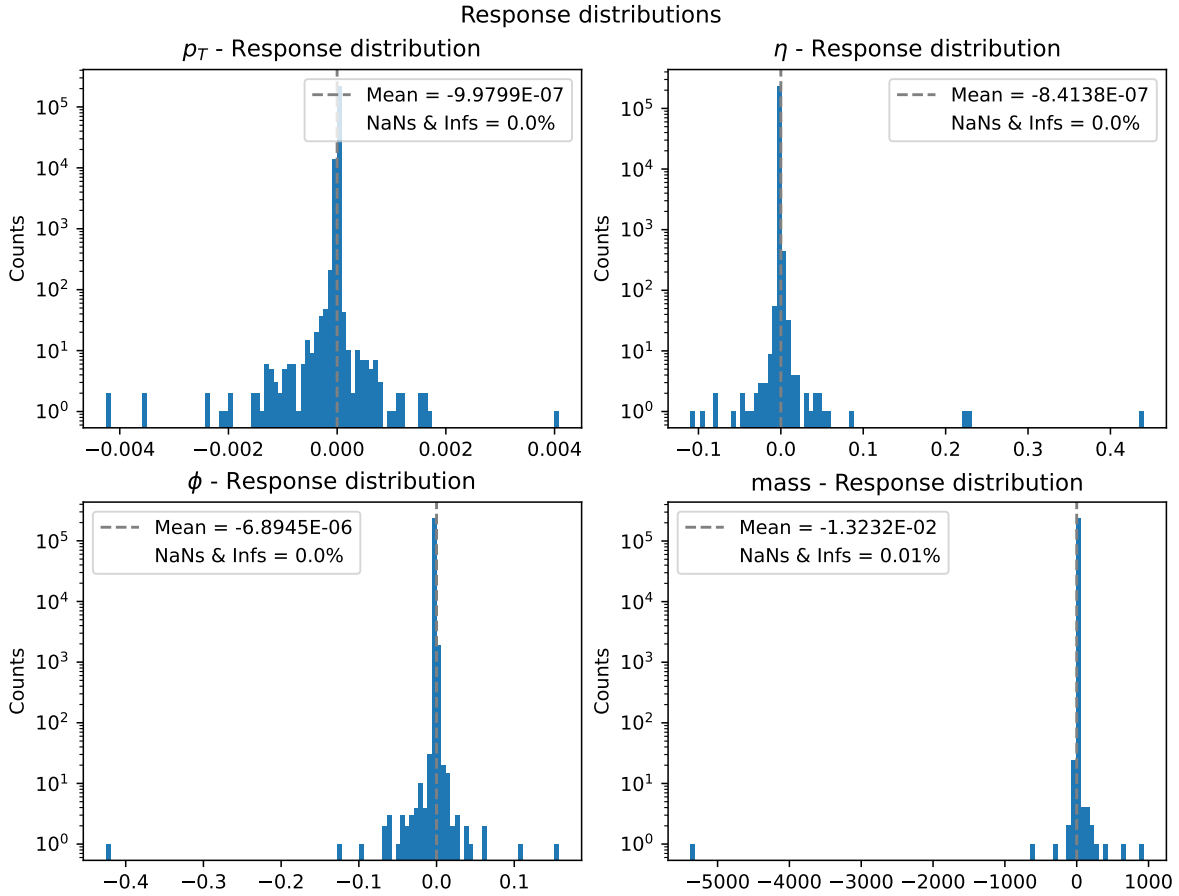

Figure 20 – Response distributions for `model1` and $N = 20$.

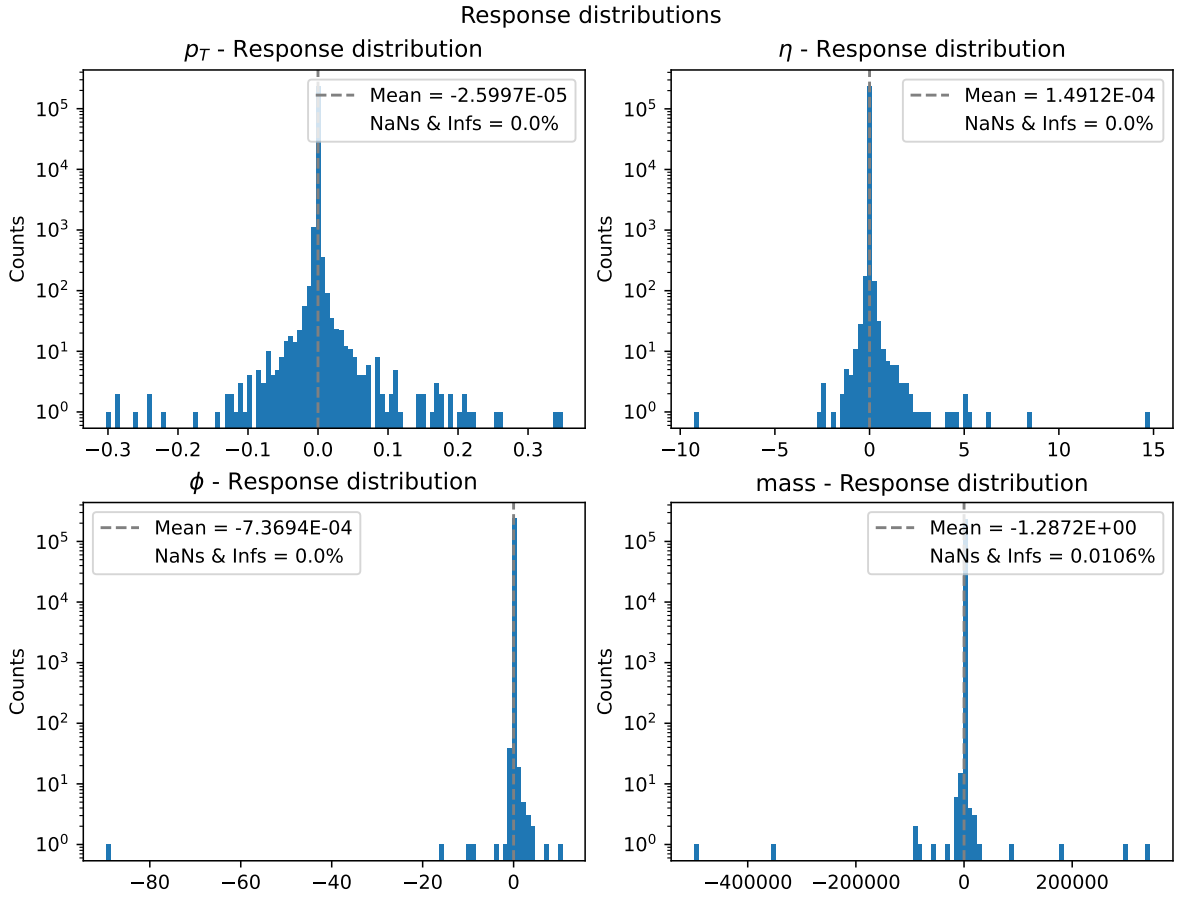

Figure 21 – Response distributions for `model2` and $N = 13$.

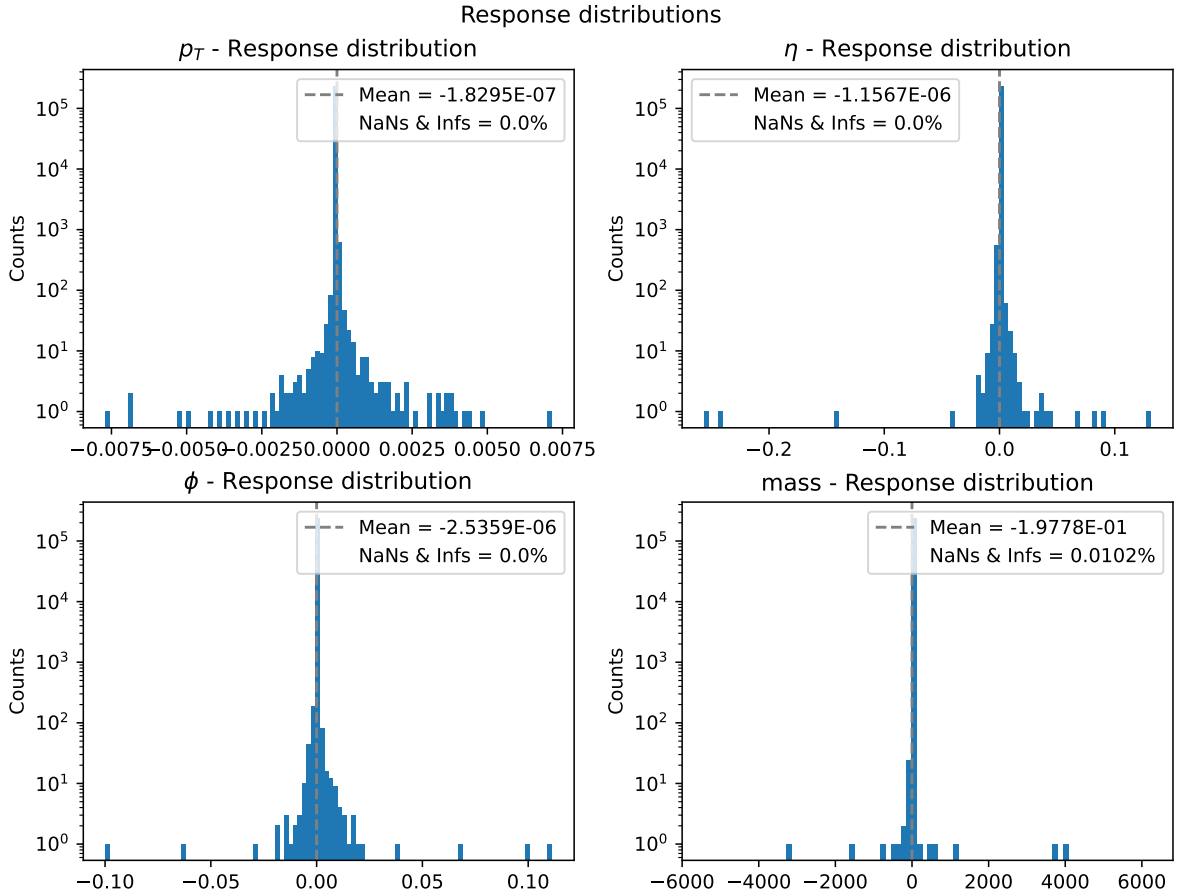

Figure 22 – Response distributions for `model2` and $N = 20$.

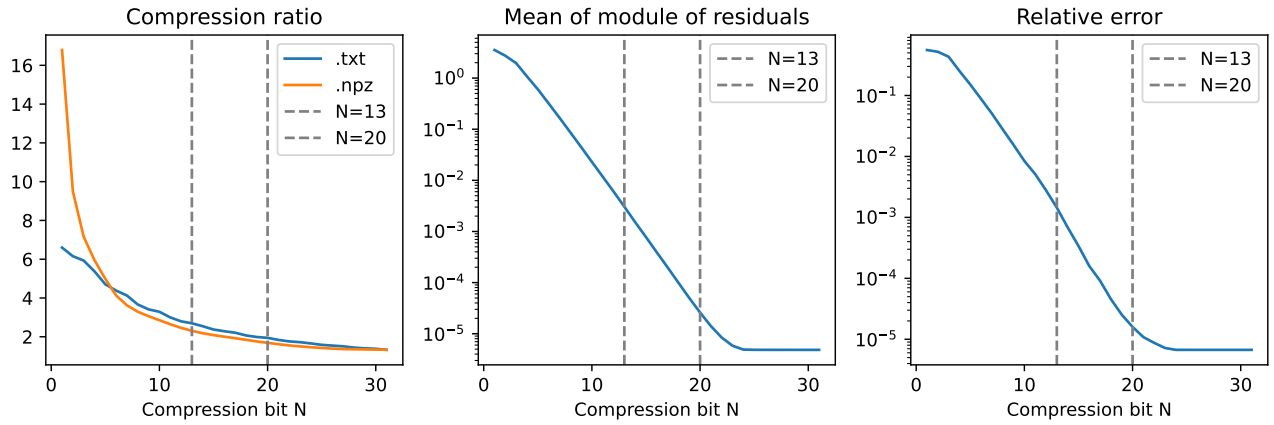

Figure 23 – Compression ratio, mean of the module of residuals and relative error as a function of N for `model1`.

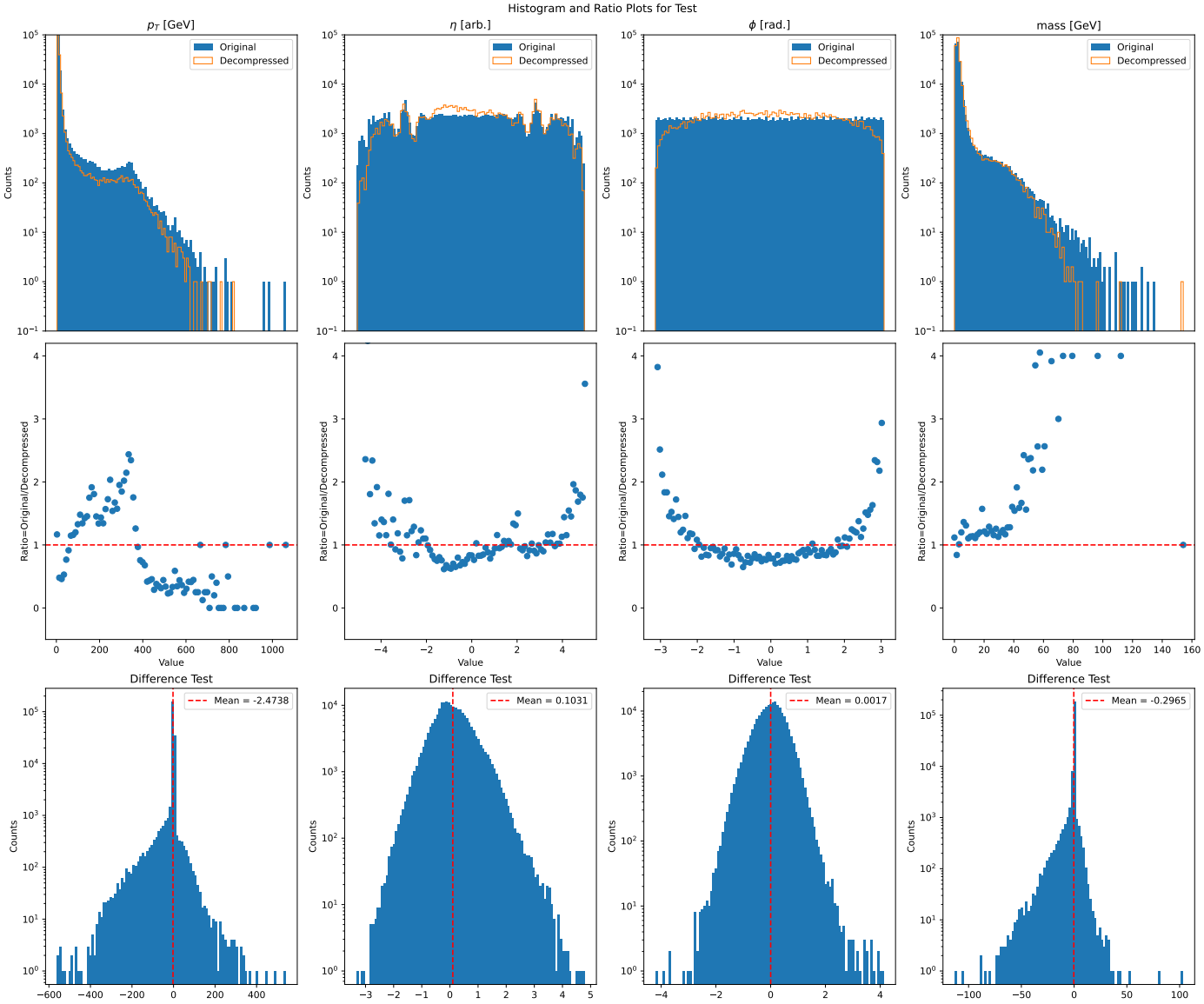

Figure 24 – Distributions, ratio and differences between original and decompressed test data for `model1` and $N = 13$.

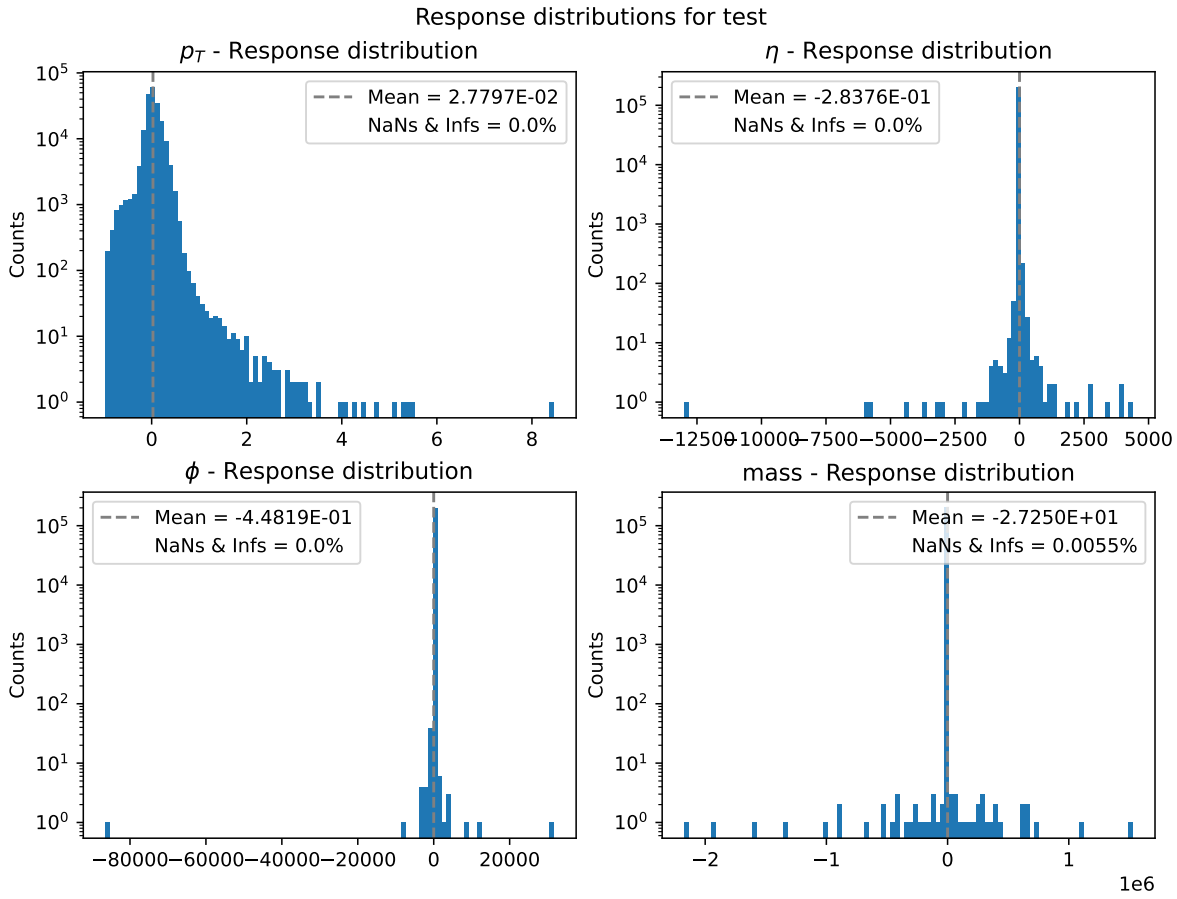

Figure 25 – Response distributions for test data, `model1` and $N = 13$.

