hyper-parameter optimization project

Authors: Mohammadjavad Abbas pour & Ahmad Yazdani

Supervisor: Dr.Taheri

Optimization in data science

Shahid Beheshti University

July , 2022

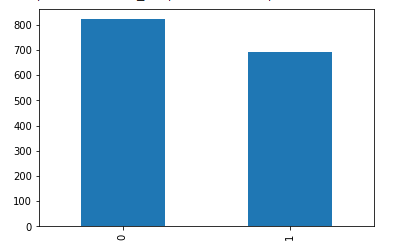
**Abstract**

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]

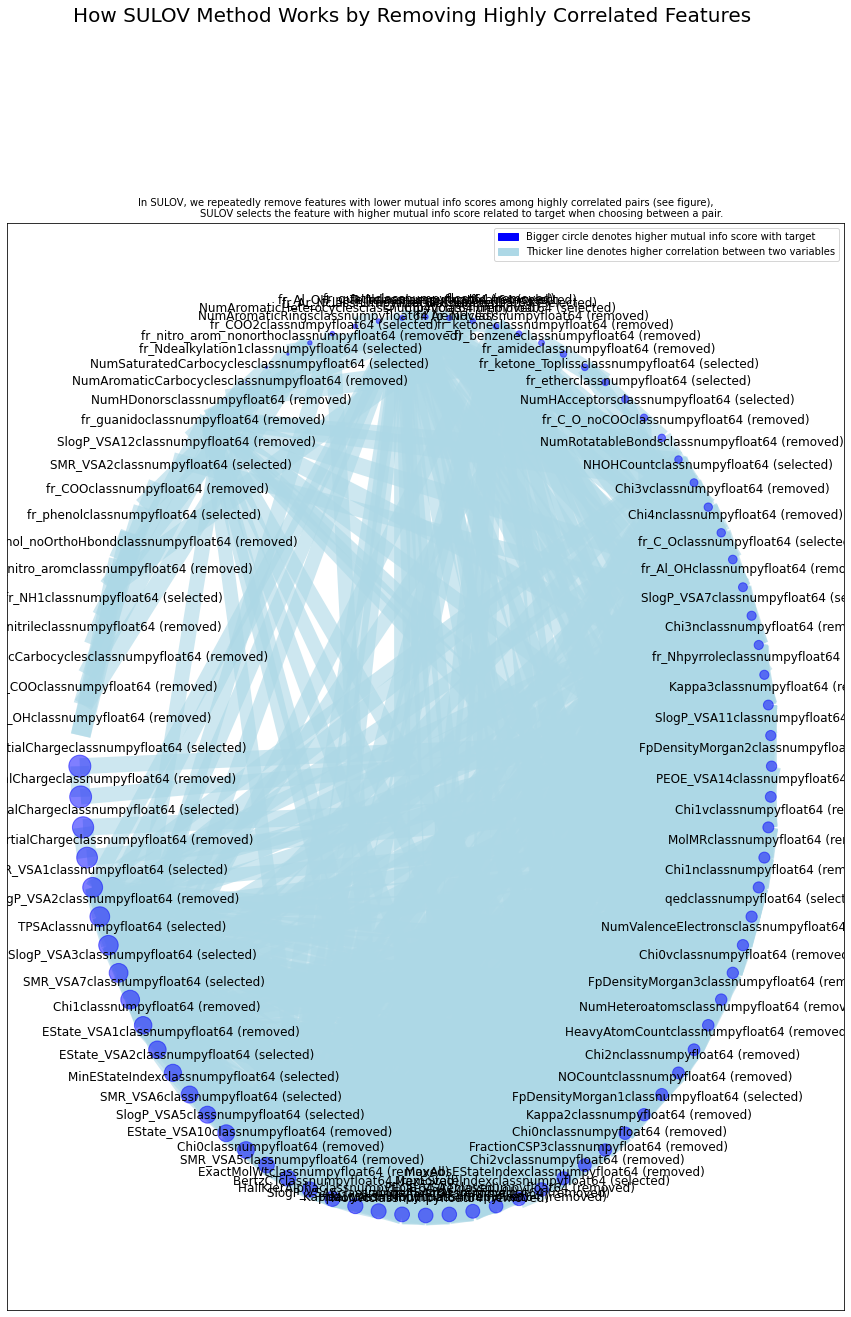
X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size= 0.15,random\_state=1234)

