AP-GCN Revisited: Replication and Alternative Approaches for Adaptive Propagation in Graph Neural Networks

GDL 2025
Group id: BEK
Project id: AP-GCN

Jonatan Bella, Tobias Erbacher, Jonas Knupp
{jonatan.bella, tobias.erbacher, jonas.knupp}@usi.ch

Abstract

In conventional Graph Convolutional Networks (GCNs), each node performs a fixed number of message passing steps. Spinelli et al. proposed the Adaptive Propagation GCN (AP-GCN), a model architecture that enables learning an individual number of message passing steps for each node. Our work consists of two main components: a replication study of the AP-GCN paper and an exploration of alternative model architectures that support per-node adaptive message passing. For the replication study, we re-implemented AP-GCN and evaluated it using the same experimental setup as in the original work. Our results closely match the accuracies reported by Spinelli et al. In the second part, we developed three adaptive propagation architectures that adhere to the same constraints as AP-GCN: each node performs at least one message passing iteration, and once halted, a node remains inactive. The architectures we explored are: RL-AP-GCN, Ponder-AP-GCN (adapted from PonderNet), and Gumbel-AP-GCN (also adapted from PonderNet). Additionally, we evaluated Co-GCN (Cooperative Graph Neural Networks) from Finkelshtein et al., which is not restricted by the aforementioned constraints. All four architectures were evaluated with the same experimental setup as AP-GCN. While our findings show that none of them outperform AP-GCN on any dataset, RL-AP-GCN, Ponder-AP-GCN and Gumbel-AP-GCN achieve competitive results in most of the experiments.

1 Introduction

This work is a replication study of the paper "Adaptive Propagation Graph Convolutional Network" by Indro Spinelli, Simone Scardapane, and Aurelio Uncini [1]. In Graph Convolutional Networks (GCNs), each node updates its representation by aggregating and transforming features from its neighbors. This process is repeated over several message passing steps, allowing nodes to incorporate information from increasingly distant parts of the graph. The core contribution of Spinelli et al. is the introduction of the Adaptive Propagation Graph Convolutional Network (AP-GCN) that allows each node in the graph to undergo an individual number of message passing steps. The number of message passing steps per node is learned during training.

In addition to the replication work, we implemented three different model architectures to allow for an individual number of message passing steps per node. The RL-AP-GCN, Ponder-AP-GCN, and Gumbel-AP-GCN model architectures closely follow the constraints and capabilities of the AP-GCN. That is, they do at least one step of message passing and once a node is halted it remains inactive thereafter. Ponder-AP-GCN and Gumbel-AP-GCN are inspired by PonderNet [2]. In addition to these three adaptive propagation model architectures, we adopt the Cooperative Graph Convolution Network (Co-GCN) architecture from Finkelshtein et al. [3] to assess whether relaxing the constraints mentioned above improves model performance. These four additional model architectures were evaluated in the same setting as AP-GCN.

We could replicate the findings of Spinelli et al. regarding the accuracy of AP-GCN on all six datasets. Although our accuracies are marginally lower than those originally reported, the differences are minimal but all outside of the 95% confidence interval. Furthermore, while none of our model architectures outperform AP-GCN, all adaptive propagation architectures achieve a reasonable accuracy — in particular considering that we did not have the resources for systematic hyperparameter tuning of each model architecture. Since Co-GCN performed worse than the other adaptive propagation methods in most cases, we conclude that relaxing the constraints did not lead to a better model performance. In addition, we found that Co-GCN is more sensitive to the choice of hyperparameters for the given datasets.

The link to our GitHub repository is available in Section 4.3.

2 Related works

The main contribution of Spinelli et al. is the introduction of the Adaptive Propagation Graph Convolutional Network (AP-GCN) which enables each node to have an individual number of message passing steps. While Spinelli et al. claim that AP-GCN was at the time the only model in which the number of message passing steps is determined independently for each node and dynamically adjusted during training, there exist model architectures that achieve a similar, or at least related, goal.

Xu et al. [4] introduced Jumping Knowledge Networks which perform a fixed number of message passes for all nodes and then use per-node LSTM attention to calculate the weighted average over the hidden vectors of the message passing rounds.

Liu et al. [5] presented GeniePath networks which not only use an attention mechanism to weight the contribution of neighboring nodes but also rely on an LSTM-like gating mechanism that controls the flow of information from one message passing step to the next. In principle, GeniePath networks can learn to stop propagating information individually for each node.

Lai et al. [6] presented Policy-GNN which consists of two modules: A meta-policy module that relies on Deep Q-Learning predicts the required number of message passing steps per node and a GNN module that uses the meta-policy to learn graph representations.

Banino et al. [2] proposed the PonderNet architecture, which dynamically adjusts its computational effort based on the complexity of the given problem. While the authors do not explicitly discuss PonderNet in the context of GNNs, the architecture itself is adaptable to various neural network designs.

Xiao et al. [7] presented the Learning to Propagate (L2P) framework. They use a latent discrete variable for each node that represents its optimal number of message passing steps. These variables are learned using variational Expectation-Maximization.

Finkelshtein et al. [3] introduced Cooperative Graph Neural Networks (Co-GNNs) where nodes are in one of the states: Standard, Listen, Broadcast, or Isolate. The message passing behavior of each node is governed by its current state. Nodes in

the Broadcast state do not receive any messages but can still send messages —— A situation similar to the halting mechanism in AP-GCN.

3 Methodology

In this section we first introduce the problem of having an individual number of message passing steps per node. Then, we describe the AP-GCN by Spinelli et al. We also introduce three alternative model architectures that allow each node to dynamically adjust its number of message passing steps. In accordance with AP-GCN, we designed all adaptive propagation variants to always do at least one step of message passing per node. Finally, we adopted Co-GCN from Finkelshtein et al., as it enables each node to change its message passing behavior based on its state.

In traditional message passing graph neural networks, the number of propagation steps is identical for all nodes in the graph. This uniformity implies that each node aggregates information from its neighbors through a fixed number of iterations, regardless of its individual characteristics or position within the graph structure. Spinelli et al. focus on the case where the message passing is implemented by a Graph Convolution Network (GCN) [8]. In GCN, the embedding for node $\mathbf{z}_i$ after k message passing iterations is:

$$\mathbf{z}_i^k = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}} \left(\mathbf{W}^\top \cdot \mathbf{z}_j^{k-1}\right) + \mathbf{b} \quad (1)$$

3.1 AP-GCN

The proposed Adaptive Propagation GCN (AP-GCN) enables each node in the graph to perform a distinct number of message passing iterations, with the number being learned during training rather than set in advance. AP-GCN is inspired by the adaptive computation time in RNNs [9].

A classifier is added to each node that predicts, based on the hidden embedding of a node after k message passing steps, whether the propagation should stop. That is, the classifier predicts whether a node should stop aggregating information from its neighbors after k message passing steps. The output of the classifier, h_i^k, is the probability the propagation should stop for node i after having performed k steps of message passing.

$$h_i^k = \sigma(\mathbf{a}\mathbf{z}_i^k + b) \quad (2)$$

$\mathbf{a}$ and b are trainable parameters and are shared between all nodes. There are two ways the propagation can stop. First, the propagation stops if the number of propagations reaches a predefined maximum number of allowed steps T. Second, the propagation stops when the propagation budget has been used up. The budget is defined as $1 - \epsilon$ where ϵ is a hyperparameter set to a small value. When $k = K_i$ the budget for node i has been reached after k iterations and the propagation stops. K_i, the last permitted step before the budget is used up, is defined as:

$$K_i = \arg\min_k \sum_{k=1}^{k} h_i^k \geq 1 - \epsilon \quad (3)$$

According to Spinelli et al. the halting probabilities can be combined as

$$p_i^k = \begin{cases} R_i = 1 - \sum_{k=1}^{K_i - 1} h_i^k & \text{if } k = K_i \text{ or } k = T \\ \sum_{l=1}^{k} h_i^l & \text{otherwise} \end{cases} \quad (4)$$

so that the sequence $\{p_i^k\}$ for $k \in \{1, 2, ..., \min(K_i, T)\}$ forms a *cumulative probability distribution* (CDF) for the halting probabilities $\{h_i^k\}$ for $k \in \{1, 2, ..., \min(K_i, T)\}$. However, the sequence $\{p_i^k\}$ does not constitute a CDF, because for $\{p_i^k\}$ to be a valid CDF, it must satisfy the condition $p_i^k = 1$ when $k = \min(K_i, T)$. This requirement arises from the fact that a CDF must converge to 1 at its maximum value, reflecting the total probability mass. Clearly, the definition by Spinelli et al., in general, violates this constraint. It could be that the error arose because AP-GCN is inspired by Graves [9] where in equation 6, Graves defines the $\{p_t^n\}$ to form a *probability mass function* (PMF) for the $\{h_t^n\}$ in a similar way. Since preliminary experiments showed that there is no large difference in accuracy when addressing this issue, we remained consistent with the description of Spinelli et al. in our implementation. In the original of Equation 4 by Spinelli et al. (equation 8), there is a second mistake: They sum from $k = 1$ to K_i in the "otherwise" branch. However, this is just a typo in the paper, in the code they do it as we describe it in Equation 4.

The output of the AP-GCN message passing layer for node i is $\hat{\mathbf{z}}_i$ which is defined as

$$\hat{\mathbf{z}}_i = \frac{1}{K_i} \sum_{k=1}^{K_i} p_i^k \mathbf{z}_i^k + (1 - p_i^k)\mathbf{z}_i^{k-1} \quad (5)$$

Although Spinelli et al. do not state it, this way of accumulating the contribution of the embeddings from the different propagation steps is likely inspired by the soft-conditional output from Scardapane et al. [10].

The propagation cost $\mathcal{S}_i$ for node i is given by the sum of K_i and R_i. Adding $\mathcal{S}_i$ to the loss term incentivizes the model to choose a limited number of propagation steps for node i. The propagation penalty α controls the trade-off between accuracy and compute time. A larger α encourages the model to restrict the number of propagation steps per node, whereas a smaller α allows for deeper message passing, potentially improving accuracy at the cost of increased computation. Spinelli et al. decided to update the halting unit once every five epochs. However, the parameters unrelated to adaptive propagation are updated every epoch as usual. Thus, the final loss is $\hat{\mathcal{L}} = \mathcal{L} + \alpha \sum_{i \in \mathcal{V}} \mathcal{S}_i$.

3.2 RL-AP-GCN

In Graph Neural Networks, node representations tend to homogenize and converge as message passing progresses, a phenomenon known as over-smoothing. Rather than treating this as a limitation, this behavior can serve as a useful signal for an adaptive halting mechanism. Specifically, a reinforcement learning (RL) agent can observe how a node's embedding evolves across propagation steps and learn to halt when changes become minimal, ideally capturing the point of maximum expressiveness for that node. This stabilization suggests diminishing returns from additional message passing, indicating that sufficient neighborhood information has already been aggregated. By relying solely on the task reward (e.g., classification accuracy), the policy network can learn to stop propagation at this convergence point, without requiring explicit regularization terms or penalties tied to the number of steps. In doing so, the halting decision becomes a function of information saturation rather than an externally imposed computational constraint.

The reinforcement learning architecture treats each node as an actor critic agent but the policy and value networks are shared among all nodes. We first compute the initial node embeddings.

Then, a loop begins, where, at each step, the shared policy network (actor) evaluates the current embedding of active nodes to output a halting probability, while a shared value network (critic) estimates the expected future reward (baseline) from that state. A halting action is sampled (incorporating exploration noise during training), and propagation continues for non-halted nodes. Upon halting (or reaching the maximum iterations), a reward is calculated based primarily on the final classification correctness and the system learns by backpropagating a combined loss: a supervised loss for the GNN's classification output, a RL loss comprising a policy gradient term (using the advantage calculated as actual reward minus the critic's baseline), a value function loss for the critic, and an entropy bonus to encourage exploration. In more detail, the corresponding operations are:

Halting policy. Let $\mathbf{z}_i^k \in \mathbb{R}^h$ be the embedding of node i after k message passing iterations. At every step the policy network produces the halting probability:

$$\pi_i^k = \sigma\big(g_\theta(\mathbf{z}_i^k)\big) \tag{6}$$

where the parameters θ are shared across all nodes. In addition, exploration noise, with the corresponding exploration factor hyperparameter, is injected by perturbing π_i^k with Gaussian noise that decays exponentially with the epoch number. This means that even if the policy network is fairly certain a node should continue propagating, the added noise might, for example, push high enough for the node to halt and vice-versa, resulting in a broader exploration of halting steps. A Bernoulli action is sampled

$$a_i^k \sim \text{Bernoulli}(\pi_i^k) \tag{7}$$

with $a_i^k = 1$ meaning *halt*. Propagation for node i stops at the first step such that:

$$K_i = \min\{k \le T \mid a_i^k = 1\}, \tag{8}$$

where T is a hard cap on the number of iterations.

Reward and advantage. After halting, node i receives a reward:

$$r_i = \mathbb{I}[\hat{y}_i = y_i] - cK_i \tag{9}$$

where $c \ge 0$ is the computation penalty coefficient — that we set to 0 during our experiments to test our hypothesis that the computation penalty is not required. A shared value network V_ϕ provides a baseline, and the resulting advantage is:

$$A_i = r_i - V_\phi(\mathbf{z}_i^{K_i}) \tag{10}$$

Loss function. Denote by $\mathcal{V}$ the set of training nodes. The overall objective for $\mathcal{L}$ to be minimized combines a standard supervised term (plus L2 regularization $\mathcal{L}_{\text{reg}}$ applied separately) and three reinforcement-learning components, averaged over the training nodes:

$$\mathcal{L} = \left(\frac{1}{|\mathcal{V}|}\sum_{i\in\mathcal{V}} \mathcal{L}_{\text{NLL}}(y_i, \hat{y}_i)\right) + \mathcal{L}_{\text{reg}} \tag{11}$$

$$- \frac{1}{|\mathcal{V}|}\sum_{i\in\mathcal{V}} \log \pi_i^{K_i} A_i^{\text{detach}} \tag{12}$$

$$+ \lambda_v \frac{1}{|\mathcal{V}|}\sum_{i\in\mathcal{V}} \big(r_i - V_\phi(\mathbf{z}_i^{K_i})\big)^2 \tag{13}$$

$$- \lambda_e \frac{1}{|\mathcal{V}|}\sum_{i\in\mathcal{V}} H(\pi_i^{K_i}) \tag{14}$$

with the binary entropy $H(p) = -\big[p\log p + (1-p)\log(1-p)\big]$. Here $\mathcal{L}_{\text{NLL}}(y_i, \hat{y}_i)$ is the negative log-likelihood (NLL) loss for node i, and $\mathcal{L}_{\text{reg}}$ represents the L2 regularization term (e.g., $\frac{\lambda_{\text{reg}}}{2}\|\theta_{\text{reg}}\|^2$ applied to specific parameters θ_{reg}). A_i^{detach} represents the advantage $r_i - V_\phi(\mathbf{z}_i^{K_i})$ where the gradient is not propagated back through the value function estimate $V_\phi(\mathbf{z}_i^{K_i})$ during the policy update (as indicated by "detach") to ensure stable training by preventing the policy update from interfering with the value function update, as the policy might try to manipulate the value estimates to make its own actions look better, rather than simply improving the policy based on the current value estimate. Hyperparameters λ_v (corresponding to "value_weight" in the code) and λ_e (corresponding to "entropy_weight") weight the value-regression and entropy-regularization terms, respectively. The signs reflect the objective minimization: The negative sign in Term 12 implements policy gradient ascent (maximizing expected rewards), and the negative sign in Term 14 encourages entropy maximization (promoting exploration).

There are several extensions that could enhance the implementation. The sparse, delayed reward signal could be augmented with denser, intermediate rewards reflecting confidence gains at each step. The RL state representation could be enriched beyond the current embedding to include neighboring node embeddings difference or a history of embeddings (using RNNs or attention), giving the policy more context. Employing more advanced RL algorithms such as PPO for policy updates or GAE for advantage estimation could improve training stability and sample efficiency compared to the current REINFORCE-with-baseline approach. Finally, refining the training strategy, perhaps by alternating between supervised and RL optimization steps, could further stabilize learning and potentially improve final performance.

3.3 Ponder-AP-GCN

Strongly inspired by PonderNet [2], we adopted its core ideas to implement a halting mechanism for GCN. We define λ_i^k as the conditional probability that node i halts after k message passing steps given that node i has not halted before. Then, the unconditional probability of node i halting after k message passing iterations is

$$p_i^k = \lambda_i^k \prod_{j=1}^{k-1}(1-\lambda_i^j) \tag{15}$$

which is similar to the geometric distribution but with variable λ. This would be a valid distribution if we integrated over all possible message passing steps, which is infeasible in practice. For this reason, we define a maximum number of propagation steps T and assign all remaining probability to the last step. This ensures that the $\{p_i^k\}$ for $k = \in \{1, 2, ..., T\}$ sum to 1. The embedding for node i and therefore implicitly also how many message passing steps are done is then sampled from the random variable $\mathcal{Z}_i$ with probability density distribution $P(\mathcal{Z}_i = \hat{\mathbf{z}}_i^k) = p_i^k$.

The model is trained end-to-end by minimizing the loss function that consists of a sum of two terms, the construction loss and the regularization loss, weighted by the hyperparameter β:

$$\mathcal{L} = \frac{1}{|\mathcal{V}|}\sum_{i=1}^{|\mathcal{V}|}\sum_{k=1}^{T} \underbrace{p_i^k \mathcal{L}(y_i, \hat{y}_i^k)}_{\text{reconstruction}} + \beta \underbrace{KL(\mathbf{p}_i || p_G(\lambda_p))}_{\text{regularization}} \tag{16}$$

where $|V|$ is the number of nodes in the graph.

The reconstruction loss per node is the cross-entropy loss averaged over the T steps weighted by their corresponding halting probability $\mathbf{p}_i$. The regularization term per node incentivizes the distribution $\mathbf{p}_i$ to stay close to a predefined prior geometric distribution p_G with parameter λ_p. In comparison to AP-GCN, this regularization term has the advantage that it is easily interpretable. In particular, setting the prior parameter $\lambda_p = \frac{1}{N}$ encourages the model towards halting after an average of N steps. A disadvantage of Ponder-AP-GCN, compared to AP-GCN, is that during training it must compute the halting distribution over all T steps for each node, whereas AP-GCN can stop computations as soon as a node halts. However, since all the computations in AP-GCN are implemented using matrix operations to deal with all the nodes simultaneously, it is questionable whether this is an actual advantage in practice.

Focusing on the implementation in code, the model architecture starts with an MLP that processes the node features, followed by the recurrent GCN propagation steps — similar to the AP-GCN. A key component is the shared linear halting unit that computes logits at each step, which are passed through a sigmoid function to yield the conditional halting probabilities λ_i^k. The unconditional halting distribution $\mathbf{p}_i$ is then derived iteratively from these λ_i^k values, ensuring the probabilities sum to one over the maximum N steps. The training loss combines the expected loss (weighted by $\mathbf{p}_i$) and the KL divergence between the learned $\mathbf{p}_i$ and the geometric prior p_G, scaled by β. Inference can be performed either by first computing the full distribution $\mathbf{p}_i$ and selecting the step via argmax or sampling, or through the computationally efficient sequential Bernoulli sampling at each step as proposed in the PonderNet paper. We implemented the first approach and use the temperature parameter to control the randomness in the model's output. A low temperature makes the model more likely to select high-probability halting steps from $\mathbf{p}_i$, whereas a high temperature increases the likelihood of selecting lower-probability halting steps.

3.4 Gumbel-AP-GCN

The Gumbel-AP-GCN architecture is based on the Ponder-AP-GCN architecture but instead of calculating the loss as an expectation over the halting steps as in Equation 16, a halting step h_i for node i is sampled from the learned distribution $\mathbf{p}_i$ of halting steps. We use the Gumbel-Softmax technique [11, 12] to draw a sample in a differentiable way. Thus, the halting step for node i is defined as

$$h_i = \arg\max \text{GumbelSoftmax}(\{\log p_i^1, ..., \log p_i^T\}, \tau) \qquad (17)$$

Where τ is the temperature parameter controlling how sharp (low τ) or smooth (high τ) the output distribution is. We apply temperature annealing by starting with a higher temperature τ_{initial} to encourage a uniform exploration in the early stages of training. τ_{warmup} defines the number of epochs during which the temperature is kept constant at τ_{initial}. Gradually it is lowered to produce less uniform distributions as training progresses and τ_{decay} specifies then the number of epochs over which the temperature is linearly annealed from τ_{initial} to its τ_{final}. Therefore, $\tau = \tau_{\text{initial}}$ during *warmup* period, $\tau = \tau_{\text{final}}$ after *warmup* and *decay*. Between these two phases, τ is defined as follows:

$$\tau = \tau_{\text{initial}} - \frac{\text{epoch} - \tau_{\text{warmup}}}{\tau_{\text{decay}}} \cdot (\tau_{\text{initial}} - \tau_{\text{final}}) \qquad (18)$$

Instead of computing the expected loss over all halting steps like in Ponder-AP-GCN, we compute it from the output corresponding to the sampled step h_i:

$$\mathcal{L} = \frac{1}{|\mathcal{V}|} \sum_{i=1}^{|\mathcal{V}|} \underbrace{p_i^{h_i} \mathcal{L}(y_i, \hat{y}_i^{h_i})}_{\text{reconstruction}} + \beta \underbrace{KL(\mathbf{p}_i || p_G(\lambda_p))}_{\text{regularization}} \qquad (19)$$

The KL divergence hyperparameter is set to 0, so that the halting policy is learned solely by optimizing the task performance at the sampled steps without explicit regularization.

The inference is implemented similarly to the training. A halting step is sampled using the GumbelSoftmax function and the output corresponding to that halting step is returned. As in the Ponder-AP-GCN, the inference could be implemented more efficiently using sequential Bernoulli sampling.

3.5 Co-GCN

The Cooperative Graph Neural Network (Co-GNN) [3] architecture is different from the other previously mentioned architectures in the way that we allow a different type of flexibility in message passing. In particular, we train a policy that decides which state to assign to a node — Standard (S), Broadcast (B), Listen (L), or Isolate (I). These states can be reassigned at each message passing step, so a state that previously was in Isolate could become Standard for the next message passing step, and all combinations thereof. In a sense, the action network modifies the graph in each step, where undirected edges may become directed or simply nonexistent, see Figure 1. In the Standard state, a node both broadcasts to neighbors that are listening and listens to neighbors that are broadcasting. In the Listen state, the node only listens to neighbors that are broadcasting. In the Broadcast state, the node only broadcasts to neighbors that are listening. In the Isolate state, the node neither broadcasts nor listens.

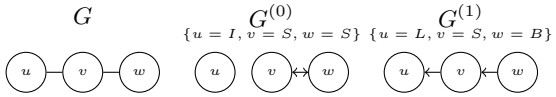


Figure 1. A graph G and the states $s \in \{S, B, L, I\}$ potentially assigned by a policy at different message passing layers.

A benefit of the dynamically changing graph structure in each layer is that the policy can adjust how to propagate the information at each layer, i.e., for long range tasks it may enable edges while for short range tasks it may isolate most edges for the final propagation layers.

Note that in contrast to the other AP-techniques, the number of propagation is fixed in Co-GNN via the number of layers that the environment network is composed of, however it can still work in a similar manner if the action network were to choose to isolate a node in all successive layers.

In addition, the Co-GNN approach is quite modular since we are able to choose various different architectures for the action and environment networks. In our case, to make the experiments more comparable we will restrain ourselves to use only a GCN architecture, but it is entirely feasible to use other GNN architectures such as GATs [13], GINs [14] etc. as well. For this reason we denote this architecture as Co-GCN instead of Co-GNN.

4 Implementation

We implemented AP-GCN based on the description in the paper and compared our implementation with the reference code provided

by Spinelli et al., available on their GitHub repository*. We found that in cases where nodes reach the maximum propagation limit T and are eligible to propagate further, the implementation by Spinelli et al. incorrectly reports one additional step, resulting in an off-by-one error. Importantly, the implementation does not actually perform an extra propagation step, it only misreports the number of steps taken. For the loading of the datasets, we used the code provided by Spinelli et al.

We asked Prof. Spinelli, among other questions, why they placed the feature mapping layer that projects to the number of classes before the message passing layer. However, we did not receive a response to our email.

4.1 Datasets

We downloaded the Citeseer [15], Cora-ML [16], PubMed [17], MS-Academic [18], A. Computer, and A. Photo datasets from the GitHub repository of Spinelli et al. along with the code to load and process them. The A. Computer and A. Photo datasets are portions of the Amazon co-purchase graph originally presented in [19]. In the MS-Academic dataset, the edges represent co-authorship while in the Citeseer, Cora-ML, and PubMed datasets, the edges represent citations. All these datasets consists of a single graph and are node classification problems. Only the largest connected component in each dataset is used. We calculated the statistics for each dataset, as shown in Table 1. Except for a few values in the average degrees, which may be due to rounding, our results are consistent with those reported by Spinelli et al. Note that the homophily ratio was not reported by Spinelli et al. All the graphs are quite strongly homophilic.

Table 1. Dataset Statistics.

Dataset	Classes	Features	Nodes	Edges	Avg. Degr.	Homo. Ratio
Citeseer	6	3703	2110	3668	6.95	0.74
Cora-ML	7	2879	2810	7981	11.36	0.78
PubMed	3	500	19717	44324	8.99	0.80
MS-Academic	15	6805	18333	81894	17.87	0.81
A.Computer	10	767	13381	245778	73.47	0.78
A.Photo	8	745	7487	119043	63.60	0.83

4.2 Hyperparameters

In the experiments with AP-GCN we used the hyperparameters described in the paper by Spinelli et al. They are described in Appendix A.1. In particular, we used the same propagation penalties as Spinelli et al. The hyperparameters used in the experiments with RL-AP-GCN, Ponder-AP-GCN, Gumbel-AP-GCN, and Co-GCN are available in Appendices A.2, A.3, A.4, and A.5. We did not perform systematic hyperparameter optimization for RL-AP-GCN, Ponder-AP-GCN, Gumbel-AP-GCN, and Co-GCN. Instead, we mostly tested each model on the Cora-ML dataset and once we obtained reasonably good hyperparameters we reused them for the other datasets.

4.3 Experimental setup

We follow the experimental setup of Spinelli et al., splitting each dataset into a development and a test set. Spinelli et al. follow the experimental setup of [20]. The test set serves to evaluate the final model performance. The development set is split into the training and validation set. The training set consists of 20 instances per class sampled from the development set. The rest of the development set is assigned to the validation set which is used for early stopping. For the MS-Academic dataset, the development set consists of 5,000 samples, whereas for all other datasets it contains 1,500 samples. This larger size is chosen for MS-Academic because a smaller development set of 1,500 samples would likely result in fewer than 20 samples per class.

