https://drive.google.com/drive/folders/1_bGKFHFaYi1eEcCMs8Jv9LthpwimVA_8?usp=share_link

https://github.com/Alx-Ho/team64-project-cs598-dlh

In [44]:
```
!pip install rdkit
```

```
Requirement already satisfied: rdkit in /usr/local/lib/python3.10/dist-packa
ges (2023.9.6)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packa
ges (from rdkit) (1.25.2)
Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-pack
ages (from rdkit) (9.4.0)
```

In [45]:
```
!git clone https://github.com/Alx-Ho/team64-project-cs598-dlh.git
```

```
fatal: destination path 'team64-project-cs598-dlh' already exists and is not
an empty directory.
```

In [46]:
```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, random_split, Subset
import pandas as pd
from rdkit import Chem, DataStructs
from rdkit.Chem import AllChem
import numpy as np
import torch.optim as optim
from sklearn.metrics import accuracy_score
import random
from sklearn.decomposition import PCA
from datetime import datetime
```

Introduction

Predicting drug-drug interactions (DDIs) is challenging due to the complex nature of biological systems and the vast number of possible drug and food combinations. The human body's response to these interactions can vary significantly, making accurate predictions difficult. A major hurdle is the lack of comprehensive data on many drugs and food constituents, including their effects on the body and their metabolic pathways. Moreover, the mechanisms through which DDIs occur are diverse, adding another layer of complexity to prediction efforts. The constant development of new drugs and the updating of existing drug information necessitate continuous revisions of prediction models and databases.

The paper presents a computational framework named DeepDDI [1], designed to predict drug–drug interactions (DDIs) using only the structural information of the drug constituent pairs as inputs. DeepDDI employs a deep neural network (DNN) optimized to accurately predict 86 DDI types, generating predictions as human-readable sentences with a mean accuracy of 92.4% using the DrugBank gold standard DDI dataset. The input structural information is provided in the Simplified Molecular-Input Line-Entry System (SMILES) format, which describes the chemical compound's structure. This framework aims to enhance the understanding of DDIs and DFIs, providing critical information for drug prescription and dietary suggestions during medication.

Scope of Reproduciblity

The proposed model works on SMILES text representations for molecules, and is thus very memory efficient and accessible computational-wise for reproducibility. However, the original paper's repository https://bitbucket.org/kaistsystemsbiology/deepddi/src/master/ did not contain the training scripts for their deep learning algorithm nor the model definition. The ChemicalX library https://github.com/AstraZeneca/chemicalx contains an implemenation of the model as described in the original paper, but it is not maintained so it proved difficult to get the environment set up so that it would work in Google Colab's environment. Therefore, we implemented the model without the use of the ChemicalX library, and instead defined a new PyTorch model class using the ChemicalX implemenation as a reference.

Training to the full extent of the original paper is difficult, however. The full dataset contains 192,303 DDI samples and was trained for up to 100 epochs. Due to compute credit limits with Google Colab, we limited training to 5 epochs and the dataset to 8,192 samples for training, 2,048 for validation, and 2,048 for hold-out testing.

Methodology

In [ ]:
```
!python --version
```

```
Python 3.10.12
```

In [ ]:
```
!pip show torch
```

```
Name: torch
Version: 2.2.1+cu121
Summary: Tensors and Dynamic neural networks in Python with strong GPU accel
eration
Home-page: https://pytorch.org/
Author: PyTorch Team
Author-email: packages@pytorch.org
License: BSD-3
Location: /usr/local/lib/python3.10/dist-packages
Requires: filelock, fsspec, jinja2, networkx, nvidia-cublas-cu12, nvidia-cud
a-cupti-cu12, nvidia-cuda-nvrtc-cu12, nvidia-cuda-runtime-cu12, nvidia-cudnn
-cu12, nvidia-cufft-cu12, nvidia-curand-cu12, nvidia-cusolver-cu12, nvidia-c
usparse-cu12, nvidia-nccl-cu12, nvidia-nvtx-cu12, sympy, triton, typing-exte
nsions
Required-by: fastai, torchaudio, torchdata, torchtext, torchvision
```

```
In [ ]: !pip show pandas
```

```
Name: pandas
Version: 2.0.3
Summary: Powerful data structures for data analysis, time series, and statis
tics
Home-page:
Author:
Author-email: The Pandas Development Team <pandas-dev@python.org>
License: BSD 3-Clause License

        Copyright (c) 2008-2011, AQR Capital Management, LLC, Lambda Foundr
y, Inc. and PyData Development Team
        All rights reserved.

        Copyright (c) 2011-2023, Open source contributors.

        Redistribution and use in source and binary forms, with or without
        modification, are permitted provided that the following conditions a
re met:

        * Redistributions of source code must retain the above copyright not
ice, this
          list of conditions and the following disclaimer.

        * Redistributions in binary form must reproduce the above copyright
notice,
          this list of conditions and the following disclaimer in the docume
ntation
          and/or other materials provided with the distribution.

        * Neither the name of the copyright holder nor the names of its
          contributors may be used to endorse or promote products derived fr
om
          this software without specific prior written permission.

        THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS"
        AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED T
O, THE
        IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR P
URPOSE ARE
        DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS B
E LIABLE
        FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQU
ENTIAL
        DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GO
ODS OR
        SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) H
OWEVER
        CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT L
IABILITY,
        OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT O
```

```
F THE USE
        OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Location: /usr/local/lib/python3.10/dist-packages
Requires: numpy, python-dateutil, pytz, tzdata
Required-by: altair, arviz, bigframes, bokeh, bqplot, cmdstanpy, cudf-cu12,
cufflinks, datascience, db-dtypes, dopamine-rl, fastai, geemap, geopandas, g
oogle-colab, gspread-dataframe, holoviews, ibis-framework, mizani, mlxtend,
pandas-datareader, pandas-gbq, panel, plotnine, prophet, pymc, seaborn, skle
arn-pandas, statsmodels, vega-datasets, xarray, yfinance
```

In [ ]:
```
!pip show rdkit
```

```
Name: rdkit
Version: 2023.9.6
Summary: A collection of chemoinformatics and machine-learning software writ
ten in C++ and Python
Home-page: https://github.com/kuelumbus/rdkit-pypi
Author: Christopher Kuenneth
Author-email: chris@kuenneth.dev
License: BSD-3-Clause
Location: /usr/local/lib/python3.10/dist-packages
Requires: numpy, Pillow
Required-by:
```

In [ ]:
```
!pip show numpy
```

```
Name: numpy
Version: 1.25.2
Summary: Fundamental package for array computing in Python
Home-page: https://www.numpy.org
Author: Travis E. Oliphant et al.
Author-email:
License: BSD-3-Clause
Location: /usr/local/lib/python3.10/dist-packages
Requires:
Required-by: albumentations, altair, arviz, astropy, autograd, blis, bokeh,
bqplot, chex, cmdstanpy, contourpy, cudf-cu12, cufflinks, cupy-cuda12x, cvxp
y, datascience, db-dtypes, dopamine-rl, ecos, flax, folium, geemap, gensim,
gym, h5py, holoviews, hyperopt, ibis-framework, imageio, imbalanced-learn, i
mgaug, jax, jaxlib, librosa, lightgbm, matplotlib, matplotlib-venn, missingn
o, mizani, ml-dtypes, mlxtend, moviepy, music21, nibabel, numba, numexpr, op
encv-contrib-python, opencv-python, opencv-python-headless, opt-einsum, opta
x, orbax-checkpoint, osqp, pandas, pandas-gbq, pandas-stubs, patsy, plotnin
e, prophet, pyarrow, pycocotools, pyerfa, pymc, pytensor, python-louvain, Py
Wavelets, qdldl, qudida, rdkit, rmm-cu12, scikit-image, scikit-learn, scipy,
scs, seaborn, shapely, sklearn-pandas, soxr, spacy, stanio, statsmodels, tab
les, tensorboard, tensorflow, tensorflow-datasets, tensorflow-hub, tensorflo
w-probability, tensorstore, thinc, tifffile, torchtext, torchvision, transfo
rmers, wordcloud, xarray, xarray-einstats, xgboost, yellowbrick, yfinance
```

In [ ]:
```
!pip show scikit-learn
```

```
Name: scikit-learn
Version: 1.2.2
Summary: A set of python modules for machine learning and data mining
Home-page: http://scikit-learn.org
Author: 
Author-email: 
License: new BSD
Location: /usr/local/lib/python3.10/dist-packages
Requires: joblib, numpy, scipy, threadpoolctl
Required-by: bigframes, fastai, imbalanced-learn, librosa, mlxtend, qudida, sklearn-pandas, yellowbrick
```

Data

We are using the datasets listed in the Supporting Information section for the PNAS publication link: https://www.pnas.org/doi/suppl/10.1073/pnas.1803294115

The two that we need for this project are Datasets S01 and S02, which are also under `data/supplement_excel` in the GitHub project. S01 describes the different Drug-Drug Interaction (DDI) types and the sentence structure associated with each. S02 contains the base data which was used in the original paper, however it is missing some crucial information necessary for model training including the SMILES representation and the DDI type. This notebook will primarily process S02 to create a ready-to-use dataset for model training.

The dataset provided for this project came in the form of Drug Bank IDs for each pair of drugs that have an interaction specified as a DDI type. For example,

`The metabolism of DB01621 can be decreased when combined with DB00477.,training`

From the text and format, we can also determine that this is DDI type 6 based on a table provided with the original paper. It also indicates that this sample with drug pair DB01621 (drug B) and DB00477 (drug A) was used in the training set.

