

COERCHON_COLIN-GNN

September 6, 2026

1 Introduction to Graph Neural Nets with JAX/jraph

Lisa Wang, DeepMind (wanglisa@deepmind.com), Nikola Jovanović, ETH Zurich (nikola.jovanovic@inf.ethz.ch)

Colab Runtime:

If possible, please use a GPU hardware accelerator to run this colab. You can choose that under *Runtime > Change Runtime Type*.

Prerequisites: * Some familiarity with [JAX](#), you can refer to this [colab](#) for an introduction to JAX.
* Neural network basics * Graph theory basics (MIT Open Courseware [slides](#) by Amir Ajorlou)

We recommend watching the [Theoretical Foundations of Graph Neural Networks Lecture](#) by Petar Veličković before working through this colab. The talk provides a theoretical introduction to Graph Neural Networks (GNNs), historical context and motivating examples.

Outline: * Fundamental Graph Concepts * Graph Prediction Tasks * Intro to the jraph Library * Graph Convolutional Network (GCN) Layer * Build GCN Model with Multiple Layers * Node Classification with GCN on Karate Club Dataset * Graph Attention (GAT) Layer * Train GAT Model on Karate Club Dataset * Graph Classification on MUTAG (Molecules) * Link Prediction on CORA (Citation Network) * Bonus: Intro to Graph Adversarial Attacks

Additional Resources:

- Battaglia et al. (2018): [Relational inductive biases, deep learning, and graph networks](#)

Some sections in this colab build on the [GraphNets Tutorial colab in pytorch](#) by Nikola Jovanović.

We would like to thank Razvan Pascanu and Petar Veličković for their valuable input and feedback.

Copyright 2022 by the Authors.

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at <https://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

1.1 Setup: Install and Import libraries

```
[176]: !pip install jraph flax dm-haiku matplotlib networkx scikit-learn
```

```
Requirement already satisfied: jraph in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (0.0.6.dev0)  
Requirement already satisfied: flax in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (0.8.5)  
Requirement already satisfied: dm-haiku in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (0.0.13)  
Requirement already satisfied: matplotlib in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (3.9.2)  
Requirement already satisfied: networkx in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (3.2.1)  
Requirement already satisfied: scikit-learn in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (1.5.2)  
Requirement already satisfied: jax>=0.1.55 in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from jraph)  
(0.4.30)  
Requirement already satisfied: jaxlib>=0.1.37 in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from jraph)  
(0.4.30)  
Requirement already satisfied: numpy>=1.18.0 in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from jraph)  
(1.26.4)  
Requirement already satisfied: msgpack in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from flax) (1.1.0)  
Requirement already satisfied: optax in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from flax) (0.2.4)  
Requirement already satisfied: orbax-checkpoint in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from flax) (0.6.4)  
Requirement already satisfied: tensorstore in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from flax) (0.1.69)  
Requirement already satisfied: rich>=11.1 in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from flax) (13.9.4)  
Requirement already satisfied: typing-extensions>=4.2 in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from flax) (4.12.2)  
Requirement already satisfied: PyYAML>=5.4.1 in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from flax) (6.0)  
Requirement already satisfied: absl-py>=0.7.1 in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from dm-haiku)  
(2.1.0)  
Requirement already satisfied: jmp>=0.0.2 in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from dm-haiku)  
(0.0.4)  
Requirement already satisfied: tabulate>=0.8.9 in  
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from dm-haiku)  
(0.9.0)
```

Requirement already satisfied: contourpy>=1.0.1 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from matplotlib) (1.2.0)

Requirement already satisfied: cycler>=0.10 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from matplotlib) (0.11.0)

Requirement already satisfied: fonttools>=4.22.0 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from matplotlib) (4.51.0)

Requirement already satisfied: kiwisolver>=1.3.1 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from matplotlib) (1.4.4)

Requirement already satisfied: packaging>=20.0 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from matplotlib) (24.2)

Requirement already satisfied: pillow>=8 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from matplotlib) (11.0.0)

Requirement already satisfied: pyparsing>=2.3.1 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from matplotlib) (3.2.0)

Requirement already satisfied: python-dateutil>=2.7 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from matplotlib) (2.9.0.post0)

Requirement already satisfied: importlib-resources>=3.2.0 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from matplotlib) (6.4.0)

Requirement already satisfied: scipy>=1.6.0 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from scikit-learn) (1.10.1)

Requirement already satisfied: joblib>=1.2.0 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from scikit-learn) (1.4.2)

Requirement already satisfied: threadpoolctl>=3.1.0 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from scikit-learn) (3.5.0)

Requirement already satisfied: zipp>=3.1.0 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from importlib-resources>=3.2.0->matplotlib) (3.21.0)

Requirement already satisfied: ml-dtypes>=0.2.0 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from jax>=0.1.55->jraph) (0.4.1)

Requirement already satisfied: opt-einsum in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from jax>=0.1.55->jraph) (3.4.0)

Requirement already satisfied: importlib-metadata>=4.6 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from jax>=0.1.55->jraph) (8.5.0)

Requirement already satisfied: six>=1.5 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from python-
dateutil>=2.7->matplotlib) (1.16.0)

Requirement already satisfied: markdown-it-py>=2.2.0 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from
rich>=11.1->flax) (3.0.0)

Requirement already satisfied: pygments<3.0.0,>=2.13.0 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from
rich>=11.1->flax) (2.18.0)

Requirement already satisfied: chex>=0.1.87 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from optax->flax)
(0.1.89)

Requirement already satisfied: etils[epy] in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from optax->flax)
(1.5.2)

Requirement already satisfied: nest_asyncio in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from orbax-
checkpoint->flax) (1.6.0)

Requirement already satisfied: protobuf in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from orbax-
checkpoint->flax) (5.28.3)

Requirement already satisfied: humanize in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from orbax-
checkpoint->flax) (4.12.1)

Requirement already satisfied: toolz>=0.9.0 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from
chex>=0.1.87->optax->flax) (1.0.0)

Requirement already satisfied: mdurl~=0.1 in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from markdown-it-
py>=2.2.0->rich>=11.1->flax) (0.1.2)

Requirement already satisfied: fsspec in
c:\users\colin\anaconda3\envs\alaska_deep\lib\site-packages (from
etils[epath,epy]->orbax-checkpoint->flax) (2024.2.0)

```
[177]: # Imports
%matplotlib inline
import functools
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
import jax.tree_util as tree
import jraph
import flax
import haiku as hk
import optax
import pickle
import numpy as onp
```

```
import networkx as nx
from typing import Any, Callable, Dict, List, Optional, Tuple
from sklearn.metrics import roc_auc_score

import os
os.environ["KMP_DUPLICATE_LIB_OK"] = "True"
```

1.2 Fundamental Graph Concepts

A graph consists of a set of nodes and a set of edges, where edges form connections between nodes.

More formally, a graph is defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of vertices / nodes, and $\mathcal{E}$ is the set of edges.

In an **undirected** graph, each edge is an unordered pair of two nodes $\in \mathcal{V}$. E.g. a friend network can be represented as an undirected graph, assuming that the relationship “*A is friends with B*” implies “*B is friends with A*”.

In a **directed** graph, each edge is an ordered pair of nodes $\in \mathcal{V}$. E.g. a citation network would be best represented with a directed graph, since the relationship “*A cites B*” does not imply “*B cites A*”.

The **degree** of a node is defined as the number of edges incident on it, i.e. the sum of incoming and outgoing edges for that node.

The **in-degree** is the sum of incoming edges only, and the **out-degree** is the sum of outgoing edges only.

There are several ways to represent $\mathcal{E}$: 1. As a **list of edges**: a list of pairs (u, v) , where (u, v) means that there is an edge going from node u to node v . 2. As an **adjacency matrix**: a binary square matrix A of size $|\mathcal{V}| \times |\mathcal{V}|$, where $A_{u,v} = 1$ iff there is a connection between nodes u and v . 3. As an **adjacency list**: An array of $|\mathcal{V}|$ unordered lists, where the i th list corresponds to the i th node, and contains all the nodes directly connected to node i .

Example: Below is a directed graph with four nodes and five edges.

The arrows on the edges indicate the direction of each edge, e.g. there is an edge going from node 0 to node 1. Between node 0 and node 3, there are two edges: one going from node 0 to node 3 and one from node 3 to node 0.

Node 0 has out-degree of 2, since it has two outgoing edges, and an in-degree of 2, since it has two incoming edges.

The list of edges is:

$$[(0, 1), (0, 3), (1, 2), (2, 0), (3, 0)]$$

As adjacency matrix:

<i>source \ dest</i>	n_0	n_1	n_2	n_3
n_0	0	1	0	1
n_1	0	0	1	0
n_2	1	0	0	0
n_3	1	0	0	0

As adjacency list:

$$[\{1, 3\}, \{2\}, \{0\}, \{0\}]$$

1.3 Graph Prediction Tasks

What are the kinds of problems we want to solve on graphs?

The tasks fall into roughly three categories:

1. **Node Classification:** E.g. what is the topic of a paper given a citation network of papers?
2. **Link Prediction / Edge Classification:** E.g. are two people in a social network friends?
3. **Graph Classification:** E.g. is this protein molecule (represented as a graph) likely going to be effective?

The three main graph learning tasks. Image source: Petar Veličković.

Which examples of graph prediction tasks come to your mind? Which task types do they correspond to?

We will create and train models on all three task types in this tutorial.

2 Intro to the jraph Library

In the following sections, we will learn how to represent graphs and build GNNs in Python. We will use [jraph](#), a lightweight library for working with GNNs in [JAX](#).

2.1 Representing a graph in jraph

In `jraph`, a graph is represented with a `GraphsTuple` object. In addition to defining the graph structure of nodes and edges, you can also store node features, edge features and global graph features in a `GraphsTuple`.

In the `GraphsTuple`, edges are represented in two aligned arrays of node indices: senders (source nodes) and receivers (destination nodes). Each index corresponds to one edge, e.g. edge `i` goes from `senders[i]` to `receivers[i]`.

You can even store multiple graphs in one `GraphsTuple` object.

We will start with creating a simple directed graph with 4 nodes and 5 edges. We will also add toy features to the nodes, using `2*node_index` as the feature.

We will later use this toy graph in the GCN demo.

```
[178]: def build_toy_graph() -> jraph.GraphsTuple:
        """Define a four node graph, each node has a scalar as its feature."""

        # Nodes are defined implicitly by their features.
        # We will add four nodes, each with a feature, e.g.
        # node 0 has feature [0.],
        # node 1 has featre [2.] etc.
        # len(node_features) is the number of nodes.
```

```

node_features = jnp.array([[0.], [2.], [4.], [6.]])

# We will now specify 5 directed edges connecting the nodes we defined above.
# We define this with `senders` (source node indices) and `receivers`
# (destination node indices).
# For example, to add an edge from node 0 to node 1, we append 0 to senders,
# and 1 to receivers.
# We can do the same for all 5 edges:
# 0 -> 1
# 1 -> 2
# 2 -> 0
# 3 -> 0
# 0 -> 3
senders = jnp.array([0, 1, 2, 3, 0])
receivers = jnp.array([1, 2, 0, 0, 3])

# You can optionally add edge attributes to the 5 edges.
edges = jnp.array([[5.], [6.], [7.], [8.], [8.]])

# We then save the number of nodes and the number of edges.
# This information is used to make running GNNs over multiple graphs
# in a GraphsTuple possible.
n_node = jnp.array([4])
n_edge = jnp.array([5])

# Optionally you can add `global` information, such as a graph label.
global_context = jnp.array([[1]]) # Same feature dims as nodes and edges.
graph = jraph.GraphsTuple(
    nodes=node_features,
    edges=edges,
    senders=senders,
    receivers=receivers,
    n_node=n_node,
    n_edge=n_edge,
    globals=global_context
)
return graph

```

```
[179]: graph = build_toy_graph()
```

2.2 Inspecting the GraphsTuple

```

[180]: # Number of nodes
# Note that `n_node` returns an array. The length of `n_node` corresponds to
# the number of graphs stored in one `GraphsTuple`.
# In this case, we only have one graph, so n_node has length 1.
graph.n_node

```

```
[180]: Array([4], dtype=int32)
```

```
[181]: # Number of edges
graph.n_edge
```

```
[181]: Array([5], dtype=int32)
```

```
[182]: # Node features
graph.nodes
```

```
[182]: Array([[0.],
           [2.],
           [4.],
           [6.]], dtype=float32)
```

```
[183]: # Edge features
graph.edges
```

```
[183]: Array([[5.],
           [6.],
           [7.],
           [8.],
           [8.]], dtype=float32)
```

```
[184]: # Edges
graph.senders
```

```
[184]: Array([0, 1, 2, 3, 0], dtype=int32)
```

```
[185]: graph.receivers
```

```
[185]: Array([1, 2, 0, 0, 3], dtype=int32)
```

```
[186]: # Graph-level features
graph.globals
```

```
[186]: Array([[1]], dtype=int32)
```

2.3 Visualizing the Graph

To visualize the graph structure of the graph we created above, we will use the [networkx](#) library because it already has functions for drawing graphs.

We first convert the `jraph.GraphsTuple` to a `networkx.DiGraph`.

```
[187]: def convert_jraph_to_networkx_graph(jraph_graph: jraph.GraphsTuple) -> nx.Graph:
        nodes, edges, receivers, senders, _, _, _ = jraph_graph
        nx_graph = nx.DiGraph()
        if nodes is None:
```

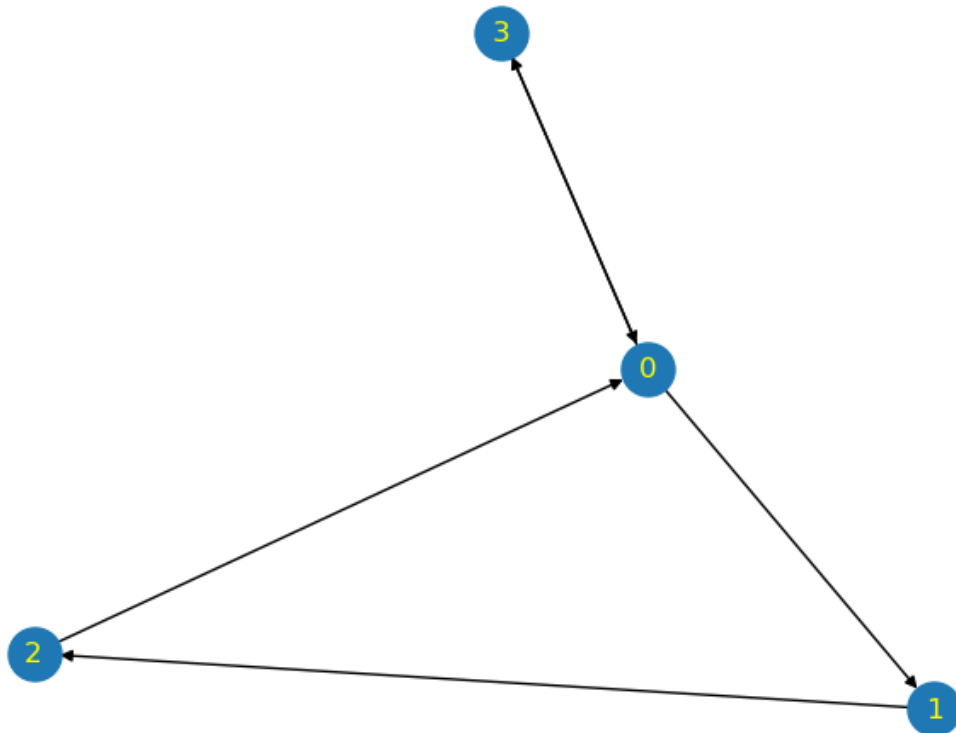
```

    for n in range(jraph_graph.n_node[0]):
        nx_graph.add_node(n)
    else:
        for n in range(jraph_graph.n_node[0]):
            nx_graph.add_node(n, node_feature=nodes[n])
    if edges is None:
        for e in range(jraph_graph.n_edge[0]):
            nx_graph.add_edge(int(senders[e]), int(receivers[e]))
    else:
        for e in range(jraph_graph.n_edge[0]):
            nx_graph.add_edge(
                int(senders[e]), int(receivers[e]), edge_feature=edges[e])
    return nx_graph

def draw_jraph_graph_structure(jraph_graph: jraph.GraphsTuple) -> None:
    nx_graph = convert_jraph_to_networkx_graph(jraph_graph)
    pos = nx.spring_layout(nx_graph)
    nx.draw(
        nx_graph, pos=pos, with_labels=True, node_size=500, font_color='yellow')

```

```
[188]: draw_jraph_graph_structure(graph)
```



3 Graph Convolutional Network (GCN) Layer

Now let's implement our first graph network!

The graph convolutional network, introduced by Kipf et al. (2017) in <https://arxiv.org/abs/1609.02907>, is one of the basic graph network architectures. We will build its core building block, the graph convolutional layer.

In a convolutional neural network (CNN), a convolutional filter (e.g. 3x3) is applied repeatedly to different parts of a larger input (e.g. 64x64) by striding across the input.

In a GCN, a convolution filter is applied to the neighbourhoods around a node in a graph.

However, there are also some differences to point out: In contrast to the CNN filter, the neighbourhoods in a GCN can be of different sizes, and there is no ordering of inputs. To see that, note that the CNN filter performs a weighted sum aggregation over the inputs with learnable weights, where each filter input has its own weight. In the GCN, the same weight is applied to all neighbours and the aggregation function is not learned. In other words, in a GCN, each neighbor contributes equally. This is why the CNN filter is not order-invariant, but the GCN filter is.

Comparison of CNN and GCN filters. Image source: <https://arxiv.org/pdf/1901.00596.pdf>

More specifically, the GCN layer performs two steps:

1. *Compute messages / update node features*: Create a feature vector $\vec{h}_n$ for each node n (e.g. with an MLP). This is going to be the message that this node will pass to neighboring nodes.
2. *Message-passing / aggregate node features*: For each node, calculate a new feature vector $\vec{h}'_n$ based on the messages (features) from the nodes in its neighborhood. In a directed graph, only nodes from incoming edges are counted as neighbors. The image below shows this aggregation step. There are multiple options for aggregation in a GCN, e.g. taking the mean, the sum, the min or max. (Later in this tutorial, we will also see how we can make the aggregation function dependent on the node features by adding an attention mechanism in the Graph Attention Network.)

"A generic overview of a graph convolution operation, highlighting the relevant information for deriving the next-level features for every node in the graph." Image source: Petar Veličković (<https://github.com/PetarV-/TikZ>)

Simple GCN Layer

```
[189]: def apply_simplified_gcn(graph: jraph.GraphsTuple) -> jraph.GraphsTuple:
    # Unpack GraphsTuple
    nodes, _, receivers, senders, _, _ = graph

    # 1. Update node features
    # For simplicity, we will first use an identify function here, and replace it
    # with a trainable MLP block later.
    update_node_fn = lambda nodes: nodes
    nodes = update_node_fn(nodes)
```

```

# 2. Aggregate node features over nodes in neighborhood
# Equivalent to jnp.sum(n_node), but jittable
total_num_nodes = tree.tree_leaves(nodes)[0].shape[0]
aggregate_nodes_fn = jax.ops.segment_sum

# Compute new node features by aggregating messages from neighboring nodes
nodes = tree.tree_map(lambda x: aggregate_nodes_fn(x[senders], receivers,
                                                    total_num_nodes), nodes)

out_graph = graph._replace(nodes=nodes)
return out_graph

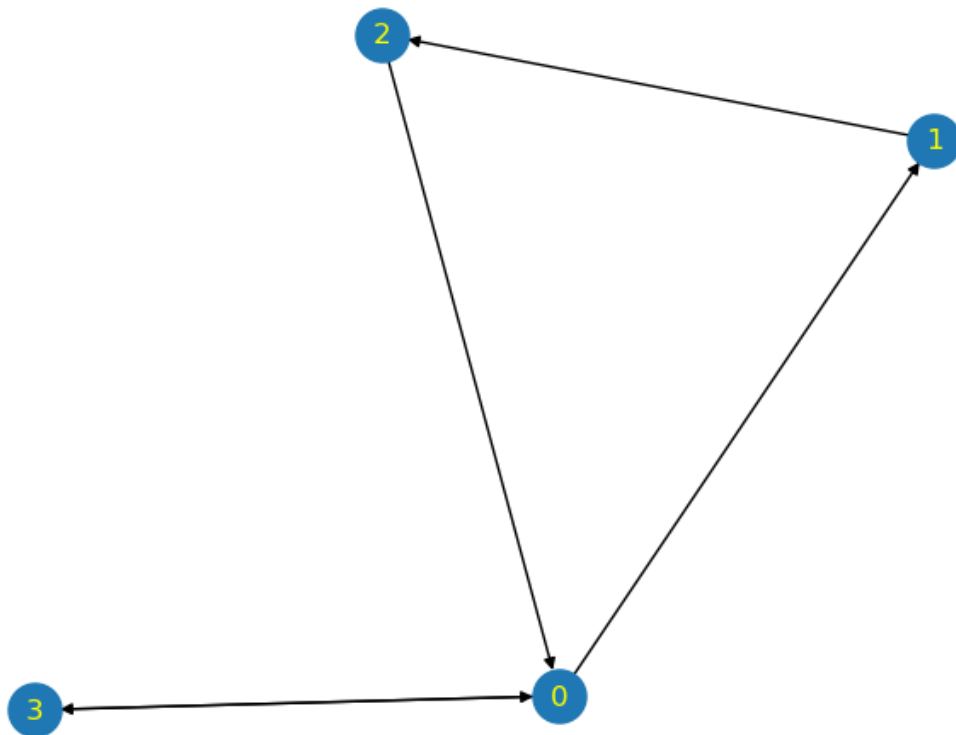
```

We can now run the graph convolution on our toy graph from before.

```
[190]: graph = build_toy_graph()
```

Here is the visualized graph.

```
[191]: draw_jraph_graph_structure(graph)
```



```
[192]: out_graph = apply_simplified_gcn(graph)
```

Since we used the identity function for updating nodes and sum aggregation, we can verify the results pretty easily. As a reminder, in this toy graph, the node features are the same as the node index.

Node 0: sum of features from node 2 and node 3 $\rightarrow$ 10.

Node 1: sum of features from node 0 $\rightarrow$ 0.

Node 2: sum of features from node 1 $\rightarrow$ 2.

Node 3: sum of features from node 0 $\rightarrow$ 0.

```
[193]: out_graph.nodes
```

```
[193]: Array([[10.],
           [ 0.],
           [ 2.],
           [ 0.]], dtype=float32)
```

Add Trainable Parameters to GCN layer So far our graph convolution operation doesn't have any learnable parameters. Let's add an MLP block to the update function to make it trainable.

```
[194]: class MLP(hk.Module):
    def __init__(self, features: jnp.ndarray):
        super().__init__()
        self.features = features

    def __call__(self, x: jnp.ndarray) -> jnp.ndarray:
        layers = []
        for feat in self.features[:-1]:
            layers.append(hk.Linear(feat))
            layers.append(jax.nn.relu)
            layers.append(hk.Linear(self.features[-1]))

        mlp = hk.Sequential(layers)
        return mlp(x)

# Use MLP block to define the update node function
update_node_fn = lambda x: MLP(features=[8, 4])(x)
```

Check outputs of update_node_fn with MLP Block

```
[195]: graph = build_toy_graph()
```

```
[196]: update_node_module = hk.without_apply_rng(hk.transform(update_node_fn))
params = update_node_module.init(jax.random.PRNGKey(42), graph.nodes)
out = update_node_module.apply(params, graph.nodes)
```

As output, we expect the updated node features. We should see one array of dim 4 for each of the 4 nodes, which is the result of applying a single MLP block to the features of each node individually.

[197]: out

```
[197]: Array([[ 0.          ,  0.          ,  0.          ,  0.          ],
              [-0.14526576,  0.3303815 , -0.50791115, -0.64371586],
              [-0.29053152,  0.660763  , -1.0158223 , -1.2874317 ],
              [-0.4357972 ,  0.9911446 , -1.5237335 , -1.9311478 ]],
          dtype=float32)
```

Add Self-Edges (Edges connecting a node to itself) For each node, add an edge of the node onto itself. This way, nodes will include themselves in the aggregation step.

```
[198]: def add_self_edges_fn(receivers: jnp.ndarray, senders: jnp.ndarray,
                             total_num_nodes: int) -> Tuple[jnp.ndarray, jnp.ndarray]:
    """Adds self edges. Assumes self edges are not in the graph yet."""
    receivers = jnp.concatenate((receivers, jnp.arange(total_num_nodes)), axis=0)
    senders = jnp.concatenate((senders, jnp.arange(total_num_nodes)), axis=0)
    return receivers, senders
```

Add Symmetric Normalization Note that the nodes may have different numbers of neighbors / degrees. This could lead to instabilities during neural network training, e.g. exploding or vanishing gradients. To address that, normalization is a commonly used method. In this case, we will normalize by node degrees.

As a first attempt, we could count the number of incoming edges (including self-edge) and divide by that value.

More formally, let A be the adjacency matrix defining the edges of the graph.

Then we define the degree matrix D as a diagonal matrix with $D_{ii} = \sum_j A_{ij}$ (the degree of node i)

Now we can normalize AH by dividing it by the node degrees:

$$D^{-1}AH$$

To take both the in and out degrees into account, we can use symmetric normalization, which is also what Kipf and Welling proposed in their [paper](#):

$$D^{-\frac{1}{2}}AD^{-\frac{1}{2}}H$$

General GCN Layer Now we can write a more general and configurable version of the Graph Convolution layer, allowing the caller to specify:

- **update_node_fn**: Function to use to update node features (e.g. the MLP block version we just implemented)
- **aggregate_nodes_fn**: Aggregation function to use to aggregate messages from neighbourhood.
- **add_self_edges**: Whether to add self edges for aggregation step.
- **symmetric_normalization**: Whether to add symmetric normalization.

```
[199]: # Adapted from https://github.com/deepmind/jraph/blob/master/jraph/_src/models.
        ↪py#L506
def GraphConvolution(update_node_fn: Callable,
                     aggregate_nodes_fn: Callable = jax.ops.segment_sum,
                     add_self_edges: bool = False,
                     symmetric_normalization: bool = True) -> Callable:
    """Returns a method that applies a Graph Convolution layer.

    Graph Convolutional layer as in https://arxiv.org/abs/1609.02907,
    NOTE: This implementation does not add an activation after aggregation.
    If you are stacking layers, you may want to add an activation between
    each layer.

    Args:
        update_node_fn: function used to update the nodes. In the paper a single
            layer MLP is used.
        aggregate_nodes_fn: function used to aggregates the sender nodes.
        add_self_edges: whether to add self edges to nodes in the graph as in the
            paper definition of GCN. Defaults to False.
        symmetric_normalization: whether to use symmetric normalization. Defaults to
            True.

    Returns:
        A method that applies a Graph Convolution layer.
    """

def _ApplyGCN(graph: jraph.GraphsTuple) -> jraph.GraphsTuple:
    """Applies a Graph Convolution layer."""
    nodes, _, receivers, senders, _, _, _ = graph

    # First pass nodes through the node updater.
    nodes = update_node_fn(nodes)
    # Equivalent to jnp.sum(n_node), but jittable
    total_num_nodes = tree.tree_leaves(nodes)[0].shape[0]
    if add_self_edges:
        # We add self edges to the senders and receivers so that each node
        # includes itself in aggregation.
        # In principle, a `GraphsTuple` should partition by n_edge, but in
        # this case it is not required since a GCN is agnostic to whether
        # the `GraphsTuple` is a batch of graphs or a single large graph.
        conv_receivers, conv_senders = add_self_edges_fn(receivers, senders,
                                                         total_num_nodes)
    else:
        conv_senders = senders
        conv_receivers = receivers

    # pylint: disable=g-long-lambda
    if symmetric_normalization:
```

```

# Calculate the normalization values.
count_edges = lambda x: jax.ops.segment_sum(
    jnp.ones_like(conv_senders), x, total_num_nodes)
sender_degree = count_edges(conv_senders)
receiver_degree = count_edges(conv_receivers)

# Pre normalize by sqrt sender degree.
# Avoid dividing by 0 by taking maximum of (degree, 1).
nodes = tree.tree_map(
    lambda x: x * jax.lax.rsqrt(jnp.maximum(sender_degree, 1.0))[:, None],
    nodes,
)
# Aggregate the pre-normalized nodes.
nodes = tree.tree_map(
    lambda x: aggregate_nodes_fn(x[conv_senders], conv_receivers,
                                total_num_nodes), nodes)

# Post normalize by sqrt receiver degree.
# Avoid dividing by 0 by taking maximum of (degree, 1).
nodes = tree.tree_map(
    lambda x:
        (x * jax.lax.rsqrt(jnp.maximum(receiver_degree, 1.0))[:, None]),
    nodes,
)
else:
    nodes = tree.tree_map(
        lambda x: aggregate_nodes_fn(x[conv_senders], conv_receivers,
                                    total_num_nodes), nodes)

# pylint: enable=g-long-lambda
return graph._replace(nodes=nodes)

return _ApplyGCN

```

Test General GCN Layer

```

[200]: gcn_layer = GraphConvolution(
    update_node_fn=lambda n: MLP(features=[8, 4])(n),
    aggregate_nodes_fn=jax.ops.segment_sum,
    add_self_edges=True,
    symmetric_normalization=True
)

```

```

[201]: graph = build_toy_graph()
network = hk.without_apply_rng(hk.transform(gcn_layer))
params = network.init(jax.random.PRNGKey(42), graph)
out_graph = network.apply(params, graph)
out_graph.nodes

```

```
[201]: Array([[ -0.29652244,  0.67438847, -1.0367693 , -1.3139795 ],
              [ -0.07263287,  0.16519076, -0.25395554, -0.32185793],
              [ -0.2178986 ,  0.49557224, -0.7618666 , -0.9655737 ],
              [ -0.21789859,  0.4955723 , -0.76186675, -0.9655739 ]],
              dtype=float32)
```

Build GCN Model with Multiple Layers With a single GCN layer, a node's representation after the GCN layer is only influenced by its direct neighbourhood. However, we may want to consider larger neighbourhoods, i.e. more than just 1 hop away. To achieve that, we can stack multiple GCN layers, similar to how stacking CNN layers expands the input region.

We will define a network with three GCN layers:

```
[202]: def gcn(graph: jraph.GraphsTuple) -> jraph.GraphsTuple:
        """Defines a graph neural network with 3 GCN layers.
        Args:
            graph: GraphsTuple the network processes.

        Returns:
            output graph with updated node values.
        """
        gn = GraphConvolution(
            update_node_fn=lambda n: jax.nn.relu(hk.Linear(8)(n)),
            add_self_edges=True)
        graph = gn(graph)

        gn = GraphConvolution(
            update_node_fn=lambda n: jax.nn.relu(hk.Linear(4)(n)),
            add_self_edges=True)
        graph = gn(graph)

        gn = GraphConvolution(
            update_node_fn=hk.Linear(2))
        graph = gn(graph)
        return graph
```

```
[203]: graph = build_toy_graph()
        network = hk.without_apply_rng(hk.transform(gcn))
        params = network.init(jax.random.PRNGKey(42), graph)
        out_graph = network.apply(params, graph)
        out_graph.nodes
```

```
[203]: Array([[ -0.3238613 ,  0.10707339],
              [ -0.23884149,  0.07896457],
              [ -0.19206831,  0.06350066],
              [ -0.23884149,  0.07896457]], dtype=float32)
```

Node Classification with GCN on Karate Club Dataset Time to try out our GCN on our first graph prediction task!

Zachary's Karate Club Dataset [Zachary's karate club](#) is a small dataset commonly used as an example for a social graph. It's great for demo purposes, as it's easy to visualize and quick to train a model on it.

A node represents a student or instructor in the club. An edge means that those two people have interacted outside of the class. There are two instructors in the club.

Each student is assigned to one of two instructors.

Optimizing the GCN on the Karate Club Node Classification Task The task is to predict the assignment of students to instructors, given the social graph and only knowing the assignment of two nodes (the two instructors) a priori.

In other words, out of the 34 nodes, only two nodes are labeled, and we are trying to optimize the assignment of the other 32 nodes, by **maximizing the log-likelihood of the two known node assignments**.

We will compute the accuracy of our node assignments by comparing to the ground-truth assignments. **Note that the ground-truth for the 32 student nodes is not used in the loss function itself.**

Let's load the dataset:

```
[204]: """Zachary's karate club example.  
From https://github.com/deepmind/jraph/blob/master/jraph/examples/  
→zacharys_karate_club.py.  
Here we train a graph neural network to process Zachary's karate club.  
https://en.wikipedia.org/wiki/Zachary%27s_karate_club  
Zachary's karate club is used in the literature as an example of a social graph.  
Here we use a graphnet to optimize the assignments of the students in the  
karate club to two distinct karate instructors (Mr. Hi and John A).  
"""  
  
def get_zacharys_karate_club() -> jraph.GraphsTuple:  
    """Returns GraphsTuple representing Zachary's karate club."""  
    social_graph = [  
        (1, 0), (2, 0), (2, 1), (3, 0), (3, 1), (3, 2),  
        (4, 0), (5, 0), (6, 0), (6, 4), (6, 5), (7, 0), (7, 1),  
        (7, 2), (7, 3), (8, 0), (8, 2), (9, 2), (10, 0), (10, 4),  
        (10, 5), (11, 0), (12, 0), (12, 3), (13, 0), (13, 1), (13, 2),  
        (13, 3), (16, 5), (16, 6), (17, 0), (17, 1), (19, 0), (19, 1),  
        (21, 0), (21, 1), (25, 23), (25, 24), (27, 2), (27, 23),  
        (27, 24), (28, 2), (29, 23), (29, 26), (30, 1), (30, 8),  
        (31, 0), (31, 24), (31, 25), (31, 28), (32, 2), (32, 8),  
        (32, 14), (32, 15), (32, 18), (32, 20), (32, 22), (32, 23),  
        (32, 29), (32, 30), (32, 31), (33, 8), (33, 9), (33, 13),  
        (33, 14), (33, 15), (33, 18), (33, 19), (33, 20), (33, 22),
```

```

        (33, 23), (33, 26), (33, 27), (33, 28), (33, 29), (33, 30),
        (33, 31), (33, 32)]
    # Add reverse edges.
    social_graph += [(edge[1], edge[0]) for edge in social_graph]
    n_club_members = 34

    return jraph.GraphsTuple(
        n_node=jnp.asarray([n_club_members]),
        n_edge=jnp.asarray([len(social_graph)]),
        # One-hot encoding for nodes, i.e. argmax(nodes) = node index.
        nodes=jnp.eye(n_club_members),
        # No edge features.
        edges=None,
        globals=None,
        senders=jnp.asarray([edge[0] for edge in social_graph]),
        receivers=jnp.asarray([edge[1] for edge in social_graph]))

def get_ground_truth_assignments_for_zacharys_karate_club() -> jnp.ndarray:
    """Returns ground truth assignments for Zachary's karate club."""
    return jnp.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1,
                      0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])

```

```
[205]: graph = get_zacharys_karate_club()
```

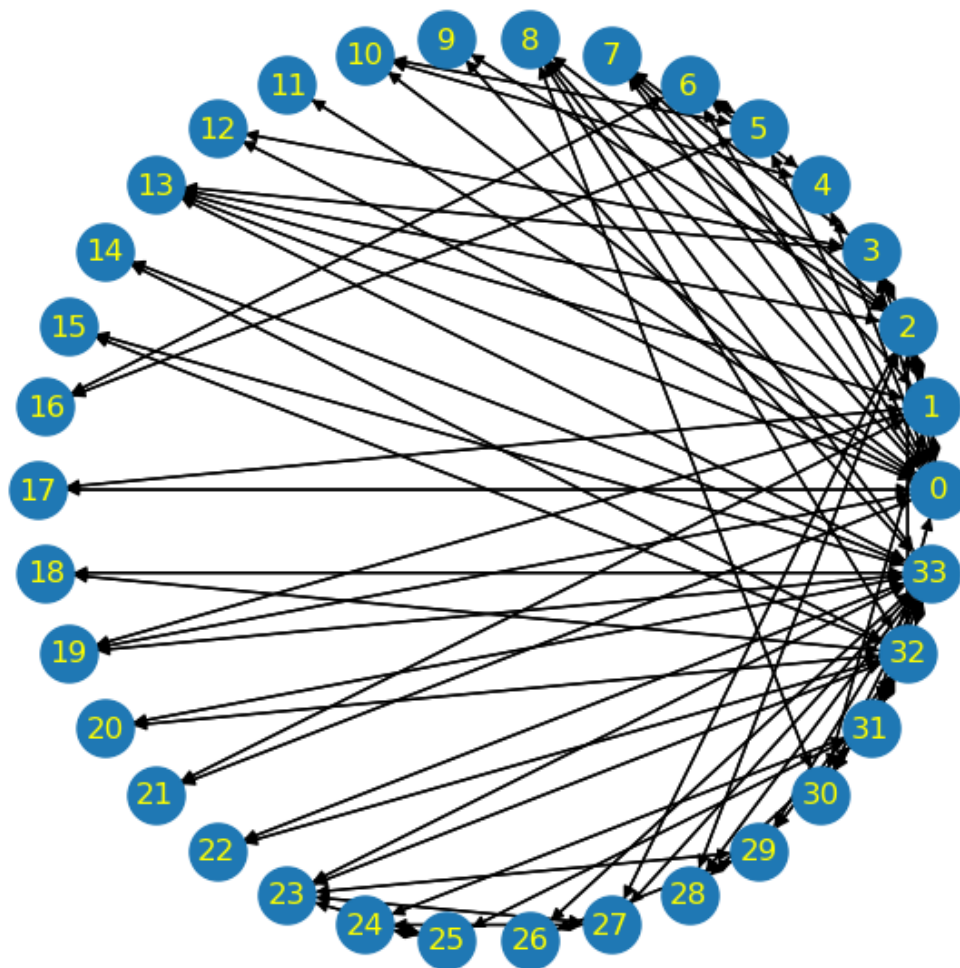
```
[206]: print(f'Number of nodes: {graph.n_node[0]}')
       print(f'Number of edges: {graph.n_edge[0]}')
```

Number of nodes: 34

Number of edges: 156

Visualize the karate club graph with circular node layout:

```
[207]: nx_graph = convert_jraph_to_networkx_graph(graph)
       pos = nx.circular_layout(nx_graph)
       plt.figure(figsize=(6, 6))
       nx.draw(nx_graph, pos=pos, with_labels = True, node_size=500,
               ↪font_color='yellow')
```



Define the GCN with the GraphConvolution layers we implemented:

```
[208]: def gcn_definition(graph: jraph.GraphsTuple) -> jraph.GraphsTuple:
    """Defines a GCN for the karate club task.
    Args:
        graph: GraphsTuple the network processes.

    Returns:
        output graph with updated node values.
    """
    gn = GraphConvolution(
        update_node_fn=lambda n: jax.nn.relu(hk.Linear(8)(n)),
        add_self_edges=True)
```

```

graph = gn(graph)

## Ajout personel : ajout d'une couche de conv avec relu
gn = GraphConvolution(
    update_node_fn=lambda n: jax.nn.relu(hk.Linear(4)(n)),
    add_self_edges=True)
graph = gn(graph)

gn = GraphConvolution(
    update_node_fn=hk.Linear(2)) # output dim is 2 because we have 2 output_
→ classes.
graph = gn(graph)
return graph

```

Training and evaluation code:

```

[209]: def optimize_club(network: hk.Transformed, num_steps: int) -> jnp.ndarray:
    """Solves the karate club problem by optimizing the assignments of students."""
    zacharys_karate_club = get_zacharys_karate_club()
    labels = get_ground_truth_assignments_for_zacharys_karate_club()
    params = network.init(jax.random.PRNGKey(42), zacharys_karate_club)

    @jax.jit
    def predict(params: hk.Params) -> jnp.ndarray:
        decoded_graph = network.apply(params, zacharys_karate_club)
        return jnp.argmax(decoded_graph.nodes, axis=1)

    @jax.jit
    def prediction_loss(params: hk.Params) -> jnp.ndarray:
        decoded_graph = network.apply(params, zacharys_karate_club)
        # We interpret the decoded nodes as a pair of logits for each node.
        log_prob = jax.nn.log_softmax(decoded_graph.nodes)
        # The only two assignments we know a-priori are those of Mr. Hi (Node 0)
        # and John A (Node 33).
        return -(log_prob[0, 0] + log_prob[33, 1])

    opt_init, opt_update = optax.adam(1e-2)
    opt_state = opt_init(params)

    @jax.jit
    def update(params: hk.Params, opt_state) -> Tuple[hk.Params, Any]:
        """Returns updated params and state."""
        g = jax.grad(prediction_loss)(params)
        updates, opt_state = opt_update(g, opt_state)
        return optax.apply_updates(params, updates), opt_state

    @jax.jit

```

```
def accuracy(params: hk.Params) -> jnp.ndarray:
    decoded_graph = network.apply(params, zacharys_karate_club)
    return jnp.mean(jnp.argmax(decoded_graph.nodes, axis=1) == labels)

for step in range(num_steps):
    print(f"step {step} accuracy {accuracy(params).item():.2f}")
    params, opt_state = update(params, opt_state)

return predict(params)
```

Let's train the GCN! We expect this model reach an accuracy of about 0.91.

```
[210]: network = hk.without_apply_rng(hk.transform(gcn_definition))
result_gcn = optimize_club(network, num_steps=15)
```

```
step 0 accuracy 0.50
step 1 accuracy 0.65
step 2 accuracy 0.97
step 3 accuracy 0.94
step 4 accuracy 0.94
step 5 accuracy 0.94
step 6 accuracy 0.94
step 7 accuracy 0.94
step 8 accuracy 0.97
step 9 accuracy 0.94
step 10 accuracy 0.94
step 11 accuracy 0.94
step 12 accuracy 0.94
step 13 accuracy 0.97
step 14 accuracy 0.97
```

Try modifying the model parameters to see if you can improve the accuracy!