**Feature selection**

I used Corrolation matrix and featurewiz library to select important featuresof 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 ###

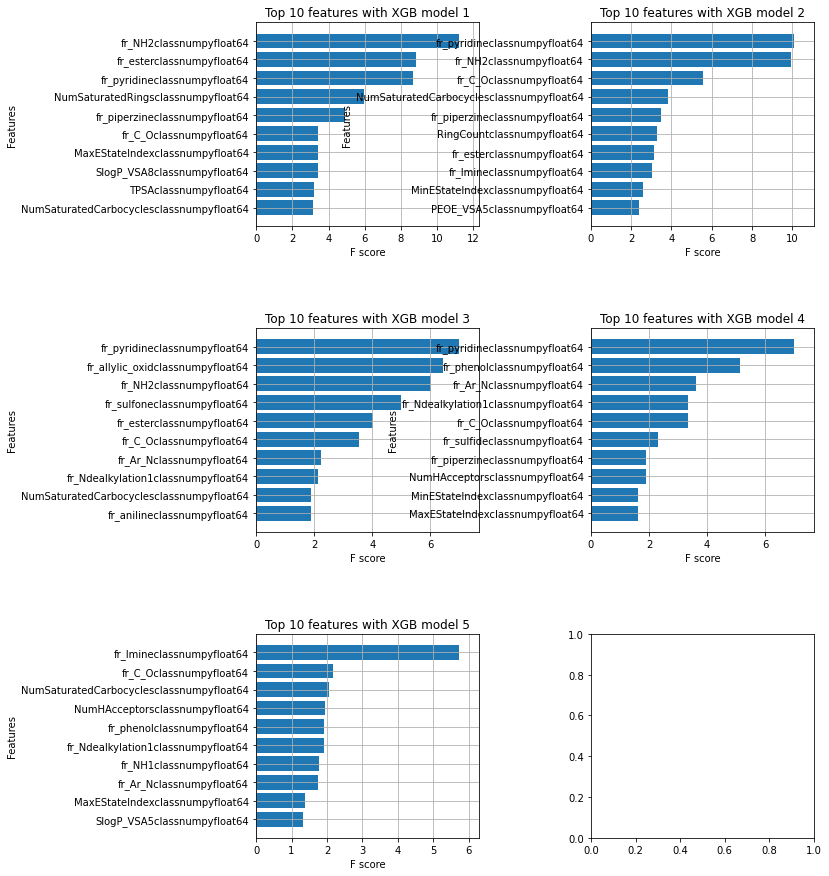
############################################################################################  
############ F A S T F E A T U R E E N G G A N D S E L E C T I O N ! ########  
# Be judicious with featurewiz. Don't use it to create too many un-interpretable features! #  
############################################################################################  
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:



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)

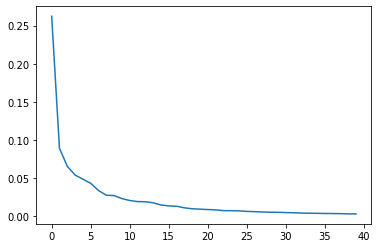
pca**.**fit(X\_train)

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() else '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, 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

### **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","tanh"))

        self.linear("num\_layers") = nn.Linear(self.hidden\_dim("num\_layers")), 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, 'hidden\_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, 'hidden\_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, 'hidden\_dim2': 170.0, 'hidden\_dim3': 100.0,'num\_layares':4}

Best trial final **validation score**: 0.8190438871473355

Best trial **test set score**: 0.808862171931479

**Final result with best hyperparameters on test score:** 0.788749847528569

**Conclusion**

In this part I compare results: