Deep Label Propagation for Semi-Supervised Learning

Anonymous Author(s)
Affiliation
Address
email

Abstract

The goal in graph-based semi-supervised learning is to effectively utilize the graph for producing accurate predictions. One way to do this is by encouraging smoothness across edges using a (weighted) Laplacian regularizer. This idea is prevalent in classic propagation algorithms as well as in current deep graph networks. Although the success of these methods crucially depends on the values of weights, learning is focused on the predictive model, and weights are typically set using local heuristics.

In this work we present a method for jointly learning both the weights and the predictive model in a discriminative fashion. We do this by casting the original Label Propagation algorithm as a deep neural network whose layers correspond to the unrolling of the algorithm's iterations. We then propose several novel neural components that provide a powerful generalization of the original algorithm. The model has very few parameters, and can be trained by applying gradient descent to an appropriate loss function. Experimental results in two settings demonstrate the utility of our approach.

1 Introduction

We study the problem of graph-based semi-supervised learning (SSL), where the goal is to correctly label all nodes of a graph given the labels of only a small subset of them. Methods for this problem are often based on assumptions regarding the relation between the graph and the predicted labels. One such assumption is *smoothness*, which states that adjacent nodes are likely to have similar labels. Smoothness can be encouraged by optimizing an objective where a loss term $\mathcal{L}$ over the labeled nodes is augmented with a graph-based quadratic penalty over all adjacent nodes:

$$\min_f \mathcal{L}(y_S, f_S) + \lambda \sum_{(i,j)\in E} w_{ij}(f_i - f_j)^2 \tag{1}$$

Here, f are the predictions, y are the true labels, S is the set of labeled nodes, and w are non-negative edge weights. The quadratic term in Eq. (1) is often referred to as Laplacian Regularization since (for symmetric weights) it can equivalently be expressed using the graph Laplacian [4].

Many SSL methods follow the general form of Eq. (1) [36, 35, 2, 5, 1, 28, 32]. Classic algorithms like the seminal Label Propagation algorithm [36] are simple, efficient, and theoretically grounded, but are limited in two important ways. First, extensive work has shown that good weights are crucial for success [36, 15, 31, 2, 16], but these are often set heuristically. Second, predictions are treated simply as optimizable variables, rather than as the output of a learned parametric classifier. Some works address the first point by proposing disciplined ways for learning w. These, however, either consider weights disjointly from predictions [31, 21] or assume specific parameterizations [34, 16]. Recent deep methods for SSL address the second point by offering intricate predictive models that are trained

Submitted to 32nd Conference on Neural Information Processing Systems (NIPS 2018). Do not distribute.

discriminatively [32, 24, 33, 19, 11, 12, 23]. These, however, often suffer from over-parameterization, do not explicitly utilize label information in the model, and still require w as input.

In this work we propose a framework for jointly learning both a parametric predictive model *and* its corresponding edge weights. We begin by showing that the original Label Propagation algorithm can be cast as a differentiable neural network $H(w; S)$ whose layers correspond to the "unrolling" of the algorithm's iterative updates. We then propose an appropriate loss function that allows for discriminatively training the network using standard gradient methods. The key modeling point here is that labeled nodes serve as input to both the loss *and* the network. This lets the network's hidden layers directly operate on predictions (rather than manipulate feature representations indirectly). Overall, our framework can be viewed as optimizing the weights of an approximate solution to the quadratic objective.

Building on this, we continue by introducing several novel non-linear components that can be incorporated in H. These include a non-linear parameterization of w, an information-theoretic attention-like mechanism, and a "bifurcation" operator for controlling the rate of label convergence. Each component is explicitly designed to operate on label distributions. Our final model offers a slim design that is tailored for semi-supervised tasks; at full capacity, it requires only a handful of parameters, and only a single hyper-parameter (depth). Together, these provide a powerful generalization of the original propagation algorithm that can be trained efficiently. Experiments on benchmark data in two distinct learning settings show that our model compares favorably against state-of-the-art baselines.

1.1 Related Work

Many SSL methods are based on Eq. (1) or on similar quadratic forms. These differ in their assumed input, the optimization objective, and the parametric form of predictions.

Classic methods such as Label Propagation [36] assume no parametric form for predictions, and require edge weights as inputs. When node features are available, weights are often set heuristically according to some feature similarity measure (e.g., $w_{ij} = \exp(\|x_i - x_j\|_2^2/\sigma^2)$). This makes the graph a discrete approximation of the density manifold. Label Propagation constrains predictions on S to agree with their true labels. Other propagation methods relax this assumption [2, 5], add regularization terms [1], allow for label uncertainty [28], or use other Laplacian forms [35].

Some methods aim to learn weights for Eq. (1), but do not directly optimize for accuracy. Instead, they either model the relations between the graph and features [31, 21] or simply require f as input [8, 14, 9]. Other methods focus on accuracy, but are constrained to specific parameterizations (e.g., weighted RBF) or assumptions (e.g., local similarity) [16]. Parameters are usually set heuristically [36]. For the weighted RBF parameterization, this can be done by optimizing the leave-one-out loss, but requries a series of costly matrix inversions [34].

Several of the recent works on deep networks for graph inputs have been devoted to SSL. The common theme is that a (weighted) graph is used to create meaningful vector representations of nodes, which are then fed into a classifier. When the input includes only the graph, features are generated using embedding techniques [24, 29, 11, 12]. When node features are available, representations are created by either convolving features over graph edges [33, 19, 23] or using attention [30]. These methods use the graph to learn highly-parametrized classifiers. However, in contrast to propagation algorithms, labeled information is used implicitly through the loss, rather than explicitly as input to the classifier.

1.2 Preliminaries

We begin by describing the semi-supervised learning setup and introducing some notations. The input includes a graph $G = (V, E)$ for which a subset of nodes $S \subset V$ are labeled by $y_S = \{y_i\}_{i \in S}$. Our method supports directed graphs, and we denote incoming neighbors as $N_i = \{j : (j, i) \in E\}$. We refer to S as the "seed" set, and denote the unlabeled nodes by $U = V \setminus S$. We use $n = |V|, m = |E|, \ell = |S|$, and $u = |U|$ so that $n = \ell + u$. In a typical task, we expect ℓ to be much smaller than n. Here we focus on the transductive learning setting where the goal is to predict the labels of the unlabeled examples in U, but note that generalizing across graphs is possible for some of the settings we consider. In same cases, the input may also include features for all nodes $x = \{x_i\}_{i \in V}$.

Importantly, we do *not* assume the input includes edge weights $w = \{w_e\}_{e \in E}$. Rather, our goal is to *learn* them in a way which maximizes predictive accuracy. We denote by W the weighted adjacency matrix of w, and use $\tilde{W}$ and $\tilde{w}$ for the respective (row)-normalized weights.

We consider a multiclass setting with labels $y \in \{1, \ldots, C\}$.[1] Most methods (as well as ours) output "soft" labels $f_i \in \Delta_C$, where Δ_C is the C-dimensional simplex. For convenience we treat "hard" labels y_i as indicator vectors in Δ_C. All predictions are encoded as a matrix f with entries $f_{ic} = \mathbb{P}[y_i = c]$. For any matrix M, we will use M_A to denote the sub-matrix with rows corresponding to A. Under this notation, given G, S, y_S, and possibly x, our goal is to generate predictions f_U that match y_U.

2 Label Propagation as a deep neural network

Many semi-supervised methods are based on the notion that predictions should be smooth across edges. A popular way to encourage such smoothness is to optimize a (weighted) quadratic objective. Intuitively, the objective encourages the predictions of all adjacent nodes to be similar. There are many variations on this idea; here we adopt the formulation of [36] where predictions are set to minimize a quadratic term subject to an agreement constraint on the labeled nodes:

$$f^*(w; S) = \operatorname*{argmin}_{f : f_S = y_S} \sum_{(i,j) \in E} w_{ij} \|f_i - f_j\|_2^2 \tag{2}$$

In typical applications, w is assumed to be given as input. In contrast, our goal here is to *learn* them in a discriminative manner. A naïve approach would be to directly minimize the empirical loss. For a loss function L, regularization term R, and regularization constant λ, the objective would be:

$$\min_w \frac{1}{\ell} \sum_{i \in S} L\left(y_i, f_i^*(w; S)\right) + \lambda R(w) \tag{3}$$

While appealing, this approach introduces two main difficulties. First, f^* is in itself the solution to an optimization problem (Eq. (2)), and so optimizing Eq. (3) is not straightforward. Second, the constraints in Eq. (2) ensure that $f_i^* = y_i$ for every $i \in S$, making the loss term in Eq. (3) redundant. While some methods solve this by replacing these with weak constraints, a third issue is that it is still not clear why optimizing edge weights for y_S should generalize well to the unlabeled nodes.

In what follows, we describe how to overcome these issues. We begin by showing that a simple algorithm for approximating f^* can be cast as a deep neural network. Under this view, the weights (as well as the algorithm itself) can be parametrized and optimized using gradient methods. We then propose a loss function suited to SSL, and show how the above network can be trained efficiently.

2.1 Label Propagation

Recall that we would like to learn $f^*(w; S)$. When w is symmetric, the objective in Eq. (2) is convex and has a closed form solution. This solution, however, requires an inversion of a large matrix, which can be costly, does not preserve sparsity, and is non-trivial to optimize. The Label Propagation algorithm [36] circumvents this issue by approximating f^* using simple iterative averaging. Let $f^{(t)}$ be the set of soft labels at iteration t and $\tilde{w}_{ij} = \sum_j w_{ij}$, then for the following recursive relation:

$$f_i^{(t+1)} = \sum_{j \in N_i} \tilde{w}_{ij} f_j^{(t)} \qquad \forall i \in U \tag{4}$$

it holds that $\lim_{t \to \infty} f^{(t)} = f^*$ for any initial $f^{(0)}$ [36]. In practice, the iterative algorithm is run up to some iteration T, and predictions are given using $f^{(T)}$. This dynamic process can be thought of as labels propagating over the graph from labeled to unlabeled nodes.

Motivated by the above, the idea behind our method is to directly learn weights for $f^{(T)}$, rather than for f^*. In other words, instead of optimizing the quadratic solution, our goal is to learn weights under which Label Propagation preforms well. This is achieved by first designing a neural architecture whose layers correspond to an "unrolling" of the iterative updates in Eq. (4), and then training this model with an appropriate loss function. We elaborate on this construction in the next section.

[1] Note that other setting such as regression are also possible (see [2]).

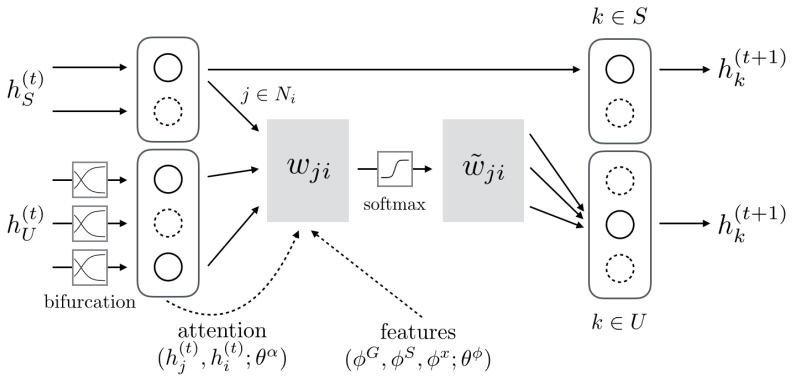

Figure 1: The label-propagation layer

2.2 Architecture

We begin by describing the basic *label-propagation layer* which is used as the main building block of our model. The basic layer takes in two main inputs: a set of (predicted) soft labels $h = \{h_i\}_{i=1}^n$ for all nodes, and the set of true labels y_A for some $A \subseteq S$. For clarity we use $A = S$ throughout this section. As output, the layer produces a new set of soft labels $h' = \{h'_i\}_{i=1}^n$ for all nodes. Note that both h_i and h'_i are in Δ_C. Hence, rather than mapping vector representations of nodes, layers can be thought of as modifying the set of soft predictions. The layer's functional form borrows from the Label Propagation update rule in Eq. (4) where unlabeled nodes are assigned the weighted-average values of their neighbors, and labeled nodes are fixed to their true labels. For a given w, the output is:

$$h'_U = \tilde{W}_U h, \qquad h'_S = y_S \tag{5}$$

where $\tilde{W}$ is the row-normalized matrix corresponding to w.

The full network is obtained by composing T identical layers with shared weights w:

$$H(w; S) = h^{(T)} \circ h^{(T-1)} \circ \cdots \circ h^{(0)} \tag{6}$$

The input layer $h^{(0)}$ is initialized to y_i for each $i \in S$ and to some prior ρ_i (e.g., uniform) for each $i \in U$. Since each layer $h^{(t)}$ corresponds to a single iterative update, the full network H can be thought of as an unrolling of the full Label Propagation algorithm. We will therefore use the terms "depth", "time", and "iterations" interchangeably. The the network depth T is the model's only hyper-parameter. Since weights are shared (and due to cleverly implemented control-flow constructs), fairly deep models (e.g., with $T = 100$) can still be trained very efficiently. As we later discuss, parameterizing the weights further reduces the computational burden.

3 Generalizing Label Propagation

So far we have shown how the Label Propagation algorithm can be cast as a neural network whose weights can be learned. The current network, however, has two limitations. First, a full parameterization of H by w is likely to cause overfitting, especially when ℓ is small and m is large. Second, the expressive power of H is limited since non-linearity is used only for normalizing weights.

