Online Structure Learning of Probabilistic Circuits

Stefan Wagner[1] **Anish Kuttetira**[1] **YooJung Choi**[1]

[1]School of Computing and Augmented Intelligence, Arizona State University

Abstract

Probabilistic circuits (PCs) are a class of probabilistic models that guarantee exact and efficient inference of various probabilistic queries by enforcing certain structural properties. Despite recent advances in learning PCs, existing methods are mostly limited to offline batch learning. In particular, while tractable inference of many queries requires determinism, there are no online structure learners for deterministic PCs, to the best of our knowledge. To bridge this gap, we introduce an online incremental structure learning framework for deterministic and structured-decomposable PCs. We use distribution-preserving *splits* to dynamically increase the expressivity of the PC structure and *merge*, when necessary, to manage size blowup, both leveraging tractable heuristics and closed-form parameter updates using the circuit properties. We pair this with a tunable decay factor to systematically adapt to distribution shifts. Empirical evaluations on density estimation benchmark datasets show that our method achieves stable training through non-static data streams and offers robust adaptation via parameter decay.

1 INTRODUCTION

Probabilistic circuits (PCs) [Choi et al., 2020] are powerful models that guarantee tractable inference for complex probabilistic queries by following structural properties such as decomposability and determinism. They encompass many families of tractable probabilistic models including sum product networks [Poon and Domingos, 2011], arithmetic circuits [Darwiche, 2002], cutset networks [Rahman et al., 2014], probabilistic generating circuits [Zhang et al., 2021], and sum of squares circuits [Loconte et al., 2025]. There has been significant progress in scalable learning algorithms for the structure and parameters of many expressive classes of PCs [Liang et al., 2017, Peharz et al., 2018, Dang et al., 2022b, 2023, Liu et al., 2024, Gala et al., 2024, Loconte et al., 2024]. However, they focus on batch learning in static environments, whereas real-world data is rarely stationary. Some previous efforts have explored online structure learning for sum-product networks—a kind of decomposable PCs [Lee et al., 2013, Dennis and Ventura, 2017, Kalra et al., 2018]; however, these methods do not support, nor can they be easily extended to, learning PC structures that also satisfy *determinism*, a property integral for tractable computation of queries such as the entropy or maximum-a-posteriori inference. In a nutshell, this is due to the fact that deterministic PCs recursively partition the support, which clustering based structural learners do not inherently guarantee when partitioning based on data alone.

We address these shortcomings with an online incremental structure learning framework for deterministic and structured-decomposable PCs. Similar to STRUDEL [Dang et al., 2022b] and LearnPSDD [Liang et al., 2017], we grow a circuit structure through operations that guarantee determinism at every learning stage. In particular, our approach dynamically captures emerging data patterns via distribution-preserving conditional splits and performs subgraph merging to prevent size blowup. By exploiting determinism, parameter estimation after each structural change is done in closed form using sufficient statistics, based on *edge flows*, which take linear space in the circuit size; this can easily be paired with a decay factor to adapt to distribution shifts. Moreover, the selection heuristics for splits and merge can be computed efficiently using determinism, for the former, and additionally structured decomposability, for the latter.

2 PRELIMINARIES

Probabilistic circuits (PCs) recursively represent distributions through compositions of sum, product, and univariate leaf nodes. They can tractably compute various probabilistic

Algorithm 1 Online Structure Learning

Require: Initial circuit C, stream $\{B_t\}_{t=1}^{T}$, decay factor λ, maximum & downsizing size limit $N_{\text{max}}, N_{\text{downsize}}$

splitting $\leftarrow True$
for each incoming batch B_t **do**
  if *splitting* **then**
    $(e^\star, X^\star) \leftarrow$ FIND_EDGE_TO_SPLIT(C)
    if $(e^\star, X^\star)$ is valid **then**
      $C \leftarrow$ SPLIT$(C, e^\star, X^\star)$
    if SIZE$(C) > N_{\text{max}}$ **then** *splitting* $\leftarrow False$
  else
    $C \leftarrow$ MERGE(C)
    if SIZE$(C) < N_{\text{downsize}}$ **then** *splitting* $\leftarrow True$
  $C \leftarrow$ UPDATE_STATISTICS(C, B_t)
  $C \leftarrow$ DECAY_STATISTICS(C, λ)
return C

inference queries based on different structural constraints. Most notably, PCs are typically assumed to be *smooth* and *decomposable*, allowing for exact marginal and conditional queries in polytime. In this work, we additionally consider *determinism* and *structured decomposability*, which enable tractable inference of larger classes of queries, as well as closed-form maximum-likelihood parameter estimation and improved model interpretability. A PC is deterministic if every sum node partitions its support (assignments with non-zero probability); and it is structured decomposable if any two product nodes of shared scope decompose the variables the same way. Detailed definitions are in Appendix A.

Closely related to our approach is STRUDEL [Dang et al., 2022b], a heuristic-driven algorithm for learning deterministic and structured-decomposable PCs in the offline setting. It first learns a Chow-Liu tree (CLT) [Chow and Liu, 1968] to compile an initial PC structure and then iteratively expands the circuit using structural transformations through conditioning (i.e. *splits*). While STRUDEL is highly effective at discovering complex dependencies in static datasets, its core methodology caters to an offline paradigm.

3 ONLINE STRUCTURE LEARNING

We introduce online incremental structure learning for deterministic and structured-decomposable PCs. Our proposed framework consists of the following components that are maintained and updated with each incoming mini-batch[1] of data in a streaming fashion. We start with (1) an initial structure and (2) maintain sufficient statistics based on *edge flows*; in each iteration, we either (3) grow the circuit structure with a *split* or (4) reduce its size with a *merge*, when necessary; lastly, (5) a decay factor can be used to better adapt to non-stationary data distributions. Algorithm 1 summarizes the order of operations.

[1]Our mini-batch representation is mathematically equivalent to streaming single data points and performing structural updates (split, merge, decay) every n data points.

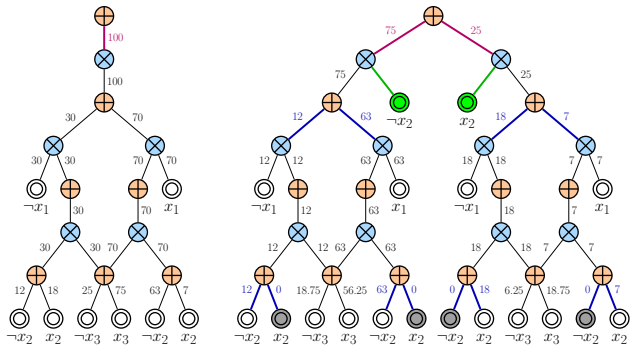

Figure 1: A PC before (left) and after (right) a split on x_2.

Flow-based Sufficient Statistics We use the concept of (circuit) edge flows [Choi et al., 2021, Dang et al., 2022a] to maintain an accurate representation of how incoming datapoints interact with the current circuit structure.

Definition 1 (Edge Flows)**.** *Let $\mathcal{C}$ be a PC over variables $\mathbf{X}$, $\mathcal{D}$ be a dataset, and $\mathbb{I}(\cdot)$ denote the indicator function evaluating to 1 or 0. For an edge $e_{i,j}$ connecting node n_i to child node n_j, the **aggregate flow** $a(i,j;\mathcal{D})$ is the number of samples in $\mathcal{D}$ whose evaluation path traverses $e_{i,j}$:*

$$a(i,j;\mathcal{D}) = \sum_{\mathbf{x}\in\mathcal{D}} \mathbb{I}(e_{i,j} \textit{ is active given } \mathbf{x})$$

*For any variables X_u, X_v in the scope of n_i, the **marginal flow** m_u and **joint flow** $J_{u,v}$ capture the variable-level co-occurrences traversing the edge:*

$$m_u(i,j;\mathcal{D}) = \sum_{\mathbf{x}\in\mathcal{D}} \mathbb{I}(e_{i,j} \textit{ is active given } \mathbf{x}) \cdot \mathbf{x}_u$$

$$Ju,v(i,j;\mathcal{D}) = \sum_{\mathbf{x}\in\mathcal{D}} \mathbb{I}(e_{i,j} \textit{ is active given } \mathbf{x}) \cdot \mathbf{x}_u\mathbf{x}_v$$

where $\mathbf{x}_u, \mathbf{x}_v \in \{0,1\}$ denote the assignments to X_u, X_v in sample $\mathbf{x}$.

For deterministic PCs, the closed-form maximum likelihood estimation for $\theta_{i,j}$ is simply the ratio of its aggregate flow to the total flow out of its parent node: $\theta_{i,j} = \frac{a(i,j;\mathcal{D})}{\sum_k a(i,k;\mathcal{D})}$.

Distribution-Preserving Splits To dynamically increase the expressivity of the PC and capture emerging statistical dependencies, we perform localized conditional splits similar to STRUDEL [Dang et al., 2022b]. Specifically, we select a target edge and a variable within its scope, then perform the split by duplicating the sub-circuit conditioned on that variable (Figure 1).

A good heuristic for determining a candidate edge for splitting needs to be both computationally cheap and informative.

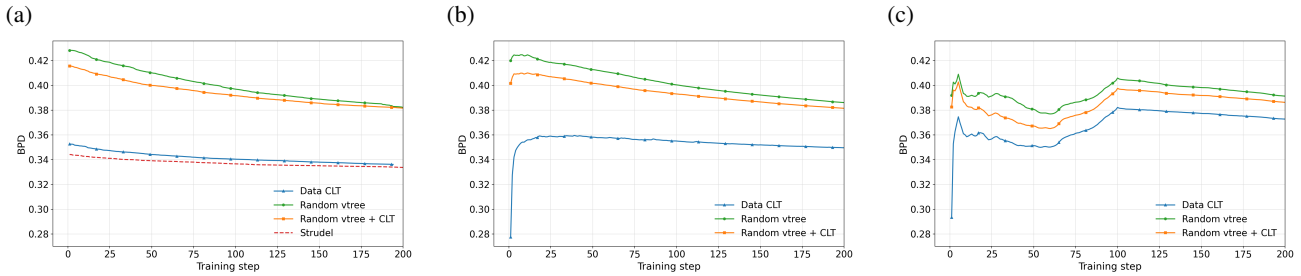

Figure 2: Observed average training BPD (↓) over 20 datasets for 200 structural updates for **(a)** our offline baseline with $\lambda = 0$, **(b)** static data streams with $\lambda = 0.99$, and **(c)** non-static data streams with $\lambda = 0.99$. For **Random vtree** and **Random vtree + CLT**, we average across three runs per dataset. Per-dataset results are provided in Appendix C.2.

For this reason, we rely on aggregate flows: the edge $e_{i,j}$ routing the highest mass of data is always prioritized for splitting. Aggregate flows easily highlight high impact edges and require only a single scalar per edge that can be updated during inference.

Once a candidate edge $e_{i,j}$ is selected, we must choose an optimal variable X_k within its scope to condition on. The purpose of creating these localized conditional splits is to dynamically increase the expressivity of the probabilistic circuit, allowing us to capture emerging statistical dependencies. Because mutual information gives us a strong estimate of how much information can be gained from conditioning on a specific variable, we select X_k by maximizing the average pairwise mutual information towards all other variables in the scope: $X_k = \text{argmax}_{X_k} \overline{\text{MI}}_{i,j}(X_k)$. If an edge has no variable left to meaningfully split on, the next best edge is chosen and variable selection reevaluated. We only consider mutual information between variables that are structurally separated by a node's children, which prevents unnecessarily deep circuits.

Although we cannot use the entire dataset to update the parameters after a split, as in batch learning, we have sufficient information on how data historically flows through the circuit to distribute the flows in a newly-split circuit in a distribution-preserving manner, illustrated in Figure 1. By splitting edge $e_{i,j}$ on variable X_k, the subgraph at product node n_j is duplicated and conditioned on X_k: descendants whose scope includes X_k are deep-copied, while others are partially copied based on depth or mutual information. The distribution is preserved through this split using marginal and joint flows, as detailed in Appendix B.1, and is subsequently updated using future data streams.

Merging To minimize structural blowup, we *merge* existing conditional branch pairs that follow similar distributions once the circuit reaches a size threshold. In particular, we use Jeffreys' divergence (symmetric Kullback-Leibler divergence) as our similarity metric, which can be computed in quadratic time for deterministic and structured-decomposable PCs [Liang and Van den Broeck, 2017, Vergari et al., 2021]: $\text{JD}(P \parallel Q) = \frac{1}{2} D_{\text{KL}}(P \parallel Q) + \frac{1}{2} D_{\text{KL}}(Q \parallel P)$. The candidates for potential merges are two sibling branches that share the same parent sum node and differ only by conditioning on a variable X_k. Each iteration, we choose the candidate pair with the lowest divergence, prune the branch with more parameters, undo the conditioning on X_k, and project its flow statistics into the retained branch. We further discuss this process in Appendix B.2.

Decay Factor and Replay Buffer To adapt to evolving distributions, we introduce a tunable decay parameter $\lambda \in (0, 1]$ that multiplicatively scales down edge flows (i.e. $\lambda = 1$ means no decay). This constant factor ensures that older observations exponentially lose their influence over time, enabling the model to prioritize recent data. Additionally, we implement an optional replay buffer that statistically samples incoming data points to maintain a size-restricted empirical estimate of the distribution. We utilize a weighted buffer that prioritizes removal of older data points if the maximum size is reached.

4 EXPERIMENTS

We evaluate our framework under *static* and *shifting* distributions on twenty density-estimation benchmarks [Van Haaren and Davis, 2012, Lowd and Davis, 2010] and downsized (14×14) binarized MNIST inspired by Larochelle and Murray [2011]. We compare three initializations: Data CLT, which uses mutual information for data-driven hierarchical conditioning (akin to STRUDEL); Random vtree, an unconditioned random deterministic circuit initialization; and Random vtree + CLT, which adds CLT conditioning to a random topology to isolate the effect of data-driven structure. Complete settings and additional per-dataset, merging, and runtime results are provided in Appendix C.

