RESEARCH

Jet Energy Calibration with Deep Learning as a Kubeflow Pipeline

Daniel Holmberg[1] · Dejan Golubovic[2] · Henning Kirschenmann[3]

Abstract

Precise measurements of the energy of jets emerging from particle collisions at the LHC are essential for a vast majority of physics searches at the CMS experiment. In this study, we leverage well-established deep learning models for point clouds and CMS open data to improve the energy calibration of particle jets. To enable production-ready machine learning based jet energy calibration an end-to-end pipeline is built on the Kubeflow cloud platform. The pipeline allowed us to scale up our hyperparameter tuning experiments on cloud resources, and serve optimal models as REST endpoints. We present the results of the parameter tuning process and analyze the performance of the served models in terms of inference time and overhead, providing insights for future work in this direction. The study also demonstrates improvements in both flavor dependence and resolution of the energy response when compared to the standard jet energy corrections baseline.

Keywords LHC · CMS · Open Data · Jet energy · Kubeflow · MLOps · GNN

Introduction

The adoption of machine learning methods has had a profound impact on the field of high energy physics, greatly increasing the discovery potential in the data measured by the particle detectors at the Large Hadron Collider (LHC) [1]. Deep learning especially has proven very useful [2] with graph neural networks being one of the most expressive and versatile architectures to choose for many tasks [3] ranging from reconstructing particle tracks [4] to classifying complete events [5].

In this paper, we study the application of deep learning for calibrating the energy of particle jets at the Compact Muon Solenoid (CMS) experiment [6]. Jets in this context originate from high energy proton–proton collisions producing color charged partons that undergo hadronization forming collimated sprays of color neutral particles. Calibrating the energy of jets is an involved process, split into several factorized steps, some based only on simulations and some on comparisons with data [7]. A precise calibration of the jet energy scale is crucial for a wide variety of physics analyses, most prominently, e.g., measurements of the top quark mass [8] and inclusive jet cross-section measurements.

Deep learning has been successfully applied by the CMS collaboration in the past to increase the energy resolution of bottom jets with a feedforward neural network [9]. These efforts are here extended to all jet flavors in a QCD-jet data sample publicly accessible on the CERN OpenData portal [10]. Additionally, by adopting recent advancements in representation learning, specifically ones made for jet classification [11, 12], more information about jet constituents can be included in the training process.

Furthermore, since operationalizing machine learning workflows is a challenge in itself for many organizations [13], we introduce a cloud native pipeline for running jet energy calibration experiments. It runs on the Kubeflow platform [14] that comes with readily available components for hyperparameter tuning and model serving among others. Kubeflow has been used by researchers in various domains such as bioinformatics to achieve rapid scaling with containers [15], or as a means to create automated machine learning workflows for a service-aware 5 G network model adapting to drift in the input data [16]. Adopting a cloud native

✉ Daniel Holmberg
  daniel.holmberg@helsinki.fi

  Dejan Golubovic
  dejan.golubovic@cern.ch

  Henning Kirschenmann
  henning.kirschenmann@cern.ch

[1] Department of Computer Science, University of Helsinki,
   00560 Helsinki, Finland

[2] CERN, 1211 Geneva 23, Switzerland

[3] Helsinki Institute of Physics, 00560 Helsinki, Finland

workflow enables the workload to be smoothly deployed also on public cloud resources, as was done for fast simulation of electromagnetic showers using generative deep learning at CERN [17].

The rest of this paper is structured as follows. In the "Dataset Definition and Conventional Jet Energy Calibration" section, we explain the contents of the CMS open dataset used for this study. The "Jet Energy Regression" section provides an overview of the data distribution, feature sets and models used to calibrate jet energy. In the "Kubeflow Pipeline" section we introduce the Kubeflow pipeline used for training and serving our models on internal cloud resources. The "Results" section presents the results that our models yield, and lastly in the "Conclusion" section we offer our final thoughts and takeaways.

Dataset Definition and Conventional Jet Energy Calibration

In this study, we utilize a dataset prepared in the context of the CMS OpenData effort [10]. The dataset consists of particle jets extracted from simulated proton–proton (pp) collision events at $\sqrt{s} = 13$ TeV. These events are generated at leading-order perturbative QCD with PYTHIA 8 [18], and include CMS detector simulation and event reconstruction.

"Particle-level jets" or "generator jets" are clustered using the anti-k_T algorithm [19] with radius parameter of 0.4 from stable (decay length $c\tau > 1$ cm) final-state particles resulting from the hadronization of partons originating from the pp collisions. As these particles propagate through the detector, they leave signals in detector components such as the tracker and the electromagnetic and hadronic calorimeter. The modeling of the interaction with the material and detector response relies on the CMS Full simulation, based on GEANT4 [20].

The types of quarks or gluons that initiate the formation of the jet determines the flavor of a jet. Flavor labeling is done using a technique called "ghost association" [21], where heavy flavor hadrons and light quark and gluon partons are added as "ghost particles" to the clustering process. If heavy flavor hadron ghosts are found in the jet, it is labeled as a heavy flavor (b or c) jet. If no ghost hadron is found, the jet is checked for light flavor (uds) or gluon (g) partons, identifying it as a corresponding jet [22].

The Particle Flow (PF) approach at CMS [23] is a method that attempts to reconstruct each particle in the event individually, prior to the jet clustering, based on information from all relevant sub-detectors, resulting in a list of "PF candidates". Various methods for per-particle pileup (additional pp collisions occurring in the same bunch crossing as the event of interest) mitigation can be applied [24]. The charged hadron subtraction (CHS) is the default for narrow

radius jets in Run 2 CMS data, meaning that charged particles associated with pileup vertices are removed prior to jet clustering, thereby reducing the impact of pileup on jets. The remaining reconstructed PF candidates are then used as input to jet clustering algorithms to form a "reconstructed jet" (anti-k_T, $R = 0.4$) that is supposed to be as close to the particle-level jet in terms of kinematic quantities as possible.

The aim of jet energy corrections is to correct the energy of the reconstructed jets—on average—back to the jet energy of particle-level jets. The measured jet energy is affected by various effects, such as energy loss in the detector material, non-linear response of the detector, and pileup. CMS factorizes the jet energy corrections into levels to individually correct for various effects [7]: the L1 correction corrects for offset energy induced by pileup. It is determined from simulated samples with/without pileup, parametrizing the offset energy as a function of median event energy density, jet area, p_T, and η. The L2L3 correction corrects for remaining detector response dependence and is parametrized as a function of jet p_T and η. The L2L3Residual correction corrects for any remaining differences between experimental data and simulation, but is not applicable in this MC-only study.

These conventional jet energy corrections do not take into account the substructure of the jets. The PF approach already leads to a reduced dependence of, e.g., the jet response on the flavor of a jet in comparison to calorimeter-only reconstruction. However, taking the substructure of jets into account promises potential for reduced flavor response differences and associated uncertainties as well as an improved jet-energy resolution.

Jet Energy Regression

Standard jet energy corrections can be further improved upon by using supervised machine learning. An important step in this direction has been taken by the CMS Collaboration for b jets specifically [9]. For these jets a significant part of the jet energy is carried by semileptonically decaying b-hadrons, decaying to charged leptons and neutrinos. Neutrinos escape detection by the CMS detector since they only interact via the weak force leading to jet energy being underestimated. However, using a neural network trained on a sample of simulated top quarks event decaying into b jets and W bosons significant improvements in the energy resolution for b jets were achieved.

These efforts can be generalized to other jet flavors too. Using a QCD sample such as the one described in the "Dataset Definition and Conventional Jet Energy Calibration" section enables the training of a regression model for multiple jet flavors. This approach can potentially address any flavor discrepancies in the energy response. Notably, the response of light quark jets and gluon jets can vary significantly due

to the higher color charge of gluon jets, which results in more and softer particles with a lower calorimeter response. Although the PF algorithm can reduce the response difference substantially by replacing the non-linear calorimeter measurement of charged hadron energy with the corresponding track momentum, 15% of jet energy is still carried by neutral hadrons subject to the calorimeter response non-linearity [23].

Data Distribution and Input Features

The jets in the CMS open dataset span a large spectrum both in terms of p_T and pseudorapidity η as seen in Fig. 1. However, the very low p_T region experiences high pileup, resulting in lower-quality jets. Additionally, the forward region of the detector lacks tracking information leading to a worse event reconstruction quality and less reliable measurements to train on. To address these issues, the dataset is filtered by $p_T^{gen} > 20$ GeV and $|\eta| < 2.5$. In total 1.42M jets were used, 60% of which were allocated to the training set. The model was validated at the end of every training epoch on a separate set with 20% of all jets. Once the training had finished, model performance was evaluated on a test set with the remaining 20% of all jets.

