Profiling and Energy Estimation of ML-based compression algorithm (Baler) using HEP data

Leonid Didukh 09.10.2023
Mentor: Caterina Doglioni

Outline:

1. Motivation for the profiling and improving energy consumption of AI (green AI)
2. Results of profiling on training
3. Energy Meter report:
    a. Zeus-ML
    b. CodeCarbon
    c. Eco2AI
4. How to speed up the training and reduce the energy cost?

Why to do computational and energy profiling:

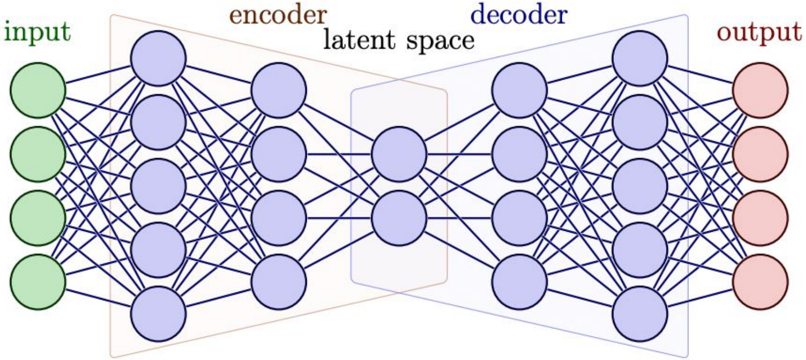
- With the growing size of DNN architecture and data the number of the operation is increasing as well therefore the training and inference consumes more electricity.
- Profiling can speed up the software execution
- It can also help us:
  - Reduce the cost of execution
  - Reduce the CO(2) emission

Dataset and Model

- Baler - Machine Learning Based Compression of Scientific Data
- It utilize the autoencoder architecture in order to provide the compressed data.
- Currently there are several benchmarks for the HEP and CFD.
- It provides the interface for the compression and decompression of data

Dataset and Model

Baler -- Machine Learning Based Compression of Scientific Data https://arxiv.org/abs/2305.02283



Metric	Value
Flops	31,283,939 FLOPs or approx. 0.03 GFLOPs
MAC	31.283M
Parameters	2457800
Operation of $Encoder_1$	2457800
Operation of $Encoder_2$	10240100
Operation of $Encoder_3$	2560050
Operation of $Encoder_4$	384015
Operation of $Decoder_1$	384050
Operation of $Decoder_2$	2560100
Operation of $Decoder_3$	10240200
Operation of $Decoder_4$	2457624

Table 1: Number of the operations and parameters in AE model

Dataset	Shape	Size
Small dataset	(520000, 24)	99847032
Big Dataset	(7689853, 24)	1476458808

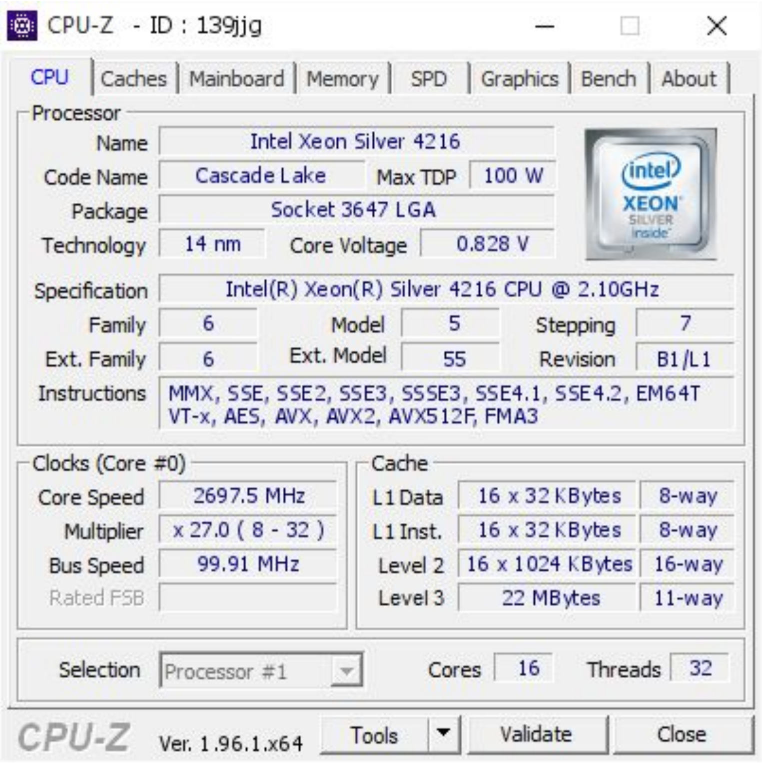
Table 2: Size of the dataset that was tested for training and inference.

Setup:

1. 1000 epoch of training
2. small hep dataset: 1 file of CMS open data
3. Batch size: 512
4. Optimizer
5. Hardware:
    a. Intel(R) Xeon(R) Silver
    b. Tesla T4

```
# === Configuration options ===        Radek Skaroupka, 5 months ago • Added new di


def set_config(c):
    c.input_path = "workspaces/CMS_workspace/data/example_CMS_data.npz"
    c.data_dimension = 1
    c.compression_ratio = 1.6
    c.apply_normalization = True
    c.model_name = "AE"
    c.epochs = 25
    c.lr = 0.001
    c.batch_size = 512
    c.early_stopping = True
    c.lr_scheduler = True

    # === Additional configuration options ===

    c.early_stopping_patience = 100
    c.min_delta = 0
    c.lr_scheduler_patience = 50
    c.custom_norm = False
    c.reg_param = 0.001
    c.RHO = 0.05
    c.test_size = 0
    # c.number_of_columns = 24
    # c.latent_space_size = 15
    c.extra_compression = False
    c.intermittent_model_saving = False
    c.intermittent_saving_patience = 100
    c.mse_avg = False
    c.mse_sum = True
    c.emd = False
    c.l1 = True
```

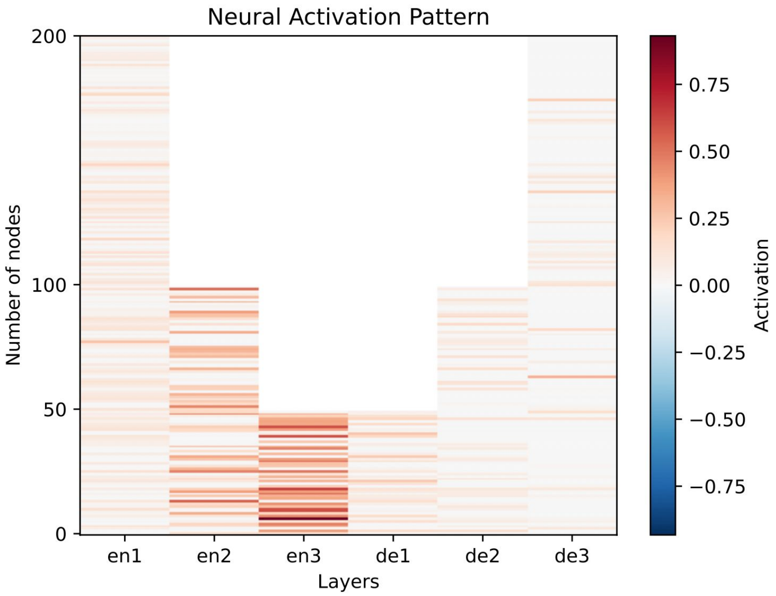
GPU execution

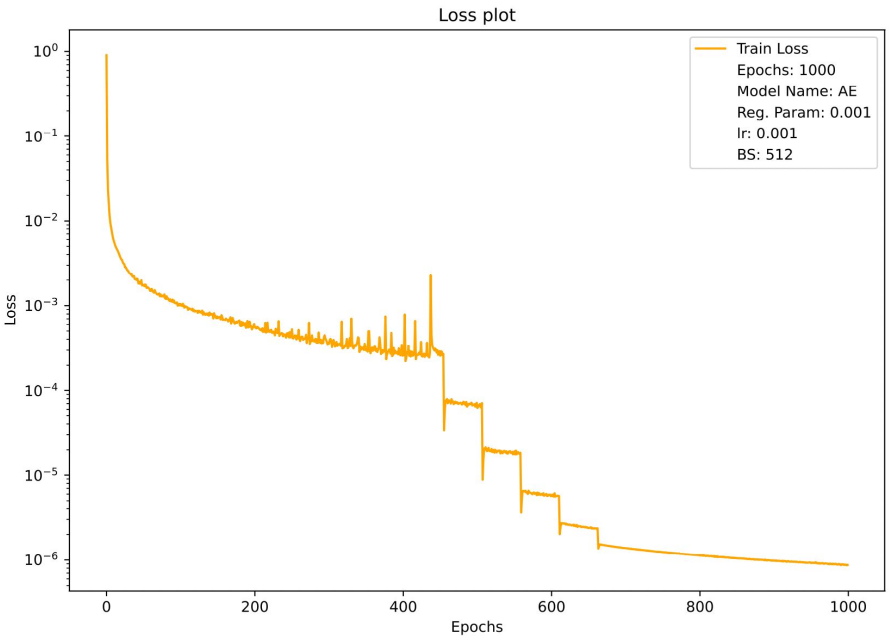


The GPU Specification (Source)