4.1 STATIC DISTRIBUTIONS

Our initial experiments establish the baseline performance of our method on stable, non-shifting data distributions.

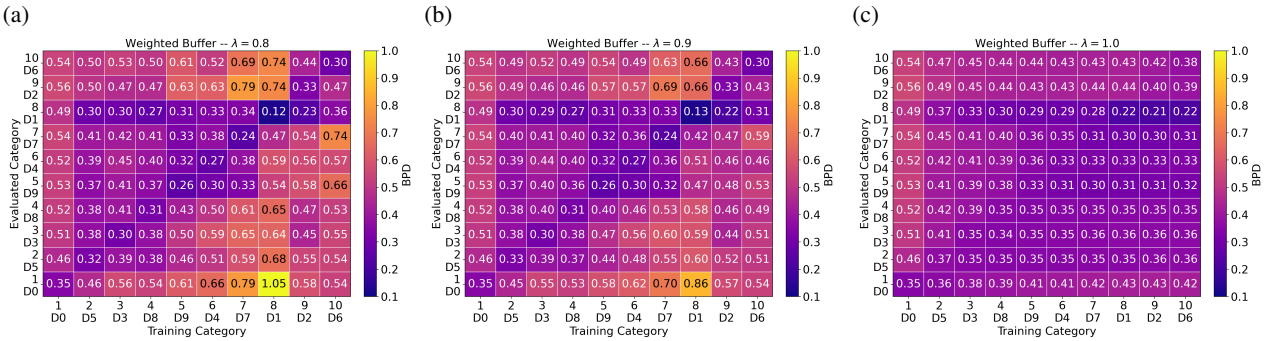

Figure 3: Average training BPD on binarized MNIST under the label-induced shift experiment with replay buffer. Evaluations are performed after training on each digit (x-axis), measuring the average BPD of the specified digits (y-axis). Subfigures correspond to **(a)** $\lambda = 0.8$, **(b)** $\lambda = 0.9$, and **(c)** $\lambda = 1.0$.

Offline Baseline We first validate our method in a full-batch setting, where the entire training set is processed between every structural manipulation and previously accumulated statistics are fully reset ($\lambda = 0$). Figure 2a shows that our method performs almost identical to STRUDEL [Dang et al., 2022b] (measured by bits-per-dimension (BPD)), the state-of-the-art for offline learning of deterministic PCs, when using the same data-based CLT initializations, which significantly outperform random-vtree initializations.

Continuous Static Stream We next simulate a stationary online environment by randomly partitioning each dataset into 100 batches per epoch, with a structural update performed between successive batches. Because batches are formed randomly, each follows the same underlying distribution in expectation, differing only due to sampling variation. As seen in Figure 2b, our method has similar convergence rates for all initializations and a stable continuous decrease in BPD after the first few iterations.

4.2 SHIFTING DISTRIBUTIONS

We additionally evaluate the performance of our method in data streams with non-static distributions.

Simulated Shift We introduce distribution shifts while maintaining the same interleaved structural-update schedule. Rather than randomly partitioning each dataset, we form 100 batches using k-means clustering. We sequence these batches by starting with the cluster whose centroid is furthest from the global average, then greedily traversing to the nearest unvisited centroid in Euclidean space, producing a smoothly evolving data stream. Figure 2c shows similar stable convergence behaviors as the static data stream once all data has been observed at least once. The advantage of data-based initialization decreases substantially in the online settings, specifically for non-static data streams.

Label-Induced Shift We apply a similar distribution shift to the downsized binarized MNIST, using the actual digit labels to define the data clusters. We sequence the exposure of these label-based clusters using the same greedy distance heuristic between centroids. However, to simulate prolonged exposure before a definitive shift, we expose the model to each digit for 50 full passes before transitioning to the next; each pass consists of updating flows once for every sample, followed by a single structural update. Figure 3 highlights the difference between three different decay settings, showing that smaller λ values adapt more strongly to the current distribution at the cost of more aggressive forgetting, while larger values retain information more evenly across prior observations. Figure 3c further shows that performance on previously observed digits can remain stable or even improve after the stream has transitioned to later digits. Similar behavior is observed without the replay buffer; see Appendix C.3.

5 CONCLUSION

This work proposed a novel algorithm for online structure learning of deterministic and structured-decomposable probabilistic circuits (PCs). Our method is able to perform efficient restructuring of PCs and aggregate parameter updates that allow the model to adapt to distribution shifts with varying strength. Empirical evaluations demonstrate that our method achieves stable learning in both static and evolving environments. A limitation is the lack of principled scheduling for structural updates. While divergence-based merging is compatible with our method, the divergence threshold is difficult to tune: high values can merge away useful conditional structure, while low values allow rapid circuit growth. Splits are likewise triggered after a fixed number of samples. We therefore simply alternate between splitting and merging based on circuit size and leave adaptive scheduling to future work.

Acknowledgements

This work was supported in part by the AFOSR grant FA9550-25-1-0320.

References

Y Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic circuits: A unifying framework for tractable probabilistic models. *UCLA. URL: http://starai. cs. ucla. edu/papers/ProbCirc20. pdf*, 6, 2020.

YooJung Choi, Meihua Dang, and Guy Van den Broeck. Group fairness by probabilistic modeling with latent fair decisions. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, Feb 2021.

CKCN Chow and Cong Liu. Approximating discrete probability distributions with dependence trees. *IEEE transactions on Information Theory*, 14(3):462–467, 1968.

Meihua Dang, Anji Liu, and Guy Van den Broeck. Sparse probabilistic circuits via pruning and growing. *Advances in Neural Information Processing Systems*, 35: 28374–28385, 2022a.

Meihua Dang, Antonio Vergari, and Guy Van den Broeck. Strudel: A fast and accurate learner of structured-decomposable probabilistic circuits. *International Journal of Approximate Reasoning*, 140:92–115, 2022b.

Meihua Dang, Anji Liu, Xinzhu Wei, Sriram Sankararaman, and Guy Van den Broeck. Tractable and expressive generative models of genetic variation data. *bioRxiv*, 2023.

Adnan Darwiche. A logical approach to factoring belief networks. *KR*, 2:409–420, 2002.

Aaron Dennis and Dan Ventura. Online structure-search for sum-product networks. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 155–160. IEEE, 2017.

Gennaro Gala, Cassio de Campos, Antonio Vergari, and Erik Quaeghebeur. Scaling continuous latent variable models as probabilistic integral circuits. *Advances in Neural Information Processing Systems*, 37:10959–10987, 2024.

Agastya Kalra, Abdullah Rashwan, Wei-Shou Hsu, Pascal Poupart, Prashant Doshi, and Georgios Trimponias. Online structure learning for feed-forward and recurrent sum-product networks. *Advances in Neural Information Processing Systems*, 31, 2018.

Hugo Larochelle and Iain Murray. The neural autoregressive distribution estimator. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 29–37. JMLR Workshop and Conference Proceedings, 2011.

