Scaling Up Diffusion and Flow-based XGBoost Models

Jesse C. Cresswell
Layer 6 AI, Toronto, Canada
jesse@layer6.ai

Taewoo Kim
Layer 6 AI, Toronto, Canada
taewoo@layer6.ai

Abstract

Novel machine learning methods for tabular data generation are often developed on small datasets which do not match the scale required for scientific applications. We investigate a recent proposal to use XGBoost as the function approximator in diffusion and flow-matching models on tabular data, which proved to be extremely memory intensive, even on tiny datasets. In this work, we conduct a critical analysis of the existing implementation from an engineering perspective, and show that these limitations are not fundamental to the method; with better implementation it can be scaled to datasets $370\times$ larger than previously used. Our efficient implementation also unlocks scaling models to much larger sizes which we show directly leads to improved performance on benchmark tasks. We also propose algorithmic improvements that can further benefit resource usage and model performance, including multi-output trees which are well-suited to generative modeling. Finally, we present results on large-scale scientific datasets derived from experimental particle physics as part of the Fast Calorimeter Simulation Challenge. Code is available at github.com/layer6ai-labs/calo-forest.

1 Introduction

The design of neural network (NN) architectures with appropriate inductive biases for a given data modality has lead to incredible breakthroughs on text [92], audio [42], image [57, 39], graph [51], and many other modalities. However, tabular data stands out in that tree-based architectures still often outperform NNs [38, 67]. This can largely be attributed to the lack of exploitable structure in tabular data that NN design typically relies on.

Despite the success of boosted tree architectures like XGBoost [17] on *discriminative* tasks, they are rarely used for *generative* modeling [74, 75]. This is surprising, as XGBoost brings several other meaningful advantages: XGBoost does not require significant data pre-processing (NNs are highly sensitive to data scale and distribution); XGBoost can operate on data that contains null values

Figure 1: Comparison of training time and memory usage between the original implementation and ours. The × indicates job failure, and the horizontal line indicates the maximum system memory.

(NNs require null values to be imputed or entire columns dropped); XGBoost can be trained efficiently on CPU or GPU (NNs usually require GPU training); and XGBoost has superior explainability (Shapley values [84] are intractable for large NNs, but the TreeSHAP algorithm makes them tractable for trees [63, 64]). Similar to MLP networks, XGBoost is a universal function approximator [31, 32] and can be used to fit any function, at least in principle.

Recently, Jolicoeur-Martineau et al. [46] proposed a method for training diffusion [88] and flow-matching [60, 6, 58] generative models on tabular data by using XGBoost as the function approximator for a learnable vector field. Given the discussion above, this idea shows great promise. However, the original implementation was only benchmarked on small datasets (up to 11,000 datapoints with 16 features), and proved to be incredibly memory intensive (Figure 1 solid lines). Important scientific and industrial applications of tabular generative modeling typically operate at much
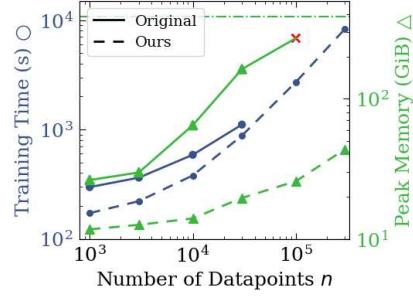
larger scales, such as the Fast Calorimeter Simulation Challenge for generative modeling of particle physics interactions [26] with tabular datasets $370\times$ larger than those used in [46].

In this work we conduct a deep and critical analysis of the implementation of diffusion and flow-matching models backed by XGBoost using engineering best-practices, and provide a new implementation re-engineered from the ground-up. Our implementation reduces CPU memory requirements from roughly quadratic in dataset size to linear and greatly reduces the memory overhead (Figure 1 dashed lines), showing that the algorithm is much more broadly applicable than previously thought. Our efficient implementation allows models to be scaled up in size which directly leads to model performance gains. In addition, we demonstrate novel techniques to improve generative quality, including the use of multi-output trees for generative modeling, which can more accurately represent high-dimensional joint distributions. Finally, we demonstrate that the methods are feasible in practice by applying them to large-scale scientific datasets.

2   Background

First, we briefly review diffusion and flow matching models and then describe how to train them with XGBoost function approximators.[1] For a more extensive description of diffusion and flow matching models, see [61]. Finally, we introduce the application of interest from experimental particle physics.

2.1   Diffusion Models

Score-based diffusion models [88] corrupt data $\mathbf{x}_0 \sim p_0$ by progressively adding noise as $t \in [0, 1]$ increases, in a process modeled by a stochastic differential equation (SDE). Reversing this process enables the generation of data from pure noise. The reverse SDE involves a novel term, the score function $\nabla_{\mathbf{x}_t} \log p_t(\mathbf{x}_t)$, where p_t is the density corresponding to data at noise level t. Since the data density p_0 is not known in closed form, neither is p_t, so $\nabla_{\mathbf{x}_t} \log p_t(\mathbf{x}_t)$ cannot be directly computed. However, it can be estimated using a denoising score matching approach [43, 93] with the loss

$$L_{\mathrm{SM}}(\theta) = \mathbb{E}_{t \sim \mathcal{U}(0,1)} w(t) \mathbb{E}_{\mathbf{x}_0 \sim p_0, \mathbf{x}_t \sim p_t(\cdot | \mathbf{x}_0)} \| \mathbf{s}_\theta(\mathbf{x}_t, t) - \nabla_{\mathbf{x}_t} \log p_t(\mathbf{x}_t | \mathbf{x}_0) \|_2^2. \tag{1}$$

Here, $\mathbf{s}_\theta$ is a parameterized vector field that is directly regressed on the score function, while $w(t)$ is a positive-valued weighting function that can be chosen freely. In words, for a t sampled uniformly, and $\mathbf{x}_0$ drawn from the data distribution, we sample $\mathbf{x}_t$ from $p_t(\cdot \mid \mathbf{x}_0)$ which is Gaussian [88],

$$p_t(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\Big(\mathbf{x}_t; \sqrt{1 - \sigma_t^2} \mathbf{x}_0, \sigma_t^2 \mathbf{I}_D\Big), \tag{2}$$

since p_t is the result of a linear SDE starting with a point mass at $\mathbf{x}_0$. The standard deviation σ_t depends on the details of the forward SDE. For generation, $\mathbf{s}_\theta$ replaces the score function in the reverse SDE which is then solved numerically.

2.2   Flow Matching

Like continuous normalizing flows [16], flow matching interpolates probability densities p_t for $t \in [0, 1]$ [60, 6, 58]. We consider p_0 as the data, and select a simple prior $p_1 = \mathcal{N}(\mathbf{x}_1 \mid 0, \sigma^2)$. The interpolation is determined by a vector field at each time $\boldsymbol{\mu}_t$, which transports datapoints $\mathbf{x}_t$ via the ODE $d\mathbf{x}_t = \boldsymbol{\mu}_t(\mathbf{x}_t) dt$. When p_t and $\boldsymbol{\mu}_t$ jointly satisfy the continuity equation

$$\frac{d}{dt} p_t + \nabla_{\mathbf{x}} \cdot (p_t \boldsymbol{\mu}_t) = 0, \tag{3}$$

then p_t will be a properly normalized density at each t. To perform flow matching, one would train a model $\boldsymbol{\nu}_\theta(\mathbf{x}_t, t)$ of the vector field $\boldsymbol{\mu}_t(\mathbf{x}_t)$ by direct regression,

$$L_{\mathrm{FM}}(\theta) = \mathbb{E}_{t \sim \mathcal{U}(0,1), \, \mathbf{x}_t \sim p_t} \| \boldsymbol{\nu}_\theta(\mathbf{x}_t, t) - \boldsymbol{\mu}_t(\mathbf{x}_t) \|_2^2. \tag{4}$$

However, in practice neither p_t nor $\boldsymbol{\mu}_t$ is uniquely determined, we can only sample from p_t for $t = 0$ (data) and 1 (prior), and we do not have access to $\boldsymbol{\mu}_t$ for evaluation at $\mathbf{x}_t$.

As a workaround, conditional flow matching (CFM) proposes to use conditional densities $p_t(\mathbf{x}_t \mid (\mathbf{x}_0, \mathbf{x}_1))$ and vector fields $\boldsymbol{\mu}_t(\mathbf{x}_t \mid (\mathbf{x}_0, \mathbf{x}_1))$, where $\mathbf{x}_0 \sim p_0$ is a training datapoint and $\mathbf{x}_1 \sim p_1$ is noise, such that both are tractable. For example, when we define

---

[1]We refer to XGBoost throughout, but other tree-based regressors could be used. Our implementation takes advantage of XGBoost features that are currently unavailable in other gradient-boosted decision tree libraries.

$$p_t(\mathbf{x}_t \mid (\mathbf{x}_0, \mathbf{x}_1)) = \mathcal{N}\big(\mathbf{x}_t; t\mathbf{x}_1 + (1-t)\mathbf{x}_0, \sigma^2\mathbf{I}_D\big), \qquad \boldsymbol{\mu}_t(\mathbf{x}_t \mid (\mathbf{x}_0, \mathbf{x}_1)) = \mathbf{x}_1 - \mathbf{x}_0, \quad (5)$$

for some $\sigma \geq 0$, the continuity equation (Eq. 3) is satisfied [90]. Now sampling data conditionally as $\mathbf{x}_t \sim p_t(\cdot \mid (\mathbf{x}_0, \mathbf{x}_1))$, the CFM loss

$$L_{\text{CFM}} = \mathbb{E}_{t \sim \mathcal{U}(0,1), \, \mathbf{x}_0 \sim p_0, \, \mathbf{x}_1 \sim p_1, \, \mathbf{x}_t \sim p_t(\cdot \mid (\mathbf{x}_0, \mathbf{x}_1))} \|\boldsymbol{\nu}_\theta(\mathbf{x}_t, t) - \boldsymbol{\mu}_t(\mathbf{x}_t \mid (\mathbf{x}_0, \mathbf{x}_1))\|_2^2, \qquad (6)$$

has the same gradients as Eq. 4, and therefore will lead to the same model $\boldsymbol{\nu}_\theta(\mathbf{x}_t, t)$, but is actually tractable. Finally, new datapoints are generated by solving the ODE starting from $\mathbf{x}_1 \sim p_1$ but using the learned vector field $\boldsymbol{\nu}_\theta(\mathbf{x}_t, t)$ instead of $\boldsymbol{\mu}_t(\mathbf{x}_t)$.

2.3 ForestDiffusion and ForestFlow

There is a clear commonality between the two methods: both regress a parameterized vector field. In almost all applications to date, NNs are used to parameterize the vector field. Jolicoeur-Martineau et al. [46] made the interesting observation that an XGBoost regressor [17] could be used instead, potentially harnessing the successes of tree-based learning in a generative setting. However, there are several major differences between how NNs and XGBoost are trained that must be overcome.

First, when using NNs Eq. 1 or 6 would be optimized by sampling a minibatch of data $\mathbf{x}_0 \sim p_0$, sampling $t \sim \mathcal{U}(0,1)$ independently for each $\mathbf{x}_0$, sampling fresh noise $\mathbf{x}_1 \sim p_1$ (Eq. 6 only), and then sampling $\mathbf{x}_t$ from the Gaussian in Eq. 2 or 5, respectively. In particular, the timestep t and random vector $\mathbf{x}_t$ would be sampled anew every batch, eventually leading to good coverage of the distributions in the loss function expectations. XGBoost is not trained with minibatches; it takes an entire dataset at once and minimizes the loss overall. Hence, the random vector $\mathbf{x}_t$ for each training point $\mathbf{x}_0$ would only be sampled once. For better coverage of the distribution, [46] proposed to duplicate each of the n training datapoints K times, and generate different $\mathbf{x}_t$ for each copy.

Second, whereas a NN can easily be designed with a number of outputs equal to the number of features p in $\mathbf{x}$ (the same size as the target vector field), standard decision trees only output a scalar. A brute-force workaround is to train a different XGBoost ensemble to predict each feature [46].

Third, whereas with a NN the time step t could be fed in as an additional input to the network during training and generation, [46] argued that simply adding t as a feature to XGBoost is unlikely to give sufficient emphasis to it, instead proposing to discretize t into n_t uniform steps and train a different XGBoost ensemble for each. The expectation over t is removed in the loss function Eq. 1 or 6, and t is instead treated as a constant for each of n_t separate loss functions.

Fourth, when conditional generation on a class label y is required, a NN can accept y as an input to adapt its behaviour while sharing parameters. Like conditioning on t, conditioning on y is better done by training a separate XGBoost ensemble for each of the n_y classes [46].

Combining these four solutions, [46] proposed ForestDiffusion and ForestFlow, aiming to realize the promises of tree-based generative modeling laid out in Section 1. While [46] reported excellent model performance, there are clear limitations, mainly the memory requirements from data duplication, and inefficient parameter use from predicting each column with separate XGBoost ensembles. In total, on a tabular dataset of size $[n, p]$, both methods require training $n_t \cdot n_y \cdot p$ XGBoost ensembles on n_t different datasets of size $[n_i \cdot K, p]$, where n_i is the number of datapoints with label i such that

Table 1: **Top:** The largest training datasets from [46] in terms of n, p, and np. N/A means y is continuous. **Bottom:** We scale to calorimeter datasets which are up to $370\times$ larger in np.

Dataset	Datapoints n	Columns p	Classes n_y
calif.	16,512	9	N/A
libras	288	90	15
bean	10,888	16	7
Photons	121,000	368	15
Pions	120,800	533	15

$\sum_{i=1}^{n_y} n_i = n$. The recommended settings are $n_t \approx 50$ and $K \approx 100$, whereas $n_i \approx n/n_y$ for class-balanced data. To emphasize the scaling issues, the *libras* dataset featured in [46] with $n = 288$ training datapoints required 151 GiB of CPU memory using the original implementation. We address these scaling issues below with novel techniques and better implementation.

2.4 Calorimeter Simulation

To motivate the need for scalable tabular data generation, we consider an important scientific application - calorimeter simulation. Measuring particle energy with calorimeters is one of the major components in particle accelerator experiments. To understand predictions from theory, physicists

simulate interactions within calorimeters, but doing so from first principles is computationally expensive [4, 7, 8]. Generative models have seen remarkable uptake as surrogates for fast simulation, spurred on by the Fast Calorimeter Simulation Challenge [26] which provides large tabular datasets of calorimeter measurements, and evaluation metrics that are scientifically relevant. A comparison of the training dataset sizes from [46] and from the Challenge is given in Table 1. In Appendix A we provide an extensive review of machine learning methods for calorimeter simulation.

3 Scaling Up

In this section we provide a step-by-step breakdown of the implementation of the main algorithm from [46], shown in pseudocode in Algorithm 1, then re-engineer it to scale to calorimeter data. After resolving implementation issues, we offer new techniques to further improve algorithm and model performance, including by completely changing the tree structure used by XGBoost.

3.1 Limitations of the Existing Implementation

Algorithm 1: ForestDiffusion and ForestFlow

Input: Dataset X_0 of size $[n, p]$, K, n_t.
$X_0' \leftarrow$ K-fold duplicate of the rows of X_0
$X_1 \leftarrow$ Dataset of $\mathbf{x}_1 \sim \mathcal{N}(0, \mathbf{I}_p)$ with the size of X_0'
for *Timestep $t \in range(n_t)$* **do**
    for *Class $y \in range(n_y)$* **do**
        for *Feature $p_i \in range(p)$* **do**
            $X_{0,y}' \leftarrow$ rows of X_0' with label y
            $X_{1,y} \leftarrow$ corresponding rows of X_1
            // Create $\mathbf{x}_t$ and regression targets
            $X_{t,y}', Z_{t,p_i} \leftarrow$ Forward$(X_{0,y}', X_{1,y}, t, p_i)$
            $f_{t,y,p_i} \leftarrow$ Regress on Z_{t,p_i} given $X_{t,y}'$
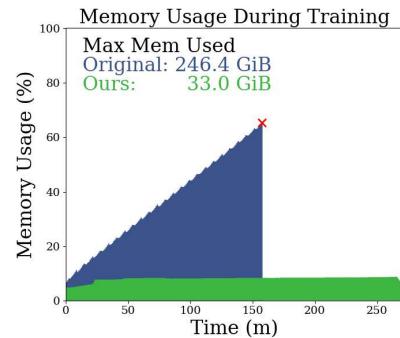**return** $\{f_{t,y,p_i}\}$ // Set of $n_t n_y p$ XGBoost ensembles

---

Figure 2: Memory usage during training for the original implementation and ours. The red cross $\times$ indicates job failure.

