HEP Software Foundation

Deep-compression for High Energy Physics data

Honey Gupta

Google Summer of Code 2020

Mentors: Caterina Doglioni, Baptiste Ravina, Rebeca Gonzalez Suarez, Antonio Boveia, Lukas Heinrich

Jun-Aug, 2020

Chapter 1

Introduction

This report contains the details of the Google Summer of Code 2020 project on Deep-compression for High Energy Physics (HEP) data. This project is in collaboration with Caterina Doglioni, Baptiste Ravina, Rebeca Gonzalez Suarez, Antonio Boveia and Lukas Heinrich.

Motivation.
At CERN's Large Hadron Collider (LHC), proton collisions are performed to study the fundamental particles and their interactions. To detect and record the outcome of these collisions, multiple detectors with different focus points have been built. The ATLAS detector is one such general purpose detectors at the LHC. There are approximately 1.7 billion events or collisions occurring inside the ATLAS detector each second and storage is one of the main limiting factors to the recording of information from these events. To filter out irrelevant information, the ATLAS experiment uses trigger systems which selects and sends interesting events to the data storage system while throwing away the rest. Storage of these events is limited by the amount of information to be stored and a reduction of the event size can allow for searches that were not previously possible.

To this end, this project aims to investigate the use of deep neural autoencoders to compress event-level data generated by the HEP detector. The existing preliminary work investigates deep-compression algorithms on jets, which is the most common type of particle. The work shows promising results towards using deep-compression on HEP data. We build upon the existing work and extend the compression algorithm to event-level data, which means that the data contains information for multiple particles rather than just jets particles. We experiment with two open-sourced datasets and perform ablation studies to investigate the effect of deep compression on different particles from multiple processes.

Deep-compression.
Deep compression refers to the usage of autoencoders for performing data compression. The aim is to learn the data distribution by projecting it to a lower-dimension and then reprojecting. The project's idea is to use deep compression for HEP data and check their efficacy. Therefore, the objective while learning the neural network is to maintain the data's fidelity after performing compression and decompression.

In Chapter 2, we discuss the existing work in some detail and present the results of the validation experiments we performed on the available pre-trained model. This chapter also contains some details of the work done directly on private ATLAS data. Next, we briefly describe the opens-sourced datasets used for our experiments in Chapter 3. In Chapters 4 and 5, we describe the experiments performed during the project and discuss our observations on the open-sourced PhenoML v1 and PhenoML challenge datasets, respectively.

Chapter 2

Description and validation of existing network

An autoencoder (AE) is a neural network that tries to implement an approximation of the identity function. It generally consists of an encoder, a latent space, and a decoder. The encoder encodes the information present in the input into a lower-dimensional latent space, and the decoder reconstructs the original input as best as it can. The latent space representation that has a lower dimension as compared to the input-space can be used as a compressed representation of the input and can be stored along with the decoder network to reconstruct the data.

The existing work focuses on using AEs for compression by using the latent space as a compressed representation of the HEP data comprising only of jet-particles. They use the commonly used Mean Squared Error (MSE) as the loss function for training their networks. The metrics used for evaluating the accuracy of the reconstructions after performing compression and decompression are difference $(x_{out} - x_{in})$, relative difference $(\frac{x_{out}-x_{in}}{x_{in}})$ and the mean and standard deviations of both of them. Here, x_{in} represents the input (4-D or 27-D) data and x_{out} represents the reconstructed data. The mean and standard deviation are averaged across the test samples. We investigate the performance of the existing work on two compression types on the ATLAS trigger data. The first compression type is from a 4-dimensional data and the second compression is from a 27-dimensional data.

Compression of 4D data

We analyse the compression of 4-D data to a latent space of 3 dimensions. We used a network with 7 fully-connected layers of 200, 100, 50, 3, 50, 100, 200 nodes and a *tanh* activation layer after each layer, as mentioned in the existing work.

Normalization

We experiment with three different types of normalizations for the compression of 4D data from ATLAS - (1) No normalization, (2) Standard normalization and (3) Custom normalization. Here, standard normalization refers to assuming a Gaussian distribution and centralizing and scaling the data to make the mean=0 and variance=1. Whereas, custom normalization refers to manually scaling each variable from the data so that the distribution lies between [0,1] or [-1,1].

We observed a high bias towards few variables in the non-normalised model and very poor reconstruction accuracy overall. One of the variables had a low error (MSE) for this normalization case whereas one of them had a high error. For standard normalization, we observed that the model had highest error for most of the parameters among the three normalization types. Custom normalization has better performance for most of the parameters as compared to the others. We conclude that custom normalization seems to be optimal option.

Different variants of the network

We tried 2 variants of the autoencoder models that had the same 7 layer autoencoder with similar node configuration as the previous model. The dissimilarity was in the activation layer and batchnorm layer. The first model has *tanh* activation and no batchnorm layer (the base model). The second has Leaky Rectified Linear Unit (ReLU) activation and batchnorm after each layer. The third model has Exponential Linear Unit (ELU) and batchnorm after each layer.

Our experiments indicated that LeakyReLU model has moderate performance (based on the variance of relative error and MSE on the test-set). *tanh* and ELU have comparable performance. *tanh* has lower variance and mean for the relative error but ELU has lower MSE. Hence, ELU model can be said to be the better among the two by considering both the relative error and MSE, since there is not much difference between ReLU and ELU's MSE for the test-set.

We also experimented by using a L1 loss as a loss function but found that there was not very significant difference between the performance of the model trained with MSE and the one trained with a L1 loss. Hence, we continue using MSE as the loss function for the rest of the experiments.

Performance of the network

Next we analyzed the execution time and memory allocation of the compression model while testing for 4D compression. The model configuration is the same as above.

Specifications.

- Test-set size: 27945
- Hardware for computation: Intel(R) Xeon(R) CPU E5-1620 v4 @ 3.50GHz, 8 cores

Results.

- Data loading execution time: 0.10674s
- Data loading memory: 3.668 MBs

The findings are summarized in Table 2.1. We observed that since ELU has an exponential component, it's runtime is higher. *tanh* also has an exponential component, but

Model	Initialization time (s)	Model load time (s)	Encoding time (s)	Decoding time (s)	Encoding memory alloc (MB)	Decoding memory (MB)
Tanh, no NN	2.5174	0.05932	0.03934	0.02113	0.0024	0.1780
LeakyReLU, BN	2.5903	0.08180	0.12613	0.10033	0.0076	0.2240
ELU, BN	2.4373	0.07940	0.16376	0.15237	0.0064	0.1945

Table 2.1: Performance summary

the batchnorm layer in the other two models increases the execution time. Overall, we can say that a model without the exponential component i.e. with a ReLU activation and without Batch-normalisation should perform the best in terms of execution time.

Compression of 27-D data

In order to transition to 27-D data, we analysed the available 27-D data by plotting the distribution of each variable and comparing the plots with the ones presented by the prior experiments. We tested the available pre-trained model and created plots from them. We also compared and validated the published results. Next, we trained the model on the available ATLAS 27-D data and created response and correlation plots to analyse the performance.

The compression was performed from 27-D data to a 20-dimensional latent space. This data contained kinematic information only from jet particles and is the same as the previously mentioned 4-D data, but with more variables. The model contained LeakyReLU as activation units and used batch-normalization. We used custom-normalization for scaling the training and testing data. The model configuration was [27-200-200-200-20-200-200-200-27], where each number represents the nodes at each layer of the autoencoder.

The compared the data distribution plots and the response plots, which is a plot of the relative error for different variables. They showed that the our data and results are very similar to the published results. After training the existing network on ATLAS data, our 1D response plots were similar to the published results, which showed that the relative errors for most of the variables are zero centered and have low variance. This depicts that the compression model performs fairly well for these variables. We also observed that errors for different variables have considerable correlation among themselves. This gave a glimpse of the ability of the network to compress different variables.

Chapter 3

Datasets

PhenoML datasets

We used two open-sourced datasets for expanding the existing method to compress data at an event-level. These datasets are

1. PhenoML v1 dataset (Link)
2. PhenoML challenge dataset (Link)

The first dataset, PhenoML v1 dataset contains a set of simulated LHC events, corresponding to 10 [1]fb^{-1} of data. The events from the dataset can be used as a benchmark for comparison of different detection algorithms. The dataset contains different processes from both Standard Model (SM) and beyond the SM (BSM) models.