Table 1 presents the features used for training our regression model selected from a comprehensive list of variables available in the CMS open dataset [10]. Eight of the features describe jets as a whole. The reconstructed p_T is log-transformed to reduce the width of the otherwise long-tailed distribution. In addition to p_T, jet coordinates η and ϕ as well as jet mass and catchment area are also included in the set of training features. The last three jet-level variables $p_T D$,
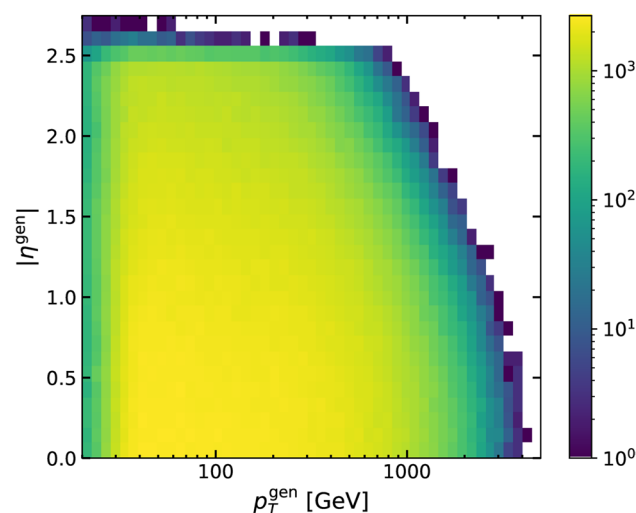
Table 1 Overview of the input features used for training the regression model, categorized into jet-level features and PF candidate features

Category	Variable	Description
Jet features	$\log p_T$	Logarithm of a jet's p_T
	η	Pseudorapidity of a jet
	ϕ	Azimuthal angle of a jet
	m	Mass of a jet
	A	Catchment area of a jet
	$p_T D$	Fragmentation distribution for a jet's p_T
	σ_2	Minor ellipse axis of a jet
	Multiplicity	Jet constituent multiplicity
PF features	$\log p_{T_i}$	Logarithm of a particle's p_T
	$\log \frac{p_{T_i}}{p_T}$	Logarithm of the fractional p_T of a particle
	η_i	Pseudorapidity of a particle
	ϕ_i	Azimuthal angle of a particle
	θ_i	Polar angle of a particle
	R_i	Distance from a particle to the center of the jet

Jet features describe the overall properties of the jet, while PF features characterize the individual particles within the jet

σ_2 and multiplicity can help to discriminate between quark and gluon jets [25].

Every particle within a jet has six associated features. The two most statistically significant for the regression task at hand are: (1) the log-transformed p_{T_i} of a PF candidate indexed with i, and (2) the same variable, but relative to the p_T of the whole jet, also log-transformed. Additionally, each PF candidate has four location-based features: the η_i, ϕ_i and θ_i detector coordinates, along with the distance R_i from the particle to the center of the jet.

Regression Target and Loss Function

The aim of the regression is to train a model to map a set of features describing a reconstructed jet towards the transverse momentum of the corresponding generator-level jet. However, by itself the particle-level p_T^{gen} follows a decreasing exponential distribution that covers many orders of magnitude in the energy spectrum as seen in Fig. 1. To counteract this, p_T^{gen} can be divided by the similarly distributed p_T^{reco} that is part of the training set. This gives a target distribution on the order of one with a reduced variance compared to the original target distribution [9]. To further correlate the target with the input features the logarithm is taken yielding the final regression target as $y = \log\left(p_T^{gen}/p_T^{reco}\right)$. The distribution of the target is narrow and centralized around zero as seen in Fig. 2. In order to get the corrected transverse momentum p_T^{corr}, the exponential function is applied to the prediction $\hat{y}$, and the resulting correction factor is multiplied
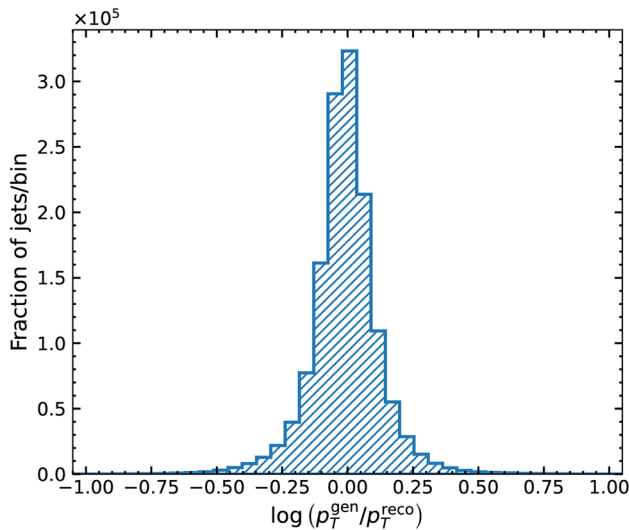

Fig. 1 A heatmap illustrating the distribution of jets in the dataset with respect to the generated transverse momentum (p_T^{gen}) and the absolute value of generated pseudorapidity ($|\eta^{gen}|$)

Fig. 2 Distribution of the regression target

by $p_{\mathrm{T}}^{\mathrm{reco}}$. The per-jet energy response can then be defined as $R = p_{\mathrm{T}}^{\mathrm{corr}}/p_{\mathrm{T}}^{\mathrm{gen}}$.

The mean absolute error (MAE) is selected as the loss function to minimize for this problem, as it assigns less importance to outliers compared to the more commonly used mean squared error (MSE) loss. In line with the previous study on b jet energy regression [9], a loss function with reduced sensitivity to the tails of the target distribution is preferred. To prevent potential spikes in the training loss, jets with a target value smaller than -1 or larger than 1 are excluded, as they are considered too poorly reconstructed to be taken into account. The loss function used is thus:

$$L = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i| I_{|y_i| < 1}. \tag{1}$$

A further motivation for the use of the MAE loss function is that the statistic it learns is the median of the target distribution, whereas for example the minimum of the MSE loss lies on the function that maps input features to the expected value of the target. Predicting the median in this case can be seen as beneficial since it is a robust measure of central tendency. By learning the median of the target distribution, the model is more likely to make accurate predictions even in the presence of outliers.

Deep Learning Models

During recent years many new approaches of applying deep learning in jet physics have emerged, especially for the purpose of jet tagging where the aim is to classify jets based on the particle initiating them. Some proposed approaches to do this is for example to treat the jets as images [26], sequences [27] or trees [28]. While these methods perform well and surpass traditional multivariate methods, the way they represent jet constituents is not ideal. A particle jet can contain up to $\mathcal{O}(100)$ particles, whereas an image of a jet will contain $\mathcal{O}(1000)$ pixels, and as noted in [26] the images are indeed very sparse with 5–10% of pixels being active. When instead considering a sequence or tree of particles as the representation a notable issue that arises is that the particles must be ordered in some fashion to be used by the deep learning model, e.g., a recurrent neural network or recursive neural network. However, the constituents of a particle jet have no intrinsic ordering.

More natural ways of representing particles have been found by adopting point cloud-based formalisms from the wider machine learning community as particle clouds [12]. This kind of representation treats a collection of particles as a graph structure $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each individual particle serves as a node $\mathcal{V} = \{1, \ldots, n\}$ with potential edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ connecting them. In this work, we compare two separate models that operate on data represented in this way.

The simplest case of representation learning on particle clouds is when the set of edges is empty $\mathcal{E} = \varnothing$. This will lead to the particles taking on the form of an unordered set, and was originally proposed for the Particle Flow Network (PFN) model [11] adapting from the Deep Sets [29] framework. The key idea here is that input feature vectors $\mathbf{x}_i \in \mathbb{R}^F$ are mapped with an equivariant function into a latent feature vector $\mathbf{h}_i = \psi(\mathbf{x}_i)$. In practice that would be a multilayer perceptron (MLP) with shared weights for all elements of the set.

To make predictions for the particle jet as a whole the latent feature vectors must be aggregated using a permutation invariant pooling, such as summing, averaging or taking the max value. Following the PFN and Deep Sets papers, summation $\sum_{i \in \mathcal{V}}$ is chosen as the global pooling operation. Figure 3b in conjunction with Fig. 3c shows the complete network architecture where the output of the Deep Sets block connects to global pooling in the network head. The global particle representation is concatenated with jet features before being passed into a final MLP mapping towards the regression target. Note that the rectified linear unit (ReLU) [30] is used as activation function, and dropout [31] is applied in the head of the network.

Spatial information can be used to further increase the expressivity of a point cloud-based model. The ParticleNet [12] architecture uses edge convolution (EdgeConv), first introduced as a building block of dynamic graph convolutional neural networks (DGCNN) [32], to incorporate information on the local neighborhood of each particle. Detector coordinates in the (η, ϕ)-plane are used to calculate the Euclidean distance matrix from pairwise distances between particles. The k-NN algorithm is then applied to
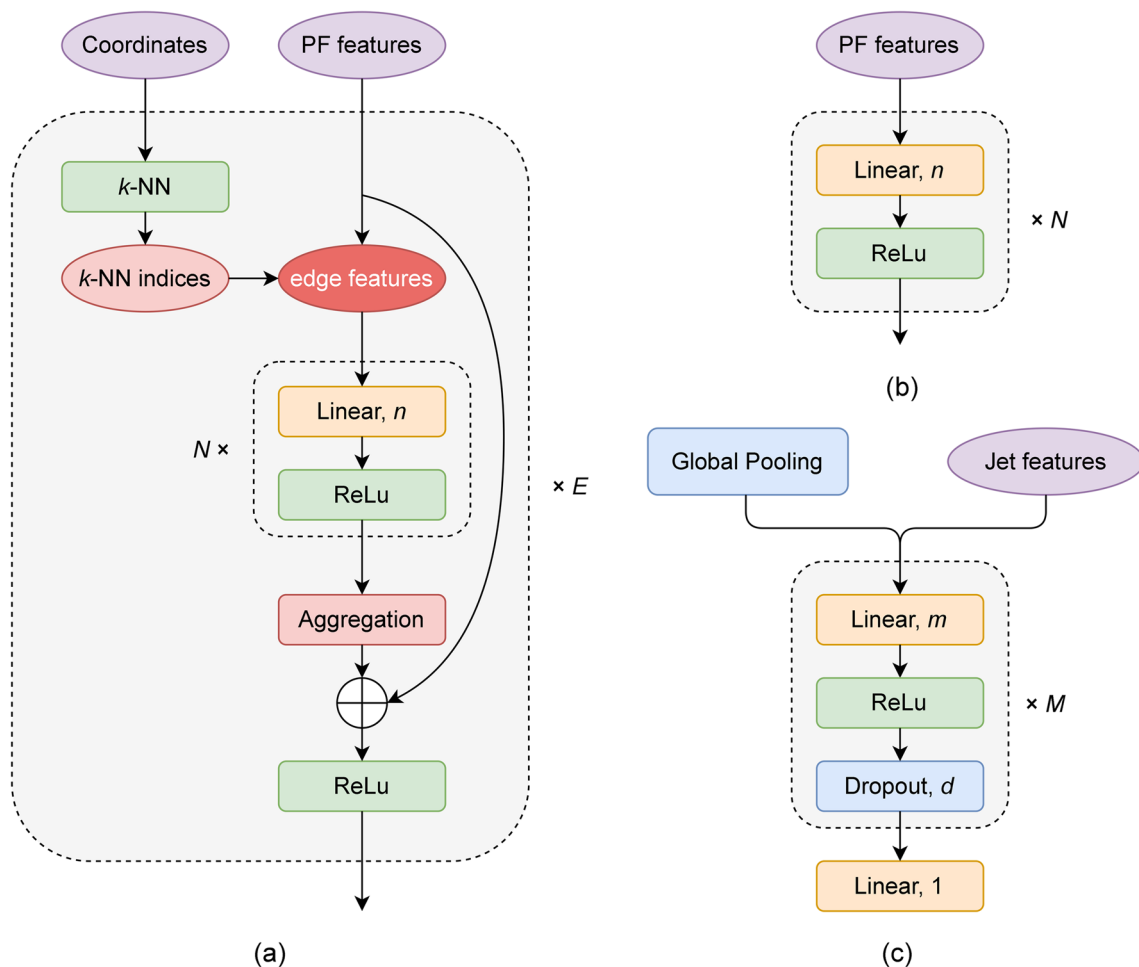
Fig. 3 Illustration of model architectures: **a** EdgeConv block for ParticleNet, **b** Deep Sets block for PFN, and **c** network head shared by both models

construct edges connecting each particle to its k nearest neighboring particles.

Messages between every point $\mathbf{x}_i$ and its neighbors $\mathbf{x}_j$ are learned using an asymmetric edge function $\psi(\mathbf{x}_i, \mathbf{x}_j - \mathbf{x}_i)$ implemented as an MLP with shared weights. Permutation invariant aggregation in the form of averaging $\frac{1}{k} \sum_{i \in \mathcal{N}_i^k}$ over the learned edge features for the k nearest neighbors is used to update every node in the particle cloud. A shortcut connection [33] from the original node features is added to the output of the aggregation before passed being passed through the ReLU activation function. This concludes the EdgeConv block shown in Fig. 3a.

If multiple EdgeConv blocks are stacked after one another the input graphs are dynamically updated by calculating the pairwise distance matrix from the latent feature space learned by the previous block. Global average pooling $\frac{1}{n} \sum_{i \in \mathcal{V}}$ is applied on the output of the last EdgeConv block as it is passed into the network head in Fig. 3c. An identical procedure to that in the PFN model is applied, where the jet features are concatenated with the pooled particle features, and finally passed through one last MLP mapping towards the regression target.

The models are implemented in PyTorch [34] as part of the weaver deep learning framework for high energy physics [35]. Weaver supports handling common particle physics data formats such as ROOT [36] or Awkward Array [37], as well as distributed training, and model inference. To scale up the training on cluster resources where GPUs are distributed over separate nodes the code must support collective communications to synchronize gradients over machines. Different backends can be chosen in PyTorch for this purpose such as Message Passing Interface (MPI) [38] for CPU parallelization, or NVIDIA Collective Communication Library (NCCL) [39] for multi-GPU communication.

Kubeflow Pipeline

The analysis was carried out on the Kubeflow-based machine learning platform at CERN [14]. Kubeflow is an open-source machine learning toolkit that supports the entire machine learning lifecycle by providing features such as notebooks, pipelines, hyperparameter optimization, distributed training, model serving, and model monitoring. Kubeflow is built on top of Kubernetes [40], a container orchestrator, leveraging the scalability, ease of use and integration of cutting-edge infrastructure technologies. Deployed as a set of Kubernetes resources, Kubeflow application code runs as a collection of micro-services that communicate with each other and process user workloads.

Machine learning workflows typically take the form of a directed acyclic graph that begins with data processing, proceeds through model training, and ends with an inference phase of the trained model. Kubeflow facilitates developing such workflows by offering several features, including a web interface for managing, tracking, and running pipelines, an engine for scheduling pipeline steps, a software development kit (SDK) for defining and running pipelines using Python, and higher-level abstraction tools like KALE [41], which convert notebooks to pipelines. Regardless of how a pipeline is defined, it is always converted to a Kubernetes YAML [42] definition file before being submitted for execution. A pipeline runs as a sequence of pods (containers), with each pipeline step waiting for its dependencies to complete successfully before proceeding. The pipeline developed for this study is shown as part of the Kubeflow user interface (UI) in Fig. 4.

Pipelines offer benefits in resource utilization. By allowing users to define hardware requirements for each step, pipelines ensure that GPUs are only utilized when needed, for example during training and inference steps. Other steps that use CPU-only resources make GPUs available for other users in the cluster. Additional features include support for pipeline scheduling which enables automatic execution of repeated or periodic workflows, running pipelines with different input parameters without any code changes, and grouping pipeline runs into experiments making it easier to track and compare similar runs.

AutoML Experiment

The Kubeflow Katib component [43] offers a streamlined process for automated machine learning (AutoML) supporting hyperparameter tuning, early stopping and neural architecture search. Here we use hyperparameter optimization that includes three main steps: (1) implementing a script that trains a model and takes hyperparameters as command line arguments, (2) building a docker image with all dependencies to run the script, and (3) specifying a YAML file with the definition of the hyperparameters. The YAML file defines the search algorithm, early stopping options, maximum number of parallel jobs, hardware resources, and other options. The YAML file can be generated manually, using an SDK, or using the high-level KALE tool.

Katib schedules search jobs in the form of multiple trials, each trial corresponding to a unique combination of

Fig. 4 Kubeflow UI for a jet energy regression pipeline run. The pipeline consists of three steps: (1) hyperparameter tuning using Kubeflow's AutoML component Katib, (2) exporting the optimal PyTorch model to the ONNX format, and (3) serve the exported model over HTTP with KServe

hyperparameters as seen in Fig. 5. A trial can be a Kubernetes job running a script in a single pod, a pipeline where multiple pods run sequentially or a distributed training job where multiple pods run synchronously utilizing multiple GPUs to train the model. Being framework-agnostic, Kubeflow supports running search jobs with any machine learning framework.

In this study, Katib trials were executed using the PyTorchJob Kubernetes custom resource [44]. The Kubeflow training operator component provides distribution on the level of containerization; training a single model using multiple cluster GPUs that can be located in different machines. In addition to implementing the training to use distribution strategies, it is necessary to specify a YAML definition of a distributed job. The YAML definition includes many attributes, including the number of worker replicas, memory, and command line arguments to the machine learning code. The PyTorchJob also includes S3 credentials to access the training data stored in a bucket on CERN object storage.

To optimize the performance of ParticleNet and PFN, Katib was configured to use the Random Search algorithm [45] to find the optimal set of parameters for minimizing the test loss. The total number of trials was set to 60 for both

Fig. 5 The structure of an Experiment Kubernetes custom resource. An AutoML experiment consists of multiple trials, with each trial representing a unique combination of hyperparameters. Each trial monitors a PyTorchJob, which can submit one or more workers to train the model. Once completed, the Experiment resource retains the outcomes of all trials and the optimal trial result based on predefined metrics

models with 10 trials running in parallel using one GPU each to achieve fast scheduling of pods on the cluster. Each trial ran for 50 epochs, with the batch size set to 500. The initial learning rate is set as part of the hyperparameter optimization process. After 70% of epochs, 35 in this case, a scheduler starts decreasing the learning rate exponentially on a per-epoch basis down to 1% of the initial value at the end of training. The model with the lowest validation loss is chosen as the final model for each trial, and is then run on the evaluation set to get the test loss that we are trying to minimize.

With reference to Fig. 3b, c, the search space for the hyperparameter tuning was defined as follows for PFN:

– linear layers: $N \in \{1, 2, 3, 4, 5\}$
– linear layer units: $n \in \{50, 100, 200, 400\}$
– linear layers: $M \in \{1, 2, 3, 4, 5\}$
– linear layer units: $m \in \{50, 100, 200, 400\}$.

