HYPER-PARAMETER OPTIMIZATION PROJECT

Authors: Mohammad javad Abbas pour & Ahmad Yazdani

Supervisor: Dr.Taheri

Optimization in data science

Shahid Beheshti University

July , 2022

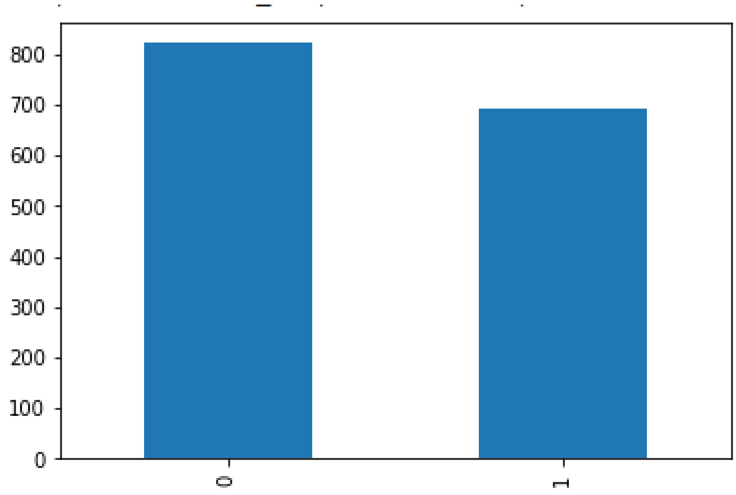
Abstract

This project is a hyper-parameter-optimization of the neural network, which has been implemented on Bace molecular data set . I use ray , pytorch ,schedulers and search-algorithms and more objects in this project.

Introduction of dataset

Type of data set is molecular. The number of records is 1513 , target variables are 0 or 1 and the number of features is 200.

Distribution of labels plot:



In this case the distribution of labels are balance. There is an balance between the class of one and zero labels in Bace dataset.

Preprocessing

Data partitioning, Feature selection, and Missing are the important things for prepare the data for final processing.

Import libraries

```
%%capture
try:
    import ray
except:
    !pip install -U ray
    import ray
try:
    import optuna
except:
    %pip install optuna
    import optuna
    %%capture
try:
    from featurewiz import featurewiz
except:
    !pip install featurewiz==0.1.70

import numpy as np
import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset
from pandas import read_csv
from sklearn.impute import SimpleImputer
import io
import matplotlib as mpl
import matplotlib.pyplot as plt
import random
from functools import partial
import os
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import random_split
from torchsummary import summary
from ray import tune
from ray.tune import CLIReporter
from ray.tune.schedulers import ASHAScheduler
```

Data reading

```
from google.colab import files
uploaded = files.upload()

dataX = pd.read_csv(io.BytesIO(uploaded['bace_global_cdf_rdkit.csv']))
dataY = pd.read_csv(io.BytesIO(uploaded1['bace(lables).csv']))
```

Data splitting

```
from sklearn.model_selection import train_test_split
Y = dataY.iloc[:, 1]
X = dataX.iloc[:,1:201]
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.15,random_state=1234)
```

Feature selection

I used Corrolation matrix and featurewiz library to select important features of Bace data set, then I've observed that featurewiz is better.

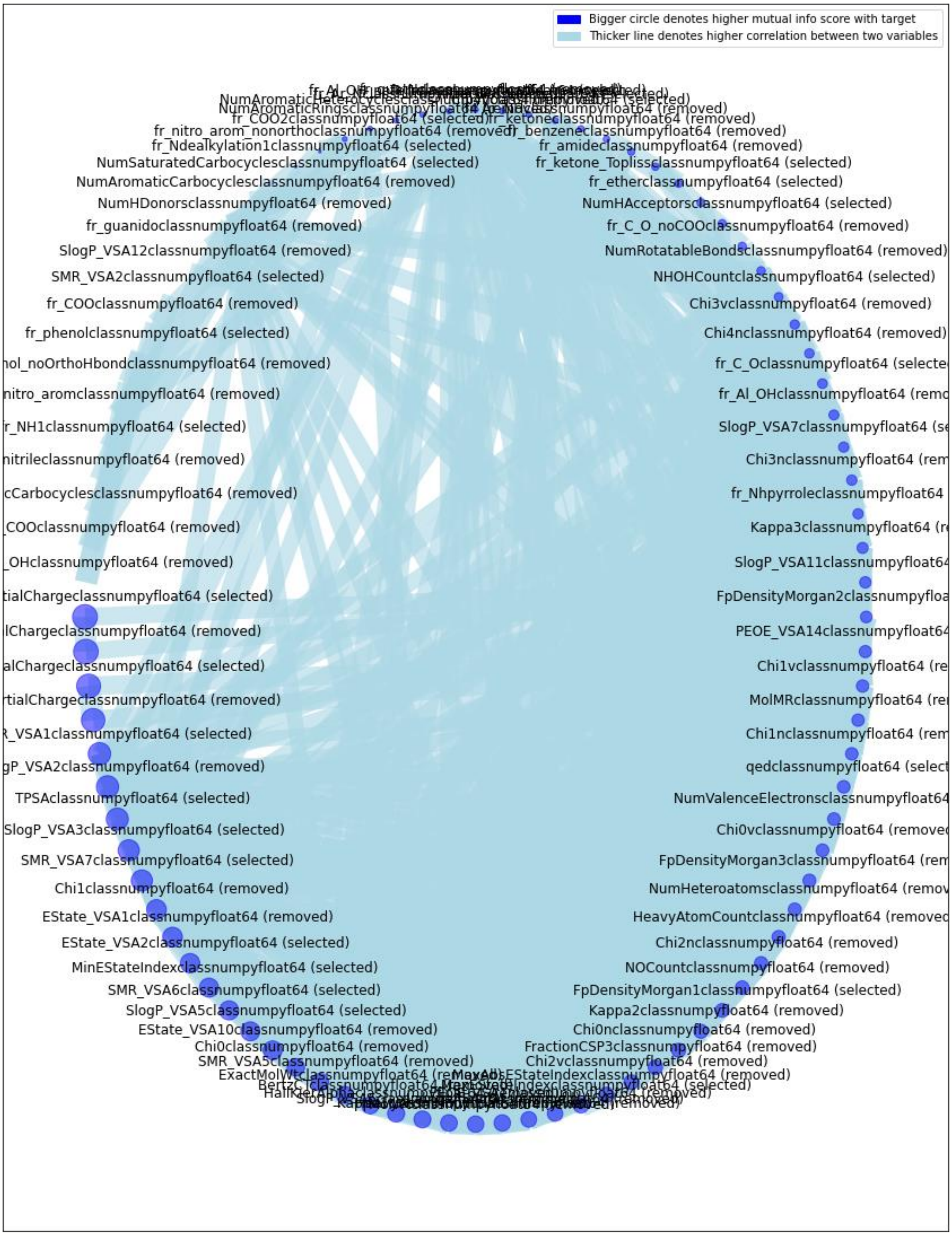
```
from featurewiz import FeatureWiz
features = FeatureWiz(corr_limit=0.70, feature_engg='', category_encoders='', dask_xgboost_flag=False, nrows=None, verbose=2)
X_train = features.fit_transform(X_train, Y_train)
X_test = features.transform(X_test)
features.features  ### provides the list of selected features ###

###################################################################################
############          F A S T   F E A T U R E  E N G G    A N D    S E L E C T I O N ! ########
# Be judicious with featurewiz. Don't use it to create too many un-interpretable features! #
###################################################################################
Skipping feature engineering since no feature_engg input...
Skipping category encoding since no category encoders specified in input...
**INFO: featurewiz can now read feather formatted files. Loading train data...
    Shape of your Data Set loaded: (1286, 201)
    Loaded train data. Shape = (1286, 201)
    Some column names had special characters which were removed...
No test data filename given...
###############################################################################
######################## C L A S S I F Y I N G  V A R I A B L E S
```

```
####################
###########################################################################
#############
Classifying variables in data set...
    200 Predictors classified...
        33 variable(s) to be removed since ID or low-information variables
      more than 33 variables to be removed; too many to print...
train data shape before dropping 33 columns = (1286, 201)
      train data shape after dropping columns = (1286, 168)
    Converted pandas dataframe into a Dask dataframe ...
GPU active on this device
    Tuning XGBoost using GPU hyper-parameters. This will take time...
    After removing redundant variables from further processing, features
left = 167
No interactions created for categorical vars since feature engg does not
specify it
#### Single_Label Binary_Classification problem ####
###########################################################################
#############
#####  Searching for Uncorrelated List Of Variables (SULOV) in 167
features ############
###########################################################################
#############
    there are no null values in dataset...
    Removing (62) highly correlated variables:
```

How SULOV Method Works by Removing Highly Correlated Features

In SULOV, we repeatedly remove features with lower mutual info scores among highly correlated pairs (see figure),
SULOV selects the feature with higher mutual info score related to target when choosing between a pair.

