

Configuration Manual

MSc Research Project
MSc in Data Analytics

Jaswinder Singh
Student ID: x19219997

School of Computing
National College of Ireland

Supervisor: Dr. Paul Stynes
Dr. Pramod Pathak

**National College of Ireland
Project Submission Sheet
School of Computing**



Student Name:	Jaswinder Singh
Student ID:	x19219997
Programme:	MSc in Data Analytics
Year:	2021
Module:	MSc Research Project
Supervisor:	Dr. Paul Stynes, Dr. Pramod Pathak
Submission Due Date:	16/08/2021
Project Title:	Configuration Manual
Word Count:	
Page Count:	22

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	22nd September 2021

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	<input type="checkbox"/>
Attach a Moodle submission receipt of the online project submission , to each project (including multiple copies).	<input type="checkbox"/>
You must ensure that you retain a HARD COPY of the project , both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Jaswinder Singh
x19219997

1 Initial Environment Setting

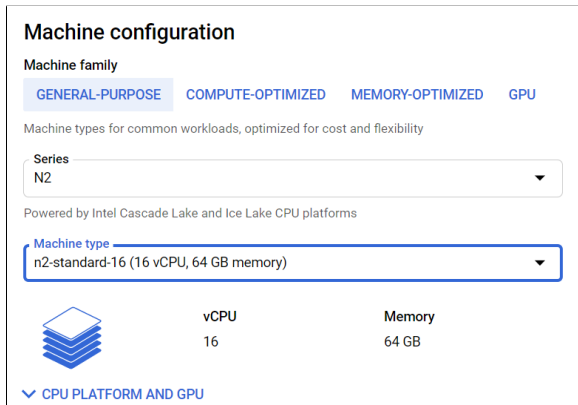
The software specifications for the setup are described in the table below:

Programming Language	Python (v3.6)
Cloud Platform	Google Cloud Platform (GCP)
Virtual Machine OS	Ubuntu 18.04LTS
CPU	No. of cores: 16; Memory: 64 GB
GPU	No. of GPUs: 2; Type: NVIDIA Tesla V100
IDE	Jupyter-Lab, VS-Code, Google Colab, Atom

1.1 Setting up Google Cloud Platform

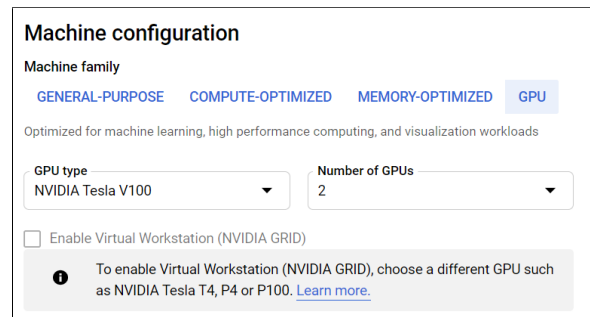
For setting up the google cloud platform for distributed training, the following steps can be followed:

1. First login with a gmail account to th GCP website
2. Choose the default options for free account for first time login. The first time users get a 300\$ credit into their account to be used by three months. We used these credits only to train our final model on the platform.
3. Choose the *Create VM Instance* option.
4. Now choose the CPU and GPU preferences as shown below. The user can choose whatever GPU and CPU specifications he/she wants, but for the purpose of this project, we chose the options as shown in figure 1.



The screenshot shows the 'Machine configuration' page for a new VM instance. Under 'Machine family', the 'GENERAL-PURPOSE' tab is selected. The 'Series' dropdown is set to 'N2'. Below it, a note states 'Powered by Intel Cascade Lake and Ice Lake CPU platforms'. The 'Machine type' dropdown is set to 'n2-standard-16 (16 vCPU, 64 GB memory)'. At the bottom, a summary shows 'vCPU' as 16 and 'Memory' as 64 GB. A blue icon of stacked disks is also visible.

(a) CPU specifications



The screenshot shows the 'Machine configuration' page with the 'GPU' tab selected under 'Machine family'. The 'GPU type' dropdown is set to 'NVIDIA Tesla V100' and the 'Number of GPUs' dropdown is set to '2'. A checkbox for 'Enable Virtual Workstation (NVIDIA GRID)' is present but unchecked. A warning icon and text state: 'To enable Virtual Workstation (NVIDIA GRID), choose a different GPU such as NVIDIA Tesla T4, P4 or P100. [Learn more.](#)'

(b) GPU specifications

Figure 1: Configurations for GPU and CPU for the virutal instance on GCP

1.2 Setting up the Google Colab Environment

To set up the code execution in the google colab environment, the following steps can be followed:

1. Login to the account through gmail.
2. Open the google drive tab and upload the entire dataset folder. this may take a while to complete since the dataset size is quite large (2 GB)
3. Open the google colab notebook and choose the GPU enabled environment for execution of the code.
4. Mount the drive to the colab environment using the below code:

```
from google.colab import drive
drive.mount('/content/drive')
```

5. Once finished uploading the dataset and mounting, open the first *.ipynb* notebook from the artefacts and start the cell execution one by one. Execute the notebooks sequentially to avoid errors.

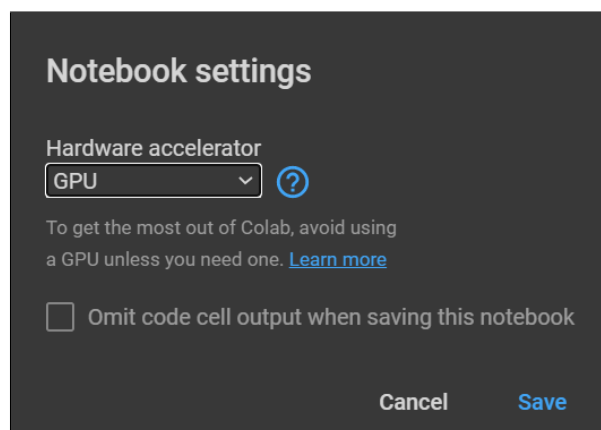


Figure 2: Setting the GPU enabled environment in google colab

2 Data Pre-processing and EDA

2.1 Data Preparation

2.1.1 Initial Setup

1. Download all the dataset files for the CHAMPS dataset from kaggle (figure 3).
2. For the pre-processing stage, simply upload the data on google drive and follow the steps described in section 1.2. For the model training and later steps put the dataset files in a folder and name it *data*.

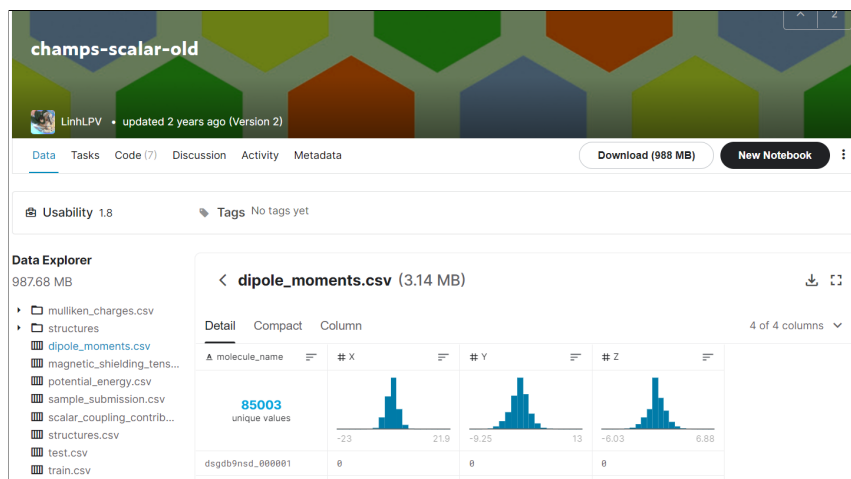


Figure 3: CHAMPS dataset on kaggle

2.2 Data Pre-processing

For pre-processing, follow the following series of steps:

1. Put all the python (.py) files in the *Custom_Modules* folder inside your main directory, i.e the directory in which the current code file is present. These files are actually imported as modules in the following notebook as the functions used in them are used in a lot of files, therefore it makes sense to import them as modules to increase functionality.
2. Create different folders defined in the *constants.py* file in the *Custom_Modules* folder like *data*, *tmp*, *proc_data*, *predictions*, *oofs*. The csv files created after the pre-processing file has been run successfully are stored in these directories.
3. Set up the conda environment for colab through following script:

```
!pip install -q condacolab
import condacolab
condacolab.install()
!conda install -c conda-forge rdkit
```

4. Clone the *xyz2mol* GitHub repository

```
!git clone https://github.com/jensengroup/xyz2mol.git
```

5. Install the modules in the *requirement.txt* file along with the *deepchem* and *utils* library. The easiest way to do that is to install the files contents using pip as follows.

```
!pip install -r requirements.txt
!pip install deepchem utils
```

6. Import the necessary libraries, including the custom modules. Remember the custom modules in this script are *xyz2mol*, *constants*

```

import gc
import numpy as np
import pandas as pd
from itertools import combinations
from glob import glob

import deepchem as dc
from rdkit.Chem import rdmolops, ChemicalFeatures

from xyz2mol import read_xyz_file, xyz2mol
import utils
import constants as C

```

2.2.1 Feature Engineering

1. The raw data path is defined in the *constants.py* file in the *Custom_modules* folder. This is the path where all the files from the CHAMPS dataset are stored. Store the xyz files into a python list using the glob function. (refer to figure 4)

```

mol_feat_columns = ['ave_bond_length', 'std_bond_length', 'ave_atom_weight']
xyz_filepath_list = list(glob(C.RAW_DATA_PATH + 'structures/*.xyz'))
xyz_filepath_list.sort()

```

Figure 4: Storing xyz files into a python list

2. Write the function to create the RDKit molecule objects. The *networkx* library is required if the quick variable is to be set to TRUE.

```

## Functions to create the RDKit mol objects
def mol_from_xyz(filepath, add_hs=True, compute_dist_centre=False):
    """Wrapper function for calling xyz2mol function."""
    charged_fragments = True # alternatively radicals are made

    # quick is faster for large systems but requires networkx
    # if you don't want to install networkx set quick=False and
    # uncomment 'import networkx as nx' at the top of the file
    quick = True

    atomicNumList, charge, xyz_coordinates = read_xyz_file(filepath)
    mol, dMat = xyz2mol(atomicNumList, charge, xyz_coordinates,
                        charged_fragments, quick, check_chiral_stereo=False)

    return mol, np.array(xyz_coordinates), dMat

```

Figure 5: Creating RDKit molecule objects

3. Now, we create the molecules and their distance matrices from the molecule objects created in the previous step. This function returns the molecule names, their ids, molecule features, xyz coordinates, Euclidean distance matrices and graph distance matrices (refer to figure 6).

```
def get_molecules():
    mols, mol_ids, mol_feats = {}, {}, {}
    xyzs, dist_matrices, graph_dist_matrices = {}, {}, {}
    print('Create molecules and distance matrices.')
    for i in range(C.N_MOLS):
        print_progress(i, C.N_MOLS)
        filepath = xyz_filepath_list[i]
        mol_name = filepath.split('/')[-1][:-4]
        mol, xyz, dist_matrix = mol_from_xyz(filepath)
        mols[mol_name] = mol
        xyzs[mol_name] = xyz
        dist_matrices[mol_name] = dist_matrix
        mol_ids[mol_name] = i

    # make padded graph distance matrix dataframes
    n_atoms = len(xyzs)
    graph_dist_matrix = pd.DataFrame(np.pad(
        rdmolops.GetDistanceMatrix(mol),
        [(0, 0), (0, C.MAX_N_ATOMS - n_atoms)], 'constant'
    ))
    graph_dist_matrix['molecule_id'] = n_atoms * [i]
    graph_dist_matrices[mol_name] = graph_dist_matrix

    # compute molecule level features
    adj_matrix = rdmolops.GetAdjacencyMatrix(mol)
    atomic_num_list, _, _ = read_xyz_file(filepath)
    dists = dist_matrix.ravel()[np.tril(adj_matrix).ravel()>=1]
    mol_feats[mol_name] = pd.Series(
        [np.mean(dists), np.std(dists), np.mean(atomic_num_list)],
        index=mol_feat_columns
    )

    return mols, mol_ids, mol_feats, xyzs, dist_matrices, graph_dist_matrices
```

Figure 6: Creating molecules and distance matrices

4. Add the *Euclidean Distance*, *xyz coordinates* to a dataframe df. Apply the mean transformation on the feature *atom_index* in *dist* column. (refer to figure 7)

```
def map_atom_info(df, atom_idx, struct_df):
    """Adds xyz-coordinates of atom_{atom_idx} to 'df'."""
    df = pd.merge(df, struct_df, how='left',
        left_on=['molecule_name', f'atom_index_{atom_idx}'],
        right_on=['molecule_name', 'atom_index'])
    df = df.drop('atom_index', axis=1)
    df = df.rename(columns={'atom': f'atom_{atom_idx}',
        'x': f'x_{atom_idx}',
        'y': f'y_{atom_idx}',
        'z': f'z_{atom_idx}'})
    return df

def add_dist(df, struct_df):
    """Adds euclidean distance between scalar coupling atoms to 'df'."""
    df = map_atom_info(df, 0, struct_df)
    df = map_atom_info(df, 1, struct_df)
    p_0 = df[['x_0', 'y_0', 'z_0']].values
    p_1 = df[['x_1', 'y_1', 'z_1']].values
    df['dist'] = np.linalg.norm(p_0 - p_1, axis=1)
    df.drop(columns=['x_0', 'y_0', 'z_0', 'x_1', 'y_1', 'z_1'], inplace=True)
    return df

def transform_per_atom_group(df, a_idx, col='dist', trans='mean'):
    """Apply transformation 'trans' on feature in 'col' to scalar coupling
    constants grouped at the atom level."""
    return df.groupby(
        ['molecule_name', f'atom_index_{a_idx}'])[col].transform(trans)

def inv_dist_per_atom(df, a_idx, d_col='dist', power=3):
    """Compute sum of inverse distances of scalar coupling constants grouped at
    the atom level."""
    trans = lambda x: 1 / sum(x ** -power)
    return transform_per_atom_group(df, a_idx, d_col, trans=trans)
```

Figure 7: Mapping atom information to the dataframe and applying mean transformation

5. Calculate the cosine and dihedral angle for the atoms using the cross product for-

mula. (refer to figure 8)

```
def dihedral(p):
    """Praxeolitic formula: 1 sqrt, 1 cross product"""
    p0 = p[0]
    p1 = p[1]
    p2 = p[2]
    p3 = p[3]

    b0 = -1.0*(p1 - p0)
    b1 = p2 - p1
    b2 = p3 - p2

    # normalize b1 so that it does not influence magnitude of vector
    # rejections that come next
    b1 /= np.linalg.norm(b1)

    # vector rejections
    v = b0 - np.dot(b0, b1)*b1
    w = b2 - np.dot(b2, b1)*b1

    # angle between v and w in a plane is the torsion angle
    # v and w may not be normalized but that's fine since tan is y/x
    x = np.dot(v, w)
    y = np.dot(np.cross(b1, v), w)
    return np.arctan2(y, x)

def cosine_angle(p):
    p0, p1, p2 = p[0], p[1], p[2]
    v1, v2 = p0 - p1, p2 - p1
    return np.dot(v1, v2) / np.sqrt(np.dot(v1, v1) * np.dot(v2, v2))
```