For each dataset, we use 20 random seeds to sample a training set from the development set, and for each seed, a model is trained five times with different weight initializations, resulting in 100 trained models per dataset for a fixed set of hyperparameters. While Spinelli et al. provided the 20 seeds for the sampling of the training sets, there is still a degree of randomness involved because Spinelli et al. did not provide seeds for the model initializations.

Our code is available on GitHub†.

4.4 Computational requirements

The experiments involving the adaptive propagation architectures were carried out on a MacBook Pro equipped with an Apple M2 Pro chip and 16 GB of memory, using the MPS backend in PyTorch. We encountered a peculiar issue with the Co-GCN model on this backend: After a certain number of epochs, the gradient began to contain NaN (not-a-number) values. Despite trying several potential fixes, the problem persisted. However, when switching to a different backend, either CPU or CUDA, with the same hyperparameters, the issue did not occur. As a result, we opted to run the Co-GCN models on machines with a Nvidia GPU. Specifically, experiments on the Amazon datasets were conducted on a machine with a Tesla T4 GPU (16 GB VRAM), while those on all other datasets were run on a GeForce RTX 3050 Ti Laptop (4 GB VRAM). Although the hardware differences make direct comparisons of the training time per epoch challenging, all systems used are consumer-grade and relatively similar in capability.

Table 2 shows the compute time per model architecture and the total compute time. Note that the reported compute times only cover the main experiments described in Section 4.3 not the compute time related to debugging the model architectures and the setup of the benchmarking code.

Table 2. Compute time by model architecture and total compute time.

Model Architecture	Total Compute Time [h]
RL-AP-GCN	20.23
Ponder-AP-GCN	8.77
Gumbel-AP-GCN	11.49
Co-GCN	8.10
AP-GCN replication	7.63
Total	56.22

5 Results

We report the results of our experiments with the AP-GCN, RL-AP-GCN, Ponder-AP-GCN, Gumbel-AP-GCN, and Co-GCN architectures. Table 3 shows the average accuracy, with uncertainties showing the 95% confidence level calculated by bootstrapping. The uncertainty is defined as the larger absolute difference between the mean and the 2.5th or 97.5th percentile of the bootstrap distribution. This follows the code in the Jupyter notebook provided by Spinelli et al. The accuracies of the replicated AP-GCN are consistently lower than the ones reported by Spinelli et al. The largest difference of 2.02% occurs on the Cora-ML

*https://github.com/spindro/AP-GCN

†https://github.com/TobiasErbacher/gdl

dataset. Given that we did not use the exact same setup (library versions and seeds for the model initializations) as Spinelli et al. we think these differences are acceptable.

In Figure 2, we report the distribution of the halting steps, that is, the steps at which each node halts for the adaptive propagation model architectures. Since we have data from 100 runs per model architecture, we first average the halting steps over the 100 runs for each node and then plot the distribution obtained using kernel density estimation (KDE) over these averaged halting steps per node. Note that because of the averaging and the KDE there can be probability mass at non-integer values even though in reality all the models can only halt at an integer number of steps. This approach follows the methodology used by Spinelli et al. The step distribution for AP-GCN has a spike for the MS-Academic (ca. 2.6) and A.Computer (ca. 3.9) datasets at around 2 steps. All other models do not show such pronounced spikes. Furthermore, the density distributions of different datasets under AP-GCN differ significantly from those of other model architectures, where the peaks of the distributions are typically closer together. In Gumbel-AP-GCN the densities for all datasets are almost identical and centered around 3.5 steps. We believe that one reason behind this phenomenon is that it requires a more explorative set of parameters with a higher initial τ. Since we did not perform an extensive search for these hyperparameters, besides a hand-picked choice based on a few validation loss samples in an individual dataset, the results indicate that further exploration could increase the difference between the step distributions among the datasets. Subsequent tests indicate that using a higher initial τ combined with a shorter warmup period may be a more effective alternative to the hyperparameters used in our results. An important result is that the datasets do not enforce a certain step distribution independent of the model architecture. The reason for this could be that the optimal number of message passing iterations per node depends on the behavior of the neighboring nodes. Consequently, we suspect that there might exist many good combinations of halting steps for a graph.

Figure 3 shows the distribution of the standard deviations of the halting steps per node over the 100 experiments for each dataset. A low median standard deviation is desirable since it shows that a model often returns similar halting steps for the same node over experiments with different seeds and weight initialization. Furthermore, it is desirable that the median standard deviation is consistent across the different datasets. AP-GCN tends to have a very low standard deviation per node. RL-AP-GCN has a high standard deviation per node for each dataset. Interestingly, the medians of the boxplots are almost identical (around 3.6). Ponder-AP-GCN has low median standard deviation per node in the Cora-ML, MS-Academic, A.Computer, and A.Photo datasets and high median standard deviation per node in the Citeseer and PubMed datasets. Gumbel-AP-GCN has a high standard deviation per node but still lower than RL-AP-GCN.

For the Co-GCN we tracked how the nodes are distributed among the four possible states at each step instead of reporting the distribution of halting steps. Figure 4 displays boxplots representing the variation in the proportion of nodes assigned to each state (Standard, Broadcast, Listen, and Isolate) across 100 independent runs, for each dataset and step. In the experiments on Citeseer, Cora-ML, PubMed, and MS-Academic, the majority of nodes remain in the standard state across nearly all 100 runs and at every step. In contrast, for the A.Computer and A.Photo experiments, nodes are nearly evenly distributed across different states at each step. Interestingly, the models perform significantly worse on these two datasets compared to the other four. Our hypothesis regarding the different state distributions is that due to the main difference of these two datasets compared to the others being the average degree of a node, the policy will assign the state Standard to fewer nodes in order to single out the important nodes for a cleaner information flow and thus more nodes get assigned to the remaining states. Moreover, for higher degree graphs, we expected that proportionally fewer nodes will be required to preserve proper information flow through the graph via redundant paths.

The average training time per epoch for each model is shown in Table 4. While the training times we obtained for AP-GCN differ slightly from those reported by Spinelli et al., this is likely due to differences in hardware; nonetheless, the values remain comparable. Across most datasets, Ponder-AP-GCN is nearly as fast as our AP-GCN replication and even outperforms it on the Cora-ML and PubMed datasets. Unsurprisingly, RL-AP-GCN has the longest training times. This is due to the reinforcement learning implementation, which involves iterative, per-node calls to the policy and value networks at each step for every active node — each followed by backpropagation through both networks. Additionally, RL-specific loss components further extend the training process, making each epoch particularly computationally intensive. Lastly, the variation in training time per epoch across datasets is more pronounced for Co-GCN than for AP-GCN.

6 Discussion and conclusion

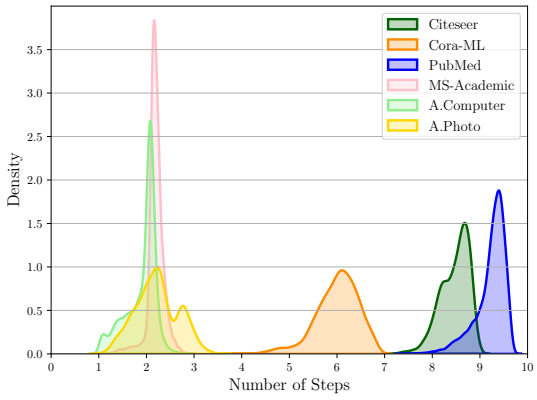
Although AP-GCN performs well in practice, several of its design choices lack clear theoretical justification in our opinion. For example, Spinelli et al. refer to h_i^k as the probability that a node should halt after k iterations. However, this is not what actually happens in the model: A node halts only when its propagation budget is used up or the maximum number of steps is reached. For instance, assume we have $\mathbf{h}_i = [0.3, 0.4, 0.25]$ and a propagation budget of 0.9. AP-GCN will not halt at node i with probability 0.3 after the first step. In fact, it will never halt after the first step. It will only halt after the third step, once the budget is fully consumed. Similarly, the way embeddings are aggregated over steps in Equation 5 lacks a clear theoretical foundation.

We particularly appreciate how seamlessly the PonderNet framework can be adopted to adaptive propagation within graph convolutional networks. Unlike AP-GCN, Ponder-AP-GCN rests on a solid theoretical basis. Although its empirical performance trails AP-GCN slightly, more systematic hyperparameter optimization could possibly allow Ponder-AP-GCN to surpass AP-GCN.

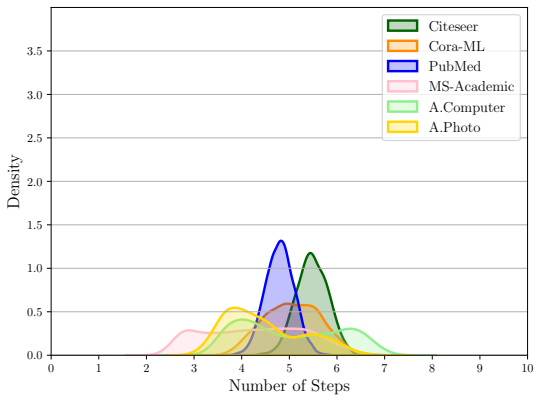
Experiments with the RL-AP-GCN and Gumbel-AP-GCN architectures showed that competitive results can be achieved without explicitly incorporating a computation penalty. This suggests that these models are capable of identifying a suitable halting point by recognizing when additional message-passing steps yield diminishing returns.

For Co-GCN, future work could explore modifying the architecture to share the environment network across steps, thereby reducing the overall parameter count and harnessing potential synergy effects.

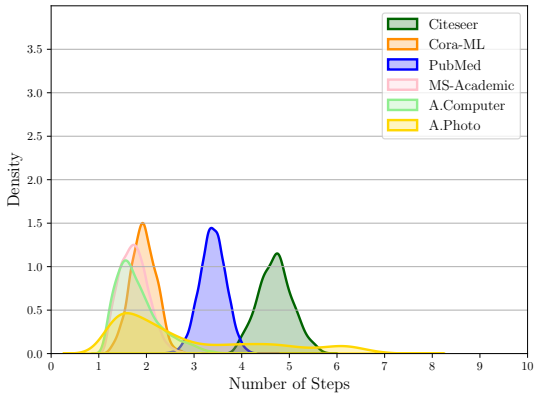
Lastly, training the models on datasets comprising a greater number of training instances would be interesting to account for intra-class variance.



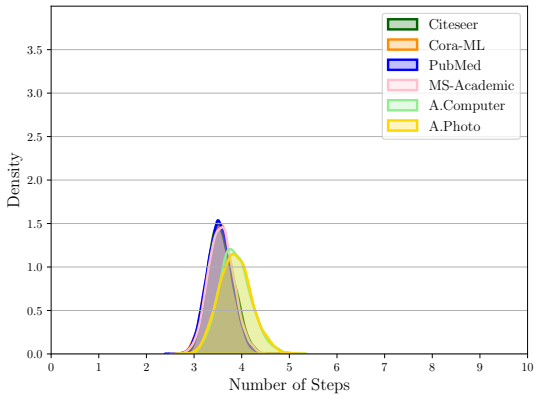
(a) AP-GCN Replication



(b) RL-AP-GCN

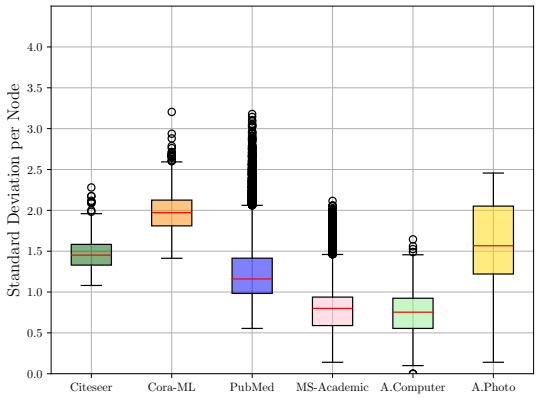


(c) Ponder-AP-GCN

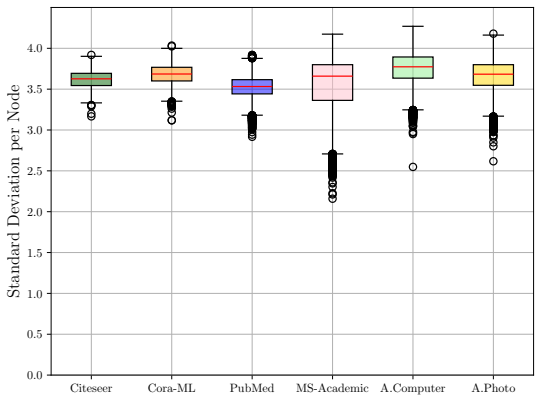


(d) Gumbel-AP-GCN

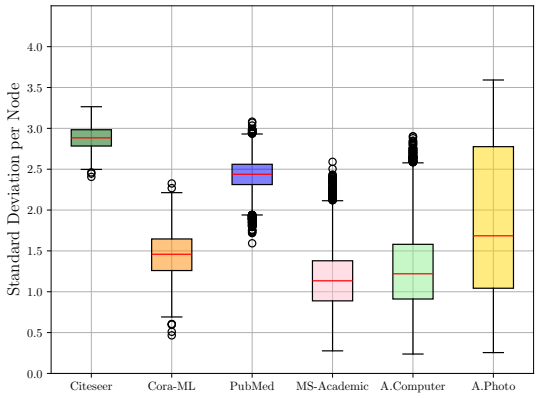
Figure 2. Average density distribution of the halting steps for the adaptive propagation model architectures for each dataset.



