Adversarial Weight Matrix Perturbations for Graph Semi-Supervised Poisson Learning

Alexander Zheng

Submitted under the supervision of Professor Jeffrey Calder to the University Honors Program at the University of Minnesota-Twin Cities in partial fulfillment of the requirements for the degree of Bachelor of Science, *summa cum laude* in Mathematics.

May 9, 2022

Acknowledgements

This paper is inspired by the "Adversarial Learning" senior project written by Quinten Norton under the supervision of Professor Jeff Calder.

I want to express my sincerest gratitude to Professor Jeff Calder for serving as my Honors Thesis advisor. His mentorship was pivotal in helping me to discover the beauty within the intersection of computer science and mathematics.

I would also like to thank my thesis readers, Professors William Leeb and Gilad Lerman, for supporting me throughout the writing process.

Finally, a large thank you to my brother David as well as my parents, Jibin and Sifang, for the continuous support in my studies.

Contents

1 Abstract

This paper will explore the adversarial learning problem for Poisson Learning, a graph-based semi-supervised machine learning algorithm that excels at low-label rates. A natural adversarial attack for Poisson learning is to modify the dataset's weight matrix by adding a weighted perturbation matrix. We will calculate the perturbation matrix that maximizes the norm of the perturbation solution of the Poisson Learning equation via gradient ascent and provide computational evidence that the derived perturbation provides more degradation in accuracy than random perturbations. Additionally, this paper analyzes the effect of inserting new edges where the perturbation matrix is greatest.

2 Introduction

Machine learning models have a number of applications including image and face recognition, text-to-speech, and medical imaging analysis. On a high level, machine learning algorithms are developed to learn relationships between data and labels. For setup, we have datapoints $\{x_1, x_2, \ldots, x_n\} \subseteq \mathcal{X}$ where $\mathcal{X}$ denotes the space in which our datapoints live, and we have their corresponding labels $\{y_1, y_2, \ldots, y_n\} \subseteq \mathcal{Y}$ where $\mathcal{Y}$ denotes the label space, often listed as $\mathbb{R}^k$. Our data x_i comes in all shapes and forms, such as images, text, and in our case, graph structures. A datapoint x_i may also have a label y_i attached, which groups it with other datapoints of the same label. The group of datapoints which share the same label is typically called a *class* which identifies them as the same type or sharing the same characteristics. For example, a doctor could employ machine learning by using medical image recognition technology to identify cancerous tumors on an X-ray scan. In this case, the machine learning model would be trained on X-ray data which are known to either contain or not contain cancerous tumors. These X-ray pictures could be labeled as "cancerous" or "non-cancerous."

These techniques allow for computers to perform traditionally computationally difficult and abstract problems such as image recognition into solvable contexts. Machine learning algorithms typically fall into three broad categories: (i) fully-supervised, (ii) semi-supervised, and (iii) unsupervised. The purpose of fully-supervised algorithms is to take all of the labeled training data $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$ and *learn* a function $f : \mathcal{X} \to \mathcal{Y}$ that generalizes the rule $x_i \mapsto y_i$. We deem such a function to generalize well if it gives the correct label to some datapoint $x_i \in \mathcal{X}$ not contained in the training or validation set. *Semi-supervised* machine learning employs a (typically) small amount of labeled data $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ along with a set of unlabeled datapoints $x_{m+1}, \ldots, x_n$ to obtain enhanced learning outcomes compared to exclusively using labeled data. *unsupervised* learning produces labeling functions by making use of exclusively unlabeled data $x_1, x_2, \ldots, x_n$.

While this concept is mathematically grounded, it is often difficult to find an accurate function f in practice. An important hurdle to overcome is the difficulty in obtaining labeled data. Fully-supervised learning requires that each sample be manually labeled, which is not only costly, but also counterintuitive because machine learning algorithms are developed to tackle this exact challenge.

Thus, we often study less costly machine learning techniques in the form of *unsupervised* and *semi-supervised* models. In this paper, we will turn our focus to *semi-supervised* learning, specifically in the graph-based setting.

3 Graph-Based Learning

We pivot to a subset of machine learning algorithms that operate on graphs. First let us define $\mathcal{G} = (\mathcal{X}, \mathcal{W})$ to be a graph with vertices $\mathcal{X}$ and edge weights $\mathcal{W} = (w_{xy})_{x,y \in \mathcal{X}}$. Each edge weight w_{xy} encodes the similarity between datapoints x and y and is large when the two are similar and is small or zero when there are dissimilar. In our case, edge weights are assumed to be nonnegative, and the weight matrix is assumed to be symmetric, that is, $w_{xy} = w_{yx}$ for all $x, y \in \mathcal{X}$. Paired with the graph is the label space $\mathcal{Y} = \{e_1, e_2, \ldots, e_k\}$ for a dataset with k classes, where $e_i \in \mathbb{R}^k$ is the i-th standard basis vector in $\mathbb{R}^k$ which corresponds to the i-th class.

In the semi-supervised graph learning setting, we consider the label subset $\Gamma \subset \mathcal{X}$ for which we have the labeling function $g : \Gamma \to \mathcal{Y}$. Due to the costliness of obtaining labeled data, usually $|\Gamma| \ll |\mathcal{X}|$. The goal of semi-supervised graph learning is to accurately predict the labels of the unlabeled datapoints $\mathcal{X} \setminus \Gamma$ by leveraging properties of the more cost-effective unlabeled datapoints compared to simply using exclusively labeled data. We often think of this as propogating the labels from labeled nodes within the graph to similar unlabeled nodes.

We leverage the *semi-supervised smoothness assumption* [3], which dictates that similar datapoints in dense parts of the graph should have similar labels. We may enforce this assumption by defining a smoothness function $\mathcal{E}$ that measures the smoothness of a labeling function $u : \mathcal{X} \to \mathbb{R}^k$. We consider $u(x)$ to be a k-dimensional vector for which each u_i component measures the likelihood that x_i belongs to class i. From here, we can solve the optimization problem

$$\begin{cases} \text{Minimize } \mathcal{E}(u) \text{ over all u} \\ \text{subject to } u = g \text{ on } \Gamma \end{cases} . \tag{1}$$

Once the optimization problem is solved, we can deduce our learned labels for each $x \in \mathcal{X} \setminus \Gamma$ through the function $\ell : \mathcal{X} \setminus \Gamma \to \{1, \dots, k\}$ defined as

$$\ell(x) = \operatorname*{argmax}_{i \in \{1,\dots,k\}} u_i(x).$$

In the following sections, we will introduce commonly used versions of the smoothness function $\mathcal{E}$.

3.1 Laplacian Learning

A widely used smoothness function is the graph Dirichlet energy

$$\mathcal{E}_2(u) := \frac{1}{2} \sum_{x,y \in \mathcal{X}} w_{xy} |u(y) - y(x)|^2.$$

Setting $\mathcal{E} = \mathcal{E}_2$ in (1) we get what is called *Laplacian regularization* [8]. Due to its popularity, Laplacian regularization has spawned many variants as methods for semi-supervised learning. One such variant is the p-Laplace regularization, which studies the energy function $\mathcal{E} = \mathcal{E}_p$ defined as

$$\mathcal{E}_p(u) := \frac{1}{2} \sum_{x,y \in \mathcal{X}} w_{xy} |u(y) - u(x)|^p.$$

p-Laplace regularization was initially introduced in [7], and proposed with large p in [4] to be the superior regularizer for very low label rates which cause Laplacian regularization to perform poorly. The case where $p = \infty$, called *Lipschitz learning*, has also been studied in [5].