Figure 8: Angles Calculations for the molecules

- Now, add the scalar coupling edge (euclidean distance) and molecule level (atomic radius, electronegativity) features to the dataframe (refer to figure 9)

```
def add_sc_features(df, structures_df, mol_feats, xyzs, dist_matrices, mol_ids):
    """Add scalar coupling edge and molecule level features to 'df'."""
    # add euclidean distance between scalar coupling atoms
    df = add_dist(df, structures_df)

    # compute distance normalized by scalar coupling type mean and std
    gb_type_dist = df.groupby('type')['dist']
    df['normed_dist'] = ((df['dist'] - gb_type_dist.transform('mean'))
                        / gb_type_dist.transform('std'))

    # add distance features adjusted for atom radii and electronegativity
    df['R0'] = df['atom_0'].map(C.ATOMIC_RADIUS)
    df['R1'] = df['atom_1'].map(C.ATOMIC_RADIUS)
    df['E0'] = df['atom_0'].map(C.ELECTRO_NEG)
    df['E1'] = df['atom_1'].map(C.ELECTRO_NEG)
    df['dist_min_rad'] = df['dist'] - df['R0'] - df['R1']
    df['dist_electro_neg_adj'] = df['dist'] * (df['E0'] + df['E1']) / 2
    df.drop(columns=['R0', 'R1', 'E0', 'E1'], inplace=True)

    # map scalar coupling types to integers and add dummy variables
    df['type'] = df['type'].map(C.TYPES_MAP)
    df = pd.concat((df, pd.get_dummies(df['type'], prefix='type')), axis=1)

    # add angle related features
    df = add_sc_angle_features(df, xyzs, dist_matrices)

    # add molecule level features
    mol_feat_df = pd.concat(mol_feats, axis=1).T
    mol_feat_dict = mol_feat_df.to_dict()
    for f in mol_feat_columns:
        df[f] = df['molecule_name'].map(mol_feat_dict[f])
```

Figure 9: Adding features to the dataframe

7. Now, create the atom and bond level features. The node (atom) level features include the element type, hybridization, number of linked hydrogen atoms, atomic number, aromatic, etc. On the other hand, the edge (bond) level features include bond length, bond type, etc. For the purpose of our study, we engineer some of these features and encode some of them using the one-hot encoding method (refer to figure 10 and 11).

```
## Functions to create atom and bond level features
def one_hot_encoding(x, set):
    one_hot = [int(x == s) for s in set]
    return one_hot

def get_bond_features(mol, eucl_dist):
    """
    Compute the following features for each bond in 'mol':
    - bond type: categorical {1: single, 2: double, 3: triple,
      4: aromatic} (one-hot)
    - is conjugated: bool {0, 1}
    - is in ring: bool {0, 1}
    - euclidean distance: float
    - normalized eucl distance: float
    """
    n_bonds = mol.GetNumBonds()
    features = np.zeros((n_bonds, C.N_BOND_FEATURES))
    bond_idx = np.zeros((n_bonds, 2))
    for n, e in enumerate(mol.GetBonds()):
        i = e.GetBeginAtomIdx()
        j = e.GetEndAtomIdx()
        dc_e_feats = dc.feats.graph_features.bond_features(e).astype(int)
        features[n, :6] = dc_e_feats
        features[n, 6] = eucl_dist[i, j]
        bond_idx[n] = i, j
    sorted_idx = bond_idx[:,0].argsort()
    dists = features[:, 6]
    features[:, 7] = (dists - dists.mean()) / dists.std() # normed_dist
    return features[sorted_idx], bond_idx[sorted_idx]
```

Figure 10: Creating bond level features

```
def get_atom_features(mol, dist_matrix):
    """
    Compute the following features for each atom in 'mol':
    - atom type: H, C, N, O, F (one-hot)
    - degree: 1, 2, 3, 4, 5 (one-hot)
    - Hybridization: SP, SP2, SP3, UNSPECIFIED (one-hot)
    - is aromatic: bool {0, 1}
    - formal charge: int
    - atomic number: float
    - average bond length: float
    - average weight of neighboring atoms: float
    - donor: bool {0, 1}
    - acceptor: bool {0, 1}
    """
    n_atoms = mol.GetNumAtoms()
    features = np.zeros((n_atoms, C.N_ATOM_FEATURES))
    adj_matrix = rdmolops.GetAdjacencyMatrix(mol)
    for a in mol.GetAtoms():
        idx = a.GetIdx()
        if sum(adj_matrix[idx]) > 0:
            ave_bond_length = np.mean(dist_matrix[idx][adj_matrix[idx]==1])
            ave_neighbor_wt = np.mean(
                [n.GetAtomicNum() for n in a.GetNeighbors()])
        else:
            ave_bond_length, ave_neighbor_wt = 0.0, 0.0

        sym = a.GetSymbol()
        a_feats = one_hot_encoding(sym, C.SYMBOLS) \
            + one_hot_encoding(a.GetDegree(), C.DEGREES) \
            + one_hot_encoding(a.GetHybridization(), C.HYBRIDIZATIONS) \
            + [a.GetIsAromatic(), a.GetFormalCharge(), a.GetAtomicNum(),
              ave_bond_length, ave_neighbor_wt]
        features[idx, :len(a_feats)] = np.array(a_feats)
```

Figure 11: Creating atom level features

8. Now store all the graph distances and Euclidean distances into the dataframe and export the dataframe to a csv file to be used later in the modelling process. Also export the atoms and bond level features created in previous steps (7 & 8) to a csv file. (refer to figures 12, 13)

```
## Functions to store distance matrices
def store_graph_distances(graph_dist_matrices):
    graph_dist_df = pd.concat(graph_dist_matrices)
    graph_dist_df.reset_index(drop=True, inplace=True)
    graph_dist_df.replace(1e8, 10, inplace=True) # fix for one erroneous atom
    graph_dist_df = graph_dist_df.astype(int)
    graph_dist_df.to_csv(C.PROC_DATA_PATH + 'graph_dist_df.csv')

def store_eucl_distances(dist_matrices, atom_df):
    dist_df = pd.DataFrame(np.concatenate(
        [np.pad(dm, [(0,0), (0, C.MAX_N_ATOMS-dm.shape[1])], mode='constant')
         for dm in dist_matrices.values()]
    ))
    dist_df['molecule_id'] = atom_df['molecule_id']
    dist_df.to_csv(C.PROC_DATA_PATH + 'dist_df.csv')
```

Figure 12: Storing the distances into a dataframe and exporting the dataframe to a csv file

```
# Create a PROC_DATA folder along with other directories as also described in the constants.py file in the Custom_modules folder
def store_atom_and_bond_features(atom_df, bond_df):
    atom_df.to_csv(C.PROC_DATA_PATH + 'atom_df.csv')
    bond_df.to_csv(C.PROC_DATA_PATH + 'bond_df.csv')
```

Figure 13: Exporting the bond and atom features dataframe to a csv file

9. Read the csv files created in the earlier steps using pandas to make sure all the features have been stored properly into the datframes. Merge all the features dataset into a single dataframe and concatenate train and test into a single dataframe *all_df*. Create the features using the functions defined above by calling them and giving the appropriate dataframes in the arguments (refer to figure 14).

```
if __name__ == '__main__':
    # import data
    train_df = pd.read_csv(C.RAW_DATA_PATH + 'train.csv', index_col=0)
    test_df = pd.read_csv(C.RAW_DATA_PATH + 'test.csv', index_col=0)
    structures_df = pd.read_csv(C.RAW_DATA_PATH + 'structures.csv')

    # concatenate train and test into one dataframe
    all_df = pd.concat((train_df, test_df), sort=True)
    if 'id' in all_df.columns: all_df.drop(columns='id', inplace=True)
    _clear_memory(['train_df', 'test_df'])

    # create molecules
    mols, mol_ids, mol_feats, xyzs, dist_matrices, graph_dist_matrices = \
        get_molecules()

    # create and store features
    all_df = add_sc_features(
        all_df, structures_df, mol_feats, xyzs, dist_matrices, mol_ids)
    store_train_and_test(all_df)
    _clear_memory(['all_df'])

    atom_df, bond_df = get_atom_and_bond_features(mols, mol_ids, dist_matrices)
    store_atom_and_bond_features(atom_df, bond_df)

    store_graph_distances(graph_dist_matrices)
    store_eucl_distances(dist_matrices, atom_df) # only used for MPNN model

    structures_df = process_and_store_structures(structures_df, mol_ids)
    _, _ = get_all_cosine_angles(bond_df, structures_df, mol_ids, store=True)
```

Figure 14: Creating and storing the features into dataframes

10. The last step in the pre-processing is to create a validation subset using K-fold cross validation technique. We create a function which create the K folds for molecules using the molecules' ID. The validation and training ids are then exported to respective csv files (refer to figure 15).

```
train_df = pd.read_csv(C.PROC_DATA_PATH + 'train_proc_df.csv', index_col=0)
mol_ids = train_df['molecule_id'].unique()

folds = KFold(C.N_FOLDS, shuffle=True, random_state=100).split(mol_ids)
folds = [(pd.Series(mol_ids[f[0]]), pd.Series(mol_ids[f[1]])) for f in folds]
train_idxes = pd.concat([f[0] for f in folds], axis=1)#.dropna().astype(int)
val_idxes = pd.concat([f[1] for f in folds], axis=1)#.dropna().astype(int)

train_idxes.to_csv(f'{C.PROC_DATA_PATH}train_idxes_{C.N_FOLDS}_fold_cv.csv')
val_idxes.to_csv(f'{C.PROC_DATA_PATH}val_idxes_{C.N_FOLDS}_fold_cv.csv')
```

Figure 15: K-fold cross validation on the data

3 Designing Utility Functions

In this section, we show how to create some utility functions. The first step is to import the necessary libraries (refer to figure 16).

```
import random
import copy
import numpy as np
import pandas as pd
import torch
from time import strftime, localtime

import constants as C
```

Figure 16: Importing the required libraries for designing utility functions

1. First create a random seed setter function for pytorch to increase the functionality of the code. This function also checks if the pytorch package has all the necessary CUDA drivers available for execution or not (refer to figure 17).

```
def set_seed(seed=100):
    """Set the seed for all relevant RNGs."""
    # python RNG
    random.seed(seed)

    # pytorch RNGs
    torch.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
    if torch.cuda.is_available(): torch.cuda.manual_seed_all(seed)

    # numpy RNG
    np.random.seed(seed)
```

Figure 17: Seed Setter Function

- Now, we define a *scatter_add* function which adds all the elements from the *src* dataframe into *out* dataframe specified by the id of the molecule. The index 'idx' nly has to match the size of 'src' in dimension 'dim' (refer to figure 18)

```
def scatter_add(src, idx, num=None, dim=0, out=None):
    """Adds all elements from 'src' into 'out' at the positions specified by
    'idx'.
```

Args:

- src: The index 'idx' only has to match the size of 'src' in dimension 'dim'. If 'out' is None it is initialized to zeros of size 'num' along 'dim' and of equal dimension to 'src' at all other dimensions.

```
    """
    if not num: num = idx.max().item() + 1
    sz, expanded_idx_sz = src.size(), src.size()
    sz = sz[:dim] + torch.Size((num,)) + sz[(dim+1):]
    expanded_idx = idx.unsqueeze(-1).expand(expanded_idx_sz)
    if out is None: out = torch.zeros(sz, dtype=src.dtype, device=src.device)
    return out.scatter_add(dim, expanded_idx, src)

def scatter_mean(src, idx, num=None, dim=0, out=None):
    return (scatter_add(src, idx, num, dim, out)
            / scatter_add(torch.ones_like(src), idx, num, dim).clamp(1.0))
```

Figure 18: Defining the *scatter_add* function

- Now, we design the functions to store the predictions obtained into csv files, which can used later for evaluating thee results and model (refer to figure 19)

```
def store_submit(predictions, name, print_head=False):
    if not isinstance(predictions, pd.DataFrame):
        submit = pd.read_csv(C.RAW_DATA_PATH + 'Sample_Predictions.csv')
        submit['scalar_coupling_constant'] = predictions
    else:
        submit = predictions
    submit.to_csv(f'{C.SUB_PATH}{name}-Predictions.csv', index=False)
    if print_head: print(submit.head())

#Storing the oof prediction
def store_oof(predictions, name, print_head=False):
    if not isinstance(predictions, pd.DataFrame):
        oof = pd.DataFrame(predictions, columns=['scalar_coupling_constants'])
    else:
        oof = predictions
    oof.to_csv(f'{C.OOF_PATH}{name}-oof.csv')
    if print_head: print(oof.head())
```

Figure 19: Functions for storing the results into a csv file

- Scale the features between the values 0 and 1. This is done by subtracting mean of the values of the features from the original values and then dividing by the standard deviation (refer to figure 20).

```

# Scaling the different features
def scale_features(df, features, train_mol_ids=None, means=None, stds=None,
                  return_mean_and_std=False):
    if ((df[features].mean().abs()>0.1).any()
        or ((df[features].std()-1.0).abs()>0.1).any()):
        if train_mol_ids is not None:
            idx = df['molecule_id'].isin(train_mol_ids)
            means = df.loc[idx, features].mean()
            stds = df.loc[idx, features].std()
        else:
            assert means is not None
            assert stds is not None
        df[features] = (df[features] - means) / stds
    if return_mean_and_std: return df, means, stds
    else: return df

```

Figure 20: Function for scaling the features

4 Layers Initialization

This section deals with the designing of the general layout of the layers that will be used later in the next section for designing the proposed Message Passing Molecular Transformer (MPMT) architecture. The first step as usual is to import the necessary libraries.

```

import torch.nn as nn
from layernorm import LayerNorm

```

1. First, define the `layernorm()` class to initialize the parameters according to the default initialization of batch normalization layers in the pytorch package. We also define the hidden layer function which appends the layer normalization and dropout after each one-dimensional batch normalization layer (refer to figure 21).

```

class LayerNorm(nn.LayerNorm):
    """Class overriding pytorch default layernorm initialization."""
    def reset_parameters(self):
        if self.elementwise_affine:
            nn.init.uniform_(self.weight)
            nn.init.zeros_(self.bias)

# Defining the hidden layers in the model
def hidden_layer(d_in, d_out, batch_norm, dropout, layer_norm=False, act=None):
    layers = []
    layers.append(nn.Linear(d_in, d_out))
    if act: layers.append(act)
    if batch_norm: layers.append(nn.BatchNorm1d(d_out))
    if layer_norm: layers.append(LayerNorm(d_out))
    if dropout != 0: layers.append(nn.Dropout(dropout))
    return layers

```

Figure 21: Defining default layer normalization procedure

2. Next, define the fully connected neural net class. This serves as the general purpose neural network with fully connected layers. It stacks together the batch normalization, dropout and hidden layers through the pre-defined *Sequential()* function in the pytorch library.

```

# Defining general purpose fully connected layer
class FullyConnectedNet(nn.Module):
    """General purpose neural network class with fully connected layers."""
    def __init__(self, d_input, d_output=None, layers=[], act=nn.ReLU(True),
                  dropout=[], batch_norm=False, out_act=None, final_bn=False,
                  layer_norm=False, final_ln=False):
        super().__init__()
        sizes = [d_input] + layers
        if d_output:
            sizes += [d_output]
            dropout += [0.0]
        layers_ = []
        for i, (d_in, d_out, dr) in enumerate(zip(sizes[:-1], sizes[1:],
                                                  dropout)):
            act_ = act if i < len(layers) else out_act
            batch_norm_ = batch_norm if i < len(layers) else final_bn
            layer_norm_ = layer_norm if i < len(layers) else final_ln
            layers_ += hidden_layer(
                d_in, d_out, batch_norm_, dr, layer_norm_, act_)
        self.layers = nn.Sequential(*layers_)

    def forward(self, x):
        return self.layers(x)

```

Figure 22: General class neural network with fully connected layers

5 Model Designing

This section is extremely crucial and at the heart of the implementation of the proposed MPMT architecture. It discusses the steps involved in the designing of the entire MPMT architecture from scratch using pytorch. First, the following libraries have to be imported:

```

import math
import copy
import torch
import torch.nn as nn
import torch.nn.functional as F
from fcnet import FullyConnectedNet, hidden_layer
from scatter import scatter_mean
from layernorm import LayerNorm

```

1. First we design a clone function for producing the identical layers. Then we define the *sublayerConnection()* function which defines a residual connection followed by a layer normalization procedure. It also applies the residual connection to any sublayer with the same size (refer to figure 23

```

def clones(module, N):
    """Produce N identical layers."""
    return torch.nn.ModuleList([copy.deepcopy(module) for _ in range(N)])

class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """
    def __init__(self, size, dropout):
        super().__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        """Apply residual connection to any sublayer with the same size."""
        return x + self.dropout(sublayer(self.norm(x)))

def _gather_nodes(x, idx, sz_last_dim):
    idx = idx.unsqueeze(-1).expand(-1, -1, sz_last_dim)
    return x.gather(1, idx)

```

Figure 23: Defining clone() and sublayerconnection() functions

- Now, define the edge network message passing procedure (same as proposed by Gilmer et al. (2017)). It adds additional cosine angle based attention mechanism over incoming messages (refer to figures 24 and 25). There is an *add_message()* function which computes and updates the message for an atom using the following mechanism.

$$M_i = \sum_j [attn_{ij} A_{ij} x_j]$$

```

class ENNMessage(nn.Module):
    """
    The edge network message passing function from the MPNN paper. Optionally
    adds and additional cosine angle based attention mechanism over incoming
    messages.
    """
    PAD_VAL = -999

    def __init__(self, d_model, d_edge, kernel_sz, enn_args={}, ann_args=None):
        super().__init__()
        assert kernel_sz <= d_model
        self.d_model, self.kernel_sz = d_model, kernel_sz
        self.enn = FullyConnectedNet(d_edge, d_model*kernel_sz, **enn_args)
        if ann_args: self.ann = FullyConnectedNet(1, d_model, **ann_args)
        else: self.ann = None

    def forward(self, x, edges, pairs_idx, angles=None, angles_idx=None, t=0):
        """Note that edges and pairs_idx raw inputs are for a unidirectional
        graph. They are expanded to allow bidirectional message passing."""
        if t==0:
            self.set_a_mat(edges)
            if self.ann: self.set_attn(angles)
            # concat reversed pairs_idx for bidirectional message passing
            self.pairs_idx = torch.cat([pairs_idx, pairs_idx[:, :, [1, 0]]], dim=1)
            return self.add_message(torch.zeros_like(x), x, angles_idx)

    def set_a_mat(self, edges):
        n_edges = edges.size(1)
        a_vect = self.enn(edges)
        a_vect = a_vect / (self.kernel_sz ** .5) # rescale
        mask = edges[:, :, 0, None].expand(a_vect.size()) == self.PAD_VAL
        a_vect = a_vect.masked_fill(mask, 0.0)
        self.a_mat = a_vect.view(-1, n_edges, self.d_model, self.kernel_sz)
        # concat a_mats for bidirectional message passing
        self.a_mat = torch.cat([self.a_mat, self.a_mat], dim=1)

```

Figure 24: Edge Network MP layer

```

def set_attn(self, angles):
    angles = angles.unsqueeze(-1)
    self.attn = self.ann(angles)
    mask = angles.expand(self.attn.size())==self.PAD_VAL
    self.attn = self.attn.masked_fill(mask, 0.0)

def add_message(self, m, x, angles_idx=None):
    """Add message for atom_{i}: m_{i} += sum_{j}[attn_{ij} A_{ij}x_{j}]."""
    # select the 'x_{j}' feeding into the 'm_{i}'
    x_in = _gather_nodes(x, self.pairs_idx[:, :, 1], self.d_model)

    # do the matrix multiplication 'A_{ij}x_{j}'
    if self.kernel_sz==self.d_model: # full matrix multiplication
        ax = (x_in.unsqueeze(-2) @ self.a_mat).squeeze(-2)
    else: # do a convolution
        x_padded = F.pad(x_in, self.n_pad)
        x_unfolded = x_padded.unfold(-1, self.kernel_sz, 1)
        ax = (x_unfolded * self.a_mat).sum(-1)

    # apply attention
    if self.ann:
        n_pairs = self.pairs_idx.size(1)
        # average all attn(angle_{ijk}) per edge_{ij}.
        # i.e.: attn_{ij} = sum_{k}[attn(angle_{ijk})] / n_angles_{ij}
        ave_att = scatter_mean(self.attn, angles_idx, num=n_pairs, dim=1,
                               out=torch.ones_like(ax))
        ax = ave_att * ax

```

Figure 25: Edge Network MP layer (cont.)

- Now design the first multi-head attention layer which computes the attention through Euclidean Distances between the nodes (refer to figure 26).

```

class MultiHeadedDistAttention(nn.Module):
    """Generalizes the euclidean and graph distance based attention layers."""
    def __init__(self, h, d_model):
        super().__init__()
        self.d_model, self.d_k, self.h = d_model, d_model // h, h
        self.attn = None
        self.linears = clones(nn.Linear(d_model, d_model), 2)

    def forward(self, dists, x, mask):
        batch_size = x.size(0)
        x = self.linears[0](x).view(batch_size, -1, self.h, self.d_k)
        x, self.attn = self.apply_attn(dists, x, mask)
        x = x.view(batch_size, -1, self.h * self.d_k)
        return self.linears[-1](x)

    def apply_attn(self, dists, x, mask):
        attn = self.create_raw_attn(dists, mask)
        attn = attn.transpose(-2, -1).transpose(1, 2)
        x = x.transpose(1, 2)
        x = torch.matmul(attn, x)
        x = x.transpose(1, 2).contiguous()
        return x, attn

    def create_raw_attn(self, dists, mask):
        pass

```

Figure 26: First multi-head attention layer

- Design the second multi-head attention layers which computes the attention by utilizing the embedding of the distance matrix of the graph (refer to figure 27).


```

class MultiHeadedGraphDistAttention(MultiHeadedDistAttention):
    """Attention based on an embedding of the graph distance matrix."""
    MAX_GRAPH_DIST = 10
    def __init__(self, h, d_model):
        super().__init__(h, d_model)
        self.embedding = nn.Embedding(self.MAX_GRAPH_DIST+1, h)

    def create_raw_attn(self, dists, mask):
        emb_dists = self.embedding(dists)
        mask = mask.unsqueeze(-1).expand(emb_dists.size())
        emb_dists = emb_dists.masked_fill(mask==0, -1e9)
        return F.softmax(emb_dists, dim=-2).masked_fill(mask==0, 0)

```

Figure 27: Second multi-head attention layer

- Next, define the third attention layer which takes the parametrized euclidean distance matrix of the molecule as input.

```

class MultiHeadedEuclDistAttention(MultiHeadedDistAttention):
    """Attention based on a parameterized normal pdf taking a molecule's
    euclidean distance matrix as input."""
    def __init__(self, h, d_model):
        super().__init__(h, d_model)
        self.log_prec = nn.Parameter(torch.Tensor(1, 1, 1, h))
        self.locs = nn.Parameter(torch.Tensor(1, 1, 1, h))
        nn.init.normal_(self.log_prec, mean=0.0, std=0.1)
        nn.init.normal_(self.locs, mean=0.0, std=1.0)

    def create_raw_attn(self, dists, mask):
        dists = dists.unsqueeze(-1).expand(-1, -1, -1, self.h)
        z = torch.exp(self.log_prec) * (dists - self.locs)
        pdf = torch.exp(-0.5 * z ** 2)
        return pdf / pdf.sum(dim=-2, keepdim=True).clamp(1e-9)

```

Figure 28: Third Attention Layer

- Define the multi-headed self attention class exactly as in the transformer paper (Vaswani et al. (2017)) (refer to figure 29).

```

class MultiHeadedSelfAttention(nn.Module):
    """Applies self-attention as described in the Transformer paper."""
    def __init__(self, h, d_model, dropout=0.1):
        super().__init__()
        self.d_model, self.d_k, self.h = d_model, d_model // h, h
        self.attn = None
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.dropout = nn.Dropout(p=dropout) if dropout > 0.0 else None

    def forward(self, x, mask):
        # Same mask applied to all h heads.
        mask = mask.unsqueeze(1)
        batch_size = x.size(0)

        # 1) Do all the linear projections in batch from d_model => h x d_k
        query, key, value = [
            l(x).view(batch_size, -1, self.h, self.d_k).transpose(1, 2)
            for l in self.linears[:3]
        ]

        # 2) Apply attention on all the projected vectors in batch.
        x, self.attn = attention(query, key, value, mask, self.dropout)

        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous()
        x = x.view(batch_size, -1, self.d_model)
        return self.linears[-1](x)

```

Figure 29: Defining the multi-head self attention class similar to transformer architecture

7. Now, stack the three attention layers and the point-wise feed forward neural network (refer to figure 30).

```
class AttendingLayer(nn.Module):
    """Stacks the three attention layers and the pointwise feedforward net."""
    def __init__(self, size, eucl_dist_attn, graph_dist_attn, self_attn, ff, dropout):
        super().__init__()
        self.eucl_dist_attn = eucl_dist_attn
        self.graph_dist_attn = graph_dist_attn
        self.self_attn = self_attn
        self.ff = ff
        self.subconns = clones(SublayerConnection(size, dropout), 4)
        self.size = size

    def forward(self, x, eucl_dists, graph_dists, mask):
        eucl_dist_sub = lambda x: self.eucl_dist_attn(eucl_dists, x, mask)
        x = self.subconns[0](x, eucl_dist_sub)
        graph_dist_sub = lambda x: self.graph_dist_attn(graph_dists, x, mask)
        x = self.subconns[1](x, graph_dist_sub)
        self_sub = lambda x: self.self_attn(x, mask)
        x = self.subconns[2](x, self_sub)
        return self.subconns[3](x, self.ff)
```

Figure 30: Stacking the three attention layers

8. Define the *MessagePassingLayer()* class which stacks the bond and scalar coupling pair MP layers together (refer to figure 31)

```
class MessagePassingLayer(nn.Module):
    """Stacks the bond and scalar coupling pair message passing layers."""
    def __init__(self, size, bond_mess, sc_mess, dropout, N):
        super().__init__()
        self.bond_mess = bond_mess
        self.sc_mess = sc_mess
        self.linears = clones(nn.Linear(size, size), 2*N)
        self.subconns = clones(SublayerConnection(size, dropout), 2*N)

    def forward(self, x, bond_x, sc_pair_x, angles, mask, bond_idx, sc_idx, angles_idx, t=0):
        bond_sub = lambda x: self.linears[2*t](
            self.bond_mess(x, bond_x, bond_idx, angles, angles_idx, t))
        x = self.subconns[2*t](x, bond_sub)
        sc_sub = lambda x: self.linears[(2*t)+1](
            self.sc_mess(x, sc_pair_x, sc_idx, t=t))
        return self.subconns[(2*t)+1](x, sc_sub)
```

Figure 31: Message Passing Layer

9. Define the *Encoder* class which stacks the N attention and one message passing layers together. The *forward()* function passes and masks the input through each encoder block in turn (refer to figure 32).

```
class Encoder(nn.Module):
    """Encoder stacks N attention layers and one message passing layer."""
    def __init__(self, mess_pass_layer, attn_layer, N):
        super().__init__()
        self.mess_pass_layer = mess_pass_layer
        self.attn_layers = clones(attn_layer, N)
        self.norm = LayerNorm(attn_layer.size)

    def forward(self, x, bond_x, sc_pair_x, eucl_dists, graph_dists, angles, mask, bond_idx, sc_idx, angles_idx):
        """Pass the inputs (and mask) through each block in turn. Note that for each block the same message passing layer is used."""
        for t, attn_layer in enumerate(self.attn_layers):
            x = self.mess_pass_layer(x, bond_x, sc_pair_x, angles, mask, bond_idx, sc_idx, angles_idx, t)
            x = attn_layer(x, eucl_dists, graph_dists, mask)
        return self.norm(x)
```

Figure 32: Encoder class for stacking the MPNN and transformer elements together

- Now define the final feed forward neural network used for calculating the individual scalar coupling contributions from each of the four terms and the final prediction of SCC using these four values (refer to figure 33).

```
def create_contrib_head(d_in, d_ff, act, dropout=0.0, layer_norm=True):
    layers = hidden_layer(d_in, d_ff, False, dropout, layer_norm, act)
    layers += hidden_layer(d_ff, 1, False, 0.0) # output layer
    return nn.Sequential(*layers)

class ContribsNet(nn.Module):
    """The feed-forward net used for the sc contribution and final sc constant
    predictions."""
    N_CONTRIBS = 5
    CONTRIB_SCALES = [1, 250, 45, 35, 500] # scales used to make the 5 predictions of similar magnitude

    def __init__(self, d_in, d_ff, vec_in, act, dropout=0.0, layer_norm=True):
        super().__init__()
        contrib_head = create_contrib_head(d_in, d_ff, act, dropout, layer_norm)
        self.blocks = clones(contrib_head, self.N_CONTRIBS)

    def forward(self, x):
        ys = torch.cat(
            [b(x)/s for b,s in zip(self.blocks, self.CONTRIB_SCALES)], dim=-1)
        return torch.cat([ys[:,-1], ys.sum(dim=-1, keepdim=True)], dim=-1)
```

Figure 33: Defining the feed-forward NN

- Join the scalar coupling type specific residual block with the scalar coupling contribution block defined in the previous step (refer to figure 34)

```
class MyCustomHead(nn.Module):
    """Joins the sc type specific residual block with the sc contribution
    feed-forward net."""
    PAD_VAL = -999
    N_TYPES = 8

    def __init__(self, d_input, d_ff, d_ff_contribs, pre_layers=[],
                 post_layers=[], act=nn.ReLU(True), dropout=3*[0.], norm=False):
        super().__init__()
        fc_pre = hidden_layer(d_input, d_ff, False, dropout[0], norm, act)
        self.preproc = nn.Sequential(*fc_pre)
        fc_type = hidden_layer(d_ff, d_input, False, dropout[1], norm, act)
        self.types_net = clones(nn.Sequential(*fc_type), self.N_TYPES)
        self.contribs_net = ContribsNet(
            d_input, d_ff_contribs, d_ff, act, dropout[2], layer_norm=norm)

    def forward(self, x, sc_types):
        # stack inputs with a .view for easier processing
        x, sc_types = x.view(-1, x.size(-1)), sc_types.view(-1)
        mask = sc_types != self.PAD_VAL
        x, sc_types = x[mask], sc_types[mask]

        x_ = self.preproc(x)
        x_types = torch.zeros_like(x)
        for i in range(self.N_TYPES):
            t_idx = sc_types==i
            if torch.any(t_idx): x_types[t_idx] = self.types_net[i](x_[t_idx])
            else: x_types = x_types + 0.0 * self.types_net[i](x_) # fake call (only necessary for
        x = x + x_types
        return self.contribs_net(x)
```

Figure 34: Joining residual block to type specific block

- Build the final MPMT architecture by stacking all the layers and blocks together (refer to figure 35)

```

class Transformer(nn.Module):
    """Molecule transformer with message passing."""
    def __init__(self, d_atom, d_bond, d_sc_pair, d_sc_mol, N=6, d_model=512,
                  d_ff=2048, d_ff_contrib=128, h=8, dropout=0.1, kernel_sz=128,
                  enn_args={}, ann_args={}):
        super().__init__()
        assert d_model % h == 0
        self.d_model = d_model
        c = copy.deepcopy
        bond_mess = ENNMessage(d_model, d_bond, kernel_sz, enn_args, ann_args)
        sc_mess = ENNMessage(d_model, d_sc_pair, kernel_sz, enn_args)
        eucl_dist_attn = MultiHeadedEuclDistAttention(h, d_model)
        graph_dist_attn = MultiHeadedGraphDistAttention(h, d_model)
        self_attn = MultiHeadedSelfAttention(h, d_model, dropout)
        ff = FullyConnectedNet(d_model, d_model, [d_ff], dropout=[dropout])

        message_passing_layer = MessagePassingLayer(
            d_model, bond_mess, sc_mess, dropout, N)
        attending_layer = AttendingLayer(
            d_model, c(eucl_dist_attn), c(graph_dist_attn), c(self_attn), c(ff),
            dropout
        )

        self.projection = nn.Linear(d_atom, d_model)
        self.encoder = Encoder(message_passing_layer, attending_layer, N)
        self.write_head = MyCustomHead(
            2 * d_model + d_sc_mol, d_ff, d_ff_contrib, norm=True)

    def forward(self, atom_x, bond_x, sc_pair_x, sc_mol_x, eucl_dists,
                graph_dists, angles, mask, bond_idx, sc_idx, angles_idx,
                sc_types):
        x = self.encoder(
            self.projection(atom_x), bond_x, sc_pair_x, eucl_dists, graph_dists,
            angles, mask, bond_idx, sc_idx, angles_idx
        )

```

Figure 35: Stacking all the layers to build MPMT’s final architecture

6 Model Training

The model training is achieved through the fastai library. First we import the necessary fastai utilities and packages necessary through the training (figure 36). The training code is similar to the training tutorials with pytorch provided on the fastai webpage.

```

import argparse
import pandas as pd
import numpy as np
from functools import partial

import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from fastai.callbacks import SaveModelCallback
from fastai.basic_data import DataBunch, DeviceDataLoader, DatasetType
from fastai.train import *
from fastai.distributed import *

from moldataset import MoleculeDataset, collate_parallel_fn
from model import Transformer
from utils import scale_features, set_seed, store_submit, store_oof
from callbacks import GradientClipping, GroupMeanLogMAE
from losses_and_metrics import rmse, mae, contribs_rmse_loss
import constants as C

from fastai.basic_train import Learner, LearnerCallback, Callback, add_metrics
from fastai.callback import annealing_cos
from fastai.callbacks.general_sched import TrainingPhase, GeneralScheduler
from losses_and_metrics import group_mean_log_mae, reshape_targs

```

Figure 36: Importing packages for model training through fastai

1. Parse the arguments like batch size, no. of epochs, learning rate, etc. through the argument parser function in pytorch (refer to figure 37).

```
# parse arguments
parser = argparse.ArgumentParser()
parser.add_argument('--batch_size', type=int, default=10)
parser.add_argument('--epochs', type=int, default=100)
parser.add_argument('--lr', type=float, default=4e-5, help='learning rate')
parser.add_argument('--d_model', type=int, default=650,
                    help='dimension of node state vector')
parser.add_argument('--N', type=int, default=10,
                    help='number of encoding layers')
parser.add_argument('--h', type=int, default=10,
                    help='number of attention heads')
parser.add_argument('--wd', type=float, default=1e-2, help='weight decay')
parser.add_argument('--dropout', type=float, default=0.0)
parser.add_argument('--start_epoch', type=int, default=0)
parser.add_argument('--fold_id', type=int, default=1)
parser.add_argument('--version', type=int, default=1)
parser.add_argument('--local_rank', type=int)
args = parser.parse_args()
```

Figure 37: Parsing the model parameters as arguments

2. Check if the distributed training functionality is available and set the model description (refer to figure 38).

```
# check if distributed training is possible and set model description
distributed_train = torch.cuda.device_count() > 1
model_str = f'mol_transformer_v{args.version}_fold{args.fold_id}'
```

Figure 38: Checking for the distributed training on GPUs

3. Now read all the csv files that were generated from the pre-processing stage and store them in appropriate pandas dataframes. Also scale the features using the *SC.FEATS_TO_SCALE()* function in the *constants.py* file/module (refer to figure 39).

```
# importing the data for Training phase
# These csv files are created after running the 01_Preprocessing(MPMT) notebook first.
train_df = pd.read_csv(C.PROC_DATA_PATH+'train_proc_df.csv', index_col=0)
test_df = pd.read_csv(C.PROC_DATA_PATH+'test_proc_df.csv', index_col=0)
atom_df = pd.read_csv(C.PROC_DATA_PATH+'atom_df.csv', index_col=0)
bond_df = pd.read_csv(C.PROC_DATA_PATH+'bond_df.csv', index_col=0)
angle_in_df = pd.read_csv(C.PROC_DATA_PATH+'angle_in_df.csv', index_col=0)
angle_out_df = pd.read_csv(C.PROC_DATA_PATH+'angle_out_df.csv', index_col=0)
graph_dist_df = pd.read_csv(
    C.PROC_DATA_PATH+'graph_dist_df.csv', index_col=0, dtype=np.int32)
structures_df = pd.read_csv(
    C.PROC_DATA_PATH+'structures_proc_df.csv', index_col=0)

train_mol_ids = pd.read_csv(C.PROC_DATA_PATH+'train_idx_8_fold_cv.csv',
                           usecols=[0, args.fold_id], index_col=0
                           ).dropna().astype(int).iloc[:,0]
val_mol_ids = pd.read_csv(C.PROC_DATA_PATH+'val_idx_8_fold_cv.csv',
                           usecols=[0, args.fold_id], index_col=0
                           ).dropna().astype(int).iloc[:,0]
test_mol_ids = pd.Series(test_df['molecule_id'].unique())

# scale features
train_df, sc_feat_means, sc_feat_stds = scale_features(
    train_df, C.SC_FEATS_TO_SCALE, train_mol_ids, return_mean_and_std=True)
test_df = scale_features(
    test_df, C.SC_FEATS_TO_SCALE, means=sc_feat_means, stds=sc_feat_stds)
atom_df = scale_features(atom_df, C.ATOM_FEATS_TO_SCALE, train_mol_ids)
bond_df = scale_features(bond_df, C.BOND_FEATS_TO_SCALE, train_mol_ids)
```