Preprocessing (in GitHub notebook `preprocess_ds.ipynb`, conda environment provided)

First, Dataset S02 is manually converted to CSV format and columns renamed

'Sentences describing the reported drug-drug interactions' --> 'gt'

'Data type used to optimize the DNN architecture' --> 'set'

The resulting file is in 'data/ds_s2.csv'

We can see from reading through Dataset S01 that there are three types of sentence structures for Dataset S02 regarding the order of the drugs mentioned:

1. Drug A ... Drug B ... (AB)
2. Drug B ... Drug A ... (BA)
3. Drug B ... Drug B ... Drug A (BBA)

The DDI types were manually sorted into these three types and saved in `ddi_types_ab.csv`, `ddi_types_ba.csv`, and `ddi_types_bba.csv` where the column `type` is the DDI type listed in Dataset S01, and the column `structure` is the original sentence structure listed in Dataset S01. From these structures, we can create regular expression patterns to extract the Drug Bank ID from Dataset S02.

Next, we extract the Drug Bank IDs for Drug A and Drug B from the ground truth sentences in Dataset S02 using the regular expression structures generated earlier for each of the three sentence structure types.

Now we need to get the SMILES representation for each drug, which we can get from go.drugbank.com using the extracted DrugBank ID.

Finally, we can merge the extracted Drug Bank IDs with the retrieved SMILES representations to create the full dataset.

For the full implementation and details, see the GitHub project and preprocessing notebook.

In [47]:
```
torch.manual_seed(0)
random.seed(0)
np.random.seed(0)
```

The model's implementation is based on the ChemicalX implementation [2] and can be seen below. It takes in a feature vector (the size of which we can specify with `drug_feature_length`), and outputs the probabilities for each class (the number of which we specify with `num_categories`). We can also change the complexity of the model by setting `num_hidden_layers` and `hidden_dimension`. The model itself is a simple feed forward network with ReLU activations and batch normalization. A pretrained model with hyperparameters 2 layers and 128 hidden dimension can be found in the GitHub project as `2_layer_128_dim_model.pth`.

In [48]:
```
class DeepDDI(nn.Module):
    """An implementation of the DeepDDI model from [ryu2018]_.

    .. seealso:: This model was suggested in https://github.com/AstraZeneca/

    .. [ryu2018] Ryu, J. Y., *et al.* (2018). `Deep learning improves predic
       of drug–drug and drug–food interactions <https://doi.org/10.1073/pnas
       *Proceedings of the National Academy of Sciences*, 115(18), E4304–E43
    """
    def __init__(
        self,
        drug_feature_length: int,
        num_hidden_layers: int,
        hidden_dimension: int,
        num_categories: int,
    ):
        """Instantiate the DeepDDI model.

        :param drug_feature_length: The length of the input feature vector f
        :param num_hidden_layers: The number of hidden layers.
        :param hidden_dimension: The size of the hidden linear layer dimensi
        :param num_categories: The number of output categories that the mode
        """
        super().__init__()
        assert num_hidden_layers > 1
        layers = [
            nn.Linear(drug_feature_length * 2, hidden_dimension),
            nn.ReLU(),
            nn.BatchNorm1d(num_features=hidden_dimension, affine=True, momen
            nn.ReLU(),
        ]
        for _ in range(num_hidden_layers - 1):
            layers.extend(
                [
                    nn.Linear(hidden_dimension, hidden_dimension),
                    nn.ReLU(),
                    nn.BatchNorm1d(num_features=hidden_dimension, affine=Tru
                    nn.ReLU(),
                ]
            )
        layers.extend([nn.Linear(hidden_dimension, num_categories), nn.Sigmo
        self.final = nn.Sequential(*layers)

    def _combine_sides(self, left: torch.FloatTensor, right: torch.FloatTens
        return torch.cat([left, right], dim=1)

    def forward(
        self,
        drug_features_a: torch.FloatTensor,
        drug_features_b: torch.FloatTensor,
    ) -> torch.FloatTensor:
        """Run a forward pass of the DeepDDI model.

        :param drug_features_a: feature vector for drug A.
        :param drug_features_b: feature vector for drug B.
        :returns: A column vector of predicted interaction scores.
        """
        hidden = self._combine_sides(drug_features_a, drug_features_b)
        return self.final(hidden)


class DDIDataset(Dataset):
    def __init__(self, csv_file, feature_size, num_categories=86):
        self.data = pd.read_csv(csv_file)
        self.fpgen = AllChem.GetRDKitFPGenerator()
        self.feature_size = feature_size

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        row = self.data.iloc[idx]
        drug_a_smiles = row['drug_a_smiles']
        drug_b_smiles = row['drug_b_smiles']
        drug_a_id = row['drug_a']
        drug_b_id = row['drug_b']
        interaction = row['interaction']

        interaction_tensor = torch.tensor(interaction, dtype=torch.int64)
        interaction_tensor -= 1 # set indexing to start at 0
        interaction_one_hot_label = nn.functional.one_hot(interaction_tensor

        drug_a_fp = self.get_fingerprint(drug_a_smiles)
        drug_b_fp = self.get_fingerprint(drug_b_smiles)

        drug_a_tensor = torch.tensor(drug_a_fp, dtype=torch.float32)
        drug_b_tensor = torch.tensor(drug_b_fp, dtype=torch.float32)
        label_tensor = interaction_one_hot_label.float()

        return drug_a_tensor, drug_b_tensor, label_tensor

    def get_fingerprint(self, smiles):
        ms = Chem.MolFromSmiles(smiles)
        if ms is not None:
            fp = self.fpgen.GetFingerprint(ms)
            array = np.zeros((self.feature_size,), dtype=np.float32)
            DataStructs.ConvertToNumpyArray(fp, array)
            return array
        else:
            return np.zeros((self.feature_size,), dtype=np.float32)


def train_pca_model(ddi_dataset, n_components):
    # Collect all drug fingerprints from the dataset
    fingerprints = []
    for i in range(len(ddi_dataset)):
        drug_a_tensor, drug_b_tensor, _ = ddi_dataset[i]
        fingerprints.append(drug_a_tensor.numpy())
        fingerprints.append(drug_b_tensor.numpy())

    # Convert the list of fingerprints to a numpy array
    fingerprints_array = np.array(fingerprints)

    # Create a PCA model
    pca_model = PCA(n_components=n_components)
    # Fit the PCA model to the fingerprints
    pca_model.fit(fingerprints_array)

    return pca_model


class DDIDatasetPCA(Dataset):
    def __init__(self, ddi_dataset, pca_model):
        self.ddi_dataset = ddi_dataset
        self.pca_model = pca_model

    def __len__(self):
        return len(self.ddi_dataset)

    def __getitem__(self, idx):
        drug_a_tensor, drug_b_tensor, label_tensor = self.ddi_dataset[idx]

        # Apply PCA dimensionality reduction to drug fingerprints
        drug_a_reduced = self.apply_pca(drug_a_tensor)
        drug_b_reduced = self.apply_pca(drug_b_tensor)

        return drug_a_reduced, drug_b_reduced, label_tensor

    def apply_pca(self, tensor):
        # Convert tensor to numpy array
        array = tensor.numpy()

        # Reshape the array to 2D if necessary
        if array.ndim == 1:
            array = array.reshape(1, -1)

        # Apply PCA transformation
        reduced_array = self.pca_model.transform(array)

        # Convert the reduced array back to a tensor
        reduced_tensor = torch.from_numpy(reduced_array).float()

        return reduced_tensor.squeeze()
```

Training

We follow the original paper's hyperparameters for the learning rate (0.0001), batch size (256), and optimizer (Adam). The paper claimed that the 9 layer model with 2048 hidden dimension size gave the best performance on a hold-out test set. Therefore, we tested a similar configuration (8 layers, 2048 hidden dimension size) along with other models of varying complexity in the range of [2, 4, 8] layers and [128, 2048] hidden dimension size.

Training a model takes on the order of 4-15 minutes for 5 epochs with 8,192 training samples, and in total we had 15 models. Therefore, the total GPU time with a single T4 was approximately 3-4 hours.

In [49]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f'Using {device}')

# Set hyperparameters
drug_feature_length = 2048
num_hidden_layers = 2 # 2 4 8
hidden_dimension = 2**7 # 2**7 2**11
num_categories = 86
learning_rate = 0.0001

# Initialize the model
model = DeepDDI(
    drug_feature_length=drug_feature_length,
    num_hidden_layers=num_hidden_layers,
    hidden_dimension=hidden_dimension,
    num_categories=num_categories,
).to(device)

optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

```
Using cuda
```

In [50]:
```
# Set hyperparameters
num_epochs = 5
batch_size = 2**8

# Define the loss function
criterion = nn.CrossEntropyLoss()

# Prepare the dataset
train_dataset = DDIDataset("/content/team64-project-cs598-dlh/data/train_ds.
                     drug_feature_length,
                     num_categories,
                     )

val_dataset = DDIDataset("/content/team64-project-cs598-dlh/data/val_ds.csv"
                     drug_feature_length,
                     num_categories,
                     )

train_subset_size = 2**13 # len(train_dataset) # reduce for quicker training
train_subset_indices = random.sample(range(len(train_dataset)), train_subset
train_subset_dataset = Subset(train_dataset, train_subset_indices)

val_subset_size = 2**12 # len(val_dataset) # reduce for quicker training
val_subset_indices = random.sample(range(len(val_dataset)), val_subset_size)
val_subset_dataset = Subset(val_dataset, val_subset_indices)

# Split the dataset into train, validation, and test sets
val_size = int(0.5 * len(val_subset_dataset))
test_size = len(val_subset_dataset) - val_size
```

```
val_dataset, test_dataset = random_split(val_subset_dataset, [val_size, test

train_dataloader = DataLoader(train_subset_dataset, batch_size=batch_size, s
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=Fals
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=Fa
```

```
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:558:
UserWarning: This DataLoader will create 8 worker processes in total. Our su
ggested max number of worker in current system is 2, which is smaller than w
hat this DataLoader is going to create. Please be aware that excessive worke
r creation might get DataLoader running slow or even freeze, lower the worke
r number to avoid potential slowness/freeze if necessary.
  warnings.warn(_create_warning_msg(
```

