Machine Learning and Algorithms for Data Mining

Assessment 2*

Analysis of graph-structured data

Sebastian Borgeaud dit Avocat

February 22, 2018

Abstract

In this report I explore different algorithms that solve the task of transductive classification on a graph and report their accuracy on the Cora dataset [1]. First, I present a simple baseline method based on support vector machines that only uses the features available at individual nodes. Next, I propose two methods that incorporate the features of neighbouring nodes by averaging over them. As expected, incorporating knowledge about the graph structure gives a significant improvement in classification accuracy, even when incorporated in a simple way. Finally, I reimplement two state-of-the-art methods: Graph Convolutional Networks [2], and Graph Attention Networks [3] which achieve the highest accuracy amongst the tested methods.

1 Main contributions, Key Concepts and Ideas

1.1 Introduction

Many real world datasets occur naturally in the form of graphs: examples from different fields include protein-protein interaction networks in biology [4], social networks in sociology [5], and scientific collaboration networks [6]. I focus on the problem of node classification for citation networks in a transductive setting and present multiple approaches to solve this problem.

*Word count: 2467 — Computed using TexCount

In the transductive case the entire structure of the network is known at training time but only some of the nodes are labelled. At test time, the task is to to infer the labels of the remaining nodes. This differs from supervised learning in two ways. i) The data points are connected to each other through edges. Using these relations might be helpful for classification. ii) As the entire graph is known at test time, the features of the nodes that will be classified at test time are known in advance.

1.2 Related work

It is generally possible to distinguish between two categories of approaches for graph-based semi-supervised learning: methods based on an explicit Laplacian regularisation cost and methods based on graph-embedding approaches.

In the first approach, the problem is seen as a graph-based semi-supervised learning problem, where the label information is smoothed over the graph via some form of explicit graph-based regularisation. Examples of such approaches include the label propagation algorithm (Zhu et. al [7]) and manifold regularisation (Belkin et al. [8]).

In the second approach, the problem consists of learning embeddings for the nodes using the local graph structure. The embeddings are then used in a semi-supervised setting to learn the corresponding class labels. For example, DeepWalk (Perozzi et al. [9]) learns embeddings using sampled random walks on the graph. This method is extended with more sophisticated random walk search schemes in LINE (Tang et al. [10]) and node2vec (Grover & Leskovec [11]).

1.3 Cora Dataset

To evaluate the different models I focus on the Cora dataset[1] [1]. The dataset consists of N = 2708 nodes representing machine learning publications. Each publication is classified into one of C = 7 classes: 'Case Based', 'Genetic Algorithms', 'Neural Networks', 'Probabilistic Methods', 'Reinforcement Learning', 'Rule Learning' or 'Theory'. The features associated to each node are a binary bag-of-word feature vector for a vocabulary of F = 1433 unique words. Let $\mathbf{X}$ be the $N \times F$ matrix containing the features of the nodes in its

[1] `https://linqs.soe.ucsc.edu/data`

rows and let $\mathbf{A}$ be the adjacency matrix of the graph. The i^{th} row of $\mathbf{X}$, $\mathbf{X}_i$, contains the bag-of-word binary vector for node i. The edges are directed and represent citations, where an edge $pub_1 \rightarrow pub_2$ means that publication pub_2 is cited in publication pub_1. The papers were selected in such a way that every paper cites or is cited by at least one other paper.

In particular, I use the data split introduced by Kipf and Welling [2]: 140 training nodes (about 6.7% of the nodes), 500 validation nodes and 1000 test nodes. Let $\mathbf{X}_L$ and $\mathbf{y}_L$ be the matrices containing the features and the labels of the labelled nodes. Furthermore, the edge orientations are ignored.

1.3.1 Node degree distribution

The distribution of node degrees is plotted in figures 1 and 2. Most nodes have few edges, with 59.9% of the nodes having 3 edges or less and over 96.5% having 10 or less. There seems to be one outlier publication with 168 edges, whereas the second most citing paper has only 78 edges.

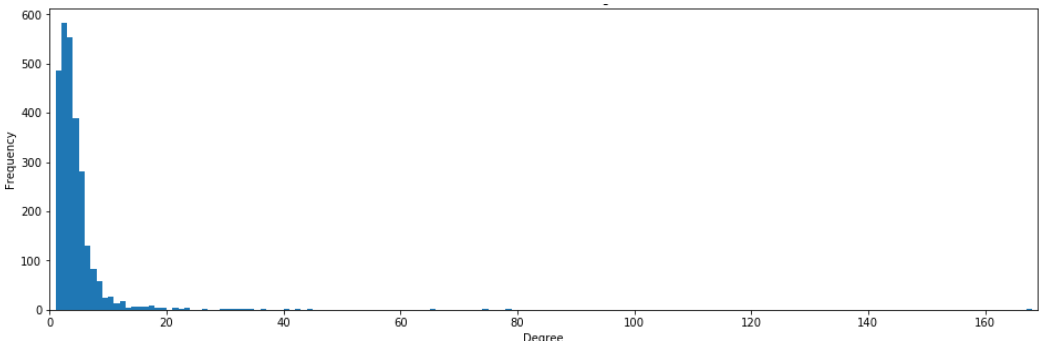

Figure 1: Distribution of the degree of the nodes in the Cora dataset

1.3.2 Clusters

Furthermore, it is interesting to explore how densely clustered the network is. Using a simple breath-first search algorithm, I found that there are a total of 78 clusters. However, most of the nodes are in the largest cluster which contains 2485 nodes, that is about 91.7% of the nodes. The full distribution of cluster sizes is shown in table 1.

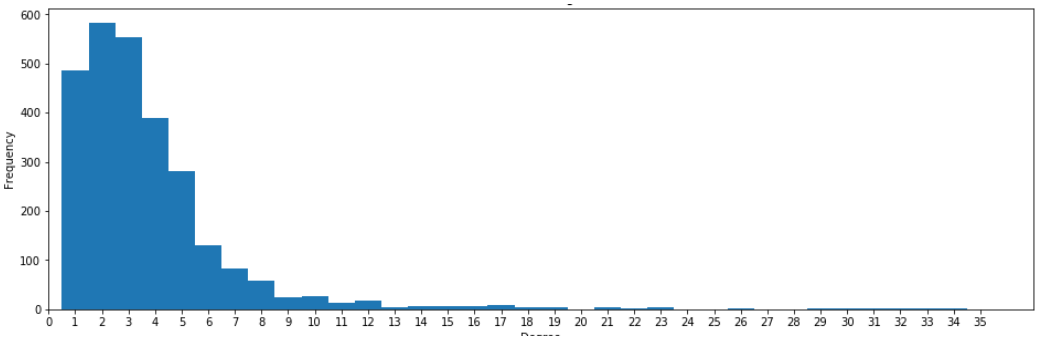

Figure 2: Distribution of the degrees of the nodes with degree below 40. This includes 2701 of the 2708 publications, or above 99.7% of the nodes.

Cluster size	2	3	4	5	6	8	9	26	2485
Frequency	57	7	6	3	1	1	1	1	1

Table 1: Distribution of cluster sizes in the Cora dataset.

2 Methods

In this section I present the algorithms I implement to solve the classification task.

2.1 SVM on node features (node SVM)

The first method provides a simple baseline model and doesn't incorporate any information about the structure of the network. The problem reduces to a supervised learning task, which I solve with a support vector machine (SVM) trained on the features of the nodes.

Using scikit-learn[2] [12] I train a SVM on the labelled training nodes. The trained model is then used to predict the class labels of the test publications.

2.2 SVM with neighbour features (average SVM)

For the second approach, I incorporate information about the graph structure into the SVM model. To do this I combine the neighbours of a node by taking

[2] `http://scikit-learn.org/`

the average of their features and concatenating the resulting vector to the feature vector of the node itself. That is, we train the SVM on the features $\mathbf{X}'$ where for each node i,

$$\mathbf{X}'_i = \left[\mathbf{X}_i \middle|\middle| \mathbf{X}_i^{neighbours}\right]$$

where

$$\mathbf{X}_i^{neighbours} = \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} \mathbf{X}_j,$$

$\mathcal{N}(i)$ is the set containing the neighbour nodes of node i, and $[\cdot||\cdot]$ denotes vector concatenation. Note that $\mathbf{X}'$ is a N $\times$ 2F matrix and thus contains twice as many entries as $\mathbf{X}$.

2.3 Multilayer perceptron (average MLP)

Next, I train a simple multilayer perceptron using the concatenated features $\mathbf{X}'$. The network consists of two fully-connected layers. The first layer consists of 64 hidden units and has a ReLU activation function. The second layer consists of $C = 7$ output units, to which is applied the softmax activation to obtain a probability distribution over the 7 classes. Furthermore, as there are only few training nodes, I make heavy use of two regularisation techniques: i) Dropout [13] applied to the input layer and to the hidden layer, and ii) L2 regularisation on the weights of the first hidden layer.

2.4 Graph Convolutional Network (GCN)

The fourth algorithm I implement is the Graph Convolutional Network, presented by Kipf and Welling [2]. Here, the graph structure is encoded directly using a neural network $f(\mathbf{X}, \mathbf{A})$ where $\mathbf{X}$ is the usual feature matrix and $\mathbf{A}$ is the adjacency matrix. The model is trained on a single supervised loss $\mathcal{L}_0$. However, as the adjacency matrix is also inputed to the network, the gradient information is distributed back from $\mathcal{L}_0$ to all nodes and therefore representations are learnt for nodes with and without labels [2].

A Graph Convolutional Network (GCN) is a neural network with the following layer-wise propagation rule, where the activations in each layer $\mathbf{H}^{(l)}$ are computed as

$$\mathbf{H}^{(l+1)} = \sigma\big(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\big)$$

Here, $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_\mathrm{N}$ is the adjacency matrix with added self-connections and $\tilde{\mathbf{D}}$ is a diagonal matrix with $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. $\mathbf{W}^{(l)}$ are the trainable neural network weights in layer l. Finally, $\sigma(\cdot)$ is an activation function.

For the Cora dataset, the authors present a 2 layer GCN with a ReLU activation in the first layer and a softmax activation in the second layer. The computation of the forward pass of the neural network can therefore be described concisely as

$$f(\mathbf{X}, \mathbf{A}) = \mathrm{softmax}\big(\hat{\mathbf{A}}\ \mathrm{ReLU}\big(\hat{\mathbf{A}} X \mathbf{W}^{(0)}\big)\ \mathbf{W}^{(1)}\big)$$

where $\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}}$. $\mathbf{W}^{(0)}$ is an N $\times$ H weight matrix, where H is the number of nodes in the hidden layer and $\mathbf{W}^{(1)}$ is an H $\times$ C weight matrix, where C = 7 is the number of classes. The ReLU activation function just clips the input to 0,

$$\mathrm{ReLU}(x) = \max(0, x)$$

and the softmax activation computes for each component x_i

$$\mathrm{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

During training the cross-entropy loss

$$\mathcal{L} = -\sum_{l \in \mathbf{y}_L} \sum_{c=1}^{C} \mathbb{1}(\mathbf{y}_L^{(l)} = c) \ln f(X, A)_{lc}$$

is minimised over all labelled examples $\mathbf{y}_L$ using gradient descent. Dropout [13] is applied to the input features and to the output of the first layer to reduce overfitting.

2.5 Graph Attention Networks

The final method I explore is the Graph Attention Network (GAT) proposed by Veličković et al. [3] based on an attention mechanism. Attention mechanisms give a neural network the ability to learn on which part of the input to focus. More specifically, the GAT uses self-attention: the attention mechanism is used to compute a representation of a single sequence. The main idea behind the algorithm is to compute a hidden representation (also called

an embedding) for each node in the graph by attending over all its neighbours using self-attention. That is, the network computes an embedding for each node using the nodes in its neighbourhood and by choosing how much attention to pay to each individual neighbour using the self-attention mechanism.

The GAT architecture is best described in terms of its layers. In each layer, the input features $\mathbf{X}$ are first linearly transformed into $\mathbf{X}' = \mathbf{X}\mathbf{W}$ where W is a $\mathrm{F} \times \mathrm{F}'$ weight matrix. Thus, each node feature vector $\mathbf{h}_i = \mathbf{X}_i$ is linearly transformed to a feature vector $\mathbf{h}'_i = \mathbf{X}'_i = \mathbf{h}_i\mathbf{W}$ of length F'. Then, self-attention is performed on the nodes by applying a shared attentional mechanism defined as a function $a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R}$. This function maps two transformed input feature vectors to a real attention coefficient:

$$e_{ij} = a(\mathbf{h}'_i, \mathbf{h}'_j) = a(\mathbf{h}_i\mathbf{W}, \mathbf{h}_j\mathbf{W})$$

Intuitively, e_{ij} indicates how important the features of node j are to node i. To inject graph structural information, we attend over the direct neighbours of i, including i itself. Hence, we have that

$$e_{ij} = \begin{cases} a(\mathbf{h}_i\mathbf{W}, \mathbf{h}_j\mathbf{W}) & \text{if } A_{ij} = 1 \text{ or } i = j \\ 0 & \text{otherwise} \end{cases}$$

Then, the coefficients are normalised using the softmax function:

$$\alpha_{ij} = \mathrm{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik})}$$

This is done for every pair of nodes i and j to obtain an $\mathrm{N} \times \mathrm{N}$ matrix α_{ij} of attention coefficients.

The attention mechanism is defined to be a single-layer neural network, parametrised by weights $\mathbf{a}$, and using a LeakyReLU nonlinearity:

$$\mathrm{LeakyReLU}_\alpha(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases}$$

The attention coefficients can therefore be described concisely as

$$\alpha_{ij} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^T\left[\mathbf{h}_i\mathbf{W} || \mathbf{h}_j\mathbf{W}\right]\right)\right)}{\sum_{k \in \mathcal{N}(i)} \exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^T\left[\mathbf{h}_i\mathbf{W} || \mathbf{h}_k\mathbf{W}\right]\right)\right)}$$

Next, we use the attention coefficients to compute a linear combination of the transformed features and apply a nonlinearity. This gives the final output of the layer

$$g_i = \sigma\big(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} \mathbf{h}_j \mathbf{W}\big)$$

Furthermore, we can apply multi-head attention to stabilise the learning process. Instead of using a single attention mechanism, we use K_a independent attention heads, concatenating their output:

$$g_i = \Big|\Big|_{k=1}^{K_a} \sigma\big(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^k \mathbf{h}_j \mathbf{W}^k\big)$$

Note that in this case the output feature vectors will have size $K_a F'$.

Veličković et al. propose a GAT network consisting of two graph attention layers. The first layer has $K_a = 8$ parallel attention heads and computes a hidden representation of features of size $F' = 8$. The nonlinearity used is the exponential linear unit ELU:

$$\mathrm{ELU}_\alpha(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\exp(x) - 1) & \text{otherwise} \end{cases}$$

The second layer has C features, a single activation head $K_a = 1$, and uses a softmax activation.

L2 regularisation is added and Dropout [13] is applied to the layers' inputs and to the attention coefficients. Thus at each training epoch, a node is only exposed to a sample of its neighbourhood.

3 Results & analysis

The results obtained with the various methods described above are shown in table 2.

Next, I go through the implementation details for each of the methods presented and explain how the hyper-parameters are optimised.

Method	Accuracy	
	validation	test
linear node SVM	0.558	0.583
RBF node SVM	0.566	0.576
linear average SVM	0.714	0.728
RBF average SVM	0.716	0.733
average MLP	0.762	0.769
GCN	0.785	0.802
GAT	0.769	0.794

Table 2: Test and validation accuracy obtained using the different models.

3.1 SVM on node features

Using the 500 validation nodes I optimise the hyper-parameters of the SVM by using a grid-search approach over a range of parameters. I consider both linear and a radial basis function (RBF) kernels. For the linear kernel C is taken at regular intervals on a logarithmic scale ranging from 2^{-15} to 2^3. For the RBF kernel, C is taken from a logarithmic scale ranging from 2^{-15} to 2^5 and γ is taken from the range 2^{-15} to 2^3. The validation accuracy obtained for these hyper-parameters ranges are presented in figures 3 and 4. The validation accuracy is maximised at 56.6%, obtained with the RBF kernel, $C = 1.86$ and $\gamma = 2^{-7}$.

Although significantly below the current state-of-the-art accuracy, this provides a simple baseline accuracy on the test nodes of 57.6%. This is significantly better than a random baseline, which would have an expected accuracy of $\frac{1}{7}$ (14.3%). This suggests that, rather unsurprisingly, a lot of the information required for classification is contained in the bag-of-word vectors.

