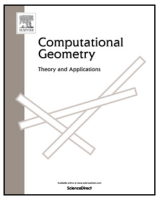
Contents lists available at ScienceDirect

Computational Geometry: Theory and Applications

journal homepage: www.elsevier.com/locate/comgeo



Topological regularization via persistence-sensitive optimization

Arnur Nigmetov[a,*,1], Aditi Krishnapriyan[a,b,1], Nicole Sanderson[a], Dmitriy Morozov[a]

[a] *Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, 94720, CA, USA*
[b] *University of California, Berkeley, 110 Sproul Hall, Berkeley, 94720, CA, USA*

A R T I C L E   I N F O

Article history:
Received 2 August 2021
Received in revised form 18 August 2022
Accepted 16 February 2024
Available online 28 February 2024

Keywords:
Topological simplification
Regularization
Merge tree

A B S T R A C T

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 [4], 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 [23,20], 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. Those that use persistent homology use it to measure either the shape of the data set or the topological complexity of the learned function. For instance, Chen et al. [8] 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

* Corresponding author.
E-mail addresses: anigmetov@lbl.gov (A. Nigmetov), aditik1@berkeley.edu (A. Krishnapriyan), nikki.f.sanderson@gmail.com (N. Sanderson), dmorozov@lbl.gov (D. Morozov).
[1] Equal contribution.

https://doi.org/10.1016/j.comgeo.2024.102086

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 [8,5,18] 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. For example, consider the Vietoris–Rips filtration of a point cloud. The birth and death values b_i and d_i are critical values of two simplices of dimension q and $q+1$. By definition, a critical value of a simplex is the length of its longest edge, so $b_i = \|x_{i,1} - x_{i,2}\|$ and $d_i = \|x_{i,3} - x_{i,4}\|$ for some four points $x_{i,k}$, $k = 1, 2, 3, 4$ of the point cloud. Thus the gradient will only back-propagate through the coordinates of these four points. Same is true for the *weak α-filtration* used in [5], since a critical value in [5] is also defined as the length of the longest edge. 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., $\mathcal{L}(f) = \sum_{(d_i - b_i) \le \varepsilon} (d_i - b_i)^2$. *Persistence-sensitive simplification* [11,2,3] offers a direct solution to this problem. It prescribes how to 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: by replacing a complicated non-convex problem with a smooth unconstrained convex problem, we get gradients not only on the critical points of f, but also on the regular points whose values must be changed to topologically simplify the function; see Fig. 1.

Our contributions are:

- a method to control the topological complexity of a function 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 that illustrate that our method is significantly faster than optimizing a loss by only back-propagating through the diagram;
- experiments with data that illustrate the utility of controlling the topology of the learned function.

Put together, these findings motivate a more extensive study of techniques for topological optimization beyond only those based on back-propagating through the diagram. The approach described in this paper works for only one loss formulation, but its underlying ideas are applicable more broadly.

We note that topological methods have found a much broader use in machine learning than regularization. The authors of [15] differentiate through persistence diagrams to learn filter functions on graphs. In [6], the authors study the convergence of stochastic subgradient descent of a functional based on persistence. An important line of work involves developing techniques to incorporate topological features detected in data into machine learning algorithms [14,7,1,17]. Although there is some overlap in methods with these research directions (notably propagating loss through the persistence diagram), our work is focused on regularization. Persistence has also been used more broadly to characterize the behavior of neural networks [13,22,12].

2. Background

We recall the relevant background in topological data analysis [10], focusing specifically on 0-dimensional persistent homology, which we introduce using an auxiliary computational construction, merge trees.

Merge trees Let $f : X \to \mathbb{R}$ 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 $f : G \to \mathbb{R}$ 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.

