Topological Regularization via Persistence-Sensitive Optimization

Arnur Nigmetov* **Aditi S. Krishnapriyan*** **Nicole Sanderson** **Dmitriy Morozov**
Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720

Abstract

Optimization, a key tool in machine learning and statistics, relies on regularization to reduce overfitting. Traditional regularization methods control a norm of the solution to ensure its smoothness. Recently, topological methods have emerged as a way to provide a more precise and expressive control over the solution, relying on persistent homology to quantify and reduce its roughness. All such existing techniques back-propagate gradients through the persistence diagram, which is a summary of the topological features of a function. Their downside is that they provide information only at the critical points of the function. We propose a method that instead builds on persistence-sensitive simplification and translates the required changes to the persistence diagram into changes on large subsets of the domain, including both critical and regular points. This approach enables a faster and more precise topological regularization, the benefits of which we illustrate with experimental evidence.

1 Introduction

Regularization is key to many practical optimization techniques. It allows the user to add a prior about the expected solution — e.g., that it needs to be smooth or sparse — and optimize it together with the main objective function. Classical regularization techniques [1], such as ℓ_1- and ℓ_2-norm regularization, have been studied in statistics and signal processing since at least the 1970s. These techniques are especially important in machine learning, where problems are often ill-posed and regularization helps prevent overfitting. Accordingly, various regularization techniques are not only used in machine learning research [2, 3], but are also incorporated into the standard optimization software and routinely used in applications.

Recently, several authors have begun to explore the use of topological methods to regularize the objective function. All of them use persistent homology to measure either the shape of the data set or the topological complexity of the learned function. For instance, Chen et al. [4] use persistence to describe the complexity of the decision boundary in a classifier and add terms to the loss to keep this boundary topologically simple. Brüel-Gabrielsson et al. [5] use persistence as a descriptor of the topology of the data and introduce a family of losses to control the shape of the data once it passes through a neural network.

All the methods that incorporate persistence into the loss function [4, 5, 6] rely on the same observation. Persistent homology describes data via a diagram, a collection of points $\{b_i, d_i\}$ in the plane, that encodes the topological features of the data: components of the decision boundary, "wrinkles" in the learned function, cycles in the point set once it passes through the neural network. Each point represents the birth b_i and death d_i of a topological feature. Each coordinate depends on the value of the function on a set of points. In the simplest case, $(b_i, d_i) = (f(x), f(y))$ for some x, y in the input, where f is the learned function. In the more sophisticated cases, each point in the persistence diagram is generated by a handful of input points (e.g., four [5]). Accordingly, if a loss $\mathcal{L}$ prescribes moving a point in the persistence diagram via a gradient $(\partial\mathcal{L}/\partial b_i, \partial\mathcal{L}/\partial d_i)$, one can back-propagate it to update the model parameters.

Although persistent homology describes a family of topological features of different dimensions (connected components, loops, voids), most practical examples have focused on 0-dimensional features (connected components generated by the extrema of the input function). In this case, a natural loss is one that penalizes and tries to remove low-persistence features, which are interpreted as noise: e.g., $\mathcal{L}(f) = \sum_{(d_i - b_i) \leq \varepsilon} (d_i - b_i)^2$. *Persistence-sensitive simplification* [7, 8, 9] offers a direct solution to this problem. It prescribes how to

*Equal contribution.

modify a given input function f to find a function g that is ε-close to f, but without the noisy features. Given such a g, which by construction minimizes the diagram loss $\mathcal{L}$ above, one can use $\|f-g\|^2$ as a term in the loss. In the context of learning, this approach offers a major advantage: instead of supplying gradients only on the critical points of f, we also get gradients on the regular points of f whose values must be changed to topologically simplify the function; see Figure 1.

Our contributions are:

- a method to control the topological complexity of a function, represented by a neural network, by incorporating persistence-sensitive simplification into the training;
- comparison of the training results after back-propagating gradients through the diagram vs. using persistence-sensitive optimization;
- experiments with data that illustrate the utility of controlling the topology of the learned function.

We note that topological methods have found a much broader use in machine learning than regularization. An important line of work involves developing techniques to incorporate topological features detected in data into machine learning algorithms [10, 11, 12]. Although there is some overlap in methods between the two research directions (notably propagating loss through the persistence diagram), our work is focused on regularization.

2 Background

We recall the relevant background in topological data analysis [13], focusing specifically on 0-dimensional persistent homology, which we introduce using an auxiliary computational construction, merge trees.

Merge trees. Let $f\colon X \to \mathbb{R}$ be a function on a topological space X. A *merge tree* tracks evolution of connected components in the sub-level sets $f^{-1}(-\infty, a]$ of the function, as we vary the threshold a. Formally, we identify two points x, y of X, if $f(x) = f(y) = a$ and x and y belong to the same connected component of the sub-level set $f^{-1}(-\infty, a]$. The quotient of X by this equivalence relation is called a merge tree of f.

