Data Compression using Deep Autoencoders

Vedant Shah

Aim

- To compress the 4 momentum of a sample of simulated particles (only jets) into 3 variables using Deep Autoencoders

Materials

Dataset

- The dataset contains about 6399 events, each of which can consist of variable number of particles. This data is present in the *monojet_Zp2000.0_DM_50.0_chan3.csv* file.
- A particle can be either a jet ('j'), a b-jet ('b') or a photon ('g'). We are required to isolate the details of only the jets and compress their 4-momentum to 3-variables.
- The 4-momentum of a particle is characterized by the following features:
  - E (in GeV) *
  - Transverse momentum (pT) (in GeV)
  - Eta (in rad) *
  - Phi (in rad) *

* - Couldn't find the details

Final Dataset

- The data for all the ‘j’ particles from all the events was isolated from the rest of the data using the script *make_dataset.py* and stored in the file *final_data.csv*.
- Number of datapoints in the final dataset : 22661 (= number of jet particles encountered throughout the events)
- Consists of 4 columns corresponding to E, pT, eta and phi for each particle

Software Stack

- python 3.7.10
- pytorch 1.8.0+cu101
- fastai 1.0.61

Compute Details* (Training done on Google Colab)

- Memory: 13 GB
- CPU: 2vCPU @ 2.2GHz
- GPU: 1xTesla K80, 2496 CUDA Cores, 12GB RAM
- CUDA: 11.2

* - GPU used for all the experiments

Method

Data Pre-processing

- The following basic transforms(referred to as base transforms in the results) were applied to the data for all the experiments. Different features of the dataset have very different scales. These transforms help in rectifying that
  - E → log(E)
  - pT → log(pT)
  - η → $\eta/5$
  - Φ → $\Phi/5$

- Apart from this experiments were performed using different techniques of scaling, normalization and standardization on the data. These are standard techniques help in bringing all the features to comparable scales, which further help the machine learning algorithms learn better and more efficiently. The following combinations were investigated
  - Scaling the data to [0, 1]
  - Standardizing the data
  - Scaling the data between [0, 1] and then standardizing
  - Standardizing the data and the scaling it to [0, 1]

Train – Test Splits

- The following train-test splits were tested for performance:
  - 0.8 train, 0.2 test
  - 0.9 train, 0.1 test
- Data was sampled randomly for each of these splits

Network Architecture

- All the different Autoencoders that were used for experimentations consisted of a basic 9 linear layer structure (including the input and the output layers, excluding activation and batch norm layers) : 4 layers of the encoder and 4 layers of the decoder with a bottleneck of 3
- Different types of activation functions, which help introduce non-linearity to the model, were tested with these activation layers added after every linear layer
  - Tanh() – Default pytorch layer parameters: Mean of tanh is close to zero which has a normalizing effect on the output of the layers
  - LeakyReLU() – *negative_slope* = 0.1 : LeakyReLU is computationally less expensive and also doesn't suffer with vanishing gradients unlike ReLU
  - ELU() – Default pytorch parameters : Similar to ReLU but converges the performance faster
- Different combinations of sizes of layers were tested:
  - 4(input), 200, 200, 20, 3, 20, 200, 200, 4(output)
  - 4(input), 400, 400, 200, 3, 200, 400, 400, 4(output)
  - 4 (input), 200, 100, 50, 3, 50, 100, 200, 4(output)
- Experiments were also performed with BatchNorm1D layers added after every linear-layer + activation layer combination. BatchNormalization is added

Training Details

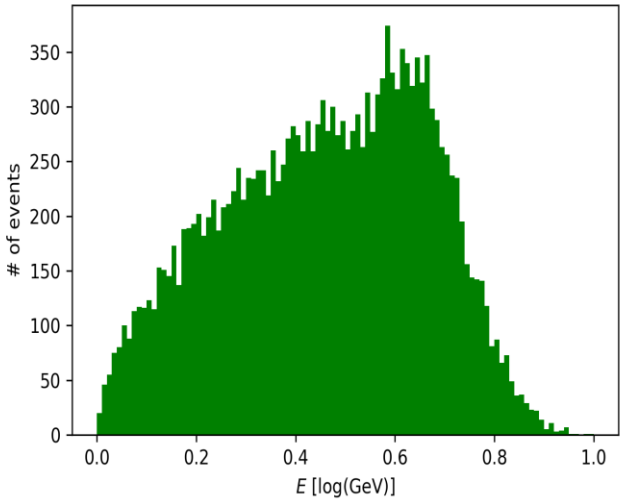
- All the experiments were performed on Colab GPUs
- The different models used along with different pre-processing techniques were trained on different number of epochs. Models were trained for 100, 200, 250, 300 epochs
- The experiments were performed on different batch sizes of 256 and 512 (standard batch sizes)
- Mean Squared Error loss was used as the loss function throughout all the experiments. We use MSE because it reflects the physical quantity that we want to minimize [(input-output)/input]

