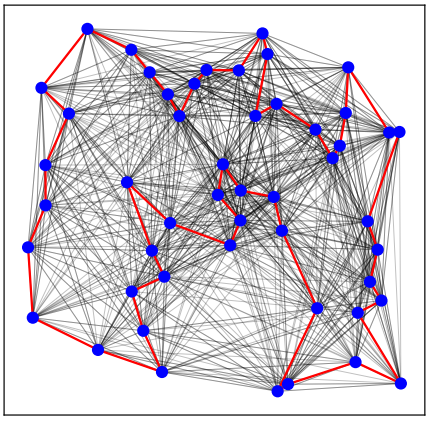
Graph Neural Networks, Homework A

Course: Advanced machine learning using neural networks, TIF360/FYM360 (2024)
Contact: Mats Granath, mats.granath@physics.gu.se
Marvin Richter, marvin.richter@chalmers.se

---

1 Background

The archetypical problem for neural network applications is image classification. Here the convolutional network (CNN) is the standard architecture. CNN's can be trained to identify generic features of images, and apply those in a parameter efficient way across the input. Mathematically images are represented as data points on a regular grid, e.g. integer values in a $3 \times 28 \times 28$ tensor for the CIFAR10 RGB images. However, other types of real world data may be better represented as graphs, i.e. nodes (or vertices) connected by edges (or links). Both the nodes and the edges may have features, typically represented by scalars or vectors. What characterizes a graph as opposed to a grid, is the irregular structure where the number of adjacent neighbors will vary from node to node, and the number of nodes may vary from graph to graph even within the same data set.



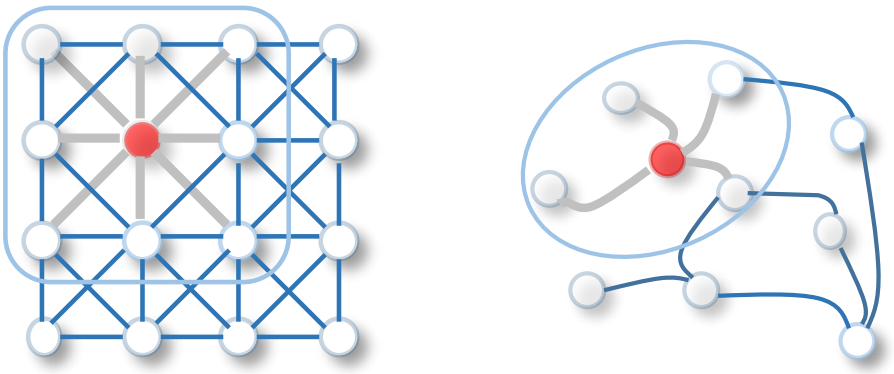
Figur 1: Complete graph with 50 nodes where edges have a scalar feature correspond to euclidean distance. The red edges marks the TSP solution on this graph. Image from [1].

An example of graph data (fig. 1) is the euclidean distances between cities, where finding the shortest cycle through all the nodes corresponds to the travelling salesperson (TSP) problem. To express this data on a regular grid would give a very sparse representation, which is likely to make the standard convolutional network inefficient and difficult to train and to generalize between problem instances. The machine learning objective would be to predict the correct edges that proposes a solution to the TSP for a graph, given a set of labeled training graphs. This corresponds to what's known as a link prediction task; for each potential link predict whether it's in the TSP cycle of the graph or not. Other types of machine learning tasks that may be relevant for graphs is node classification (e.g. categorizing people according to subcommunities in social networks), graph regression (e.g. to what extent does the molecule described by a graph have a particular chemical property), or graph classification. Depending on application and type of data,

graph neural networks can be used for both supervised learning (e.g. graph classification) or semi-supervised learning (e.g. node classification). The first part of the homework will focus on the latter, where the graph proximity of labeled and unlabeled nodes can be utilized to improve performance over supervised learning algorithms on the same data. The second part of the homework is a supervised graph classification task.

Two recommended references that provide summaries of graph neural networks are [1] and [2]. A more general background to 'Geometric deep learning' can be found in [3].

2 Graph convolutions



Figur 2: Illustration of a standard convolutional kernel with a fixed size compared to a graph convolution where the kernel size effectively depends on the connectivity of the graph. Image from [2].

A challenge with handling graphs in a neural network pipeline is that the data is irregular: the size of the graph given by the number of nodes and the degree of each node (corresponding to the number of neighbors) will vary both between and within graphs. A graph with n nodes is represented by an adjacency matrix A_{ij} with $i, j \in \{1, ...n\}$ with binary (1 or 0) entries indicating edge or no edge. Typically, this is a sparse object represented numerically as a list of edges $\{\{i, j\}\}$. (The graph may or may not be directed, corresponding to a symmetric or not adjacency matrix.) The diagonal degree matrix is given by $D_{ii} = \sum_j A_{ij}$. Each node has d features represented by a vector $\vec{x}_i$ or the corresponding $n \times d$ matrix X. There may also be m edge features given by a vector $\vec{e}_{ij}$ for the edge connecting node i to node j.

The basic structure of a graph convolution is a mapping from a graph with feature matrix X to an isomorphic graph (i.e. with the same adjacency matrix) with feature matrix X'. The feature vectors after the graph convolution may have a different dimension d', so that X' is an $n \times d'$ matrix. This is similar to how a dense layer with d' neurons would map a d dimensional input vector to a d' dimensional output, but implemented in a way that takes into account information from the feature vectors on adjacent nodes of the graph.

One of the earlier and standard forms of a graph convolution was suggested in [4] (GCNConv in PyTorch Geometric) as follows. The node feature matrix X^{l+1} in layer

$l+1$, is given by

$$X^{l+1} = \sigma(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}X^l W^l) \tag{1}$$

where $\tilde{A} = A + I_N$ (added self edges), $\tilde{D}$ is the corresponding degree matrix (which acts to normalize the feature vector irrespectively of the node connectivity), W is a trainable $d_l \times d_{l+1}$ weight matrix, and σ is the non-linearity (e.g. ReLU) that acts element-wise (and may also include a trainable bias per feature). After a trainable linear transformation (with W) the new feature vector at each node is thus an average over all the nodes to which it is connected processed through the non-linearity. With each successive layer, feature information travels in from further out in the graph.

Note that the weight matrix W is equivalent to that of a dense layer $d_\ell \to d_{\ell+1}$, which makes the graph convolution quite distinct from the standard grid convolution which has different trainable weights to each input pixel in the kernel. The reason for this is the variable size of the input (number of neighboring nodes) which would require a kernel of varying size to use the standard CNN type. Note also that this form of the graph convolution does not take edge features into account but only the structure of the graph.

There are of course a large number of more advanced convolutions that expand on this. For example, in the case that the edge features are a scalar, i.e. an edge weight, e_{ij} which is 0 if there is no edge, there is GraphConv which has the simple intuitive form

$$\vec{x}_i^{\,l+1} = \sigma(W_1^l \vec{x}_i^{\,l} + \sum_j e_{ij} W_2^l \vec{x}_j^{\,l}) \tag{2}$$

where W_1 and W_2 are two independent $d_{l+1} \times d_l$ weight matrices. Another example of graph convolutions, GATConv [5] is based on the attention mechanism where the idea is to train a more selective (anisotropic) sampling of the feature vectors of the adjacent nodes. The convolution (feature vector $d \to d'$) is standard,

$$\vec{x}_i' = \sigma(\sum_j \alpha_{i,j} W \vec{x}_j)\,, \tag{3}$$

but with sophistication hiding in the attention coefficients $\alpha_{i,j}$. The latter are given by $\alpha_{ij} = \mathrm{softmax}(\epsilon_{ij})$ with

$$\epsilon_{ij} = \mathrm{LeakyReLU}(\vec{a} \cdot [W\vec{x}_i || W\vec{x}_j])\tilde{A}_{ij} \tag{4}$$

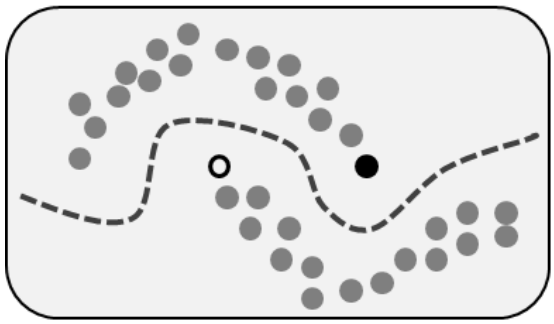
where $\vec{a}$ is a dimension $2d'$ trainable weight vector, $[\cdot||\cdot]$ is the concatenation (i.e. putting the two vectors on top of each other), and $\tilde{A}$ is included to only allow attending on adjacent nodes or the node itself. This can also be extended to having several attention heads ϵ_{ij}^k, $k = 1, ..., K$, each with its own weight matrices and attention vectors.

The graph convolutional layers are typically capped by a dense network and pooling layers that give the required output, depending on the structure of the targets. (See, e.g. appendix of [1].) For the purpose of node classification on a single graph there is actually no need for any post convolution layers. The graph convolution output is of the form of a feature vector per node that may be given appropriate length on which to apply softmax activiation to provide a distribution over the labels. (Additional dense layers could of course also be added.) For graph classification, on the other hand, a single label per graph requires that the information from all the nodes must be gathered and processed to give a single output vector for the graph. Various pooling structures may be used for this, with the simplest being a mean pooling layer that takes the mean feature vector of all nodes which can then be processed through a dense network or go directly to softmax.

3 Semi-supervised learning

As mentioned previously, a hallmark of machine learning on graphs is the use of semi-supervised learning, as exemplified by the task of node classification. Given a number of labeled nodes, the objective is to classify all nodes. Using only the node features of the labeled nodes individually for training would correspond to supervised learning, but this would not make use of the implicit information about the relations between labeled and unlabeled nodes that is contained in the graph adjacency matrix. Using a graph convolution, the feature vectors of adjacent unlabeled nodes are transferred to the labeled nodes and vice versa and incorporated in the network training, making it semi-supervised through the use of both labeled and unlabeled data.

An intuitive example of semi-supervised learning is shown in figure 3, demonstrating how a classifier can be constructed from two labeled datapoints complemented by a clustering algorithm. (The latter could be in the form of a non-linear support vector machine, that aims to make a minimal complexity separation of the data into two sets.)



Figur 3: Semi-supervised learning, illustration. Two labeled data points (empty/black) complimented by unlabeled (gray) that are identified by an unsupervised clustering algorithm. A GNN for node classification works along the same principles by using the graph connectivity to identify unlabeled with labeled nodes. (By Techerin, wikimedia.org).

4 Assignment

For this assignment you will consider two different datasets, using semi-supervised learning for node classification and supervised learning for graph classification respectively. The recommended library to use for the assignment is PyTorch Geomtric (PyG).

4.1 Warm up exercise

The first set of problems deal with the Cora dataset[6], which is citation graph of scientific papers. The nodes (papers) have features given by sparse bag-of-words vector that correspond to 1433 keywords. There are seven categories of papers based on a classification of the keywords. Out of the 2708 nodes 140 are labeled in the standard training set, 20 in each class. The dataset can be found: https://github.com/kimiyoung/planetoid or through PyG.

You are welcome to use the Colab Notebook on node classification on the Cora dataset provided in the PyTorch Geometric documentation. Either modify and run directly in

Google Colab or reuse relevant parts of the code in your own Jupyter notebook or other environment.

1) [2p]

- a) From the Planetoid Cora dataset, extract the number of nodes that are in the training set, validation set, and test set. Discuss briefly the implications for supervised versus semi-supervised learning.
- b) Train both a standard dense network using supervised learning and network using the convolution from equation 1 using semi-supervised learning.
- c) Use t-SNE to visualize the results.

All of this is essentially provided in the Colab notebook referred to above. You are also welcome to try out varying the depth and width of the network to improve the accuracy.

4.2 Decoding a quantum memory

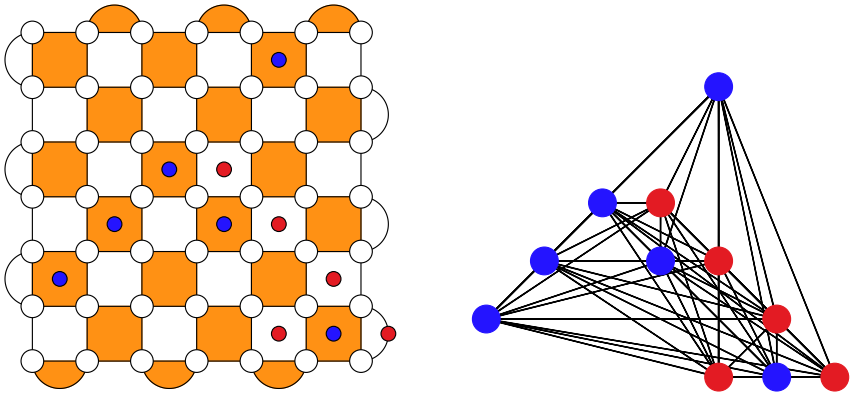
The task will be to train a network for graph classification.

