Investigation of Graph Convolutional Networks

Christopher Fifty [*1] **Alex Libman** [*1] **Danny Qiu** [*1]

Abstract

We present a novel, extensible, and research-oriented implementation of the popular Graph Convolutional Network. Our system is implemented in Python 3.6 and exclusively uses NumPy primitives to avoid the overhead and framework-specific knowledge inherent in using a popular deep learning framework such as TensorFlow or PyTorch. Moreover, the NumPy Graph Convolutional Network (numpyGCN) framework outperforms both TensorFlow and PyTorch GCN implementations in wall-clock training/inference time. numpyGCN is particularly suited to a research setting due to its extensibility and usage of low-level operations. Lastly, we implement and investigate Graph Convolutional Network compression using the hashing trick and find that the size of the model's weight parameters can be compressed by a factor of 8 with comparable accuracy to the uncompressed baseline. The hashed GCN vastly outperforms a reduced-size GCN with an equivalent number of parameters.

1. Introduction and Related Work

Semi-supervised learning paradigms exploit structure, regularities, and features of unlabeled data – in conjunction with a small set of label information – to make predictions. Tasks of this nature can take a variety of forms; however, one of the most popular tasks involve learning over graph-structured data. In this paradigm, we would like to use a small subset of labeled nodes to classify the vast majority of unlabeled nodes within a graph.

Conventional graph-based semi-supervised learning techniques employ a form of Laplacian regulariation (Zhou et al., 2004) in the loss function to smooth label information from the labeled nodes across the entire graph.

$$\mathcal{L} = \mathcal{L}_0 + \lambda \mathcal{L}_{reg} \tag{1}$$

Here $\mathcal{L}_0$ represents the supervised loss with respect to the labeled portion of the graph and $\mathcal{L}_{reg}$ is the Laplacian regularization term. $\mathcal{L}_{reg}$ propagates label information from the labeled nodes to the unlabeled ones such that nodes which are connected are likely to have the same label. A minimization of this two-fold loss function estimates a function f that assigns each node to a label. However, recent work in the domain of graph-structured semi-supervised learning has shifted focus to deep learning methodologies.

Graph Convolutional Networks (GCNs) (Kipf & Welling, 2016) are an efficient variant of CNNs which operate directly on graphs. GCNs are motivated by first-order Chebyshev approximations of learned spectral filters, thereby avoiding an explicit graph-based regularization term and allowing for a more flexible model. GCN layer k propagates feature information from the k^{th}-order neighborhood of node n_i to adapt its feature representation. The final j^{th} GCN layer consolidates information from the 1^{st}, 2^{nd}, ..., $j-1^{th}$ order neighborhood (weighing lower-order neighborhoods more heavily than higher-order neighborhoods) to predict node labels in the graph. GCNs achieve state-of-the-art results on a number of semi-supervised graph classification tasks, and in particular, citation networks.

The currently available implementations of GCNs exist solely in Deep Learning frameworks such as TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2017). Such frameworks construct a unified dataflow graph for the model and its necessary computations. Because the computations and data exist only in the framework, TensorFlow and PyTorch provide auto-differentiation capabilities that remove the need to manually derive the backprogation rule for gradient updates. With our GCN implementation in NumPy, we believe that this is the first paper to show the derivation of the backpropagation rule for GCNs, in particular, the two-layer GCN used in the original paper (Kipf & Welling, 2016).

With the vast increase in model sizes in recent years as well as the desire to be able to run such models in settings where computational resources are limited (such as on mobile devices), compression techniques have been developed to al-

[*]Equal contribution [1]Department of Computer Science, Cornell University, Ithaca, New York. Correspondence to: Christopher Fifty <cjf92@cornell.edu>, Alex Libman <asl237@cornell.edu>, Danny Qiu <dq29@cornell.edu>.

Proceedings of Advanced Machine Learning Systems 2018, Ithaca, NY, Upson Hall 142, 2018.

low for neural networks to be encoded into a more compact form which encodes a very similar model, reducing memory usage as well as improving inference times in some architectures. Compression of neural networks using the hashing trick (Chen et al., 2015) assumes that many weights learned during training are similar, and constrains random groups of weight parameters in between layers using a hash function to reference a single stored parameter. This reduces the number of weights needed to be stored during training and inference while avoiding cutting out connections entirely.

2. numpyGCN

We introduce a novel GCN implementation (numpyGCN), specifically designed for machine learning researchers and students, to allow for low-level control over every operation and augmentation to a Graph Convolutional Network architecture. numpyGCN is created with minimal overhead, optimized matrix operations, and intelligent use of sparse Coordinate List (COO) matrices for maximum performance. Moreover, numpyGCN's wall-clock performance is superior to both TensorFlow and PyTorch implementations (Kipf & Welling, 2016).

2.1. Essential Network Operations

Identical to the methods presented by Kipf and Welling (Kipf & Welling, 2016), we define numpyGCN's forward pass to be:

$$H^{l+1} = \sigma(\hat{A}H^l W_l) \tag{2}$$

H^l is the activations matrix in the l^{th} layer such that H^0 = X (i.e. the input feature matrix). $\hat{A} = \widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}$ is the normalized adjacency matrix. We define $\widetilde{A} = I + A$ to be the adjacency matrix of our graph with self-loops and $\widetilde{D} = I + D$ is the diagonal matrix of $\widetilde{A}$. The weight matrix W_l is the learned parameters at layer l.

We implement the following functions using exclusively NumPy primitives: forward pass, backpropagation, gradient descent update, calculate loss, compute accuracy, and predict. We have made our code publicly available to encourage researchers and students to use our framework as well as demonstrate the reproducibility of our results [1]. The derivation of the backpropagation rule for the two-layer GCN in Equation 3 can be found in the Appendix.

2.2. Network Augmentations

To demonstrate the extensibility and low-level control of our framework, we implement early stopping, dropout, L2 weight decay (first layer only), gradient descent, and adaptive learning rate. Our implementation of popular deep learning techniques highlight the capability of our architecture to implement the most state-of-the-art deep learning augmentations. While we discussed implementing the Adam Optimizer (Kingma & Ba, 2014), we settled on a gradient descent optimizer with an adaptive learning rate so we could spend more time focusing on other aspects of our model.

[1] https://github.com/cfifty/numpyGCN

2.3. Case Study

One of the primary advantages of numpyGCN over TensorFlow and PyTorch is its customizability. Any feature of the model can be modified and changed – regardless of whether or not TensorFlow or PyTorch implements an API for it – and many low-level changes can be implemented much simpler than in TensorFlow and PyTorch. For example, let's consider the case of adding random noise $n \sim \mathcal{N}(0, 0.001)$ to our weights as a regularization technique as described in (Kawakami, 2008).

The algorithm for weight noise regularization is straightforward:

```
for each node in our graph:
    Save initial weights
    Add N(0, 0.001) noise to weights
    Calculate the gradient update via bp
    Restore the initial weights
    Update weights with gradient
```

Implementing this work in numpyGCN is just as simple and can be accomplished with the following basic and intuitive modifications to the architecture:

```
# before the forward pass during bp
if self.random_noise:
    tmp_W1 = np.copy(self.W_1)
    tmp_W2 = np.copy(self.W_2)
    self.W_1 += np.random.normal(0,
        0.001, self.W_1.shape)
    self.W_2 += np.random.normal(0,
        0.001, self.W_2.shape)

# after gradients are computed
if self.random_noise:
    self.W_1, self.W_2 = tmp_W1, tmp_W2
```

However, in TensorFlow, this minor modification is somewhat difficult to integrate into the model. Restoring the pre-noise weights requires overriding the gradient update in TensorFlow for that layer. TensorFlow allows this override with the `custom_gradient` decorator, which gives users the ability to provide their own gradient function for the backwards pass alongside the forward computation. With `custom_gradient`, we can return an identical gradient as originally computed by TensorFlow and subtract out the added Gaussian noise from the weights before the gradient update is applied. However, this change requires refactoring the forward pass function inside the layer class by adding a nested function to perform the forward computation, as the custom gradient decorator cannot be applied to a class

Dataset	Type	Nodes	Edges	Classes	Features	Train	Validation	Test
Citeseer	Citation Network	3,327	4,732	6	4,703	140	500	1000
Cora	Citation Network	2,708	5,429	7	1,433	140	500	1000

Table 1. Dataset statistics: reported in (Yang et al., 2016)

level function. Furthermore, issues arise with typing when sparse inputs are passed into the custom gradient decorator. Ultimately, although we explored incorporating this into the TensorFlow implementation, it proved too challenging.

3. Experiments

3.1. Datasets

We evaluate the performance of a NumPy, Tensorflow (tfGCN), and PyTorch (torchGCN) GCN implementation on two citation network datasets. Citation networks show how a group of documents are related to each other through citations, and as a result, are typically represented with a graph-like data structure. In this paradigm, nodes represent individual documents within the citation network and a directed edge from node v_i to node v_j indicates that the document represented by v_i cites the document represented by v_j. To further simplify this challenge, we represent citations between documents with undirected edges.

Building upon the work of (Sen et al., 2008) we consider two citation network datasets, Citeseer and Cora, for use in our numpyGCN analysis. The features corresponding to each document consist of a sparse bag-of-words model, and the sparse adjacency matrix included in this dataset is representative of citation links among documents.

To maximize comparison with related methods as well as past work, we use the same dataset splits as in (Yang et al., 2016) with 500 additional nodes in our validation set for hyperparameter tuning. Precise statistics about the nodes, edges, classes, and features for each citation network dataset are provided in Table 1.

3.2. Setup

In our NumPy implementation, we train a two-layer GCN and evaluate its predictive performance on the Citeseer and Cora datasets. Following the work done in (Glorot & Bengio, 2010) we randomly initialize the weight parameters following a uniform distribution pursuant to the size of the hidden dimension, and row-normalize the input feature vectors as well.

We use a gradient descent optimizer with a custom adaptive learning rate and cross entropy loss to train a 2-layer numpyGCN model. We choose to use a ReLU activation between the layers of our model, except after the final layer, at which point we use a softmax activation function. Accordingly, our output $Z \in \mathbb{R}^{n\times|o|}$ (where $|o|$ is the size of the output layer) is:

$$Z = \mathrm{softmax}\left(\hat{A}\,\mathrm{ReLU}\left(\hat{A}XW_0\right)W_1\right) \tag{3}$$

Our numpyGCN is augmented with early stopping with a window size of 10, dropout, and L2 weight decay. We used a first-layer dropout rate of 0.0010, a L2 weight decay value of 0.0008, a learning rate of 0.1475, and halved the learning rate if the training loss increased between epochs. All hyperparameters were chosen by running 500 iterations of $hyperopt$'s tree-structure Parzen estimator (Bergstra et al., 2011) on the validation set for the Cora dataset. Lastly, to maximize comparability with the TensforFlow and PyTorch GCN implementations, we train our model for a maximum of 200 epochs.

As described in (Kipf & Welling, 2016), the TensorFlow model is similarly trained for a maximum of 200 epochs, with an initial learning rate of 0.01, a dropout rate of 0.5, a L2 weight decay of 0.0005, and a 10 step early stopping window. The TensorFlow GCN's hyperparameters were tuned on Cora's validation set as well. Unlike the TensorFlow implementation, torchGCN is a fundamentally different model. The adjacency matrix is normalized with $\widetilde{D}^{-1}\widetilde{A}$, and different dataset splits are used, but it should have comparable accuracy and performance to the TensorFlow model. Lastly, to reduce the impact of randomness on our analysis, we use the mean and standard deviation of 10 runs of our model for all subsequent charts and graphs.

3.3. HashNet Compression

We implement compression of neural network weight parameters inside TensorFlow using the hashing trick, which uses a hash function to group and constrain multiple connection weights between layers in the network to the same weight value. We compare the statistical performance (via test set accuracy) of a GCN that shares weights with this method versus that of the baseline feed-forward GCN on the Cora dataset. To limit the effect of randomness in a non-convex optimization paradigm, we use 5 randomly seeded runs of the model at each level of compression and report the median value as well as the standard deviation of the samples. All hashed models are trained with early stopping, dropout, L2 weight decay, and optimized with Adam.

Model	Cora Accuracy	Citeseer Accuracy	Cora Time(s)	Citeseer Time (s)
torchGCN	**83.4 ± .3**	-	7.57	-
tfGCN	81.5 ± .6	**70.5 ± .8**	3.86	8.49
numpyGCN	80.2 ± .4	69.8 ± .8	**2.96**	**7.35**

Table 2. Summary of results in terms of classification accuracy and training time

4. Results

Our numpyGCN implementation is comparable in performance to that of the TensorFlow and PyTorch implementations as shown in Table 2, and we reason that any differences between numpyGCN and tfGCN are due to the choice of optimizer (standard gradient descent vs Adam). Differences between torchGCN and alternative models are also easily explained. As mentioned in section 3.2, torchGCN normalizes the adjacency matrix differently and uses different dataset splits. Moreover, the wall-clock performance of numpyGCN is superior to both tfGCN and torchGCN. The Citeseer dataset results are not reported for the PyTorch GCN due to implementation constraints and dataset intricacies.

4.1. Ablation Studies

According to (Kipf & Welling, 2016), Graph Convolutional networks function optimally when regularization techniques are heavily applied. After tuning hyperparameters, a dropout rate of 0.5, L2 weight decay of $5 \cdot 10^{-4}$, and 10-step early stopping window are employed in their TensorFlow model for optimal results. However, given the size of their network for the Cora dataset (2-layer GCN, $W_0 \in \mathbb{R}^{1433\times16}$, $W_1 \in \mathbb{R}^{16\times7}$) we chose to investigate the effect of regularization on the performance of GCNs within the numpyGCN framework as an ablation study.

In Figure 1 we see that the presence of dropout weight regularization and early stopping regularization do not significantly affect the test accuracy on both the Cora and Citeseer dataset. Drawing from the ideas presented in (Caruana et al., 2001), we believe the relatively small hidden layer size of 16 can be attributed to the model's resistance to overfitting. Our ablation study indicates the the regularization techniques commonly employed in most – if not all – GCN architectures are superfluous, and the inclusion of these techniques within the GCN architecture may be derived from an over reliance on past deep learning literature and standard practice.

4.2. HashNet Results

Figure 2*(a)* shows the performance of a GCN where the compression only occurs in the first layer. We opt to do this since the base GCN only contains 112 connections between the first and second layer, whereas the incoming connections

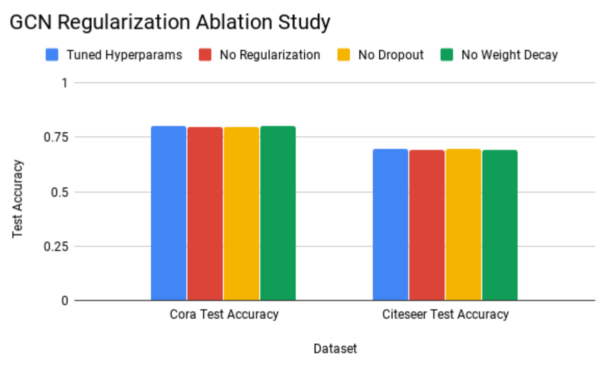
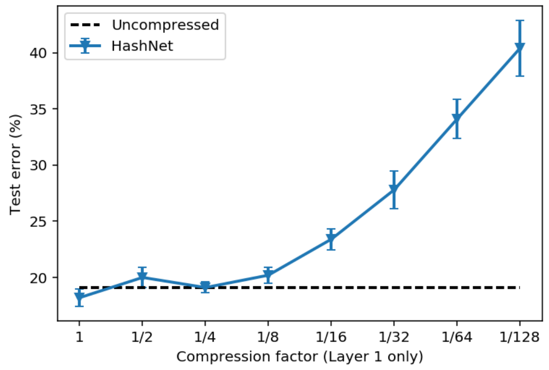


