Korea Advanced Institute of Technology

Department of Mathematical Sciences

Semi-supervised graph node classification of CORA dataset using GCN

Author:
Junghyun Lee

Instructor:
Prof. Chang Dong Yoo

Homework #4

EE531: Statistical Learning Theory, Fall 2019

December 24, 2019

Abstract

In this homework, I have implemented a simple version of GCN(Graph Convolutional Network) to do a semi-supervised graph node classification of CORA dataset. The algorithm is based upon the work by Kipf & Welling, 2016.

(Coding done in Google Colaboratory.)

Contents

Chapter 1

Implementation

Implementation was done using the Google Colaboratory. All the modifications that I've done to the original template are accompanied by the comments that I've put.

Below, I've added additional explanations for some of the features that I've used or modified.

1.1 Initialization

I have initialized some boolean parameters to be used later on for the training/evaluation/testing.

```
# Dropout setting
#DROPOUT = 0
DROPOUT = 0.5

# Boolean value for Xavier initialization
# Used uniform xavier initialization from Glorot & Bengio, 2010[2]
XAVIER = True
#XAVIER = False
```

Figure 1.1: Initialization

1.2 TensorBoardColab

TensorBoard is a visualization toolkit that supports various experimentation. Especially, it is useful in tracking and visualizing metrics such as loss and accuracy.

It is supported in multiple platforms, including Tensorflow, Google Colaboratory...etc.

```
# Initialize tensorboard for visualization
# Note : click the Tensorboard link to see the visualization of training/testing results
tbc = TensorBoardColab()
```

Figure 1.2: TensorBoardColab

1.3 GraphConvolutionLayer

1.3.1 Parameter Initialization

Here I have given myself two choices for initializing weights:

1. Sampling from $U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$[2]

2. Xavier initialization[1]

For either case, bias was initialized by sampling from $U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$.

```python
def init_parameters(self):
    # ========= YOU CODE HERE =========
    # Initializing weight and bias
    stdv = 1. / math.sqrt(self.weight.size(1))

    # With Xavier Initialization
    if XAVIER:
      init.xavier_uniform_(self.weight)
    # Without Xavier Initialization
    else:
      self.weight.data.uniform_(-stdv, stdv)

    # Xavier Initialization is only applied to weights, not bias!
    self.bias.data.uniform_(-stdv, stdv)

    # ================================
```

Figure 1.3: Parameter initialization

1.3.2 Forward Function

For forward propagation, I have used the heuristic of

$$forward(H) = AHW + b$$

where A is the adjacency matrix, H is the input matrix, W is the weight matrix, and b is the bias matrix.

```python
def forward(self, input, adj):
    # ========= YOU CODE HERE =========
    # Convolving your input graph with your weight corresponding adjacent
    # matrix, don't forget to add bias
    s = torch.mm(input, self.weight)
    out = torch.mm(adj, s) + self.bias
    # ================================
    return out
```

Figure 1.4: Forward propagation

1.4 GCN

1.4.1 Network Initialization

Following the work of Kipf & Welling[3], I have decided to implemented the two-layered GCN. To do that, two *GraphConvolutionLayer*'s, along with *dropout*, were initialized.

```python
def __init__(self, nfeat, nhid, nclass, dropout):
    super(GCN, self).__init__()

    # ========= YOU CODE HERE =========
    # Declaring your multiple graph convolution layers, remembering that
    # output of final layer has dimension equal to number of classes
    self.gc1 = GraphConvolutionLayer(nfeat, nhid)
    self.gc2 = GraphConvolutionLayer(nhid, nclass)
    self.dropout = dropout
    # ================================
```

Figure 1.5: Network initialization

1.4.2 Forward Function

I have used the forward model of the form:

$$forward(X, A) = \text{softmax}\left(\hat{A}\,\text{ReLU}\left(\hat{A}XW^{(0)}\right)W^{(1)}\right)$$

```python
def forward(self, x, adj):
    # ========= YOU CODE HERE =========
    # Feeding your input graph and adjacent matrix into multiple graph
    # convolution layers and activation function
    x = F.relu(self.gc1(x, adj))
    x = F.dropout(x, self.dropout, training = self.training)
    x = self.gc2(x, adj)
    # =================================
    out = F.log_softmax(x, dim=1)
    return out
```

Figure 1.6: Forward propagation

1.5 Training/Validation/Testing

(Even though I only describe one of the three here, the rest are the same. Just change the index set to the appropriate one.)

I have utilized the negative log likelihood loss. Also, I have updated the tensorboard plot for every epoch.

```python
# ========= YOU CODE HERE =========
# Feeding your input graph (nodes's feature and adjacent matrix) into your
# model and computing loss between prediction and ground truth

# input: feature matrix and the graph's adjacency matrix
# output: predicted labels
output = model(features, adj)
# Use negative log likelihood loss(nll_loss) for the datas assigned for trainig
# idx_train: range of indices for data to be used in the training process
loss_train = F.nll_loss(output[idx_train], labels[idx_train])

# Update the tensorboard plot
tbc.save_value('Loss',' train_loss', epoch, loss_train.item())
# =================================

acc_train = accuracy(output[idx_train], labels[idx_train])
# Update the tensorboard plot
tbc.save_value('Accuracy',' train_accuracy', epoch, acc_train.item())
```

Figure 1.7: Training phase

1.6 Confusion Matrix

For the confusion matrix plot, I have utilized the external github open repository: `https://github.com/wcipriano/pretty-print-confusion-matrix`.

```python
from sklearn.metrics import confusion_matrix

# Create numpy arrays for true/predicted labels
with torch.no_grad():
  true = labels[idx_test].cpu().numpy()
  predicted = model(features, adj)[idx_test].max(1)[1].type_as(labels).cpu().numpy()

# Plot confusion matrix
true_size = len(true) > 10

plot_confusion_matrix_from_data(true, predicted, columns,
  annot = True, cmap = 'Oranges', fmt = '.2f', fz = (lambda x: 12 if x else 9)(true_size),
  lw = 0.5, cbar = False,  figsize = (lambda x: [9,9] if x else [12,12])(true_size),
  show_null_values = 2, pred_val_axis = 'y')
```

Figure 1.8: Confusion matrix

Chapter 2

Results

Here,

- Original model: One with uniform initialization with dropouts(rate=0.5).
- Ver. 1: Original model with Xavier (Uniform) Initialization
- Ver. 2: Original model without dropouts
- Ver. 3: Original model with Xavier Initialization and without dropouts

In all the plots, x-axis corresponds to the epoch. As for the graphs,

- Accuracy graph
  - Dark red: train accuracy
  - Bright red: validation accuracy
- Loss graph
  - Dark blue: train loss
  - Bright blue: validation loss

2.1 Original model

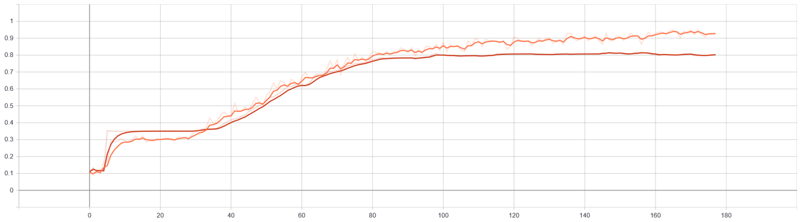

Figure 2.1: Accuracy graph

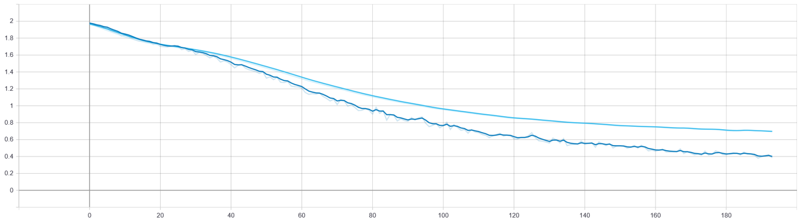

Figure 2.2: Loss graph

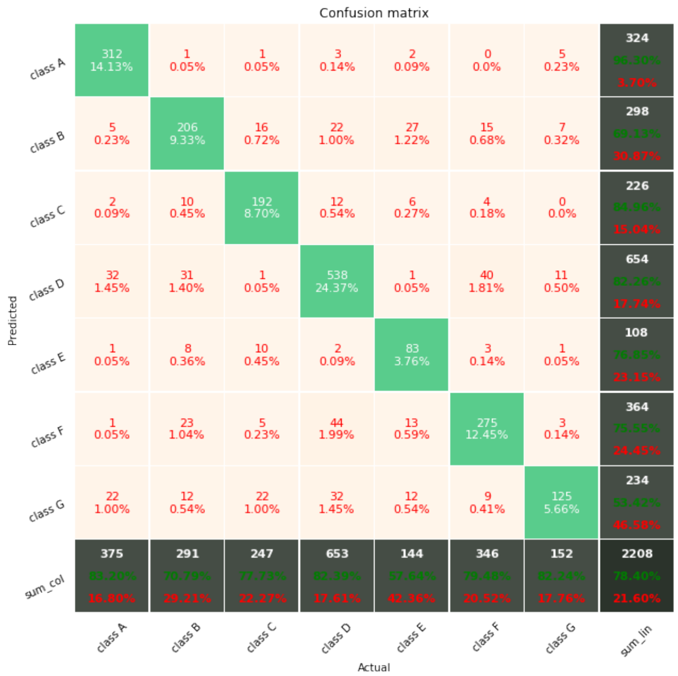

Figure 2.3: Confusion matrix

2.2 Modified Model (Ver. 1)

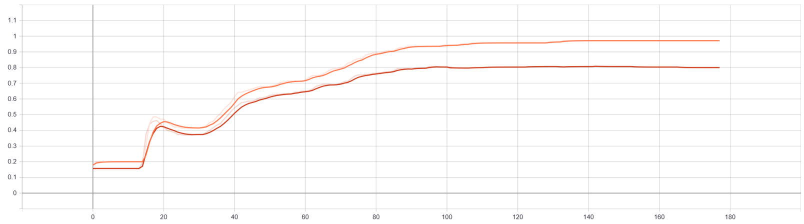

Figure 2.4: Accuracy graph

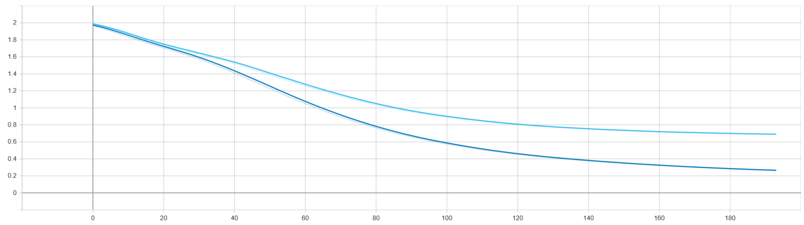

Figure 2.5: Loss graph

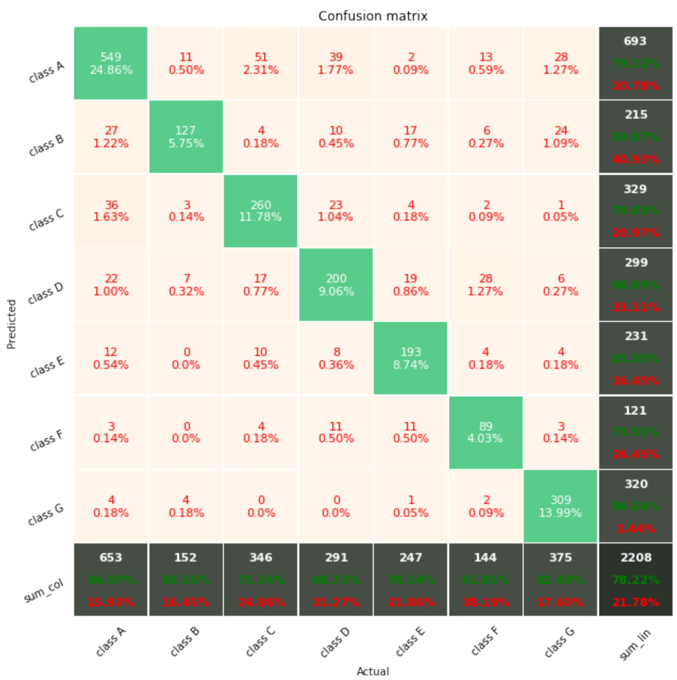

Figure 2.6: Confusion matrix

2.3 Modified Model (Ver. 2)

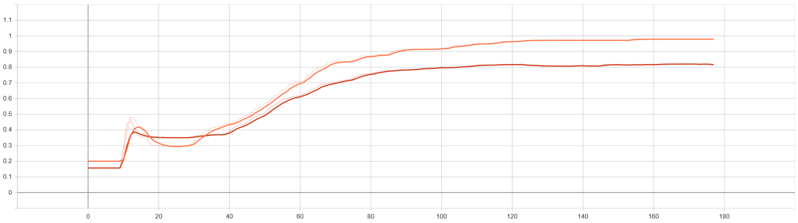

Figure 2.7: Accuracy graph

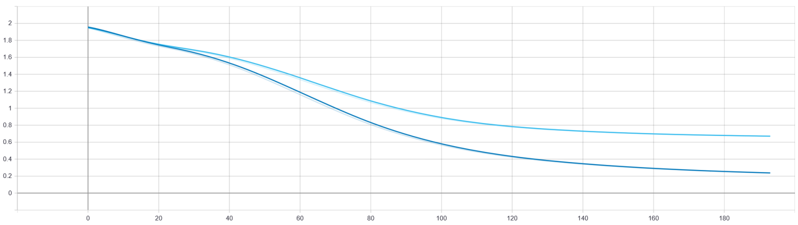

Figure 2.8: Loss graph

Figure 2.9: Confusion matrix

2.4 Modified Model (Ver. 3)

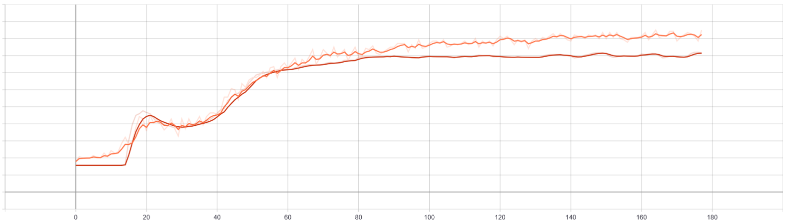

Figure 2.10: Accuracy graph

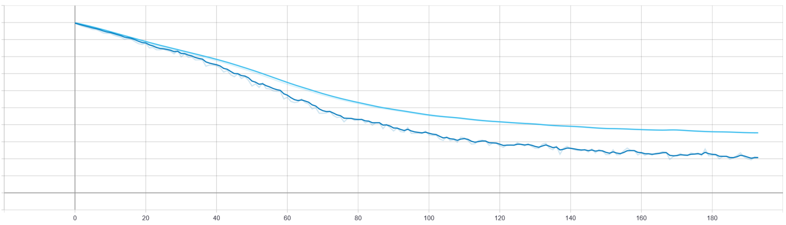

Figure 2.11: Loss graph

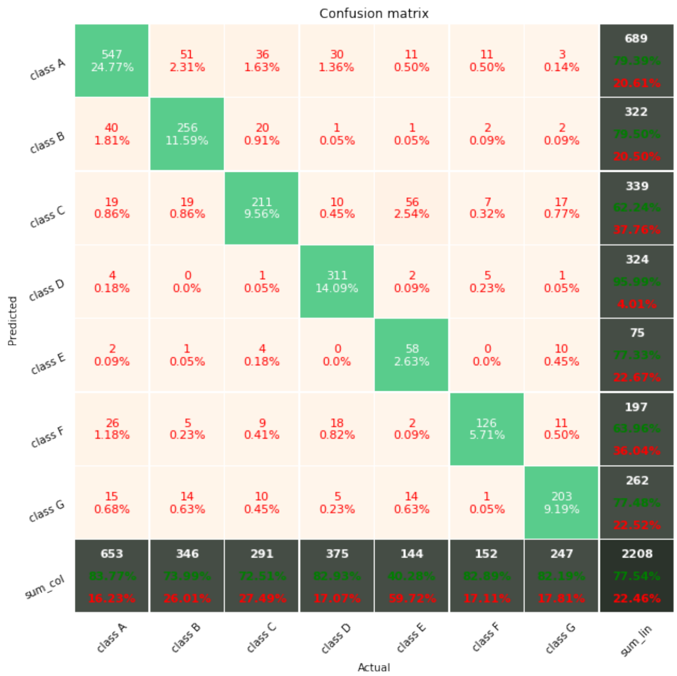

Figure 2.12: Confusion matrix

Chapter 3

Conclusion

It seems that the slight modifications that I've tried did nothing to improve the model's accuracy. Indeed, the original model reports accuracy of 78.40%, which tied with Ver. 2 and is strictly greater than the other two versions.

This accuracy is actually lower than the reported accuracy from Kipf & Welling, which is actually 81.5%. I can't really explain why this is the case... But one guess is that maybe getting rid of the bias term might increase the accuracy? (<− just a wild guess)

One interesting observation is that Ver. 3, the model with the most complexity out of the 4 tested, performed the worst with accuracy 77.54%. It shows that adding in feature/model complexity doesn't always increase the accuracy; it might even decrease it!

Bibliography

[1] 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, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*. 2010, pp. 249–256. URL: `http://proceedings.mlr.press/v9/glorot10a.html`.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[3] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. 2017. URL: `https://openreview.net/forum?id=SJU4ayYgl`.