4.2.1 Background

In order for a quantum computer to eventually outperform classical computers on certain tasks it should execute millions of gates on thousands of qubits. A major challenge is to make the operation fault tolerant, such that inevitable errors do not destroy the calculation. Topological stabilizer codes, such as the surface code (Fig. 4), encode logical qubit states as entangled states in a matrix of noisy physical qubits. Errors such as bit-flips or phase-flips on physical qubits are detected through non-destructive measurements. (For a recent breakthrough implementation, see [7].) Logical qubit states are protected up to strings of errors that span the code, hence the term topological codes. Measurements provide a "syndrome" of the physical errors, but do not uniquely identify them. In fact, an exponentially large number of error configurations are consistent with any given syndrome. What nevertheless makes error correction tractable is that the errors that correspond to a particular measured syndrome comes in four different logical classes $\{I, X, Z, Y\}$ (identity, bit-flip, phase-flip, or both). The decoder is an algorithm that takes as input the syndrome and predicts the correct class. Such decoder algorithms are typically based on numerically intensive search or graph algorithms. Here we will explore a pure data-driven approach. By generating a large amount of data, corresponding to syndrome graphs labeled by class, we train a graph neural network to work as decoder. For more details, see [8].

4.2.2 Problem

Train a graph neural network for graph classification using the dataset provided on Google drive. This notebook provides basic code for loading the data and relevant modules and gives some basic information about its structure.
3) [8p]



Figur 4: Illustration of a surface code quantum memory, consisting of a grid of 7×7 qubits (empty circles). The code is dynamically stabilized by doing a full set of parity measurements over four (two on boundary) qubits on each square plaquette. (White and yellow plaquettes correspond to two different types of parity measurements.) The syndrome, red and blue dots, indicate that an error has occurred on one or three of the surrounding 4 qubits. (Plaquettes that are not marked did not record an error in this measurement round.) The syndrome is mapped to a to a graph for decoding, i.e. classifying the syndrome into one of four logical sectors, as required to do do error correction. (Image from P. Havström and O. Heuts, Machine Learning Assisted Quantum Error Correction Using Scalable Neural Network Decoders).

- Divide the dataset 95/5 into training set and test set.
- Design a graph neural network with the objective to optimize the accuracy on the test set. You should be able to get a test accuracy of at least 95%, possibly up to 98%.
- Plot training and test accuracy over at least 10 epochs of training.
- Be ready to discuss your code and the components of your network.

Referenser

[1] Dwivedi, V. P., Joshi, C. K., Laurent, T., Bengio, Y. & Bresson, X. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982* (2020). URL https://arxiv.org/abs/2003.00982.

[2] Wu, Z. *et al.* A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* (2020). URL https://arxiv.org/abs/1901.00596.

[3] Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A. & Vandergheynst, P. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine* **34**, 18–42 (2017). URL https://arxiv.org/abs/1611.08097.

[4] Kipf, T. N. & Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016). URL https://arxiv.org/abs/1609.02907.

[5] Veličković, P. *et al.* Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017). URL https://arxiv.org/abs/1710.10903.

[6] Yang, Z., Cohen, W. & Salakhudinov, R. Revisiting semi-supervised learning with graph embeddings. In *International conference on machine learning*, 40–48 (PMLR, 2016). URL https://arxiv.org/abs/1603.08861.

[7] Google Quantum AI. Suppressing quantum errors by scaling a surface code logical qubit. *Nature* **614**, 676–681 (2023). URL https://www.nature.com/articles/s41586-022-05434-1.

[8] Lange, M. *et al.* Data-driven decoding of quantum error correcting codes using graph neural networks (2023). URL https://arxiv.org/abs/2307.01241.