The major drawback as p increases is the additional computational burden which limits practical usage. Additionally, a fundamental issue arises in any Laplace learning algorithm at low label rates: a large constant bias appears in the solution of the Laplace equation which can account for *nearly all* of the breakdown in accuracy for Laplace learning. This bias appears only at low label rates and was what Poisson Learning [2] was introduced to address.

3.2 Poisson Learning

We begin with an interpretation of the constant bias as a random walk. We again consider our graph $\mathcal{G} = (\mathcal{X}, \mathcal{W})$. Suppose we have labeled vertices $(x_1, x_2, \ldots, x_m)$ and unlabeled vertices $(x_{m+1}, x_{m+2}, \ldots, x_n)$ where $n = |\mathcal{X}|$, our total number of datapoints. Let $x \in \mathcal{X}$ and let $\mathcal{X}_0^x, \mathcal{X}_1^x, \mathcal{X}_2^x, \ldots$ be a random walk on $\mathcal{X}$ starting at $\mathcal{X}_0^x = x$ with the transition probabilities

$$\mathbb{P}(\mathcal{X}_{k+1}^x = x_j | \mathcal{X}_k^x = x_i) = \frac{w_{ij}}{d_i}$$

where d_i denotes the *degree* of vertex i which we define as $d_i = \sum_{j=1}^n w_{ij}$.

Let us define a stopping time τ of the random walk to be the time a random walker takes to reach a labeled vertex for the first time, i.e.

$$\tau = \inf\{k \geq 0 : \mathcal{X}_k^x \in \{x_1, x_2, \ldots, x_m\}\}$$

If we let $i_\tau \leq m$ denote the index of the vertex reached so $\mathcal{X}_\tau^x = x_{i_\tau}$, we find that

$$u(x) = \mathbb{E}[y_{i_\tau}]$$

where $u(x)$ is the solution to the Laplace learning problem. In this interpretation, we allow a random walker to walk from x until it hits a labeled point, and we record that label $y_{i_\tau} \in \mathcal{Y}$. We then average these labels over many random walkers to obtain the value of $u(x)$.

When we are in the setting of low label rates, the stopping time τ becomes so large that it passes the *mixing time* of the random walk, and the distribution of $\mathcal{X}_\tau^x$ becomes close to the stationary distribution of the random walk, which omits

$$u(x) \approx \frac{\sum_{i=1}^m d_i y_i}{\sum_{j=1}^m d_j} =: \overline{y}_w.$$

The function u is nearly constant, except for sharp spikes at the labeled vertices. This can have disastrous consequences for the accuracy of Laplace learning since many vertices are given the same label. Note that the constant vector $\overline{y}_w$ depends only on the degrees of the labeled nodes in the graph, which is sensitive to the local graph structure surrounding those nodes.

Poisson learning instead releases a random walker from labeled vertices and counts each time it visits any particular vertex. In this sense, we define

$$w_T(x_i) = \mathbb{E}\left[\sum_{k=0}^{T}\sum_{j=1}^{m} y_j \mathbb{1}_{\{\mathcal{X}_k^{x_j} = x_i\}}\right]$$

where $\mathbb{1}_{\{\mathcal{X}_k^{x_j} = x_i\}}$ is our indicator function for the random walker visiting x_i and T represents the amount we run our random walk. For small T, the quantity is meaningful, but as T grows increasingly large, we encounter the same issue as Laplace learning where we grow closer to the stationary distribution.

At a high level, we may discard the long-term behavior in order to only record the short-term behavior of the random walk by subtracting off $\overline{y} = \frac{1}{m}\sum_{j=1}^m y_j$ within the sum and normalizing by

d_i. Then w_T becomes

$$u_T(x_i) = \mathbb{E}\left[\sum_{k=0}^{T} \frac{1}{d_i} \sum_{j=1}^{m} (y_j - \overline{y}) \mathbb{1}_{\{\mathcal{X}_k^{x_j} = x_i\}}\right].$$

It turns out that as $T \to \infty$, the function $u_T(x_i)$ converges to the solution of the Poisson equation, which we define next.

Let $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ if $i \neq j$. We have the Poisson equation

$$\mathcal{L}u(x_i) = \sum_{j=1}^{m} (y_j - \overline{y}) \delta_{ij} \tag{2}$$

satisfying $\sum_{i=1}^{n} d_i u(x_i) = 0$ to obtain a unique solution. After determining the solution of the Poisson equation, we obtain our learned labels through the function ℓ defined by

$$\ell(x_i) = \underset{j \in \{1,\ldots,k\}}{\operatorname{argmax}} \{u_i(x) \cdot e_j\}.$$

We continue to study Poisson learning in the adversarial case.

4 Adversarial Poisson Learning Problem

All machine learning algorithms, from unsupervised to fully-supervised, require the use of some combination of datapoints and their labels to "train" their models. Adversarial attacks seek to hurt machine learning models' performances while still remaining inconspicuous. We study adversarial machine learning problems to ensure these models are robust to bad actors. For example, consider self-driving cars which use computer vision to visualize its surroundings. If a self-driving car fails to interpret road signage such as stop or yield signs, human lives may be put in danger. Thus, defending against such adversarial attacks are of utmost importance.

The goal of this paper is to test one method an adversary may employ to degrade Poisson Learning's accuracy. This method will seek to corrupt the original training data by perturbing the dataset's weight matrix, but will not modify the Poisson equation in any way.

4.1 Adversarial Poisson Equation

Let our graph $\mathcal{G} = (\mathcal{X}, \mathcal{W})$ where $\mathcal{X}$ denotes the vertices of our graph and $\mathcal{W}$ refers to the graph's symmetric weight matrix. We assume that an adversarial attack would affect $\mathcal{W}$ by corrupting either the training data or unlabeled data. The adversary would change the weight matrix $\mathcal{W}$ to $\mathcal{W} + \epsilon\mathcal{V}$, where $\epsilon > 0$ and $\mathcal{V}$ is the symmetric perturbation matrix chosen by the adversary. We assume that $\mathcal{W}+\epsilon\mathcal{V}$ is non-negative in order to be a valid weight matrix. We write the corresponding solution $u_\epsilon = u + \epsilon v$ for a perturbed solution v. Denote the Laplacian matrix for some symmetric weight matrix $\mathcal{W}$ and its diagonal degree matrix $\mathcal{D} = \operatorname{diag}(d_i)_{i=1}^n$ as $\mathcal{L}_\mathcal{W} := \mathcal{W} - \mathcal{D}$.

We may now write the Poisson equation for the perturbed weight matrix as

$$\mathcal{L}_{\mathcal{W}+\epsilon\mathcal{V}}(u + \epsilon v) = f \tag{3}$$

where f corresponds to the right hand side of (2). Let us write the corresponding solution $u_\epsilon \approx u+\epsilon v$ for a perturbed solution v. Because $\mathcal{L}_\mathcal{W}$ is linear in $\mathcal{W}$, We can rewrite the equation as

$$(\mathcal{L}_\mathcal{W} + \epsilon\mathcal{L}_\mathcal{V})(u + \epsilon v) = f$$

This can be expanded to

$$\mathcal{L}_\mathcal{W}u + \epsilon(\mathcal{L}_\mathcal{V}u + \mathcal{L}_\mathcal{W}v) + \epsilon^2\mathcal{L}_\mathcal{V}v = f.$$

We discard the ϵ^2 term and use the equality $\mathcal{L}_\mathcal{W}u = f$ to arrive at

$$\mathcal{L}_\mathcal{W}v = -\mathcal{L}_\mathcal{V}u. \tag{4}$$

To isolate v on the left hand side, we first diagonalize the graph Laplacian $\mathcal{L}_\mathcal{W}$. We may do this because $\mathcal{L}_\mathcal{W}$ is real $n \times n$ symmetric matrix. Then there exists an $n \times n$ orthogonal matrix V (i.e. $VV^T = V^TV = I$) and an $n \times n$ diagonal matrix Λ such that

$$\mathcal{L}_\mathcal{W} = V\Lambda V^T.$$

The columns of V are exactly the eigenvectors of $\mathcal{L}_{\mathcal{W}}$, and the diagonal entries of Λ are the corresponding non-negative eigenvalues. Let us write $\lambda_i = \Lambda_{ii}$ for convenience. We also assume that the eigenvalues are ordered in increasing order, so

$$0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n.$$

We observe that the first eigenvalue $\lambda_1 = 0$. This is due to the fact that the first eigenvalue $\lambda_1 = 0$ since the constant vector $v = (1, 1, \ldots, 1)$ satisfies $\mathcal{L}_{\mathcal{W}} v = 0$. Intuitively, this is because the graph Laplacian's diagonal entries are simply the negative of the sum over their respective rows, or weights, and hence cancel to 0 when dotted with the constant vector (Recall that $\mathcal{L}_{\mathcal{W}} := \mathcal{W} - \mathcal{D}$ with the i-th entry of $\mathcal{D}$ defined as $d_i = \sum_{j=1}^{n} w_{ij}$). We also assume that the graph $\mathcal{G}$ is connected, so λ_1 is the only eigenvalue equal to 0 with corresponding constant eigenvector $\mathbb{1}$. The existence of a null eigenvalue will lead us to impose the mean-zero condition on our equality (4).

To solve (4), let us first write the equality as

$$V \Lambda V^T v = -\mathcal{L}_{\mathcal{V}} u.$$

We continue by multiplying by V^T on both sides so

$$\Lambda V^T v = -V^T \mathcal{L}_{\mathcal{V}} u.$$

Typically, multiplying by the inverse of a diagonal matrix is simple. Normally the inverse would simply be the same diagonal matrix but with each diagonal entry inverted. However, $\lambda_1 = 0$. We now impose the same *mean-zero condition* used within the original Poisson learning paper [2]. If we choose to look for a solution v with mean-value 0, i.e. $\sum_{i=1}^{n} d_i v(x_i) = 0$, then we can ignore λ_1. Then we may use the pseudo-inverse Λ^+ which we define as

$$\Lambda^+ := diag(\lambda_1^{-1} \equiv 0, \lambda_2^{-1}, \lambda_3^{-1}, ..., \lambda_n^{-1}).$$

Thus, our solution formula may be written as

$$v = -V\Lambda^{+}V^{T}\mathcal{L}_{\mathcal{V}}u. \tag{5}$$

For convenience, let us write the pseudo-inverse of $\mathcal{L}_{\mathcal{W}}$ as $M := V\Lambda^{+}V^{T}$, so

$$v = -M\mathcal{L}_{\mathcal{V}}u.$$

We also assume that the adversary should have some bound on the size of their weight matrix perturbation to remain inconspicuous. Suppose this bound is $\|\mathcal{V}\| \leq 1$. A reasonable goal for the adversary is to choose a perturbation within the aforementioned constraint which maximizes some norm of the perturbed solution $\|v\|$. Thus, the adversary may attempt to solve the adversarial learning problem shown below

$$\max_{\|\mathcal{V}\|\leq 1}\{\|v\|\}.$$

4.2 Gradient Calculation

We begin by defining the function f through which we will maximize $\|v\|$:

$$\begin{aligned} f(\mathcal{V}) :&= \frac{1}{2}\|v\|^2 \\ &= \frac{1}{2}\|-M\mathcal{L}_{\mathcal{V}}u\|^2 \\ &= \frac{1}{2}\|M\mathcal{L}_{\mathcal{V}}u\|^2 \end{aligned}$$

To perform the gradient calculation, it is easier to calculate $M\mathcal{L}_{\mathcal{V}}u$ defined by a single entry. First, we note that any entry of $\mathcal{L}_{\mathcal{V}}u$ can be defined as

$$[\mathcal{L}_{\mathcal{V}}u]_{ip} = \sum_{j=1}\mathcal{V}_{ij}(u_p(x_i) - u_p(x_j)) \tag{6}$$

where $u_p(x_i)$ represents the p-th entry of the row vector $u(x_i) \in \mathbb{R}^k$ for k classes and some $x_i \in \mathcal{X}$. Using (6), the individual entries of $M\mathcal{L}_{\mathcal{V}}u$ can be re-written as

$$[M\mathcal{L}_{\mathcal{V}}u]_{qp} = \sum_{i=1}^{n} M_{qi}[\mathcal{L}_{\mathcal{V}}u]_{ip} = \sum_{i=1}^{n} M_{qi} \sum_{j=1}^{n} \mathcal{V}_{ij}(u_p(x_i) - u_p(x_j)) \tag{7}$$

We can now use equation (7) to split apart our matrix norm into sums:

$$\begin{aligned} f(\mathcal{V}) &= \frac{1}{2}\|M\mathcal{L}_{\mathcal{V}}u\|^2 \\ &= \frac{1}{2}\sum_{q=1}^{n}\sum_{p=1}^{k}\left(\sum_{j=1}^{n} M_{qi}\sum_{j=1}^{n}\mathcal{V}_{ij}(u_p(x_i) - u_p(x_j))\right)^2 \end{aligned}$$

We proceed with our calculation of the matrix gradient.

$$\begin{aligned} \nabla f(\mathcal{V})_{\alpha\beta} &= \frac{\partial f}{\partial \mathcal{V}_{\alpha\beta}} \frac{1}{2}\sum_{q=1}^{n}\sum_{p=1}^{k}\left(\sum_{i=1}^{n}\sum_{j=1}^{n} M_{qi}\mathcal{V}_{ij}(u_p(x_i) - u_p(x_j))\right)^2 \\ &= \sum_{q=1}^{n}\sum_{p=1}^{k}\left(\sum_{i=1}^{n}\sum_{j=1}^{n} M_{qi}\mathcal{V}_{ij}(u_p(x_i) - u_p(x_j))\right)\frac{\partial f}{\partial \mathcal{V}_{\alpha\beta}}\left(\sum_{i'=1}^{n}\sum_{j'=1}^{n} M_{qi'}\mathcal{V}_{i'j'}(u_p(x_{i'}) - u_p(x_{j'}))\right) \\ &= \sum_{q=1}^{n}\sum_{p=1}^{k}\left(\sum_{i=1}^{n}\sum_{j=1}^{n} M_{qi}\mathcal{V}_{ij}(u_p(x_i) - u_p(x_j))\right)\left(\sum_{i'=1}^{n}\sum_{j'=1}^{n} M_{qi'}\delta_{i'=\alpha}\delta_{j'=\beta}(u_p(x_{i'}) - u_p(x_{j'}))\right) \\ &= \sum_{q=1}^{n}\sum_{p=1}^{k}\underbrace{\left(\sum_{i=1}^{n}\sum_{j=1}^{n} M_{qi}\mathcal{V}_{ij}(u_p(x_i) - u_p(x_j))\right)}_{[M\mathcal{L}_{\mathcal{V}}u]_{qp}} M_{q\alpha}(u_p(x_\alpha) - u_p(x_\beta)) \end{aligned}$$