The data is provided as in CSV files, the format of which is as follows:

```
event ID; process ID; event weight; MET; METphi; obj1, E1, pt1, eta1,
                phi1; obj2, E2, pt2, eta2, phi2; ...
```

Each process is identified by a process ID that is mentioned as the name of the CSV file. The data/file for each process contains data in a one-line-per-event text format, where each line has variable lengths. The object identifiers `(obj1,obj2, ...)` are strings identifying each object in the event. The particle to keyword mapping can be found in Table 3.1. The data distribution plots generated by converting the events into 4-D data from one of the processes from the SM model: `atop_10fb` is shown in Fig. 3.1. The data in the plots are custom normalized and represent a sample train/test dataset.

For our experiments, we look at jet events first for training and testing the model. Since, the existing works validate deep compression for HEP data on only jets, we started with jets and moved on to other particles. Moreover, the kinematic properties of particles other than jets and b-jets, as mentioned in Table 3.1, is similar to the jets data distribution based on the physics processes involved. Hence, we focus on training the model on jets and then testing on different flavours of the dataset containing different kinds of particles.

[1]fb^{-1} corresponds to sample size of thousands to millions, depending on the type of event considered. It should be enough to provide a statistically meaningful training sample

Keyword	**Particle**
`j`	jet
`b`	b-jet
`e-`	electron (e^-)
`e+`	positron (e^+)
`m-`	muon (μ^-)
`m+`	antimuon (μ^+)
`g`	photon (γ)

Table 3.1: Definition of the keywords used in the PhenoML dataset for different particles.

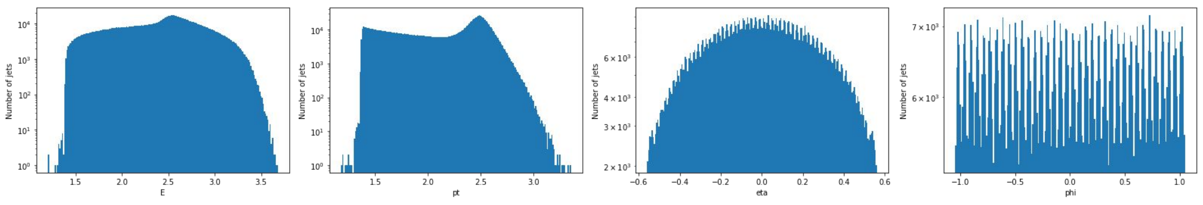

Figure 3.1: Data distribution of the 4 variables from the process - atop_10fb data(SM) from the PhenoML v1 dataset, after custom normalization.

We also tried the other way around by training on only 'other' particles and testing on jets. For this experiment, we used the PhenoML challenge dataset, which contains data having all processes as mixed. Also, the fraction of 'other' particles as compared to jets is comparatively higher in this dataset as compared to the PhenoML v1 dataset. More details of the experiments and results are mentioned in the following sections.

Chapter 4

Tests of the network on PhenoML dataset v1

We trained a deep autoencoder on the `njets_10fb` dataset which majorly contains jet particles. The model compresses an input 4D data into 3D latent space and then again back to 4D with the help of encoders and decoders. The 4D data that we used for training contained event-level data converted to 4D by considering all particles as the same.

Model features:

- `Activation = LeakyReLU`
- `Batch-normalization = True`
- `Data normalization type = Custom`
- `Layer configuration = [4, 400, 400, 200, 3, 200, 400, 400, 4]`

As mentioned earlier, we use the response plots to analuse the properties of the reconstructed data. The response plots are the plots of the relative difference or the residuals between the input and output data. The plots for the jets-trained model tested on a dataset derived from the same `njets_10fb` process are shown in Fig. 4.1.

We then tested the jets-trained model on data from two other processes: `atop_10fb` and `ttbar_10fb`. The aim was to analyse the ffect of compression on different particles when the model is trained on one type of particle - jets. The response/residual plots for `atop_10fb` are shown in Fig. 4.2. We observed that the response plots generated from the compression were very similar to the plots in Fig. 4.1, which indicated that the jets-trained model performs fairly well for other particles as well.

We next performed a more deeper analysis to check the the affect of compression on data at an event level. We tested the jets-trained model on three different version of the data from `atop_10fb` and `ttbar_10fb`.

Different versions of the test dataset:

- Dataset created by considering all particles from all the events in the process
- Dataset created by considering all the particles but from those events that contained only jets

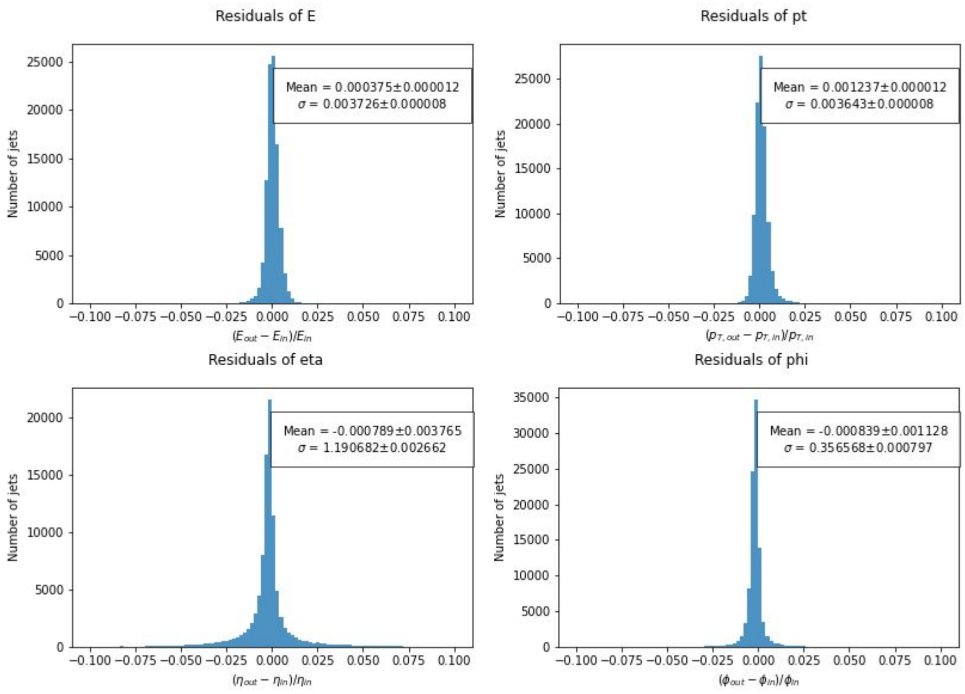

Figure 4.1: Response plots for the results from the jets-trained model when tested on jets.

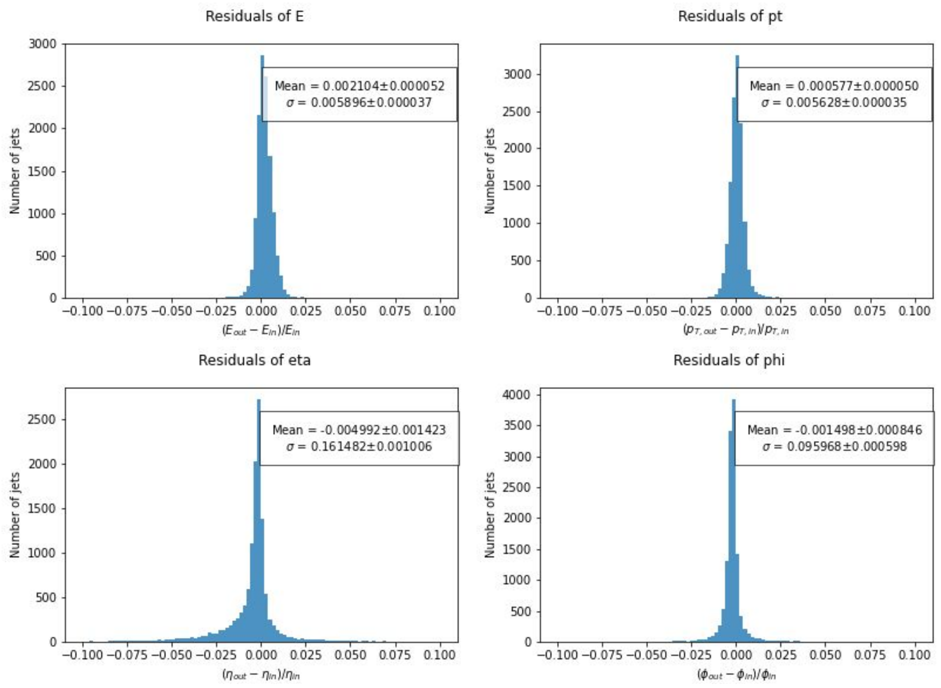

Figure 4.2: Response plots for the results from the jets-trained model when tested on `atop_10fb`.

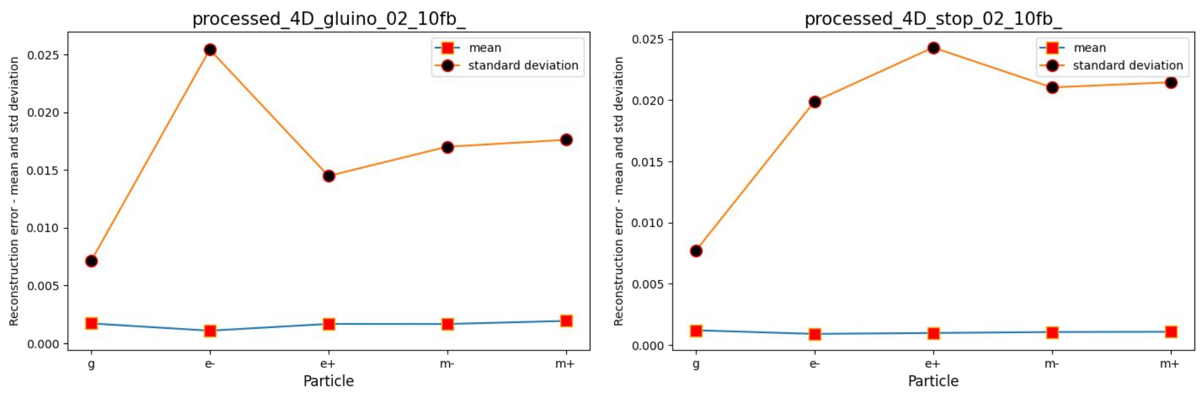

Figure 4.3: Mean and standard deviation of the relative difference error of p_t for various particles from the `gluino_02` and `gluino_02` BSM processes.

- Dataset created by considering all events but taking 4D data from only jet particles

For all the above three flavours of event-level data, we again observed that the response plots had minor variations from the ones in Fig.4.1. The variations were smaller in the case of E and p_t variables and more in the case of η and ϕ.

We next analysed the compression performance of the jet-trained model on test-sets that contained only 'other' particles, individually. For this, we considered two BSM processes - `stop_02_p_p_to_t1_t10̃_5.69774999996_48` and `gluino_02_p_p_to_go_go_0_0.0508105_30`. To create the dataset, we extracted the 4D information corresponding to each 'other' particle separately. We then tested these particles independently on the jet-trained model. The mean and standard deviation of the response for p_t of different particles is shown in the Fig. 4.3 below. The stacked response plots for the `stop_02` process is shown in Fig. 4.4. The response plots for this experiment have the means close to 0 and have low-variance, which shows that the compression on individual particles using a jet-trained model also works considerably well.

Our overall conclusion from the experiments is that using a jet-trained network to perform compression should be sufficient and training on other particles might not be necessary. To validate this further, we performed the experiments with the PhenoML challenge dataset, as mentioned in the next chapter.