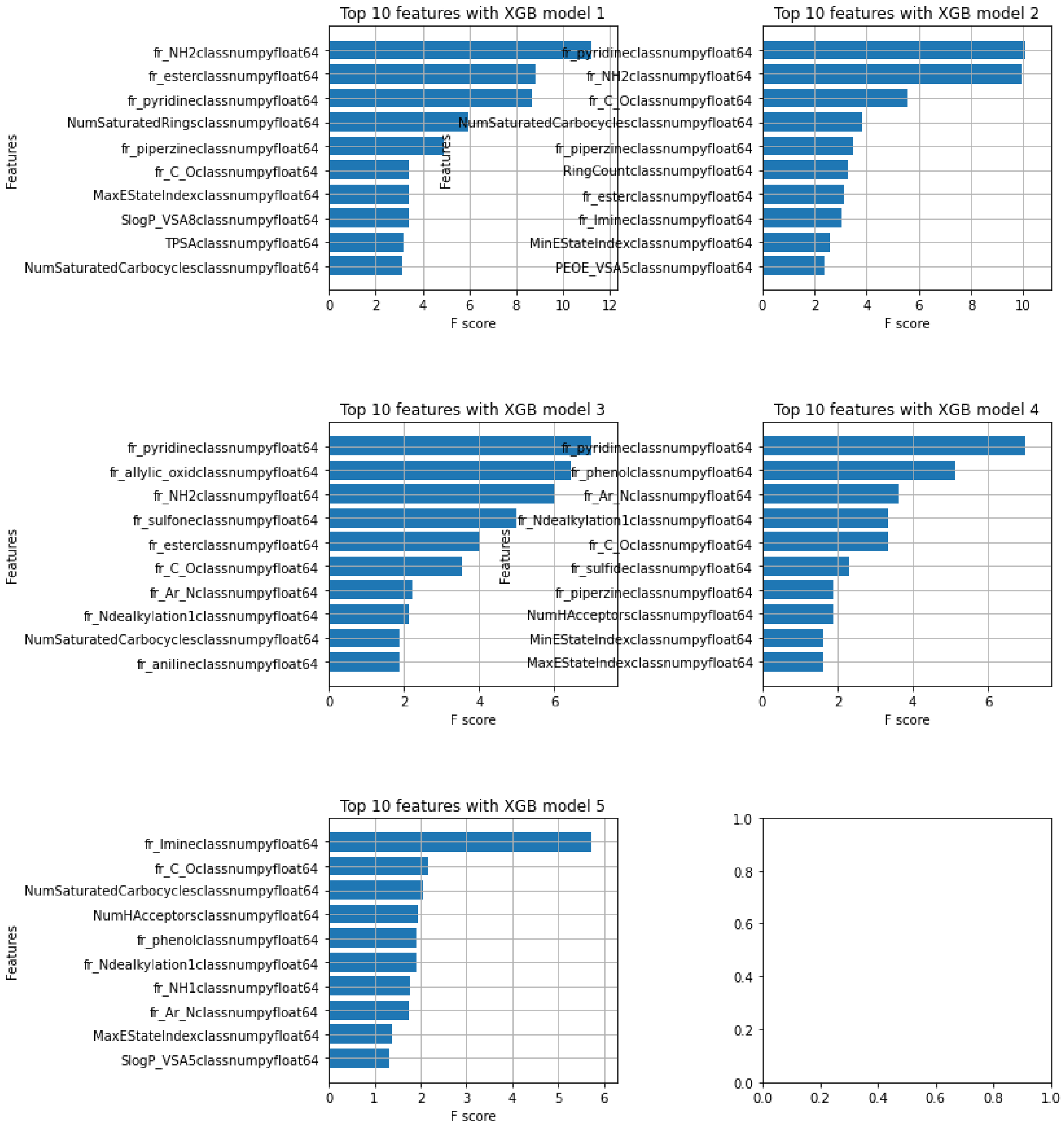


```
Time taken for SULOV method = 4 seconds
    Adding 0 categorical variables to reduced numeric variables  of 105
Final list of selected vars after SULOV = 105
Readying dataset for Recursive XGBoost by converting all features to
numeric...
##########################################################################
#############
#####    R E C U R S I V E   X G B O O S T : F E A T U R E   S E L E C T I
O N  #######
##########################################################################
#############
    using regular XGBoost
Train and Test loaded into Dask dataframes successfully after feature_engg
completed
Current number of predictors = 105
    XGBoost version: 1.6.1
Number of booster rounds = 100
        using 105 variables...
            Time taken for regular XGBoost feature selection = 2 seconds
        using 84 variables...
            Time taken for regular XGBoost feature selection = 5 seconds
        using 63 variables...
            Time taken for regular XGBoost feature selection = 7 seconds
        using 42 variables...
            Time taken for regular XGBoost feature selection = 8 seconds
        using 21 variables...
            Time taken for regular XGBoost feature selection = 9 seconds

            Total time taken for XGBoost feature selection = 12 seconds
##########################################################################
#############
#####          F E A T U R E   S E L E C T I O N   C O M P L E T E D
#######
##########################################################################
#############
Selected 56 important features. Too many to print...

    Time taken for feature selection = 15 seconds
Returning 2 dataframes: dataname and test_data with 56 important features.
    Time taken to create entire pipeline = 17 second(s)

["('fr_NH2', <class 'numpy.float64'>)",
 "('fr_ester', <class 'numpy.float64'>)",
 "('fr_pyridine', <class 'numpy.float64'>)",
 "('NumSaturatedRings', <class 'numpy.float64'>)",
 "('fr_piperzine', <class 'numpy.float64'>)",
 "('fr_C_O', <class 'numpy.float64'>)",
 "('MaxEStateIndex', <class 'numpy.float64'>)",
 "('SlogP_VSA8', <class 'numpy.float64'>)",