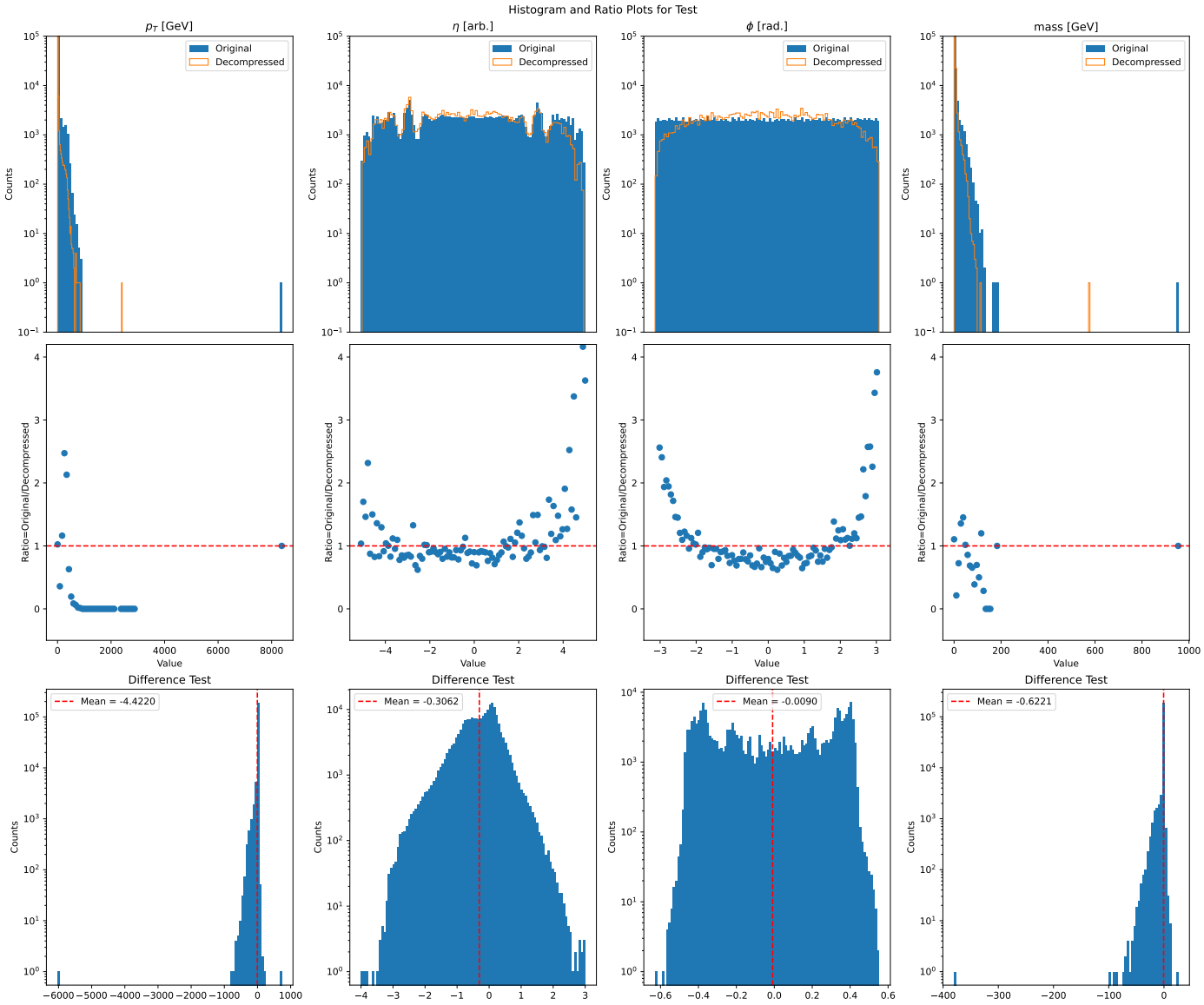

Figure 26 – Distributions, ratio and differences between original and decompressed test data for `model2` and $N = 13$.