Figure 39: Reading the csv files from pre-processing stage

4. Set up the fastai dataset objects and databunch by using the test, train and validation dataframes from the previous step (refer to figure 40).

```

train_ds = MoleculeDataset(
    train_mol_ids, gb_mol_sc, gb_mol_atom, gb_mol_bond, gb_mol_struct,
    gb_mol_angle_in, gb_mol_angle_out, gb_mol_graph_dist
)
val_ds = MoleculeDataset(
    val_mol_ids, gb_mol_sc, gb_mol_atom, gb_mol_bond, gb_mol_struct,
    gb_mol_angle_in, gb_mol_angle_out, gb_mol_graph_dist
)
test_ds = MoleculeDataset(
    test_mol_ids, test_gb_mol_sc, gb_mol_atom, gb_mol_bond, gb_mol_struct,
    gb_mol_angle_in, gb_mol_angle_out, gb_mol_graph_dist
)

train_dl = DataLoader(train_ds, args.batch_size, shuffle=True, num_workers=8)
val_dl = DataLoader(val_ds, args.batch_size, num_workers=8)
test_dl = DeviceDataLoader.create(
    test_ds, args.batch_size, num_workers=8,
    collate_fn=partial(collate_parallel_fn, test=True)
)

db = DataBunch(train_dl, val_dl, collate_fn=collate_parallel_fn)
db.test_dl = test_dl

```

Figure 40: Creating fastai dataset objects and databunch

5. Set up the model using the transformer class defined in previous section and the no. of features defined in the *constants.py* file (refer to figure 41).

```

# set up model
set_seed(100)
d_model = args.d_model
enn_args = dict(layers=3*[d_model], dropout=3*[0.0], layer_norm=True)
ann_args = dict(layers=1*[d_model], dropout=1*[0.0], layer_norm=True,
                out_act=nn.Tanh())
model = Transformer(
    C.N_ATOM_FEATURES, C.N_BOND_FEATURES, C.N_SC_EDGE_FEATURES,
    C.N_SC_MOL_FEATURES, N=args.N, d_model=d_model, d_ff=d_model*4,
    d_ff_contrib=d_model//4, h=args.h, dropout=args.dropout,
    kernel_sz=min(128, d_model), enn_args=enn_args, ann_args=ann_args
)

# Distributed training on CUDA using pytorch

# initialize distributed
if distributed_train:
    torch.cuda.set_device(args.local_rank)
    torch.distributed.init_process_group(backend='nccl', init_method='env://')

```

Figure 41: Setting up the MPMT model

6. Define the gradient clipping and the group log MAE callback functions for report the results during the training. The gradient clipping function is mainly used to avoid the problem of exploding gradients during training the deep neural networks (refer to figure 42).

```

# Creating callback classes and functions for training on Fastai
class GradientClipping(LearnerCallback):
    """Gradient clipping during training after 'start_it' number of steps."""
    def __init__(self, learn:Learner, clip:float = 0., start_it:int = 100):
        super().__init__(learn)
        self.clip, self.start_it = clip, start_it

    def on_backward_end(self, iteration, **kwargs):
        """Clip the gradient before the optimizer step."""
        if self.clip and (iteration > self.start_it):
            torch.nn.utils.clip_grad_norm_(
                self.learn.model.parameters(), self.clip)

class GroupMeanLogMAE(Callback):
    """Callback to report the group mean log MAE during training. Also supports
    correct computation of the metric during snapshot ensembling."""
    _order = -20 # Needs to run before the recorder

    def __init__(self, learn, snapshot_ensemble=False, **kwargs):
        self.learn = learn
        self.snapshot_ensemble = snapshot_ensemble

    def on_train_begin(self, **kwargs):
        metric_names = ['group_mean_log_mae']
        if self.snapshot_ensemble: metric_names += ['group_mean_log_mae_es']
        self.learn.recorder.add_metric_names(metric_names)
        if self.snapshot_ensemble: self.val_preds = []

    def on_epoch_begin(self, **kwargs):
        self.sc_types, self.output, self.target = [], [], []

```

Figure 42: Defining callback functions for training

7. Train the model using the callback functions, and using group mean log MAE as the evaluation metric for each epoch. Obtain the predictions and store them into a csv file format (refer to figure 43).

```

# train model
callback_fns = [
    partial(GradientClipping, clip=10), GroupMeanLogMAE,
    partial(SaveModelCallback, every='improvement', mode='min',
        monitor='group_mean_log_mae', name=model_str)
]
learn = Learner(db, model, metrics=[rmse, mae], callback_fns=callback_fns,
    wd=args.wd, loss_func=contribs_rmse_loss)
if args.start_epoch > 0:
    learn.load(model_str)
    torch.cuda.empty_cache()
if distributed_train: learn = learn.to_distributed(args.local_rank)

learn.fit_one_cycle(args.epochs, max_lr=args.lr, start_epoch=args.start_epoch)

# make predictions for the target variable i.e the scalar coupling constant
val_contrib_preds = learn.get_preds(DatasetType.Valid)
test_contrib_preds = learn.get_preds(DatasetType.Test)
val_preds = val_contrib_preds[0][:,-1].detach().numpy() * C.SC_STD + C.SC_MEAN
test_preds = test_contrib_preds[0][:,-1].detach().numpy() * C.SC_STD + C.SC_MEAN

# store results in csv files
store_submit(test_preds, model_str, print_head=True)
store_oof(val_preds, model_str, print_head=True)

```

Figure 43: Training the model and storing the results in the dataframes

7 Model Evaluation

The model evaluation is done through the *log MAE*, *RMSE* and *contrib_rmse()* demonstrated in figure 44 after importing all the necessary packages described below:

```

import numpy as np
import pandas as pd
import torch
import torch.nn.functional as F
import constants as C

```

```

def group_mean_log_mae(y_true, y_pred, types, sc_mean=0, sc_std=1):
    def proc(x):
        if isinstance(x, torch.Tensor): return x.cpu().numpy().ravel()
    y_true, y_pred, types = proc(y_true), proc(y_pred), proc(types)
    y_true = sc_mean + y_true * sc_std
    y_pred = sc_mean + y_pred * sc_std
    maes = pd.Series(y_true - y_pred).abs().groupby(types).mean()
    gmlmae = np.log(maes).mean()
    return gmlmae

def contribs_rmse_loss(preds, targs):
    """
    Returns the sum of RMSEs for each scalar coupling (sc) contribution and
    the sc constant in a batch.

    Args:
        - preds: tensor of shape (n_sc_batch, 5) containing predictions. Last
            column is the scalar coupling constant.
        - targs: tensor of shape (batch_size, max_n_sc_per_molecule, 5)
            containing true values. Last column is the scalar coupling constant.
    """
    targs = reshape_targs(targs)
    return torch.mean((preds - targs) ** 2, dim=0).sqrt().sum()

def rmse(preds, targs):
    targs = reshape_targs(targs)
    return torch.sqrt(F.mse_loss(preds[:, -1], targs[:, -1]))

def mae(preds, targs):
    targs = reshape_targs(targs)
    return torch.abs(preds[:, -1] - targs[:, -1]).mean()

```

Figure 44: Evaluation functions

References

- Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O. and Dahl, G. E. (2017). Neural message passing for quantum chemistry, *International conference on machine learning*, PMLR, pp. 1263–1272.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. and Polosukhin, I. (2017). Attention is all you need, *Advances in neural information processing systems*, pp. 5998–6008.