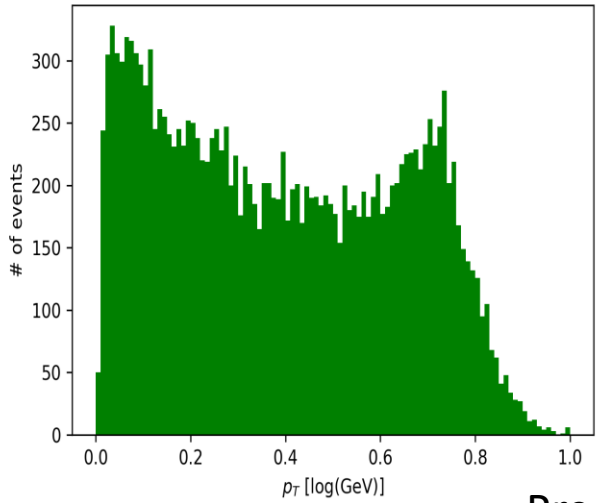
- The *fit_one_cycle* method of *fastai* learner is used for the training. The *lr_max* is set to *lr_min/10* where *lr_min* is the learning rate with the minimum loss. *lr_max = lr_steep* was also tested but didn't give satisfactory results
- We also use the *ReduceLROnPlateau* callback for some experiments to reduce the learning rate when the performance starts plateauing. This helps in avoiding overshooting and missing the minimum of the cost function and also helps in preventing overfitting
- All of the permutations of the above mentioned permutations weren't tested. If a particular parameter or a particular value of a parameter didn't give better performance or performance comparable to other configurations, it wasn't considered in further experiments. For example max_min_scaling seemed to give better performance than other other combinations, hence other combinations weren't tested after a particular point.

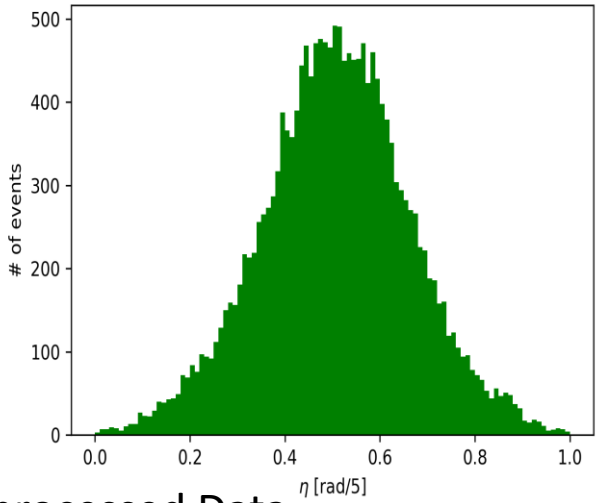
Results

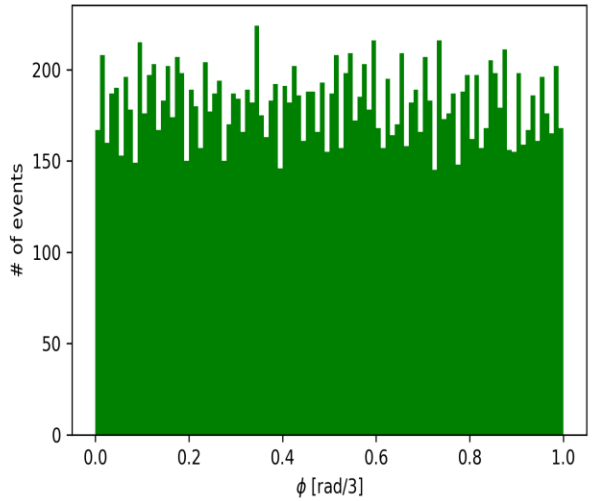
- 26 experiments were performed, results of all of which have been meticulously documented here:
  ATLAS Data Compression Experiments
- Evaluation metric : Validation score obtained using *fastai.learner.learner.validate()*
- The best performance (i.e. the minimum validation score) was obtained with the following configuration:
  - Scaling the data to [0, 1] (using sklearn's MinMaxScaler())
  - Train-Test Split of 0.8:0.1
  - Model with linear layer sizes : [4, 400, 400, 200, 3, 200, 400, 400, 4]
  - LeakyReLU activation layers with the *negative_slope* set to 0.1 after each linear layer
  - No Batch Normalization used
  - Trained for 256 epochs
- The best validation score obtained was: *7.8405e-07*

Plots (For the best performance):

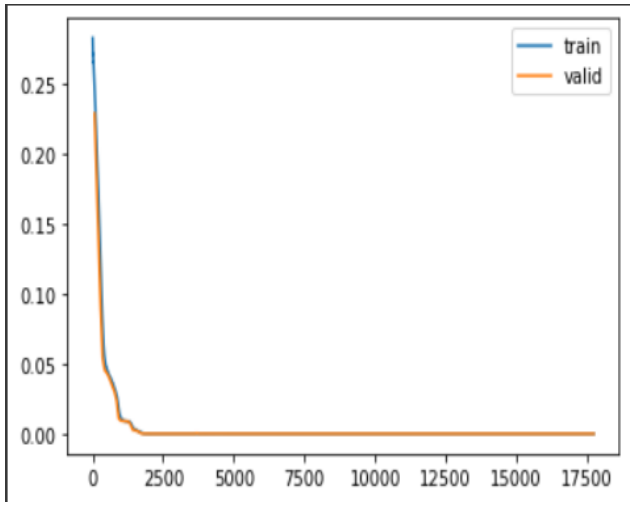




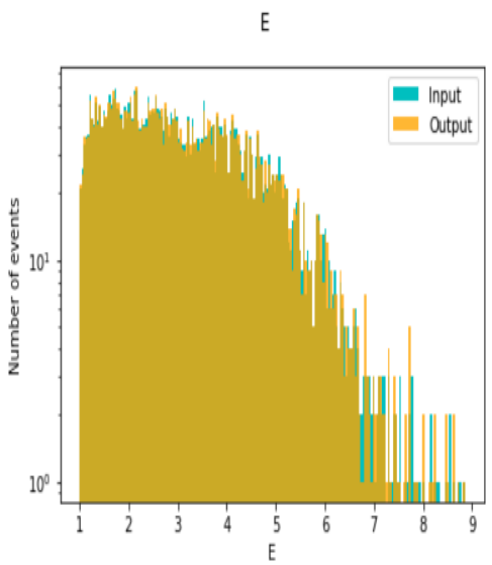


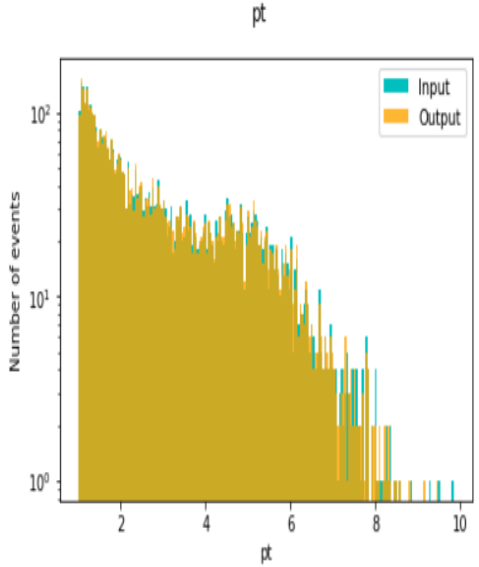


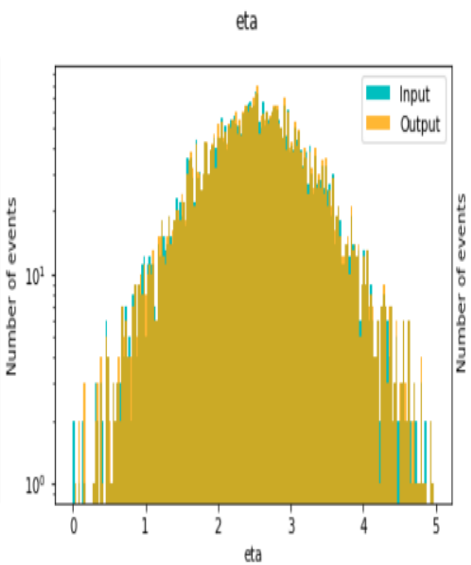
Pre-processed Data

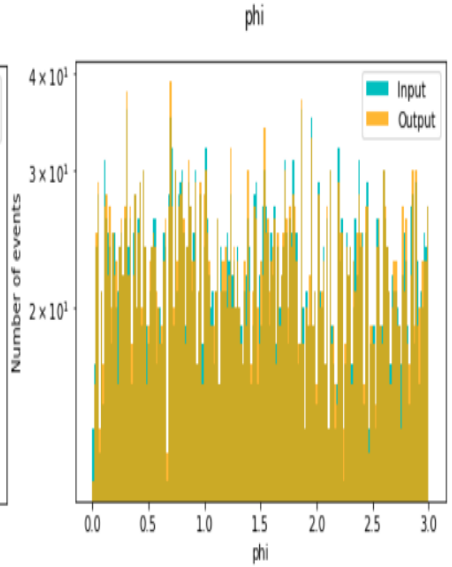


Training Loss Curve









Reconstructed Data on Pre-processed Data for

Discussion

- From the plots and the validation score, the model reconstructs the 4-momentum data satisfactorily which implies that the encoder part of the autoencoder can be used for effectively compressing the 4-momentum data into 3 variables. Although the compression is lossy, the performance seems to be acceptable
- In the reconstructed data, the 'E' attribute is reconstructed the most closely to the original data, whereas the 'phi' attribute is reconstructed the least closely
- Further work can include looking into other variants of autoencoders such as Variational Autoencoder (VAEs) and β – Variational Autoencoders (β-VAEs) for the same task.

References

[1] Wulff, Eric. (2020) *Deep Autoencoders for Data Compression in High Energy Physics,* Lund University, Sweden

[2] Wallin, Erik. (2020) *Tests of Autoencoder Compression of Trigger Jets in the ATLAS experiment*, Lund University, Sweden

[3] How to best use fit_one_cycle in practice? - Part 1 (2019) - Deep Learning Course Forums (fast.ai)

[4] Deep Compression for High Energy Physics Data, Google Summer of Code'20, Honey Gupta

THANK YOU