The search space for ParticleNet, with respect to Fig. 3a, c, was defined as:

– EdgeConv blocks: $E \in \{1, 2, 3\}$
– nearest neighbors: $k \in \{4, 8, 16\}$
– linear layers: $N \in \{1, 2, 3\}$
– linear layer units: $n \in \{50, 100, 200\}$
– linear layers: $M \in \{1, 2, 3\}$
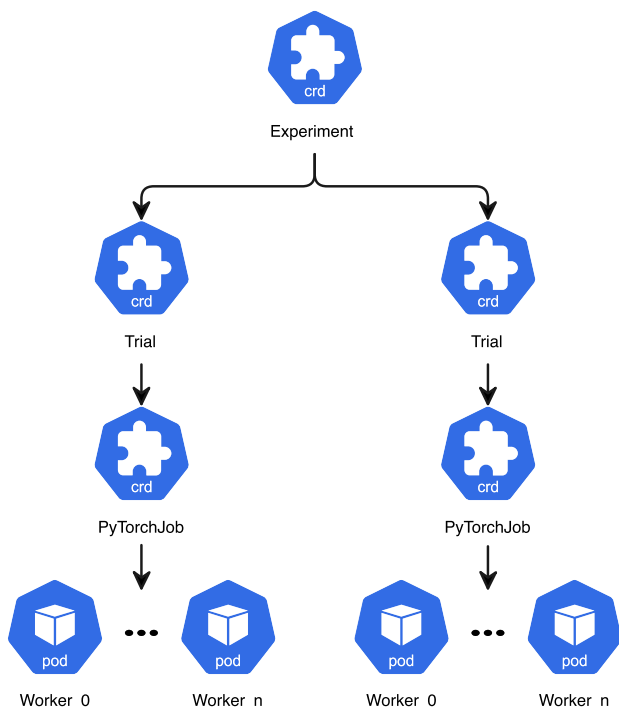– linear layer units: $m \in \{50, 100, 200\}$.

Lastly, the mutual hyperparameters considered were:

– dropout rate: $d \in [0; 0.5]$
– initial learning rate: $\text{lr} \in [10^{-5}; 10^{-2}]$
– optimizer: $\text{optim} \in \{\text{AdaGrad, Adam, AdamW, Ranger, RMSProp}\}$.

The completion of trials can be viewed directly from the Katib UI, while Kubeflow's TensorBoard component allows tracking the progress of individual training runs. To set up a TensorBoard server, it is necessary to specify a model output path, either on a persistent volume claim (PVC) within the cluster or an S3 object storage endpoint. As long as the model training is writing to the specified location, the model performance can be monitored in real-time using TensorBoard servers.

Exporting the Optimal Model

Once the optimal hyperparameters for both PFN and ParticleNet models have been determined through the hyperparameter tuning process, the optimal PyTorch model is exported to the ML framework-agnostic Open Neural Network Exchange (ONNX) format [46]. This

enables seamless integration with other ML tools, such as NVIDIA Triton Inference Server [47] for model serving, and eases the deployment process.

The second step of the Kubeflow pipeline involves running a PyTorchJob to carry out this conversion. It retrieves the optimal PFN and ParticleNet models from the S3 bucket, converts them to ONNX format, and stores the resulting ONNX models back into the same S3 bucket. The PyTorchJob can be defined using a YAML file that specifies the necessary configurations, such as the PyTorch model input path and the ONNX model output path, network configuration, and hardware resources to run the export job.

When exporting a model to ONNX, a configuration file is created alongside the model file to facilitate serving the model using Triton. We made a schema in the Protocol Buffers (protobuf) [48] format with model input/output dimensions, data types (32-bit float) and graph optimization level for ONNX Runtime. This was compiled into a Python file that can be used to automatically generate model configuration in protobuf text format with the correct input and output dimensions when exporting a model in PyTorch.

ONNX Runtime defines a static computational graph for the model as opposed to the dynamic one used by PyTorch during training, which allows for various graph optimizations that can improve inference performance, such as graph-level transformations, node eliminations, node fusions, and layout optimizations. An extended optimization level is available that enables complex node fusions. However, these optimizations were found to cause issues when serving ParticleNet, and as a result they were only applied to PFN. A more basic graph optimization level with semantics-preserving graph rewrites that removes redundant nodes and redundant computation was chosen for ParticleNet.

To allow Triton to accept dynamically varying batch sizes, we configured the maximum batch size to be 100k in the model configuration file. Batch requests that large are not necessarily recommended due to the spiky network load they would produce and the excessive amount of memory that must be allocated to the inference server.

The export job produces an output directory structure, as shown in Fig. 6, that follows Triton's specifications. The base model repository is in our case an S3 bucket path and a unique id for every pipeline run. The top-level repository can contain many subdirectories (or pseudo-folders since object storage has a flat address space), each representing distinct models. The optimal ONNX model is placed in a numeric sub-folder signifying model version during exportation, and the automatically generated model configuration file is placed alongside that folder.

```
<model-repository-path>/
└── <model-name>/
    ├── config.pbtxt
    └── <version>/
        └── model.onnx
    ├── ...
└── ...
```
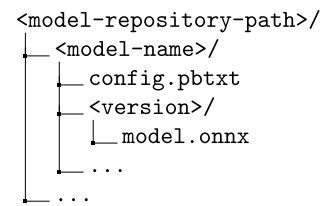
Fig. 6 Model repository layout for Triton Inference Server with ONNX backend

Model Serving

After exporting the optimal PFN and ParticleNet PyTorch models to the ONNX format and storing them in an S3 bucket, the models are served using custom InferenceService resources. An InferenceService is the interface used for deploying models on Kubeflow's inference platform KServe [49].

The InferenceService can be specified using a YAML file with various configuration options, such as hardware allocation for the server (CPU, GPU and memory), runtime version of the predictor, and the path to the model repository. In order to make authenticated requests to S3 storage, a Kubernetes ServiceAccount with the required access rights was deployed on the cluster and attached to every InferenceService.

KServe relies on Knative [50] for scaling serverless workloads and supports scale-to-zero, optimizing cost efficiency. Istio [51], another key technology in KServe, acts as a service mesh that uses Kubernetes sidecars (containers deployed alongside a main container in a pod) for network traffic management, providing features such as progressive "canary" model rollouts, traffic routing, ingress management, logging, load balancing, and security.

Triton is utilized as the predictor for the InferenceService. It is an open-source inference server capable of serving multiple models concurrently, supporting various machine learning frameworks. We have specified the ONNX Runtime as platform in the model configuration file telling Triton explicitly which backend to use.

The complete inference workflow is illustrated in Fig. 7. The InferenceService creates Triton pods that retrieves the ONNX model and configuration from S3 object storage. The pods act as REST endpoints [52] that can be queried over HTTP. When user sends inference requests, a load balancer is responsible for receiving them and distribute them to available inference server pods. The pods can scale up or down dynamically based on the volume of incoming requests. Servers pass the input data through the deep learning model and return model output to the load balancer that sends it back to the user.
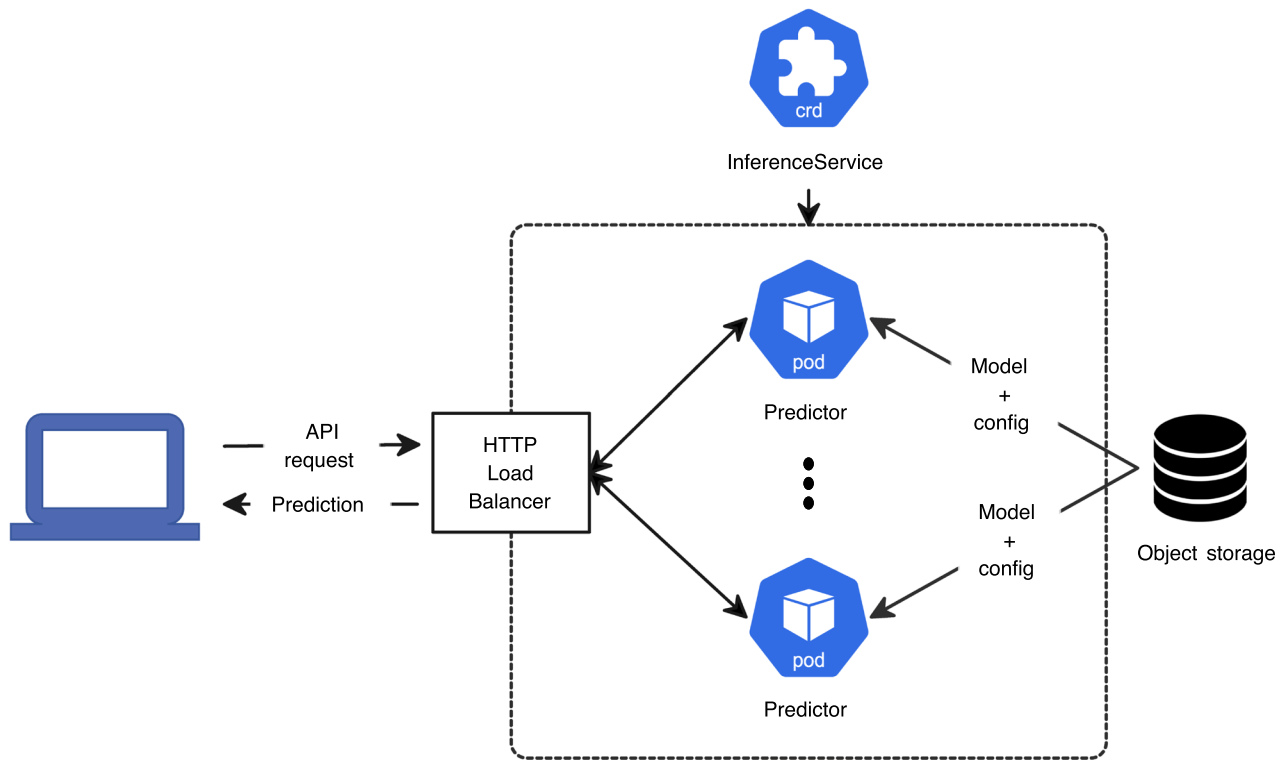
Fig. 7 A diagram depicting model serving using KServe, highlighting load balancing of user inference requests and the scalability of predictor pods