In the original paper, the feature representation is not the direct molecular fingerprint of the drug, but rather a dimensionality-reduced version created by using Principal Component Analysis on the training set. Here, we use the direct molecular fingerprint, which is a binary encoding of length 2048. For the draft, we reduce the complexity of the model compared to the paper (8 layers --> 3 layers), and we also limit the training by reducing the dataset size as well as the number of training epochs (100 to 5). Furthermore, we only predict a single DDI type, so the output is a single probability of interaction, whereas the original paper defines an output of 86 probabilities for each tested DDI type.

```
In [ ]: # Training loop
start = datetime.now()
print(f'Starting at {start}')

for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(train_dataloader):
        inputs_left, inputs_right, labels = data
        inputs_left, inputs_right = inputs_left.to(device), inputs_right.to(

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs_left, inputs_right)
        inputs_left, inputs_right = inputs_left.cpu(), inputs_right.cpu()
        labels = labels.to(device)
        loss = criterion(outputs, labels)
        labels = labels.cpu()
        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        # Print the loss for each batch
```

```
            print(f"Epoch [{epoch+1}/{num_epochs}], Batch [{i+1}/{len(train_data

        # Print the average loss for the epoch
        epoch_loss = running_loss / len(train_dataloader)
        print(f"Epoch [{epoch+1}/{num_epochs}], Average Training Loss: {epoch_lo

        # Validation
        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for i, data in enumerate(val_dataloader):
                inputs_left, inputs_right, labels = data
                inputs_left, inputs_right = inputs_left.to(device), inputs_right
                outputs = model(inputs_left, inputs_right)
                inputs_left, inputs_right = inputs_left.cpu(), inputs_right.cpu(
                labels = labels.to(device)
                loss = criterion(outputs.squeeze(), labels)
                labels = labels.cpu()
                val_loss += loss.item()
                print(f"Epoch [{epoch+1}/{num_epochs}], Batch [{i+1}/{len(val_da


        val_loss /= len(val_dataloader)
        print(f"Epoch [{epoch+1}/{num_epochs}], Average Validation Loss: {val_lo
        model.train()
end = datetime.now()
print(f"Training finished! Training took {end - start} ({start} to {end})")
```

```
Starting at 2024-05-08 04:20:40.286331
```

```
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.for
k() was called. os.fork() is incompatible with multithreaded code, and JAX i
s multithreaded, so this will likely lead to a deadlock.
  self.pid = os.fork()
```

```
Epoch [1/5], Batch [1/32], Training Loss: 4.4751
Epoch [1/5], Batch [2/32], Training Loss: 4.4654
Epoch [1/5], Batch [3/32], Training Loss: 4.4528
Epoch [1/5], Batch [4/32], Training Loss: 4.4604
Epoch [1/5], Batch [5/32], Training Loss: 4.4515
Epoch [1/5], Batch [6/32], Training Loss: 4.4320
Epoch [1/5], Batch [7/32], Training Loss: 4.4275
Epoch [1/5], Batch [8/32], Training Loss: 4.4342
Epoch [1/5], Batch [9/32], Training Loss: 4.4186
Epoch [1/5], Batch [10/32], Training Loss: 4.4068
Epoch [1/5], Batch [11/32], Training Loss: 4.4039
```

```
[04:21:07] SMILES Parse Error: syntax error while parsing: OC1=CC=CC(=C1)C-1
=C2\CCC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C
1)C1=CC(O)=CC=C1)\C1=CC(O)=CC=C1
[04:21:07] SMILES Parse Error: Failed parsing SMILES 'OC1=CC=CC(=C1)C-1=C2\C
CC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=
CC(O)=CC=C1)\C1=CC(O)=CC=C1' for input: 'OC1=CC=CC(=C1)C-1=C2\CCC(=N2)\C(=C
2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=CC(O)=CC=C1)
\C1=CC(O)=CC=C1'
```

```
Epoch [1/5], Batch [12/32], Training Loss: 4.3998
Epoch [1/5], Batch [13/32], Training Loss: 4.3950
Epoch [1/5], Batch [14/32], Training Loss: 4.3940
Epoch [1/5], Batch [15/32], Training Loss: 4.3756
Epoch [1/5], Batch [16/32], Training Loss: 4.3859
Epoch [1/5], Batch [17/32], Training Loss: 4.3730
Epoch [1/5], Batch [18/32], Training Loss: 4.3639
Epoch [1/5], Batch [19/32], Training Loss: 4.3705
Epoch [1/5], Batch [20/32], Training Loss: 4.3596
Epoch [1/5], Batch [21/32], Training Loss: 4.3578
Epoch [1/5], Batch [22/32], Training Loss: 4.3518
Epoch [1/5], Batch [23/32], Training Loss: 4.3525
Epoch [1/5], Batch [24/32], Training Loss: 4.3377
Epoch [1/5], Batch [25/32], Training Loss: 4.3423
Epoch [1/5], Batch [26/32], Training Loss: 4.3456
Epoch [1/5], Batch [27/32], Training Loss: 4.3381
Epoch [1/5], Batch [28/32], Training Loss: 4.3246
Epoch [1/5], Batch [29/32], Training Loss: 4.3186
Epoch [1/5], Batch [30/32], Training Loss: 4.3147
```

```
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.for
k() was called. os.fork() is incompatible with multithreaded code, and JAX i
s multithreaded, so this will likely lead to a deadlock.
  self.pid = os.fork()
```

```
Epoch [1/5], Batch [31/32], Training Loss: 4.3238
Epoch [1/5], Batch [32/32], Training Loss: 4.3038
Epoch [1/5], Average Training Loss: 4.3830
Epoch [1/5], Batch [1/8], Validation Loss: 4.2725
Epoch [1/5], Batch [2/8], Validation Loss: 4.2327
Epoch [1/5], Batch [3/8], Validation Loss: 4.2424
Epoch [1/5], Batch [4/8], Validation Loss: 4.2479
Epoch [1/5], Batch [5/8], Validation Loss: 4.2401
Epoch [1/5], Batch [6/8], Validation Loss: 4.2531
Epoch [1/5], Batch [7/8], Validation Loss: 4.2493
Epoch [1/5], Batch [8/8], Validation Loss: 4.2766
Epoch [1/5], Average Validation Loss: 4.2518
Epoch [2/5], Batch [1/32], Training Loss: 4.2816
Epoch [2/5], Batch [2/32], Training Loss: 4.2791
Epoch [2/5], Batch [3/32], Training Loss: 4.2910
Epoch [2/5], Batch [4/32], Training Loss: 4.2638
Epoch [2/5], Batch [5/32], Training Loss: 4.2702
Epoch [2/5], Batch [6/32], Training Loss: 4.2668
Epoch [2/5], Batch [7/32], Training Loss: 4.2542
Epoch [2/5], Batch [8/32], Training Loss: 4.2572
Epoch [2/5], Batch [9/32], Training Loss: 4.2422
Epoch [2/5], Batch [10/32], Training Loss: 4.2514
Epoch [2/5], Batch [11/32], Training Loss: 4.2552
Epoch [2/5], Batch [12/32], Training Loss: 4.2475
Epoch [2/5], Batch [13/32], Training Loss: 4.2429
Epoch [2/5], Batch [14/32], Training Loss: 4.2481
Epoch [2/5], Batch [15/32], Training Loss: 4.2374
Epoch [2/5], Batch [16/32], Training Loss: 4.2342
Epoch [2/5], Batch [17/32], Training Loss: 4.2440
Epoch [2/5], Batch [18/32], Training Loss: 4.2210
Epoch [2/5], Batch [19/32], Training Loss: 4.2163
Epoch [2/5], Batch [20/32], Training Loss: 4.2366
Epoch [2/5], Batch [21/32], Training Loss: 4.2270
Epoch [2/5], Batch [22/32], Training Loss: 4.2392
Epoch [2/5], Batch [23/32], Training Loss: 4.2046
Epoch [2/5], Batch [24/32], Training Loss: 4.2105
```

```
[04:22:36] SMILES Parse Error: syntax error while parsing: OC1=CC=CC(=C1)C-1
=C2\CCC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C
1)C1=CC(O)=CC=C1)\C1=CC(O)=CC=C1
[04:22:36] SMILES Parse Error: Failed parsing SMILES 'OC1=CC=CC(=C1)C-1=C2\C
CC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=
CC(O)=CC=C1)\C1=CC(O)=CC=C1' for input: 'OC1=CC=CC(=C1)C-1=C2\CCC(=N2)\C(=C
2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=CC(O)=CC=C1)
\C1=CC(O)=CC=C1'
```

```
Epoch [2/5], Batch [25/32], Training Loss: 4.2091
Epoch [2/5], Batch [26/32], Training Loss: 4.2177
Epoch [2/5], Batch [27/32], Training Loss: 4.2069
Epoch [2/5], Batch [28/32], Training Loss: 4.2030
Epoch [2/5], Batch [29/32], Training Loss: 4.1939
Epoch [2/5], Batch [30/32], Training Loss: 4.2089
Epoch [2/5], Batch [31/32], Training Loss: 4.1869
Epoch [2/5], Batch [32/32], Training Loss: 4.2043
Epoch [2/5], Average Training Loss: 4.2360
Epoch [2/5], Batch [1/8], Validation Loss: 4.2003
Epoch [2/5], Batch [2/8], Validation Loss: 4.1800
Epoch [2/5], Batch [3/8], Validation Loss: 4.1808
Epoch [2/5], Batch [4/8], Validation Loss: 4.1857
Epoch [2/5], Batch [5/8], Validation Loss: 4.1910
Epoch [2/5], Batch [6/8], Validation Loss: 4.2082
Epoch [2/5], Batch [7/8], Validation Loss: 4.1973
Epoch [2/5], Batch [8/8], Validation Loss: 4.2151
Epoch [2/5], Average Validation Loss: 4.1948
Epoch [3/5], Batch [1/32], Training Loss: 4.1791
Epoch [3/5], Batch [2/32], Training Loss: 4.1797
Epoch [3/5], Batch [3/32], Training Loss: 4.1791
Epoch [3/5], Batch [4/32], Training Loss: 4.1719
Epoch [3/5], Batch [5/32], Training Loss: 4.1752
Epoch [3/5], Batch [6/32], Training Loss: 4.1630
Epoch [3/5], Batch [7/32], Training Loss: 4.1698
Epoch [3/5], Batch [8/32], Training Loss: 4.1633
Epoch [3/5], Batch [9/32], Training Loss: 4.1648
Epoch [3/5], Batch [10/32], Training Loss: 4.1522
Epoch [3/5], Batch [11/32], Training Loss: 4.1493
Epoch [3/5], Batch [12/32], Training Loss: 4.1510
Epoch [3/5], Batch [13/32], Training Loss: 4.1453
Epoch [3/5], Batch [14/32], Training Loss: 4.1453
Epoch [3/5], Batch [15/32], Training Loss: 4.1500
Epoch [3/5], Batch [16/32], Training Loss: 4.1436
Epoch [3/5], Batch [17/32], Training Loss: 4.1381
Epoch [3/5], Batch [18/32], Training Loss: 4.1265
Epoch [3/5], Batch [19/32], Training Loss: 4.1280
Epoch [3/5], Batch [20/32], Training Loss: 4.1339
Epoch [3/5], Batch [21/32], Training Loss: 4.1271
Epoch [3/5], Batch [22/32], Training Loss: 4.1300
Epoch [3/5], Batch [23/32], Training Loss: 4.1363
Epoch [3/5], Batch [24/32], Training Loss: 4.1169
```

```
[04:23:28] SMILES Parse Error: syntax error while parsing: OC1=CC=CC(=C1)C-1
=C2\CCC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C
1)C1=CC(O)=CC=C1)\C1=CC(O)=CC=C1
[04:23:28] SMILES Parse Error: Failed parsing SMILES 'OC1=CC=CC(=C1)C-1=C2\C
CC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=
CC(O)=CC=C1)\C1=CC(O)=CC=C1' for input: 'OC1=CC=CC(=C1)C-1=C2\CCC(=N2)\C(=C
2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=CC(O)=CC=C1)
\C1=CC(O)=CC=C1'
```

```
Epoch [3/5], Batch [25/32], Training Loss: 4.1194
Epoch [3/5], Batch [26/32], Training Loss: 4.1301
Epoch [3/5], Batch [27/32], Training Loss: 4.1124
Epoch [3/5], Batch [28/32], Training Loss: 4.1150
Epoch [3/5], Batch [29/32], Training Loss: 4.1068
Epoch [3/5], Batch [30/32], Training Loss: 4.1213
Epoch [3/5], Batch [31/32], Training Loss: 4.1005
Epoch [3/5], Batch [32/32], Training Loss: 4.1138
Epoch [3/5], Average Training Loss: 4.1418
Epoch [3/5], Batch [1/8], Validation Loss: 4.1319
Epoch [3/5], Batch [2/8], Validation Loss: 4.1121
Epoch [3/5], Batch [3/8], Validation Loss: 4.1179
Epoch [3/5], Batch [4/8], Validation Loss: 4.1282
Epoch [3/5], Batch [5/8], Validation Loss: 4.1373
Epoch [3/5], Batch [6/8], Validation Loss: 4.1426
Epoch [3/5], Batch [7/8], Validation Loss: 4.1338
Epoch [3/5], Batch [8/8], Validation Loss: 4.1458
Epoch [3/5], Average Validation Loss: 4.1312
Epoch [4/5], Batch [1/32], Training Loss: 4.0906
Epoch [4/5], Batch [2/32], Training Loss: 4.0991
Epoch [4/5], Batch [3/32], Training Loss: 4.0872
Epoch [4/5], Batch [4/32], Training Loss: 4.0890
Epoch [4/5], Batch [5/32], Training Loss: 4.0803
Epoch [4/5], Batch [6/32], Training Loss: 4.0820
Epoch [4/5], Batch [7/32], Training Loss: 4.0869
Epoch [4/5], Batch [8/32], Training Loss: 4.0867
```

```
[04:23:55] SMILES Parse Error: syntax error while parsing: OC1=CC=CC(=C1)C-1
=C2\CCC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C
1)C1=CC(O)=CC=C1)\C1=CC(O)=CC=C1
[04:23:55] SMILES Parse Error: Failed parsing SMILES 'OC1=CC=CC(=C1)C-1=C2\C
CC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=
CC(O)=CC=C1)\C1=CC(O)=CC=C1' for input: 'OC1=CC=CC(=C1)C-1=C2\CCC(=N2)\C(=C
2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=CC(O)=CC=C1)
\C1=CC(O)=CC=C1'
```

```
Epoch [4/5], Batch [9/32], Training Loss: 4.0838
Epoch [4/5], Batch [10/32], Training Loss: 4.0634
Epoch [4/5], Batch [11/32], Training Loss: 4.0759
Epoch [4/5], Batch [12/32], Training Loss: 4.0751
Epoch [4/5], Batch [13/32], Training Loss: 4.0553
Epoch [4/5], Batch [14/32], Training Loss: 4.0783
Epoch [4/5], Batch [15/32], Training Loss: 4.0758
Epoch [4/5], Batch [16/32], Training Loss: 4.0550
Epoch [4/5], Batch [17/32], Training Loss: 4.0706
Epoch [4/5], Batch [18/32], Training Loss: 4.0763
Epoch [4/5], Batch [19/32], Training Loss: 4.0679
Epoch [4/5], Batch [20/32], Training Loss: 4.0623
Epoch [4/5], Batch [21/32], Training Loss: 4.0616
Epoch [4/5], Batch [22/32], Training Loss: 4.0592
Epoch [4/5], Batch [23/32], Training Loss: 4.0590
Epoch [4/5], Batch [24/32], Training Loss: 4.0540
Epoch [4/5], Batch [25/32], Training Loss: 4.0594
Epoch [4/5], Batch [26/32], Training Loss: 4.0562
Epoch [4/5], Batch [27/32], Training Loss: 4.0517
Epoch [4/5], Batch [28/32], Training Loss: 4.0655
Epoch [4/5], Batch [29/32], Training Loss: 4.0516
Epoch [4/5], Batch [30/32], Training Loss: 4.0540
Epoch [4/5], Batch [31/32], Training Loss: 4.0591
Epoch [4/5], Batch [32/32], Training Loss: 4.0470
Epoch [4/5], Average Training Loss: 4.0694
Epoch [4/5], Batch [1/8], Validation Loss: 4.0958
Epoch [4/5], Batch [2/8], Validation Loss: 4.0708
Epoch [4/5], Batch [3/8], Validation Loss: 4.0779
Epoch [4/5], Batch [4/8], Validation Loss: 4.0853
Epoch [4/5], Batch [5/8], Validation Loss: 4.0948
Epoch [4/5], Batch [6/8], Validation Loss: 4.0999
Epoch [4/5], Batch [7/8], Validation Loss: 4.0935
Epoch [4/5], Batch [8/8], Validation Loss: 4.1027
Epoch [4/5], Average Validation Loss: 4.0901
```

```
[04:24:48] SMILES Parse Error: syntax error while parsing: OC1=CC=CC(=C1)C-1
=C2\CCC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C
1)C1=CC(O)=CC=C1)\C1=CC(O)=CC=C1
[04:24:48] SMILES Parse Error: Failed parsing SMILES 'OC1=CC=CC(=C1)C-1=C2\C
CC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=
CC(O)=CC=C1)\C1=CC(O)=CC=C1' for input: 'OC1=CC=CC(=C1)C-1=C2\CCC(=N2)\C(=C
2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=CC(O)=CC=C1)
\C1=CC(O)=CC=C1'
```

```
Epoch [5/5], Batch [1/32], Training Loss: 4.0335
Epoch [5/5], Batch [2/32], Training Loss: 4.0448
Epoch [5/5], Batch [3/32], Training Loss: 4.0217
Epoch [5/5], Batch [4/32], Training Loss: 4.0177
Epoch [5/5], Batch [5/32], Training Loss: 4.0285
Epoch [5/5], Batch [6/32], Training Loss: 4.0271
Epoch [5/5], Batch [7/32], Training Loss: 4.0051
Epoch [5/5], Batch [8/32], Training Loss: 4.0164
Epoch [5/5], Batch [9/32], Training Loss: 4.0210
Epoch [5/5], Batch [10/32], Training Loss: 4.0184
Epoch [5/5], Batch [11/32], Training Loss: 4.0127
Epoch [5/5], Batch [12/32], Training Loss: 4.0220
Epoch [5/5], Batch [13/32], Training Loss: 4.0201
Epoch [5/5], Batch [14/32], Training Loss: 4.0129
Epoch [5/5], Batch [15/32], Training Loss: 4.0139
Epoch [5/5], Batch [16/32], Training Loss: 4.0068
Epoch [5/5], Batch [17/32], Training Loss: 4.0037
Epoch [5/5], Batch [18/32], Training Loss: 4.0178
Epoch [5/5], Batch [19/32], Training Loss: 4.0093
Epoch [5/5], Batch [20/32], Training Loss: 4.0125
Epoch [5/5], Batch [21/32], Training Loss: 4.0115
Epoch [5/5], Batch [22/32], Training Loss: 4.0041
Epoch [5/5], Batch [23/32], Training Loss: 3.9962
Epoch [5/5], Batch [24/32], Training Loss: 3.9970
Epoch [5/5], Batch [25/32], Training Loss: 3.9938
Epoch [5/5], Batch [26/32], Training Loss: 3.9911
Epoch [5/5], Batch [27/32], Training Loss: 3.9940
Epoch [5/5], Batch [28/32], Training Loss: 3.9962
Epoch [5/5], Batch [29/32], Training Loss: 3.9917
Epoch [5/5], Batch [30/32], Training Loss: 3.9932
Epoch [5/5], Batch [31/32], Training Loss: 3.9879
Epoch [5/5], Batch [32/32], Training Loss: 3.9920
Epoch [5/5], Average Training Loss: 4.0098
Epoch [5/5], Batch [1/8], Validation Loss: 4.0429
Epoch [5/5], Batch [2/8], Validation Loss: 4.0184
Epoch [5/5], Batch [3/8], Validation Loss: 4.0200
Epoch [5/5], Batch [4/8], Validation Loss: 4.0352
Epoch [5/5], Batch [5/8], Validation Loss: 4.0473
Epoch [5/5], Batch [6/8], Validation Loss: 4.0437
Epoch [5/5], Batch [7/8], Validation Loss: 4.0443
Epoch [5/5], Batch [8/8], Validation Loss: 4.0466
Epoch [5/5], Average Validation Loss: 4.0373
Training finished! Training took 0:05:02.993448 (2024-05-08 04:20:40.286331
to 2024-05-08 04:25:43.279779)
```

```
In [ ]: torch.save(model.state_dict(), f'./{num_hidden_layers}_layer_{hidden_dimensi
```

With the current configuration, we get a test set accuracy of over 80%, with this specific model checkpoint achieving 82.8% accuracy with the particular random seed set. The original paper only measured overall accuracy for each DDI type, so we reflect that by only measuring the accuracy for an evaluation metric. The original paper also used 0.47 as a cutoff for prediction that there is a drug-drug interaction, so we also modified this from standard practice of using 0.5.

```
In [ ]: model.load_state_dict(torch.load(f'./{num_hidden_layers}_layer_{hidden_dimer
```

```
Out[ ]: <All keys matched successfully>
```

Evaluation

We follow the same metric for evaluation as the original paper by recording the average accuracy across each of the 86 DDI types. The original paper did not note if the averages were weighted, so we assumed a simple average of the accuracies. The implementation is below:

```
In [ ]: # Test set evaluation
        model.eval()
        predictions = []
        true_labels = []
        with torch.no_grad():
            for data in test_dataloader:
                inputs_left, inputs_right, labels = data
                inputs_left, inputs_right = inputs_left.to(device), inputs_right.to(

                outputs = model(inputs_left, inputs_right)
                predicted_labels = (outputs.squeeze() > 0.47).float()
                predictions.extend(predicted_labels.tolist())
                true_labels.extend(labels.tolist())

        ddi_accuracies = {}
        for ddi_type in range(len(true_labels[0])):
          ddi_true = [true_label[ddi_type] for true_label in true_labels]
          ddi_pred = [pred_label[ddi_type] for pred_label in predictions]
          ddi_accuracy = accuracy_score(ddi_true, ddi_pred)
          ddi_accuracies[ddi_type] = ddi_accuracy
          print(f'DDI {ddi_type+1} accuracy: {ddi_accuracy}')
```

```
DDI 1 accuracy: 0.966796875
DDI 2 accuracy: 0.85302734375
DDI 3 accuracy: 0.83154296875
DDI 4 accuracy: 0.97509765625
DDI 5 accuracy: 0.87744140625
DDI 6 accuracy: 0.390625
DDI 7 accuracy: 0.57763671875
DDI 8 accuracy: 0.921875
DDI 9 accuracy: 0.078125
DDI 10 accuracy: 0.3935546875
DDI 11 accuracy: 0.80517578125
DDI 12 accuracy: 0.892578125
DDI 13 accuracy: 0.166015625
DDI 14 accuracy: 0.83056640625
DDI 15 accuracy: 0.87548828125
DDI 16 accuracy: 0.96484375
DDI 17 accuracy: 0.70751953125
DDI 18 accuracy: 0.95703125
DDI 19 accuracy: 0.966796875
DDI 20 accuracy: 0.96728515625
DDI 21 accuracy: 0.64453125
DDI 22 accuracy: 0.92138671875
DDI 23 accuracy: 0.69677734375
DDI 24 accuracy: 0.96875
DDI 25 accuracy: 0.9970703125
DDI 26 accuracy: 0.515625
DDI 27 accuracy: 0.96240234375
DDI 28 accuracy: 0.97607421875
DDI 29 accuracy: 0.80810546875
DDI 30 accuracy: 0.869140625
DDI 31 accuracy: 0.93212890625
DDI 32 accuracy: 0.94921875
DDI 33 accuracy: 0.9501953125
DDI 34 accuracy: 0.84814453125
DDI 35 accuracy: 0.73779296875
DDI 36 accuracy: 0.99609375
DDI 37 accuracy: 0.9619140625
DDI 38 accuracy: 0.662109375
DDI 39 accuracy: 0.875
DDI 40 accuracy: 0.97216796875
DDI 41 accuracy: 0.92626953125
DDI 42 accuracy: 0.98388671875
DDI 43 accuracy: 0.9716796875
DDI 44 accuracy: 0.80908203125
DDI 45 accuracy: 0.84130859375
DDI 46 accuracy: 0.50927734375
DDI 47 accuracy: 0.72021484375
DDI 48 accuracy: 0.7744140625
DDI 49 accuracy: 0.962890625
DDI 50 accuracy: 0.796875
DDI 51 accuracy: 0.9775390625
DDI 52 accuracy: 0.52294921875
DDI 53 accuracy: 0.95703125
DDI 54 accuracy: 0.87158203125
DDI 55 accuracy: 0.9931640625
DDI 56 accuracy: 0.927734375
DDI 57 accuracy: 0.67822265625
DDI 58 accuracy: 0.970703125
DDI 59 accuracy: 0.84423828125
DDI 60 accuracy: 0.97265625
DDI 61 accuracy: 0.642578125
DDI 62 accuracy: 0.98095703125
DDI 63 accuracy: 0.96826171875
DDI 64 accuracy: 0.62939453125
DDI 65 accuracy: 0.833984375
DDI 66 accuracy: 0.75341796875
DDI 67 accuracy: 0.46533203125
DDI 68 accuracy: 0.970703125
DDI 69 accuracy: 0.80517578125
DDI 70 accuracy: 0.8369140625
DDI 71 accuracy: 0.923828125
DDI 72 accuracy: 0.5791015625
DDI 73 accuracy: 0.98291015625
DDI 74 accuracy: 0.7900390625
DDI 75 accuracy: 0.65087890625
DDI 76 accuracy: 0.314453125
DDI 77 accuracy: 0.9287109375
DDI 78 accuracy: 0.65966796875
DDI 79 accuracy: 0.8544921875
DDI 80 accuracy: 0.97119140625
DDI 81 accuracy: 0.86279296875
DDI 82 accuracy: 0.88427734375
DDI 83 accuracy: 0.89013671875
DDI 84 accuracy: 0.94384765625
DDI 85 accuracy: 0.9814453125
DDI 86 accuracy: 0.98583984375
```

In [ ]:
```
sum(list(ddi_accuracies.values()))/len(ddi_accuracies)
```

Out[ ]:
```
0.8179732921511628
```

In [ ]:
```
n_components = 30  # Specify the desired number of components
pca_model = train_pca_model(train_subset_dataset, n_components)
```

```
[04:51:22] SMILES Parse Error: syntax error while parsing: OC1=CC=CC(=C1)C-1
=C2\CCC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C
1)C1=CC(O)=CC=C1)\C1=CC(O)=CC=C1
[04:51:22] SMILES Parse Error: Failed parsing SMILES 'OC1=CC=CC(=C1)C-1=C2\C
CC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=
CC(O)=CC=C1)\C1=CC(O)=CC=C1' for input: 'OC1=CC=CC(=C1)C-1=C2\CCC(=N2)\C(=C
2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=CC(O)=CC=C1)
\C1=CC(O)=CC=C1'
```

In [ ]:
```
# Initialize the model
model = DeepDDI(
    drug_feature_length=n_components,
    num_hidden_layers=num_hidden_layers,
    hidden_dimension=hidden_dimension,
    num_categories=num_categories,
).to(device)

optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Create an instance of DDIDatasetPCA
pca_train_dataset = DDIDatasetPCA(train_subset_dataset, pca_model)
pca_val_dataset = DDIDatasetPCA(val_dataset, pca_model)
pca_test_dataset = DDIDatasetPCA(test_dataset, pca_model)

train_dataloader = DataLoader(pca_train_dataset, batch_size=batch_size, shuf
val_dataloader = DataLoader(pca_val_dataset, batch_size=batch_size, shuffle=
test_dataloader = DataLoader(pca_test_dataset, batch_size=batch_size, shuffl
```

In [ ]:
```
# Training loop
```

```
start = datetime.now()
print(f'Starting at {start}')

for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(train_dataloader):
        inputs_left, inputs_right, labels = data
        inputs_left, inputs_right = inputs_left.to(device), inputs_right.to(

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass

        outputs = model(inputs_left, inputs_right)
        inputs_left, inputs_right = inputs_left.cpu(), inputs_right.cpu()
        labels = labels.to(device)
        loss = criterion(outputs, labels)
        labels = labels.cpu()
        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        # Print the loss for each batch
        print(f"Epoch [{epoch+1}/{num_epochs}], Batch [{i+1}/{len(train_data

    # Print the average loss for the epoch
    epoch_loss = running_loss / len(train_dataloader)
    print(f"Epoch [{epoch+1}/{num_epochs}], Average Training Loss: {epoch_lo

    # Validation
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for i, data in enumerate(val_dataloader):
            inputs_left, inputs_right, labels = data
            inputs_left, inputs_right = inputs_left.to(device), inputs_right
            outputs = model(inputs_left, inputs_right)
            inputs_left, inputs_right = inputs_left.cpu(), inputs_right.cpu(
            labels = labels.to(device)
            loss = criterion(outputs.squeeze(), labels)
            labels = labels.cpu()
            val_loss += loss.item()
            print(f"Epoch [{epoch+1}/{num_epochs}], Batch [{i+1}/{len(val_da

    val_loss /= len(val_dataloader)
    print(f"Epoch [{epoch+1}/{num_epochs}], Average Validation Loss: {val_lo
    model.train()

end = datetime.now()
```

```
print(f"Training finished! Training took {end - start} ({start} to {end})")
```

```
Starting at 2024-05-08 04:51:59.569979
Epoch [1/5], Batch [1/32], Training Loss: 4.4333
Epoch [1/5], Batch [2/32], Training Loss: 4.4262
Epoch [1/5], Batch [3/32], Training Loss: 4.4174
Epoch [1/5], Batch [4/32], Training Loss: 4.4182
Epoch [1/5], Batch [5/32], Training Loss: 4.4320
[04:52:10] SMILES Parse Error: syntax error while parsing: OC1=CC=CC(=C1)C-1
=C2\CCC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C
1)C1=CC(O)=CC=C1)\C1=CC(O)=CC=C1
[04:52:10] SMILES Parse Error: Failed parsing SMILES 'OC1=CC=CC(=C1)C-1=C2\C
CC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=
CC(O)=CC=C1)\C1=CC(O)=CC=C1' for input: 'OC1=CC=CC(=C1)C-1=C2\CCC(=N2)\C(=C
2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=CC(O)=CC=C1)
\C1=CC(O)=CC=C1'
Epoch [1/5], Batch [6/32], Training Loss: 4.4139
Epoch [1/5], Batch [7/32], Training Loss: 4.4084
Epoch [1/5], Batch [8/32], Training Loss: 4.4103
Epoch [1/5], Batch [9/32], Training Loss: 4.4046
Epoch [1/5], Batch [10/32], Training Loss: 4.4115
Epoch [1/5], Batch [11/32], Training Loss: 4.4050
Epoch [1/5], Batch [12/32], Training Loss: 4.4058
Epoch [1/5], Batch [13/32], Training Loss: 4.4019
Epoch [1/5], Batch [14/32], Training Loss: 4.3963
Epoch [1/5], Batch [15/32], Training Loss: 4.3886
Epoch [1/5], Batch [16/32], Training Loss: 4.3874
Epoch [1/5], Batch [17/32], Training Loss: 4.3847
Epoch [1/5], Batch [18/32], Training Loss: 4.3882
Epoch [1/5], Batch [19/32], Training Loss: 4.3888
Epoch [1/5], Batch [20/32], Training Loss: 4.3761
Epoch [1/5], Batch [21/32], Training Loss: 4.3905
Epoch [1/5], Batch [22/32], Training Loss: 4.3808
Epoch [1/5], Batch [23/32], Training Loss: 4.3738
Epoch [1/5], Batch [24/32], Training Loss: 4.3832
Epoch [1/5], Batch [25/32], Training Loss: 4.3703
Epoch [1/5], Batch [26/32], Training Loss: 4.3786
Epoch [1/5], Batch [27/32], Training Loss: 4.3612
Epoch [1/5], Batch [28/32], Training Loss: 4.3645
Epoch [1/5], Batch [29/32], Training Loss: 4.3625
Epoch [1/5], Batch [30/32], Training Loss: 4.3596
Epoch [1/5], Batch [31/32], Training Loss: 4.3561
Epoch [1/5], Batch [32/32], Training Loss: 4.3604
Epoch [1/5], Average Training Loss: 4.3919
Epoch [1/5], Batch [1/8], Validation Loss: 4.3638
Epoch [1/5], Batch [2/8], Validation Loss: 4.3372
Epoch [1/5], Batch [3/8], Validation Loss: 4.3353
Epoch [1/5], Batch [4/8], Validation Loss: 4.3474
Epoch [1/5], Batch [5/8], Validation Loss: 4.3511
Epoch [1/5], Batch [6/8], Validation Loss: 4.3502
Epoch [1/5], Batch [7/8], Validation Loss: 4.3487
Epoch [1/5], Batch [8/8], Validation Loss: 4.3517
Epoch [1/5], Average Validation Loss: 4.3482
Epoch [2/5], Batch [1/32], Training Loss: 4.3652
Epoch [2/5], Batch [2/32], Training Loss: 4.3458
Epoch [2/5], Batch [3/32], Training Loss: 4.3433
Epoch [2/5], Batch [4/32], Training Loss: 4.3390
Epoch [2/5], Batch [5/32], Training Loss: 4.3409
Epoch [2/5], Batch [6/32], Training Loss: 4.3478
Epoch [2/5], Batch [7/32], Training Loss: 4.3456
Epoch [2/5], Batch [8/32], Training Loss: 4.3344
Epoch [2/5], Batch [9/32], Training Loss: 4.3409
Epoch [2/5], Batch [10/32], Training Loss: 4.3381
Epoch [2/5], Batch [11/32], Training Loss: 4.3413
Epoch [2/5], Batch [12/32], Training Loss: 4.3314
Epoch [2/5], Batch [13/32], Training Loss: 4.3317
Epoch [2/5], Batch [14/32], Training Loss: 4.3096
Epoch [2/5], Batch [15/32], Training Loss: 4.3101
Epoch [2/5], Batch [16/32], Training Loss: 4.3249
Epoch [2/5], Batch [17/32], Training Loss: 4.3177
Epoch [2/5], Batch [18/32], Training Loss: 4.3177
Epoch [2/5], Batch [19/32], Training Loss: 4.3231
Epoch [2/5], Batch [20/32], Training Loss: 4.3276
```

```
[04:54:58] SMILES Parse Error: syntax error while parsing: OC1=CC=CC(=C1)C-1
=C2\CCC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C
1)C1=CC(O)=CC=C1)\C1=CC(O)=CC=C1
[04:54:58] SMILES Parse Error: Failed parsing SMILES 'OC1=CC=CC(=C1)C-1=C2\C
CC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=
CC(O)=CC=C1)\C1=CC(O)=CC=C1' for input: 'OC1=CC=CC(=C1)C-1=C2\CCC(=N2)\C(=C
2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=CC(O)=CC=C1)
\C1=CC(O)=CC=C1'
```

```
Epoch [2/5], Batch [21/32], Training Loss: 4.3188
Epoch [2/5], Batch [22/32], Training Loss: 4.2948
Epoch [2/5], Batch [23/32], Training Loss: 4.3068
Epoch [2/5], Batch [24/32], Training Loss: 4.3046
Epoch [2/5], Batch [25/32], Training Loss: 4.3018
Epoch [2/5], Batch [26/32], Training Loss: 4.2935
Epoch [2/5], Batch [27/32], Training Loss: 4.3036
Epoch [2/5], Batch [28/32], Training Loss: 4.2966
Epoch [2/5], Batch [29/32], Training Loss: 4.2981
Epoch [2/5], Batch [30/32], Training Loss: 4.2935
Epoch [2/5], Batch [31/32], Training Loss: 4.2835
Epoch [2/5], Batch [32/32], Training Loss: 4.2955
Epoch [2/5], Average Training Loss: 4.3208
Epoch [2/5], Batch [1/8], Validation Loss: 4.2926
Epoch [2/5], Batch [2/8], Validation Loss: 4.2610
Epoch [2/5], Batch [3/8], Validation Loss: 4.2617
Epoch [2/5], Batch [4/8], Validation Loss: 4.2776
Epoch [2/5], Batch [5/8], Validation Loss: 4.2830
Epoch [2/5], Batch [6/8], Validation Loss: 4.2799
Epoch [2/5], Batch [7/8], Validation Loss: 4.2764
Epoch [2/5], Batch [8/8], Validation Loss: 4.2833
Epoch [2/5], Average Validation Loss: 4.2769
Epoch [3/5], Batch [1/32], Training Loss: 4.2947
Epoch [3/5], Batch [2/32], Training Loss: 4.2924
Epoch [3/5], Batch [3/32], Training Loss: 4.2856
Epoch [3/5], Batch [4/32], Training Loss: 4.2753
Epoch [3/5], Batch [5/32], Training Loss: 4.2869
Epoch [3/5], Batch [6/32], Training Loss: 4.2634
Epoch [3/5], Batch [7/32], Training Loss: 4.2754
Epoch [3/5], Batch [8/32], Training Loss: 4.2570
Epoch [3/5], Batch [9/32], Training Loss: 4.2841
Epoch [3/5], Batch [10/32], Training Loss: 4.2777
Epoch [3/5], Batch [11/32], Training Loss: 4.2657
Epoch [3/5], Batch [12/32], Training Loss: 4.2604
Epoch [3/5], Batch [13/32], Training Loss: 4.2664
Epoch [3/5], Batch [14/32], Training Loss: 4.2838
Epoch [3/5], Batch [15/32], Training Loss: 4.2603
Epoch [3/5], Batch [16/32], Training Loss: 4.2573
Epoch [3/5], Batch [17/32], Training Loss: 4.2612
Epoch [3/5], Batch [18/32], Training Loss: 4.2482
Epoch [3/5], Batch [19/32], Training Loss: 4.2381
Epoch [3/5], Batch [20/32], Training Loss: 4.2564
```

```
[04:57:09] SMILES Parse Error: syntax error while parsing: OC1=CC=CC(=C1)C-1
=C2\CCC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C
1)C1=CC(O)=CC=C1)\C1=CC(O)=CC=C1
[04:57:09] SMILES Parse Error: Failed parsing SMILES 'OC1=CC=CC(=C1)C-1=C2\C
CC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=
CC(O)=CC=C1)\C1=CC(O)=CC=C1' for input: 'OC1=CC=CC(=C1)C-1=C2\CCC(=N2)\C(=C
2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=CC(O)=CC=C1)
\C1=CC(O)=CC=C1'
```

```
Epoch [3/5], Batch [21/32], Training Loss: 4.2354
Epoch [3/5], Batch [22/32], Training Loss: 4.2441
Epoch [3/5], Batch [23/32], Training Loss: 4.2446
Epoch [3/5], Batch [24/32], Training Loss: 4.2536
Epoch [3/5], Batch [25/32], Training Loss: 4.2459
Epoch [3/5], Batch [26/32], Training Loss: 4.2344
Epoch [3/5], Batch [27/32], Training Loss: 4.2430
Epoch [3/5], Batch [28/32], Training Loss: 4.2425
Epoch [3/5], Batch [29/32], Training Loss: 4.2212
Epoch [3/5], Batch [30/32], Training Loss: 4.2334
Epoch [3/5], Batch [31/32], Training Loss: 4.2324
Epoch [3/5], Batch [32/32], Training Loss: 4.2332
Epoch [3/5], Average Training Loss: 4.2579
Epoch [3/5], Batch [1/8], Validation Loss: 4.2288
Epoch [3/5], Batch [2/8], Validation Loss: 4.1943
Epoch [3/5], Batch [3/8], Validation Loss: 4.1971
Epoch [3/5], Batch [4/8], Validation Loss: 4.2168
Epoch [3/5], Batch [5/8], Validation Loss: 4.2226
Epoch [3/5], Batch [6/8], Validation Loss: 4.2172
Epoch [3/5], Batch [7/8], Validation Loss: 4.2115
Epoch [3/5], Batch [8/8], Validation Loss: 4.2229
Epoch [3/5], Average Validation Loss: 4.2139
Epoch [4/5], Batch [1/32], Training Loss: 4.2359
Epoch [4/5], Batch [2/32], Training Loss: 4.2195
Epoch [4/5], Batch [3/32], Training Loss: 4.2089
Epoch [4/5], Batch [4/32], Training Loss: 4.2226
Epoch [4/5], Batch [5/32], Training Loss: 4.2227
Epoch [4/5], Batch [6/32], Training Loss: 4.2144
Epoch [4/5], Batch [7/32], Training Loss: 4.2180
Epoch [4/5], Batch [8/32], Training Loss: 4.2280
Epoch [4/5], Batch [9/32], Training Loss: 4.2052
Epoch [4/5], Batch [10/32], Training Loss: 4.2159
Epoch [4/5], Batch [11/32], Training Loss: 4.2153
[04:58:41] SMILES Parse Error: syntax error while parsing: OC1=CC=CC(=C1)C-1
=C2\CCC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C
1)C1=CC(O)=CC=C1)\C1=CC(O)=CC=C1
[04:58:41] SMILES Parse Error: Failed parsing SMILES 'OC1=CC=CC(=C1)C-1=C2\C
CC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=
CC(O)=CC=C1)\C1=CC(O)=CC=C1' for input: 'OC1=CC=CC(=C1)C-1=C2\CCC(=N2)\C(=C
2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=CC(O)=CC=C1)
\C1=CC(O)=CC=C1'
Epoch [4/5], Batch [12/32], Training Loss: 4.2210
Epoch [4/5], Batch [13/32], Training Loss: 4.2149
Epoch [4/5], Batch [14/32], Training Loss: 4.2138
Epoch [4/5], Batch [15/32], Training Loss: 4.2069
Epoch [4/5], Batch [16/32], Training Loss: 4.1994
Epoch [4/5], Batch [17/32], Training Loss: 4.2039
Epoch [4/5], Batch [18/32], Training Loss: 4.1955
Epoch [4/5], Batch [19/32], Training Loss: 4.1782
Epoch [4/5], Batch [20/32], Training Loss: 4.1931
Epoch [4/5], Batch [21/32], Training Loss: 4.2054
Epoch [4/5], Batch [22/32], Training Loss: 4.1892
Epoch [4/5], Batch [23/32], Training Loss: 4.1892
Epoch [4/5], Batch [24/32], Training Loss: 4.1786
Epoch [4/5], Batch [25/32], Training Loss: 4.1886
Epoch [4/5], Batch [26/32], Training Loss: 4.1697
Epoch [4/5], Batch [27/32], Training Loss: 4.1772
Epoch [4/5], Batch [28/32], Training Loss: 4.1979
Epoch [4/5], Batch [29/32], Training Loss: 4.1925
Epoch [4/5], Batch [30/32], Training Loss: 4.1921
Epoch [4/5], Batch [31/32], Training Loss: 4.1730
Epoch [4/5], Batch [32/32], Training Loss: 4.1648
Epoch [4/5], Average Training Loss: 4.2016
Epoch [4/5], Batch [1/8], Validation Loss: 4.1711
Epoch [4/5], Batch [2/8], Validation Loss: 4.1353
Epoch [4/5], Batch [3/8], Validation Loss: 4.1397
Epoch [4/5], Batch [4/8], Validation Loss: 4.1629
Epoch [4/5], Batch [5/8], Validation Loss: 4.1687
Epoch [4/5], Batch [6/8], Validation Loss: 4.1612
Epoch [4/5], Batch [7/8], Validation Loss: 4.1532
Epoch [4/5], Batch [8/8], Validation Loss: 4.1681
Epoch [4/5], Average Validation Loss: 4.1575
Epoch [5/5], Batch [1/32], Training Loss: 4.1672
Epoch [5/5], Batch [2/32], Training Loss: 4.1625
Epoch [5/5], Batch [3/32], Training Loss: 4.1889
Epoch [5/5], Batch [4/32], Training Loss: 4.1724
Epoch [5/5], Batch [5/32], Training Loss: 4.1790
Epoch [5/5], Batch [6/32], Training Loss: 4.1855
Epoch [5/5], Batch [7/32], Training Loss: 4.1584
Epoch [5/5], Batch [8/32], Training Loss: 4.1608
Epoch [5/5], Batch [9/32], Training Loss: 4.1663
Epoch [5/5], Batch [10/32], Training Loss: 4.1345
Epoch [5/5], Batch [11/32], Training Loss: 4.1451
Epoch [5/5], Batch [12/32], Training Loss: 4.1592
Epoch [5/5], Batch [13/32], Training Loss: 4.1381
Epoch [5/5], Batch [14/32], Training Loss: 4.1556
Epoch [5/5], Batch [15/32], Training Loss: 4.1487
Epoch [5/5], Batch [16/32], Training Loss: 4.1548
Epoch [5/5], Batch [17/32], Training Loss: 4.1466
Epoch [5/5], Batch [18/32], Training Loss: 4.1570
Epoch [5/5], Batch [19/32], Training Loss: 4.1304
Epoch [5/5], Batch [20/32], Training Loss: 4.1351
Epoch [5/5], Batch [21/32], Training Loss: 4.1495
Epoch [5/5], Batch [22/32], Training Loss: 4.1332
Epoch [5/5], Batch [23/32], Training Loss: 4.1505
Epoch [5/5], Batch [24/32], Training Loss: 4.1541
Epoch [5/5], Batch [25/32], Training Loss: 4.1460
Epoch [5/5], Batch [26/32], Training Loss: 4.1348
Epoch [5/5], Batch [27/32], Training Loss: 4.1362
Epoch [5/5], Batch [28/32], Training Loss: 4.1427
Epoch [5/5], Batch [29/32], Training Loss: 4.1514
Epoch [5/5], Batch [30/32], Training Loss: 4.1215
```

```
[05:01:35] SMILES Parse Error: syntax error while parsing: OC1=CC=CC(=C1)C-1
=C2\CCC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C
1)C1=CC(O)=CC=C1)\C1=CC(O)=CC=C1
[05:01:35] SMILES Parse Error: Failed parsing SMILES 'OC1=CC=CC(=C1)C-1=C2\C
CC(=N2)\C(=C2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=
CC(O)=CC=C1)\C1=CC(O)=CC=C1' for input: 'OC1=CC=CC(=C1)C-1=C2\CCC(=N2)\C(=C
2/N\C(\C=C2)=C(/C2=N/C(/C=C2)=C(\C2=CC=C\-1N2)C1=CC(O)=CC=C1)C1=CC(O)=CC=C1)
\C1=CC(O)=CC=C1'
Epoch [5/5], Batch [31/32], Training Loss: 4.1335
Epoch [5/5], Batch [32/32], Training Loss: 4.1149
Epoch [5/5], Average Training Loss: 4.1505
Epoch [5/5], Batch [1/8], Validation Loss: 4.1186
Epoch [5/5], Batch [2/8], Validation Loss: 4.0824
Epoch [5/5], Batch [3/8], Validation Loss: 4.0879
Epoch [5/5], Batch [4/8], Validation Loss: 4.1150
Epoch [5/5], Batch [5/8], Validation Loss: 4.1204
Epoch [5/5], Batch [6/8], Validation Loss: 4.1109
Epoch [5/5], Batch [7/8], Validation Loss: 4.1008
Epoch [5/5], Batch [8/8], Validation Loss: 4.1178
Epoch [5/5], Average Validation Loss: 4.1067
Training finished! Training took 0:10:06.018729 (2024-05-08 04:51:59.569979
to 2024-05-08 05:02:05.588708)
```

```
In [ ]: torch.save(model.state_dict(), f'./{n_components}_pca_{num_hidden_layers}_la
```

```
In [ ]: model.load_state_dict(torch.load(f'./{n_components}_pca_{num_hidden_layers}_
```

```
Out[ ]: <All keys matched successfully>
```

```
In [ ]: # Test set evaluation
        model.eval()
        predictions = []
        true_labels = []
        with torch.no_grad():
            for data in test_dataloader:
                inputs_left, inputs_right, labels = data
                inputs_left, inputs_right = inputs_left.to(device), inputs_right.to(

                outputs = model(inputs_left, inputs_right)
                predicted_labels = (outputs.squeeze() > 0.47).float()
                predictions.extend(predicted_labels.tolist())
                true_labels.extend(labels.tolist())

        ddi_accuracies = {}
        for ddi_type in range(len(true_labels[0])):
          ddi_true = [true_label[ddi_type] for true_label in true_labels]
          ddi_pred = [pred_label[ddi_type] for pred_label in predictions]
          ddi_accuracy = accuracy_score(ddi_true, ddi_pred)
          ddi_accuracies[ddi_type] = ddi_accuracy
          print(f'DDI {ddi_type+1} accuracy: {ddi_accuracy}')
```

```
DDI 1 accuracy: 0.75244140625
DDI 2 accuracy: 0.49072265625
DDI 3 accuracy: 0.79541015625
DDI 4 accuracy: 0.701171875
DDI 5 accuracy: 0.89697265625
DDI 6 accuracy: 0.20556640625
DDI 7 accuracy: 0.19091796875
DDI 8 accuracy: 0.94287109375
DDI 9 accuracy: 0.09326171875
DDI 10 accuracy: 0.19677734375
DDI 11 accuracy: 0.8818359375
DDI 12 accuracy: 0.857421875
DDI 13 accuracy: 0.10205078125
DDI 14 accuracy: 0.90966796875
DDI 15 accuracy: 0.6650390625
DDI 16 accuracy: 0.98681640625
DDI 17 accuracy: 0.81640625
DDI 18 accuracy: 0.8837890625
DDI 19 accuracy: 0.916015625
DDI 20 accuracy: 0.9970703125
DDI 21 accuracy: 0.89697265625
DDI 22 accuracy: 0.8955078125
DDI 23 accuracy: 0.90234375
DDI 24 accuracy: 0.92431640625
DDI 25 accuracy: 0.97265625
DDI 26 accuracy: 0.3447265625
DDI 27 accuracy: 0.89013671875
DDI 28 accuracy: 0.89794921875
DDI 29 accuracy: 0.91943359375
DDI 30 accuracy: 0.90966796875
DDI 31 accuracy: 0.96728515625
DDI 32 accuracy: 0.96337890625
DDI 33 accuracy: 0.88037109375
DDI 34 accuracy: 0.7841796875
DDI 35 accuracy: 0.4892578125
DDI 36 accuracy: 0.77392578125
DDI 37 accuracy: 0.8564453125
DDI 38 accuracy: 0.974609375
DDI 39 accuracy: 0.9501953125
DDI 40 accuracy: 0.78369140625
DDI 41 accuracy: 0.82666015625
DDI 42 accuracy: 0.8642578125
DDI 43 accuracy: 0.95166015625
DDI 44 accuracy: 0.97412109375
DDI 45 accuracy: 0.93896484375
DDI 46 accuracy: 0.35546875
DDI 47 accuracy: 0.9287109375
DDI 48 accuracy: 0.7880859375
DDI 49 accuracy: 0.888671875
DDI 50 accuracy: 0.4951171875
DDI 51 accuracy: 0.94287109375
DDI 52 accuracy: 0.40087890625
DDI 53 accuracy: 0.81494140625
DDI 54 accuracy: 0.93310546875
DDI 55 accuracy: 0.95751953125
DDI 56 accuracy: 0.859375
DDI 57 accuracy: 0.8232421875
DDI 58 accuracy: 0.83544921875
DDI 59 accuracy: 0.8544921875
DDI 60 accuracy: 0.92333984375
DDI 61 accuracy: 0.90625
DDI 62 accuracy: 0.7919921875
DDI 63 accuracy: 0.935546875
DDI 64 accuracy: 0.54296875
DDI 65 accuracy: 0.68310546875
DDI 66 accuracy: 0.90087890625
DDI 67 accuracy: 0.16845703125
DDI 68 accuracy: 0.80712890625
DDI 69 accuracy: 0.80810546875
DDI 70 accuracy: 0.90185546875
DDI 71 accuracy: 0.994140625
DDI 72 accuracy: 0.66845703125
DDI 73 accuracy: 0.96044921875
DDI 74 accuracy: 0.81201171875
DDI 75 accuracy: 0.86279296875
DDI 76 accuracy: 0.07373046875
DDI 77 accuracy: 0.974609375
DDI 78 accuracy: 0.80859375
DDI 79 accuracy: 0.7607421875
DDI 80 accuracy: 0.99755859375
DDI 81 accuracy: 0.8173828125
DDI 82 accuracy: 0.97265625
DDI 83 accuracy: 0.96630859375
DDI 84 accuracy: 0.9521484375
DDI 85 accuracy: 0.9775390625
DDI 86 accuracy: 0.79296875
```

```
In [ ]: sum(list(ddi_accuracies.values()))/len(ddi_accuracies)
```

```
Out[ ]: 0.7843556958575582
```

