Treasure pipeline discussion

Merve Nazlim Agaras

20.5.26

The content might be too much, as we did not meet past weeks.
I will not go through them one by one.

Intro

* Today's update
  - BNL -> heptokens integration status
  - Tokenization and model workflow and added functionalities
  - Comparison of the two frameworks

* Main decision
  - Which framework to continue for which tasks
  - 3 options
    - BNL
    - Heptokens
    - Refined heptokens Nazlim's fork (tokenization + model training)

* **** Nicholas and I are working on the fork of heptokens which contains the new implementations.

Decision Point: Where Should This Workflow Live?

* We worked on integrating the BNL event-level workflow into heptokens, focusing on two parts: event tokenization and downstream event-level modeling.
  - BNL code https://gitlab.cern.ch/vcavalie/bnl-treasure
  - Heptokens code https://github.com/MerveNazlim/heptokens/tree/event_level

* BNL workflow (script based) -> event level tokenisation + classification model pre-training/fine tunning
* Heptokens (Package/config-driven framework) -> jet level tokenisation + jet classification

* I tested how far the BNL workflow can be integrated into heptokens. The technical integration is possible, but we need to decide how to do.
  - should the full workflow live in **heptokens** or
  - should **heptokens** only provide tokenizers/data loading while the event-level model stays in a separate package (using heptokens framework)
  - or event-level model stays in a separate BNL package (using BNL script based framework)?

What Was Moved From BNL/added into Heptokens type fr

* Data
  - Event-level HDF5 dataloader for ATLAS-style files
  - Configurable object collections instead of hardcoded paths
  - Tokenized Parquet reader
* Tokenization:
  - one tokenizer across all objects
  - one tokenizer per object type
  - separate object outputs for future studies
  - VQ-VAE tokenizer training through Hydra/Pixi
* Model
  - Event sequence transformer
  - masked-token pretraining
  - direct classification
  - fine-tuning from pretrained backbone
* Post-training diagnostics:
  - reconstruction vs original
  - residuals
  - codebook usage
  - dead tokens / token frequency

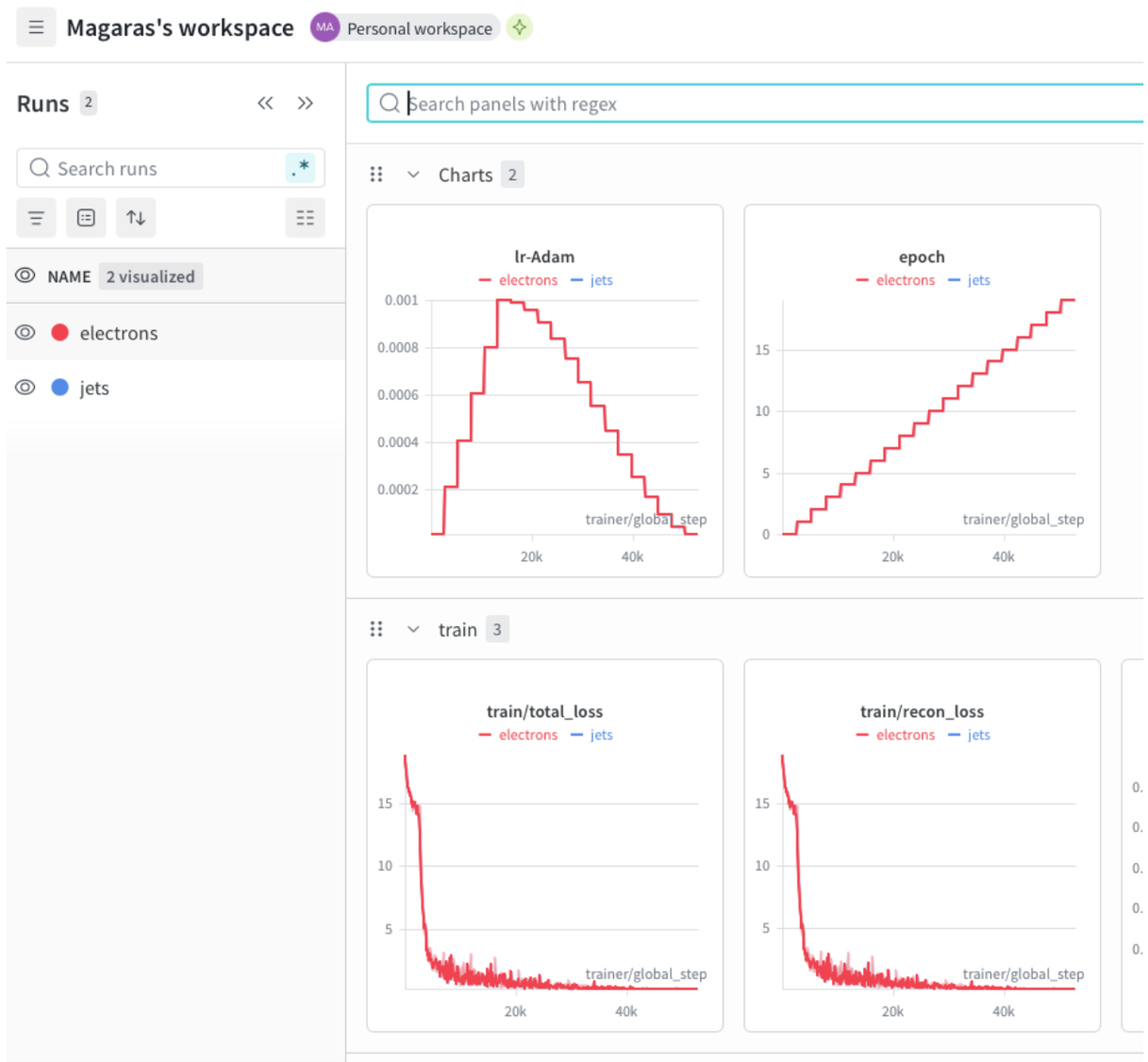
I know its alot in one go, but I wanted to test the full pipeline, the review can be done separately

“

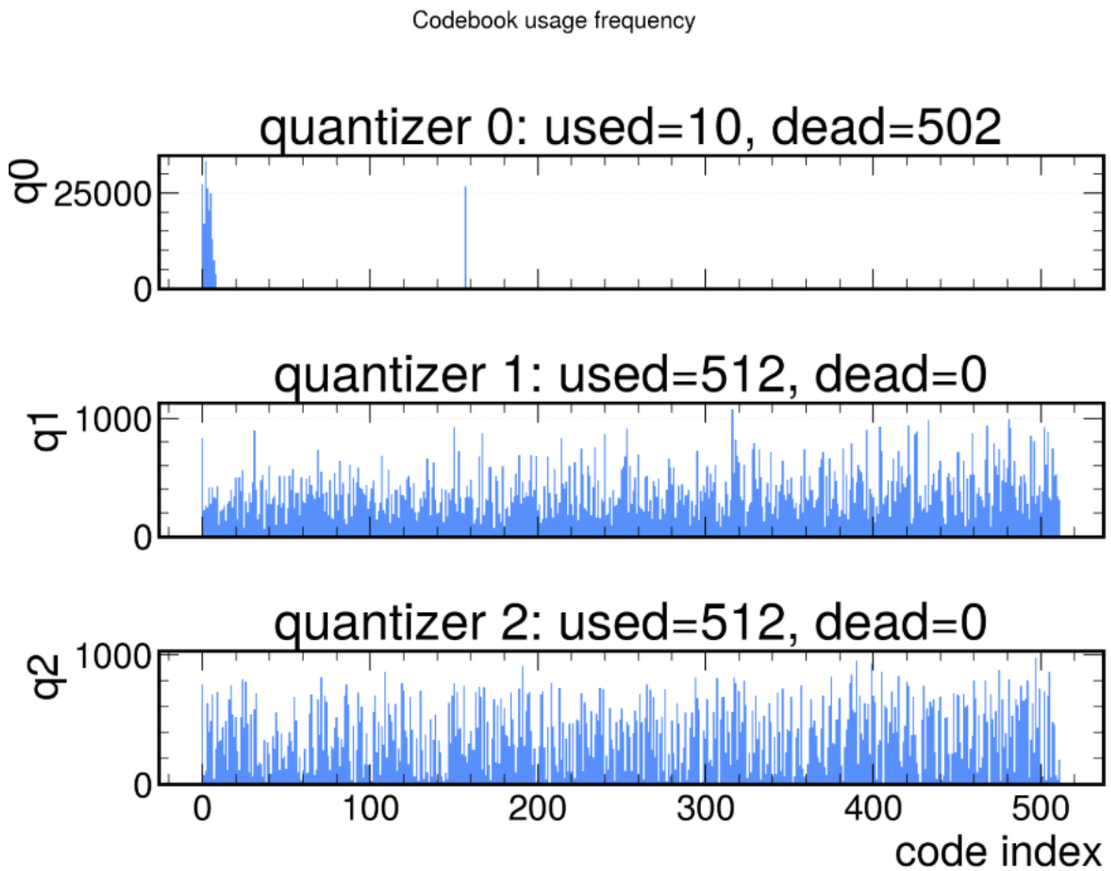
Event tokenization

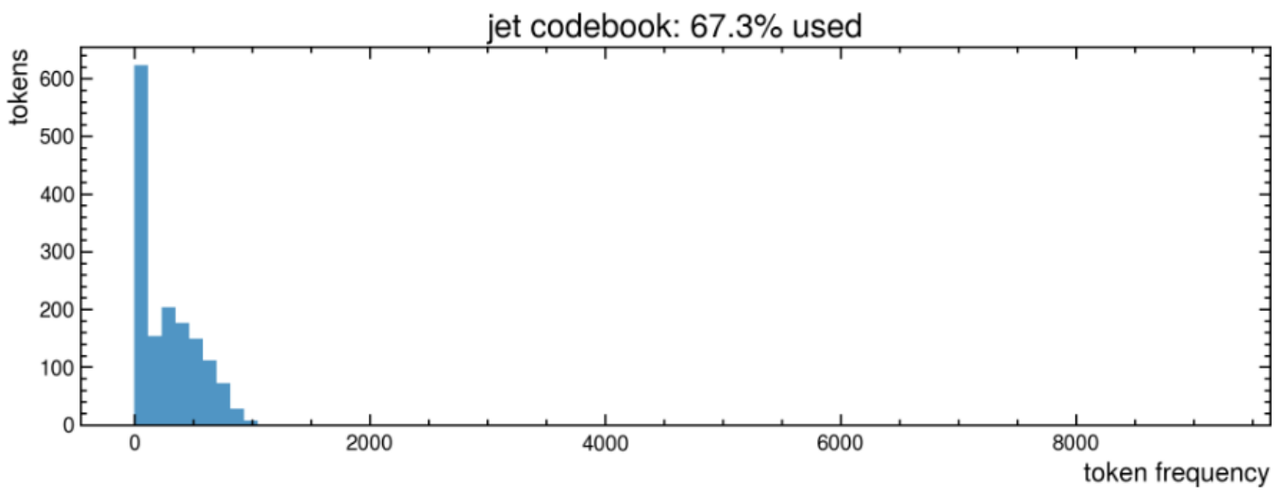
Event tokenization from BNL to Heptokens

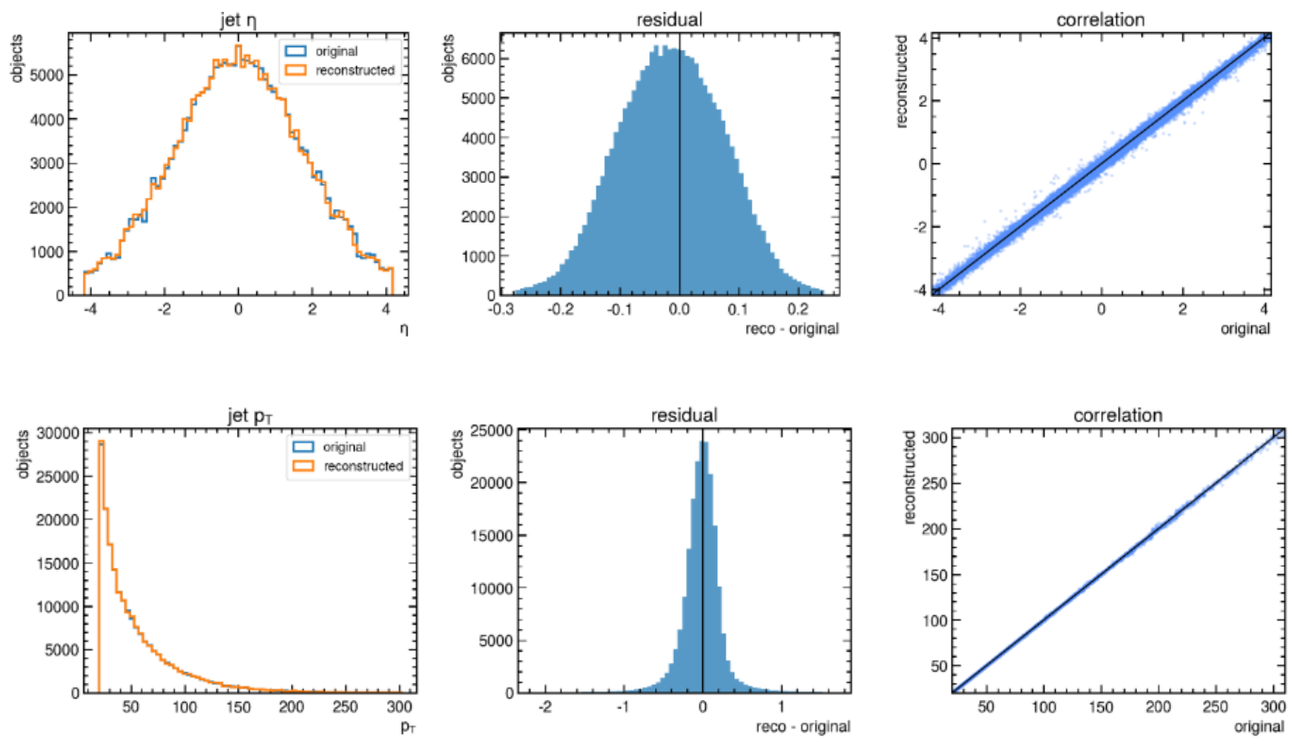
* Output modes:
  - **combined**: one tokenizer across object types; concatenates objects into csts/mask
  - **object**: one tokenizer for one object type; set object_type below
  - **separate**: one tensor per object type for future joint event-tokenizer studies
* Per object setup:
  - Each job produces one VQ-VAE checkpoint. Those checkpoints are then applied together to build event-level token sequences
* **Added** a script to produce the same style parquet files with Masks, embeddings etc.
* Fast training ~
  - TOTAL events = 4M
  - TOTAL jets = 17M

