Neural network programming: graph neural networks

Erik Spence

SciNet HPC Consortium

23 May 2023

Today's code and slides

You can get the slides and code for today's class at the SciNet Education web page.

`https://scinet.courses/1271`

Click on the link for the class, and look under "Lectures", click on "Graph Neural Networks".

Today's class

This class will cover the following topics:

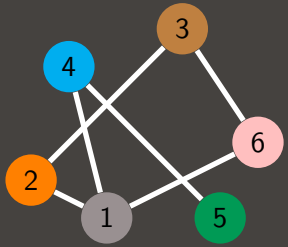
- Graphs.
- Graph neural networks.
- Graph convolutional neural networks.
- Example.

Please ask questions if something isn't clear.

What is a graph?

What is a graph?

- A "graph" is a type of data structure.
- It is built of two main components:
  - "nodes": also called "vertices", which are usually the "things" that make up the graph,
  - "edges": connections between the nodes which describe some sort of relationship between them.
- The nodes can contain labels or features or both.
- The edges can also contain weights, directions, or other characteristics that quantify the relationship between the nodes.
- This data structure shows up in any situation where there are relationships between components.



Applying neural networks to this type of data has been an active area of research for many years.

Examples of graphs

Graphs show up in many different situations.

- social networks,
- traffic problems,
- chemistry, drug discovery,
- recommendation systems,
- images,
- text,
- and many many other applications.

Graphs show up in many areas where you might not expect it.

Graph neural networks

What are graph neural networks (GNNs)?

- Graphs contain many different types of data within them:
  - nodes: labels and features,
  - edges: connections between nodes, and associated weights, labels or features,
  - whole graph: may also have labels and features.
- As with CNNs, where we adapted the network to the nature of the data, to treat a graph properly using a neural network we need to adapt to this disparate-but-structured data format.
- Graph neural networks are neural networks which have been specially-crafted to deal with the unique characteristics of graphs.

To use the graph as input data to a neural network requires modifications to the network we've already been using.

Graph neural network problems

What sort of problems can GNNs solve?

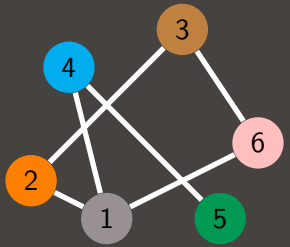
- Node problems:
  - classifying the labels of nodes
  - predicting features of nodes
- Edge problems:
  - predicting weights of edges
  - predicting missing edges
- Whole-graph problems:
  - classifying graphs
  - predicting features of the whole graph
- Combinations of the above.

Pretty much, if you can imagine it you can try to apply a GNN to it.

How do you include a graph as input?

At first it's not be obvious how you would use a graph as an input to a neural network.

- Why? Because unlike array-like data, each data point will have a different number of nodes and edges. How do we input that into the network?
- You could include the nodes, with their labels and features, as the inputs and targets of the network.
- You could do the same thing with the edges, and their labels and features.
- But both of these approaches ignore the interconnectedness of the nodes.
- We need a way of encoding the fact that the nodes are connected to specific neighbours.



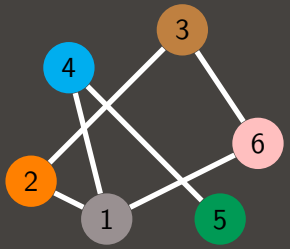
The solution to this problem is to use an Adjacency Matrix.

Adjacency matrices

The Adjacency matrix of a graph encodes which nodes have connections between them.

In its simplest form, without direction or weights, the matrix just consists of a collection of zeros and ones. The matrix below corresponds to the graph to the right.

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$



Note that the diagonal is populated by zeros. There are no edges that connect a node to itself.

Graph Convolutional Networks (2017)

How do you work the adjacency matrix into the input of the network, or a layer?

- If both the node features and the adjacency matrix are used as input to a given NN layer, we can encode neighbour information into the network.
- There are several ways that this might be accomplished. One early way of doing this was

$$f_\ell(H_{\ell-1}, A) = g\left(AH_{\ell-1}w_\ell\right)$$

where

- f_ℓ is the output of the layer,
- $H_{\ell-1}$ is the output of the previous layer (the input to this layer),
- g is the activation function,
- A is the adjacency matrix,
- w_ℓ are the layer's weights.

This approach to GNNs is known as a Graph Convolutional Network (GCN).

Graph Convolutional Networks, continued

$$f_\ell(H_{\ell-1}, A) = g\left(AH_{\ell-1}w_\ell\right)$$

How does this work?

- A is the adjacency matrix, with dimension (num_nodes, num_nodes),
- $H_{\ell-1}$ is the output of the previous layer, with dimension (num_nodes, num_node_features),
- w_ℓ are the layer's weights (num_node_features, num_node_features).

This matrix multiplication leads to a layer output of dimension (num_nodes, num_node_features), which can then be fed into the next layer. All of the layer's trainable parameters are in the w_ℓ array. Note that there are no neurons as such.

Also note that these dimensions only apply to a single input graph, since each graph will have a different number of nodes.

Graph Convolutional Networks, continued more

Using just the matrix product $\boldsymbol{A}\boldsymbol{H}_{\ell-1}\boldsymbol{w}_{\ell}$ as the output of the NN layer is not the best approach. A couple of modifications need to be added.

The first problem we encounter is with $\boldsymbol{A}$ itself. If we multiply by $\boldsymbol{A}$, then each node gets interaction information about its neighbours, but the information about itself is lost (assuming that the diagonals of $\boldsymbol{A}$ are zero). This is fixed by putting ones on the diagonal of $\boldsymbol{A}$.

Thus, in practice, we use $\hat{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$, where $\boldsymbol{I}$ is the identity matrix.

Graph Convolutional Networks, continued some more

The second problem has to do with the number of edges each node has.

- If $\boldsymbol{A}$ is not normalized then the scale of the values of the features will be changed, depending on the number of connections a given node has. This is fixed by normalizing each row of $\boldsymbol{A}$.
- If $\boldsymbol{D}$ is a diagonal matrix containing the degree of each row of $\boldsymbol{A}$ (the number of connections each node has), then we can normalize by using $\boldsymbol{D^{-1}A}$. This corresponds to averaging the features of the neighbouring nodes.
- In practice, a symmetric normalization is usually used: $\boldsymbol{D^{-\frac{1}{2}}AD^{-\frac{1}{2}}}$.

$$\boldsymbol{f(H_{\ell-1}, A) = g\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H_{\ell-1}w_{\ell}\right)}$$

where $\hat{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$, $\boldsymbol{I}$ is the identity matrix, and $\hat{\boldsymbol{D}}$ is the degree matrix of $\hat{\boldsymbol{A}}$.

Graph Convolutional Networks, continued even more

$$f(H_{\ell-1}, A) = g\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H_{\ell-1}w_{\ell}\right)$$

This is the operation that is applied at each GCN layer.

- Note that there aren't any neurons, as such. All the weights are buried in the matrix, w_{ℓ}.
- Note also that there aren't any biases, though in theory we could add them.
- The activation function is applied to the resulting matrix, element-by-element.
- By applying this matrix operation, information is spread from a single node to its immediate neighbours.
- Consequently, to spread information about a given node further than the immediate neighbours, multiple of such operations must be performed.

To build such an operation we will create a custom Keras layer.

Custom Keras layers

To use a layer that performs custom operations requires creating a new type of layer.

- Keras allows you to define custom layers for your neural network.
- The layer is defined as an extension to the tensorflow.keras.layers.Layer class.
- A custom layer in Keras must define three methods:
  - __init__: the constructor, defines layer attributes and weights that do not depend on the size of the input,
  - build(input_shape): the function which builds the layer, and defines the weights and biases that depend on the input shape.
  - call(inputs): the function which defines how the inputs are processed through the layer, and output.
- Once defined, the layer can be used like any other layer.

We will define two types of such layers for our GCN.

Our custom Keras layer

The custom layer is a class.

- The class extends Keras' "Layer" class.
- The first thing we define is the constructor.
- All we do here is define the activation function.
- The "super" function accesses the kl.Layer functions, in this case the base constructor.

You will need to understand object oriented programming in Python to understand what's going on here.

```python
# mutag_layers.py
import tensorflow as tf
import tensorflow.keras.activations as ka
import tensorflow.keras.layers as kl

class GCNLayer(kl.Layer):

  def __init__(self, activation = None, **kwargs):
    super(GCNLayer, self).__init__(**kwargs)
    self.activation = ka.get(activation)

  def build(self, input_shape):
    # Get the input shape, then build the weights.
    node_shape, adj_shape = input_shape
    self.w = self.add_weight(
      shape = (node_shape[2], node_shape[2]),
      name = 'w', trainable = True)
```

Our custom Keras layer, continued

The final piece is the call function.

- All that needs to be done is craft the operation of the layer.
- Note that the adjacency matrix is returned as well as the output, so it can be used in the subsequent layer.

```
# mutag_layers.py, continued
def call(self, inputs):
  # The input is a tuple, containing H and the
  # adjacency matrix.
  H, adj = inputs

  # The degree of the adjacency matrix, normalized
  degree = tf.reduce_sum(adj, axis = -1)
  norm_degree = tf.linalg.diag(tf.pow(degree, -0.5))

  # Normalize the adjacency matrix.
  norm_adj = tf.matmul(norm_degree,
    tf.matmul(adj, norm_degree))

  # The output, before activation.
  output = tf.matmul(norm_adj, tf.matmul(H, self.w))

  return self.activation(output), adj
```

Example

Let's do an example. Consider the "MUTAG" data set.

- A collection of graph data sets from University of Dortmund.
- Consists of molecules, each with a label, -1, or 1.
- Each node has a label, which will be one-hot-encoded into a feature.
- Each node also has as an adjacency matrix.
- The edges also have labels.
- There are 188 molecules in total.

The goal is to classify the molecules into the correct category. The data can be found here:

`https://www.chrsmrrs.com/graphkerneldatasets/MUTAG.zip`

Example, our approach

How shall we solve this problem?

- Build a neural network consisting of several Graph Convolutional layers.
- We will use the labels of the nodes as input, as well as the adjacency matrix. We will not use the edge labels in this example.
- As shown previously, we'll pass both the output of the previous layer's calculation, and the adjacency matrix, to each subsequent layer.
- But for this to work we'll need another custom layer that can deal with the special output of the Graph Convolutional layer, the adjacency matrix in particular.
- We'll follow this by a fully-connected layer.
- This will then be followed by a 2-node output layer, of the type we usually use for classification problems.

The goal is to classify the molecules into the correct category.

A second custom Keras layer

We need a second custom layer to transition from our GCN layers to regular Keras layers.

- The GCN layer passes both the calculated output and the adjacency matrix to the successive layer.
- That's fine for successive GCN layers.
- But we need a special layer to transition between our GCN layer and fully-connected layers.

This simple class ignores the incoming adjacency matrix, and reduces the data down by taking the mean.

```
# mutag_layers.py, continued

class GraphReduction(kl.Layer):

  def __init__(self, **kwargs):
    super(GraphReduction, self).__init__(**kwargs)

  def call(self, inputs):
    # Ignore the adjacency matrix.
    # Just reduce the inputs.
    H, adj = inputs
    reduction = tf.reduce_mean(H, axis = 1)
    return reduction
```

The data is reduced in dimensionality from (num_nodes, num_node_features) to (num_node_features).

Example, getting the data

Getting the data prepared is quite involved. Here is an outline of the steps. These are found in mutag_utilities.py:

- The data is scattered across 4 files. The files contain the list of edges, the labels for the graphs, the labels for the nodes, and which graph each node belongs to.
- A data frame containing the nodes, and their associated labels, is built.
- The networkx Python package is used to build the graph for each molecule.
- This graph is used to build the adjacency matrix for the molecule. The identity matrix is added to this matrix.
- A generator is created, which returns (yields)
  - the array of features for each node in a given graph (molecule),
  - the adjacency matrix for the graph, and
  - the one-hot-encoded category for the graph.
- A tensorflow Dataset object is created from this generator.

Example, getting the data, continued

A special point needs to be made about each data point:

- The dimensionality of the incoming data is different for each molecule (graph).
- Why? Because the number of nodes (atoms) in each molecule is different.
- What are the dimensions of the data?
  - the input array is (num_nodes, num_node_features).
  - the adjacency matrix has dimension (num_nodes, num_nodes).
- The array of weights is of constants size (num_node_features, num_node_features).

We will only be able to run on batches of 1 in this example, since data can only be 'batched' when each data point has the same dimension.

Example, building the model

Because there are two inputs we use Keras' functional syntax.

The only constant dimension in the incoming data is the size of the one-hot-encoded node labels.

Note that we only specify the number of nodes for the fully-connected layers.

```
# mutag_model.py
import tensorflow.keras.models as km, tensorflow.keras.layers as kl
import mutag_layers as ml

def build_model(num_node_labels):
  node_input = kl.Input(shape = (None, num_node_labels))
  adj_input = kl.Input(shape = (None, None))

  x = ml.GCNLayer(activation = 'relu')([node_input, adj_input])
  x = ml.GCNLayer(activation = 'relu')(x)
  x = ml.GCNLayer(activation = 'relu')(x)
  x = ml.GCNLayer(activation = 'relu')(x)
  x = ml.GraphReduction()(x)
  x = kl.Dense(16, activation = 'tanh')(x)
  x = kl.Dense(2, activation = 'softmax')(x)

  return = km.Model(inputs = [node_input, adj_input], outputs = x)
```

Example, driver script

```python
# mutag_driver.py
import mutag_utilities as mu
import mutag_model as mm

# We need to know the characteristics of the
# data before we can build a Tensorflow Dataset.
num_node_labels = 7
num_data = 188

data = mu.create_data(num_node_labels)

num_test = round(num_data * 0.2)

# Put chuncks of data into different splits.
test_data = data.take(num_test)
train_data = data.skip(num_test)
```

```python
# mutag_driver.py, continued

model = mm.build_model(num_node_labels)

print(model.summary())

model.compile(optimization = 'adam',
  loss = 'categorical_crossentropy',
  metrics = ['accuracy'])

# Note that there is only a single data point
# in each batch.
result = model.fit(train_data.batch(1),
  epochs = 100, verbose = 2)

print(model.evaluate(test_data.batch(1)))
```

Our network using Keras, continued

```
ejspence@mycomp ~> python mutag_driver.py
Layer (type)                 Output Shape                              Param #
=================================================================================
input_1 (InputLayer)         [(None, None, 7)]                         0
input_2 (InputLayer)         [(None, None, None)]                      0
gcn_layer (GCNLayer)         ((None, None, 7), (None, None, None))     49
gcn_layer_1 (GCNLayer)       ((None, None, 7), (None, None, None))     49
gcn_layer_2 (GCNLayer)       ((None, None, 7), (None, None, None))     49
gcn_layer_3 (GCNLayer)       ((None, None, 7), (None, None, None))     49
GRLayer (GraphReduction)     (None, 7)                                 0
dense (Dense)                (None, 16)                                128
dense_1 (Dense)              (None, 2)                                 34
=================================================================================
Total params:  358
Trainable params:  358
Non-trainable params:  0
```

Our network using Keras, continued

```
Epoch 1/100
150/150 - 17s - loss:  0.6885 - accuracy:  0.5536
Epoch 2/100
150/150 - 15s - loss:  0.6696 - accuracy:  0.6250
Epoch 3/100
150/150 - 16s - loss:  0.6632 - accuracy:  0.6250
⋮
Epoch 98/100
150/150 - 15s - loss:  0.4836 - accuracy:  0.7533
Epoch 99/100
150/150 - 16s - loss:  0.4831 - accuracy:  0.7533
Epoch 100/100
150/150 - 17s - loss:  0.4832 - accuracy:  0.7533
38/38 [==============================] - 6s 15ms/step - loss:  0.5502 - accuracy:  0.8158
[0.5502200126647949, 0.8157894611358643]
ejspence@mycomp ~>
```

Example, notes

A few notes about our example.

- Obviously, the biggest problem with what we have done here is the use of a batch size of one, which eliminates any averaging which might occur with each batch.
- How could we get around this? The easiest way would be to pad the number of nodes in the graphs to the largest graph size in the data set.
- However, the presence of many zeros in the interaction matrices, and the feature data, might cause problems.
- Another approach is to represent n graphs together in a single block-diagonal adjacency matrix. This would allow several graphs to be processed simultaneously.

This would be an obvious problem to address if you started working with this family of neural networks.

GNNs, other approaches

More-generalized versions of what we have done today have been developed since 2017.

- Spectral Graph Convolutions: the matrix representation of matrix opperation given earlier are expanded in terms of Chebyshev polynomials.
- A generalization of graph convolutions is known as "message passing". This works similarly to the final custom layer we used in our GNN, but is done within the GCN layer. The information aggregated from each node's neighbour is passed through an "update function" before being passed to the next layer.
- The convolutions which we performed in our GCN layers are a simplified form of message passing.

These are approaches to explore if you go down this road.

Linky goodness

GNN APIs:

- https://graphneural.network
- https://stellargraph.readthedocs.io
- https://github.com/deepmind/graph_nets
- https://blog.tensorflow.org/2021/11/introducing-tensorflow-gnn.html

GNNs:

- https://arxiv.org/abs/1901.00596
- https://distill.pub/2021/gnn-intro
- https://cnvrg.io/graph-neural-networks