Sang-Woo Lee, Min-Oh Heo, and Byoung-Tak Zhang. Online incremental structure learning of sum–product networks. In *International conference on neural information processing*, pages 220–227. Springer, 2013.

Yitao Liang and Guy Van den Broeck. Towards compact interpretable models: Shrinking of learned probabilistic sentential decision diagrams. In *IJCAI 2017 Workshop on Explainable Artificial Intelligence (XAI)*, 2017.

Yitao Liang, Jessa Bekker, and Guy Van den Broeck. Learning the structure of probabilistic sentential decision diagrams. In *Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI)*, 2017.

Anji Liu, Kareem Ahmed, and Guy Van den Broeck. Scaling tractable probabilistic circuits: A systems perspective. *arXiv preprint arXiv:2406.00766*, 2024.

Lorenzo Loconte, Aleksanteri Sladek, Stefan Mengel, Martin Trapp, Arno Solin, Nicolas Gillis, and Antonio Vergari. Subtractive mixture models via squaring: Representation and learning. In *International Conference on Learning Representations*, volume 2024, pages 15751–15785, 2024.

Lorenzo Loconte, Stefan Mengel, and Antonio Vergari. Sum of squares circuits. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 19077–19085, 2025.

Daniel Lowd and Jesse Davis. Learning markov network structure with decision trees. In *2010 IEEE International Conference on Data Mining*, pages 334–343. IEEE, 2010.

Robert Peharz, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Kristian Kersting, and Zoubin Ghahramani. Probabilistic deep learning using random sum-product networks. *arXiv preprint arXiv:1806.01910*, 2018.

Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 689–690. IEEE, 2011.

Tahrima Rahman, Prasanna Kothalkar, and Vibhav Gogate. Cutset networks: A simple, tractable, and scalable approach for improving the accuracy of chow-liu trees. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 630–645. Springer, 2014.

Jan Van Haaren and Jesse Davis. Markov network structure learning: A randomized feature generation approach. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, pages 1148–1154, 2012.

Antonio Vergari, YooJung Choi, Anji Liu, Stefano Teso, and Guy Van den Broeck. A compositional atlas of tractable circuit operations: From simple transformations to complex information-theoretic queries. *arXiv preprint arXiv:2102.06137*, 2021.

Honghua Zhang, Brendan Juba, and Guy Van den Broeck. Probabilistic generating circuits. In *International conference on machine learning*, pages 12447–12457. PMLR, 2021.

Online Structure Learning of Probabilistic Circuits (Supplementary Material)

Stefan Wagner[1] **Anish Kuttetira**[1] **YooJung Choi**[1]

[1]School of Computing and Augmented Intelligence, Arizona State University

A PROBABILISTIC CIRCUITS BACKGROUND

Definition 2 (Probabilistic Circuits)**.** *A probabilistic circuit (PC) C is a directed acyclic graph (DAG), where each node n is either a leaf, product, or sum node parameterized by θ. A leaf node captures a function over a set of random variables $\boldsymbol{X}_n$, also called its scope $\phi(n)$. Sum and product nodes receive their inputs from their children in(n), and their scope is given by $\phi(n) = \cup_{c \in in(n)} \phi(c)$. The probability distribution $p_n(\boldsymbol{x})$ captured by any given node in the circuit can be recursively defined as:*

$$p_n(\boldsymbol{X}_n) = \begin{cases} f_n(\boldsymbol{X}_n) & \text{if } n \text{ is a leaf node}, \\ \prod_{c \in in(n)} p_c(\boldsymbol{X}_c) & \text{if } n \text{ is a product node}, \\ \sum_{c \in in(n)} \theta_{n,c} p_c(\boldsymbol{X}_c) & \text{if } n \text{ is a sum node}. \end{cases}$$

Definition 3 (Smoothness)**.** *A probabilistic circuit C is smooth if all sum nodes $n \in C$ only have children that share the same scope: $\phi(n) = \phi(c), \forall c \in in(n)$.*

Definition 4 (Decomposability)**.** *A probabilistic circuit C is decomposable if all product nodes $n \in C$ only have children with disjoint scopes $\cap_{c \in in(n)} \phi(c) = \emptyset$.*

Definition 5 (Determinism)**.** *A probabilistic circuit C is deterministic if for every sum node n at most one child node provides non-zero input for every possible circuit input: $\forall \mathbf{x} \in \boldsymbol{X}, |\{c \in in(n) \mid p_c(\mathbf{x}_c) > 0\}| \leq 1$.*

Definition 6 (Structured decomposability)**.** *A probabilistic circuit C is structured decomposable if any two product nodes $n, m \in C$ of shared scope decompose the same way: $\phi(n) = \phi(m) \implies \{\phi(c) \mid c \in in(n)\} = \{\phi(c') \mid c' \in in(m)\}$.*

B EDGE FLOW MAINTENANCE

B.1 SPLITTING

For nodes within the copied branches that contain X_k in their scope, we set their aggregate flows based on the marginal flows to preserve the distributions.

$$a^*(i, j; \mathcal{D}) = \begin{cases} m_k(i, j; \mathcal{D}), & \text{if } x_k = 1, \\ a(i, j; \mathcal{D}) - m_k(i, j; \mathcal{D}), & \text{otherwise.} \end{cases}$$

Similarly, we use the joint flows to set the marginal flows to preserve as much statistical information as possible throughout iterations.

$$m_u^*(i, j; \mathcal{D}) = \begin{cases} J_{u,k}(i, j; \mathcal{D}), & \text{if } x_k = 1, \\ m_u(i, j; \mathcal{D}) - J_{u,k}(i, j; \mathcal{D}), & \text{otherwise.} \end{cases}$$

Finally, to initialize the joint flows for the newly created branches, we assume local independence between the variables within the newly conditioned context. Because the structural split fundamentally alters the underlying data routing, perfectly

partitioning the historical off-diagonal joint statistics is intractable without maintaining higher-order variable interactions. Thus, we reconstruct the new joint flows as the normalized product of the updated marginals:

$$J^*_{u,v}(i,j;\mathcal{D}) = \frac{m^*_u(i,j;\mathcal{D}) \cdot m^*_v(i,j;\mathcal{D})}{a^*(i,j;\mathcal{D})}$$

For compactness and further reference in Appendix B.2, let $\mathcal{F}_{i,j}$ denote the flow statistics stored at an edge. We denote the conditional partition defined above by

$$\mathcal{F}(e_{i,j}) = \big(a(i,j;\mathcal{D}), \mathbf{m}(i,j;\mathcal{D}), \mathbf{J}(i,j;\mathcal{D})\big), \qquad \big(\mathcal{F}^+(e_{i,j}), \mathcal{F}^-(e_{i,j})\big) = \mathrm{Cond}_{X_k}(\mathcal{F}(e_{i,j}))\,.$$

Here, the aggregate and marginal flows are obtained from the available marginal and joint statistics, while off-diagonal conditional joint flows use the stated second-order approximation. $\mathcal{F}^+(e_{i,j})$ and $\mathcal{F}^-(e_{i,j})$ represent the copy of an edge $e_{i,j}$ in the branch where $x_k = 1$ and $x_k = 0$ respectively.

For all other nodes that do not contain X_k in its scope, we leave the ratios of edge flows unchanged and normalize their magnitude using the weights to match the current structure.

B.2 MERGING

Let e^{keep} and e^{remove} denote the retained and removed branch edges. Because we cannot assume that both branches share the same history of conditional branching, we approximate the flows of the removed branch with its parent edge e^{remove}. The flow routing mechanism is akin to recreating the splits of the retained branch for our $\mathcal{F}(e^{\mathrm{remove}})$ flows as discussed in Appendix B.1.

Let $\Delta\mathcal{F}(e^{\mathrm{keep}}) = \mathcal{F}(e^{\mathrm{remove}})$. For any edge pair $(e_{i,+}, e_{i,-})$ that either conditions n_i on X_k or represents a leaf distribution over X_k in the retained subgraph, we can define the flow updates as:

$$\big(\Delta\mathcal{F}(e_{i,+}), \Delta\mathcal{F}(e_{i,-})\big) = \sum_{p \in \mathit{out}(n_i)} \mathrm{Cond}_{X_k}\big(\Delta\mathcal{F}(e_{p,i})\big),$$

where $\mathit{out}(n_i)$ defines the set of parent nodes for node n_i. For all other edges, we simply set them equivalent to their parent flow contribution:

$$\Delta\mathcal{F}(e_{i,j}) = \sum_{p \in \mathit{out}(n_i)} \big(\Delta\mathcal{F}(e_{p,i})\big),$$

Finally, we update the flows in our retained branch:

$$\mathcal{F}^*(e_{i,j}) = \mathcal{F}(e_{i,j}) + \Delta\mathcal{F}(e_{i,j})$$

Repeated application of the conditioning operator successfully preserves the aggregate, marginal, and joint information at each step. However, routing data perfectly through deep conditional structures would require higher-order statistics that our method does not track. Because of this, reconstructing joint flows relies on an approximation, and this error can accumulate along paths with multiple consecutive splits. Despite this limitation, our method ensures the resulting circuit remains deterministic and normalized.

In practice, our divergence-based candidate selection naturally controls the impact of these approximation errors. Smaller subgraphs with fewer downstream splits typically have a lower Jeffreys divergence, making them the most likely candidates for merging. In contrast, deeper branches with highly differentiated structures are structurally more susceptible to greater divergence. As a result, most merges occur exactly where they are safest: in areas of the circuit that require only a few successive conditional approximations, preventing error blowup.

C EXPERIMENTAL DETAILS

C.1 GENERAL EXPERIMENTAL SETUP

Unless otherwise specified, we run all our experiments with a decay factor of $\lambda = 0.99$. To prevent immediate reversals of structural operations, edges created by a split must remain unchanged for 20 structural updates before becoming eligible for merging, while edges affected by a merge cannot be split again for 5 structural updates. Unless otherwise specified, experiments using random-vtree initialization are averaged over three independently seeded runs. We use a batch size of 256 for the parameter update pass in all our experiments.

C.2 20-DATASET BENCHMARK DETAILS

We evaluate our methodology on the 20 binary density-estimation benchmark datasets, specifically the train and test data found in `github.com/UCLA-StarAI/Density-Estimation-Datasets` without any preprocessing. We run STRUDEL with greedy splitting as a baseline.

The full-batch static evaluations use a decay factor of $\lambda = 0$; for the stationary and non-static streams we use $\lambda = 0.99$. We set the upscale limit of each run to $N_{\text{max}} = 30,000$, and we allow downscaling to $N_{\text{downsize}} = 24,000$ when merging is enabled. Runs were allowed up to 1,000 iterations, but not all runs reached the ceiling within runtime constraints. We therefore use the common first 200 structural updates for cross-dataset comparisons in Figure 2, since all selected runs reach this horizon. Individual runs for our *offline baseline*, *static data streams*, and *non-static data streams* are shown in Figure 4, Figure 5, and Figure 6 respectively.

C.3 BINARIZED MNIST DETAILS

For the label-shift experiment from Section 4.2, we initialize circuits as a random vtree and set $N_{\text{max}} = 25,000$ and $N_{\text{downsize}} = 23,750$ ($N_{\text{max}} \cdot 0.95$). We trained on decay factors $\lambda = \{0.8, 0.9, 1.0\}$ both with and without a weighted buffer. When enabled, the buffer has a capacity equal to 25% of the dataset, and each incoming observation is added with probability 0.1. Figure 7 reports the corresponding results without the replay buffer.

C.4 MERGING EFFICIENCY

We evaluate whether repeated split-merge cycles can improve structure learning without increasing the final circuit size. Across the 20 benchmark datasets, we target a parameter budget of $N_{\text{target}} = 10,000$ and consider three $(N_{\text{max}}; N_{\text{downsize}})$ pairs: $(11,000; 9,000)$, $(12,000; 8,000)$, and $(13,000; 7,000)$. For each setting, we run Algorithm 1 for n complete upscale-downscale phases, followed by an additional 100-epoch settling period of parameter-only updates over the entire dataset. All circuits are initialized from unconditioned random vtrees, and results are averaged over three independently seeded runs per dataset.

Figure 8 shows both the resulting phase trajectories and the relative BPD improvement as the number of completed merge phases increases. Repeated split-merge cycles consistently improve BPD while returning the circuit toward the same target parameter budget, indicating that merging can serve not only as a mechanism for controlling model size, but also as a means of revising and improving the learned structure.

C.5 LIMITATION OF MERGING

We do not use CLT initialization in experiments involving merging. Compiling a CLT produces a highly shared DAG. This representation remains compatible with splitting, but merging a conditioned branch can require modifying a subgraph that is also referenced by other contexts. Preserving those contexts then requires duplicating the shared subgraph, which can offsetor even reversethe intended reduction in circuit size. Consequently, our current merge experiments use unconditioned random-vtree initializations. The reported merge results should therefore not be interpreted as demonstrating size control for CLT-initialized circuits. Supporting shared-subgraph merges without duplication is left for future work.

C.6 RUNTIME

We evaluate runtime over 500 structural updates without imposing a maximum circuit size. For each of the 20 benchmark datasets, we initialize an unconditioned random vtree and average over three independently seeded runs. We use the stationary streaming setting, sampling batches of 256 observations from the dataset and performing one structural update after every batch, resulting in 500 splits per run. The results can be seen in Table 1. Mean update time remains below 0.9 seconds per step for every dataset, but runtime and variability increase substantially for certain datasets where the circuit undergoes the greatest structural growth, such as `kosarek` and `tretail`.