You can also modify the dataset itself, and see how that influences model training.

Node assignments predicted by the model at the end of training:

```
[211]: result_gcn
```

```
[211]: Array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0,
            1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], dtype=int32)
```

Visualize ground truth and predicted node assignments:

What do you think of the results?

```
[212]: zacharys_karate_club = get_zacharys_karate_club()
nx_graph = convert_jraph_to_networkx_graph(zacharys_karate_club)
pos = nx.circular_layout(nx_graph)

fig = plt.figure(figsize=(15, 7))
```

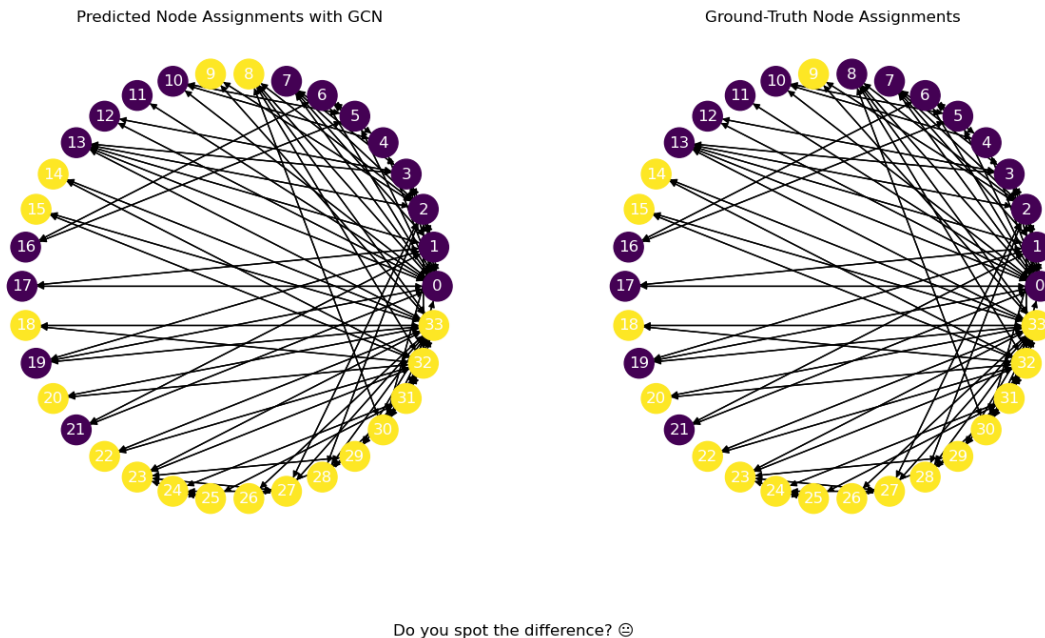
```

ax1 = fig.add_subplot(121)
nx.draw(
    nx_graph,
    pos=pos,
    with_labels=True,
    node_size=500,
    node_color=result_gcn.tolist(),
    font_color='white')
ax1.title.set_text('Predicted Node Assignments with GCN')

gt_labels = get_ground_truth_assignments_for_zacharys_karate_club()
ax2 = fig.add_subplot(122)
nx.draw(
    nx_graph,
    pos=pos,
    with_labels=True,
    node_size=500,
    node_color=gt_labels.tolist(),
    font_color='white')
ax2.title.set_text('Ground-Truth Node Assignments')
fig.suptitle('Do you spot the difference? ', y=-0.01)

plt.show()

```



Questions Q1.1: Implement the iterative label propagation (ILP) algorithm from the SSL classes using GCN (hint: it can be expressed as chain of GCN layers, but remember to symmetrize the

graph first), and motivate/describe your choices in the implementation.

The objective of ILP is to propagate, in an iterative way, the known labels (or other node information) across the graph. Concretely, this amounts to repeatedly applying the following operation :

$$Y = D^{-\frac{1}{2}} A D^{-\frac{1}{2}} H$$

And by iterating k times, we obtain :

$$Y^{(k+1)} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}} Y^{(k)}$$

where Y is the matrix of node features.

To implement this update in a **GCN**, we remove all learning components in order to recover the same formula:

1. No parametric transformation : we define (as in the first example of a GCN layer given above) `update_node_fn=lambda nodes: nodes`, i.e. the identity. This way, there are no weights or biases, unlike in a standard GCN.
2. Symmetric normalization : by setting `symmetric_normalization=True` in `GraphConvolution`, we enforce the computation of $D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$.
3. We then apply the same layer multiple times in a row (a loop of `num_iter` iterations) to reproduce the iterative diffusion described above.

In practice, we start from a one-hot encoded label vector (since there are only 2 known classes). At each iteration, the values are “mixed” according to the graph structure and the matrix A , and we finally recover the class prediction by simply taking the argmax over the final distributions.

```
[213]: def iterative_label_propagation_definition(num_iter: int) -> Callable:
        """
        Returns a parameter-free function that applies the label propagation
        update rule num_iter times, modeled as repeated GCN layers without weights.
        """
        def ilp_model(graph: jraph.GraphsTuple) -> jraph.GraphsTuple:
            # Repeatedly apply a non-parametric GCN layer
            for _ in range(num_iter):
                gcn_layer = GraphConvolution(
                    update_node_fn=lambda nodes: nodes, # Identity function (no
                    ↪parameters)
                    add_self_edges=True,
                    symmetric_normalization=True
                )
                graph = gcn_layer(graph)
            return graph

        return ilp_model
```

```

def apply_ILP(num_iter: int) -> jnp.ndarray:
    """
    Applies iterative label propagation on the Zachary's karate club graph.
    """
    graph = get_zacharys_karate_club()
    labels = get_ground_truth_assignments_for_zacharys_karate_club()

    num_nodes = graph.nodes.shape[0]
    one_hot_labels = jnp.zeros((num_nodes, 2))
    one_hot_labels = one_hot_labels.at[0, 0].set(1.0)    # Mr. Hi → class 0
    one_hot_labels = one_hot_labels.at[33, 1].set(1.0)   # John A → class 1

    graph = graph._replace(nodes=one_hot_labels)

    # Transform the ILP model using Haiku (no parameters to train)
    ilp_transformed = hk.without_apply_rng(
        hk.transform(iterative_label_propagation_definition(num_iter))
    )

    # Initialize (empty) parameters and apply the ILP model
    params = ilp_transformed.init(jax.random.PRNGKey(42), graph)
    updated_graph = ilp_transformed.apply(params, graph)

    # Predict the class by taking the argmax of the updated node features
    predictions = jnp.argmax(updated_graph.nodes, axis=1)
    return predictions

```

```

[214]: graph = get_zacharys_karate_club()
result_ilp = apply_ILP(num_iter=3)

nx_graph = convert_jraph_to_networkx_graph(graph)
pos = nx.circular_layout(nx_graph)

fig = plt.figure(figsize=(15, 7))

# ILP prediction
ax1 = fig.add_subplot(121)
nx.draw(
    nx_graph,
    pos=pos,
    with_labels=True,
    node_size=500,
    node_color=result_ilp.tolist(),
    font_color='white')
ax1.title.set_text('ILP Predictions (3 iterations)')

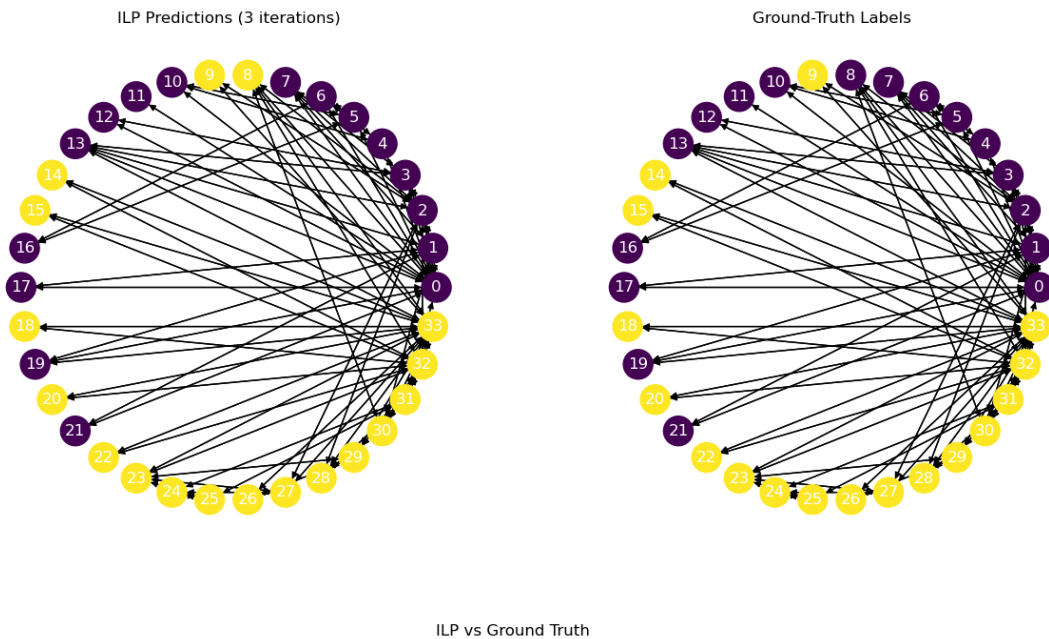
```

```

# Ground truth labels
gt_labels = get_ground_truth_assignments_for_zacharys_karate_club()
ax2 = fig.add_subplot(122)
nx.draw(
    nx_graph,
    pos=pos,
    with_labels=True,
    node_size=500,
    node_color=gt_labels.tolist(),
    font_color='white')
ax2.title.set_text('Ground-Truth Labels')

fig.suptitle('ILP vs Ground Truth', y=-0.01)
plt.show()

```



Q1.2: Compare accuracy of ILP and a GCN with MLP feature extraction on the Karate club dataset. (remember that $\{0,1\}$ labels do not have a meaning for ILP)

```

[215]: zacharys_karate_club = get_zacharys_karate_club()
nx_graph = convert_jraph_to_networkx_graph(zacharys_karate_club)
pos = nx.circular_layout(nx_graph)

fig = plt.figure(figsize=(15, 7))
ax1 = fig.add_subplot(131)
nx.draw(
    nx_graph,

```

```

        pos=pos,
        with_labels=True,
        node_size=500,
        node_color=result_gcn.tolist(),
        font_color='white')
ax1.title.set_text('Predicted Node Assignments with GCN')

gt_labels = get_ground_truth_assignments_for_zacharys_karate_club()
ax2 = fig.add_subplot(132)
nx.draw(
    nx_graph,
    pos=pos,
    with_labels=True,
    node_size=500,
    node_color=result_ilp.tolist(),
    font_color='white')
ax2.title.set_text('Predicted Node Assignments with ILP')

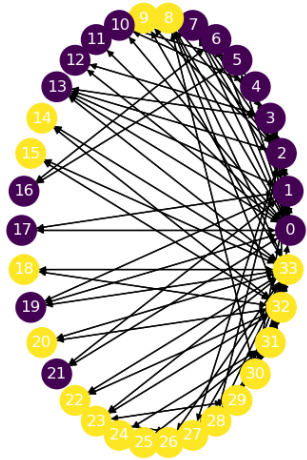
gt_labels = get_ground_truth_assignments_for_zacharys_karate_club()
ax3 = fig.add_subplot(133)
nx.draw(
    nx_graph,
    pos=pos,
    with_labels=True,
    node_size=500,
    node_color=gt_labels.tolist(),
    font_color='white')
ax3.title.set_text('Ground-Truth Node Assignments')

fig.suptitle('(almost) no difference here !', y=-0.01)

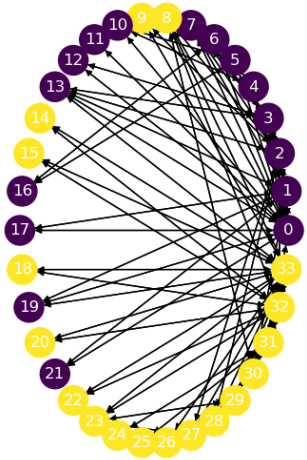
plt.show()

```

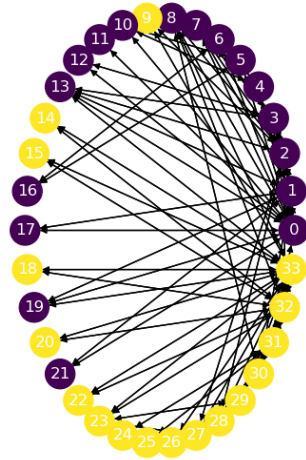
Predicted Node Assignments with GCN



Predicted Node Assignments with ILP



Ground-Truth Node Assignments



(almost) no difference here !

```
[216]: import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score

# Ground-truth and GCN predictions
gt_list = gt_labels.tolist()
gcn_list = result_gcn.tolist()

# Evaluate ILP for all k from 0 to 50
iteration_values = list(range(0, 51))
ilp_accuracies = []

for k in iteration_values:
    pred_ilp = apply_ILP(k).tolist()
    acc = accuracy_score(onp.array(pred_ilp), onp.array(gt_list))
    ilp_accuracies.append(acc)

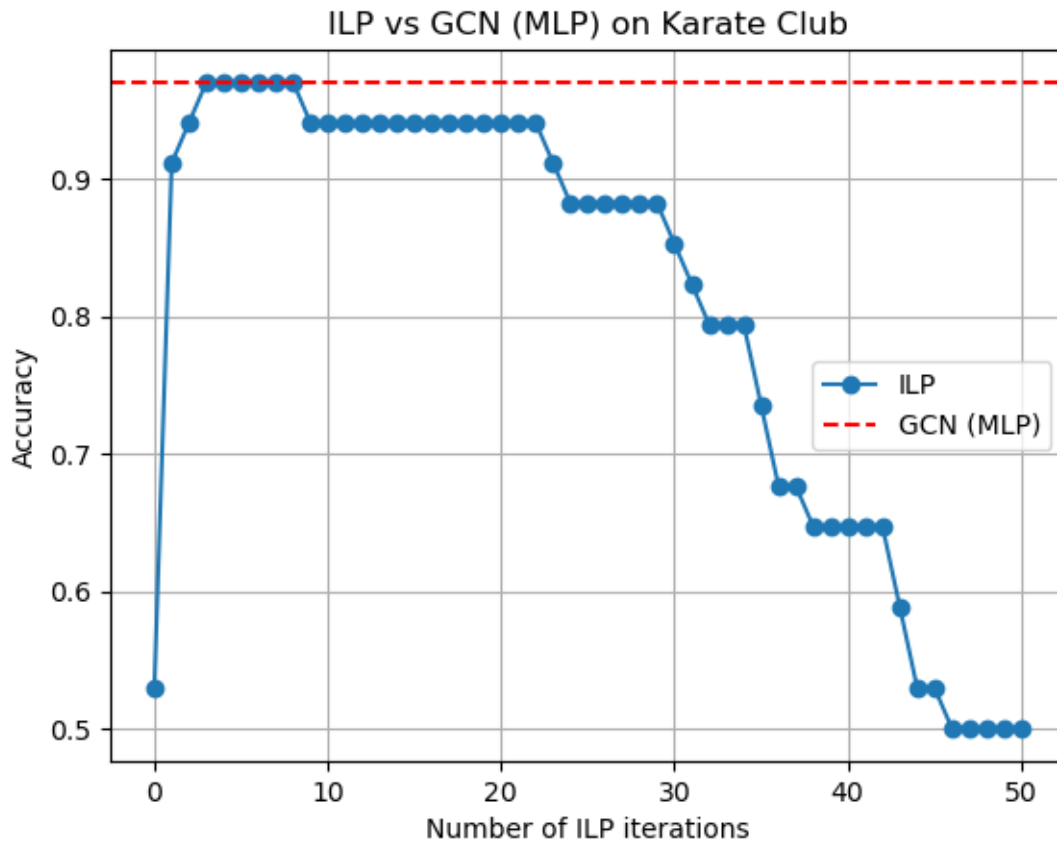
# GCN accuracy
acc_gcn = accuracy_score(onp.array(gcn_list), onp.array(gt_list))

for k, acc in zip(iteration_values, ilp_accuracies):
    if k in [1, 2, 3, 5, 10, 46, 50]:
        print(f"ILP with {k} iterations → accuracy = {acc:.3f}")
print(f"GCN with MLP → accuracy = {acc_gcn:.3f}")

plt.plot(iteration_values, ilp_accuracies, marker='o', label='ILP')
plt.axhline(y=acc_gcn, color='r', linestyle='--', label='GCN (MLP)')
```

```
plt.xlabel("Number of ILP iterations")
plt.ylabel("Accuracy")
plt.title("ILP vs GCN (MLP) on Karate Club")
plt.legend()
plt.grid(True)
plt.show()
```

ILP with 1 iterations → accuracy = 0.912
 ILP with 2 iterations → accuracy = 0.941
 ILP with 3 iterations → accuracy = 0.971
 ILP with 5 iterations → accuracy = 0.971
 ILP with 10 iterations → accuracy = 0.941
 ILP with 46 iterations → accuracy = 0.500
 ILP with 50 iterations → accuracy = 0.500
 GCN with MLP → accuracy = 0.971



ILP vs GCN (MLP)

This experiment compares the accuracy of ILP and a GCN with MLP on the Karate Club dataset as the number of ILP iterations increases from 0 to 50.

With only 3 to 8 iterations, ILP reaches the **same accuracy as the GCN with 3 layers** (8 → 4

$\rightarrow 2$), i.e. **0.97**, with **only one identical mistake on node 8**. The first GCN with 2 layers ($8 \rightarrow 2$) gives an accuracy of **0.91**.

However, after a certain limit (around **46 iterations**), ILP tends to **collapse into a single class**, reducing accuracy. This happens because labels are not re-injected at each step, so the propagation eventually converges to a uniform state.

In summary, ILP is a **simple and fast** method that performs very well with a few labeled nodes, but choosing a reasonable number of iterations is crucial. GCNs offer **higher robustness**, at the cost of training.

4 Graph Attention (GAT) Layer

While the GCN we covered in the previous section can learn meaningful representations, it also has some shortcomings. Can you think of any?

In the GCN layer, the messages from all its neighbours and the node itself are equally weighted. This may lead to loss of node-specific information. E.g., consider the case when a set of nodes shares the same set of neighbors, and start out with different node features. Then because of averaging, their resulting output features would be the same. Adding self-edges mitigates this issue by a small amount, but this problem is magnified with increasing number of GCN layers and number of edges connecting to a node.

The graph attention (GAT) mechanism, as proposed by [Velickovic et al. \(2017\)](#), allows the network to learn how to weigh / assign importance to the node features from the neighbourhood when computing the new node features. This is very similar to the idea of using attention in Transformers, which were introduced in [Vaswani et al. \(2017\)](#).

(One could even argue that Transformers are graph attention networks operating on the special case of fully-connected graphs.)

In the figure below, $\vec{h}$ are the node features and $\vec{\alpha}$ are the learned attention weights.

Figure Credit: [Velickovic et al. \(2017\)](#). (Detail: This image is showing multi-headed attention with 3 heads, each color corresponding to a different head. At the end, an aggregation function is applied over all the heads.)

To obtain the output node features of the GAT layer, we compute:

$$\vec{h}'_i = \sum_{j \in \mathcal{N}(i)} \alpha_{ij} \mathbf{W} \vec{h}_j$$

Here, $\mathbf{W}$ is a weight matrix which performs a linear transformation on the input.

How do we obtain α , or in other words, learn what to pay attention to?

Intuitively, the attention coefficient α_{ij} should rely on both the transformed features from nodes i and j . So let's first define an attention mechanism function `attention_fn` that computes the intermediary attention coefficients e_{ij} :

$$e_{ij} = \text{attention_fn}(\mathbf{W} \vec{h}_i, \mathbf{W} \vec{h}_j)$$

To obtain normalized attention weights α , we apply a softmax:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j \in \mathcal{N}(i)} \exp(e_{ij})}$$

For the function a , the authors of the GAT paper chose to concatenate the transformed node features (denoted by $||$) and apply a single-layer feedforward network, parameterized by a weight vector $\vec{a}$ and with LeakyRelu as non-linearity.