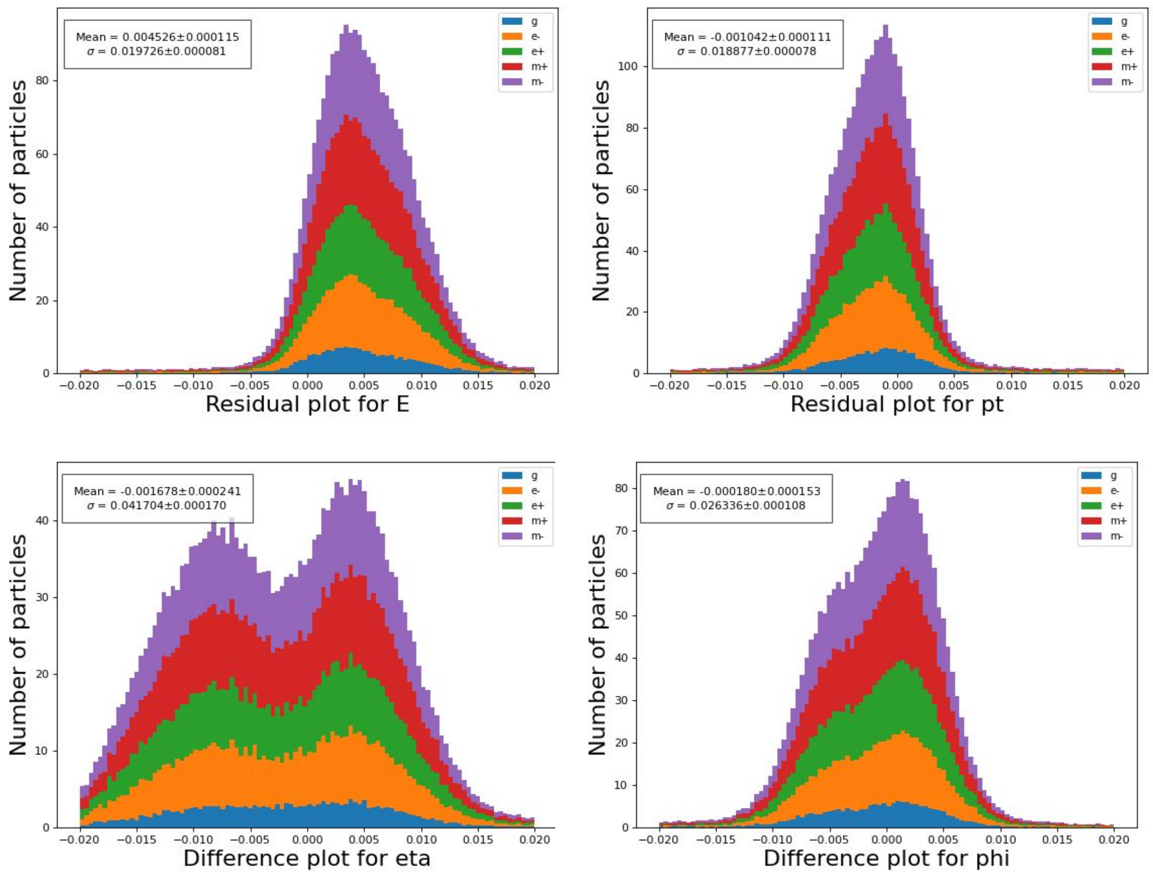

Figure 4.4: The response plots for various particles from the `gluino_02` BSM process.

Chapter 5

Tests of the network on PhenoML challenge dataset

To check the compression performance when a network is trained on ‘other’ particles, we used the PhenoML challenge dataset that contains data from mixed processes. We use the data from `chan2a` that contains mixed particles to train the model and use `chan3` data that contains mostly jets to test the model. Fig. 5.1 shows the loss *vs* epoch plots for networks trained with jets data (`njets_10fb`) and the one trained with other particles from `chan2a`. We can see that the model trained on jets converges better than the one trained on other particles. This could be because the mixed processes data has less correlated particles. Thus, the kinematic properties are also less correlated as compared to using just jets having similar kinematics. This dissimilarity can lead to poor convergence for the network while training.

Fig. 5.2 shows the response plots for the 4D-3D compression when data from `chan3` containing mostly jets is tested on a model trained on data from `chan2a`. We can observe that the response plots have higher variance as compared to Figs. 4.1 and 4.4, which shows that training a model on jets gives better performance as compared to training on mixed particles.

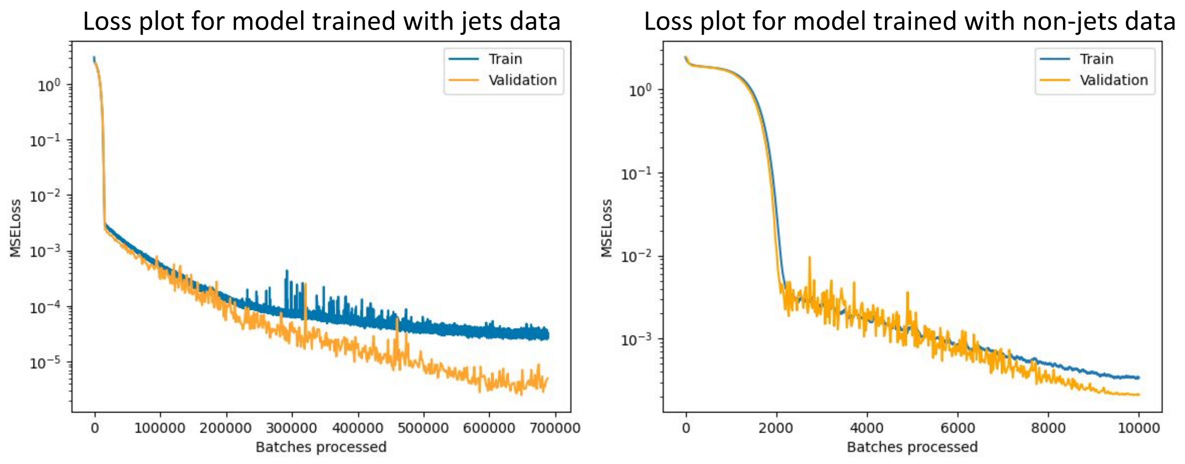

Figure 5.1: Loss plots for networks trained on jets and non-jets (other particles) data.

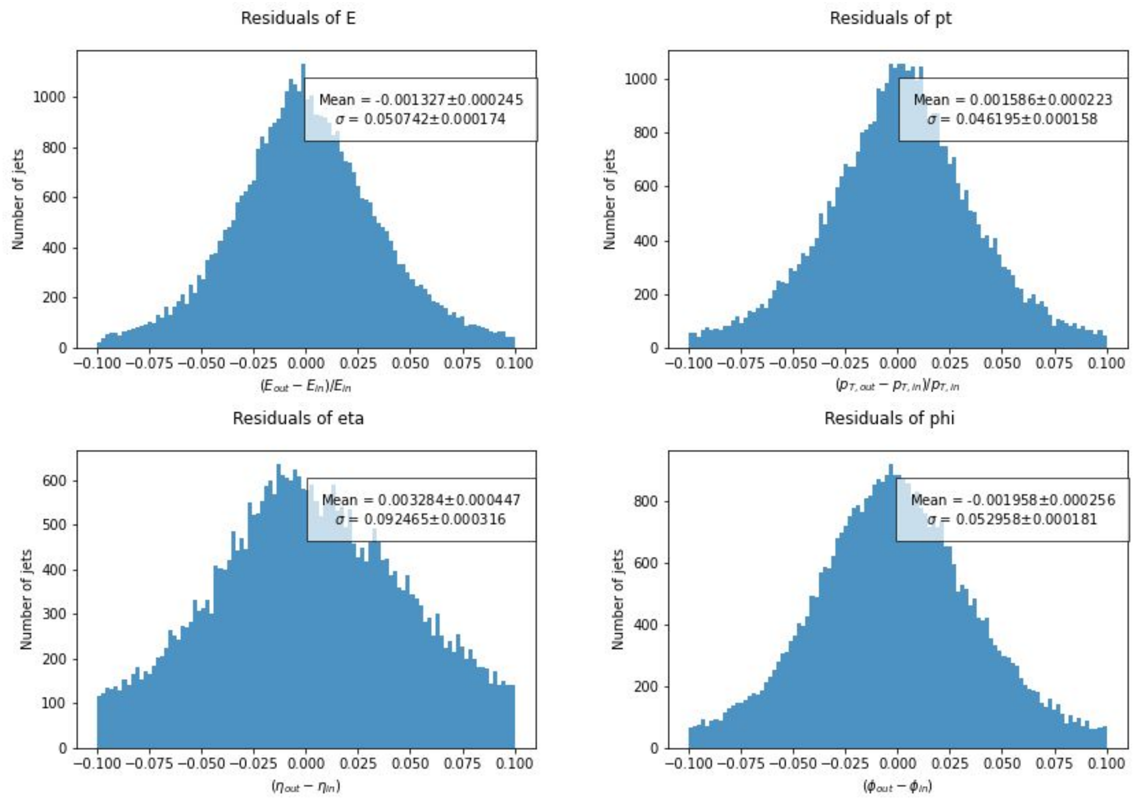

Figure 5.2: The response plots for all particles when jets-data is tested on a model trained on other particles.

Chapter 6

Code and documentation

Project Artifacts:

- Code Release:
  `https://github.com/Autoencoders-compression-anomaly/AE-Compression-pytorch/releases/tag/GSoC20`
- Slides containing the experiments summary, generated plots, observations and conclusions: `https://bit.ly/3jrPTB4`

The code for performing the experiments, analysis and generating the plots mentioned in this report can be found in the GitHub repository at `https://github.com/Autoencoders-compression-anomaly/AE-Compression-pytorch`, release tag - GSoC20. The repository includes the documentation for performing the experiments and analysing the models. The scripts are written in `Python` language and we use `fastai` and `PyTorch` frameworks for implementing and executing the neural networks.