In light of this, in this section we introduce several non-linear components which together generalize the basic propagation layer in Eq. (5). With these, an unrolling of the network can now be seen as running a generalized Label Propagation algorithm with powerful non-linear updates. The generalized layer replaces both weights and inputs with *functions*, and the general form becomes:

$$h^{(t+1)} = \tilde{A}(h^{(t)}, \phi; \theta_1)\mu(h^{(t)}, t; \theta_2) \tag{7}$$

Here, $\phi \in \mathbb{R}^{m \times d}$ are edge features, $\tilde{A}(\cdot)$ returns a row-normalized weight matrix (thus replacing $\tilde{W}$), $\mu(\cdot)$ returns a soft-label input matrix (thus replacing $h^{(t)}$), and $\theta = [\theta_1, \theta_2]$ are the learned parameters. We refer to the corresponding network as $H(\theta; S)$.

The edge-weight function $\tilde{A}$ has two distinct roles. As a function of ϕ, it allows for parameterizing edge weights using features. As a function of $h^{(t)}$, it offers an attention-like mechanism that dynamically allocates weights according the 'states' of a node and its neighbors. While these can operate together, for clarity we describe them separately. The labeling function μ is a time-dependent bifurcation mechanism which dynamically controls the rate of label convergence. We next describe both $\tilde{A}$ and μ in detail.

3.1 Weight parameterization

Although possible, parameterizing H directly by w will likely lead to overfitting. Intuitively, this is because the loss in Sec. 4 can be determined only by a small subset of edges, such as those on paths between labeled nodes. Instead, we further parametrize weights as a function of edge features $\phi_{ij} \in \mathbb{R}^d$ and parameters $\theta^\phi \in \mathbb{R}^d$ by:

$$w_{ij} = \langle \theta^\phi, \phi_{ij} \rangle \tag{8}$$

Normalization is then applied using a softmax function over incoming neighbors:

$$\tilde{A}_{ij} = \frac{\exp(w_{ij})}{\sum_{k \in N_i} \exp(w_{kj})} = \mathrm{softmax}_i(w_{ij}) \tag{9}$$

When the data includes "raw" node features $\{x_i\}_{i=1}^n, x_i \in \mathbb{R}^D$, the standard approach is to define ϕ_{ij} using x_i and x_j (and set $d = D$). For example, setting $(\phi_{ij})_l = ((x_i)_l - (x_j)_l)^2$ for $l = 1, \ldots, d$ recovers the popular weighted RBF [36]. This, however, can still overfit when D is large, does not generalize across datasets with different features, and is impossible when node features are not available. As in alternative, we opt for a compact approach and propose to instead use a small set of carefully designed edge features. In addition to incorporating raw features (when available), these also utilize the graph structure and the seed set S. We use three main types of features:

Graph features (ϕ^G): graph-related properties such as node attributes (e.g., source-node degree), edge centrality measures (e.g., edge betweenness),path-ensemble features (e.g., Katz distance),and graph-partitions (e.g., k-cores).These allow generalization across nodes based on their local attributes and global roles in the graph.

Seed features (ϕ^S): relations between the edge (i, j) and nodes in S, such the minimal (unweighted) distance to i from some $s \in S$. Since closer nodes are more likely to be labeled correctly, these features are used to quantify the reliability of nodes as sources of information. These can be made class-specific.

Raw features (ϕ^x): un-parametrized feature similarity measures such as cosine similarity ($x_i^\top x_j / \|x_i\| \|x_j\|$) and Gaussian similarity ($\exp\{-\|x_i - x_j\|_2^2 / \sigma^2\}$). These offer fixed but varied ways to measure proximity in feature space.

In our experiments we use the concatenated features $\phi = [\phi^G, \phi^S, \phi^x]$, and use a total of 20 features (see supplementary material for details). All of these features (except class-dependent ones) are transferable across datasets, and thus allows for generalizing across graphs. We leave this for future work. Note that some of the features are asymmetric, and hence Eq. (2) is no longer necessarily convex in f. Nonetheless, we have found that the expressive power gained by this provides a significant gain in performance.

3.2 Attention for probabilistic inputs

Instead of fixed weights, an *attention mechanism* α can be used to implement *dynamic* weights that change over time. For node i, we can think of $h_i^{(t)}$ as its "state" at time t, and define:

$$A_{ji}^{(t+1)} = \alpha_{ji}(h_j^{(t)}, h_i^{(t)}; \theta^\alpha) \tag{10}$$

where θ^α are the attention parameters, and $\tilde{A}$ is obtained by standard normalization. Since each state h_i is a distribution over labels, relations between states can be defined using information theoretic measures. We use negative class-weighted entropy $\boldsymbol{e}$ to quantify the certainty of a label, and negative

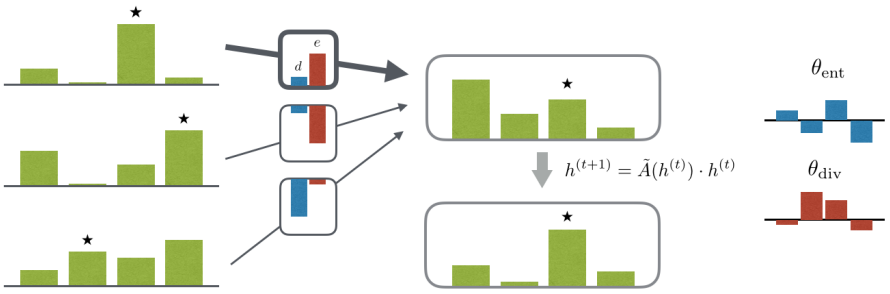

Figure 2: An illustration of the attention mechanism. Green bars denote soft labels, stars denote true labels, red and blue bars denote values of θ (right) and $\boldsymbol{e}$ or $\boldsymbol{d}$ (left). With the learned θ, attention is given to an informative and distributionally-similar neighbor (top left), and the update amplifies the predicted value of the correct label.

class-weighted KL-divergence $\boldsymbol{d}$ to measure label similarity. Attention can thus be focused based on informativeness (with entropy) and distributional similarity (with divergence). Overall, we use:

$$\alpha_{ji}(h_j, h_i; \theta^\alpha) = \exp\left(\boldsymbol{e}(h_j; \theta^{\boldsymbol{e}}) + \boldsymbol{d}(h_j, h_i; \theta^{\boldsymbol{d}})\right), \qquad \theta^\alpha = [\theta^{\boldsymbol{e}}, \theta^{\boldsymbol{d}}] \tag{11}$$

where:

$$\boldsymbol{e}(p; \eta) = -\sum_{c=1}^{C} \eta_c p_c \log(1/p_c), \qquad \boldsymbol{d}(p, q; \eta) = -\sum_{c=1}^{C} \eta_c p_c \log(p_c/q_c) \tag{12}$$

In a typical setting, labeled nodes are fixed to their true labels, unlabeled nodes start out with uniform labels, and the overall entropy is high. As updates are iteratively applied, labeled information propagates, and the the overall entropy decreases. As labels change over time, the attention-based weights change accordingly. Figure 2 provides an illustration of this process.

3.3 Bifurcation for controlling label convergence

Although the updates in Eq. (4) converge for any w, this convergence can be slow. As a result, the final outputs of a finite number of updates are often close to uniform, and thus prone to noise [25]. One solution to this problem is to dynamically bootstrap confident predictions to their corresponding hard labels [20, 10]. This process can be thought of as a way to speed up the rate of convergence by further decreasing the entropy of low-entropy input labels.

Here we generalize this idea, and propose a flexible *bifurcation* mechanism μ. The mechanism allows for dynamically *increasing* or *decreasing* the entropy of labels. Specifically, for a node i and some $\tau \in \mathbb{R}$, each entry h_{ic} is modified by:

$$\mu_c(h_i; \tau) = \frac{(h_{ic})^\tau}{\sum_{c'=1}^{C} (h_{ic'})^\tau} = \text{bifurcate}_c(h_i; \tau) \tag{13}$$

Note that since $h_i \in \Delta_C$, for any τ it holds that $\mu(h_i) \in \Delta_C$ as well.

When $\tau > 1$ and as τ increases, entropy decreases, and confident labels are amplified (in a similar manner to softmax). In contrast, when $0 < \tau < 1$ and as approaches 0, entropy decreases, and labels become uniform. For $\tau < 0$ the effects are reversed, and setting $\tau = 1$ gives $\mu(h_i) = h_i$, showing that Eq. (7) generalizes (5). Hence, τ acts as a bifurcation parameter that changes the point of convergence when the μ is repetitively applied.

In practice, it is useful to parameterize τ as a function of time, for instance by:

$$\tau(t; \theta^\tau) = a \cdot t + b + 1 \tag{14}$$

where $\theta^\tau = [a, b]$ are learned. Thus, when $\theta \neq 0$, Eq. (14) allows for time-dependent variation in the bifurcation effects of Eq. (13).

4 Learning

Recall that our goal is to learn the parameters θ, η of the network $H(\theta)$. Note that by Eq. (5), for all $i \in S$ it holds that $H_i(\theta; S) = y_i$. This deems the standard empirical loss ineffective, as it penalizes $H_i(\theta; S)$ according to y_i. As an alternative, we propose an objective based on the *leave-one-out* loss:

$$\mathcal{L}_{loo}(\theta; S) = \frac{1}{\ell} \sum_{i \in S} L\left(y_i, H_i(\theta; S_{-i})\right) + \lambda R(\theta) \tag{15}$$

where $S_{-i} = S \setminus \{i\}$, L is a loss function (e.g., cross-entropy), R is a regularization term (e.g., $R = \|\cdot\|_2^2$), and λ is the regularization coefficient. Here, each true label y_i is compared to the model's prediction when given all labeled points *except* i. Thus, the model is encouraged to propagate the labels of all nodes but one in a way which is consistent with the held-out node.

The leave-one-out loss is well-studied estimator of the expected loss with strong generalization guarantees [17, 6, 13]. In typical settings, estimating the model on multiple datasets introduces a significant computational overhead. However, this becomes feasible in SSL when ℓ is small [34]. For larger values of ℓ, a possible solution is to minimize the leave-k-loss over a training set with randomly sampled held-out sets of size k. In practice we have found it useful to weight examples in the loss by the inverse class ratio (estimated on S).

When λ is small, θ is unconstrained, and the model can easily overfit. Intuitively, this can happen when only a small subset of edges is sufficient for correctly propagating labels between nodes in S. In this case, weights of all other edges are constrained only by the model, and most nodes in U will receive noisy label information. In the other extreme, when λ grows, w approaches 0, and by Eq. (9) the normalized weights become uniform, namely $\tilde{w}_{ij} = 1/deg(j)$ for all $(i, j) \in E$.

Implementation: Eqs. (5) and (7) are implemented in TensorFlow using sparse tensor operations. Layer composition (Eq. (6)) is implemented using control-flow while loops, which easily allows for efficiently training very deep networks. Since all of the model's components are differentiable, Eq. (15) can be optimized using gradient methods. Note that θ is shared across all layers. As a result, for the basic model, w is also shared, and the overall number of variables is $O(m)$.

5 Experiments

We evaluate our method in two different experimental settings, one where the input includes both a graph and node features, and one where only the graph is available. These are in line with the two main modeling themes currently explored in deep graph-based SSL, namely graph convolution networks and graph embedding networks.

Our evaluation scheme follows the standard graph-based SSL experimental setup in line with [7, 36, 35, 34, 24, 11, 29, 23]. First we sample k labeled nodes uniformly at random, and ensure there is at least one node from each class. We then use the graph, labeled set, and node features (when available) to generate soft labels for all remaining nodes, from which hard labeled are inferred by taking the argmax. We repeat this for 10 random draws for $k = 1\%$ of the nodes, and report average accuracies. For each dataset we used the largest connected component. The supplementary material includes further details on various aspects of the experimental setup.

For tasks that include node features, we used the LINQS dataset collection [27] as a benchmark.[2] The baselines we compared to include Label Propagation (LP [36]) and Label Propagation with RBF weights (LP_{RBF}), the label propagation variant ADSORPTION [1], the Iterative Classification algorithm (ICA [22]), Graph Convolutional Networks (GCN [19]). We also include the graph embedding method NODE2VEC [11] as graph-only baselines, and Ridge Regression (RIDGEREG) as a features-only baseline. For tasks that do not include features, we used the FLIP dataset collection [26].[3] Baseline included the spectral embedding method Laplacian Eigenmaps (LEM [3]) and the deep graph embedding methods DEEPWALK [24], NODE2VEC [11], and LINE [29]. For all tasks we also compare to vanilla Label Propagation with uniform weights (LP [36]). All baselines were trained with the default parameters.[4] For methods that require regularization an ℓ_2 penalty was applied.

[2] `http://linqs.umiacs.umd.edu/projects/projects/lbc/`

[3] `http://cs.gmu.edu/~tsaha/Homepage/Projects.html`

[4] GCN requires a validation set for early stopping, for which we used an 80:20 split.

Table 1: Results on datsets with features (left) and without features (right).