Throughout the paper we use graphs to approximate continuous spaces, so we briefly dissect the above definition for functions on graphs. Let $f\colon G \to \mathbb{R}$ be a function on a graph $G = (V, E)$, defined on the vertices and linearly interpolated on the edges. For simplicity, we assume that all the values of f on the vertices are distinct and index the vertices $V = \{v_i\}$ so that $f(v_i) < f(v_{i+1})$. The *merge tree* of f is a graph $T = (V, E_T)$ such that an edge (v_i, v_j) for $i < j$ is present in T if and only if v_i and v_j belong to the same connected component $\mathcal{C}$ of $f^{-1}(-\infty, f(v_j)]$ and there does not exist k such that $i < k < j$ and $v_k \in \mathcal{C}$. A merge tree T is not necessarily a tree — it is a forest, with a tree for every connected component of G — but the distinction is minor for this paper.

T is naturally decomposed into *branches*; see Figure 1. A branch $B \subseteq V$ tracks a component of the sub-level set of f that first appears at a local minimum $v_b \in B$. This component disappears by merging into another branch B' that appeared at a lower local minimum v'_b. B merges into B' at a saddle $v_d \in B'$. We say that B is born at $f(v_b)$ and it dies at $f(v_d)$. The branch of the tree, born at the global minimum, that never merges into a deeper branch dies at ∞, by definition. The *persistence* $\mathrm{pers}(B)$ of a branch B is defined as the absolute value of the difference between its death and birth values.

Persistence. A 0-dimensional persistence diagram, denoted $\mathrm{Dgm}(f)$, is another summary of the connectivity of the sub-level sets of f. It is a multiset of points in the (extended) plane: a branch B, born at $f(v_b)$ that dies at $f(v_d)$ is summarized by the point $(f(v_b), f(v_d))$. Points closer to the diagonal represent shorter branches and we interpret them as noise.

Although we have defined everything in terms of the sub-level sets, the definition for super-level sets, $f^{-1}[a, \infty)$ is symmetric, with maxima replacing the minima. We use both constructions throughout the paper.

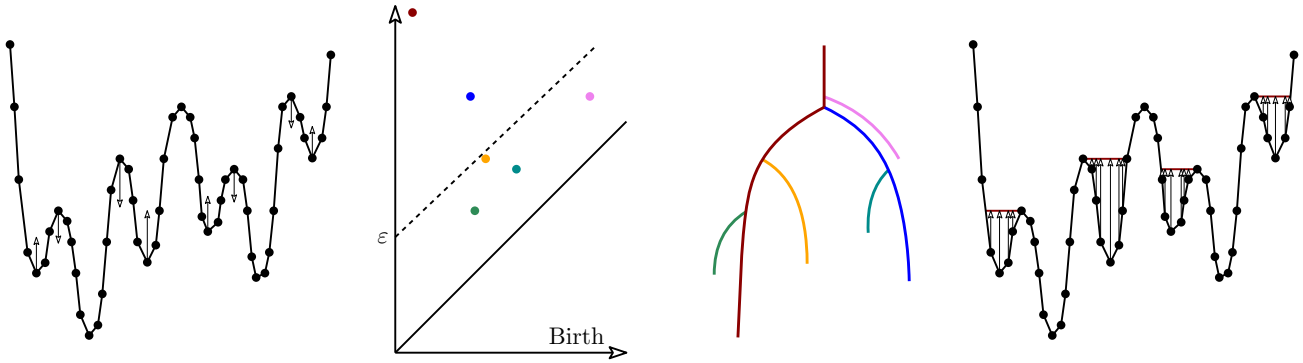
If graph G has n vertices and m edges, then a merge tree on G can be computed in $O(n \log n + m\alpha(m))$, where α is the inverse Ackermann function. It follows that a 0-dimensional persistence diagram can be computed in the same time.

To visualize the topological changes in the model during optimization, we stack persistence diagrams next to each other. The resulting *vineyard* of a family of functions f_i is a multiset of points $(i, |d^i_j - b^i_j|)$, where $\{(b^i_j, d^i_j)\}$ is the persistence diagram of f_i. In other words, over each i (for example, a training epoch) we plot all persistences of the corresponding diagram.

Simplification. An important property of persistence is stability: a small perturbation of function f causes a small perturbation of the persistence diagram $\mathrm{Dgm}(f)$. The formal statement is the celebrated Stability Theorem:

$$d_B(\mathrm{Dgm}(f), \mathrm{Dgm}(g)) \leq \|f - g\|_\infty,$$

where f and g are two real-valued functions on the same domain and d_B denotes the *bottleneck distance*. This theorem is one of the justifications for treating points close to the diagonal as topological noise.



(a) (b) (c) (d)

Figure 1: (a) Function on a graph, with gradients on critical points prescribed by the diagram loss. (b) Persistence diagram of this function. Points closer to the diagonal correspond to smaller fluctuations in the function, and we interpret them as topological noise. ε indicates the level of desired simplification that generates the gradients in (a) and (d). (c) Merge tree of the function, with branches highlighted in different color. The branches translate into the points in the persistence diagram of the matching color. (d) Gradients prescribed by the persistence-sensitive optimization (PSO loss). The gradients are present both on critical and regular points.

This view suggests getting rid of the topological noise. Let $f\colon G \to \mathbb{R}$ be a function on a graph G. A function $g\colon G \to \mathbb{R}$ is called its *ε-simplification*, if $\|f-g\|_\infty \leq \varepsilon$ and $\mathrm{Dgm}(g) = \{(b,d) \in \mathrm{Dgm}(f) \mid |d-b| > \varepsilon\}$. In other words, g is ε-close to f but its persistence diagram has only those points whose persistence exceeds ε. In the case of 0-dimensional persistence, ε-simplification always exists and can be computed in the same time as a merge tree [7, 8, 9].