In Algorithm 1 the triple loop must be parallelized well for the method to scale to large datasets (see Appendix B for discussion). Since the training dataset X_0 is duplicated K times and n_t different versions are generated to represent $\mathbf{x}_t$ at different timesteps, memory can become a severe issue which is compounded by the number of XGBoost ensembles to be trained. Unfortunately, these challenges are not handled well in the implementation published by [46]. To demonstrate the need for rework, we show the resource usage during training for a small dataset with $n = 1000$, $p = 100$, and $n_y = 10$ in Figure 2. Despite our computer having 385 GiB of CPU memory available, the training job failed. There are three clear problems that this example shows: (1) even on a modest dataset the absolute amount of memory consumed is unexpectedly high (250 GiB); (2) memory usage grows at a constant rate during training, potentially leading to out-of-memory errors hours into a job; (3) training can fail due to memory issues even when the system maximum has not been reached.

3.2 Outlining the Existing Implementation

To begin unravelling the causes of the three undesirable behaviours above, we consider in detail the Python implementation by [46].[2] First, the dataset X_0 of size $[n, p]$ is given as a Numpy array, and if the dataset has discrete labels for conditioning, these are denoted as y. A min-max scaler applied to X_0 fits all data into the range $[-1, 1]$, after which the dataset is duplicated K times, giving X_0' of size $[nK, p]$, and noise X_1 is sampled with the same shape. For conditioning on the n_y distinct classes in y, n_y Boolean masks are created over X_0'. With this preparatory work done, the regression inputs and targets are created, denoted by X_t' and Z_t respectively. X_t' represents samples $\mathbf{x}_t$ from the distribution in Equation 2 or 5, while Z_t is either the score function $\nabla_{\mathbf{x}_t} \log p_t(\mathbf{x}_t \mid \mathbf{x}_0)$ from Equation 1 or the conditional vector field $\boldsymbol{\mu}_t(\mathbf{x}_t \mid (\mathbf{x}_0, \mathbf{x}_1))$ from Equation 6. Finally, all $n_t \cdot n_y \cdot p$ models are trained in a parallel triple loop with the widely used Python parallelization library Joblib.

---

[2]We base our discussion on `github.com/SamsungSAILMontreal/ForestDiffusion`, commit hash 855281b dated Nov. 2, 2023. After this point significant changes involving the XGBoost data iterator were added. We discuss the data iterator and reasons for not using a more recent version in Appendix B.3.

These steps are shown in the following code snippet, which has been compressed to show only the crucial information, and uses ForestFlow as a representative example.

```
      Python Implementation of ForestFlow from [46]
 1  from sklearn.preprocessing import MinMaxScaler
 2  import numpy, xgboost
 3  from joblib import delayed, Parallel
 4  X0, y, K, n_t, xgb_kwargs, n_jobs = inputs()
 5  n, p = X0.shape
 6  # Scale data so that range matches noise variance
 7  X0 = MinMaxScaler(feature_range=(-1, 1)).fit_transform(X0)
 8  # Duplicate data and generate noise
 9  X0 = numpy.tile(X0, (K, 1)) # [n*K, p]
10  X1 = numpy.random.normal(size=X0.shape) # [n*K, p]
11  # Create Boolean masks for class conditioning
12  mask = {} # Boolean mask for which rows of X0 have label y_i
13  y_uniq = numpy.unique(y)
14  for y_i in y_uniq:
15      mask[y_i] = numpy.tile(y == y_i, K)
16  # Create regression targets (ForestFlow)
17  t = numpy.linspace(0, 1, num=n_t) # Discretize time into n_t equal steps
18  X_train = t*X1 + (1-t)*X0 # [n_t, n*K, p]
19  Z_train = X1 - X0 # [n*K, p]
20  # Train models in parallel triple loop over timesteps, classes, and features
21  def train_parallel(X_tr_i, Z_tr_i):
22      model = xgboost.XGBRegressor(**xgb_kwargs)
23      return model.fit(X_tr_i, Z_tr_i)
24  regressors = Parallel(n_jobs)(
25      delayed(train_parallel)(
26          X_train[t_i][mask[y_i], :], Z_train[mask[y_i], p_i]
27      ) for t_i in range(n_t) for y_i in y_uniq for p_i in range(p)
28  ) # regressors is list of n_t*n_y*p XGBoost ensembles
```

3.3 Analysis and Improvement of the Implementation

While the implementation above looks innocuous, Figure 2 shows that there are serious engineering issues lurking. We aim to answer the following specific questions based on the observations above: (1) Why are memory requirements high for tiny datasets? (2) Why does memory usage increase during training? (3) Why do jobs fail before memory usage reaches 100%? We proceed by identifying issues, proposing solutions, and quantifying the benefits, starting with a simple observation.

Issue 1: Regression inputs `X_train` for all timesteps are created in memory at once, which is a large array of shape `[n_t, n*K, p]` (Line 18). Using the recommended values of `n_t` and `K` creates an array 5000 times the size of the training dataset, making even small datasets burdensome (Question 1).

Solution 1: Each XGBoost training call requires only the information at a single timestep `X_train[t_i]`, and so this data should be generated on-the-fly within the `n_t` loop.

Benefit 1: We avoid holding an array of size `[n_t, n*K, p]` in memory. This is in fact the first issue encountered when trying to apply the implementation from [46] to calorimeter data, since a `numpy.float64` array of size `[50, 120800*100, 533]` (Table 1) requires **2.34 TiB** of memory.

```
      Issue 1: Original
 1  t = numpy.linspace(0, 1, num=n_t)
 2  X_train = t*X1 + (1-t)*X0
 3  # [n_t, n*K, p] array in memory
```

```
      Issue 1: Improvement
 1  def train_parallel(X0,X1,Z_tr_i,t_i):
 2      X_tr_i = t_i*X1 + (1-t_i)*X0
 3      model = xgboost.XGBRegressor()
 4      return model.fit(X_tr_i, Z_tr_i)
```

Issue 2: Parallelization of the many XGB training calls is necessary for an efficient implementation. We discuss this requirement and other approaches to parallelization in Appendix B. Due to the Python global interpreter lock, parallelization is often handled through multiprocessing with Joblib, wherein the main process spawns worker processes and sends them copies of data. When a large array is assigned to multiple workers, Joblib automatically puts it into shared memory as a memory-mapped file, and passes only the reference to the workers. When an indexed array like `X_train[t_i][mask[y_i], :]` is passed to Joblib's `Parallel` call (Line 26), it is treated as a

distinct array, even though the same indexed array might appear for many jobs. This is because "advanced indexing" in Numpy, such as when the array `mask[y_i]` is used as the selection object, always creates a copy of the data. Hence, Joblib creates a new array in shared memory for every copy throughout every call in the triple parallel loop. By default, Joblib stores the memory-mapped arrays in RAM disk, a virtual disk on RAM, and does not free that memory until all parallel jobs are done. This continuously increases RAM disk usage during training (Question 2) and can lead to out-of-memory errors if RAM disk usage reaches its maximum capacity. For example, the maximum shared memory size on our Ubuntu 22.04 machine had been set to 189 GiB by default, and the failure in Figure 2 was caused by the RAM disk reaching this limit, even though RAM itself (at 385 GiB) was not at 100% usage (Question 3).

Solution 2: Instead of passing indexed arrays in the `Parallel` call, pass the entire array and index it inside worker processes. Upon the first call, Joblib puts the array into shared memory, but in subsequent calls, Joblib identifies the same array being requested and passes only a reference to workers.

Benefit 2: In the original implementation, each `train_parallel` call in the triple loop over (`n_t`, `n_y`, `p`) creates a copy of the inputs in shared memory. Looking only at `X_train`, the triple loop consumes `n_t*p*sizeof(X_train[t_i])=n_t*p*(n*K*p*8)` bytes in shared memory, which is p times more than Issue 1 (Question 1). For the Pions dataset (Table 1), this would amount to **1.22 PiB**. Our solution holds only one copy of `X0` and `X1` in shared memory, a factor of `n_t*p` less.

```
Issue 2: Improvement
1  def train_parallel(X0, X1, Z_train, t_i, mask_i, p_i):
2      X_tr_i = t_i*X1[mask_i, :] + (1-t_i)*X0[mask_i, :]
3      Z_tr_i = Z_train[mask_i, p_i]
4      model = xgboost.XGBRegressor(**xgb_kwargs)
5      return model.fit(X_tr_i, Z_tr_i)
6  regressors = Parallel(n_jobs)(
7      delayed(train_parallel)(
8          X0, X1, Z_train, t_i, mask[y_i], p_i
9      ) for t_i in t for y_i in y_uniq for p_i in range(p) )
```

Issue 3: All trained XGBoost models are held in memory until the end of training, causing a steady increase of consumed memory as training progresses (Question 2). The memory consumed by these models is independent of `n`, but increases with `p` and `n_y` (Question 1).

Solution 3: After a model is trained, write it to disk, and delete it from memory. Use the Universal Binary JSON (UBJ) format as it is compatible across XGBoost versions, and is the fastest format overall for reading and writing with the best compression on disk.

Benefit 3: Workers writing their trained model to disk prevents the growth of memory usage over training, and bypasses the need to return models from the worker to the main process via pickling. Furthermore, it serves as a checkpoint so that training can be easily resumed upon system failure. ForestFlow requires `n_t*n_y*p` XGBoost ensembles, each made of `n_tree` trees which themselves have up to `2**(d+1)-1` nodes where `d` is the maximum depth. Each node of an XGBoost tree uses 53 bytes to store parameters, metadata, and training statistics. Using the recommended defaults of `n_tree=100`, `d=7`, and no regularization, essentially all trees will have the maximum 255 nodes, and would require in total **503 GiB** on the Pions dataset (Table 1).

```
Issue 3: Original
1  def train_parallel(...):
2      ...
3      return model.fit(X_tr_i, Z_tr_i)
```

```
Issue 3: Improvement
1  def train_parallel(...):
2      ...
3      model.fit(X_tr_i, Z_tr_i)
4      model.save_model(f"{path}.ubj")
```

At this point we have reviewed the most significant training issues which together explain the three problematic observations from Figure 2, and when fixed, provide the bulk of resource improvements that we report. In Appendix B.1 we list four additional issues and solutions of smaller magnitude, and present a Python implementation with all of our changes. Figure 2 shows that our implementation solves the three observed problems. However, training is only one part of the story. In Appendix B.2 we investigate improvements to generation speed, with our method proving to be more than an order of magnitude faster.
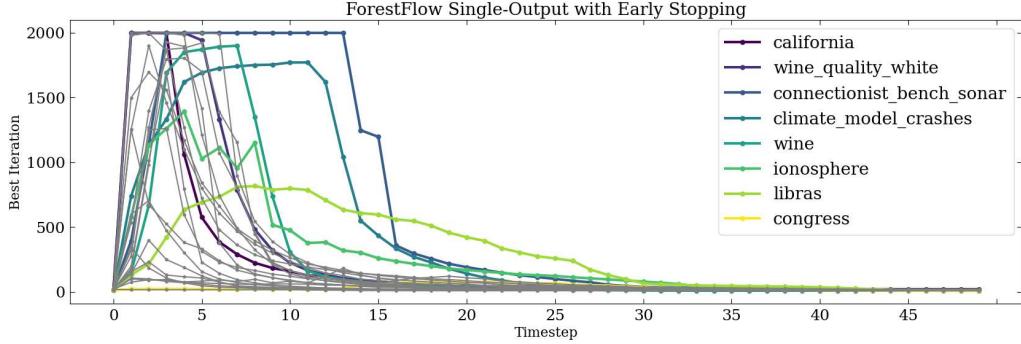
Figure 3: Number of trees at the best iteration of the validation loss by timestep and dataset. Selected datasets from all 27 are highlighted for comparison to MO in App. C.2. Early stopping after $n_{ES} = 20$ rounds with no improvement prevents wasteful training where no progress is being made.

3.4 Algorithmic Modifications

Beyond improving the implementation of Algorithm 1, we also offer modifications that can improve model performance and resource efficiency. Our two most significant changes are described below, with further minor recommendations in Appendix C, along with ablation studies to understand the impact of the proposed changes and key hyperparameters.

Multi-output trees: First, we propose a significant change to the structure of trees to make them more suited to the high-dimensional outputs required in generative modelling. Instead of training p single-output (SO) trees, one for each feature, we propose to use multi-output (MO) trees [100, 99, 66, 44, 81], where a single tree outputs p values. This is clearly advantageous for parameter efficiency: p times fewer XGBoost ensembles are required to represent the vector field, which is a massive benefit for generation speed and trained model memory requirements. However, multi-output trees also have the potential to better model the joint distribution of data. Consider how a set of p single-output trees generates a vector output. From identical inputs, each tree independently identifies the appropriate leaf node and outputs a scalar – there is no dependence between elements during generation. This is clearly not desirable for generative models where we aim to model the joint distribution of the data, not merely the marginals. Multi-output trees can better represent these correlations since generated elements come from a single tree with interactions during generation.

Early stopping: Second, our improved implementation unlocks the ability to greatly scale up model size by increasing K for better coverage of the expectations in Eq. 1 or 6, and by increasing the number of trees per ensemble, n_{tree}, for more model capacity. Jolicoeur-Martineau et al. [46] noted that underfitting was a major concern, and hence used no regularization. Our findings are more nuanced; underfitting is a concern, but moreso for values of t near 0 (data) than 1 (noise). Models trained for large values of t require much less capacity. This is related to the well-known phenomenon that noise scheduling has a strong impact on diffusion model performance [73, 18]. Rather than tune n_{tree} for different t, we propose to adaptively regularize each model using early stopping. This could be done with a held-out validation set, but since many benchmark tabular datasets are small to begin with, reducing the training set further is disadvantageous. Instead, because the regression targets for both diffusion and CFM are generated by adding noise to the training data, we validate using the training set X_0 but with fresh noise X_1. An example of how the best iteration varies with timestep is shown in Figure 3, stopping after $n_{ES} = 20$ trees without improvement. Coupling a very large $n_{tree} = 2000$ with early stopping proves to be an effective regularizer as it allows wide ensembles for the more challenging training iterations while preventing overfitting on the easier steps. We show below in Section 4.1 that adding early stopping to the default settings greatly speeds up training, as it avoids wasted computation for ensembles which have converged.

4 Experimental Results

In this section we demonstrate the improved resource scaling of our implementation, provide benchmarking on small tabular datasets for model performance improvements, then present our results on the much larger calorimeter datasets. Complete details on datasets and hyperparameters are given in Appendix D for reproducibility.
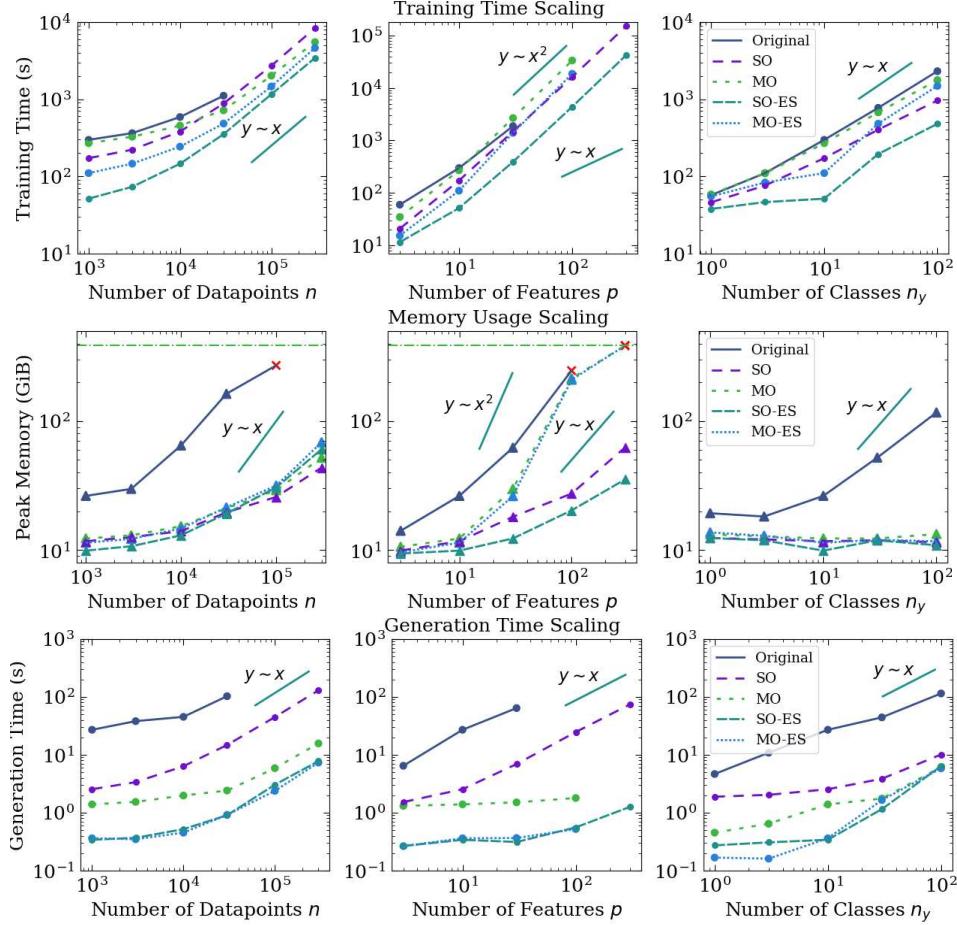
Figure 4: Resource usage of the ForestFlow implementation from [46], compared to our implementation (SO), including with multi-output trees (MO), and early stopping (ES). **Top:** Training time. **Middle:** Peak memory usage. **Bottom:** Generation time. A red cross $\times$ for memory indicates job failure, and hence corresponding points in other plots are unavailable. A horizontal line indicates the maximum system memory used for all models at 385 GiB.

4.1 Resource Usage Scaling

We quantify resource usage using synthetic datasets of controllable size. Features and labels are randomly generated, with one of n, p, or n_y altered from base values of $n = 1000$, $p = 10$, and $n_y = 10$. Increasing n should increase training time and memory usage, but at most linearly due to XGBoost's `hist` method [17]. Increasing n_y with n held fixed means increasing the number of ensembles trained, but with each on a smaller dataset, hence memory usage should slightly decrease. Meanwhile, increasing p leads to quadratic scaling in time since it increases the number of ensembles required, and the size of each dataset, the latter of which implies linear memory requirements.

Figure 4 shows our controlled experiments for training time, memory, and generation time scaling. We compare the Original implementation of ForestFlow to ours (SO) which produces the same models. Additionally, we show the multi-output (MO) variant with the same learning hyperparameters, and our proposal for adaptive early stopping (SO-ES, MO-ES) which trains much smaller models. Each plot is on a log-log scale, and we display line segments with linear or quadratic behaviour so that scaling exponents can easily be read off.[3]

---

[3]In some cases the constant overhead can be significant, and the independent variable cannot be made large enough to reach the asymptotic regime. For example, in the second row, first column, memory scaling may appear sublinear for methods other than Original due to the constant, but is actually linear.

Table 2: Average rank (standard error) of generated data quality over 27 datasets. Lower is better.

	$W1_{train}$	$W1_{test}$	Cov_{train}	Cov_{test}	R^2_{gen}	$F1_{gen}$	P_{bias}	cov_{rate}	Avg.
GaussianCopula [45]	10.1±0.3	10.2±0.3	10.2±0.3	10.3±0.3	9.1±0.1	9.6±0.3	8.6±1.4	10.6±0.5	9.8±0.1
TVAE [95]	8.1±0.3	8.0±0.3	8.5±0.3	8.5±0.3	9.7±0.6	9.0±0.5	10.7±0.5	10.1±0.3	9.1±0.0
CTGAN [95]	11.4±0.1	11.4±0.2	11.3±0.2	11.1±0.2	11.6±0.2	11.4±0.2	8.4±1.2	10.6±0.5	10.9±0.1
CTAB-GAN+ [101]	9.8±0.3	9.7±0.3	10.1±0.3	9.9±0.3	10.0±0.2	10.1±0.3	10.7±0.6	9.3±0.9	10.0±0.1
STaSy [49]	9.1±0.2	9.3±0.2	8.1±0.3	8.0±0.4	9.0±1.0	8.0±0.3	7.4±1.1	7.4±1.1	8.3±0.2
TabDDPM [54]	4.9±0.9	6.4±0.7	4.9±0.7	5.4±0.6	2.0±0.7	6.2±0.8	3.3±1.4	3.4±0.6	4.6±0.1
FD-Original [46]	6.7±0.1	6.3±0.2	6.7±0.2	5.9±0.5	5.7±0.7	4.1±0.5	6.7±1.1	5.7±1.2	6.0±0.1
FD-SO-Scaled	3.8±0.3	4.1±0.3	4.3±0.3	4.4±0.3	5.6±0.4	3.4±0.5	4.6±0.7	4.6±1.1	4.4±0.1
FD-MO-Scaled	4.5±0.2	4.2±0.3	5.0±0.3	4.8±0.4	5.7±0.4	4.6±0.4	5.1±0.8	5.9±0.7	5.0±0.1
FF-Original [46]	5.0±0.2	4.3±0.3	4.0±0.3	3.9±0.4	4.3±0.8	4.8±0.5	3.6±0.8	3.8±0.7	4.2±0.1
FF-SO-Scaled	1.7±0.2	2.3±0.2	2.1±0.3	2.9±0.3	2.1±0.1	3.1±0.5	4.4±0.5	3.0±0.5	2.7±0.0
FF-MO-Scaled	2.8±0.2	1.9±0.2	2.7±0.3	3.0±0.3	3.1±0.7	3.8±0.4	4.4±0.7	3.6±0.8	3.2±0.1

All methods show linear training time in n as nearly all compute time is spent in calls to XGBoost. However, SO reduces overhead which can be up to half the training time for small n. Meanwhile, early stopping provides up to a $3\times$ speedup for SO-ES over SO, and a smaller but still useful speedup for MO-ES over MO (see Figure 10 in Appendix C.2 for an illustration of early stopping's effect).

Our implementation greatly reduces memory overhead, and peak memory usage scales linearly with n. The Original implementation shows worse-than-linear scaling (after accounting for the overhead), and leads to job failures with as few as $n = 10,000$ datapoints. The number of features p has the biggest impact on resource use, but our implementation reduces the memory scaling from quadratic to linear. Finally, for n_y our implementation uses a constant amount of memory, whereas the Original scales linearly. While MO behaves similarly to SO in n and n_y, MO suffers from worse scaling in p. Multi-output trees reduce the number of ensembles required by a factor of p, but each tree's training is more memory intensive as XGBoost must search over a higher dimensional leaf space.

The bottom of Figure 4 shows the time required to generate 5 batches of n datapoints using the corresponding models trained for the other plots. Not only is our implementation more than an order of magnitude faster for most settings, we see an even greater improvement for multi-output trees. The near-constant scaling in p showcases the benefit of generating all p outputs with a single ensemble making multi-output trees a strong candidate for applications that require large volumes of generated data. Since the ES variants drastically reduce the trained model size, generation time is commensurately reduced.

In summary, our implementation and algorithmic improvements together improve the efficiency of ForestDiffusion and ForestFlow for all major resources – training time, memory, and generation time – which allows us to scale up to much larger models and datasets in the following sections.

4.2 Model Performance on Benchmark Datasets

We scaled up models by increasing the duplication factor K from 100 to 1000, and the maximum number of trees n_{tree} from 100 to 2000. In this section, we directly compare scaled-up models using our proposed algorithmic improvements from Section 3.4 to the original ForestDiffusion (FD) and ForestFlow (FF) on 27 datasets [69], across 8 metrics, which we averaged over 5 generated datasets for each of 3 seeds. The metrics convey the quality of generated samples along four dimensions: distributional distance (Wasserstein-1 distance to the training or test set), diversity (Coverage [70] of the training or test set), usefulness for training discriminative models (R^2_{gen} and $F1_{gen}$), and usefulness for statistical inference (P_{bias} and cov_{rate}). For comparison, 6 baseline generative models are shown ranging from statistical methods to tabular diffusion models. The 27 datasets, 8 evaluation metrics, and 6 baseline methods are repeated from [46], and described fully in Appendix D along with exact hyperparameter settings in Table 9, and experimental details.

Table 2 shows the average rank that each method obtained on each metric where the average and standard error are computed over the 27 datasets, similar to previous tabular generation papers [36, 37]. We also include "Avg.", the row-wise average and standard error to summarize the table, and highlight the single best method on each metric. While the original ForestFlow already outperforms advanced methods like TabDDPM [54] on several metrics, our performance improvements and scaling-up of the single-output case further establish it as a state-of-the-art tabular generative model. Multi-output trees are also competitive when scaled up, and notably achieve the lowest Wasserstein

Table 3: Model performance on calorimeter data. Lower is better.

Photons	AUC	$E_{\text{dep}}/E_{\text{inc}}$	$E_{\text{dep,L0}}$	$CE_{\eta,\text{L1}}$	$CE_{\phi,\text{L1}}$	$\text{Width}_{\eta,\text{L1}}$	$\text{Width}_{\phi,\text{L1}}$
CaloMan [20]	0.9998	0.0020	0.0001	0.0462	0.0394	0.0366	0.0865
CaloForest (Ours)	0.8392	0.0778	0.0033	0.0056	0.0029	0.0241	0.0228

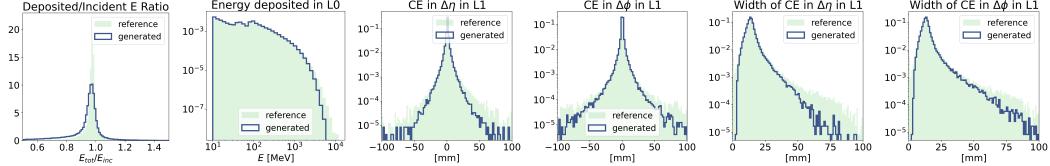| **Pions**	AUC	$E_{\text{dep}}/E_{\text{inc}}$	$E_{\text{dep,L0}}$	$CE_{\eta,\text{L1}}$	$CE_{\phi,\text{L1}}$	$\text{Width}_{\eta,\text{L1}}$	$\text{Width}_{\phi,\text{L1}}$
CaloMan [20]	0.9986	0.0404	0.0002	0.0477	0.0282	0.2380	0.2183
CaloForest (Ours)	0.9119	0.0625	0.0384	0.0268	0.0266	0.1935	0.1978

Figure 5: Histograms of high-level features comparing generated Photons samples to the test set. Note the log scale of the y-axis for all but the first plot.

distance to the test set, showing the best ability to capture the underlying data distribution. Raw metric values are plotted in Appendix D.5.

4.3 CaloForest - Flow-based XGBoost Models for Calorimeter Data

To show that our implementation scales to much larger datasets, we model Photons and Pions (Table 1) from the Fast Calorimeter Simulation Challenge [26]. Each dataset comes with training and test splits of size $\approx 121,000$, where each datapoint represents the energies deposited in voxels of a calorimeter by an incident particle.

Competitive NN-based approaches perform extensive pre-processing to the data to facilitate training [56, 68, 20]. Since XGBoost is robust to features on different scales and with different distributions we need only perform min-max scaling on each class. We used our SO variant of ForestFlow as the MO variant still has high memory costs for large p (Figure 4). We discretized time into $n_t = 100$ steps, and duplicated each datapoint $K = 20$ times. Each XGBoost ensemble had $n_{\text{tree}} = 20$ trees of maximum depth 7, a learning rate of 1.5, and all other XGBoost hyperparameters left as defaults. We trained up to 20 XGBoost ensembles in parallel, each with 2 CPUs, on a single desktop workstation with 250 GiB RAM and 40 CPUs (Intel Xeon Silver 4114T). In total, for the Photons model 552,000 XGBoost ensembles were trained in 135 hours with a peak memory burden of 54 GiB, while the Pions model used 799,500 ensembles, completed in 281 hours, and required 78 GiB of memory. Generation of n datapoints (matching the number in the training and test sets) took 231 s for Photons (1.91 ms per datapoint), and 347 s for Pions (2.87 ms per datapoint), which can be compared to 40 ms per datapoint for score-based NNs on a GPU [68], or anywhere from 100 ms to 3 s for the widely used Geant4 simulator [2].

The Challenge uses three types of metrics to evaluate models: resource usage, especially generation time as discussed above; distributional closeness to the test set judged by the χ^2 separation power between histograms in features crafted by domain experts; and ROCAUC of a binary classifier trained on a mix of real and generated data. We describe these metrics in full detail in Appendix A.1. The latter two types of metric are shown in Table 3 as compared to a NN-based approach [20]. We see that ForestFlow produces more "realistic" datapoints in that they are harder for a classifier to distinguish from the test set. Feature histograms are shown in Figure 5 which confirms an accurate representation of the true distribution. Complete results for all metrics are given in Appendix A.2).

5 Conclusions and Limitations

In this work, we have pushed the boundaries of tabular data generation backed not by neural networks, but by XGBoost. As discussed, XGBoost offers several tantalizing advantages over NNs for generative modelling: XGBoost's better performance on discriminative tabular tasks may translate to better tabular generation; it is robust without data pre-processing; it natively handles missing values; it can be efficiently trained on CPU; and finally it offers improved explainability. However, the great

differences in the mechanics of XGBoost training compared to NNs led Jolicoeur-Martineau et al. [46] to propose overparameterized models that do achieve state-of-the-art performance, but potentially at the cost of practicality and scalability. Our contributions re-engineered the inner workings of these models leading to faster training and generation, as well as much lower peak memory burdens and better scaling with dataset size, allowing us to train on datasets $370\times$ bigger than previously tested. The improved implementation unlocked the ability to scale up model size which directly led to improved performance when properly regularized. Finally, we proposed the use of multi-output trees which are more suited to the high-dimensional outputs required in generative modelling, and showed that generation time can be reduced by an additional order of magnitude for applications that require large volumes of generated data. Still, it is clear that the methods we discuss have limitations. Models trained with ForestDiffusion and ForestFlow are highly overparameterized (Section 4.3), still require significant computational resources especially when p is large (Figure 4), and our proposal to use multi-output trees comes at a cost of somewhat slower and more memory-intensive training (Figure 4).

References

[1] Congressional Voting Records. UCI Machine Learning Repository, 1987. DOI: https://doi.org/10.24432/C5C01P.

[2] Georges Aad, Alexander Kupco, Jay Chan, Miguel Angel Principe Martin, Pierre Antoine Delsart, Timo Dreyer, Yufeng Wang, Karl Jakobs, Ana Maria Rodriguez Vera, Savanna Shaw, et al. AtlFast3: the next generation of fast simulation in ATLAS. Technical report, ATLAS-SIMU-2018-04-003, 2021.

[3] Stefan Aeberhard and M. Forina. Wine. UCI Machine Learning Repository, 1991. DOI: https://doi.org/10.24432/C5PC7J.

[4] S. Agostinelli et al. Geant4—a simulation toolkit. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, 2003. ISSN 0168-9002. doi: https://doi.org/10.1016/S0168-9002(03) 01368-8.

[5] David Aha. Tic-Tac-Toe Endgame. UCI Machine Learning Repository, 1991. DOI: https://doi.org/10.24432/C5688J.

[6] Michael S Albergo and Eric Vanden-Eijnden. Building normalizing flows with stochastic interpolants. In *International Conference on Learning Representations*, 2023.

[7] J. Allison et al. Geant4 developments and applications. *IEEE Transactions on Nuclear Science*, 53(1):270–278, 2006. doi: 10.1109/TNS.2006.869826.

[8] J. Allison et al. Recent developments in Geant4. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 835: 186–225, 2016. ISSN 0168-9002. doi: https://doi.org/10.1016/j.nima.2016.06.125.

