PersLay

Tao Hou, CS410/510

University of Oregon

Acknowledgment

Contents of the slide (including figures) are based on the paper:

- Carriere, M., Chazal, F., Ike, Y., Lacombe, T., Royer, M. and Umeda, Y., 2020, June. Perslay: A neural network layer for persistence diagrams and new graph topological signatures. In International Conference on Artificial Intelligence and Statistics (pp. 2786-2796). PMLR.

PersLay

- A general and versatile framework for learning vectorizations of persistence diagrams, which encompasses most of the vectorization techniques used in the literature

PersLay

- A general and versatile framework for learning vectorizations of persistence diagrams, which encompasses most of the vectorization techniques used in the literature
- Aka. it is still a vectorization technique, but somehow not a fixed one (e.g., parameter wise)

PersLay

- A general and versatile framework for learning vectorizations of persistence diagrams, which encompasses most of the vectorization techniques used in the literature
- Aka. it is still a vectorization technique, but somehow not a fixed one (e.g., parameter wise)
- Instead, the parameters used for the vectorization can be learned from the data using gradient descent

PersLay

- A general and versatile framework for learning vectorizations of persistence diagrams, which encompasses most of the vectorization techniques used in the literature
- Aka. it is still a vectorization technique, but somehow not a fixed one (e.g., parameter wise)
- Instead, the parameters used for the vectorization can be learned from the data using gradient descent
- So that the layer is truly a part of the model (neural network), you will have a "custom-made" vectorization layer targeting your data specifically

- Inspired by *Deep Set* architecture, which focuses on learning from features that are sets
  - Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. Deep sets. In Advances in Neural Information Processing Systems, pages 33913401, 2017.
- Major goal is to be permutation invariant and to accommodate inputs of different size

- Inspired by *Deep Set* architecture, which focuses on learning from features that are sets
  - Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. Deep sets. In Advances in Neural Information Processing Systems, pages 33913401, 2017.
- Major goal is to be permutation invariant and to accommodate inputs of different size
- Generic neural network layer:

$$\textsc{PersLay}(PD) := \mathrm{op}(\{w(p) \cdot \phi(p)\}_{p \in PD})$$

- Inspired by *Deep Set* architecture, which focuses on learning from features that are sets
  - Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. Deep sets. In Advances in Neural Information Processing Systems, pages 33913401, 2017.
- Major goal is to be permutation invariant and to accommodate inputs of different size
- Generic neural network layer:

$$\textsc{PersLay}(PD) := \mathrm{op}(\{w(p) \cdot \phi(p)\}_{p \in PD})$$

- Note:
  - $w(p)$ is a scalar weight
  - $\phi(p)$ may be a vector (aka. a function $\phi : \mathbb{R}^2 \to \mathbb{R}^q$)
  - op is an "aggregate" function performed on each component of the vector $\phi(p)$ but over all points p in PD, which could be any permutation invariant operation (such as minimum, maximum, sum, k-th largest value...)

For $PD = \{p_1, \ldots, p_n\}$, suppose:

- $w(p_1) \cdot \phi(p_1) = [x_1^1, \ldots, x_q^1]$
- $\vdots$
- $w(p_n) \cdot \phi(p_n) = [x_1^n, \ldots, x_q^n]$

For $PD = \{p_1, \ldots, p_n\}$, suppose:

- $w(p_1) \cdot \phi(p_1) = [x_1^1, \ldots, x_q^1]$
- $\vdots$
- $w(p_n) \cdot \phi(p_n) = [x_1^n, \ldots, x_q^n]$

Let op $=$ sum and the output vector be $[z_1, \ldots, z_q]$ then:

For $PD = \{p_1, \ldots, p_n\}$, suppose:

- $w(p_1) \cdot \phi(p_1) = [x_1^1, \ldots, x_q^1]$
- $\vdots$
- $w(p_n) \cdot \phi(p_n) = [x_1^n, \ldots, x_q^n]$

Let op $=$ sum and the output vector be $[z_1, \ldots, z_q]$ then:

- $z_i = \text{sum}(x_i^1, \ldots, x_i^n)$

For $PD = \{p_1, \ldots, p_n\}$, suppose:

- $w(p_1) \cdot \phi(p_1) = [x_1^1, \ldots, x_q^1]$
- $\vdots$
- $w(p_n) \cdot \phi(p_n) = [x_1^n, \ldots, x_q^n]$

Let $\mathrm{op} = \mathrm{sum}$ and the output vector be $[z_1, \ldots, z_q]$ then:

- $z_i = \mathrm{sum}(x_i^1, \ldots, x_i^n)$

Setting the different ϕ, w, op give you different vectorizations

Recovering persistence landscape

- $\phi := \phi_\Lambda$, where

$$\phi_\Lambda(p) = [\Lambda_p(t_1), \Lambda_p(t_2), \ldots, \Lambda_p(t_q)],$$

  Λ_p associated to a point $p = (x, y) \in \mathbb{R}^2$ is

$$\Lambda_p(t) = \max\{0, y - |t - x|\},$$

  and $t_1, \ldots, t_q \in \mathbb{R}$.
- $w(p) = 1$
- $\mathrm{op} = \max_k$

Recovering persistence silhouette

- $\phi := \phi_\Lambda$ as before
- $w(p)$ arbitrary
- $\mathrm{op} = \mathrm{sum}$

Recovering persistence image

- $\phi := \phi_\Gamma$, where

$$\phi_\Gamma(p) = [\Gamma_p(t_1), \Gamma_p(t_2), \ldots, \Gamma_p(t_q)],$$

  Γ_p associated to a point $p = (x, y) \in \mathbb{R}^2$ is

$$\Gamma_p(t) = \exp\left(-\|p - t\|_2^2/(2\sigma^2)\right)$$

  and $t_1, \ldots, t_q \in \mathbb{R}^2$.
- $w(p)$ arbitrary
- $\mathtt{op} = \mathtt{sum}$

Another ϕ

- $\phi := \phi_\Lambda$, where
$$\phi_\Lambda(p) = [L_{\Delta_1}(p), L_{\Delta_2}(p), \ldots, L_{\Delta_q}(p)],$$
the line function L_Δ associated to a line Δ with direction vector $e_\Delta \in \mathbb{R}^2$ and bias $b_\Delta \in \mathbb{R}$ is
$$L_\Delta(p) = \langle p, e_\Delta \rangle + b_\Delta,$$
$\Delta_1, \ldots, \Delta_q$ are q lines in the plane.
- It can be used to recover the Sliced Wasserstein kernel:
  - Mathieu Carriere, Marco Cuturi, and Steve Oudot. Sliced Wasserstein kernel for persistence diagrams. In International Conference on Machine Learning, volume 70, pages 664 - 673, jul 2017.

Choosing the weight in practice

In the application that follows, the authors adopts a way to choose the weight $w(p)$ for a $p \in PD$ that is also general enough:

- First normalize all points in PD to a unit square $[0, 1] \times [0, 1]$
- Divide the unit square into $N \times N$ grids, assign a weight $w_{i,j}$ to each point p falling in cell (i, j)

Choosing the weight in practice

In the application that follows, the authors adopts a way to choose the weight $w(p)$ for a $p \in PD$ that is also general enough:

- First normalize all points in PD to a unit square $[0, 1] \times [0, 1]$
- Divide the unit square into $N \times N$ grids, assign a weight $w_{i,j}$ to each point p falling in cell (i, j)
- All the weights, $\{w_{i,j}\}_{1 \leq i,j \leq N}$, are learnable
- For this, we have to show that the weight is "differentiable"

For $PD = \{p_1, \ldots, p_n\}$, suppose:

- $\phi(p_1) = [x_1^1, \ldots, x_q^1]$
- $\vdots$
- $\phi(p_n) = [x_1^n, \ldots, x_q^n]$

For $PD = \{p_1, \ldots, p_n\}$, suppose:

- $\phi(p_1) = [x_1^1, \ldots, x_q^1]$
- $\vdots$
- $\phi(p_n) = [x_1^n, \ldots, x_q^n]$

We have:

- $w_{c(p_1)} \cdot \phi(p_1) = [w_{c(p_1)} \cdot x_1^1, \ldots, w_{c(p_1)} \cdot x_q^1]$
- $\vdots$
- $w_{c(p_n)} \cdot \phi(p_n) = [w_{c(p_n)} \cdot x_1^n, \ldots, w_{c(p_n)} \cdot x_q^n]$

For $PD = \{p_1, \ldots, p_n\}$, suppose:

- $\phi(p_1) = [x_1^1, \ldots, x_q^1]$
- $\vdots$
- $\phi(p_n) = [x_1^n, \ldots, x_q^n]$

We have:

- $w_{c(p_1)} \cdot \phi(p_1) = [w_{c(p_1)} \cdot x_1^1, \ldots, w_{c(p_1)} \cdot x_q^1]$
- $\vdots$
- $w_{c(p_n)} \cdot \phi(p_n) = [w_{c(p_n)} \cdot x_1^n, \ldots, w_{c(p_n)} \cdot x_q^n]$

Let op $=$ sum and the output vector be $[z_1, \ldots, z_q]$ where

$$z_i = \sum_{j=1}^{n} w_{c(p_j)} \cdot x_i^j = \sum_{1 \leq \alpha, \beta \leq N} \left(w_{\alpha,\beta} \sum_{c(p_j)=(\alpha,\beta)} x_i^j \right)$$

For $PD = \{p_1, \ldots, p_n\}$, suppose:

- $\phi(p_1) = [x_1^1, \ldots, x_q^1]$
- $\vdots$
- $\phi(p_n) = [x_1^n, \ldots, x_q^n]$

We have:

- $w_{c(p_1)} \cdot \phi(p_1) = [w_{c(p_1)} \cdot x_1^1, \ldots, w_{c(p_1)} \cdot x_q^1]$
- $\vdots$
- $w_{c(p_n)} \cdot \phi(p_n) = [w_{c(p_n)} \cdot x_1^n, \ldots, w_{c(p_n)} \cdot x_q^n]$

Let op = sum and the output vector be $[z_1, \ldots, z_q]$ where

$$z_i = \sum_{j=1}^{n} w_{c(p_j)} \cdot x_i^j = \sum_{1 \le \alpha, \beta \le N} \left(w_{\alpha,\beta} \sum_{c(p_j)=(\alpha,\beta)} x_i^j \right)$$

Then

$$\frac{\partial z_i}{\partial w_{\alpha,\beta}} = \sum_{c(p_j)=(\alpha,\beta)} x_i^j$$

Example of PersLay: Graph classification

- Task: We have a collection of graphs (social networks, medical or biological frameworks) which are labelled as different classes, and we want to train a classifier

Example of PersLay: Graph classification

- Task: We have a collection of graphs (social networks, medical or biological frameworks) which are labelled as different classes, and we want to train a classifier
- Graph classifications are generally hard even with the current deep neural networks, because it's not so easily vectorized (it's two sets but with some additional structures)

Example of PersLay: Graph classification

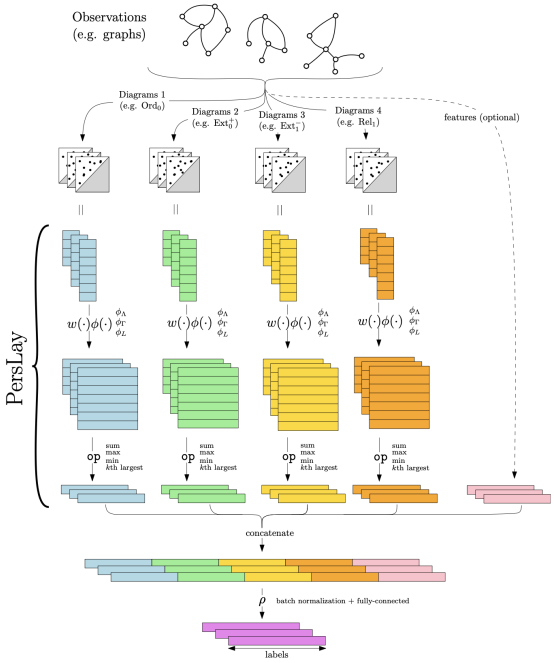
- Task: We have a collection of graphs (social networks, medical or biological frameworks) which are labelled as different classes, and we want to train a classifier
- Graph classifications are generally hard even with the current deep neural networks, because it's not so easily vectorized (it's two sets but with some additional structures)
- The first thing the authors do is to convert graphs (the raw data) into PDs (features)
- Of course, we need to build filtration on graphs, and here, we assign values to the vertices and build sublevelset filtration

Example of PersLay: Graph classification

- Task: We have a collection of graphs (social networks, medical or biological frameworks) which are labelled as different classes, and we want to train a classifier
- Graph classifications are generally hard even with the current deep neural networks, because it's not so easily vectorized (it's two sets but with some additional structures)
- The first thing the authors do is to convert graphs (the raw data) into PDs (features)
- Of course, we need to build filtration on graphs, and here, we assign values to the vertices and build sublevelset filtration
- Notice that the vertices do not automatically come with real values, so the authors utilize *Heat Kernel Signatures* (HKS) which assigns a value to a vertex based on the graph structure

Setting

- They use a very simple network architecture, namely a two-layer network
- The first layer is PersLay, which processes persistence diagrams
- The second is a fully-connected layer whose output is used for predictions
- This simplistic two-layer architecture is designed to understanding the effect of the PD and PersLay rather than achieving the best possible performances.


Observations
(e.g. graphs)
Diagrams 1
(e.g. Ord_0)
Diagrams 2
(e.g. Ext_0^+)
Diagrams 3
(e.g. Ext_1^-)
Diagrams 4
(e.g. Rel_1)
features (optional)
PersLay
$w(\cdot)\phi(\cdot)$
ϕ_Λ
ϕ_Γ
ϕ_L
op
sum
max
min
kth largest
concatenate
ρ
batch normalization + fully-connected
labels

Test

- They run on 11 datasets some of which are social networks and others are from biology or medical areas
- For each dataset, they also perform ten-fold validation and report the average and best ten-fold results

Test Results

Comparison of using the traditional graph features (the eigenvalues of the normalized graph Laplacian along with the deciles of the computed HKS), PD alone, and combination

	Spectral alone	PD alone		PersLay
		Extended	Ordinary	
REDDIT5K	49.7($\pm$0.3)	55.0	52.5	55.6($\pm$0.3)
REDDIT12K	39.7($\pm$0.1)	44.2	40.1	47.7($\pm$0.2)
COLLAB	67.8($\pm$0.2)	71.6	69.2	76.4($\pm$0.4)
IMDB-B	67.6($\pm$0.6)	68.8	64.7	71.2($\pm$0.7)
IMDB-M	44.5($\pm$0.4)	48.2	42.0	48.8($\pm$0.6)
COX2 *	78.2($\pm$1.3)	81.5	79.0	80.9($\pm$1.0)
DHFR *	69.5($\pm$1.0)	78.2	71.8	80.3($\pm$0.8)
MUTAG *	85.8($\pm$1.3)	85.1	70.2	89.8($\pm$0.9)
PROTEINS *	73.5($\pm$0.3)	72.2	69.7	74.8($\pm$0.3)
NCI1 *	65.3($\pm$0.2)	72.3	68.9	73.5($\pm$0.3)
NCI109 *	64.9($\pm$0.2)	67.0	66.2	69.5($\pm$0.3)

Comparison with other methods

The authors compare performances with five other graph classification methods with general good performance:

- Scale-Variant topo: leverages a kernel for ordinary persistence diagrams computed on point cloud used to encode the graphs
- RetGK: a kernel method for graphs that leverages eventual attributes on the graph vertices and edges
- FGSD: a finite-dimensional graph embedding that does not leverage attributes
- GCNN and GIN: two graph neural network approaches that reach top-tier results

Test Results

Dataset	SV[1]	RetGK* [2]	FGSD [3]	GCNN [4]	GIN [5]	PersLay	
						Mean	Max
REDDIT5K	—	56.1	47.8	52.9	57.0	55.6	56.5
REDDIT12K	—	48.7	—	46.6	—	47.7	49.1
COLLAB	—	81.0	80.0	79.6	80.1	76.4	78.0
IMDB-B	72.9	71.9	73.6	73.1	74.3	71.2	72.6
IMDB-M	50.3	47.7	52.4	50.3	52.1	48.8	52.2
COX2*	78.4	80.1	—	—	—	80.9	81.6
DHFR*	78.4	81.5	—	—	—	80.3	80.9
MUTAG*	88.3	90.3	92.1	86.7	89.0	89.8	91.5
PROTEINS*	72.6	75.8	73.4	76.3	75.9	74.8	75.9
NCI1*	71.6	84.5	79.8	78.4	82.7	73.5	74.0
NCI109*	70.5	—	78.8	—	—	69.5	70.1

Gudhi implementation

https://gudhi.inria.fr/python/latest/representations_tflow_itf_ref.html