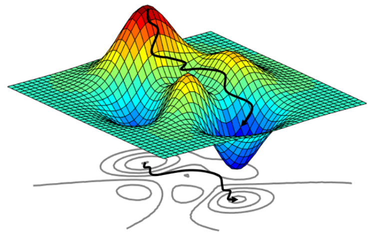
hyper-parameter optimization project report

Arash Sajjadi
Optimization in data science
Shahid Beheshti University
July 26, 2022



Keywords: Deep learning, Neural network, Hyper-parameter optimization, Bayesian optimization, Optuna search algorithm, Ray tune

ABSTRACT

The upcoming project is a hyper-parameter optimization of the neural network, which has been implemented on three datasets of Bbbp, HIV, and FreeSolv. Two datasets describe classification problems, and one dataset that I am working on is regression. Next, I will explain each data set. One of my most essential references during this project will be the ray library article. [1]

CONTENTS

Contents

PRE-INTRODUCTION

The data set that I am going to work on during this project is the molecular data set. Since my focus in this project is only on optimizing the hyper-parameters of a neural network, I will refrain from additional explanations about the nature of the data and introduce them briefly.

	Dataset	Task Type	Tasks	Compunds	Category
1	Bbbp	Classification	1	$2,039$	Physiology
2	HIV	Classification	1	$41,127$	Biophysics
3	FreeSolv	Regression	1	642	Pysical Chemistry

1 INTRODUCTION

In this section, I would like to put a summary table of the features and labels of the data sets I am dealing with. It is necessary to explain that some features represent a constant value that I have deleted in all datasets after calling in Python. Therefore, I have specified each record's algebraic dimension of the features vector in this table. It should be mentioned that I have partitioned the data set to the training set, validation, and testing in the same proportion.

	Number of features	The algebraic dimension of the features	Number of records	Target variable	Training set Validation set Test set
Bbbp	200	185	$2,039$	$\{0,1\}$	70%-20%-10%
HIV	200	192	$41,127$	$\{0,1\}$	70%-20%-10%
FreeSolv	200	161	642	$\mathbb{R}$	70%-20%-10%

Also, since I'm presenting all three projects in one report, I've highlighted all the code for the **Bbbp** dataset with a **light red** background, the **HIV** dataset with a **light green** background, and the **FreeSolv** dataset with a **light blue** background for more straightforward diagnosis.

1.1 Distribution of labels

Examining the distribution of labels is one of the first things we should do when dealing with any data set. I have also plotted this distribution for all three data sets. [1]

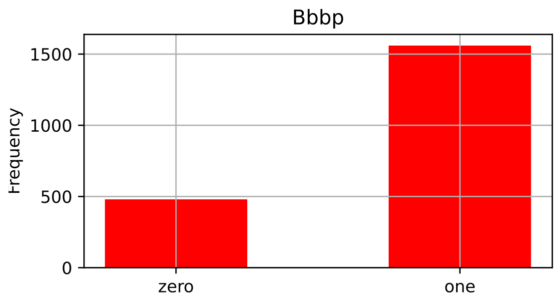


Figure 1: Distribution of Bbbp dataset labels

Since there is an imbalance between the class of one and zero labels in Bbbp's dataset (Fig 1), I will partition the data with the same proportion of distribution. [2]

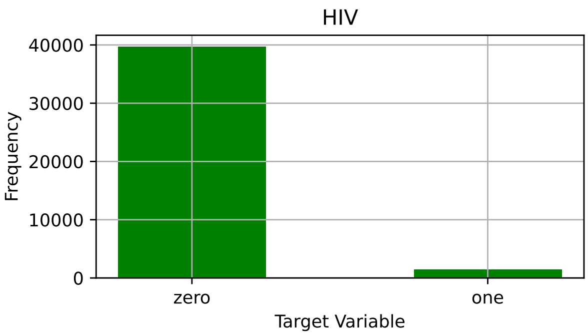


Figure 2: Distribution of HIV dataset labels

Since there is an imbalance between the class of one and zero labels in HIV's dataset (Fig 2), I will partition the data with the same proportion of distribution.

It is clear that this imbalance is significantly more apparent in the HIV dataset than in the Bbbp dataset.

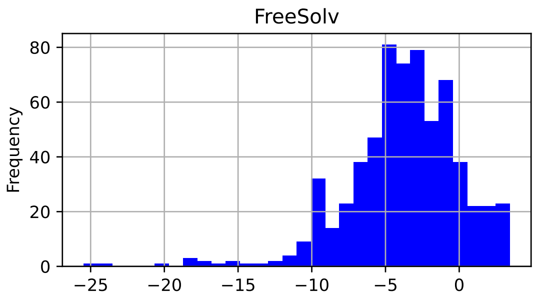


Figure 3: Distribution of FreeSolv dataset labels

The best way to interpret Figure 3 is to present a frequency table describing this graph. Therefore, I draw your attention to this table.

count	mean	std	min	25%	50%	75%	max
642	-3.803	3.847	-25.47	-5.727	-3.53	-1.215	3.43

1.2 Type of the features

Undoubtedly, all the features specify different properties of chemicals. It should be noted that all these data have been normalized by the CDF [3] scaler method. According to my research, the standard and min-max scaler methods are not applicable for these types of data sets. [2] Also, I should point out that none of the features represent categorical features (or at least I have access to their numerically encoded version).

2 PREPROCESSING

Data preprocessing includes Data partitioning, Feature selection, and Missing values subsections that prepare the data for final processing.

2.1 Data partitioning

I have chosen an evaluation method for the best model for these three projects, which I will explain in detail. For a better visual understanding, I drew Figure 4 to make it easier for me to explain what is going to happen. I will use a semi-cross-validation system to train the neural network, which needs to partition the data like Figure 4.

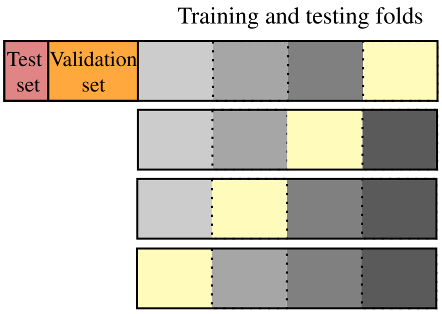


Figure 4: Data partitioning diagram in all three data sets

```
%%capture
try:
    import ray
except:
    %pip install ray
    import ray

%pip install ray
try:
    import optuna
except:
    %pip install optuna
    import optuna

try:
    from torchmetrics import ConfusionMatrix
except:
    %pip install torchmetrics
    from torchmetrics import ConfusionMatrix
import numpy as np
import torch
import torch.optim as optim
import torch.nn as nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, TensorDataset, Dataset
import torch.nn.functional as F
from ray import tune
import os
import ray
from ray.tune.schedulers import ASHAScheduler
import warnings
warnings.filterwarnings('ignore')
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import balanced_accuracy_score
import matplotlib as mpl
import matplotlib.pyplot as plt
import math
from sklearn.metrics import roc_auc_score
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import HistGradientBoostingRegressor
confmat = ConfusionMatrix(num_classes=2)

%%capture
try:
    from featurewiz import featurewiz
except:
    !pip install featurewiz==0.1.70
    from featurewiz import featurewiz

#read data
initial_targets=pd.read_csv("bbbp.csv")
initial_features=pd.read_csv("bbbp_global_cdf_rdkit.csv")

initial_features=initial_features.loc[:, (initial_features != initial_features.iloc[0]).any()]

shuffled_targets=initial_targets.sample(frac=1, random_state=1234).reset_index(drop=True)
shuffled_features=initial_features.sample(frac=1, random_state=1234).reset_index(drop=True)
#shuffling the data to randomize the sequence

X_train, X_test, y_train, y_test = train_test_split(shuffled_features,shuffled_targets["p_np"], test_size=0.1, random_state=1234, stratify=shuffled_targets["p_np"])
X_train, X_valid, y_train, y_valid = train_test_split(X_train,y_train,test_size=0.22, random_state=1234, stratify=y_train)
X_train12 ,X_train34 , y_train12, y_train34=train_test_split(X_train,y_train,test_size=0.5, random_state=1234,stratify=y_train)
X_train1 ,X_train2 , y_train1, y_train2=train_test_split(X_train12,y_train12,test_size=0.5, random_state=1234,stratify=y_train12)
X_train3 ,X_train4 , y_train3, y_train4=train_test_split(X_train34,y_train34,test_size=0.5, random_state=1234,stratify=y_train34)
del X_train12
del X_train34
del y_train12
del y_train34
k_fold_X_train=[X_train1 ,X_train2,X_train3 ,X_train4]
k_fold_y_train=[y_train1 ,y_train2,y_train3 ,y_train4]
del X_train1
del X_train2
del X_train3
del X_train4
del y_train1
del y_train2
del y_train3
del y_train4

Data=pd.concat([X_train, y_train], axis=1) #reshape data for featurewiz feature selection
target = ['p_np']
from featurewiz import featurewiz
feature_selection = featurewiz(Data, target, corr_limit=0.97, verbose=2,header=0, nrows=None, outputs="features")

X_train=X_train[feature_selection[0]]
X_valid=X_valid[feature_selection[0]]
X_test=X_test[feature_selection[0]]
y_train=pd.DataFrame(data=y_train)
y_valid=pd.DataFrame(data=y_valid)
```

```python
y_test=pd.DataFrame(data=y_test)

for i in np.arange(0,4):
  k_fold_X_train[i]=k_fold_X_train[i][feature_selection[0]]
  k_fold_y_train[i]=pd.DataFrame(data=k_fold_y_train[i])
```