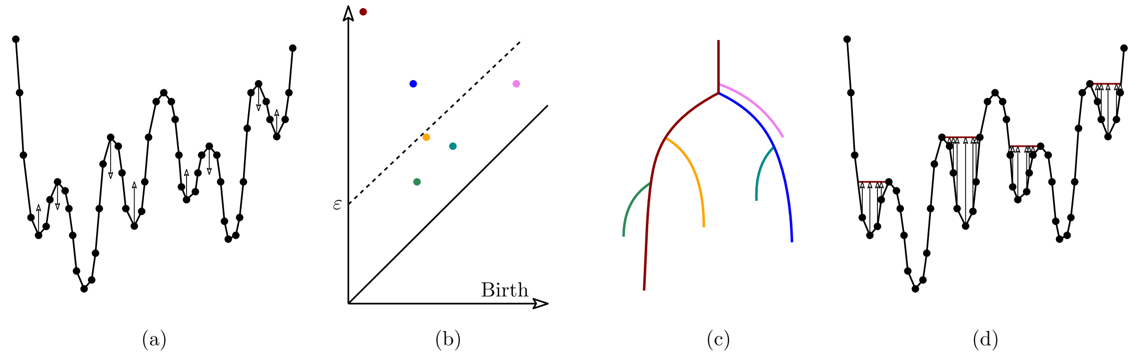


Fig. 1. (a) Function on a graph, with gradients on critical points prescribed by the diagram loss. (b) Persistence diagram of this function. Points closer to the diagonal correspond to smaller fluctuations in the function, and we interpret them as topological noise. ε indicates the level of desired simplification 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. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

T is naturally decomposed into *branches*; see Fig. 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 to obtain what is called a *vineyard*. Formally, the *vineyard* of a family of functions f_i is a multiset of points

$$\{(i, b^i_j, d^i_j) \mid (b^i_j, d^i_j) \text{ are points of the persistence diagram of } f_i\}.$$

Thus, vineyards are 3-dimensional objects with coordinates $(\mathit{time}, \mathit{birth}, \mathit{death})$. In order to plot them, we project a vineyard into $(\mathit{time}, \mathit{persistence})$ coordinates. In all our figures that show vineyards, we plot the multiset of points $(i, |d^i_j - b^i_j|)$. In other words, over each i (for example, a training epoch) we plot all persistences of the corresponding diagram. In this projection, the smaller the persistence of a point, the closer the point to the horizontal axis.

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:

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

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.

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 $\|f - g\|_\infty \le \varepsilon$ 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 [11,2,3].

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. Fig. 1 shows an example. We choose ε that preserves only the two longest branches: the branch corresponding to the global minimum (the purple branch on the left, with infinite persistence) and the blue branch on the right. Four shorter branches (light-green, yellow, dark-green, and pink) are removed, if we change the function values as indicated by the arrows.

Definition 3.1. Given ε, let g be an ε-simplification of f_θ. We view g as constant with respect to weights θ. The *PSO loss* is defined as

$$\mathcal{L}_{PSO}(f) = \frac{1}{2}\|f_\theta - g\|^2 = \frac{1}{2}\sum_{v \in G} |f_\theta(v) - g(v)|^2.$$

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. This technique is similar to the alternating minimization algorithm studied by [24] for separable convex problems.

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 [8,5] 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, $f_\theta : \mathbb{R}^d \to \mathbb{R}^m$, 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. 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

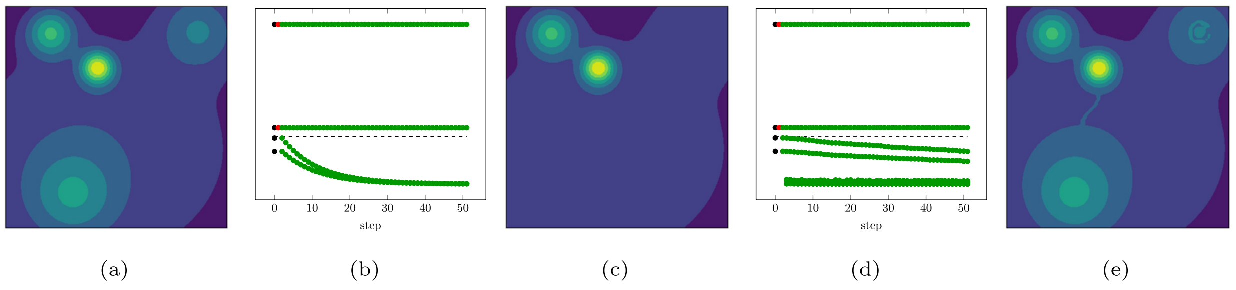


Fig. 2. Optimization of the values. (a) Original function. (b) Vineyard of simplification with PSO loss. (c) Function simplified with PSO loss. (d) Vineyard of simplification with diagram loss. (e) Function simplified with diagram loss.