Results

Hyperparameter Optimization

The results of the hyperparameter tuning using Random Search can be analyzed to some extent with Pearson's correlation coefficient. Table 2 shows the correlation of all continuous and ordinal hyperparameters with the inverted test loss for the models with the lowest validation loss during each training run. Because the choice of hyperparameters is stochastic for every trial it is difficult to isolate the impact of a single hyperparameter on the model's performance, and thus particularly high correlation scores are not expected here. Note also that the correlation is limited to the search space laid out in the "AutoML Experiment" section.

The table suggests that for PFN, having more linear layers with fewer units in the Deep Sets block is weakly associated with a lower loss. For ParticleNet, having more EdgeConv

Table 2 A twofold presentation of the results from hyperparameter tuning

	Model	E	k	N	n	M	m	d	lr	Optim	L	Gain (%)
$\rho_{\text{hp},1/L}$	PFN			0.27	-0.21	-0.08	0.33	-0.54	-0.39			
	ParticleNet	0.22	-0.04	0.17	-0.21	0.02	0.40	-0.39	-0.34			
Top trials	PFN			5	50	3	400	9.45e$-$3	1.08e$-$3	Ranger	8.768e$-$2	7.52
				5	50	4	100	4.13e$-$2	7.60e$-$4	Adam	8.770e$-$2	7.49
				3	200	4	400	8.58e$-$2	7.33e$-$4	RMSProp	8.770e$-$2	7.49
	ParticleNet	3	16	3	50	3	200	1.16e$-$2	8.72e$-$3	Ranger	8.746e$-$2	7.79
		3	16	2	100	2	100	1.14e$-$1	1.84e$-$3	Ranger	8.752e$-$2	7.71
		3	8	2	50	3	200	8.70e$-$2	2.50e$-$3	AdamW	8.755e$-$2	7.68

The upper part of the table shows the Pearson correlation between hyperparameters and the inverted test loss for PFN and ParticleNet. The bottom part lists the top three best sets of hyperparameters found for both models with the corresponding test loss values for those trials. Gain signifies the relative improvement in loss for each trial compared to the standard corrections baseline ($L_{\text{Baseline}} = 9.427e-2$), and is computed as $\Delta L/L_{\text{Model}}$

blocks with additional linear layers and fewer units offers an advantage. The number of nearest neighbors k in ParticleNet's particle graph appears uncorrelated with a lower loss. The remaining correlation values exhibit similar behavior for both models. The loss is relatively unaffected by the number of linear layers in the network head, but more units in these layers tend to yield a lower loss. Increased dropout negatively impacts the regression task with the most certainty among all correlation results. Finally, a lower learning rate tends to produce better results.

The best hyperparameters found for both models, as listed in the lower section of Table 2, do tend to align with the Pearson correlation scores. For the PFN model, the top three trials all used a configuration with more linear layers (3–5) in the Deep Sets block and fewer units (50–200), which aligns with the correlation scores observed for these parameters. Similarly for ParticleNet, the top three trials incorporate more EdgeConv blocks (3) with a higher number of linear layers (2–3) and fewer units (50–100), reflecting the corresponding correlations. The initial learning rate and dropout across the top trials are, with exception for the optimal ParticleNet trial's learning rate, notably low as suggested by the Pearson correlation.

To further highlight the impact of poorly adjusted dropout and learning rate, we can compare the average of those parameters for trials that fall in the upper and lower quartiles ranked by test loss. For PFN, the average initial learning rate for trials in the lower quartile of losses was 3.0e−3, while for trials in the upper quartile it was higher, at 5.6e−3. Similarly, the average dropout for trials in the lower quartile was 0.11, compared to a significantly higher 0.31 in the upper quartile. A similar pattern was observed with the ParticleNet model, with the average initial learning rate for trials in the lower quartile being 3.3e−3, compared to 6.3e−3 in the upper quartile, and the average dropout for trials in the lower quartile being 0.16, compared to 0.31 in the upper quartile.

A high learning rate allows the model to learn quickly, but it may also cause the model to overshoot the optimal solution and not converge well. The aim of dropout on the other hand is for the model to learn more robust, generalizable representations of the data. However, if the dropout rate is too high, as is the case for many trials in the upper loss quartile, the model may struggle to learn from the data at all, leading to underfitting.

The optimizer algorithm as a nominal variable falls outside the scope of the correlation analysis. However, Ranger proved to be the most successful. It combines LookAhead [53] with $k = 6$ and $\alpha = 0.5$, and an inner RAdam optimizer [54] with $\beta_1 = 0.95$, $\beta_2 = 0.999$ and $\epsilon = 10^{-5}$. RAdam can help to stabilize the learning rate and adapt it based on the variance of the gradient, making it a robust option when the learning rate is ill-adjusted. Furthermore, LookAhead has empirically been shown to improve convergence by

considering multiple directions in the parameter space. It also mitigates the impact of poorly chosen hyperparameters on training by smoothing out noisy gradient updates.

The gain measure in Table 2 represents the percentage improvement in test loss achieved by the PFN and ParticleNet models over the standard jet energy corrections loss value. This metric serves as an indicator of the models' relative performance, providing a quantifiable measure of the benefits realized through hyperparameter optimization. The gain observed for the optimal PFN configuration is 7.52%, whereas the optimal ParticleNet model achieves a 7.79% improvement over the baseline.

In this study, Random Search proved straightforward to set up and it showcased favorable practical properties. Random state is the only input parameter, the algorithm allows for trials to be discontinued or restarted without jeopardizing the experiment, and compared to grid search, it is more efficient for a given computational budget [45]. However, Katib offers several other AutoML algorithms such as Bayesian optimization [55] or Hyperband [56] that could be considered for future work since they have great potential to more effectively find an optimal set of hyperparameters.

It should be noted that more advanced algorithms often rely on additional input parameters that may alter the outcome which adds a level of complexity in the setup. Furthermore, while Random Search is embarrassingly parallel, Bayesian Optimization uses Gaussian process regression to iteratively model the search space and is therefore inherently sequential. Trials can still be queued in parallel, but the choice of parameter configuration for back-to-back trials is less informed than when the algorithm is run sequentially. Hyperband, on the other hand, is an extension of Random Search, and offers more efficient resource allocation to trials that matter by invoking early stopping on poorly performing configurations. However, adjusting resource allocation when candidate configurations have different convergence rates is an open challenge [56], a circumstance occurring in our experiment with varying learning rates and models with differing numbers of layers and hidden units.

Model Complexity and Inference Performance

Table 3 compares the complexity of the optimal PFN and ParticleNet models in terms of loss, number of parameters, and Multiply-Accumulate operations (MACs). MACs

Table 3 Comparison of model complexity for the optimal configuration of PFN and ParticleNet

Model	Loss	# Params	MACs
PFN	8.768e−2	355.45k	1.43M
ParticleNet	8.746e−2	123.47k	47.59M

represent the number of multiplications and additions performed during a single forward pass, indicating computational complexity. While ParticleNet achieves a lower test loss than PFN due to the inclusion of particle locality information, it has significantly higher computational complexity. A smaller number of nearest neighbors in the particle graph and fewer channels in the linear layers can be considered for reducing the complexity while still maintaining good performance [12].

The optimal PFN and ParticleNet models are served as REST endpoints using the Triton Inference Server running on top of KServe. We used the Python Triton client to request predictions for different batch sizes to evaluate how these models compare, and how request batch size affects roundtrip time, inference time, and overhead. The roundtrip time encompasses the total duration for a request to be processed, including both HTTP request time and inference time. Meanwhile, the overhead, calculated as the difference between roundtrip time and inference time, represents additional delay induced by factors such as data serialization/deserialization and network latency. The results of these tests for both models when served either on a CPU or a Tesla V100 GPU are displayed in Fig. 8.

In the context of model performance, PFN achieve lower roundtrip times than ParticleNet due to its much lower computational complexity. However, the difference is less pronounced for smaller batch sizes. The larger overhead at small batch sizes affects both models similarly, thereby reducing the relative performance difference between the models.