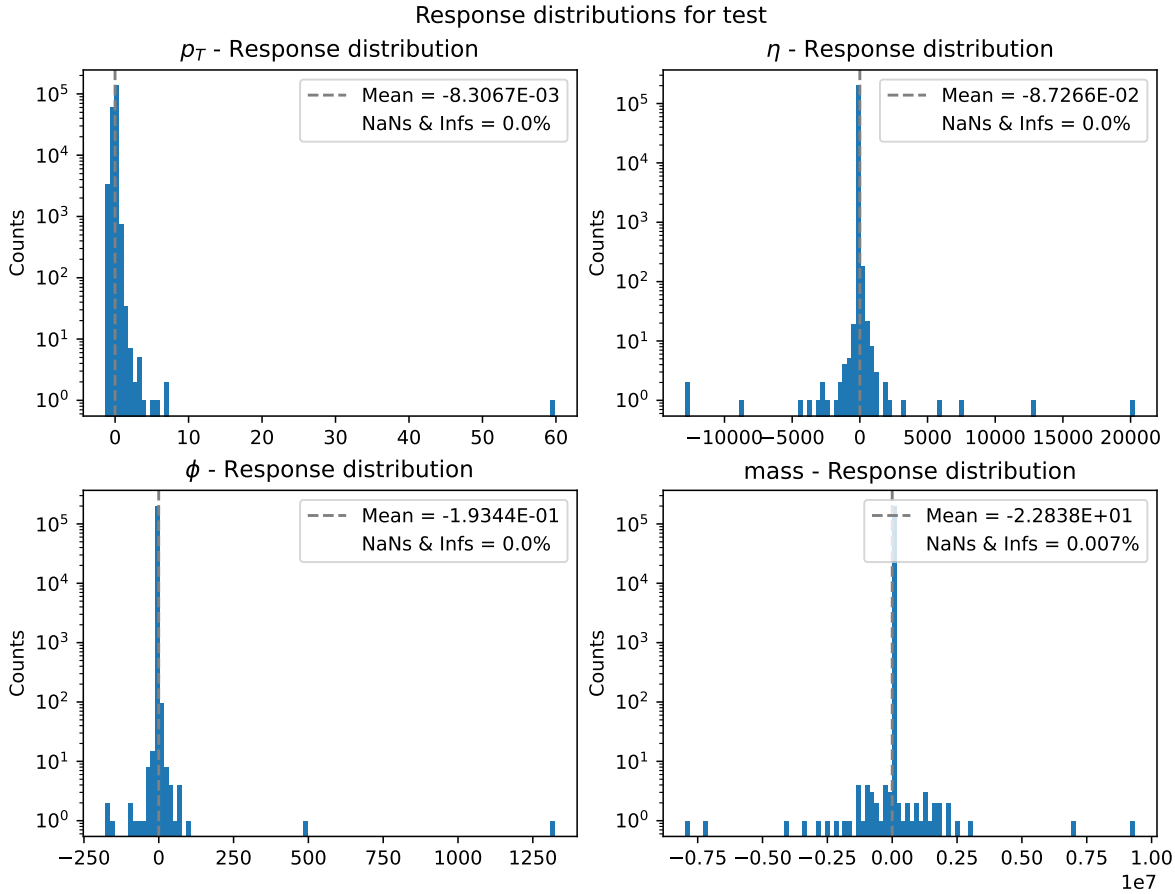

Figure 27 – Response distributions for test data, `model2` and $N = 13$.