 "('TPSA', <class 'numpy.float64'>)",
 "('NumSaturatedCarbocycles', <class 'numpy.float64'>)",
 "('PEOE_VSA4', <class 'numpy.float64'>)",
 "('SMR_VSA10', <class 'numpy.float64'>)",
 "('fr_aryl_methyl', <class 'numpy.float64'>)",
 "('NumHAcceptors', <class 'numpy.float64'>)",
 "('MinEStateIndex', <class 'numpy.float64'>)",
 "('SlogP_VSA5', <class 'numpy.float64'>)",
 "('MaxPartialCharge', <class 'numpy.float64'>)",
 "('SlogP_VSA4', <class 'numpy.float64'>)",
 "('VSA_EState10', <class 'numpy.float64'>)",
 "('PEOE_VSA12', <class 'numpy.float64'>)",
 "('SlogP_VSA1', <class 'numpy.float64'>)",
 "('PEOE_VSA11', <class 'numpy.float64'>)",
 "('PEOE_VSA2', <class 'numpy.float64'>)",
 "('SMR_VSA6', <class 'numpy.float64'>)",
 "('fr_bicyclic', <class 'numpy.float64'>)",
 "('RingCount', <class 'numpy.float64'>)",
 "('fr_Imine', <class 'numpy.float64'>)",
 "('PEOE_VSA5', <class 'numpy.float64'>)",
 "('fr_phenol', <class 'numpy.float64'>)",
 "('fr_NH0', <class 'numpy.float64'>)",
 "('fr_para_hydroxylation', <class 'numpy.float64'>)",
 "('fr_allylic_oxid', <class 'numpy.float64'>)",
 "('fr_sulfone', <class 'numpy.float64'>)",
 "('fr_Ar_N', <class 'numpy.float64'>)",
 "('fr_Ndealkylation1', <class 'numpy.float64'>)",
 "('fr_aniline', <class 'numpy.float64'>)",
 "('fr_Ndealkylation2', <class 'numpy.float64'>)",
 "('fr_oxazole', <class 'numpy.float64'>)",
 "('fr_imidazole', <class 'numpy.float64'>)",
 "('qed', <class 'numpy.float64'>)",
 "('fr_unbrch_alkane', <class 'numpy.float64'>)",
 "('fr_sulfide', <class 'numpy.float64'>)",
 "('SlogP_VSA3', <class 'numpy.float64'>)",
 "('SMR_VSA7', <class 'numpy.float64'>)",
 "('MaxAbsPartialCharge', <class 'numpy.float64'>)",
 "('NHOHCount', <class 'numpy.float64'>)",
 "('Chi4v', <class 'numpy.float64'>)",
 "('SMR_VSA1', <class 'numpy.float64'>)",
 "('fr_ketone_Topliss', <class 'numpy.float64'>)",
 "('fr_NH1', <class 'numpy.float64'>)",
 "('fr_ether', <class 'numpy.float64'>)",
 "('FpDensityMorgan1', <class 'numpy.float64'>)",
 "('SlogP_VSA7', <class 'numpy.float64'>)",
 "('fr_COO2', <class 'numpy.float64'>)",
 "('SMR_VSA2', <class 'numpy.float64'>)",
 "('fr_Nhpyrrole', <class 'numpy.float64'>)"]
```



