CS224W Final Project: Policy-GNN/Policy-GAT

Sam Lowe
Department of Computer Science
Stanford University
Stanford, CA 94305
samlowe@stanford.edu

1 Abstract

Graph-based machine learning is a paradigm that has seen increased focus in recent years due to advances in deep learning techniques for graphs and their widespread application to domains such as chemistry, biology, and social psychology. However, many complications arise in these models with increased depth. While additional layers can help to produce more informative features, they come at a cost of increased computational complexity and introduce the threat of oversmoothing. Policy-GNN [1], presented in the 2020 KDD paper *Policy-GNN: Aggregation Optimization for Graph Neural Networks*, is a meta-policy framework designed to address these issues by dually training a policy that selects the number of layers of aggregation needed to classify graph features and a flexible graph neural network (GNN) model that can perform the given number of aggregation steps. Since the paper presented Policy-GNN as a framework, the given implementation was built on standard models used in graph machine learning and deep reinforcement learning research, Graph Convolutional Networks [2] and Double Deep Q-Learning [3]. In this work, I apply Policy-GNN to the setting of node classification, explore potential gains in accuracy by comparing the performance of more advanced graph convolutional layers, and show that graph attention layers [4] provide the greatest performance boost, leading to a new model, Policy-GAT.

2 Introduction

2.1 Background

Graphs are an intuitive data structure for representing many real-world phenomena, like the molecular interaction of proteins [5] and social networks [6], and can be leveraged for several different tasks across the graph hierarchy, like node classification, link prediction, and graph property prediction [7]. A core technique underlying these applications is graph representation learning [8], which extracts low-dimensional representations that encode hierarchical and structural information from the graph. Given the recent success of deep-learning based techniques in many problem settings, much research focus has been given to developing deep models to learn these representations and iterating on their designs to achieve state-of-the-art performance on graph-based inference tasks across a variety of datasets.

2.2 Dataset

For this project, I decided to focus on the task of node-level prediction, as most of my previous work in machine learning has centered around object-level classification tasks, and I was interested in exploring how relational features can be employed for such goals. To better standardize my results and compare them against other methods in a rigorous way, I am utilizing the data processing and evaluation pipelines implemented in Open Graph Benchmark (OGB) [9]. Since the project was prototyped and executed utilizing Colab's resources, I settled on the `obgn-arxiv` dataset for its compact size. This dataset consists of nodes for CS-area research papers connected by directed edges

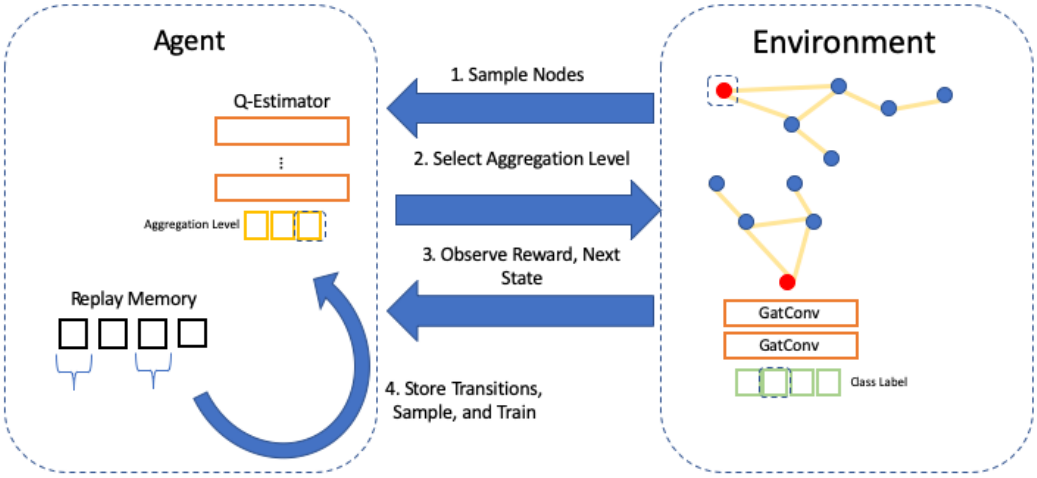

Figure 1: A visual illustration of high-level information flow between the agent and environment in Policy-GAT.

representing citations. The task for this dataset is to classify the nodes into one of forty subject areas, like AI or operating systems.

3 Policy-GNN

The key idea underlying work on GNN models is the concept of neighborhood aggregation. Graphs consist of nodes connected by edges, and we can gather information about a particular node in a graph by looking at its local neighborhood, typically defined by a k-hop distance where k is the maximum number of edges between the central node and its neighbors. Graph convolutional operators aggregate information from a node's immediate, 1-hop neighborhood and combine them with the node's previous representation to produce a new feature embedding. Given this behavior, the effect of additional graph convolutional layers in a GNN is to aggregate information from neighbors that are progressively farther away. This increased structural information can benefit inference tasks by providing a greater range of local, structural information, but comes at the risk of oversmoothing, in which gradients vanish and embeddings for distinct nodes converge to the same value [10].

Building on the observation that different nodes often require different levels of aggregation to most effectively classify them, Policy-GNN proposes a method to explicitly learn the trade-off between efficient, distinguishable embeddings smaller networks produce and the increased structural information enabled by additional layers. This is facilitated by modeling the choice of aggregation layer count as a Markov Decision Process which allows for the use of deep reinforcement learning methods to train a policy that can make the decision based on node features. Policy-GNN is a model **that has not been previously implemented for the OGB leaderboard**.

3.1 Previous Work

Several other works have attempted to exploit the performance of deeper GNN models while still mitigating the effects of the over-smoothing problem. DeepGCNs [11] and DGNs [12] propose novel message passing architectures and processing steps within the network. Graph U-nets [13] and similar models have utilized skip connections in deep architectures to enable deeper networks that avoid the over-smoothing problem by forwarding information across layers. However, a common shortcoming among all these models is that they still define fixed-size neighborhoods, or explicitly defined connections in the network in the case of U-nets.

Almasan, et al [14] proposed applying deep reinforcement learning methods to the setting of graph machine learning, but their application focused on improving the generalizability of the model to topologies that were not observed at training time. Work in other applications of deep reinforcement learning have demonstrated the success of meta-learning to guide machine learning tasks [15], motivating the decision to apply the meta-policy paradigm to the domain of GNNs.

3.2 Method

Policy-GNN trains a meta-policy that selects layer count for different nodes by modeling this choice as a Markov Decision Process (MDP) of the form $M = (S, A, P_T, R, y)$, where S is a set of states corresponding to node features, A is the set of actions that represent the number of aggregation layers, P_T is the transition distribution from a state and action pair to the subsequent state, R is our reward function that measures the marginal improvement on a fixed set of nodes (in the paper this was the validation set, but to align with the rules for OGB, I substituted a pre-selected set of training nodes), and y is a discount factor applied to future rewards. Within the setting of the MDP, at each timestep t, an agent observes a state and selects an action according to some policy, and receives the next state and reward. The problem of aggregation optimization is framed as finding the policy that maps node feature states to the number of aggregation layers that optimizes the downstream prediction task.

MDPs such as the one defined above can be solved by a variety of deep reinforcement learning algorithms, depending on what portions of the MDP are observable or provided to the agent [16]. One of the most popular methods in deep RL is Deep Q-Learning, which is a model-free algorithm that uses a neural network to estimate the Q-function of an MDP. Q-Functions map state and action pairs to the expected sum of rewards accrued by taking that action in the given state and then acting optimally at all subsequent timesteps. This training goal is explicitly given by: $Q(s, a) = \mathbb{E}_{s'}\ [R(s^{'}) + y(\max_{a'} Q(s^{'}, a^{'})]$. After training an agent to learn this Q-function, the optimal policy can be recovered by taking the action that maximizes the Q-function at the observed state. Policy-GNNs utilize an improved variant of Deep Q-Learning called Double Deep Q-Learning [3], which uses two Q-function estimation networks, one which is actively updated during training and a time-delayed target network that is used to calculate the Q-function update based on the above formula.

There are two main components to the implementation of Policy-GNNs: an OpenAI Gym environment [17] and a Double DQN Agent. The Gym environment, Policy-GNN Environment, defines the state and action spaces for the MDP, controls the MDP flow with step and reset functions, and contains the flexible GNN that computes node classifications based on the node features and selected number of aggregation steps. The Double DQN Agent contains the Q-Estimator and Q-Target networks and a replay memory buffer to store and sample observed transitions for training.

Training for Policy-GNN proceeds in two steps, the first focusing on training the Q-networks to recover the best policy and then re-trains a newly initialized flexible GNN with the optimal policy in the second step. To start the learning process, an instance of the Policy-GNN Environment and Double DQN Agent are initialized. Then for a set number of epochs, we train the policy over a given number of steps in the environment. The initial state is provided by a call to the environment's reset function. Then at each step, the agent selects an action, observes the subsequent state and reward, and feeds the entire observed transition of the form (state, action, next state, reward) to the memory buffer. When selecting actions during training, we want to encourage the agent to explore the environment and avoid local minima, so an epsilon value is set such that with probability epsilon, the agent returns a random action in lieu of the action that maximizes the Q-function at that point. After taking the prescribed number of steps, we train the Q-Estimator by sampling a batch of transitions from our memory buffer, calculating the Q-values at that state, and taking the gradient of the loss between those Q-values and the values determined by our target network and the expectation formula for the Q-function. Finally, if we have taken the number of training steps at which we want to replace our target network, we replace its weights with those of the current Q-Estimator.

In the second training phase, we create a new instance of our environment (which contains a newly initialized flexible GNN), and focus solely on optimizing the GNN model with our trained policy. After generating an initial state with the environment's reset function, we proceed for a separately-specified number of GNN training epochs. In each epoch, we recover actions for our current state with our trained policy and train on any full buffers, as described below in Policy-GNN's training efficiency improvements.

After proceeding through the two phases of training for the policy and then the model, testing the model after training is a simple as passing a set of states to the environment's evaluate function, which gets the best actions according to our trained policy and then predicts node labels based on their features and the selected level of aggregation.

The original paper also implemented efficiency improvements in the form of parameter sharing and a buffer mechanism for graph representation learning. The flexible GNN is initialized with the maximum number of convolutional layers and stacks the selected number of layers in their initialization order during forward passes. This parameter sharing greatly increases training efficiency by preventing us from having to train independent GNNs of fixed size, which would require a far greater number of parameters and larger training times as a result. Training time would also increase if we had to perform the time-intensive GNN construction and computation for every individual state, action pair, so training of the flexible GNN is performed in batches. Each time the environment receives an action for a given state, the state is stored in a buffer corresponding to that action. When the buffer reaches our pre-determined batch size, we train the flexible GNN on that batch and then reset the buffer.

4 Improving Policy-GNN: Advanced Convolutional Layers

In an effort to improve over the baseline performance of Policy-GNN, I explored several candidate graph convolutional layers and implemented efficiency improvements. With the discovery that graph attention layers provide the greatest performance gains, my final model, Policy-GAT, functions as depicted in Figure 1.

4.1 Motivation

In the original paper, Policy-GNN is presented primarily as a framework. Deep reinforcement learning techniques had not previously been used to optimize GNNs in this fashion, so the novelty of the application of meta-policy learning to aggregation optimization was the main focus of the work. The original authors also wanted to show the strength of their framework by demonstrating its performance when applying it to some of the simplest methods in graph machine learning and deep RL. However, many more advanced graph convolutional architectures have been proposed and demonstrated since the original Graph Convolutional Networks paper [2], relying on different forms and aggregations of message passing between neighbors in the network. Given the potential performance improvements that could be gained in the Policy-GNN framework by utilizing these more advanced methods, I believed that exploring these alternative convolutional layers would be a worthwhile contribution to the development of the Policy-GNN framework. For my experiments, I decided to compare the GraphConv [18], GraphSAGE [19], Graph Attenion [4], and Simple Graph Convolution [20] operators. The GraphConv operator is a convolutional layer that can take into account higher-order graph structures. GraphSAGE proposes an inductive approach to convolutional architectures, which I believed would provide potential improvements as a result of the design for the data-splitting procedure for the `obgn-arxiv` dataset. Graph Attention layers use attention scores to facilitate weighted aggregation of neighborhood features, which is an approach that has led to many state-of-the-art models (As of the time of writing, 10 of the top 12 models on the OGB leaderboard for `obgn-arxiv` utilize graph attention techniques). Finally, while not necessarily a more advanced method for graph convolutions, Simple Graph Convolutions are a simplified convolutional architecture designed to optimize computational efficiency by removing non-linearities and collapsing weight matrices between layers, which I was interested in exploring in this context as a way to offset the computational costs of a dual training regime.

Given that training for Policy-GNN takes a relatively long time compared to other methods due to the fragmented training procedure, I also felt that implementing any potential efficiency improvements would be beneficial. To that end, I restructured all of the reset, step, and prediction functions to operate on batches of states at a time, rather than individual nodes. Also, in their original paper, the transition from state to next state was made by selecting a random neighbor from the k-hop subgraph induced by the aggregation selection. In my work to re-implement their specifications, I discovered that this did not provide any considerable advantage over randomly selected transitions but incurred additional computational costs for the k-hop calculations, so I ultimately opted for the random transition model.

Table 1: Performance Table

Model	Training Acc.	Validation Acc.	Test Acc.
MLP	63.58	57.65	55.50
Node2Vec	76.43	71.29	70.07
GCN	78.87	73.00	71.74
GraphSAGE	82.35	72.77	71.49
Policy-GNN (GCN)	49.99	49.97	43.87
Policy-GNN (GraphConv)	26.13	19.53	10.18
Policy-GNN (GraphSAGE)	51.02	51.41	43.99
Policy-GNN (SGConv)	48.98	49.81	43.69
Policy-GNN (GATConv)	49.48	49.66	44.34
Policy-GAT	47.95±5.7	49.22±4.99	43.99±4.95

5 Experiments

5.1 Model Design and Hyperparameters

As a result of the complexity of the overall framework design, training a Policy-GNN model requires a large number of model design choices and hyperparameters. The flexible GNN network is composed of a variable number of convolutional layers, up to a maximum depth (and maximum action choice) of 5, with a hidden dimension size of 32. Between each layer is a relu nonlinearity and a dropout layer with probabilty 0.5. The final layer is passed through a logarithmic softmax function to return a class label. The flexible GNN network is trained with NLLLoss and an Adam optimizer with learning rate 0.01 and weight decay of 0.0005. The Q-function estimators are modeled as a series of six linear layers with tanh nonlinearities in between. The hidden dimensions between the linear layers are of size 32, 64, 128, 64, and 32, respectively. The Q-Estimator network is trained using MSELoss and an Adam optimizer with learning rate 0.0005. The Q-Target network is not trained but is updated every 1000 training steps. The DQN Agent has a memory buffer size of 10,000 transitions, a reward discount factor of 0.95, and begins with an epsilon probability of 1.0 that decays each step by 0.008 to a minimum value of 0.2. The reward function is calculated on the marginal increase in accuracy on a randomly-selected but fixed set of 500 nodes, and the accuracy is compared to the 50 most recent model performance scores. The first phase of training proceeds for 100 epochs and within each epoch, the agent takes 20 steps in the environment. The second phase trains the flexible GNN with the best policy for 500 epochs. Finally, all batched operations are performed with a batch size of 128.

5.2 Results

The results presented in Table 1 consist of the baseline performance metrics from the original OGB paper [9], the performance of Policy-GNN as described in its original paper [1], intermediate results from experiments with differing convolutional architectures, and the final confidence interval results for Policy-GAT tested over five different random seeds. The Policy-GNN metrics and intermediate results were all tested with the same seed.