GPU Architecture	**NVIDIA Turing**
NVIDIA Turing Tensor Cores	**320**
NVIDIA CUDA® Cores	**2,560**
Single-Precision	**8.1 TFLOPS**
Mixed-Precision (FP16/FP32)	**65 TFLOPS**
INT8	**130 TOPS**
INT4	**260 TOPS**
GPU Memory	**16 GB GDDR6 300 GB/sec**
ECC	**Yes**
Interconnect Bandwidth	**32 GB/sec**
System Interface	**x16 PCIe Gen3**
Form Factor	**Low-Profile PCIe**
Thermal Solution	**Passive**
Compute APIs	**CUDA, NVIDIA TensorRT™, ONNX**

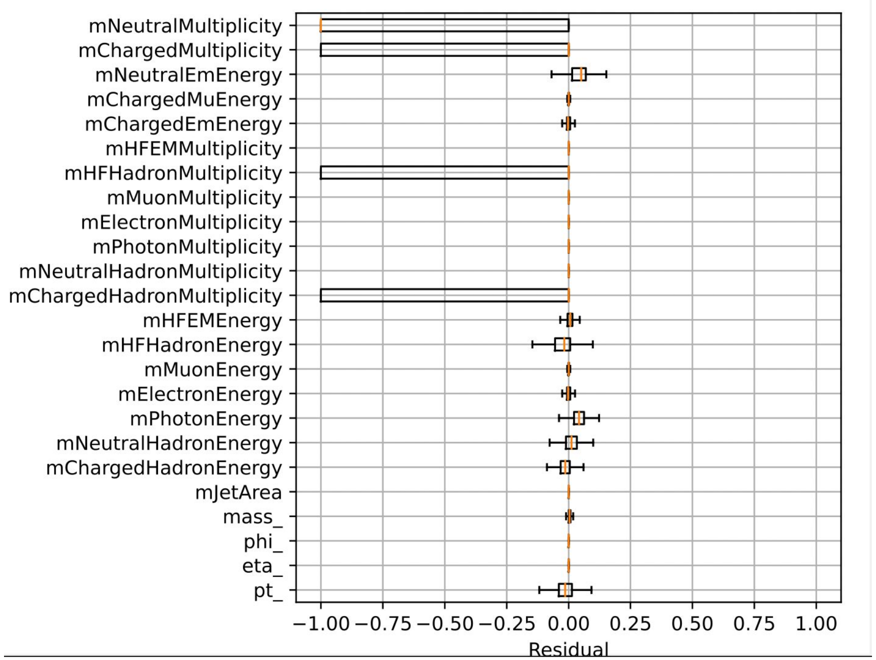
Result of Training:

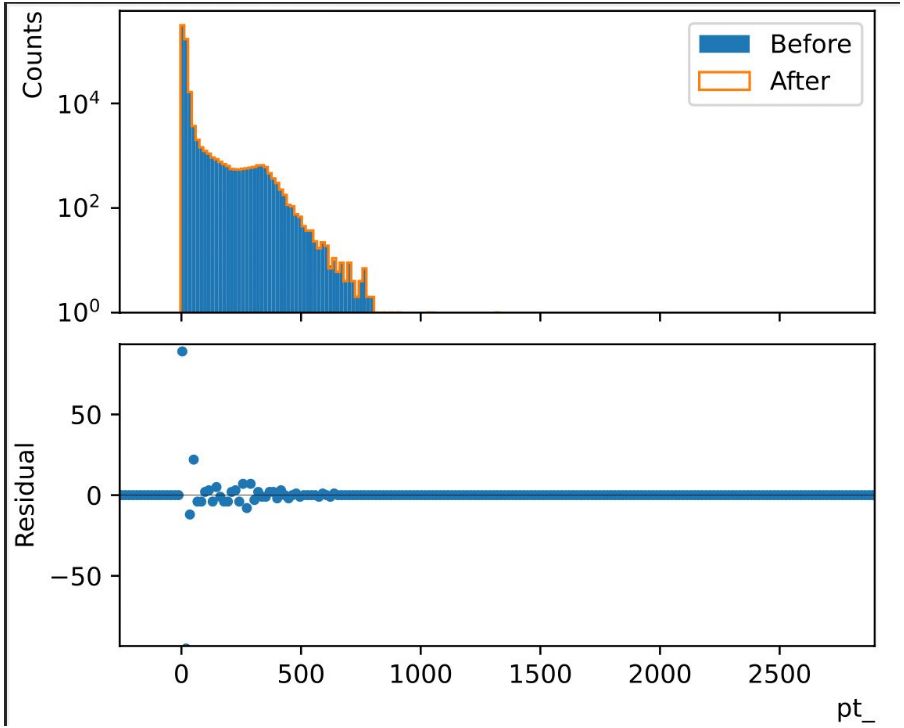




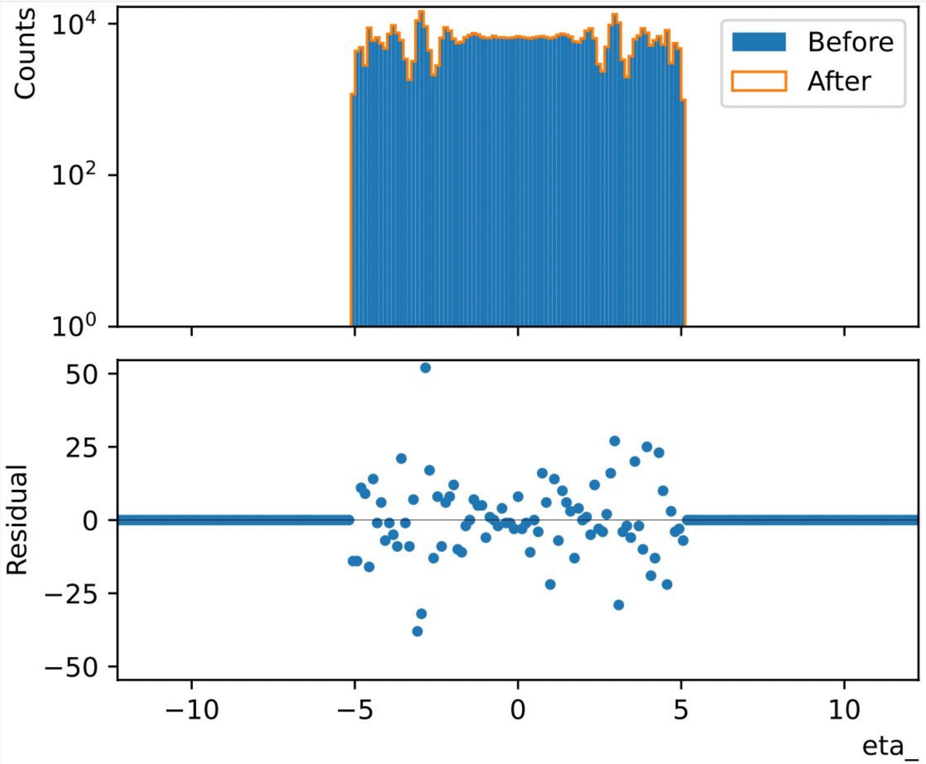
The activation function plot and the loss dynamics of training procedure.