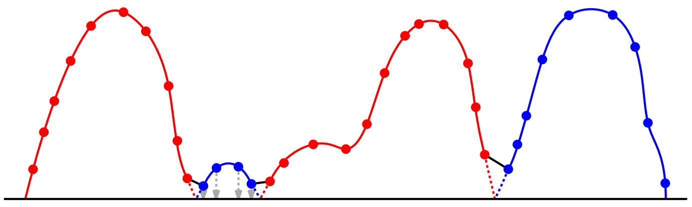


Fig. 3. Confidence function on a real line. There are two classes, encoded in red and blue colors. The edges in black that correspond to the dashed parts of the continuous function are removed. After that, in this case there is only one small 'bump' that we want to remove during simplification. The desired effect of the PSO loss shown with arrows.

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 the 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.

A schematic example is shown in Fig. 3. We assume that $k = 2$, so each point is connected to its two neighbors and the underlying graph is a line. There are 4 points of the blue class that correspond to a region of low confidence. If we remove the edges in black, then these 4 points are isolated and the merge tree has a connected component capturing these points. We want to make the network even less confident in its predictions in this region, and simplifying the confidence to 0 expresses exactly that.

4. Comparison with diagram simplification

Earlier work on applying topological regularization to neural networks [8,5] relied on backpropagation through persistence diagrams. Let us re-state this approach in our notation. We build k-NN graph G and view the model f_θ as a piecewise-linear function $f_\theta : G \to \mathbb{R}$, i.e., we evaluate f_θ at each vertex x of G and extend it to the edges by linearity. Since we have a function on a 1-dimensional simplicial complex, we can compute its 0-dimensional persistence diagram using a lower-star filtration (sublevelset persistent homology of f_θ). Each point (b_i, d_i) of the diagram records a component that is born at local minimum x_i with $f_\theta(x_i) = b_i$ and dies at saddle y_i with $f_\theta(y_i) = d_i$.

Definition 4.1. Given ε, the *diagram loss* is defined as

$$\mathcal{L}_{Dgm}(f) = \frac{1}{2} \sum_{|d_i - b_i| < \varepsilon} (d_i - b_i)^2 = \frac{1}{2} \sum_{|d_i - b_i| < \varepsilon} (f_\theta(y_i) - f_\theta(x_i))^2,$$

where we sum over all points (b_i, d_i) of the diagram with persistence less than ε.

If one adds a regularization term of the form $\lambda \mathcal{L}_{Dgm}(f)$, then one can back-propagate the gradient to the function values $f_\theta(x)$, $f_\theta(y)$ and then to the model parameters θ, i.e., the weights of the network.

Let us compare the (sub)gradients of the diagram loss and the PSO loss. For simplicity, we assume general position (all function values are distinct), so that both losses are differentiable, and we can talk about gradients. In terminology of [19], we assume that we are in the highest-dimensional stratum. First of all, $\frac{\partial \mathcal{L}_{Dgm}}{\partial x} = 0$ for every non-critical $x \in G$, while for the PSO loss the gradient is non-zero for all x that belong to branches with persistence less than ε. Furthermore, even for critical points, the components of the gradients are different for two reasons. First, the diagram loss pushes the point (b_i, d_i) to its diagonal projection $((b_i + d_i)/2, (b_i + d_i)/2)$. The birth point should move up, and the death point should move

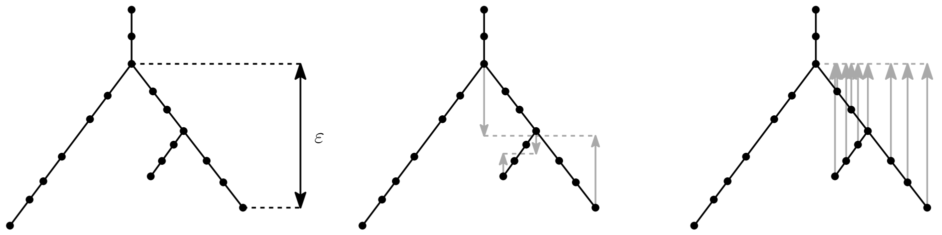


Fig. 4. Comparison of the gradients of the diagram and PSO losses. *Left*: A merge tree of a function with 3 minima. The merge tree has 3 branches, and ε is such that the two shorter branches should be removed. *Middle*: Gradient of the diagram loss pushes critical points towards the midpoint of their branch. *Right*: Gradient of the PSO loss pushes all points of the outermost branch that should be simplified and all shorter branches that merge into it towards the saddle value.