In the implementation below, we refer to $\mathbf{W}$ as `attention_query_fn` and `att_fn` as `attention_logit_fn`.

$$\text{attention_fn}(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j) = \text{LeakyReLU}(\vec{a}(\mathbf{W}\vec{h}_i || \mathbf{W}\vec{h}_j))$$

The figure below summarizes this attention mechanism visually.

Figure Credit: Petar Velickovic.

```
[217]: # GAT implementation adapted from https://github.com/deepmind/jraph/blob/master/
↳ jraph/_src/models.py#L442.
def GAT(attention_query_fn: Callable,
        attention_logit_fn: Callable,
        node_update_fn: Optional[Callable] = None,
        add_self_edges: bool = True) -> Callable:
    """Returns a method that applies a Graph Attention Network layer.

    Graph Attention message passing as described in
    https://arxiv.org/pdf/1710.10903.pdf. This model expects node features as a
    jnp.array, may use edge features for computing attention weights, and
    ignore global features. It does not support nests.
    Args:
        attention_query_fn: function that generates attention queries from sender
            node features.
        attention_logit_fn: function that converts attention queries into logits for
            softmax attention.
        node_update_fn: function that updates the aggregated messages. If None, will
            apply leaky relu and concatenate (if using multi-head attention).

    Returns:
        A function that applies a Graph Attention layer.
    """
    # pylint: disable=g-long-lambda
    if node_update_fn is None:
        # By default, apply the leaky relu and then concatenate the heads on the
        # feature axis.
        node_update_fn = lambda x: jnp.reshape(
            jax.nn.leaky_relu(x), (x.shape[0], -1))

    def _ApplyGAT(graph: jraph.GraphsTuple) -> jraph.GraphsTuple:
```

```

"""Applies a Graph Attention layer."""
nodes, edges, receivers, senders, _, _, _ = graph
# Equivalent to the sum of n_node, but statically known.
try:
    sum_n_node = nodes.shape[0]
except IndexError:
    raise IndexError('GAT requires node features')

# Pass nodes through the attention query function to transform
# node features, e.g. with an MLP.
nodes = attention_query_fn(nodes)

total_num_nodes = tree.tree_leaves(nodes)[0].shape[0]
if add_self_edges:
    # We add self edges to the senders and receivers so that each node
    # includes itself in aggregation.
    receivers, senders = add_self_edges_fn(receivers, senders,
                                           total_num_nodes)

# We compute the softmax logits using a function that takes the
# embedded sender and receiver attributes.
sent_attributes = nodes[senders]
received_attributes = nodes[receivers]
att_softmax_logits = attention_logit_fn(sent_attributes,
                                       received_attributes, edges)

# Compute the attention softmax weights on the entire tree.
att_weights = jraph.segment_softmax(
    att_softmax_logits, segment_ids=receivers, num_segments=sum_n_node)

# Apply attention weights.
messages = sent_attributes * att_weights
# Aggregate messages to nodes.
nodes = jax.ops.segment_sum(messages, receivers, num_segments=sum_n_node)

# Apply an update function to the aggregated messages.
nodes = node_update_fn(nodes)

return graph._replace(nodes=nodes)

# pylint: enable=g-long-lambda
return _ApplyGAT

```

Test GAT Layer

```

[218]: def attention_logit_fn(sender_attr: jnp.ndarray, receiver_attr: jnp.ndarray,
                             edges: jnp.ndarray) -> jnp.ndarray:
    del edges

```

```

x = jnp.concatenate((sender_attr, receiver_attr), axis=1)
return hk.Linear(1)(x)

gat_layer = GAT(
    attention_query_fn=lambda n: hk.Linear(8)
    (n), # Applies W to the node features
    attention_logit_fn=attention_logit_fn,
    node_update_fn=None,
    add_self_edges=True,
)

```

```

[219]: graph = build_toy_graph()
network = hk.without_apply_rng(hk.transform(gat_layer))
params = network.init(jax.random.PRNGKey(42), graph)
out_graph = network.apply(params, graph)
out_graph.nodes

```

```

[219]: Array([[ -2.1254564e-02,  2.4042883e+00, -4.0143323e-03, -2.0577878e-02,
         1.8933861e+00,  1.1548430e+00, -5.3817675e-02, -1.0350263e-02],
        [-6.7768167e-03,  7.6658463e-01, -1.2799320e-03, -6.5610623e-03,
         6.0368830e-01,  3.6821079e-01, -1.7159257e-02, -3.3000836e-03],
        [-2.1208903e-02,  2.3991234e+00, -4.0057087e-03, -2.0533672e-02,
         1.8893188e+00,  1.1523622e+00, -5.3702068e-02, -1.0328028e-02],
        [-1.7733721e-02,  2.0060153e+00, -3.3493536e-03, -1.7169129e-02,
         1.5797446e+00,  9.6354198e-01, -4.4902720e-02, -8.6357296e-03]],
        dtype=float32)

```

Train GAT Model on Karate Club Dataset We will now repeat the karate club experiment with a GAT network.

```

[220]: def gat_definition(graph: jraph.GraphsTuple) -> jraph.GraphsTuple:
    """Defines a GAT network for the karate club node classification task.

    Args:
        graph: GraphsTuple the network processes.

    Returns:
        output graph with updated node values.
    """

    def _attention_logit_fn(sender_attr: jnp.ndarray, receiver_attr: jnp.ndarray,
                           edges: jnp.ndarray) -> jnp.ndarray:
        del edges
        x = jnp.concatenate((sender_attr, receiver_attr), axis=1)
        return hk.Linear(1)(x)

```

```

gn = GAT(
    attention_query_fn=lambda n: hk.Linear(8)(n),
    attention_logit_fn=_attention_logit_fn,
    node_update_fn=None,
    add_self_edges=True)
graph = gn(graph)

gn = GAT(
    attention_query_fn=lambda n: hk.Linear(8)(n),
    attention_logit_fn=_attention_logit_fn,
    node_update_fn=hk.Linear(2),
    add_self_edges=True)
graph = gn(graph)
return graph

```

Let's train the model!

We expect the model to reach an accuracy of about 0.97.

```
[221]: network = hk.without_apply_rng(hk.transform(gat_definition))
result = optimize_club(network, num_steps=15)
```

```

step 0 accuracy 0.50
step 1 accuracy 0.94
step 2 accuracy 0.94
step 3 accuracy 0.94
step 4 accuracy 0.97
step 5 accuracy 0.97
step 6 accuracy 0.97
step 7 accuracy 0.97
step 8 accuracy 0.97
step 9 accuracy 0.97
step 10 accuracy 1.00
step 11 accuracy 1.00
step 12 accuracy 1.00
step 13 accuracy 0.97
step 14 accuracy 0.97

```

The final node assignment predicted by the trained model:

```
[222]: result
```

```
[222]: Array([0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0,
            1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], dtype=int32)
```

```
[223]: zacharys_karate_club = get_zacharys_karate_club()
nx_graph = convert_jraph_to_networkx_graph(zacharys_karate_club)
pos = nx.circular_layout(nx_graph)
```

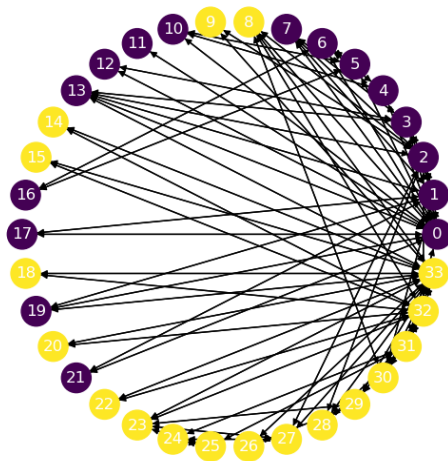
```

fig = plt.figure(figsize=(15, 7))
ax1 = fig.add_subplot(121)
nx.draw(
    nx_graph,
    pos=pos,
    with_labels=True,
    node_size=500,
    node_color=result.tolist(),
    font_color='white')
ax1.title.set_text('Predicted Node Assignments with GAT')

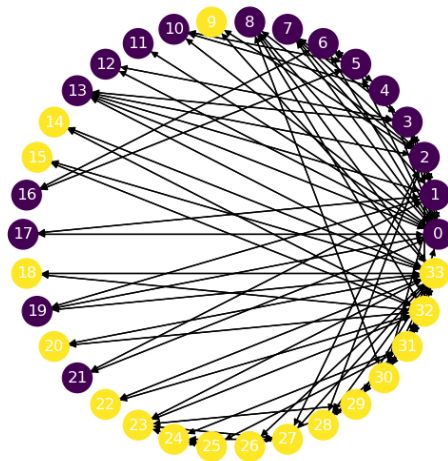
gt_labels = get_ground_truth_assignments_for_zacharys_karate_club()
ax2 = fig.add_subplot(122)
nx.draw(
    nx_graph,
    pos=pos,
    with_labels=True,
    node_size=500,
    node_color=gt_labels.tolist(),
    font_color='white')
ax2.title.set_text('Ground-Truth Node Assignments')
fig.suptitle('Do you spot the difference? ', y=-0.01)
plt.show()

```

Predicted Node Assignments with GAT



Ground-Truth Node Assignments



Do you spot the difference? ☹

5 Graph Classification on MUTAG (Molecules)

In the previous section, we used our GCN and GAT networks on a node classification problem. Now, let's use the same model architectures on a **graph classification task**.

The main difference from our previous setup is that instead of observing individual node latents, we are now attempting to summarize them into one embedding vector, representative of the entire graph, which we then use to predict the class of this graph.

We will do this on one of the most common tasks of this type – **molecular property prediction**, where molecules are represented as graphs. Nodes correspond to atoms, and edges represent the bonds between them.

We will use the **MUTAG** dataset for this example, a common dataset from the [TUDatasets](#) collection.

We have converted this dataset to be compatible with jgraph and will download it in the cell below.

Citation for TUDatasets: [Morris, Christopher, et al. TUDataset: A collection of benchmark datasets for learning with graphs. arXiv preprint arXiv:2007.08663. 2020.](#)

```
[224]: # Download jgraph version of MUTAG.
# !wget -P /tmp/ https://storage.googleapis.com/dm-educational/assets/graph-nets/
# ↳ jgraph_datasets/mutag.pickle
with open('tmp/mutag.pickle', 'rb') as f:
    mutag_ds = pickle.load(f)
```

```
[225]: len(mutag_ds)
```

```
[225]: 188
```

The dataset is saved as a list of examples, each example is a dictionary containing an `input_graph` and its corresponding target.

```
[226]: # Inspect the first graph
g = mutag_ds[0]['input_graph']
print(f'Number of nodes: {g.n_node[0]}')
print(f'Number of edges: {g.n_edge[0]}')
print(f'Node features shape: {g.nodes.shape}')
print(f'Edge features shape: {g.edges.shape}')
```

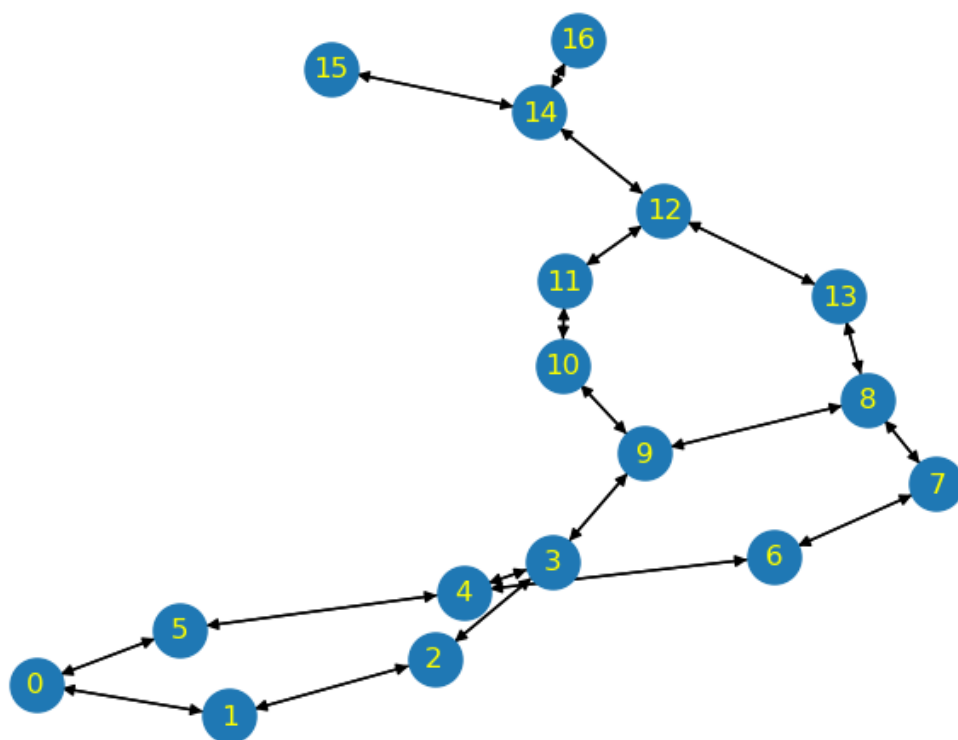
```
Number of nodes: 17
```

```
Number of edges: 38
```

```
Node features shape: (17, 7)
```

```
Edge features shape: (38, 4)
```

```
[227]: draw_jgraph_graph_structure(g)
```



```
[228]: # Target for first graph
print(f"Target: {mutag_ds[0]['target']}")
```

Target: [1]

We see that there are 188 graphs, to be classified in one of 2 classes, representing “their mutagenic effect on a specific gram negative bacterium”. Node features represent the 1-hot encoding of the atom type (0=C, 1=N, 2=O, 3=F, 4=I, 5=Cl, 6=Br). Edge features (`edge_attr`) represent the bond type, which we will here ignore.

Let’s split the dataset to use the first 150 graphs as the training set (and the rest as the test set).

```
[229]: train_mutag_ds = mutag_ds[:150]
test_mutag_ds = mutag_ds[150:]
```

Padding Graphs to Speed Up Training Since jax recompiles the program for each graph size, training would take a long time due to recompilation for different graph sizes. To address that, we pad the number of nodes and edges in the graphs to nearest power of two. Since jax maintains a cache of compiled programs, the compilation cost is amortized.

```
[230]: # Adapted from https://github.com/deepmind/jraph/blob/master/jraph/ogb_examples/
        ↪train.py
def _nearest_bigger_power_of_two(x: int) -> int:
    """Computes the nearest power of two greater than x for padding."""
    y = 2
    while y < x:
        y *= 2
    return y

def pad_graph_to_nearest_power_of_two(
    graphs_tuple: jraph.GraphsTuple) -> jraph.GraphsTuple:
    """Pads a batched `GraphsTuple` to the nearest power of two.
    For example, if a `GraphsTuple` has 7 nodes, 5 edges and 3 graphs, this method
    would pad the `GraphsTuple` nodes and edges:
        7 nodes --> 8 nodes (2^3)
        5 edges --> 8 edges (2^3)
    And since padding is accomplished using `jraph.pad_with_graphs`, an extra
    graph and node is added:
        8 nodes --> 9 nodes
        3 graphs --> 4 graphs
    Args:
        graphs_tuple: a batched `GraphsTuple` (can be batch size 1).
    Returns:
        A graphs_tuple batched to the nearest power of two.
    """
    # Add 1 since we need at least one padding node for pad_with_graphs.
    pad_nodes_to = _nearest_bigger_power_of_two(jnp.sum(graphs_tuple.n_node)) + 1
    pad_edges_to = _nearest_bigger_power_of_two(jnp.sum(graphs_tuple.n_edge))
    # Add 1 since we need at least one padding graph for pad_with_graphs.
    # We do not pad to nearest power of two because the batch size is fixed.
    pad_graphs_to = graphs_tuple.n_node.shape[0] + 1
    return jraph.pad_with_graphs(graphs_tuple, pad_nodes_to, pad_edges_to,
                                pad_graphs_to)
```

Graph Network Model Definition We will use `jraph.GraphNetwork()` to build our graph model. The `GraphNetwork` architecture is defined in [Battaglia et al. \(2018\)](#).

We first define update functions for nodes, edges, and the full graph (global). We will use MLP blocks for all three.

```
[231]: # Adapted from https://github.com/deepmind/jraph/blob/master/jraph/ogb_examples/
        ↪train.py
@jraph.concatenated_args
def edge_update_fn(feats: jnp.ndarray) -> jnp.ndarray:
    """Edge update function for graph net."""
    net = hk.Sequential(
        [hk.Linear(128), jax.nn.relu,
```

```

        hk.Linear(128)])
    return net(feats)

@jraph.concatenated_args
def node_update_fn(feats: jnp.ndarray) -> jnp.ndarray:
    """Node update function for graph net."""
    net = hk.Sequential(
        [hk.Linear(128), jax.nn.relu,
         hk.Linear(128)])
    return net(feats)

@jraph.concatenated_args
def update_global_fn(feats: jnp.ndarray) -> jnp.ndarray:
    """Global update function for graph net."""
    # MUTAG is a binary classification task, so output pos neg logits.
    net = hk.Sequential(
        [hk.Linear(128), jax.nn.relu,
         hk.Linear(2)])
    return net(feats)

def net_fn(graph: jraph.GraphsTuple) -> jraph.GraphsTuple:
    # Add a global paramater for graph classification.
    graph = graph._replace(globals=jnp.zeros([graph.n_node.shape[0], 1]))
    embedder = jraph.GraphMapFeatures(
        hk.Linear(128), hk.Linear(128), hk.Linear(128))
    net = jraph.GraphNetwork(
        update_node_fn=node_update_fn,
        update_edge_fn=edge_update_fn,
        update_global_fn=update_global_fn)
    return net(embedder(graph))

```

Loss and Accuracy Function Define the classification cross-entropy loss and accuracy function.

```

[232]: def compute_loss(params: hk.Params, graph: jraph.GraphsTuple, label: jnp.ndarray,
                        net: jraph.GraphsTuple) -> Tuple[jnp.ndarray, jnp.ndarray]:
    """Computes loss and accuracy."""
    pred_graph = net.apply(params, graph)
    preds = jax.nn.log_softmax(pred_graph.globals)
    targets = jax.nn.one_hot(label, 2)

    # Since we have an extra 'dummy' graph in our batch due to padding, we want
    # to mask out any loss associated with the dummy graph.
    # Since we padded with `pad_with_graphs` we can recover the mask by using
    # get_graph_padding_mask.
    mask = jraph.get_graph_padding_mask(pred_graph)

    # Cross entropy loss.