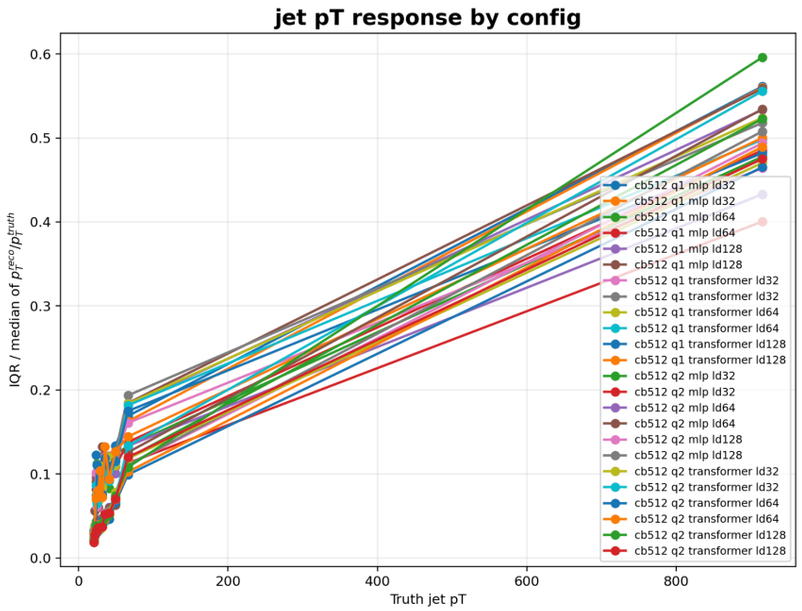


Diagnostics









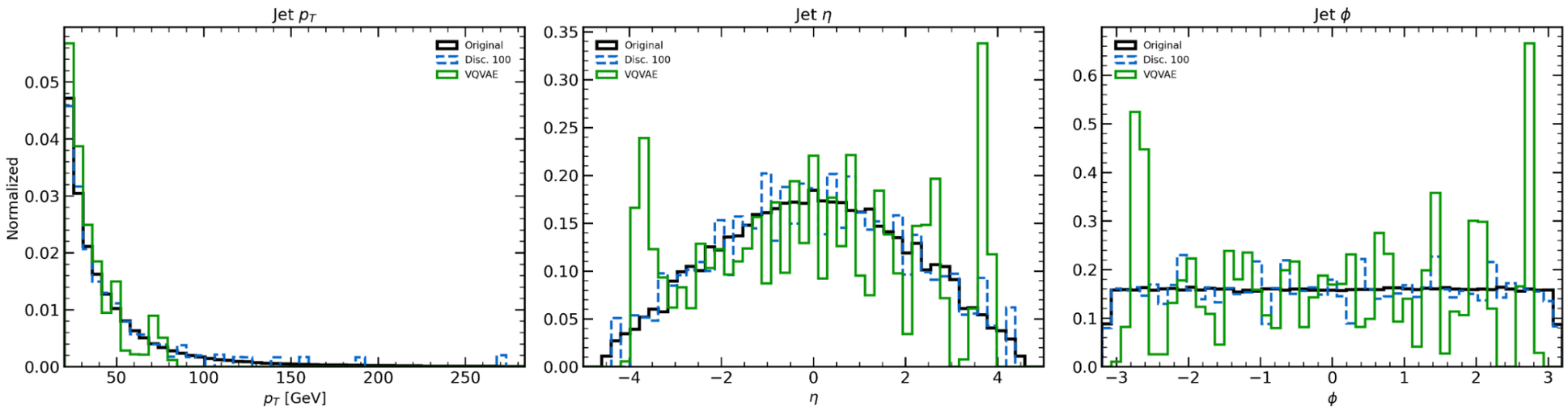
Difference between heptokens and BNL

BNL
quantile normalization
single VQ-VAE codebook with EMA updates
dead-code reset
MSE reconstruction loss
max 2M fit events / up to 1M jet

Heptokens
no normalization
ResidualVQ-style model
3 quantizers
L1 reconstruction loss
all loaded jet objects from your files

* Was able to run over 2M events only for jets (adapted the code)
* Memory issues for more

BNL

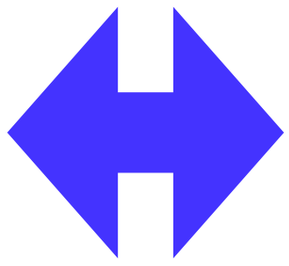


“

Downstream tasks

Downstream model from BNL to Heptokens

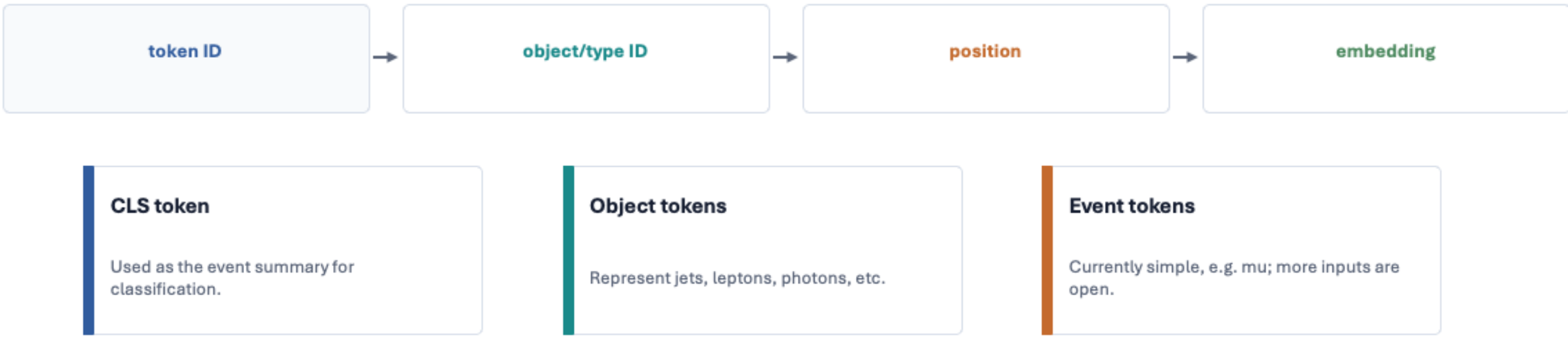
* I refactored the foundation-model logic into the heptokens package layout:
  - sequence batch handling
  - Parquet loading
  - shared transformer backbone
  - masked sequence pretraining
  - sequence classification/fine-tuning



```
src/heptokens/data/
  sequence.py
  token_parquet.py

src/heptokens/models/
  sequence_backbone.py
  foundation.py
  classifier.py
```

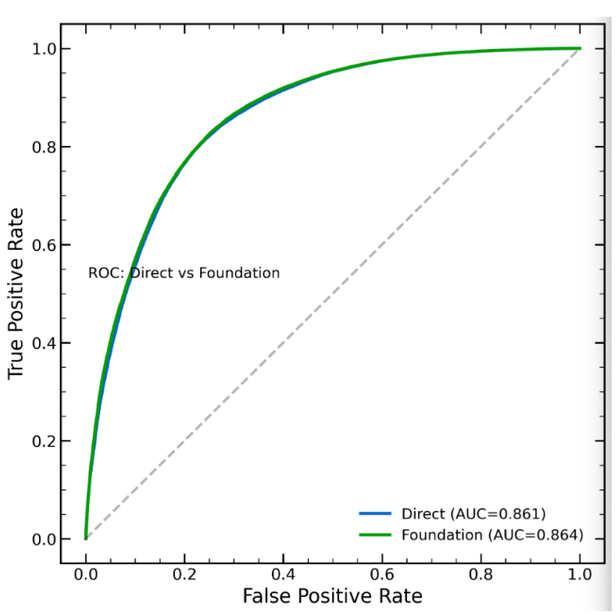
* This keeps the model independent of how the tokens were produced.
* What transformer sees?
  - For each sequence position, the model receives a learned vector. Self-attention then compares all tokens in the event (BNL implementation).

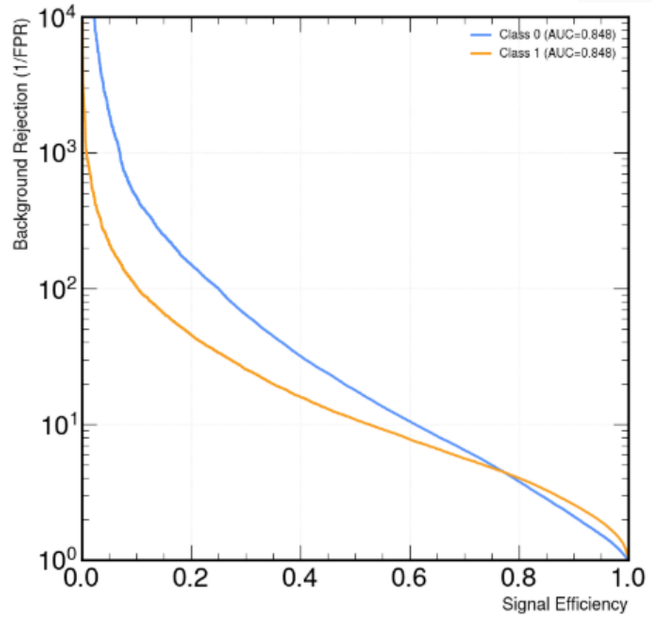


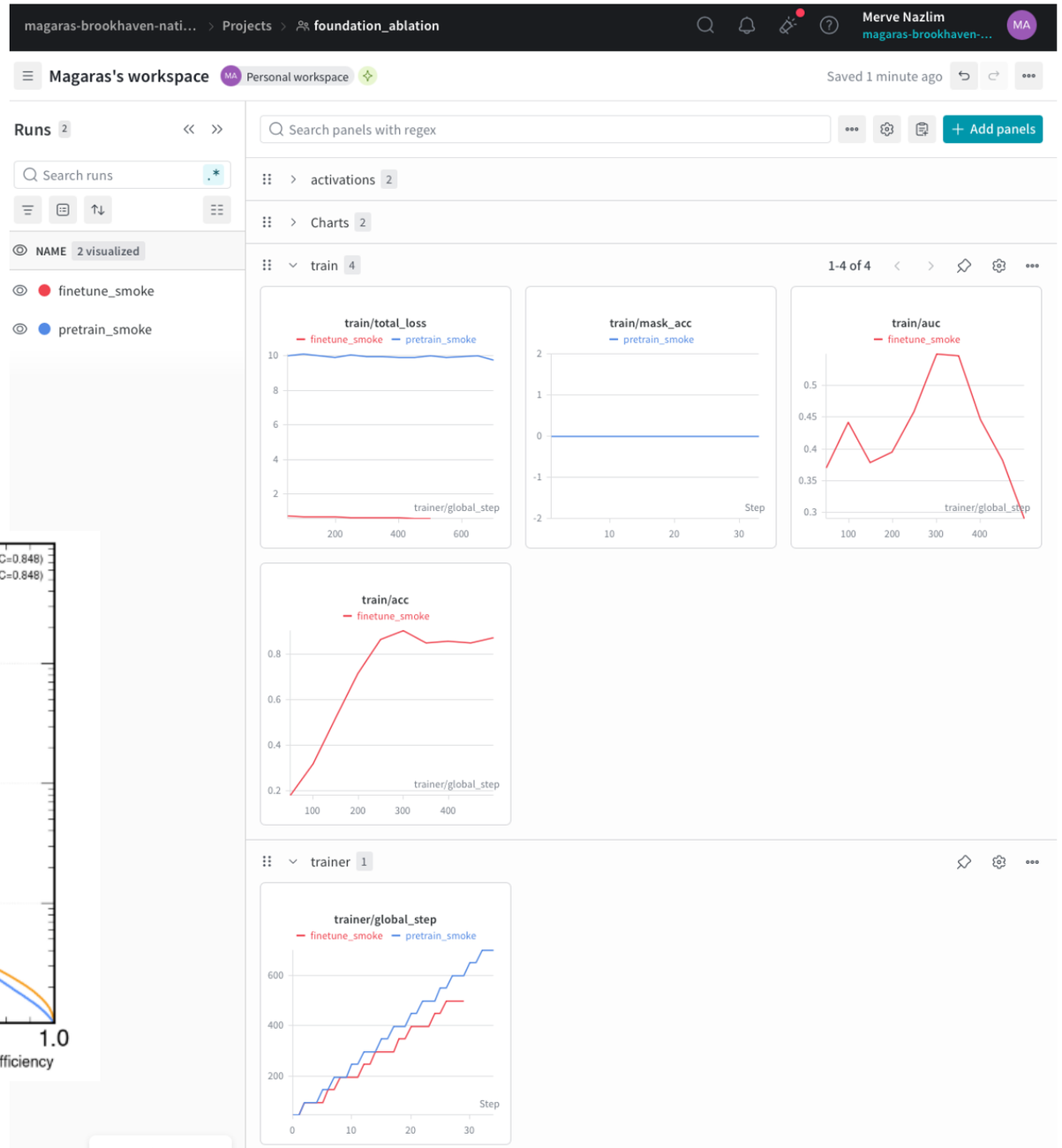
Downstream model from BNL to Heptokens

* A pretraining run successfully started on multiple GPUs.
* The model loaded:
  - ~10.4M tokenized sequences
  - 5.6M model parameters
  - W&B offline logging enabled

BNL vs heptokens







What worked and what remains open

* Working
  - Technical integration is working
  - BNL-style token Parquet files can be read in the new training setup
  - Foundation pretraining and direct classification run through Hydra/Pixi
  - Multi-GPU training with Lightning DDP was tested
  - Offline W&B logging and checkpointing work for long-running jobs
  - Event-level HDF5 loading now supports configurable object collections
* Open
  - Decide heptokens vs BNL repo boundary
  - Validate tokenization quality across object types
  - Run hyperparameter scan
  - Run different setups
    - One vs all tokenisation, context aware etc
  - Check for leakage from sequence length/type IDs
    - Too good results