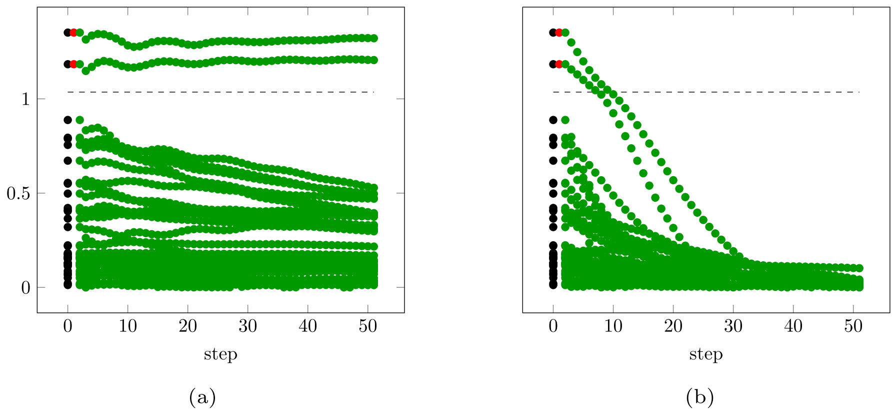


Fig. 5. Optimization of the weights. (a) Vineyard of simplification with PSO loss. (b) Vineyard of simplification with diagram loss.

down, to meet in the middle. Our simplification algorithm lifts all points to the saddle value, thus pushing (b_i, d_i) to the point (d_i, d_i). The death point does not move at all, and the birth point and all intermediate points are moved up. Secondly, consider a point (b_i, d_i) that corresponds to a short branch that merges into another short branch (b_j, d_j), $|d_j - b_j| < \varepsilon$, which in turn merges into a branch with persistence greater than ε. The diagram loss will try to move $f(x_i)$ and $f(y_i)$ towards the midpoint. The PSO loss will detect that it is not enough to eliminate the i-th branch alone, but the whole j-th branch also has to be destroyed, and it will send $b_i = f(x_i)$ and $d_i = f(y_i)$ to $d_j = f(y_j)$. See Fig. 4 for an illustration.

The first disadvantage of the diagram loss is that only critical points generate pairs in the persistence diagram. Accordingly, as explained in the previous paragraph, most input points are not used and receive no information during the backpropagation. To illustrate this, we take $f : \mathbb{R}^2 \to \mathbb{R}$ 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. Fig. 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 Figs. 2c and 2e, respectively.

Figs. 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 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.

Another significant advantage of the PSO loss is that it only requires computing the merge tree (to get the simplified function) once per epoch. On the other hand, the diagram loss requires recomputing the merge tree (to get a persistence diagram) after each step. This not only makes the process slower, but also introduces additional topological noise, evident in the vineyard. In experiments on real datasets, the diagram loss made topological phase 2–20 times longer than a phase with the PSO regularization, see Table 1 in Section 6.

Fig. 5 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.

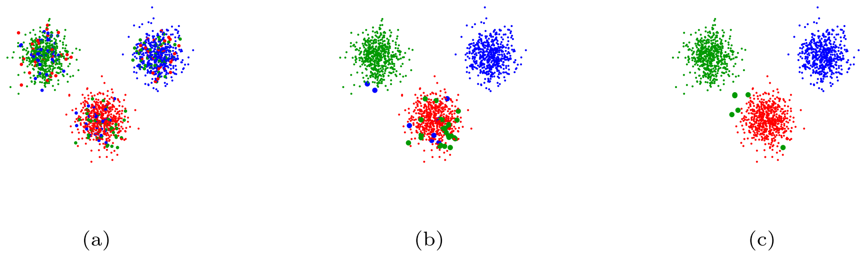


Fig. 6. (a) Input data: 1000 points sampled from three Gaussians, representing three distinct classes, with 20% of the labels randomly shuffled. (b) Labels predicted at epoch 500 without regularization. (c) Labels predicted at epoch 500 with PSO.