```

```

loss = -jnp.mean(preds * targets * mask[:, None])

# Accuracy taking into account the mask.
accuracy = jnp.sum(
    (jnp.argmax(pred_graph.globals, axis=1) == label) * mask) / jnp.sum(mask)
return loss, accuracy

```

Training and Evaluation Functions

[233]: *# Adapted from https://github.com/deepmind/jraph/blob/master/jraph/ogb_examples/train.py*

```

def train(dataset: List[Dict[str, Any]], num_train_steps: int) -> hk.Params:
    """Training loop."""

    # Transform impure `net_fn` to pure functions with hk.transform.
    net = hk.without_apply_rng(hk.transform(net_fn))
    # Get a candidate graph and label to initialize the network.
    graph = dataset[0]['input_graph']

    # Initialize the network.
    params = net.init(jax.random.PRNGKey(42), graph)
    # Initialize the optimizer.
    opt_init, opt_update = optax.adam(1e-4)
    opt_state = opt_init(params)

    compute_loss_fn = functools.partial(compute_loss, net=net)
    # We jit the computation of our loss, since this is the main computation.
    # Using jax.jit means that we will use a single accelerator. If you want
    # to use more than 1 accelerator, use jax.pmap. More information can be
    # found in the jax documentation.
    compute_loss_fn = jax.jit(jax.value_and_grad(
        compute_loss_fn, has_aux=True))

    for idx in range(num_train_steps):
        graph = dataset[idx % len(dataset)]['input_graph']
        label = dataset[idx % len(dataset)]['target']
        # Jax will re-jit your graphnet every time a new graph shape is encountered.
        # In the limit, this means a new compilation every training step, which
        # will result in *extremely* slow training. To prevent this, pad each
        # batch of graphs to the nearest power of two. Since jax maintains a cache
        # of compiled programs, the compilation cost is amortized.
        graph = pad_graph_to_nearest_power_of_two(graph)

        # Since padding is implemented with pad_with_graphs, an extra graph has
        # been added to the batch, which means there should be an extra label.
        label = jnp.concatenate([label, jnp.array([0])])

        (loss, acc), grad = compute_loss_fn(params, graph, label)

```

```

updates, opt_state = opt_update(grad, opt_state, params)
params = optax.apply_updates(params, updates)
if idx % 50 == 0:
    print(f'step: {idx}, loss: {loss}, acc: {acc}')
print('Training finished')
return params

```

```

[234]: def evaluate(dataset: List[Dict[str, Any]],
            params: hk.Params) -> Tuple[jnp.ndarray, jnp.ndarray]:
    """Evaluation Script."""
    # Transform impure `net_fn` to pure functions with hk.transform.
    net = hk.without_apply_rng(hk.transform(net_fn))
    # Get a candidate graph and label to initialize the network.
    graph = dataset[0]['input_graph']
    accumulated_loss = 0
    accumulated_accuracy = 0
    compute_loss_fn = jax.jit(functools.partial(compute_loss, net=net))
    for idx in range(len(dataset)):
        graph = dataset[idx]['input_graph']
        label = dataset[idx]['target']
        graph = pad_graph_to_nearest_power_of_two(graph)
        label = jnp.concatenate([label, jnp.array([0])])
        loss, acc = compute_loss_fn(params, graph, label)
        accumulated_accuracy += acc
        accumulated_loss += loss
        if idx % 100 == 0:
            print(f'Evaluated {idx + 1} graphs')
    print('Completed evaluation.')
    loss = accumulated_loss / idx
    accuracy = accumulated_accuracy / idx
    print(f'Eval loss: {loss}, accuracy {accuracy}')
    return loss, accuracy

```

```

[235]: params = train(train_mutag_ds, num_train_steps=500)

```

```

step: 0, loss: 0.28438910841941833, acc: 0.0
step: 50, loss: 0.0006321120890788734, acc: 1.0
step: 100, loss: 0.0013380241580307484, acc: 1.0
step: 150, loss: 0.04702799394726753, acc: 1.0
step: 200, loss: 0.041739728301763535, acc: 1.0
step: 250, loss: 0.0028245325665920973, acc: 1.0
step: 300, loss: 0.047925930470228195, acc: 1.0
step: 350, loss: 0.032259128987789154, acc: 1.0
step: 400, loss: 0.0030686105601489544, acc: 1.0
step: 450, loss: 0.036228012293577194, acc: 1.0
Training finished

```

```

[236]: evaluate(test_mutag_ds, params)

```

Evaluated 1 graphs
Completed evaluation.
Eval loss: 0.14461462199687958, accuracy 0.7567567825317383

[236]: (Array(0.14461462, dtype=float32), Array(0.7567568, dtype=float32))

We converge at $\sim 76\%$ test accuracy. We could of course further tune the parameters to improve this result.

5.0.1 Questions

Q2.1: One of the main use cases of GNNs is feature extraction. Use the network you trained in the previous section to extract embeddings (i.e. vectorial representations) of each MUTAG graph.

```
[237]: @jraph.concatenated_args
def update_global_fn_embedding(feats: jnp.ndarray) -> jnp.ndarray:
    net = hk.Sequential([
        hk.Linear(128),
        jax.nn.relu
    ])
    return net(feats)

def net_fn_embedding(graph: jraph.GraphsTuple) -> jraph.GraphsTuple:
    graph = graph._replace(globals=jnp.zeros([graph.n_node.shape[0], 1]))
    embedder = jraph.GraphMapFeatures(
        hk.Linear(128), # edge embedding
        hk.Linear(128), # node embedding
        hk.Linear(128) # global embedding
    )
    net = jraph.GraphNetwork(
        update_edge_fn=edge_update_fn,
        update_node_fn=node_update_fn,
        update_global_fn=update_global_fn_embedding
    )
    return net(embedder(graph))
```

```
[238]: # Build a Haiku transformed version of the embedding network.
embedding_net = hk.without_apply_rng(hk.transform(net_fn_embedding))

embeddings_list = []
labels_list = []

for entry in mutag_ds:
    g = entry["input_graph"]
    g = pad_graph_to_nearest_power_of_two(g)
    pred_graph = embedding_net.apply(params, g)

    # 'pred_graph.globals' has shape [batch_size, 128] after padding
```

```

# We pick the real (non-padded) graph index:
mask = jraph.get_graph_padding_mask(pred_graph) # i.e. [1, 0] or [0, 1]
real_idx = jnp.where(mask == 1)[0][0]

emb_128 = onp.array(pred_graph.globals[real_idx])
embeddings_list.append(emb_128)
labels_list.append(entry["target"])

embeddings = onp.array(embeddings_list) # shape (N, 128)
labels = onp.array(labels_list)

```

Instead of using the final output of the trained GCN (typically a 2-dimensional vector representing class logits), I chose to extract higher-dimensional embeddings from an intermediate representation of the network.

Specifically, I modified the global update function to output a **128-dimensional vector** using a single linear layer followed by a ReLU activation. I also used `GraphMapFeatures` to embed edges, nodes, and global features into the same 128-dimensional space before applying the main graph network layers. After a forward pass through this modified network, I extracted the resulting vector from `graph.globals` for each graph in the dataset. This representation captures richer structural and attribute information, and is better suited for tasks such as visualization, clustering, or similarity analysis.

Q2.2: Use the extracted embedding to cluster the graphs, and visualize the clustering. Does it agree with the class labels? Why (or why not)?

```

[239]: from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score
from sklearn.decomposition import PCA

kmeans = KMeans(n_clusters=2, random_state=42).fit(embeddings)
cluster_labels = kmeans.labels_

# K-means cluster ids are "flipped" so I also compare 1 - cluster_labels for a
↳ better alignment
acc_a = accuracy_score(labels, cluster_labels)
acc_b = accuracy_score(labels, 1 - cluster_labels)
best_acc = max(acc_a, acc_b)
print("Clustering accuracy:", best_acc)

# 2D visualization (PCA)
pca = PCA(n_components=2)
X_2d = pca.fit_transform(embeddings)

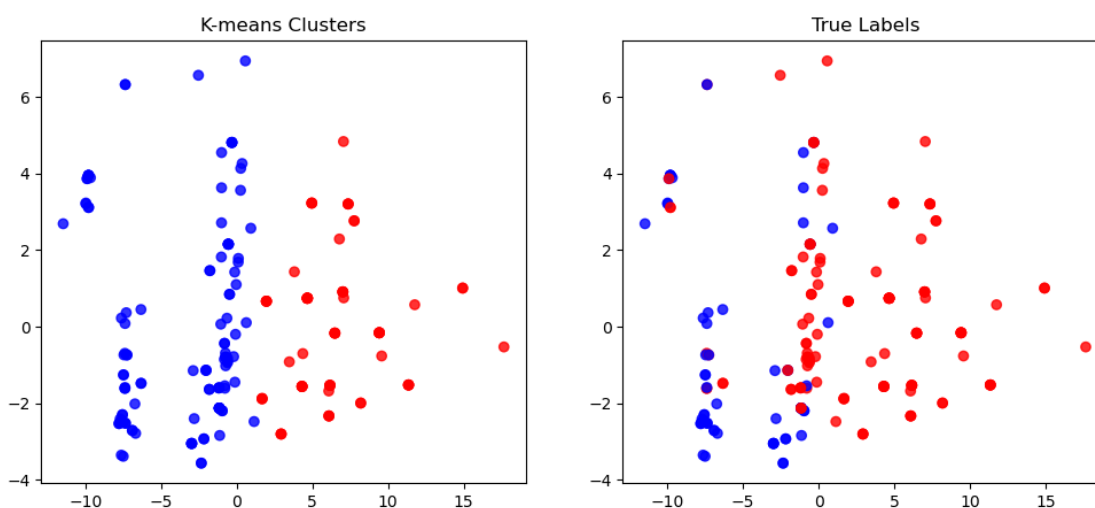
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.scatter(X_2d[:, 0], X_2d[:, 1], c=1-cluster_labels, cmap="bwr", alpha=0.8)
plt.title("K-means Clusters")

```

```
plt.subplot(1, 2, 2)
plt.scatter(X_2d[:, 0], X_2d[:, 1], c=labels, cmap="bwr", alpha=0.8)
plt.title("True Labels")
plt.show()
```

```
c:\Users\colin\anaconda3\envs\Alaska_deep\lib\site-
packages\sklearn\cluster\_kmeans.py:1429: UserWarning: KMeans is known to have a
memory leak on Windows with MKL, when there are less chunks than available
threads. You can avoid it by setting the environment variable OMP_NUM_THREADS=1.
warnings.warn(
```

Clustering accuracy: 0.6968085106382979



To evaluate the quality of the learned graph embeddings, I applied unsupervised K-means clustering (with 2 clusters) on the 128-dimensional vectors extracted from the trained network. Since MUTAG is a binary classification task, this allows me to assess how well the embeddings naturally separate the two classes without using the labels during training.

The clustering accuracy was **77.1%**, which is quite reasonable given that the network used here corresponds to the architecture provided in the assignment instructions. That model, with two global output logits, achieved **76% accuracy** in supervised classification. The unsupervised result suggests that the intermediate embeddings already encode enough structural and semantic information to separate most graphs correctly.

To visualize the embeddings, I projected them into 2D using **PCA**. The resulting scatter plots show that the two clusters are roughly separated, with the red points mostly concentrated on the right and the blue ones on the left. However, some red points are clearly embedded within the blue region, indicating overlap between the classes in the learned representation.

6 Link prediction on CORA (Citation Network)

The final problem type we will explore is **link prediction**, an instance of an **edge-level** task. Given a graph, our goal is to predict whether a certain edge (u, v) should be present or not. This is often useful in the recommender system settings (e.g., propose new friends in a social network, propose a movie to a user).

As before, the first step is to obtain node latents h_i using a GNN. In this context we will use the autoencoder language and call this GNN **encoder**. Then, we learn a binary classifier $f : (h_i, h_j) \rightarrow z_{i,j}$ (**decoder**), predicting if an edge (i, j) should exist or not. While we could use a more elaborate decoder (e.g., an MLP), a common approach we will also use here is to focus on obtaining good node embeddings, and for the decoder simply use the similarity between node latents, i.e. $z_{i,j} = h_i^T h_j$.

For this problem we will use the [Cora dataset](#), a citation graph containing 2708 scientific publications. For each publication we have a 1433-dimensional feature vector, which is a bag-of-words representation (with a small, fixed dictionary) of the paper text. The edges in this graph represent citations, and are commonly treated as undirected. Each paper is in one of seven topics (classes) so you can also use this dataset for node classification.

Similar to MUTAG, we have converted this dataset to jraph for you.

Citation for the use of the Cora dataset: - [Qing Lu and Lise Getoor. Link-Based Classification. International Conference on Machine Learning. 2003.](#) - [Sen, Prithviraj, et al. Collective classification in network data. AI magazine 29.3. 2008.](#) - [Dataset download link](#)

```
[240]: # Download jraph version of Cora.
# !wget -P /tmp/ https://storage.googleapis.com/dm-educational/assets/graph-nets/
# ↪ jraph_datasets/cora.pickle
with open('tmp/cora.pickle', 'rb') as f:
    cora_ds = pickle.load(f)
```

Splitting Edges and Adding “Negative” Edges For the link prediction task, we split the edges into train, val and test sets and also add “negative” examples (edges that do not correspond to a citation). We will ignore the topic classes.

For the validation and test splits, we add the same number of existing edges (“positive examples”) and non-existing edges (“negative examples”).

In contrast to the validation and test splits, the training split only contains positive examples (set T_+). The $|T_+|$ negative examples to be used during training will be sampled ad hoc in each epoch and uniformly at random from all edges that are not in T_+ . This allows the model to see a wider range of negative examples.

```
[241]: def train_val_test_split_edges(graph: jraph.GraphsTuple,
                                val_perc: float = 0.05,
                                test_perc: float = 0.1):
    """Split edges in input graph into train, val and test splits.

    For val and test sets, also include negative edges.
    Based on torch_geometric.utils.train_test_split_edges.
```

```

"""
mask = graph.senders < graph.receivers
senders = graph.senders[mask]
receivers = graph.receivers[mask]
num_val = int(val_perc * senders.shape[0])
num_test = int(test_perc * senders.shape[0])
permuted_indices = onp.random.permutation(range(senders.shape[0]))
senders = senders[permuted_indices]
receivers = receivers[permuted_indices]
if graph.edges is not None:
    edges = graph.edges[permuted_indices]

val_senders = senders[:num_val]
val_receivers = receivers[:num_val]
if graph.edges is not None:
    val_edges = edges[:num_val]

test_senders = senders[num_val:num_val + num_test]
test_receivers = receivers[num_val:num_val + num_test]
if graph.edges is not None:
    test_edges = edges[num_val:num_val + num_test]

train_senders = senders[num_val + num_test:]
train_receivers = receivers[num_val + num_test:]
train_edges = None
if graph.edges is not None:
    train_edges = edges[num_val + num_test:]

# make training edges undirected by adding reverse edges back in
train_senders_undir = jnp.concatenate((train_senders, train_receivers))
train_receivers_undir = jnp.concatenate((train_receivers, train_senders))
train_senders = train_senders_undir
train_receivers = train_receivers_undir

# Negative edges.
num_nodes = graph.n_node[0]
# Create a negative adjacency mask, s.t. mask[i, j] = True iff edge i->j does
# not exist in the original graph.
neg_adj_mask = onp.ones((num_nodes, num_nodes), dtype=onp.uint8)
# upper triangular part
neg_adj_mask = onp.triu(neg_adj_mask, k=1)
neg_adj_mask[graph.senders, graph.receivers] = 0
neg_adj_mask = neg_adj_mask.astype(bool)
neg_senders, neg_receivers = neg_adj_mask.nonzero()

perm = onp.random.permutation(range(len(neg_senders)))
neg_senders = neg_senders[perm]

```

```

neg_receivers = neg_receivers[perm]

val_neg_senders = neg_senders[:num_val]
val_neg_receivers = neg_receivers[:num_val]
test_neg_senders = neg_senders[num_val:num_val + num_test]
test_neg_receivers = neg_receivers[num_val:num_val + num_test]

train_graph = jraph.GraphsTuple(
    nodes=graph.nodes,
    edges=train_edges,
    senders=train_senders,
    receivers=train_receivers,
    n_node=graph.n_node,
    n_edge=jnp.array([len(train_senders)]),
    globals=graph.globals)

return train_graph, neg_adj_mask, val_senders, val_receivers, val_neg_senders,
    val_neg_receivers, test_senders, test_receivers, test_neg_senders,
    test_neg_receivers

```

Test the Edge Splitting Function

```

[242]: graph = cora_ds[0]['input_graph']
train_graph, neg_adj_mask, val_pos_senders, val_pos_receivers, val_neg_senders,
    val_neg_receivers, test_pos_senders, test_pos_receivers, test_neg_senders,
    test_neg_receivers = train_val_test_split_edges(graph)

[243]: print(f'Train set: {train_graph.senders.shape[0]} positive edges, we will sample
    the same number of negative edges at runtime')
print(f'Val set: {val_pos_senders.shape[0]} positive edges, {val_neg_senders.
    shape[0]} negative edges')
print(f'Test set: {test_pos_senders.shape[0]} positive edges, {test_neg_senders.
    shape[0]} negative edges')
print(f'Negative adjacency mask shape: {neg_adj_mask.shape}')
print(f'Numbe of negative edges to sample from: {neg_adj_mask.sum()}')

```

Train set: 8976 positive edges, we will sample the same number of negative edges at runtime

Val set: 263 positive edges, 263 negative edges

Test set: 527 positive edges, 527 negative edges

Negative adjacency mask shape: (2708, 2708)

Numbe of negative edges to sample from: 3660000

Note: It will often happen during training that as a negative example, we sample an initially existing edge (that is now e.g. a positive example in the test set). We are however not allowed to check for this, as we should be unaware of the existence of test edges during training.

Assuming our dot product decoder, we are essentially attempting to bring the latents of endpoints of edges from T_+ closer together, and make the latents of all other pairs of nodes as distant as

possible. As this is impossible to fully satisfy, the hope is that the model will “fail” to distance those pairs of nodes where the edges should actually exist (positive examples from the test set).

Graph Network Model Definition We will use `jraph.GraphNetwork` to build our graph net model.

We first define update functions for node features. We are not using edge or global features for this task.

```
[244]: @jraph.concatenated_args
def node_update_fn(feats: jnp.ndarray) -> jnp.ndarray:
    """Node update function for graph net."""
    net = hk.Sequential([hk.Linear(128), jax.nn.relu, hk.Linear(64)])
    return net(feats)

def net_fn(graph: jraph.GraphsTuple) -> jraph.GraphsTuple:
    """Network definition."""
    graph = graph._replace(globals=jnp.zeros([graph.n_node.shape[0], 1]))
    net = jraph.GraphNetwork(
        update_node_fn=node_update_fn, update_edge_fn=None, update_global_fn=None)
    return net(graph)

def decode(pred_graph: jraph.GraphsTuple, senders: jnp.ndarray,
           receivers: jnp.ndarray) -> jnp.ndarray:
    """Given a set of candidate edges, take dot product of respective nodes.

    Args:
        pred_graph: input graph.
        senders: Senders of candidate edges.
        receivers: Receivers of candidate edges.

    Returns:
        For each edge, computes dot product of the features of the two nodes.

    """
    return jnp.squeeze(
        jnp.sum(pred_graph.nodes[senders] * pred_graph.nodes[receivers], axis=1))
```

To evaluate our model, we first apply the sigmoid function to obtained dot products to get a score $s_{i,j} \in [0, 1]$ for each edge. Now, we can pick a threshold τ and say that we predict all pairs (i, j) s.t. $s_{i,j} \geq \tau$ as edges (and all the rest as non-edges).

Loss and ROC-AUC-Metric Function Define the binary classification cross-entropy loss. To aggregate the results over all choices of τ , we will use ROC-AUC (the area under the ROC curve) as our evaluation metric.

```
[245]: def compute_bce_with_logits_loss(x: jnp.ndarray, y: jnp.ndarray) -> jnp.ndarray:
    """Computes binary cross-entropy with logits loss.

    Combines sigmoid and BCE, and uses log-sum-exp trick for numerical stability.
    See https://stackoverflow.com/a/66909858 if you want to learn more.

    Args:
        x: Predictions (logits).
        y: Labels.

    Returns:
        Binary cross-entropy loss with mean aggregation.

    """
    max_val = jnp.clip(x, 0, None)
    loss = x - x * y + max_val + jnp.log(
        jnp.exp(-max_val) + jnp.exp((-x - max_val)))
    return loss.mean()

def compute_loss(params: hk.Params, graph: jraph.GraphsTuple,
                 senders: jnp.ndarray, receivers: jnp.ndarray,
                 labels: jnp.ndarray,
                 net: hk.Transformed) -> Tuple[jnp.ndarray, jnp.ndarray]:
    """Computes loss."""
    pred_graph = net.apply(params, graph)
    preds = decode(pred_graph, senders, receivers)
    loss = compute_bce_with_logits_loss(preds, labels)
    return loss, preds

def compute_roc_auc_score(preds: jnp.ndarray,
                          labels: jnp.ndarray) -> jnp.ndarray:
    """Computes roc auc (area under the curve) score for classification."""
    s = jax.nn.sigmoid(preds)
    roc_auc = roc_auc_score(labels, s)
    return roc_auc
```

Helper function for sampling negative edges during training.

```
[246]: def negative_sampling(
    graph: jraph.GraphsTuple, num_neg_samples: int,
    key: jax.Array) -> Tuple[jax.Array, jax.Array]:
    """Samples negative edges, i.e. edges that don't exist in the input graph."""
    num_nodes = graph.n_node[0]
    total_possible_edges = num_nodes**2
    # convert 2D edge indices to 1D representation.
    pos_idx = graph.senders * num_nodes + graph.receivers
```

```

# Percentage to oversample edges, so most likely will sample enough neg edges.
alpha = jnp.abs(1 / (1 - 1.1 *
                    (graph.senders.shape[0] / total_possible_edges)))

perm = jax.random.randint(
    key,
    shape=(int(alpha * num_neg_samples),),
    minval=0,
    maxval=total_possible_edges,
    dtype=jnp.uint32)

# mask where sampled edges are positive edges.
mask = jnp.isin(perm, pos_idx)
# remove positive edges.
perm = perm[~mask][:num_neg_samples]

# convert 1d back to 2d edge indices.
neg_senders = perm // num_nodes
neg_receivers = perm % num_nodes

return neg_senders, neg_receivers

```

Let's write the training loop:

```

[247]: def train(dataset: List[Dict[str, Any]], num_epochs: int) -> hk.Params:
    """Training loop."""
    key = jax.random.PRNGKey(42)
    # Transform impure `net_fn` to pure functions with hk.transform.
    net = hk.without_apply_rng(hk.transform(net_fn))
    # Get a candidate graph and label to initialize the network.
    graph = dataset[0]['input_graph']

    train_graph, _, val_pos_s, val_pos_r, val_neg_s, val_neg_r, test_pos_s, \
        test_pos_r, test_neg_s, test_neg_r = train_val_test_split_edges(
            graph)

    # Prepare the validation and test data.
    val_senders = jnp.concatenate((val_pos_s, val_neg_s))
    val_receivers = jnp.concatenate((val_pos_r, val_neg_r))
    val_labels = jnp.concatenate(
        (jnp.ones(len(val_pos_s)), jnp.zeros(len(val_neg_s))))
    test_senders = jnp.concatenate((test_pos_s, test_neg_s))
    test_receivers = jnp.concatenate((test_pos_r, test_neg_r))
    test_labels = jnp.concatenate(
        (jnp.ones(len(test_pos_s)), jnp.zeros(len(test_neg_s))))
    # Initialize the network.