Method	CiteSeer	CoRA	PubMed
DEEPLP$_\phi$	**56.2**	**66.6**	**75.0**
LP [36]	50.0	43.4	65.2
LP$_{\text{RBF}}$ [36]	50.0	41.3	65.4
ADSORP [1]	51.1	58.8	70.5
ICA [22]	46.4	46.1	37.8
GCN [19]	52.0	61.3	41.9
NODE2VEC [11]	46.4	53.1	74.4
RIDGEREG	23.7	26.9	39.5

Method	CoRA	DBLP	Flickr	IMDb	Industry
DEEPLP$_\alpha$	**67.0**	**73.6**	68.2	52.9	**25.5**
LP [36]	44.0	57.5	45.2	51.3	20.5
LEM [3]	46.0	56.9	63.8	**59.0**	21.6
DEEPWALK [24]	48.3	65.6	80.0	52.0	21.7
LINE [29]	30.9	44.6	80.0	51.4	19.8
NODE2VEC [11]	53.3	67.0	**81.4**	55.5	21.7

We evaluate two variants of our method (DEEPLP). For tasks that include features we use feature-parametrized weights as in Sec. 3.1 and denote the model by DEEPLP$_\phi$. We use a small set of raw and graph features and one seed feature per class (see supplementary material for details). For tasks without features we use the attention mechanism of Sec. 3.2, and denote the model by DEEPLP$_\alpha$. Both models use bifurcation with linear time dependency (Sec. 3.3), and the total number of parameters $|\theta|$ are $22 + 2C$ and $2 + 2C$, respectively. All models include bi-directional weight variables. For training, we use a class-balanced cross-entropy loss with ℓ_2 regularization, and set λ by 5-fold cross-validation. For optimization we used Adam [18] with learning rate 0.01.

For all tasks DEEPLP was initialized with $\theta = 0$. Hence, when the network's depth is T, learning starts in a state where a forward pass of DEEPLP is equivalent to running LP with uniform weights for T iterations, denoted LP$_T$. To tune T (our model's only hyper-parameter), we use this equivalence and tune choose $T \in \{10, 20, \ldots, 100\}$ by running 5-fold cross-validation on LP$_T$ (note that LP$_T$ typically outperforms LP). This process is fast, involves no learning, and gives DEEPLP a strong starting point. In addition, by controlling the speed of label convergence, bifurcation acts as a regularizer for T whose parameters are learned.

Results: Table 1 shows average accuracies for all methods in both learning settings. As can be seen, DEEPLP outperforms other baselines on most tasks, and consistently ranks high. DEEPLP outperforms LP by 13.5% and LP$_{\text{RBF}}$ by 13.7% on average. This quantifies the gain in accuracy by learning weights, and demonstrates the compromising effect of heuristically choosing them. While some deep methods perform well on some datasets, they fail on others, and their overall performance varies. This is true for both learning settings. A possible explanation for this is their large number of parameters and hyper-parameters. Deep methods are typically evaluated on larger seed sets and allowed access to a held-out validation data, and hyper-parameters are tuned per dataset or on a single dataset (see, e.g., [33, 19, 23, 30]).

6 Conclusions

In this work we presented a deep network for graph-based SSL. Our design process revolved around three main ideas: that edge weights should be learned, that labeled points should serve as input to the model, and that the number of parameters must be kept small. We began by revisiting the classic Label Propagation algorithm, whose simple structure allowed us to encode it as a differentiable neural network. This basic structure then served as the basis for other ad-hoc components such as attention and bifurcation. The resulting model is a powerful generalization of the original algorithm. While few labeled points make it challenging to achieve high accuracy, the upside is that more powerful tools can be used for training. This allowed us to train using the leave-one-out loss, which in most cases is intractable.

We point out two interesting avenues for future work. First, we point out that Laplacian regularization is a well-developed concept with many theoretical benefits. One interesting property is that the Laplacian's eigenvalues often play a key role in the generalization bounds of many spectral methods. An interesting question is whether these can be used to construct regularizers for the learned weights. Second, since many of the suggested components of our method depend only on the graph, it is interesting to explore how learned models generalize across different graphs and domains.

References

[1] Shumeet Baluja, Rohan Seth, D Sivakumar, Yushi Jing, Jay Yagnik, Shankar Kumar, Deepak Ravichandran, and Mohamed Aly. Video suggestion and discovery for youtube: taking random walks through the view graph. In *Proceedings of the 17th international conference on World Wide Web*, pages 895–904. ACM, 2008.

[2] Mikhail Belkin, Irina Matveeva, and Partha Niyogi. Regularization and semi-supervised learning on large graphs. In *International Conference on Computational Learning Theory*, pages 624–638. Springer, 2004.

[3] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6):1373–1396, 2003.

