Lagrangian Propagation Graph Neural Networks

Anonymous Author(s)
Affiliation
Address
email

Abstract

In the last years, the popularity of deep learning techniques has renewed the interest in neural models able to process complex patterns, that are naturally encoded as graphs. In particular, different architectures have been proposed to extend the original Graph Neural Network (GNN) model. GNNs exploit a set of state variables, each assigned to a graph node, and a diffusion mechanism among neighbor nodes, to implement an iterative state update procedure that computes the fixed point of the (learnable) state transition function. In this paper, we propose a novel approach to state computation and learning for GNNs, based on a constraint optimisation task solved in the Lagrangian framework. The state convergence procedure is implicitly expressed by the constraint satisfaction mechanism and does not require a separate iterative phase for each epoch of the learning procedure. In fact, the computational structure is based on the search for saddle points of the Lagrangian in the adjoint space of weights, neural outputs (node states), and Lagrange multipliers. The proposed approach is compared experimentally with other popular models for processing graphs.

1 Introduction

Graph Neural Networks (Scarselli et al., 2009) learn the encoding for nodes in a graph to solve a given task, taking into account both the information local to each node and the whole graph topology. The computation is based on an iterative scheme implying the diffusion of information among neighboring nodes, aimed at reaching an equilibrium for the node state representations that provide a local encoding of the graph for the given task. The encoding is a computationally expensive relaxation process, that computes the fixed point of the state transition function. A maximum number of iterations can be defined, but this limits the local encoding to a maximum depth of the neighborhood of each node. Some proposals were aimed at simplifying this step, such as the scheme proposed in (Li et al., 2016) that exploits gated recurrent units. Recent approaches differ in the choice of the neighborhood aggregation method and the graph level pooling scheme, and can be categorized into two main areas. *Spectral approaches* exploit particular embeddings of the graph and the convolution operation defined in the spectral domain (Bruna et al., 2014). Characterized by computational drawbacks, simplified approaches consider smooth reparametrization (Henaff, Bruna, and LeCun, 2015) or approximation of the spectral filters (Defferrard, Bresson, and Vandergheynst, 2016). Graph Convolutional Networks (GCNs) (Kipf and Welling, 2017), restrict filters to operate in a 1-hop neighborhood of each node. *Spatial methods*, like PATCHY-SAN (Niepert, Ahmed, and Kutzkov, 2016; Duvenaud et al., 2015), DCNNs (Atwood and Towsley, 2016), GraphSAGE (Hamilton, Ying, and Leskovec, 2017), GATs (Veličković et al., 2017), SortPooling (Zhang et al., 2018), GIN (Xu et al., 2018), exploit directly the graph topology, without the need of an intermediate representation, mainly distinguished by the definition and capabilities of the aggregation operator used to compute the node states.

In this paper, we propose a new learning mechanism for GNNs based on a Lagrangian formulation, that allows the embedding of the fixed point computation into the optimization problem constraints. In the proposed scheme the network state representations and the weights are jointly optimized without the need of applying the fixed point relaxation procedure at each weight update epoch. A Lagrangian formulation of learning neural networks can be found in the seminal work of LeCun et al. (1988), that studies a theoretical framework for Backpropagation. More recently Carreira-Perpinan and Wang (2014) and Taylor et al. (2016) introduced the idea of training networks, transformed into a constraint–based representation, through an extension of the space of the learnable parameters. By framing the optimization of neural networks in the Lagrangian framework, where neuron computations are expressed as constraints, the main goal is to obtain a local algorithm, in which different layers can be updated in parallel. On the contrary in the proposed approach, we use a novel mixed strategy. In particular, the majority of the computations still rely on Backpropagation, whereas constraints are exploited only to express the diffusion mechanism. This allows us to carry out both the optimization of the neural network weights and the diffusion process at the same time, instead of alternating them into two distinct phases (like in Scarselli et al. (2009)).
It has already been shown that algorithms on graphs can be effectively learned exploiting a constrained fixed-point

formulation. For example, SSE (Dai et al., 2018) exploits the Reinforcement Learning *policy iteration* algorithm for the interleaved evaluation of the fixed point equation and the improvement of the transition and output functions. Our approach, starting from similar assumptions, exploits the unifying Lagrangian framework for learning both the transition and the output functions. Thus, by framing the optimization algorithm into a standard gradient descent/ascent scheme, we are allowed to use recent update rules (e.g. Adam) without the need to resort to ad-hoc moving average updates.
The paper is organized as follows. Section 2 briefly reviews the GNN model, then in Section 3 the proposed constrained optimization problem is formulated. Section 4 reports the experimental evaluation of Lagrangian learning for GNNs. Finally, Section 5 draws the conclusions and proposes the future research directions.

2 Graph Neural Networks

The term Graph Neural Network (GNN) refers to a general computational model, that exploits the processing and learning schemes of neural networks to process non Euclidean data, i.e. data organized as graphs.

Given an input graph $G = (V, E)$, where V is a finite set of *nodes* and $E \subseteq V \times V$ collects the *arcs*, GNNs apply a two-phase computation on G. In the *encoding* (or *aggregation*) phase the model computes a state vector for each node in V by (iteratively) combining the states of neighboring nodes (i.e. nodes $u, v \in V$ that are connected by an arc $(u, v) \in E$). In the second phase, usually referred to as *output* (or *readout*), the latent representations encoded by the node states are exploited to compute the model output. The GNN can implement either a *node-focused* function, where an output is produced for each node of the input graph, or a *graph-focused* function, where the representations of all the nodes are aggregated to yield a single output for the input graph.

The GNN is defined by the *state transition* function f_a required in the encoding phase and the *output* function f_r exploited in the output phase, as follows:

$$x_v^{(t)} = f_a(x_{\text{ne}[v]}^{(t-1)}, l_{\text{ne}[v]}, l_{(v,\text{ch}[v])}, l_{(\text{pa}[v],v)}, x_v^{(t-1)}, l_v|\theta_{f_a}), \tag{1}$$

$$y_v = f_r(x_v^{(T)}|\theta_{f_r}), \tag{2a}$$

$$y_G = f_r(\{x_v^{(T)}, v \in V\}|\theta_{f_r}), \tag{2b}$$

where $x_v^{(t)} \in \mathbb{R}^s$ is the state of the node v at iteration t, $\text{pa}[v] = \{u \in V : (u, v) \in E\}$ is the set of the *parents* of node v in G, $\text{ch}[v] = \{u \in V : (v, u) \in E\}$ are the *children* of v in G, $\text{ne}[v] = \text{pa}[v] \cup \text{ch}[v]$ are the *neighbors* of the node v in G, $l_u \in \mathbb{R}^m$ is the feature vector available for node $u \in V$, and $l_{(u,w)} \in \mathbb{R}^d$ is the feature vector available for the arc $(u, w) \in E$[1]. The vectors θ_{f_a} and θ_{f_r} collect the model parameters (the neural network weights) to be adapted during the learning procedure. Equations (2a) and (2b) are the two variants of the output function for node-focused or graph-focused tasks, respectively. Different implementations have been proposed for f_a. For instance, the original scheme, proposed in (Scarselli et al., 2009), aggregates the contributions of the neighbor nodes by a sum as

$$\sum_{u \in \text{ne}[v]} h(x_u, l_u, l_{(v,u)}, l_{(u,v)}, x_v, l_v|\theta_h) , \tag{3}$$

where the function $h()$ is implemented by a feedforward neural network with s outputs and $2s + 2m + 2d$ inputs obtained by the concatenation of its arguments.
The recursive application of the state transition function f_a on the graph nodes implements a diffusion mechanism that is iterated for T steps to yield the node states. In fact, by stacking t times the aggregation of 1-hop neighborhoods by f_a, the information computed at a given node can be transferred to the nodes that are distant at most t-hops.

In the original GNN model (Scarselli et al., 2009), eq. (1) is executed until convergence of the state representation, i.e. until $x_v^{(t)} \simeq x_v^{(t-1)}, v \in V$. This scheme corresponds to the computation of the *fixed point* of the state transition function f_a on the input graph. In order to guarantee the convergence of this phase, the transition function is required to be a *contraction map*.
Basically, the encoding phase, through the iteration of f_a, finds a solution to the fixed point problem defined by the constraint[2]

$$\forall v \in V, x_v = f_{a,v} \tag{4}$$

This diffusion mechanism is more general than executing only a fixed number of iterations. However, it can be computationally expensive and, hence, many recent GNN architectures apply only a fixed number of iterations for all nodes.

