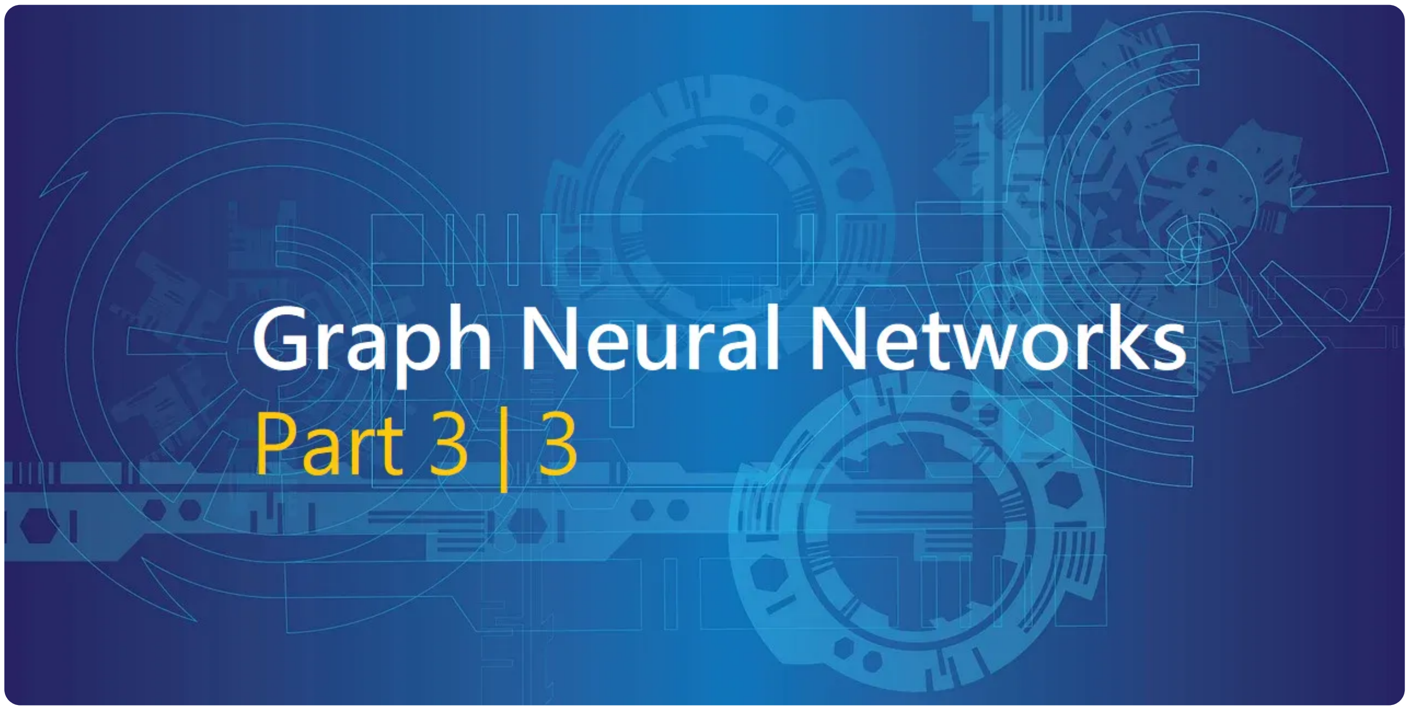
September 2, 2020

Understanding Graph Neural Networks | Part 3/3



Welcome back to the final part of this Blog Series on Graph Neural Networks! 🙂
In the following I'll give a quick introduction to PyTorch Geometric and afterwards we will build our first Graph Neural Network with this library!

This part of the series is also available as a Google Colab Notebook.

Here you find part 3 on YouTube.

PyTorch Geometric Introduction

Installation

PyTorch Geometric is a Python library for deep learning on irregular data structures, such as Graphs. It is commonly used as framework for building Graph Neural Networks. First we need to install it via pip, which can be done by the following lines:

Installing PyTorch Geometric

```
pip install -q torch-scatter==latest+cu101 -f https://pytorch-geometric.com/whl/tor
pip install -q torch-sparse==latest+cu101 -f https://pytorch-geometric.com/whl/torc
pip install -q git+https://github.com/rusty1s/pytorch_geometric.git
```

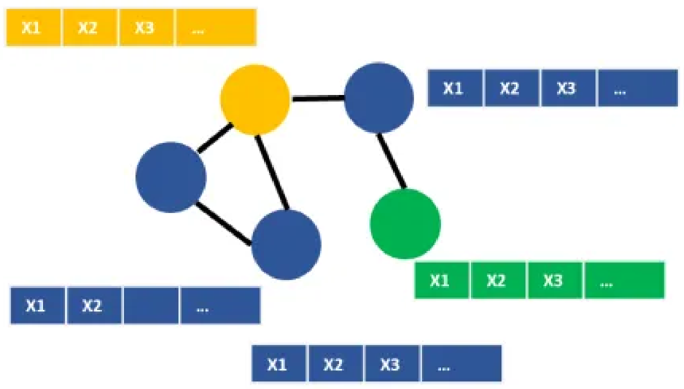
You might have to check your CUDA version, the installation above only works for CUDA 10.1 – you can find more related information in the official documentation.

How Data is represented in PyTorch Geometric

In the following we will quickly go over the key concepts of this library. As the name suggests, it is an extension of PyTorch and hence works the same way as building torch models. Data can be stored in a dedicated data object that holds different attributes, which are explained below:

- `data.x`: Node feature matrix with shape `[num_nodes, num_node_features]`

This means, we have a node feature-vector for each node in the graph, which can be represented as a matrix. If we recall the graph from the previous episodes, data.x simply holds the bars next to the nodes on the visualization, stacked as a matrix.



- `data.edge_index`: Graph connectivity in COO format with shape `[2, num_edges]` and type `torch.long`

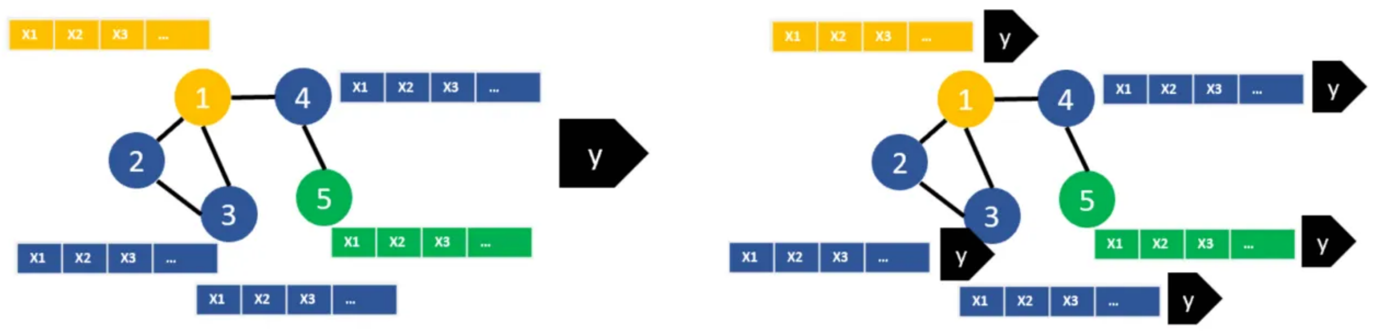
COO is a special format that is used to represent sparse matrices and stands for *coordinate list*. This means it contains 2-tuples of elements that are connected. This is an alternative form to the already mentioned adjacency matrix.

- `data.edge_attr`: Edge feature matrix with shape `[num_edges, num_edge_features]`

As explained before, edges can also have features, which are stored the same way as for the nodes – resulting in a matrix.

- `data.y`: Target to train against (may have arbitrary shape), *e.g.*, node-level targets of shape `[num_nodes, *]` or graph-level targets of shape `[1, *]`

Depending on the learning task we might have the predictions *y* on the node level, e.g. predictions for specific atoms or on the graph level, e.g. one prediction for the whole molecule. This is also illustrated in the following image:



Of course there can also be other kinds of predictions, such as link (edge) predictions, which would have the annotations on the edges.

Finally, the Data can be loaded with the included *Data Loader,* which supports batching, iterating, shuffling, efficient handling of the graph structure and many more functionalities.

In the following we will use a dataset that is already included in PyTorch Geometric. If you are interested in adding your own custom dataset you can find tutorials here.

Building Models with PyTorch Geometric

Building a neural network with PyTorch Geometric works the same way as building a neural network with PyTorch. We simply **inherit from torch.nn.Module** and build our architecture. The only fundamental difference is that we use the provided GNN-layers, such as *GCNConv*. Additionally, we can also define our own GNN layers with custom aggregate and update functions as explained here.

The library makes it very simple to build GNNs and still provides enough flexibility. Let’s have a look at this in practice!

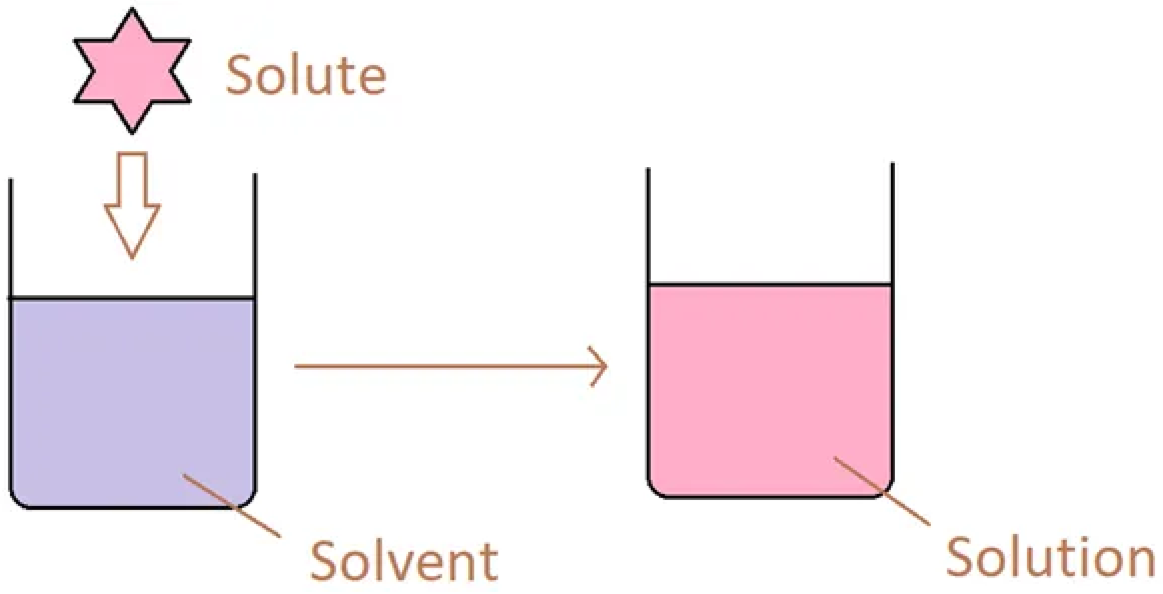
Our example dataset: ESOL

In the following we will use a dataset provided in the dataset collection of PyTorch Geometric (Here you find all datasets). The Dataset comes from the MoleculeNet collection, which can be found here.

This is the official description:

> “ ESOL is a small dataset consisting of water solubility data for 1128 compounds. The dataset has been used to train models that estimate solubility directly from chemical structures (as encoded in SMILES strings). Note that these structures don’t include 3D coordinates, since solubility is a property of a molecule and not of its particular conformers.

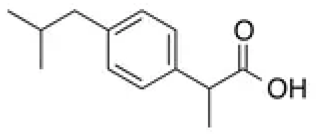
In other words: Our Machine Learning Task is to predict how different molecules dissolve in water.



Source

We can already see, that this Machine Learning Task is based on *continuous* **graph-level predictions**, which means we have to represent the whole graph structure as one embedding. This will then be fed into a Linear Layer for the Regression output. In a second we will see how simple it is to build a model for that with PyTorch Geometric.

Before continuing, let's quickly go over a couple of concepts that are related to molecule data. Molecules can be represented using the so called SMILES (*Simplified Molecular Input Line Entry Specification*) format. The image below illustrates the SMILES string of Ibuprofen.



Ibuprofen
CC(C)Cc1ccc(cc1)C(C)C(O)=O

Source

You might think now: **Why not simply using this String as input for the Neural Network?**

The first problem with this approach is that a molecule might have multiple SMILES Strings and is therefore not uniquely identified. More precisely, there exist different notations of this format which makes the data alignment difficult.

Additionally, the model will focus on the *grammar* rather than the *actual structure* of the molecule. Chemical graphs however, are invariant to permutations (which is not the case for grammar), and therefore we need another architecture to handle this setup – Graph Neural Networks.

Loading the Data

The ESOL-data can be directly loaded in PyTorch Geometric, however we first need to install another library: RDKit. This chemoinformatics framework is commonly used when dealing with molecule data – it contains various functions to work with graph-structured chemical data. **Currently RDKit can only be installed using Anaconda!** Please have a look at the Colab Notebook to find out how Rdkit can be installed in the Cloud.

Then we can import the the dataset using:

```
import rdkit
from torch_geometric.datasets import MoleculeNet

# Load the ESOL dataset
data = MoleculeNet(root=".", name="ESOL")
```

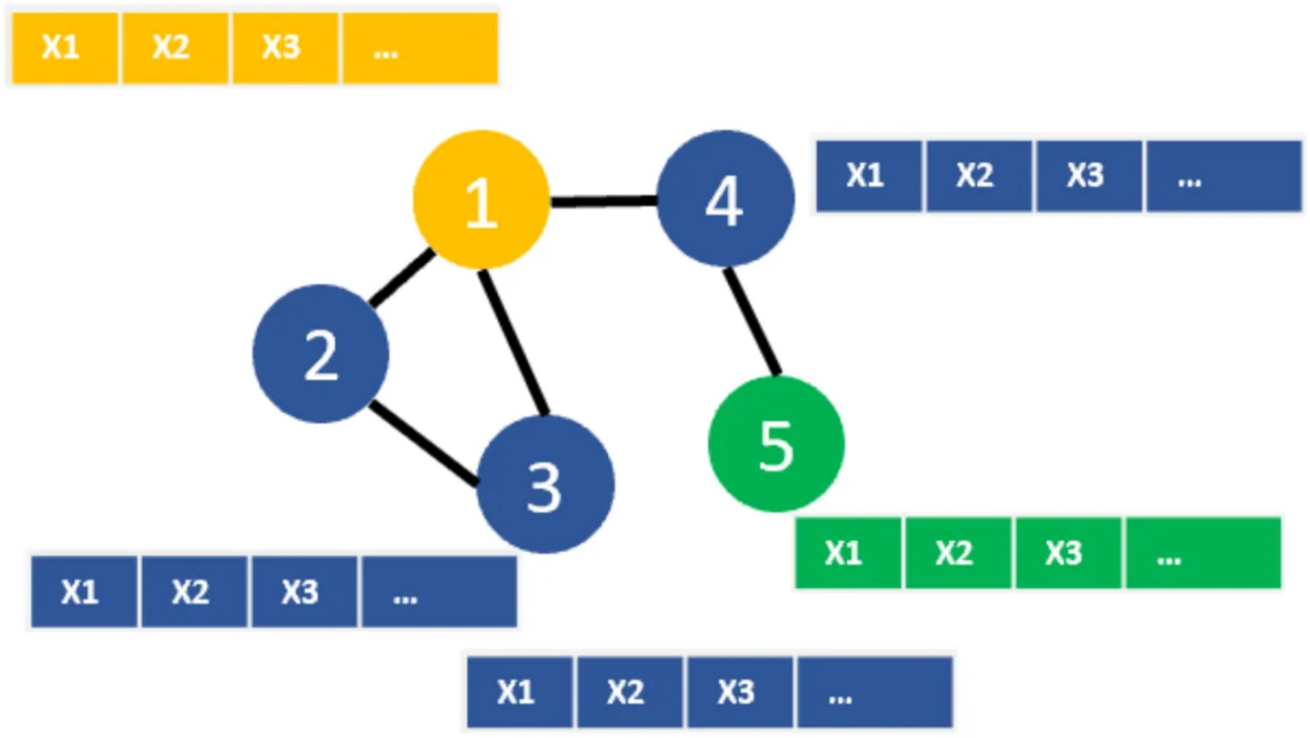
The data variable now contains our Graph Data, which has the type: **torch_geometric.datasets.molecule_net.MoleculeNet**.

We can observe some basic insights by running:

```
print("Dataset features: ", data.num_features)
print("Dataset target: ", data.num_classes)
print("Dataset length: ", data.len)
print("Dataset sample: ", data[0])
print("Sample  nodes: ", data[0].num_nodes)
print("Sample  edges: ", data[0].num_edges)

# OUTPUT:
Dataset type:  <class 'torch_geometric.datasets.molecule_net.MoleculeNet'>
Dataset features:  9
Dataset target:  1
Dataset length:  <bound method InMemoryDataset.len of ESOL(1128)>
Dataset sample:  Data(edge_attr=[68, 3], edge_index=[2, 68], smiles="OCC3OC(OCC2OC(
Sample  nodes:  32
Sample  edges:  68
```

We see that the dataset contains 1128 molecules and we have 9 features on the node level. As illustrated before, this means that the vector next to each node contains 9 elements x_1, …, x_9. Additionally, if we take a closer look at the first molecule, we see the SMILES string and the number of nodes (32) and edges (68).



We can actually also print the node features for the first node and get something like this:

```
print(data[0].x)
# OUTPUT:
tensor([[8, 0, 2, 5, 1, 0, 4, 0, 0],
        [6, 0, 4, 5, 2, 0, 4, 0, 0],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        ...
        [8, 0, 2, 5, 1, 0, 4, 0, 0],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        [8, 0, 2, 5, 1, 0, 4, 0, 0]])
```

The same way we can investigate the edge information by looking at the edge index:

```
print(data[0].edge_index.t())
# OUTPUT:
tensor([[ 0,  1],
        [ 1,  0],
        [ 1,  2],
        [ 2,  1],
        [ 2,  3],
      ...
        [29, 28],
        [30,  2],
        [30, 28],
        [30, 31],
        [31, 30]])
```

We see the aforementioned tuples of the COO representation. The first two entries for example tell us that node 0 is connected to node 1 and vice versa.

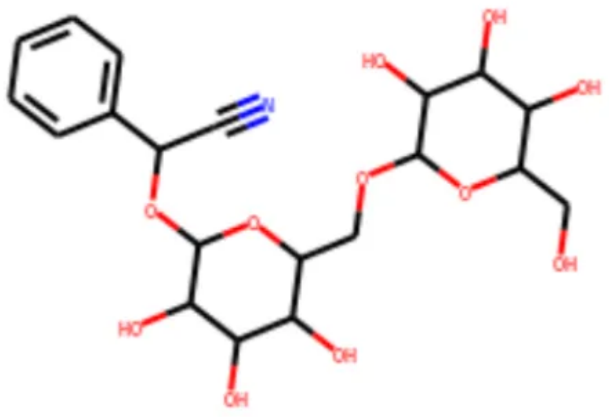
Finally, if we look at the output variable y, we can see that we have **one value for each molecule**, e.g. data[0].y has the value -0.770. This means that we operate on a graph-level, as visualized before and we need to combine the hidden states of the nodes.

Visualizing molecules with RDKit / Feature Engineering

Before we finally built the model, let's quickly see how we can visualize the molecule data very easily with RDKit by utilizing the SMILES representation. For example we can use the following code to build an image of the molecule including the atoms and bonds.

```
from rdkit import Chem
from rdkit.Chem.Draw import IPythonConsole
molecule = Chem.MolFromSmiles(data[0]["smiles"])

# Display the molecule image in a Jupyter Notebook
molecule
```



RDKit includes a powerful drawing engine and you can find more examples to play around here.

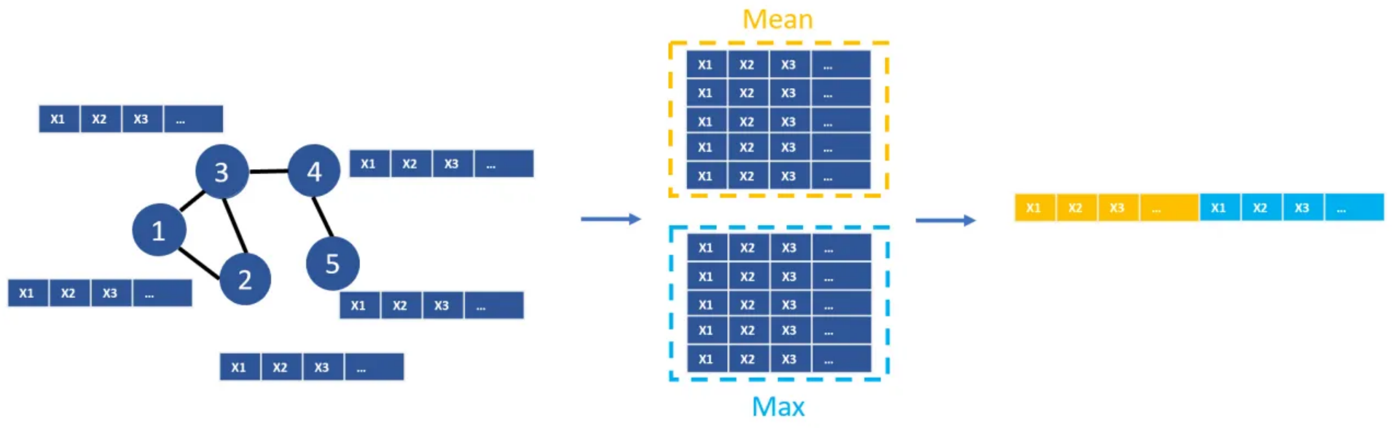
In our case we have the node features already provided in the loaded dataset of PyTorch Geometric. For real world data this is certainly not the case and therefore you will have to get those features by yourself. This could be done using the various functions from RDKit applied on loaded molecules (objects of type *rdkit.Chem.rdchem.Mol*). For instance, you can iterate over the atoms of the molecule object and store the one-hot encoded atom type (such as hydrogen or carbon) as a first feature. This great getting started page can answer all your questions regarding the feature engineering part.

Building the Graph Neural Network Model

The code below builds our simple GNN model. First we import the GCNConv layer from PyTorch Geometric and start with a first layer that translates our node features into the size of the embedding. After that we stack three more Message Passing Layers. This means that we have in total four neighborhood hops to gather information.

We use a tanh activation function between the layers. Afterwards, we combine the node embeddings to one embedding vector by applying a pooling operation. In our case we use both a *mean* and *max* operation over the node states. This is necessary as we want to obtain a prediction on a graph-level, and need a combined embedding. In contrast, when dealing with node-level predictions, you usually have to work with binary masks for the labels, but this is a topic for another blog post.

The illustration below shows how the information of individual node embeddings (of the **final** message passing layer…) can be combined to obtain a representation for the whole graph.



PyTorch Geometric provides various other Pooling layers, but here we want to keep it simple and use this combination of mean and max.

Finally, a linear output layer ensures that we get a continuous unbounded output value. It's input is the flattened vector from the drawing above.

```
import torch
from torch.nn import Linear
import torch.nn.functional as F
from torch_geometric.nn import GCNConv, TopKPooling, global_mean_pool
from torch_geometric.nn import global_mean_pool as gap, global_max_pool as gmp
embedding_size = 64

class GCN(torch.nn.Module):
    def __init__(self):
        # Init parent
        super(GCN, self).__init__()
        torch.manual_seed(42)

        # GCN layers
        self.initial_conv = GCNConv(data.num_features, embedding_size)
        self.conv1 = GCNConv(embedding_size, embedding_size)
        self.conv2 = GCNConv(embedding_size, embedding_size)
        self.conv3 = GCNConv(embedding_size, embedding_size)

        # Output layer
        self.out = Linear(embedding_size*2, data.num_classes)

    def forward(self, x, edge_index, batch_index):
        # First Conv layer
 (Projection)
        hidden = self.initial_conv(x, edge_index)
        hidden = F.tanh(hidden)

        # Other Conv layers
        hidden = self.conv1(hidden, edge_index)
        hidden = F.tanh(hidden)
        hidden = self.conv2(hidden, edge_index)
        hidden = F.tanh(hidden)
        hidden = self.conv3(hidden, edge_index)
        hidden = F.tanh(hidden)

        # Global Pooling Layer (stack different aggregations)
        hidden = torch.cat([gmp(hidden, batch_index),
                            gap(hidden, batch_index)], dim=1)

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

        return out, hidden

model = GCN()
print(model)
print("Number of parameters: ", sum(p.numel() for p in model.parameters()))

# OUTPUT:
# GCN(
#   (initial_conv): GCNConv(9, 64)
#   (conv1): GCNConv(64, 64)
#   (conv2): GCNConv(64, 64)
#   (conv3): GCNConv(64, 64)
#   (out): Linear(in_features=128, out_features=1, bias=True)
# )
#Number of parameters:  13249
```

If we print the model we get a summary of the layers and see that our 9 features are fed into the Message Passing layers, which produce hidden states of size 64, which are finally combined using the mean and max operation to obtain a regression output in the linear layer. The choice of the embedding size (64) is a hyperparameter and depends on factors such as the size of the graphs in the dataset.

Finally, our model has 13249 parameters, which seems reasonable, as we only have 1128 samples.

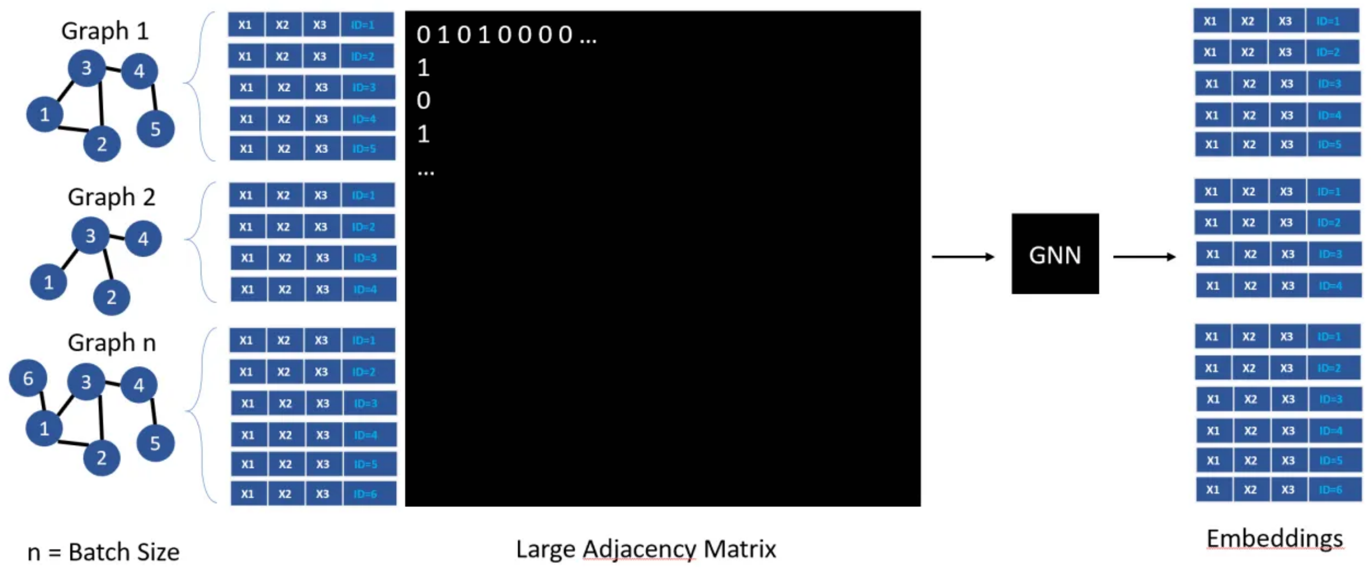
Training the Graph Neural Network Model

Next we want to train our model. As loss metric we use the root mean squared error (RMSE), such as suggested by the MoleculeNet website. As optimizer, Adam (Adaptive Movement Estimation) with a initial learning rate of 0.0007 is selected.

Next we wrap our dataset (separately for train and test) in the aforementioned Data Loader. This loader will do all the batching work for us.

Because the batching strategy (for parallelizing training with data) with graphs is slightly different than usually, I want to give some remarks on how it internally works in PyTorch Geometric's Data Loader.

As we have graphs with varying node sizes, the library cannot simply *hardcode* the dimensions of the batches. Instead it is required to combine the individual graphs into a large graph (the batch) that can be fed into the model. This means the Data Loader simply concatenates all node features and builds a large adjacency matrix that represents our combined graph. This way, we can simply feed in the mini-batches with differing number of nodes and edges into the GNN and directly obtain new embeddings, as illustrated below. If you are interested in this topic, you can find more information about it here.



Now back to the Code: We use a batch size of 64 (this means we have 64 graphs in our batch) and select the shuffle option. The first 80% of the data will be our train data, the rest is the test data.

Then we simply iterate over the batches of the Data Loader, which thankfully does all the work for us, as it is done in the *train* function. We call this train function # *epochs* times, in this case 2000.

```
from torch_geometric.data import DataLoader
import warnings
warnings.filterwarnings("ignore")

# Root mean squared error
loss_fn = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0007)

# Use GPU for training
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = model.to(device)

# Wrap data in a data loader
data_size = len(data)
NUM_GRAPHS_PER_BATCH = 64
loader = DataLoader(data[:int(data_size * 0.8)],
                    batch_size=NUM_GRAPHS_PER_BATCH, shuffle=True)
test_loader = DataLoader(data[int(data_size * 0.8):],
                         batch_size=NUM_GRAPHS_PER_BATCH, shuffle=True)

def train(data):
    # Enumerate over the data
    for batch in loader:
      # Use GPU
      batch.to(device)
      # Reset gradients
      optimizer.zero_grad()
      # Passing the node features and the connection info
      pred, embedding = model(batch.x.float(), batch.edge_index, batch.batch)
      # Calculating the loss and gradients
      loss = torch.sqrt(loss_fn(pred, batch.y))
      loss.backward()
      # Update using the gradients
      optimizer.step()
    return loss, embedding

print("Starting training...")
losses = []
for epoch in range(2000):
    loss, h = train(data)
```

```
      losses.append(loss)
      if epoch % 100 == 0:
        print(f"Epoch {epoch} | Train Loss {loss}")
```

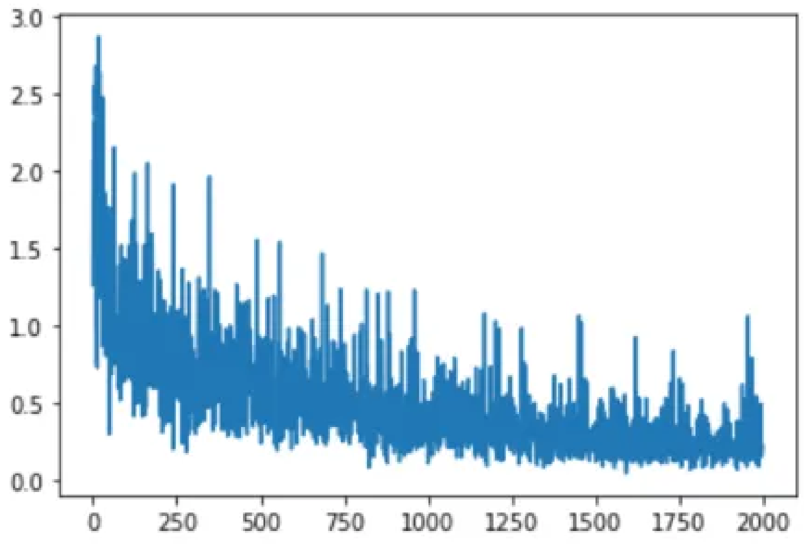
My training output looked like this:

```
Starting training...
Epoch 0 | Train Loss 1.2618728876113892
Epoch 100 | Train Loss 1.0729552507400513
Epoch 200 | Train Loss 1.2997239828109741
Epoch 300 | Train Loss 0.7074698209762573
Epoch 400 | Train Loss 0.8143591284751892
...
Epoch 1600 | Train Loss 0.243491068482399
Epoch 1700 | Train Loss 0.3084290325641632
Epoch 1800 | Train Loss 0.23583029210567474
Epoch 1900 | Train Loss 0.15630505979061127
```

We can also visualize the training loss (which was previously stored) using this code:

```
# Visualize learning (training loss)
import seaborn as sns
losses_float = [float(loss.cpu().detach().numpy()) for loss in losses]
loss_indices = [i for i,l in enumerate(losses_float)]
plt = sns.lineplot(loss_indices, losses_float)
plt
```

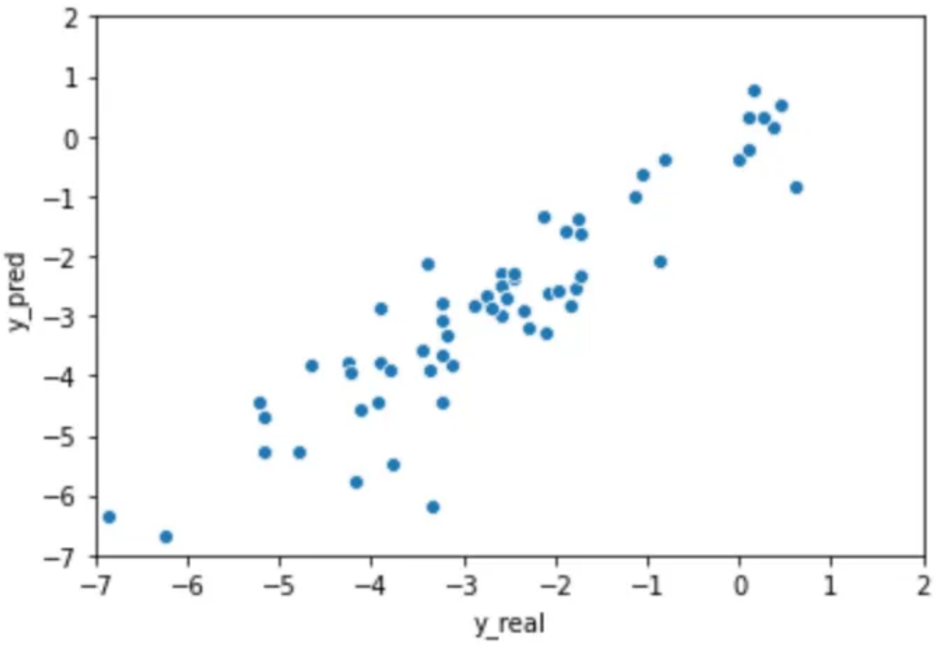
As result we get something like this:



As we can see, the loss decreases, even tough it is a bit noisy. This can be further improved by proper hyperparameter optimization, such as tweaking the learning rate (feel free to play around :)).

Validating the predictions

For a test batch we can now roughly check how good the water solubility predictions are. Therefore I simply plotted the real values against the predictions.



We can see that the outputs of our model strongly correlate with the actual values as we are close to a straight line (perfect model) – therefore the model isn't too bad 🙂

We have successfully trained our first GNN!

What did our model learn?

If you want to get a feeling for how the learned embeddings might look like, I can recommend a visualization of the latent spaces on this blog article (video at the end). Graphs which are similiar, are transformed by the GNN into similar latent spaces.

Improving the model

Of course you can add several improvements to the simple model of above. Here are just some ideas:

- Apply Dropouts
- Other (more intelligent) Pooling Layers (all layers here: https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#)
- Global Pooling Layers
- Batch Normalization
- More / Less MP layers
- Other hidden state sizes
- Add test metrics and Hyperparameter optimization
- …

I hope you enjoyed this series on Graph Neural Networks and had fun listening and playing around with the code. If you have questions or need help, feel free to drop a comment and I will do the best to support you! 🙂

Share this:

Twitter Facebook

You may also like