Inductive Node Classification with Attention

Tianqi Zhao,Wei Zhang,Tyng-Jiun Kuo

July 2019

1 Abstract

Graph attention networks with masked self-attentional layers are novel network architectures operate on graph-structured data. They are capable of specifying different weights to different nodes in a neighborhood in stacking layers without any kind of costly matrix operation. Experiments have confirmed that GAT models have achieved or matched state-of-the-art results across four established transductive and inductive graph benchmarks:the Cora, Citeseer and Pubmed citation network datasets, as well as a protein-protein interaction dataset.

2 Task Description

Nowadays, Convolutional Neural Networks (CNNs) have been widely used to tackle problems which has a grid-like structure, such as image classification, or machine translation. These architectures reuse the local filters, applying them and learnable parameter to all the input positions.
However, for task in social networks, telecommunication networks or brain connections, these tasks involve data that can not be represented in a grid-like structure. In contrast, they can usually be represented by graphs. Therefore, there is an increasing interest in generalizing convolutions to the graph domain. More recently, Hamilton et al. (2017) introduced GraphSAGE to compute node representations in an inductive manner. The approach is, first sampling a fixed-size neighborhood of each node, and then performing a specific aggregator over it.
Since then, attention mechanisms have become popular in sequence-based tasks. For a single sequence, it is usually referred to self-attention or intra-attention. However, self-attention can not only improve a method based on RNNs or convolutions, but also sufficient for constructing a powerful model obtaining state-of-the-art performance on the machine translation task.
According to this concept. The author introduced an attention-based architecture to perform node classification of graph-structured data. The idea is to compute the hidden representations of each node in the graph, by attending over its neighbors, following a self-attention strategy. This architecture has several

properties, first it is efficient, second it can be applied to graph node for any degrees, third the model can be directly used to inductive learning problems. In this model, relations between objects are aggregated pair-wise, by employing a shared mechanism, and then use a neighborhood attention operation to compute attention coefficients between different objects in an environment. The author used LLE to select neighbors around each data point, and learns a weight coefficient for each neighbor to reconstruct each point as a weighted sum of its neighbors. Also, the author interpreted the neighborhood of a node as the memory, which is used to compute the node features by attending over its values, and then is updated by storing the new features in the same position.

3 Approach

3 Approaches to create the Graph attentional layer

For a sigle graph attentional layer, the input is a set of node features $\mathbf{h} = \left\{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\right\}$, each h in this set is a vector itself,which has F rows. The output from this layer is also a set of node features: $\mathbf{h}' = \left\{\vec{h}'_1, \vec{h}'_2, \ldots, \vec{h}'_N\right\}, \vec{h}'_i \in R^{F'}$.

Firstly, a weight matrix was applied to each node in first layer in order to transfer features into high-level features.

Secondly, a shared attentional mechanism," self-attention" was performed, in order to compute attention coefficient which indicate the importance of node j's features to node i. $e_{ij} = a\left(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j\right)$.This allows every node to attend on every other node undependent from the graph structure. In this essay, the author injected the graph structure into the mechanism by performing masked attention—they only compute attention coefficient $e_{ij} = a\left(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j\right)$ for nodes $j \in \mathcal{N}_i$,which j belongs to the neighborhood of node i. To make the coefficient easily to compare with each other, softmax function was used to normalize them:

$$\alpha_{ij} = \operatorname{softmax}_j\left(e_{ij}\right) = \frac{\exp(e_{ij})}{\sum_{k\in\mathcal{N}_i}\exp(e_{ik})}.$$

The attention mechanism used in this experiment is a single-layer feedforward neural network applying the LeakyReLU nonlinearity with negative input slope (alpha) = 0.2. After fully expended, the attention coefficient can be present as:

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

The normalized attention coefficients are used to compute a linear combination of the features corresponding to them, applying with a potentially nonlinearity (sigma), provide a final output feature for every node:

$$\vec{h}'_i = \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}\mathbf{W}\vec{h}_j\right).$$

With extending the mechanism to employ multi-head attention can stabilize the learning process of self-attention. In this case, there will be K independent attention mechanism execute the transformation of Equation 4, and their features are concatenated

$$\vec{h}'_i = \|_{k=1}^{K}\sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k\mathbf{W}^k\vec{h}_j\right).$$

Then we have the final output h' just like described before. It will have kf' features for each node instead of f' features.
Specially, when perform multi-head attention on the final layer, concatenation will be no longer sensible, so the author made some adjustment. They employed averaging and applied non-linearity at final:

$$\vec{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k\mathbf{W}^k\vec{h}_j\right).$$

4 Experiments

We made some changes in the original model and used the dataset Cora to test the performance of the changed model.

1) Change the activation function
We tried to use Relu as activation function and ran the model 4 times to test the accuracy of the results. Some of the test results are slightly worse than the results showed in the paper. The activation function used in the paper was Elu and the accuracy was $83.0 \pm 0.7\%$. When we use Relu as activation function, the average accuracy is 82.3749%, which is still in the range of the accuracy using Elu. However, the worst accuracy is 81.0999%. The reason for that is that, compared to Relu, the output of elu is negative when $x < 0$. We can push the output of the activation function to 0 to reach the effect of batchnormlization and the amount of the calculation is reduced.

2) Add a layer to the model
The model used in the code reference has 8 units in the hidden layer. We tried to add a same hidden layer to see if it can improve the accuracy of the results. After running 3 times, we found that the performance of the original model has decreased to: 78.299%, 77.199%, 77.899%.

To analyse the reason for that, we think that what model should learn is the attention of the nodes in the original graph. If we add one hidden layer to the model, what is learned is the attention of the transformed nodes representation, which means we would lose part of the information about the original nodes. If we use only one hidden layer, the direct attention among the nodes would be learned.

3) Use GradientDescentOptimizer
Instead of using the AdamOptimizer as in the code reference, we tried to use GradientDecentOptimizer. After running the model 6 times, we got an average accuracy of 16.53%, which is a very bad performance compared to the original model's $83.0 \pm 0.7\%$. Nevertheless, in the worst case, we got an accuracy of 12.899%. According to our analysation, the reason is that when we use the AdamOptimizer, it intergrates the Momentum and RMSProp. As a result, when we seek the optimal point, the path would be gentle. But if we use GradientDecentOptimizer, the path oscillates heavily. It leads to a nun-monotonous change of the loss value. Thus, the early-stop will be triggered, so we got a very bad accuracy.

References

[1]Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, Yoshua Bengio: Graph Attention Networks,2018