In the above calculation, $\delta_{i'=\alpha}$ and $\delta_{j'=\beta}$ are defined as kronecker deltas for i' and j'. We continue by noticing (7) can simplify our equation further

$$\nabla f(\mathcal{V})_{\alpha\beta} = \sum_{q=1}^{n}\sum_{p=1}^{k}[M\mathcal{L}_{\mathcal{V}}u]_{qp}M_{q\alpha}(u_p(x_\alpha) - u_p(x_\beta))$$

Reordering the summations allows $M_{q\alpha}$ and $[M\mathcal{L}_{\mathcal{V}}u]_{qp}$ to be grouped within the summation as a matrix multiplication.

$$\begin{aligned}\nabla f(\mathcal{V})_{\alpha\beta} &= \sum_{p=1}^{k}\left(\sum_{q=1}^{n} M_{q\alpha}[M\mathcal{L}_{\mathcal{V}}u]_{qp}\right)(u_p(x_\alpha)-u_p(x_\beta)) \\ &= \sum_{p=1}^{k}\left(\sum_{q=1}^{n} M^T_{\alpha q}[M\mathcal{L}_{\mathcal{V}}u]_{qp}\right)(u_p(x_\alpha)-u_p(x_\beta))\end{aligned}$$

Subsequently, the summation within the parentheses may be reexpressed as the component-form $[M^TM\mathcal{L}_{\mathcal{V}}u]_{\alpha p}$, and we arrive at

$$\nabla f(\mathcal{V})_{\alpha\beta} = \sum_{p=1}^{k}[M^TM\mathcal{L}_{\mathcal{V}}u]_{\alpha p}(u_p(x_\alpha)-u_p(x_\beta))$$

As mentioned before, we can interpret $u(x_i) \in \mathbb{R}^k$ as a row vector for any i, and consequently $(u_p(x_\alpha)-u_p(x_\beta)) = [u(x_\alpha)-u(x_\beta)]_{1p}$.

$$\begin{aligned}\nabla f(\mathcal{V})_{\alpha\beta} &= \sum_{p=1}^{k}[M^TM\mathcal{L}_{\mathcal{V}}u]_{\alpha p}(u_p(x_\alpha)-u_p(x_\beta)) \\ &= \sum_{p=1}^{k}[M^TM\mathcal{L}_{\mathcal{V}}u]_{\alpha p}[u(x_\alpha)-u(x_\beta)]_{1p} \\ &= \sum_{p=1}^{k}\underbrace{[M^TM\mathcal{L}_{\mathcal{V}}u]_{\alpha p}}_{n\times k}\underbrace{[u(x_\alpha)-u(x_\beta)]^T_{p1}}_{k\times 1}\end{aligned}$$

If we fix $k=1$ so $u : \mathcal{X} \to \mathbb{R}$, we can vectorize the gradient in another way.

$$\nabla f(\mathcal{V})_{\alpha\beta} = [M^TM\mathcal{L}_{\mathcal{V}}u]_{\alpha,1}[u(x_\alpha)-u(x_\beta)]^T$$

The full gradient matrix can then be written as

$$\begin{aligned}\nabla f(\mathcal{V}) &= \underbrace{[M^T M\mathcal{L}_{\mathcal{V}}u]}_{n\times 1} \odot \underbrace{[\nabla u]}_{n\times n} \\ &= diag([M^T M\mathcal{L}_{\mathcal{V}}u]_1, [M^T M\mathcal{L}_{\mathcal{V}}u]_2, ..., [M^T M\mathcal{L}_{\mathcal{V}}u]_n)[\nabla u]\end{aligned}$$

where $\odot$ refers to multiplying $M^T M\mathcal{L}_{\mathcal{V}}u$ element-wise into each column and $[\nabla u]$ refers to the $n \times n$ graph gradient matrix.

If we allow p to vary, then we must sum over all p, i.e. all classes. Define u_p to be the pth-column of the solution matrix u, and ∇u_p to be the graph gradient of u_p, that is, $\nabla u_p(x, y) = u_p(y) - u_p(x)$. Denote $G = M^T M\mathcal{L}_{\mathcal{V}}u_p \in \mathbb{R}^n$. Then our gradient becomes

$$\nabla f(\mathcal{V}) = \sum_{p=1}^{k} diag(G(1), G(2), ..., G(n))[\nabla u_p]$$

For the sake of computations later on, we can slightly simplify $M^T M$ to be written in terms of V and Λ^+ as shown below:

$$\begin{aligned}M^T M &= (V\Lambda^+V^T)^T V\Lambda^+V^T \\ &= V(\Lambda^+)^T(V^T V)\Lambda^+V^T \\ &= V\Lambda^+\Lambda^+V^T \\ &= V(\Lambda^+)^2V^T\end{aligned}$$

where in the second equality, we use the fact that $V^T V = I$ since V is an orthogonal matrix.

4.3 Spectral Numerical Methds

Due to the computational intensity of calculating eigenvectors and eigenvalues for the sake of diagonalizing the pseudo-inverse of $\mathcal{L}_{\mathcal{W}}$, we will now also introduce spectral numerical methods to approximate the diagonalization. Given a spectral cutoff N, let V_N denote the $n \times N$ matrix consisting of the first N columns of V, i.e.

$$V_N = \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \cdots & v_N \\ | & | & & | \end{bmatrix}$$

where $N \ll n$. We also denote our approximate pseudoinverse diagonal eigenvalue matrix $\Lambda_N^+ \in \mathbb{R}^{N\times N}$ as

$$\Lambda_N^+ = \begin{bmatrix} 0 & & & \\ & \lambda_2^{-1} & & \\ & & \ddots & \\ & & & \lambda_N^{-1} \end{bmatrix}.$$

Then our pseudoinverse M may be rewritten to have the spectral approximation $V_N \Lambda_N^+ V_N^T$, and $M^T M = V_N (\Lambda_N^+)^2 V_N^T$.

Although the spectral approximation does not provide an exact solution to equation (5), it does describe the smoothly-varying portion of the solution because the error is confined to the higher frequencies within the spectrum of the solution. To demonstrate this, we rewrite the solution formulas

$$v = \lambda_2^{-1} v_2 v_2^T F + \lambda_3^{-1} v_3 v_3^T F + \dots \lambda_n^{-1} v_n v_n^T F$$

and

$$v_N = \lambda_2^{-1} v_2 v_2^T F + \lambda_3^{-1} v_3 v_3^T F + \dots \lambda_N^{-1} v_N v_N^T F$$

where $F := \mathcal{L}_{\mathcal{V}} u$. Then the difference between the two solutions becomes

$$v - v_N = \lambda_{N+1}^{-1} v_{N+1} v_{N+1}^T F + \lambda_{N+2}^{-1} v_{N+2} v_{N+2}^T F + \ldots + \lambda_n^{-1} v_n v_n^T F$$

and we can see that the error can be found within the higher frequencies exclusively after the spectral cutoff. Hence,

$$\|v - v_N\| = \lambda_{N+1}^{-1}\|v_{N+1}v_{N+1}^T F\| + \lambda_{N+2}^{-1}\|v_{N+2}v_{N+2}^T F\| + \ldots + \lambda_n^{-1}\|v_n v_n^T F\|.$$

In general, we expect the eigenvalues to grow rapidly as $i \to n$, so the error correspondingly fades quickly. However, we must also consider that the number of eigenvalues/eigenvectors kept corresponds to the number of clusters identified in the graph, so a small spectral cutoff will allow us to filter out noisy clusters identified in the higher frequency spectrum.

4.4 Gradient Ascent Algorithm

We now present the pseudocode for the GradientAscent algorithm, as well as the procedure AddEdges which adds m new edges to the weight matrix.

The procedure GradientAscent requires these arguments:

- $\mathcal{W}$: the dataset's weight matrix
- $\mathcal{T}$: the number of iterations to run gradient descent
- N: the spectral cutoff for eigenvectors and eigenvalues
- $\|\cdot\|$: some matrix norm by which to normalize the perturbation matrix during each iteration of gradient ascent

Before performing gradient ascent, we begin by initializing the perturbation matrix to have all elements equal to 0.5 and dimension equal to that of the original weight matrix. GradientAscent then computes $\mathcal{T}$ iterations of gradient ascent while normalizing by $\|\mathcal{V}\|$ between each iteration. Finally, if desired, an optional argument of m new edges can be added. Note that AddEdges will constrain the edges of $\mathcal{V}$ to corresponding nonzero edges in $\mathcal{W}$ before adding new edges. If AddEdges is not called, then we must perform this restriction of perturbation edges to those of corresponding nonzero edges in $\mathcal{W}$ within the GradientAscent procedure instead. We can formalize this constraint as $\mathcal{W}_{ij} = 0 \implies \mathcal{V}_{ij} = 0$. The final perturbation matrix is then symmetrized by taking the average of $\mathcal{V}$ and $\mathcal{V}^T$.

Algorithm 1 Gradient Ascent Algorithm

```
1: procedure GradientAscent($\mathcal{W}$, $\mathcal{T}$, $N$, $\|\cdot\|$, $m$: optional)
2:     $\mathcal{V} \leftarrow 0.5\times$ones$(n, n)$
3:     for $t = 0$ to $\mathcal{T}$ do
4:         $\nabla f(\mathcal{V}) \leftarrow (V_N(\Lambda_N^+)^2 V_N^T \mathcal{L}_{\mathcal{V}} u) \odot [\nabla u]$
5:         $\mathcal{V} \leftarrow \mathcal{V} + dt \cdot \nabla f(\mathcal{V})$
6:         if $\|\mathcal{V}\| > 1$ then
7:             $\mathcal{V} \leftarrow \mathcal{V}/\|\mathcal{V}\|$                ▷ Normalize $\mathcal{V}$ by $\|\cdot\|$
8:         end if
9:         $\mathcal{V} \leftarrow (\mathcal{V} + \mathcal{V}^T)/2$                ▷ symmetrize $\mathcal{V}$
10:    end for
11:    $\mathcal{V} \leftarrow$ AddEdges$(\mathcal{W}, \mathcal{V}, m)$
12:    if AddEdges not called then
13:        for all $(x, y)$ such that $\mathcal{W}_{xy} = 0$ do
14:            $\mathcal{V}_{xy} \leftarrow 0$
15:        end for
16:    end if
17:    return $(\mathcal{V} + \mathcal{V}^T)/2$
18: end procedure
```

We also define a function AddEdges (2) for which we additionally perturb the weight matrix $\mathcal{W}$ by adding m new edges to $\mathcal{W}$. AddEdges adds the m edges which have gradient greatest in magnitude to the perturbation matrix $\mathcal{V}$ while discarding all other edges in $\mathcal{V}$ which are 0 in $\mathcal{W}$. Here, we use Ξ to refer to all nonzero edges in $\mathcal{W}$ and Δ to refer to the set of edges added to the perturbation matrix.

5 Experiments

Equipped with the functions GradientAscent and AddEdges, we tested our adversarial perturbation matrix vs a random perturbation matrix with nonnegative random values between 0 and 1, in the Poisson learning setting. A range of ϵ values were selected to compare the accuracies of Poisson learning with our adversarially perturbed weight matrix $\mathcal{W} + \epsilon\mathcal{V}_{\text{adversary}}$ against the average

Algorithm 2 Add New Edges to the Perturbation Matrix

```
1: procedure AddEdges($\mathcal{W}$, $\mathcal{V}$, $m$)
2:     $\mathcal{V}' \leftarrow$ copy($\mathcal{V}$)
3:     for all $(x, y)$ such that $\mathcal{W}_{xy} = 0$ do
4:         $\mathcal{V}'_{xy} \leftarrow 0$
5:     end for
6:     $\Xi \leftarrow \{(x, y) : \mathcal{W}_{xy} \neq 0\}$
7:     $\Delta \leftarrow \emptyset$
8:     for $p = 0$ to $m$ do
9:         $\beta \leftarrow \text{argmax}_{(x,y)\in(\mathcal{X}\times\mathcal{X})\setminus(\Xi\cup\Delta)}\{|\mathcal{V}_{xy}|\}$
10:        $\mathcal{V}'_{\beta} \leftarrow \mathcal{V}_{\beta}$
11:        $\Delta \leftarrow \Delta \cup \beta$
12:    end for
13:    return $\mathcal{V}'$
14: end procedure
```

of 5 randomly perturbed matrices $\mathcal{W} + \epsilon\mathcal{V}_{\text{random}}$ for each value of ϵ.

We tested Poisson learning with unperturbed, adversarially perturbed, and randomly perturbed weight matrices on three different datasets: MNIST, FashionMNIST, Cifar-10 loaded from the `graphlearning` python package available here: `https://github.com/jwcalder/GraphLearning` [1]. To construct the graph structure of each dataset while extracting the data's important features, the MNIST and FashionMNIST datasets were fed through variational autoencoders, while Cifar-10 utilized the AutoEncodingTransformations architecture from [6]. The final graph for each dataset was constructed as a 10-nearest neighbor graph with Gaussian weights over a 500 sample subset of the datasets with 2 classes. For further detail on the graph construction, this paper uses the same construction process as outlined in [2].

Four trials were completed for each dataset over various epsilon values and were conducted with these variables:

- $\mathcal{T} = 50$ iterations
- x we truncated negative edge weights to 10^{-10} in the resulting weight matrix
- $n = 500$ samples (number of samples per class may differ)

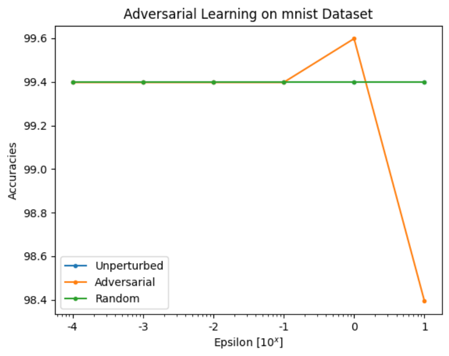
- $k = 2$ classes
- $\gamma = 1$ label per class
- $\|\cdot\| =$ the *matrix infinity norm* $\|\cdot\|_\infty$ defined as $\|\cdot\|_\infty = \max_{x,y\in\mathcal{X}}\{|\mathcal{V}_{xy}|\}$

All experiments were conducted for 50 iterations with a spectral cutoff of 10. After adding the perturbation matrix (adversarial or random) scaled by ϵ, any negative edge weights were truncated to 10^{-10} due to our assumption that the edge weights must be non-negative to be a valid weight matrix (it would be obvious that an adversary perturbed the weight matrix if it did not follow its own constraints). The choice of the matrix infinity norm $\|\cdot\|_\infty$ was to account for the large size of the weight matrices. If we recall that we perform gradient ascent under the constraint that $\|\mathcal{V}\| \leq 1$, other norm choices such as the euclidean norm $\|\cdot\|_2$ become too restrictive because the normalization factor explodes. As a result, the values within the perturbation matrix $\mathcal{V}$ may shrink to levels that are not consequential to our calculation. We should note that our original perturbed Poisson equation (3) is an approximation, so miniscule perturbations across the entirety of the weight matrix may not even be completely accurate to our original solution to the perturbed Poisson equation.

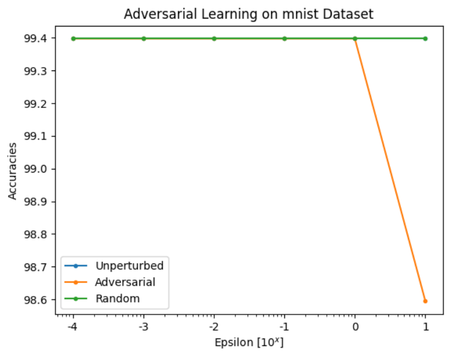
5.1 Results for Adversarial Poisson Learning

Our first round of simulations solely determined the perturbation matrix through gradient ascent without adding new edges. The adversarial and random perturbation matrices within each trial were scaled by $\epsilon \in \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 10^{0}, 10^{1}\}$ and then added to the weight matrix. These results are displayed in figures 1, 2, and 3.

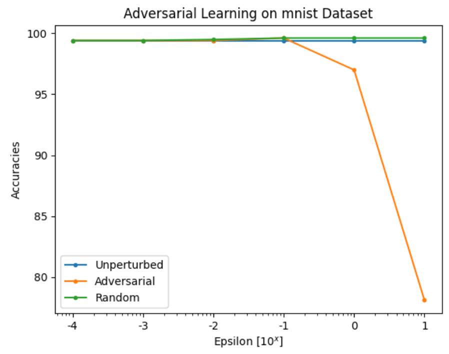
Experiments ran on the FashionMNIST and Cifar-10 datasets consistently saw small drops (less than two percent) in accuracies by $\epsilon = 10^{-1}$, while $\epsilon > 10^{-1}$ demonstrated significantly greater accuracy degradation. A few Cifar-10 trials, namely those in figures 3b and 3c, suffered accuracy decreases of more than five percent by the time ϵ reached 10^{-1}. Interestingly, the trials ran on the MNIST dataset had slightly more sporadic changes in accuracy up until $\epsilon = 10^{1}$, in which three of the four trials decreased more than 0.8 percent in performance.



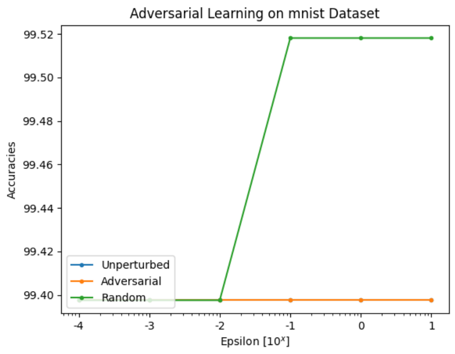
(a) MNIST trial 1. $\|\mathcal{V}\|_\infty = 0.1435$



(b) MNIST trial 2. $\|\mathcal{V}\|_\infty = 0.1303$

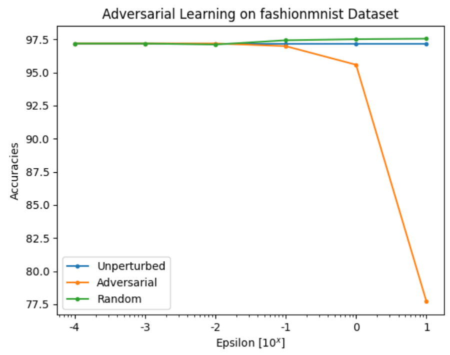


(c) MNIST trial 3. $\|\mathcal{V}\|_\infty = 0.2286$

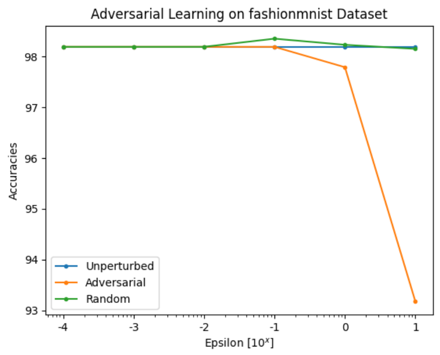


(d) MNIST trial 4. $\|\mathcal{V}\|_\infty = 9.256 \times 10^{-15}$

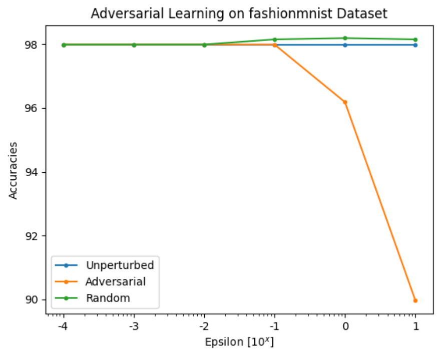
Figure 1: Accuracy of Poisson learning on unperturbed, adversarially perturbed, and randomly perturbed weight matrices over the MNIST dataset with 500 samples (229 in the "0" class, and 271 in the "1" class). $\|\mathcal{V}\|_\infty$ is provided for each trial to assist in understanding the scale of the perturbation. Note that $\|\mathcal{W}\|_\infty = 1$ for all trials.



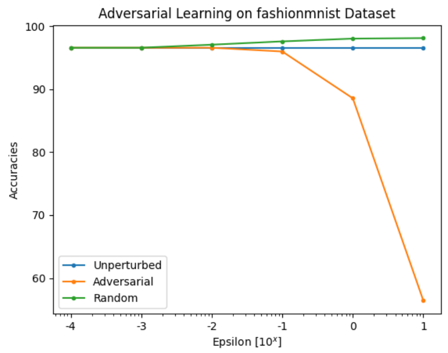
(a) FashionMNIST trial 1. $\|\mathcal{V}\|_\infty = 0.1355$



(b) FashionMNIST trial 2. $\|\mathcal{V}\|_\infty = 0.1098$

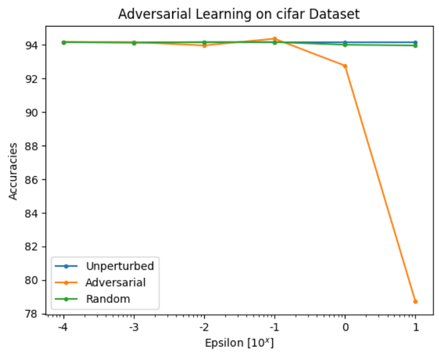


(c) FashionMNIST trial 3. $\|\mathcal{V}\|_\infty = 0.2060$

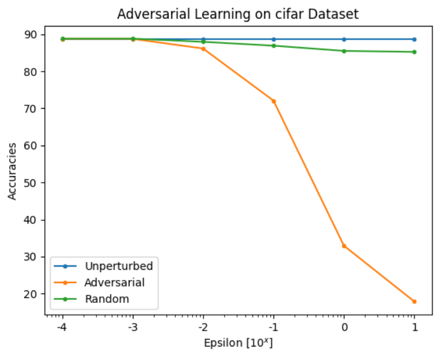


(d) FashionMNIST trial 4. $\|\mathcal{V}\|_\infty = 0.2258$

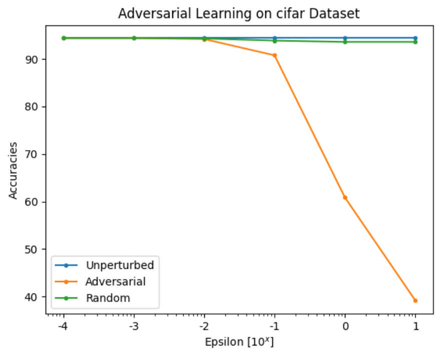
Figure 2: Accuracy of Poisson learning on unperturbed, adversarially perturbed, and randomly perturbed weight matrices over the FashionMNIST dataset with 500 samples (258 in the ”0” class, and 242 in the ”1” class). $\|\mathcal{V}\|_\infty$ is provided for each trial to assist in understanding the scale of the perturbation. Note that $\|\mathcal{W}\|_\infty = 1$ for all trials.



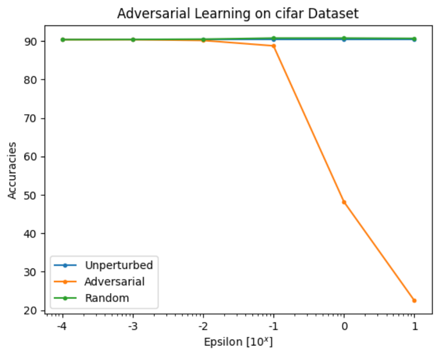
(a) Cifar-10 trial 1. $\|\mathcal{V}\|_\infty = 0.2274$



(b) Cifar-10 trial 2. $\|\mathcal{V}\|_\infty = 0.2243$



(c) Cifar-10 trial 3. $\|\mathcal{V}\|_\infty = 0.4814$



(d) Cifar-10 trial 4. $\|\mathcal{V}\|_\infty = 0.2442$

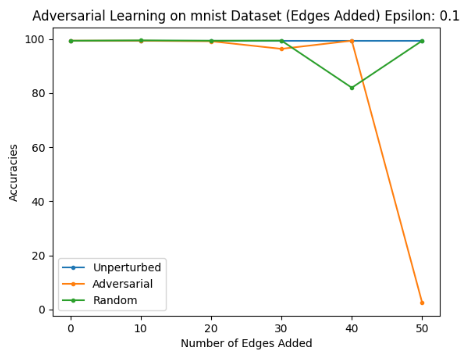
Figure 3: Accuracy of Poisson learning on unperturbed, adversarially perturbed, and randomly perturbed weight matrices over the Cifar-10 dataset with 500 samples (256 in the ”0” class, and 244 in the ”1” class). $\|\mathcal{V}\|_\infty$ is provided for each trial to assist in understanding the scale of the perturbation. Note that $\|\mathcal{W}\|_\infty = 1$ for all trials.

5.2 Results for Adversarial Poisson Learning with Added Edges

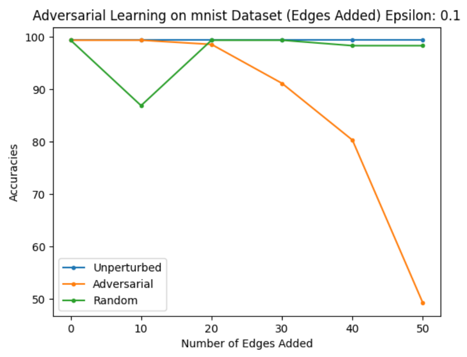
A second round of experiments tested the effect of adding an increasing number of new edges to the weight matrix in addition to the perturbation matrix. Figures 4, 5, and 6 show the results of adding $m \in \{0, 10, 20, 30, 40, 50\}$ new edges to the weight matrices. During these trials, the scaling factor ϵ was fixed to 10^{-1} because it consistently was the first to cause noticeable changes in accuracy.

We can observe that adding previously non-existent edges almost always led to monotonically decreasing accuracies of which the only exceptions were MNIST trials 4a and 4c. For all three datasets, adding 20 edges provided a good chance of lowering accuracy compared to the unperturbed model, while 30 or more edges almost guaranteed a significant hit to accuracy.

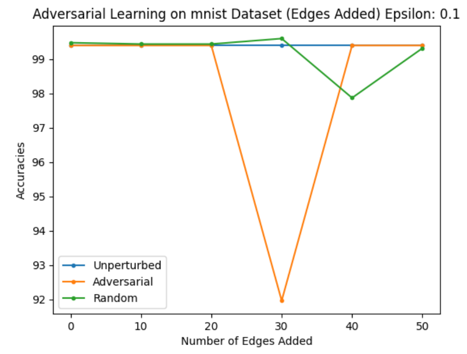
We also listed the rate at which new edges were formed between two different classes. We predicted that creating edges between different classes would allow random walkers starting at a certain labeled point to walk to differently-labeled points more readily. This would cause ccuracy degradation as there are more inter-class connections within the graph, allowing classes to "communicate" more strongly which makes them less separate. We would expect our algorithm to add edges along where the gradient is greatest, which would be between samples of different classes. Our experiments validate these predictions as they largely demonstrate a decrease in Poisson learning's performance coupled with edges inserted between exclusively different classes except in trial 6c.



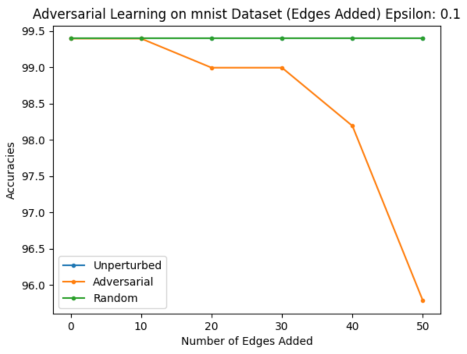
(a) MNIST trial 1. Rate of new edges between different classes: 100%



(b) MNIST trial 2. Rate of new edges between different classes: 100%

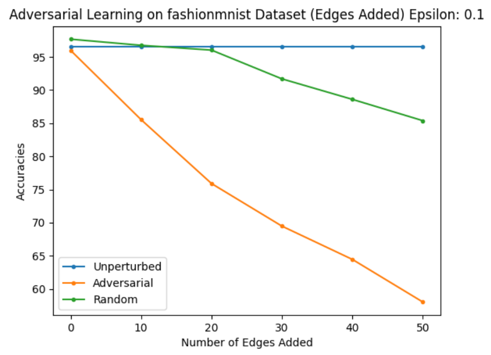


(c) MNIST trial 3. Rate of new edges between different classes: 100%

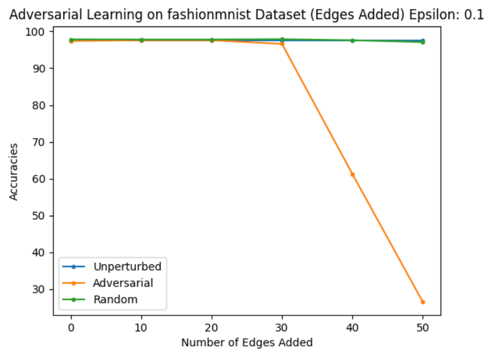


(d) MNIST trial 4. Rate of new edges between different classes: 100%

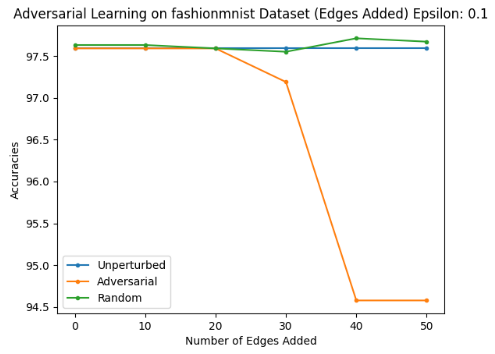
Figure 4: Accuracy of Poisson learning on unperturbed, adversarially perturbed, and randomly perturbed weight matrices over the MNIST dataset with 500 samples (229 in the "0" class, and 271 in the "1" class) while adding additional edges. Note that $\|\mathcal{W}\|_\infty = 1$ for all trials.