[9] Oz Amram and Kevin Pedro. CaloDiffusion with GLaM for High Fidelity Calorimeter Simulation. *arXiv:2308.03876*, 2023.

[10] Rajen Bhatt. Planning Relax. UCI Machine Learning Repository, 2012. DOI: https://doi.org/10.24432/C5T023.

[11] Marko Bohanec. Car Evaluation. UCI Machine Learning Repository, 1997. DOI: https://doi.org/10.24432/C5JP48.

[12] Thomas Brooks, D. Pope, and Michael Marcolini. Airfoil Self-Noise. UCI Machine Learning Repository, 2014. DOI: https://doi.org/10.24432/C5VW2C.

[13] Matthew R. Buckley, Claudius Krause, Ian Pang, and David Shih. Inductive CaloFlow. *arXiv:2305.11934*, 2023.

[14] Erik Buhmann, Sascha Diefenbacher, Engin Eren, Frank Gaede, Gregor Kasieczka, Anatolii Korol, William Korcari, Katja Krüger, and Peter McKeown. CaloClouds: Fast geometry-independent highly-granular calorimeter simulation. *JINST*, 18(11):P11025, 2023. doi: 10.1088/1748-0221/18/11/P11025.

[15] Magorzata Charytanowicz, Jerzy Niewczas, Piotr Kulczycki, Piotr Kowalski, and Szymon Lukasik. seeds. UCI Machine Learning Repository, 2012. DOI: https://doi.org/10.24432/C5H30K.

[16] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, volume 31, 2018.

[17] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016. ISBN 9781450342322.

[18] Ting Chen. On the importance of noise scheduling for diffusion models. *arXiv:2301.10972*, 2023.

[19] Paulo Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis. Wine Quality. UCI Machine Learning Repository, 2009. DOI: https://doi.org/10.24432/C56S3T.

[20] Jesse C Cresswell, Brendan Leigh Ross, Gabriel Loaiza-Ganem, Humberto Reyes-Gonzalez, Marco Letizia, and Anthony L Caterini. CaloMan: Fast generation of calorimeter showers with density estimation on learned manifolds. *Machine Learning and the Physical Sciences Workshop at NeurIPS 2022*, 2022.

[21] David Deterding, Mahesan Niranjan, and Tony Robinson. Connectionist Bench (Vowel Recognition - Deterding Data). UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C58P4S.

[22] Daniel Dias, Sarajane Peres, and Helton Bscaro. Libras Movement. UCI Machine Learning Repository, 2009. DOI: https://doi.org/10.24432/C5GC82.

[23] Sascha Diefenbacher, Engin Eren, Frank Gaede, Gregor Kasieczka, Claudius Krause, Imahn Shekhzadeh, and David Shih. L2LFlows: Generating high-fidelity 3D calorimeter images. *JINST*, 18(10):P10017, 2023. doi: 10.1088/1748-0221/18/10/P10017.

[24] Florian Ernst, Luigi Favaro, Claudius Krause, Tilman Plehn, and David Shih. Normalizing Flows for High-Dimensional Detector Simulations. *arXiv:2312.09290*, 2023.

[25] Michele Faucci Giannelli and Rui Zhang. CaloShowerGAN, a Generative Adversarial Networks model for fast calorimeter shower simulation. *arXiv:2309.06515*, 2023.

[26] Michele Faucci Giannelli, Gregor Kasieczka, Claudius Krause, Ben Nachman, Dalila Salamani, David Shih, and Anna Zaborowska. Fast Calorimeter Simulation Challenge 2022, 2022. URL https://calochallenge.github.io/homepage/.

[27] Luigi Favaro, Ayodele Ore, Sofia Palacios Schweitzer, and Tilman Plehn. Calodream - detector response emulation via attentive flow matching. *arXiv:2405.09629*, 2024.

[28] R. A. Fisher. Iris. UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C56C76.

[29] Rémi Flamary, Nicolas Courty, Alexandre Gramfort, Mokhtar Z. Alaya, Aurélie Boisbunon, Stanislas Chambon, Laetitia Chapel, Adrien Corenflos, Kilian Fatras, Nemo Fournier, Léo Gautheron, Nathalie T.H. Gayraud, Hicham Janati, Alain Rakotomamonjy, Ievgen Redko, Antoine Rolet, Antony Schutz, Vivien Seguy, Danica J. Sutherland, Romain Tavenard, Alexander Tong, and Titouan Vayer. POT: Python Optimal Transport. *Journal of Machine Learning Research*, 22(78):1–8, 2021.

[30] Yoav Freund and Robert E. Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory*, pages 23–37, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-540-49195-8.

[31] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting (With discussion and a rejoinder by the authors). *The Annals of Statistics*, 28(2):337 – 407, 2000. doi: 10.1214/aos/1016218223.

[32] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5), 2001. doi: 10.1214/aos/1013203451.

[33] B. German. Glass Identification. UCI Machine Learning Repository, 1987. DOI: https://doi.org/10.24432/C5WW2P.

[34] J. Gerritsma, R. Onnink, and A. Versluis. Yacht Hydrodynamics. UCI Machine Learning Repository, 2013. DOI: https://doi.org/10.24432/C5XG7R.

[35] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, 2014.

[36] Yury Gorishniy, Ivan Rubachev, Valentin Khrulkov, and Artem Babenko. Revisiting deep learning models for tabular data. In *Advances in Neural Information Processing Systems*, volume 34, pages 18932–18943, 2021.

[37] Yury Gorishniy, Ivan Rubachev, Nikolay Kartashev, Daniil Shlenskii, Akim Kotelnikov, and Artem Babenko. TabR: Unlocking the power of retrieval-augmented tabular deep learning. *arXiv:2307.14338*, 2023.

[38] Leo Grinsztajn, Edouard Oyallon, and Gael Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? In *Advances in Neural Information Processing Systems*, volume 35, 2022.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, June 2016.

[40] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33:6840–6851, 2020.

[41] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282 vol.1, 1995. doi: 10.1109/ICDAR.1995.598994.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9 (8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.

[43] Aapo Hyvärinen. Estimation of non-normalized statistical models by score matching. *Journal of Machine Learning Research*, 6(24):695–709, 2005.

[44] Leonid Iosipoi and Anton Vakhrushev. Sketchboost: Fast gradient boosted decision tree for multioutput problems. In *Advances in Neural Information Processing Systems*, 2022.

[45] H. Joe. *Dependence Modeling with Copulas*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 2014. ISBN 9781466583221.

[46] Alexia Jolicoeur-Martineau, Kilian Fatras, and Tal Kachman. Generating and imputing tabular data via diffusion and flow-based gradient-boosted trees. In *Proceedings of The 27th International Conference on Artificial Intelligence and Statistics*, 2024.

[47] Benno Käch and Isabell Melzer-Pellmann. Attention to Mean-Fields for Particle Cloud Generation. *arXiv:2305.15254*, 2023.

[48] Markelle Kelly, Rachel Longjohn, and Kolby Nottingham. The UCI Machine Learning Repository. https://archive.ics.uci.edu. Accessed: 2023-12-09.

[49] Jayoung Kim, Chaejeong Lee, and Noseong Park. STaSy: Score-based Tabular data Synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.

[50] Diederik P Kingma and Max Welling. Auto-encoding Variational Bayes. *ICLR*, 2014.

[51] Thomas N Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*, 2016.

[52] Dmitrii Kobylianskii, Nathalie Soybelman, Etienne Dreyer, and Eilam Gross. CaloGraph: Graph-based diffusion model for fast shower generation in calorimeters with irregular geometry. *arXiv:2402.11575*, 2024.

[53] Murat Koklu and Ilker Ali Ozkan. Multiclass classification of dry beans using computer vision and machine learning techniques. *Computers and Electronics in Agriculture*, 174: 105507, 2020.

[54] Akim Kotelnikov, Dmitry Baranchuk, Ivan Rubachev, and Artem Babenko. TabDDPM: Modelling Tabular Data with Diffusion Models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 17564–17579. PMLR, 23–29 Jul 2023.

[55] Claudius Krause and David Shih. CaloFlow: Fast and Accurate Generation of Calorimeter Showers with Normalizing Flows. *arXiv:2106.05285*, 2021.

[56] Claudius Krause and David Shih. CaloFlow II: Even Faster and Still Accurate Generation of Calorimeter Showers with Normalizing Flows. *arXiv:2110.11377*, 2021.

[57] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25, 2012.

[58] Yaron Lipman, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matthew Le. Flow matching for generative modeling. In *International Conference on Learning Representations*, 2023.

[59] Max Little. Parkinsons. UCI Machine Learning Repository, 2008. DOI: https://doi.org/10.24432/C59C74.

[60] Xingchao Liu, Chengyue Gong, and Qiang Liu. Flow straight and fast: Learning to generate and transfer data with rectified flow. In *International Conference on Learning Representations*, 2023.

[61] Gabriel Loaiza-Ganem, Brendan Leigh Ross, Rasa Hosseinzadeh, Anthony L Caterini, and Jesse C Cresswell. Deep generative models through the lens of the manifold hypothesis: A survey and new connections. *arXiv:2404.02954*, 2024.

[62] DD Lucas, R Klein, J Tannahill, D Ivanova, S Brandon, D Domyancic, and Y Zhang. Climate Model Simulation Crashes. UCI Machine Learning Repository, 2013. DOI: https://doi.org/10.24432/C5HG71.

[63] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[64] Scott M Lundberg, Gabriel G Erion, and Su-In Lee. Consistent individualized feature attribution for tree ensembles. *arXiv:1802.03888*, 2018.

[65] Kamel Mansouri, Tine Ringsted, Davide Ballabio, Roberto Todeschini, and Viviana Consonni. QSAR biodegradation. UCI Machine Learning Repository, 2013. DOI: https://doi.org/10.24432/C5H60M.

[66] Alexander März. Multi-Target XGBoostLSS Regression. *arXiv:2210.06831*, 2022.

[67] Duncan McElfresh, Sujay Khandagale, Jonathan Valverde, Ganesh Ramakrishnan, Micah Goldblum, Colin White, et al. When do neural nets outperform boosted trees on tabular data? In *Advances in Neural Information Processing Systems*, volume 36, 2023.

[68] Vinicius Mikuni and Benjamin Nachman. Score-based generative models for calorimeter shower simulation. *Phys. Rev. D*, 106(9):092009, 2022. doi: 10.1103/PhysRevD.106.092009.

[69] Boris Muzellec, Julie Josse, Claire Boyer, and Marco Cuturi. Missing data imputation using optimal transport. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 7130–7140. PMLR, 13–18 Jul 2020.

[70] Muhammad Ferjad Naeem, Seong Joon Oh, Youngjung Uh, Yunjey Choi, and Jaejun Yoo. Reliable fidelity and diversity metrics for generative models. In *International Conference on Machine Learning*, pages 7176–7185. PMLR, 2020.

[71] Kenta Nakai. Ecoli. UCI Machine Learning Repository, 1996. DOI: https://doi.org/10.24432/C5388M.

[72] Kenta Nakai. Yeast. UCI Machine Learning Repository, 1996. DOI: https://doi.org/10.24432/C5KG68.

[73] Alexander Quinn Nichol and Prafulla Dhariwal. Improved denoising diffusion probabilistic models. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139, pages 8162–8171, 2021.

[74] Richard Nock and Mathieu Guillame-Bert. Generative trees: Adversarial and copycat. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162, pages 16906–16951, 2022.

[75] Richard Nock and Mathieu Guillame-Bert. Generative forests. *arXiv:2308.03648*, 2023.

[76] R Kelley Pace and Ronald Barry. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3):291–297, 1997.

[77] Michela Paganini, Luke de Oliveira, and Benjamin Nachman. CaloGAN: Simulating 3D high energy particle showers in multilayer electromagnetic calorimeters with generative adversarial networks. *Phys. Rev. D*, 97:014021, Jan 2018. doi: 10.1103/PhysRevD.97.014021.

[78] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. The synthetic data vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics*, pages 399–410, 2016. doi: 10.1109/DSAA.2016.49.

[79] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[80] Moritz Alfons Wilhelm Scham, Dirk Krücker, Benno Käch, and Kerstin Borras. DeepTreeGAN: Fast Generation of High Dimensional Point Clouds. *arXiv:2311.12616*, 2023.

[81] Lena Schmid, Alexander Gerharz, Andreas Groll, and Markus Pauly. Tree-based ensembles for multi-output regression: Comparing multivariate approaches with separate univariate ones. *Computational Statistics & Data Analysis*, 179:107628, 2023. ISSN 0167-9473. doi: 10.1016/j.csda.2022.107628.

[82] Simon Schnake, Dirk Krücker, and Kerstin Borras. CaloPointFlow II Generating Calorimeter Showers as Point Clouds. *arXiv:2403.15782*, 2024.

[83] Terry Sejnowski and Paul R Gorman. Connectionist Bench (Sonar, Mines vs. Rocks). UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5T01Q.

[84] Lloyd S Shapley. Notes on the n-person game—ii: The value of an n-person game. 1951.

[85] Vincent G Sigillito, Simon P Wing, Larrie V Hutton, and Kile B Baker. Ionosphere. UCI Machine Learning Repository, 1989. DOI: https://doi.org/10.24432/C5W01B.

[86] M. Sklar. *Fonctions de Répartition À N Dimensions Et Leurs Marges*. Université Paris 8, 1959.

[87] Yang Song and Stefano Ermon. Generative Modeling by Estimating Gradients of the Data Distribution. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[88] Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. In *International Conference on Learning Representations*, 2021.

[89] George Stein, Jesse Cresswell, Rasa Hosseinzadeh, Yi Sui, Brendan Ross, Valentin Villecroze, Zhaoyan Liu, Anthony L Caterini, Eric Taylor, and Gabriel Loaiza-Ganem. Exposing flaws of generative model evaluation metrics and their unfair treatment of diffusion models. In *Advances in Neural Information Processing Systems*, volume 36, pages 3732–3784, 2023.

[90] Alexander Tong, Kilian Fatras, Nikolay Malkin, Guillaume Huguet, Yanlei Zhang, Jarrid Rector-Brooks, Guy Wolf, and Yoshua Bengio. Improving and generalizing flow-based generative models with minibatch optimal transport. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856.

[91] S. van Buuren. *Flexible Imputation of Missing Data*. Chapman & Hall/CRC Interdisciplinary Statistics. CRC Press, 2nd edition, 2018. ISBN 9780429960352.

[92] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[93] Pascal Vincent. A connection between score matching and denoising autoencoders. *Neural Computation*, 23(7):1661–1674, 2011.

[94] William Wolberg, Olvi Mangasarian, Nick Street, and W. Street. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository, 1995. DOI: https://doi.org/10.24432/C5DW2B.

[95] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Modeling Tabular data using Conditional GAN. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[96] I-Cheng Yeh. Concrete Compressive Strength. UCI Machine Learning Repository, 2007. DOI: https://doi.org/10.24432/C5PK67.

[97] I-Cheng Yeh. Blood Transfusion Service Center. UCI Machine Learning Repository, 2008. DOI: https://doi.org/10.24432/C5GS39.

[98] I-Cheng Yeh. Concrete Slump Test. UCI Machine Learning Repository, 2009. DOI: https://doi.org/10.24432/C5FG7D.

[99] ZhenZhe Ying, Zhuoer Xu, Zhifeng Li, Weiqiang Wang, and Changhua Meng. MT-GBM: A multi-task gradient boosting machine with shared decision trees. *arXiv:2201.06239*, 2022.

[100] Zhendong Zhang and Cheolkon Jung. GBDT-MO: Gradient-Boosted Decision Trees for Multiple Outputs. *IEEE Transactions on Neural Networks and Learning Systems*, 32(7):3156–3167, 2021. doi: 10.1109/TNNLS.2020.3009776.

[101] Zilong Zhao, Aditya Kunar, Robert Birke, Hiek Van der Scheer, and Lydia Y. Chen. CTAB-GAN+: Enhancing Tabular Data Synthesis. *Frontiers in Big Data*, 6, 2024. ISSN 2624-909X. doi: 10.3389/fdata.2023.1296508.
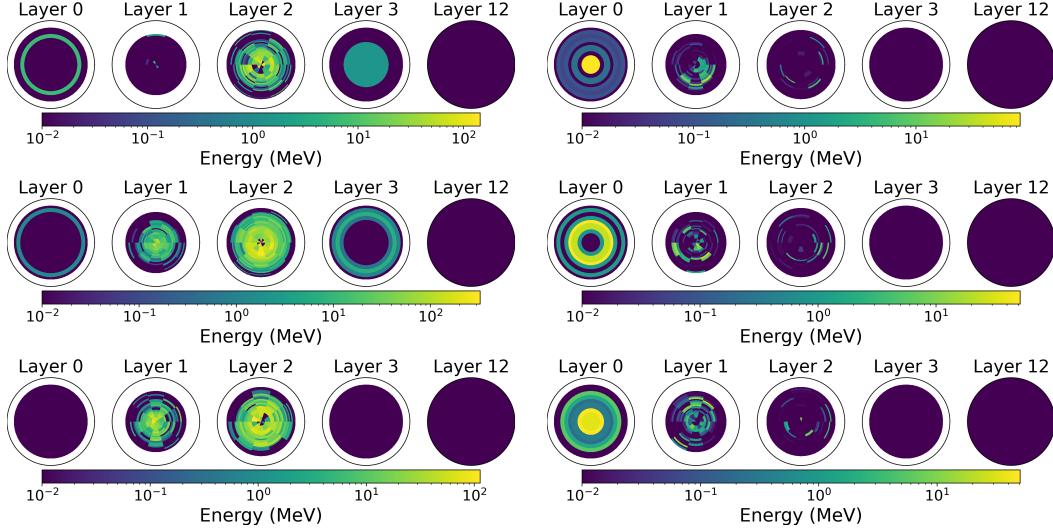
Figure 6: Individual showers shown as energy deposited per voxel for the Photons test dataset (left), and generated by CaloForest (right). Note the nested cylindrical geometry of voxels which is inconsistent across layers, meaning the data must be treated as tabular, rather than as images.

A   Fast Calorimeter Simulation

Particle accelerator experiments in high energy physics utilise several components in their detectors to measure properties of particles created in collisions. Calorimeters are one component that measure the energy of particles. Upon entering the calorimeter, the incident particle begins interacting with the material of the calorimeter and progressively deposits its energy. The interactions form a branching tree-like structure called a shower. Energy deposits are measured in an array of voxels allowing the 3d reconstruction of shower shape. Since nature is inherently probabilistic, a given incident particle with fixed energy gives rise to a probability distribution of possible showers.

Physicists desire to sample from these distributions as one component of detector simulation. By simulating detector responses using known theory, physicists can define their prior for what is expected to be measured when the actual experiment is run. Measured deviations from the prior expectation may indicate new physics, leading to a deeper understanding of nature.

However, sampling calorimeter showers using precise simulation of physical processes from first-principles is incredibly slow. Currently, simulations at the largest particle accelerator, the Large Hadron Collider (LHC), are done with Geant4 [4, 7, 8], which is CPU-based and can take upwards of ten minutes per shower. Since billions of simulations are needed to provide accurate background statistics, the computational burden of exact simulation is immense.

Generative modelling heralds a solution by directly generating showers using surrogate models instead of simulating them from first-principles. The first method in this line of research used GANs [77], eventually leading to actual deployment of GAN-based generators in the experimental pipeline of the LHC [2]. Following work explored other deep generative techniques like normalizing flows [55, 56].

The initial success of these methods at reducing simulation time, while accurately representing the distribution of showers, led to the public release of large-scale training datasets and a call to the community to explore new methods in the Fast Calorimeter Simulation Challenge [26]. The four datasets involve different types of particles incident on the calorimeter. The datasets represent the calorimeters with a cylindrical pattern of voxels, and each datapoint's features represent the energy deposited in one voxel. This allows the visualization of individual showers (Figure 6), as well as averages across the dataset (Figure 7).

Early submissions to the Challenge branched out to test score-based methods [68] and models that learned the low-dimensional structure of showers [20]. Overall, submissions can be classified by the generative modelling paradigm they build off of, with GANs [25, 47, 80], normalizing flows
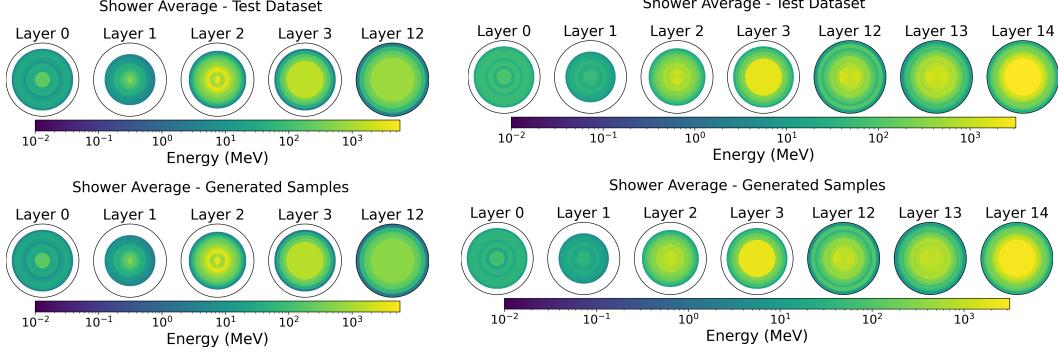
Figure 7: Average deposited energy per voxel for Photons (left) and Pions (right) on the test dataset (top), and samples generated conditionally using the test set class distribution (bottom).

[23, 13, 24, 82], diffusion models [9, 14, 52], and conditional flow matching [27] being popular choices.

Notably, every prior submission to the Fast Calorimeter Simulation Challenge uses deep neural networks as function approximators. This is despite the need for GPU resources to train and generate with NNs, whereas existing scientific computing infrastructure for shower simulation is largely CPU-based. Our method, CaloForest, provides an alternative, as it is the only attempt to use tree-based approximators for the Challenge's large-scale tabular datasets.

A.1 Evaluation Metrics

Since the generated showers are meant to be used in actual scientific experiments, custom evaluation metrics have been defined for the Challenge using domain knowledge.

First, computational resources are important (hence "Fast" in the Challenge's title). Training should be accomplished with as little time and memory as possible, but the most important resource metric is shower generation time, as billions of generated showers will be needed in practice. Hence, we track the training time, generation time, and peak memory usage during training. These results are given in Section 4.3.

Second, generated showers must accurately represent the actual distribution of showers predicted by theory. Calculating this distribution in closed form from theory is not feasible, so instead the ground truth is taken from theory-based simulations using Geant4. A test dataset of showers generated in the same way as the training data is provided with each of the Challenge's datasets. Using domain knowledge, physicists defined high-level features from voxel-level information. The one-dimensional distributions of each feature can be compared between the test set and a generated set using the χ^2 separation power between histograms which is defined as

$$\chi^2(h_1, h_2) = \frac{1}{2} \sum_i \frac{(h_{1,i} - h_{2,1})^2}{h_{1,i} + h_{2,i}}, \tag{7}$$

where $h_{j,i}$ is the fraction of all datapoints falling into bin i of histogram j, such that $\sum_i h_{j,i} = 1$. The metric is normalized such that $\chi^2(h_1, h_2) = 0$ if and only if the histograms are the same, $h_1 = h_2$, whereas when the histograms have no overlap $\chi^2(h_1, h_2) = 1$. The high-level features denote the ratio of deposited energy to incident energy, the total deposited energy in each layer of the calorimeter, the center of energy in angular directions η and ϕ per layer, and the width of the center of energy in angular directions per layer. Example χ^2 metrics are given in Table 3 with example histogram plots in Figures 5 and 8, while the complete lists of metrics are shown below in Tables 4 and 5.

Third, a binary classifier is trained to distinguish generated showers from the test set using the architecture and training details provided by the Challenge [26]. The ROCAUC of the classifier on a balanced, held-out set of samples should be as low as possible, indicating that generated samples are indistinguishable from test datapoints. We present the ROCAUC metrics in Table 3.

18

Table 4: Photons dataset histogram χ^2 separation powers in domain expert features. L denotes layer. CE is the center of energy. Lower is better.

FEATURE	CaloMan [20]	CaloForest
$E_{\text{dep}}/E_{\text{inc}}$	0.0020	0.0778
E_{dep}, L0	0.00005	0.0033
E_{dep}, L1	0.00008	0.0036
E_{dep}, L2	0.00002	0.0031
E_{dep}, L3	0.00001	0.0018
E_{dep}, L12	0.00002	0.0037
CE in η, L1	0.0462	0.0056
CE in η, L2	0.0419	0.0014
CE in ϕ, L1	0.0394	0.0029
CE in ϕ, L2	0.0367	0.0017
Width in CE in η, L1	0.0366	0.0241
Width in CE in η, L2	0.0696	0.0108
Width in CE in ϕ, L1	0.0865	0.0228
Width in CE in ϕ, L2	0.0649	0.0097

Table 5: Pions dataset histogram χ^2 separation powers in domain expert features. L denotes layer. CE is the center of energy. Lower is better.

FEATURE	CaloMan [20]	CaloForest
$E_{\text{dep}}/E_{\text{inc}}$	0.0404	0.0625
E_{dep}, L0	0.0002	0.0384
E_{dep}, L1	0.0347	0.1440
E_{dep}, L2	0.0052	0.0532
E_{dep}, L3	0.0001	0.0178
E_{dep}, L12	0.0008	0.0046
E_{dep}, L13	0.0001	0.0102
E_{dep}, L14	0.0002	0.0085
CE in η, L1	0.0477	0.0268
CE in η, L2	0.0808	0.0168
CE in η, L12	0.0477	0.0641
CE in η, L13	0.0808	0.1377
CE in ϕ, L1	0.0282	0.0266
CE in ϕ, L2	0.0240	0.0155
CE in ϕ, L12	0.0282	0.0573
CE in ϕ, L13	0.0240	0.1203
Width in CE in η, L1	0.2380	0.1935
Width in CE in η, L2	0.2074	0.1121
Width in CE in η, L12	0.2380	0.1758
Width in CE in η, L13	0.2074	0.2384
Width in CE in ϕ, L1	0.2183	0.1978
Width in CE in ϕ, L2	0.2067	0.1141
Width in CE in ϕ, L12	0.2183	0.1788
Width in CE in ϕ, L13	0.2067	0.2399

A.2 Extended Results

Here we present complete information on the histogram evaluation metrics for the Challenge that are obtained with our method CaloForest. Tables 4 and 5 show the χ^2 separation powers for histograms of the generated and test set samples. Compared to a NN-based approach designed for the challenge called CaloMan [20], CaloForest better captures the distribution of Centers of Energy and their Widths. CaloMan was designed with a separate module to predict the deposited energy in each layer, and thus has better performance in those metrics.
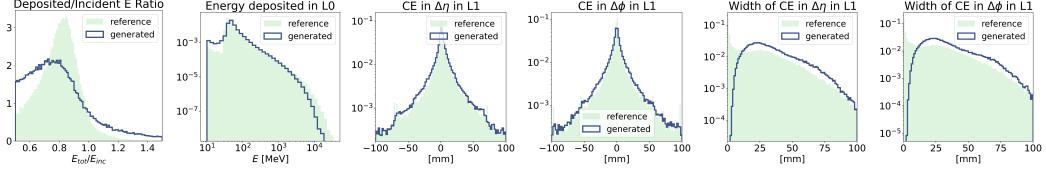
Figure 8: Histograms of high-level features comparing generated Pions samples to the test set. Note the log scale of the y-axis for all but the first plot.

We also add feature histogram plots for the Pions dataset in Figure 8 to complement the Photons results shown in the main text (Figure 5).

As mentioned above, average per-voxel energy deposits are shown for the models trained on both datasets in Figure 7 where it is clear that these distributions are learned almost perfectly.

B Re-engineering the ForestDiffusion and ForestFlow Implementation

In this Appendix we continue our analysis of the original implementation of ForestDiffusion and ForestFlow training (Algorithm 1) provided by [46], and present a summary of our improvements as a unified implementation. We further consider how to optimize data generation.

First, we comment on the pros and cons of parallelization. Most of the memory issues experienced when using ForestDiffusion and ForestFlow are a result of training many XGBoost ensembles in parallel. Parallelization using multiprocessing requires copying data arrays to worker processes, so one may ask about alternatives. Apart from multiprocessing, parallelism in Python can also be achieved through multithreading. This avoids spawning new processes with their own memory spaces and can allow threads to share memory in the main process, however, due to the Python global interpreter lock (GIL), multithreading can only be done on tasks that release the GIL while running. In fact, calls to XGBoost training do release the GIL, as XGBoost runs native C++ code, so multithreading is a potential solution for ForestDiffusion and ForestFlow. However, in our preliminary tests we found that multithreading was more prone to memory not being properly released, causing increased usage over training. We were not able to find a definitive reason for this but we suspect that the Python garbage collector in a multithreaded process does not effectively free up used memory. On the other hand, we observed that multiprocessing is very effective for releasing all used memory when the corresponding job is completed.

Alternatively, one may wonder why creating parallel jobs is necessary at all when a single XGBoost training job can make use of multiple CPUs. In fact XGBoost is not perfectly efficient in its use of additional CPUs, especially on small datasets like those from Table 8; training time is reduced by less than 50% when two CPUs are used instead of one, and this efficiency becomes even worse as more CPUs are provided. Hence, there is a tradeoff between speed and memory: assigning all CPUs to a single job uses the least memory but may be slow, whereas assigning one CPU to N training jobs will be much faster, while using roughly N times as much memory. Figure 9 demonstrates this by training our implementation of ForestFlow on a dataset with $n = 1000$,

Figure 9: XGBoost is not efficient with multiple CPUs, especially for small datasets. Running single-CPU jobs in parallel is the most time-efficient method as long as adequate memory is available.

$p = 10$, and $n_y = 10$ for various assignments of CPUs per job (cf. Figure 4). On our machine with 40 CPUs, we set the number of CPUs assigned to each job to the values {1, 2, 4, 10, 20, 40} (and correspondingly set the number of parallel jobs to {40, 20, 10, 4, 2, 1}). Generally, the number of parallel jobs times the number of CPUs assigned per job should not exceed the number of CPUs available in total, otherwise thread contention can degrade performance. When memory is a limitation, Figure 9 shows that assigning a few CPUs per job and reducing the number of jobs can greatly decrease peak memory requirements at a marginal increase in training time. Hence, we used two CPUs per job when training on large-scale calorimeter data.
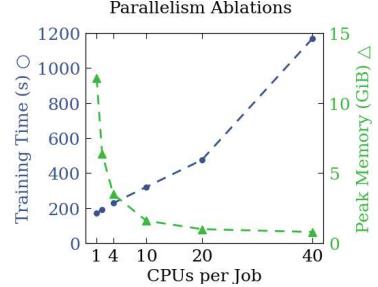
For the sake of our resource benchmarking across methods, we always use multiprocessing, and assign one CPU per worker, with the number of workers equal to the available CPUs.

B.1 Continued Analysis and Improvement of the Implementation

In Section 3.3 we began our analysis with the most impactful issues and solutions. We recommended to avoid creating large arrays in memory and instead create slices on-the-fly as needed within parallel loops (Issue 1). We found that improper use of multiprocessing could lead to excessive duplication of arrays in shared memory that could not be freed by the system, and explained how to properly share an array across processes (Issue 2). Finally, we recommended to write XGBoost models to disk as their training completed to prevent them piling up in active memory (Issue 3). These three issues and solutions accounted for the vast majority of memory improvements we observed and explained the three problematic behaviours pointed out in Section 3.1. Nevertheless, we pick up where we left off and present several additional improvements that further optimize memory usage and runtime while adhering to engineering best-practices.