Comparison

Table 1

Area	BNL-treasure	Heptokens before our work	Heptokens after our work
Main purpose	End-to-end prototype: tokenize H5, write Parquet, train event model	Mainly jet/object tokenization framework	Can now start supporting event-level tokenization studies
Input data	ATLAS/common-style H5 files	Existing jet-style inputs	ATLAS/common-style H5 files through configs
Object support	jets, electrons, muons, photons, taus, tracks, MET, event info	Mostly jet-centered workflow	Multiple object collections configurable
Tokenizer setup	Separate tokenizer per object	Existing tokenizer model, but not good for full event objects	event_object mode supports one tokenizer per object type
Combined event tokenization	Has logic to assemble event token sequences	Not really available	combined mode added as a simple all-object baseline
Separate object outputs	can tokenize objects separately	Not available for event data	object and separate modes added config-wise
Feature def	Hardcoded in Python	Not centralized for event objects	Config-driven object collections
Event-level inputs	mu, pvx/pvy/pvz	Not supported	Yes, still needs cleaner design
Training workflow	Script-based	Hydra/Pixi/Lightning workflow	Hydra/Pixi/Lightning workflow
Checkpoints	Saves tokenizer objects/ checkpoints from scripts	Existing Lightning checkpoint flow	checkpoints saved through heptokens callbacks
Diagnostics	BNL had plotting/check scripts	Limited for new event tokenizer workflow	Added reconstruction, residual, codebook usage, dead-token diagnostics
Downstream foundation model	Included in BNL-style prototype	Not part of heptokens	Yes
Flexibility	Fast to prototype, but more hardcoded, slow	Cleaner package, but missing event pieces	More flexible, but still early and needs some cleanup

Discussion (long term)

* Foundation model direction -> I think we need to agree on what kind of tasks we want in this model, and how flexible we want this to be.
  - Build one event-level model that can help many standard analyses, instead of every analysis training its own classifier from scratch.
  - Start from signal vs background and scale it to multi-class classification
  - Higgs, QCD, Z, W, top, etc.

* Masked modeling vs classification
  - BERT needed? if you already have labels from simulation, then it is not obvious whether masked modeling is better than just supervised classification.
  - Do we want anomaly detection? If yes, what kind of tasks BSM, LLPs etc.

“

Backup

Current h5 Viviana

```
Input event sequence
[CLS] [event] [electron_tok] [muon_tok] [jet_tok] [track_tok] [MET_tok] ...

  │
  ├── Token Embedding Layer
  │       maps token ID → learned vector (dim = d_model)
  │
  ├── Position Embedding Layer
  │       token order / slot index
  │
  ├── Type Embedding Layer
  │       electron / muon / jet / track / MET / special token
  │
  ├── Sum all embeddings
  │       x = token + position + type
  │
  ├── LayerNorm
  ├── Dropout
  │
  ├── (Optional GroupAttentionLayer)
  │       split token embedding into sub-vectors
  │       inner attention
  │       residual (recombine) + norm (transformer)
  │
  │
  └── Transformer Encoder Blocks × N
          repeated N times
```

```
each block:

    ┌──────────────────────────┐
    │ Multi-Head Self Attention │
    └──────────────────────────┘
                 │
          Residual Add
                 │
          LayerNorm
                 │
    ┌──────────────────────────┐
    │ Feed Forward MLP         │
    │ Linear(d→4d)             │
    │ GELU                     │
    │ Linear(4d→d)             │
    └──────────────────────────┘
                 │
          Residual Add
                 │
          LayerNorm

  │
  └── Final hidden states for every token ->pooling

Take hidden state of [CLS]

  │
  └── Classification Head

      Linear(d,d)
      ReLU
      Dropout
      Linear(d,2)
```

Data-layer

* I added a sequence data convention consistent with the repo's token classifier style
* The adapter converts old Parquet columns into the repo-style batch format:
  - input_ids -> tokens
  - attention_mask -> mask
  - token_type_ids -> type_ids
* token_parquet.py (TokenParquetPretrainModule and TokenParquetClassificationModule)
  - read Parquet files
  - find token columns
  - convert them into tensors
  - assign labels for signal/background classification
  - split into train/val/test
  - create PyTorch/Lightning dataloaders
* added Parquet support through pyarrow

Model-layer

* I added:
  - **SequenceBackbone**: shared transformer backbone for token sequences
  - **LitMaskedSequenceModel**: Lightning module for **masked** sequence pretraining (BERT style)
  - **LitSequenceClassifier**: Lightning module for direct classification or fine-tuning from a pretrained backbone
* The backbone reuses the repo's existing transformer building block where possible.
* Existing **TokenClassifier** expects batch["tokens"] and batch["mask"], but it does not do masked-token pretraining.
* Existing **Classifier** is designed around a saved JetBackbone, not a BERT-style masked sequence backbone.
* Existing **Transformer** was reusable, so the new **SequenceBackbone** uses it instead of rewriting transformer logic.
  - Wraps Transformer with missing sequence specific parts token embeddings, type embeddings, position embeddings, optional group-attention logic
  - SequenceBackbone = token IDs -> embeddings -> Transformer -> hidden states