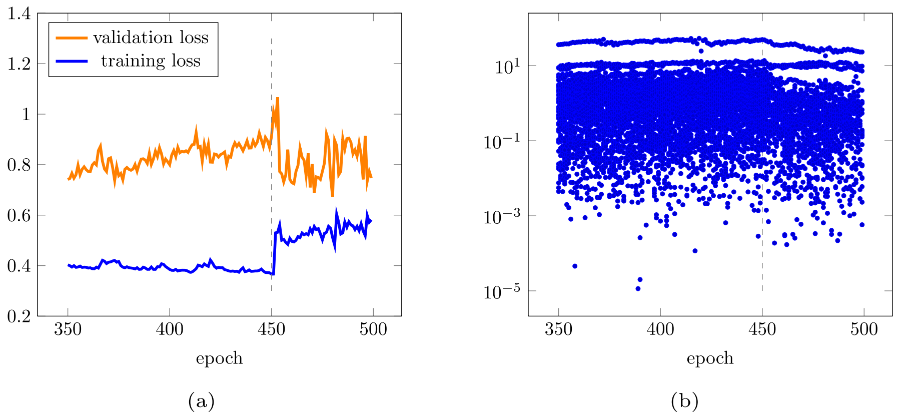


Fig. 7. (a) Training and validation loss, restricted to the later epochs when the network overfits the data. Simplification is applied after every epoch, following epoch 450, marked with a dashed line. (b) Vineyard of the confidence function during training.

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 [16], 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 Fig. 6a. 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.

Fig. 7a illustrates the training and validation losses, and Fig. 7b 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 indistinguishable from the noisy points.

Starting with epoch 450, we apply ten steps of topological simplification after every training epoch (i.e., we apply ten optimization steps to minimize the PSO loss). 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 Fig. 7a illustrates, after turning on simplification at epoch 450, the validation loss decreases by over 20%. Fig. 7b 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

Table 1
Timing of diagram loss and PSO loss, in seconds. Total column contains the running time of the topological regularization part in one phase (the first time regularization was applied). Loss computation column contains the total time spent computing the loss function, during the entire phase.

	Total		Loss computation		Ratio Dgm/PSO	
	PSO	Dgm	PSO	Dgm	Total	Loss only
Boston	0.074	0.140	0.010	0.100	1.896	9.665
Iran housing	0.066	0.094	0.007	0.061	1.426	8.174
Wine	0.129	0.305	0.023	0.252	2.362	10.938
Protein	3.701	9.511	0.771	8.299	2.570	10.762
Wireless	0.152	0.377	0.027	0.311	2.486	11.404
Wine (class)	0.163	0.409	0.031	0.341	2.512	11.158
Vertebral	0.063	0.106	0.006	0.065	1.696	10.130
SPECT	0.064	0.098	0.006	0.058	1.524	8.988
Wisconsin cancer	0.079	0.147	0.011	0.109	1.863	9.916
Semeion	0.156	0.334	0.024	0.261	2.144	11.030

Table 2
RMSD results on regression datasets comparing no regularization, ℓ_2 regularization of the weights, and topological simplification. Δ is the percentage of improvement from None to PSO.

	Regularization				Hyperparameters (best)			
Datasets	None	ℓ_2	PSO	Δ	k	t	n	σ
Wine	0.784	0.772	**0.759**	2.6%	15	0.001	6	0.001
Iran housing	0.122	0.113	**0.102**	16.7%	10	0.01	9	0.001
Boston	0.334	0.323	**0.314**	6.1%	20	0.001	9	0.001
Concrete	0.310	0.297	**0.288**	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	0.638	0.633	**0.624**	3.1%	20	0.001	6	0.001

these peaks which drives the network to match the class labels of the dominant class around the outliers. The effect can be clearly seen in Fig. 6c, which contains much fewer misclassified points than Fig. 6b, where no regularization was applied. Note that we consider the original labels (before random shuffling) to be the correct ones, and we use 'misclassified' with respect to them.

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 occurred.

6. Experiments

We study the performance of persistence-sensitive optimization on six regression problems and seven classification problems from the UCI repository [9]. 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.

Timing Table 1 lists the time spent in the topological regularization phase, either using our PSO method or the diagram loss. The times were measured during the first regularization phase in any given experiment, so that identical networks are used as input in both cases. Each method takes 10 steps. Besides listing the total time spent in the phase (including back-propagating weights through the network and taking steps with the optimizer), the table also breaks out the time spent only computing the loss. Because diagram loss has to be recomputed after every step, whereas PSO computes the diagram only once, this part of the computation is usually 10 times faster with PSO. The speed-up in total time is less dramatic — between 1.4 and 2.5 times — because the data sets are small, and persistence computation is much faster than back-propagation and weight updates.

Hyperparameters 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.