3 Method

We start with the standard supervised learning problem. Given training data x_i with labels y_i, we want to learn a model f_θ, with parameters θ, that approximates y_i given x_i. Although this framework applies more generally, throughout the paper we focus on the case where f_θ is a neural network.

Suppose we are solving a regression problem. In this case, the input labels are scalars, $y_i \in \mathbb{R}$, and our network maps from some (typically) Euclidean space into reals, $f_\theta\colon \mathbb{R}^d \to \mathbb{R}$. The learning process is usually a form of gradient descent on the network parameters with respect to a user-chosen loss, for example, the mean-squared error (MSE), $\mathcal{L}(\theta) = \sum (f_\theta(x_i) - y_i)^2/n$.

Ideally, we would like to topologically simplify the model f_θ either on its entire domain, or at least on the "data manifold," the subset of the domain that contains all possible data. Unfortunately, there are no algorithms to solve this problem — topological methods require a combinatorial representation of the domain — so we resort to a standard approximation.

We take the domain of the network f_θ to be the k-nearest neighbors graph on the training set $\hat{X}$: each training sample is a vertex, and two vertices are connected if and only if one of them is among the k-nearest neighbors of the other one. The k-NN graph G approximates the data manifold. We can increase the quality of this approximation by sampling additional points in the neighborhood of our input. In the experiments in Section 6, we draw n additional points from a normal distribution, centered on each training data point, $x \in \hat{X}$, which results in a graph with $(n+1)\cdot|\hat{X}|$ vertices. (Although we don't know the true label on the extra points, we don't need it for the topological simplification.) Both because computing a k-NN graph is expensive for high-dimensional data and because it helps to control noise, in some experiments we build the k-NN graph on the lower-dimensional projection of $\hat{X}$ using PCA.

We use merge trees to compute an ε-simplification g of our model f_θ. For every vertex v, we find its first ancestor u that lies on a branch with persistence at least ε. (If v is already on such a branch, then $u = v$.) We set $g(v) = f_\theta(u)$. The effect of this operation on the merge tree is that all the branches with persistence less than ε are removed; see Figure 1.

Applying simplification. Given an ε-simplification g of f_θ, we could add a term $\lambda\cdot\|f_\theta - g\|^2$ to the loss and use a single optimizer. Instead, we opted for a different approach by alternating between the standard training and the topological phases, with a separate optimizer for each phase. A key advantage of this separation is that it keeps two histories of the gradients, one for each phase, so that the topological loss does not influence the momentum in the standard training.

An important decision is when to switch to the topological phase. We use a heuristic that depends on the validation loss. In each epoch, we first iterate over all batches and perform standard training using the first optimizer. Then, if the validation loss increases, compared to the previous epoch, by more than some threshold (a hyperparameter), we compute the ε-simplification g and take 5 to 10 steps with the second optimizer to minimize $\|f_\theta - g\|^2$. We use the norms of the gradients of the ordinary training loss and of the topological loss, to set a learning rate for the latter that ensures that we update the model parameters θ by comparable amounts in both phases.

Choice of ε. A key decision in implementing our method is how to choose ε, to decide which points to keep and which to remove in the persistence diagram. Earlier works [4, 5] prescribe a fixed number of points to keep in a certain region of the persistence diagram. For instance, some of the losses in [5] penalize all but j of the most persistent points. We can optimize such a loss by setting $\varepsilon = (p_j + p_{j+1})/2$, where p_i is the persistence of each point, sorted in descending order.

Another alternative, used in topological data analysis to automatically distinguish between persistent and noisy points, is the *largest-gap heuristic*. To apply it, we find index j such that the difference $p_j - p_{j+1}$ is maximized.

Finally, the heuristic that we found most effective and use for all experiments in Section 6 is to use validation loss as our ε. Validation loss tells us how far we are from a function that gives perfect answers on the validation set. Using it as ε, we find the topologically simplest function g that is within the same distance from our model f_θ.

Classification. For regression, the network itself serves as a real-valued function amenable to topological analysis. Classification requires a little more work. We assume that the data has m classes and the network has m output channels, $f_\theta \colon \mathbb{R}^d \to \mathbb{R}^m$, with the predicted class chosen as $p = \arg\max_i f_\theta(x)[i]$. We define the *confidence function*, $\phi : \mathbb{R}^d \to \mathbb{R}$, to measure how much higher the value in the predicted channel is compared to the second highest candidate:

$$\phi(x) = f_\theta(x)[p] - \max_{i \neq p} f_\theta(x)[i].$$

When $\phi(x)$ is close to 0, the network is not confident whether to classify x as the top class p or the second-best guess. The zero set $\phi^{-1}(0)$ is the decision boundary, by definition. Outliers of one class scattered among the points of another introduce spurious extrema in the confidence function. By driving optimization towards the simplified version of ϕ, we can reduce overfitting.