3.2 Average features SVM

I train the second SVM this time using the concatenated features $\mathbf{X}'$. I consider both linear and a RBF kernels. The validation accuracies are presented

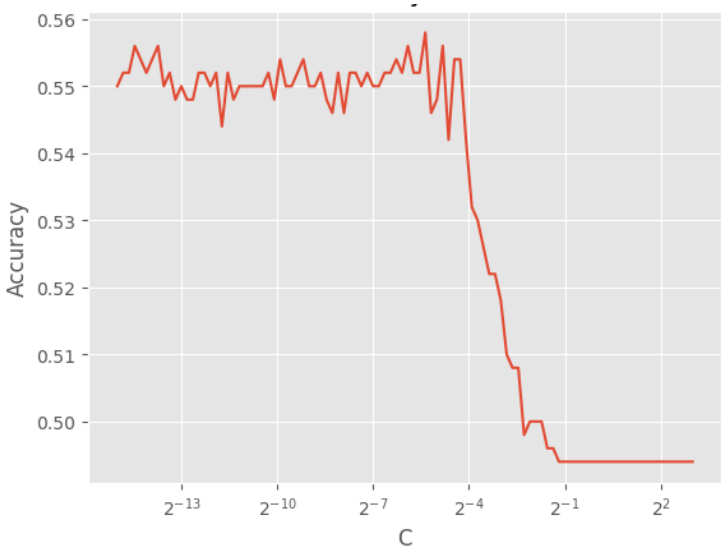

Figure 3: Validation accuracy obtained for various values of C for the SVM with a linear kernel trained on the node features alone. The best validation accuracy is obtain at $C = 0.024$, with a validation accuracy of 55.8%.

in figures 5 and 6. Again, the RBF kernel performs slightly better with a validation accuracy of 71.6% compared to 71.4% for the linear kernel. Using the best hyper-parameters with the RBF kernel: $C = 5.411$ and $\gamma = 2^{-9}$, I obtain an accuracy of 73.3% on the test nodes.

This is a significant improvement over the accuracy obtain with the SVM trained on the features of each node individually. This shows that incorporating knowledge about the graph structure does indeed improve the classification accuracy significantly even when incorporated only in a very simplistic way, i.e. by only considering the direct neighbours and averaging their features.

3.3 Multilayer perceptron

To find the optimal hyper-parameters, I again use a grid search method, validating the accuracy using the 500 validation nodes. The network is trained using every possible combination of parameters with `batch_size` in [32, 64, 128, 140], `num_epochs` in [50, 100, 200, 400], `dropout_p` in [0.0, 0.1, 0.2, 0.4, 0.6] and λ in [0.05, 0.005, 0.001, 0.0001] where λ is the L2 regularisation