Listing 1: Import libraries, data reading, train test splitting, adn feature selection

```python
import ray
import optuna
from torchmetrics import ConfusionMatrix
import numpy as np
import torch
import torch.optim as optim
import torch.nn as nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, TensorDataset, Dataset
import torch.nn.functional as F
from ray import tune
import os
import ray
from ray.tune.schedulers import ASHAScheduler
import warnings
warnings.filterwarnings('ignore')
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import balanced_accuracy_score
import matplotlib as mpl
import matplotlib.pyplot as plt
import math
from sklearn.metrics import roc_auc_score
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import HistGradientBoostingRegressor
confmat = ConfusionMatrix(num_classes=2)
from featurewiz import featurewiz
from sklearn.impute import KNNImputer
#read data
initial_targets=pd.read_csv("Downloads/Hiv dataset/hiv.csv")
initial_features=pd.read_csv("Downloads/Hiv dataset/hiv_global_cdf_rdkit.csv")

initial_features=initial_features.loc[:, (initial_features != initial_features.iloc[0]).any()]

shuffled_targets=initial_targets.sample(frac=1, random_state=1234).reset_index(drop=True).drop("smiles",axis=1)
shuffled_features=initial_features.sample(frac=1, random_state=1234).reset_index(drop=True)
X_train, X_test, y_train, y_test = train_test_split(shuffled_features,shuffled_targets,test_size=0.1, random_state=1234,stratify=shuffled_targets["HIV_active"])
X_train, X_valid, y_train, y_valid = train_test_split(X_train,y_train,test_size=0.22, random_state=1234, stratify=y_train["HIV_active"])

#feature selection
Data=pd.concat([X_train, y_train], axis=1)
target = ['HIV_active']
feature_selection = featurewiz(Data.dropna(), target, corr_limit=0.8, verbose=2,header=0, nrows=None, outputs="features")

#dealing with missings
X_train=X_train[feature_selection[0]]
X_valid=X_valid[feature_selection[0]]
X_test=X_test[feature_selection[0]]
y_train=pd.DataFrame(data=y_train)
y_valid=pd.DataFrame(data=y_valid)
y_test=pd.DataFrame(data=y_test)


imputer = KNNImputer(n_neighbors=3)
X_train=pd.DataFrame(data=imputer.fit_transform(X_train))
X_valid=pd.DataFrame(data=imputer.transform(X_valid))
X_test=pd.DataFrame(data=imputer.transform(X_test))


X_train12 ,X_train34 , y_train12, y_train34=train_test_split(X_train,y_train,test_size=0.5, random_state=1234,stratify=y_train["HIV_active"])
X_train1 ,X_train2 , y_train1, y_train2=train_test_split(X_train12,y_train12,test_size=0.5, random_state=1234,stratify=y_train12["HIV_active"])
X_train3 ,X_train4 , y_train3, y_train4=train_test_split(X_train34,y_train34,test_size=0.5, random_state=1234,stratify=y_train34["HIV_active"])
del X_train12
del X_train34
del y_train12
del y_train34
k_fold_X_train=[X_train1 ,X_train2,X_train3 ,X_train4]
k_fold_y_train=[y_train1 ,y_train2,y_train3 ,y_train4]
del X_train1
del X_train2
del X_train3
del X_train4
del y_train1
del y_train2
del y_train3
del y_train4

k_fold_y_train[0]=pd.DataFrame(data=k_fold_y_train[0])
k_fold_y_train[1]=pd.DataFrame(data=k_fold_y_train[1])
k_fold_y_train[2]=pd.DataFrame(data=k_fold_y_train[2])
k_fold_y_train[3]=pd.DataFrame(data=k_fold_y_train[3])
```

Listing 2: Import libraries, data reading, train test splitting, feature selection, and deal with missing data

```python
%%capture
try:
    import ray
except:
    %pip install ray
    import ray


%pip install ray
try:
    import optuna
except:
    %pip install optuna
    import optuna

import numpy as np
import torch
import torch.optim as optim
import torch.nn as nn
from torchvision import datasets, transforms
```

```
from torch.utils.data import DataLoader, TensorDataset, Dataset
import torch.nn.functional as F
from ray import tune
import os
import ray
from ray.tune.schedulers import ASHAScheduler
import warnings
warnings.filterwarnings('ignore')
import pandas as pd
from sklearn.model_selection import train_test_split
import matplotlib as mpl
import matplotlib.pyplot as plt
import math
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import HistGradientBoostingRegressor
try:
    from featurewiz import featurewiz
except:
    !pip install featurewiz==0.1.70
    from featurewiz import featurewiz
#read data
initial_targets=pd.read_csv("freesolv.csv")
initial_features=pd.read_csv("sampl(freesolv)_global_cdf_rdkit.csv")

initial_features=initial_features.loc[:, (initial_features != initial_features.iloc[0]).any()]

shuffled_targets=initial_targets.sample(frac=1, random_state=1234).reset_index(drop=True)
shuffled_features=initial_features.sample(frac=1, random_state=1234).reset_index(drop=True)
#shuffling the data to randomize the sequence

X_train, X_test, y_train, y_test = train_test_split(shuffled_features,shuffled_targets["freesolv"], test_size=0.1, random_state=1234)
X_train, X_valid, y_train, y_valid = train_test_split(X_train,y_train,test_size=0.22, random_state=1234)
X_train12 ,X_train34 , y_train12, y_train34=train_test_split(X_train,y_train,test_size=0.5, random_state=1234)
X_train1 ,X_train2 , y_train1, y_train2=train_test_split(X_train12,y_train12,test_size=0.5, random_state=1234)
X_train3 ,X_train4 , y_train3, y_train4=train_test_split(X_train34,y_train34,test_size=0.5, random_state=1234)
del X_train12
del X_train34
del y_train12
del y_train34
k_fold_X_train=[X_train1 ,X_train2,X_train3 ,X_train4]
k_fold_y_train=[y_train1 ,y_train2,y_train3 ,y_train4]
del X_train1
del X_train2
del X_train3
del X_train4
del y_train1
del y_train2
del y_train3
del y_train4

Data=pd.concat([X_train, y_train], axis=1) #reshape data for featurewiz feature selection
target = ["freesolv"]
from featurewiz import featurewiz
feature_selection = featurewiz(Data, target, corr_limit=0.97, verbose=2,header=0, nrows=None, outputs="features")

X_train=X_train[feature_selection[0]]
X_valid=X_valid[feature_selection[0]]
X_test=X_test[feature_selection[0]]
y_train=pd.DataFrame(data=y_train)
y_valid=pd.DataFrame(data=y_valid)
y_test=pd.DataFrame(data=y_test)

for i in np.arange(0,4):
  k_fold_X_train[i]=k_fold_X_train[i][feature_selection[0]]
  k_fold_y_train[i]=pd.DataFrame(data=k_fold_y_train[i])
```

Listing 3: Import libraries, data reading, train test splitting, and feature selection

In the continuation of the project, I will explain how to use our semi-cross-validation on the partitioned data.

2.2 Feature selection

The codes related to this section are also available in the previous listings. I have used the featurewiz library to select important features, which uses very up-to-date methods for feature selection. One of the main methods that featurewiz uses is exploiting the XGBoost function from the sklearn library.[3]

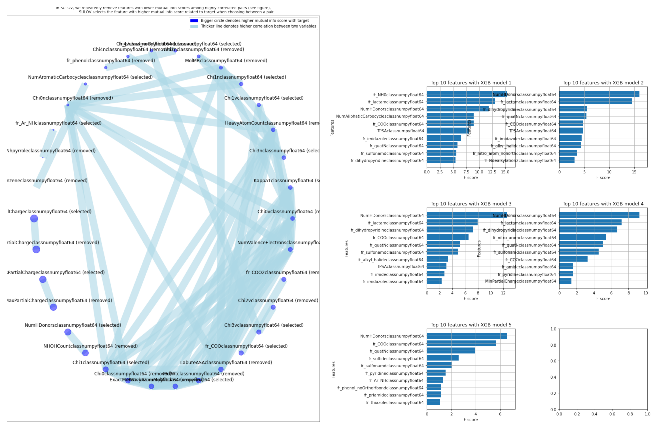


Figure 5: featurewiz output on Bbbp dataset

Out of **185** features, 20 features were removed due to being highly correlated with other features. Therefore, the remaining 165 features were processed by recursive feature selection XGBoost, and finally, **71** features were selected as important and influencing features on the target variable in the Bbbp dataset. (Fig.5)

The processing of this feature selection took 56 seconds on a Google colab server with two 2.30GHz core processors.

Out of **192** features, 51 features were removed due to being highly correlated with other features. Therefore, the remaining 141 features were processed by recursive feature selection XGBoost, and finally, **65** features were selected as important and influencing features on the target variable in the HIV dataset. (Fig.6)

The processing of this feature selection took 540 seconds on a private Ubuntu server with 24- 3.30GHz core processors.

Out of **161** features, 17 features were removed due to being highly correlated with other features. Therefore, the remaining 144 features

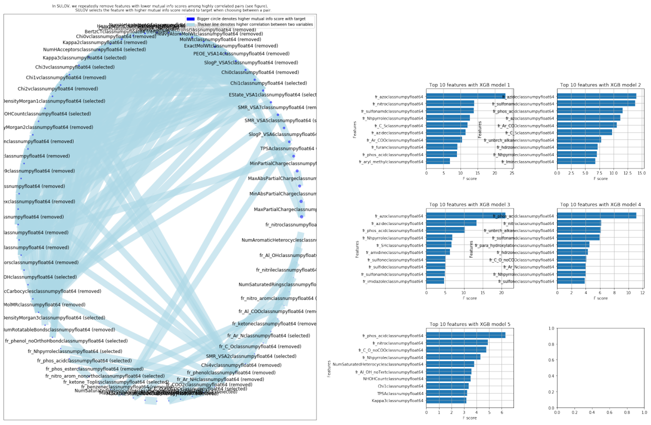


Figure 6: featurewiz output on HIV dataset

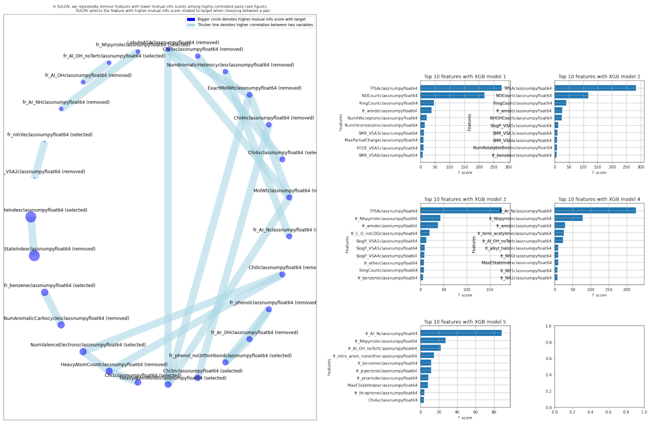


Figure 7: featurewiz output on FreeSolv dataset

were processed by recursive feature selection XGBoost, and finally, **65** features were selected as important and influencing features on the target variable in the FreeSolv dataset. (Fig.7)

The processing of this feature selection took 20 seconds on a Google colab server with two 2.30GHz core processors.

It should be noted that the feature selection section is processed **only based on the information in the training set**.

2.3 Missing values

Among these three data sets, the only one data set that has missing values. It is the HIV data set. I used knn imputer to deal with this problem. This method helps me use a predictive system to guess missing values. In fact, instead of using indicators such as median and mean, we estimate the amount of missing data more intelligently according to k close data.

Due to the large amount of data, this process was a bit time-consuming. But the result was ultimately satisfactory. Note that this step is also applied to the training, validation, and test sets based on the training set.

2.4 Accuracy and error measurement criteria

I use `ROC`[4]`_AUC`[5] score [4] for two classification projects (Bbbp,HIV) and the `MSE` [6]loss meter for the regression project.

```python
from sklearn.metrics import roc_auc_score

def compute_score(model, data_loader, device="cpu"):
    model.eval()
    metric = roc_auc_score
    with torch.no_grad():
        prediction_all= torch.empty(0, device=device)
        labels_all= torch.empty(0, device=device)
        for i, (feats, labels) in enumerate(
    data_loader):
            feats=feats.to(device)
            labels=labels.to(device)
            prediction = model(feats).to(device)
            prediction = torch.sigmoid(prediction).to(
    device)
            prediction_all = torch.cat((prediction_all
    , prediction), 0)
            labels_all = torch.cat((labels_all, labels
    ), 0)
        try:
            t = metric(labels_all.int().cpu(),
    prediction_all.cpu()).item()
        except ValueError:
            t = 0
    return t
```

Listing 4: ROC_AUC score

```python
from sklearn.metrics import roc_auc_score

def compute_score(model, data_loader, device="cpu"):
    model.eval()
    metric = roc_auc_score
    with torch.no_grad():
        prediction_all= torch.empty(0, device=device)
        labels_all= torch.empty(0, device=device)
        for i, (feats, labels) in enumerate(
    data_loader):
            feats=feats.to(device)
            labels=labels.to(device)
            prediction = model(feats).to(device)
            prediction = torch.sigmoid(prediction).to(
    device)
            prediction_all = torch.cat((prediction_all
    , prediction), 0)
            labels_all = torch.cat((labels_all, labels
    ), 0)
        try:
            t = metric(labels_all.int().cpu(),
    prediction_all.cpu()).item()
        except ValueError:
            t = 0
    return t
```

Listing 5: ROC_AUC score

```python
from sklearn.metrics import mean_squared_error

def compute_loss(model, data_loader, device="cpu"):
    model.eval()
    metric = mean_squared_error
    with torch.no_grad():
        prediction_all= torch.empty(0, device=device)
        labels_all= torch.empty(0, device=device)
        for i, (feats, labels) in enumerate(
    data_loader):
            feats=feats.to(device)
            labels=labels.to(device)
            prediction = model(feats).to(device)
            prediction_all = torch.cat((prediction_all
    , prediction), 0)
            labels_all = torch.cat((labels_all, labels
    ), 0)
            t = metric(labels_all.int().cpu()
    return t
```

Listing 6: MSE loss

3 NEURAL NETWORKS AND INTRODUCTION OF HYPER-PARAMETERS

In the optimization process of hyper-parameters, I do not decide to generally consider the number of nodes of each hidden layer as a hyper-parameter. Since I intuitively feel that the number of nodes should gradually decrease, I define the general shape of the neural network and add some nodes to each hidden layer at each step. These numbers can be defined as hyper-parameters. The figure below can give me a general idea of what I am looking for.

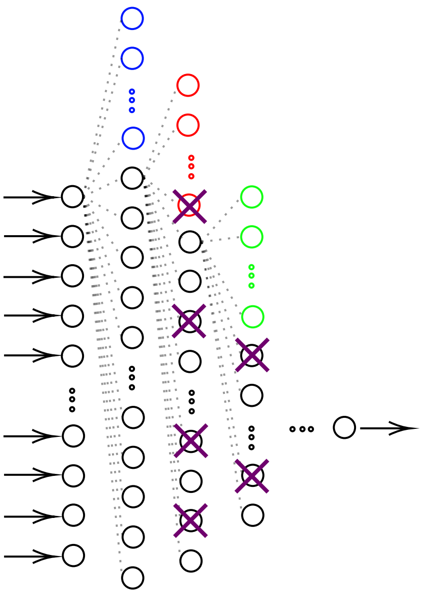


Figure 8: The general structure of the neural networks of this project

3.1 Defining data ladders

At first, according to what was said in the previous part and concerning figure 4, I will define different data loaders for different goals. Note that I separated the data during the last section completely. Here these data loaders help me to use them in the PyTorch[5] environment.

```
class BbbpDataset_train (Dataset):
    def __init__(self,transform=None):
        #data loading
        self.x=X_train.to_numpy().astype("float32")
        self.y=y_train.to_numpy().astype("float32")
        self.n_samples=X_train.shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class BbbpDataset_train (Dataset):
    def __init__(self,transform=None):
        #data loading
        self.x=X_train.to_numpy().astype("float32")
        self.y=y_train.to_numpy().astype("float32")
        self.n_samples=X_train.shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class kf_BbbpDataset_train (Dataset):
    def __init__(self,k,transform=None):
        #data loading np.delete(np.array([0,1,2,3]),k)
        self.x=(   k_fold_X_train[   np.delete(np.array([0,1,2,3]),k)[0]   ].append(k_fold_X_train[   np.delete(np.array([0,1,2,3]),k)[1]   ]).append(k_fold_X_train[   np.delete(np.array([0,1,2,3]),k)[2]   ])     ).to_numpy().astype("float32")
        self.y=(   k_fold_y_train[   np.delete(np.array([0,1,2,3]),k)[0]   ].append(k_fold_y_train[   np.delete(np.array([0,1,2,3]),k)[1]   ]).append(k_fold_y_train[   np.delete(np.array([0,1,2,3]),k)[2]   ])     ).to_numpy().astype("float32")
        self.n_samples=(k_fold_X_train[   np.delete(np.array([0,1,2,3]),k)[0]   ].append(k_fold_X_train[   np.delete(np.array([0,1,2,3]),k)[1]   ]).append(k_fold_X_train[   np.delete(np.array([0,1,2,3]),k)[2]   ])).shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class kf_BbbpDataset_valid (Dataset):
    def __init__(self,k,transform=None):
        #data loading
        self.x=(k_fold_X_train[k]).to_numpy().astype("float32")
        self.y=(k_fold_y_train[k]).to_numpy().astype("float32")
        self.n_samples=(k_fold_X_train[k]).shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class BbbpDataset_valid (Dataset):
    def __init__(self,transform=None):
        #data loading
        self.x=X_valid.to_numpy().astype("float32")
        self.y=y_valid.to_numpy().astype("float32")
        self.n_samples=X_valid.shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class BbbpDataset_test (Dataset):
    def __init__(self,transform=None):
        #data loading
        self.x=X_test.to_numpy().astype("float32")
        self.y=y_test.to_numpy().astype("float32")
        self.n_samples=X_test.shape[0]
        self.transform=transform
    def __getitem__(self, index):
```

```python
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class ToTensor():
    def __call__(self,sample):
        inputs,targets=sample
        inputs=torch.from_numpy(inputs.astype("float32"))
        targets=torch.tensor(targets.astype("float32"))
        #targets=targets.view(targets.shape[0],1)
        return inputs,targets

#training set --
training_set = BbbpDataset_train(transform=ToTensor())
train_loader = DataLoader(dataset=training_set,
                          batch_size=64,
                          shuffle=True)

dataiter_train = iter(train_loader)
data_train = dataiter_train.next()


def trainloader(config):
    return  DataLoader(dataset=training_set,
                          batch_size=config["batch_size"],
                          shuffle=True,num_workers=2)

#kf-training set --
def kf_trainloader(config,k):
    return  DataLoader(dataset=kf_BbbpDataset_train(k,transform=ToTensor())    ,
                          batch_size=config["batch_size"],
                          shuffle=True,num_workers=2)
#kf-training set --
def kf_validloader(config,k):
    return  DataLoader(dataset=kf_BbbpDataset_valid(k,transform=ToTensor())    ,
                          batch_size=config["batch_size"],
                          shuffle=True,num_workers=2)
#validation set --
validation_set = BbbpDataset_valid(transform=ToTensor())
valid_loader = DataLoader(dataset=validation_set,
                          batch_size=64,
                          shuffle=True)

dataiter_valid = iter(valid_loader)
data_valid = dataiter_valid.next()

def validloader(config):
    return  DataLoader(dataset=validation_set,
                          batch_size=config["batch_size"],
                          shuffle=True,num_workers=2)
#test set --
test_set = BbbpDataset_test(transform=ToTensor())
test_loader = DataLoader(dataset=test_set,
                          batch_size=8,
                          shuffle=False)

dataiter_test = iter(test_loader)
data_test = dataiter_test.next()

def testloader(config):
    return  DataLoader(dataset=test_set,
                          batch_size=4,
                          shuffle=False,num_workers=2)
```

Listing 7: Defining data ladders

```python
class HIVDataset_train (Dataset):
    def __init__(self,transform=None):
        #data loading
        self.x=X_train.to_numpy().astype("float32")
        self.y=y_train.to_numpy().astype("float32")
        self.n_samples=X_train.shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class HIVDataset_train (Dataset):
    def __init__(self,transform=None):
        #data loading
        self.x=X_train.to_numpy().astype("float32")
        self.y=y_train.to_numpy().astype("float32")
        self.n_samples=X_train.shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class kf_HIVDataset_train (Dataset):
    def __init__(self,k,transform=None):
        #data loading np.delete(np.array([0,1,2,3]),k)
        self.x=(   k_fold_X_train[  np.delete(np.array([0,1,2,3]),k)[0]  ].append(k_fold_X_train[  np.delete(np.array([0,1,2,3]),k)[1]  ]).append(k_fold_X_train[  np.delete(np.array([0,1,2,3]),k)[2]  ])    ).to_numpy().astype("float32")
        self.y=(   k_fold_y_train[  np.delete(np.array([0,1,2,3]),k)[0]  ].append(k_fold_y_train[  np.delete(np.array([0,1,2,3]),k)[1]  ]).append(k_fold_y_train[  np.delete(np.array([0,1,2,3]),k)[2]  ])    ).to_numpy().astype("float32")
        self.n_samples=(k_fold_X_train[  np.delete(np.array([0,1,2,3]),k)[0]  ].append(k_fold_X_train[  np.delete(np.array([0,1,2,3]),k)[1]  ]).append(k_fold_X_train[  np.delete(np.array([0,1,2,3]),k)[2]  ])).shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class kf_HIVDataset_valid (Dataset):
    def __init__(self,k,transform=None):
        #data loading
        self.x=(k_fold_X_train[k]).to_numpy().astype("float32")
        self.y=(k_fold_y_train[k]).to_numpy().astype("float32")
        self.n_samples=(k_fold_X_train[k]).shape[0]
        self.transform=transform
```

```
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class HIVDataset_valid (Dataset):
    def __init__(self,transform=None):
        #data loading
        self.x=X_valid.to_numpy().astype("float32")
        self.y=y_valid.to_numpy().astype("float32")
        self.n_samples=X_valid.shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class HIVDataset_test (Dataset):
    def __init__(self,transform=None):
        #data loading
        self.x=X_test.to_numpy().astype("float32")
        self.y=y_test.to_numpy().astype("float32")
        self.n_samples=X_test.shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class ToTensor():
    def __call__(self,sample):
        inputs,targets=sample
        inputs=torch.from_numpy(inputs.astype("float32"))
        targets=torch.tensor(targets.astype("float32"))
        #targets=targets.view(targets.shape[0],1)
        return inputs,targets

#training set --
training_set = HIVDataset_train(transform=ToTensor())
train_loader = DataLoader(dataset=training_set,
                          batch_size=64,
                          shuffle=True)

dataiter_train = iter(train_loader)
data_train = dataiter_train.next()


def trainloader(config):
    return  DataLoader(dataset=training_set,
                          batch_size=config["batch_size"],
                          shuffle=True,num_workers=2)

#kf-training set --
def kf_trainloader(config,k):
    return  DataLoader(dataset=kf_HIVDataset_train(k,transform=ToTensor())    ,
                          batch_size=config["batch_size"],
                          shuffle=True,num_workers=2)
#kf-training set --
def kf_validloader(config,k):
    return  DataLoader(dataset=kf_HIVDataset_valid(k,transform=ToTensor())    ,
                          batch_size=config["batch_size"],
                          shuffle=True,num_workers=2)
#validation set --
validation_set = HIVDataset_valid(transform=ToTensor())
valid_loader = DataLoader(dataset=validation_set,
                          batch_size=64,
                          shuffle=True)

dataiter_valid = iter(valid_loader)
data_valid = dataiter_valid.next()

def validloader(config):
    return  DataLoader(dataset=validation_set,
                          batch_size=config["batch_size"],
                          shuffle=True,num_workers=2)
#test set --
test_set = HIVDataset_test(transform=ToTensor())
test_loader = DataLoader(dataset=test_set,
                          batch_size=8,
                          shuffle=False)

dataiter_test = iter(test_loader)
data_test = dataiter_test.next()

def testloader(config):
    return  DataLoader(dataset=test_set,
                          batch_size=4,
                          shuffle=False,num_workers=2)
```

Listing 8: Defining data ladders

```
class freesolvDataset_train (Dataset):
    def __init__(self,transform=None):
        #data loading
        self.x=X_train.to_numpy().astype("float32")
        self.y=y_train.to_numpy().astype("float32")
        self.n_samples=X_train.shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class freesolvDataset_train (Dataset):
    def __init__(self,transform=None):
        #data loading
        self.x=X_train.to_numpy().astype("float32")
        self.y=y_train.to_numpy().astype("float32")
        self.n_samples=X_train.shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class kf_freesolvDataset_train (Dataset):
    def __init__(self,k,transform=None):
        #data loading np.delete(np.array([0,1,2,3]),k)
```

```
        self.x=(   k_fold_X_train[  np.delete(np.array([0,1,2,3]),k)[0]  ].append(k_fold_X_train[  np.delete(np.array([0,1,2,3]),k)[1]  ]).append(k_fold_X_train[  np.delete(np.array([0,1,2,3]),k)[2]  ])   ).to_numpy().astype("float32")
        self.y=(   k_fold_y_train[  np.delete(np.array([0,1,2,3]),k)[0]  ].append(k_fold_y_train[  np.delete(np.array([0,1,2,3]),k)[1]  ]).append(k_fold_y_train[  np.delete(np.array([0,1,2,3]),k)[2]  ])   ).to_numpy().astype("float32")
        self.n_samples=(k_fold_X_train[  np.delete(np.array([0,1,2,3]),k)[0]  ].append(k_fold_X_train[  np.delete(np.array([0,1,2,3]),k)[1]  ]).append(k_fold_X_train[  np.delete(np.array([0,1,2,3]),k)[2]  ])).shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class kf_freesolvDataset_valid (Dataset):
    def __init__(self,k,transform=None):
        #data loading
        self.x=(k_fold_X_train[k]).to_numpy().astype("float32")
        self.y=(k_fold_y_train[k]).to_numpy().astype("float32")
        self.n_samples=(k_fold_X_train[k]).shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class freesolvDataset_valid (Dataset):
    def __init__(self,transform=None):
        #data loading
        self.x=X_valid.to_numpy().astype("float32")
        self.y=y_valid.to_numpy().astype("float32")
        self.n_samples=X_valid.shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class freesolvDataset_test (Dataset):
    def __init__(self,transform=None):
        #data loading
        self.x=X_test.to_numpy().astype("float32")
        self.y=y_test.to_numpy().astype("float32")
        self.n_samples=X_test.shape[0]
        self.transform=transform
    def __getitem__(self, index):
        sample=self.x[index], self.y[index]
        if self.transform:
            sample=self.transform(sample)
        return sample
    def __len__(self):
        return self.n_samples

class ToTensor():
    def __call__(self,sample):
        inputs,targets=sample
        inputs=torch.from_numpy(inputs.astype("float32"))
        targets=torch.tensor(targets.astype("float32"))
        #targets=targets.view(targets.shape[0],1)
        return inputs,targets

#training set --
training_set = freesolvDataset_train(transform=ToTensor())
train_loader = DataLoader(dataset=training_set,
                          batch_size=64,
                          shuffle=True)

dataiter_train = iter(train_loader)
data_train = dataiter_train.next()


def trainloader(config):
    return  DataLoader(dataset=training_set,
                          batch_size=config["batch_size"],
                          shuffle=True,num_workers=2)

#kf-training set --
def kf_trainloader(config,k):
    return  DataLoader(dataset=kf_freesolvDataset_train(k,transform=ToTensor()),
                          batch_size=config["batch_size"],
                          shuffle=True,num_workers=2)
#kf-training set --
def kf_validloader(config,k):
    return  DataLoader(dataset=kf_freesolvDataset_valid(k,transform=ToTensor()),
                          batch_size=config["batch_size"],
                          shuffle=True,num_workers=2)
#validation set --
validation_set = freesolvDataset_valid(transform=ToTensor())
valid_loader = DataLoader(dataset=validation_set,
                          batch_size=64,
                          shuffle=True)

dataiter_valid = iter(valid_loader)
data_valid = dataiter_valid.next()

def validloader(config):
    return  DataLoader(dataset=validation_set,
                          batch_size=config["batch_size"],
                          shuffle=True,num_workers=2)
#test set --
test_set = freesolvDataset_test(transform=ToTensor())
test_loader = DataLoader(dataset=test_set,
                          batch_size=8,
                          shuffle=False)

dataiter_test = iter(test_loader)
data_test = dataiter_test.next()

def testloader(config):
    return  DataLoader(dataset=test_set,
                          batch_size=4,
                          shuffle=False,num_workers=2)
```

Listing 9: Defining data ladders

3.2 Defining Neural Networks

According to what I explained in Figure 5, I define neural networks for all three data sets. Of course, slight and sometimes fundamental differences exist between the neural networks of these three models. For example, it can be clearly seen that the last layer of the regression dataset's neural network does not have an activation function. Of course, there are other differences.

But in general, in all these neural networks, it is clear that at least some activation functions, three natural numbers that regulate the number of nodes in some layers, are defined as hyper-parameters. Of course, the learning rate is also a hyper-parameter that is defined, and it is better to introduce it in the following sections

```
n_samples,n_features=X_train.shape
class NeuralNetwork (nn.Module):
    def __init__(self,n_input_features,l1, l2,l3, config):
        super (NeuralNetwork, self).__init__()
        self.config = config
        self.linear1=nn.Linear(n_input_features,4*math.floor(n_input_features/2)+l1)
        self.linear2=nn.Linear(l1+4*math.floor(n_input_features/2),math.floor(n_input_features/2)+l2)
        self.D1=torch.nn.Dropout(config.get("drop_out_ratio1"))
        self.linear3=nn.Linear(math.floor(n_input_features/2)+l2,math.floor(n_input_features/4)+l3)
        self.D2=torch.nn.Dropout(config.get("drop_out_ratio2"))
        self.linear5=nn.Linear(math.floor(n_input_features/4)+l3,1)

        self.a1 = self.config.get("a1")
        self.a2 = self.config.get("a2")
        self.a3 = self.config.get("a3")


    @staticmethod
    def activation_func(act_str):
        if act_str=="tanh" or act_str=="sigmoid":
            return eval("torch."+act_str)
        elif act_str=="silu" or act_str=="relu" or act_str=="leaky_relu" or act_str=="gelu":
            return eval("torch.nn.functional."+act_str)

    def forward(self,x):
        out=self.linear1(x)
        out=self.activation_func(self.a1)(out.float())
        out=self.linear2(out)
        out=self.D1(out)
        out=self.activation_func(self.a2)(out.float())
        out=self.linear3(out)
        out=self.activation_func(self.a3)(out.float())
        out=self.D1(out)
        out=self.linear5(out)
        out=torch.sigmoid(out)
        y_predicted=out
        return y_predicted
```

Listing 10: Neural network for the Bbbp dataset

```
n_samples,n_features=X_train.shape
class NeuralNetwork (nn.Module):
    def __init__(self,n_input_features,l1, l2,l3, config):
        super (NeuralNetwork, self).__init__()
        self.config = config
        self.linear1=nn.Linear(n_input_features, n_input_features+l1)
        self.linear2=nn.Linear(n_input_features+l1, math.floor(n_input_features/2)+l2)
        self.D1=torch.nn.Dropout(config.get("drop_out_ratio1"))
        self.linear3=nn.Linear(math.floor(n_input_features/2)+l2,math.floor(n_input_features/4)+l3)
        self.D2=torch.nn.Dropout(config.get("drop_out_ratio2"))
        self.linear5=nn.Linear(math.floor(n_input_features/4)+l3,1)

        self.a1 = self.config.get("a1")
        self.a2 = self.config.get("a2")
        self.a3 = self.config.get("a3")


    @staticmethod
    def activation_func(act_str):
        if act_str=="tanh" or act_str=="sigmoid":
            return eval("torch."+act_str)
        elif act_str=="silu" or act_str=="relu" or act_str=="leaky_relu" or act_str=="gelu":
            return eval("torch.nn.functional."+act_str)

    def forward(self,x):
        out=self.linear1(x)
        out=self.activation_func(self.a1)(out.float())
        out=self.linear2(out)
        out=self.D1(out)
        out=self.activation_func(self.a2)(out.float())
        out=self.linear3(out)
        out=self.activation_func(self.a3)(out.float())
        out=self.D1(out)
        out=self.linear5(out)
        out=torch.sigmoid(out)
        y_predicted=out
        return y_predicted
```

Listing 11: Neural network for the HIV dataset

```
n_samples,n_features=X_train.shape
class NeuralNetwork (nn.Module):
    def __init__(self,n_input_features,l1, l2,l3, config):
        super (NeuralNetwork, self).__init__()
        self.config = config
        self.linear1=nn.Linear(n_input_features,4*math.floor(n_input_features/2)+l1)
        self.linear2=nn.Linear(l1+4*math.floor(n_input_features/2),math.floor(n_input_features/2)+l2)
        self.D1=torch.nn.Dropout(config.get("drop_out_ratio1"))
        self.linear3=nn.Linear(math.floor(n_input_features/2)+l2,math.floor(n_input_features/4)+l3)
        self.D2=torch.nn.Dropout(config.get("drop_out_ratio2"))
        self.linear5=nn.Linear(math.floor(n_input_features/4)+l3,1)

        self.a1 = self.config.get("a1")
        self.a2 = self.config.get("a2")
        self.a3 = self.config.get("a3")


    @staticmethod
    def activation_func(act_str):
```

```
        if act_str=="tanh" or act_str=="sigmoid":
            return eval("torch."+act_str)
        elif act_str=="silu" or act_str=="relu" or act_str=="leaky_relu" or act_str=="gelu":
            return eval("torch.nn.functional."+act_str)

    def forward(self,x):
        out=self.linear1(x)
        out=self.activation_func(self.a1)(out.float())
        out=self.linear2(out)
        out=self.D1(out)
        out=self.activation_func(self.a2)(out.float())
        out=self.linear3(out)
        out=self.activation_func(self.a3)(out.float())
        out=self.D1(out)
        out=self.linear5(out)
        y_predicted=out
        return y_predicted
```

Listing 12: Neural network for the FreeSolv dataset

3.3 Training loops

Perhaps the most critical part of this project is this part. In this section, two neural network training processes are done separately. It would help if you remembered that I discussed a semi-cross-validation in figure 4. My goal in this project was not to get high accuracy. I am looking for a mindset to optimize the hyper-parameters of a neural network.

Let's talk a little more about this semi-cross-validation. Why did I call it semi-cross-validation? The truth is that I do not stop the process of training the neural network weights when the local validation set is changed, so I look for an overfit model on the entire training set. After that, by resetting the neural network weights with the same hyper-parameters, I train the neural network as is customary in all projects. But what is the application of this mindset? I'm looking for hyper-parameters that have good ultimate accuracy and don't overfit easily compared to other hyper-parameters. Having the overfitted models' results helps me choose hyper-parameters that do not easily lead to overfitting on the training set.

A criticism may be raised that says that this process cannot be justified because the failure of a neural network may be largely dependent on the choice of learning rate and not due to the hyper-parameters themselves. I answer that my optimizer in these three projects is Adam, and the learning rate changes during the learning process. Therefore, our learning rate specifies only an initial value, which of course, I do not deny its effect in any way, but I have given up its direct impact on this project. On the other hand, I am looking for a package of hyper-parameters, including the learning rate itself. So, I do not ignore the learning rate's effect in model training.

```
def train_Bbbp(config,checkpoint_dir=None,max_iter=11):
    net = NeuralNetwork(np.shape(feature_selection[0])[0],config["l1"],config["l2"],config["l3"],config)


    device = "cpu"
    if torch.cuda.is_available():
        device = "cuda:0"
        if torch.cuda.device_count() > 1:
            net = nn.DataParallel(net)
    net.to(device)

    #Define my loss function and optimizer
    criterion=nn.BCELoss()
    optimizer=torch.optim.Adam(net.parameters(), lr=config["lr"])


    # The `checkpoint_dir` parameter gets passed by Ray Tune when a checkpoint
    # should be restored.
    if checkpoint_dir:
        checkpoint = os.path.join(checkpoint_dir, "checkpoint")
        model_state, optimizer_state = torch.load(checkpoint)
        net.load_state_dict(model_state)
        optimizer.load_state_dict(optimizer_state)



    localiter=0
    for epoch in range(max_iter):  # loop over the dataset multiple times
        running_loss1 = 0.0
        epoch_steps1 = 0
        for i, data in enumerate(kf_trainloader(config,1), 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss1 += loss.item()
            epoch_steps1 += 1
            if i % 2000 == 1999:  # print every 2000 mini-batches
                print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss1 / epoch_steps1))
                running_loss1 = 0.0


        # Validation score
        score1 = compute_score(net, kf_validloader(config,1), device="cpu")

    #second loop -

        running_loss2 = 0.0
        epoch_steps2 = 0
        for i, data in enumerate(kf_trainloader(config,2), 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
```

```
            running_loss2 += loss.item()
            epoch_steps2 += 1
            if i % 2000 == 1999:  # print every 2000 mini-batches
                print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss2 / epoch_steps2))
                running_loss2 = 0.0


        # Validation score

        score2 = compute_score(net, kf_validloader(config,2), device="cpu")

    #third loop -

        running_loss3 = 0.0
        epoch_steps3 = 0
        for i, data in enumerate(kf_trainloader(config,3), 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss3 += loss.item()
            epoch_steps3 += 1
            if i % 2000 == 1999:  # print every 2000 mini-batches
                print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss3 / epoch_steps3))
                running_loss3 = 0.0


        # Validation score

        score3 = compute_score(net, kf_validloader(config,3), device="cpu")

    #forth loop -
    for epoch in range(max_iter):  # loop over the dataset multiple times
        running_loss4 = 0.0
        epoch_steps4 = 0
        for i, data in enumerate(kf_trainloader(config,0), 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss4 += loss.item()
            epoch_steps4 += 1
            if i % 2000 == 1999:  # print every 2000 mini-batches
                print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss4 / epoch_steps4))
                running_loss4 = 0.0


        # Validation score

        score4 = compute_score(net, kf_validloader(config,0), device="cpu")



        # Validation score

        kf_score=np.min([score1,score2,score3,score4])
        #print(f"score1: {score1:.4f}, score2: {score2:.4f}, score3: {score3:.4f}. score4: {score4:.4f}--->kf_score: {kf_score:.5f}")

        localiter=localiter+1
        val_score = compute_score(net, validloader(config), device="cpu")

        score=np.mean([val_score,val_score,kf_score])-((localiter/max_iter)**2)*0.033-(kf_score-val_score)/4




        with tune.checkpoint_dir(epoch) as checkpoint_dir:
            path = os.path.join(checkpoint_dir, "checkpoint")
            torch.save((net.state_dict(), optimizer.state_dict()), path)
        tune.report(score=score,kf_score=kf_score,val_score=val_score)

    print("Finished Training")
```

Listing 13: Training loops

```
def train_HIV(config,checkpoint_dir=None,max_iter=11):
    net = NeuralNetwork(np.shape(feature_selection[0])[0],config["l1"],config["l2"],config["l3"],config)


    device = "cpu"
    if torch.cuda.is_available():
        device = "cuda:0"
        if torch.cuda.device_count() > 1:
            net = nn.DataParallel(net)
    net.to(device)

    #Define my loss function and optimizer
    criterion=nn.BCELoss()
    optimizer=torch.optim.Adam(net.parameters(), lr=config["lr"])


    # The `checkpoint_dir` parameter gets passed by Ray Tune when a checkpoint
    # should be restored.
    if checkpoint_dir:
        checkpoint = os.path.join(checkpoint_dir, "checkpoint")
```

```
    model_state, optimizer_state = torch.load(checkpoint)
    net.load_state_dict(model_state)
    optimizer.load_state_dict(optimizer_state)



localiter=0
for epoch in range(max_iter):  # loop over the dataset multiple times
    running_loss1 = 0.0
    epoch_steps1 = 0
    for i, data in enumerate(kf_trainloader(config,1), 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss1 += loss.item()
        epoch_steps1 += 1
        if i % 2000 == 1999:  # print every 2000 mini-batches
            print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss1 / epoch_steps1))
            running_loss1 = 0.0


    # Validation score
    score1 = compute_score(net, kf_validloader(config,1), device="cpu")

#second loop -

    running_loss2 = 0.0
    epoch_steps2 = 0
    for i, data in enumerate(kf_trainloader(config,2), 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss2 += loss.item()
        epoch_steps2 += 1
        if i % 2000 == 1999:  # print every 2000 mini-batches
            print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss2 / epoch_steps2))
            running_loss2 = 0.0



    # Validation score

    score2 = compute_score(net, kf_validloader(config,2), device="cpu")

#third loop -

    running_loss3 = 0.0
    epoch_steps3 = 0
    for i, data in enumerate(kf_trainloader(config,3), 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss3 += loss.item()
        epoch_steps3 += 1
        if i % 2000 == 1999:  # print every 2000 mini-batches
            print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss3 / epoch_steps3))
            running_loss3 = 0.0


    # Validation score

    score3 = compute_score(net, kf_validloader(config,3), device="cpu")

#forth loop -
  # loop over the dataset multiple times
    running_loss4 = 0.0
    epoch_steps4 = 0
    for i, data in enumerate(kf_trainloader(config,0), 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss4 += loss.item()
        epoch_steps4 += 1
        if i % 2000 == 1999:  # print every 2000 mini-batches
            print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss4 / epoch_steps4))
            running_loss4 = 0.0
```

```


        # Validation score

        score4 = compute_score(net, kf_validloader(config,0), device="cpu")

    #global loop -
    for layer in net.children():
      if hasattr(layer, 'reset_parameters'):
        layer.reset_parameters()
    for epoch in range(max_iter):
        running_loss = 0.0
        epoch_steps = 0
        for i, data in enumerate(trainloader(config), 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
            epoch_steps += 1
            if i % 2000 == 1999:  # print every 2000 mini-batches
                print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss / epoch_steps))
                running_loss = 0.0


        # Validation score

        kf_score=np.min([score1,score2,score3,score4])


        localiter=localiter+1
        val_score = compute_score(net, validloader(config), device="cpu")

        score=np.mean([val_score,val_score,kf_score])-((localiter/max_iter)**2)*0.04-(kf_score-val_score)/4




        with tune.checkpoint_dir(epoch) as checkpoint_dir:
            path = os.path.join(checkpoint_dir, "checkpoint")
            torch.save((net.state_dict(), optimizer.state_dict()), path)
        tune.report(score=score,kf_score=kf_score,val_score=val_score)

    print("Finished Training")
```

Listing 14: Training loops

```
def train_freesolv(config,checkpoint_dir=None,max_iter=11):
    net = NeuralNetwork(np.shape(feature_selection[0])[0],config["l1"],config["l2"],config["l3"],config)


    device = "cpu"
    if torch.cuda.is_available():
        device = "cuda:0"
        if torch.cuda.device_count() > 1:
            net = nn.DataParallel(net)
    net.to(device)

    #Define my loss function and optimizer
    criterion=nn.MSELoss()
    optimizer=torch.optim.Adam(net.parameters(), lr=config["lr"])


    # The `checkpoint_dir` parameter gets passed by Ray Tune when a checkpoint
    # should be restored.
    if checkpoint_dir:
        checkpoint = os.path.join(checkpoint_dir, "checkpoint")
        model_state, optimizer_state = torch.load(checkpoint)
        net.load_state_dict(model_state)
        optimizer.load_state_dict(optimizer_state)



    localiter=0
    for epoch in range(max_iter):  # loop over the dataset multiple times
        running_loss1 = 0.0
        epoch_steps1 = 0
        for i, data in enumerate(kf_trainloader(config,1), 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss1 += loss.item()
            epoch_steps1 += 1
            if i % 2000 == 1999:  # print every 2000 mini-batches
                print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss1 / epoch_steps1))
                running_loss1 = 0.0


        # Validation loss
        loss1 = compute_loss(net, kf_validloader(config,1), device="cpu")


    #second loop -

        running_loss2 = 0.0
        epoch_steps2 = 0
```

```
    for i, data in enumerate(kf_trainloader(config,2), 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss2 += loss.item()
        epoch_steps2 += 1
        if i % 2000 == 1999:  # print every 2000 mini-batches
            print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss2 / epoch_steps2))
            running_loss2 = 0.0


    # Validation loss

    loss2 = compute_loss(net, kf_validloader(config,2), device="cpu")


#third loop -

    running_loss3 = 0.0
    epoch_steps3 = 0
    for i, data in enumerate(kf_trainloader(config,3), 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss3 += loss.item()
        epoch_steps3 += 1
        if i % 2000 == 1999:  # print every 2000 mini-batches
            print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss3 / epoch_steps3))
            running_loss3 = 0.0


    # Validation loss

    loss3 = compute_loss(net, kf_validloader(config,3), device="cpu")


#forth loop -
    # loop over the dataset multiple times
    running_loss4 = 0.0
    epoch_steps4 = 0
    for i, data in enumerate(kf_trainloader(config,0), 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss4 += loss.item()
        epoch_steps4 += 1
        if i % 2000 == 1999:  # print every 2000 mini-batches
            print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss4 / epoch_steps4))
            running_loss4 = 0.0


    # Validation loss

    loss4 = compute_loss(net, kf_validloader(config,0), device="cpu")




    # Validation loss

    kf_loss=np.max([loss1,loss2,loss3,loss4])

    #global loop -
for layer in net.children():
  if hasattr(layer, 'reset_parameters'):
    layer.reset_parameters()
for epoch in range(max_iter):
    running_loss = 0.0
    epoch_steps = 0
    for i, data in enumerate(trainloader(config), 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        epoch_steps += 1
        if i % 2000 == 1999:  # print every 2000 mini-batches
            print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss / epoch_steps))
```

```
                running_loss4 = 0.0

        localiter=localiter+1
        val_loss = compute_loss(net, validloader(config), device="cpu")

        loss=np.mean([val_loss,val_loss,kf_loss])+((localiter/max_iter)**2)*0.2+(val_loss-kf_loss)/2





        with tune.checkpoint_dir(epoch) as checkpoint_dir:
            path = os.path.join(checkpoint_dir, "checkpoint")
            torch.save((net.state_dict(), optimizer.state_dict()), path)
        tune.report(loss=loss,kf_loss=kf_loss,val_loss=val_loss)

    print("Finished Training")
```

Listing 15: Training loops

I have to explain the leading indicator I chose to choose the best model. As I said before, I have penalized the model for overfitting the neural networks in more iterations. In addition, the sooner Ray concludes that the neural network does not need more training; in other words, the training process stops in fewer iterations, the less likely the final model will be overfit. (I first observed this experimentally while working with the data and was inspired to choose the final criterion) Therefore, I also penalized the models that use the maximum possible iteration for training.

Ultimately, I expect my chosen metric to be an estimate of the result of the test set. In the following, we will examine how successful I have been.

3.4 Applying the best model to the test set

After the process of training and generating the best model, it is apparent that we have to apply the final algorithm to the test set. This will help us to understand whether we are caught in the Biased Optimization trap or not.

```
def test_best_model(best_trial):
    best_trained_model = NeuralNetwork(np.shape(feature_selection[0])[0],best_trial.config["l1"],best_trial.config["l2"],best_trial.config["l3"],best_trial.config)
    device = "cuda:0" if torch.cuda.is_available() else "cpu"
    best_trained_model.to(device)

    checkpoint_path = os.path.join(best_trial.checkpoint.value, "checkpoint")

    model_state, optimizer_state = torch.load(checkpoint_path)
    best_trained_model.load_state_dict(model_state)

    test_score = compute_score(best_trained_model, testloader(best_trial.config), device)
    print("Best trial test set score:              - {}".format(test_score))
    return best_trial.config, best_trained_model
```

Listing 16: Applying the best model to the test set

```
def test_best_model(best_trial):
    best_trained_model = NeuralNetwork(np.shape(feature_selection[0])[0],best_trial.config["l1"],best_trial.config["l2"],best_trial.config["l3"],best_trial.config)
    device = "cuda:0" if torch.cuda.is_available() else "cpu"
    best_trained_model.to(device)

    checkpoint_path = os.path.join(best_trial.checkpoint.value, "checkpoint")

    model_state, optimizer_state = torch.load(checkpoint_path)
    best_trained_model.load_state_dict(model_state)

    test_score = compute_score(best_trained_model, testloader(best_trial.config), device)
    print("Best trial test set score:              - {}".format(test_score))
    return best_trial.config, best_trained_model
```

Listing 17: Applying the best model to the test set

```
def test_best_model(best_trial):
    best_trained_model = NeuralNetwork(np.shape(feature_selection[0])[0],best_trial.config["l1"],best_trial.config["l2"],best_trial.config["l3"],best_trial.config)
    device = "cuda:0" if torch.cuda.is_available() else "cpu"
    best_trained_model.to(device)

    checkpoint_path = os.path.join(best_trial.checkpoint.value, "checkpoint")

    model_state, optimizer_state = torch.load(checkpoint_path)
    best_trained_model.load_state_dict(model_state)

    test_loss = compute_loss(best_trained_model, testloader(best_trial.config), device)
    print("Best trial test set loss:               - {}".format(test_loss))
    return best_trial.config, best_trained_model
```

Listing 18: Applying the best model to the test set

3.5 Defining the search space

In this part, I specify the space in which we are going to find the optimal hyper-parameters. In this part, all three data sets are similar.

```
config = {
      "l1": tune.choice([2**6,2**7,2**8]),
      "l2": tune.choice([2**6,2**7,2**8]),
      "l3": tune.choice([2**6,2**7,2**8]),
      "lr": tune.quniform(0.0001, 0.1,0.0001),
      "drop_out_ratio1": tune.quniform(0.3, 0.65,0.01),
      "drop_out_ratio2": tune.quniform(0.01, 1,0.01),
      "a1":tune.choice(["relu","leaky_relu","silu"]),
      "a2":tune.choice(["gelu","leaky_relu"]),
      "a3":tune.choice(["relu", "gelu"]),
      "batch_size": tune.choice([ 32, 64, 128]),
      }
```

Listing 19: Config

```
config = {
      "l1": tune.choice([2**6,2**7,2**8]),
      "l2": tune.choice([2**6,2**7,2**8]),
      "l3": tune.choice([2**6,2**7,2**8]),
```

```
        "lr": tune.quniform(0.0001, 0.1,0.0001),
        "drop_out_ratio1": tune.quniform(0.3, 0.65,0.01),
        "drop_out_ratio2": tune.quniform(0.01, 1,0.01),
        "a1":tune.choice(["relu","leaky_relu","silu"]),
        "a2":tune.choice(["gelu","leaky_relu"]),
        "a3":tune.choice(["relu", "gelu"]),
        "batch_size": tune.choice([ 32, 64, 128]),
        }
```

Listing 20: Config

```
config = {
        "l1": tune.choice([2**6,2**7,2**8]),
        "l2": tune.choice([2**6,2**7,2**8]),
        "l3": tune.choice([2**6,2**7,2**8]),
        "lr": tune.quniform(0.0001, 0.1,0.0001),
        "drop_out_ratio1": tune.quniform(0.3, 0.65,0.01),
        "drop_out_ratio2": tune.quniform(0.01, 1,0.01),
        "a1":tune.choice(["relu","leaky_relu","silu"]),
        "a2":tune.choice(["gelu","leaky_relu"]),
        "a3":tune.choice(["relu", "gelu"]),
        "batch_size": tune.choice([ 32, 64, 128]),
        }
```

