# Graph Classification with Graph Neural Networks

*This noteboook is a julia adaptation of the Pytorch Geometric tutorials that can be found here.*

In this tutorial session we will have a closer look at how to apply **Graph Neural Networks (GNNs) to the task of graph classification**. Graph classification refers to the problem of classifiying entire graphs (in contrast to nodes), given a **dataset of graphs**, based on some structural graph properties. Here, we want to embed entire graphs, and we want to embed those graphs in such a way so that they are linearly separable given a task at hand.

The most common task for graph classification is **molecular property prediction**, in which molecules are represented as graphs, and the task may be to infer whether a molecule inhibits HIV virus replication or not.

The TU Dortmund University has collected a wide range of different graph classification datasets, known as the **TUDatasets**, which are also accessible via MLDatasets.jl. Let's load and inspect one of the smaller ones, the **MUTAG dataset**:

In [1]:
```
using Flux
using Flux: DataLoader
using Flux: logitcrossentropy, onecold, onehotbatch
using GraphNeuralNetworks
using LinearAlgebra
using MLDatasets
using Random
using Statistics
Random.seed!(1);
```

In [2]:
```
dataset = TUDataset("MUTAG")
```

```
dataset TUDataset:
  name        =>    MUTAG
  metadata    =>    Dict{String, Any} with 1 entry
  graphs      =>    188-element Vector{MLDatasets.Graph}
  graph_data  =>    (targets = "188-element Vector{Int64}",)
  num_nodes   =>    3371
  num_edges   =>    7442
  num_graphs  =>    188
```

In [3]:
```
dataset.graph_data.targets |> union
```

```
2-element Vector{Int64}:
  1
 -1
```

In [4]:
```
g1, y1  = dataset[1] #get the first graph and target
```

```
(graphs = Graph(17, 38), targets = 1)
```

In [5]:
```
reduce(vcat, g.node_data.targets for (g,_) in dataset) |> union
```

```
7-element Vector{Int64}:
 0
 1
 2
 3
 4
 5
 6
```

In [6]:
```
reduce(vcat, g.edge_data.targets for (g,_) in dataset)|> union
```

```
4-element Vector{Int64}:
 0
 1
 2
 3
```

This dataset provides **188 different graphs**, and the task is to classify each graph into **one out of two classes**.

By inspecting the first graph object of the dataset, we can see that it comes with **17 nodes** and **38 edges**. It also comes with exactly **one graph label**, and provides additional node labels (7 classes) and edge labels (4 classes). However, for the sake of simplicity, we will not make use of edge labels.

We have some useful utilities for working with graph datasets, *e.g.*, we can shuffle the dataset and use the first 150 graphs as training graphs, while using the remaining ones for testing:

In [7]:
```
graphs = mldataset2gnngraph(dataset)
graphs = [GNNGraph(g,
                                        ndata=Float32.(onehotbatch(g.ndata.t
                                        edata=nothing)
                        for g in graphs]

shuffled_idxs = randperm(length(graphs))
train_idxs = shuffled_idxs[1:150]
test_idxs = shuffled_idxs[151:end]
train_graphs = graphs[train_idxs]
test_graphs = graphs[test_idxs]
ytrain = onehotbatch(dataset.graph_data.targets[train_idxs], [-1, 1])
ytest = onehotbatch(dataset.graph_data.targets[test_idxs], [-1, 1]);
```

## Mini-batching of graphs

Since graphs in graph classification datasets are usually small, a good idea is to **batch the graphs** before inputting them into a Graph Neural Network to guarantee full GPU

utilization. In the image or language domain, this procedure is typically achieved by **rescaling** or **padding** each example into a set of equally-sized shapes, and examples are then grouped in an additional dimension. The length of this dimension is then equal to the number of examples grouped in a mini-batch and is typically referred to as the `batchsize`.

However, for GNNs the two approaches described above are either not feasible or may result in a lot of unnecessary memory consumption. Therefore, GNN.jl opts for another approach to achieve parallelization across a number of examples. Here, adjacency matrices are stacked in a diagonal fashion (creating a giant graph that holds multiple isolated subgraphs), and node and target features are simply concatenated in the node dimension (the last dimension).

This procedure has some crucial advantages over other batching procedures:

1. GNN operators that rely on a message passing scheme do not need to be modified since messages are not exchanged between two nodes that belong to different graphs.

2. There is no computational or memory overhead since adjacency matrices are saved in a sparse fashion holding only non-zero entries, *i.e.*, the edges.