5.3 Discussion

As I predicted based on the results I had observed elsewhere in the field of graph machine learning, graph attention layers do appear to lead to the best performance. Unfortunately, my results do not compare favorably with the baselines provided by OGB, but it is worthwhile to consider the factors that may have led to these outcomes. The primary cause is likely the inherent instability in deep reinforcement learning algorithms [16]. This effect is easy to see upon observation of the wide confidence interval for the final performance metrics on Policy-GAT. One method to mitigate these issues would be to decrease the learning rate for the deep RL policy and compensate with increased epochs, however this leads us to another possible explanation for our relatively low final accuracies: limited computational resources. All of these models were developed and tested on Google Colab, and the time limits and timeouts on the platform limited my ability to run longer experiments, and ultimately restrained my hyperparameter search spaces in regards to learning rates and epochs. I did

search over several faster training regimes (higher learning rates and fewer epochs) but these were prone to gradient collapse. As a result, I ran the longest experiments that were feasible.

6 Conclusion

In this work, I presented an implementation of Policy-GNN built on the standardized OGB data processing and evaluation pipeline, explored the performance gains when utilizing various convolutional operators with the framework, and introduced several computational efficiency gains for the training regime.

6.1 Future Work

Despite my attempts to directly replicate Policy-GNN as specified in the paper, I did not succeed in replicating the success of their results, which in the paper were reported to be $\approx 90\%$ across the benchmark datasets Cora, Citeseer, and Pubmed [1]. I hypothesize that the instability issue that is frequently encountered when training deep reinforcement learning models is the primary causes of this discrepancy. As a result, while this work focused primarily on the performance gains that could be achieved with more advanced graph machine learning methods, future work on better reward engineering and implementing more modern and stable variations of deep reinforcement learning algorithms could prove to be an invaluable benefit to the framework.

References

[1] Lai, K., et al. (2020) Policy-GNN: Aggregation Optimization for Graph Neural Networks. In *KDD*.

[2] Kipf, T. & Welling, M. (2017) Semi-supervised Classification with Graph Convolutional Networks. In *ICLR*.

[3] Hasselt, H., Guez, A., & Silver, S. (2016) Deep Reinforcement Learning with Double Q-Learning. In *AAAI Conference on Artificial Intelligence*

[4] Velickovic, P., et al. (2018) Graph Attention Networks. In *ICLR*.

[5] Yang, F., et al. (2020) Graph-based Prediction of Protein-Protein Interactions with Attributed Signed Graph Embedding. In *BMC Bioinformatics*.

[6] Tan, Q., Liu, N. & Hu, X. (2019) Deep Representation Learning for Social Network Analysis. In *Front. Big Data*.

[7] Zhou, G., et al. (2019) Graph Neural Networks: A Review of Methods and Applications. In *arXiv e-prints*.

[8] Hamilton, W., Ying, R., Leskovec, J. (2017) Representation Learning on Graphs: Methods and Applications. In *IEEE Data Engineering Bulletin*.

[9] Hu, W., et al. (2020) Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *arXiv preprints*.

[10] Gürel, N., et al. (2019) An Anatomy of Graph Neural Networks Going Deep via the Lens of Mutual Information: Exponential Decay vs. Full Preservation. In *arXiv preprint*.

[11] Li, G., et al. (2019) DeepGCNs: Can GCNs Go as Deep as CNNs? In *IEEE International Conference on Computer Vision*.

[12] Gao, H. & Ji, S. (2019) Graph U-nets. In *Proceedings of the International Conference on Machine Learning*.

[13] Zhou, K., et al. (2020) Towards Deeper Graph Neural Networks with Differentiable Group Normalization. In *arXiv e-prints*.

[14] Almasan, P., et al. (2019) Deep Reinforcement Learning Meets Graph Neural Networks: Exploring a Routing Optimization Use Case. In *arXiv e-prints*.

[15] Zha, D., et al. (2019) Experience Replay Optimization. In *IJCAI*.

[16] Arulkumaran, K., et al. (2017) A Brief Survey of Deep Reinforcement Learning. In *IEEE Signal Processing Magazine*.

[17] Brockman, G., et al. (2016) OpenAI Gym. In *arXiv preprints*.

[18] Morris, C., et al. (2019) Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks. In *AAAI Conference on Artificial Intelligence*.

[19] Hamilton, W., Ying, R., and Leskovec, J. (2017). Inductive Representation Learning on Large Graphs. In *NIPS*.

[20] Wu, F., et al. (2019) Simplifying Graph Convolutional Networks. In *ICML*.