Neural Computing and Applications (2021) 33:4807–4814
https://doi.org/10.1007/s00521-020-05283-x

ORIGINAL ARTICLE

# A noise injection strategy for graph autoencoder training

Yingfeng Wang[1,4] · Biyun Xu[2] · Myungjae Kwak[1] · Xiaoqin Zeng[3]

Received: 27 March 2020 / Accepted: 5 August 2020 / Published online: 11 August 2020


**Abstract**
Graph autoencoder can map graph data into a low-dimensional space. It is a powerful graph embedding method applied in graph analytics to lower the computational cost. Researchers have developed different graph autoencoders for addressing different needs. This paper proposes a strategy based on noise injection for graph autoencoder training. This is a general training strategy that can flexibly fit most existing training algorithms. The experimental results verify this general strategy can significantly reduce overfitting and identify the noise rate setting for consistent training performance improvement.

**Keywords** Graph autoencoder · Noise injection · Training algorithm · Overfitting

## 1 Introduction

Autoencoder is a neural network composed of encoder and decoder [2]. Encoder converts the input data into an abstract representation, while decoder reconstructs the original input data from the output of encoder. Graph autoencoder is based on graph neural network, whose input data are graph information. The great potential of graph autoencoder in dimensionality reduction has motivated scientists to apply it in graph embedding [3–5]. Graph autoencoder embeds graph data based on matrix factorization [3, 6]. It aims to preserve the graph structure of the input matrix, e.g., adjacency matrix, in a low-dimensional space by matrix factorization [7].

Since the importance of graph autoencoder is recognized, more and more training algorithms have been designed for meeting different needs. Conventional graph autoencoders, such as GAE [8], SDNE [9], and DRNE [10], aim to preserve the graph information in a low-dimensional space. Variational graph autoencoders, e.g., VGAE [8], focus on building a graph autoencoder allowing users to feed random data or interpolated data to the decoder. Recently, Samanta et al. proposed NeVAE, a variational graph autoencoder using a deep generative model, for processing molecular graphs [11], while Grover et al. developed Graphite, a variational graph autoencoder using a scalable deep generative model [12]. Besides preserving the graph information, the encoder needs to learn the distribution of training samples in training. Denoising graph autoencoders, e.g., DNGR [13], can automatically filter out noise. Therefore, the related training algorithms use corrupted input in training, while the desired output of decoder is the original input. Adversarially regularized graph autoencoders, e.g., ARGE [6], ARVGE [6], and NetRA [14], attempt to train an autoencoder with the feature of distinguishing faked encoded representation. The related training platform is composed of generator and discriminator [15]. The former is trained to confuse the latter, while the latter is trained to classify true samples and generated data. The corresponding training algorithm based on this platform is good at reducing autoencoder learning failures caused by too much capacity of encoder and decoder.

The success of a graph autoencoder is strongly correlated to the performance of its training algorithm. A general training strategy, which can improve most training algorithms of graph autoencoder, will advance the study

A preliminary version of this work appeared in the proceedings of the international conference on machine learning and computing (ICMLC 2020) [1].

✉ Yingfeng Wang
yingfeng-wang@utc.edu

[1] Department of Information Technology, Middle Georgia State University, Macon, GA 31206, USA

[2] Beijing Kubao Technology Company, Beijing 100124, China

[3] Institute of Intelligence Science and Technology, Hohai University, Nanjing 210098, Jiangsu, China

[4] Present Address: Department of Computer Science and Engineering, University of Tennessee at Chattanooga, Chattanooga, TN 37403, USA

and application of graph autoencoder significantly. The goal of this work is to develop such a training strategy. Noise injection is a general training strategy, which was proposed three decades ago [16]. Most studies of this strategy focus on conventional feedforward neural networks such as multilayer perceptron (MLP) [17–28] and radial basis function (RBF) [27, 28]. Although the successes of these works strongly encourage us to apply the noise injection strategy in the training of graph autoencoder, there are still some challenges. The first challenge is how to inject noise. All existing works directly add random noise to the input vector. However, this method does not fit graph information. For instance, all elements in the adjacency matrix are one or zero, which represent the connection status between nodes. Directly adding random noise to elements of the adjacency matrix makes the values of these elements hard to interpret. Furthermore, the experience of setting the noise rate on traditional feedforward neural networks may not work on a graph neural network.

This paper proposes a noise injection strategy for graph encoder training with addressing the above-mentioned challenges. We developed a simple noise injection method for injecting noise into the input graph information. Additionally, this work investigates the impact of noise rate on training performance in experiments based on conventional graph autoencoders, variational graph autoencoders, and adversarially regularized graph autoencoders.

The rest of this paper is organized as follows. Section 2 introduces the basic architecture of a graph autoencoder. The noise injection strategy is presented in Sect. 3. Section 4 details and discusses the results of experiments conducted for verifying the effectiveness of the proposed strategy. Finally, Sect. 5 concludes the paper and proposes future work.

## 2 The architecture of graph autoencoder

A graph autoencoder is composed of an encoder and a decoder. The upper part of Fig. 1 is a diagram of a general graph autoencoder. The input graph data are encoded by the encoder. The output of encoder is the input of decoder. Decoder can reconstruct the original input graph data. This paper focuses on GAE, VGAE, ARGE, and ARVGE for covering conventional graph autoencoders, variational graph autoencoders, and adversarially regularized graph autoencoders. We use the basic architecture of these models as an example. The diagram of this architecture is given in the lower part of Fig. 1. The encoder, which was first proposed by Kipf and Welling [8], is based on a graph convolutional network (GCN) [29]. The input graph data can be represented by $(A, X)$, where $A$ is the adjacency matrix, while $X$ is the node feature matrix. If the number of nodes is $n$, $A$ is a $n \times n$ matrix. It is worth noting that all diagonal elements of A are set to one. If the number of features is $m$, $X$ is a $n \times m$ matrix. We can build the $n \times n$ degree matrix $D$ based on $A$ by the following expression.

$$D_{ij} = \begin{cases} \sum_{k=1}^{n} A_{ik}, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \tag{1}$$

where $1 \leq i, j \leq n$. Assume the GCN model used in the encoder has $k$ layers. The output of the $l$th layer of GCN, represented by $Z^{(l)}$, can be computed by the following formula.

$$Z^{(l)} = f^{(l)}\left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}Z^{(l-1)}W^{(l)}\right), \tag{2}$$

where $1 \leq l \leq k$, $Z^0 = X$, $f^{(l)}$ is the activation function of the $l$th layer, $W^{(l)}$ is the weight matrix of the $l$th layer, and $D^{-\frac{1}{2}}$ can be calculated by the following expression.

$$D_{ij}^{-\frac{1}{2}} = \begin{cases} D_{ii}^{-\frac{1}{2}}, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \tag{3}$$

The training algorithm adjusts the weight parameters of all layers. In GAE, VGAE, ARGE, and ARVGE, the two-layer GCN of the encoder uses the following ReLU activation function for the first layer.

$$f_{\text{Relu}}(t) = \max(0, t) \tag{4}$$

And the following linear function is used for the second layer.

$$f_{\text{linear}}(t) = t \tag{5}$$

We use $Z$ to represent the output of the encoder. In GAE and ARGE,

$$Z = Z^{(2)}, \tag{6}$$

while in VGAE and ARVGE, the output of the encoder is calculated by the following expression.

$$Z = Z^{(2)} + N_{n\times r}\left(0, \text{Exp}\left(Z^{(2)}\right)\right), \tag{7}$$

where $r$ is the number of units in hidden layer 2, $\text{Exp}(\cdot)$ element-wisely computes the natural exponential of the input matrix, and $N_{n\times r}(0,)$ returns a $n \times r$ matrix filled with random values with $(0, \cdot)$ normal distribution.

The input of the decoder is $Z$, which is the output of the encoder. Here, we focus on reconstructing the adjacency matrix, $A$. The reconstructed adjacency matrix is denoted by $\hat{A}$ and can be computed by the following formula.

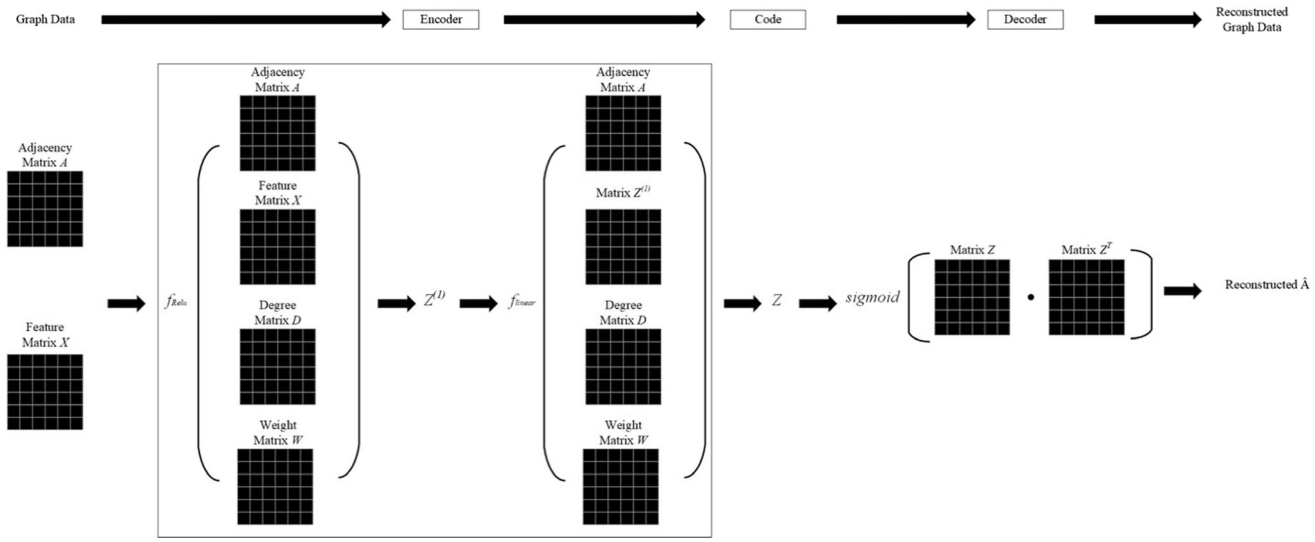


**Fig. 1** The upper part is the diagram of a general graph autoencoder. The lower part is the diagram of a graph autoencoder proposed by Kipf and Welling [8]

$$\hat{A} = sigmoid\left(ZZ^T\right), \tag{8}$$

where $Z^T$ is the transpose matrix of Z and the details of $sigmoid(\cdot)$ are given as follows.

$$sigmoid(t) = \frac{1}{1+e^{-t}} \tag{9}$$

## 3 Training strategy

This paper proposes a noise injection strategy based on the classic training framework for graph autoencoder. To reconstruct the adjacency matrix, the classic framework uses the input adjacency matrix as the desired output of decoder. Our strategy injects noise into the original adjacency matrix of a training sample and uses the noisy input to replace the original input and the desired output.

An element of an adjacency matrix is either one or zero. One indicates that the two corresponding nodes are connected, while zero suggests there is no connection between the two corresponding nodes. Therefore, the traditional noise injection method, which directly adds random noise to the input vector, does not fit the input data of graph autoencoder. DNGR [13], a denoising graph autoencoder, adds noise to the input by randomly changing some matrix elements from one to zero. This method decreases the number of edges. It may bring some concerns in sparse matrixes, which only contain a few edges. To overcome this potential issue, we developed a simple noise injection approach by randomly removing some existing edges and add the same number of new random edges. In comparison with DNGR, our approach can keep the number of edges stable. Edge removal is to change one into zero, while edge addition is to change zero into one. It is worth noting that we only focus on the edges between different nodes, so the elements of the diagonal of the adjacency matrix are ignored. The current version of our approach is designed for sparse adjacency matrixes, while it can also be adapted for dense adjacency matrix. We use $p$ and $u$ to represent the noise rate and the number of edges, respectively. After users specify $p$, which is between zero and one, our approach randomly removes $\lfloor pu \rfloor$ edges and also randomly adds $\lfloor pu \rfloor$ edges, where $\lfloor \cdot \rfloor$ is the floor function. The adjacency matrix of an undirected graph is symmetrical. Therefore, we first focus on the upper triangular matrix of the adjacency matrix in practice. This approach randomly changes $\lfloor pu \rfloor$ elements from one to zero and $\lfloor pu \rfloor$ elements from zero to one. After the upper triangular matrix is updated, our approach updates the lower triangular matrix symmetrically.

The noise injection process does not change the number of edges, but randomly shifts edges. In each training iteration, we randomly inject noise to the original training input and use the noisy input to replace the original input and the desired output, while the original training algorithm has no change. It allows this strategy to be flexibly applied to most existing training algorithms. Since the noise injection is conducted in each iteration, the original training input is not directly used in the whole training. The proposed strategy is summarized as follows.

Training Strategy (noise rate $p, 0 < p < 1$):
For each iteration:
    randomly remove edges with rate $p$;
    randomly add new edges with rate $p$;
    replace the original input by the noisy input
    replace the desired output by the noisy input
    use the original training algorithm with the noisy training data
End For

It is worth noting that our training strategy is different from that used in denoising autoencoders. After noise injection, our strategy uses the updated noisy input as desired output, while denoising autoencoders still use the original desired output, which has no noise.

## 4 Experiments and discussion

To verify the effectiveness of our strategy and investigate the suitable noise rates, we conduct experiments of link prediction on two data sets: Cora [30] and Citeseer [31]. The information of these two data sets is given in Table 1 [6, 8]. Both data sets were used for the experiments of [6, 8]. Our experiments test four autoencoders that are GAE, VGAE, ARGE, and ARVGE. Besides the original input, our experiments also test noisy input with noise rate 0.001, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, and 0.7 on both data sets. We follow the same experimental setting of [8]. The dimensions of the first and second layers are 32 and 16, respectively. In each data set, 5% connected node pairs and 5% non-connected node pairs are randomly picked for the validation set. Similarly, 10% connected node pairs and 10% non-connected node pairs are randomly picked for the test set. The rest is used for the training set. The training algorithm is Adam algorithm [32], while the learning rate is 0.01. The experiments apply area under the ROC curve (AUC) and average precision (AP) scores [33–35] to measure performance. As for each autoencoder, we repeatedly train ten times on both data sets. In order to easily reproduce training results, the random seed of the $i$th time is set to $i$. This paper reports average AUC and AP scores on the testing data sets. All autoencoders were trained 200 iterations in the experiments of [6, 8] on Cora and Citeseer. Because overfitting is more likely to happen with more iterations, the experiments report average AUC and AP scores with 200, 500, 1000, 2000, and 5000 iterations. As for other parameters, we use the default setting of each autoencoder.

**Table 1** The information of data sets used in experiments

| | Cora | Citeseer |
|---|---|---|
| The number of nodes | 2708 | 3327 |
| The number of edges | 5429 | 4732 |
| The number of features | 1433 | 3703 |

The experimental results on Cora and Citeseer are given in Figs. 2 and 3, respectively. It is worth noting that noise rate 0 refers to the original input without noise injection. Figure 2 shows that inputs with noise rates 0.05, 0.1, and 0.2 consistently outperform original input (noise rate 0) in AUC and AP scores with all four autoencoders, while other positive noise rates perform inconsistently. Figure 3 shows similar results. It suggests that noise rates 0.05, 0.1, and 0.2 consistently reduce overfitting in all tests. It also indicates noise rates less than 0.05 or greater than 0.2 may bring performance change, but not always performance improvement.

The impact of noise injection is clearly reflected in Figs. 2 and 3. Without noise injection (noise rate 0), the training performances with 200 iterations are consistently the best due to the overfitting problem. However, the noise injection strategy may achieve better performance with much more iterations by reducing the overfitting. For example, ARGE on both data sets can achieve better performance with iteration number 5000 and noise rate 0.2 than iteration number 200 and noise rate 0. It does not only encourage us to apply the proposed strategy in training, but also leaves an open question: which is the best noise rate? Although experimental results show 0.05, 0.1, and 0.2 can consistently improve performance, they are not always the best rate for a given graph autoencoder on a given data set. Tables 2 and 3 list the best noise rates for all categories. It is clear that 0.001 and 0.01 are not the best rates regardless of autoencoder and data set. It could be because they are too small for regularization. Furthermore, some rates greater than 0.2 are the best rates in some categories. It encourages us to develop new methods that can dynamically adjust the noise rate for gaining the best training performance in the future.

## 5 Conclusion and future work

This paper proposes a noise injection strategy for graph autoencoder training. This strategy can be flexibly applied in most existing graph autoencoders. The experimental results verify the effectiveness of this training strategy and suggest noise rates 0.05, 0.1, and 0.2 consistently improve training performance in all tests, although other higher rates may achieve better performance in some categories. In our future work, we will develop new methods with the feature of dynamically adjusting the noise rate during the training.

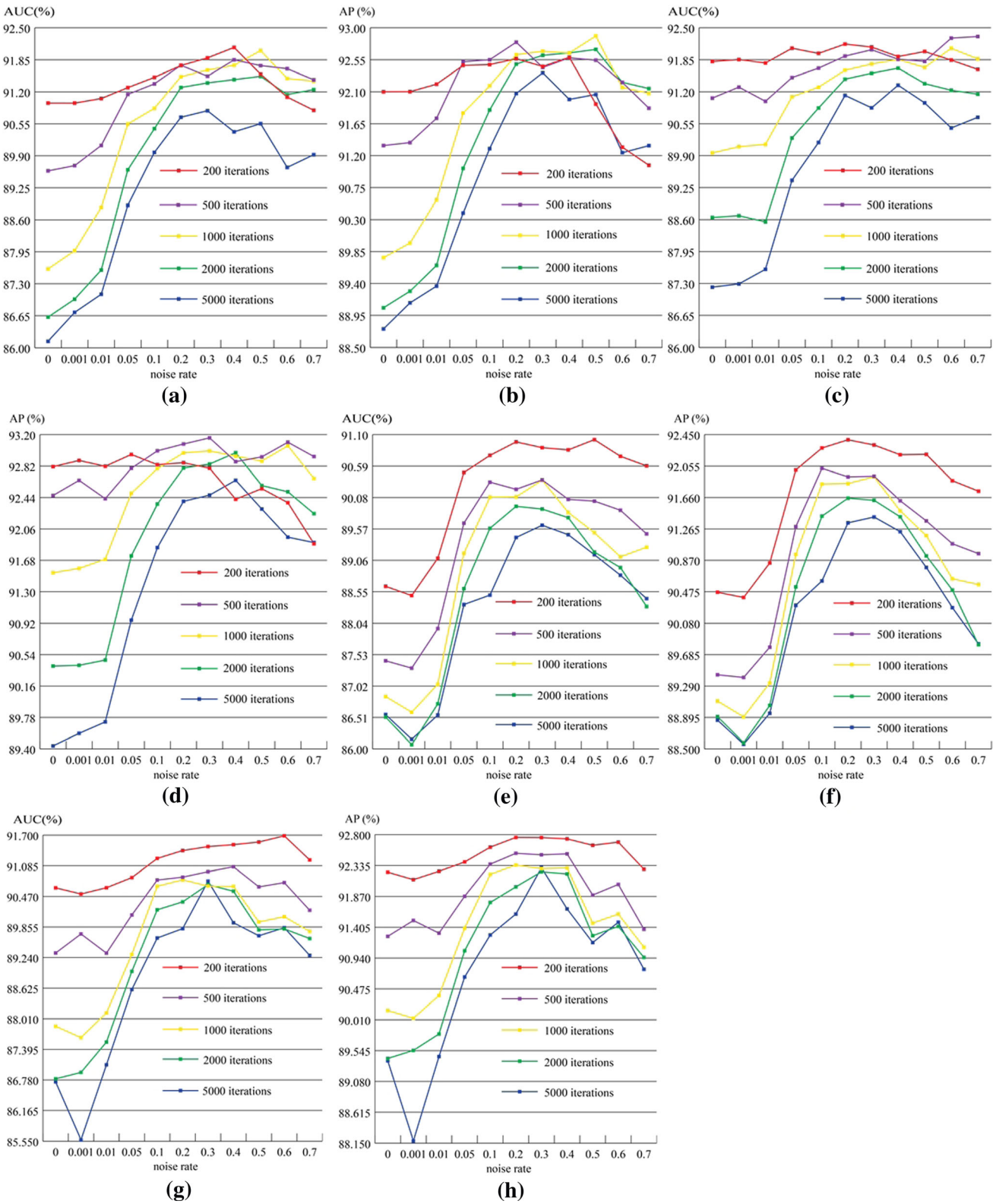


**Fig. 2** AUC and AP scores with 200, 500, 1000, 2000, and 5000 training iterations on Cora. Noise rate 0 refers to the original training input without noise injection. **a** GAE and AUC scores, **b** GAE and AP scores, **c** VGAE and AUC scores, **d** VGAE and AP scores, **e** ARGE and AUC scores, **f** ARGE and AP scores, **g** ARVGE and AUC scores, **h** ARVGE and AP scores

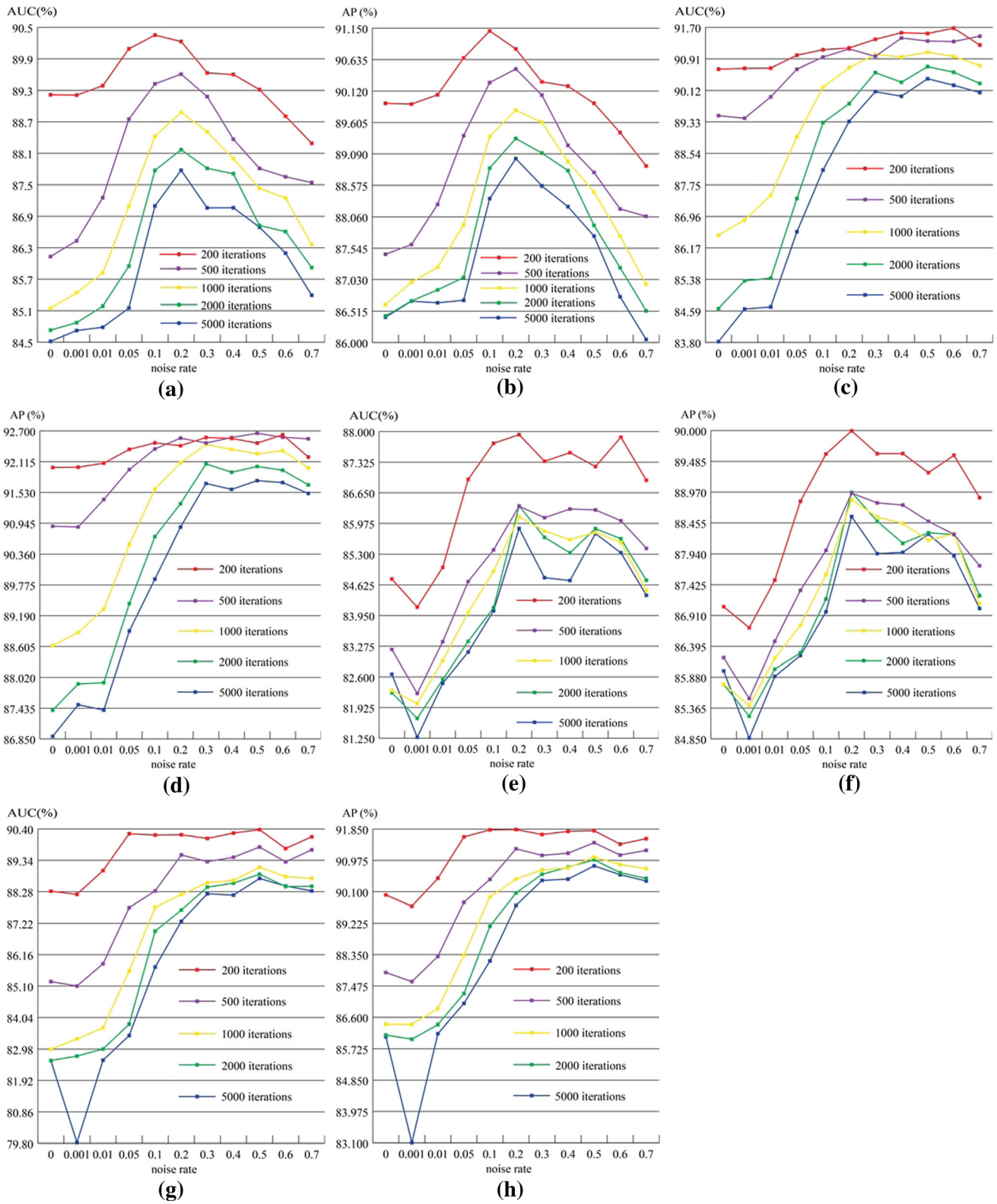


**Fig. 3** AUC and AP scores with 200, 500, 1000, 2000, and 5000 training iterations on Citeseer. Noise rate 0 refers to the original training input without noise injection. **a** GAE and AUC scores, **b** GAE and AP scores, **c** VGAE and AUC scores, **d** VGAE and AP scores, **e** ARGE and AUC scores, **f** ARGE and AP scores, **g** ARVGE and AUC scores, **h** ARVGE and AP scores

**Table 2** The best noise rates with different autoencoders and iteration times on Cora

| Autoencoder | GAE | | VGAE | |
|---|---|---|---|---|
| Measurement | AUC | AP | AUC | AP |
| Number of iterations | | | | |
| 200 | 0.4 | 0.4 | 0.2 | 0.05 |
| 500 | 0.4 | 0.2 | 0.7 | 0.3 |
| 1000 | 0.5 | 0.5 | 0.6 | 0.6 |
| 2000 | 0.5 | 0.5 | 0.4 | 0.4 |
| 5000 | 0.3 | 0.3 | 0.4 | 0.4 |
| Autoencoder | ARGE | | ARVGE | |
| Measurement | AUC | AP | AUC | AP |
| Number of iterations | | | | |
| 200 | 0.5 | 0.2 | 0.6 | 0.2 |
| 500 | 0.3 | 0.1 | 0.4 | 0.2 |
| 1000 | 0.3 | 0.3 | 0.2 | 0.2 |
| 2000 | 0.2 | 0.2 | 0.3 | 0.3 |
| 5000 | 0.3 | 0.3 | 0.3 | 0.3 |

**Table 3** The best noise rates with different autoencoders and iteration times on Citeseer

| Autoencoder | GAE | | VGAE | |
|---|---|---|---|---|
| Measurement | AUC | AP | AUC | AP |
| Number of iterations | | | | |
| 200 | 0.1 | 0.1 | 0.6 | 0.6 |
| 500 | 0.2 | 0.2 | 0.7 | 0.5 |
| 1000 | 0.2 | 0.2 | 0.5 | 0.3 |
| 2000 | 0.2 | 0.2 | 0.5 | 0.3 |
| 5000 | 0.2 | 0.2 | 0.5 | 0.5 |
| Autoencoder | ARGE | | ARVGE | |
| Measurement | AUC | AP | AUC | AP |
| Number of iterations | | | | |
| 200 | 0.2 | 0.2 | 0.5 | 0.2 |
| 500 | 0.2 | 0.2 | 0.5 | 0.5 |
| 1000 | 0.2 | 0.2 | 0.5 | 0.5 |
| 2000 | 0.2 | 0.2 | 0.5 | 0.5 |
| 5000 | 0.2 | 0.2 | 0.5 | 0.5 |

**Acknowledgements** This work was partially supported by the National Science Foundation under Grant Number 1813252.

## Compliance with ethical standards

**Conflict of interest** All authors declare that they have no conflict of interest.

## References

1. Wang Y, Xu B, Kwak M, Zeng X (2020) A simple training strategy for graph autoencoder. In: Proceedings of the international conference on machine learning and computing (ICMLC), pp 341–345. https://doi.org/10.1145/3383972.3383985
2. Tahmasebi H, Ravanmehr R, Mohamadrezaei R (2020) Social movie recommender system based on deep autoencoder network using Twitter data. Neural Comput Appl. https://doi.org/10.1007/s00521-020-05085-1
3. Cai H, Zheng VW, Chang KCC (2018) A comprehensive survey of graph embedding: Problems, techniques, and applications. IEEE Trans Knowl Data Eng 30:1616–1637. https://doi.org/10.1109/TKDE.2018.2807452
4. Li B, Pi D (2020) Network representation learning: a systematic literature review. Neural Comput Appl. https://doi.org/10.1007/s00521-020-04908-5
5. Pan S, Hu R, Fung SF et al (2020) Learning graph embedding with adversarial training methods. IEEE Trans Cybern 50:2475–2487. https://doi.org/10.1109/TCYB.2019.2932096
6. Pan S, Hu R, Long G, et al (2018) Adversarially regularized graph autoencoder for graph embedding. In: Proceedings of 27th international joint conference artificial intelligence, pp 2609–2615. https://doi.org/10.1523/JNEUROSCI.1317-08.2008
7. Zhang D, Yin J, Zhu X, Zhang C (2018) Network Representation Learning: A Survey. IEEE Trans Big Data. https://doi.org/10.1109/tbdata.2018.2850013
8. Kipf TN, Welling M (2016) Variational graph auto-encoders. In: NIPS workshop on bayesian deep learning
9. Wang D, Cui P, Zhu W (2016) Structural deep network embedding. In: Proceedings of the ACM SIGKDD international conference on knowledge discovery and data mining, pp 1225–1234
10. Tu K, Cui P, Wang X, et al (2018) Deep recursive network embedding with regular equivalence. In: Proceedings of ACM SIGKDD international conference knowledge discovery data min, pp 2357–2366. https://doi.org/10.1145/3219819.3220068
11. Samanta B, DE A, Jana G, et al (2019) NeVAE: A deep generative model for molecular graphs. In: Proceedings of the AAAI conference on artificial intelligence. pp 1110–1117
12. Grover A, Zweig A, Ermon S (2019) Graphite: Iterative generative modeling of graphs. In: Proceedings of machine learning research. pp 2434–2444
13. Cao S, Lu W, Xu Q (2016) Deep neural networks for learning graph representations. In: Proceedings of 30th AAAI conference on artificial intelligence, pp 1145–1152
14. Yu W, Zheng C, Cheng W, et al (2018) Learning deep network representations with adversarially. In: Proceedings of the international conference on knowledge discovery and data mining, pp 2663–2671
15. Goodfellow I, Pouget-Abadie J, Mirza M et al (2014) Generative adversarial nets. Adv Neural Inf Process Syst 27:2672–2680
16. Elman JL, Zipser D (1988) Learning the hidden structure of speech. J Acoust Soc Am 83:1615–1626. https://doi.org/10.1121/1.395916
17. Sietsma J, Dow RJF (1991) Creating artificial neural networks that generalize. Neural Netw 4:67–79. https://doi.org/10.1016/0893-6080(91)90033-2
18. Holmstrom L, Koistinen P (1992) Using additive noise in back propagation training. IEEE Trans Neural Netw 3:24–38. https://doi.org/10.1109/72.105415
19. Skurichina M, Raudys Š, Duin RPW (2000) K-nearest neighbors directed noise injection in multilayer perceptron training. IEEE Trans Neural Netw 11:504–511. https://doi.org/10.1109/72.839019