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

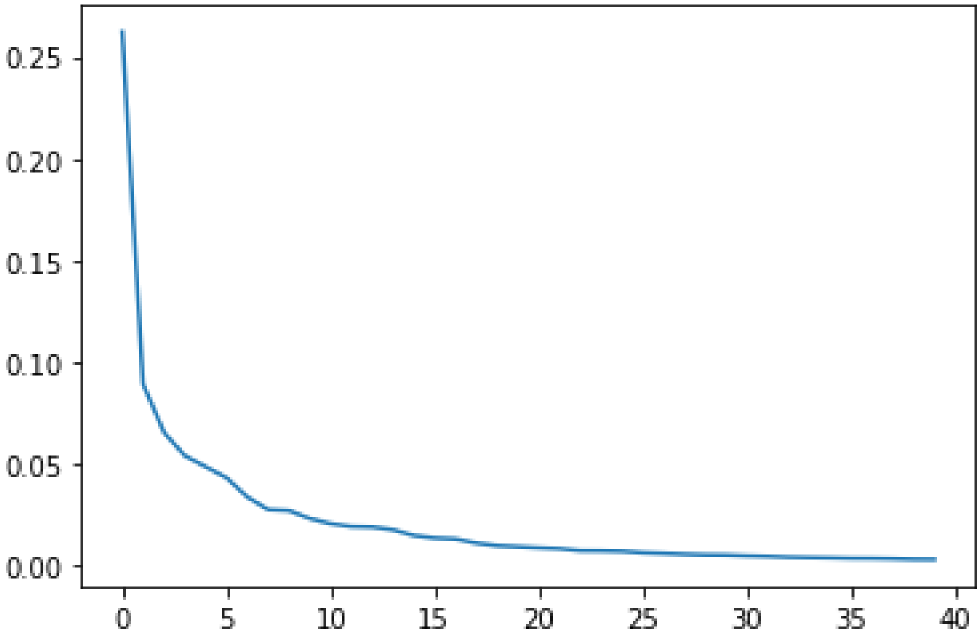
56 features were selected as influencing features on the target variable in the Bace dataset**.**

PCA

I have used PCA but the result of our model is not good in this way, cause of that ignored it.

```
from sklearn.decomposition import PCA as sklearnPCA
pca = sklearnPCA(n_components=40)
```

```
pca.fit(X_train)
X_train=pca.transform(X_train)
X_test=pca.transform(X_test)
plt.plot(np.arange(40),pca.explained_variance_ratio_)
plt.show()
```



Scailing data

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
import random
np.random.seed(1234)

from sklearn import preprocessing
scaler_data = preprocessing.MinMaxScaler()
X_train = scaler_data.fit_transform(X_train)
X_test = scaler_data.transform(X_test)
scaler_labels = preprocessing.MinMaxScaler()
```

I have not used MinMaxScaler method on lables, because they are 0 or 1.

Transform to torch tensor

```
device = torch.device('cuda:0' if torch.cuda.is_available() el
se 'cpu')

tensor_x = torch.tensor(X_train, dtype=torch.float).to(device)

tensor_x2 = torch.tensor(X_test, dtype=torch.float).to(device)


tensor_y = torch.tensor(Y_train, dtype=torch.float).to(device)
tensor_y2 = torch.tensor(Y_test, dtype=torch.float).to(device)
 # create your dataset
trainset = TensorDataset(tensor_x, tensor_y)
testset = TensorDataset(tensor_x2,tensor_y2)
```

Loading the Data

```
def load_data(data_dir=None):
    return trainset, testset
```

Missing values

Bace data set has not any missing values.

Accuracy and error measurement

I used ROC_AUC score for Bace data set ,cause that is a classification project.

```
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, predic
tion), 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
```

Configurable neaural network

We can tune those parameters that are configurable(Learning rate ,Hidden dim,Number of layers, Activation functions).

Note: In this case I define a loop for assigning Hidden_dim in our Nueral Network model.

```
class Net(nn.Module):
    def __init__(self, config):
        super().__init__()

        self.config = config
        self.hidden_dim1 = int(self.config.get("hidden_dim1", 100))
        self.hidden_dim2 = int(self.config.get("hidden_dim2", 100))
        self.hidden_dim3 = int(self.config.get("hidden_dim3", 100))
        hidden_dim={}

        self.act1 = self.config.get("act1", "relu")
        self.act2 = self.config.get("act2", "relu")
        self.act3 = self.config.get("act3", "relu")

        self.linear1 = nn.Linear(200, self.hidden_dim1)
        self.linear2 = nn.Linear(self.hidden_dim1, self.hidden_dim2)
        self.linear3 = nn.Linear(self.hidden_dim2, self.hidden_dim3)
        self.linear4 = nn.Linear(self.hidden_dim3, 1)

        for i in range (4,num_layers):
          self.config.update("hidden_dim"+str(i):tune.quniform(150,300,
10))
          self.config.update("linear"+str(i):nn.Linear("hidden_dim"+str
(i-1), "hidden_dim"+str(i)))
          self.config.update("act"+str(i):tune.choice("relu","selu","ta
nh"))
        self.linear("num_layers") = nn.Linear(self.hidden_dim("num_laye
rs")), 1)
    @staticmethod
    def activation_func(act_str):
        if act_str=="tanh":
            return eval("torch."+act_str)
        elif act_str=="selu" or act_str=="relu":
            return eval("torch.nn.functional."+act_str)

    def forward(self, x):
        output = self.linear1(x)
        output = self.activation_func(self.act1)(output)
        output = self.linear2(output)
        output = self.activation_func(self.act2)(output)
        output = self.linear3(output)
        output = self.activation_func(self.act3)(output)
        output = self.linear4(output)
        output = self.linear("linear")(output)
        output = self.activation_func(self.act)(output)
        output =torch.sigmoid(output)
        predictY=output
        return predictY
```

The train function

We wrap the training script in a function "trainable_func".

As you can guess, the "config" parameter will receive the hyperparameters we would like to train with. The "checkpoint_dir" parameter is used to restore checkpoints. The "data_dir" specifies the directory where we load and store the data, so multiple runs can share the same data source.

In this function I defined a loss function (criterion=nn.BCELoss())

And an optimizer(SGD).

We also split the training data into a training and validation subset. We thus train on 70% of the data and calculate the validation loss on the remaining 15% . The batch sizes with which we iterate through the training and test sets are configurable as well.

Here we first save a checkpoint and then report some metrics back to Ray Tune. Specifically we send the validation loss and accuracy back to Ray Tune. Ray Tune can then use these metrics to decide which hyperparameter configuration lead to the best results. These metrics can also be used to stop bad performing trials early in order to avoid wasting resources on those trials.

Also, by saving the checkpoint we can later load the trained models and validate them.

```
def trainable_func(config, checkpoint_dir=None, data_dir=None,
 epochs=10):

    net = Net(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 a loss function
    '''
    ## Classification
    #criterion = nn.CrossEntropyLoss()
    criterion=nn.BCELoss()

    # Define an optimizer
    optimizer = torch.optim.SGD(net.parameters(), lr=config.get("lr",0.0003))

    if checkpoint_dir:
        model_state, optimizer_state = torch.load(
            os.path.join(checkpoint_dir, "checkpoint"))
        net.load_state_dict(model_state)
        optimizer.load_state_dict(optimizer_state)

    # Load data
    trainset, testset = load_data(data_dir)

    # Split the dataset into training and validation sets
    train_size = int(len(trainset) * 0.825)
    train_subset, val_subset = random_split(trainset, [train_size, len(trainset) - train_size])

    # Define data loaders (which combines a dataset and a sampler, and provides an iterable over the given dataset)
    trainloader = torch.utils.data.DataLoader(
        train_subset,
        batch_size=int(config.get("batch_size",32)),
        shuffle=True,
        num_workers=2)
    valloader = torch.utils.data.DataLoader(
        val_subset,
        batch_size=int(config.get("batch_size",32)),
        shuffle=True,
        num_workers=2)

    for epoch in range(epochs):  # loop over the dataset multiple times
        epoch_train_loss = 0.0
        # epoch_steps = 0
        net.train() # Prepare model for training
        for i, data in enumerate(trainloader):
            # 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()