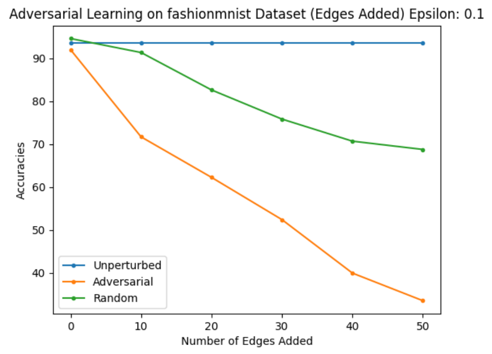
(a) FashionMNIST trial 1. Rate of new edges between different classes: 100%



(b) FashionMNIST trial 2. Rate of new edges between different classes: 100%

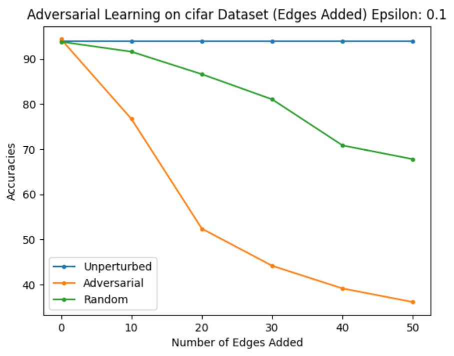


(c) FashionMNIST trial 3. Rate of new edges between different classes: 100%

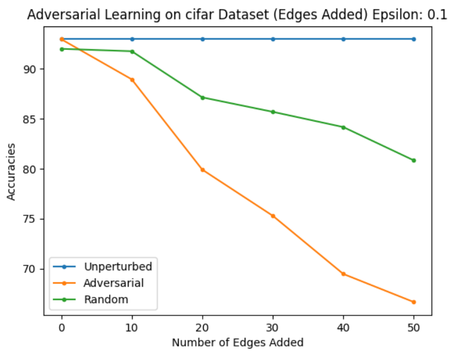


(d) FashionMNIST trial 4. Rate of new edges between different classes: 100%

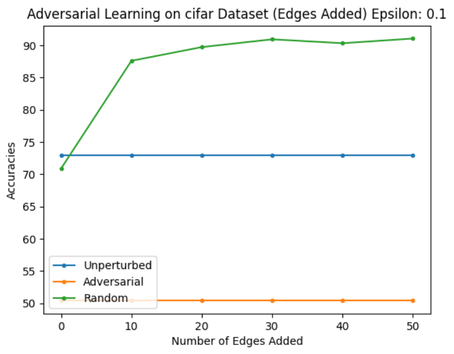
Figure 5: Accuracy of Poisson learning on unperturbed, adversarially perturbed, and randomly perturbed weight matrices over the FashionMNIST dataset with 500 samples (258 in the ”0” class, and 242 in the ”1” class) while adding additional edges. Note that $\|\mathcal{W}\|_\infty = 1$ for all trials.



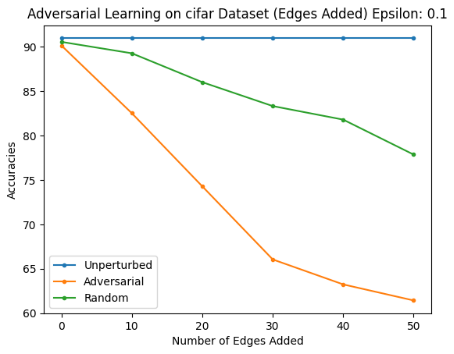
(a) Cifar-10 trial 1. Rate of new edges between different classes: 100%



(b) Cifar-10 trial 2. Rate of new edges between different classes: 100%



(c) Cifar-10 trial 3. Rate of new edges between different classes: 0.0%



(d) Cifar-10 trial 4. Rate of new edges between different classes: 100%

Figure 6: Accuracy of Poisson learning on unperturbed, adversarially perturbed, and randomly perturbed weight matrices over the Cifar-10 dataset with 500 samples (256 in the ”0” class, and 244 in the ”1” class) while adding additional edges. Note that $\|\mathcal{W}\|_\infty = 1$ for all trials.

6 Conclusion

We introduced the abstract mathematical conceptualization of machine learning and discussed one of its subsets, namely semi-supervised graph-based learning. We then investigated the advantages and disadvantages of a popular graph-based learning method called Laplacian learning. The disadvantages of Laplacian learning and its variants led us to the introduction of Poisson learning, a novel graph-based learning framework which is highly successful at low-label rates.

To test the ability of Poisson learning to withstand one kind of adversarial attack to the dataset's weight matrix, we approximately solved the perturbed Poisson equation and determined a gradient ascent scheme with respect to the perturbation matrix in order to maximize the norm of the perturbed solution to the Poisson equation. The gradient ascent scheme was developed with a small spectral cutoff to prevent intractable computational complexity.

Finally, we conducted simulations on several subsets of common datasets with adversarial perturbation matrices and compared its effectiveness against random perturbation matrices. Experiments demonstrate that adding the perturbation matrix and inserting new edges can consistently degrade the performance of Poisson learning under conditions assumed during simulations.

References

[1] Calder, J. Graphlearning python package, Jan. 2022.

[2] Calder, J., Cook, B., Thorpe, M., and Slepcev, D. Poisson learning: Graph based semi-supervised learning at very low label rates. In *Proceedings of the 37th International Conference on Machine Learning* (13–18 Jul 2020), H. D. III and A. Singh, Eds., vol. 119 of *Proceedings of Machine Learning Research*, PMLR, pp. 1306–1316.

[3] Chapelle, O., Scholkopf, B., and Zien, Eds., A. Semi-supervised learning (chapelle, o. et al., eds.; 2006) [book reviews]. *IEEE Transactions on Neural Networks 20*, 3 (2009), 542–542.

[4] El Alaoui, A., Cheng, X., Ramdas, A., Wainwright, M. J., and Jordan, M. I. Asymptotic behavior of\ell_p-based laplacian regularization in semi-supervised learning. In *Conference on Learning Theory* (2016), PMLR, pp. 879–906.

[5] Kyng, R., Rao, A., Sachdeva, S., and Spielman, D. A. Algorithms for lipschitz learning on graphs. In *Conference on Learning Theory* (2015), PMLR, pp. 1190–1223.

[6] Zhang, L., Qi, G.-J., Wang, L., and Luo, J. Aet vs. aed: Unsupervised representation learning by auto-encoding transformations rather than data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2019), pp. 2547–2555.

[7] Zhou, D., and Schölkopf, B. Regularization on discrete spaces. In *Joint Pattern Recognition Symposium* (2005), Springer, pp. 361–368.

[8] Zhu, X., Ghahramani, Z., and Lafferty, J. D. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International conference on Machine learning (ICML-03)* (2003), pp. 912–919.