GNN.jl can **batch multiple graphs into a single giant graph** with the help of `collate` option of `DataLoader` that implicitly calls `Flux.batch` on the data:

In [8]:
```
train_loader = DataLoader((train_graphs, ytrain), batchsize=64, shuffle=true
test_loader = DataLoader((test_graphs, ytest), batchsize=10, shuffle=false,
```

```
4-element DataLoader(::Tuple{Vector{GNNGraph{Tuple{Vector{Int64}, Vector{In
t64}, Nothing}}}, OneHotArrays.OneHotMatrix{UInt32, Vector{UInt32}}}, batch
size=10, collate=Val{true}())
  with first element:
  (GNNGraph{Tuple{Vector{Int64}, Vector{Int64}, Nothing}}, 2×10 OneHotMatri
x(::Vector{UInt32}) with eltype Bool,)
```

In [9]:
```
first(train_loader)
```

```
(GNNGraph(1169, 2592) with x: 7×1169 data, Bool[1 0 … 0 0; 0 1 … 1 1])
```

In [10]:
```
Flux.batch(train_graphs[1:64])
```

```
GNNGraph:
  num_nodes: 1183
  num_edges: 2630
  num_graphs: 64
  ndata:
        x = 7×1183 Matrix{Float32}
```

Here, we opt for a `batch_size` of 64, leading to 3 (randomly shuffled) mini-batches, containing all $2 \cdot 64 + 22 = 150$ graphs.

Furthermore, each batched graph object is equipped with a **`graph_indicator` vector**, which maps each node to its respective graph in the batch:

$$\text{graph-indicator} = [1, \ldots, 1, 2, \ldots, 2, 3, \ldots]$$

## Training a Graph Neural Network (GNN)

Training a GNN for graph classification usually follows a simple recipe:

1. Embed each node by performing multiple rounds of message passing
2. Aggregate node embeddings into a unified graph embedding (**readout layer**)
3. Train a final classifier on the graph embedding

There exists multiple **readout layers** in literature, but the most common one is to simply take the average of node embeddings:

$$\mathbf{x}_{\mathcal{G}} = \frac{1}{|\mathcal{V}|} \sum_{v \in \mathcal{V}} x_v^{(L)}$$

GNN.jl provides this functionality via `GlobalPool(mean)`, which takes in the node embeddings of all nodes in the mini-batch and the assignment vector `graph_indicator` to compute a graph embedding of size `[hidden_channels, batchsize]`.

The final architecture for applying GNNs to the task of graph classification then looks as follows and allows for complete end-to-end training:

In [11]:
```
function create_model(nin, nout)
        return GNNChain( GCNConv(nin => nh, relu),
                                        GCNConv(nh => nh, relu),
                                        GCNConv(nh => nh), # after this, [H
                                        GlobalPool(mean), # after this, [H]
                                        Dropout(0.5),
                                        Dense(nh, nout))
end
```

```
create_model (generic function with 1 method)
```

Here, we again make use of the `GCNConv` with $\mathrm{ReLU}(x) = \max(x, 0)$ activation for obtaining localized node embeddings, before we apply our final classifier on top of a graph readout layer.

Let's train our network for a few epochs to see how well it performs on the training as well as test set:

In [12]:
```
function eval_loss_accuracy(model, data_loader, device)
    loss = 0.
    acc = 0.
    ntot = 0
```

```
    for (g, y) in data_loader
        g, y = g |> device, y |> device
        n = length(y)
        ŷ = model(g, g.ndata.x)
        loss += logitcrossentropy(ŷ, y) * n
        acc += mean((ŷ .> 0) .== y) * n
        ntot += n
    end
    return (loss = round(loss/ntot, digits=4), acc = round(acc*100/ntot, dig
end

function train!(model; epochs=200, η=1e-2, infotime=10)
        # device = Flux.gpu # uncomment this for GPU training
        device = Flux.cpu
        model = model |> device
        opt_state = Flux.setup(Adam(η), model)


    function report(epoch)
        train = eval_loss_accuracy(model, train_loader, device)
        test = eval_loss_accuracy(model, test_loader, device)
        println("# epoch = $epoch")
        println("train = $train")
        println("test = $test")
    end

    report(0)
    for epoch in 1:epochs
        for (g, y) in train_loader
            g, y = g |> device, y |> device
            grads = Flux.gradient(model) do model
                ŷ = model(g, g.ndata.x)
                logitcrossentropy(ŷ, y)
            end
            Flux.Optimise.update!(opt_state, model, grads[1])
        end
                epoch % infotime == 0 && report(epoch)
    end
end
```

```
train! (generic function with 1 method)
```

