



Deep Learning

Graph Convolutional Network - Lab 2

U Kang
Seoul National University



In This Lecture

- Improve the previous implementation of graph convolutional networks (GCN)
 - Support various options for the structure
 - Improve the computational efficiency
- We mainly focus on the adjacency matrix \mathbf{A}
 - Choose the way of normalizing \mathbf{A}
 - Use a sparse implementation of \mathbf{A}



Outline

- ➡ ☐ Requirements
- ☐ Skeleton Codes
- ☐ Answers



Motivation (1)

- The current implementation is inefficient
- The main reason is that we use a dense implementation of the adjacency matrix A
 - A real-world graph has high sparsity
 - 99.93% for our Cora dataset
 - The zero elements waste our resources
- Using a sparse implementation can boost the computational efficiency of our GCN



Motivation (2)

- The current implementation fixes many choices for the structure of a GCN
 - Way to normalize the matrix \mathbf{A}
 - The number of layers
 - The size of each layer
 - ...
- We aim at implementing a flexible class of a GCN that supports structure options as arguments



Requirements

- Use a sparse implementation of the matrix \mathbf{A}
 - The computation should be $O(E)$ instead of $O(V^2)$
- Make it possible to choose a normalization type
 - Symmetric normalization: $\tilde{\mathbf{D}}^{-1/2} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-1/2}$
 - Row normalization: $\tilde{\mathbf{D}}^{-1} \tilde{\mathbf{A}}$
 - Column normalization: $\tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-1}$
- (Optional) Make it possible to choose the structure of a GCN such as the number of layers



Sparse Implementations

- You should utilize the *tf.sparse* package
 - An official implementation of sparse tensors
 - https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/sparse
- Create a sparse tensor from indices and values
 - Such as *tf.sparse.SparseTensor(indices, values, (N, N))*
 - *Indices* is like $[(1, 3), (4, 5), (8, 6), \dots, (10, 11)]$
 - *Values* is like $[1, 1, 1, \dots, 1]$



Outline

☒ Introduction

 ☐ **Skeleton Codes**

☐ Answers



Data Preprocessing (1)

■ Import packages including TensorFlow

```
1 import numpy as np
2 import tensorflow as tf
3 import pandas as pd
```

■ Read the edges of our Cora graph

```
1 path = './cora/cora.content'
2 cora_content = pd.read_csv(path, sep='\t', header=None)
3 cora_content.head()
```



Data Preprocessing (2)

■ Extract graph information from the contents

```
1 ids = cora_content[0].values # paper(node) ids
2 vecs = cora_content.iloc[:, 1:1434].values # node features
3 labels = cora_content[1434].values # node labels
4
5 for l in np.unique(labels):
6     print(l, labels[labels == l].shape[0])
```



Data Preprocessing (3)

■ Transform the labels into one-hot vectors

```
1 from sklearn.preprocessing import LabelEncoder, OneHotEncoder
2
3 # node label one hot encoding
4 labels_onehot = LabelEncoder().fit_transform(labels)
5 labels_onehot = np.expand_dims(labels_onehot, axis=1)
6 labels_onehot = OneHotEncoder().fit_transform(labels_onehot)
7 labels_onehot = labels_onehot.toarray()
8
9 print(labels_onehot[:5])
```



Data Preprocessing (4)

■ Split the nodes into training and test sets

```
1  from sklearn.model_selection import train_test_split
2
3  num_classes = 7
4  num_per_train = 10
5  num_per_test = 100
6
7  y_train, y_test, idx_train, idx_test = train_test_split(
8      y, inds, stratify=y, random_state=42,
9      train_size=num_classes * num_per_train,
10     test_size=num_classes * num_per_test)
11
12  idx_train, idx_valid = train_test_split(
13     idx_train, stratify=y_train, random_state=42,
14     train_size=int(num_classes * num_per_train * 0.8),
15     test_size=int(num_classes * num_per_train * 0.2))
16
17  print(idx_train.shape, idx_valid.shape, idx_test.shape)
```



GCN Class

- Our GCN class has only two functions:
 - `self.__init__`: The class initializer
 - `self.apply_layers()`: The core function that applies a series of graph convolutions on placeholders
- We provide the initializer as a skeleton code
- You may need the following packages

```
1 from tensorflow import sparse
2 from tensorflow.layers import Dense
```



GCN_INITIALIZER (1)

- Our initializer is divided into a few sections
- Set the hyperparameters and layers of a GCN

```
class GCN2(Model):  
    def __init__(self, indices, values, input_dim=1433,  
                 hid_dim=64, num_classes=7, num_nodes=2708,  
                 num_layers=2):  
        super(GCN2, self).__init__()  
  
        # Hyperparameters of a model  
        self.num_nodes = num_nodes  
        self.input_dim = input_dim  
        self.num_classes = num_classes  
        self.hid_dim = hid_dim  
        self.num_layers = num_layers  
  
        self.indices = indices  
        self.values = tf.cast(values, dtype='float32')  
  
        # Define layers  
        self.dense_layers = [Dense(self.hid_dim, kernel_initializer='he_normal', activation='relu')  
                              for _ in range(self.num_layers)]  
        self.dense_layers.append(Dense(self.num_classes, kernel_initializer='he_normal'))
```



GCN Initializer (2)

■ Set a loss function

```
def loss_fn(self, logits, labels, indices):  
    _labels = tf.gather_nd(labels, indices)  
    _logits = tf.gather_nd(logits, indices)  
    loss = tf.nn.softmax_cross_entropy_with_logits(labels=_labels, logits=_logits)  
    return tf.reduce_mean(loss)
```

■ Evaluation function

```
def evaluate(self, x, labels, indices):  
    logits = self.call(x)  
    loss = self.loss_fn(logits, labels, indices)  
    _logits = tf.gather_nd(logits, indices)  
    _labels = tf.gather_nd(labels, indices)  
  
    pred = tf.argmax(_logits, axis=1)  
    ans = tf.argmax(_labels, axis=1)  
    correct = tf.equal(pred, ans)  
    acc = tf.reduce_mean(tf.cast(correct, tf.float32))  
    return loss, acc
```



Training Process (1)

■ Define training function for the class GCN2

```
def train(self, x, labels, idx_train, idx_val, optimizer, max_epochs=20):
    for epoch in range(1, max_epochs+1):
        with tf.GradientTape() as tape:
            logits = self.call(x)
            train_loss = self.loss_fn(logits, labels, idx_train)

            grad_list = tape.gradient(train_loss, self.weights)
            grads_and_vars = zip(grad_list, self.weights)
            optimizer.apply_gradients(grads_and_vars)

        # Evaluation
        train_loss, train_acc = self.evaluate(x, labels, idx_train)
        valid_loss, valid_acc = self.evaluate(x, labels, idx_val)
        print(f"Epoch {epoch:3d}: {train_loss:.4f}, {train_acc*100:.2f}%",
              f"{valid_loss:.4f}, {valid_acc*100:.2f}%")
```




Training Process (2)

- Initialize a GCN and necessary variables
- You should implement `get_adj_matrix()` and `normalize()` to support our requirements

```
num_nodes = ids.shape[0]
indices, values = get_adj_matrix(ids)
values = normalize(indices, values, num_nodes, way='both')

train_mask = np.expand_dims(idx_train, axis=1)
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

gcn2 = GCN2(indices=indices, values=values,
            input_dim=x.shape[1], hid_dim=64, num_classes=num_classes,
            num_nodes=x.shape[0], num_layers=2)
_idx_train = np.expand_dims(idx_train, axis=1)
_idx_val = np.expand_dims(idx_valid, axis=1)

gcn2.train(x=x, labels=y, idx_train=_idx_train, idx_val=_idx_val, optimizer=optimizer, max_epochs=20)
```




Summary

- You should implement the three functions
 - GCN2.call()
 - get_adj_matrix()
 - normalize()



Outline

- ☒ Requirements
- ☒ Skeleton Codes
-  ☐ Answers



call

- Create a sparse **A** and apply graph convolutions
- We adopt a residual connection (why?)

```
def call(self, x):  
    A_size = (self.num_nodes, self.num_nodes)  
    A = sparse.SparseTensor(  
        self.indices, self.values, A_size)  
  
    L = tf.cast(x, 'float32')  
    for l in range(self.num_layers):  
        L_new = sparse.sparse_dense_matmul(A, L)  
        L_new = self.dense_layers[l](L_new)  
        if l > 0:  
            L_new = L + L_new # A residual connection  
        L = L_new  
    return self.dense_layers[-1](L)
```



get_adj_matrix

- Generate an adjacency matrix with self-loops

```
1 def get_adj_matrix(ids):
2     num_nodes = ids.shape[0]
3     cites = np.loadtxt('./cora/cora.cites', dtype=np.int32)
4     id_map = {v: u for u, v in enumerate(ids)}
5     indices = [(e, e) for e in range(num_nodes)]
6     for node1, node2 in cites:
7         if node1 != node2:
8             idx1 = id_map[node1]
9             idx2 = id_map[node2]
10            indices.append((idx1, idx2))
11            indices.append((idx2, idx1))
12    indices = np.array(indices)
13    values = np.ones(indices.shape[0])
14    return indices, values
```



normalize

- Normalize the matrix based on the argument

```
1 def normalize(indices, values, num_nodes, way='both'):  
2     values_sum = np.ones(num_nodes)  
3     for node1, node2 in indices:  
4         values_sum[node1] += 1  
5         values_sum[node2] += 1  
6  
7     if way == 'both':  
8         values /= np.sqrt(values_sum[indices[:, 1]])  
9         values /= np.sqrt(values_sum[indices[:, 0]])  
10    elif way == 'row':  
11        values /= values_sum[indices[:, 0]]  
12    elif way == 'col':  
13        values /= values_sum[indices[:, 1]]  
14    else:  
15        raise ValueError()  
16    return values
```



What You Need To Know

- How to improve the basic implementation of a TF model by generalizing its functions
- How to use sparse tensors in TF for efficiency
- How to implement an efficient GCN which is scalable to a large dataset for real-world scenarios



Questions?