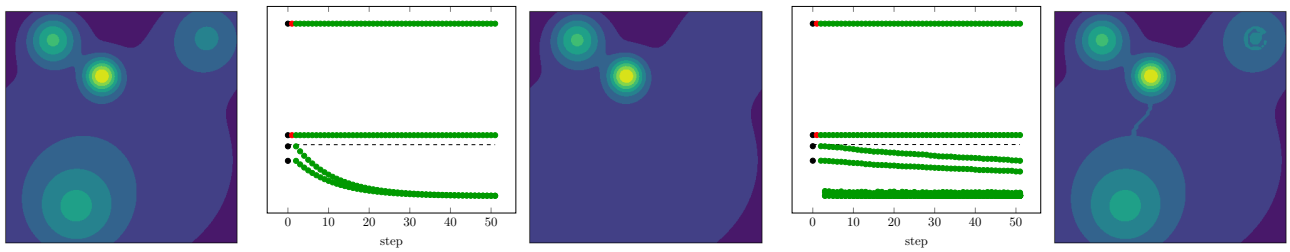
Because generically $\phi(x)$ is never zero on an input point $x \in \hat{X}$, we need an extra step to capture the topology of the decision boundary. If two vertices u and v, connected by an edge in the k-NN graph, are assigned two different classes by the network, then the decision boundary passes somewhere between them. In this case, we remove the edge (u, v) from the graph. This pruning results in multiple connected components, at least one per class. We compute the merge tree — forest in this case — of the confidence function on the pruned graph, with respect to the super-level sets, i.e., tracking persistence of the maxima. Because confidence function is never negative, we restrict the infinite branches in the merge tree to die at 0. This obviates special treatment of separate connected components in the graph: if one of them produces a low-persistence merge tree, we simplify it by setting the values of all of its vertices to 0.

4 Comparison with Diagram Simplification

Earlier work on applying topological regularization to neural networks [4, 5] relied on backpropagation through persistence diagrams. For piecewise-linear functions on a graph, each point in the 0-dimensional persistence diagram corresponds to a pair of vertices, $(b_i, d_i) = (f(x), f(y))$. If one adds a regularization term of the form $\sum(d_i - b_i)^2$, where the sum is taken over all points (b_i, d_i) with persistence less than ε, then one can back-propagate the gradient to the function values and then to the model parameters, i.e., the weights of the network. We call this loss the *diagram loss*, and the loss proposed in the previous section, the *PSO loss*.

The first disadvantage of the diagram loss is that only critical points generate pairs in the persistence diagram. Accordingly, most input points are not used and receive no information during the backpropagation. To illustrate this, we take $f \colon \mathbb{R}^2 \to \mathbb{R}$ to be the sum of 4 Gaussians and evaluate f on the uniform grid over unit square $[0, 1] \times [0, 1]$ with $10,000$ vertices. Figure 2a illustrates the plot of f. We pick ε so that the two lower persistence points in the diagram of f (corresponding to the two Gaussians with lower peaks) are simplified, and take 50 steps of gradient descent using the PSO loss and the diagram loss directly on values of f at each vertex. The simplified functions appear in Figures 2c and 2e, respectively.

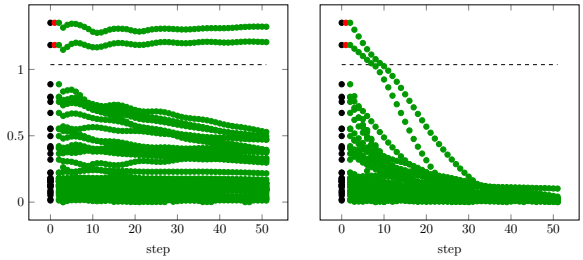
Figures 2b and 2d show the vineyards of the two optimization processes. In both vineyards, we show the original persistence values in black, the desired values in red, and the values at each step of the optimization in green. With PSO loss, this is an unconstrained convex problem, so the optimizer quickly eliminates



(a) (b) (c) (d) (e)

Figure 2: Optimization of the values. (a) Original function. (b) Vineyard of simplification with PSO loss. (c) Function simplified with PSO loss. (d) Vineyard of simplification with diagram loss. (e) Function simplified with diagram loss.

the two noisy bumps of the function, while preserving its persistent part. In contrast, each step of the diagram loss changes values only at critical points, making the optimization process much slower — after 50 steps both bumps are still present. It also requires recomputing persistence diagram after each step. This not only makes the process slower, but also introduces additional topological noise, evident in the vineyard.



(a) (b)

Figure 3: Optimization of the weights. (a) Vineyard of simplification with PSO loss. (b) Vineyard of simplification with diagram loss.

Figure 3 shows the effect of the two losses on a neural network. We train a fully connected network with 5 layers for 100 epochs and then perform 30 steps of topological optimization. The key difference from the previous example is that we do not have direct control over function values, but only over the weights of the network. The diagram loss provides information only for the critical points of the function, and the optimizer ends up minimizing this loss by pushing the whole function towards a constant: in the vineyard on the right-hand side, all points, not just the points below ε, are moving to 0. Since the PSO loss penalizes changes to the high-persistence parts of the function, its optimization does not suffer from the same problem, as the vineyard on the left-hand side shows.