For very fast models such as PFN running on GPU the inference time is minimal due to the low model complexity and the high parallel processing capabilities of the GPU. This leads to overhead being the dominating factor in the roundtrip time. Conversely, for slower models like ParticleNet, the inference time especially when processing large batch sizes is substantially longer than the overhead time. As

a result, the overhead comprises a fraction of the roundtrip time in that scenario.

We can note that PFN with extended ONNX graph optimization (PFN Opt.) shows performance enhancements compared to PFN with basic graph optimizations. Extended graph optimizations go beyond the simple, semantics-preserving transformations of basic optimizations, and apply complex node fusions after graph partitioning, tailoring the computation to the specific execution provider (CPU or GPU). This results in more efficient computations that are better suited to the architecture of the hardware on which the model runs.

The faster inference time on GPU compared to CPU for both PFN and ParticleNet models can be primarily attributed to the difference in the underlying architecture of these hardware platforms. GPUs are specifically designed for high-throughput, parallel processing and are capable of executing thousands of threads simultaneously. This characteristic is particularly beneficial for inference tasks which involve large-scale matrix operations that can be parallelized effectively. In contrast, CPUs have fewer cores and are optimized for sequential tasks. Furthermore, the performance gap can be widened when the batch size is large, as larger batch sizes enable better utilization of the GPU's parallel processing capabilities, leading to a faster per-jet inference time.

GPU inference and overhead times are also more consistent compared to CPU times. When allocating a processor in a Kubernetes cluster the instance is assigned a virtual processor (vCPU) shared across different processes, and due to their general-purpose nature, CPUs are typically tasked with managing a wider variety of processes, including system operations and other applications running in the background. A scheduler managed by the kernel has to coordinate time slots on the physical CPUs, which can lead to more variability in the availability of resources for the inference tasks. In contrast, GPU allocations on Kubeflow will currently result
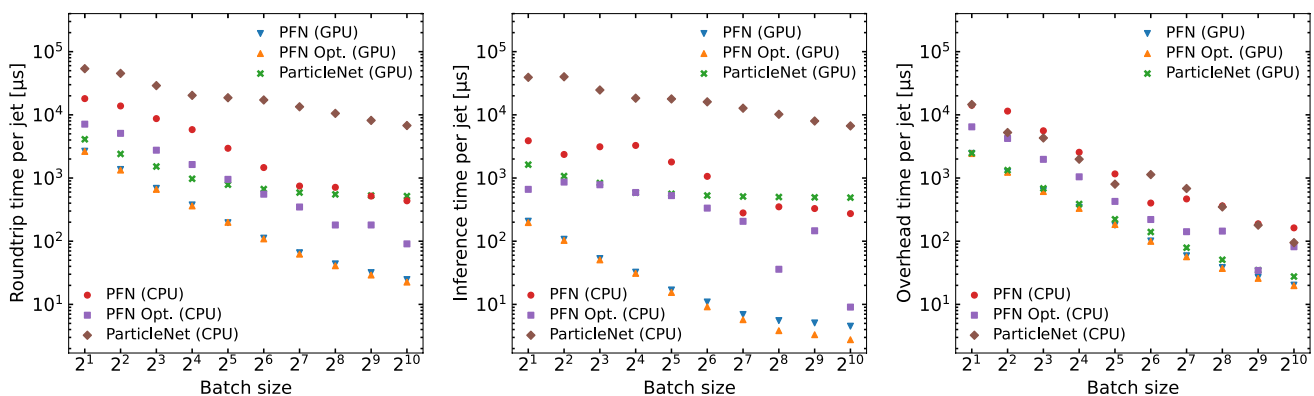
Fig. 8 Comparison of prediction request roundtrip time (left), device inference time (middle) and overhead (right) for PFN and ParticleNet ONNX models served with Triton. The represented values are based on the average processing time for 1k repetitions with randomly selected jets, covering a range of batch sizes from 2 to 1024

in a dedicated GPU for the task, resulting in more consistent performance.

When faced with the choice between CPU and GPU for deep learning inference it really depends on the hardware resources available and application requirements. Hardware accelerators such as GPUs provide superior throughput but often come with increased costs and power consumption. For applications with moderate inference workloads and less stringent response time requirements, CPU-based inference may be more cost-effective. In contrast, high-throughput, low-latency applications can benefit significantly from GPU investment. It is also noteworthy that while a dedicated GPU will yield good results when benchmarking, virtualization of these resources as vGPUs could lead to better resource utilization [57], which is especially important as demand for hardware accelerators continues to increase.

Jet Energy Response Flavor Dependence

In this section, we present the analysis of the flavor dependence of the calibrated energy response produced by the optimal PFN and ParticleNet models. Figure 9 displays the median response for each jet flavor obtained using the two deep learning models and the standard corrections baseline. Both models exhibit a reduction in the differences between jet flavors compared to the baseline. A notable improvement in flavor dependence of the energy calibration is observed between light quark jets and gluon jets, which has been a known shortcoming in standard jet energy corrections.

The uncertainty for each data point, represented by the error bars in Fig. 9, is calculated using the statistical bootstrapping method. By randomly sampling all response values 30 times, new sets of response values are generated similar in magnitude to the original dataset. The median is computed for each sample, followed by the standard deviation of the 30 median values, which provides an uncertainty for

each point. There are fewer s, c and b jets in the QCD sample compared to the amount of u, d and g jets contributing to them having a higher bootstrapped uncertainty. The sample also has fewer jets in the endcap region compared to barrel region resulting in a higher uncertainty for the median response in the right plot in Fig. 9 compared to the left one.

The sum of absolute errors (SAE) is used to evaluate the improvement in flavor dependence. It can be computed directly from the points in Fig. 9 by summing the absolute difference between the median response for each flavor and the mean of the same points. Mathematically, it can be expressed as:

$$SAE = \sum_{flavor} |R_{50\%, \, flavor} - \frac{1}{n} \sum_{flavor} (R_{50\%, \, flavor})|, \qquad (2)$$

where flavor $= \{u, d, s, c, b, g\}$ represents the complete set of jet flavors in the QCD sample. The relative improvement in flavor dependence for a model compared to the standard JEC is denoted by α and is defined as:

$$\alpha = 1 - SAE_{Model}/SAE_{Baseline}. \qquad (3)$$

The values that α can take on ranges from any negative value to one, where a negative value indicates that the model performs worse than the standard correction, zero corresponds to no improvement, and $\alpha = 1$ would mean that the median response produced by the deep learning model is identical for all jet flavors.

As seen in Table 4, the improvement in flavor dependence varies across different p_T intervals for both models. In the very low p_T region ($30 \, \text{GeV} < p_T^{gen} < 100 \, \text{GeV}$), the improvements in flavor dependence are modest. As the p_T intervals become larger, both models demonstrate more significant improvements. Which of the two models perform better in a certain p_T interval or detector region varies. However, the differences in performance between them become
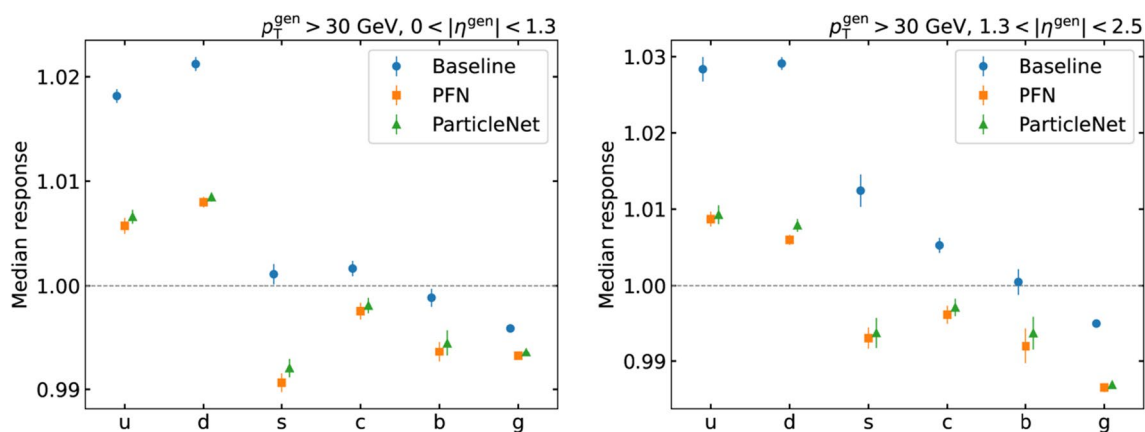
Fig. 9 Median energy response separated by jet flavor in the barrel region (left) and the endcap region (right). The improvement over the baseline is about 40% in the barrel region and 50% in the endcap region of the detector for both models

Table 4 Summary of jet energy regression results

Interval (GeV)	Model	α_{barrel} (%)	β_{barrel} (%)	α_{endcap} (%)	β_{endcap} (%)
$30 < p_{\mathrm{T}}^{\mathrm{gen}} < 100$	PFN	8.04	0.23	12.01	0.97
	ParticleNet	6.71	0.64	17.97	1.25
$100 < p_{\mathrm{T}}^{\mathrm{gen}} < 300$	PFN	23.25	1.43	49.59	6.93
	ParticleNet	24.52	1.61	45.07	7.07
$300 < p_{\mathrm{T}}^{\mathrm{gen}} < 1000$	PFN	57.52	4.53	68.05	12.90
	ParticleNet	56.11	4.81	70.02	11.93
$p_{\mathrm{T}}^{\mathrm{gen}} > 1000$	PFN	68.62	7.95	37.91	4.97
	ParticleNet	68.34	7.75	37.56	9.37

Improvements in flavor dependence and energy resolution produced by PFN and ParticleNet compared to baseline JEC are presented in multiple p_{T} intervals and detector regions

less pronounced as p_{T} increases. It is also worth noting that the improvement in flavor dependence is not uniform across the barrel and endcap regions. For both models, the improvement is generally larger in the endcap region, especially in the intermediate and high p_{T} intervals.