Results

Below is the summarized table of results for both the original hypothesis of whether we can accurately predict hold-out DDI samples (90%+ reported in the paper), and for the ablations:

Number of Layers	Hidden Dimension Size	PCA Dimension	Average DDI Accuracy
2	128	NA	0.818
2	128	10	0.818
2	128	30	0.784
2	128	50	0.789
4	128	NA	0.784

4	128	50	0.743
8	128	NA	0.754
8	128	50	0.560
2	2048	NA	0.981
2	2048	50	0.961
4	2048	NA	0.984
4	2048	50	0.975
8	2048	NA	0.948
8	2048	50	0.961

We can see from the results that several models performed well above 90% in average DDI accuracy. The paper reported that the 9-layer model with a hidden dimension size of 2,048 performed the best, however we saw that the 4-layer model with a hidden dimension size of 2,048 performed best. We can also see that all models with a hidden dimension size of 2,048 achieved 90%+ mean accuracy. Therefore, it is likely that even with our small subset of the full dataset that we can accurately predict unseen DDIs.

For the ablations, we first modified the complexity of the model by scaling the number of layers and hidden dimension size. We can see the trend that with the paper's 50 principal component dimensionality reduction and holding hidden dimension size constant, there is a negative correlation between the number of layers and performance. It is possible that with our reduced dataset that the model is overfitting, and the shallower models have better generalization.

Additionally, there is a clear positive correlation between model performance and the hidden dimension size, however intermediate hidden dimension sizes between 128 and 2048 were not tested.

Finally, we can see with our ablation study determining the impact of the PCA dimensionality reduction of the molecular fingerprint, there is no clear trend. Holding the number of layers (2) and the hidden dimension size (128) constant, we can there may be a negative correlation between the PCA dimension size and performance, however this contradicts the original paper's findings.

Discussion

The paper was fairly difficult to reproduce without the refrence to ChemicalX [2]. The amount of compute given the dataset size also made it difficult to reproduce at the same scale. The original paper [1] does include a link to their repository containing "source code," however it does not contain any code for training the models nor does it contain any model weights. Moreover, the dataset was fairly intensive for preprocessing, and the paper does not provide a clean version of their data for immediate use. That being said, the model itself is very simple being a standard feed-forward network, so model implementation was straightforward.

Overall, the findings and takeaways from the study still hold true in that predicting unseen drug-drug interactions is possible using molecular fingerprints based on structural similarity.

I would recommend to the authors that the model definition and trianing code be included in the repository, and the data used with SMILES strings be released as well as opposed to only listing the Drug Bank IDs.

References

1. Ryu, J. Y., Kim, H. U., & Lee, S. Y. (2018). Deep learning improves prediction of drug–drug and drug–food interactions. Proceedings of the national academy of sciences, 115(18), E4304-E4311.

2. Rozemberczki, B., Hoyt, C. T., Gogleva, A., Grabowski, P., Karis, K., Lamov, A., ... & Gyori, B. M. (2022, August). Chemicalx: A deep learning library for drug pair scoring. In Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (pp. 3819-3828).