An End-to-End Deep Learning Architecture for Graph Classification

Jessica Bojorquez [1] **Peter Kim** [1] **Tameez Latib** [1] **Puja Trivedi** [1]

Abstract

Neural networks are typically designed to deal with data in tensor forms. While many types of data are naturally presented with order, graphs usually lack a tensor representation. Given a dataset containing graphs in the form (G,y) where G is a graph and y is its class, the goal is to create a novel neural network architecture that is able to directly accept graphs of arbitrary structure. There are two main challenges when using graph datasets directly in neural networks: 1) how to extract useful features characterizing the rich information encoded in a graph for classification purposes , and 2) how to sequentially read a graph in a meaningful and consistent order. To address the first challenge we used a localized graph convolution model and to address the second challenge, we used novel SortPooling layer which sorts graph vertices in a consistent order so that traditional neural networks can be trained on the graphs (Zhang et al. 2018). This architecture allows end-to-end gradient based training with original graphs, without the need to first transform graphs into vectors.

1. Introduction

The application of neural networks in domains such as image classification, natural language processing, and reinforcement learning has increased over the past few years. Neural networks work the best when data is presented in tensor form due to the connection structure between layers in the network. For data that lack a tensor representation there is limited applicability of neural networks on them. While many types of data are naturally presented with order, there is another major category of structured data, namely graphs, which usually lack a tensor representation with fixed ordering. This issue has created a growing interest in generalizing neural networks to directly accept graphs of arbitrary structure and to learn classification and predictive functions. Graphs do not have a fixed structure; therefore, it is also important to create a model that map isomorphic graphs to the same representation and output the same prediction. This is an important area of research because it removes the need to first transform graphs into vectors when using neural network models and provides a different approach to solving graph problems.

One of the most common neural networks, convolutional neural network (CNN) are usually set up such that the data must first be converted into tensor form before being processed. Typically, CNNs analyze an image and train by running the tensors through multiple layers of neurons where certain operations are performed on the tensors, eventually resulting in a set of parameters. A benefit to using neural networks is that they are very rudimentary to implement while producing effective predictions. An obstacle that may make neural networks more difficult to implement is the fact that data must first be converted to tensor form beforehand. This ties back to the problem that neural networks will have problems handling data graphs with the form (G,y).

2. A comprehensive literature review

The paper proposed a state-of-the-art solution for typical neural networks by introducing a new SortPooling layer. The intention is to develop a neural network that mitigates the need to convert the dataset to tensor form while still using the features that a neural network provides, resulting in what the paper names a Deep Graph Convolutional Neural Network (DGCNN). The con of using DGCNN is that since it is specifically designed to deal with data graphs with dynamic shaping (unable to be converted to tensor form), it is better off using common neural network approaches like a CNN for more typical data-sets.

3. Deep Graph Convolutional Neural Networks

3.1. Algorithm

Our algorithm has three steps. We first pass the input through a Graph Convolution Network (GCN). Next, we use the novel Sortpool layer, which fixes the output size. Lastly, we apply a Convolutional Neural Network to the output of the Sortpool. In the next sections I will describe in detail what each part does and the parameters behind them.

3.2. GCN

Our GCN takes in the input graphs and applies graph convolution layers to it. Although there are many such convolution layers in literature, such as the one we learnt in class by Kipf and Welling, the paper suggests their own layer as follows:

$$Z^{t+1} = f(\hat{D}^{-1}\hat{A}Z^tW^t)$$

with Z^t being the features of t'th graph. That is, the ith column of Z^t correspond to the latent features of vertex/node i. Similarly, $Z^0 = X$ is the matrix where the ith column of X is the attributes of the ith node in the graph. In the case of MUTAG and many other datasets, this is a 1-hot encoding corresponding to what atom that vertice represents. The edges then encode the bond between them, which is stored in the adjacency matrix A. However, note that in the above formula we use $\hat{A} = A + I$. Similarly, $\hat{D} = D + I$, as in both matrices contain self loops. This is that the matrix multiplication of AZ^t has more information, which can then be given higher or lower weight based on W^t. Also, f can be any activation function but is chosen to be tanh in this case.

This GCN is also special in that it incorporates a RESNET-like highway structure, where the output of the GCN is not Z^h, the final layer (for an h-layer net), but instead the layer outputs are all feeded to the output and concatenated. That is, the concatenation of $Z^1, Z^2, ...Z^h$ which is defined as $Z^{1:h}$ is the output of our GCN. For the MUTAG dataset, typical results (unless otherwise specified) have $h = 4$, so there are 4 convolution layers. There has been experimentation with more or less layers, see the Experimental Evaluation for more.

3.3. SortPool

Next is the Sortpool layer, which takes in $Z^{1:h}$ which has variable input size. However, we are able to convert this to a fixed size output using the sortpool. In order to do this, we take advantage of the Weisfeiler-Leman (WL) colors of Z^h. The WL starts with each node in its own class based on node degree (or some other initialisation), and iteratively updates each node based on its own class and the classes of its neighbors. We can then create a consistent order based on these classes, my mapping each class to an integer and then taking the nodes of the class corresponding to the largest integer. The sortpool layer uses WL on Z^h, and then picks the top k (chosen by us, with discussion in later section) nodes to propagate forward as our output. Therefore, the output of sortpool is k columns from each $Z^1, Z^2, ...Z^h$, which will be denoted $Z_k^{1:h}$. For the MUTAG dataset, if we only have m atoms, then the shape of $Z_k^{1:h}$ is $h \times k \times m$. Whereas the original $Z^{1:h}$ had shape $h \times k' \times m$, where k' depends on the input but now k is fixed! Note that if $k' < k$, then the sortpool pads with zeros instead of truncating the result. However, for most of the graphs we have that $k < k'$ so we truncate. This new $Z_k^{1:h}$ is our embedding of the input graph. Comparing to other methods like SVD where we only take the top N eigenvectors/values, we only take the top m nodes, so there is some parallels to existing methods. Further details can be seen in the original authors paper [1], where more theoretical arguments are made and proven. However, I feel that the intuition of the WL colors producing a ranking of nodes is similar to how SVD uses a ranking of eigenvectors based on eigenvalues.

3.4. CNN

After sortpool, we are guaranteed data has fixed size! We flatten and then use 1D convolution and Fully connected (FC) layers next. This is a traditional CNN, which has been proven empirically to have quite high accuracy. However, it's important that our input has some order to it otherwise the CNN will not be able to capture any good information from our latent state and will behave poorly. This is why we need the sortpooling layer, since it provides consistency! That is, if you have an input molecule from the MUTAG dataset, the arrangement of the columns of X (which node is treated as the first node, etc) does not matter too much since the WL Colors will eventually rank the nodes of Z^h and so we will only extract the useful nodes, and in the order from best to worst color (so swapping columns in X / switching how a molecule is represented) has no effect on the input to our CNN. We only use 1 Conv layer and 2 FC layers with ReLU activation and softmax output. We also use incorporate dropout in the FC layer to reduce over-fitting. We explore adding an extra layer in a later section.

3.5. Data Pipeline

Putting all 3 of the above together, we have the what is shown in figure 1. That is, input graph $\rightarrow$ GCN $\rightarrow$ SortPool$\rightarrow$ CNN $\rightarrow$ output class. Furthermore note that each of these operations contains a gradient and so can utilize backpropagation. Thus, we have an end-to-end system which only needs data in the form of (G,y) and we can produce loss and perform gradient descent. In later sections we discuss different optimizers that we tried and the results thereof.

4. Experimental Evaluation

To evaluate results of the model, we performed a grid search using the MUTAG dataset. The dataset and code is available at https://github.com/tluccs/pytorch_DGCNN.

The experimental evaluation consisted of running a grid search that found best results for different values of dropout, hidden layers, and optimizer. Within each iteration of

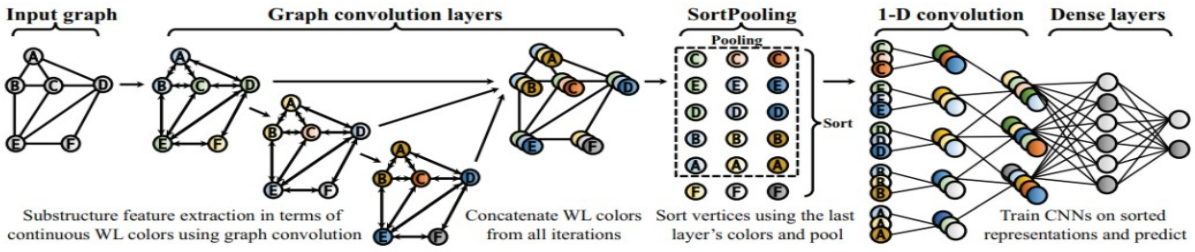

Figure 1. Our model.

dropout and hidden layer, we performed a second grid search to find the best parameters for the optimizers as well.

4.1. MUTAG Dataset

The MUTAG dataset contains 188 samples of nitroaromatic chemical compounds and describes their mutagenicity on Salmonella Typhimurium. Each sample is classified into two classes according to their mutagenic effect on bacterium. The mutagenic effect depends on the structure of the chemical compound and are greater influenced by heavy atoms. The dataset is in the format (G,y) where G represents the chemical compounds, the vertices represent the atoms, and the edges represent the bonds between atoms.

4.2. Optimizer

We will be choosing between the Adam optimizer and Stochastic Gradient Descent (SGD). SGD + momentum and Adam are known to perform similarly to each other; however, the choice of optimizer ultimately depends on the type of data that we worked with. In our case, we performed a grid search to obtain the optimal values of model hyperparameters for both the Adam optimizer and SGD; from there we picked the optimizer with the higher accuracy.

The Adam optimizer is a replacement optimization algorithm that can be applied to stochastic gradient descent. According to its documentation, this optimizer combines Adaptive Gradient Descent (AdaGrad) and Root Mean Squared Propagation (RMSProp), which results in an optimization algorithm that can handle sparse gradients on noisy gradients. Additionally, this optimizer is effective for problems with large data and parameter quantity. [7]

SGD, is well know and applied in many problems. This optimizer takes batches of data to compute the gradient descent, this saves time as it does not require to use all the data in order to compute the gradient at each point. The SGD optimizer also maintains a non-changing learning rate, whereas Adam optimizer does not maintain a non-changing learning rate. One can conclude that the Adam optimizer might do better due to its more complex nature, but as showed in our experimental results, it will depend a lot on the data that is being learned by the model.

4.3. Momentum

Stochastic Gradient Descent with momentum has faster convergence. The equation for the momentum is $U = mgh$, where U is momentum, m is mass, g is gravity, and h is height. In gradient descent, the loss can be perceived as "the height of a hilly terrain" [7], and hence we know that height (loss) is proportional to the momentum U. The force on a particle can be perceived as related to the gradient of potential energy, $F = -\nabla U$. Additionally, the formula for force is $F = ma$, so we can find a formulation to express proportionality to acceleration to speed up Gradient Descent. It is important to keep in mind that in reality, the momentum variable behaves more like a coefficient of friction, since it reduces the kinetic energy of the system.

To further enhance the results of SGD we used Nesterov momentum. Nesterov momentum is an extension of momentum that involves calculating the decaying moving average of the gradients of projected positions in the search space rather than the actual positions themselves. When the current iteration computes the position of a parameter vector, we know the momentum alone acts with some effect upon that parameter vector. When computing the gradient, we can treat the future approximate position plus the affected parameter vector as a "look ahead". As described in source [7], Figure 2 describes the scenario of Nesterov Momentum update:

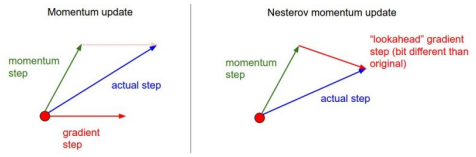

Figure 2. Evaluate gradient at the "looked ahead" position

4.4. Our Results

We performed a grid search to find the best possible parameters for a higher accuracy. Our grid search consisted of searching through:

1. Dropout percentages: $0\%, 40\%, 50\%, 60\%, 70\%$
2. Hidden layers: $128, 256, 512$
3. Optimizers: $Adam, SGD$

Where SGD consisted of parameters for momentum, dampening, learning rate, weight decay and the option of using Nesterov.

"An End-to-End Deep Learning Architecture for Graph Classification" was initially using the Adam optimizer with hidden size of 128, dropout of 0.5, and learning rate of 0.0001. The lower value of hidden size that they used could have been under-fitting the model, whereas the lower drop percentage that they used could have been over fitting the model. To see the accuracy graph with original parameters and Adam optimizer, refer to Figure 3.

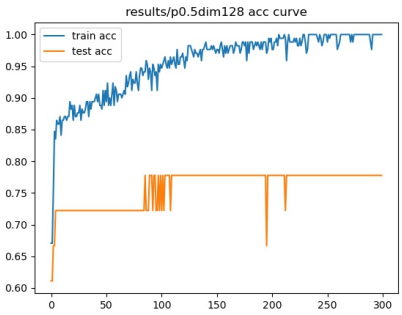

Figure 3. Parameters: 50% dropout, dimension 128, MLP Classifier with Adam

It is important to keep in might that this paper did have a better accuracy for a larger set of data-sets and not only for the MUTAG dataset, and hence they might have not reached the optimal for this specific dataset MUTAG. Since we made adjustments for MUTAG during this specific grid search, we had an increase in accuracy of 3%, see Figure 4 for an accuracy plot.

The best results were obtained with dropout of 60% and hidden size 256, while using the SGD optimizer. We believe that these numbers are best for the MUTAG dataset for the specific model that we worked with.

Additionally, we might have obtained higher results if we tested other optimizers and performed grid search for their respective parameters. In our case, we were only able to explore these two optimizers in detail.

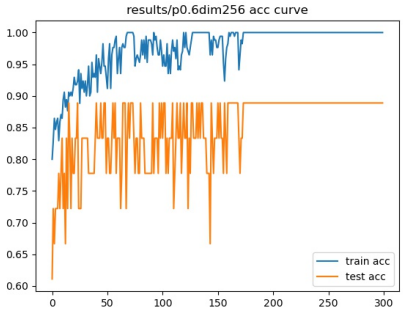

Figure 4. Parameters: 60% dropout, dimension 256, MLP Classifier with SGD, and 88% testing accuracy

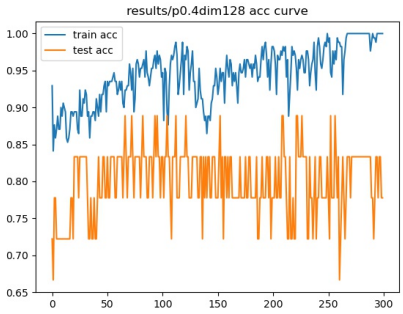

Figure 5. Parameters: 40% dropout, dimension 128, MLP Classifier with SGD, and 77% testing accuracy

4.5. Benchmarks on MUTAG dataset

There are multiple papers that have published higher results for the MUTAG dataset compared to "An End-to-End Deep Learning Architecture for Graph Classification" [8]. We believe that this can be due to difference in architecture. These higher results could be obtained due to models being more focused towards the MUTAG data-set. Additionally, the models that performed higher on MUTAG, may perform badly on other data-sets.

By doing a grid-search on dropout, hidden layers, and optimizer, we were able to increase the accuracy by 3%. The original accuracy that this paper reached was 85.83%, the highest score in Papers with Code was 95%, and out results were 88%.

4.6. Adding a layer to the CNN

An example plot from our findings is shown in Figure 6. We find that when adding extra layers to the CNN, we do not see an improvement in results. There can be many reasons for this, but the intuition is that it does not matter how deep the CNN is if the embedding does not contain enough information. Here we see even with large dropout (60 to

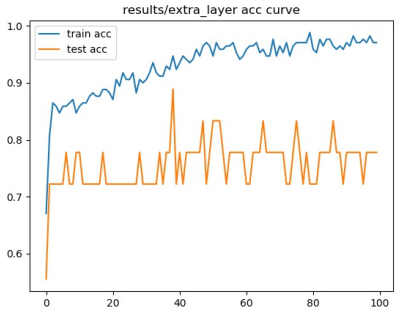

Figure 6. Accuracy plot for adding another layer to CNN

80 percent) we have a classic overfitting example. This is because with more parameters we can easily achieve 1.0 train accuracy (even with dropout), but the problem seems to be that the input itself is not general enough.

4.7. Adding more convolutions to GCN

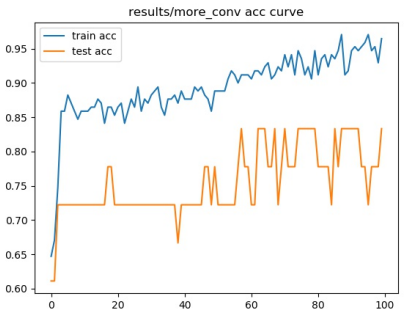

Figure 7. Accuracy plot for adding another layer to GCN

An example plot from our findings is shown in Figure 7. We find that when adding extra layers to the GCN, i.e. increasing h, we do not see an improvement in results. Here I added one more layer to make $h = 5$, but it seems that the results are very similar to baseline in the paper that reaches 85 percent. I think it's possible that having a deeper GCN does not necessarily encode more information. Maybe there is a limit on the amount of useful information that can be feasibly stored in the embedding using the GCN layers provided. Therefore it still overfits because the representation of the test data is so different from the representation of the training data. When exploring even greater h (of 6, 7, 8) we find similar results to the graph in Figure 7

4.8. Experimenting with sortpool output

An example plot from our findings is shown in Figure 8. We find that when adding lowering the k value by half, we have very poor results. This shows us that we do need a large

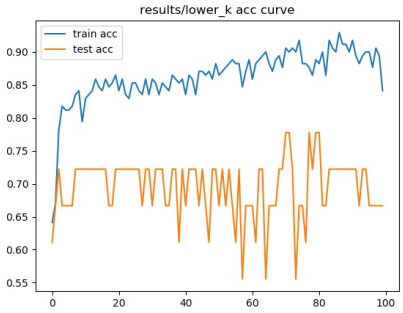

Figure 8. Accuracy plot for decreasing k to half of baseline value

k value for our embedding to encode the data. This means that we overfit by training on all the very low-dimensional training data, but cannot generalise to the test data because it has a different distribution. Other results show that small k decreases accuracy even further, whereas a larger k increases accuracy up to a certain point. This can indicate that the embedding needs larger output dimension, but can only embed data and crucial information up to a certain point. That is, if we increase k to be the size of the largest input graph, the problem is that we pad many zeros in the graphs. This is not too good for our CNN though, and also shows poor accuracy.

5. Challenges, work's limitation, and how this work can be further extended

There are two main challenges: 1) how to extract useful features characterizing the rich information encoded in a graph for classification purposes, and 2) how to sequentially read a graph in a meaningful and consistent order. To address the first challenge, "An End-to-End Deep Learning Architecture for Graph Classification" designed a localized graph convolution model and showed its connection with two graph kernels. To address the second challenge, they designed a novel SortPooling layer which sorts graph vertices in a consistent order so that traditional neural networks can be trained on the graphs. In addition, because graphs do not have fixed size, it is hard to use traditional Neural Network approaches. Therefore, special care is needed to ensure that our model trains accurately and can handle a variety of different graphs. Most previous works have relatively worse performance on graph classification tasks. One reason for this is that after extracting localized vertex features, these features are directly summed up as a graph-level feature used for graph classification.

6. Conclusion

When comparing our results to the "An End-to-End Deep Learning Architecture for Graph Classification" results, we observed the accuracy for our target dataset, MUTAG, was higher after the optimizer and other parameters was changed. This is most likely because the paper's algorithm emphasized generality and application on multiple data-sets, whereas our algorithm was focused on achieving the highest accuracy possible specifically for the MUTAG dataset. Our optimization demonstrates that the parameters of the DGCNN can be adjusted per dataset to achieve a very accurate model.

In conclusion, the paper that we used for our project was not in the top 10 best results for our dataset but was instead more generalized to other data-sets. After searching for better accuracy, we were able to tailor it more for our dataset MUTAG.

7. Division of work

7.1. Jessica

I worked on the initial research to decide which paper to work on. Worked one proposal but can't remember which section :) I then worked on experimental evaluation, and running the different optimizers to pick which to use and which parameters to focus on for the grid search. The parameters explored were hidden layer, dropout, optimizer, and within optimizer, the other parameters explored where learning rate, momentum, rate decay and Nesterov. For the final report I researched more about momentum and Nesterov for SGD, and wrote section 4.2 4.3 4.4 4.5 and conclusion.

7.2. Puja

I focused on the experimental evaluation. I researched about different optimizers (Adam and SGD) as well as the hyper parameters we were trying to tune using the gridsearch (hidden layer, learning rate, momentum, rate decay, dropout). For the project proposal I wrote the answers for all the questions regarding the "Problem Formulation". For the final paper I wrote the abstract, part of the introduction, and part of section 4.

7.3. Peter

I along with the rest of the group initially researched the topic that we eventually did our paper on. I also developed the project schedule and planned division of work for the rest of the quarter. In the project proposal, I wrote the comprehensive review portion. In the final report, I aided in the introduction, comprehensive literature review (from the project proposal), section 4.1, and the conclusion.

7.4. Tameez

I created plotting features to show us the loss and accuracy and save our results in txt files if we need to review it. I also created the "gridsearch.sh" script which helps us run the program through many different settings like changing dropout or hidden dimension. I tested varying the sortpool k parameter and tested the number of conv layers h. Lastly, I added a new layer that can be commented in/out in the CNN. For the final paper I wrote section 3 and part of section 4

8. Citations and References

[1] https://github.com/muhanzhang/pytorch_DGCNN

[2] https://github.com/toenshoff/CRaWl

[3] https://github.com/rusty1s/pytorch_geometric

[4] https://github.com/navid-naderi/WEGL

[5] https://github.com/diningphil/gnn-comparison

[6] https://machinelearningmastery.com

[7] https://cs231n.github.io/neural-networks-3/

[8] https://paperswithcode.com/sota/graph-classification-on-mutag

[9]