Flavor response differences and even more so their differences between different generators are an input to both ATLAS and CMS flavor-related uncertainties. If the flavor responses become more alike, because the underlying jet properties are taken into account, as demonstrated in Table 4 and Fig. 9, then one can also expect the uncertainties based on generator differences to be decreased.

Jet Energy Resolution

The performance of the regression models can also be assessed by examining the relative jet energy resolution. We define it here as the interquartile range (IQR) divided by the median for the response:

$$\bar{s} = \frac{R_{75\%} - R_{25\%}}{R_{50\%}}. \tag{4}$$

The IQR serves as a measure of response resolution. Both the median and IQR are robust statistics, meaning that they are less affected by outliers compared to the mean and standard deviation, respectively. The uncertainty of the relative resolution is measured using the same bootstrapping technique as in the previous section and results are shown in Fig. 10. As the high p_{T} and endcap region are less populated as indicated in Fig. 1, the uncertainties are sizeable for endcap high p_{T} jets.

The improvement in relative resolution for a model with respect to standard corrections is denoted here as β. We define it as one minus the ratio of relative jet energy resolution between the models and the baseline:

$$\beta = 1 - \bar{s}_{\mathrm{Model}}/\bar{s}_{\mathrm{Baseline}}. \tag{5}$$

When examining the results presented in Table 4, we observe that both PFN and ParticleNet models achieve improvements
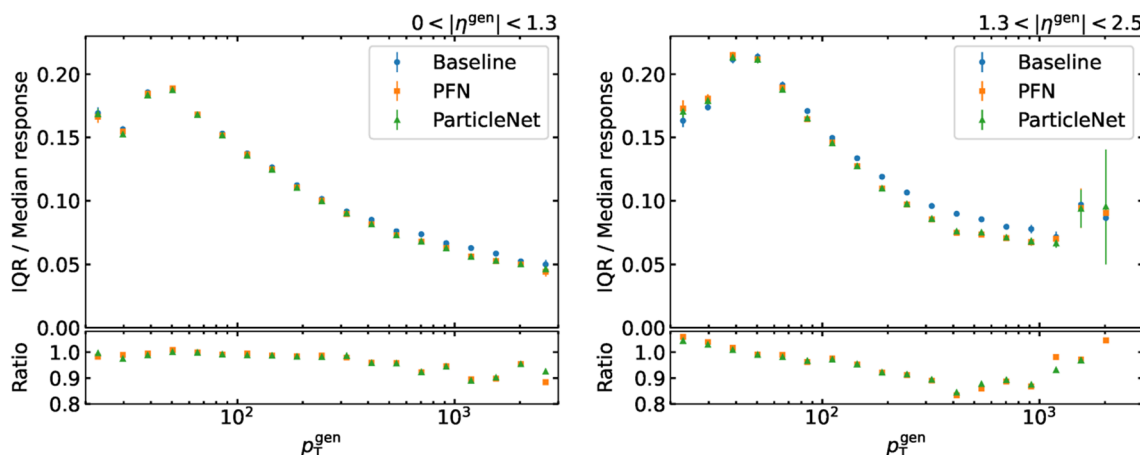
Fig. 10 Relative jet energy resolution in the barrel region (left) and the endcap region (right) binned logarithmically. The bottom panels show the ratio between the relative resolution produced by the deep learning models and the relative resolution after standard jet energy corrections

in the energy resolution compared to the baseline. The improvements vary across different p_T intervals, with the largest improvements observed in the intermediate to high p_T intervals for both models. This behavior can be attributed to several factors. High p_T jets generally have more complex substructures and are more likely to undergo hard parton splittings, resulting in a higher multiplicity for the jet. This increased complexity leaves more room for improvement for machine learning-based approaches. The effect of pileup also diminishes at higher p_T, resulting in less noisy data to train on. Owing to the limited amount of training data in the endcap region for jets with $p_T^{gen} > 1000$ GeV and reaching the kinematic limit of phase space in that regime, the improvements achieved through deep learning are comparatively smaller there.

Conclusion

In this paper, we presented a deep learning-based workflow for calibrating the energy of particle jets in the CMS detector. By utilizing advancements in learning on particle clouds in the form of the PFN and ParticleNet models, we managed to improve upon standard jet energy corrections derived solely from kinematic quantities. The results, categorized into jet energy resolution and flavor dependence, suggest that the performance of both networks is generally comparable, with larger improvements at higher p_T. The most notable difference between the two models is that the inclusion of locality information in ParticleNet results in a slightly better energy resolution at the expense of higher model complexity and inference time.

We have also demonstrated the potential of the Kubeflow platform for operationalizing ML workflows in high energy physics. As the field is witnessing a growing integration of ML techniques, the capabilities offered by Kubeflow, supporting the continual development of scalable ML solutions, are becoming increasingly more relevant. The pipeline we developed for this work enabled us to efficiently scale up our AutoML experiments on cloud resources and serve the optimal models as easily queryable REST endpoints. Having each step in the pipeline defined using Kubernetes custom resources allows for fine-grained access to hardware resources on the cloud and well-versioned, reusable machine learning workflows.

Author Contributions All authors contributed to the writing and reviewing of the manuscript. Corresponding author DH prepared the figures and the tables.

Declarations

References

1. Radovic A, Williams M, Rousseau D, Kagan M, Bonacorsi D, Himmel A, Aurisano A, Terao K, Wongjirad T (2018) Machine learning at the energy and intensity frontiers of particle physics. Nature 560(7716):41–48. https://doi.org/10.1038/s41586-018-0361-2

2. Guest D, Cranmer K, Whiteson D (2018) Deep learning and its application to LHC physics. Annu Rev Nuclear Particle Sci 68(1):161–181. https://doi.org/10.1146/annurev-nucl-101917-021019

3. Shlomi J, Battaglia P, Vlimant J-R (2020) Graph neural networks in particle physics. Mach Learn Sci Technol 2(2):021001. https://doi.org/10.1088/2632-2153/abbf9a

4. Ju X, Farrell S, Calafiura P, Murnane D, Gray L, Klijnsma T, Pedro K, Cerati G, Kowalkowski J, Perdue G et al (2019) Graph neural networks for particle reconstruction in high energy physics detectors. In: Advances in neural information processing systems, vol 32. https://doi.org/10.48550/arXiv.2003.11603

5. Choma N, Monti F, Gerhardt L, Palczewski T, Ronaghi Z, Prabhat P, Bhimji W, Bronstein MM, Klein SR, Bruna J (2018) Graph neural networks for IceCube signal classification. In: IEEE international conference on machine learning and applications, vol 17, p 386–391. https://doi.org/10.1109/ICMLA.2018.00064

6. The CMS Collaboration (2008) The CMS experiment at the CERN LHC. J Instrum 3(08):08004. https://doi.org/10.1088/1748-0221/3/08/S08004

7. The CMS Collaboration (2017) Jet energy scale and resolution in the CMS experiment in pp collisions at 8 TeV. J Instrum 12(02):02014. https://doi.org/10.1088/1748-0221/12/02/P02014

8. The CMS Collaboration (2023) Measurement of the top quark mass using a profile likelihood approach with the lepton+jets final states in proton-proton collisions at $\sqrt{s} = 13$ TeV. Technical report, CERN, Geneva. https://cds.cern.ch/record/2848244

9. The CMS Collaboration (2020) A deep neural network for simultaneous estimation of b jet energy and resolution. Comput Softw Big Sci 4(1):10. https://doi.org/10.1007/s41781-020-00041-z

10. Kallonen K (2019) Sample with jet properties for jet-flavor and other jet-related ML studies JetNTuple_QCD_RunII_13TeV_MC. CERN Open Data Portal. https://doi.org/10.7483/OPENDATA.CMS.RY2V.T797

11. Komiske PT, Metodiev EM, Thaler J (2019) Energy flow networks: deep sets for particle jets. J High Energy Phys 2019(1):121. https://doi.org/10.1007/JHEP01(2019)121

12. Qu H, Gouskos L (2020) Jet tagging via particle clouds. Phys Rev D 101:056019. https://doi.org/10.1103/PhysRevD.101.056019

13. Mäkinen S, Skogström H, Laaksonen E, Mikkonen T (2021) Who needs MLOps: what data scientists seek to accomplish and how can MLOps help? In: IEEE/ACM workshop on AI engineering—software engineering for AI, vol 1, p 109–112. https://doi.org/10.1109/WAIN52551.2021.00024

14. Golubovic D, Rocha R (2021) Training and Serving ML workloads with Kubeflow at CERN. In: 25th international conference on computing in high-energy and nuclear physics, vol 251, p 02067. https://doi.org/10.1051/epjconf/202125102067