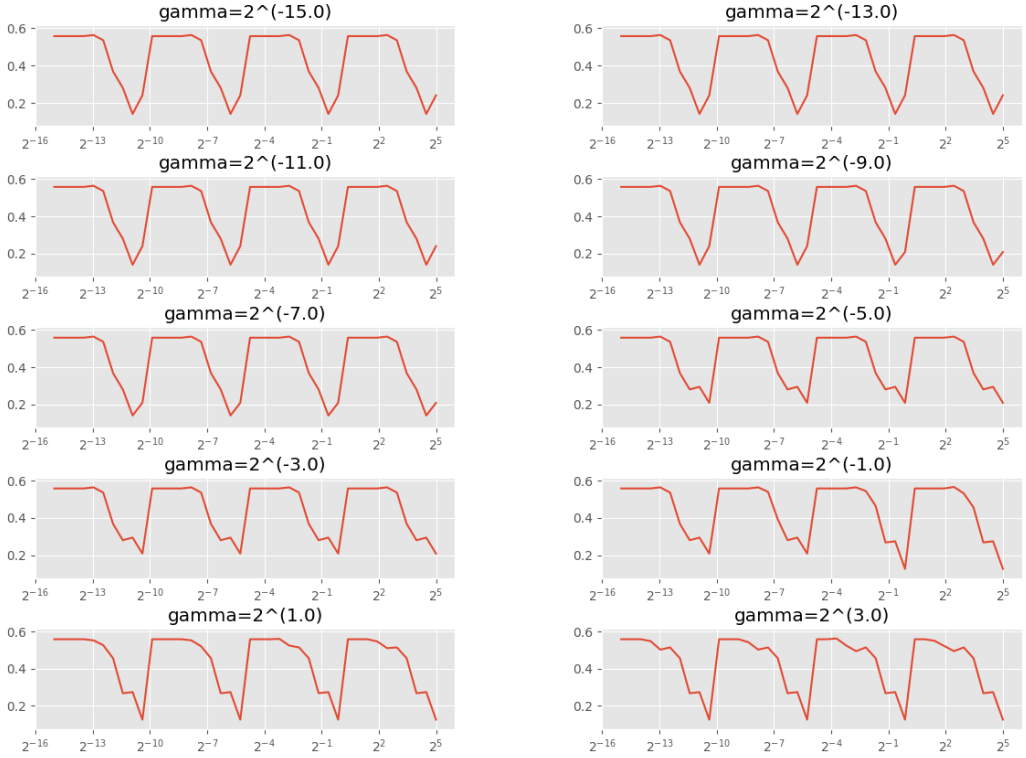

Figure 4: Validation accuracy obtained for various values of C and γ for the SVM with an RBF kernel trained on the node features alone. The best validation accuracy is obtain at $C = 1.86$ and $\gamma = 2^{-7}$, with a validation accuracy of 55.6%.

weight. The best validation accuracy of 76.2% was obtained using the entire training set in a single batch, trained over 400 epochs with a dropout rate of 0.6 and setting $\lambda = 0.0001$. Using these hyper-parameters I obtain a test accuracy of 76.9%. Figure 7 shows the loss and accuracy at training time.

The multilayer perceptron therefore performs better than the SVM trained on the same input features. This could be explained by the higher complexity of the MLP compared to the SVM.

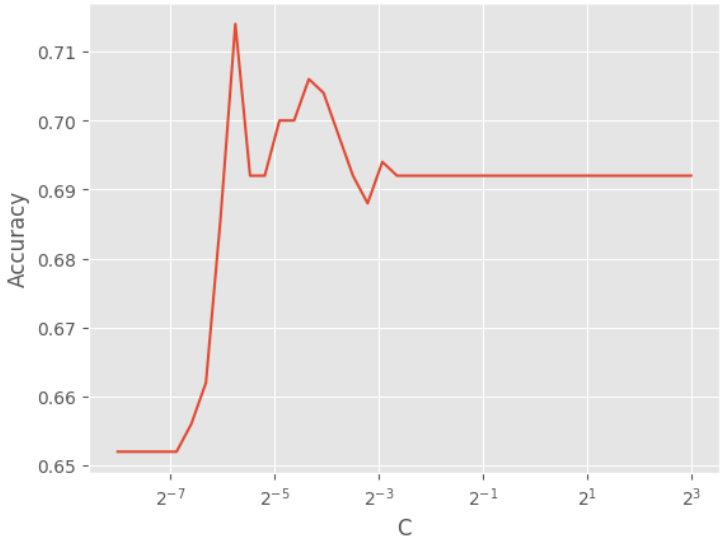

Figure 5: Validation accuracy obtained for various values of C for the SVM with a linear kernel trained on the node features and the average neighbouring features. The best validation accuracy is obtain at $C = 0.018$, with a validation accuracy of 7.14%.

3.4 Graph Convolutional Network

I implement the GCN model using TensorFlow with the optimised hyperparameters presented by the authors. The model is trained over 200 epochs using the Adam optimiser [14]. The optimal learning rate was found to be 0.01. The weights are initialised using a Glorot initialization [15]. Furthermore, the input feature vectors are row-normalized. The dropout rate was set to $p = 0.5$, the L2 regularisation weight to $\lambda = 0.0005$. The number of hidden inputs was set to 16. Averaging the result of 10 runs, I obtain a validation accuracy of 78.5% and a test accuracy of 80.3%.

The GCN gives a significant improvement over the MLP. This could be accounted for by multiple reasons. First, the graph structure is incorporated in a more principled way, using the adjacency matrix and the degree matrix of the network. Second, as the network consists of two layers, it has the ability to incorporate information not only from the direct neighbours of a node but also from the neighbours of the neighbours of a node.