Listing 21: Config

3.6 Main function

Almost everything happens in this function. Note that this is where the functions are finally one by one called, and the optimization of the hyper-parameters begins. In this section, I use the search algorithm of OptunaSearch[6], which is based on Bayesian optimization. In addition, ASHAScheduler [7] as a scheduler plays an active role in reducing computational costs for me.

```
from ray.tune import CLIReporter
from functools import partial
def main(num_samples=10, max_num_epochs=100, gpus_per_trial=2):

    config = {
      "l1": tune.choice([2**6,2**7,2**8]),
      "l2": tune.choice([2**6,2**7,2**8]),
      "l3": tune.choice([2**6,2**7,2**8]),
      "lr": tune.quniform(0.0001, 0.1,0.0001),
      "drop_out_ratio1": tune.quniform(0.3, 0.65,0.01),
      "drop_out_ratio2": tune.quniform(0.01, 1,0.01),
      "a1":tune.choice(["relu","leaky_relu","silu"]),
      "a2":tune.choice(["gelu","leaky_relu"]),
      "a3":tune.choice(["relu", "gelu"]),
      "batch_size": tune.choice([ 32, 64, 128]),
    }

    from ray.tune.suggest.optuna import OptunaSearch

    from ray.tune.suggest import ConcurrencyLimiter
    search_alg = OptunaSearch(
       metric="score", #or accuracy, etc.
       mode="max", #or max
       seed = 42,
       )
    search_alg = ConcurrencyLimiter(search_alg, max_concurrent=10)

    scheduler = ASHAScheduler(
        metric ="score",
        mode="max",
        max_t=max_num_epochs,
        reduction_factor=2,
        grace_period=4,
        brackets=5
        )

    reporter = CLIReporter(
        metric_columns=["score","val_score","kf_score","training_iteration"]
        )

    # wrap data loading and training for tuning using `partial`
    # (note that there exist other methods for this purpose)
    max_iter=max_num_epochs
    result = tune.run(
        partial(train_Bbbp,max_iter=max_iter),
        scheduler=scheduler,
        search_alg=search_alg,
        num_samples=num_samples,
        config=config,
        verbose=3,
        checkpoint_score_attr="score",
        checkpoint_freq=0,
        keep_checkpoints_num=1,
        progress_reporter=reporter,
        resources_per_trial={"cpu": 1, "gpu": gpus_per_trial},
        stop={"training_iteration": max_iter},
        )

    best_trial = result.get_best_trial("score", "max", "last")
    print("Best trial config: {}".format(best_trial.config))
    print("Average ROC_AUC score of the chosen model for different validation sets in 4-fold cross validation: {}".format(best_trial.last_result["kf_score"]))
    print("Best trial final validation ROC_AUC:    -- {} ".format(best_trial.last_result["val_score"]))
    print("Best trial final score:             -- {}".format(best_trial.last_result["score"]))

    if ray.util.client.ray.is_connected():
        # If using Ray Client, we want to make sure checkpoint access
        # happens on the server. So we wrap `test_best_model` in a Ray task.
        # We have to make sure it gets executed on the same node that
        # ``tune.run`` is called on.
        from ray.util.ml_utils.node import force_on_current_node
        remote_fn = force_on_current_node(ray.remote(test_best_model))
        ray.get(remote_fn.remote(best_trial))
    else:
       best_trial.config, best_trained_model= test_best_model(best_trial)
    return best_trial.config, best_trained_model

configuration,Bneuralnetwork=main(num_samples=100, max_num_epochs=14, gpus_per_trial=0)
```

Listing 22: Main function

```
Best trial config: {'l1': 256, 'l2': 256, 'l3': 256, 'lr': 0.00023721028804998172, 'drop_out_ratio1': 0.5269833252256684, 'drop_out_ratio2': 0.012888067179906312, 'a1': 'relu', 'a2': 'leaky_relu', 'a3': 'relu', 'batch_size': 128}
```

```
Average ROC_AUC score of the chosen model for different validation sets in 4-fold cross validation: 0.9841849148418491
Best trial final validation ROC_AUC:0.9066445239312796
Best trial final score:0.9001562145698741
Best trial test set score:0.9153247029986053
```

Listing 23: Minimized output

```
from ray.tune import CLIReporter
from functools import partial
def main(num_samples=10, max_num_epochs=100, gpus_per_trial=2):

    config = {
      "l1": tune.choice([2**6,2**7,2**8]),
      "l2": tune.choice([2**6,2**7,2**8]),
      "l3": tune.choice([2**6,2**7,2**8]),
      "lr": tune.quniform(0.0001, 0.1,0.0001),
      "drop_out_ratio1": tune.quniform(0.3, 0.65,0.01),
      "drop_out_ratio2": tune.quniform(0.01, 1,0.01),
      "a1":tune.choice(["relu","leaky_relu","silu"]),
      "a2":tune.choice(["gelu","leaky_relu"]),
      "a3":tune.choice(["relu", "gelu"]),
      "batch_size": tune.choice([ 32, 64, 128]),
     }

    from ray.tune.suggest.optuna import OptunaSearch

    from ray.tune.suggest import ConcurrencyLimiter
    search_alg = OptunaSearch(
       metric="score", #or accuracy, etc.
       mode="max", #or max
       seed = 42,
       )
    search_alg = ConcurrencyLimiter(search_alg, max_concurrent=10)

    scheduler = ASHAScheduler(
        metric ="score",
        mode="max",
        max_t=max_num_epochs,
        reduction_factor=2,
        grace_period=4,
        brackets=5
        )

    reporter = CLIReporter(
        metric_columns=["score","val_score","kf_score","training_iteration"]
        )

    # wrap data loading and training for tuning using `partial`
    # (note that there exist other methods for this purpose)
    max_iter=max_num_epochs
    result = tune.run(
        partial(train_HIV,max_iter=max_iter),
        scheduler=scheduler,
        search_alg=search_alg,
        num_samples=num_samples,
        config=config,
        verbose=3,
        checkpoint_score_attr="score",
        checkpoint_freq=0,
        keep_checkpoints_num=1,
        progress_reporter=reporter,
        resources_per_trial={"cpu": 2, "gpu": gpus_per_trial},
        stop={"training_iteration": max_iter},
        )

    best_trial = result.get_best_trial("score", "max", "last")
    print("Best trial config: {}".format(best_trial.config))
    print("Average ROC_AUC score of the chosen model for different validation sets in 4-fold cross validation: {}".format(best_trial.last_result["kf_score"]))
    print("Best trial final validation ROC_AUC:   -- {} ".format(best_trial.last_result["val_score"]))
    print("Best trial final score:             -- {}".format(best_trial.last_result["score"]))

    if ray.util.client.ray.is_connected():
        # If using Ray Client, we want to make sure checkpoint access
        # happens on the server. So we wrap `test_best_model` in a Ray task.
        # We have to make sure it gets executed on the same node that
        # ``tune.run`` is called on.
        from ray.util.ml_utils.node import force_on_current_node
        remote_fn = force_on_current_node(ray.remote(test_best_model))
        ray.get(remote_fn.remote(best_trial))
    else:
       best_trial.config, best_trained_model=test_best_model(best_trial)
    return best_trial.config, best_trained_model

configuration,Bneuralnetwork=main(num_samples=10, max_num_epochs=8, gpus_per_trial=0)
```

Listing 24: Main function

```
Best trial config: {'l1': 256, 'l2': 64, 'l3': 256, 'lr': 0.00032476735706274504, 'drop_out_ratio1': 0.32254266574935124, 'drop_out_ratio2': 0.7912456325487695, 'a1': 'relu', 'a2': 'leaky_relu', 'a3': 'relu', 'batch_size': 128}
Average ROC_AUC score of the chosen model for different validation sets in 4-fold cross validation:0.92654115475632112
Best trial final validation ROC_AUC:0.810365456975426
Best trial final score:0.799459873651485
Best trial test set score:0.81422115885625425
```

Listing 25: Minimized output

```
from ray.tune import CLIReporter
from functools import partial
def main(num_samples=10, max_num_epochs=100, gpus_per_trial=2):

    config = {
      "l1": tune.choice([2**6,2**7,2**8]),
      "l2": tune.choice([2**6,2**7,2**8]),
      "l3": tune.choice([2**6,2**7,2**8]),
      "lr": tune.quniform(0.0001, 0.1,0.0001),
      "drop_out_ratio1": tune.quniform(0.3, 0.65,0.01),
      "drop_out_ratio2": tune.quniform(0.01, 1,0.01),
      "a1":tune.choice(["relu","leaky_relu","silu"]),
      "a2":tune.choice(["gelu","leaky_relu"]),
      "a3":tune.choice(["relu", "gelu"]),
      "batch_size": tune.choice([ 32, 64, 128]),
     }

```

```
    from ray.tune.suggest.optuna import OptunaSearch

    from ray.tune.suggest import ConcurrencyLimiter
    search_alg = OptunaSearch(
       metric="loss", #or accuracy, etc.
       mode="min", #or max
       seed = 42,
       )
    search_alg = ConcurrencyLimiter(search_alg,
    max_concurrent=10)

    scheduler = ASHAScheduler(
        metric ="loss",
        mode="min",
        max_t=max_num_epochs,
        reduction_factor=2,
        grace_period=4,
        brackets=5
        )

    reporter = CLIReporter(
        metric_columns=["loss","val_loss","kf_loss" ,"
    training_iteration"]
        )

    # wrap data loading and training for tuning using
    `partial`
    # (note that there exist other methods for this
    purpose)
    max_iter=max_num_epochs
    result = tune.run(
        partial(train_freesolv,max_iter=max_iter),
        scheduler=scheduler,
        search_alg=search_alg,
        num_samples=num_samples,
        config=config,
        verbose=3,
        checkpoint_score_attr="loss",
        checkpoint_freq=0,
        keep_checkpoints_num=1,
        progress_reporter=reporter,
        resources_per_trial={"cpu": 1, "gpu":
    gpus_per_trial},
        stop={"training_iteration": max_iter},
        )



    best_trial = result.get_best_trial("loss", "min",
    "last")
    print("Best trial config: {}".format(best_trial.
    config))
    print("Average MSE loss of the chosen model for
    different validation sets in 4-fold cross
    validation: {}".format(best_trial.last_result["
    kf_loss"]))
    print("Best trial final validation MSEloss:
       - - {} ".format(best_trial.last_result["
    val_loss"]))
    print("Best trial final loss:              {}".
    format(best_trial.last_result["loss"]))

    if ray.util.client.ray.is_connected():
        # If using Ray Client, we want to make sure
    checkpoint access
        # happens on the server. So we wrap `
    test_best_model` in a Ray task.
        # We have to make sure it gets executed on the
     same node that
        # ``tune.run`` is called on.
        from ray.util.ml_utils.node import
    force_on_current_node
        remote_fn = force_on_current_node(ray.remote(
    test_best_model))
        ray.get(remote_fn.remote(best_trial))
    else:
        best_trial.config, best_trained_model=
    test_best_model(best_trial)
    return best_trial.config, best_trained_model

configuration,Bneuralnetwork=main(num_samples=15,
    max_num_epochs=30, gpus_per_trial=0)
```

Listing 26: Main function

```
Best trial config: {'l1': 64, 'l2': 64, 'l3': 256, 'lr
    ': 0.004048778818153414, 'drop_out_ratio1':
    0.3383871708118728, 'drop_out_ratio2':
    0.4616014195190429, 'a1': 'relu', 'a2': '
    leaky_relu', 'a3': 'gelu', 'batch_size': 64}
Average MSE loss of the chosen model for different
    validation sets in 4-fold cross validation
    :0.9645641363221736
Best trial final validation MSEloss:1.13998745615874
Best trial final loss:1.3503651421559874
Best trial test set loss:1.190654159875324
```

Listing 27: Main function

4 FINAL RESULTS

Now we have the optimal hyper-parameters. But have only the initial weights of the neural network caused acceptable results? To answer this question, we again train the neural network from the beginning and check the results on all three datasets.

```
def final_traing(config,max_iter=14):
    net = NeuralNetwork(np.shape(feature_selection[0])[0],config["l1"],config["l2"],config["l3"],config)


    device = "cpu"
    if torch.cuda.is_available():
        device = "cuda:0"
        if torch.cuda.device_count() > 1:
            net = nn.DataParallel(net)
    net.to(device)

    #Define my loss function and optimizer
    criterion=nn.BCELoss()
    optimizer=torch.optim.Adam(net.parameters(), lr=config["lr"])

    for epoch in range(max_iter):
      running_loss = 0.0
      epoch_steps = 0
      for i, data in enumerate(trainloader(config), 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        # zero the parameter gradients
        optimizer.zero_grad()
        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        # print statistics
        running_loss += loss.item()
        epoch_steps += 1
        if i % 2000 == 1999:  # print every 2000 mini-batches
          print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss / epoch_steps))
          running_loss = 0.0

        # Validation score
    return net

neurlnet=final_traing(configuration,max_iter=14)
with torch.no_grad():
    y_predicted=neurlnet(X_test_)
    y_predicted_cls=y_predicted.round()
    acc= y_predicted_cls.eq(y_test_).sum()/float(y_test_.shape[0])

    #print(f'accuracy={acc:.4f}')
    a=confmat( y_predicted_cls.int(),y_test_.int())
    print(f'accuracy={acc:.19f} balanced_accuracy_score={balanced_accuracy_score(y_predicted_cls.int(),y_test_.int()):.19f} ROC_AUC={compute_score(neurlnet, testloader(configuration), device="cpu")   :.19f}')
    print(a)
```

Listing 28: Retraining the neural network

```
accuracy=0.8970588235294118
balanced_accuracy_score=0.8622641509433963
ROC_AUC=0.9235299485542957124 8
tensor([[ 36  12],
        [  9 147]])
```

Listing 29: Final results

```
def final_traing(config,max_iter=14):
    net = NeuralNetwork(np.shape(feature_selection[0])[0],config["l1"],config["l2"],config["l3"],config)


    device = "cpu"
    if torch.cuda.is_available():
        device = "cuda:0"
        if torch.cuda.device_count() > 1:
            net = nn.DataParallel(net)
    net.to(device)

    #Define my loss function and optimizer
    criterion=nn.BCELoss()
    optimizer=torch.optim.Adam(net.parameters(), lr=config["lr"])

    for epoch in range(max_iter):
      running_loss = 0.0
      epoch_steps = 0
      for i, data in enumerate(trainloader(config), 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        # zero the parameter gradients
        optimizer.zero_grad()
        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        # print statistics
        running_loss += loss.item()
        epoch_steps += 1
        if i % 2000 == 1999:  # print every 2000 mini-batches
          print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss / epoch_steps))
          running_loss = 0.0

        # Validation score
    return net

neurlnet=final_traing(configuration,max_iter=9)
with torch.no_grad():
    y_predicted=neurlnet(X_test_)
    y_predicted_cls=y_predicted.round()
    acc= y_predicted_cls.eq(y_test_).sum()/float(y_test_.shape[0])

    #print(f'accuracy={acc:.4f}')
    a=confmat( y_predicted_cls.int(),y_test_.int())
    print(f'accuracy={acc:.19f} balanced_accuracy_score={balanced_accuracy_score(y_predicted_cls.int(),y_test_.int()):.19f} ROC_AUC={compute_score(neurlnet, testloader(configuration), device="cpu")   :.19f}')
    print(a)
```

Listing 30: Retraining the neural network

```
accuracy=0.9720398735716023
balanced_accuracy_score=0.8807050684974516
```

```
ROC_AUC=0.83159499309512508965
tensor([[3961,   10],
        [ 105,   37]])
```

Listing 31: Final results

```
def final_traing(config,max_iter=14):
    net = NeuralNetwork(np.shape(feature_selection[0])[0],config["l1"],config["l2"],config["l3"],config)


    device = "cpu"
    if torch.cuda.is_available():
        device = "cuda:0"
        if torch.cuda.device_count() > 1:
            net = nn.DataParallel(net)
    net.to(device)

    #Define my loss function and optimizer
    criterion=nn.MSELoss()
    optimizer=torch.optim.Adam(net.parameters(), lr=config["lr"])

    for epoch in range(max_iter):
      running_loss = 0.0
      epoch_steps = 0
      for i, data in enumerate(trainloader(config), 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        # zero the parameter gradients
        optimizer.zero_grad()
        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        # print statistics
        running_loss += loss.item()
        epoch_steps += 1
        if i % 2000 == 1999:  # print every 2000 mini-batches
          print("[%d, %5d] loss: %.3f" % (epoch + 1, i + 1,running_loss / epoch_steps))
          running_loss = 0.0
    return net

neurlnet=final_traing(configuration,max_iter=30)
loss=compute_loss(neurlnet, testloader(config), device="cpu")
print(f'loss={loss:.10f}')
```

Listing 32: Retraining the neural network

```
loss=1.24510621941294373
```

Listing 33: Final results

5 CONCLUSION

Among these three datasets, I have examined the results of the Bbbp dataset more than the other datasets. Therefore, I thought it would be good to put the results of working with this data in the form of a table.

		ROC_AUC	Search Algorithm	Changes compared to the previous version	Optimizer
1	Primary neural network	0.8920	-	-	SGD
2	hyper-parameter tuning with ray tune	0.9066	Random search	Using Ray tune	SGD
3	hyper-parameter tuning with ray tune	0.9110	Random search	Introducing the activation functions as hyper-parameters	SGD
4	hyper-parameter tuning with ray tune	0.9155	Random search	Change the optimizer	Adam
5	hyper-parameter tuning with ray tune	0.9222	Optuna	Change the search algorithm	Adam
6	hyper-parameter tuning with ray tune	0.9054	Optuna	add semi_CV -Changing the score criterion	Adam
7	hyper-parameter tuning with ray tune	0.91052	Optuna	Changing the max possible iterations	Adam
8	hyper-parameter tuning with ray tune	0.9074	Optuna	Changing the score criterion	Adam
9	hyper-parameter tuning with ray tune	**0.9235**	Optuna	Limit the search space & New neural network model training with the optimized hyper-parameters	Adam

In addition, referring to the report I wrote earlier in the applied machine learning course about the HIV dataset is not harmful. My meter in that project was balanced accuracy. In the following table, you can have a better comparison of the final accuracy of these two data sets. [7]

	Best algorithm	Dimension reduction	Search Algorithm	Balanced accuracy	Sensitivity	Specificity
1	KNN	PCA	Gridsearchcv	0.7326	0.4861	**0.9791**
				Accuracy= 0.96131		
2	Neural Network	featurewiz	Optuna	**0.8807**	**0.7872**	0.9741
				Accuracy= 0.97203		

I would have loved to partition the data set like the article FunQG: Molecular Representation Learning Via Quotient Graphs using Scaffold Split. Still, the little time I had on this project prevented me. I hope to have this opportunity in the future.[8]

NOTES

[1]Matplotlib library has been used in most visualizations of this report, either directly or indirectly. [9]

[2]I am not sure that using `stratify` in `train_test_split` [10] does not cause any information leakage. However, this is a standard method, and I will use this method in this project.

[3]Cumulative Distribution Function

[4]Receiver Operating characteristic curve

[5]Area under the (ROC) Curve

[6]Mean Squared Error

[7]In Tune, some hyper-parameter optimization algorithms are written as "scheduling algorithms". These Trial Schedulers can early terminate bad trials, pause trials, clone trials, and alter hyper-parameters of a running trial.[11]

ACKNOWLEDGEMENTS

I would like to express my special gratitude to my dear professors, **Dr.Zahra Taheri** and **Dr.Bijan Ahmadi**. Both of these dignitaries gave me the golden opportunity to do this wonderful project on the topic of hyper-parameter optimization, which also helped me do a lot of research that has led to learning fascinating and valuable information in this field.

It is also appropriate to mention **Professor Andrew Ng**, who introduced me to the structure of neural networks through his online training courses. In addition, let me say that I learned to work with the PyTorch library through Mr. *Patrick Loeber*'s YouTube channel.

I would like to mention that I consulted with my good friends *Dr. Behrad Taghi Beiglo* and *Kian Adib* in this project. I should also thank my good teammate, Mr. *Mahmoud Hashempour*.

REFERENCES

[1] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[2] Kevin Yang, Kyle Swanson, Wengong Jin, Connor Coley, Philipp Eiden, Hua Gao, Angel Guzman-Perez, Timothy Hopper, Brian Kelley, Miriam Mathea, et al. Analyzing learned molecular representations for property prediction. *Journal of chemical information and modeling*, 59(8):3370–3388, 2019.

[3] AutoViML. Autoviml/featurewiz: Use advanced feature engineering strategies and select best features from your data set with a single line of code.

[4] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006. ROC Analysis in Pattern Recognition.

[5] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[6] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[7] Arash Sajjadi. *HIV project report*. 2022.

[8] Hossein Hajiabolhassan, Zahra Taheri, Ali Hojatnia, and Yavar Taheri Yeganeh. Funqg: Molecular representation learning via quotient graphs, 2022.

[9] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[11] trial schedulers (tune.schedulers) - ray 1.13.0.