3 A constraint-based formulation of GNNs

Neural network learning can be cast as a Lagragian optimization problem by a formulation that requires the minimization of the classical data fitting loss (and eventually a regularization term) and the satisfaction of a set of *architectural* constraints that describe the computation performed on the data. Given this formulation, the solution can be computed by finding the *saddle points* of the associated Lagrangian in the space defined by the original network parameters and the *Lagrange multipliers*. The constraints can be exploited to enforce the computational structure that characterizes the GNN models.

The computation of Graph Neural Networks is driven by the input graph topology that defines the constraints among the computed state variables $x_v, v \in V$. In particular, the fixed point computation aims at solving eq. (4), that imposes a constraint between the node states and the way they are computed by the state transition function.
In the original GNN learning algorithm, the computation of the fixed point is required at each epoch of the learning

[1]With abuse of notation, we denote the set $\{x_u^{(t-1)} : u \in \text{ne}[v]\}$ by $x_{\text{ne}[v]}^{(t-1)}$. Similar definitions apply for $l_{\text{ne}[v]}$, $l_{(v,\text{ch}[v])}$, and $l_{(\text{pa}[v],v)}$.

[2]Henceforth, $f_{a,v}$ is used to denote the state transition function applied to a node $v \in V$, i.e. $f_{a,v} = f_a(x_{\text{ne}[v]}, l_{\text{ne}[v]}, l_{(v,\text{ch}[v])}, l_{(\text{pa}[v],v)}, x_v, l_v|\theta_{f_a})$.

procedure, as implemented by the iterative application of the transition function. Moreover, also the gradient computation requires to take into account the relaxation procedure, by a backpropagation schema through the replicas of the state transition network exploited during the iterations for the fixed point computation. This procedure may be time consuming when the number of iterations T for convergence to the fixed point is high (for instance in the case of large graphs).

We consider a Lagrangian formulation of the problem by adding free variables corresponding to the node states x_v, such that the fixed point is directly defined by the constraints themselves, as

$$\forall v \in V,\ \mathcal{G}\left(x_v - f_{a,v}\right) = 0$$

where $\mathcal{G}(x)$ is a function characterized by $\mathcal{G}(0) = 0$, such that the satisfaction of the constraints implies the solution of eq. (4). Possible choices are $\mathcal{G}(x) = x$, $\mathcal{G}(x) = x^2$, or $\mathcal{G}(x) = \max(||x||_1 - \epsilon, 0)$, where $\epsilon \geq 0$ is a parameter that can be used to allow tolerance in the satisfaction of the constraint. The hard formulation of the problem requires $\epsilon = 0$, but by setting ϵ to a small positive value it is possible to obtain a better generalization and tolerance to noise.

In the following, for simplicity, we will refer to a node-focused task, such that for some (or all) nodes $v \in S \subseteq V$ of the input graph G, a target output y_v is provided as a supervision[3]. If $L(f_r(x_v|\theta_{fr_r}), y_v)$ is the loss function used to measure the target fitting approximation for node $v \in S$, the formulation of the learning task is:

$$\begin{aligned} \min_{\theta_{f_a}, \theta_{f_r}, X} \quad & \sum_{v \in S} L(f_r(x_v|\theta_{f_r}), y_v) \\ \text{subject to} \quad & \mathcal{G}\left(x_v - f_{a,v}\right) = 0, \quad \forall\, v \in V \end{aligned} \tag{5}$$

where θ_{f_a} and θ_{f_r} are the weights of the MLPs implementing the state transition function and the output function, respectively, and $X = \{x_v : v \in V\}$ is the set of the introduced free state variables.

This problem statement implicitly includes the definition of the fixed point of the state transition function in the optimal solution, since for any solution the constraints are satisfied and hence the computed optimal x_v are solutions of eq. (4). As shown in the previous subsection, the constrained optimization problem of eq. (5) can framed into the Lagrangian framework by introducing for each constraint a Lagrange multiplier λ_v, to define the Lagrangian function

$$\begin{aligned} \mathcal{L}(\theta_{f_a}, \theta_{f_r}, X, \Lambda) = \sum_{v \in S} \big[& L(f_r(x_v|\theta_{f_r}), y_v) + \\ & + \lambda_v \mathcal{G}\left(x_v - f_{a,v}\right) \big], \end{aligned} \tag{6}$$

where Λ is the set of the $|V|$ Lagrangian multipliers. Finally, we can define the unconstrained optimization problem as the search for saddle points in the adjoint space $(\theta_{f_a}, \theta_{f_r}, X, \Lambda)$ as:

$$\min_{\theta_{f_a}, \theta_{f_r}, X} \max_{\Lambda} \mathcal{L}(\theta_{f_a}, \theta_{f_r}, X, \Lambda)$$

that can be solved by gradient descent with respect to the variables $\theta_{f_a}, \theta_{f_r}, X$ and gradient ascent with respect to the Lagrange multipliers Λ.
The gradient can be computed locally to each node, given the local variables and those of the neighboring nodes[4]. Being f_a and f_r implemented by feedforward neural networks, their derivatives are obtained easily by applying a classical backpropagation scheme, in order to optimize the Lagrangian function in the descent-ascent scheme, aiming at the saddle point, following Platt and Barr (1988).

Even if the proposed formulation adds the free state variables x_v and the Lagrange multipliers λ_v, $v \in V$, there is no significant increase in the memory requirements since the state variables are also required in the original formulation and there is just a Lagrange multiplier for each node.

The learning algorithm is based on a mixed strategy where *(i)* Backpropagation is used to efficiently update the weights of the neural networks that implement the state transition and output functions, and, *(ii)* the diffusion mechanism evolves gradually by enforcing the convergence of the state transition function to a fixed point by virtue of the constraints. This last point is a novel approach in training GNNs. In fact, in classical approaches, the encoding phase (see Section 2) is completely executed during the forward pass to compute the node states and, only after this phase is completed, the backward step is applied to update the weights of f_a and f_r. In the proposed scheme, both the neural network weights and the node state variables are simultaneously updated, forcing the state representation function towards a fixed point of f_a in order to satisfy the constraints.

Common graph models exploit synchronous updates among all nodes and multiple iterations for the node state embedding, with a computational complexity for each parameter update $\mathcal{O}(T(|V| + |E|))$, where T is the number of iterations, $|V|$ the number of nodes and $|E|$ the number of edges. By simultaneously carrying on the optimization of neural models and the diffusion process, our scheme relies only on 1-hop neighbors for each parameter update, hence showing a computational cost of $\mathcal{O}(|V| + |E|)$. From the memory cost viewpoint, the persistent state variable matrix requires $\mathcal{O}(|V|)$ space. However, it represents a much cheaper cost than most of GNN models, usually requiring $\mathcal{O}(T|V|)$ space. In fact, those methods need to store all the intermediate state values of all the iterations, for a latter use in back-propagation.

4 Experiments

The evaluation was carried out on two classes of tasks. Artificial tasks (Subgraph matching and Clique detection) are commonly exploited as benchmarks for GNNs, thus, allowing a direct comparison of the proposed constraint based optimization algorithm with respect to the original GNN learning scheme, on the same architecture. The second class

[3]For the sake of simplicity we consider only the case when a single graph is provided for learning. The extension for more graphs is straightforward for node-focused tasks, since they can be considered as a single graph composed by the given graphs as disconnected components.

[4]The derivatives of the Lagrangian are reported in the Appendix B.

of tasks consists of graph classification in the domains of social networks and bioinformatics.

In the experiments we considered two different implementations of the state transition function $f_{a,v}$:

$$f_{a,v}^{(\text{SUM})} = \sum_{u \in ne[v]} h(x_u, l_u, l_{(v,u)}, l_{(u,v)}, x_v, l_v | \theta_h) \quad (7)$$

$$f_{a,v}^{(\text{AVG})} = \frac{1}{|ne[v]|} \sum_{u \in ne[v]} h(x_u, l_u, l_{(v,u)}, l_{(u,v)}, x_v, l_v | \theta_h) \quad (8)$$

4.1 Artificial Tasks

We designed a batch of experiments on the following two tasks to validate our simple local optimization approach to constraint-based networks. In particular, we want to show that our optimization scheme can learn better transition and output functions than the corresponding GNN model of Scarselli et al. (2009). Moreover, we want to investigate the behaviour of the algorithm for different choices of the function $\mathcal{G}(x)$, i.e. when using different modalities to enforce the state convergence constraints. In particular, we tested functions with different properties: *ϵ-insensitive functions*, i.e $\mathcal{G}(x) = 0,\ \forall x : -\epsilon \leq x \leq \epsilon$, *unilateral functions*, i.e. $\mathcal{G}(x) \in \mathbb{R}^+$, and *bilateral functions*, i.e. $\mathcal{G}(x) \in \mathbb{R}$ (a $\mathcal{G}$ function is either unilateral or bilateral). The considered choices for $\mathcal{G}(x)$ are: x (*lin*), $\max(x, \epsilon) - \max(-x, \epsilon)$ (*lin* $- \epsilon$), $|x|$ (*abs*), $\max(|x| - \epsilon, 0)$ (*abs* $- \epsilon$), and x^2 (*squared*).

Subgraph Matching. Given a graph G and a graph S such that $|S| \leq |G|$, the subgraph matching problem consists in finding the nodes of a subgraph $\hat{S} \subset G$ which is isomorphic to S. The task is that of learning a function τ, such that $\tau_S(G, n) = 1, n \in V$, when the node n belongs to the given subgraph S, otherwise $\tau_S(G, n) = 0$. The problem of finding a given subgraph is common in many practical problems and corresponds, for instance, to finding a particular small molecule inside a greater compound. An example of a subgraph structure is shown, on the left, in Fig. 1. Our dataset is composed of 100 different graphs, each one having 7 nodes. The number of nodes of the target subgraph S is instead 3.

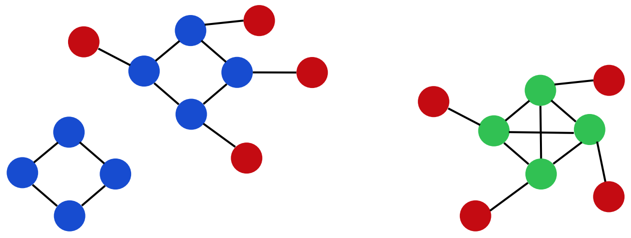
Figure 1: On the left, an example of a subgraph matching problem, where the graph with the blue nodes is matched against the bigger graph. On the right, an example of a graph containing a clique. The green nodes represent a fully connected subgraph of dimension 4, whereas the red nodes do not belong to the clique.

Clique localization A clique is a complete graph, i.e. a graph in which each node is connected with all the others. In a network, overlapping cliques (i.e. cliques that share some nodes) are admitted. Clique localization is a particular instance of the subgraph matching problem, with S being complete. However, the several symmetries contained in a clique makes the graph isomorphism test more difficult. Indeed, it is known that the graph isomorphism has polynomial time solutions only in absence of symmetries. A clique example is shown in Fig. 1. In the experiments, we consider a dataset composed by graphs having 7 nodes each, where the dimension of the maximal clique is 3 nodes.

Results. Following the experimental setting of Scarselli et al. (2009), we exploited a training, validation and test set having the same size, i.e. 100 graphs each. We performed a tuning of the hyperparemeters[5]. The proposed model is compared with the GNN model of Scarselli et al. (2009), with the same number of hidden neurons of the f_a and f_r functions. Results are presented in Table 1.

Constraints characterized by *unilateral functions* usually offer better performances than equivalent bilateral constraints. This might be due to the fact that keeping constraints positive (as in unilateral constraints) provides a more stable learning process. Moreover, smoother constraints (i.e *squared*) or ϵ–insensitive constraints tend to perform slightly better than the hard versions. This can be due to the fact that as the constraints move closer to 0 they tend to give a small or null contribution, for *squared* and *abs*-ϵ respectively, acting as regularizers.

4.2 Graph Classification

We used 4 benchmarks in bioinformatics (MUTAG, PTC, NCI1, PROTEINS) and 2 in social network analysis (IMDB-BINARY, IMDB-MULTI) (Yanardag and Vishwanathan, 2015). In MUTAG, PTC, NCI1, and PROTEINS, the graph nodes have categorical input labels (e.g. the atom symbol). In the social network datasets, there are no node labels and we followed the approach in Xu et al. (2018), where the nodes were labeled by one-hot encodings of their degrees. Dataset statistics are summarized in Table 2.

We compared the proposed Lagrangian Propagation GNN (LP-GNN) scheme with some of the state-of-the-art neural models for graph classification, such as Graph Convolutional Neural Networks. All the GNN-like models have a number of layers/iterations equal to 5. An important difference with these models is that, by using a different transition function at each iteration, at a cost of a much larger number of parameters, they have a much higher representational power. Even though our model could, in principle, stack multiple diffusion processes at different levels (i.e. different latent representation of the nodes) and, then, have multiple transition functions, we have not explored this direction in this evaluation. We applied the settings of (Niepert, Ahmed, and Kutzkov, 2016), using the same models[6]. In particular, we

[5]See Appendix C for the details.

[6]See Appendix D for a detailed list of the references.

Model			Subgraph		Clique	
	$\mathcal{G}$	ϵ	*Acc(avg)*	*Acc(std)*	*Acc(avg)*	*Acc(std)*
LP-GNN	*abs*	0.00	96.25	0.96	88.80	4.82
		0.01	**96.30**	0.87	88.75	5.03
		0.10	95.80	0.85	85.88	4.13
	lin	0.00	95.94	0.91	84.61	2.49
		0.01	95.94	0.91	85.21	0.54
		0.10	95.80	0.85	85.14	2.17
	squared	-	96.17	1.01	**93.07**	2.18
Scarselli et al. (2009)	-	-	95.86	0.64	91.86	1.12

Table 1: Accuracy on the artificial datasets, for the proposed model (Lagrangian Propagation GNN - LP-GNN) and the original GNN model for different settings.

Datasets	IMDB-B	IMDB-M	MUTAG	PROT.	PTC	NCI1
# graphs	1000	1500	188	1113	344	4110
# classes	2	3	2	2	2	2
Avg # nodes	19.8	13.0	17.9	39.1	25.5	29.8
DCNN	49.1	33.5	67.0	61.3	56.6	62.6
PATCHYSAN	71.0 ± 2.2	45.2 ± 2.8	92.6 ± 4.2	75.9 ± 2.8	60.0 ± 4.8	78.6 ± 1.9
DGCNN	70.0	47.8	85.8	75.5	58.6	74.4
AWL	74.5 ± 5.9	51.5 ± 3.6	87.9 ± 9.8	–	–	–
GIN	75.1 ± 5.1	52.3 ± 2.8	89.4 ± 5.6	76.2 ± 2.8	64.6 ± 7.0	82.7 ± 1.7
GNN	60.9 ± 5.7	41.1 ± 3.8	88.8 ± 11.5	76.4 ± 4.4	61.2 ± 8.5	51.5 ± 2.6
LP-GNN*	71.2 ± 4.7	46.6 ± 3.7	90.5 ± 7.0	77.1 ± 4.3	64.4 ± 5.9	68.4 ± 2.1

Table 2: Average accuracy and standard deviation for the graph classification benchmarks, evaluated on the test set, for different GNN models. The proposed model is denoted as LP_GNN and marked with a star.

performed 10-fold cross-validation and reported both the average and standard deviation of validation accuracy across the 10 folds within the cross-validation. The stopping epoch is selected as the epoch with the best cross-validation accuracy averaged over the 10 folds. The hyperparameters were tuned by grid search[7]. Results are shown in Table 2.

As previously stated, differently from the baseline models, our approach does not rely on a deep stack of layers based on differently learnable filters. Despite of this fact, the simple GNN model trained by the proposed scheme offers performances that, on average, are preferable or on-par to the ones obtained by more complex models that exploit a larger amount of parameters. Moreover, it is interesting to note that for current GNN models, the role of the architecture depth is twofold. First, as it is common in deep learning, depth is used to perform a multi-layer feature extraction of node inputs. Secondly, it allows node information to flow through the graph fostering the realisation of a diffusion mechanism. Conversely, our model strictly splits these two processes. We believe this distinction to be a fundamental ingredient for a clearer understanding of which mechanism, between diffusion and node deep representation, is concurring in achieving specific performances. Indeed, in this paper, we show that the diffusion mechanism paired only with a simple shallow representation of nodes is sufficient to match performances of much deeper and complex networks.

[7]See Appendix D for details.

5 Conclusions and Future Work

We proposed a formulation of the GNN learning task as a constrained optimization that allows us to avoid the explicit computation of the fixed point needed to encode the graph. The proposed framework defines how to jointly optimize the model weights and the state representation without the need of separate phases. This approach simplifies the computational scheme of GNNs and allows us to incorporate alternative strategies in the fixed point optimization by the choice of the constraint function $\mathcal{G}()$. As shown in the experimental evaluation, the appropriate functions may affect generalization and robustness to noise.

Future work will be devoted to explore systematically the properties of the proposed algorithm in terms of convergence and complexity. Furthermore, the proposed constraint-based scheme can be extended to all the other methods proposed in the literature that exploit more sophisticated architectures. Finally, LP-GNN can be extended allowing the diffusion mechanism to take place at multiple layers allowing a *controlled* integration of diffusion and deep feature extraction mechanisms.

References

Atwood, J., and Towsley, D. 2016. Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, 1993–2001.

Bruna, J.; Zaremba, W.; Szlam, A.; and LeCun, Y. 2014. Spectral networks and locally connected networks on graphs. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*.

Carreira-Perpinan, M., and Wang, W. 2014. Distributed optimization of deeply nested systems. In *Artificial Intelligence and Statistics*, 10–19.

Dai, H.; Kozareva, Z.; Dai, B.; Smola, A.; and Song, L. 2018. Learning steady-states of iterative algorithms over graphs. In *International Conference on Machine Learning*, 1114–1122.

Defferrard, M.; Bresson, X.; and Vandergheynst, P. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, 3837–3845.

Duvenaud, D. K.; Maclaurin, D.; Aguilera-Iparraguirre, J.; Gómez-Bombarelli, R.; Hirzel, T.; Aspuru-Guzik, A.; and Adams, R. P. 2015. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, 2224–2232.

Hamilton, W. L.; Ying, R.; and Leskovec, J. 2017. Inductive representation learning on large graphs. In *NIPS*.

Henaff, M.; Bruna, J.; and LeCun, Y. 2015. Deep convolutional networks on graph-structured data. *CoRR* abs/1506.05163.

Ivanov, S., and Burnaev, E. 2018. Anonymous walk embeddings. *arXiv preprint arXiv:1805.11921*.

Kipf, T. N., and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

LeCun, Y.; Touresky, D.; Hinton, G.; and Sejnowski, T. 1988. A theoretical framework for back-propagation. In *Proceedings of the 1988 connectionist models summer school*, volume 1, 21–28. CMU, Pittsburgh, Pa: Morgan Kaufmann.

Li, Y.; Tarlow, D.; Brockschmidt, M.; and Zemel, R. S. 2016. Gated graph sequence neural networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

Niepert, M.; Ahmed, M.; and Kutzkov, K. 2016. Learning convolutional neural networks for graphs. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 2014–2023.

Platt, J. C., and Barr, A. H. 1988. Constrained differential optimization. In *Neural Information Processing Systems*, 612–621.

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2009. The graph neural network model. *IEEE Trans. Neural Networks* 20(1):61–80.

Taylor, G.; Burmeister, R.; Xu, Z.; Singh, B.; Patel, A.; and Goldstein, T. 2016. Training neural networks without gradients: A scalable admm approach. In *International conference on machine learning*, 2722–2731.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; and Bengio, Y. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903*.

Xu, K.; Hu, W.; Leskovec, J.; and Jegelka, S. 2018. How powerful are graph neural networks? *CoRR* abs/1810.00826.

Yanardag, P., and Vishwanathan, S. 2015. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1365–1374. ACM.

Zhang, M.; Cui, Z.; Neumann, M.; and Chen, Y. 2018. An end-to-end deep learning architecture for graph classification. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, 4438–4445.

A Appendix

In Table 3 we list some implementations of the transition function f_a, as proposed in the literature. It should be noted that this function may depend on a variable number of inputs, given that the nodes $v \in V$ may have different degrees $\text{de}[v] = |\text{ne}[v]|$. Moreover, in general, the proposed implementations are invariant with respect to permutations of the nodes in $\text{ne}[v]$, unless some predefined ordering is given for the neighbors of each node.

B Lagrangian derivatives

The gradient can be computed locally to each node, given the local variables and those of the neighboring nodes. In fact, the derivatives of the Lagrangian with respect to the considered parameters are:

$$\frac{\partial \mathcal{L}}{\partial x_v} = L' f'_{r,v} + \lambda_v \mathcal{G}'_v (1 - f'_{a,v}) - \sum_{w: v \in ne[w]} \lambda_w \mathcal{G}'_w f'_{a,w} \quad (9)$$

$$\frac{\partial \mathcal{L}}{\partial \theta_{f_a}} = - \sum_{v \in S} \lambda_v \mathcal{G}'_v f'_{a,v} \quad (10)$$

$$\frac{\partial \mathcal{L}}{\partial \theta_{f_r}} = \sum_{v \in S} L' f'_{r,v} \quad (11)$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_v} = \mathcal{G}_v \quad (12)$$

where, $f_{a,v} = f_a(x_{\text{ne}[v]}, l_{\text{ne}[v]}, l_{(v,\text{ch}[v])}, l_{(\text{pa}[v],v)}, x_v, l_v | \theta_{f_a})$, $f'_{a,v}$ is its first derivative, which is computed with respect to the same argument as in the partial derivative on the left side, $f_{r,v} = f_r(x_v|\theta_{f_r})$, $f'_{r,v}$ is its first derivative, $\mathcal{G}_v = \mathcal{G}(x_v - f_{a,v})$ and $\mathcal{G}'_v$ is its first derivative, and, finally, L' is the first derivative of L. Note that when parameters are vectors, the reported gradients should be considered element-wise.

C Artificial Tasks experimental settings

We tuned the hyperparameters on the validation data, by selecting the node state dimension from the set $\{5, 10, 35, \}$, the dropout drop-rate from the set $\{0., 0.7\}$, the state *transition function* from $\{f^{(\text{AVG})}_{a,v}, f^{(\text{SUM})}_{a,v}\}$ and their number of hidden units from $\{5, 20, 50\}$. We used the Adam optimizer (TensorFlow). Learning rate for parameters θ_{f_a} and θ_{f_r} is selected from the set $\{10^{-5}, 10^{-4}, 10^{-3}\}$, and the learning rate for the variables x_v and λ_v from the set $\{10^{-4}, 10^{-3}, 10^{-2}\}$.

D Graph Classification experimental settings

The models used in the comparison are: Diffusion-Convolutional Neural Networks (DCNN) (Atwood and Towsley, 2016), PATCHY-SAN (Niepert, Ahmed, and Kutzkov, 2016), Deep Graph CNN (DGCNN) (Zhang et al., 2018), AWL (Ivanov and Burnaev, 2018) , GIN-GNN (Xu et al., 2018), original GNN (Scarselli et al., 2009). Apart from original GNN, we report the accuracy as reported in the referred papers.

We tuned the hyperparameters by searching: (1) the number of hidden units for both the f_a and f_r functions from the set $\{5, 20, 50, 70, 150\}$; (2) the state transition function from $\{f^{(\text{AVG})}_{a,v}, f^{(\text{SUM})}_{a,v}\}$; (3) the dropout ratio from $\{0, 0.7\}$; (4) the size of the node state x_v from $\{10, 35, 50, 70, 150\}$; (5) learning rates for both the θ_{f_a}, θ_{f_r}, x_v and λ_v from $\{0.1, 0.01, 0.001\}$.

Function	Implementation
Scarselli et al. (2009) Sum	$\sum_{u \in \text{ne}[v]} h(x_u, l_u, l_{(v,u)}, l_{(u,v)}, x_v, l_v \vert \theta_h)$
Xu et al. (2018) Sum	$h(x_v + \sum_{u \in \text{ne}[v]} x_u)$
Kipf and Welling (2017) Mean	$h\left(\frac{1}{\vert\text{ne}[v]\vert+1}(x_v + \sum_{u \in \text{ne}[v]} x_u)\right)$
Hamilton, Ying, and Leskovec (2017) Max	$\max_{u \in \text{ne}[v]} h(x_u)$

Table 3: Simplified implementations of the state transition function $f_a()$. The function $h()$ is implemented by a feedforward neural network with s outputs, whose input is the concatenation of its arguments (f.i. in the first case the input consists of a vector of $2s + 2m + 2d$ entries, with $l_{(u,v)} \in \mathbb{R}^d$ and $l_u \in \mathbb{R}^m$). For the sake of clarity, some of these formulas are reported in a simplified form w.r.t. the original proposal. For example, the "mean" function in Kipf and Welling (2017) is a weighted mean, where the weights come from the normalized graph adjacency matrix, or the "max" function in Hamilton, Ying, and Leskovec (2017) is followed by a concatenation.

	lin	*lin-*ϵ	*abs*	*abs-*ϵ	*squared*
$\mathcal{G}(x)$	x	$\max(x, \epsilon) - \max(-x, \epsilon)$	$\vert x\vert$	$\max(\vert x\vert - \epsilon, 0)$	x^2
Unilateral	$\times$	$\times$	✓	✓	✓
ϵ-insensitive	$\times$	✓	$\times$	✓	$\times$

Table 4: The considered variants of the $\mathcal{G}$ function. By introducing ϵ-insensitive constraint satisfaction, we can inject into our hard-optimization scheme a controlled amount (i.e. ϵ) of unsatisfaction tolerance.