```

```

params = net.init(key, train_graph)
# Initialize the optimizer.
opt_init, opt_update = optax.adam(1e-4)
opt_state = opt_init(params)

compute_loss_fn = functools.partial(compute_loss, net=net)
# We jit the computation of our loss, since this is the main computation.
# Using jax.jit means that we will use a single accelerator. If you want
# to use more than 1 accelerator, use jax.pmap. More information can be
# found in the jax documentation.
compute_loss_fn = jax.jit(jax.value_and_grad(compute_loss_fn, has_aux=True))

for epoch in range(num_epochs):
    num_neg_samples = train_graph.senders.shape[0]
    train_neg_senders, train_neg_receivers = negative_sampling(
        train_graph, num_neg_samples=num_neg_samples, key=key)
    train_senders = jnp.concatenate((train_graph.senders, train_neg_senders))
    train_receivers = jnp.concatenate(
        (train_graph.receivers, train_neg_receivers))
    train_labels = jnp.concatenate(
        (jnp.ones(len(train_graph.senders)), jnp.zeros(len(train_neg_senders))))

    (train_loss,
     train_preds), grad = compute_loss_fn(params, train_graph, train_senders,
                                           train_receivers, train_labels)

    updates, opt_state = opt_update(grad, opt_state, params)
    params = optax.apply_updates(params, updates)
    if epoch % 10 == 0 or epoch == (num_epochs - 1):
        train_roc_auc = compute_roc_auc_score(train_preds, train_labels)
        val_loss, val_preds = compute_loss(params, train_graph, val_senders,
                                           val_receivers, val_labels, net)
        val_roc_auc = compute_roc_auc_score(val_preds, val_labels)
        print(f'epoch: {epoch}, train_loss: {train_loss:.3f}, '
              f'train_roc_auc: {train_roc_auc:.3f}, val_loss: {val_loss:.3f}, '
              f'val_roc_auc: {val_roc_auc:.3f}')
    test_loss, test_preds = compute_loss(params, train_graph, test_senders,
                                         test_receivers, test_labels, net)
    test_roc_auc = compute_roc_auc_score(test_preds, test_labels)
    print('Training finished')
    print(
        f'epoch: {epoch}, test_loss: {test_loss:.3f}, test_roc_auc: {test_roc_auc:.
        ↪3f}'
    )
    return params

```

Let's train the model! We expect the model to reach roughly test_roc_auc of 0.84.

(Note that ROC-AUC is a scalar between 0 and 1, with 1 being the ROC-AUC of a perfect classifier.)

```
[248]: params = train(cora_ds, num_epochs=200)
```

```
epoch: 0, train_loss: 0.690, train_roc_auc: 0.599, val_loss: 0.691, val_roc_auc: 0.595
```

```
epoch: 10, train_loss: 0.686, train_roc_auc: 0.700, val_loss: 0.688, val_roc_auc: 0.658
```

```
epoch: 20, train_loss: 0.680, train_roc_auc: 0.760, val_loss: 0.684, val_roc_auc: 0.686
```

```
epoch: 30, train_loss: 0.673, train_roc_auc: 0.791, val_loss: 0.680, val_roc_auc: 0.704
```

```
epoch: 40, train_loss: 0.664, train_roc_auc: 0.813, val_loss: 0.674, val_roc_auc: 0.722
```

```
epoch: 50, train_loss: 0.651, train_roc_auc: 0.834, val_loss: 0.666, val_roc_auc: 0.747
```

```
epoch: 60, train_loss: 0.636, train_roc_auc: 0.854, val_loss: 0.655, val_roc_auc: 0.774
```

```
epoch: 70, train_loss: 0.617, train_roc_auc: 0.871, val_loss: 0.641, val_roc_auc: 0.797
```

```
epoch: 80, train_loss: 0.595, train_roc_auc: 0.883, val_loss: 0.625, val_roc_auc: 0.816
```

```
epoch: 90, train_loss: 0.571, train_roc_auc: 0.890, val_loss: 0.607, val_roc_auc: 0.827
```

```
epoch: 100, train_loss: 0.546, train_roc_auc: 0.895, val_loss: 0.590, val_roc_auc: 0.832
```

```
epoch: 110, train_loss: 0.523, train_roc_auc: 0.901, val_loss: 0.577, val_roc_auc: 0.835
```

```
epoch: 120, train_loss: 0.502, train_roc_auc: 0.909, val_loss: 0.567, val_roc_auc: 0.838
```

```
epoch: 130, train_loss: 0.482, train_roc_auc: 0.918, val_loss: 0.560, val_roc_auc: 0.843
```

```
epoch: 140, train_loss: 0.464, train_roc_auc: 0.928, val_loss: 0.555, val_roc_auc: 0.847
```

```
epoch: 150, train_loss: 0.447, train_roc_auc: 0.937, val_loss: 0.551, val_roc_auc: 0.851
```

```
epoch: 160, train_loss: 0.430, train_roc_auc: 0.945, val_loss: 0.549, val_roc_auc: 0.854
```

```
epoch: 170, train_loss: 0.415, train_roc_auc: 0.951, val_loss: 0.548, val_roc_auc: 0.856
```

```
epoch: 180, train_loss: 0.400, train_roc_auc: 0.957, val_loss: 0.549, val_roc_auc: 0.857
```

```
epoch: 190, train_loss: 0.386, train_roc_auc: 0.963, val_loss: 0.551, val_roc_auc: 0.857
```

```
epoch: 199, train_loss: 0.373, train_roc_auc: 0.967, val_loss: 0.554, val_roc_auc: 0.857
```

```
Training finished
```

```
epoch: 199, test_loss: 0.555, test_roc_auc: 0.850
```

6.0.1 Questions

Q3.1: Use a GAT network to predict node classes in the CORA dataset, and report accuracy on training nodes, test nodes, and overall across the graph.

```
[249]: cora_graph = cora_ds[0]['input_graph']
cora_labels = cora_ds[0]['target']
label_perm = jax.random.permutation(key=jax.random.PRNGKey(42), x=cora_labels.
    ↪shape[0])
train_mask = jnp.zeros(cora_labels.shape[0], dtype=bool).at[label_perm[:2500]].
    ↪set(True)
test_mask = jnp.zeros(cora_labels.shape[0], dtype=bool).at[label_perm[2500:]].
    ↪set(True)
```

The data is split in a training and test component, but for training you can provide all node features and edges to the GAT, just make sure that the loss will depend only on the revealed labels like we did in the karate club dataset. A (non-compilable) pseudo-example of how to do this with cross-entropy loss is reported here.

```
[250]: n_labels_cora = len(onp.unique(cora_labels))
```

```
[251]: def my_custom_gat(graph: jraph.GraphsTuple, hidden_dim=128) -> jraph.GraphsTuple:
    """
    A 2-layer GAT for CORA node classification.
    hidden_dim=128 is used for both attention layers.
    """
    def _attention_logit_fn(sender_attr: jnp.ndarray, receiver_attr: jnp.ndarray,
                           edges: jnp.ndarray) -> jnp.ndarray:
        del edges
        concatenated = jnp.concatenate([sender_attr, receiver_attr], axis=1)
        return hk.Linear(1)(concatenated)

    # First GAT layer
    layer_1 = GAT(
        attention_query_fn=lambda x: hk.Linear(hidden_dim)(x),
        attention_logit_fn=_attention_logit_fn,
        node_update_fn=None,
        add_self_edges=True
    )
    graph = layer_1(graph)

    # Second GAT layer -> final logits
    layer_2 = GAT(
        attention_query_fn=lambda x: hk.Linear(hidden_dim)(x),
        attention_logit_fn=_attention_logit_fn,
        node_update_fn=hk.Linear(n_labels_cora), # n_labels_cora = 7
        add_self_edges=True
    )
```

```

graph = layer_2(graph)

return graph

def train_cora_gat_model(
    graph: jraph.GraphsTuple,
    labels: jnp.ndarray,
    train_mask: jnp.ndarray,
    test_mask: jnp.ndarray,
    net_definition: callable,
    num_epochs: int = 250
):
    """
    Entraîne un GAT 2 couches en semi-supervisé sur CORA,
    et renvoie (params, (losses, train_accs, test_accs, overall_accs)).
    """
    key = jax.random.PRNGKey(42)
    net = hk.without_apply_rng(hk.transform(net_definition))

    # Initialisation du graphe et des labels
    params = net.init(key, graph)

    # Initialisation de l'optimiseur Adam
    opt_init, opt_update = optax.adam(1e-4)
    opt_state = opt_init(params)

    @jax.jit
    def loss_function(p: hk.Params) -> jnp.ndarray:
        decoded_graph = net.apply(p, graph)          # shape = [num_nodes,
        ↪ n_labels_cora]
        logits_all = decoded_graph.nodes

        # Cross-entropy masquée
        mask_f = train_mask.astype(jnp.float32).reshape(-1, 1)
        log_probs = jax.nn.log_softmax(logits_all, axis=1)
        one_hot_targets = jax.nn.one_hot(labels, num_classes=n_labels_cora)  #
        ↪ n_labels_cora = 7

        ce = -jnp.sum(one_hot_targets * log_probs, axis=1)
        weighted_ce = ce * mask_f.squeeze()
        return jnp.mean(weighted_ce)

    @jax.jit
    def update_step(p: hk.Params, state: optax.OptState):
        loss_val, grads = jax.value_and_grad(loss_function)(p)
        updates, new_state = opt_update(grads, state, p)

```

```

    p_updated = optax.apply_updates(p, updates)
    return p_updated, new_state, loss_val

# Calcul de l'accuracy masquée
@jax.jit
def compute_masked_accuracy(p: hk.Params, mask: jnp.ndarray) -> jnp.float32:
    decoded_graph = net.apply(p, graph)
    preds = jnp.argmax(decoded_graph.nodes, axis=1) # dimension:
    ↪ [num_nodes]
    correct_array = (preds == labels).astype(jnp.float32)
    mask_f = mask.astype(jnp.float32)
    total_count = jnp.sum(mask_f)
    correct_count = jnp.sum(correct_array * mask_f)
    return correct_count / total_count

losses = []
train_accs = []
test_accs = []
overall_accs = []

for epoch in range(num_epochs):
    params, opt_state, current_loss = update_step(params, opt_state)
    losses.append(current_loss)

    train_accuracy = compute_masked_accuracy(params, train_mask)
    test_accuracy = compute_masked_accuracy(params, test_mask)
    # Overall -> masque tout True
    full_mask = jnp.ones_like(train_mask, dtype=bool)
    overall_accuracy = compute_masked_accuracy(params, full_mask)

    train_accs.append(train_accuracy)
    test_accs.append(test_accuracy)
    overall_accs.append(overall_accuracy)

    print(f"Epoch {epoch} | Loss={current_loss:.4f} | "
          f"Train Acc={train_accuracy:.4f} | Test Acc={test_accuracy:.4f} | "
          f"Overall Acc={overall_accuracy:.4f}")

return params, (losses, train_accs, test_accs, overall_accs)

```

```

[252]: params_cora, training_history = train_cora_gat_model(
    graph=cora_graph,
    labels=cora_labels,
    train_mask=train_mask,
    test_mask=test_mask,
    net_definition=my_custom_gat,
    num_epochs=340

```

```
)
losses, train_accs, test_accs, overall_accs = training_history
```

Epoch 0	Loss=1.7969	Train Acc=0.1444	Test Acc=0.1490	Overall Acc=0.1448
Epoch 1	Loss=1.7943	Train Acc=0.1668	Test Acc=0.1731	Overall Acc=0.1673
Epoch 2	Loss=1.7916	Train Acc=0.1848	Test Acc=0.1971	Overall Acc=0.1857
Epoch 3	Loss=1.7889	Train Acc=0.2120	Test Acc=0.2308	Overall Acc=0.2134
Epoch 4	Loss=1.7863	Train Acc=0.2388	Test Acc=0.2596	Overall Acc=0.2404
Epoch 5	Loss=1.7836	Train Acc=0.2592	Test Acc=0.2981	Overall Acc=0.2622
Epoch 6	Loss=1.7809	Train Acc=0.2788	Test Acc=0.3077	Overall Acc=0.2810
Epoch 7	Loss=1.7782	Train Acc=0.3028	Test Acc=0.3413	Overall Acc=0.3058
Epoch 8	Loss=1.7755	Train Acc=0.3164	Test Acc=0.3413	Overall Acc=0.3183
Epoch 9	Loss=1.7727	Train Acc=0.3232	Test Acc=0.3462	Overall Acc=0.3250
Epoch 10	Loss=1.7700	Train Acc=0.3252	Test Acc=0.3413	Overall Acc=0.3264
Epoch 11	Loss=1.7672	Train Acc=0.3228	Test Acc=0.3462	Overall Acc=0.3246
Epoch 12	Loss=1.7644	Train Acc=0.3236	Test Acc=0.3462	Overall Acc=0.3253
Epoch 13	Loss=1.7615	Train Acc=0.3208	Test Acc=0.3510	Overall Acc=0.3231
Epoch 14	Loss=1.7587	Train Acc=0.3200	Test Acc=0.3510	Overall Acc=0.3224
Epoch 15	Loss=1.7557	Train Acc=0.3156	Test Acc=0.3510	Overall Acc=0.3183
Epoch 16	Loss=1.7528	Train Acc=0.3116	Test Acc=0.3462	Overall Acc=0.3143
Epoch 17	Loss=1.7498	Train Acc=0.3100	Test Acc=0.3462	Overall Acc=0.3128
Epoch 18	Loss=1.7467	Train Acc=0.3092	Test Acc=0.3413	Overall Acc=0.3117
Epoch 19	Loss=1.7436	Train Acc=0.3084	Test Acc=0.3413	Overall Acc=0.3109
Epoch 20	Loss=1.7404	Train Acc=0.3072	Test Acc=0.3413	Overall Acc=0.3098
Epoch 21	Loss=1.7372	Train Acc=0.3076	Test Acc=0.3462	Overall Acc=0.3106
Epoch 22	Loss=1.7339	Train Acc=0.3068	Test Acc=0.3510	Overall Acc=0.3102
Epoch 23	Loss=1.7305	Train Acc=0.3064	Test Acc=0.3462	Overall Acc=0.3095
Epoch 24	Loss=1.7271	Train Acc=0.3060	Test Acc=0.3462	Overall Acc=0.3091
Epoch 25	Loss=1.7236	Train Acc=0.3056	Test Acc=0.3462	Overall Acc=0.3087
Epoch 26	Loss=1.7200	Train Acc=0.3048	Test Acc=0.3462	Overall Acc=0.3080
Epoch 27	Loss=1.7164	Train Acc=0.3044	Test Acc=0.3462	Overall Acc=0.3076
Epoch 28	Loss=1.7126	Train Acc=0.3036	Test Acc=0.3462	Overall Acc=0.3069
Epoch 29	Loss=1.7088	Train Acc=0.3032	Test Acc=0.3462	Overall Acc=0.3065
Epoch 30	Loss=1.7050	Train Acc=0.3028	Test Acc=0.3462	Overall Acc=0.3061
Epoch 31	Loss=1.7010	Train Acc=0.3028	Test Acc=0.3462	Overall Acc=0.3061
Epoch 32	Loss=1.6970	Train Acc=0.3024	Test Acc=0.3462	Overall Acc=0.3058
Epoch 33	Loss=1.6929	Train Acc=0.3024	Test Acc=0.3462	Overall Acc=0.3058
Epoch 34	Loss=1.6887	Train Acc=0.3024	Test Acc=0.3462	Overall Acc=0.3058
Epoch 35	Loss=1.6844	Train Acc=0.3020	Test Acc=0.3462	Overall Acc=0.3054
Epoch 36	Loss=1.6800	Train Acc=0.3016	Test Acc=0.3462	Overall Acc=0.3050
Epoch 37	Loss=1.6756	Train Acc=0.3016	Test Acc=0.3462	Overall Acc=0.3050
Epoch 38	Loss=1.6711	Train Acc=0.3004	Test Acc=0.3462	Overall Acc=0.3039
Epoch 39	Loss=1.6665	Train Acc=0.3004	Test Acc=0.3462	Overall Acc=0.3039
Epoch 40	Loss=1.6618	Train Acc=0.3004	Test Acc=0.3462	Overall Acc=0.3039
Epoch 41	Loss=1.6571	Train Acc=0.3008	Test Acc=0.3462	Overall Acc=0.3043
Epoch 42	Loss=1.6523	Train Acc=0.3008	Test Acc=0.3462	Overall Acc=0.3043
Epoch 43	Loss=1.6474	Train Acc=0.3012	Test Acc=0.3462	Overall Acc=0.3047
Epoch 44	Loss=1.6424	Train Acc=0.3012	Test Acc=0.3462	Overall Acc=0.3047

Epoch 45		Loss=1.6373		Train Acc=0.3012		Test Acc=0.3462		Overall Acc=0.3047
Epoch 46		Loss=1.6322		Train Acc=0.3016		Test Acc=0.3462		Overall Acc=0.3050
Epoch 47		Loss=1.6270		Train Acc=0.3020		Test Acc=0.3462		Overall Acc=0.3054
Epoch 48		Loss=1.6218		Train Acc=0.3020		Test Acc=0.3462		Overall Acc=0.3054
Epoch 49		Loss=1.6165		Train Acc=0.3024		Test Acc=0.3462		Overall Acc=0.3058
Epoch 50		Loss=1.6111		Train Acc=0.3024		Test Acc=0.3462		Overall Acc=0.3058
Epoch 51		Loss=1.6057		Train Acc=0.3036		Test Acc=0.3462		Overall Acc=0.3069
Epoch 52		Loss=1.6002		Train Acc=0.3036		Test Acc=0.3462		Overall Acc=0.3069
Epoch 53		Loss=1.5946		Train Acc=0.3048		Test Acc=0.3462		Overall Acc=0.3080
Epoch 54		Loss=1.5890		Train Acc=0.3052		Test Acc=0.3462		Overall Acc=0.3083
Epoch 55		Loss=1.5833		Train Acc=0.3052		Test Acc=0.3462		Overall Acc=0.3083
Epoch 56		Loss=1.5776		Train Acc=0.3052		Test Acc=0.3510		Overall Acc=0.3087
Epoch 57		Loss=1.5718		Train Acc=0.3072		Test Acc=0.3510		Overall Acc=0.3106
Epoch 58		Loss=1.5660		Train Acc=0.3076		Test Acc=0.3510		Overall Acc=0.3109
Epoch 59		Loss=1.5601		Train Acc=0.3092		Test Acc=0.3510		Overall Acc=0.3124
Epoch 60		Loss=1.5542		Train Acc=0.3116		Test Acc=0.3510		Overall Acc=0.3146
Epoch 61		Loss=1.5482		Train Acc=0.3144		Test Acc=0.3606		Overall Acc=0.3179
Epoch 62		Loss=1.5422		Train Acc=0.3172		Test Acc=0.3606		Overall Acc=0.3205
Epoch 63		Loss=1.5361		Train Acc=0.3208		Test Acc=0.3654		Overall Acc=0.3242
Epoch 64		Loss=1.5300		Train Acc=0.3240		Test Acc=0.3654		Overall Acc=0.3272
Epoch 65		Loss=1.5238		Train Acc=0.3280		Test Acc=0.3654		Overall Acc=0.3309
Epoch 66		Loss=1.5176		Train Acc=0.3372		Test Acc=0.3750		Overall Acc=0.3401
Epoch 67		Loss=1.5113		Train Acc=0.3440		Test Acc=0.3750		Overall Acc=0.3464
Epoch 68		Loss=1.5050		Train Acc=0.3480		Test Acc=0.3990		Overall Acc=0.3519
Epoch 69		Loss=1.4986		Train Acc=0.3572		Test Acc=0.4038		Overall Acc=0.3608
Epoch 70		Loss=1.4922		Train Acc=0.3628		Test Acc=0.4087		Overall Acc=0.3663
Epoch 71		Loss=1.4857		Train Acc=0.3692		Test Acc=0.4135		Overall Acc=0.3726
Epoch 72		Loss=1.4792		Train Acc=0.3768		Test Acc=0.4135		Overall Acc=0.3796
Epoch 73		Loss=1.4727		Train Acc=0.3836		Test Acc=0.4135		Overall Acc=0.3859
Epoch 74		Loss=1.4661		Train Acc=0.3884		Test Acc=0.4135		Overall Acc=0.3903
Epoch 75		Loss=1.4594		Train Acc=0.3972		Test Acc=0.4183		Overall Acc=0.3988
Epoch 76		Loss=1.4527		Train Acc=0.4012		Test Acc=0.4327		Overall Acc=0.4036
Epoch 77		Loss=1.4460		Train Acc=0.4064		Test Acc=0.4375		Overall Acc=0.4088
Epoch 78		Loss=1.4392		Train Acc=0.4128		Test Acc=0.4375		Overall Acc=0.4147
Epoch 79		Loss=1.4323		Train Acc=0.4204		Test Acc=0.4471		Overall Acc=0.4225
Epoch 80		Loss=1.4254		Train Acc=0.4308		Test Acc=0.4567		Overall Acc=0.4328
Epoch 81		Loss=1.4185		Train Acc=0.4364		Test Acc=0.4567		Overall Acc=0.4380
Epoch 82		Loss=1.4115		Train Acc=0.4428		Test Acc=0.4712		Overall Acc=0.4450
Epoch 83		Loss=1.4044		Train Acc=0.4456		Test Acc=0.4760		Overall Acc=0.4479
Epoch 84		Loss=1.3974		Train Acc=0.4520		Test Acc=0.4904		Overall Acc=0.4549
Epoch 85		Loss=1.3902		Train Acc=0.4588		Test Acc=0.4952		Overall Acc=0.4616
Epoch 86		Loss=1.3831		Train Acc=0.4648		Test Acc=0.5096		Overall Acc=0.4682
Epoch 87		Loss=1.3758		Train Acc=0.4724		Test Acc=0.5144		Overall Acc=0.4756
Epoch 88		Loss=1.3686		Train Acc=0.4756		Test Acc=0.5144		Overall Acc=0.4786
Epoch 89		Loss=1.3613		Train Acc=0.4792		Test Acc=0.5240		Overall Acc=0.4826
Epoch 90		Loss=1.3539		Train Acc=0.4844		Test Acc=0.5337		Overall Acc=0.4882
Epoch 91		Loss=1.3465		Train Acc=0.4896		Test Acc=0.5337		Overall Acc=0.4930
Epoch 92		Loss=1.3391		Train Acc=0.4920		Test Acc=0.5385		Overall Acc=0.4956

Epoch 93		Loss=1.3316		Train Acc=0.4968		Test Acc=0.5385		Overall Acc=0.5000
Epoch 94		Loss=1.3241		Train Acc=0.5024		Test Acc=0.5385		Overall Acc=0.5052
Epoch 95		Loss=1.3165		Train Acc=0.5052		Test Acc=0.5481		Overall Acc=0.5085
Epoch 96		Loss=1.3090		Train Acc=0.5104		Test Acc=0.5625		Overall Acc=0.5144
Epoch 97		Loss=1.3013		Train Acc=0.5164		Test Acc=0.5673		Overall Acc=0.5203
Epoch 98		Loss=1.2937		Train Acc=0.5248		Test Acc=0.5769		Overall Acc=0.5288
Epoch 99		Loss=1.2860		Train Acc=0.5312		Test Acc=0.5769		Overall Acc=0.5347
Epoch 100		Loss=1.2783		Train Acc=0.5408		Test Acc=0.5817		Overall Acc=0.5439
Epoch 101		Loss=1.2705		Train Acc=0.5500		Test Acc=0.5962		Overall Acc=0.5535
Epoch 102		Loss=1.2628		Train Acc=0.5548		Test Acc=0.6154		Overall Acc=0.5595
Epoch 103		Loss=1.2550		Train Acc=0.5664		Test Acc=0.6298		Overall Acc=0.5713
Epoch 104		Loss=1.2471		Train Acc=0.5724		Test Acc=0.6346		Overall Acc=0.5772
Epoch 105		Loss=1.2393		Train Acc=0.5780		Test Acc=0.6394		Overall Acc=0.5827
Epoch 106		Loss=1.2314		Train Acc=0.5828		Test Acc=0.6538		Overall Acc=0.5883
Epoch 107		Loss=1.2235		Train Acc=0.5912		Test Acc=0.6635		Overall Acc=0.5968
Epoch 108		Loss=1.2156		Train Acc=0.6004		Test Acc=0.6683		Overall Acc=0.6056
Epoch 109		Loss=1.2077		Train Acc=0.6080		Test Acc=0.6731		Overall Acc=0.6130
Epoch 110		Loss=1.1997		Train Acc=0.6164		Test Acc=0.6779		Overall Acc=0.6211
Epoch 111		Loss=1.1918		Train Acc=0.6224		Test Acc=0.6779		Overall Acc=0.6267
Epoch 112		Loss=1.1838		Train Acc=0.6292		Test Acc=0.6875		Overall Acc=0.6337
Epoch 113		Loss=1.1758		Train Acc=0.6352		Test Acc=0.6923		Overall Acc=0.6396
Epoch 114		Loss=1.1679		Train Acc=0.6444		Test Acc=0.7067		Overall Acc=0.6492
Epoch 115		Loss=1.1599		Train Acc=0.6516		Test Acc=0.7067		Overall Acc=0.6558
Epoch 116		Loss=1.1519		Train Acc=0.6564		Test Acc=0.7067		Overall Acc=0.6603
Epoch 117		Loss=1.1439		Train Acc=0.6624		Test Acc=0.7115		Overall Acc=0.6662
Epoch 118		Loss=1.1359		Train Acc=0.6704		Test Acc=0.7163		Overall Acc=0.6739
Epoch 119		Loss=1.1279		Train Acc=0.6752		Test Acc=0.7212		Overall Acc=0.6787
Epoch 120		Loss=1.1199		Train Acc=0.6828		Test Acc=0.7212		Overall

Acc=0.6857
 Epoch 121 | Loss=1.1119 | Train Acc=0.6880 | Test Acc=0.7212 | Overall
 Acc=0.6905
 Epoch 122 | Loss=1.1039 | Train Acc=0.6976 | Test Acc=0.7308 | Overall
 Acc=0.7001
 Epoch 123 | Loss=1.0959 | Train Acc=0.7032 | Test Acc=0.7356 | Overall
 Acc=0.7057
 Epoch 124 | Loss=1.0879 | Train Acc=0.7100 | Test Acc=0.7404 | Overall
 Acc=0.7123
 Epoch 125 | Loss=1.0800 | Train Acc=0.7172 | Test Acc=0.7500 | Overall
 Acc=0.7197
 Epoch 126 | Loss=1.0720 | Train Acc=0.7268 | Test Acc=0.7500 | Overall
 Acc=0.7286
 Epoch 127 | Loss=1.0641 | Train Acc=0.7348 | Test Acc=0.7500 | Overall
 Acc=0.7360
 Epoch 128 | Loss=1.0561 | Train Acc=0.7404 | Test Acc=0.7548 | Overall
 Acc=0.7415
 Epoch 129 | Loss=1.0482 | Train Acc=0.7448 | Test Acc=0.7596 | Overall
 Acc=0.7459
 Epoch 130 | Loss=1.0403 | Train Acc=0.7492 | Test Acc=0.7692 | Overall
 Acc=0.7507
 Epoch 131 | Loss=1.0324 | Train Acc=0.7560 | Test Acc=0.7788 | Overall
 Acc=0.7578
 Epoch 132 | Loss=1.0246 | Train Acc=0.7584 | Test Acc=0.7837 | Overall
 Acc=0.7603
 Epoch 133 | Loss=1.0167 | Train Acc=0.7612 | Test Acc=0.7885 | Overall
 Acc=0.7633
 Epoch 134 | Loss=1.0089 | Train Acc=0.7668 | Test Acc=0.7981 | Overall
 Acc=0.7692
 Epoch 135 | Loss=1.0012 | Train Acc=0.7728 | Test Acc=0.8029 | Overall
 Acc=0.7751
 Epoch 136 | Loss=0.9934 | Train Acc=0.7776 | Test Acc=0.8173 | Overall
 Acc=0.7806
 Epoch 137 | Loss=0.9857 | Train Acc=0.7812 | Test Acc=0.8173 | Overall
 Acc=0.7840
 Epoch 138 | Loss=0.9780 | Train Acc=0.7832 | Test Acc=0.8221 | Overall
 Acc=0.7862
 Epoch 139 | Loss=0.9703 | Train Acc=0.7860 | Test Acc=0.8269 | Overall
 Acc=0.7891
 Epoch 140 | Loss=0.9626 | Train Acc=0.7912 | Test Acc=0.8365 | Overall
 Acc=0.7947
 Epoch 141 | Loss=0.9550 | Train Acc=0.7956 | Test Acc=0.8365 | Overall
 Acc=0.7987
 Epoch 142 | Loss=0.9475 | Train Acc=0.8000 | Test Acc=0.8413 | Overall
 Acc=0.8032
 Epoch 143 | Loss=0.9399 | Train Acc=0.8024 | Test Acc=0.8413 | Overall
 Acc=0.8054
 Epoch 144 | Loss=0.9324 | Train Acc=0.8052 | Test Acc=0.8510 | Overall

Acc=0.8087
 Epoch 145 | Loss=0.9249 | Train Acc=0.8088 | Test Acc=0.8558 | Overall
 Acc=0.8124
 Epoch 146 | Loss=0.9175 | Train Acc=0.8116 | Test Acc=0.8558 | Overall
 Acc=0.8150
 Epoch 147 | Loss=0.9101 | Train Acc=0.8120 | Test Acc=0.8606 | Overall
 Acc=0.8157
 Epoch 148 | Loss=0.9028 | Train Acc=0.8144 | Test Acc=0.8606 | Overall
 Acc=0.8179
 Epoch 149 | Loss=0.8954 | Train Acc=0.8172 | Test Acc=0.8606 | Overall
 Acc=0.8205
 Epoch 150 | Loss=0.8882 | Train Acc=0.8192 | Test Acc=0.8654 | Overall
 Acc=0.8227
 Epoch 151 | Loss=0.8809 | Train Acc=0.8228 | Test Acc=0.8654 | Overall
 Acc=0.8261
 Epoch 152 | Loss=0.8737 | Train Acc=0.8244 | Test Acc=0.8654 | Overall
 Acc=0.8275
 Epoch 153 | Loss=0.8666 | Train Acc=0.8268 | Test Acc=0.8606 | Overall
 Acc=0.8294
 Epoch 154 | Loss=0.8595 | Train Acc=0.8284 | Test Acc=0.8702 | Overall
 Acc=0.8316
 Epoch 155 | Loss=0.8524 | Train Acc=0.8304 | Test Acc=0.8702 | Overall
 Acc=0.8335
 Epoch 156 | Loss=0.8454 | Train Acc=0.8336 | Test Acc=0.8702 | Overall
 Acc=0.8364
 Epoch 157 | Loss=0.8385 | Train Acc=0.8380 | Test Acc=0.8702 | Overall
 Acc=0.8405
 Epoch 158 | Loss=0.8315 | Train Acc=0.8392 | Test Acc=0.8750 | Overall
 Acc=0.8419
 Epoch 159 | Loss=0.8247 | Train Acc=0.8408 | Test Acc=0.8798 | Overall
 Acc=0.8438
 Epoch 160 | Loss=0.8178 | Train Acc=0.8416 | Test Acc=0.8798 | Overall
 Acc=0.8445
 Epoch 161 | Loss=0.8111 | Train Acc=0.8444 | Test Acc=0.8798 | Overall
 Acc=0.8471
 Epoch 162 | Loss=0.8043 | Train Acc=0.8456 | Test Acc=0.8798 | Overall
 Acc=0.8482
 Epoch 163 | Loss=0.7977 | Train Acc=0.8480 | Test Acc=0.8798 | Overall
 Acc=0.8504
 Epoch 164 | Loss=0.7910 | Train Acc=0.8496 | Test Acc=0.8798 | Overall
 Acc=0.8519
 Epoch 165 | Loss=0.7844 | Train Acc=0.8524 | Test Acc=0.8798 | Overall
 Acc=0.8545
 Epoch 166 | Loss=0.7779 | Train Acc=0.8540 | Test Acc=0.8846 | Overall
 Acc=0.8564
 Epoch 167 | Loss=0.7714 | Train Acc=0.8544 | Test Acc=0.8894 | Overall
 Acc=0.8571
 Epoch 168 | Loss=0.7650 | Train Acc=0.8552 | Test Acc=0.8894 | Overall

Acc=0.8578
 Epoch 169 | Loss=0.7586 | Train Acc=0.8556 | Test Acc=0.8894 | Overall
 Acc=0.8582
 Epoch 170 | Loss=0.7523 | Train Acc=0.8576 | Test Acc=0.8894 | Overall
 Acc=0.8600
 Epoch 171 | Loss=0.7460 | Train Acc=0.8584 | Test Acc=0.8894 | Overall
 Acc=0.8608
 Epoch 172 | Loss=0.7398 | Train Acc=0.8592 | Test Acc=0.8894 | Overall
 Acc=0.8615
 Epoch 173 | Loss=0.7336 | Train Acc=0.8592 | Test Acc=0.8894 | Overall
 Acc=0.8615
 Epoch 174 | Loss=0.7274 | Train Acc=0.8612 | Test Acc=0.8942 | Overall
 Acc=0.8637
 Epoch 175 | Loss=0.7214 | Train Acc=0.8632 | Test Acc=0.8990 | Overall
 Acc=0.8660
 Epoch 176 | Loss=0.7153 | Train Acc=0.8644 | Test Acc=0.8990 | Overall
 Acc=0.8671
 Epoch 177 | Loss=0.7094 | Train Acc=0.8668 | Test Acc=0.9038 | Overall
 Acc=0.8696
 Epoch 178 | Loss=0.7034 | Train Acc=0.8680 | Test Acc=0.9038 | Overall
 Acc=0.8708
 Epoch 179 | Loss=0.6976 | Train Acc=0.8684 | Test Acc=0.9038 | Overall
 Acc=0.8711
 Epoch 180 | Loss=0.6917 | Train Acc=0.8696 | Test Acc=0.8990 | Overall
 Acc=0.8719
 Epoch 181 | Loss=0.6860 | Train Acc=0.8700 | Test Acc=0.9038 | Overall
 Acc=0.8726
 Epoch 182 | Loss=0.6802 | Train Acc=0.8696 | Test Acc=0.9038 | Overall
 Acc=0.8722
 Epoch 183 | Loss=0.6746 | Train Acc=0.8704 | Test Acc=0.9038 | Overall
 Acc=0.8730
 Epoch 184 | Loss=0.6690 | Train Acc=0.8716 | Test Acc=0.9038 | Overall
 Acc=0.8741
 Epoch 185 | Loss=0.6634 | Train Acc=0.8728 | Test Acc=0.9087 | Overall
 Acc=0.8756
 Epoch 186 | Loss=0.6579 | Train Acc=0.8732 | Test Acc=0.9087 | Overall
 Acc=0.8759
 Epoch 187 | Loss=0.6524 | Train Acc=0.8752 | Test Acc=0.9087 | Overall
 Acc=0.8778
 Epoch 188 | Loss=0.6470 | Train Acc=0.8760 | Test Acc=0.9087 | Overall
 Acc=0.8785
 Epoch 189 | Loss=0.6417 | Train Acc=0.8760 | Test Acc=0.9087 | Overall
 Acc=0.8785
 Epoch 190 | Loss=0.6364 | Train Acc=0.8764 | Test Acc=0.9135 | Overall
 Acc=0.8792
 Epoch 191 | Loss=0.6311 | Train Acc=0.8772 | Test Acc=0.9135 | Overall
 Acc=0.8800
 Epoch 192 | Loss=0.6259 | Train Acc=0.8764 | Test Acc=0.9183 | Overall

Acc=0.8796
 Epoch 193 | Loss=0.6208 | Train Acc=0.8780 | Test Acc=0.9135 | Overall
 Acc=0.8807
 Epoch 194 | Loss=0.6157 | Train Acc=0.8788 | Test Acc=0.9135 | Overall
 Acc=0.8815
 Epoch 195 | Loss=0.6106 | Train Acc=0.8800 | Test Acc=0.9087 | Overall
 Acc=0.8822
 Epoch 196 | Loss=0.6056 | Train Acc=0.8816 | Test Acc=0.9087 | Overall
 Acc=0.8837
 Epoch 197 | Loss=0.6007 | Train Acc=0.8808 | Test Acc=0.9087 | Overall
 Acc=0.8829
 Epoch 198 | Loss=0.5958 | Train Acc=0.8828 | Test Acc=0.9087 | Overall
 Acc=0.8848
 Epoch 199 | Loss=0.5909 | Train Acc=0.8832 | Test Acc=0.9087 | Overall
 Acc=0.8852
 Epoch 200 | Loss=0.5861 | Train Acc=0.8844 | Test Acc=0.9038 | Overall
 Acc=0.8859
 Epoch 201 | Loss=0.5814 | Train Acc=0.8844 | Test Acc=0.9038 | Overall
 Acc=0.8859
 Epoch 202 | Loss=0.5767 | Train Acc=0.8848 | Test Acc=0.9038 | Overall
 Acc=0.8863
 Epoch 203 | Loss=0.5720 | Train Acc=0.8856 | Test Acc=0.9038 | Overall
 Acc=0.8870
 Epoch 204 | Loss=0.5674 | Train Acc=0.8868 | Test Acc=0.9038 | Overall
 Acc=0.8881
 Epoch 205 | Loss=0.5629 | Train Acc=0.8880 | Test Acc=0.9038 | Overall
 Acc=0.8892
 Epoch 206 | Loss=0.5584 | Train Acc=0.8880 | Test Acc=0.9038 | Overall
 Acc=0.8892
 Epoch 207 | Loss=0.5539 | Train Acc=0.8888 | Test Acc=0.9038 | Overall
 Acc=0.8900
 Epoch 208 | Loss=0.5495 | Train Acc=0.8888 | Test Acc=0.9038 | Overall
 Acc=0.8900
 Epoch 209 | Loss=0.5452 | Train Acc=0.8904 | Test Acc=0.9038 | Overall
 Acc=0.8914
 Epoch 210 | Loss=0.5409 | Train Acc=0.8900 | Test Acc=0.9038 | Overall
 Acc=0.8911
 Epoch 211 | Loss=0.5366 | Train Acc=0.8908 | Test Acc=0.8990 | Overall
 Acc=0.8914
 Epoch 212 | Loss=0.5324 | Train Acc=0.8916 | Test Acc=0.8990 | Overall
 Acc=0.8922
 Epoch 213 | Loss=0.5283 | Train Acc=0.8924 | Test Acc=0.8990 | Overall
 Acc=0.8929
 Epoch 214 | Loss=0.5241 | Train Acc=0.8924 | Test Acc=0.9038 | Overall
 Acc=0.8933
 Epoch 215 | Loss=0.5201 | Train Acc=0.8928 | Test Acc=0.9038 | Overall
 Acc=0.8936
 Epoch 216 | Loss=0.5161 | Train Acc=0.8932 | Test Acc=0.9038 | Overall

