pytorch_geometric_intro

July 23, 2021

1 Pytorch Geometric (PyG)

Pytorch Geometric (PyG) is a geometric deep learning extension library for PyTorch. It consists of various methods for deep learning on graphs and other irregular structures. It implements plenty of graph neural networks from the literature and allows to easily prototype new ones.

Adapted from tutorials and notebooks from https://github.com/rusty1s/pytorch_geometric

1.1 Creating Message Passing Networks

Graph neural networks can be defined in terms of a *neighborhood aggregation* or *message passing* scheme. With $\mathbf{x}_i^{(k-1)} \in \mathbb{R}^F$ denoting node features of node i in layer $(k-1)$ and $\mathbf{e}_{j,i} \in \mathbb{R}^D$ denoting (optional) edge features from node j to node i, message passing graph neural networks can be described as

$$\mathbf{x}_i^{(k)} = \gamma^{(k)} \left(\mathbf{x}_i^{(k-1)}, \square_{j \in \mathcal{N}(i)} \, \phi^{(k)} \left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i} \right) \right),$$

where $\square$ denotes a differentiable, permutation invariant function, *e.g.*, sum, mean or max, and γ and ϕ denote differentiable functions such as MLPs (Multi Layer Perceptrons).

1.2 The "MessagePassing" Base Class

PyTorch Geometric provides the `MessagePassing` base class, which helps in creating such kinds of message passing graph neural networks by automatically taking care of message propagation. The user only has to define the functions ϕ , *i.e.* `message`, and γ , *i.e.* `update`, as well as the aggregation scheme to use, *i.e.* `aggr="add"`, `aggr="mean"` or `aggr="max"`.

This is done with the help of the following methods:

- `MessagePassing(aggr="add", flow="source_to_target", node_dim=-2)`: Defines the aggregation scheme to use (`"add"`, `"mean"` or `"max"`) and the flow direction of message passing (either `"source_to_target"` or `"target_to_source"`). Furthermore, the `node_dim` attribute indicates along which axis to propagate.
- `MessagePassing.propagate(edge_index, ...)`: The initial call to start propagating messages. Takes in the edge indices and all additional data which is needed to construct messages and to update node embeddings.

- `MessagePassing.message(...)`: Constructs messages to node i in analogy to ϕ for each edge in $(j,i) \in \mathcal{E}$ if `flow="source_to_target"` and $(i,j) \in \mathcal{E}$ if `flow="target_to_source"`. Can take any argument which was initially passed to `propagate`. In addition, tensors passed to `propagate` can be mapped to the respective nodes i and j by appending $_i$ or $_j$ to the variable name, *.e.g.* x_i and x_j. Note that we generally refer to i as the central nodes that aggregates information, and refer to j as the neighboring nodes, since this is the most common notation.
- `MessagePassing.update(aggr_out, ...)`: Updates node embeddings in analogy to γ for each node $i \in \mathcal{V}$. Takes in the output of aggregation as first argument and any argument which was initially passed to `MessagePassing.propagate`.

Let us verify this by re-implementing two popular GNN variants, the `GCN layer from Kipf and Welling <https://arxiv.org/abs/1609.02907>_` and the `EdgeConv layer from Wang et al. <https://arxiv.org/abs/1801.07829>_`.

1.3 Implementing the GCN Layer

The GCN layer from Kipf and Welling is a popular GNN that was the first to bring the idea of layerwise convolutions to graphs processing. The layer is mathematically defined as

$$\mathbf{x}_i^{(k)} = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}} \cdot \left(\cdot \mathbf{x}_j^{(k-1)} \right),$$

where neighboring node features are first transformed by a weight matrix , normalized by their degree, and finally summed up. This formula can be divided into the following steps:

1. Add self-loops to the adjacency matrix.
2. Linearly transform node feature matrix.
3. Compute normalization coefficients.
4. Normalize node features in ϕ.
5. Sum up neighboring node features (`"add"` aggregation).

Steps 1-3 are typically computed before message passing takes place. Steps 4-5 can be easily processed using the `MessagePassing` base class. The full layer implementation is shown below:

[11]:
```
# Install required packages.
!pip install -q torch-scatter -f https://pytorch-geometric.com/whl/torch-1.9.
 ↪0+cu102.html
!pip install -q torch-sparse -f https://pytorch-geometric.com/whl/torch-1.9.
 ↪0+cu102.html
!pip install -q torch-geometric

import torch
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import add_self_loops, degree

class myGCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(myGCNConv, self).__init__(aggr='add')  # "Add" aggregation (Step 5).
```

```
        self.lin = torch.nn.Linear(in_channels, out_channels)

    def forward(self, x, edge_index):
        # x has shape [N, in_channels]
        # edge_index has shape [2, E] (sparse adjacency matrix as a list of edges)

        # Step 1: Add self-loops to the adjacency matrix.
        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))

        # Step 2: Linearly transform node feature matrix.
        x = self.lin(x)

        # Step 3: Compute normalization.
        row, col = edge_index
        deg = degree(col, x.size(0), dtype=x.dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        deg_inv_sqrt[deg_inv_sqrt == float('inf')] = 0
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

        # Step 4-5: Start propagating messages.
        return self.propagate(edge_index, x=x, norm=norm)

    def message(self, x_j, norm):
        # x_j has shape [E, out_channels]

        # Step 4: Normalize node features.
        return norm.view(-1, 1) * x_j
```

[12]:
```
# Helper function for visualization.
%matplotlib inline
import torch
import networkx as nx
import matplotlib.pyplot as plt


def visualize(h, color, epoch=None, loss=None):
    plt.figure(figsize=(7,7))
    plt.xticks([])
    plt.yticks([])

    if torch.is_tensor(h):
        h = h.detach().cpu().numpy()
        plt.scatter(h[:, 0], h[:, 1], s=140, c=color, cmap="Set2")
        if epoch is not None and loss is not None:
            plt.xlabel(f'Epoch: {epoch}, Loss: {loss.item():.4f}', fontsize=16)
    else:
        nx.draw_networkx(G, pos=nx.spring_layout(G, seed=42), with_labels=False,
```

```
                            node_color=color, cmap="Set2")
    plt.show()
```

2 Example: node classification

Following Kipf et al. (2017), let's dive into the world of GNNs by looking at a simple graph-structured example, the well-known **Zachary's karate club network**. This graph describes a social network of 34 members of a karate club and documents links between members who interacted outside the club. Here, we are interested in detecting communities that arise from the member's interaction.

PyTorch Geometric provides an easy access to this dataset via the `torch_geometric.datasets` subpackage:

[13]:
```
from torch_geometric.datasets import KarateClub

dataset = KarateClub()
print(f'Dataset: {dataset}:')
print('======================')
print(f'Number of graphs: {len(dataset)}')
print(f'Number of features: {dataset.num_features}')
print(f'Number of classes: {dataset.num_classes}')
```

```
Dataset: KarateClub():
======================
Number of graphs: 1
Number of features: 34
Number of classes: 4
```

After initializing the `KarateClub` dataset, we first can inspect some of its properties. For example, we can see that this dataset holds exactly **one graph**, and that each node in this dataset is assigned a **34-dimensional feature vector** (which uniquely describes the members of the karate club). Furthermore, the graph holds exactly **4 classes**, which represent the community each node belongs to.

Let's now look at the underlying graph in more detail:

[14]:
```
data = dataset[0]  # Get the first graph object.

print(data)
print('==============================================================')

# Gather some statistics about the graph.
print(f'Number of nodes: {data.num_nodes}')
print(f'Number of edges: {data.num_edges}')
print(f'Average node degree: {data.num_edges / data.num_nodes:.2f}')
print(f'Number of training nodes: {data.train_mask.sum()}')
```

```
print(f'Training node label rate: {int(data.train_mask.sum()) / data.num_nodes:.
 ↪2f}')
print(f'Contains isolated nodes: {data.contains_isolated_nodes()}')
print(f'Contains self-loops: {data.contains_self_loops()}')
print(f'Is undirected: {data.is_undirected()}')
```

```
Data(edge_index=[2, 156], train_mask=[34], x=[34, 34], y=[34])
==============================================================
Number of nodes: 34
Number of edges: 156
Average node degree: 4.59
Number of training nodes: 4
Training node label rate: 0.12
Contains isolated nodes: False
Contains self-loops: False
Is undirected: True
```

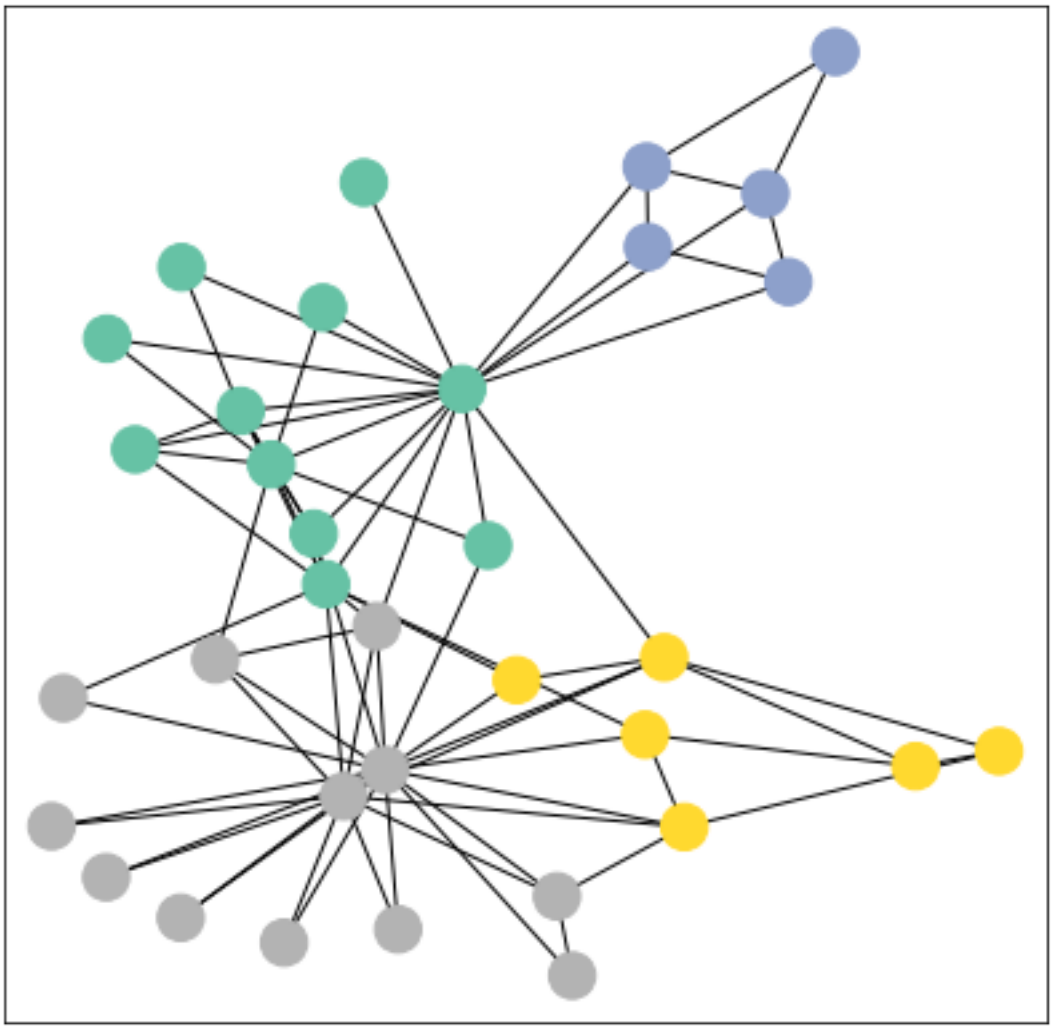
Each graph in PyTorch Geometric is represented by a single `Data` object, which holds all the information to describe its graph representation. We can print the data object anytime via `print(data)` to receive a short summary about its attributes and their shapes:

```
Data(edge_index=[2, 156], x=[34, 34], y=[34], train_mask=[34])
```

We can see that this `data` object holds 4 attributes: (1) The `edge_index` property holds the information about the **graph connectivity**, *i.e.*, a tuple of source and destination node indices for each edge. PyG further refers to (2) **node features** as `x` (each of the 34 nodes is assigned a 34-dim feature vector), and to (3) **node labels** as `y` (each node is assigned to exactly one class). (4) There also exists an additional attribute called `train_mask`, which describes for which nodes we already know their community assigments. In total, we are only aware of the ground-truth labels of 4 nodes (one for each community), and the task is to infer the community assignment for the remaining nodes.

We can further visualize the graph by converting it to the `networkx` library format, which implements, in addition to graph manipulation functionalities, powerful tools for visualization:

[15]:
```
from torch_geometric.utils import to_networkx

G = to_networkx(data, to_undirected=True)
visualize(G, color=data.y)
```



2.1 Implementing a GCN

We implement a GCN as a `torch.nn.Module` class that contains a sequence of GCNConv layers.

[27]:
```
import torch
from torch.nn import Linear
#from torch_geometric.nn import GCNConv


class GCN(torch.nn.Module):
    def __init__(self):
        super(GCN, self).__init__()
        torch.manual_seed(112233)
        self.conv1 = myGCNConv(dataset.num_features, 4)
```

```
        self.conv2 = myGCNConv(4, 4)
        self.conv3 = myGCNConv(4, 2)
        self.classifier = Linear(2, dataset.num_classes)

    def forward(self, x, edge_index):
        h = self.conv1(x, edge_index)
        h = h.tanh()
        h = self.conv2(h, edge_index)
        h = h.tanh()
        h = self.conv3(h, edge_index)
        h = h.tanh()  # Final GNN embedding space.

        # Apply a final (linear) classifier.
        out = self.classifier(h)

        return out, h

model = GCN()
print(model)
```

```
GCN(
  (conv1): myGCNConv(
    (lin): Linear(in_features=34, out_features=4, bias=True)
  )
  (conv2): myGCNConv(
    (lin): Linear(in_features=4, out_features=4, bias=True)
  )
  (conv3): myGCNConv(
    (lin): Linear(in_features=4, out_features=2, bias=True)
  )
  (classifier): Linear(in_features=2, out_features=4, bias=True)
)
```

Here, we first initialize all of our building blocks in `__init__` and define the computation flow of our network in `forward`. We first define and stack **three graph convolution layers**, which corresponds to aggregating 3-hop neighborhood information around each node (all nodes up to 3 "hops" away). In addition, the `myGCNConv` layers reduce the node feature dimensionality to 2, *i.e.*, $34 \to 4 \to 4 \to 2$. Each `myGCNConv` layer is enhanced by a tanh non-linearity.

After that, we apply a single linear transformation (`torch.nn.Linear`) that acts as a classifier to map our nodes to 1 out of the 4 classes/communities.

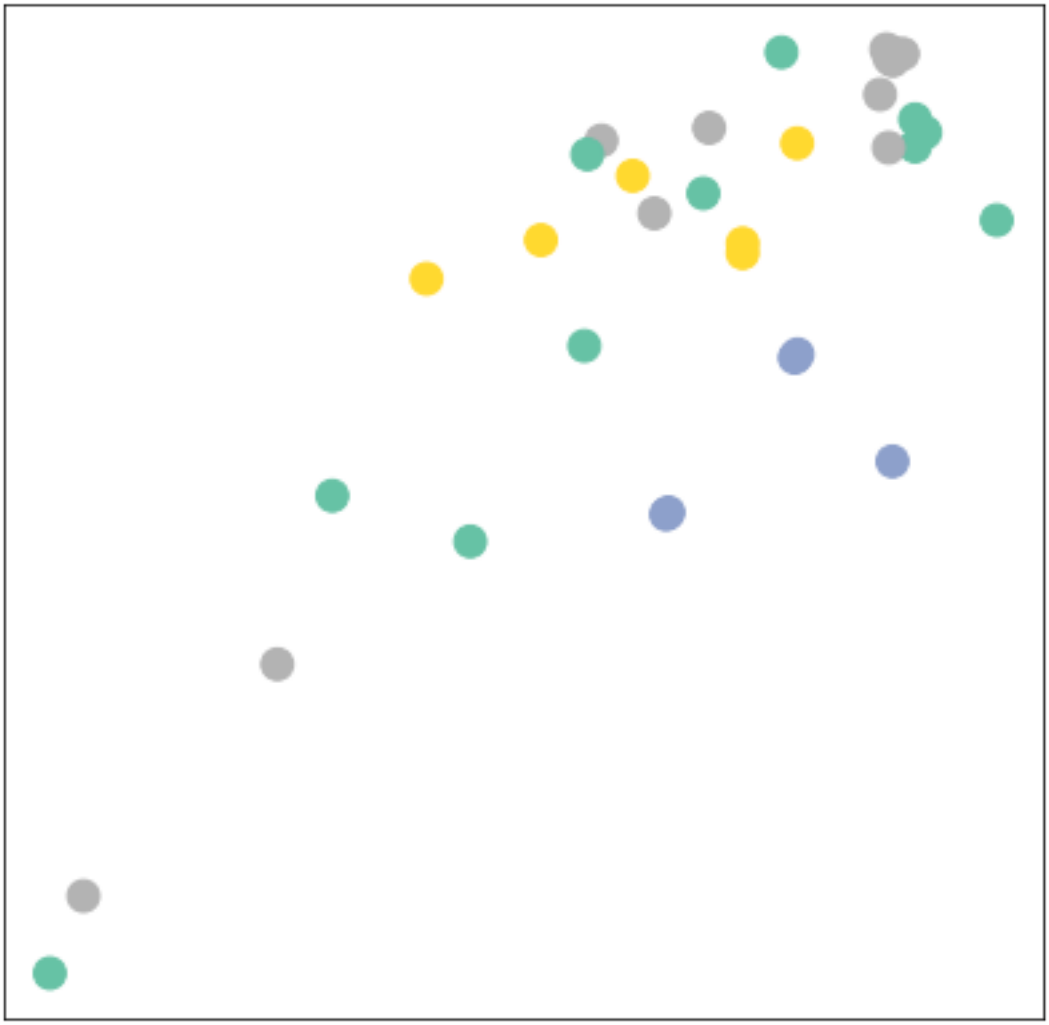
We return both the output of the final classifier as well as the final node embeddings produced by our GNN. We proceed to initialize our final model via `GCN()`, and printing our model produces a summary of all its used sub-modules.

2.1.1 Embedding the Karate Club Network

Let's take a look at the node embeddings produced by our GNN. Here, we pass in the initial node features `x` and the graph connectivity information `edge_index` to the model, and visualize its 2-dimensional embedding.

[28]:
```
model = GCN()

_, h = model(data.x, data.edge_index)
print(f'Embedding shape: {list(h.shape)}')

visualize(h, color=data.y)
```

```
Embedding shape: [34, 2]
```



Remarkably, even before training the weights of our model, the model produces an embedding

of nodes that closely resembles the community-structure of the graph. Nodes of the same color (community) are already closely clustered together in the embedding space, although the weights of our model are initialized **completely at random** and we have not yet performed any training so far! This leads to the conclusion that GNNs introduce a strong inductive bias, leading to similar embeddings for nodes that are close to each other in the input graph.

2.1.2 Training on the Karate Club Network

But can we do better? Let's look at an example on how to train our network parameters based on the knowledge of the community assignments of 4 nodes in the graph (one for each community):

Since everything in our model is differentiable and parameterized, we can add some labels, train the model and observse how the embeddings react. Here, we make use of a semi-supervised or transductive learning procedure: We simply train against one node per class, but are allowed to make use of the complete input graph data.

Training our model is very similar to any other PyTorch model. In addition to defining our network architecture, we define a loss critertion (here, `CrossEntropyLoss`) and initialize a stochastic gradient optimizer (here, `Adam`).