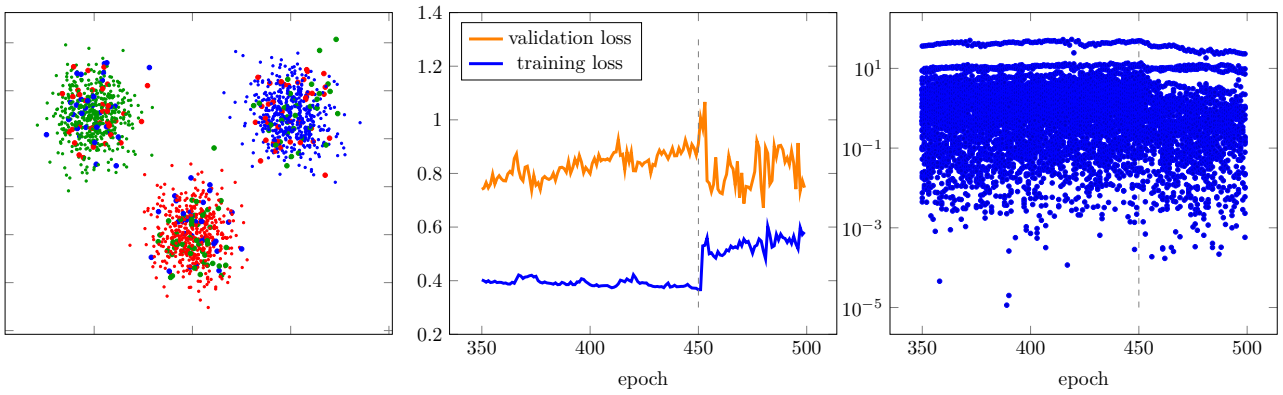
It is not clear how to fix this overzealousness of the diagram loss. The main difficulty is that the critical vertices and their pairing change after each gradient descent step. A naive fix would be to add a term that pushes high-persistence points to ∞: $-\lambda \sum_{(d_i - b_i) > \varepsilon} (b_i - d_i)^2$. We have tried this approach, but it did not perform well. Depending on weight λ, either the additional term had no influence at all, and the function was squashed to a constant; or it dominated, and the function exploded numerically.

A more principled solution would be to compute a matching between the persistence diagram after each step of the topological optimization and the target simplified diagram. The matching would translate into a loss that would simplify the diagram, while trying to preserve the high-persistence points. However, this approach has many drawbacks. The computation of the matching, even using the fast algorithms [14], is prohibitively expensive and would make this procedure completely impractical. The method itself, by construction, would only preserve the structure of the persistence diagram, not its values at individual vertices. Finally, changing the diagram loss function at each step of the gradient descent may have unexpected effects on the momentum.

5 Illustrative Example

To illustrate how topological regularization using the PSO loss can reduce overfitting, we consider a simple three-class dataset, shown in Figure 4a. It consists of points sampled from three Gaussians, 1,000 points from each, that represent three distinct classes. We randomly shuffle 20% of the labels to introduce class noise. We train a fully-connected feedforward neural network with 5 hidden layers of 100 nodes each for 500 epochs.

Figure 4b illustrates the training and validation losses, and Figure 4c shows the persistence vineyard of the confidence function for epochs 350 to 500. In the beginning of this range, the network has already overfit the labels. The growing validation loss confirms the overfitting, which is also evident in the vineyard, where the second and third highest persistence points, which represent the true classes in the data, are becoming



(a) (b) (c)

Figure 4: (a) Input data: 1,000 points sampled from each of the three Gaussians, representing three distinct classes, with 20% of the labels randomly shuffled. (b) Training and validation loss during the training of a neural network, restricted to the later epochs, where the network overfits the data. Simplification is applied after every epoch, following epoch 450, marked with a dashed line. (c) Vineyard of the confidence function during training; the start of the simplification is marked with a dashed line. The three persistent points, representing the three classes in the data, become prominent after the simplification.

indistinguishable from the noisy points.

Starting with epoch 450, we apply ten steps of topological simplification after every training epoch. Because we expect each of the three classes to be a single cluster, we set ε to keep the three highest points in the persistence diagram. This defines a PSO loss that encourages removing maxima of the confidence function that do not correspond to the 3 predominant class clusters.

As Figure 4b illustrates, after turning on simplification at epoch 450, the validation loss decreases by over 20%. Figure 4c demonstrates the abundance of high persistence features prior to epoch 450. Most of these correspond to mountains in the confidence function around noisy mislabeled points. Turning on simplification at epoch 450 reduces the persistence of these peaks which drives the network to match the class labels of the dominant class around the outliers.

This toy example demonstrates how PSO simplification identifies regions of overfitting due to class noise and reduces the confidence function near these noisy labeled points, lowering the validation loss and increasing the accuracy of the model after overfitting has occured.

6 Experiments

We study the performance of persistence-sensitive optimization on six regression problems and seven classification problems from the UCI repository [15]. To represent a variety of problem settings, the selected datasets vary in the number of features, sample size, and number of classes. We standardize the features by subtracting the mean and dividing by the standard deviation. For both regression and classification, we use a dense neural network with five hidden layers and 100 hidden nodes per layer. We use the Adam optimizer and a learning rate of 0.001 across all experiments, including regular training and training with topological simplification.

We compare performance of the networks trained (1) without regularization, (2) with ℓ_2 regularization, (3) with topological regularization. For all experiments, training with and without regularization were run for the same number of total epochs. For the ℓ_2 regularization, the square of the weights of the network is added to the loss, scaled by a factor of λ, which we choose by sweeping through a logarithmically spaced grid from $[10^{-5}, 10^{1}]$. We report the best performance across all λs for each dataset. For each dataset, we run all the models at least five times with different preset random seeds and average over all the trials.

