pjztestgat

March 7, 2024

1 Hands on with Graph Neural Networks

1.1 Installing Pytorch Geometric and RDKit

- Pytorch Geometric => Build Graph Neural Network
- RDKit => Handle Molecule Data

Note: Currently I didn't find a way to set a specific python version in colab. However, when installing rdkit, only specific python versions are supported… Hence it cannot be ensured that the notebook runs properly. For example Python 3.7.10 is not supported.

[29]:

```
# Enforce pytorch version 1.6.0
import torch
# if torch.__version__ != '1.6.0':
#   !pip uninstall torch -y
#   !pip uninstall torchvision -y
#   !pip install torch==1.6.0
#   !pip install torchvision==0.7.0

# # Check pytorch version and make sure you use a GPU Kernel
# !python -c "import torch; print(torch.__version__)"
# !python -c "import torch; print(torch.version.cuda)"
# !python --version
# !nvidia-smi
```

Make sure you clicked "RESTART RUNTIME" above (if torch version was different)!

[30]:

```
#@title
# # Install rdkit
# import sys
# import os
# import requests
# import subprocess
# import shutil
# from logging import getLogger, StreamHandler, INFO
# logger = getLogger(__name__)
# logger.addHandler(StreamHandler())
# logger.setLevel(INFO)


# def install(
#          chunk_size=4096,
#          file_name="Miniconda3-latest-Linux-x86_64.sh",
#          url_base="https://repo.continuum.io/miniconda/",
#          conda_path=os.path.expanduser(os.path.join("~", "miniconda")),
#          rdkit_version=None,
#          add_python_path=True,
#          force=False):
#     """install rdkit from miniconda
#     ```
#     import rdkit_installer
#     rdkit_installer.install()
#     ```
#     """

#     python_path = os.path.join(
#         conda_path,
#         "lib",
#         "python{0}.{1}".format(*sys.version_info),
#         "site-packages",
#     )

#     if add_python_path and python_path not in sys.path:
#         logger.info("add {} to PYTHONPATH".format(python_path))
#         sys.path.append(python_path)

#     if os.path.isdir(os.path.join(python_path, "rdkit")):
#         logger.info("rdkit is already installed")
#         if not force:
#             return

#         logger.info("force re-install")

#     url = url_base + file_name
#     python_version = "{0}.{1}.{2}".format(*sys.version_info)

#     logger.info("python version: {}".format(python_version))

#     if os.path.isdir(conda_path):
#         logger.warning("remove current miniconda")
#         shutil.rmtree(conda_path)
#     elif os.path.isfile(conda_path):
```

```
#         logger.warning("remove {}".format(conda_path))
#         os.remove(conda_path)

#     logger.info('fetching installer from {}'.format(url))
#     res = requests.get(url, stream=True)
#     res.raise_for_status()
#     with open(file_name, 'wb') as f:
#         for chunk in res.iter_content(chunk_size):
#             f.write(chunk)
#     logger.info('done')

#     logger.info('installing miniconda to {}'.format(conda_path))
#     subprocess.check_call(["bash", file_name, "-b", "-p", conda_path])
#     logger.info('done')

#     logger.info("installing rdkit")
#     subprocess.check_call([
#         os.path.join(conda_path, "bin", "conda"),
#         "install",
#         "--yes",
#         "-c", "rdkit",
#         "python==3.7.3",
#         "rdkit" if rdkit_version is None else "rdkit=={}".
 ↪format(rdkit_version)])
#     logger.info("done")

#     import rdkit
#     logger.info("rdkit-{} installation finished!".format(rdkit.__version__))


# if __name__ == "__main__":
#     install()
```

[31]:

```
# If something breaks in the notebook it is probably related to a mismatch␣
 ↪between the Python version, CUDA or torch
import torch
# pytorch_version = f"torch-{torch.__version__}.html"
# !pip install --no-index torch-scatter -f https://pytorch-geometric.com/whl/
 ↪$pytorch_version
# !pip install --no-index torch-sparse -f https://pytorch-geometric.com/whl/
 ↪$pytorch_version
# !pip install --no-index torch-cluster -f https://pytorch-geometric.com/whl/
 ↪$pytorch_version
# !pip install --no-index torch-spline-conv -f https://pytorch-geometric.com/
 ↪whl/$pytorch_version
# !pip install torch-geometric
```

1.2 Background info on the Dataset

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.

> "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."

>>> Machine Learning task: How are different molecules dissolving in water?

Source: https://www.differencebetween.com/difference-between-solubility-and-solubility-product/

1.2.1 SMILES representation and important sidenodes

Source: https://medium.com/@sunitachoudhary103/generating-molecules-using-a-char-rnn-in-pytorch-16885fd9394b

- Using the plain SMILES string as input is not suitable
- This will not consider the molecule structure but rather the grammar of the SMILES string
- The SMILES string can be different for a molecule, depending on the notation (a unique molecule can have multiple SMILES strings)
- Chemical graphs however, are invariant to permutations –> Graph Neural Networks

1.3 Looking into the Dataset

[32]:

```
import rdkit
from torch_geometric.datasets import MoleculeNet

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

[32]:

```
ESOL(1128)
```

Note: There seems to be a change in the Dataset class and somehow the target dim now equals 734 instead of one. You can simply ignore it at this point. :)

[33]:

```
# Investigating the dataset
print("Dataset type: ", type(data))
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)

# edge_index = graph connections
# smiles = molecule with its atoms
# x = node features (32 nodes have each 9 features)
# y = labels (dimension)
```

```
Dataset type:  <class 'torch_geometric.datasets.molecule_net.MoleculeNet'>
Dataset features:  9
Dataset target:  734
Dataset length:  <bound method InMemoryDataset.len of ESOL(1128)>
Dataset sample:  Data(x=[32, 9], edge_index=[2, 68], edge_attr=[68, 3],
smiles='OCC3OC(OCC2OC(OC(C#N)c1ccccc1)C(O)C(O)C2O)C(O)C(O)C3O ', y=[1, 1])
Sample  nodes:  32
Sample  edges:  68
```

[34]:

```
# Investigating the features
# Shape: [num_nodes, num_node_features]
data[0].x
```

[34]:

```
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, 0, 0, 4, 0, 1],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        [8, 0, 2, 5, 0, 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, 0, 0, 4, 0, 1],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        [8, 0, 2, 5, 0, 0, 4, 0, 0],
        [6, 0, 4, 5, 1, 0, 4, 0, 0],
        [6, 0, 2, 5, 0, 0, 2, 0, 0],
        [7, 0, 1, 5, 0, 0, 2, 0, 0],
        [6, 0, 3, 5, 0, 0, 3, 1, 1],
        [6, 0, 3, 5, 1, 0, 3, 1, 1],
        [6, 0, 3, 5, 1, 0, 3, 1, 1],
        [6, 0, 3, 5, 1, 0, 3, 1, 1],
        [6, 0, 3, 5, 1, 0, 3, 1, 1],
        [6, 0, 3, 5, 1, 0, 3, 1, 1],
        [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],
        [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],
        [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]])
```

[35]:

```
# Investigating the edges in sparse COO format
# Shape [2, num_edges]
data[0].edge_index.t()
```

[35]:

```
tensor([[ 0,  1],
        [ 1,  0],
        [ 1,  2],
        [ 2,  1],
        [ 2,  3],
        [ 2, 30],
        [ 3,  2],
        [ 3,  4],
        [ 4,  3],
        [ 4,  5],
        [ 4, 26],
        [ 5,  4],
        [ 5,  6],
        [ 6,  5],
        [ 6,  7],
        [ 7,  6],
        [ 7,  8],
        [ 7, 24],
        [ 8,  7],
        [ 8,  9],
        [ 9,  8],
        [ 9, 10],
        [ 9, 20],
        [10,  9],
        [10, 11],
        [11, 10],
        [11, 12],
        [11, 14],
        [12, 11],
        [12, 13],
        [13, 12],
        [14, 11],
        [14, 15],
        [14, 19],
        [15, 14],
        [15, 16],
        [16, 15],
        [16, 17],
        [17, 16],
        [17, 18],
        [18, 17],
        [18, 19],
        [19, 14],
        [19, 18],
        [20,  9],
        [20, 21],
        [20, 22],
        [21, 20],
        [22, 20],
        [22, 23],
        [22, 24],
        [23, 22],
        [24,  7],
        [24, 22],
        [24, 25],
        [25, 24],
        [26,  4],
        [26, 27],
        [26, 28],
        [27, 26],
        [28, 26],
        [28, 29],
        [28, 30],
        [29, 28],
        [30,  2],
        [30, 28],
        [30, 31],
        [31, 30]])
```

[36]:

```
data[0].y
```

[36]:

```
tensor([[-0.7700]])
```

In the following we will perform predictions based on the graph level. This

1. List item
2. List item

means we have one y-label for the whole graph, as shown on the left image below. The right image would be node-level-predictions.

1.4 Converting SMILES to RDKit molecules - Visualizing molecules

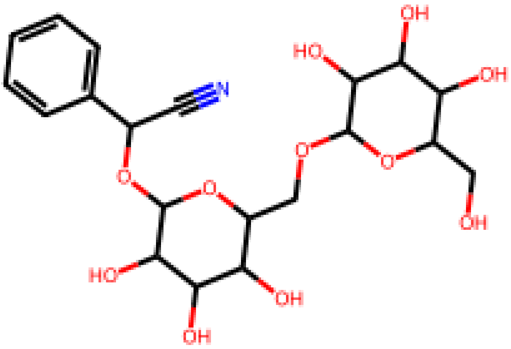
Next we want to have our SMILES molecules as graphs…

[37]:
```
data[0]["smiles"]
```

[37]:
```
'OCC3OC(OCC2OC(OC(C#N)c1ccccc1)C(O)C(O)C2O)C(O)C(O)C3O '
```

[38]:
```
from rdkit import Chem
from rdkit.Chem.Draw import IPythonConsole
molecule = Chem.MolFromSmiles(data[0]["smiles"])
molecule
```

[38]:



[39]:
```
type(molecule)
```

[39]:
```
rdkit.Chem.rdchem.Mol
```

- We can also obtain the features from this RDKit representation
- It tells us everything we need to know e.g. atom features (type, …), edges, …
- **In our case however, It's even easier as we have the information explicitly given already in the dataset**
- Otherwise we would calculate the node features from those atom properties

-> For datasets containing SMILES representations this would be the way to go

1.5 Implementing the Graph Neural Network

Building a Graph Neural Network works the same way as building a Convolutional

---

Neural Network, we simple add some layers.

The GCN simply extends torch.nn.Module. GCNConv expects: - in_channels = Size of each input sample. - out_channels = Size of each output sample.

We apply three convolutional layers, which means we learn the information about 3 neighbor hops. After that we apply a pooling layer to combine the information of the individual nodes, as we want to perform graph-level prediction.

Always keep in mind that different learning problems (node, edge or graph prediction) require different GNN architectures.

For example for node-level prediction you will often encounter masks. For graph-level predictions on the other hand you need to combine the node embeddings.

[40]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import add_self_loops

class PJzGAT(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(PJzGAT, self).__init__(aggr='add')

        self.in_channels = in_channels
        self.out_channels = out_channels

        self.W = nn.Parameter(torch.zeros(size=(in_channels, out_channels)))
        nn.init.xavier_uniform_(self.W.data, gain=1.414)

        self.a = nn.Parameter(torch.zeros(size=(2*out_channels, 1)))
        nn.init.xavier_uniform_(self.a.data, gain=1.414)

        self.leakyrelu = nn.LeakyReLU(0.2)

        self.mlp = nn.Sequential(
            nn.Linear(out_channels, out_channels),  # Adjust input channels for␣
 ↪MLP
            nn.ReLU(),
            nn.Linear(out_channels, out_channels),
            nn.ReLU()
        )

    def forward(self, x, edge_index):
        h = torch.matmul(x, self.W)  # Apply linear transformation

        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))

        # Compute attention coefficients
        a_input = torch.cat([h.repeat(1, x.size(0)).view(x.size(0) * x.size(0),␣
 ↪-1), h.repeat(x.size(0), 1)], dim=1).view(x.size(0), -1, 2 * self.
 ↪out_channels)
```

```
        e = self.leakyrelu(torch.matmul(a_input, self.a).squeeze())

        # Apply mask and activation function
        row , col = edge_index
        zero_vec = -9e15 * torch.ones_like(e)
        mask = torch.zeros_like(e)
        mask[edge_index[0], edge_index[1]] = 1
        attention = mask * e
        attention = F.leaky_relu(attention, negative_slope=0.2)
        attention = custom_exp(edge_index, attention)

        # Perform message passing with attention
        out = self.propagate(edge_index, x=h, attention=attention)

        # Optionally apply MLP after aggregation
        out = self.mlp(out)

        return out

    def message(self, x_j, attention):
        # Compute messages with attention coefficients
        attention = attention.view(-1)
        attention = attention[attention != 0].view(-1, 1)
        buffer = attention.view(-1, 1) * x_j
        return buffer
```

[41]:

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

import torch

def custom_exp(edge_index, attention_scores):
    # Extract the number of nodes (N) from the shape of the attention score␣
 ↪matrix
    N = attention_scores.size(0)

    # Initialize an empty tensor to store the attention coefficients
    attention_coefficients = torch.zeros_like(attention_scores)

    # Loop over each source node index (row index of edge_index)
    for i in range(edge_index.size(1)):
        src_idx, dst_idx = edge_index[:, i]
        numerator = torch.exp(attention_scores[src_idx, dst_idx])

        # get an array with attention scores of neighbours of src_idx
        neighbours = edge_index[1, edge_index[0] == src_idx]
        denominator = torch.sum(torch.exp(attention_scores[src_idx, neighbours]))

        attention_coefficients[src_idx, dst_idx] = numerator / denominator

    return attention_coefficients

# Example usage:
# edge_index = torch.tensor([[0, 1, 1, 2], [1, 0, 2, 1]])
# attention_scores = torch.randn(N, N)  # Replace with your attention score matrix
# attention_coefficients = custom_exp(edge_index, attention_scores)

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

        # GCN layers
        self.initial_conv = PJzGAT(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, 1)

    def forward(self, x, edge_index, batch_index):
        # First Conv layer

        hidden = nn.Linear(data.num_features, data.num_features)(x)
        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 (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()))
```

```
GCN(
  (initial_conv): PJzGAT(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:  21633
```

- We could also reduce the embeddings, but as we have large molecules we use 64
- The more layers we add, the more information we get about the graph
- For the regression problem we use a Linear layer as final output layer
- We try to use not too many parameters, as we only have ~1k samples

1.6 Training the GNN

[42]:

```
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 = 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}")
```

```
Starting training…
Epoch 0 | Train Loss 1.505754828453064
Epoch 100 | Train Loss 2.162950277328491
Epoch 200 | Train Loss 0.6054990887641907
Epoch 300 | Train Loss 0.9838536381721497
Epoch 400 | Train Loss 0.7840669751167297
Epoch 500 | Train Loss 0.33541715145111084
Epoch 600 | Train Loss 0.7452005743980408
Epoch 700 | Train Loss 0.9405662417411804
Epoch 800 | Train Loss 0.08519735932350159
Epoch 900 | Train Loss 0.697784960269928
Epoch 1000 | Train Loss 0.1554538458585739
Epoch 1100 | Train Loss 0.43753793835639954
Epoch 1200 | Train Loss 0.12322080880403519
Epoch 1300 | Train Loss 0.13829989731311798
Epoch 1400 | Train Loss 0.07302401214838028
Epoch 1500 | Train Loss 0.0774255022406578
Epoch 1600 | Train Loss 0.05965699627995491
```

1.6.1 Visualizing the Training loss

[1]:

```python
# 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(losses_float)
plt
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 3
      1 # Visualize learning (training loss)
      2 import seaborn as sns
----> 3 losses_float = [float(loss.cpu().detach().numpy()) for loss in losses]
      4 loss_indices = [i for i,l in enumerate(losses_float)]
      5 plt = sns.lineplot(losses_float)

NameError: name 'losses' is not defined
```

1.6.2 Getting a test prediction

[ ]:

```python
import pandas as pd

# Analyze the results for one batch
test_batch = next(iter(test_loader))
with torch.no_grad():
    test_batch.to(device)
    pred, embed = model(test_batch.x.float(), test_batch.edge_index, test_batch.
 ↪batch)
    df = pd.DataFrame()
    df["y_real"] = test_batch.y.tolist()
    df["y_pred"] = pred.tolist()
df["y_real"] = df["y_real"].apply(lambda row: row[0])
df["y_pred"] = df["y_pred"].apply(lambda row: row[0])
df
```

[ ]:

```
    y_real    y_pred
0    -1.92 -1.815238
1    -1.72 -2.060766
2    -3.24 -3.920323
3    -6.86 -5.725528
4    -0.85 -1.195607
..     …         …
59   -1.52 -3.488182
60   -2.38 -2.023098
```

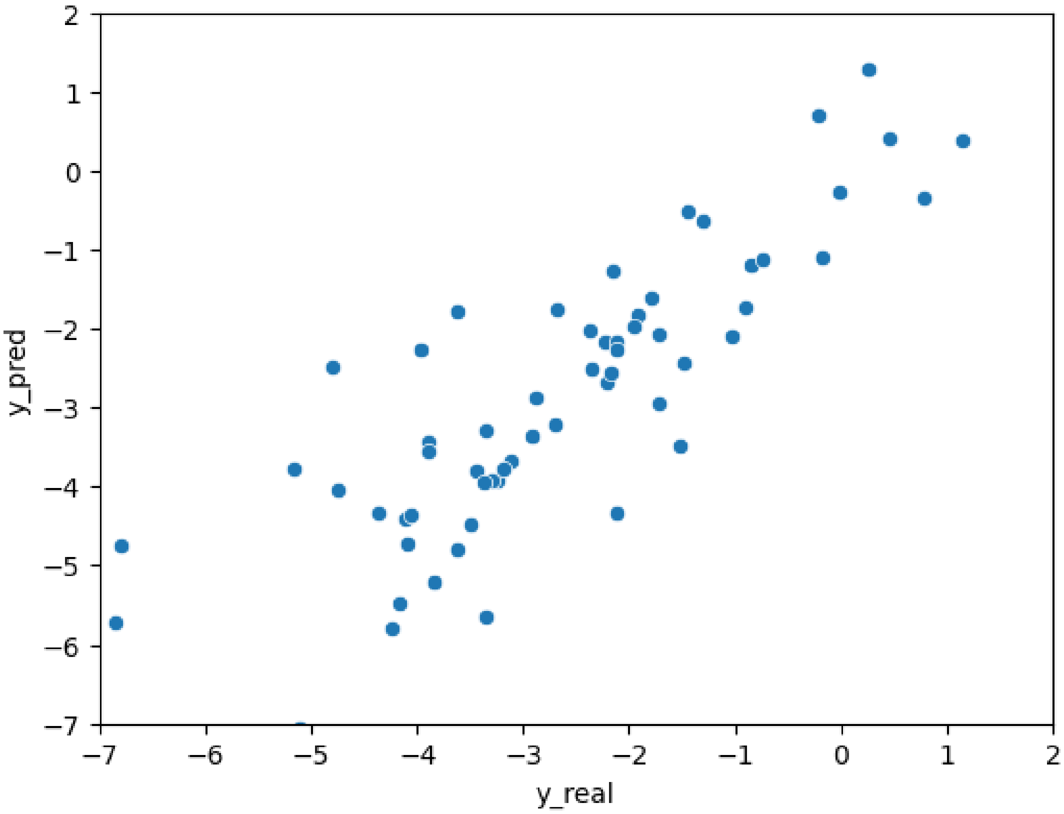
```
61   -3.84 -5.203280
62   -2.18 -2.544443
63   -0.22  0.720641

[64 rows x 2 columns]
```

[ ]:

```
plt = sns.scatterplot(data=df, x="y_real", y="y_pred")
plt.set(xlim=(-7, 2))
plt.set(ylim=(-7, 2))
plt
```

[ ]:

```
<Axes: xlabel='y_real', ylabel='y_pred'>
```



1.7 Improving the model / More to play around with

For example you can add: - 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 MP layers - Other hidden state sizes - Test metrics (test error) and Hyperparameter optimization - …

https://github.com/rusty1s/pytorch_geometric/tree/master/examples

[ ]:

```

```