Issue 4: Since worker processes created by Joblib access data saved to shared memory, the main process does not need to occupy memory by holding on to its copy of X0, X1, and Z_train. These costly array objects are merely used as keys for Joblib to identify the arrays in shared memory.
Solution 4: Explicitly save the arrays in shared memory as memory-mapped files, delete them from the main process, and retain only a reference which can be passed to worker processes.
Benefit 4: The X0, X1, and Z_train objects can be freed from the main process, amounting to **144 GiB** for the Pions dataset. Technically, we save memory-mapped files on a disk instead of a RAM disk. While a RAM disk occupies RAM space, saving to an actual disk does not, which leaves more available memory during training. Nonetheless, this does not cause slow downs from disk I/O. When a file is saved to a disk, it is first saved to cache memory (part of RAM). When the file is accessed again (potentially by a different process), if it is already in cache, the file in cache is reused. Unlike in-use RAM disk memory, this kind of cache memory can be freed upon memory pressure as it merely serves as cache for a disk.

Issue 4: Improvement

```
1  import tempfile, os
2  from joblib import dump, load
3  # Create memmap files
4  temp_folder = tempfile.mkdtemp()
5  def create_memmap(array, file_name):
6    file = os.path.join(temp_folder, file_name)
7    dump(array, file)
8    return load(file, mmap_mode='r')
9  X0_mmap = create_memmap(X0, "X0.mmap")
10 X1_mmap = create_memmap(X1, "X1.mmap")
11 Z_mmap = create_memmap(Z_train, "Z.mmap")
12 # Free memory of X0, X1, Z_train
13 del X0, X1, Z_train
```

Issue 5: Using n_y Boolean masks across the duplicated dataset to index each class's data requires n*K*n_y bytes, since the numpy.bool datatype uses one byte, not one bit. Moreover, indexing a Numpy array with the mask array (advanced indexing) creates a copy of the underlying data, as done for example in Line 26 of the original implementation (X_train[t_i][mask[y_i], :]).
Solution 5: First sort the data by class, then use Python's slice(start, end) function with the beginning and end indices of each class.
Benefit 5: On the Pions dataset these Boolean masks would occupy **173 MiB** of space. Our solution only requires 2*n_y integers, and when used for indexing only creates a view, not a copy of underlying data. However, our solution does involve sorting, though the time involved is minuscule compared to the training time for hundreds of thousands of XGBoost ensembles.

Issue 5: Original

```
1  # Create Boolean masks for class conditioning
2  mask = {} # Boolean mask for which rows of X0 have label y_i
3  y_uniq = numpy.unique(y)
```

21

```
4  for y_i in y_uniq:
5    mask[y_i] = numpy.tile(y == y_i, K)
```

Issue 5: Improvement
```
1  # Sort by label and slice for class conditioning
2  y_arg_sort = numpy.argsort(y)
3  y, X0 = y[y_arg_sort], X0[y_arg_sort]
4  y_uniq, y_counts = numpy.unique(y, return_counts=True)
5  mask = {} # Slice of X0's rows that have label y_i
6  csum = 0
7  for y_i, count in zip(y_uniq, y_counts):
8    mask[y_i] = slice(csum, csum + count)
9    csum += count
10 y_slice = {} # adjust slices for duplicated data
11 for y_i, sl in mask.items():
12    y_slice[y_i] = slice(sl.start*K, sl.stop*K)
```

Issue 6: In XGBoost training, input data is converted to a `DMatrix`, XGBoost's native data structure, and is reformatted and cached for faster access. For example, features are converted to histograms when using `hist` training, as we do. The histogram computations are redundant across jobs since the same `X_tr_i` is used for all `p_i`.

Solution 6: Ever since XGBoost version 1.6, multiple regressors trained with the same features but different targets can be encapsulated in a single `Booster` object. When the multi-dimensional target is passed to XGBoost's `fit(X,Z)` function, XGBoost internally trains an ensemble on each target sequentially while using the same `DMatrix`, avoiding redundant histogram computations over `p_i`. Multi-output trees use the same data structure and training call, so can be integrated seamlessly.

Benefit 6: This reduces `DMatrix` constructions and reduces histogram computations by a factor of p. Additionally, all ensembles over p for a given t and y are contained in the same `Booster` object, which, in turn, reduces the number of model files and metadata to be stored, and reduces file I/O overhead.

Issue 6: Original
```
1  # One Booster for each column p_i
2  Z_tr_i = Z_train[mask[y_i], p_i]
3  model.fit(X_tr_i, Z_tr_i)
```

Issue 6: Improvement
```
1  # Single Booster for all columns p_i
2  Z_tr_i = Z_train[y_slice[y_i], :]
3  model.fit(X_tr_i, Z_tr_i)
```

Issue 7: XGBoost internally uses `fp32` regardless of the input data type. However, `numpy.float64` is implicitly used in the original implementation.

Solution 7: Use `fp32` throughout the whole pipeline.

Benefit 7: Using lower precision throughout reduces memory usage without losing model accuracy and avoids implicit data type conversions.

Issue 7: Original
```
1  X0 = inputs()
2  # loaded as numpy.float64
3  X1=numpy.random.normal(size=X0.shape)
4  # default dtype is numpy.float64
```

Issue 7: Improvement
```
1  X0 = inputs()
2  X0 = X0.astype(numpy.float32)
3  X1=numpy.random.normal(size=X0.shape)
4  X1 = X1.astype(X0.dtype)
```

To summarize, our implementation making use of all our recommended changes is given below.

Our Implementation of ForestFlow Training with Single-Output Trees

```python
from sklearn.preprocessing import MinMaxScaler
import numpy, xgboost, tempfile, os
from joblib import delayed, Parallel, dump, load

X0, y, K, n_t, xgb_kwargs, n_jobs = inputs()
n, p = X0.shape
X0 = X0.astype(numpy.float32) # use XGBoost's native dtype
# Sort by label and slice for class conditioning
y_arg_sort = numpy.argsort(y)
y, X0 = y[y_arg_sort], X0[y_arg_sort]
y_uniq, y_counts = numpy.unique(y, return_counts=True)
mask = {} # Slice of X0's rows that have label y_i
csum = 0
for y_i, count in zip(y_uniq, y_counts):
  mask[y_i] = slice(csum, csum + count)
  csum += count
# Scale each class's data so that range matches noise variance
scalers = []
for y_i in y_uniq:
  scalers.append(MinMaxScaler(feature_range=(-1, 1)))
  X0[mask[y_i], :] = scalers[-1].fit_transform(X0[mask[y_i], :])
# Duplicate data and generate noise
X0 = numpy.repeat(X0, K, axis=0)
X1 = numpy.random.normal(size=X0.shape).astype(X0.dtype)
y_slice = {} # adjust slices
for y_i, sl in mask.items():
    y_slice[y_i] = slice(sl.start*K, sl.stop*K)
# Create regression targets (ForestFlow)
Z_train = X1 - X0 # regression target is constant in t, but input is not
t = numpy.linspace(0, 1, num=n_t)
# Create memmap files
temp_folder = tempfile.mkdtemp()
def create_memmap(array, file_name):
  file = os.path.join(temp_folder, file_name)
  dump(array, file)
  return load(file, mmap_mode='r')
X0_mmap = create_memmap(X0, "X0.mmap")
X1_mmap = create_memmap(X1, "X1.mmap")
Z_mmap = create_memmap(Z_train, "Z.mmap")
del X0, X1, Z_train
# Train models in double loop over timesteps and classes
def train_parallel(X0_mmap, X1_mmap, Z_mmap, t_i, y_i):
  X_tr_i = t_i*X1_mmap[y_i, :] + (1-t_i)*X0_mmap[y_i, :]
  Z_tr_i = Z_mmap[y_i, :]
  model = xgboost.XGBRegressor(**xgb_kwargs)
  model.fit(X_tr_i, Z_tr_i) # single Booster for all columns p_i
  model.save_model(f"{model_path}.ubj") # path for t_i, y_i
Parallel(n_jobs)(
  delayed(train_parallel)(
    X0_mmap, X1_mmap, Z_mmap, t_i, y_i,
  ) for t_i in t for y_i in y_slice
)
shutil.rmtree(temp_folder) # clean up memmaps
```

For a direct comparison, we show in Figure 2 the memory usage during training using the original implementation as well as ours on the same dataset with $n = 1000$, $p = 100$, and $n_y = 10$. Our implementation does not suffer from the three undesirable behaviours noted in Section 3.1.

B.2 Analysis and Improvement of Data Generation

To this point we have focused on improving the implementation of ForestFlow training. For many applications generation speed is also a critical requirement, including hosted generative model services and our running example of calorimeter simulation for experimental particle physics. In this section we turn our attention to improving the implementation of data generation with a trained ForestFlow model, starting with a summary of the existing implementation from [46].

First, for conditional sampling, labels are created using a multinomial distribution with probabilities equal to the relative prevalence of labels in the training set. Boolean masks are created to indicate the conditioning. Gaussian noise X1 is sampled to seed the generation, and Euler's method over uniformly discretized timesteps is used to solve the ODE using the trained models as the vector field. In particular, a triple `for` loop is used over timesteps, classes, and features in that order.

Python Implementation of ForestFlow Generation from [46]

```python
import numpy

y, n_t, n, p, regressors = inputs()
# Sample labels for conditioning using frequencies from the training dataset
y_uniq, y_counts = numpy.unique(y, return_counts=True)
y_probs = y_counts / numpy.sum(y_counts)
y_sample = numpy.argmax(numpy.random.multinomial(1, y_probs, size=n), axis=1)
# Create Boolean masks for class-conditioning
label_y = y_uniq[y_sample]
mask = {}
for y_i in y_uniq:
  mask[y_i] = (label_y == y_i)
# Solve ODE with Euler's method starting from noise
X1 = numpy.random.normal(size=(n, p))
h = 1 / (n_t-1) # size of timestep
for t_i in numpy.linspace(1, 0, num=n_t):
  out = numpy.zeros(shape=X1.shape)
  for y_idx, y_i in enumerate(y_uniq):
    for p_i in range(p):
      model = regressors[t_i][y_idx][p_i]
      out[mask[y_i], p_i] = model.predict(X1[mask[y_i], :])
  X1 = X1 - h * out
X0 = X1
```

Once again, we proceed by pointing out issues, offering solutions, and quantifying the benefits.

Issue 8: XGBoost's core engine is implemented in C++, and there is hidden overhead when the Python wrapper makes a call to its C-API.
Solution 8: Reduce the number of calls to the C-API by reducing the number of distinct `Booster` objects. In our training implementation, all ensembles trained over p for a given t and y are contained in the same `Booster` object (See Issue 6 in Appendix B.1). Inference on this `Booster` object produces an output shape with `[n_i, p]` containing all features.
Benefit 8: A factor of p fewer calls to the XGBoost C-API are made, and we eliminate Python's slow `for` loop over p. Additionally, cache locality is utilized more aggressively by XGBoost's C++ inference implementation.

Issue 8: Original
```python
for p_i in range(p):
  model = regressors[t_i][y_idx][p_i]
  out[mask[y_i], p_i] =
    model.predict(X1[mask[y_i], :])
```

Issue 8: Improvement
```python
model = regressors[t_i][y_idx]
out[mask[y_i], :] =
 model.predict(X1[mask[y_i], :])
```

Issue 9: Slow Numpy indexing operations are used in the triple loop.
Solution 9: Conditional generation of datapoints with different y labels uses disjoint sets of ensembles. It is not necessary to combine all partially generated datapoints over classes into a single array after every timestep. Instead, we can iterate over `y_i` in the outer loop and concatenate all the generated datapoints over classes only at the end.

Benefit 9: This eliminates writing intermediate results to non-contiguous memory `out[mask[y_i]]`. Data copying is further avoided by replacing the Boolean mask with `slice` as in Issue 5 from Appendix B.1.

```
Issue 9: Improvement
1  results = []
2  for y_idx, y_i in enumerate(y_uniq):
3    X1_i = X1[y_slice[y_i], :]
4    for t_i in numpy.linspace(1, 0, num=n_t):
5      model = regressors[t_i][y_idx]
6      X1_i = X1_i - h * model.predict(X1_i)
7    results.append(X1_i)
8  X0 = numpy.concatenate(results, axis=0)
```

To summarize, our implementation for ForestFlow generation making use of our recommended changes is given below.

```
Our Implementation of ForestFlow Generation with Single-Output Trees
1   import numpy
2
3   y, n_t, n, p, regressors = inputs()
4   # Sample labels for conditioning using frequencies from the training dataset
5   y_uniq, y_counts = numpy.unique(y, return_counts=True)
6   y_probs = y_counts / numpy.sum(y_counts)
7   y_sample = numpy.argmax(numpy.random.multinomial(1, y_probs, size=n), axis=1)
8   label_y = y_uniq[y_sample]
9   # Sort by label and slice each class
10  label_y.sort()
11  y_uniq, y_counts = numpy.unique(label_y, return_counts=True)
12  y_slice = {}
13  csum = 0
14  for y_i, count in zip(y_uniq, y_counts):
15    y_slice[y_i] = slice(csum, csum + count)
16    csum += count
17  # Solve ODE with Euler's method starting from noise
18  X1 = numpy.random.normal(size=(n, p)).astype(numpy.float32)
19  h = 1 / (n_t-1) # size of timestep
20  results = []
21  for y_idx, y_i in enumerate(y_uniq):
22    X1_i = X1[y_slice[y_i], :]
23    for t_i in numpy.linspace(1, 0, num=n_t):
24      model = regressors[t_i][y_idx]
25      X1_i = X1_i - h * model.predict(X1_i)
26    results.append(X1_i)
27  X0 = numpy.concatenate(results, axis=0)
```

B.3 Analysis of Data Iterator

In Section 3 we remarked that we branched off of the Nov. 2, 2023 version of the ForestDiffusion codebase for our analysis.[4] The codebase is active, and has seen numerous changes since that date. Due to the nature of the work presented in this paper, we must select a fixed snapshot in order to provide meaningful analysis and reproducible results. There are a few reasons we have not branched off of more recent versions for the main discussion.

The changes immediately following our branching point (see commit hash 5417806 and following) implement sweeping changes that do not directly address the issues we have noted, but instead move in an orthogonal direction with the use of a data iterator. However, many of the older implementation issues remain, and there is still value in understanding and solving them from an engineering perspective. We will discuss the merits of the data iterator below, but first note that the paper [46] does not use it for their published results, commenting only in their conclusion section that a data

---

[4]See `github.com/SamsungSAILMontreal/ForestDiffusion/tree/855281b`.

iterator could be implemented in future work. Since only the published results are available to us for comparison, we must use the earlier version of the codebase prior to the data iterator's addition.

Second, more recent versions of the codebase (up to hash `818ac3b` as of writing) do not exhibit such extreme memory consumption issues when the data iterator is used, but this is not because the data iterator itself is a solution to the issues. The code implementing the data iterator inadvertently resolved most of what we described as Issue 2 in Section 3.3. We say this was inadvertent because Issue 2 remains present in branches of the execution where the data iterator is not enabled. To disentangle the effects of the data iterator from Issue 2, we started from the earlier snapshot.

Third, we have identified unintended issues with the use of the data iterator that go beyond resource inefficiencies and actually lead to incorrectly trained models. As emphasized in Section 2.3, one drawback of XGBoost is that it does not allow for mini-batch training, instead optimizing over the entire labeled training set at once. The data iterator is an optional technique with an evolving implemention since XGBoost 1.2.0 which allows for batched processing of large datasets. XGBoost's histogram training method does not use the dataset during optimization, instead it constructs a summarized version of the dataset prior to optimization. The summary, stored in a `QuantileDMatrix` object, consists of histogram quantiles of each feature, along with histogram bin indices of each datapoint. Then, during optimization only the stored quantiles are used for split values in trees, while the bin indices indicate whether a data point falls into the left or right child node given a quantile split. The data iterator allows the histogram quantiles and bin indices to be constructed iteratively from batches of data, rather than from the entire dataset at once which can avoid memory bottlenecks with large datasets. After construction of the `QuantileDMatrix`, XGBoost does not use the data iterator or raw dataset again during training.

Unfortunately, the use of the data iterator after commit hash `5417806` is flawed since it uses fresh noise to construct X_t from X_0 each time it is consumed. XGBoost consumes the data iterator four times in total, all of which are for the creation of the `QuantileDMatrix`, when it computes the overall data shape[5],

constructs the histogram[6], stores bin indices in row-major order[7], and stores bin indices in column-major order.[8] The latter three components must be built from the same datapoints provided in the same order for correct construction, but the injection of noise by the ForestDiffusion implementation means different datasets are used each time. Seeding the randomness of the data iterator when it is initialized and reset would ensure that the same noisy datapoints appear in the same order on each of the four passes.

There is however one benefit of the data iterator relevant to ForestDiffusion and ForestFlow that we have uncovered. When passing the entire input dataset to XGBoost for `QuantileDMatrix` construction, the input dataset is held in memory by the `QuantileDMatrix` throughout optimization, although it is not used. When using the data iterator, `QuantileDMatrix` does not hold a copy of the input throughout optimization. This could become relevant when training many XGBoost regressors in parallel; say we train n_{jobs} jobs in parallel each using the appropriate subset of X_{train}, the subsets being $[n_i \cdot K, p]$ arrays, where n_i is n/n_y for balanced classes. Using the data iterator would avoid holding these n_{jobs} copies of size $[n/n_y \cdot K, p]$ within the respective `QuantileDMatrix` objects. In the spirit of our paper, we can quantify the benefit it would give on the Pions dataset which is $n_{jobs} \cdot n/n_y \cdot K \cdot p \cdot 8$ bytes (or 4 bytes after the switch to `fp32` from Issue 7 in Appendix B.1). With $n_{jobs} = 40$, $K = 100$, $n = 120,800$, and $p = 533$ we would see a further reduction of 128 GiB (or 64 GiB). While significant, this is lower by orders of magnitude than the benefits we described in Issues 1, 2, and 3.

There is also a major con to using the data iterator - it is slower. First, for general uses, the data iterator constructs the `QuantileDMatrix` histogram on one batch, and then iteratively updates it on subsequent batches. This process requires more computation overall than constructing the histogram in a single shot on the entire dataset. Second, in the particular use of the data iterator for ForestDiffusion, (seeded) noise must be generated in each of the four passes through the iterator. This is wasteful compared to the alternative of generating noise once across the entire dataset.

---

[5] github.com/dmlc/xgboost/blob/release_2.1.0/src/data/iterative_dmatrix.cc#L172

[6] github.com/dmlc/xgboost/blob/release_2.1.0/src/data/iterative_dmatrix.cc#L206

[7] github.com/dmlc/xgboost/blob/release_2.1.0/src/data/iterative_dmatrix.cc#L242

[8] github.com/dmlc/xgboost/blob/release_2.1.0/src/data/iterative_dmatrix.cc#L267

Table 6: Data from Figure 1 compared against the data iterator. **Left:** Training time (s). **Right:** Peak memory usage (GiB).

Method / n	1000	3000	10,000	30,000	100,000	Method / n	1000	3000	10,000	30,000	100,000
Original	297	363	587	1108	*crash*	Original	33	36	71	169	*crash*
Ours	172	221	380	880	2711	Ours	18	18	20	25	31
Ours-Iterator	177	230	408	957	2941	Ours-Iterator	19	20	20	23	29

Still, the data iterator can be helpful, so we have included a corrected implementation in our codebase. To demonstrate its merits, we reproduce the data from Figure 1 in Table 6 with the data iterator option included, where K batches are used such that only one copy of the raw dataset is loaded at a time during `QuantileDMatrix` construction. We see a marginal slowdown, but also reduction in peak memory usage for the reasons explained above. In particular, for the $n = 100,000$ run we note that the expected memory savings of $n_{\text{jobs}} * (n/n_y * K * p * 4) = 1.5$ GiB is consistent with the observed memory savings. In conclusion, the data iterator can be beneficial for very large datasets if there is a memory bottleneck. Otherwise, it is better to use the ordinary `QuantileDMatrix` construction method as it is faster.

C Performance Improvements and Ablation Studies

In this Appendix, we further details methods to improve the generative quality or resource utilization of ForestDiffusion and ForestFlow [46] that go beyond implementation changes, and evaluate them with ablation studies. This discussion extends Section 3.4 from the main text.

C.1 Multi-Output Trees

One obvious downside of using XGBoost regressors is that they output a scalar, whereas for generative modelling we need to output a vector $\mathbf{x}$. In practice $\mathbf{x}$ is often high dimensional, and its dimension p enters multiplicatively into the number of ensembles needed ($n_t \cdot n_y \cdot p$) for ForestDiffusion and ForestFlow.

We propose replacing single-output trees with multi-output trees, also referred to as vector-leaf trees [100, 99, 66, 44, 81]. Each leaf node in the multi-output tree provides a vector of values, and the training algorithm is modified to fit all output variables at once by minimizing the sum of losses over individual outputs [100]. Not only does this reduce the number of ensembles we require by a factor of p, but it has the potential to increase model performance by better capturing correlations between output variables during generation since generated elements do not come from independent trees. Although single-output trees may be sampling from the marginals, and not the joint distribution at any timestep, we do note that feature interactions can occur *between* timesteps in the ForestDiffusion and ForestFlow algorithms, since during generation (independent) outputs from one timestep affect all inputs in the next timestep of the SDE/ODE solve. This explains why the algorithms can show strong distribution learning abilities even with single-output trees that are limited to sampling marginally.

Past work on discriminative tasks has shown that multi-output trees may require thousands of boosting rounds before their performance surpasses their single-output counterparts [100]. Similarly, we found for our generative tasks that multi-output trees only become comparable or surpass single-output baselines when scaled up with large n_{tree} (~ 2000) and K (~ 1000) with appropriate regularization, see Table 2. However, we noticed that multi-output trees are less prone to overfitting and generalize better to the test set. This may be because multi-output trees solve a more difficult optimization task involving all outputs at once, meaning they are less prone to learning "shortcuts" that minimize the loss on individual outputs but that do not generalize.

These differences are demonstrated by comparing Table 2, which used scaled-up ensembles and early stopping, to Table 7 below without those additions. Without the benefit of wide ensembles the MO method seems to underperform SO, however, we note that both versions use the same n_{tree} and maximum depth hyperparameters across tests, meaning that the SO models essentially use p times

more parameters. When scaled up with early-stopping, single-output trees tended to stop earlier than multi-output versions, which may simply be a result of their different capacities.

Since version 2.0.0, XGBoost has implemented multi-output trees. Although there is a lack of documentation currently, to the best of our knowledge the algorithm reflects the proposal of Zhang and Jung [100]. During our testing we identified a bug in the gain computation in the official XGBoost codebase and reported it to the maintainers who implemented our proposed fix. See `github.com/dmlc/xgboost/issues/9960`. Hence, only XGBoost version 2.1.0 or later should be used for multi-output trees. Still, this version's implementation is not yet optimized for time and memory performance, so our measurements in Section 4.1 should be considered preliminary.

Figure 10: Number of trees at the best iteration of the validation loss by timestep and dataset. The number of trees is averaged over ensembles at the same timestep for different classes y and features p_i (for single-output trees). Selected datasets from the set of all 27 in Table 8 are highlighted for comparison between methods. Models are trained with early stopping after $n_{\text{ES}} = 20$ rounds. The top figure is reproduced from Figure 3 in the main text for ease of comparison.

C.2 Early Stopping with Scaled-Up Hyperparameters

Jolicoeur-Martineau et al. [46] observed that their models were underfitting the training data, and hence used no regularization on their XGBoost ensembles. This has the downside that nearly every tree in every ensemble will reach is maximal size, determined by the depth hyperparameter, resulting in very large models that slow down training and generation. To provide more capacity for models, we scaled up n_{tree} from 100 to 2000, and increased $K = 100$ to 1000 for better coverage of the loss function expectations. While scaling up the capacity of models can reduce underfitting and increase model performance, it also exacerbates training and generation speed concerns.

We have found that the severity of underfitting varies greatly across timesteps. Models at timesteps closer to $t = 1$ (noise) converge quickly, so increasing n_{tree} without regularization is computationally wasteful, and in the worst case could lead to severe overfitting. Hence, to provide capacity only where it is needed we propose to apply the well-known regularization technique of early stopping individually on each XGBoost ensemble's training loss. Taking advantage of the nature of ForestDiffusion and ForestFlow, we reused the training set X_0 with fresh noise X_1 to create the regression features and targets that we validate on.

In Figure 3 we showed how ensembles trained for t close to 1 stop improving on the validation set very early. We reproduce this figure (which used FF-SO-Scaled) along with the other scaled variants in Figure 10 for comparison. The patterns of early stopping are very similar between ForestDiffusion and ForestFlow, but differ greatly between the single-output and multi-output variants. First, a different subset of datasets ends up using very wide ensembles – as one example SO stops early at all timesteps on *congress*, whereas MO uses maximally wide ensembles for many timesteps. Second, on many datasets MO also trains wide ensembles for later timesteps. One basic implication is that early stopping provides a greater computational benefit for SO variants as the average stopping round is lower. We suppose this is due to how multi-output trees are better able to capture joint distributions and can pick up on the subtle correlations between elements which are mostly noise. However, as timesteps close to $t = 1$ in diffusion sampling are mainly responsible for global structure and not fine details, we find that single-output trees are not at a significant disadvantage here.

Finally, the fact that SO focuses almost entirely on the first fifth of the timesteps indicates that a non-uniform partitioning of the $(0, 1)$ interval could potentially improve performance. This would be similar to how NN-based diffusion models are often trained with a non-uniform noise schedule [73, 18]. We leave these explorations for future work.

To wrap us this discussion, we show hyperparameter ablations over a single dataset, *connectionist_bench_sonar*, for K, n_{tree}, and tree structure in Figure 11. The results demonstrate that SO hardly benefits from wide ensembles, while MO can continue to marginally improve its generalization to the test set even up to $n_{\text{tree}} = 2000$. The duplication factor K has a strong effect, and the default setting of 100 used in [46] is far from enough. While SO outperforms MO for most settings, there is a regime where MO can achieve the best generalization on $W1_{\text{test}}$, but this requires both high duplication ($K \geq 1000$), and wide ensembles ($n_{\text{tree}} \geq 1000$).

C.3 Class-Conditional Scalers

One advantage of XGBoost as a function approximator is its robustness to data with varying scales and distributions. This stands in stark contrast to deep NNs which require data to be carefully pre-processed for best results. While XGBoost itself is agnostic, ForestDiffusion and ForestFlow do require input data to be on the same scale as the added noise in Eq. 2 and 5. Jolicoeur-Martineau et al. [46] achieve this by applying min-max scaling over the entire input dataset. However, when using the class-conditional variant, models are trained on disjoint sets of data belonging to each class. If the classes have distinct distributions, which is often the point of distinguishing data by class, then the data subsets actually provided to the training algorithm may not be properly scaled. To rectify this, we propose class-conditional min-max scaling. This is especially beneficial on the calorimeter datasets as the classes represent particle energies increasing on an exponential scale leading to vastly different per-class distributions. Class-conditional scaling centers the data better making it more easily distinguishable as noise is added, ultimately benefiting the model performance.
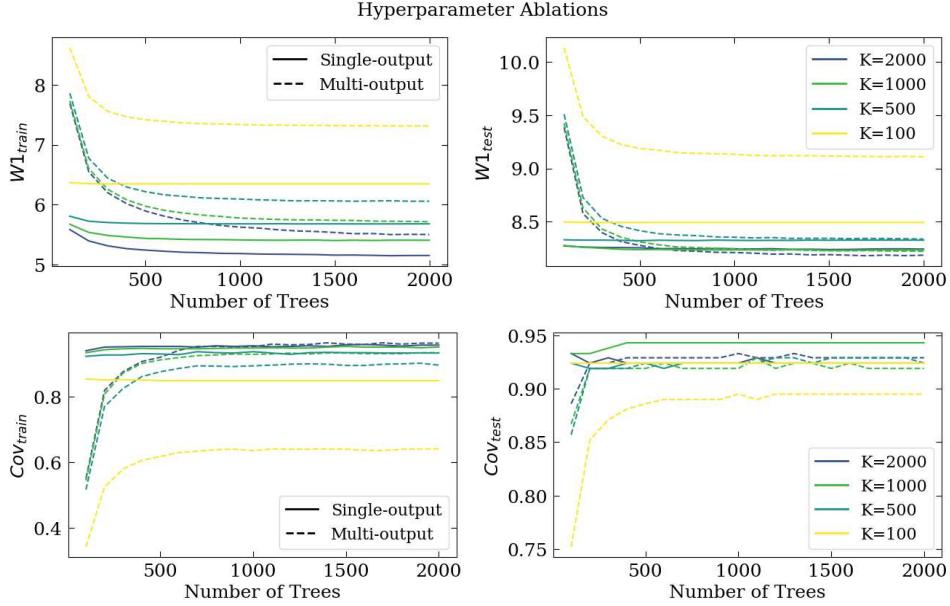
Figure 11: Effect of varying K, n_{tree}, and the tree structure (SO vs. MO) on distributional metrics.

Table 7: Average rank (standard error) of generated data quality over 27 datasets. Lower is better.

	$W1_{\text{train}}$	$W1_{\text{test}}$	$\text{Cov}_{\text{train}}$	Cov_{test}	R^2_{gen}	$F1_{\text{gen}}$	P_{bias}	cov_{rate}	Avg.
GaussianCopula [45]	$10.0_{\pm 0.3}$	$10.1_{\pm 0.3}$	$10.0_{\pm 0.4}$	$10.1_{\pm 0.4}$	$9.1_{\pm 0.1}$	$9.4_{\pm 0.4}$	$8.4_{\pm 1.5}$	$10.6_{\pm 0.5}$	$9.7_{\pm 0.1}$
TVAE [95]	$7.9_{\pm 0.4}$	$7.6_{\pm 0.4}$	$8.3_{\pm 0.4}$	$8.3_{\pm 0.4}$	$9.6_{\pm 0.7}$	$9.0_{\pm 0.6}$	$10.7_{\pm 0.5}$	$10.1_{\pm 0.3}$	$8.9_{\pm 0.0}$
CTGAN [95]	$11.4_{\pm 0.2}$	$11.3_{\pm 0.2}$	$11.2_{\pm 0.2}$	$11.1_{\pm 0.2}$	$11.6_{\pm 0.2}$	$11.4_{\pm 0.2}$	$7.9_{\pm 1.3}$	$10.6_{\pm 0.5}$	$10.8_{\pm 0.1}$
CTAB-GAN+ [101]	$9.7_{\pm 0.3}$	$9.6_{\pm 0.4}$	$9.9_{\pm 0.4}$	$9.8_{\pm 0.4}$	$10.0_{\pm 0.2}$	$9.9_{\pm 0.4}$	$10.7_{\pm 0.6}$	$8.9_{\pm 1.2}$	$9.8_{\pm 0.1}$
STaSy [49]	$9.0_{\pm 0.2}$	$9.2_{\pm 0.2}$	$8.0_{\pm 0.3}$	$8.0_{\pm 0.4}$	$8.3_{\pm 1.3}$	$7.8_{\pm 0.5}$	$6.9_{\pm 1.1}$	$6.5_{\pm 1.4}$	$8.0_{\pm 0.2}$
TabDDPM [54]	$4.4_{\pm 0.9}$	$5.9_{\pm 0.8}$	$4.1_{\pm 0.7}$	$4.9_{\pm 0.7}$	$2.0_{\pm 0.7}$	$5.6_{\pm 0.8}$	$3.3_{\pm 1.4}$	$3.2_{\pm 0.7}$	$4.2_{\pm 0.1}$
ForestDiffusion-Original	$5.0_{\pm 0.3}$	$4.8_{\pm 0.2}$	$5.0_{\pm 0.3}$	$4.5_{\pm 0.4}$	$4.0_{\pm 0.5}$	$3.2_{\pm 0.5}$	$6.0_{\pm 0.7}$	$5.1_{\pm 0.9}$	$4.7_{\pm 0.1}$
ForestDiffusion-SO	$4.4_{\pm 0.2}$	$4.3_{\pm 0.2}$	$5.4_{\pm 0.2}$	$5.1_{\pm 0.4}$	$6.9_{\pm 0.4}$	$4.5_{\pm 0.5}$	$6.0_{\pm 1.0}$	$6.1_{\pm 0.7}$	$5.3_{\pm 0.1}$
ForestDiffusion-MO	$6.6_{\pm 0.3}$	$6.5_{\pm 0.4}$	$6.9_{\pm 0.3}$	$6.3_{\pm 0.4}$	$6.0_{\pm 0.6}$	$5.0_{\pm 0.5}$	$6.0_{\pm 0.5}$	$5.4_{\pm 0.6}$	$6.1_{\pm 0.0}$
ForestFlow-Original	$3.0_{\pm 0.3}$	$2.9_{\pm 0.3}$	$2.4_{\pm 0.3}$	$2.9_{\pm 0.4}$	$2.6_{\pm 0.6}$	$4.0_{\pm 0.4}$	$3.3_{\pm 1.0}$	$3.4_{\pm 0.8}$	$3.1_{\pm 0.1}$
ForestFlow-SO	$2.0_{\pm 0.3}$	$1.8_{\pm 0.4}$	$2.2_{\pm 0.4}$	$2.7_{\pm 0.5}$	$2.9_{\pm 0.5}$	$4.0_{\pm 0.4}$	$5.3_{\pm 0.8}$	$4.3_{\pm 0.9}$	$3.1_{\pm 0.1}$
ForestFlow-MO	$4.6_{\pm 0.3}$	$4.1_{\pm 0.3}$	$4.6_{\pm 0.4}$	$4.4_{\pm 0.4}$	$5.1_{\pm 0.7}$	$4.5_{\pm 0.5}$	$3.6_{\pm 0.6}$	$3.9_{\pm 1.3}$	$4.3_{\pm 0.1}$

C.4 Sampling with the Training Set Label Distribution

For class-conditional sampling, [46] used the relative prevalence of classes in the training set to define a multinomial distribution and sampled from it to create class labels for conditioning. We found it advantageous to directly use the empirical distribution of class labels from the training set for conditioning, especially for the distributional Wasserstein metrics. For the small datasets used in benchmarking (Table 8), multinomial sampling may lead to a skewed distribution by chance; the law of large numbers may not kick in at these sizes. This type of sampling with training set labels is also mandated in the Fast Calorimeter Simulation Challenge [26].

As an ablation, in Table 7 we show performance results corresponding to Table 2 where SO and MO use the same hyperparameter settings as Original, but with class-conditional scalers and training set label conditioning (i.e. not using scaled up models nor early stopping). The results show that some gains can be made on distributional metrics simply by improving the distributions of scaled data and class labels. However, scaling up the models as done in Table 2 provides more impressive gains, especially for the MO variant.

D Experimental Details

In this Appendix we lay out the details of the experiments conducted in Section 4 and Appendix C.

D.1 Datasets

For the resource scaling experiments in Section 4.1 we used synthetic data that was randomly generated. The input data X of size $[n, p]$ was simply drawn from a identity covariance Gaussian, while the class label y was randomly drawn from the integers $[0, n_y)$. While this data is meaningless for model performance, it gives us precise control over the dataset size for analysing resource usage. Since the correlations between features are random, unregularized XGBoost regressors will use essentially their entire available capacity in learning which gives us a good upper bound on resource usage. The dataset size parameters were set at $n = 1000$, $p = 10$, and $n_y = 10$ by default, and a single one of these three was modified at a time. We swept over the values $n \in \{100, 300, 1000, 3000, 10000, 30000, 100000, 300000\}$, $p \in \{3, 10, 30, 100, 300\}$, and $n_y \in \{1, 3, 10, 30, 100\}$.

The early stopping regularization used for SO-ES and MO-ES introduces dataset dependence for training time. One may wonder if the randomness of the synthetic data affects when early stopping would typically occur. To check this, we repeated the resource scaling experiments on differently sized subsamples of the Pions data set. We found the same trends for SO-ES and MO-ES as shown in Figure 4 which gives us confidence that our reported results for training time will be typical of real-world datasets.

In Section 4.2 we used 27 datasets from the UCI Machine Learning Repository of tabular datasets [48] and from scikit-learn [79] that have previously been studied [69, 46]. These datasets are summarized in Table 8, and showcase a variety of tabular learning tasks with variation in the number of datapoints n, features p, classes n_y, and target types. In each case, we randomly held-out 20% of the dataset as a test split and trained generative models on the remaining 80%. Categorical variables are one-hot encoded.

Each of the UCI datasets is covered by a CC BY 4.0 license, while the *iris* dataset has a BSD 3-Clause License, and *california housing* has no license.

D.2 Metrics

For a fair and direct comparison in Section 4.2, we use eight performance metrics for generated data from [46] which measure quality along four different axes: distributional distance, diversity, usefulness for training discriminative models, and usefulness for statistical inference.

Distributional Distance We measure the Wasserstein-1 distance between the generated data and either the training set ($W1_{\text{train}}$) or test set ($W1_{\text{test}}$). The Wasserstein distance quantifies similarity in distribution - smaller distance to the test set is always desirable while distance to the training set should be similar in magnitude to the distance between the training and test sets, as a much smaller distance here can indicate memorization. Generally, $W1_{\text{train}}$ values are not less than the train-test distance, so we treat lower values as better. Computation of Wasserstein distances was done with the Python Optimal Transport library [29]. These metrics are omitted for the larger *bean* and *california* datasets as they scale quadratically in dataset size which is prohibitively expensive [69, 46].

Diversity Coverage [70] measures to what extent the generated data covers a reference dataset, where a reference datapoint is covered if there is at least one generated datapoint in its neighbourhood. Hence, generated data must be as diverse as the reference data to achieve high Coverage. Coverage is computed as the ratio of covered points to all points [89]

$$\text{Coverage}\left(\{x_i^g\}_{i=1}^n, \{x_j^r\}_{j=1}^m\right) = \frac{1}{m}\sum_{j=1}^m \max_{i=1,\dots,n} \mathbb{1}\left(x_i^g \in B(x_j^r, \text{NND}_k(x_j^r))\right), \quad (8)$$

where x^g are the generated datapoints, x^r are the reference datapoints, $\mathbb{1}(\cdot)$ denotes the indicator function, $B(x, r)$ denotes a ball centered at x with radius r, and $\text{NND}_k(x_j^r)$ is the nearest-neighbour distance between x_j^r and its kth nearest neighbour in $\{x_j^r\}_{j=1}^m$. We use an L1 ball to compute distances as it is more suited for mixed data types typical of tabular data. k is chosen automatically as the smallest value such that the training data has at least 95% Coverage of the test data. We calculate the Coverage using either the training ($\text{Cov}_{\text{train}}$) or test ($\text{Cov}_{\text{test}}$) dataset as the reference. $\text{Cov}_{\text{train}}$ helps to address "mode dropping" where some parts of the training dataset are ignored, while Cov_{test} helps measure the ability to generalize with sufficient diversity. These metrics were computed for all datasets.

Table 8: Tabular benchmark datasets. Training dataset sizes n are 80% of the total number of datapoints. Continuous and integer targets y are treated as an additional feature.

Dataset	Citation	# Datapoints n	# Features p	# Classes n_y	Target y type
airfoil self noise	[12]	1503	6	N/A	Continuous
bean	[53]	13611	16	7	Categorical
blood transfusion	[97]	748	4	2	Binary
breast cancer diagnostic	[94]	569	30	2	Binary
california housing	[76]	20640	9	N/A	Continuous
car evaluation	[11]	1728	6	4	Categorical
climate model crashes	[62]	540	18	2	Binary
concrete compression	[96]	1030	9	N/A	Continuous
concrete slump	[98]	103	8	N/A	Continuous
congressional voting	[1]	435	16	2	Binary
connectionist bench sonar	[83]	208	60	2	Binary
connectionist bench vowel	[21]	990	10	2	Binary
ecoli	[71]	336	7	8	Categorical
glass	[33]	214	9	6	Categorical
ionosphere	[85]	351	33	2	Binary
iris	[28]	150	4	3	Categorical
libras	[22]	360	90	15	Categorical
parkinsons	[59]	195	22	2	Binary
planning relax	[10]	182	12	2	Binary
qsar biodegradation	[65]	1055	41	2	Binary
seeds	[15]	210	7	3	Categorical
tic-tac-toe	[5]	958	9	2	Binary
wine	[3]	178	13	3	Categorical
wine quality red	[19]	1599	11	N/A	Integer
wine quality white	[19]	4898	12	N/A	Integer
yacht hydrodynamics	[34]	308	7	N/A	Continuous
yeast	[72]	1484	8	10	Categorical

Usefulness for Training Discriminative Models Tabular generative models are often motivated as a way to replace or extend training data for downstream tabular discriminative models [95, 54]. Available training data may be considered private and not suitable for directly training a discriminative model, whereas synthetic data derived from a generative model may be more palatable. Alternatively, synthetic data may be used with the hope that it leads to better performing downstream models. Hence, we measure the usefulness of generative models by training downstream discriminative models on generated data, and evaluating discriminative performance on the test set. Performance is measured either by the F1-score for classification tasks (20 datasets), or the R^2-coefficient for regression tasks (7 datasets), where higher is better. Since these metrics are highly dependent on the type of discriminative model used, we average the performance metrics over four different methods that are commonly used for tabular discriminative modelling: linear/logistic regression, AdaBoost [30], Random Forests [41], and, of course, XGBoost [17].

Usefulness for Statistical Inference Whereas the above metrics take a machine learning point of view in aiming to optimize the performance of a model, we can also consider a statistical point of view and measure the usefulness of synthetic data for inferring the importance of features [91]. By training a linear model on either the training data or generated data we can compare the regression parameters β. If the generated data accurately represents the training data, the learned regression coefficients should be similar. If these coefficients are not similar, one might conclude from the generated data that a given feature is statistically important when the same conclusion would not be reached using the training data. The percent bias measures this difference and is defined as $P_{\text{bias}} = |\mathbb{E}\frac{\hat{\beta}-\beta}{\beta}|$ using the estimated coefficients $\hat{\beta}$ on generated data and actual coefficients β from training data, with the expectation taken over generated data. From another direction, it is desirable for confidence intervals on the estimated coefficients $\hat{\beta}$ to contain the true coefficients β. This is quantified by the coverage rate cov_{rate}, the fraction of β that are contained in the confidence intervals

around $\hat{\beta}$. These metrics were computed only for the regression tasks (7 datasets). Lower is better for P_{bias}, but higher is better for cov_{rate}. Coverage rate is not to be confused with Coverage used above as a diversity metric.

D.3 Baseline Methods

In addition to comparing our approach to the original implementation of ForestDiffusion and Forest-Flow in Section 4.2, we also compare our improved models to 6 popular baseline methods for tabular generative modelling, including state-of-the-art deep learning methods, as done in [46].

GaussianCopula Many deep learning-based generative models learn a mapping between a simple distribution on latent space and the data distribution [61]. Generation is done by sampling from the latent distribution and mapping the sample to data space. This overarching idea hearkens back to copula methods [86] which model any multivariate joint distribution by its univariate marginals along with a copula describing the dependence structure. We use Gaussian copulas [45] implemented by Synthetic Data Vault (SDV) [78] using default hyperparameters.

TVAE Variational autoencoders (VAE) [50] learn an encoder and decoder with a low dimensional latent space through variational inference. As a typical example we use the tabular VAE (TVAE) from [95], again implemented by SDV using default hyperparameters.

CTGAN and CTAB-GAN+ Generative adversarial networks [35] train a generator which produces synthetic datapoints, and a discriminator that tries to classify real and synthetic datapoints. The two networks are trained simultaneously in an adversarial manner. As a typical example we use the conditional tabular GAN (CTGAN) from [95], also as implemented by SDV. We also employ a more modern tabular GAN called CTAB-GAN+ [101] as implemented by its authors. Both methods use default hyperparameters from their respective implementations.

StaSy and Tab-DDPM More recently score-based [87] and diffusion models [40, 88] have eclipsed VAEs and GANs for generative quality on the image modality. To represent these classes, we use STaSy [49], a score-based method, and Tab-DDPM [54] a denoising diffusion model [40] adapted for tabular settings. For the former we use hyperparameters found by [46], and the latter uses default hyperparameters from the author's implementation.

D.4 Experimental Setup and Hyperparameters

When measuring resource usage in Section 4.1 training time was clocked starting once data had been loaded and pre-processed, and stopped once all models had been trained (i.e. generation and evaluation are not included). For models that trained successfully (i.e. did not fail due to memory issues), we measure the time to generate five batches of data equal in size to the dataset a given model was trained on. We also monitor the used CPU memory every second (or every 10 seconds for long runs taking more than one hour), including at the beginning before any data is loaded to capture the background memory usage by the operating system. The peak memory that we report is the single highest measurement over training, with the single lowest measurement subtracted off to remove the effects of the background. The peak memory burden is more important than alternatives, like the average memory, since the peak burden determines if a job can successfully complete on a given machine.

Throughout our experiments we made efforts to keep comparisons as fair as possible by using the same hyperparameters as [46] where sensible. A summary of our settings is given in Table 9. In particular, for the resource scaling experiments in Section 4.1, all methods use the same learning and XGBoost hyperparameters, other than the number of early stopping rounds n_{ES} which is set to 20 when enabled. The data was duplicated $K = 100$ times with $n_t = 50$ discrete time steps as recommended by [46], and models were trained conditionally on y whenever $n_y > 1$. Computationally, 40 parallel training jobs were used (equal to the number of CPUs on our machine), with one CPU assigned to each job. For the sake of benchmarking, we did not reduce the number of parallel jobs when methods began failing due to memory issues. XGBoost hyperparameters were left at their defaults, other than L2 regularization which was set to $\lambda = 0$. Notably, this means $n_{\text{tree}} = 100$ trees were trained per ensemble of max depth 7.

The same hyperparameters were used for our performance benchmarking in Section 4.2 and additional tests in Appendix C, with the exception of $K = 1000$ and $n_{\text{tree}} = 2000$ for the scaled up

Table 9: Hyperparameter settings

Method	n_t	K	n_{tree}	n_{ES}	η	λ	ϵ	Scaler	Sampler
Original	50	100	100	0	0.3	0	0.001 (FD), 0 (FF)	Single	Multinomial
SO	50	100	100	0	0.3	0	0.001 (FD), 0 (FF)	Per-class	Label
MO	50	100	100	0	0.3	0	0.001 (FD), 0 (FF)	Per-class	Label
SO-ES (Figure 4)	50	100	100	20	0.3	0	0 (FF)	Per-class	Label
MO-ES (Figure 4)	50	100	100	20	0.3	0	0 (FF)	Per-class	Label
SO-Scaled (Table 2)	50	1000	2000	20	0.3	0	0.001 (FD), 0 (FF)	Per-class	Label
MO-Scaled (Table 2)	50	1000	2000	20	0.3	0	0.001 (FD), 0 (FF)	Per-class	Label
CaloForest	100	20	20	0	1.5	1	0 (FF)	Per-class	Label

models. Each method was trained with 3 different random seeds on each dataset, and for each training run 5 sets of data the size of the training dataset were generated. Each set of data was used to compute the performance metrics independently, and the results were averaged across the 5 generations per 3 seeds. These averaged performance metrics were then used to compute the relative rankings between methods. For each metric, we computed the ranking of methods on each dataset and then took the mean and standard deviation of rankings across datasets. As discussed in Appendix D.2, not all metrics could be used for all datasets, so the averages over datasets only include applicable datasets where the metric could actually be computed.

D.5 Additional Performance Benchmarking Results

Here we complement the summarized results of Table 2 by plotting the raw metric values averaged over three seeds for each metric, method, and dataset. We remark that the *bean* and *california* datasets were not evaluated with the Wasserstein metrics due to their size. Other plots show missing information for datasets when the metric is suited for either classifcation or regression tasks, but not both. In addition to the 6 baseline methods and variants of FD and FF, we also show the Oracle metric values which are obtained by using the training set as if it were generated data. Since the training set is truly from the underlying distribution, it can serve as a "gold standard" for some metrics. Notably, $W1_{\text{train}}$ will always be zero while $\text{Cov}_{\text{train}}$ will always be one – this is not desirable for generated data as it indicates memorization of the training data. However, the oracle values of $W1_{\text{test}}$ and Cov_{test} give an estimate of the best possible expected value that would be achieved with data from the correct distribution.