As described in Section 3, we set a number of hyperparameters during the topological simplification:

- topological simplification is applied when validation loss increases by more than t;
- k determines the number of neighbors in the k-NN graph used to approximate the domain of the function;
- n is the number of additional points we sample, for each input point, before building the k-NN graph;
- the points are drawn from a Gaussian with variance σ, ranging from 0.001 to 0.2.

Supplementary materials list extra details for the data sets, including what hyperparameter ranges were swept

	Regularization				Hyperparameters			
Datasets	None	ℓ_2	PSO	Δ	k	t	n	σ
Wine	0.78	0.77	**0.76**	2.6%	15	0.001	6	0.001
Iran housing	0.12	0.11	**0.10**	16.7%	10	0.01	9	0.001
Boston	0.33	0.32	**0.31**	6.1%	20	0.001	9	0.001
Concrete	0.31	0.30	**0.29**	6.4%	15	0.01	3	0.001
CT slices	0.031	0.031	**0.029**	6.4%	60	0.0001	1	0.001
Protein	0.64	0.63	**0.62**	3.1%	20	0.001	6	0.001

Table 1: RMSD results on regression datasets comparing no regularization, ℓ_2 regularization of the weights, and topological simplification, averaged over multiple trials. The best model for each dataset is in bold. As topological simplification always results in performance improvement, the percentage of improvement (decrease in RMSD), from None to PSO, is also shown (Δ). The last four columns show the hyperparameters for the best model.

during the experiments. We always set ε to the validation loss.

Regression. We evaluate the performance of topological regularization on six regression datasets. They vary in size from hundreds (Iran housing, Boston) to thousands (Wine, Concrete), to tens of thousands (CT slices, Protein) data points. For the largest dataset, CT slices, we project the data onto the first ten principle components before computing the k-NN graph. We use a 56%-19%-25% training-validation-test split, i.e., first applying a 75%-25% training-test split, and then further splitting the training set 75%-25% into a validation set. We evaluate the quality of the prediction using the root-mean-square-deviation, $\sqrt{\sum(\hat{y}_i - y_i)^2/n}$.

Table 1 presents the results of our regression experiments. Overall, topological simplification reduces RMSD across all the datasets by an average of 6.9%. Sampling each point multiple times with a small amount of perturbation improves performance. By applying simplification when validation loss increases by more than threshold t, we reduce overfitting and the resulting error. We also see that across the λ hyperparameter swept for ℓ_2 regularization, the performance is always worse than with topological simplification. We note that our method is fast enough to be used on very large datasets (we give two examples with 40,000+ points, but that's by no means the limit); previous approaches to topological regularization (using a form of diagram loss) [4] were limited to much smaller datasets (hundreds to a thousand points).

Classification. We also evaluate our method on seven classification datasets. Each one has from two to 26 classes. Similar to the regression datasets, each has hundreds (Wisconsin cancer, Vertebral, SPECT) to thousands (Wine, Semeion, Wireless) to tens of thousands (Letter recognition) data points. We use the same 56%-19%-25% training-validation-test split. When topological simplification is applied, we set ε to the cross-entropy loss and simplify the confidence function ϕ, described in Section 3. We evaluate the quality of our predictions by computing the cross-entropy (X-E) loss and accuracy.

Table 2 shows the results of our classification experiments. The X-E loss decreases when we apply topological regularization except for the Wisconsin cancer dataset, while accuracy increases for all the datasets, except the SPECT dataset (the smallest dataset in size). Overall, X-E loss decreases by an average of 14.8% and accuracy increases by an average of 2.9% across all datasets. The table shows the hyperparameters for the model with the lowest X-E loss. In contrast with regression, on average, more aggressive perturbation of the sampled points results in better performance. The best model performance across all the datasets, except letter recognition, occurs for validation loss threshold t equal to 0.0001, indicating that applying simplification as soon as validation loss increases, i.e., as soon as the model shows any sign of overfitting, helps regularize the training. Topological simplification is slightly less accurate than ℓ_2 regularization on the SPECT dataset, equally accurate on the vertebral dataset, the same in terms of X-E loss on the wireless dataset, and better on all other datasets.

Loss and vineyard. To better understand topological simplification, we examine the training and validation loss curves as well as the vineyards for regression experiments on the Wine dataset. As Figure 5 illustrates, the network quickly starts to overfit — without simplification, within 10–15 epochs — and the validation loss rises. Applying simplification quickly reduces the validation loss, seemingly pushing the system into another region of the loss landscape. This is further seen in the vineyard, where the sharp decrease in validation loss matches with the simplification of the persistence diagram.