In [13]:

```
nin = 7
nh = 64
nout = 2
model = create_model(nin, nh, nout)
train!(model)
```

```
# epoch = 0
train = (loss = 0.7051, acc = 50.0)
test = (loss = 0.6978, acc = 50.0)
# epoch = 10
train = (loss = 0.4819, acc = 74.67)
test = (loss = 0.7471, acc = 68.42)
# epoch = 20
train = (loss = 0.4786, acc = 76.67)
test = (loss = 0.6451, acc = 61.84)
# epoch = 30
train = (loss = 0.4648, acc = 77.67)
test = (loss = 0.6478, acc = 64.47)
# epoch = 40
train = (loss = 0.4543, acc = 79.0)
test = (loss = 0.7267, acc = 68.42)
# epoch = 50
train = (loss = 0.4452, acc = 78.0)
test = (loss = 0.688, acc = 68.42)
# epoch = 60
train = (loss = 0.4509, acc = 78.33)
test = (loss = 0.7459, acc = 67.11)
# epoch = 70
train = (loss = 0.4453, acc = 78.0)
test = (loss = 0.6655, acc = 69.74)
# epoch = 80
train = (loss = 0.4403, acc = 78.33)
test = (loss = 0.7645, acc = 68.42)
# epoch = 90
train = (loss = 0.4255, acc = 80.0)
test = (loss = 0.7259, acc = 68.42)
# epoch = 100
train = (loss = 0.4224, acc = 80.33)
test = (loss = 0.6988, acc = 69.74)
# epoch = 110
train = (loss = 0.4233, acc = 81.0)
test = (loss = 0.7432, acc = 67.11)
# epoch = 120
train = (loss = 0.4218, acc = 78.67)
test = (loss = 0.7177, acc = 68.42)
# epoch = 130
train = (loss = 0.4255, acc = 79.67)
test = (loss = 0.7101, acc = 71.05)
# epoch = 140
train = (loss = 0.418, acc = 81.0)
test = (loss = 0.7522, acc = 71.05)
# epoch = 150
train = (loss = 0.4137, acc = 81.0)
test = (loss = 0.7165, acc = 69.74)
# epoch = 160
train = (loss = 0.4259, acc = 81.0)
test = (loss = 0.6943, acc = 69.74)
# epoch = 170
train = (loss = 0.4156, acc = 80.0)
test = (loss = 0.7532, acc = 65.79)
# epoch = 180
train = (loss = 0.4074, acc = 81.33)
```

```
test = (loss = 0.7493, acc = 72.37)
# epoch = 190
train = (loss = 0.413, acc = 80.0)
test = (loss = 0.7774, acc = 65.79)
# epoch = 200
train = (loss = 0.4268, acc = 79.67)
test = (loss = 0.7375, acc = 73.68)
```

As one can see, our model reaches around **70% test accuracy**. Reasons for the fluctations in accuracy can be explained by the rather small dataset (only 38 test graphs), and usually disappear once one applies GNNs to larger datasets.

## Exercise 1

Can we do better than this? As multiple papers pointed out (Xu et al. (2018), Morris et al. (2018)), applying **neighborhood normalization decreases the expressivity of GNNs in distinguishing certain graph structures**. An alternative formulation (Morris et al. (2018)) omits neighborhood normalization completely and adds a simple skip-connection to the GNN layer in order to preserve central node information:

$$\mathbf{x}_i^{(\ell+1)} = \mathbf{W}_1^{(\ell+1)}\mathbf{x}_i^{(\ell)} + \mathbf{W}_2^{(\ell+1)} \sum_{j \in \mathcal{N}(i)} \mathbf{x}_j^{(\ell)}$$

This layer is implemented under the name `GraphConv` in GNN.jl.

As an exercise, you are invited to complete the following code to the extent that it makes use of `GraphConv` rather than `GCNConv`. This should bring you close to **80% test accuracy**.

In [ ]:

# Exercise 2

Define your own convolutional layer drawing inspiration from any of the already existing ones:
https://github.com/CarloLucibello/GraphNeuralNetworks.jl/blob/master/src/layers/conv.jl

You can try to:

- use MLPs instead of linear operators
- add skip connections

In [ ]: