École Polytechnique Fédérale de Lausanne
Swiss Federal Institute of Technology

Project report

Authors

Miloš Novaković
Mateja Ilić

Professors

Prof. Pascal Frossard
Dr. Dorina Thanou

Subject

EE–452: Network Machine Learning

GitHub Repository

Amazon Product Project

1 Introduction

The goal of this project is to apply the tools we learned during the Network Machine Learning course to explore a real-world, network-based dataset and perform some inference task using the provided data. We chose the Amazon Products dataset because of its practical relevance and because it is easy to interpret the data. Our main task is node classification.

The Amazon Products dataset is a projection of a bipartite graph, whose one part contains nodes that represent users and whose other part contains nodes that represent products. The users and products are connected with an edge, if the user bought the product. The projection of this graph on the product part gives us a graph of 1,569,960 nodes, that are connected with 264,339,468 edges. In other words, we obtain a graph where two nodes are connected if users that bought one, also bought the other.

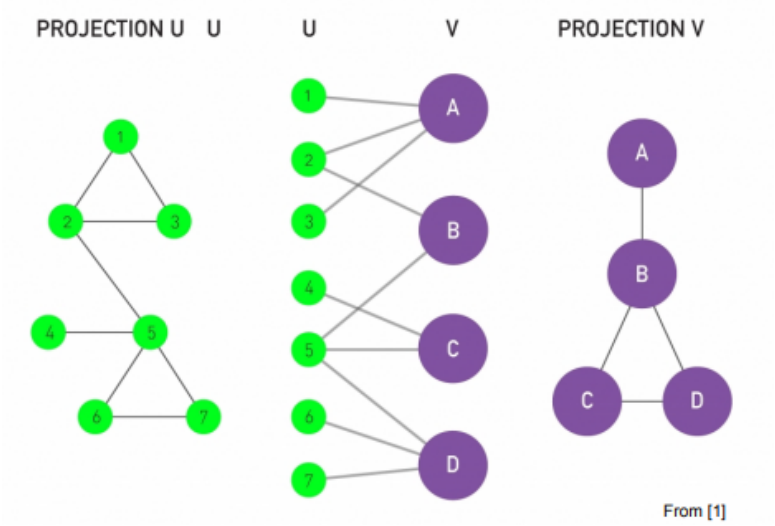

Figure 1: Bipartite graph illustration, taken from course slide 15, Lesson 1

Each node also comes with a feature vector $x \in \mathbb{R}^{200}$, which is an SVD-reduced vector of a much larger vector of 4-gram word counts from the reviews of the product. Each node is also labeled with zero or more of the 107 product category labels. A training, testing and validation mask is provided to divide up the node feature and label data when training and testing our models.

A node classification problem is an inference task where, based on the set of training node features and their corresponding labels, we need to learn a model that can accurately predict the labels on previously unseen nodes and their features. In order to solve this task we will try to employ several machine learning models.

2 Exploration

In order to better understand our data set, what it represents and what we can do with it, it is always useful to use the various tools we have from graph theory and network science to explore the network underlying our data. The main feature of this dataset that distinguishes it from the many other Big Data datasets is that a graph structure, which we will denote by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ has been recorded and preserved to go along with data we are more used to encounter - the feature matrix $X \in \mathbb{R}^{|\mathcal{V}| \times 200}$ and the corresponding label matrix $Y \in \mathbb{R}^{|\mathcal{V}| \times 107}$. This means that, besides the data explicitly tied to every product, we also have the position of the corresponding product node in the network as additional context. To better understand this context, we start by examining some general properties of our network.

In order to examine our network, we will use a Python package for network science called **networkx**. Since we are dealing with a fairly large dataset, we subclassed the regular networkx Graph class in order to disable edge attributes (since we have none anyway) and reduce the networkx memory footprint so the resulting graph instance can actually fit into our RAM. A lot of our exploration will be constricted to what is actually feasible in terms of memory and processing power available to us at the time of doing this project. This is one of the challenges that we face when we want to use very large datasets such as this one, but we refrained from doing our tasks on a subsample of the whole dataset because this was a unique opportunity to troubleshoot working with such large amounts of data. This way we can also get the full picture of what is happening with our dataset during various steps of processing.

2.1 Global graph properties

2.1.1 Basic graph properties

After successfully loading the graph into networkx, we run some quick commands to obtain the basic information about our graph:

- **Number of nodes**: 1 569 960
- **Number of edges**: 132 954 714
- **Number of self-loops**: 1 569 960
- **Number of connected components**: 492 143

As we can see, all of the nodes were successfully loaded from the edge index that was produced by the **PyTorch Geometric Dataloader** and all of the nodes have a self-loop, which makes sense because users who buy a product tautologically also buy the same product.

However, it is worth noting that the number of edges in our graph is about half of the number of edges that were in the edge index. This is our first approximation, since most of the linked products have links in both direction, we instatiated our Graph object as an instance of a regular graph subclass, instead of a digraph, because it is easier to manipulate. This is somewhat of an approximation, as we can see that our number of edges is slightly higher than 50% of the original number of edges in edge index. However, working the numbers out shows that only around 0.29% of edges weren't bi-directional, so we consider the approximation to be appropriate in this case.

Another important observation is that our graph apparently features a whopping 492 143 connected components! The average node degree of the graph, which can be easily calculated in undirected graphs as $< k >= \frac{2L}{N}$, where $L = |\mathcal{E}|$ and $N = |\mathcal{V}|$, comes out to be $< k > \approx 169.37$, also indicates that graph is sparse, since it is 4 orders of magnitude lower than than the number of nodes in the graph. This might be cause for concern, although an average node degree of 169.37 should still indicate a decent number of connections, the next thing we should check out is how are our nodes distributed in the many connected components of our graph. If our underlying network is not very informative about the relations of many nodes, it might not even be worth including it into our learning tasks.

2.1.2 The giant component

When exploring connected components, we naturally always start from the biggest one. In this case we are in luck, since the results are fairly clear:

- **Number of nodes**: 1 066 627
- **Number of edges**: 132 430 138

As we can see, the giant component contains $\approx 67.94\%$ of the nodes but $\approx 99.6\%$ of the edges. This means that most of the other components basically just consist of isolated nodes. In the context of our dataset, these nodes represent products for which there was not enough interactions to link with other products. We can interpret these results as promising, at least in terms of the nodes that belong to the giant component, since their connections should provide a lot of context that can be exploited for better classification performance. As for the other third of the nodes, we still have feature data for them that can be classified just like regular data samples. It is also possible that the absence of connections could serve as additional information to the model, and it would be interesting to check in future research if there's any difference in labeling accuracy for nodes that belong to the giant component vs those that don't.

Since our giant component is one fully connected subgraph, we can also check its diameter, i.e. the longest shortest path between any two nodes in it. Doing this using a brute force approach that actually calculates shortest paths would obviously take way too long, so we decided to use an approximate procedure that computes a lower bound on the (sub)graph diameter. **The lower bound of the GC diameter turns out to be 19.** We can see that, compared to its size, the giant component is fairly well connected, having a lower bound diameter of just 19. Since our graph is sparse, that means that our graph probably contains some large hubs through which many of the shortest paths go through. This leads us to our next exploration - we examine the degree statistics of our graph.

2.1.3 Node degree distribution

If we calculate the all the node degrees in our graph and sort them in descending order, we can see that our **top 10 hubs** feature degrees of 75135, 62409, 53409, 52950, 49801, 49446, 49384, 46101, 44767 and 44677. This indicates that there are some nodes that are really well connected with the largest hub connected to 7% of the nodes in our Giant Component.

To get an even better feel of how our nodes are connected, we generate the histogram of the node degree distribution:

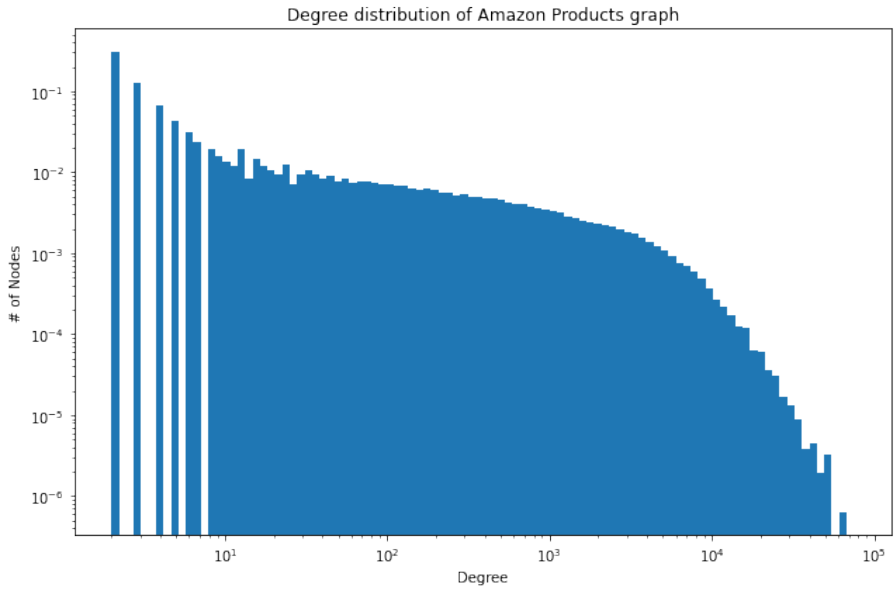

Figure 2: Logarithmic scale histogram plot of node degree distribution

The relative frequency of most of the node degrees looks very steady, as the degree grows, and then starts exponentially falling off somewhere around node degree 1000. This behaviour doesn't quite fit with either a power-law or a Poisson distribution, since there is an exponential drop-off in frequency for very high degree nodes, but the drop off isn't as steep immediately after $< k >$ as it is with networks that follow a Poisson degree distribution. It looks like our network is somewhere in the middle of those two models.

2.2 Class label distribution

Before we go into our classification task, one thing that is useful to check is how our class labels are distributed, i.e. whether we can expect a balanced dataset or not. To illustrate this, we generated a bar plot of label occurrences per category:

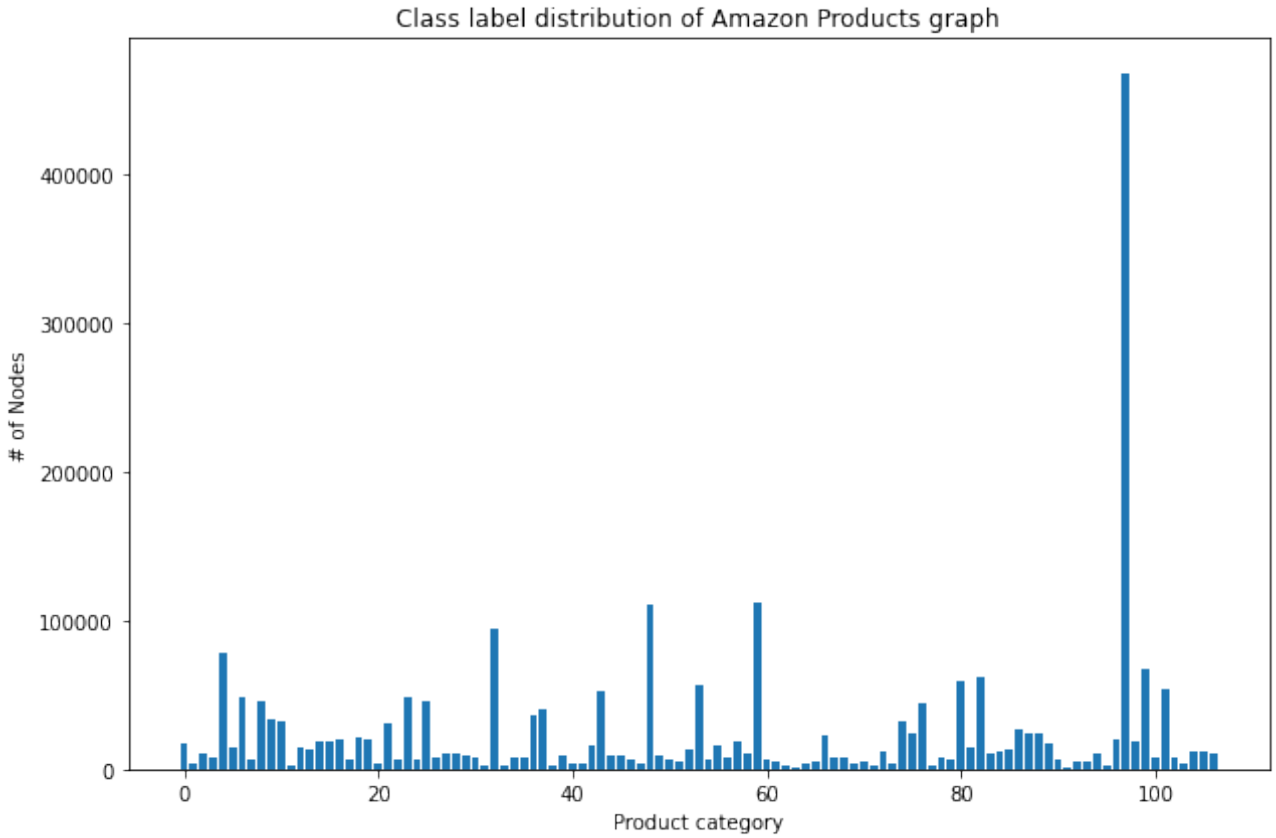

Figure 3: Bar plot of class label occurences

3 Exploitation

A classification task where each node is labeled with zero or more of the 107 class labels, which we have to infer based on the input feature matrix X (and graph structure $\mathcal{G}$ if we are using Graph Neural Networks), is called a multi-class, multi-label (node) classification problem.

As we've already mentioned, we will try multiple models to compare and contrast their classification performances and their potential to solve this problem, given enough time and data. These models will be implemented using **PyTorch** and **PyTorch Geometric** packages in python.

To test the classification performance, we are using the Python package **torchmetrics**. The torchmetrics' accuracy class offers two ways of evaluating the performance of a multi-class, multi-label classification model:

- **Micro-accuracy**, where each label of each output is compared independently between the model output and expected output, and an average is computed. This metric can be deceptive, since in our case most of the categories of every node are always labeled with 0, so if our model outputs only zeros, we will still get a substantially high accuracy, without learning anything.
- **Macro-accuracy**, enabled by setting subset_accuracy=True when instantiating the torchmetrics accuracy object, ensures that the testing module will only count the output of our model as correct if all of the labels were assigned correctly.

3.1 Setting up a baseline

Before building our GNN-based models, we wanted to establish a baseline performance. We do this by training a regular Neural Network, also known as a Multi-Layer Perceptron, without taking any of the underlying network data into account. To keep the comparison fair, we shall try to keep the number of hidden layers and hidden dimensions to the same values in all Neural Network-like models, if possible.

In this case, we set up a 1-hidden layer MLP (which means 3 layers in total), with a hidden dimension of 512 neurons. The activation function of the input and hidden layers was chosen to be ReLU, while the activation function of the output layer is the sigmoid function. This is done in multi-class, multi-label classification problems because the sigmoid function generates outputs in the $(0, 1)$ range. These outputs can be interpreted as probabilities given for all of the class labels for a given input feature vector and can be combined with a binary cross-entropy loss function that we will minimize during training.

After training using an ADAM optimizer, with a learning rate of $\eta = 0.01$ for 700 epochs, we obtain the following loss curve, after roughly 15 minutes of training:

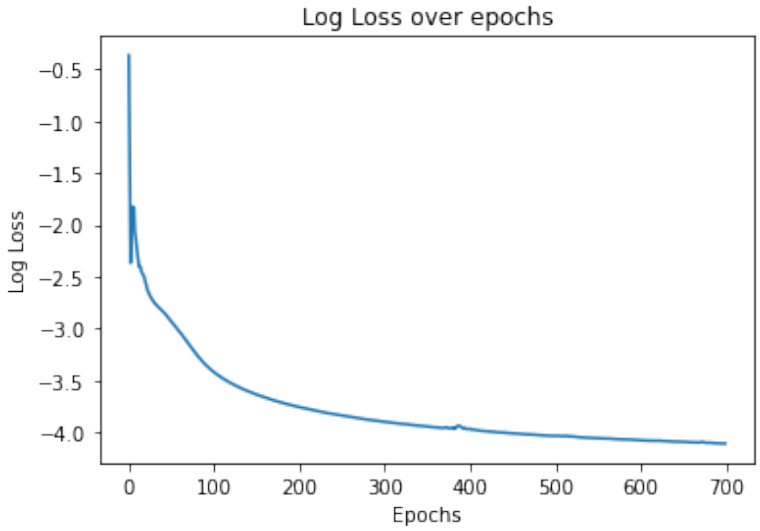

Figure 4: Training loss plot for a 1-hidden layer MLP

If we check out the macro-accuracy of our trained model, we obtain:

- **Training set accuracy**: 0.6426309943199158
- **Validation set accuracy**: 0.6415952444076538
- **Test set accuracy**: 0.6393341422080994

As we will see compared to other GNN models, the MLP trains fairly quickly and has resulted in a surprisingly high accuracy rate, even though it doesn't explicitly exploit any of the underlying network structure. We suspect that this is the case because we have a lot of labeled data (80% of 1.57M nodes) we can train on. This is also reflected in the fact that the MLP performs very similarly on the Validation and Test nodes as well, indicating that it managed to generalize as well as it architecture allowed it to. To see if we can do better than this, next we will try to use some GNN architectures that will try to embed the structure underlying our data explicitly before training an MLP classifier.

3.2 GraphSAGE

There are strong indications from the literature that exploiting the underlying structure in our data to automatically extract the features that will be the most informative for our learning task can work much better that doing manual feature engineering and much faster than just loading the raw data into one big Neural Network and hoping it extracts the structure on its own. Such an approach was successful with CNNs and 2D regular grid data (i.e. images), with RNNs and time-series etc. Graph Neural Networks try to recreate this approach for non-Euclidean structured data - in this case, our feature vectors are node attributes on a graph.

GraphSAGE is a Message Passing GNN, introduced in [1]. GraphSAGE is a model that learns node embeddings based on sampling and aggregating features from a node's local neighborhood, which should allow for better scalability. Both the function that generates embeddings and the function that aggregates them from layer to layer are learnable. This allows GraphSAGE to generate embeddings for previously unseen nodes.

We instantiated a GraphSAGE GNN module with 2 embedding layers and the same hidden/embedding dimensions as our MLP. After the node embedding, we input the data into an MLP classifier, so the rest of the architecture and training is the same as when using just an MLP for multi-class, multi-label classification tasks (i.e. sigmoid outer layer act. function + binary cross-entropy).

Unfortunately, we couldn't test our model since **the training could not be completed on any hardware that was available to us**, including Google Collab Pro +, which allocates around 51Gb of RAM to its user. It appears that, even with the added scalability of the node neighborhood subsampling, this approach is out of reach for single users working on large datasets such as this one. The original paper specifies that the initial GraphSAGE tests were ran on a machine with 256Gb of RAM. These results are why we turn to our final GNN model - GraphSAINT, that promises even better scalability through sampling of the whole graph minibatches of subgraphs, instead of subsampling node neighborhoods.

3.3 GraphSAINT

As we've already mentioned, Graph Convolution Networks are very useful for learning representations in the domain of Network Science. However, like in any other domain, the field of Graph Representation Learning also has some challenges that need be overcome. A notable one is the algorithmic complexity of some models or algorithms, which scale unfavorably as one increases the number of nodes and edges (or the size of the feature space corresponding to nodes or edges). Hence, some of the models are more powerful and expressive, in terms of their predictive power, but they be might lacking in terms of scalability. The scalability property in the world of thousand of petabytes of data is something that seriously needs to be put into the consideration when choosing a model for a specific Machine Learning application.

One class of Deep Learning models that proved itself to be efficient in this sense are Graph Convolution Networks (GCNs). GCNs can learn different (node, edge and graph) feature representations, i.e. embeddings, which can then be fed into a classic ML model to perform whatever taks we might need. When it comes to the scalability of GCNs to large graph datasets, one can see that researchers use different types of layer sampling techniques. They use the layer sampling techniques so that they do not come face to face with the exponentially growing neighborhood ("neighborhood expansion" or "neighborhood explosion").

One of the models that uses layer sampling techniques is called GraphSAINT, which we used in the realization of our project. The GraphSAINT model is used here with the purpose of handling large Amazon Products data-set graph (which has more than 1.5 million nodes). The GraphSAINT model was introduced in [2], and it features a fundamental difference compared to other sampling-based models - it takes the samples from the training graph itself, compared to other sampling-based models that take samples from the nodes or edges across the GCN layers.

Therefore, in each iteration of the training loop, the GraphSAINT builds a complete GCN from the sampled subgraph of the original graph, thus making the problem smaller and improving scalability. More importantly, we can control how large that subgraph will be (that becomes a hyperparameter in the training loop), so we can control the scalability of our problem ourselves. Because the size of the sampled subgraph is now a hyperparameter, the GraphSAINT ensures a fixed number of connected nodes in all layers.

What is also very important and useful is that GraphSAINT, by the nature of its design, separates the sampling of a graph (also referenced to as generation of subgraphs) from the training loop (forward and backward propagation). Hence, one can *store* the sampled subgraphs and use them in the training after. This gives huge reliefs because now there is no need to load the whole Graph (which can easily be tens or hundereds GB of RAM) into the RAM, we can now simply load just the sampled subgraphs.

We ran our GraphSAINT training model with two types of subgraph sampling procedures - using a Node Sample and a Random Walk sampler. The results of the training after 30 epochs can be seen below:

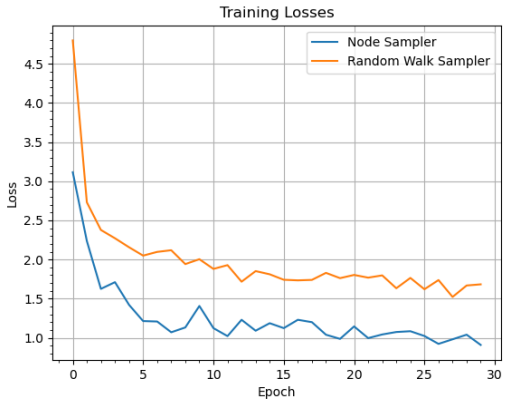

Figure 5: Training loss plot for 2 GraphSAINT runs with different samplers

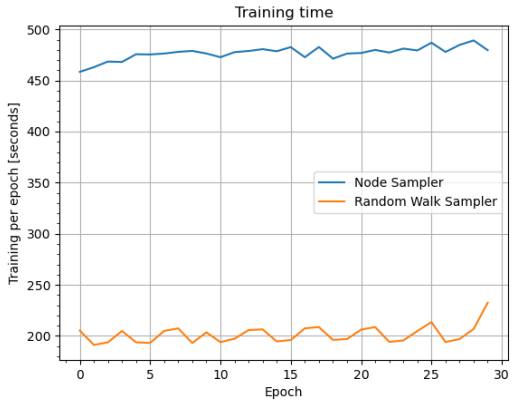

Figure 6: Training with RW sampled subgraphs is significantly faster

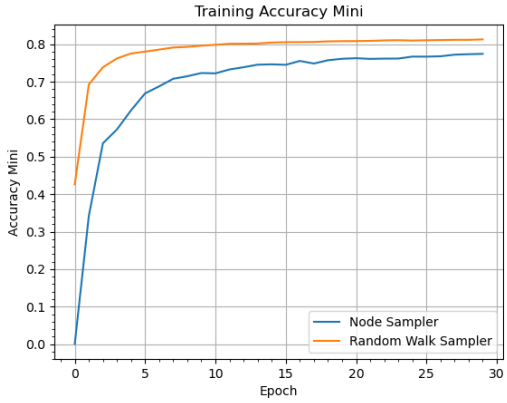

Figure 7: Training micro-accuracy plot for 2 GraphSAINT runs with different samplers

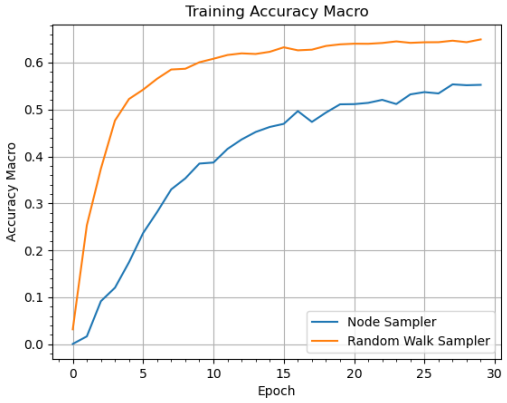

Figure 8: Training macro-accuracy plot for 2 GraphSAINT runs with different samplers

The RW sampler typically performs better on all counts:

- **Total Training Time**: 14150.79 sec
- **Average training time per epoch**: 201.58 sec
- **Test set micro-accuracy**: 0.8143
- **Test set macro-accuracy**: 0.6508

While the results after 30 epochs of training with the Node sampler aren't even better our baseline:

- **Total Training Time**: 21720.73 sec
- **Average training time per epoch**: 476.92 sec
- **Test set micro-accuracy**: 0.7757
- **Test set macro-accuracy**: 0.5526

In our opinion, the results of the GraphSAINT experiment do show promise, especially when it comes to using the RW graph sampler. The test set accuracy slightly outperforms our baseline, but we do have to remember that this is comparing 30 epochs of training to 700 that we did on the MLP. With further refinement of various GCN techniques, we might approach a time where the additional time spent implementing and training such models becomes worth it for the increase in performance.

References

[1] W.L. Hamilton, R. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs", Journal CoRR, abs/1706.02216, 2017.

[2] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan and Viktor K. Prasanna, "GraphSAINT: Graph Sampling Based Inductive Learning Method", Journal CoRR, abs/1907.04931, 2019.