            '''
            Compute train loss without scaling to print
            '''

        score = compute_score(net, valloader, device="cpu")

        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)
    print("Finished Training")
```

Test set score

Commonly the performance of a machine learning model is tested on a hold-out test set with data that has not been used for training the model. We also wrap this in a function.

I have defined this function for applying best model's score on the test set:

```
def test_score(config, net , device="cpu"):
    trainset, testset = load_data()

    testloader = torch.utils.data.DataLoader(testset, batch_size=int(config.get("batch_size",32)), shuffle=False, num_workers=2)
    best_trained_model=net
    criterion = nn.CrossEntropyLoss()

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

Configuring the search space

Ray Tune will now randomly sample a combination of parameters from these search spaces. It will then train a number of models in parallel and find the best performing one among these. We also use the ``ASHAScheduler" which will terminate bad performing trials early.

Main function

At the first, I should say that I have used some search algorithms and schedulers(You can see them on my github ,6nd version file)and choose the best ones.I used the search algorithm of OptunaSearch, which is based on Bayesian optimization. In addition, ASHAScheduler as a scheduler plays an active role in reducing computational costs for me.

This is a part where the functions are finally one by one called.The optimization of the hyper-parameters begins:

```
def main(num_samples=10, max_num_epochs=100, gpus_per_trial=2)
:

    # define data directory here if you want to load data from
 files
    data_dir = os.path.abspath("./data")
    load_data(data_dir)

    # define the search space of hyperparameters
    config = {
        "act1 ": tune.choice(["relu","tanh","selu"]),
        "act2" : tune.choice(["relu","tanh","selu"]),
        "act3" : tune.choice(["relu","tanh","selu"]),
        "l1": tune.choice([2**2,2**3,2**4,2**5,2**6,2**7,2**8]
), #tune.sample_from(lambda _: 2**np.random.randint(2, 8)),
        "l2": tune.choice([2**2,2**3,2**4,2**5,2**6,2**7,2**8]
), #tune.sample_from(lambda _: 2**np.random.randint(2, 8)),
        "l3": tune.choice([2**2,2**3,2**4,2**5,2**6,2**7,2**8]
), #tune.sample_from(lambda _: 2**np.random.randint(2, 8)),
        "lr": tune.quniform(0.0005, 0.001, 0.0001),
        "batch_size": tune.choice([8, 16, 32]),
        "hidden_dim1" : tune.quniform(150, 300, 10),
        "hidden_dim2" : tune.quniform(150, 300, 10),
        "hidden_dim3" : tune.quniform(150, 300, 10),
        "num_layers" :tune.uniform(4,10)
    }


    # Optuna search algorithm
    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(
        parameter_columns=["l1", "l2", "lr", "batch_size"],
        metric_columns=["score", "training_iteration"]
        )


    result = tune.run(
        partial(trainable_func, data_dir=data_dir, epochs=max_
num_epochs),
        scheduler=scheduler,
        search_alg=search_alg,
        num_samples=num_samples,
        config=config,
        verbose=2,
        checkpoint_score_attr="score",
        checkpoint_freq=0,
        keep_checkpoints_num=1,
        progress_reporter=reporter,
        resources_per_trial={"cpu": 0.5, "gpu": gpus_per_trial
},
        stop={"training_iteration": max_num_epochs},

        )

    best_trial = result.get_best_trial("score", "max", "last")
    print("Best trial config: {}".format(best_trial.config))
    print("Best trial final validation score: {}".format(
        best_trial.last_result["score"]))
    best_trained_model = Net(best_trial.config)

    device = "cpu"
    if torch.cuda.is_available():
        device = "cuda:0"
        if gpus_per_trial > 1:
            best_trained_model = nn.DataParallel(best_trained_
model)
    best_trained_model.to(device)

    best_checkpoint_dir = best_trial.checkpoint.value
    model_state, optimizer_state = torch.load(os.path.join(
        best_checkpoint_dir, "checkpoint"))
    best_trained_model.load_state_dict(model_state)

    test_score_value = test_score(best_trial.config, best_trained_model
, device)
    print("Best trial test set score: {}".format(test_score_value))

if __name__ == "__main__":
    main(num_samples=100, max_num_epochs=10, gpus_per_trial=0)
```