15. Yuan DY, Wildish T (2020) Bioinformatics application with Kubeflow for batch processing in clouds. In: International conference on high performance computing, p 355–367. https://doi.org/10.1007/978-3-030-59851-8_24

16. Tsourdinis T, Chatzistefanidis I, Makris N, Korakis T (2022) AI-driven service-aware real-time slicing for beyond 5G networks. In: IEEE conference on computer communications workshops, vol 41, p 1–6. https://doi.org/10.1109/INFOCOMWKSHPS54753.2022.9798391

17. Carminati F, Khattak G, Loncar V, Nguyen TQ, Pierini M, Rocha RBD, Samaras-Tsakiris K, Vallecorsa S, Vlimant J-R (2020) Generative adversarial networks for fast simulation. J Phys Conf Ser 1525(1):012064. https://doi.org/10.1088/1742-6596/1525/1/012064

18. Sjöstrand T, Ask S, Christiansen JR, Corke R, Desai N, Ilten P, Mrenna S, Prestel S, Rasmussen CO, Skands PZ (2015) An introduction to PYTHIA 8.2. Comput Phys Commun 191:159–177. https://doi.org/10.1016/j.cpc.2015.01.024

19. Cacciari M, Salam GP, Soyez G (2008) The anti-kt jet clustering algorithm. J High Energy Phys 2008(04):063. https://doi.org/10.1088/1126-6708/2008/04/063

20. Agostinelli S, Allison J, Amako Ka, Apostolakis J, Araujo H, Arce P, Asai M, Axen D, Banerjee S, Barrand G, et al (2003) GEANT4—a simulation toolkit. Nuclear instruments and methods in physics research section A: accelerators, spectrometers, detectors and associated equipment 506(3):250–303. https://doi.org/10.1016/S0168-9002(03)01368-8

21. The CMS Collaboration (2018) Identification of heavy-flavour jets with the CMS detector in pp collisions at 13 TeV. J Instrum 13(05):05011. https://doi.org/10.1088/1748-0221/13/05/P05011

22. The CMS Collaboration (2017) Jet algorithms performance in 13 TeV data. Technical report, CERN, Geneva. http://cds.cern.ch/record/2256875

23. The CMS collaboration (2017) Particle-flow reconstruction and global event description with the CMS detector. JINST 12(10):10003. https://doi.org/10.1088/1748-0221/12/10/P10003

24. The CMS Collaboration (2020) Pileup mitigation at CMS in 13 TeV data. JINST 15(09):09018. https://doi.org/10.1088/1748-0221/15/09/P09018

25. The CMS Collaboration (2013) Performance of quark/gluon discrimination in 8 TeV pp data. Technical report, CERN, Geneva. https://cds.cern.ch/record/1599732

26. de Oliveira L, Kagan M, Mackey L, Nachman B, Schwartzman A (2016) Jet-images-deep learning edition. J High Energy Phys 2016(7):1–32. https://doi.org/10.1007/JHEP07(2016)069

27. Guest D, Collado J, Baldi P, Hsu S-C, Urban G, Whiteson D (2016) Jet flavor classification in high-energy physics with deep neural networks. Phys Rev D 94(11):112002. https://doi.org/10.1103/PhysRevD.94.112002

28. Louppe G, Cho K, Becot C, Cranmer K (2019) QCD-aware recursive neural networks for jet physics. J High Energy Phys 2019(1):1–23. https://doi.org/10.1007/JHEP01(2019)057

29. Zaheer M, Kottur S, Ravanbakhsh S, Poczos B, Salakhutdinov RR, Smola AJ (2017) Deep sets. In: Advances in neural information processing systems, vol 30. https://doi.org/10.48550/arXiv.1703.06114

30. Glorot X, Bordes A, Bengio Y (2011) Deep sparse rectifier neural networks. In: Proceedings of the fourteenth international conference on artificial intelligence and statistics. Proceedings of machine learning research, vol 15, p 315–323. https://proceedings.mlr.press/v15/glorot11a

31. Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R (2014) Dropout: a simple way to prevent neural networks from overfitting. J Mach Learn Res 15(56):1929–1958

32. Wang Y, Sun Y, Liu Z, Sarma SE, Bronstein MM, Solomon JM (2019) Dynamic graph CNN for learning on point clouds. ACM Trans Graph 38(5):1–12. https://doi.org/10.1145/3326362

33. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: IEEE conference on computer vision and pattern recognition, vol 29, p 770–778. https://doi.org/10.1109/CVPR.2016.90

34. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L et al (2019) Pytorch: an imperative style, high-performance deep learning library. In: Advances in neural information processing systems, vol. 32. https://doi.org/10.48550/arXiv.1912.01703

35. Qu H (2023) Weaver: a machine learning R &D framework for high energy physics applications. https://github.com/hqucms/weaver. Accessed 10 Mar 2023

36. Brun R, Rademakers F (1997) ROOT—an object oriented data analysis framework. Nuclear Instrum Methods Phys Res Sect A Accel Spectrom Detect Assoc Equip 389(1):81–86. https://doi.org/10.1016/S0168-9002(97)00048-X

37. Pivarski J, Elmer P, Lange D (2020) Awkward arrays in Python, C++, and Numba. In: 24th international conference on computing in high energy and nuclear physics, vol 245, p 05023. https://doi.org/10.1051/epjconf/202024505023

38. Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A et al (2004) Open MPI: goals, concept, and design of a next generation MPI implementation. In: Recent advances in parallel virtual machine and message passing interface: 11th European PVM/MPI Users' group meeting, p 97–104. https://doi.org/10.1007/978-3-540-30218-6_19

39. NVIDIA Corporation (2023) NCCL: optimized primitives for collective multi-GPU communication. https://github.com/nvidia/nccl. Accessed 13 Apr 2023

40. Burns B, Grant B, Oppenheimer D, Brewer E, Wilkes J (2016) Borg, Omega, and Kubernetes. ACM Queue 14:70–93. https://doi.org/10.1145/2890784

41. Pavlou CS, Kessler FB, Katsakioris I, Kostis L, Stefano F, Alexiou T, Valerio M (2023) KALE: Kubeflow automated pipeLines engine. https://github.com/kubeflow-kale/kale. Accessed 13 Apr 2023

42. The YAML Project (2023) YAML ain't markup language. https://yaml.org. Accessed 13 Apr 2023

43. George J, Gao C, Liu R, Liu HG, Tang Y, Pydipaty R, Saha AK (2020) A scalable and cloud-native hyperparameter tuning system. arXiv eprint. https://doi.org/10.48550/arXiv.2006.02085

44. The Kubeflow Project (2023) Kubernetes custom resource and operator for PyTorch jobs. https://github.com/kubeflow/pytorch-operator. Accessed 13 Apr 2023

45. Bergstra J, Bengio Y (2012) Random search for hyper-parameter optimization. J Mach Learn Res 13(10):281–305

46. ONNX Runtime Developers (2023) ONNX runtime: a cross-platform, high performance ML inferencing and training accelerator. https://onnxruntime.ai. Accessed 13 Apr 2023

47. NVIDIA Corporation (2023) Triton inference server: an optimized cloud and edge inferencing solution. https://github.com/triton-inference-server. Accessed 13 Apr 2023

48. Google Inc. (2023) Protocol buffers: a language-neutral, platform-neutral extensible mechanism for serializing structured data. https://protobuf.dev. Accessed 30 May 2023

49. The KServe Project (2023) KServe: standardized serverless ML inference platform on kubernetes. https://github.com/kserve/kserve. Accessed 13 Apr 2023

50. The Knative Project (2023) Knative: kubernetes-based platform to build, deploy, and manage modern serverless workloads. https://knative.dev. Accessed 13 Apr 2023

51. The Istio Project (2023) Istio: connect, secure, control, and observe services. https://istio.io. Accessed 13 Apr 2023

52. Fielding RT, Taylor RN (2002) Principled design of the modern web architecture. ACM Trans Internet Technol 2(2):115–150. https://doi.org/10.1145/514183.514185

53. Zhang M, Lucas J, Ba J, Hinton GE (2019) Lookahead optimizer: k steps forward, 1 step back. In: Advances in neural information processing systems, vol 32. https://doi.org/10.48550/arXiv.1907.08610

54. Liu L, Jiang H, He P, Chen W, Liu X, Gao J, Han J (2020) On the variance of the adaptive learning rate and beyond. In: International conference on learning representations, vol 8. https://doi.org/10.48550/arXiv.1908.03265

55. Brochu E, Cora VM, de Freitas N (2009) A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. Technical report UBC TR-2009-023, University of British Columbia, Department of Computer Science. https://doi.org/10.48550/arXiv.1012.2599

56. Li L, Jamieson K, DeSalvo G, Rostamizadeh A, Talwalkar A (2018) Hyperband: a novel bandit-based approach to hyperparameter optimization. J Mach Learn Res 18(185):1–52

57. Golubovic D, Gaponcic D, Guerra D, Rocha R (2023) Efficient access to shared GPU resources part 1: mechanisms, motivations and use cases for GPU concurrency on kubernetes. https://kubernetes.web.cern.ch/blog/2023/01/09/efficient-access-to-shared-gpu-resources-part-1