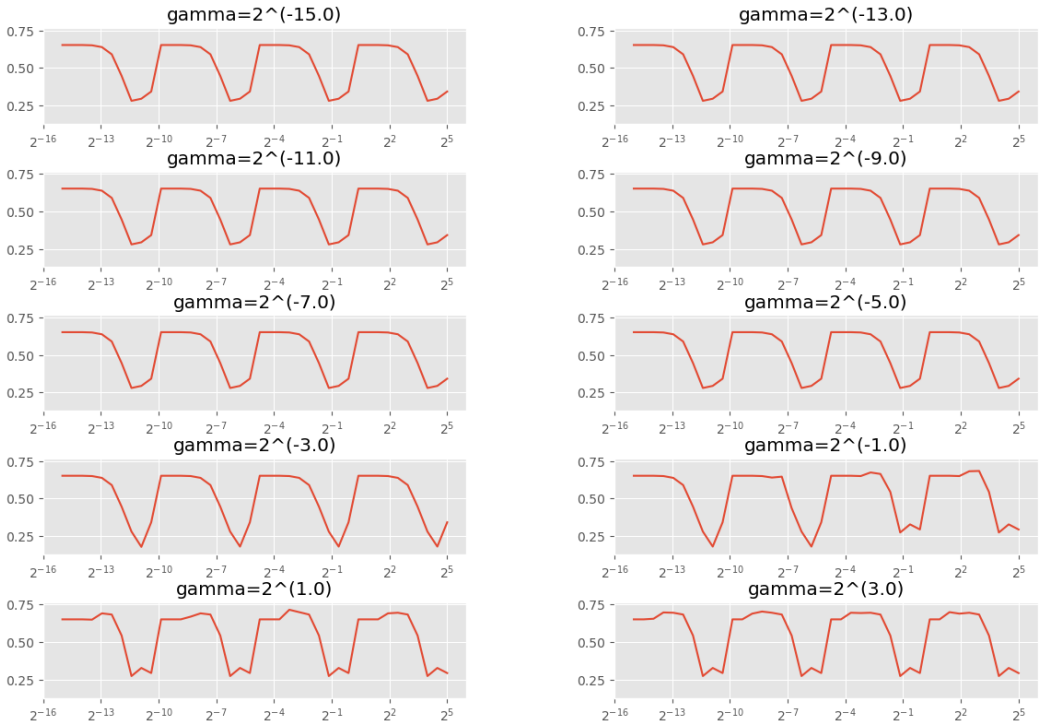

Figure 6: Validation accuracy obtained for various values of C and γ for the SVM with an RBF kernel trained on the node features and the averaged neighbouring features. The best validation accuracy is obtain at $C5.411$ and $\gamma = 2^{-9}$, with a validation accuracy of 71.6%.

3.4.1 Hidden layer visualisation

The activations in the hidden layer of the GCN can be visualised using a dimensionality reduction technique. Following the approach taken by Kipf and Welling, I use t-SNE [16] to reduce the 16-dimensional hidden activation to 2 dimensions. Figure 8 shows the resulting vectors.

3.5 Graph Attention Network

Similarly as for the GCN, I use the hyper-parameters presented by the author but reimplement the network in TensorFlow. The regularisation weight is set to $\lambda = 0.0005$ and the dropout rate is set to $p = 0.6$. The Adam optimiser [14] is used with a learning rate of $lr = 0.005$. The model is trained over 100

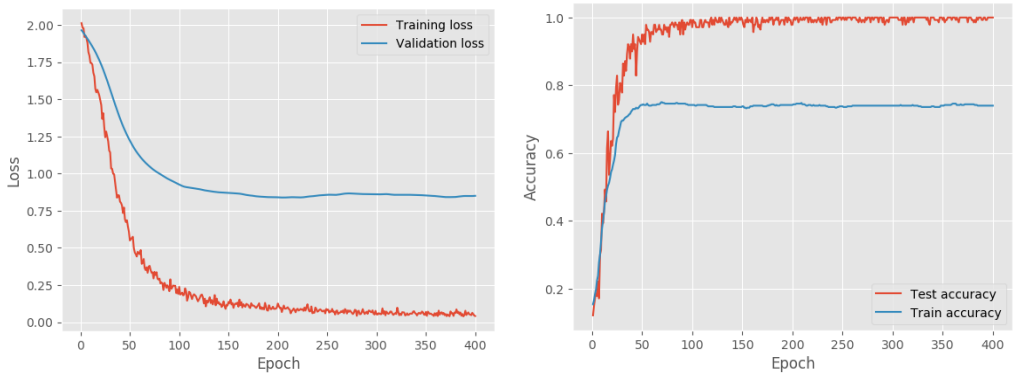

Figure 7: Accuracy and loss obtained during training of the MLP using the optimised hyper-parameters.

epochs, each with a single batch containing all 2708 nodes.