Best trial final **validation score**: 0.8190438871473355

Best trial **test set score**: 0.808862171931479

```
Best_trial_config = {'act1 ': 'tanh', 'act2': 'tanh', 'act3':
'tanh''lr': 0.0008, 'batch_size': 16, 'hidden_dim1': 130.0, 'h
idden_dim2': 170.0, 'hidden_dim3': 100.0,'num_layares':4}
```

Train the neural network from the beginning and check the results

```
import random
random.seed(1234)
Best_trial_config = {'act1 ': 'tanh', 'act2': 'tanh', 'act3':
'tanh''lr': 0.0008, 'batch_size': 16, 'hidden_dim1': 130.0, 'h
idden_dim2': 170.0, 'hidden_dim3': 100.0,'num_layares':4}
epochs = 10
config = {'act1 ': 'tanh', 'act2': 'tanh', 'act3': 'tanh''lr':
 0.0008, 'batch_size': 16, 'hidden_dim1': 130.0, 'hidden_dim2'
: 170.0, 'hidden_dim3': 100.0,'num_layares':4}

net = Net(Best_trial_config)
criterion = nn.CrossEntropyLoss()

    # # Define an optimizer
optimizer = optim.Adam(net.parameters(), lr=config.get("lr",0.
0003))
```

```
trainset, testset = load_data()

    # Split the dataset into training and validation sets
train_size = int(len(trainset) * 0.825)
train_subset, val_subset = random_split(trainset, [train_size, len(trainset) - train_size])

    # Define data loaders (which combines a dataset and a sampler, and provides an iterable over the given dataset)
trainloader = torch.utils.data.DataLoader(
    train_subset,
    batch_size=int(config.get("batch_size",32)),
    shuffle=False,
    num_workers=2)
valloader = torch.utils.data.DataLoader(
    val_subset,
    batch_size=int(config.get("batch_size",32)),
    shuffle=False,
    num_workers=2)
for epoch in range(epochs):  # loop over the dataset multiple times
        epoch_train_loss = 0.0
        # epoch_steps = 0
        net.train() # Prepare model for training
        for i, data in enumerate(trainloader):
            # 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()

            '''
            Compute train loss without scaling to print
            '''

        score = compute_score(net, valloader, device="cpu")
        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)
print("Finished Training")
device = "cpu"
test_score_value = test_score(Best_trial_config, net, device)
print(test_score_value)
```

```
Best_trial_config = {'act1 ': 'tanh', 'act2': 'tanh', 'act3':
'tanh''lr': 0.0008, 'batch_size': 16, 'hidden_dim1': 130.0, 'h
idden_dim2': 170.0, 'hidden_dim3': 100.0,'num_layares':4}
```

Best trial final **validation score**: 0.81904388871473355

Best trial **test set score**: 0.808862171931479

Final result with best hyperparameters on test score: 0.788749847528569

Conclusion

In this part I compare results of each search algorithms and schedulers in a table:

scheduler **search algorithm**	MedianStoppingRule HyperOptSearch	MedianStoppingRule OptunaSearch	HyperBandForBOHB OptunaSearch
validation score	0.6823	0.6912	0.7651
test set score	0.6654	0.7054	0.7421
rank	4	3	2

scheduler **search algorithm**	AsyncHyperBandScheduler HyperOptSearch	ASHAscheduler OptunaSearch
validation score	-	0.8190
test set score	-	0.8088
rank	Error	1

Final words

I have learned significant and essential lessons from Dr.Taheri in this term.

Special Thanks to Professor Dr. Taheri for their efforts, guidance and patience.

References

(1) Gasteiger, J.; Zupan, J. Angewandte Chemie International Edition 1993, 32, 503–527.

(2) Zupan, J.; Gasteiger, J. Neural networks in chemistry and drug design; John Wiley & Sons, Inc.

(3) Varnek, A.; Baskin, I. Journal of chemical information and modeling 2012, 52, 1413– 1437.

(4) Mitchell, J. B. Wiley Interdisciplinary Reviews: Computational Molecular Science 2014, 4

(5)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

(6) Devillers, J. Neural networks in QSAR and drug design; Academic Press, 1996. (6) Schneider, G.; Wrede, P. Progress in biophysics and molecular biology 1998, 70, 175– 222. (7) LeCun, Y.; Bengio, Y.; Hinton, G. Nature 2015, 521, 436–444. (8) Schmidhuber, J. Neural networks 2015, 61, 85–117.

(7)(https://github.com/zahta/deep-learning/tree/master/PyTorch_Tutorial)