Table 3
Cross-entropy loss and accuracy results on classification datasets comparing no regularization, ℓ_2 regularization of the weights, and topological simplification. Δ is the percentage of improvement from None to PSO.

	Crossentropy				Accuracy			
	Regularization				Regularization			
Datasets	None	ℓ_2	PSO	Δ	None	ℓ_2	PSO	Δ
W. cancer	0.133	**0.083**	0.089	30.8%	0.969	0.983	**0.990**	2.1%
Wine	0.971	0.959	**0.929**	4.1%	0.592	0.600	**0.652**	10.2%
Semeion	0.481	0.478	**0.381**	20.8%	0.868	0.880	**0.902**	3.4%
Vertebral	0.393	0.381	**0.343**	12.8%	0.819	0.839	**0.841**	2.4%
Wireless	0.070	0.061	**0.060**	14.3%	0.970	0.981	**0.982**	1.1%
SPECT	0.352	0.341	**0.334**	5.7%	0.801	**0.828**	0.803	0%
Letter	0.272	0.259	**0.232**	14.8%	0.917	0.919	**0.933**	1.1%

Table 4
Hyperparameter values for the best model.

	Hyperparameters (best)			
Datasets	k	t	n	σ
Wisconsin cancer	15	0.0001	3	0.2
Wine	15	0.0001	0	0.001
Semeion	20	0.0001	9	0.2
Vertebral	15	0.0001	9	0.1
Wireless	25	0.0001	6	0.2
SPECT	10	0.0001	15	0.01
Letter recognition	20	0.01	3	0.2

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.

In appendix we list the hyperparameter ranges that were swept 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 principal 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 2 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) [8] 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 3 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. Table 4 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

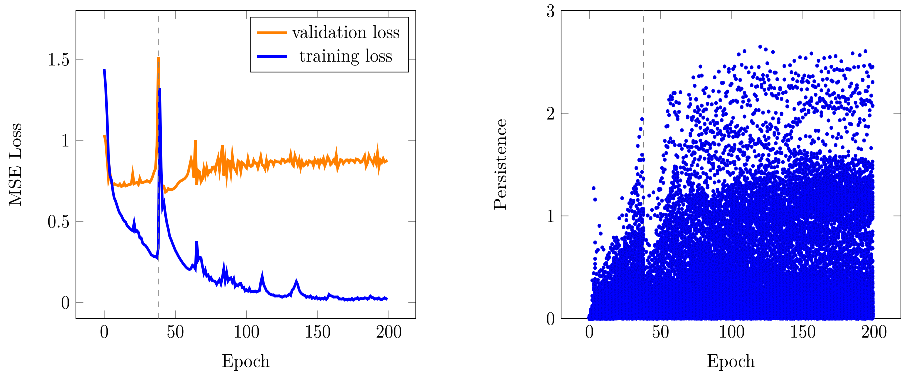


Fig. 8. (a) Training and validation loss curves for an experiment on the wine regression dataset. Performance is best at epoch 44, and simplification is applied only once, after epoch 43. (b) Vineyard over all epochs.

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 Fig. 8 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.

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 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 [8,5,18,14,7] 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 described only how to simplify extrema, i.e., 0-dimensional persistence diagrams. It is undoubtedly useful to incorporate higher-dimensional topological constraints, such as loops or voids in the data, into optimization. A key research direction is how to adapt these ideas to higher dimensional persistent homology. 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. This direction is explored in detail in the recent preprint [21].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

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. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

Appendix A. 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.

A.1. Regression

See Table A.5.

Table A.5
Hyperparameter ranges for regression datasets.

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

A.2. Classification

See Table A.6.

Table A.6
Hyperparameter ranges for classification datasets.

Datasets	k	t	n	σ
W. 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	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**

References

[1] H. Adams, T. Emerson, M. Kirby, R. Neville, C. Peterson, P. Shipman, S. Chepushtanova, E. Hanson, F. Motta, L. Ziegelmeier, Persistence images: A stable vector representation of persistent homology, J. Mach. Learn. Res. (ISSN 1532-4435) 18 (1) (2017) 218–252.

[2] D. Attali, M. Glisse, S. Hornus, F. Lazarus, D. Morozov, Persistence-sensitive simplification of functions on surfaces in linear time, Manuscript, 2009, presented at TopoInVis'09.

[3] U. Bauer, C. Lange, M. Wardetzky, Optimal topological simplification of discrete functions on surfaces, Discrete Comput. Geom. 47 (2) (2012) 347–377.