Figure 1. GCN Regularization Ablation Results on Cora and Citeseer

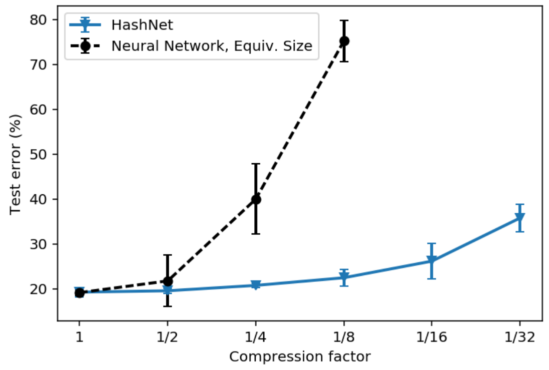
for the first layer contain roughly 99.5% of the total weight parameters in the entire model. This ensures that we can compress the model to a greater degree without introducing a severe information bottleneck. Moreover, compressing the second layer connections does not significantly reduce the model's memory constraints.

In this setting, the performance of the compressed GCN is comparable to that of the baseline up to a compression factor of $1/8$, after which the test error of the compressed model is more than 1% above the baseline. The test error continues to increase as the model becomes more compressed, however the accuracy is still acceptable (under 10% increase) even when the number of weights stored is only about 3% that of the original model. This result indicates that there are many similar weights learned in the original model which the hashing trick compression algorithm exploits.

Figure 2*(b)* compares the accuracy of a GCN compressed via the hashing trick to an equivalently shrunk GCN with an equal number of parameters. Both layers of the hashed GCN are being compressed for fair comparison with the shrunk GCN. Even with the second layer being compressed $1/16\times$ to only 14 unique stored weights, the compressed GCN still manages to achieve a sub 30% test error rate. Meanwhile, the reduced-size neural network struggles to learn the dataset once less than half the original number of weights remain. Much of the expressiveness of the model is lost due to the explicit drop in the number of connections. Notably, the variance in the test time accuracy of the



(a) Compressed first layer



(b) Both layers compressed vs. reduced-size GCN with equivalent number of parameters

Figure 2. Test error rates of GCNs compressed with the hashing trick on Cora

reduced-size GCNs is quite large, showing that the models without compression in this regime are incredibly sensitive to weight matrix initialization. Alternatively, reduced-sized models need many more training iterations to converge.

5. Conclusion and Future Work

In this work, we present a novel implementation of Graph Convolutional Networks in a system that exposes low-level neural network operations. In addition to superior wall-clock times to those of TensorFlow and PyTorch GCN implementations, numpyGCN is highly extensible and ideal for research settings. Moreover, we apply the hashing trick to GCNs and achieve an $8\times$ compression of the weight parameters without significant drop in accuracy, showcasing the effectiveness of the hashing trick on GCNs.

We hope this work inspires alternative, light-weight, and easy-to-use frameworks to be considered and developed for a research setting. The utility of TensorFlow and PyTorch make these frameworks ideal for a wide variety of applications; however, this utility and ability to ”do it all” can make TensorFlow and PyTorch unwieldy and cumbersome in a research setting.

6. Acknowledgements

We would like to thank Chris De Sa[2] for his assistance in deriving $\frac{\partial L}{\partial W_0}$ and advisement throughout the semester. We would also like to thank Kilian Weinberger, Felix Wu, and Amauri Holanda for first introducing one of our team members to the graph-based semi-supervised learning paradigms and helping him understand the mathematics and theory behind these methods.

[2]http://www.cs.cornell.edu/~cdesa/

References

Abadi, Martín, Barham, Paul, Chen, Jianmin, Chen, Zhifeng, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Irving, Geoffrey, Isard, Michael, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, 2016.

Bergstra, James S, Bardenet, Rémi, Bengio, Yoshua, and Kégl, Balázs. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pp. 2546–2554, 2011.

Caruana, Rich, Lawrence, Steve, and Giles, C Lee. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in neural information processing systems*, pp. 402–408, 2001.

Chen, Wenlin, Wilson, James, Tyree, Stephen, Weinberger, Kilian, and Chen, Yixin. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pp. 2285–2294, 2015.

Glorot, Xavier and Bengio, Yoshua. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

Kawakami, Kazuya. *Supervised Sequence Labelling with Recurrent Neural Networks*. PhD thesis, PhD thesis. Ph. D. thesis, Technical University of Munich, 2008.

Kingma, Diederik P. and Ba, Jimmy. Adam: A method for stochastic optimization. *CoRR*, 2014.

Kipf, Thomas N and Welling, Max. Semi-supervised classification with graph convolutional networks. 2016.

Paszke, Adam, Gross, Sam, Chintala, Soumith, Chanan, Gregory, Yang, Edward, DeVito, Zachary, Lin, Zeming, Desmaison, Alban, Antiga, Luca, and Lerer, Adam. Automatic differentiation in pytorch. 2017.

Sen, Prithviraj, Namata, Galileo, Bilgic, Mustafa, Getoor, Lise, Galligher, Brian, and Eliassi-Rad, Tina. Collective classification in network data. 2008.

Yang, Zhilin, Cohen, William W, and Salakhutdinov, Ruslan. Revisiting semi-supervised learning with graph embeddings. 2016.

Zhou, Denny, Bousquet, Olivier, Lal, Thomas N., Weston, Jason, and Schölkopf, Bernhard. Learning with local and global consistency. In *Advances in Neural Information Processing Systems 16*. 2004.

7. Appendix

7.1. Backpropagation

Define $X \in \mathbb{R}^{n\times 1433}$ as the input feature matrix and $Y \in \mathbb{R}^{n\times 7}$ as the matrix containing the one-hot encodings of the available labels, i.e. $Y_{ij} = 1$ if feature vector i is labeled as class j, and the rest of the entries in row i are zero.

We begin with the forward pass of a GCN:

$$Z = \text{softmax}\left(\hat{A}\ \text{ReLU}\left(\hat{A}XW_0\right)W_1\right)$$

and break this equation down into several natural and more manageable parts:

$$\begin{aligned} \text{in1} &= \hat{A}XW_0 \\ \text{out1} &= \text{ReLU}(\text{in1}) \\ \text{in2} &= \hat{A}(\text{out1})W_1 \\ Z &= \text{softmax}(\text{in2}) \end{aligned}$$

We now aim to compute $\frac{\partial L}{\partial W_1}$ and $\frac{\partial L}{\partial W_2}$ to use in gradient descent to update numpyGCN's parameters. Similar to other deep learning models with a final softmax activation, numpyGCN uses cross-entropy loss given by $L = \text{cross\_entropy}(Z, Y)$.

Computing the gradient of the loss with respect to the second-layer weight parameter W_1:

$$\begin{aligned} \frac{\partial L}{\partial\,\text{in2}} &= Z - Y \\ \frac{\partial\,\text{in2}}{\partial W_1} &= \hat{A}\cdot\text{out1}^T \\ \frac{\partial L}{\partial W_1} &= \frac{\partial\,\text{in2}}{\partial W_1}\frac{\partial L}{\partial\,\text{in2}} \\ &= (\hat{A}\cdot\text{out1})^T\cdot(Z-Y) \end{aligned}$$

Computing the gradient of the loss with respect to the first-layer weight parameter W_0:

$$\begin{aligned} \frac{\partial L}{\partial\,\text{out1}} &= \hat{A}^T\cdot(Z-Y)\cdot W_1^T \\ \frac{\partial\,\text{out1}}{\partial\,\text{in1}} &= \frac{\partial}{\partial\,\text{in1}}\text{ReLU}(\text{in1}) = \begin{cases} 1 & \text{in1}_{ij} > 0 \\ 0 & \text{in1}_{ij} \le 0 \end{cases} \\ \frac{\partial\,\text{in1}}{\partial W_0} &= (\hat{A}\cdot X)^T \\ \frac{\partial L}{\partial W_0} &= \frac{\partial\,\text{in1}}{\partial W_0}\frac{\partial\,\text{out1}}{\partial\,\text{in1}}\frac{\partial L}{\partial\,\text{out1}} \end{aligned}$$

Which yields a final gradient update for W_0 of

$$\frac{\partial L}{\partial W_0} = (\hat{A}\cdot X)^T\cdot\left[\left[\frac{\partial}{\partial\,\text{in1}}\text{ReLU}(\text{in1})\right]\odot(\hat{A}^T\cdot(Z-Y)\cdot W_1^T)\right]$$