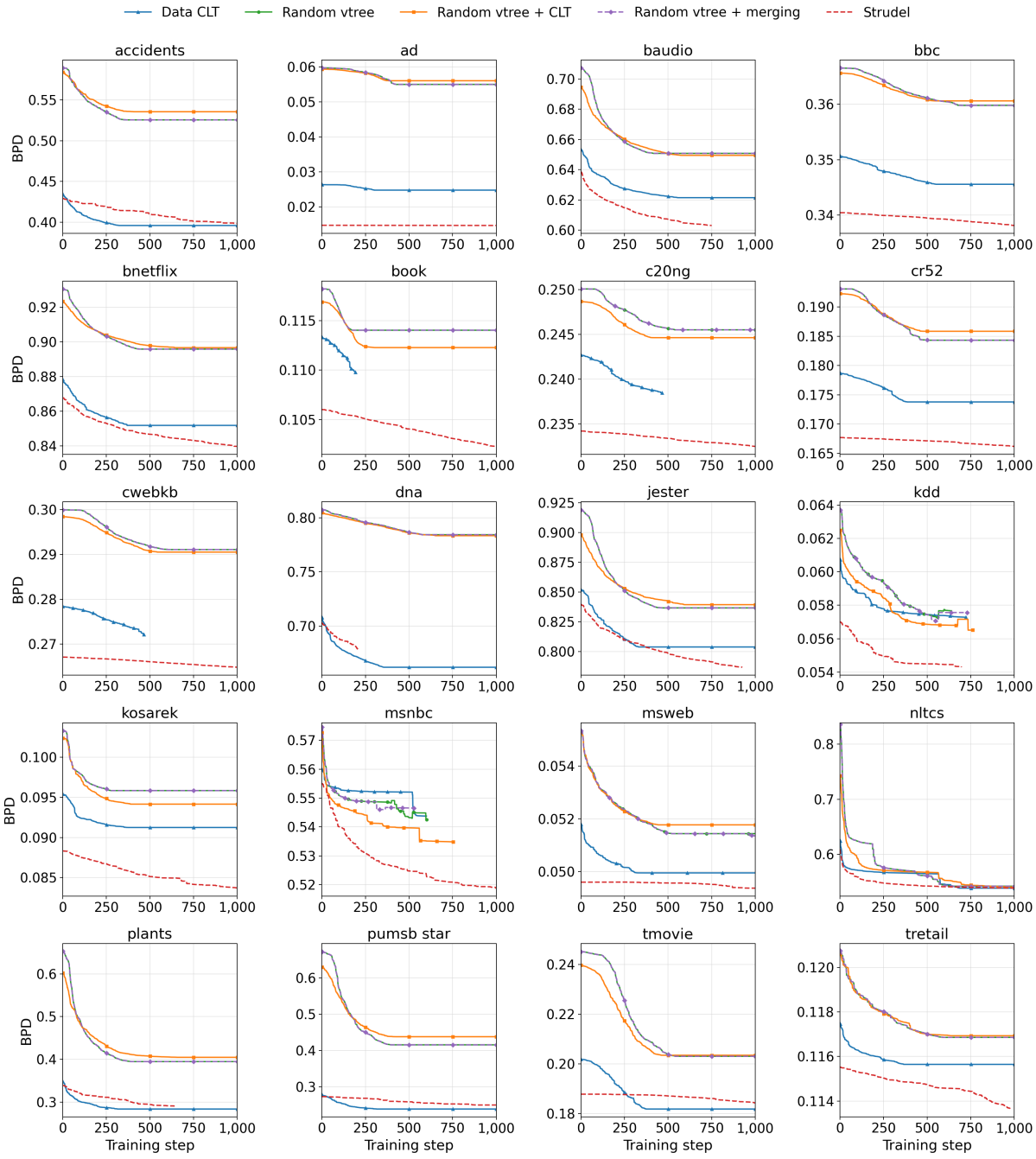

Figure 4: Per-dataset training BPD over up to 1,000 structural updates with $\lambda = 0$ for the offline baseline. **Random vtree**, **Random vtree + CLT**, and **Random vtree + merging** are averaged across three runs per dataset. All runs use $N_{\max} = 30,000$; merging runs use $N_{\text{downsize}} = 24,000$.

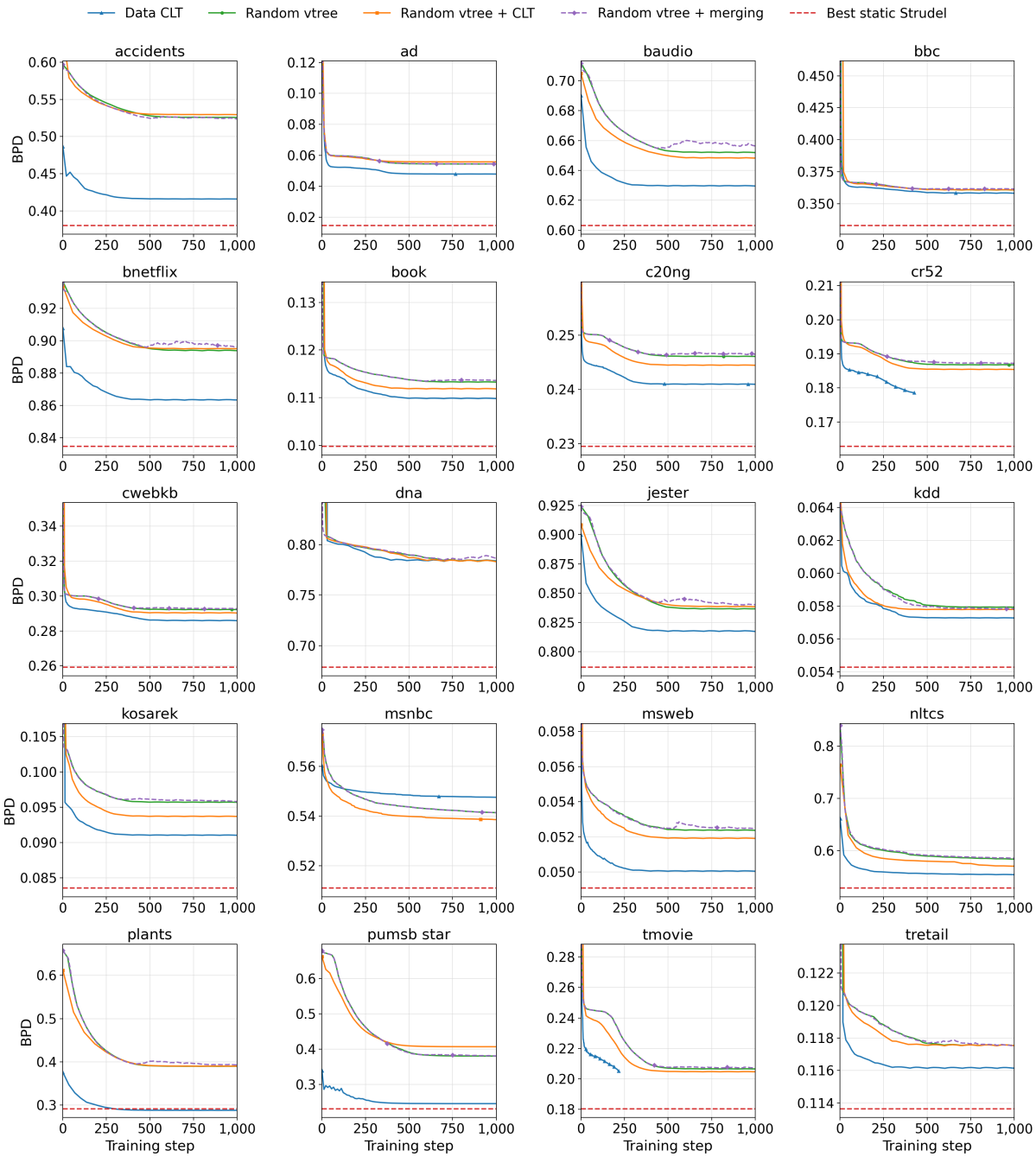

Figure 5: Per-dataset training BPD over up to 1,000 structural updates with $\lambda = 0.99$ for static data streams. **Random vtree**, **Random vtree + CLT**, and **Random vtree + merging** are averaged across three runs per dataset. All runs use $N_{\max} = 30,000$; merging runs use $N_{\text{downsize}} = 24,000$. The dashed line indicates the best offline STRUDEL BPD and is included only as a static reference; STRUDEL is not trained on the streaming sequence.

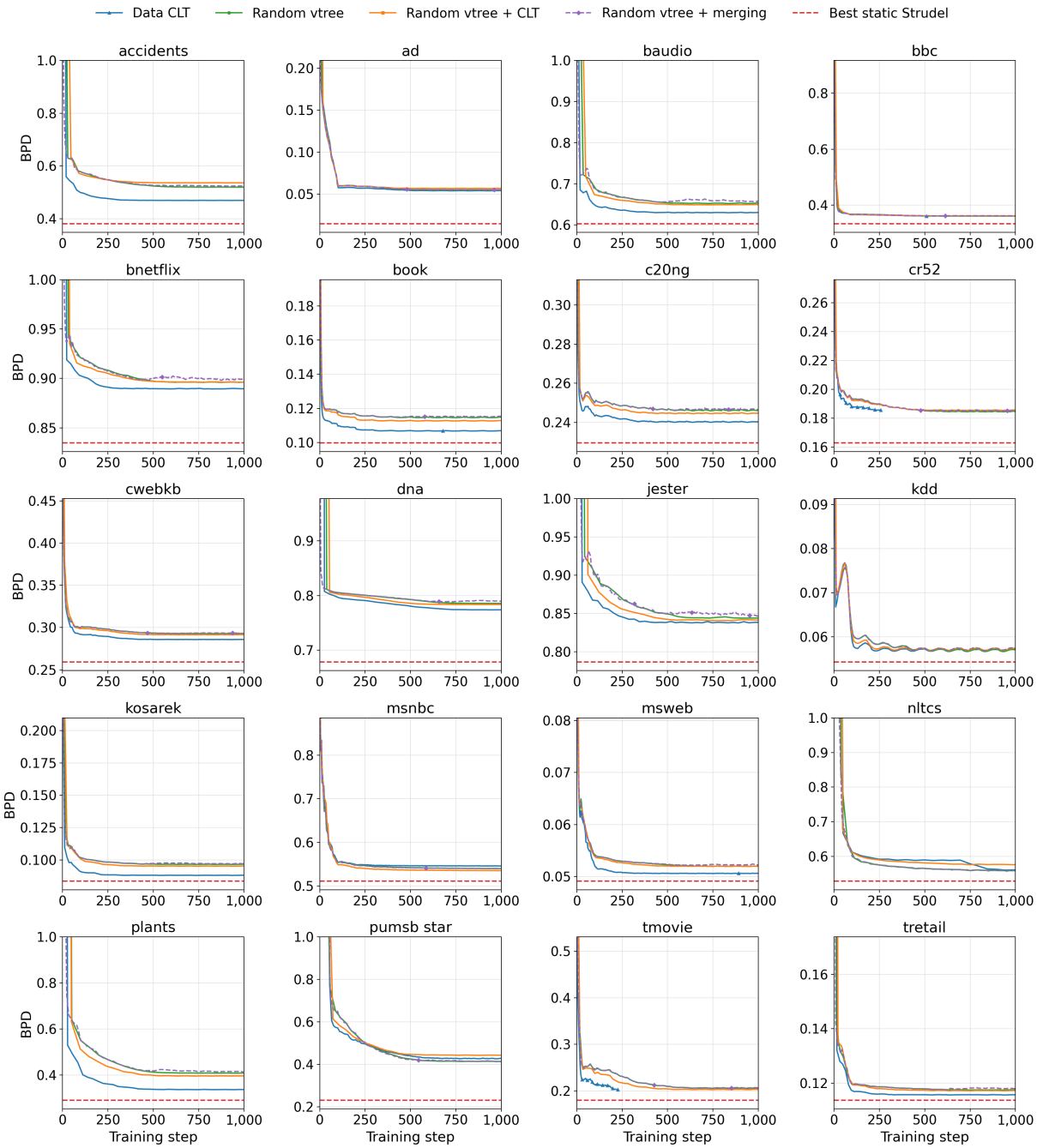

Figure 6: Per-dataset training BPD over up to 1,000 structural updates with $\lambda = 0.99$ for non-static data streams. **Random vtree**, **Random vtree + CLT**, and **Random vtree + merging** are averaged across three runs per dataset. All runs use $N_{\max} = 30,000$; merging runs use $N_{\text{downsize}} = 24,000$. The dashed line indicates the best offline STRUDEL BPD and is included only as a static reference; STRUDEL is not trained on the streaming sequence.

Dataset	Dim	Mean (s) / Iter.	Max (s) / Iter.	Total (s) 500 steps	Final parameters
accidents	111	0.5269	1.950	263.5 ± 26.9	38,095 ± 2,593
ad	1,556	0.8639	1.317	432.0 ± 23.0	8,237 ± 8
baudio	100	0.4489	1.620	224.4 ± 24.9	37,116 ± 6,209
bbc	1,058	0.4826	0.856	241.3 ± 6.8	6,914 ± 13
bnetflix	100	0.2403	0.802	120.2 ± 3.0	28,736 ± 1,099
book	500	0.1396	0.317	69.8 ± 1.8	8,144 ± 311
c20ng	910	0.5298	0.804	264.9 ± 6.2	6,504 ± 45
cr52	889	0.3745	0.594	187.3 ± 29.6	6,379 ± 27
cwebkb	839	0.3681	0.592	184.0 ± 28.0	6,430 ± 43
dna	180	0.2306	0.863	115.3 ± 8.1	23,397 ± 1,358
jester	100	0.1896	0.751	94.8 ± 9.1	34,566 ± 4,696
kdd	64	0.3658	1.106	182.9 ± 9.6	38,614 ± 2,656
kosarek	190	0.8063	6.506	403.2 ± 158.6	114,695 ± 53,914
msnbc	17	0.1116	0.371	55.8 ± 11.6	11,853 ± 2,649
msweb	294	0.2596	2.318	129.8 ± 23.5	21,083 ± 8,103
nltcs	16	0.0804	0.262	40.2 ± 2.8	12,138 ± 862
plants	69	0.3440	0.952	172.0 ± 6.1	39,067 ± 1,612
pumsb_star	163	0.1653	0.605	82.7 ± 5.3	22,495 ± 2,306
tmovie	500	0.1098	0.208	54.9 ± 0.4	7,796 ± 330
tretail	135	0.8861	5.011	443.0 ± 264.2	128,197 ± 65,590

Table 1: Runtime over 500 structural updates. **Mean** denotes the average per-update runtime; **Max** the largest single-update runtime across all three runs; **Total** the mean runtime over all 500 updates; and **Final parameters** the mean circuit parameter count after the final update (rounded to the nearest integer). Reported ± values denote standard deviations across seeds.

C.7 HARDWARE/SOFTWARE SPECIFICATIONS

All experiments were conducted on a machine running Rocky Linux 8.10 with 2×AMD EPYC 7713 CPUs and 2 TB DDR4 RAM. Multiple runs were executed simultaneously across all 128 CPU cores and without the use of a GPU. We limited each individual dataset evaluation to 12 hours of runtime and 32GB of RAM. We run our experiments using Python 3.10.17, Rust 1.92.0, Maturin 1.13.1, and NumPy 2.2.5.

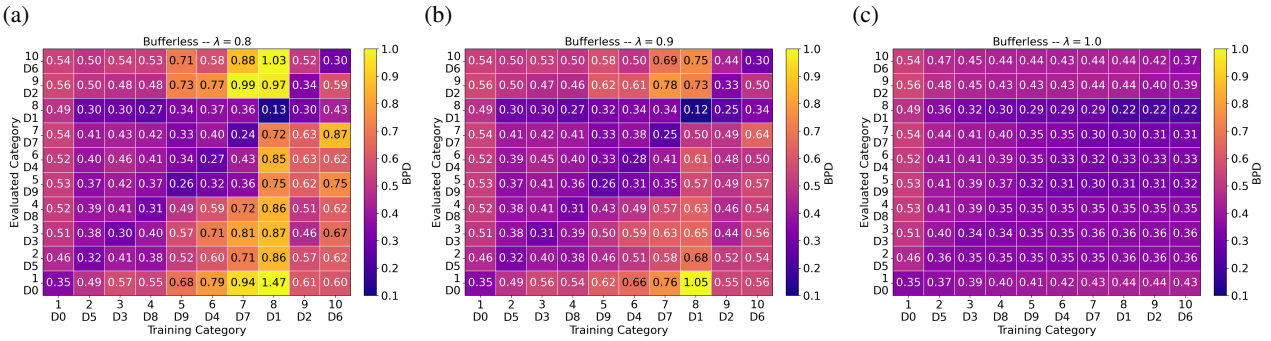

Figure 7: Average training BPD on binarized MNIST under the label-induced shift experiment without a buffer. Evaluations are performed after training on each digit (x-axis), measuring the average BPD of the specified digits (y-axis). Subfigures correspond to **(a)** $\lambda = 0.8$, **(b)** $\lambda = 0.9$, and **(c)** $\lambda = 1.0$.

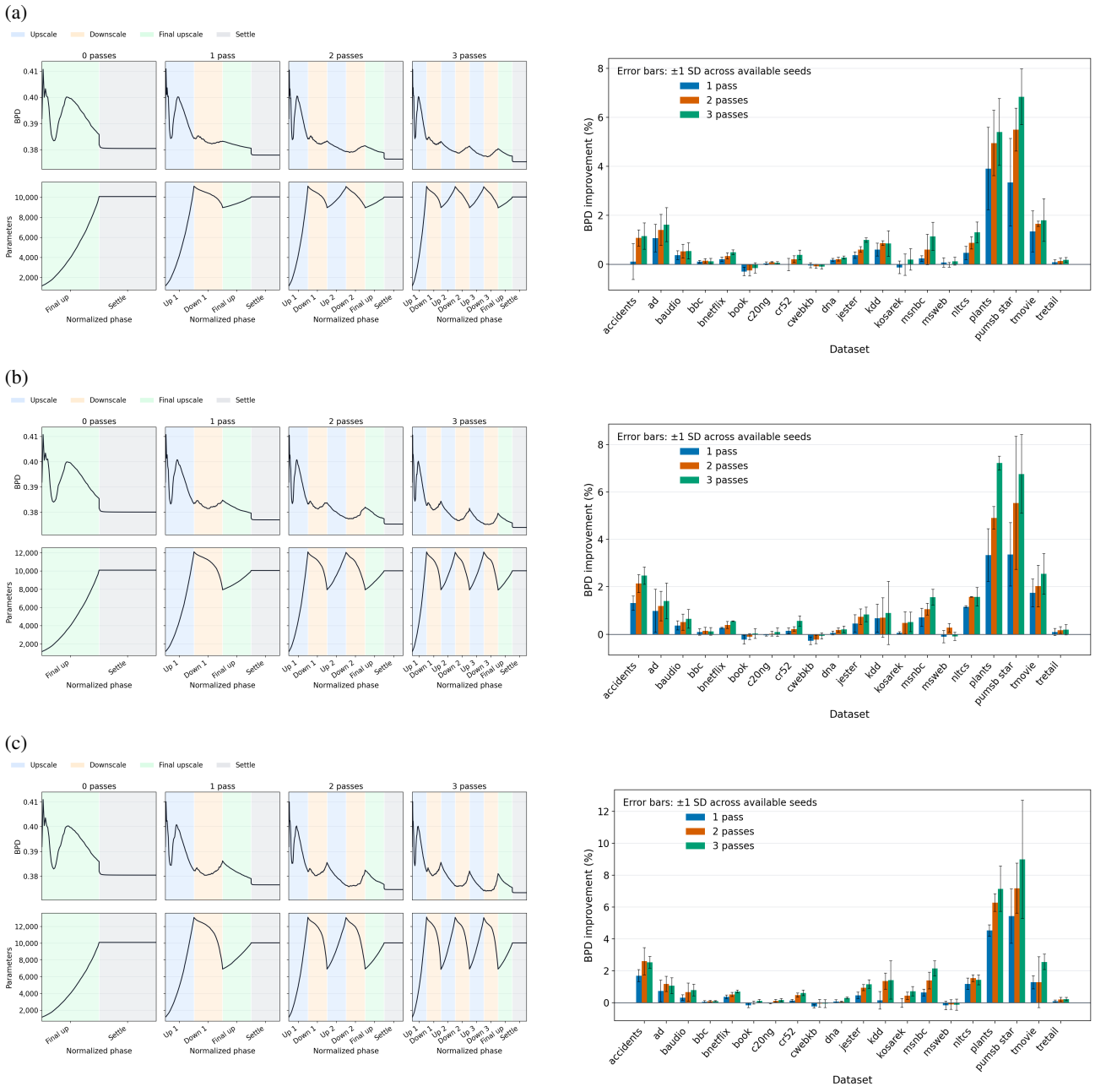

Figure 8: Phase trajectory and corresponding BPD improvement (%) from merging over the simulated distribution shift setting, with (left column) phase trajectories and (right column) BPD improvements for $(N_{\max}; N_{\text{downsize}})$ pairs of **(a)** $(11,000; 9,000)$, **(b)** $(12,000; 8,000)$, and **(c)** $(13,000; 7,000)$.