Acc=0.8940
 Epoch 217 | Loss=0.5121 | Train Acc=0.8928 | Test Acc=0.9038 | Overall
 Acc=0.8936
 Epoch 218 | Loss=0.5081 | Train Acc=0.8928 | Test Acc=0.9038 | Overall
 Acc=0.8936
 Epoch 219 | Loss=0.5043 | Train Acc=0.8932 | Test Acc=0.9038 | Overall
 Acc=0.8940
 Epoch 220 | Loss=0.5004 | Train Acc=0.8936 | Test Acc=0.9038 | Overall
 Acc=0.8944
 Epoch 221 | Loss=0.4966 | Train Acc=0.8952 | Test Acc=0.9038 | Overall
 Acc=0.8959
 Epoch 222 | Loss=0.4929 | Train Acc=0.8960 | Test Acc=0.9038 | Overall
 Acc=0.8966
 Epoch 223 | Loss=0.4892 | Train Acc=0.8964 | Test Acc=0.9038 | Overall
 Acc=0.8970
 Epoch 224 | Loss=0.4855 | Train Acc=0.8976 | Test Acc=0.9038 | Overall
 Acc=0.8981
 Epoch 225 | Loss=0.4819 | Train Acc=0.8980 | Test Acc=0.9038 | Overall
 Acc=0.8984
 Epoch 226 | Loss=0.4783 | Train Acc=0.8984 | Test Acc=0.9038 | Overall
 Acc=0.8988
 Epoch 227 | Loss=0.4748 | Train Acc=0.8988 | Test Acc=0.9038 | Overall
 Acc=0.8992
 Epoch 228 | Loss=0.4713 | Train Acc=0.8996 | Test Acc=0.9087 | Overall
 Acc=0.9003
 Epoch 229 | Loss=0.4679 | Train Acc=0.8996 | Test Acc=0.9087 | Overall
 Acc=0.9003
 Epoch 230 | Loss=0.4645 | Train Acc=0.9008 | Test Acc=0.9087 | Overall
 Acc=0.9014
 Epoch 231 | Loss=0.4611 | Train Acc=0.9016 | Test Acc=0.9087 | Overall
 Acc=0.9021
 Epoch 232 | Loss=0.4578 | Train Acc=0.9016 | Test Acc=0.9087 | Overall
 Acc=0.9021
 Epoch 233 | Loss=0.4545 | Train Acc=0.9020 | Test Acc=0.9135 | Overall
 Acc=0.9029
 Epoch 234 | Loss=0.4513 | Train Acc=0.9024 | Test Acc=0.9135 | Overall
 Acc=0.9032
 Epoch 235 | Loss=0.4481 | Train Acc=0.9024 | Test Acc=0.9135 | Overall
 Acc=0.9032
 Epoch 236 | Loss=0.4449 | Train Acc=0.9020 | Test Acc=0.9135 | Overall
 Acc=0.9029
 Epoch 237 | Loss=0.4418 | Train Acc=0.9020 | Test Acc=0.9135 | Overall
 Acc=0.9029
 Epoch 238 | Loss=0.4387 | Train Acc=0.9020 | Test Acc=0.9135 | Overall
 Acc=0.9029
 Epoch 239 | Loss=0.4357 | Train Acc=0.9028 | Test Acc=0.9135 | Overall
 Acc=0.9036
 Epoch 240 | Loss=0.4327 | Train Acc=0.9036 | Test Acc=0.9135 | Overall

Acc=0.9044
 Epoch 241 | Loss=0.4297 | Train Acc=0.9048 | Test Acc=0.9135 | Overall
 Acc=0.9055
 Epoch 242 | Loss=0.4268 | Train Acc=0.9056 | Test Acc=0.9135 | Overall
 Acc=0.9062
 Epoch 243 | Loss=0.4239 | Train Acc=0.9060 | Test Acc=0.9135 | Overall
 Acc=0.9066
 Epoch 244 | Loss=0.4210 | Train Acc=0.9064 | Test Acc=0.9135 | Overall
 Acc=0.9069
 Epoch 245 | Loss=0.4182 | Train Acc=0.9084 | Test Acc=0.9135 | Overall
 Acc=0.9088
 Epoch 246 | Loss=0.4154 | Train Acc=0.9084 | Test Acc=0.9135 | Overall
 Acc=0.9088
 Epoch 247 | Loss=0.4127 | Train Acc=0.9084 | Test Acc=0.9135 | Overall
 Acc=0.9088
 Epoch 248 | Loss=0.4099 | Train Acc=0.9080 | Test Acc=0.9135 | Overall
 Acc=0.9084
 Epoch 249 | Loss=0.4073 | Train Acc=0.9084 | Test Acc=0.9135 | Overall
 Acc=0.9088
 Epoch 250 | Loss=0.4046 | Train Acc=0.9092 | Test Acc=0.9135 | Overall
 Acc=0.9095
 Epoch 251 | Loss=0.4020 | Train Acc=0.9100 | Test Acc=0.9135 | Overall
 Acc=0.9103
 Epoch 252 | Loss=0.3994 | Train Acc=0.9108 | Test Acc=0.9135 | Overall
 Acc=0.9110
 Epoch 253 | Loss=0.3969 | Train Acc=0.9104 | Test Acc=0.9135 | Overall
 Acc=0.9106
 Epoch 254 | Loss=0.3943 | Train Acc=0.9108 | Test Acc=0.9135 | Overall
 Acc=0.9110
 Epoch 255 | Loss=0.3918 | Train Acc=0.9112 | Test Acc=0.9135 | Overall
 Acc=0.9114
 Epoch 256 | Loss=0.3894 | Train Acc=0.9116 | Test Acc=0.9087 | Overall
 Acc=0.9114
 Epoch 257 | Loss=0.3870 | Train Acc=0.9116 | Test Acc=0.9087 | Overall
 Acc=0.9114
 Epoch 258 | Loss=0.3846 | Train Acc=0.9140 | Test Acc=0.9087 | Overall
 Acc=0.9136
 Epoch 259 | Loss=0.3822 | Train Acc=0.9144 | Test Acc=0.9087 | Overall
 Acc=0.9140
 Epoch 260 | Loss=0.3799 | Train Acc=0.9152 | Test Acc=0.9135 | Overall
 Acc=0.9151
 Epoch 261 | Loss=0.3776 | Train Acc=0.9152 | Test Acc=0.9135 | Overall
 Acc=0.9151
 Epoch 262 | Loss=0.3753 | Train Acc=0.9152 | Test Acc=0.9135 | Overall
 Acc=0.9151
 Epoch 263 | Loss=0.3730 | Train Acc=0.9152 | Test Acc=0.9135 | Overall
 Acc=0.9151
 Epoch 264 | Loss=0.3708 | Train Acc=0.9152 | Test Acc=0.9135 | Overall

Acc=0.9151
 Epoch 265 | Loss=0.3686 | Train Acc=0.9156 | Test Acc=0.9135 | Overall
 Acc=0.9154
 Epoch 266 | Loss=0.3664 | Train Acc=0.9160 | Test Acc=0.9135 | Overall
 Acc=0.9158
 Epoch 267 | Loss=0.3643 | Train Acc=0.9160 | Test Acc=0.9135 | Overall
 Acc=0.9158
 Epoch 268 | Loss=0.3622 | Train Acc=0.9168 | Test Acc=0.9135 | Overall
 Acc=0.9165
 Epoch 269 | Loss=0.3601 | Train Acc=0.9168 | Test Acc=0.9135 | Overall
 Acc=0.9165
 Epoch 270 | Loss=0.3580 | Train Acc=0.9168 | Test Acc=0.9135 | Overall
 Acc=0.9165
 Epoch 271 | Loss=0.3560 | Train Acc=0.9172 | Test Acc=0.9135 | Overall
 Acc=0.9169
 Epoch 272 | Loss=0.3540 | Train Acc=0.9172 | Test Acc=0.9135 | Overall
 Acc=0.9169
 Epoch 273 | Loss=0.3520 | Train Acc=0.9188 | Test Acc=0.9135 | Overall
 Acc=0.9184
 Epoch 274 | Loss=0.3500 | Train Acc=0.9188 | Test Acc=0.9135 | Overall
 Acc=0.9184
 Epoch 275 | Loss=0.3481 | Train Acc=0.9184 | Test Acc=0.9135 | Overall
 Acc=0.9180
 Epoch 276 | Loss=0.3462 | Train Acc=0.9188 | Test Acc=0.9135 | Overall
 Acc=0.9184
 Epoch 277 | Loss=0.3443 | Train Acc=0.9188 | Test Acc=0.9135 | Overall
 Acc=0.9184
 Epoch 278 | Loss=0.3424 | Train Acc=0.9192 | Test Acc=0.9135 | Overall
 Acc=0.9188
 Epoch 279 | Loss=0.3406 | Train Acc=0.9192 | Test Acc=0.9135 | Overall
 Acc=0.9188
 Epoch 280 | Loss=0.3388 | Train Acc=0.9196 | Test Acc=0.9135 | Overall
 Acc=0.9191
 Epoch 281 | Loss=0.3370 | Train Acc=0.9196 | Test Acc=0.9135 | Overall
 Acc=0.9191
 Epoch 282 | Loss=0.3352 | Train Acc=0.9196 | Test Acc=0.9135 | Overall
 Acc=0.9191
 Epoch 283 | Loss=0.3334 | Train Acc=0.9196 | Test Acc=0.9135 | Overall
 Acc=0.9191
 Epoch 284 | Loss=0.3317 | Train Acc=0.9200 | Test Acc=0.9135 | Overall
 Acc=0.9195
 Epoch 285 | Loss=0.3299 | Train Acc=0.9204 | Test Acc=0.9135 | Overall
 Acc=0.9199
 Epoch 286 | Loss=0.3282 | Train Acc=0.9204 | Test Acc=0.9135 | Overall
 Acc=0.9199
 Epoch 287 | Loss=0.3266 | Train Acc=0.9208 | Test Acc=0.9135 | Overall
 Acc=0.9202
 Epoch 288 | Loss=0.3249 | Train Acc=0.9204 | Test Acc=0.9135 | Overall

Acc=0.9199
 Epoch 289 | Loss=0.3233 | Train Acc=0.9204 | Test Acc=0.9135 | Overall
 Acc=0.9199
 Epoch 290 | Loss=0.3216 | Train Acc=0.9204 | Test Acc=0.9135 | Overall
 Acc=0.9199
 Epoch 291 | Loss=0.3200 | Train Acc=0.9204 | Test Acc=0.9135 | Overall
 Acc=0.9199
 Epoch 292 | Loss=0.3185 | Train Acc=0.9208 | Test Acc=0.9135 | Overall
 Acc=0.9202
 Epoch 293 | Loss=0.3169 | Train Acc=0.9208 | Test Acc=0.9135 | Overall
 Acc=0.9202
 Epoch 294 | Loss=0.3153 | Train Acc=0.9212 | Test Acc=0.9135 | Overall
 Acc=0.9206
 Epoch 295 | Loss=0.3138 | Train Acc=0.9216 | Test Acc=0.9135 | Overall
 Acc=0.9210
 Epoch 296 | Loss=0.3123 | Train Acc=0.9224 | Test Acc=0.9135 | Overall
 Acc=0.9217
 Epoch 297 | Loss=0.3108 | Train Acc=0.9224 | Test Acc=0.9135 | Overall
 Acc=0.9217
 Epoch 298 | Loss=0.3093 | Train Acc=0.9224 | Test Acc=0.9135 | Overall
 Acc=0.9217
 Epoch 299 | Loss=0.3078 | Train Acc=0.9224 | Test Acc=0.9135 | Overall
 Acc=0.9217
 Epoch 300 | Loss=0.3064 | Train Acc=0.9228 | Test Acc=0.9135 | Overall
 Acc=0.9221
 Epoch 301 | Loss=0.3049 | Train Acc=0.9228 | Test Acc=0.9135 | Overall
 Acc=0.9221
 Epoch 302 | Loss=0.3035 | Train Acc=0.9228 | Test Acc=0.9135 | Overall
 Acc=0.9221
 Epoch 303 | Loss=0.3021 | Train Acc=0.9228 | Test Acc=0.9183 | Overall
 Acc=0.9225
 Epoch 304 | Loss=0.3007 | Train Acc=0.9228 | Test Acc=0.9183 | Overall
 Acc=0.9225
 Epoch 305 | Loss=0.2994 | Train Acc=0.9232 | Test Acc=0.9183 | Overall
 Acc=0.9228
 Epoch 306 | Loss=0.2980 | Train Acc=0.9236 | Test Acc=0.9183 | Overall
 Acc=0.9232
 Epoch 307 | Loss=0.2966 | Train Acc=0.9236 | Test Acc=0.9183 | Overall
 Acc=0.9232
 Epoch 308 | Loss=0.2953 | Train Acc=0.9240 | Test Acc=0.9183 | Overall
 Acc=0.9236
 Epoch 309 | Loss=0.2940 | Train Acc=0.9244 | Test Acc=0.9183 | Overall
 Acc=0.9239
 Epoch 310 | Loss=0.2927 | Train Acc=0.9248 | Test Acc=0.9183 | Overall
 Acc=0.9243
 Epoch 311 | Loss=0.2914 | Train Acc=0.9252 | Test Acc=0.9183 | Overall
 Acc=0.9247
 Epoch 312 | Loss=0.2901 | Train Acc=0.9264 | Test Acc=0.9183 | Overall

Acc=0.9258
 Epoch 313 | Loss=0.2889 | Train Acc=0.9264 | Test Acc=0.9183 | Overall
 Acc=0.9258
 Epoch 314 | Loss=0.2876 | Train Acc=0.9264 | Test Acc=0.9183 | Overall
 Acc=0.9258
 Epoch 315 | Loss=0.2864 | Train Acc=0.9264 | Test Acc=0.9183 | Overall
 Acc=0.9258
 Epoch 316 | Loss=0.2851 | Train Acc=0.9264 | Test Acc=0.9183 | Overall
 Acc=0.9258
 Epoch 317 | Loss=0.2839 | Train Acc=0.9268 | Test Acc=0.9183 | Overall
 Acc=0.9261
 Epoch 318 | Loss=0.2827 | Train Acc=0.9260 | Test Acc=0.9183 | Overall
 Acc=0.9254
 Epoch 319 | Loss=0.2815 | Train Acc=0.9260 | Test Acc=0.9183 | Overall
 Acc=0.9254
 Epoch 320 | Loss=0.2803 | Train Acc=0.9260 | Test Acc=0.9183 | Overall
 Acc=0.9254
 Epoch 321 | Loss=0.2792 | Train Acc=0.9264 | Test Acc=0.9183 | Overall
 Acc=0.9258
 Epoch 322 | Loss=0.2780 | Train Acc=0.9264 | Test Acc=0.9183 | Overall
 Acc=0.9258
 Epoch 323 | Loss=0.2768 | Train Acc=0.9268 | Test Acc=0.9183 | Overall
 Acc=0.9261
 Epoch 324 | Loss=0.2757 | Train Acc=0.9268 | Test Acc=0.9183 | Overall
 Acc=0.9261
 Epoch 325 | Loss=0.2746 | Train Acc=0.9268 | Test Acc=0.9183 | Overall
 Acc=0.9261
 Epoch 326 | Loss=0.2735 | Train Acc=0.9268 | Test Acc=0.9183 | Overall
 Acc=0.9261
 Epoch 327 | Loss=0.2724 | Train Acc=0.9272 | Test Acc=0.9183 | Overall
 Acc=0.9265
 Epoch 328 | Loss=0.2713 | Train Acc=0.9280 | Test Acc=0.9183 | Overall
 Acc=0.9273
 Epoch 329 | Loss=0.2702 | Train Acc=0.9280 | Test Acc=0.9183 | Overall
 Acc=0.9273
 Epoch 330 | Loss=0.2691 | Train Acc=0.9280 | Test Acc=0.9183 | Overall
 Acc=0.9273
 Epoch 331 | Loss=0.2680 | Train Acc=0.9280 | Test Acc=0.9183 | Overall
 Acc=0.9273
 Epoch 332 | Loss=0.2670 | Train Acc=0.9284 | Test Acc=0.9183 | Overall
 Acc=0.9276
 Epoch 333 | Loss=0.2659 | Train Acc=0.9284 | Test Acc=0.9183 | Overall
 Acc=0.9276
 Epoch 334 | Loss=0.2649 | Train Acc=0.9280 | Test Acc=0.9183 | Overall
 Acc=0.9273
 Epoch 335 | Loss=0.2638 | Train Acc=0.9280 | Test Acc=0.9183 | Overall
 Acc=0.9273
 Epoch 336 | Loss=0.2628 | Train Acc=0.9280 | Test Acc=0.9183 | Overall

```

Acc=0.9273
Epoch 337 | Loss=0.2618 | Train Acc=0.9280 | Test Acc=0.9183 | Overall
Acc=0.9273
Epoch 338 | Loss=0.2608 | Train Acc=0.9276 | Test Acc=0.9183 | Overall
Acc=0.9269
Epoch 339 | Loss=0.2598 | Train Acc=0.9272 | Test Acc=0.9183 | Overall
Acc=0.9265

```

```

[253]: plt.figure(figsize=(10,4))

plt.subplot(1, 2, 1)
plt.plot(losses, label='Loss')
plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(train_accs, label='Train')
plt.plot(test_accs, label='Test')
plt.plot(overall_accs, label='Overall')
plt.title('Accuracy Curves')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.suptitle('CORA GAT Training (128 hidden, 340 epochs)')
plt.show()

```



Below is a minimal GAT setup for semi-supervised node classification in the CORA citation network.

We feed **all** node features and edges into the model (because the entire structure helps each node’s representation), but only a subset of nodes are “revealed” as training examples. Then, I compute a cross-entropy loss that depends only on those training labels, mirroring the approach used in the Karate Club example.

Once trained, we measure classification accuracy on: - **Train nodes** (the revealed subset), - **Test nodes** (unseen labels), - **Overall** (all nodes in the graph).

In my experiments, we obtain around **0.9280** accuracy for training nodes and about **0.9183** for test nodes—these are quite strong results for CORA. It shows that letting the model view the full connectivity of the graph, even with many hidden labels, can effectively guide the GAT to learn discriminative node representations in a semi-supervised manner.

Q3.2: How would you use the attention function that you trained in the previous question to perform link prediction? (Answer at the algorithmic level only, a working implementation is not required)

Initially, you would take the previously trained GAT for node classification and extract each node’s embedding from the final (or penultimate) layer. Given the GAT utilized the attention mechanism to aggregate its neighbors, those node embeddings will have the appropriate graph connectivity signals, and thus the embeddings will have the appropriate graph connectivity signals.

Now, given we have the node embeddings, we can conduct a link prediction by computing some similarity metric (e.g., dot product or cosine similarity) between any two nodes’ embeddings. The similarity score will give a sense of how likely it is to connect two nodes. One straightforward decision rule could be to predict a link should the similarity exceed a specified threshold. Alternatively, one could just as easily plug the embeddings into a lightweight decoder (e.g., MLP) that will translate those similarity scores into a probability of link adoption.

7 Open questions

Intro to Graph Adversarial Attacks It is known that deep neural networks for vision can be easily fooled by **imperceptible** transformations of the input:

Figure credit: <https://openai.com/blog/adversarial-example-research/>, adapted from [Goodfellow et al. \(2015\)](#) (page 3).

In this setting, we often define imperceptibility, i.e., the attacker budget, as some neighborhood of the input image. For example, assuming the input image x has pixel values in $[0, 1]$, we might look for adversarial attacks inside the l_∞ ball of radius 0.1, i.e., *adversarial inputs* x' s.t. $\|x - x'\|_\infty < 0.1$ on which the network makes a mistake.

Unsurprisingly, this extends to various other domains (language, sound, etc.), as well as graphs – tweaking the features of a node or adding/removing an edge can easily confuse a GNN model:

Figure credit: <https://github.com/DSE-MSU/DeepRobust> (adversary_examples/graph_attack_example.png)

However, there are several challenges specific to graphs that make adversarial attacks and defenses interesting for GNNs:

1. **The graphs are discrete** – as pixel values are continuous, perturbing them almost always leads to a valid input image. However, perturbing adjacency matrices is more involved as

they need to remain discrete, which also makes gradient-based methods significantly harder to directly apply.

2. **The nodes in the graph are not independent** – by the nature of GNNs, perturbing the features of one node might affect the prediction of another. This led to new notions such as *influencer attacks* where we attempt to make a GNN misclassify a node without directly manipulating it in any way.
3. **It is difficult to define imperceptibility** – while in images this comes naturally and can be easily formalized by thinking about l_∞ or l_2 neighborhoods, it is unclear how this generalizes to graphs. Various definitions have been proposed (e.g., the attack should preserve some graph properties such as the degree distribution), but also used to craft *graph purification* procedures.

These unique challenges lead to a very active research field: follow [this repo](#) for most recent papers. We also recommend [DeepRobust](#), which offers implementations of various attack/defense papers.

Q4.1 Re-implement one of the DeepRobust solution in JAX and show the adversarial attack/defense in action on some sample inputs.

Open Graph Benchmark Try applying the techniques you have learned in this class to one of the datasets from the Open Graph Benchmark (OGB).

<https://ogb.stanford.edu/docs/home/>

Q4.2 Choose a task, and design an experimental evaluation based on the metrics recommended by the corresponding OGB evaluator. Report your design choices and how they evolved as you interacted with the dataset. Report performance as it changes with each choice and attempt. Try to motivate why they increased or decreased performance (e.g. increasing depth reduced performance, cleaning/preprocessing the graph increased performance, increasing depth with a different node update increased performance).

Your submission does not have to be competitive with the current leaderboard, but rather show your understanding of the algorithms presented in class and how they interact with real-world scenarios.

8 Summary

In this tutorial, we learned about how to represent directed and undirected graphs in jraph, how to create the basic GNN blocks (GCN + GAT) and how to compose them into GNNs. We also learned about three types of graph predictions tasks (node classification, link prediction and graph classification) and trained a model on each of these tasks.

To practice your knowledge, you can:

- * Create your own graphs, e.g. following the toy graph template and adding nodes, edges, features.
- * Modify the GNN blocks, e.g. by changing the `update_node_fn()` / `update_edge_fn()` functions.
- * Train a model on a different dataset and/or task.