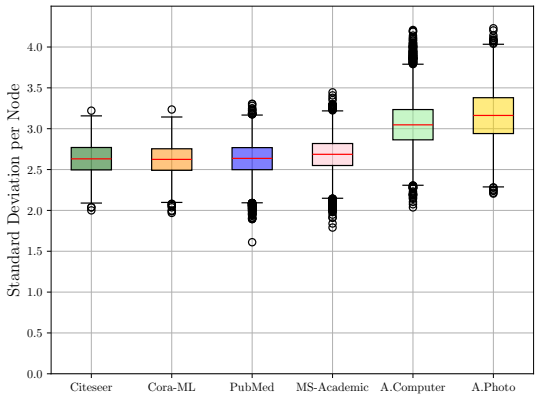
(a) AP-GCN Replication



(b) RL-AP-GCN



(c) Ponder-AP-GCN



(d) Gumbel-AP-GCN

Figure 3. Standard deviations of the halting steps per node over the 100 experiments for each dataset.

Table 3. Average test set accuracies with 95% confidence intervals, computed from 100 runs per model architecture using 20 random seeds and 5 different model initializations per seed.

Model	Citeseer	Cora-ML	PubMed	MS-Academic	A.Computer	A.Photo
RL-AP-GCN	74.66 ± 0.30	81.63 ± 1.43	77.18 ± 0.44	88.32 ± 0.45	79.97 ± 0.56	88.88 ± 0.40
Ponder-AP-GCN	74.80 ± 0.33	82.14 ± 0.28	76.37 ± 0.43	91.29 ± 0.12	83.41 ± 0.27	91.28 ± 0.23
Gumbel-AP-GCN	70.12 ± 1.43	70.86 ± 2.50	76.85 ± 0.41	87.56 ± 0.41	79.13 ± 0.55	89.22 ± 0.39
Co-GCN	68.00 ± 0.77	74.80 ± 2.14	75.89 ± 1.04	88.58 ± 0.20	30.47 ± 1.16	34.42 ± 3.57
AP-GCN replication	75.87 ± 0.27	83.69 ± 0.34	78.97 ± 0.41	91.65 ± 0.13	83.25 ± 0.34	90.69 ± 0.32
AP-GCN	76.12 ± 0.24	85.71 ± 0.22	79.80 ± 0.34	92.62 ± 0.07	85.18 ± 0.23	92.05 ± 0.22

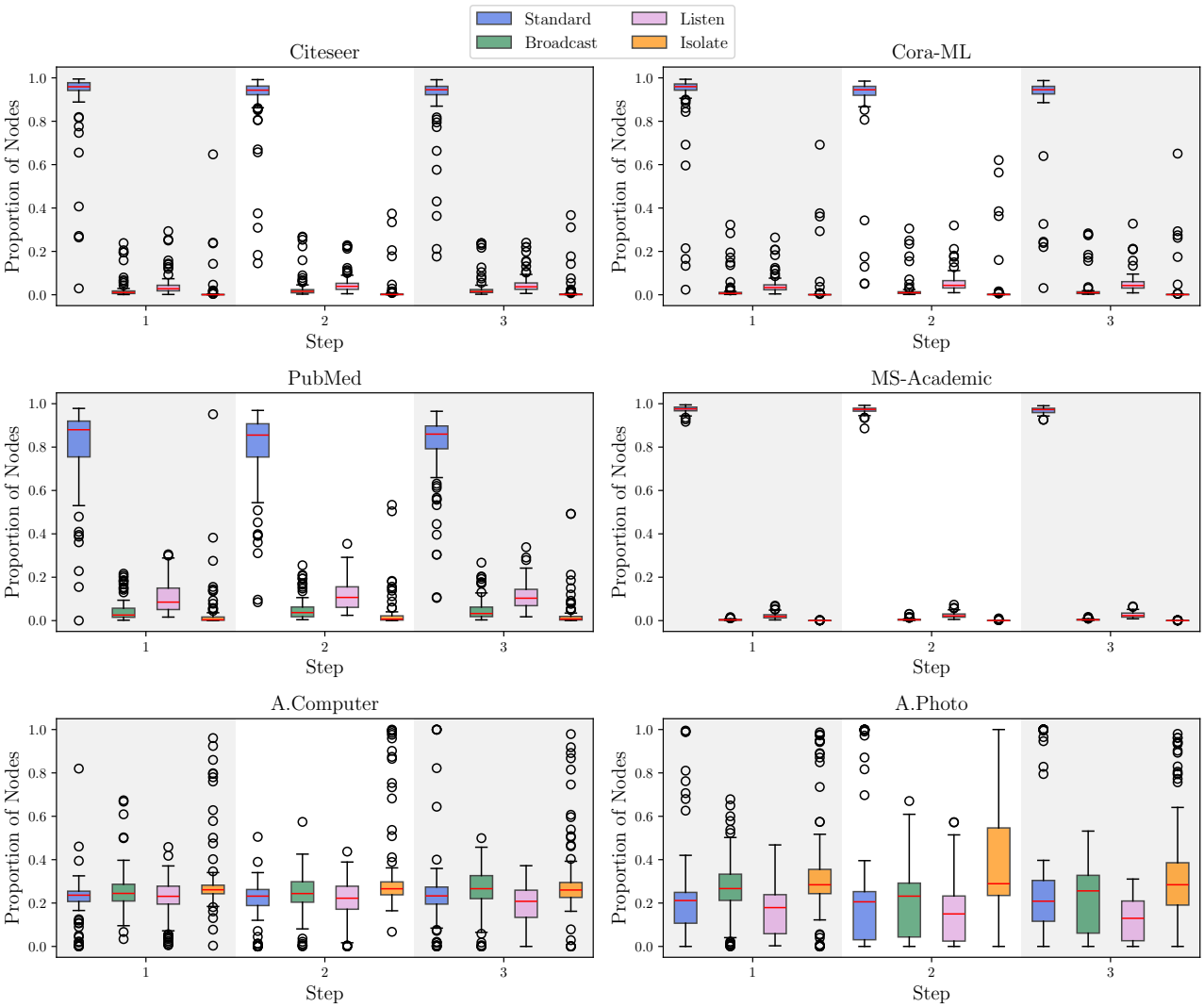


Figure 4. Each figure shows the distribution of the proportion of nodes in a given state per layer of the Co-GCN models for a dataset.

Table 4. Average training time per epoch in milliseconds, computed from 100 runs per model using 20 random seeds and 5 different model initializations per seed.