20. Brown WM, Gedeon TD, Groves DI (2003) Use of noise to augment training data: a neural network method of mineral-potential mapping in regions of limited known deposit examples. Nat Resour Res 12:141–152. https://doi.org/10.1023/A:1024218913435
21. Matsuoka K (1992) Noise injection into inputs in back-propagation learning. IEEE Trans Syst Man Cybern 22:436–440. https://doi.org/10.1109/21.370200
22. Reed R, Marks RJ, Oh S (1995) Similarities of error regularization, sigmoid gain scaling, target smoothing, and training with jitter. IEEE Trans Neural Netw 6:529–538. https://doi.org/10.1109/72.377960
23. Bishop CM (1995) Training with noise is equivalent to Tikhonov regularization. Neural Comput 7:108–116. https://doi.org/10.1162/neco.1995.7.1.108
24. Grandvalet Y, Canu S, Boucheron S (1997) Noise injection: theoretical prospects. Neural Comput 9:1093–1108. https://doi.org/10.1162/neco.1997.9.5.1093
25. An G (1996) The Effects of adding noise during backpropagation training on a generalization performance. Neural Comput 8:643–674. https://doi.org/10.1162/neco.1996.8.3.643
26. Piotrowski AP, Napiorkowski JJ (2013) A comparison of methods to avoid overfitting in neural networks training in the case of catchment runoff modelling. J Hydrol 476:97–111. https://doi.org/10.1016/j.jhydrol.2012.10.019
27. Wright WA (1999) Bayesian approach to neural-network modeling with input uncertainty. IEEE Trans Neural Netw 10:1261–1270. https://doi.org/10.1109/72.809073
28. Wright WA, Ramage G, Cornford D, Nabney IT (2000) Neural network modelling with input uncertainty: theory and application. J VLSI Signal Process Syst Signal Image Video Technol 26:169–188. https://doi.org/10.1023/A:1008111920791
29. Zhang S, Tong H, Xu J, Maciejewski R (2019) Graph convolutional networks: a comprehensive review. Comput Soc Netw 6:1–23. https://doi.org/10.1186/s40649-019-0069-y
30. McDowell LK, Gupta KM, Aha DW (2009) Cautious collective classification. J Mach Learn Res 10:2777–2836
31. Giles CL, Bollacker KD, Lawrence S (1998) CiteSeer: an automatic citation indexing system. In: Proceedings of ACM international conference digital library, pp 89–98
32. Kingma DP, Ba JL (2015) Adam: a method for stochastic optimization. In: Proceedings of the 3rd international conference on learning representations
33. Fawcett T (2006) An introduction to ROC analysis. Pattern Recognit Lett 27:861–874. https://doi.org/10.1016/j.patrec.2005.10.010
34. McClish DK (1989) Analyzing a portion of the ROC curve. Med Decis Mak 9:190–195. https://doi.org/10.1177/0272989X8900900307
35. Wikipedia entry for the Receiver operating characteristic. https://en.wikipedia.org/wiki/Receiver_operating_characteristic. Accessed 9 Jan 2019