Chapter 6

Graph Neural Networks in Practice

In Chapter 5, we introduced a number of graph neural network (GNN) architectures. However, we did not discuss how these architectures are optimized and what kinds of loss functions and regularization are generally used. In this chapter, we will turn our attention to some of these practical aspects of GNNs. We will discuss some representative applications and how GNNs are generally optimized in practice, including a discussion of unsupervised pre-training methods that can be particularly effective. We will also introduce common techniques used to regularize and improve the efficiency of GNNs.

6.1 Applications and Loss Functions

In the vast majority of current applications, GNNs are used for one of three tasks: node classification, graph classification, or relation prediction. As discussed in Chapter 1, these tasks reflect a large number of real-world applications, such as predicting whether a user is a bot in a social network (node classification), property prediction based on molecular graph structures (graph classification), and content recommendation in online platforms (relation prediction). In this section, we briefly describe how these tasks translate into concrete loss functions for GNNs, and we also discuss how GNNs can be pre-trained in an unsupervised manner to improve performance on these downstream tasks.

In the following discussions, we will use $\mathbf{z}_u \in \mathbb{R}^d$ to denote the node embedding output by the final layer of a GNN, and we will use $\mathbf{z}_{\mathcal{G}} \in \mathbb{R}^d$ to denote a graph-level embedding output by a pooling function. Any of the GNN approaches discussed in Chapter 5 could, in principle, be used to generate these embeddings. In general, we will define loss functions on the $\mathbf{z}_u$ and $\mathbf{z}_{\mathcal{G}}$ embeddings, and we will assume that the gradient of the loss is backpropagated through the parameters of the GNN using stochastic gradient descent or one of its variants [Rumelhart et al., 1986].

6.1.1 GNNs for Node Classification

Node classification is one of the most popular benchmark tasks for GNNs. For instance, during the years 2017 to 2019—when GNN methods were beginning to gain prominence across machine learning—research on GNNs was dominated by the Cora, Citeseer, and Pubmed citation network benchmarks, which were popularized by Kipf and Welling [2016a]. These baselines involved classifying the category or topic of scientific papers based on their position within a citation network, with language-based node features (e.g., word vectors) and only a very small number of positive examples given per each class (usually less than 10% of the nodes).

The standard way to apply GNNs to such a node classification task is to train GNNs in a fully-supervised manner, where we define the loss using a softmax classification function and negative log-likelihood loss:

$$\mathcal{L} = \sum_{u \in \mathcal{V}_{\text{train}}} -\log(\text{softmax}(\mathbf{z}_u, \mathbf{y}_u)). \tag{6.1}$$

Here, we assume that $\mathbf{y}_u \in \mathbb{Z}^c$ is a one-hot vector indicating the class of training node $u \in \mathcal{V}_{\text{train}}$; for example, in the citation network setting, $\mathbf{y}_u$ would indicate the topic of paper u. We use $\text{softmax}(\mathbf{z}_u, \mathbf{y}_u)$ to denote the predicted probability that the node belongs to the class $\mathbf{y}_u$, computed via the softmax function:

$$\text{softmax}(\mathbf{z}_u, \mathbf{y}_u) = \sum_{i=1}^{c} \mathbf{y}_u[i] \frac{e^{\mathbf{z}_u^\top \mathbf{w}_i}}{\sum_{j=1}^{c} e^{\mathbf{z}_u^\top \mathbf{w}_j}}, \tag{6.2}$$

where $\mathbf{w}_i \in \mathbb{R}^d, i = 1, ..., c$ are trainable parameters. There are other variations of supervised node losses, but training GNNs in a supervised manner based on the loss in Equation (6.1) is one of the most common optimization strategies for GNNs.

Supervised, semi-supervised, transductive, and inductive Note that—as discussed in Chapter 1—the node classification setting is often referred to both as supervised and semi-supervised. One important factor when applying these terms is whether and how different nodes are used during training the GNN. Generally, we can distinguish between three types of nodes:

1. There is the set of training nodes, $\mathcal{V}_{\text{train}}$. These nodes are included in the GNN message passing operations, and they are also used to compute the loss, e.g., via Equation (6.1).

2. In addition to the training nodes, we can also have *transductive* test nodes, $\mathcal{V}_{\text{trans}}$. These nodes are unlabeled and not used in the loss computation, but these nodes—and their incident edges—are still involved in the GNN message passing operations. In other words, the GNN will generate hidden representations $\mathbf{h}_u^{(k)}$ for the nodes in $u \in \mathcal{V}_{\text{trans}}$ during the GNN message passing operations. However, the

final layer embeddings $\mathbf{z}_u$ for these nodes will not be used in the loss function computation.

3. Finally, we will also have *inductive* test nodes, $\mathcal{V}_{\text{ind}}$. These nodes are not used in either the loss computation or the GNN message passing operations during training, meaning that these nodes—and all of their edges—are completely unobserved while the GNN is trained.

The term semi-supervised is applicable in cases where the GNN is tested on transductive test nodes, since in this case the GNN observes the test nodes (but not their labels) during training. The term inductive node classification is used to distinguish the setting where the test nodes—and all their incident edges—are completely unobserved during training. An example of inductive node classification would be training a GNN on one subgraph of a citation network and then testing it on a completely disjoint subgraph.

6.1.2 GNNs for Graph Classification

Similar to node classification, applications on graph-level classification are popular as benchmark tasks. Historically, kernel methods were popular for graph classification, and—as a result—some of the most popular early benchmarks for graph classification were adapted from the kernel literature, such as tasks involving the classification of enzyme properties based on graph-based representations [Morris et al., 2019]. In these tasks, a softmax classification loss—analogous to Equation (6.1)—is often used, with the key difference that the loss is computed with graph-level embeddings $\mathbf{z}_{\mathcal{G}_i}$ over a set of labeled training graphs $\mathcal{T} = \{\mathcal{G}_1, ..., \mathcal{G}_n\}$. In recent years, GNNs have also witnessed success in regression tasks involving graph data—especially tasks involving the prediction of molecular properties (e.g., solubility) from graph-based representations of molecules. In these instances, it is standard to employ a squared-error loss of the following form:

$$\mathcal{L} = \sum_{\mathcal{G}_i \in \mathcal{T}} \|\text{MLP}(\mathbf{z}_{\mathcal{G}_i}) - y_{\mathcal{G}_i}\|_2^2, \tag{6.3}$$

where MLP is a densely connected neural network with a univariate output and $y_{\mathcal{G}_i} \in \mathbb{R}$ is the target value for training graph $\mathcal{G}_i$.

6.1.3 GNNs for Relation Prediction

While classification tasks are by far the most popular application of GNNs, GNNs are also used in in relation prediction tasks, such as recommender systems [Ying et al., 2018a] and knowledge graph completion [Schlichtkrull et al., 2017]. In these applications, the standard practice is to employ the pairwise node embedding loss functions introduced in Chapters 3 and 4. In principle, GNNs can be combined with any of the pairwise loss functions discussed in those chapters, with the output of the GNNs replacing the shallow embeddings.

6.1.4 Pre-training GNNs

Pre-training techniques have become standard practice in deep learning [Goodfellow et al., 2016]. In the case of GNNs, one might imagine that pre-training a GNN using one of the neighborhood reconstruction losses from Chapter 3 could be a useful strategy to improve performance on a downstream classification task. For example, one could pre-train a GNN to reconstruct missing edges in the graph before fine-tuning on a node classification loss.

Interestingly, however, this approach has achieved little success in the context of GNNs. In fact, Veličković et al. [2019] even find that a *randomly initialized GNN* is equally strong compared to one pre-trained on a neighborhood reconstruction loss. One hypothesis to explain this finding is the fact that the GNN message passing already effectively encodes neighborhood information. Neighboring nodes in the graph will tend to have similar embeddings in a GNN due to the structure of message passing, so enforcing a neighborhood reconstruction loss can simply be redundant.

Despite this negative result regarding pre-training with neighborhood reconstruction losses, there have been positive results using other pre-training strategies. For example, Veličković et al. [2019] propose *Deep Graph Infomax (DGI)*, which involves maximizing the mutual information between node embeddings $\mathbf{z}_u$ and graph embeddings $\mathbf{z}_{\mathcal{G}}$. Formally, this approach optimizes the following loss:

$$\mathcal{L} = -\sum_{u\in\mathcal{V}_{\text{train}}} \mathbb{E}_{\mathcal{G}} \log(D(\mathbf{z}_u, \mathbf{z}_{\mathcal{G}})) + \gamma \mathbb{E}_{\tilde{\mathcal{G}}} \log(1 - D(\tilde{\mathbf{z}}_u, \mathbf{z}_{\mathcal{G}})). \tag{6.4}$$

Here, $\mathbf{z}_u$ denotes the embedding of node u generated from the GNN based on graph $\mathcal{G}$, while $\tilde{\mathbf{z}}_u$ denotes an embedding of node u generated based on a *corrupted* version of graph $\mathcal{G}$, denoted $\tilde{G}$. We use D to denote a *discriminator* function, which is a neural network trained to predict whether the node embedding came from the real graph $\mathcal{G}$ or the corrupted version $\tilde{\mathcal{G}}$. Usually, the graph is corrupted by modifying either the node features, adjacency matrix, or both in some stochastic manner (e.g., shuffling entries of the feature matrix). The intuition behind this loss is that the GNN model must learn to generate node embeddings that can distinguish between the real graph and its corrupted counterpart. It can be shown that this optimization is closely connected to maximizing the mutual information between the node embeddings $\mathbf{z}_u$ and the graph-level embedding $\mathbf{z}_{\mathcal{G}}$.

The loss function used in DGI (Equation 6.4) is just one example of a broader class of unsupervised objectives that have witnessed success in the context of GNNs [Hu et al., 2019, Sun et al., 2020]. These unsupervised training strategies generally involve training GNNs to maximize the mutual information between different levels of representations or to distinguish between real and corrupted pairs of embeddings. Conceptually, these pre-training approaches—which are also sometimes used as auxiliary losses during supervised training—bear similarities to the "content masking" pre-training approaches that have ushered in a new state of the art in natural language processing [Devlin et al., 2018]. Nonetheless, the extension and improvement of GNN pre-training approaches is an open and active area of research.

6.2 Efficiency Concerns and Node Sampling

In Chapter 5, we mainly discussed GNNs from the perspective of node-level message passing equations. However, directly implementing a GNN based on these equations can be computationally inefficient. For example, if multiple nodes share neighbors, we might end up doing redundant computation if we implement the message passing operations independently for all nodes in the graph. In this section, we discuss some strategies that can be used implement GNNs in an efficient manner.

6.2.1 Graph-level Implementations

In terms of minimizing the number of mathematical operations needed to run message passing, the most effective strategy is to use graph-level implementations of the GNN equations. We discussed these graph-level equations in Section 5.1.3 of the previous chapter, and the key idea is to implement the message passing operations based on sparse matrix multiplications. For example, the graph-level equation for a basic GNN is given by

$$\mathbf{H}^{(k)} = \sigma\left(\mathbf{A}\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}_{\text{neigh}} + \mathbf{H}^{(k-1)}\mathbf{W}^{(k)}_{\text{self}}\right), \tag{6.5}$$

where $\mathbf{H}^{(t)}$ is a matrix containing the layer-k embeddings of all the nodes in the graph. The benefit of using these equations is that there are no redundant computations—i.e., we compute the embedding $\mathbf{h}_u^{(k)}$ for each node u exactly once when running the model. However, the limitation of this approach is that it requires operating on the entire graph and all node features simultaneously, which may not be feasible due to memory limitations. In addition, using the graph-level equations essentially limits one to full-batch (as opposed to mini-batched) gradient descent.

6.2.2 Subsampling and Mini-Batching

In order to limit the memory footprint of a GNN and facilitate mini-batch training, one can work with a subset of nodes during message passing. Mathematically, we can think of this as running the node-level GNN equations for a subset of the nodes in the graph in each batch. Redundant computations can be avoided through careful engineering to ensure that we only compute the embedding $\mathbf{h}_u^{(k)}$ for each node u in the batch at most once when running the model.

The challenge, however, is that we cannot simply run message passing on a subset of the nodes in a graph without losing information. Every time we remove a node, we also delete its edges (i.e., we modify the adjacency matrix). There is no guarantee that selecting a random subset of nodes will even lead to a connected graph, and selecting a random subset of nodes for each mini-batch can have a severely detrimental impact on model performance.

Hamilton et al. [2017b] propose one strategy to overcome this issue by subsampling node neighborhoods. The basic idea is to first select a set of target nodes for a batch and then to recursively sample the neighbors of these nodes in order to ensure that the connectivity of the graph is maintained. In order to avoid the possibility of sampling too many nodes for a batch, Hamilton et al. [2017b] propose to subsample the neighbors of each node, using a fixed sample size to improve the efficiency of batched tensor operations. Additional subsampling ideas have been proposed in follow-up work [Chen et al., 2018], and these approaches are crucial in making GNNs scalable to massive real-world graphs [Ying et al., 2018a].

6.3 Parameter Sharing and Regularization

Regularization is a key component of any machine learning model. In the context of GNNs, many of the standard regularization approaches are known to work well, including L2 regularization, dropout [Srivastava et al., 2014], and layer normalization [Ba et al., 2016]. However, there are also regularization strategies that are somewhat specific to the GNN setting.

Parameter Sharing Across Layers

One strategy that is often employed in GNNs with many layers of message passing is parameter sharing. The core idea is to use the same parameters in all the AGGREGATE and UPDATE functions in the GNN. Generally, this approach is most effective in GNNs with more than six layers, and it is often used in conjunction with gated update functions (see Chapter 5) [Li et al., 2015, Selsam et al., 2019].

Edge Dropout

Another GNN-specific strategy is known as *edge dropout*. In this regularization strategy, we randomly remove (or mask) edges in the adjacency matrix during training, with the intuition that this will make the GNN less prone to overfitting and more robust to noise in the adjacency matrix. This approach has been particularly successful in the application of GNNs to knowledge graphs [Schlichtkrull et al., 2017, Teru et al., 2020], and it was an essential technique used in the original graph attention network (GAT) work [Veličković et al., 2018]. Note also that the neighborhood subsampling approaches discussed in

Section 6.2.2 lead to this kind of regularization as a side effect, making it a very common strategy in large-scale GNN applications.