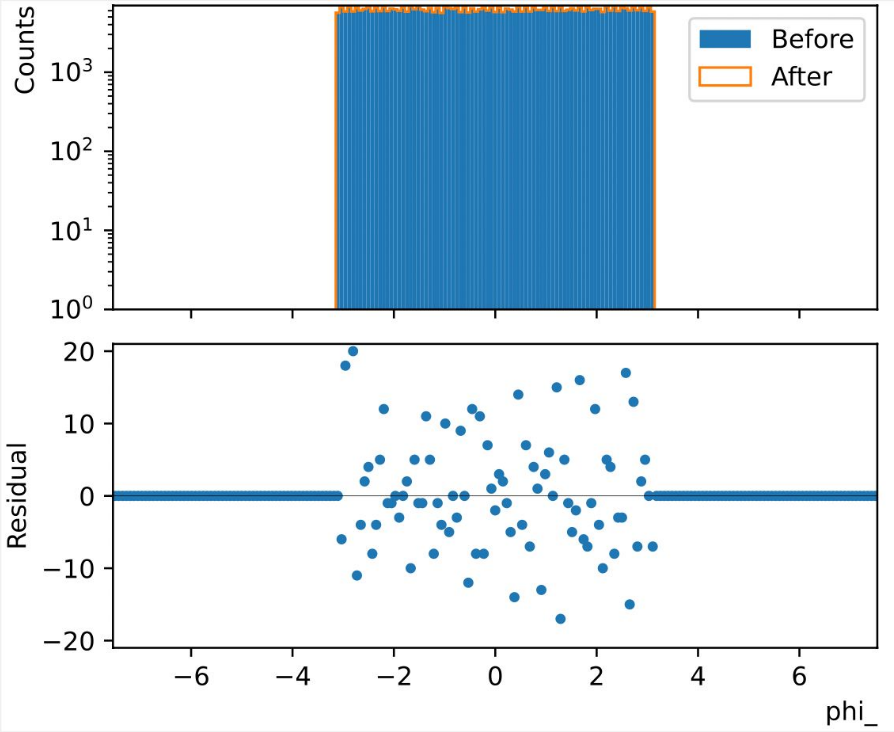
Result of Training





Result of Training





Profiling Metrics

https://jmlr.org/papers/volume21/20-312/20-312.pdf

- Wall Clock
- CPU/GPU time
- Total Time
- Number of operations:
  - MAC (Multiply-accumulate operation)

    is a floating-point multiply-add

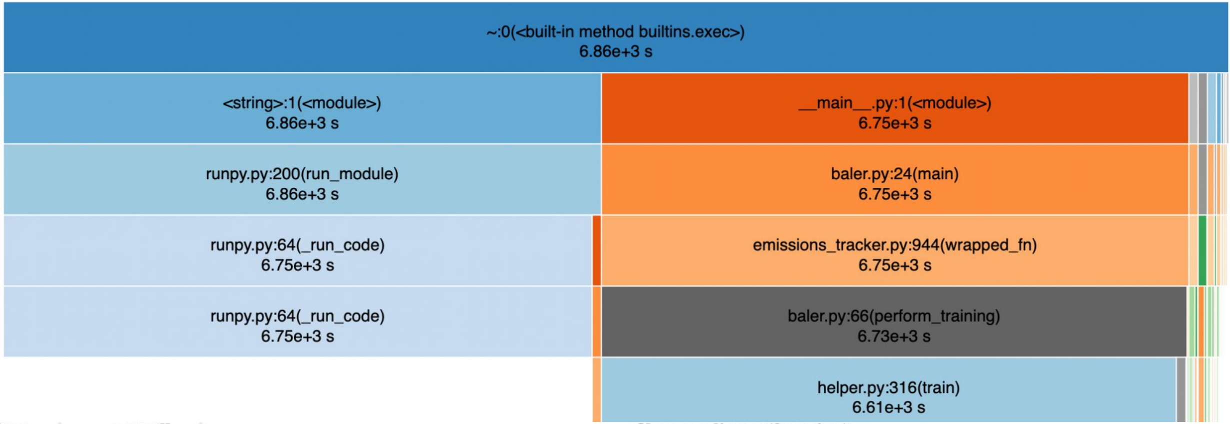
    operation performed in one step,with a single rounding
  - FLOPS - floating point operation
- Memory consumption
- Energy (Joules)
- Power consumption in Watts)

Profiling the training

Training profiling:

The most expensive operation is the the sampler

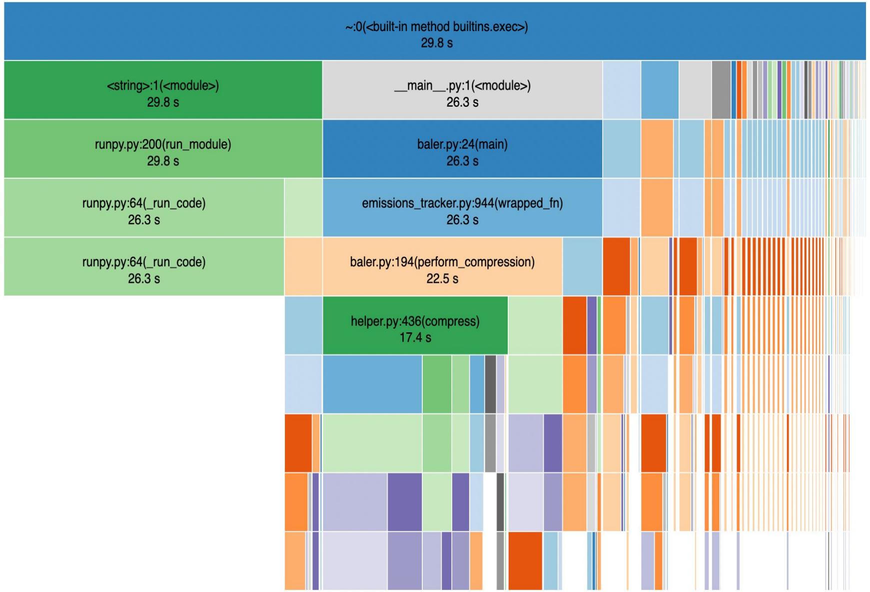
Profiling is done using cProfile and visualized by ShakeViz



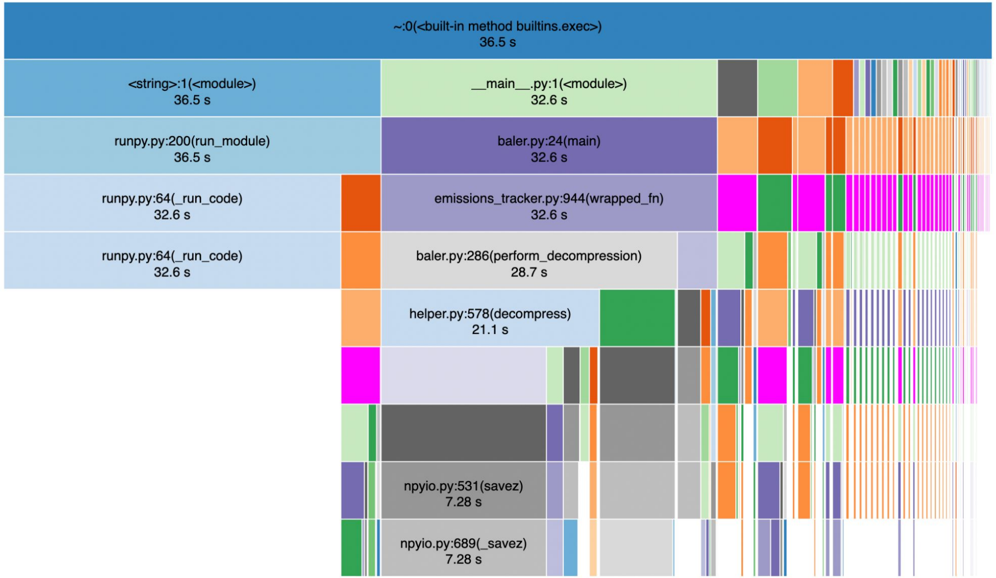
ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
273707	1242	0.004538	2444	0.008928	sampler.py:227(__iter__)
162495878	702.9	4.326e-06	702.9	4.326e-06	~:0(<method 'append' of 'list' objects>)
145115177/145094189	619.7	4.271e-06	619.9	4.272e-06	~:0(<built-in method builtins.len>)
273437	350	0.00128	350	0.00128	~:0(<method 'run_backward' of 'torch._C._EngineBase' objects>)
30624928	283.3	9.249e-06	414.7	1.354e-05	_tensor.py:1001(grad)
273437	280.3	0.001025	781.2	0.002857	_functional.py:54(adam)
273437	211.2	0.0007724	1383	0.005057	adam.py:81(step)
20126901	173.2	8.606e-06	259.2	1.288e-05	utils.py:374(<genexpr>)
273437	172.4	0.0006305	172.4	0.0006305	fetch.py:49(<listcomp>)

Result of Profiling

Compression:

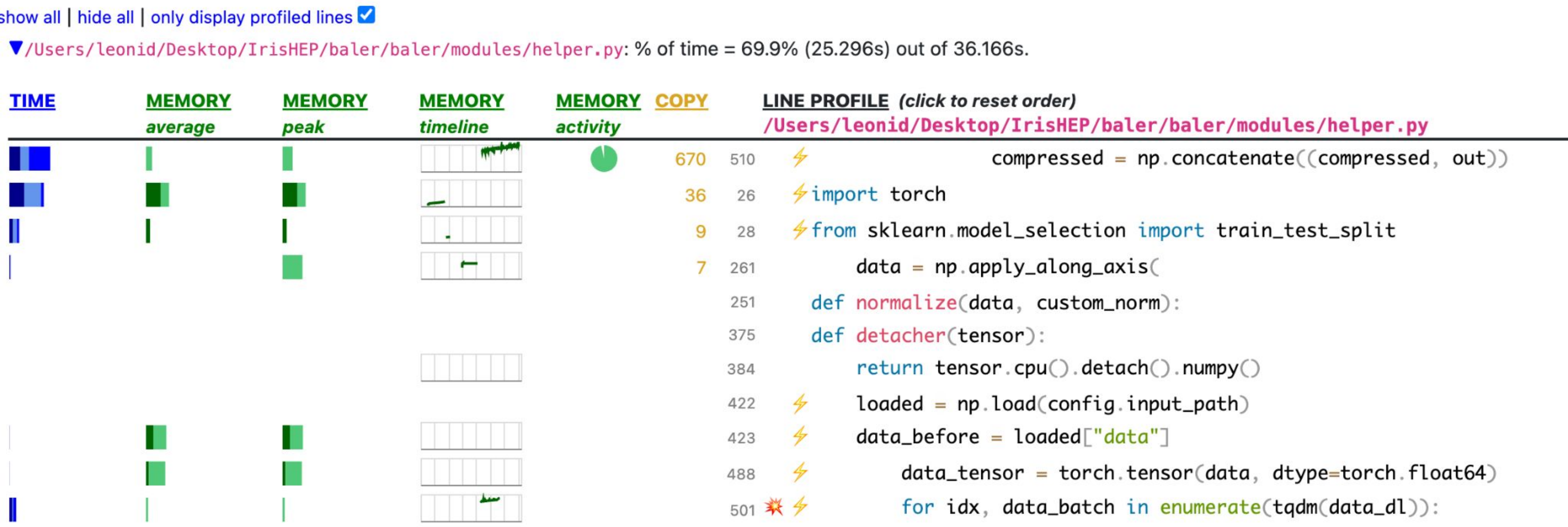


Decompression:



Profiling the Compression/decompression

Profiling is done using Scalene
Th numpy concatenation is the most costly operation and could be optimized.



Profiling the Compression

▼/Users/leonid/Desktop/IrisHEP/baler/baler/modules/models.py: % of time = 4.3% (1.553s) out of 36.166s.

TIME	MEMORY average	MEMORY peak	MEMORY timeline	MEMORY activity	COPY	LINE PROFILE (click to reset order) /Users/leonid/Desktop/IrisHEP/baler/baler/modules/models.py
						32 `self.en1 = nn.Linear(n_features, 200, dtype=torch.float64)`
						45 💥 ⚡ `def encode(self, x):`
						46 ⚡ `h1 = F.leaky_relu(self.en1(x))`
						47 ⚡ `h2 = F.leaky_relu(self.en2(h1))`
						48 ⚡ `h3 = F.leaky_relu(self.en3(h2))`
						49 ⚡ `return self.en4(h3)`

TIME	MEMORY average	MEMORY peak	MEMORY timeline	MEMORY activity	COPY	FUNCTION PROFILE (click to reset order) /Users/leonid/Desktop/IrisHEP/baler/baler/modules/models.py
						26 AE.__init__
						45 AE.encode

▼baler.py: % of time = 2.0% (720.481ms) out of 36.166s.

Profiling the Compression/decompression

show all | hide all | only display profiled lines ☑

▼/Users/leonid/Desktop/IrisHEP/baler/baler/modules/data_processing.py: % of time = 72.2% (56.250s) out of 1m:17.916s.

TIME	MEMORY average	MEMORY peak	MEMORY timeline	MEMORY activity	COPY	LINE PROFILE (click to reset order) /Users/leonid/Desktop/IrisHEP/baler/baler/modules/data_processing.py
					34	145 ⚡ `return np.array([((i * feature_range) + true_min) for i in list(input_data)])`
					3	160 ⚡ `norm_data = np.array(norm_data)`
						66 `model.to(device)`
						131 💥 ⚡ `def renormalize_std(`
						148 💥 ⚡ `def renormalize_func(norm_data: ndarray, min_list: List, range_list: List) -> ndarray:`
						166 ⚡ `renormalized_full = np.array(renormalized_full).T`

▼/Users/leonid/Desktop/IrisHEP/baler/baler/modules/models.py: % of time = 5.2% (4.056s) out of 1m:17.916s.

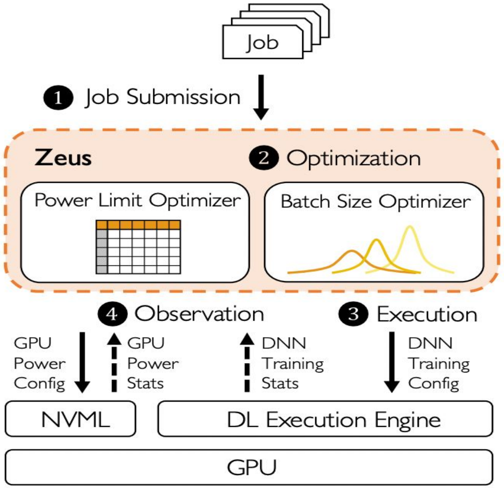
TIME	MEMORY average	MEMORY peak	MEMORY timeline	MEMORY activity	COPY	LINE PROFILE (click to reset order) /Users/leonid/Desktop/IrisHEP/baler/baler/modules/models.py
						51 💥 ⚡ `def decode(self, z):`
						52 ⚡ `h4 = F.leaky_relu(self.de1(z))`
						53 ⚡ `h5 = F.leaky_relu(self.de2(h4))`
						54 ⚡ `h6 = F.leaky_relu(self.de3(h5))`
						55 ⚡ `out = self.de4(h6)`
						209 `def encode(self, x):`

TIME	MEMORY average	MEMORY peak	MEMORY timeline	MEMORY activity	COPY	FUNCTION PROFILE (click to reset order) /Users/leonid/Desktop/IrisHEP/baler/baler/modules/models.py
						51 `AE.decode`
						162 `SourceFileLoader.AE_Dropout_BN`

Zeus-ML Energy Meter

Zeus-ML Energy meter

```
from zeus.monitor import ZeusMonitor

monitor = ZeusMonitor(gpu_indices=[0,1,2,3])

monitor.begin_window("heavy computation")
# Four GPUs consuming energy like crazy!
measurement = monitor.end_window("heavy computation")

print(f"Energy: {measurement.total_energy} J")
print(f"Time  : {measurement.time} s")
```

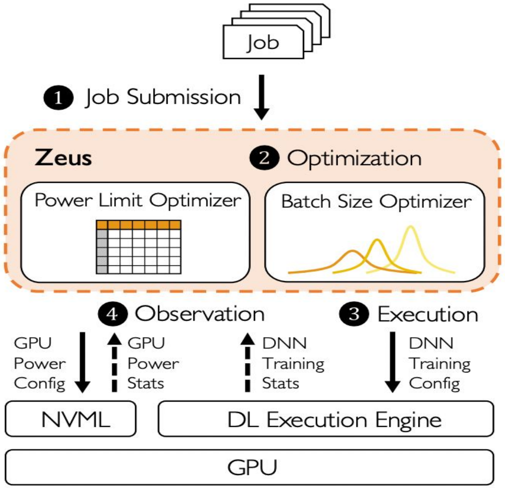


$$\text{Cost} = \eta \cdot \text{ETA} + (1 - \eta) \cdot \text{MaxPower} \cdot \text{TTA}$$

Zeus-ML Energy Meter

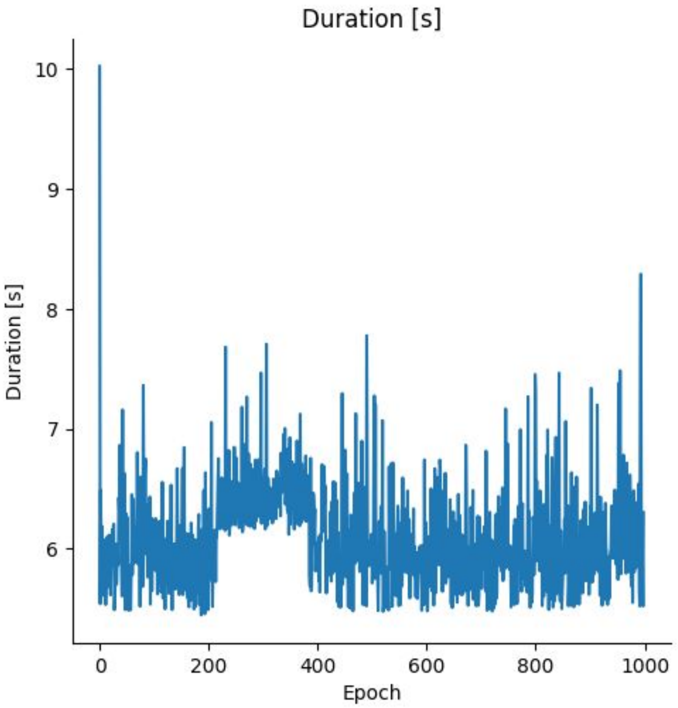
Read the data from nvml
Can optimize the power level and batch size
Cost:
Energy to Accuracy (ETA),
energy required to reach accuracy in our case is l2 score.

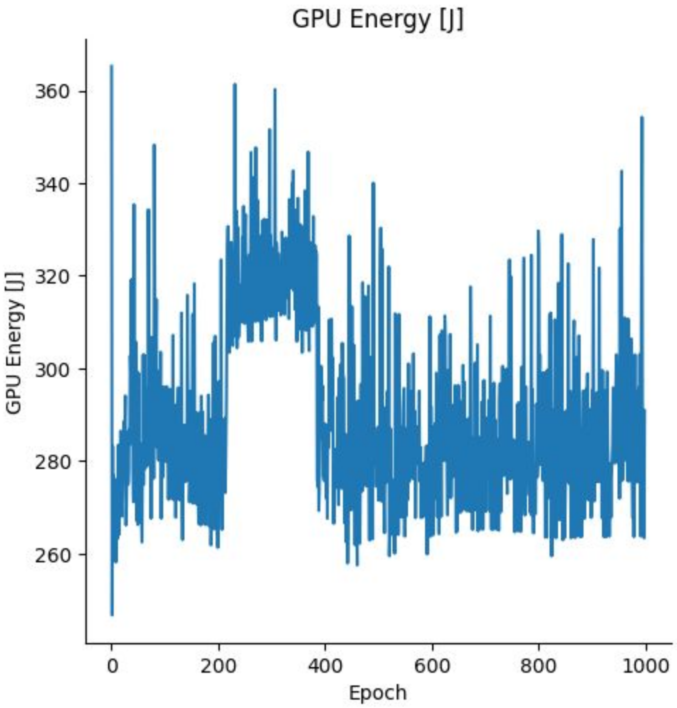
TTA - Time to Accuracy
time required to reach accuracy



$$\text{Cost} = \eta \cdot \text{ETA} + (1 - \eta) \cdot \text{MaxPower} \cdot \text{TTA}$$

Zeus-ML Energy Meter





One step took 6.220813512802124 s and 290.7010000000093 J on average.
Total duration: 1.02e+02 minutes

CodeCarbon Energy Meter

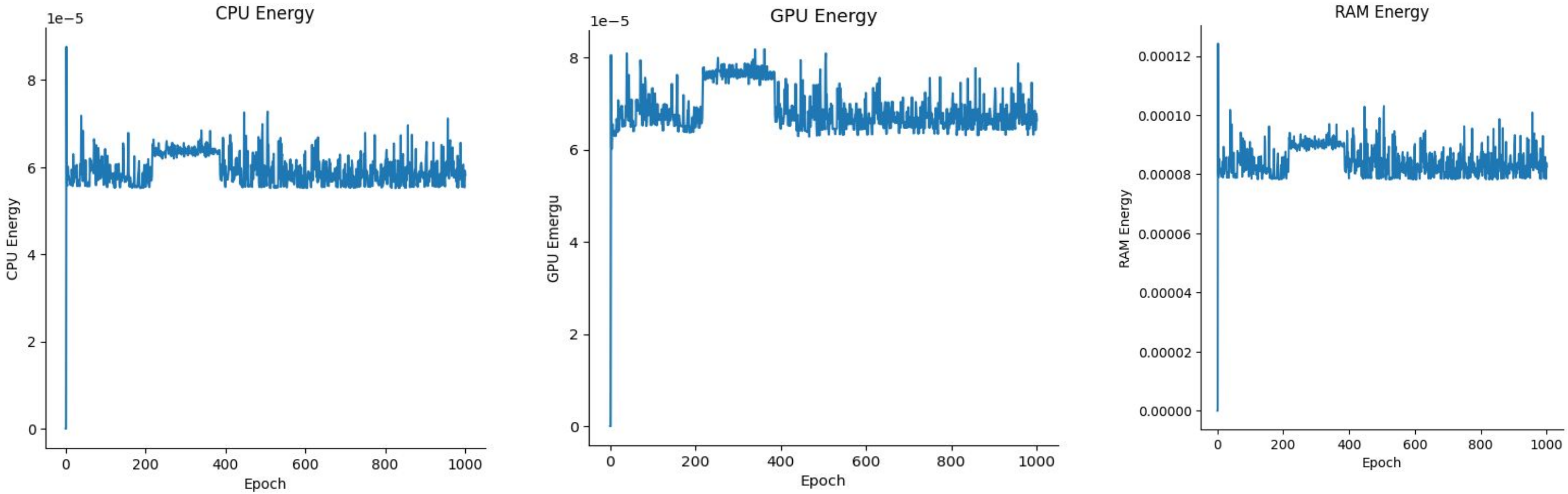
Profiling, energy and CO(2) meters #302 #303

Open neogyk wants to merge 1 commit into `baler-collaboration:main` from `neogyk:302-add-profiler-and-energy-meters`

```python
from codecarbon import EmissionsTracker
tracker = EmissionsTracker()
tracker.start()
try:
     # Compute intensive code goes here
     _ = 1 + 1
finally:
     tracker.stop()
```

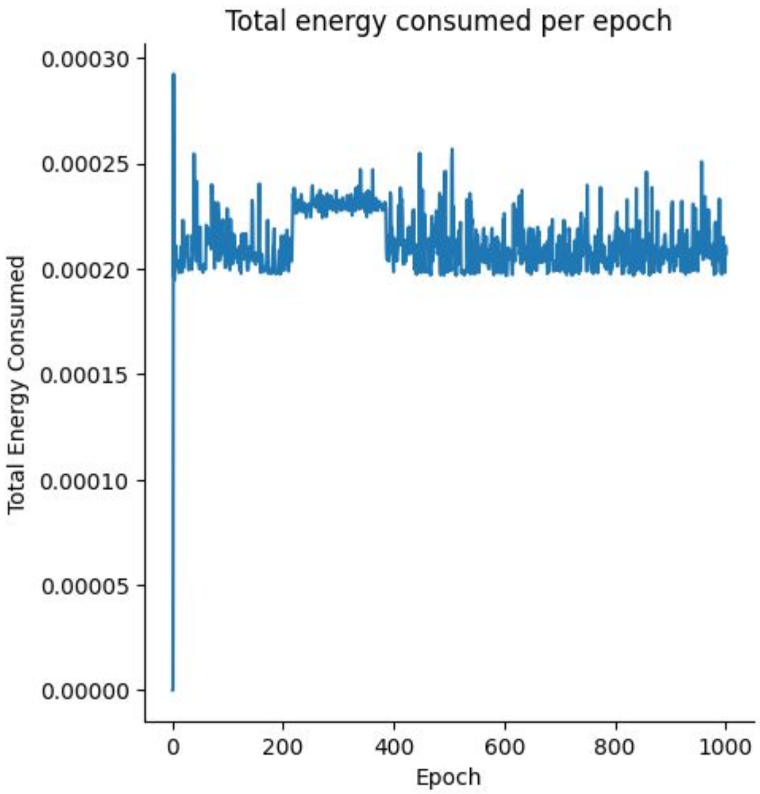
This energy meter provides the information about energy consumed by RAM, CPU, GPU and CO(2) emission.

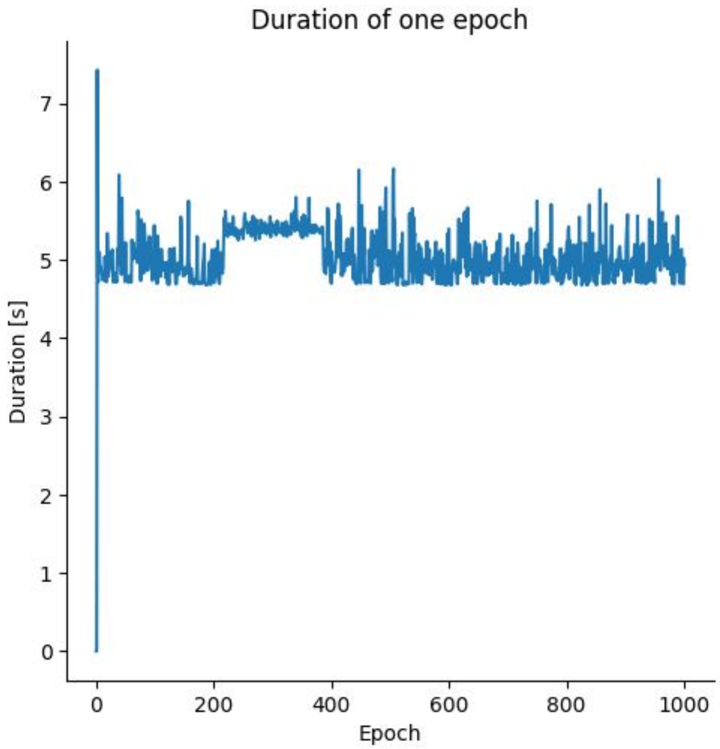
CodeCarbon Energy Meter



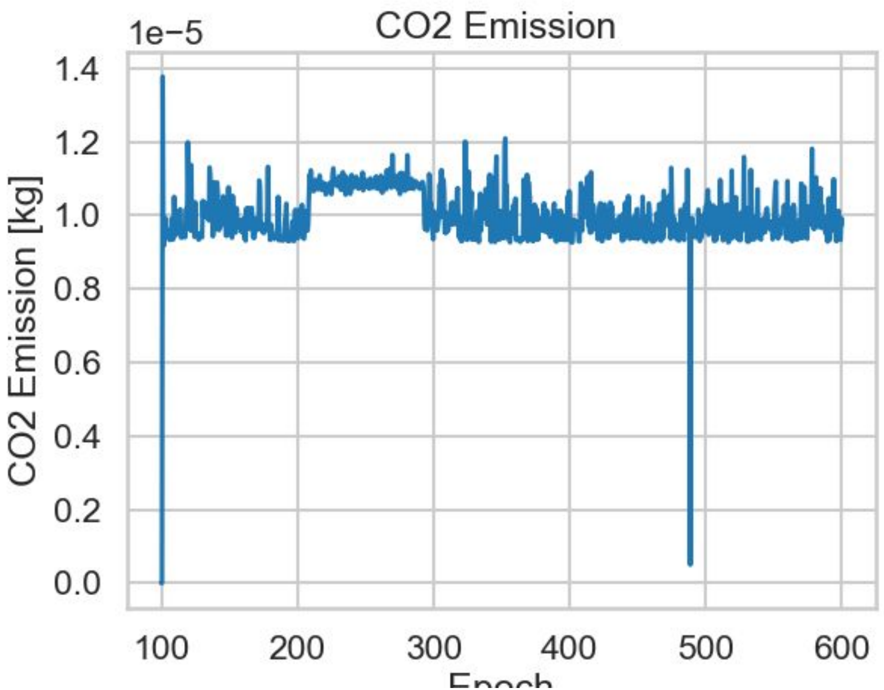
One step took 5.032286106469389 s and 290.70100000000093 J on average.
Total duration: 1.02e+02 minutes