	Crossentropy				Accuracy				Hyperparameters			
	Regularization				Regularization							
Datasets	None	ℓ_2	PSO	Δ	None	ℓ_2	PSO	Δ	k	t	n	σ
Wisconsin cancer	0.13	**0.08**	0.09	30.8%	0.97	0.98	**0.99**	2.1%	15	0.0001	3	0.2
Wine	0.97	0.96	**0.93**	4.1%	0.59	0.60	**0.65**	10.2%	15	0.0001	0	0.001
Semeion	0.48	0.48	**0.38**	20.8%	0.87	0.88	**0.90**	3.4%	20	0.0001	9	0.2
Vertebral	0.39	0.38	**0.34**	12.8%	0.82	**0.84**	**0.84**	2.4%	15	0.0001	9	0.1
Wireless	0.07	**0.06**	**0.06**	14.3%	0.97	0.98	**0.98**	1.1%	25	0.0001	6	0.2
SPECT	0.35	0.34	**0.33**	5.7%	0.80	**0.83**	0.80	0%	10	0.0001	15	0.01
Letter recognition	0.27	0.26	**0.23**	14.8%	0.92	0.92	**0.93**	1.1%	20	0.01	3	0.2

Table 2: Cross-entropy loss and accuracy results on classification datasets comparing no regularization, ℓ_2 regularization of the weights, and topological simplification, averaged over multiple trials. The best model is in bold. As the improvement from topological simplification is always greater than or equal to training the model without regularization, the percentage of improvement (decrease in the case of X-E loss and increase in the case of accuracy), from None to PSO, is also shown (Δ). The last four columns show the hyperparameters for the best model, with the lowest X-E loss.

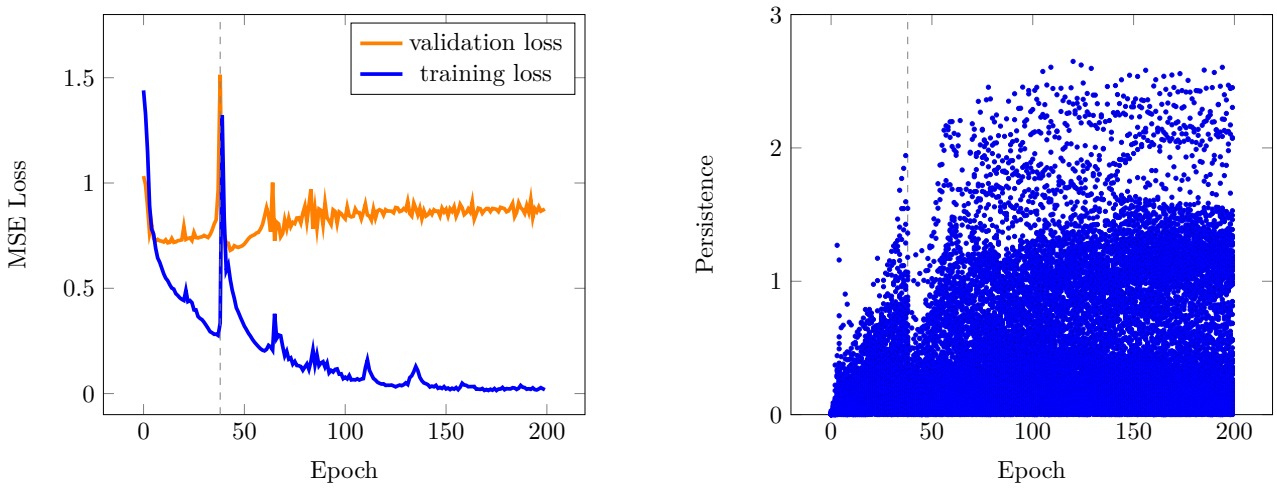


Figure 5: (a) Training and validation loss curves for an experiment on the wine regression dataset. Performance is best at epoch 44, and simplification is applied only once, after epoch 43. (b) Vineyard over all epochs.

7 Conclusion

We presented a topological regularization method that uses persistent homology, merge trees, and persistence-sensitive simplification to minimize the number of noisy extrema in a machine learning model. Unlike previous such methods, our approach is faster — requiring to compute the topological descriptor only once per simplification phase — as well as more robust and predictable in its effects on the model. The key distinction of the method is its ability to prescribe gradients on the entire domain, approximated as a k-NN graph, rather than only on the critical points. We illustrated the benefits of its use in experiments with a number of well-known data sets.

Our work has a larger implication for the use of topological methods in machine learning. The realization that one can back-propagate gradients through a persistence diagram has generated considerable interest in the community, with a number of recent works [4, 5, 6, 10, 11] exploring this idea. Our results suggest that it may be better to not treat persistence as a black box. Rather, it is a rich language that allows one to precisely express topological constraints and priors to add to a problem. The actual enforcement of these constraints can be accomplished via different methods, back-propagation through the persistence diagram being but one of them.

Building on prior work in computational topology, we describe only how to simplify extrema, i.e., 0-dimensional persistence diagrams. A key research direction is how to adapt these ideas to higher dimensional persistent homology. It is undoubtedly useful to incorporate higher-dimensional topological constraints, such as loops or voids in the data, into optimization. Doing so efficiently may require imposing constraints not only on the points in the persistence diagrams, but on the

entire representative cycles implied by those points.

8 Acknowledgements

This work was supported by Laboratory Directed Research and Development (LDRD) funding from Berkeley Lab, provided by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. A.S.K. was supported by the Alvarez Fellowship in the Computational Research Division at LBNL.

References

[1] Stephen P Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[2] Bernhard Schölkopf, Alexander J Smola, Francis Bach, and Managing Director of the Max Planck Institute for Biological Cybernetics in Tubingen Germany Profe Bernhard Scholkopf. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2002.

[3] Andrew Y Ng. Feature selection, L_1 vs. L_2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78, 2004.

[4] Chao Chen, Xiuyan Ni, Qinxun Bai, and Yusu Wang. A topological regularizer for classifiers via persistent homology. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 2573–2582, 2019.

[5] Rickard Brüel-Gabrielsson, Bradley J Nelson, Anjan Dwaraknath, Primoz Skraba, Leonidas J Guibas, and Gunnar Carlsson. A topology layer for machine learning. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 1553–1563, 2020.

[6] Jacob Leygonie, Steve Oudot, and Ulrike Tillmann. A framework for differential calculus on persistence barcodes. October 2019.

[7] Herbert Edelsbrunner, Dmitriy Morozov, and Valerio Pascucci. Persistence-sensitive simplification functions on 2-manifolds. In *Proceedings of the Annual Symposium on Computational Geometry*, pages 127–134. ACM, 2006.

[8] Dominique Attali, Marc Glisse, Samuel Hornus, Francis Lazarus, and Dmitriy Morozov. Persistence-sensitive simplification of functions on surfaces in linear time, 2009. Manuscript. Presented at TopoInVis'09.

[9] Ulrich Bauer, Carsten Lange, and Max Wardetzky. Optimal topological simplification of discrete functions on surfaces. *Discrete & computational geometry*, 47(2):347–377, 2012.

[10] Christoph D Hofer, Roland Kwitt, and Marc Niethammer. Learning representations of persistence barcodes. *Journal of machine learning research: JMLR*, 20(126):1–45, 2019.

[11] Mathieu Carriere, Frederic Chazal, Yuichi Ike, Theo Lacombe, Martin Royer, and Yuhei Umeda. PersLay: A neural network layer for persistence diagrams and new graph topological signatures. *Stat*, 1050:17, 2019.

[12] Henry Adams, Tegan Emerson, Michael Kirby, Rachel Neville, Chris Peterson, Patrick Shipman, Sofya Chepushtanova, Eric Hanson, Francis Motta, and Lori Ziegelmeier. Persistence images: A stable vector representation of persistent homology. *Journal of machine learning research: JMLR*, 18(1):218–252, 2017.

[13] Herbert Edelsbrunner and John Harer. *Computational topology: an introduction*. American Mathematical Soc., 2010.

[14] Michael Kerber, Dmitriy Morozov, and Arnur Nigmetov. Geometry helps to compare persistence diagrams. *J. Exp. Algorithmics*, 22:1.4:1–1.4:20, September 2017.

[15] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

Topological Regularization via Persistence-Sensitive Optimization: Supplementary Materials

1 Hyperparameter ranges for experiments

The computation relies on the following hyperparameters:

- topological simplification is applied when validation loss increases by more than t;
- k determines the number of neighbors in the k-NN graph used to approximate the domain of the function;
- n is the number of additional points we sample, for each input point, before building the k-NN graph;
- the points are drawn from a Gaussian with variance σ.

The tables list the values of the hyperparameters we tried for each dataset. In the main text, we report the model that has the best performance, highlighted in bold here.

1.1 Regression

Datasets	k	t	n	σ
Wine	10, **15**, 20	**0.001**, 0.01, 0.05, 0.1, 0.5	0, 3, **6**, 9,12	**0.001**, 0.01, 0.1. 0.2
Iran housing	**10**, 15, 20	0.001, **0.01**, 0.05, 0.1	0, 3, 6, **9**, 12	**0.001**, 0.01, 0.1, 0.2
Boston	10, 15, **20**	**0.001**, 0.01, 0.05, 0.1	0, 3, 6, **9**, 12	**0.001**, 0.01, 0.1, 0.2
Concrete	10, **15**, 20	0.001, **0.01**, 0.05, 0.1, 0.5	0, **3**, 6, 9, 12	**0.001**, 0.01, 0.1, 0.2
CT slices	20, 40, **60**, 80	**0.0001**, 0.001	0, **1**	**0.001**, 0.01, 0.1, 0.2
Protein	**20**, 40, 60, 80	**0.001**, 0.01, 0.1	0, 3, **6**	**0.001**, 0.01, 0.1, 0.2

Table 1: Hyperparameter ranges for regression datasets.

1.2 Classification

Datasets	k	t	n	σ
Wisconsin cancer	10, **15**, 20	**0.0001**, 0.001, 0.01, 0.1	0, **3**, 6, 9,12	0.001, 0.01, 0.1. **0.2**
Wine	10, **15**, 20	**0.0001**, 0.001, 0.01, 0.1	**0**, 3, 6, 9, 12	**0.001**, 0.01, 0.1, 0.2
Semeion	10, 15, **20**	**0.0001**, 0.001, 0.01, 0.1	0, 3, 6, **9**, 12	0.001, 0.01, 0.1, **0.2**
Vertebral	10, **15**, 20	**0.0001**, 0.001, 0.01, 0.1	0, 3, 6, **9**, 12	0.001, 0.01, **0.1**, 0.2
Wireless	10, 15, 20, **25**	**0.0001**, 0.001, 0.01, 0.1	0, 3, **6**, 9, 12	0.001, 0.01, 0.1, **0.2**
SPECT	**10**, 15, 20	**0.0001**, 0.001, 0.01, 0.1	0, 3, 6, 9, 12, **15**	0.001, **0.01**, 0.1, 0.2
Letter recognition	10, **20**, 30, 40, 50	0.0001, 0.001, **0.01**	0, **3**, 6, 9, 12, 15	0.001, 0.01, 0.1, **0.2**

Table 2: Hyperparameter ranges for classification datasets.