Note that our semi-supervised learning scenario is achieved by the following line:

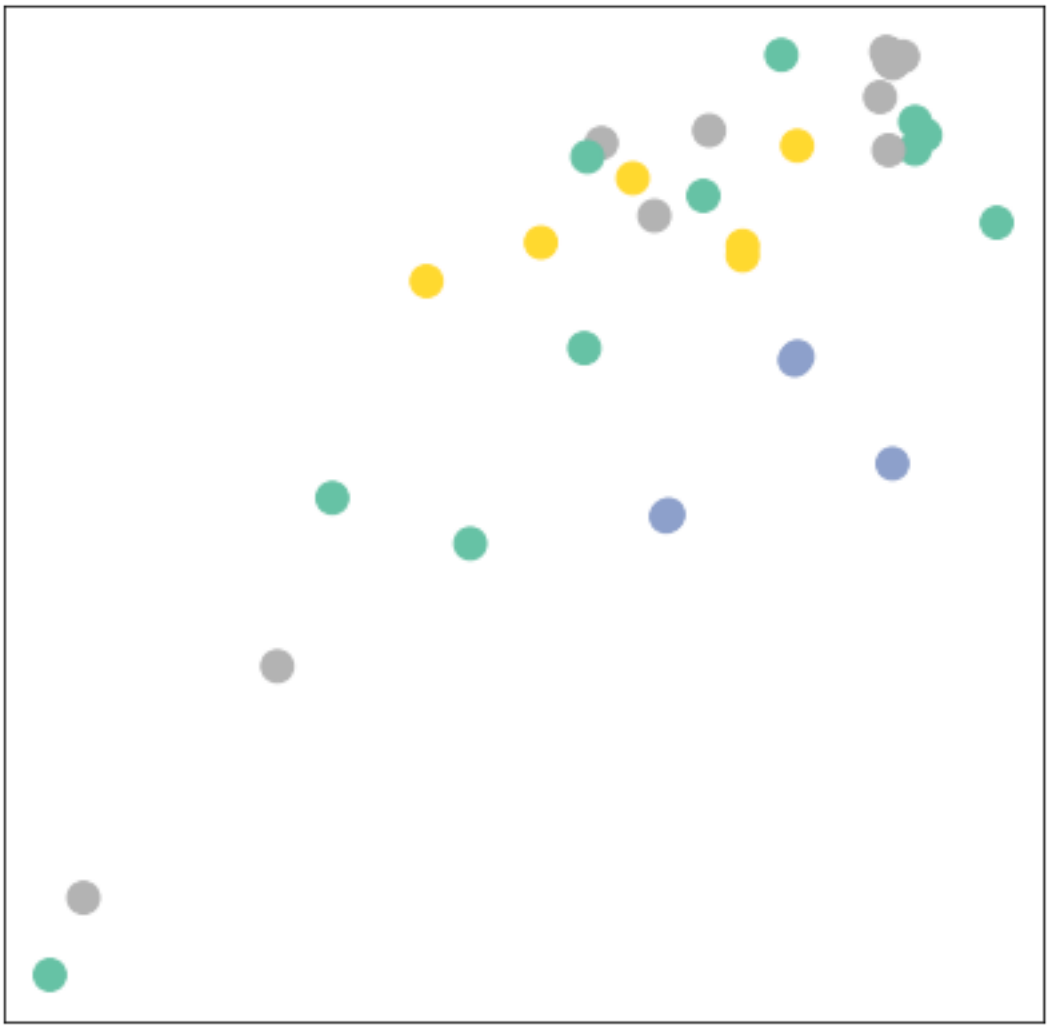
`loss = criterion(out[data.train_mask], data.y[data.train_mask])`

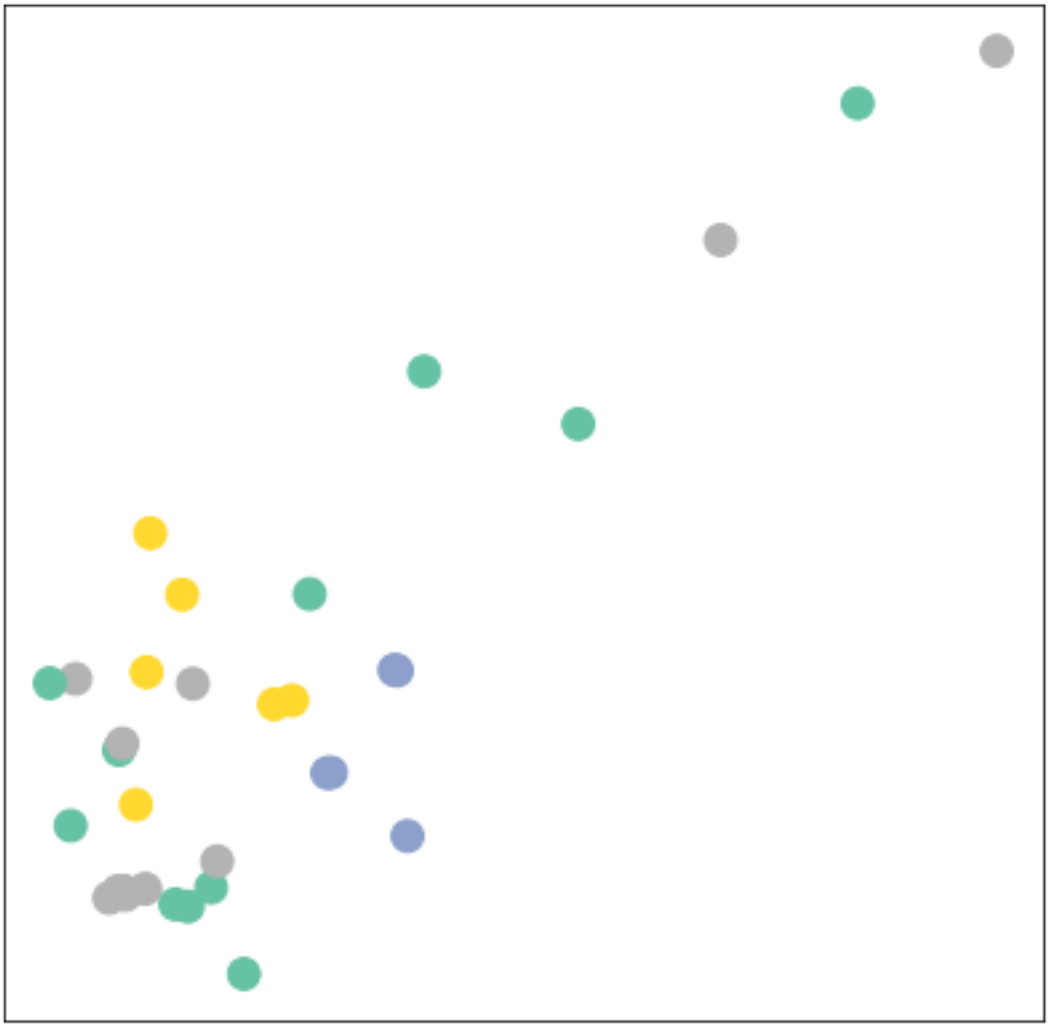
While we compute node embeddings for all of our nodes, we **only make use of the training nodes for computing the loss**. Here, this is implemented by filtering the output of the classifier `out` and ground-truth labels `data.y` to only contain the nodes in the `train_mask`.

Let us now start training and see how our node embeddings evolve over time (best experienced by explicitely running the code):

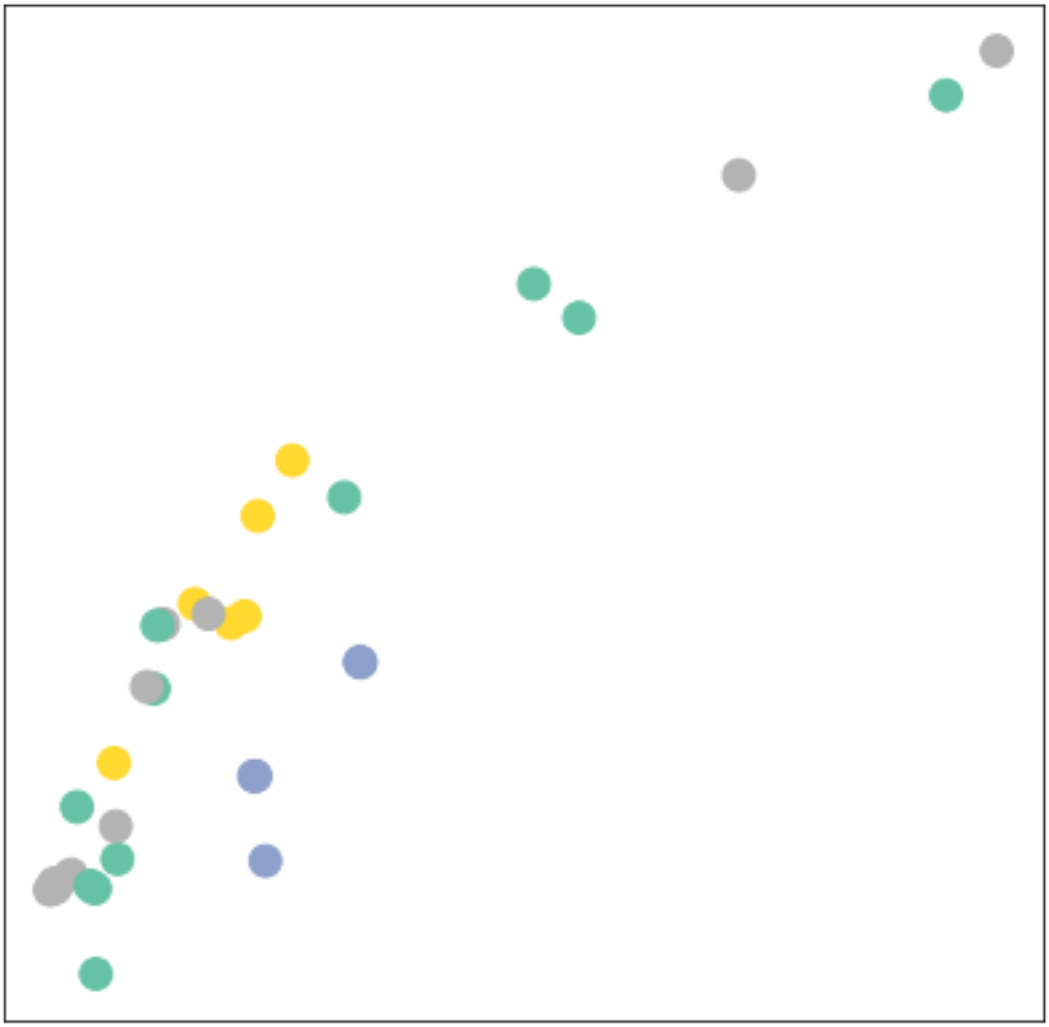
[29]:
```
import time
#from IPython.display import Javascript  # Restrict height of output cell.
#display(Javascript('''google.colab.output.setIframeHeight(0, true, {maxHeight:␣
 ↪430})'''))

model = GCN()
criterion = torch.nn.CrossEntropyLoss()  # Define loss criterion.
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)  # Define optimizer.

def train(data):
    optimizer.zero_grad()  # Clear gradients.
    out, h = model(data.x, data.edge_index)  # Perform a single forward pass.
    loss = criterion(out[data.train_mask], data.y[data.train_mask])  # Compute␣
 ↪the loss solely based on the training nodes.
    loss.backward()  # Derive gradients.
    optimizer.step()  # Update parameters based on gradients.
    return loss, h

for epoch in range(401):
```

```
    loss, h = train(data)
    if epoch % 10 == 0:
        visualize(h, color=data.y, epoch=epoch, loss=loss)
        time.sleep(0.3)
```

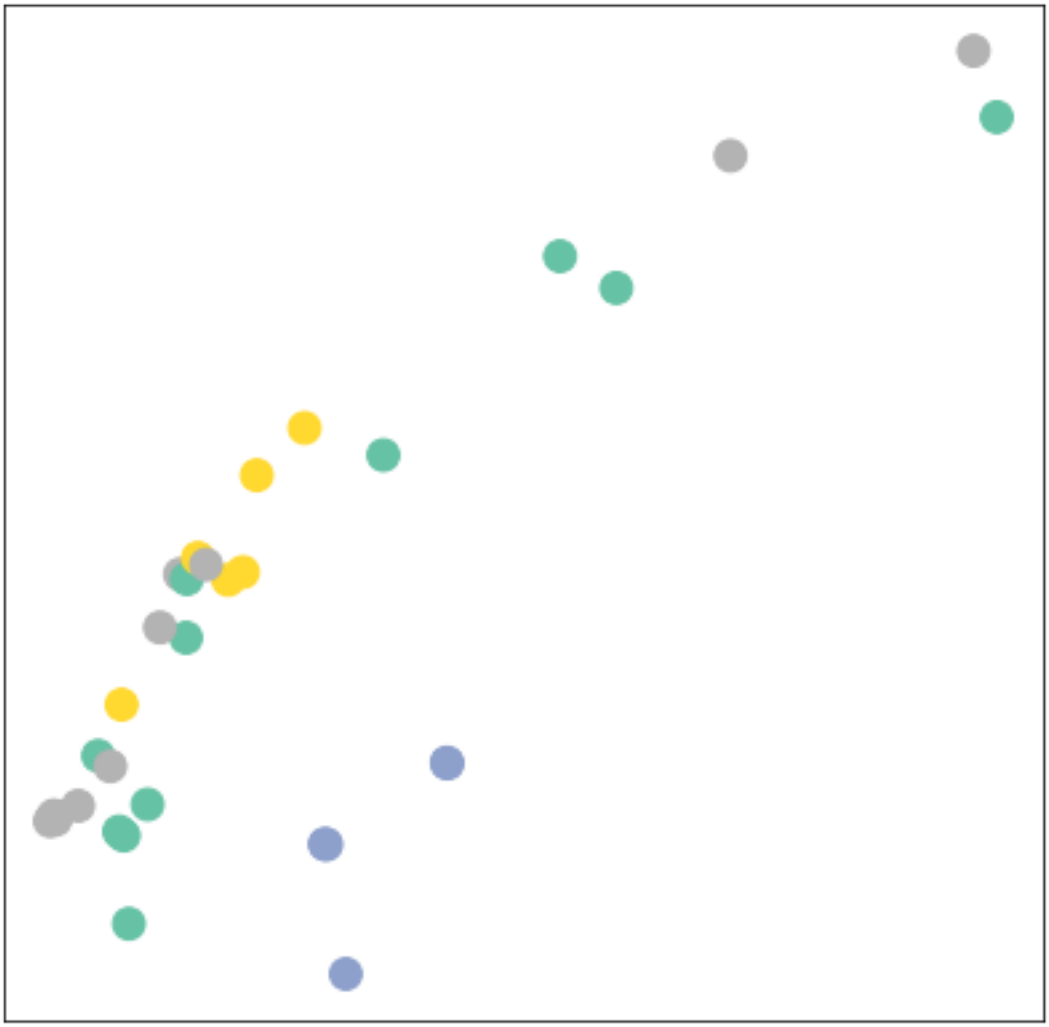




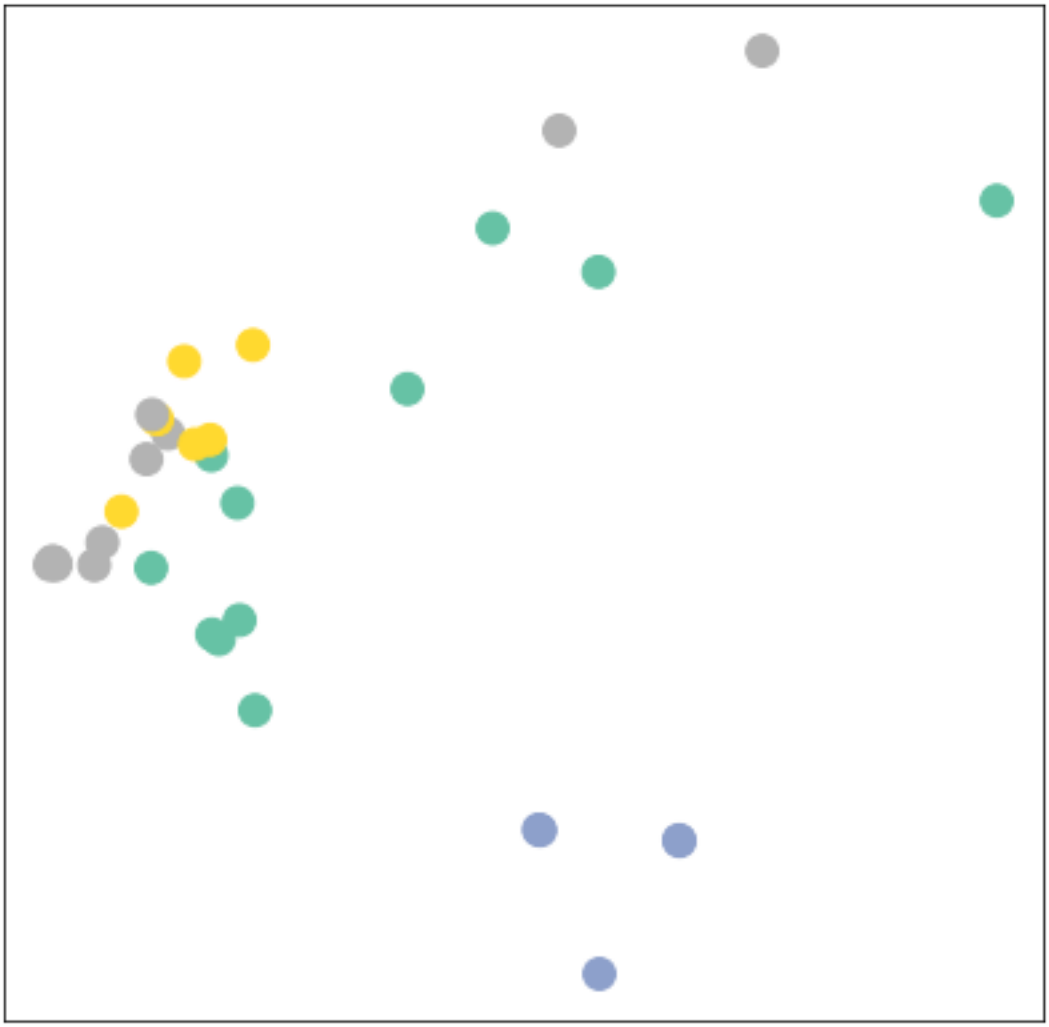
Epoch: 10, Loss: 1.3711



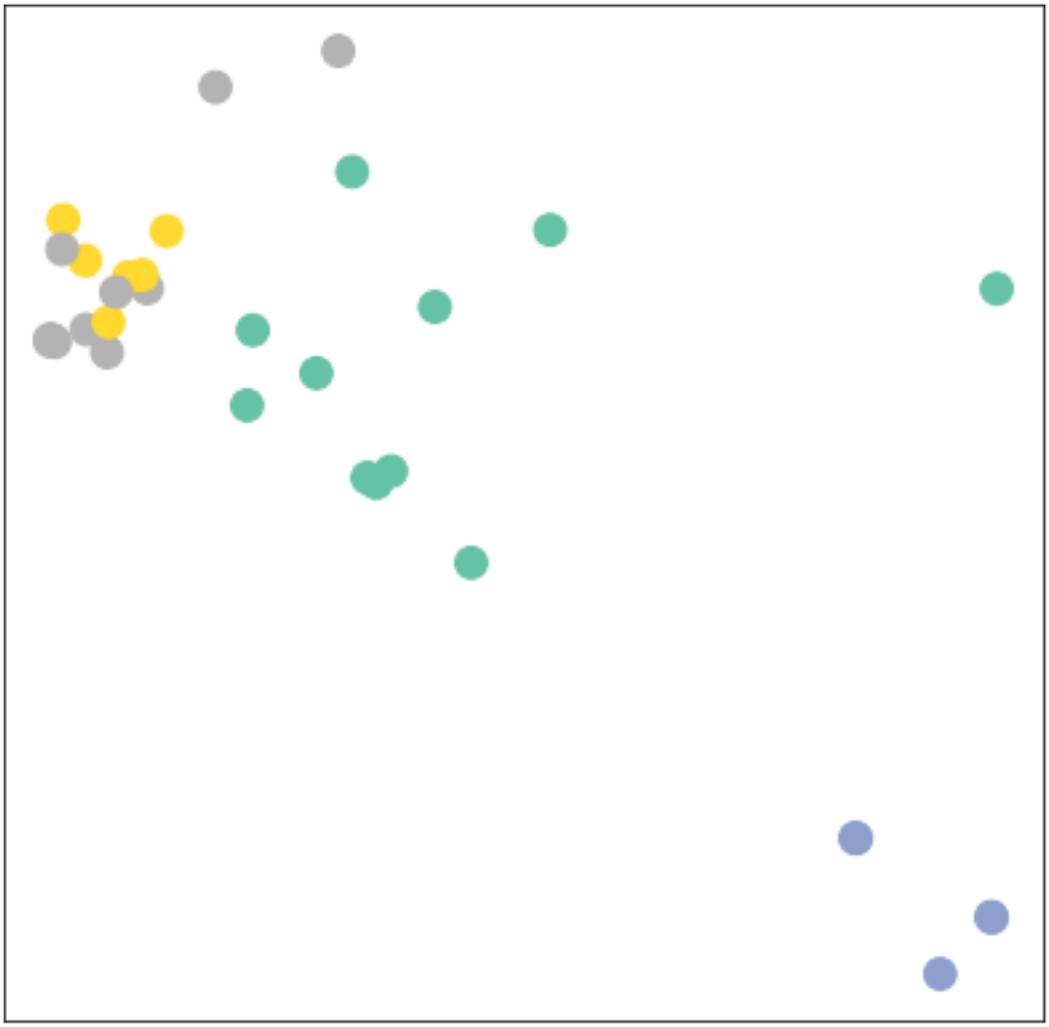
Epoch: 20, Loss: 1.3258



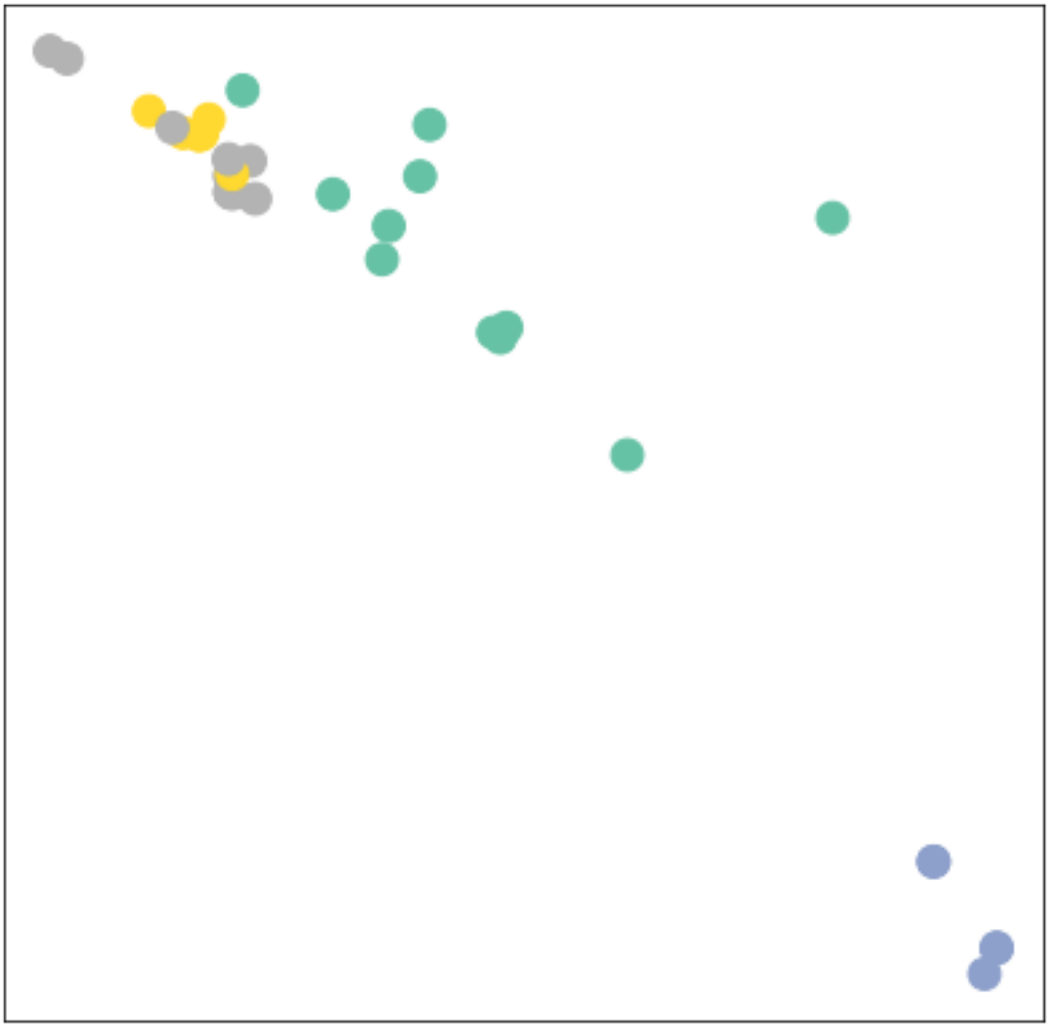
Epoch: 30, Loss: 1.2992



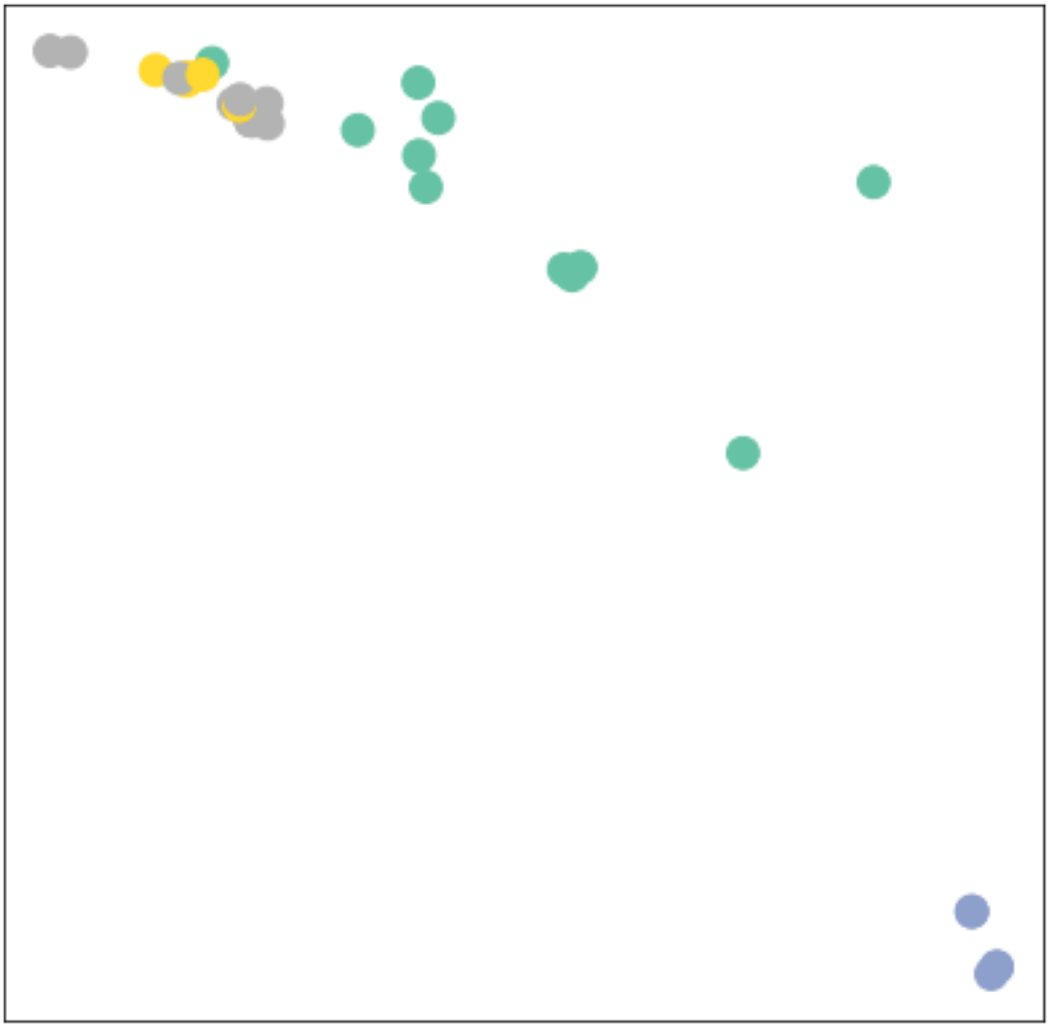
Epoch: 40, Loss: 1.2312



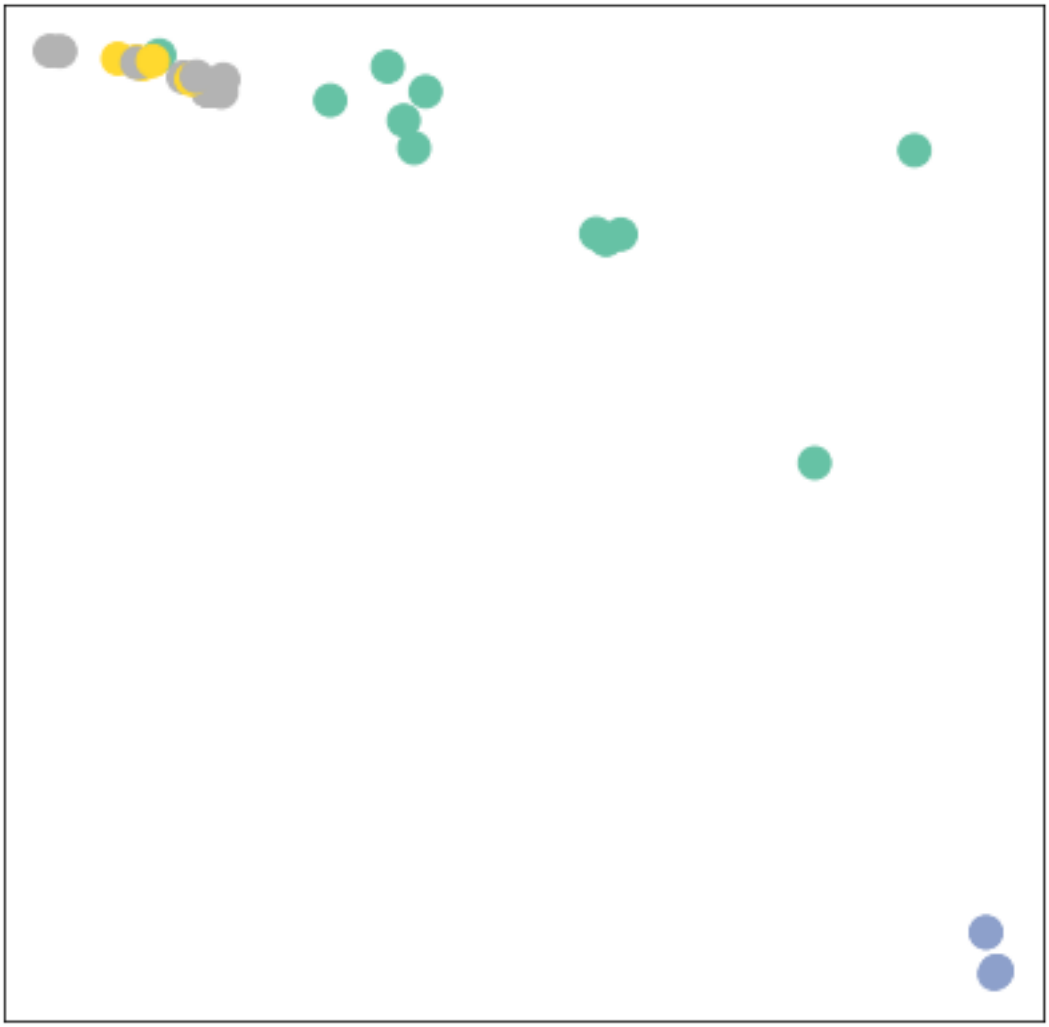
Epoch: 50, Loss: 1.0871



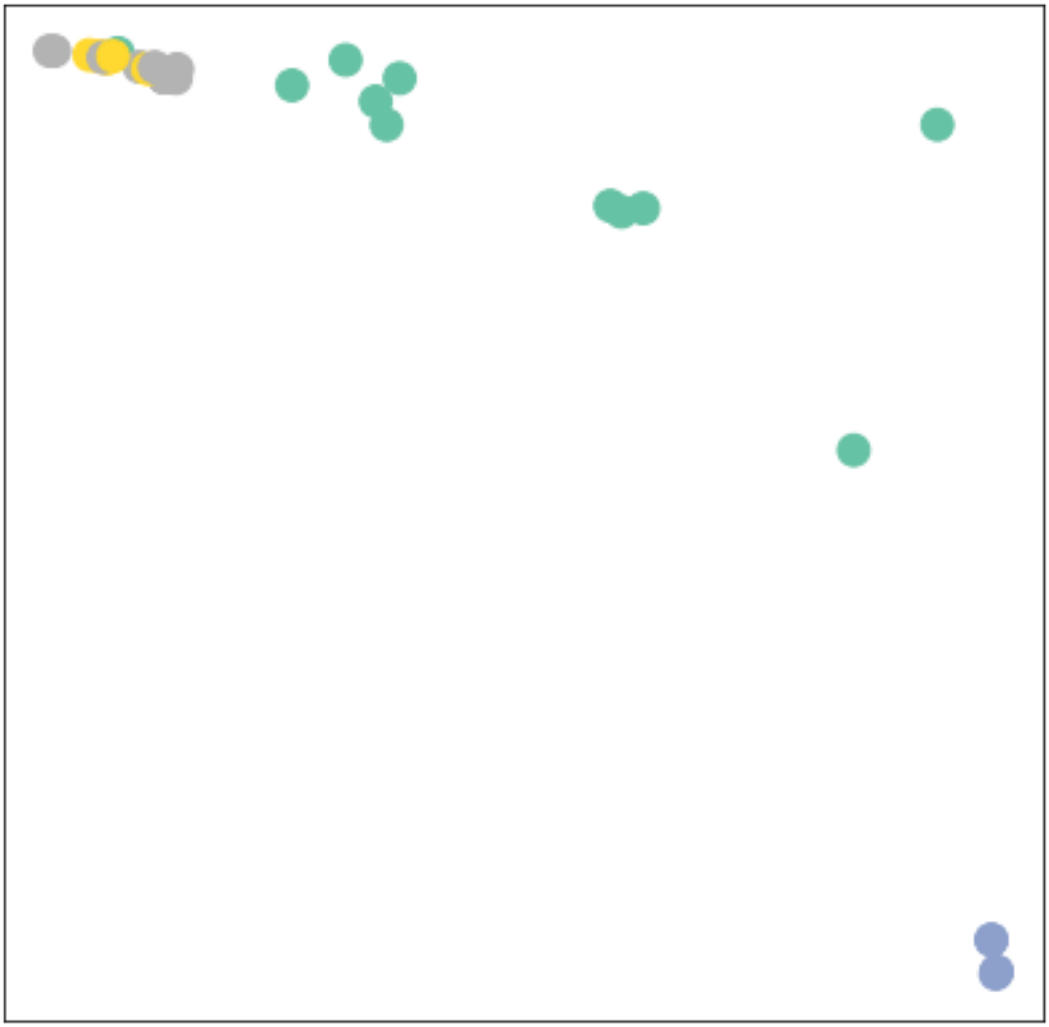
Epoch: 60, Loss: 0.8595



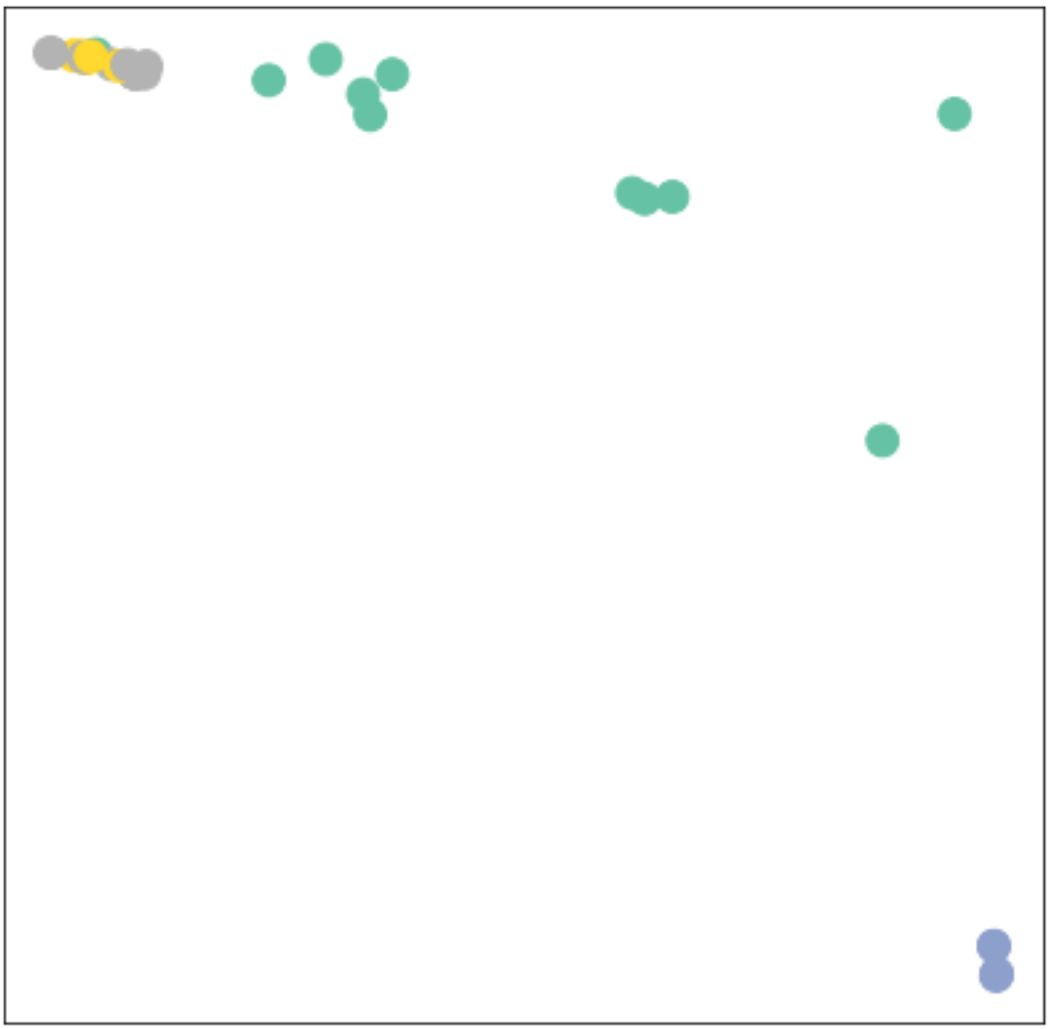
Epoch: 70, Loss: 0.6815

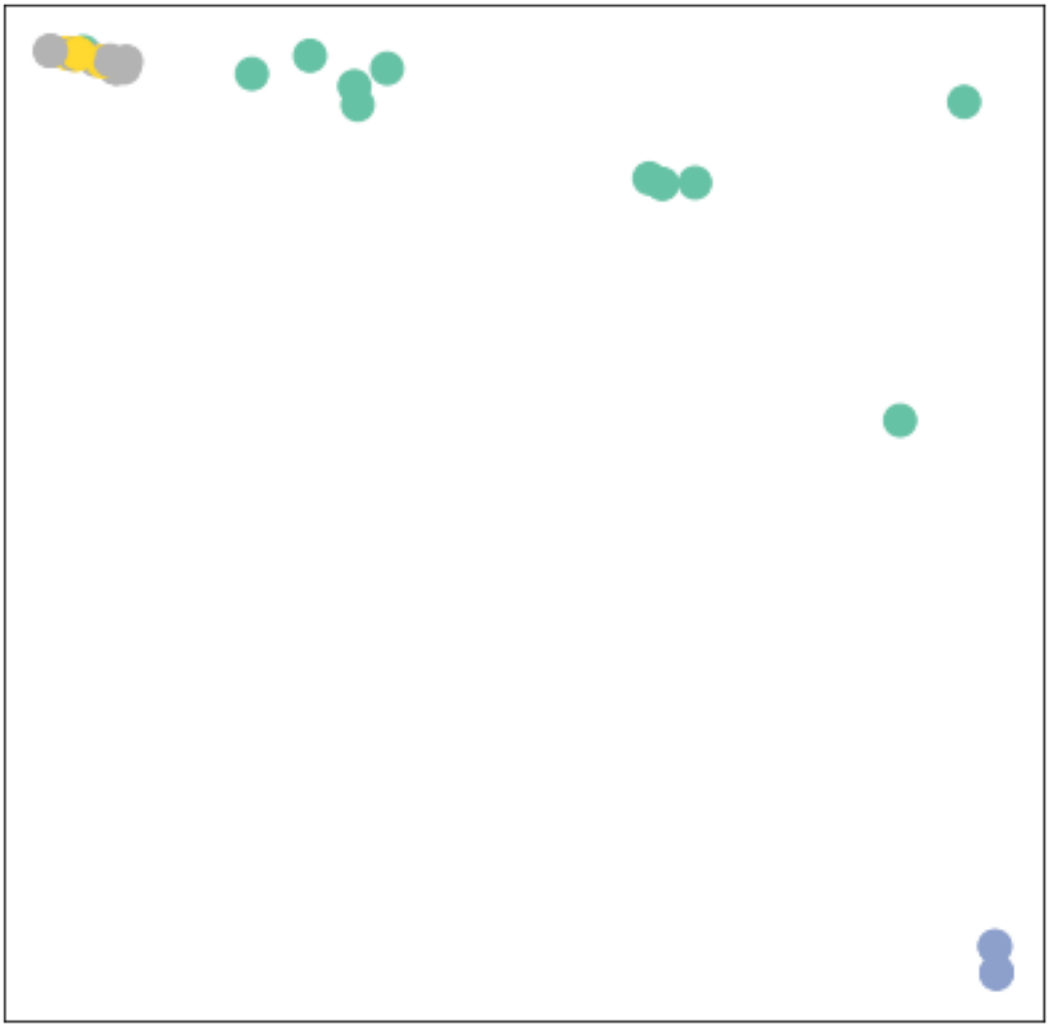


Epoch: 80, Loss: 0.5705

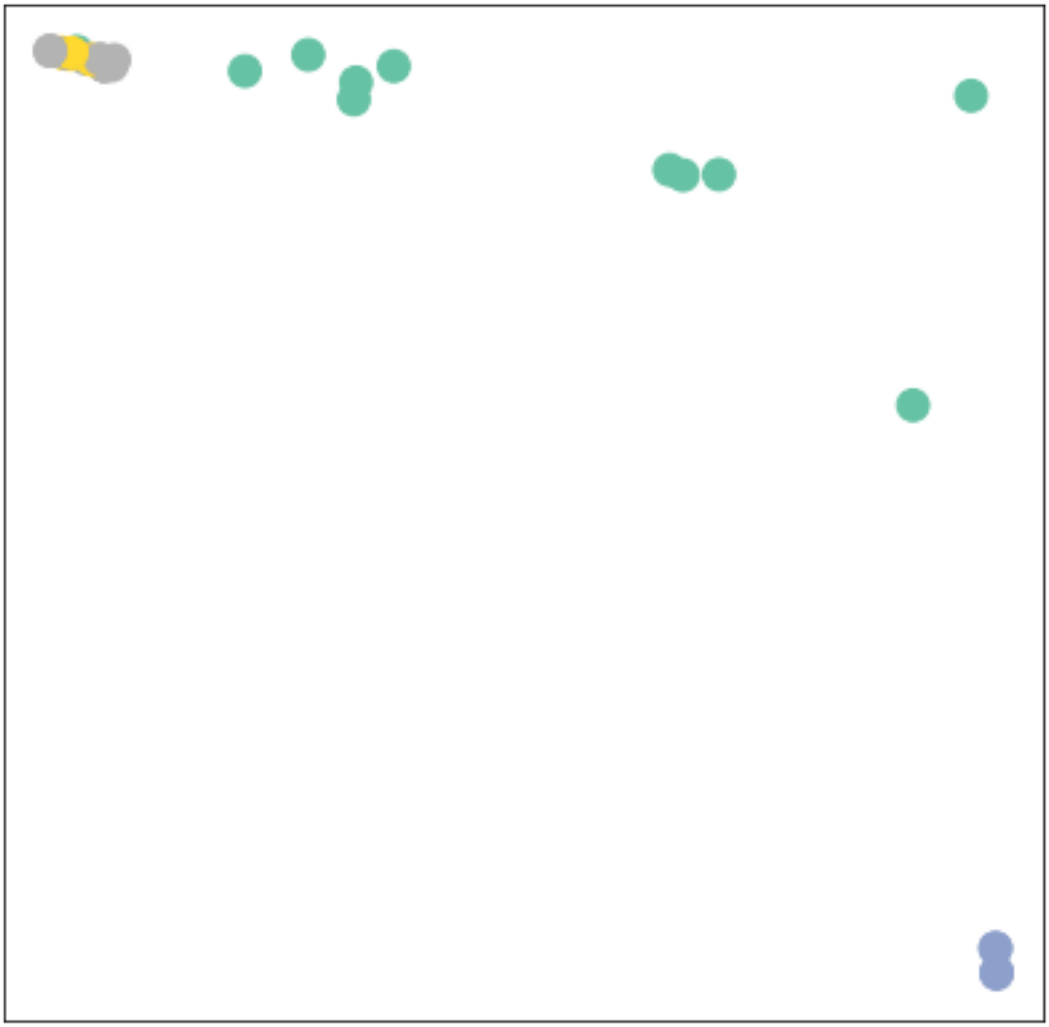


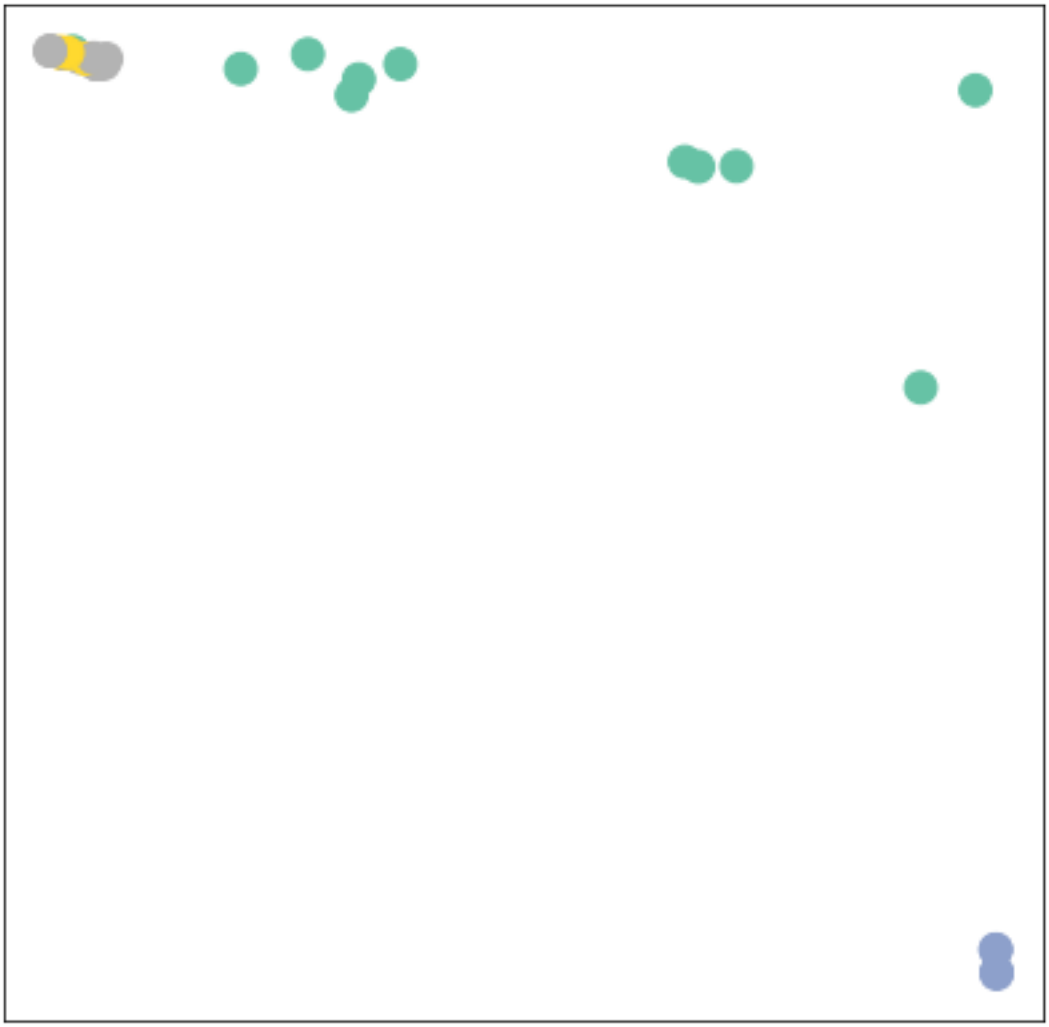
Epoch: 90, Loss: 0.5051

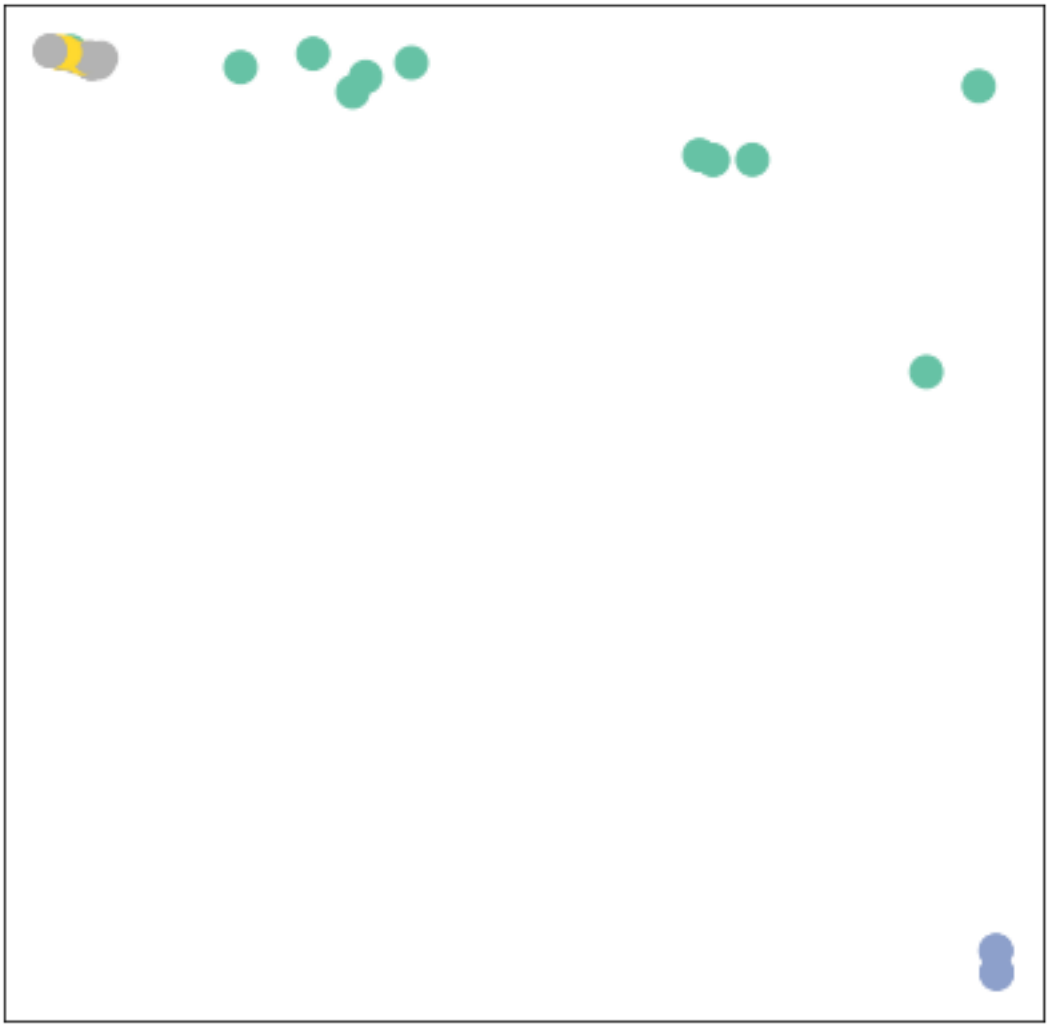

Epoch: 100, Loss: 0.4669

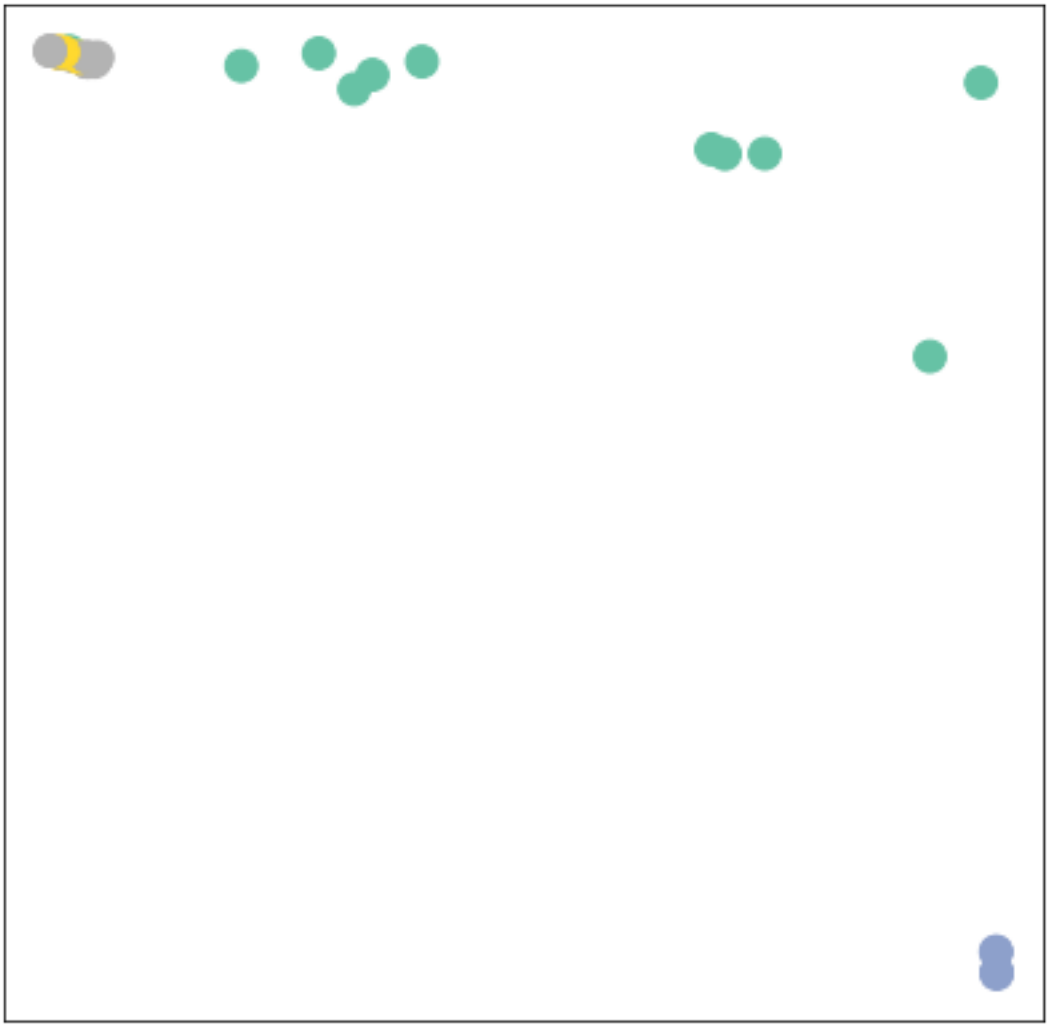


Epoch: 110, Loss: 0.4436

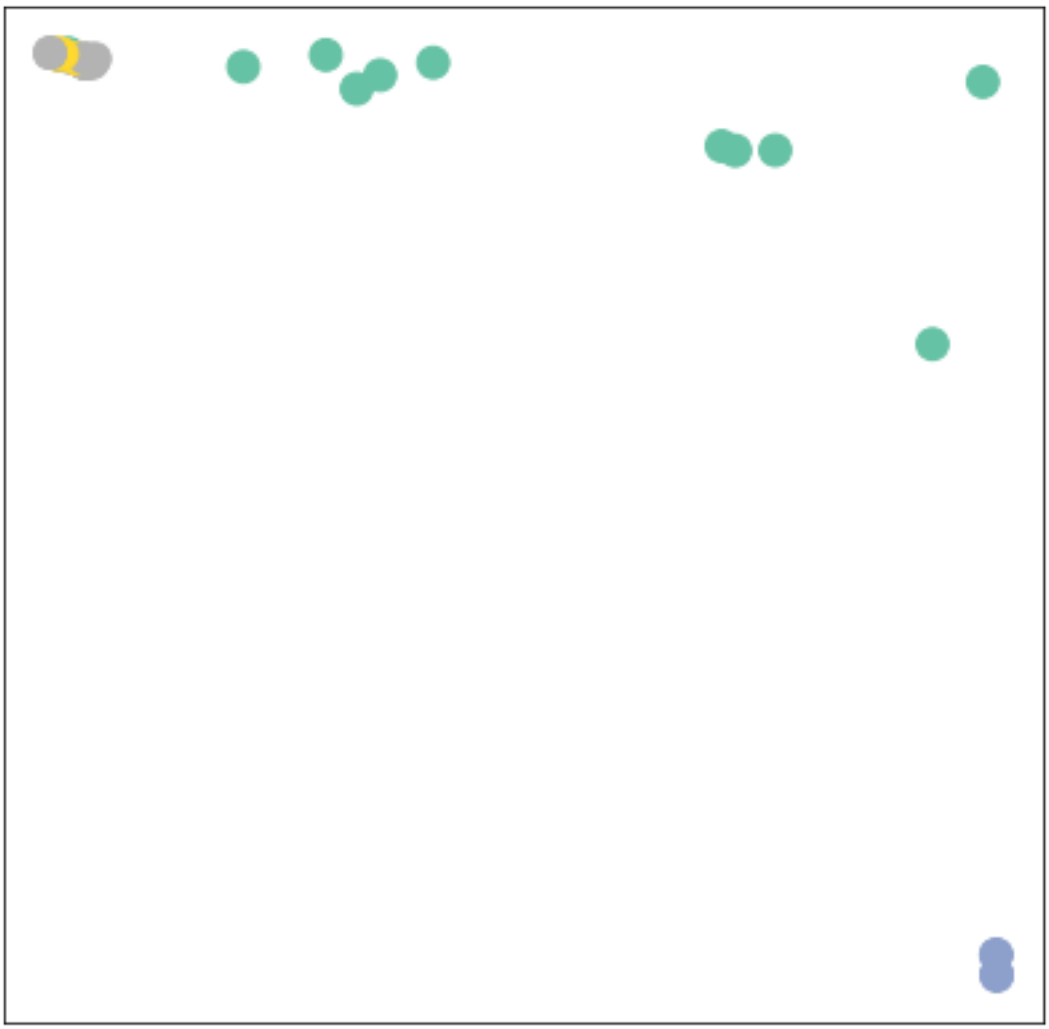

Epoch: 120, Loss: 0.4280


Epoch: 130, Loss: 0.4168

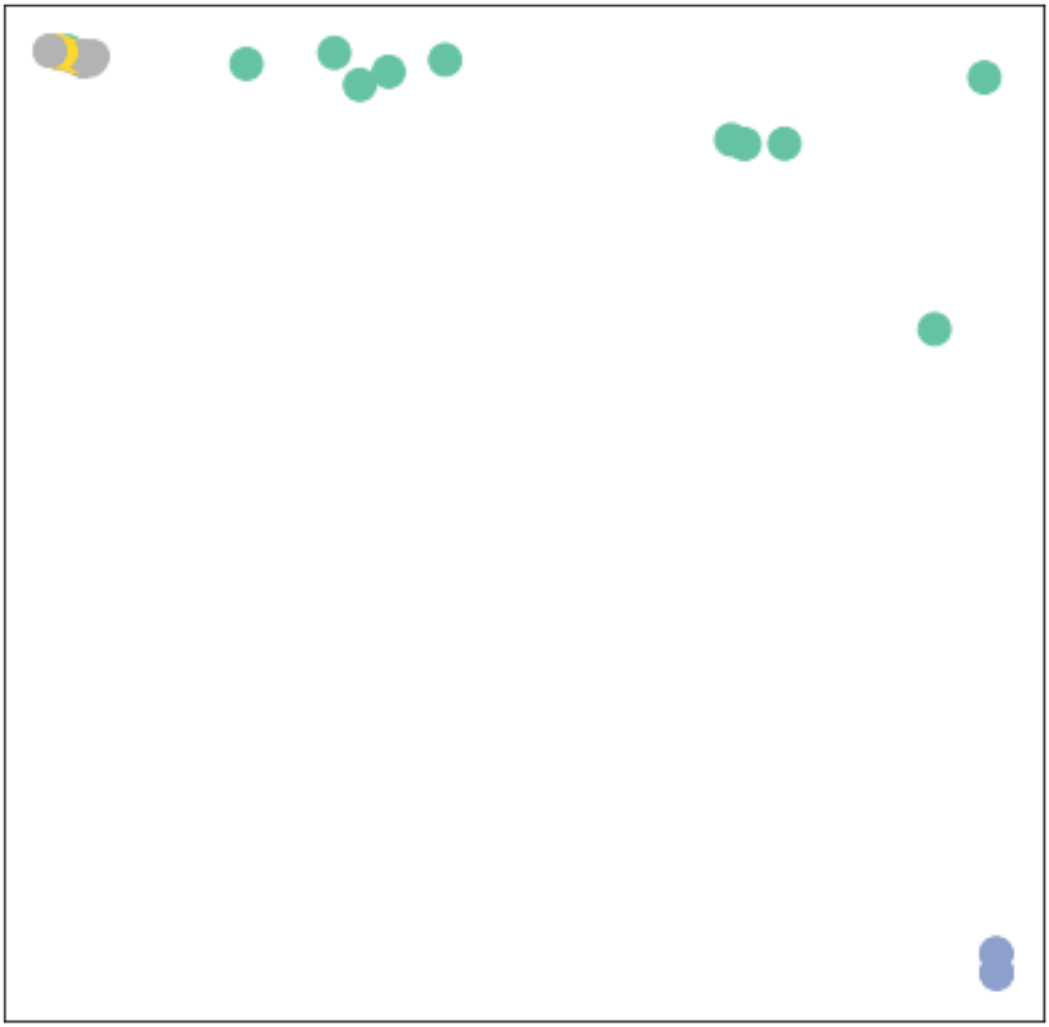

Epoch: 140, Loss: 0.4083



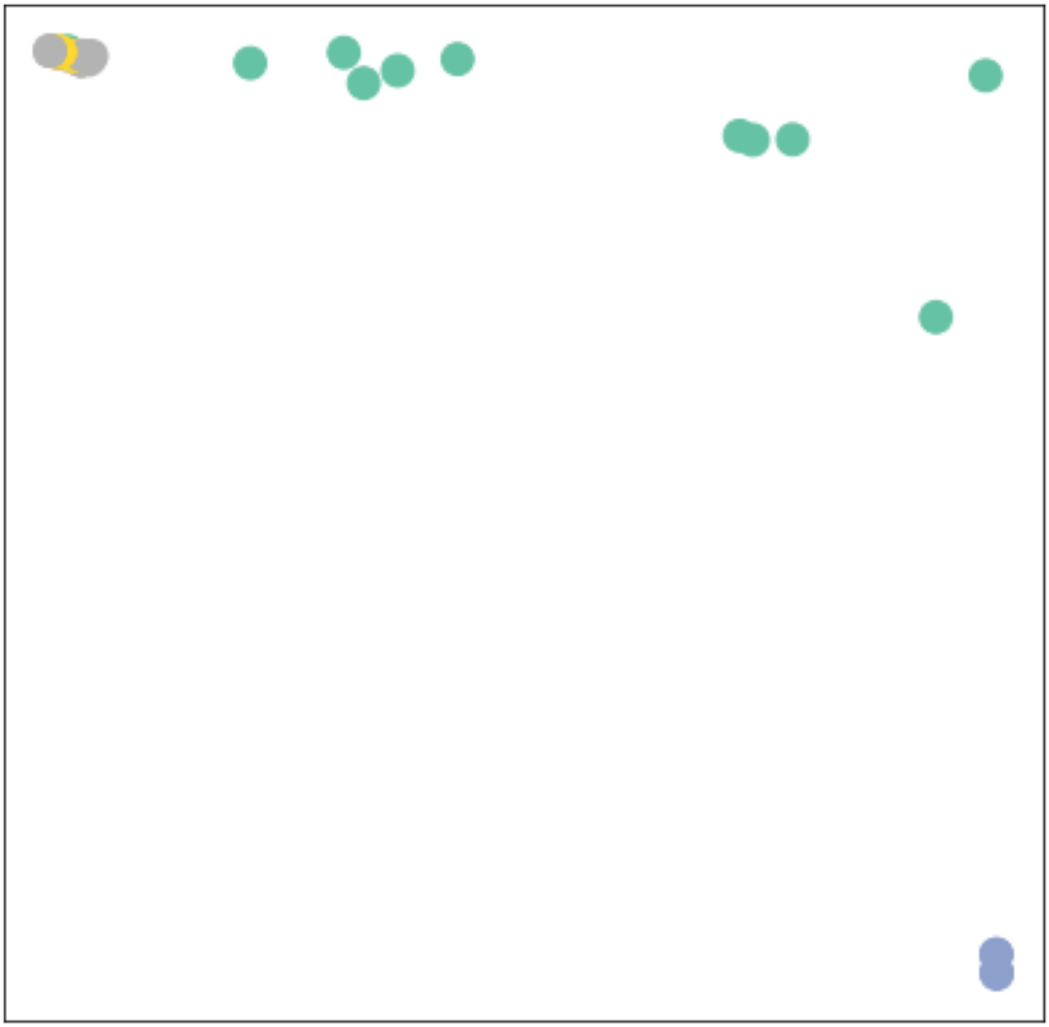
Epoch: 150, Loss: 0.4016



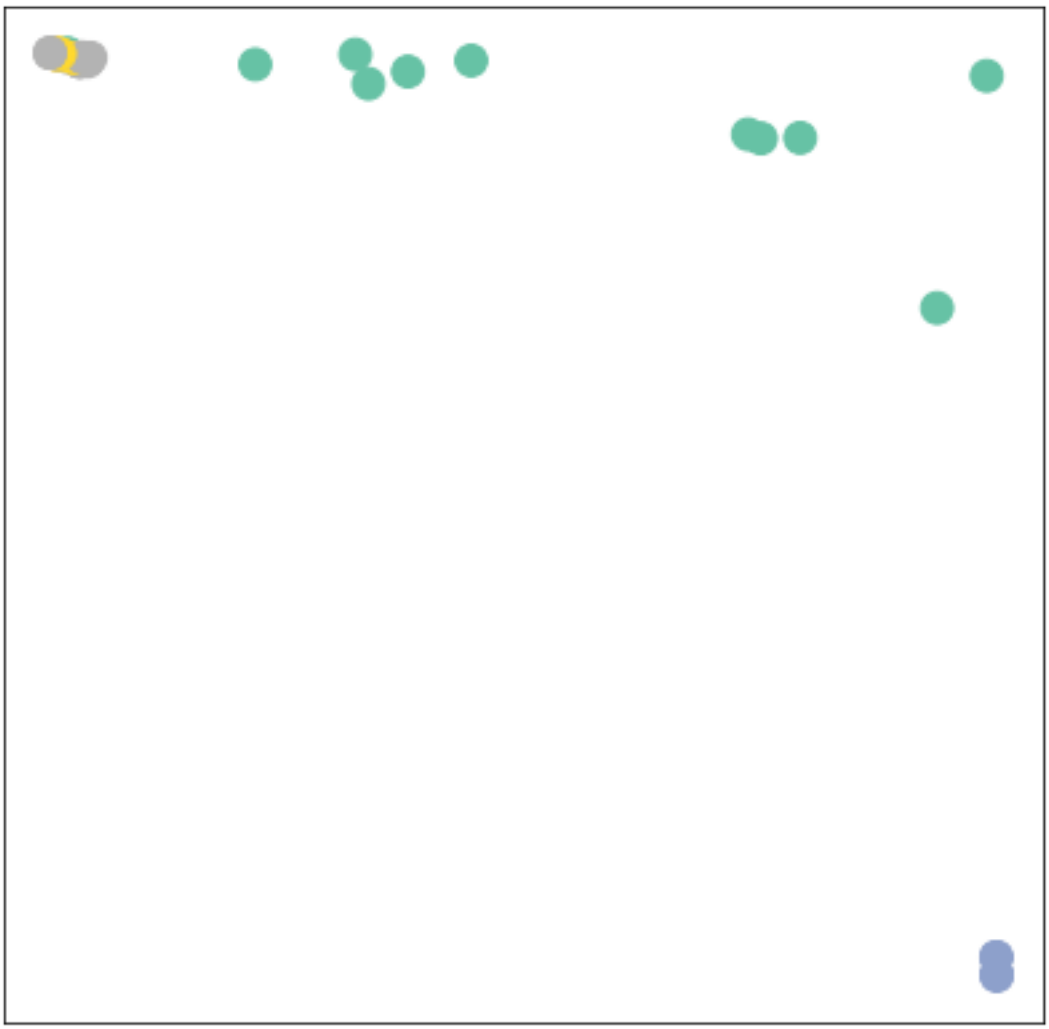
Epoch: 160, Loss: 0.3961



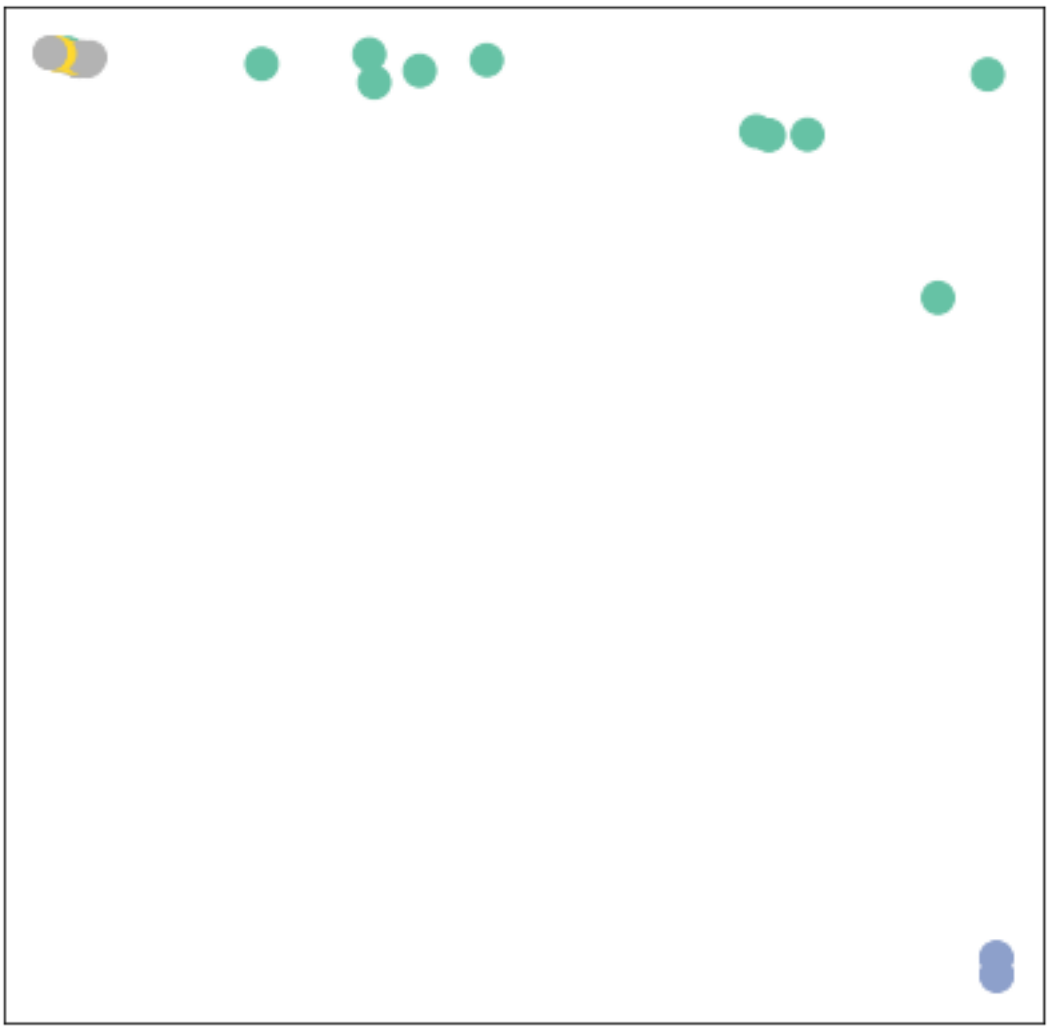
Epoch: 170, Loss: 0.3915



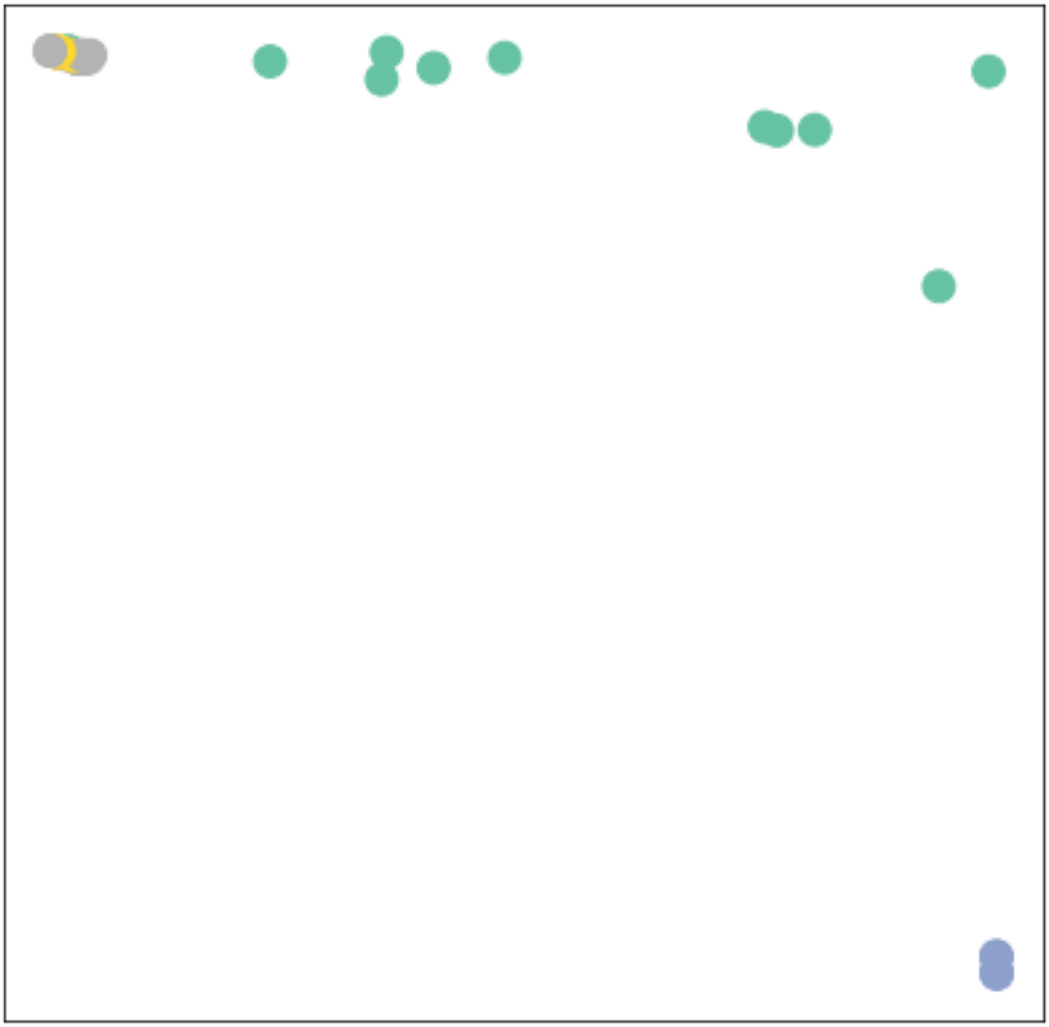
Epoch: 180, Loss: 0.3876



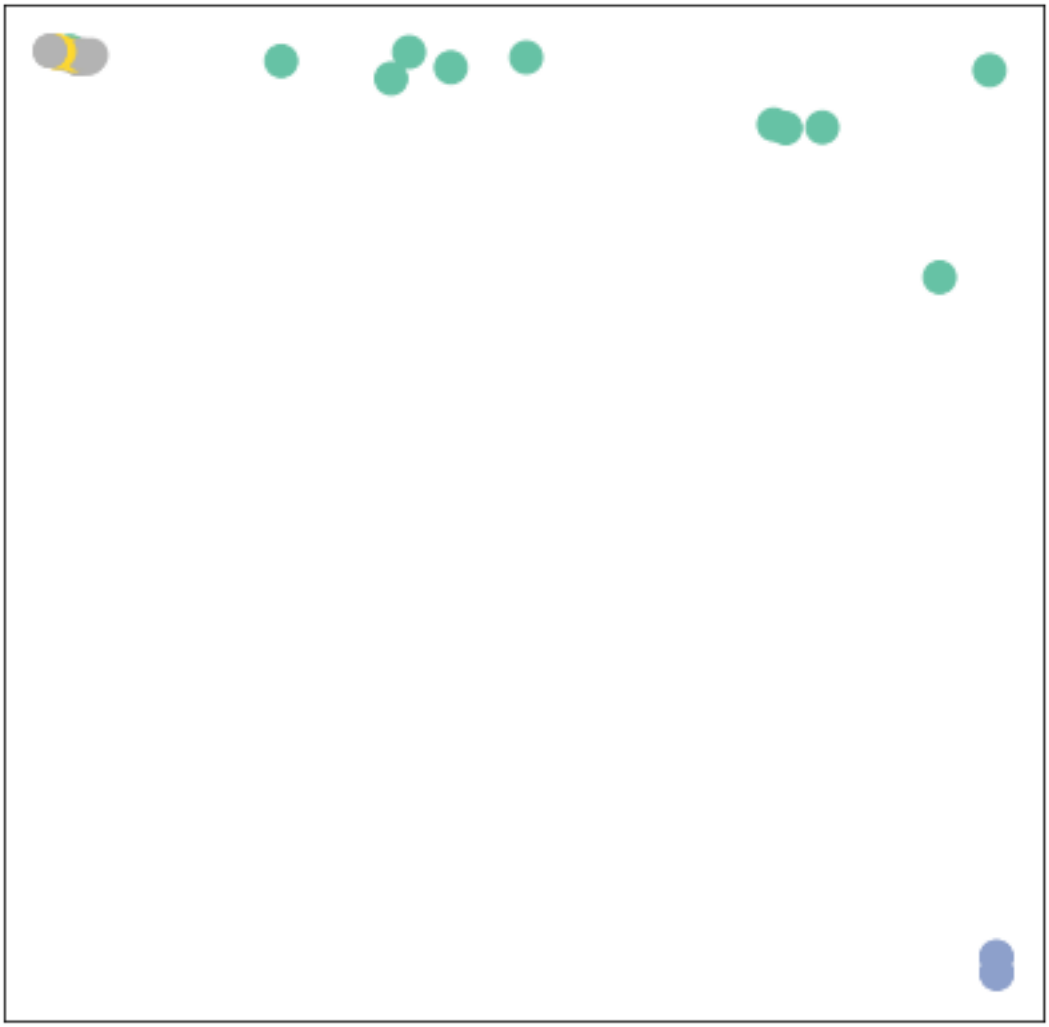
Epoch: 190, Loss: 0.3843



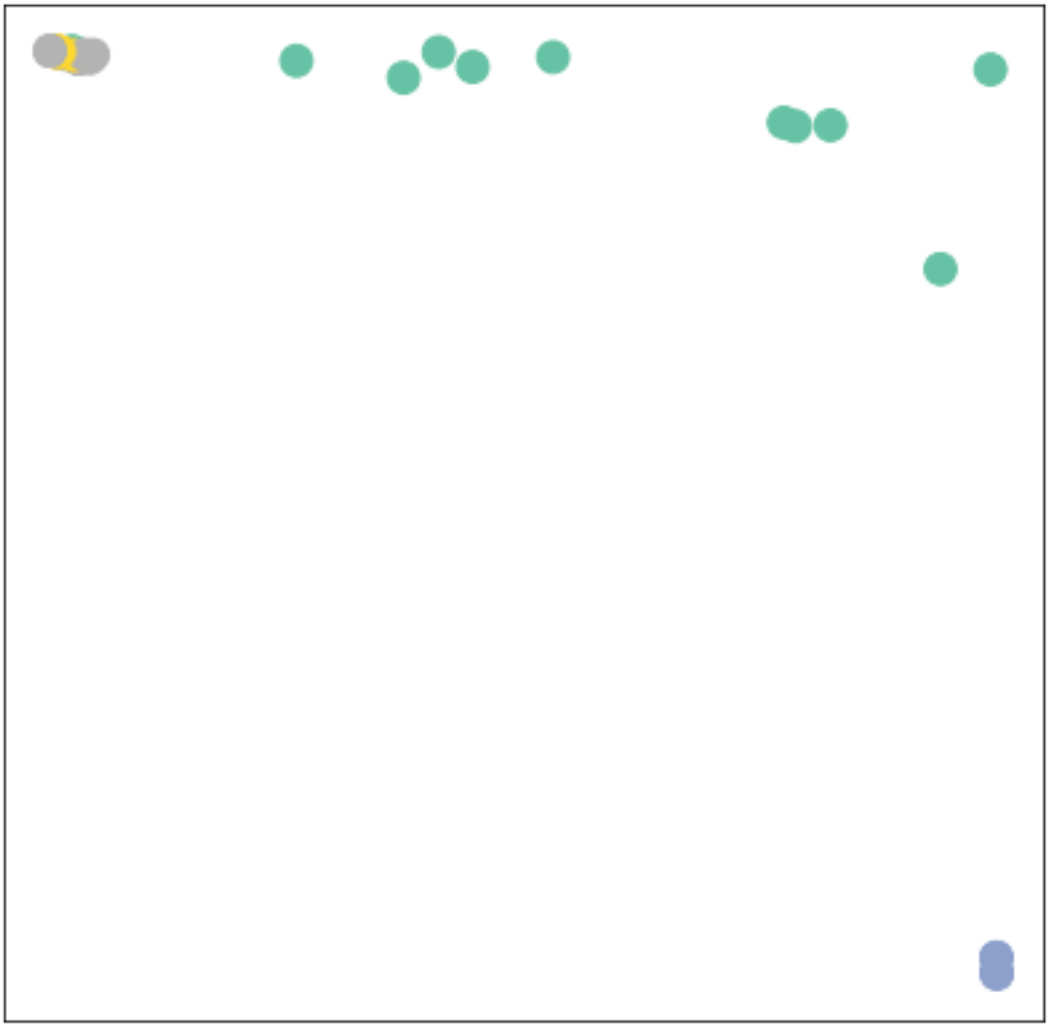
Epoch: 200, Loss: 0.3813



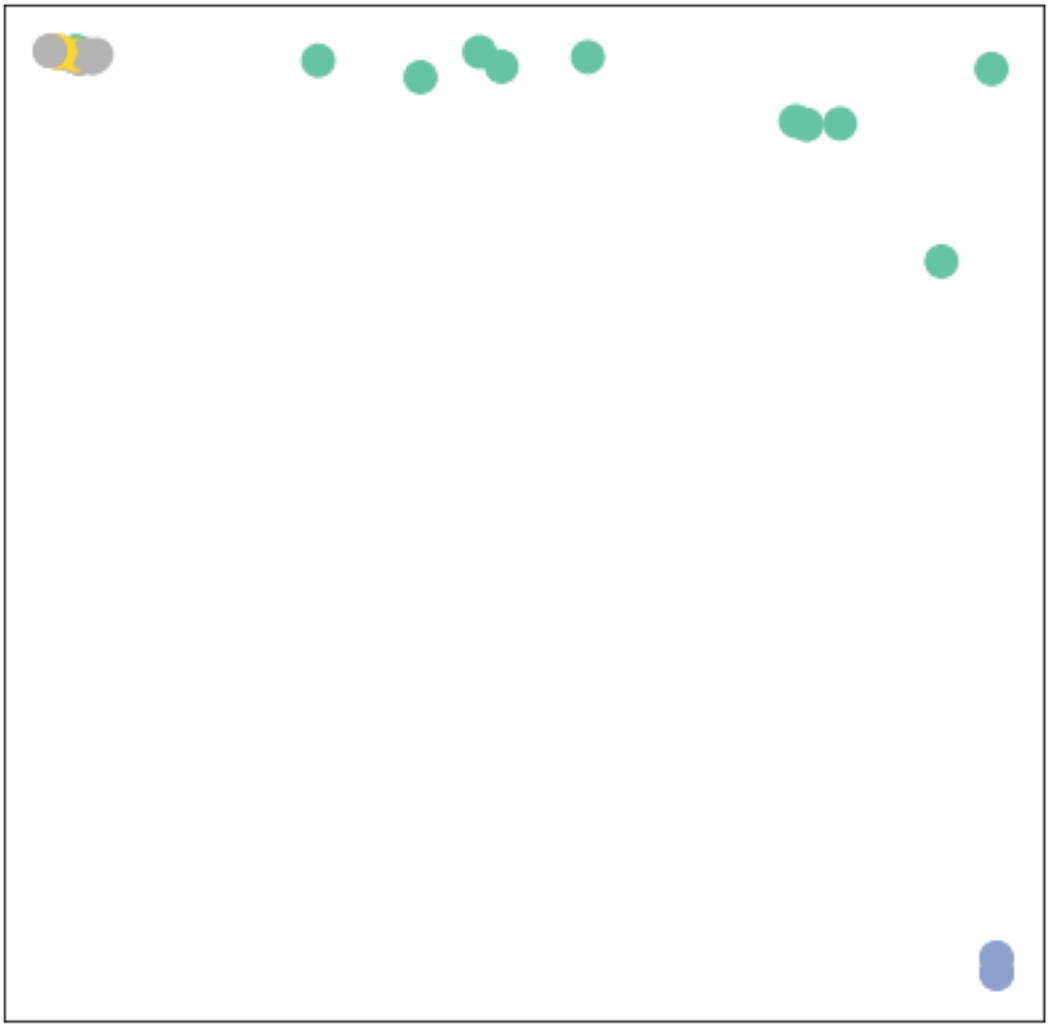
Epoch: 210, Loss: 0.3787



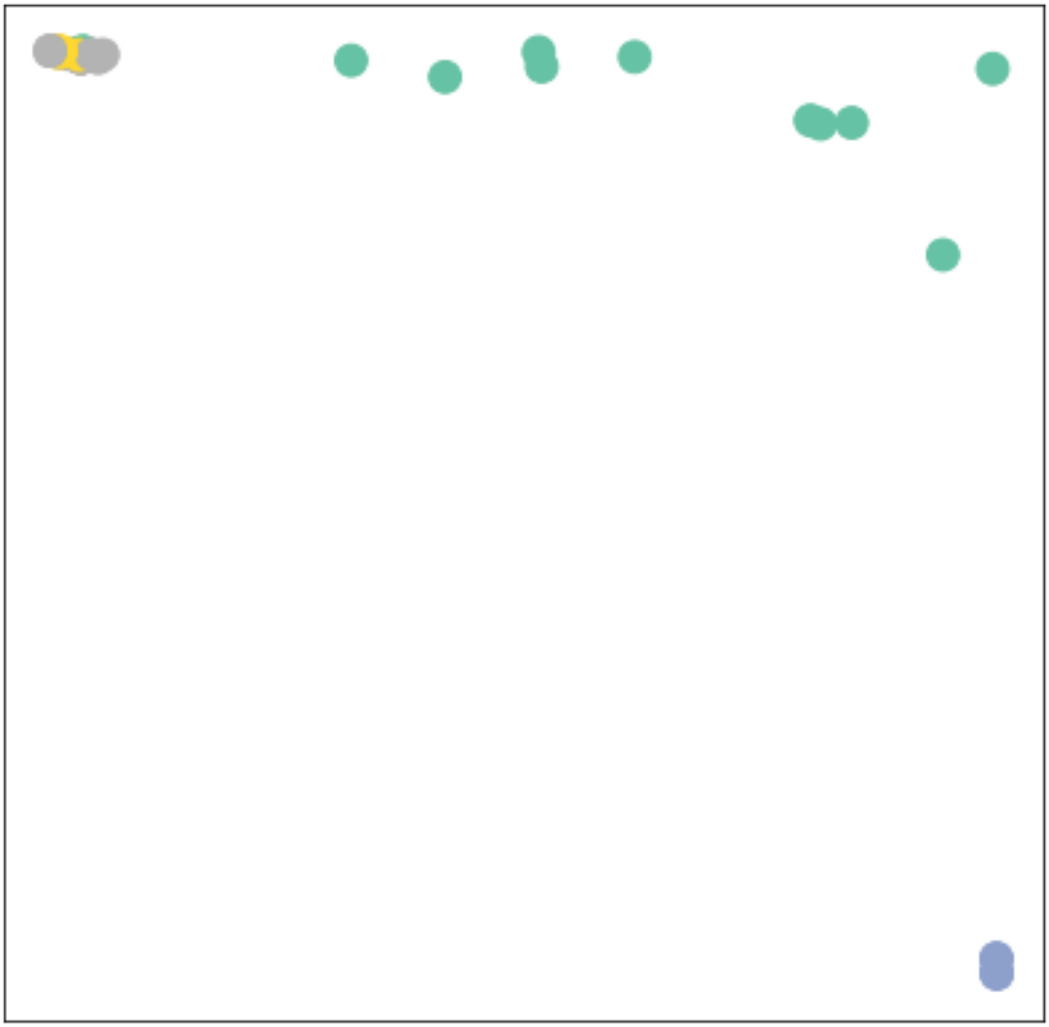
Epoch: 220, Loss: 0.3764



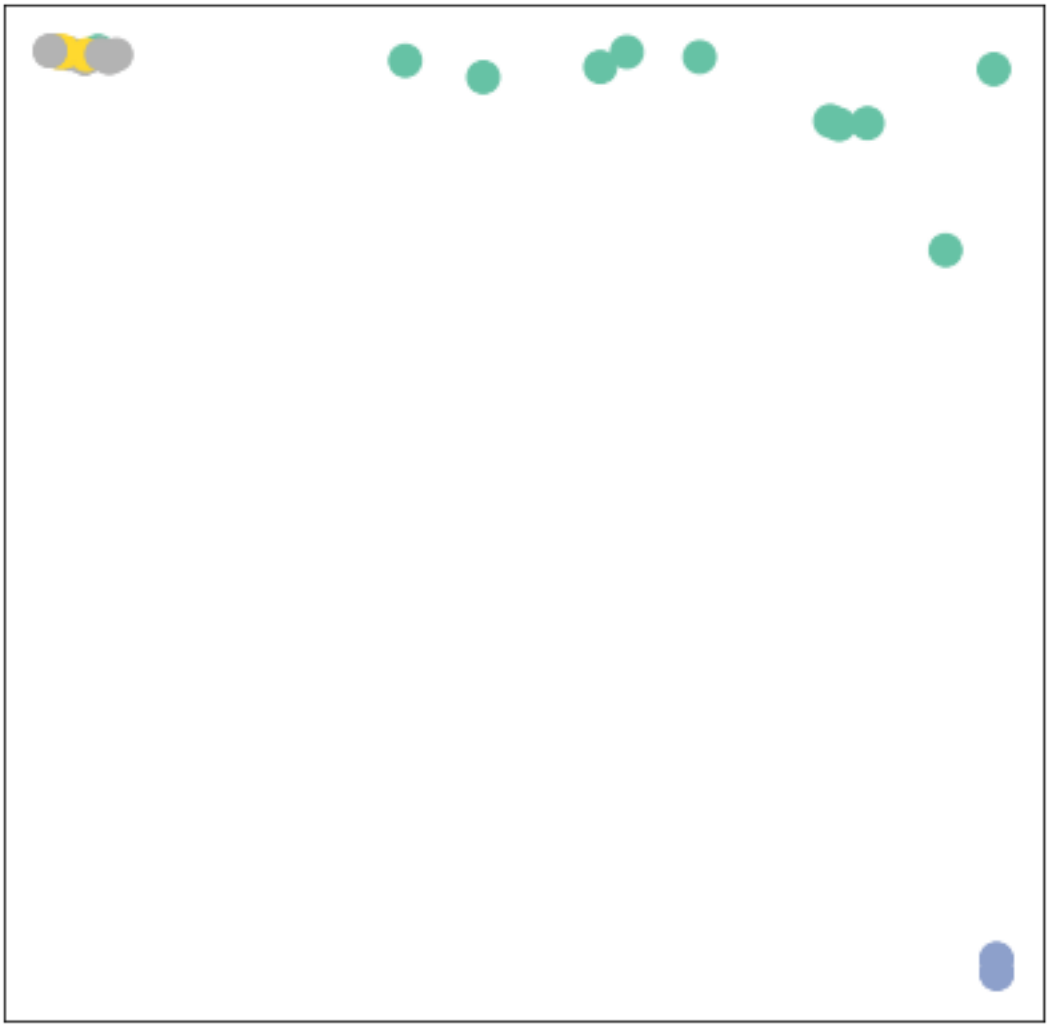
Epoch: 230, Loss: 0.3743



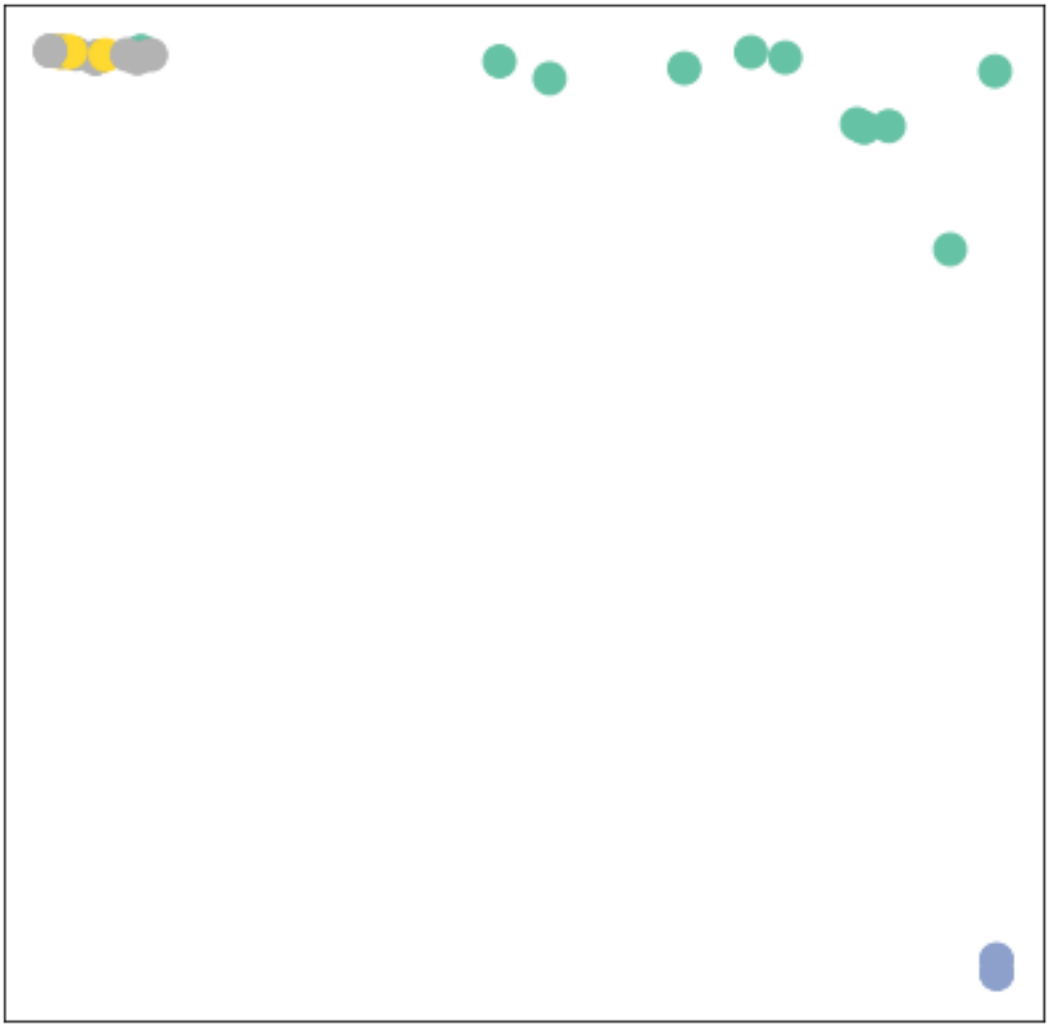
Epoch: 240, Loss: 0.3722



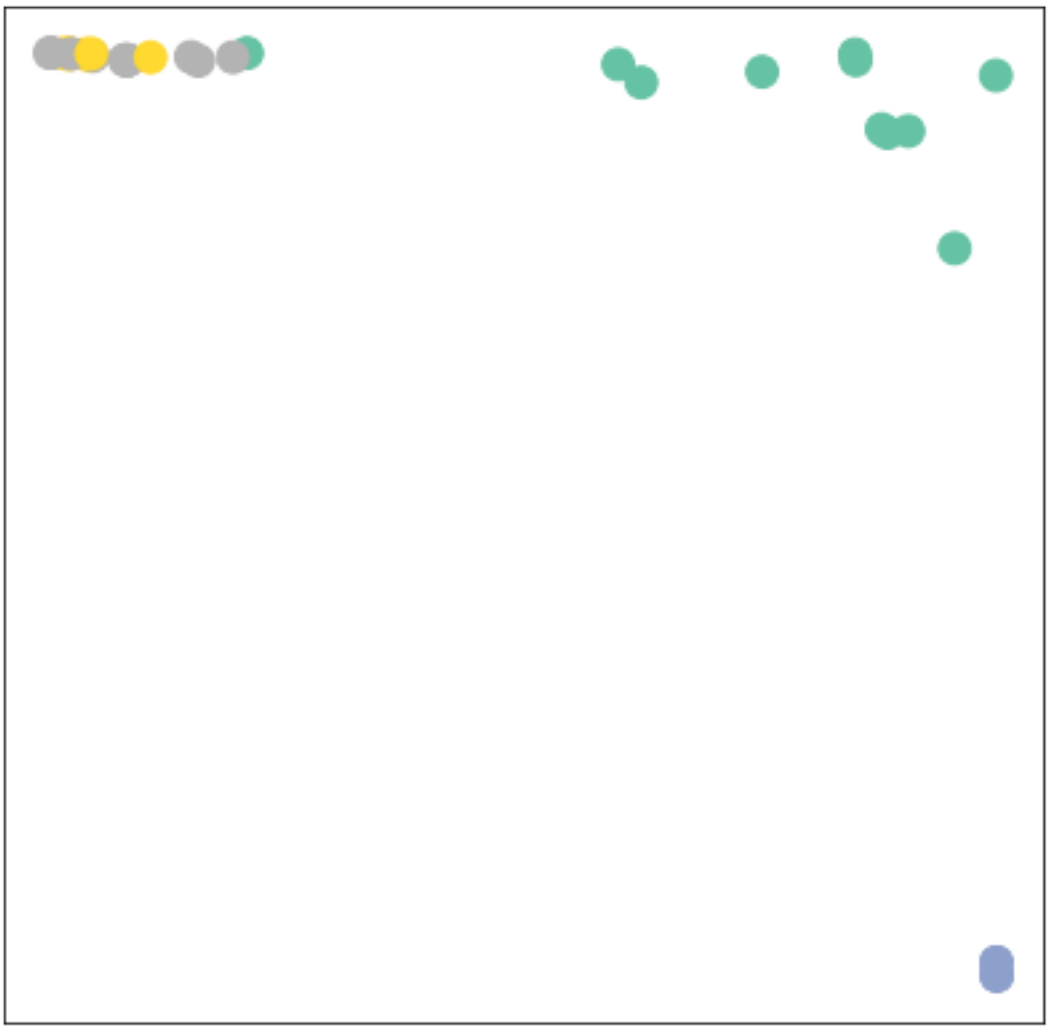
Epoch: 250, Loss: 0.3702



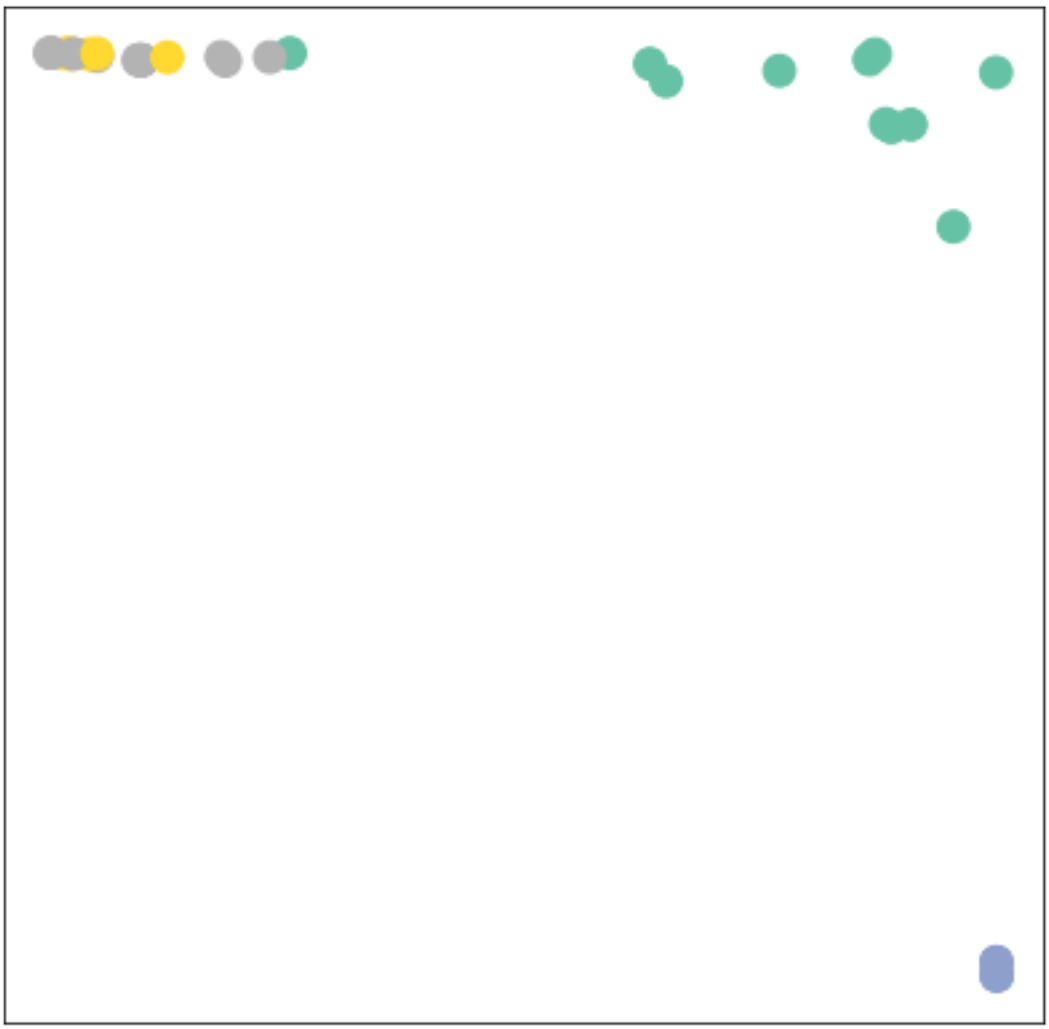
Epoch: 260, Loss: 0.3680



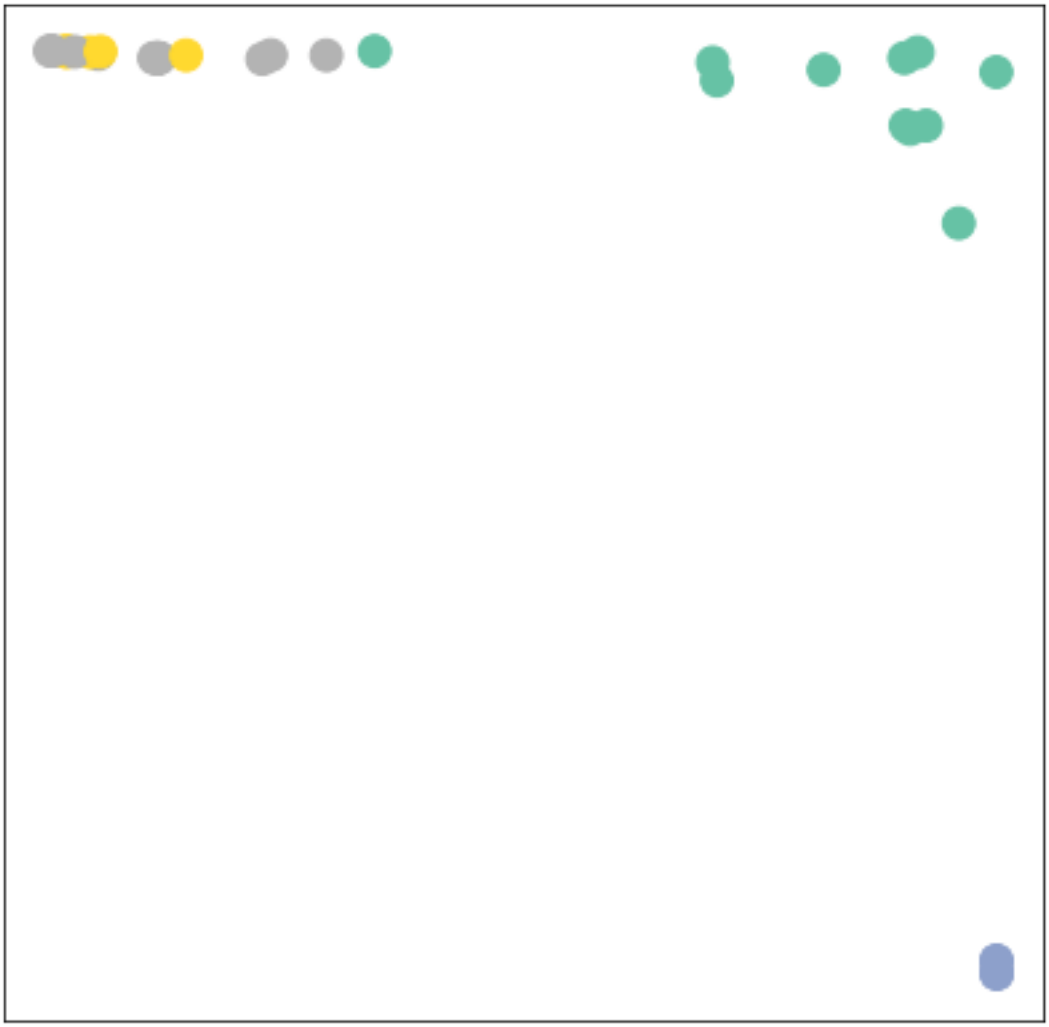
Epoch: 270, Loss: 0.3646



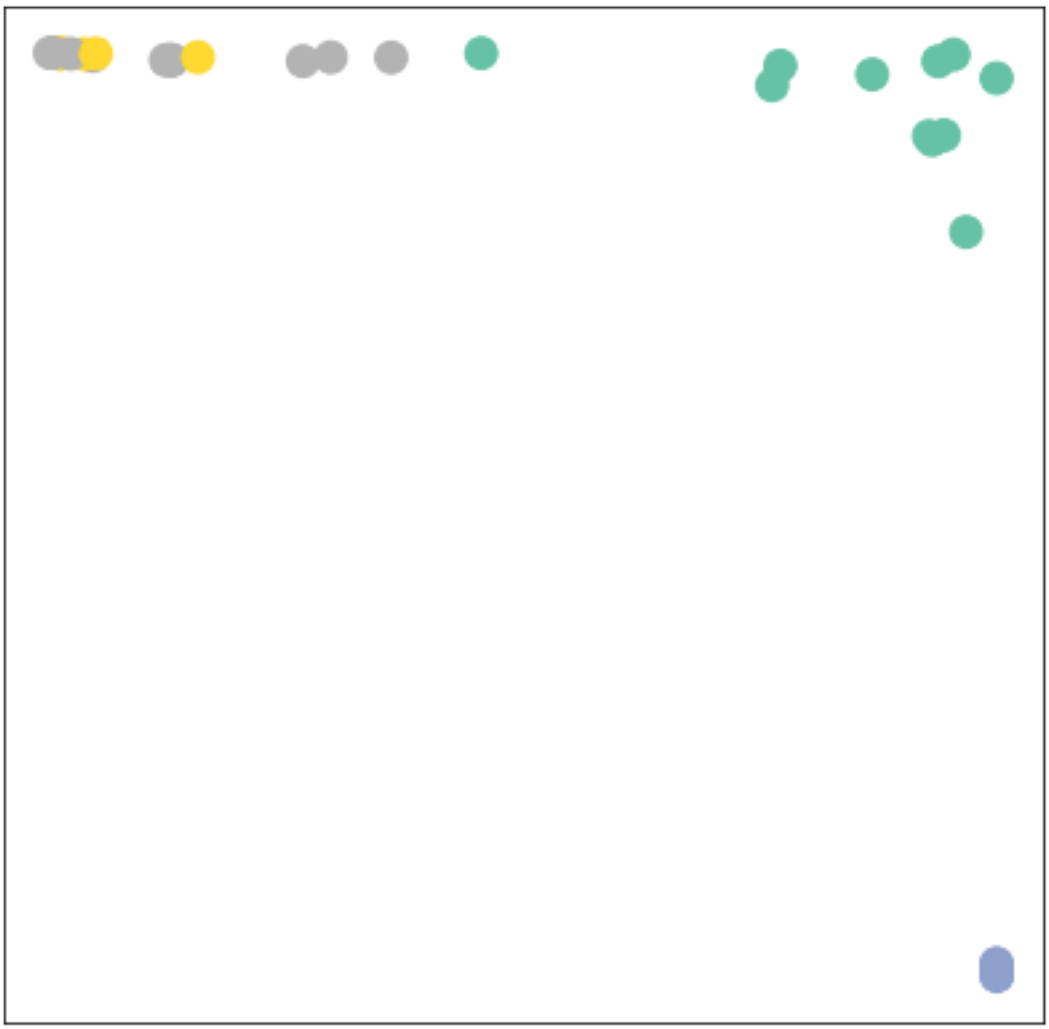
Epoch: 280, Loss: 0.3585



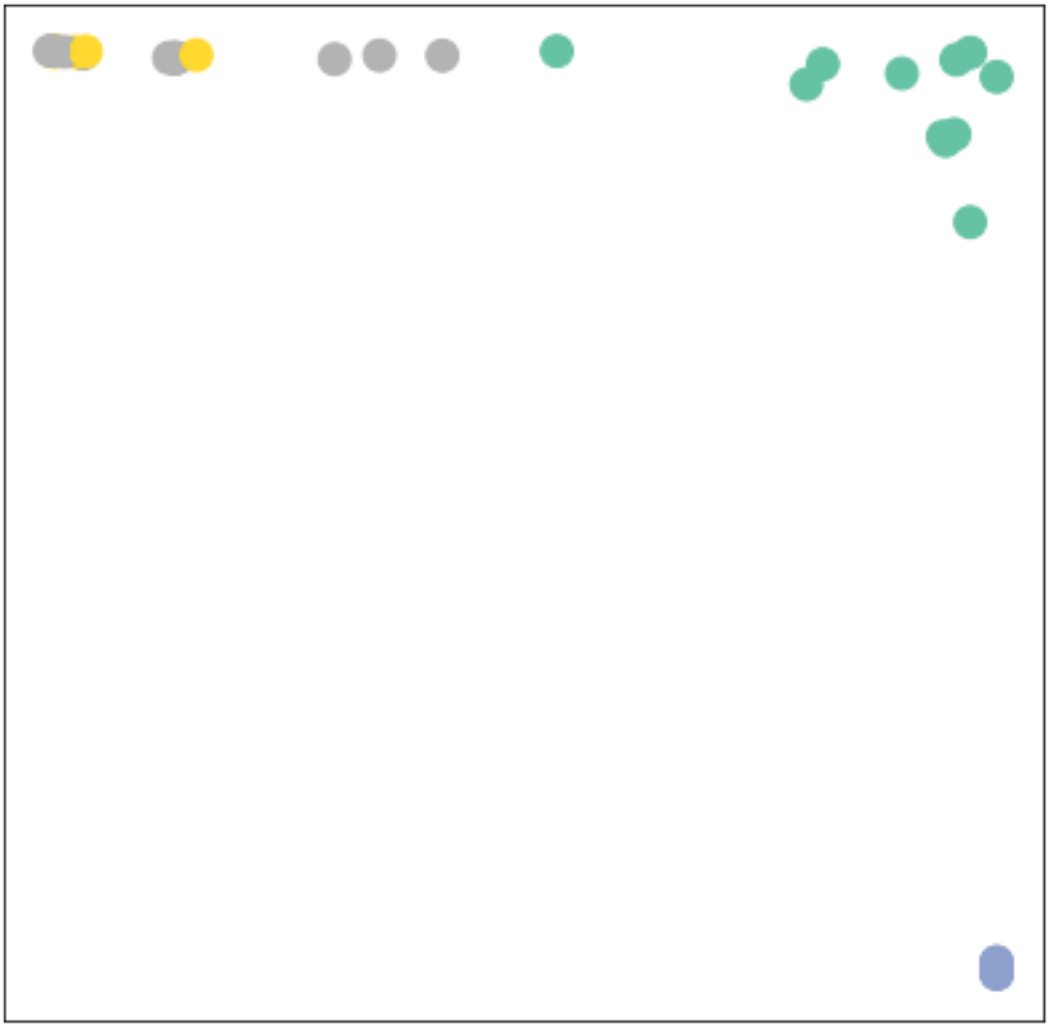
Epoch: 290, Loss: 0.3474



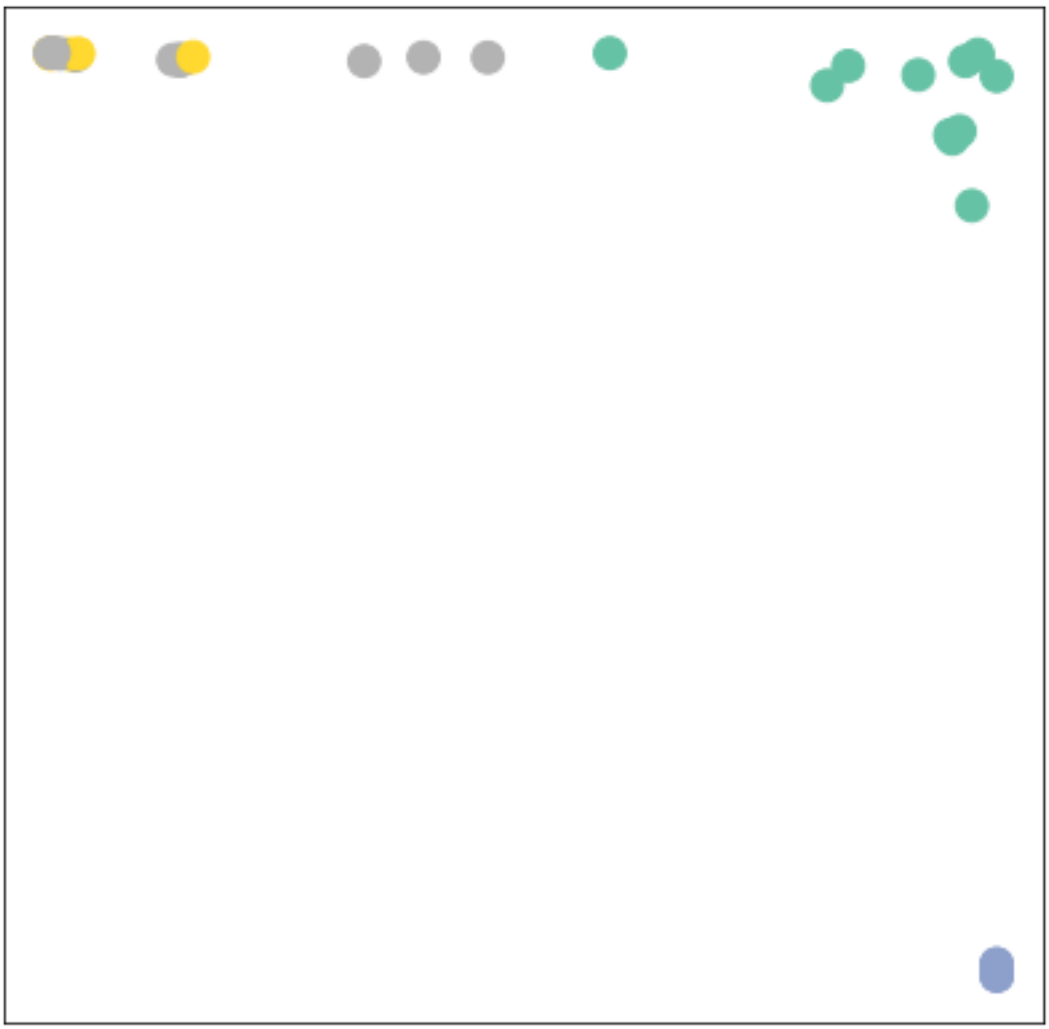
Epoch: 300, Loss: 0.3296



Epoch: 310, Loss: 0.3052



Epoch: 320, Loss: 0.2785

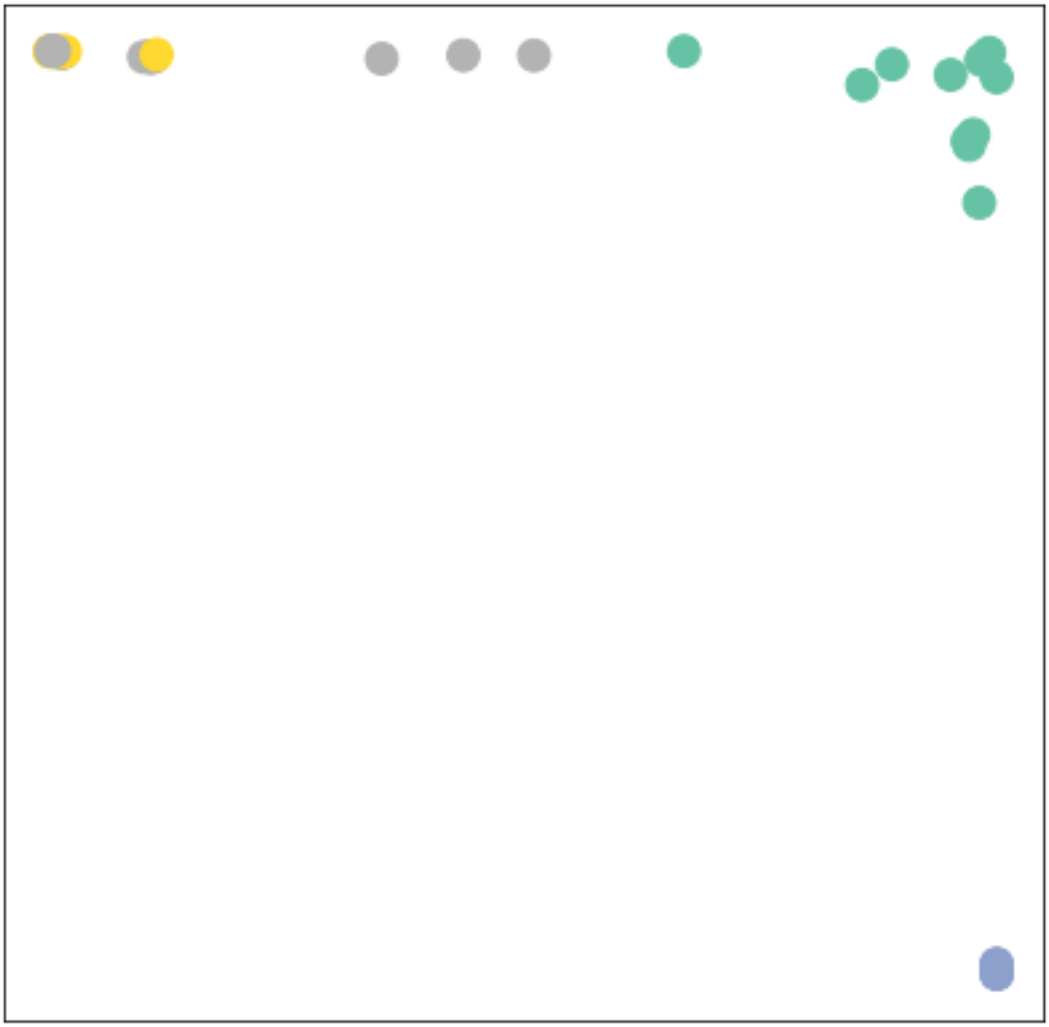


Epoch: 330, Loss: 0.2552

Epoch: 340, Loss: 0.2378

Epoch: 350, Loss: 0.2248

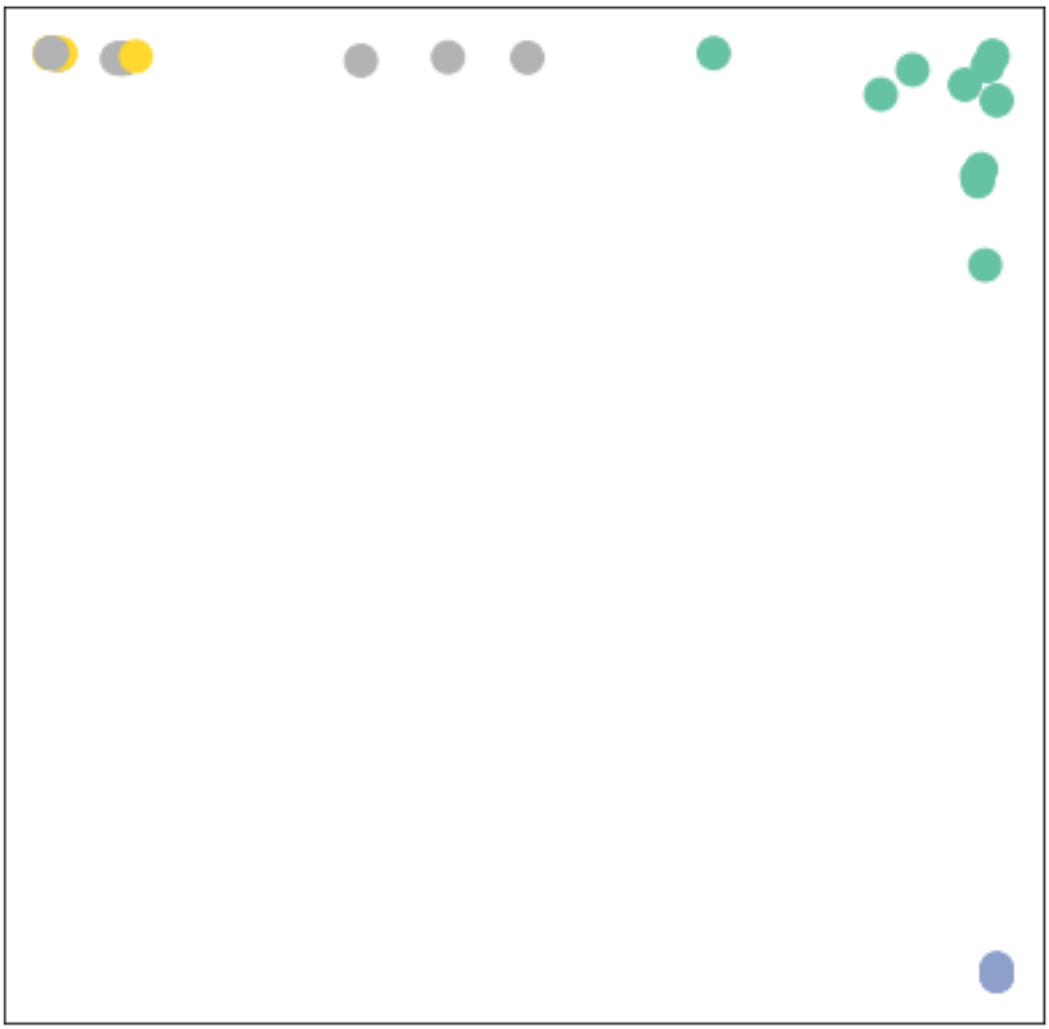
Epoch: 360, Loss: 0.2139



Epoch: 370, Loss: 0.2043

Epoch: 380, Loss: 0.1956

Epoch: 390, Loss: 0.1877



As one can see, our 3-layer GCN model manages to linearly separating the communities and classifying most of the nodes correctly.

Note that we do not need to reimplement standard GNN architectures, the library provides implementations for the most popular ones, including the GCN Layer:

`torch_geometric.nn.GCNConv`

3 Example: graph classification

Let's now have a closer look at how to apply **Graph Neural Networks (GNNs) to the task of graph classification**. Graph classification refers to the problem of classifiying entire graphs (in contrast to nodes), given a **dataset of graphs**, based on some structural graph properties. Here,

we want to embed entire graphs, and we want to embed those graphs in such a way so that they are linearly separable given a task at hand.

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

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

[32]:
```
import torch
from torch_geometric.datasets import TUDataset

dataset = TUDataset(root='data/TUDataset', name='MUTAG')

print()
print(f'Dataset: {dataset}:')
print('====================')
print(f'Number of graphs: {len(dataset)}')
print(f'Number of features: {dataset.num_features}')
print(f'Number of classes: {dataset.num_classes}')

data = dataset[0]  # Get the first graph object.

print()
print(data)
print('=============================================================')

# Gather some statistics about the first graph.
print(f'Number of nodes: {data.num_nodes}')
print(f'Number of edges: {data.num_edges}')
print(f'Average node degree: {data.num_edges / data.num_nodes:.2f}')
print(f'Contains isolated nodes: {data.contains_isolated_nodes()}')
print(f'Contains self-loops: {data.contains_self_loops()}')
print(f'Is undirected: {data.is_undirected()}')
```

```
Downloading https://www.chrsmrrs.com/graphkerneldatasets/MUTAG.zip
Extracting data/TUDataset/MUTAG/MUTAG.zip
Processing...
Done!

Dataset: MUTAG(188):
====================
Number of graphs: 188
Number of features: 7
```

```
Number of classes: 2

Data(edge_attr=[38, 4], edge_index=[2, 38], x=[17, 7], y=[1])
=============================================================
Number of nodes: 17
Number of edges: 38
Average node degree: 2.24
Contains isolated nodes: False
Contains self-loops: False
Is undirected: True
```

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

By inspecting the first graph object of the dataset, we can see that it comes with **17 nodes (with 7-dimensional feature vectors)** and **38 edges** (leading to an average node degree of 2.24). It also comes with exactly **one graph label** (`y=[1]`), and, in addition to previous datasets, provides addtional **4-dimensional edge features** (`edge_attr=[38, 4]`). However, for the sake of simplicity, we will not make use of those.

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

[33]:
```
torch.manual_seed(12345)
dataset = dataset.shuffle()

train_dataset = dataset[:150]
test_dataset = dataset[150:]

print(f'Number of training graphs: {len(train_dataset)}')
print(f'Number of test graphs: {len(test_dataset)}')
```

```
Number of training graphs: 150
Number of test graphs: 38
```

3.1 Mini-batching of graphs

Since graphs in graph classification datasets are usually small, a good idea is to **batch the graphs** before inputting them into a Graph Neural Network to guarantee full GPU utilization. In the image or language domain, this procedure is typically achieved by **rescaling** or **padding** each example into a set of equally-sized shapes, and examples are then grouped in an additional dimension. The length of this dimension is then equal to the number of examples grouped in a mini-batch and is typically referred to as the `batch_size`.

However, for GNNs the two approaches described above are either not feasible or may result in a lot of unnecessary memory consumption. Therefore, PyTorch Geometric opts for another approach to achieve parallelization across a number of examples. Here, adjacency matrices are stacked in

a diagonal fashion (creating a giant graph that holds multiple isolated subgraphs), and node and target features are simply concatenated in the node dimension:

This procedure has some crucial advantages over other batching procedures:

1. GNN operators that rely on a message passing scheme do not need to be modified since messages are not exchanged between two nodes that belong to different graphs.
2. There is no computational or memory overhead since adjacency matrices are saved in a sparse fashion holding only non-zero entries, *i.e.*, the edges.

PyTorch Geometric automatically takes care of **batching multiple graphs into a single giant graph** with the help of the `torch_geometric.data.DataLoader` class:

[34]:
```
from torch_geometric.data import DataLoader

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

for step, data in enumerate(train_loader):
    print(f'Step {step + 1}:')
    print('=======')
    print(f'Number of graphs in the current batch: {data.num_graphs}')
    print(data)
    print()
```

```
Step 1:
=======
Number of graphs in the current batch: 64
Batch(batch=[1185], edge_attr=[2624, 4], edge_index=[2, 2624], ptr=[65],
x=[1185, 7], y=[64])

Step 2:
=======
Number of graphs in the current batch: 64
Batch(batch=[1146], edge_attr=[2538, 4], edge_index=[2, 2538], ptr=[65],
x=[1146, 7], y=[64])

Step 3:
=======
Number of graphs in the current batch: 22
Batch(batch=[383], edge_attr=[832, 4], edge_index=[2, 832], ptr=[23], x=[383,
7], y=[22])
```

3.2 Training a Graph Neural Network (GNN)

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

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

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

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

PyTorch Geometric provides this functionality via `torch_geometric.nn.global_mean_pool`, which takes in the node embeddings of all nodes in the mini-batch and the assignment vector `batch` to compute a graph embedding of size `[batch_size, hidden_channels]` for each graph in the batch.

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

[38]:
```
from torch.nn import Linear
import torch.nn.functional as F
from torch_geometric.nn import GCNConv
from torch_geometric.nn import global_mean_pool


class GCN(torch.nn.Module):
    def __init__(self, hidden_channels):
        super(GCN, self).__init__()
        torch.manual_seed(1234567)
        self.conv1 = GCNConv(dataset.num_node_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.conv3 = GCNConv(hidden_channels, hidden_channels)
        self.lin = Linear(hidden_channels, dataset.num_classes)

    def forward(self, x, edge_index, batch):
        # 1. Obtain node embeddings
        x = self.conv1(x, edge_index)
        x = x.relu()
        x = self.conv2(x, edge_index)
        x = x.relu()
        x = self.conv3(x, edge_index)

        # 2. Readout layer
        x = global_mean_pool(x, batch)  # [batch_size, hidden_channels]

        # 3. Apply a final classifier
```

```
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.lin(x)

        return x

model = GCN(hidden_channels=64)
print(model)
```

```
GCN(
  (conv1): GCNConv(7, 64)
  (conv2): GCNConv(64, 64)
  (conv3): GCNConv(64, 64)
  (lin): Linear(in_features=64, out_features=2, bias=True)
)
```

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

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

[40]:
```
#from IPython.display import Javascript
#display(Javascript('''google.colab.output.setIframeHeight(0, true, {maxHeight:␣
 ↪300})'''))

model = GCN(hidden_channels=64)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = torch.nn.CrossEntropyLoss()

def train():
    model.train()

    for data in train_loader:  # Iterate in batches over the training dataset.
         out = model(data.x, data.edge_index, data.batch)  # Perform a single␣
 ↪forward pass.
         loss = criterion(out, data.y)  # Compute the loss.
         loss.backward()  # Derive gradients.
         optimizer.step()  # Update parameters based on gradients.
         optimizer.zero_grad()  # Clear gradients.

def test(loader):
     model.eval()

     correct = 0
     for data in loader:  # Iterate in batches over the training/test dataset.
         out = model(data.x, data.edge_index, data.batch)
         pred = out.argmax(dim=1)  # Use the class with highest probability.
```

```
        correct += int((pred == data.y).sum())  # Check against ground-truth␣
↪labels.
    return correct / len(loader.dataset)  # Derive ratio of correct predictions.


for epoch in range(1, 51):
    train()
    train_acc = test(train_loader)
    test_acc = test(test_loader)
    print(f'Epoch: {epoch:03d}, Train Acc: {train_acc:.4f}, Test Acc: {test_acc:
↪.4f}')
```

```
Epoch: 001, Train Acc: 0.6467, Test Acc: 0.7368
Epoch: 002, Train Acc: 0.6467, Test Acc: 0.7368
Epoch: 003, Train Acc: 0.6467, Test Acc: 0.7368
Epoch: 004, Train Acc: 0.6533, Test Acc: 0.7368
Epoch: 005, Train Acc: 0.6533, Test Acc: 0.7632
Epoch: 006, Train Acc: 0.7200, Test Acc: 0.7632
Epoch: 007, Train Acc: 0.7333, Test Acc: 0.7632
Epoch: 008, Train Acc: 0.7067, Test Acc: 0.7895
Epoch: 009, Train Acc: 0.7067, Test Acc: 0.6579
Epoch: 010, Train Acc: 0.7267, Test Acc: 0.7632
Epoch: 011, Train Acc: 0.6800, Test Acc: 0.7632
Epoch: 012, Train Acc: 0.7600, Test Acc: 0.7632
Epoch: 013, Train Acc: 0.7133, Test Acc: 0.7895
Epoch: 014, Train Acc: 0.7200, Test Acc: 0.7895
Epoch: 015, Train Acc: 0.7467, Test Acc: 0.7632
Epoch: 016, Train Acc: 0.7467, Test Acc: 0.7368
Epoch: 017, Train Acc: 0.7200, Test Acc: 0.7895
Epoch: 018, Train Acc: 0.7200, Test Acc: 0.7895
Epoch: 019, Train Acc: 0.7267, Test Acc: 0.7895
Epoch: 020, Train Acc: 0.7133, Test Acc: 0.7895
Epoch: 021, Train Acc: 0.7133, Test Acc: 0.7895
Epoch: 022, Train Acc: 0.7267, Test Acc: 0.7632
Epoch: 023, Train Acc: 0.7467, Test Acc: 0.7368
Epoch: 024, Train Acc: 0.7533, Test Acc: 0.7368
Epoch: 025, Train Acc: 0.7267, Test Acc: 0.8158
Epoch: 026, Train Acc: 0.7200, Test Acc: 0.8421
Epoch: 027, Train Acc: 0.7467, Test Acc: 0.7632
Epoch: 028, Train Acc: 0.7533, Test Acc: 0.7632
Epoch: 029, Train Acc: 0.7400, Test Acc: 0.7632
Epoch: 030, Train Acc: 0.7400, Test Acc: 0.7632
Epoch: 031, Train Acc: 0.7467, Test Acc: 0.7632
Epoch: 032, Train Acc: 0.7533, Test Acc: 0.7632
Epoch: 033, Train Acc: 0.7533, Test Acc: 0.7895
Epoch: 034, Train Acc: 0.7533, Test Acc: 0.7632
Epoch: 035, Train Acc: 0.7667, Test Acc: 0.7632
```

```
Epoch: 036, Train Acc: 0.7533, Test Acc: 0.7632
Epoch: 037, Train Acc: 0.7533, Test Acc: 0.7895
Epoch: 038, Train Acc: 0.7533, Test Acc: 0.7895
Epoch: 039, Train Acc: 0.7667, Test Acc: 0.7632
Epoch: 040, Train Acc: 0.7667, Test Acc: 0.7632
Epoch: 041, Train Acc: 0.7600, Test Acc: 0.7632
Epoch: 042, Train Acc: 0.7600, Test Acc: 0.7632
Epoch: 043, Train Acc: 0.7667, Test Acc: 0.7895
Epoch: 044, Train Acc: 0.7667, Test Acc: 0.7632
Epoch: 045, Train Acc: 0.7667, Test Acc: 0.7632
Epoch: 046, Train Acc: 0.7600, Test Acc: 0.7632
Epoch: 047, Train Acc: 0.7733, Test Acc: 0.7632
Epoch: 048, Train Acc: 0.7400, Test Acc: 0.7895
Epoch: 049, Train Acc: 0.7667, Test Acc: 0.7895
Epoch: 050, Train Acc: 0.7667, Test Acc: 0.7632
Epoch: 051, Train Acc: 0.7667, Test Acc: 0.7632
Epoch: 052, Train Acc: 0.7667, Test Acc: 0.8158
Epoch: 053, Train Acc: 0.7733, Test Acc: 0.7895
Epoch: 054, Train Acc: 0.7667, Test Acc: 0.7632
Epoch: 055, Train Acc: 0.7667, Test Acc: 0.7632
Epoch: 056, Train Acc: 0.7667, Test Acc: 0.7632
Epoch: 057, Train Acc: 0.7733, Test Acc: 0.7895
Epoch: 058, Train Acc: 0.7733, Test Acc: 0.7895
Epoch: 059, Train Acc: 0.7667, Test Acc: 0.7632
Epoch: 060, Train Acc: 0.7667, Test Acc: 0.7632
Epoch: 061, Train Acc: 0.7733, Test Acc: 0.7895
Epoch: 062, Train Acc: 0.7667, Test Acc: 0.7895
Epoch: 063, Train Acc: 0.7800, Test Acc: 0.7895
Epoch: 064, Train Acc: 0.7800, Test Acc: 0.7632
Epoch: 065, Train Acc: 0.7800, Test Acc: 0.7632
Epoch: 066, Train Acc: 0.7800, Test Acc: 0.8158
Epoch: 067, Train Acc: 0.7800, Test Acc: 0.7632
Epoch: 068, Train Acc: 0.7733, Test Acc: 0.7632
Epoch: 069, Train Acc: 0.7867, Test Acc: 0.7632
Epoch: 070, Train Acc: 0.7867, Test Acc: 0.7632
Epoch: 071, Train Acc: 0.7800, Test Acc: 0.7632
Epoch: 072, Train Acc: 0.7800, Test Acc: 0.7632
Epoch: 073, Train Acc: 0.7733, Test Acc: 0.7895
Epoch: 074, Train Acc: 0.7733, Test Acc: 0.8158
Epoch: 075, Train Acc: 0.7733, Test Acc: 0.8158
Epoch: 076, Train Acc: 0.7800, Test Acc: 0.7632
Epoch: 077, Train Acc: 0.7733, Test Acc: 0.7632
Epoch: 078, Train Acc: 0.7733, Test Acc: 0.7895
Epoch: 079, Train Acc: 0.7733, Test Acc: 0.7895
Epoch: 080, Train Acc: 0.7800, Test Acc: 0.7632
Epoch: 081, Train Acc: 0.7733, Test Acc: 0.7632
Epoch: 082, Train Acc: 0.7933, Test Acc: 0.7895
Epoch: 083, Train Acc: 0.7800, Test Acc: 0.7632
```

```
Epoch: 084, Train Acc: 0.7800, Test Acc: 0.7632
Epoch: 085, Train Acc: 0.7733, Test Acc: 0.7632
Epoch: 086, Train Acc: 0.7867, Test Acc: 0.8158
Epoch: 087, Train Acc: 0.7733, Test Acc: 0.7632
Epoch: 088, Train Acc: 0.7667, Test Acc: 0.7632
Epoch: 089, Train Acc: 0.7733, Test Acc: 0.7632
Epoch: 090, Train Acc: 0.7800, Test Acc: 0.7632
Epoch: 091, Train Acc: 0.7733, Test Acc: 0.7632
Epoch: 092, Train Acc: 0.7800, Test Acc: 0.7632
Epoch: 093, Train Acc: 0.7867, Test Acc: 0.8421
Epoch: 094, Train Acc: 0.7733, Test Acc: 0.7368
Epoch: 095, Train Acc: 0.7733, Test Acc: 0.7368
Epoch: 096, Train Acc: 0.7933, Test Acc: 0.7368
Epoch: 097, Train Acc: 0.7733, Test Acc: 0.7632
Epoch: 098, Train Acc: 0.7733, Test Acc: 0.7105
Epoch: 099, Train Acc: 0.7867, Test Acc: 0.7895
Epoch: 100, Train Acc: 0.8267, Test Acc: 0.7368
```

As one can see, our model reaches around **76% test accuracy**. Reasons for the fluctations in accuracy can be explained by the rather small dataset (only 38 test graphs), and usually disappear once one applies GNNs to larger datasets.