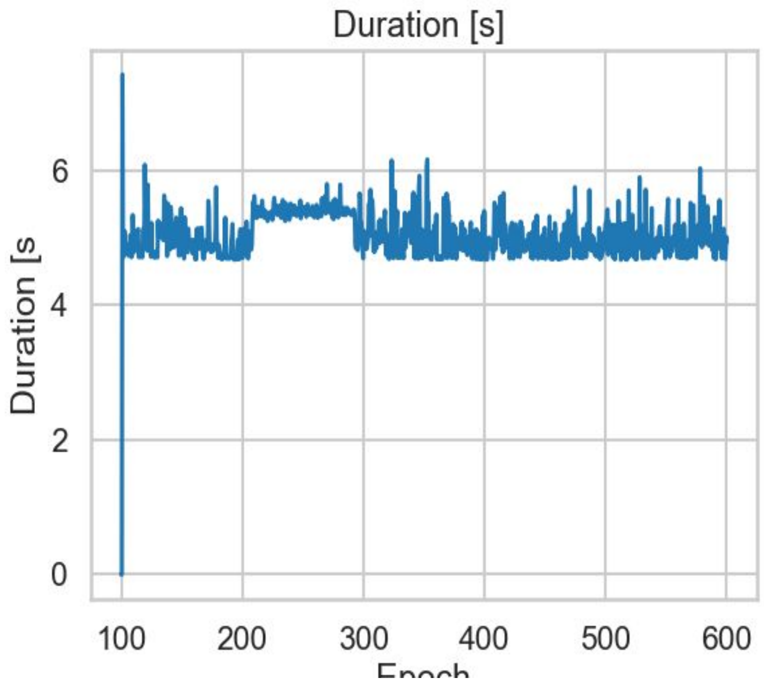
CodeCarbon Energy Meter



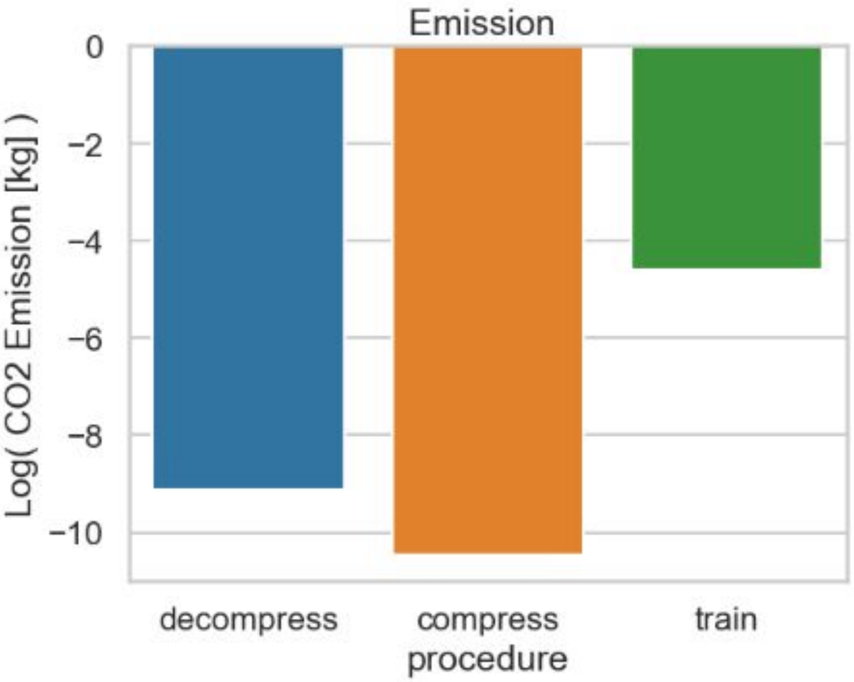


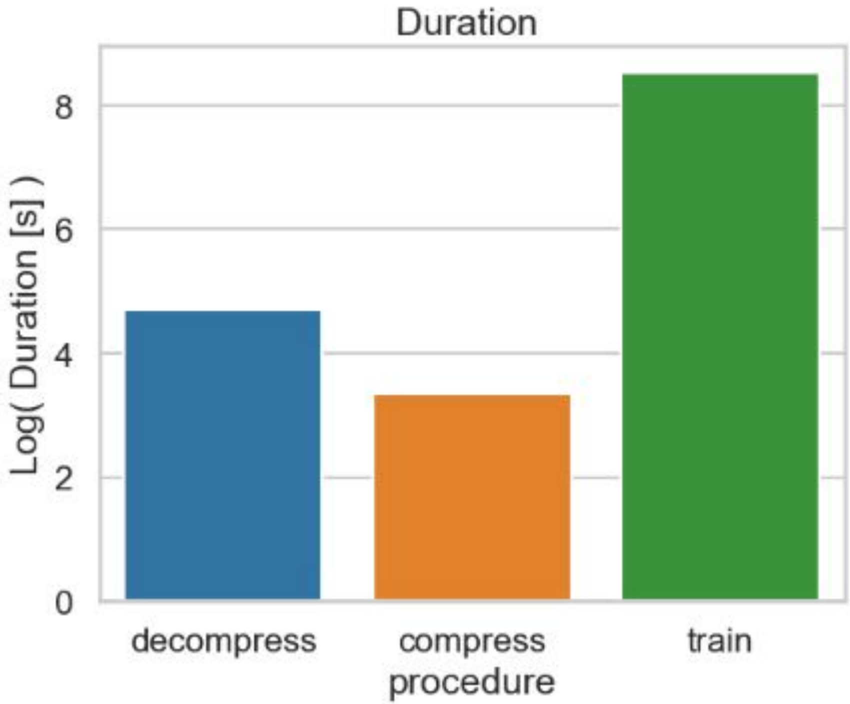
CodeCarbon Energy Meter





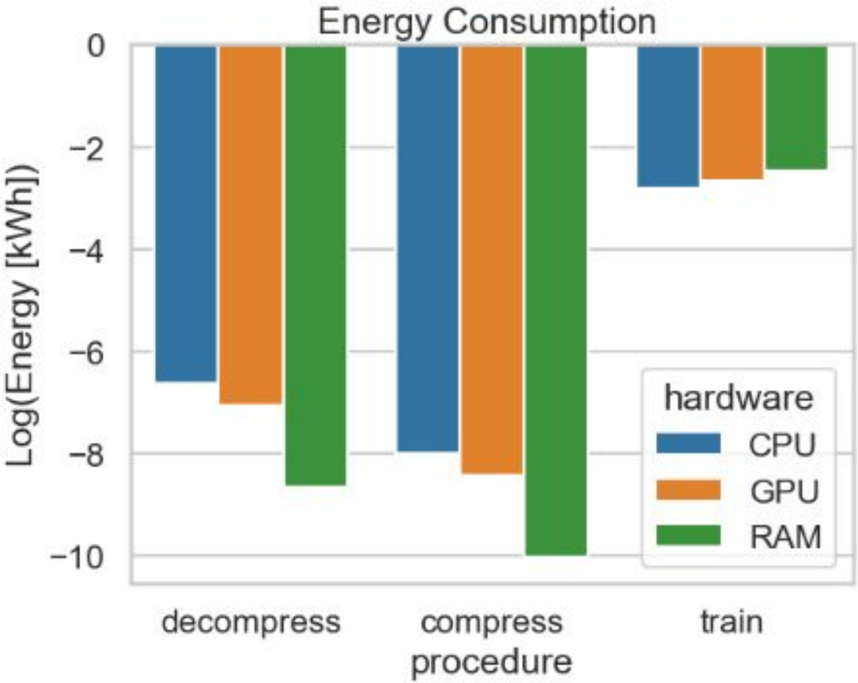
Code Carbon Energy Meter

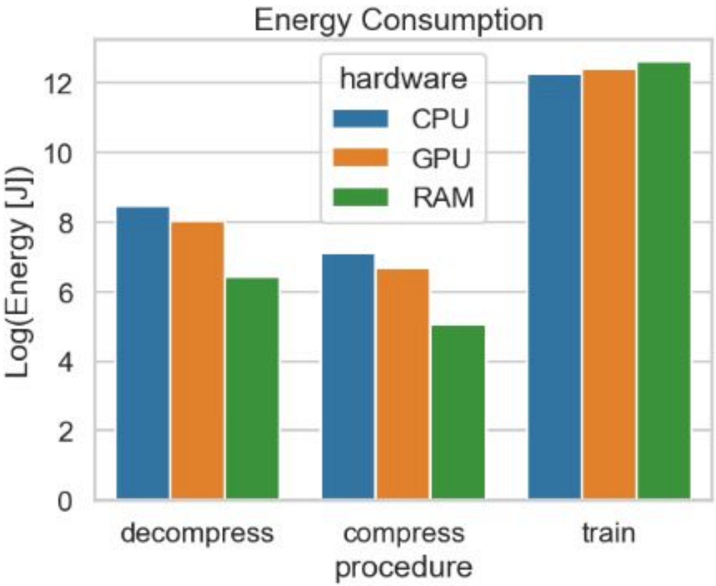





```
0.010010516955963899 kg of CO(2) emitted during training
```

Code Carbon Energy Meter



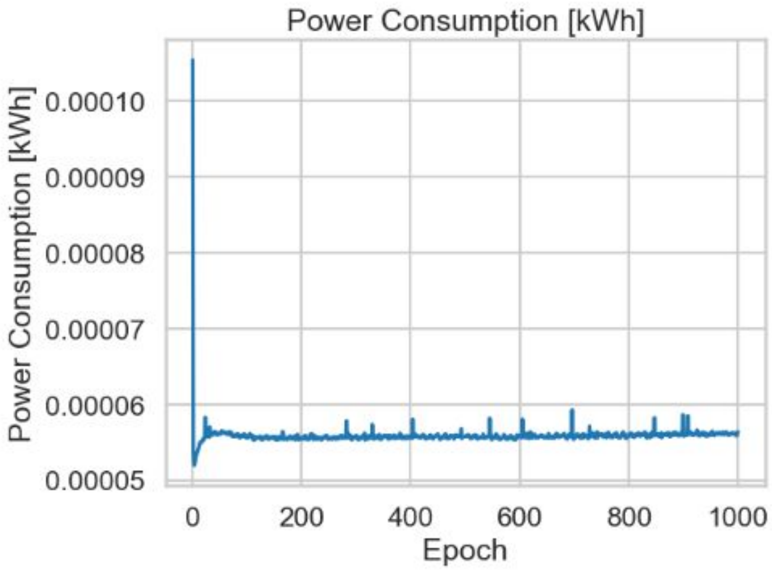


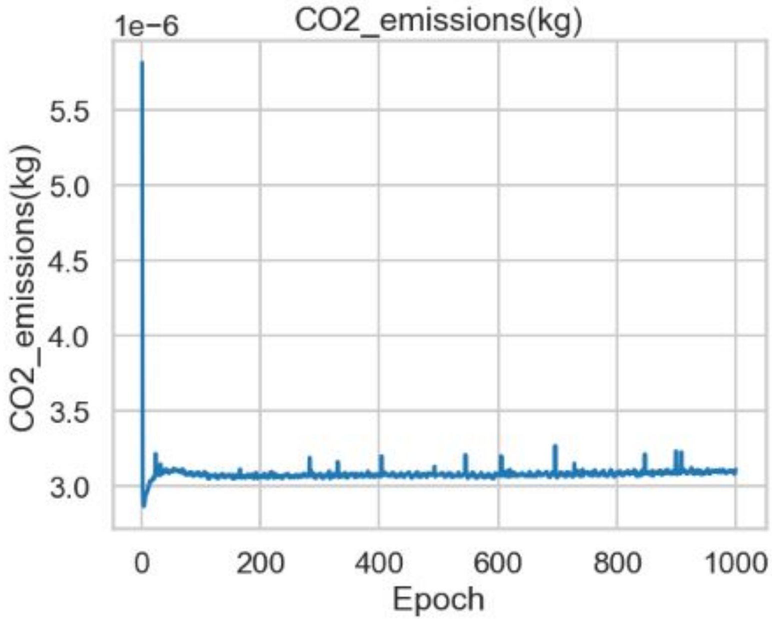
Eco2AI Energy Meter

```
import eco2ai

tracker = eco2ai.Tracker(project_name="YourProjectName", experiment_description="training the <your

tracker.start()

<your gpu &(or) cpu calculations>

tracker.stop()
```

The Eco2AI is a python library for CO_2 emission tracking. It monitors energy consumption of CPU & GPU devices and estimates equivalent carbon emissions taking into account the regional emission coefficient.

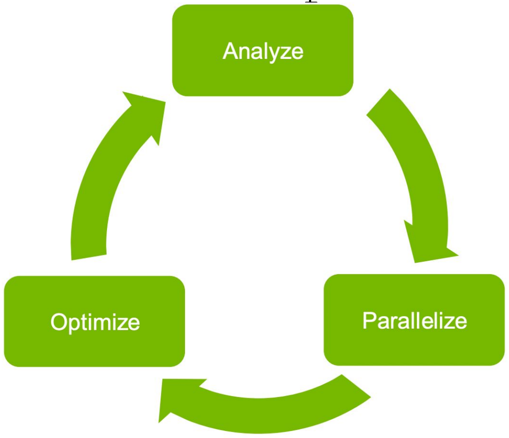
Eco2AI Energy Meter





How it's possible to optimize:

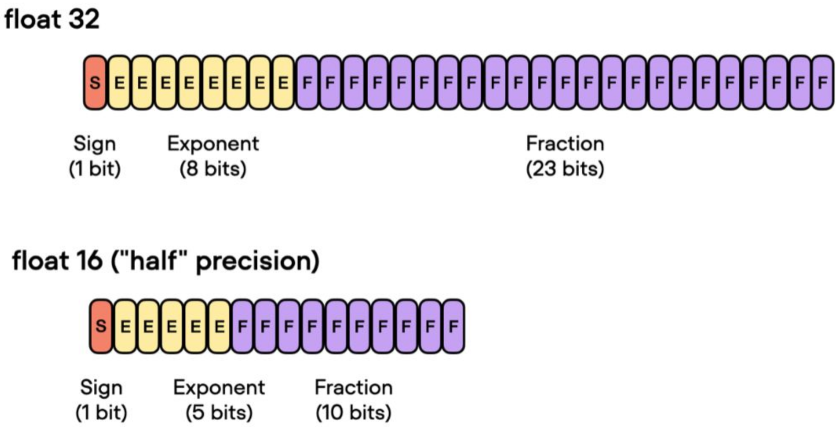
1. Optimize the GPU power
2. Optimize the batch size or other hyper parameters:
   a. Consider another LR Scheduler, Optimizer
   b. The data loading and data copying is the most costly operations in this framework
3. Use the jit library: numba, cupy
4. Use automatic mixed precision training
5. Use the Data parallel/model parallel strategies in case of distributed tr



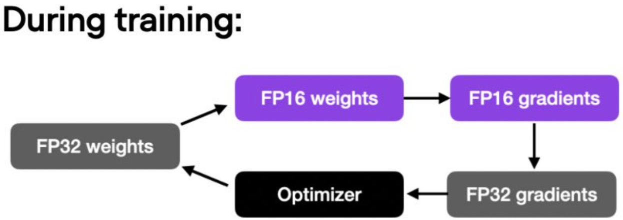
Mixed precision training

https://arxiv.org/abs/1710.03740

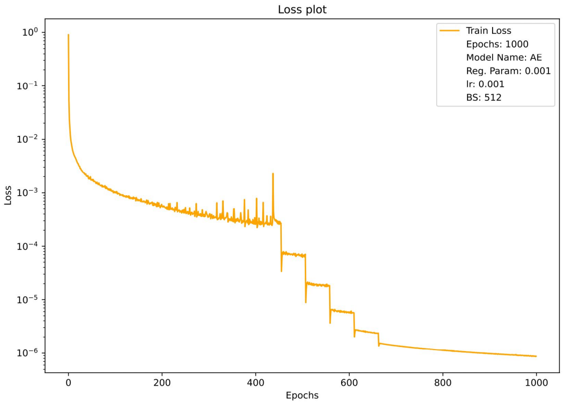
```
import torch
# Creates once at the beginning of training
scaler = torch.cuda.amp.GradScaler()

for data, label in data_iter:
   optimizer.zero_grad()
   # Casts operations to mixed precision
   with torch.amp.autocast(device_type="cuda", dtype=torch.float16):
      loss = model(data)

   # Scales the loss, and calls backward()
   # to create scaled gradients
   scaler.scale(loss).backward()

   # Unscales gradients and calls
   # or skips optimizer.step()
   scaler.step(optimizer)

   # Updates the scale for next iteration
   scaler.update()
```

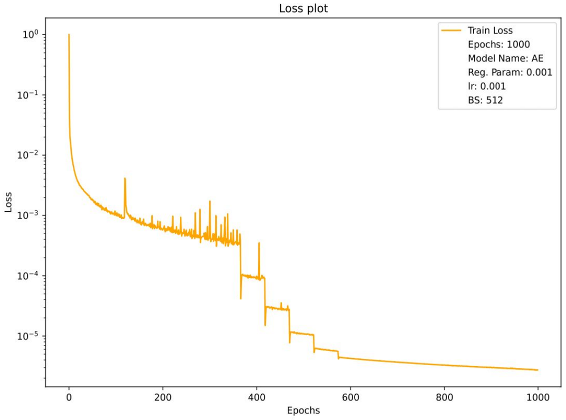


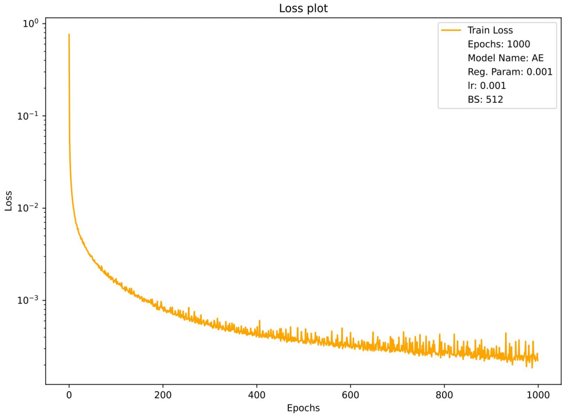
we use automatic mixed precision training, which switches between 32-bit and 16-bit floating point representations during training without sacrificing accuracy



Mixed precision training







Normal training

Total execution time: 1.02e+02 minutes
Total execution time: 6041.347 sec
Energy:0.21273390494325kWh

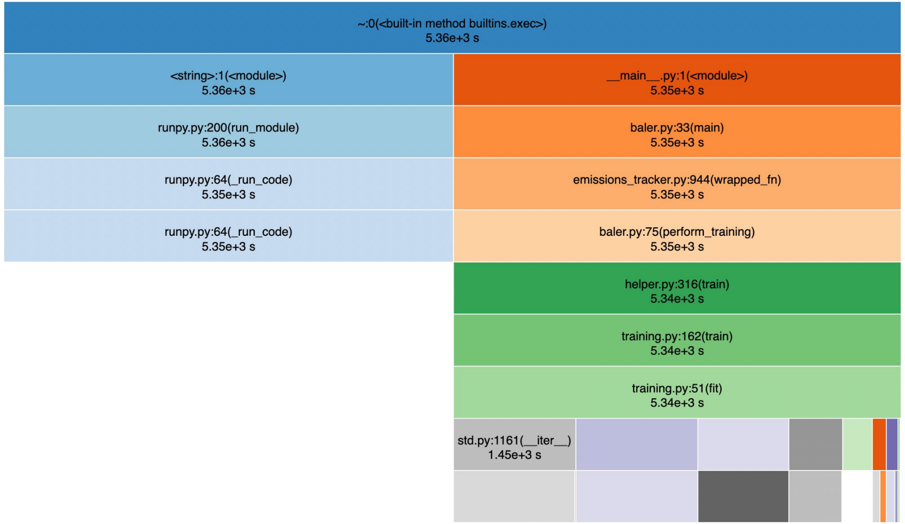
Automatic Mixed Precision without scaling

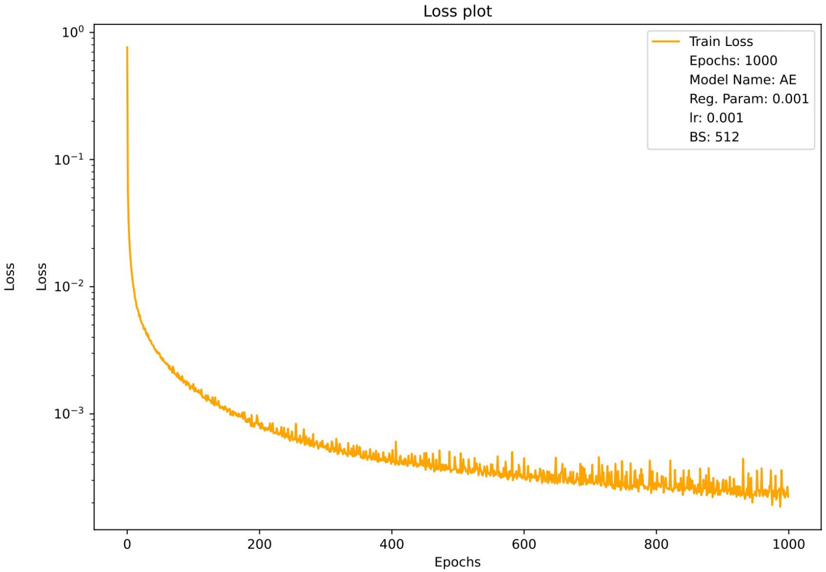
Total execution time: 89.0 minutes
Total execution time: 5338.941 sec
Energy:0.143103kWh

Automatic Mixed Precision with scaling

Total execution time: 92.8 minutes
Total execution time: 5569.683 sec
Energy:0.148851kWh

Mixed precision training





AMP can reduce the running time, but the accuracy has to be tuned.

Conclusion

- We measured the time and operation related metrics for training and inference.
- Measured the power consumption and CO(2) emission.
- Aprobated the AMP as a way to speed up the training procedure and reduce energy cost.
- Results of the experiments - https://github.com/software-energy-cost-studies/profiling

- Big thanks to Caterina Doglioni, Alexander Ekman and Baler Collaboration

Questions