I obtain a validation accuracy of 0.77% and a test accuracy of 0.79%, which is below the accuracy reported by the authors (83.0%). This discrepancy could be explained by small differences between the two implementations of the algorithm.

3.5.1 Learnt embeddings visualisation

Using t-SNE we can reduce the dimensionality of the learnt embeddings, i.e. the output of the first layer of the GAT. Reducing to 2 dimensions I plot the embeddings in a plane, as shown in figure 9.

4 Conclusion

I implemented 5 different approaches to solve the task of transductive classification on the Cora dataset [1]. Experiments show that a simple baseline method using the node features alone is sufficient to obtain significantly better results than using a random baseline. Using information given by the graph structure by concatenating the average of the neighbouring node features to the features of a node itself did increase the accuracy significantly. My implementation of the more sophisticated approaches, i.e. the GCN and the GAT obtained the best results, closely matching the current state-of-the-art results for this dataset.

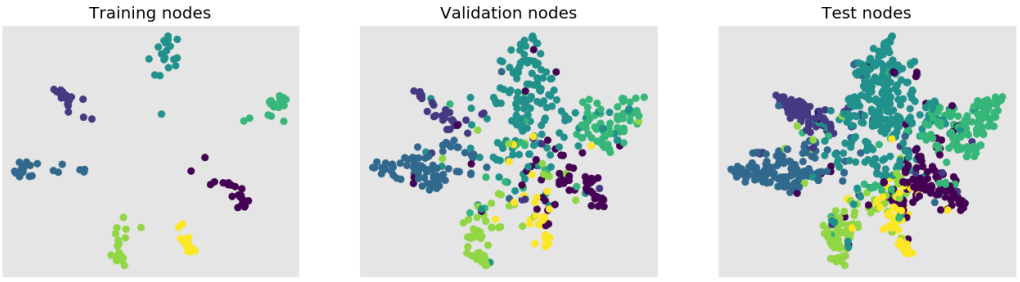

Figure 8: Activations in the hidden layer, after reducing their dimensionality using t-SNE for the training, validation and test nodes. As expected, the network is able to learn to discern the training examples well. Different colours represent the different classes.

References

[1] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93, 2008.

[2] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[3] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. accepted as poster.

[4] Benno Schwikowski, Peter Uetz, and Stanley Fields. A network of protein–protein interactions in yeast. *Nature biotechnology*, 18(12):1257, 2000.

[5] Evelien Otte and Ronald Rousseau. Social network analysis: a powerful strategy, also for the information sciences. *Journal of information Science*, 28(6):441–453, 2002.

[6] Mark EJ Newman. The structure of scientific collaboration networks. *Proceedings of the national academy of sciences*, 98(2):404–409, 2001.

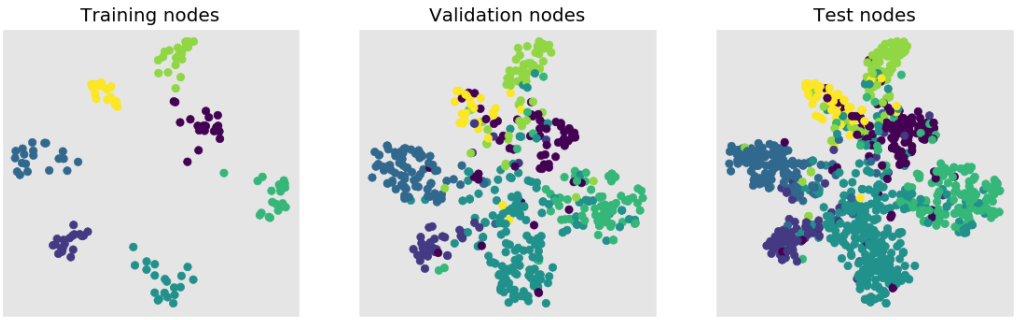

Figure 9: Visualisation of the learnt embeddings by the GAT. The dimensionality of the embeddings is reduced from 64 to 2 using t-SNE. The embeddings are shown for the training, validation and test nodes. As expected, the network is able to learn to discern the training examples well. Different colours represent the different classes.

[7] Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. 2002.

[8] Mikhail Belkin, Partha Niyogi, and Vikas Sindhwani. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *Journal of machine learning research*, 7(Nov):2399–2434, 2006.

[9] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[10] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1067–1077. International World Wide Web Conferences Steering Committee, 2015.

[11] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[13] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[16] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.