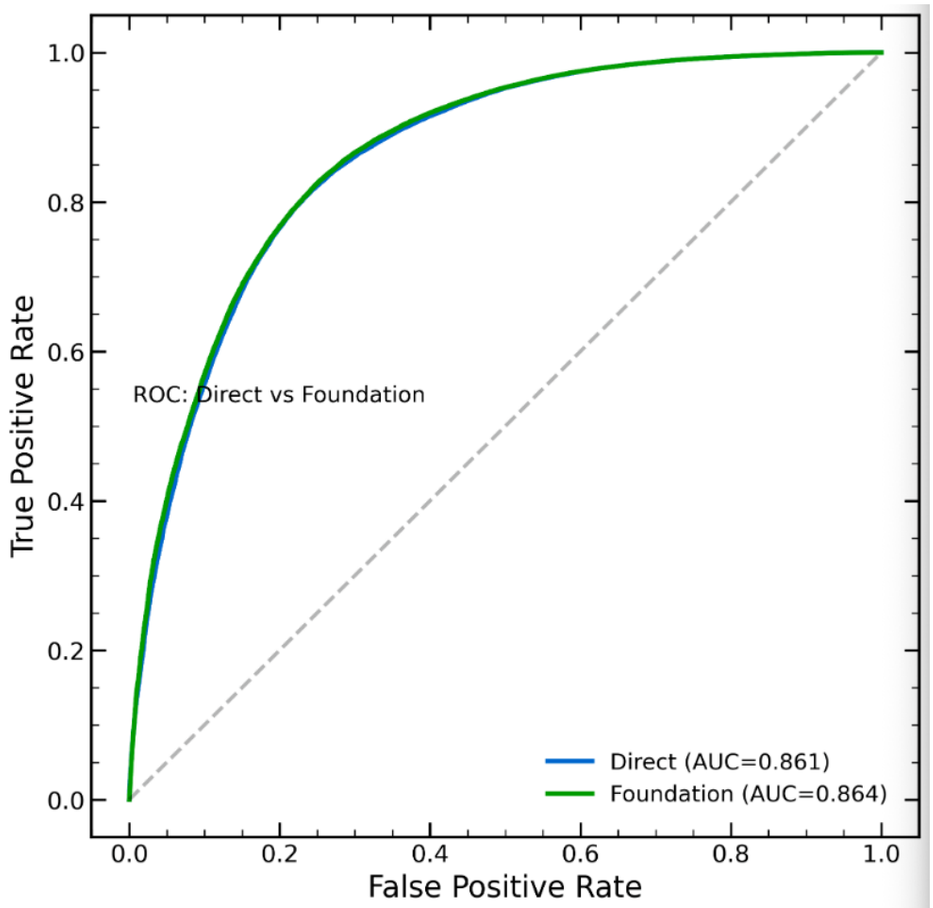
Hydra/Pixi Integration and running the workflow

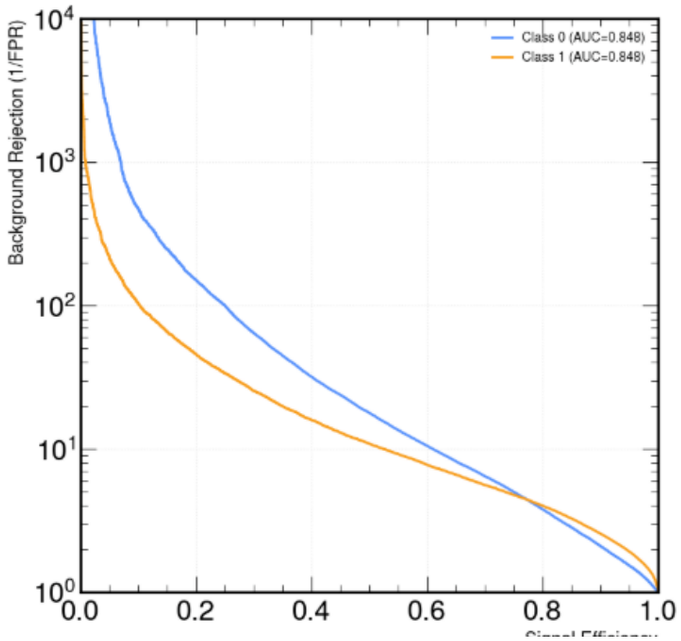
* I added Hydra configs so the workflow runs through the repo's standard entry point:
  - pixi run python scripts/train.py …
  - ```
    configs/datamodule/token_parquet_pretrain.yaml
    configs/datamodule/token_parquet_classification.yaml
    configs/model/foundation_pretrain.yaml
    configs/model/sequence_classifier.yaml
    ```
* Existing callback configs are reused:
  - ```
    callbacks=pretrain
    callbacks=classify
    ```
* The workflow is now split into separate jobs:
  1. Foundation pretraining
  2. Direct classifier baseline
  3. Foundation fine-tuning
* Example pretraining command
  * ```
    pixi run python scripts/train.py \
      datamodule=token_parquet_pretrain \
      model=foundation_pretrain \
      callbacks=pretrain \
      trainer.max_epochs=3
    ```

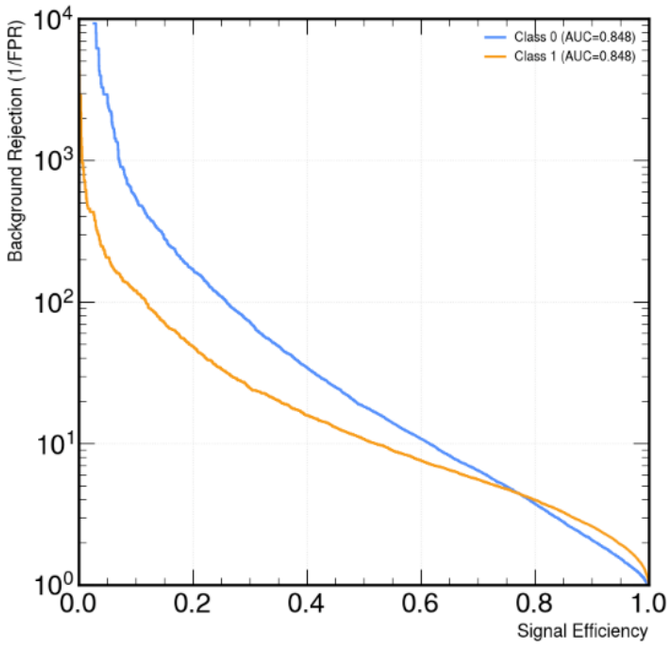
Using multi-GPU training with Lightning DDP
    ```

First Successful Run

- A pretraining run successfully started on multiple GPUs.
- The model loaded:
  - ~10.4M tokenized sequences
  - 5.6M model parameters
  - W&B offline logging enabled







Adapting the code to on the fly tokenisation

* HDF5 event objects
  - -> event tokenizer
  - -> tokens, type_ids, mask
  - -> optional labels
  - -> SequenceBackbone / foundation / classifier

* The models do not care whether the tokens came from Parquet, HDF5, or an online tokenizer.

Classifier

* token embedding
  + type embedding
  + positional embedding
  → Transformer encoder with 2 layers and 4 attention heads
  → take [CLS] token output
  → small MLP classification head
  → binary output: signal vs background

* Classifier head:
  - Linear(hidden_dim, hidden_dim)
  - ReLU
  - Dropout(0.1)
  - Linear(hidden_dim, 2)
* It uses the hidden state of the first token, the `[CLS]` token, for classification.

Z->4l vs background investigation

* This is not the best example
* Tokenisation is poor but the classifier still sees very strong “event shape” information:

```
signal mean sequence length     ≈ 24.5
background mean sequence length ≈ 12.6

signal mean electrons ≈ 1.44
background electrons  ≈ 0.009

signal mean muons ≈ 1.68
background muons  ≈ 0.014

signal mean tracks ≈ 7.68
background tracks  ≈ 1.28

signal mean taus ≈ 1.63
background taus  ≈ 0.26
```

```
tokens
-> token embedding
-> + type embedding
-> + position embedding
-> transformer encoder with self-attention
-> contextual token representations
-> masked-token prediction head
```

* The tokenized input contains more than the VQ-VAE codebook IDs. It also contains:

```
attention_mask    → tells which positions are real tokens
token_type_ids    → tells object type: electron, muon, jet, track, tau, etc.
SEP positions     → indirectly show where each object block ends
sequence length   → strongly different for signal/background
```

Object level Features Used In Tokenisation

Electrons

text

```
pt
eta
phi
charge
trk_iso03
LHLoose
LHMedium
LHTight
```

Muons

text

```
pt
eta
phi
charge
trk_iso03
quality
muonType
```

Taus

text

```
pt
eta
phi
charge
is_1prong
NNDecayMode
RNNJetScore
RNNEleScore
```

Photons

text

```
pt
eta
phi
trk_iso03
isLoose
isTight
```

Jets

text

```
pt
eta
phi
mass
n_trk
QG_nTracks
QG_tracksWidth
QG_tracksC1
DL1d_pb
DL1d_pc
DL1d_pu
GN2_pb
GN2_pc
GN2_pu
```

Tracks

text

```
pt
eta
phi
d0
z0
qOverP
chiSquared
nDoF
```

MET

text

```
met / pt
phi
sumet
```

Event level Features Used In Tokenisation

* Only uses mu
* Should we add primary vertex info?
* Should we add cross section, counts for different type of particles?
* Preselection region or signal region trainings?
  - Current tokenization is bad because we have many low pt jets but not high
* How to or should we implement the event weights in some way?

Object vs event level

* Object-level features go through the object tokenizer/VQ-VAE path:
  - ```
    object features like pt, eta, phi, charge, b-tag scores, ...
    -> normalized feature vector
    -> VQ/tokenizer code
    -> object token ID
    ```
* But event-level mu is much simpler:
  - ```
    mu
    -> clipped/binned integer
    -> event token ID
    ```
* [CLS] [event_mu_token] electron tokens [SEP] muon tokens [SEP] jet tokens ... MET token

What The Tokenizer Saves To Parquet

* The tokenizer does not save the raw features. It converts them into token sequences and writes:

```
input_ids
token_type_ids
attention_mask
event_index
source_file
```

* In the new repo we map these to:

```
input_ids        -> tokens
token_type_ids   -> type_ids
attention_mask   -> mask
```

* What The Transformer Sees

```
tokens
type_ids
mask
```

* So for each sequence position, the transformer gets:

```
token ID embedding
+ object/type embedding
+ position embedding
```

* Then self-attention learns relationships between tokens.

* Extra Sequence Information

```
CLS token
SEP tokens between object groups
PAD tokens
MASK token during pretraining
```

* And object type IDs:

```
electron -> type id 4
muon     -> type id 5
jet      -> type id 6
met      -> type id 7
photon   -> type id 8
track    -> type id 9
event    -> type id 10
tau      -> type id 11
```

Do the tokenization and model can be generalised?

1. Fit tokenizer once using mixed / generic / independent events

2. Use that same tokenizer to create:

   signal parquet

   background_zz parquet

   background_other parquet

3. Train classifier:

   signal vs background_zz

4. Test classifier:

   signal vs background_other

Tokenisation

Tokenisation checks

◯ Baseline

VQ-VAE per object
codebook_size = 512
latent_dim = 64
hidden_dim = 128

- Make reconstruction plots **per object type**:

jets → pT, eta, phi, mass
electrons → pT, eta, phi
muons → pT, eta, phi
photons → pT, eta, phi
MET → pT, phi
Plots:

- original vs reconstructed distributions
- residuals (reco – original)
- correlation (scatter)

- Check **codebook usage per object**:

how many tokens are used
distribution of token frequencies
dead tokens?

Goal:

- detect collapse
- detect under-utilisation

◯ Scan axes (ordered by importance)

- Codebook size [512, 2048, 8192, 32768]
- Number of residual quantizers [1, 2, 4]
- Encoder type ["mlp", "transformer"]
- Latent dimension [32, 64, 128]
- Hidden dimension [128, 256]
- Codebook dimension fixed = 8

Starter scan (~12 configs)
codebook_size = [512, 8192, 32768]
num_quantizers = [1, 2, 4]
encoder = ["mlp", "transformer"]
latent_dim = [64, 128, 256]

reco – original pT
reco – original eta
reco – original phi
mass resolution for jets
mll / m4l resolution for events
codebook usage
classifier AUC
training time

What to log for EACH config

A. Reconstruction

- pT resolution
- eta, phi distributions
- mass resolution (jets)
- original vs reconstructed distributions
- residuals (reco – original)
- correlation (scatter)

B. Codebook usage

- % of tokens used
- histogram of token frequency
- dead tokens

C. Downstream performance

- classifier AUC
- accuracy

D. Efficiency

- training time
- tokenization speed