[4] Mikhail Belkin, Partha Niyogi, and Vikas Sindhwani. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *Journal of machine learning research*, 7(Nov):2399–2434, 2006.

[5] Yoshua Bengio, Olivier Delalleau, and Nicolas Le Roux. Label propagation and quadratic criterion. In B. Schölkopf O. Chapelle and A. Zien, editors, *Semi-Supervised Learning*, chapter 11, pages 193–216. MIT Press, 2006.

[6] Olivier Bousquet and André Elisseeff. Stability and generalization. *Journal of machine learning research*, 2(Mar):499–526, 2002.

[7] Olivier Chapelle, Bernhard Schölkopf, Alexander Zien, et al. *Semi-supervised learning*. MIT press Cambridge, 2006.

[8] Samuel I Daitch, Jonathan A Kelner, and Daniel A Spielman. Fitting a graph to vector data. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 201–208. ACM, 2009.

[9] Xiaowen Dong, Dorina Thanou, Pascal Frossard, and Pierre Vandergheynst. Learning laplacian matrix in smooth graph signal representations. *IEEE Transactions on Signal Processing*, 64(23):6160–6173, 2016.

[10] Buchnik Eliav and Edith Cohen. Bootstrapped graph diffusions: Exposing the power of nonlinearity. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(1):10, 2018.

[11] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[12] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1025–1035, 2017.

[13] Satyen Kale, Ravi Kumar, and Sergei Vassilvitskii. Cross-validation and mean-square stability. In *In Proceedings of the Second Symposium on Innovations in Computer Science (ICS2011*. Citeseer, 2011.

[14] Vassilis Kalofolias. How to learn a graph from smooth signals. In *Artificial Intelligence and Statistics*, pages 920–929, 2016.

[15] Ashish Kapoor, Hyungil Ahn, Yuan Qi, and Rosalind W Picard. Hyperparameter and kernel learning for graph based semi-supervised classification. In *Advances in Neural Information Processing Systems*, pages 627–634, 2006.

[16] Masayuki Karasuyama and Hiroshi Mamitsuka. Manifold-based similarity adaptation for label propagation. In *Advances in neural information processing systems*, pages 1547–1555, 2013.

[17] Michael Kearns and Dana Ron. Algorithmic stability and sanity-check bounds for leave-one-out cross-validation. *Neural computation*, 11(6):1427–1453, 1999.

[18] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[20] Branislav Kveton, Michal Valko, Ali Rahimi, and Ling Huang. Semi-supervised learning with max-margin graph cuts. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 421–428, 2010.

[21] Wei Liu, Junfeng He, and Shih-Fu Chang. Large graph construction for scalable semi-supervised learning. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 679–686, 2010.

[22] Qing Lu and Lise Getoor. Link-based classification. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 496–503, 2003.

[23] Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proc. CVPR*, volume 1, page 3, 2017.

[24] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[25] Nir Rosenfeld and Amir Globerson. Semi-supervised learning with competitive infection models. In *AISTATS*, 2018.

[26] Tanwistha Saha, Huzefa Rangwala, and Carlotta Domeniconi. Flip: active learning for relational network classification. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 1–18. Springer, 2014.

[27] Prithviraj Sen, Galileo Mark Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93–106, 2008.

[28] Partha Pratim Talukdar and Koby Crammer. New regularized algorithms for transductive learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 442–457. Springer, 2009.

[29] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1067–1077. International World Wide Web Conferences Steering Committee, 2015.

[30] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *stat*, 1050:20, 2017.

[31] Fei Wang and Changshui Zhang. Label propagation through linear neighborhoods. *IEEE Transactions on Knowledge and Data Engineering*, 20(1):55–67, 2008.

[32] Jason Weston, Frédéric Ratle, Hossein Mobahi, and Ronan Collobert. Deep learning via semi-supervised embedding. In *Neural Networks: Tricks of the Trade*, pages 639–655. Springer, 2012.

[33] Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, ICML'16, pages 40–48, 2016.

[34] Xinhua Zhang and Wee S Lee. Hyperparameter learning for graph based semi-supervised learning algorithms. In *Advances in neural information processing systems*, pages 1585–1592, 2007.

[35] Denny Zhou, Olivier Bousquet, Thomas N Lal, Jason Weston, and Bernhard Schölkopf. Learning with local and global consistency. In *Advances in neural information processing systems*, pages 321–328, 2004.

[36] Xiaojin Zhu, Zoubin Ghahramani, John Lafferty, et al. Semi-supervised learning using gaussian fields and harmonic functions. In *ICML*, volume 3, pages 912–919, 2003.