[4] S.P. Boyd, L. Vandenberghe, Convex Optimization, Cambridge University Press, 2004.

[5] R. Brüel-Gabrielsson, B.J. Nelson, A. Dwaraknath, P. Skraba, L.J. Guibas, G. Carlsson, A topology layer for machine learning, in: Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS), 2020, pp. 1553–1563.

[6] M. Carrière, F. Chazal, M. Glisse, Y. Ike, H. Kannan, A note on stochastic subgradient descent for persistence-based functionals: convergence and practical aspects, arXiv:2010.08356.

[7] M. Carriere, F. Chazal, Y. Ike, T. Lacombe, M. Royer, Y. Umeda, PersLay: a neural network layer for persistence diagrams and new graph topological signatures, in: Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS), 2020, pp. 2786–2796.

[8] C. Chen, X. Ni, Q. Bai, Y. Wang, A topological regularizer for classifiers via persistent homology, in: Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS), 2019, pp. 2573–2582.

[9] D. Dua, C. Graff, UCI Machine Learning Repository, 2017.

[10] H. Edelsbrunner, J. Harer, Computational Topology: An Introduction, American Mathematical Society, 2010.

[11] H. Edelsbrunner, D. Morozov, V. Pascucci, Persistence-sensitive simplification functions on 2-manifolds, in: Proceedings of the Annual Symposium on Computational Geometry, ACM, 2006, pp. 127–134.

[12] R.B. Gabrielsson, G. Carlsson, Exposition and interpretation of the topology of neural networks, in: 2019 18th IEEE International Conference on Machine Learning and Applications (ICMLA), IEEE, 2019, pp. 1069–1076, https://doi.org/10.1109/ICMLA.2019.00180.

[13] W.H. Guss, R. Salakhutdinov, On characterizing the capacity of neural networks using algebraic topology, arXiv:1802.04443.

[14] C. Hofer, R. Kwitt, M. Niethammer, Learning representations of persistence barcodes, J. Mach. Learn. Res. (ISSN 1532-4435) 20 (126) (2019) 1–45.
[15] C. Hofer, F. Graf, B. Rieck, M. Niethammer, R. Kwitt, Graph filtration learning, in: International Conference on Machine Learning, PMLR, 2020, pp. 4314–4323.
[16] M. Kerber, D. Morozov, A. Nigmetov, Geometry helps to compare persistence diagrams, J. Exp. Algorithmics (ISSN 1084-6654) 22 (2017) 1.4, https://doi.org/10.1145/3064175.
[17] K. Kim, J. Kim, M. Zaheer, J. Kim, F. Chazal, L. Wasserman, PLLay: efficient topological layer based on persistence landscapes, in: 34th Conference on Neural Information Processing Systems (NeurIPS 2020), 2020.
[18] J. Leygonie, S. Oudot, U. Tillmann, A framework for differential calculus on persistence barcodes, Found. Comput. Math. (2021) 1–63, https://doi.org/10.1007/s10208-021-09522-y.
[19] J. Leygonie, M. Carrière, T. Lacombe, S. Oudot, A gradient sampling algorithm for stratified maps with applications to topological data analysis, Math. Program. (2023) 1–41, https://doi.org/10.1007/s10107-023-01931-x.
[20] A.Y. Ng, Feature selection, L_1 vs. L_2 regularization, and rotational invariance, in: Proceedings of the Twenty-First International Conference on Machine Learning, 2004, p. 78.
[21] A. Nigmetov, D. Morozov, Topological optimization with big steps, Discrete Comput. Geom. (2024) 1–35, https://doi.org/10.1007/s00454-023-00613-x.
[22] B. Rieck, M. Togninalli, C. Bock, M. Moor, M. Horn, T. Gumbsch, K. Borgwardt, Neural persistence: A complexity measure for deep neural networks using algebraic topology, in: International Conference on Learning Representations, 2019, https://openreview.net/forum?id=ByxkijC5FQ.
[23] B. Schölkopf, A.J. Smola, F. Bach, Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond, MIT Press, ISBN 9780262194754, 2002.
[24] P. Tseng, Applications of a splitting algorithm to decomposition in convex programming and variational inequalities, SIAM J. Control Optim. (ISSN 0363-0129) 29 (1) (1991) 119–138, https://doi.org/10.1137/0329006.