Model	Citeseer	Cora-ML	PubMed	MS-Academic	A.Computer	A.Photo
RL-AP-GCN	169.0	161.6	221.4	279.0	279.8	206.1
Ponder-AP-GCN	54.1	51.1	67.9	136.4	173.6	87.9
Gumbel-AP-GCN	81.5	84.8	84.9	157.3	188.8	112.5
Co-GCN	52.4	52.4	213.5	131.6	290.4	236.0
AP-GCN replication	48.1	51.5	79.1	117.5	113.5	78.6
AP-GCN	32.4	36.2	42.0	100.3	76.7	50.0

References

[1] Indro Spinelli, Simone Scardapane, and Aurelio Uncini. Adaptive propagation graph convolutional network. *IEEE Transactions on Neural Networks and Learning Systems*, 32 (10):4755–4760, 2021.

[2] Andrea Banino, Jan Balaguer, and Charles Blundell. Pondernet: Learning to ponder, 2021. URL https://arxiv.org/abs/2107.05407. 8th ICML Workshop on Automated Machine Learning (AutoML 2021).

[3] Ben Finkelshtein, Xingyue Huang, Michael Bronstein, and İsmail İlkan Ceylan. Cooperative graph neural networks. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[4] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5453–5462. PMLR, 10–15 Jul 2018.

[5] Ziqi Liu, Chaochao Chen, Longfei Li, Jun Zhou, Xiaolong Li, Le Song, and Yuan Qi. Geniepath: Graph neural networks with adaptive receptive paths. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):4424–4431, Jul. 2019. URL https://ojs.aaai.org/index.php/AAAI/article/view/4354.

[6] Kwei-Herng Lai, Daochen Zha, Kaixiong Zhou, and Xia Hu. Policy-gnn: Aggregation optimization for graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 461–471, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379984. URL https://doi.org/10.1145/3394486.3403088.

[7] Teng Xiao, Zhengyu Chen, Donglin Wang, and Suhang Wang. Learning how to propagate messages in graph neural networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, page 1894–1903, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383325. URL https://doi.org/10.1145/3447548.3467451.

[8] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*, ICLR '17, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

[9] Alex Graves. Adaptive computation time for recurrent neural networks, 2017. URL https://arxiv.org/abs/1603.08983.

[10] Simone Scardapane, Danilo Comminiello, Michele Scarpiniti, Enzo Baccarelli, and Aurelio Uncini. Differentiable branching in deep networks for fast inference. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4167–4171, 2020.

[11] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *Proceedings of the international conference on learning Representations*. International Conference on Learning Representations, 2017.

[12] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *Proceedings of the international conference on learning Representations*. International Conference on Learning Representations, 2017.

[13] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *Proceedings of the international conference on learning Representations*. International Conference on Learning Representations, 2018. URL https://openreview.net/forum?id=rJXMpikCZ.

[14] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *Proceedings of the international conference on learning Representations*. International Conference on Learning Representations, 2019. URL https://openreview.net/forum?id=ryGs6iA5Km.

[15] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93–106, 2008. URL https://onlinelibrary.wiley.com/doi/abs/10.1609/aimag.v29i3.2157.

[16] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3(2): 127–163, 2000. URL https://doi.org/10.1023/A:1009953814988.

[17] Galileo Namata, Ben London, Lise Getoor, and Bert Huang. Query-driven active surveying for collective classification. In *International Workshop on Mining and Learning with Graphs (MLG)*, 2012.

[18] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. In *Relational Representation Learning Workshop (R2L) at NeurIPS*, 2018.

[19] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton van den Hengel. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '15, pages 43–52, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336215. URL https://doi.org/10.1145/2766462.2767755.

[20] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Predict then propagate: Graph neural networks meet personalized pagerank. In *Proceedings of the international conference on learning Representations*. International Conference on Learning Representations, 2019.

A Hyperparameters

The maximum number of steps T was set to 10 for all experiments, except those involving the Co-GCN model architecture.

A.1 AP-GCN

We use 2 feature transformation layers, a dropout rate of 0.5, 64 hidden units, and the Adam optimizer with learning rate 0.01. We applied $L2$ regularization with coefficient 0.008 on the parameters of the first layer. For the A. Photo and A. Computer no regularization was used.

A.2 RL-AP-GCN

We used the Adam optimizer with a learning rate of 0.01. The model employs a dropout rate of 0.5 and 64 hidden units. L_2 regularization with a coefficient of 0.008 was applied to the parameters of the first layer, except for the A. Photo and A. Computer datasets, where no regularization was used. The entropy weight was set to 0.01, and the value weight to 0.5. Gradient norm clipping was applied across all model parameters with a maximum norm of 1.0. Additionally, we set the computation penalty to 0, used an exploration factor of 0.01, and did not employ any scheduling for the exploration penalty.

A.3 Ponder-AP-GCN

We used the Adam optimizer with a learning rate of 0.01. The model incorporates a dropout rate of 0.5, an edge dropout rate of 0.3, and 64 hidden units. Edge dropout was applied only in the section focused on adaptive message propagation. L_2 regularization with a coefficient of 0.008 was applied to the parameters of the first layer, except for the A. Photo and A. Computer datasets, where no regularization was used. We set β to 0.01 and λ_p to 0.2. During inference, a temperature value of 5.0 was used.

A.4 Gumbel-AP-GCN

We employed the Adam optimizer with a learning rate of 0.01. The model uses a dropout rate of 0.5, an edge dropout rate of 0.3, and 64 hidden units. Edge dropout was applied exclusively in the adaptive message propagation component. L_2 regularization with a coefficient of 0.008 was applied to the parameters of the first layer, except for the A. Photo and A. Computer datasets, where no regularization was used. We set β to 0, indicating that the model learns without a geometric prior, rendering λ_p irrelevant. During training we use a scheduler for τ with $\tau_{\text{initial}} = 10.0$, $\tau_{\text{warmup}} = 20.0$, $\tau_{\text{decay}} = 50.0$, and $\tau_{\text{final}} = 1.0$. During evaluation, τ was fixed at 0.01.

A.5 Co-GCN

For this approach, in all experiments we set the optimizer to be Adam with a learning rate of 0.005 and apply a weight decay of 0.003 across all parameters. During training, we did not perform gradient clipping. Apart from this, the hyperparameters for the temperature, environment and action networks are shown in Table 5.

Table 5. Co-GCN Hyperparameters.

Parameter	Value
Temperature Network	
Learnable Temperature	`True`
Model Type	`GCN`
τ_0	0.5
`GIN MLP` Function	`None`
Action Network	
Model Type	`GCN`
Hidden Units	64
# Message Passing (Hidden) Layers	2
Activation Function	`ReLU`
Dropout Probability	0.5
`GIN MLP` Function	`None`
Environment Network	
Model Type	`GCN`
Hidden Units	32
# Message Passing (Hidden) Layers	3
# Decoder Layers	1
Activation Function	`ReLU`
Dropout Probability	0.5
Layer Normalization	`True`
Residual Connections	`False`
Dataset Encoder	`None`
`GIN MLP` Function	`None`