CS5062_ASSESSMENT_1_ZHIXI_TANG_52097136

Student Name: ZHIXI TANG

Student ID: 52097136

INTORDUCTION

This is the report of CS5062 Assessment I including 2 tasks. The .ipynb file including the code of tasks and can be run on a local machine and Google Colab.

The local machine environment is:

- System: Linux Ubuntu 20.04
- Python Version: 3.8

If you are going to implement the code snippets on Google Colab, you might have to upload the data file to your Google Drive firstly, and revise the corresponding directory, then execute the following command:

```python
# mount Google Drive
from google.colab import drive

drive.mount('/content/drive')
```

```python
# unzip the .zip data file
!unzip -q
/content/drive/MyDrive/Machine_Learning_UoA/CS5062_AssessmentII_Dataset.zip
```

The usage of all functions was written as comments in the code snippets. To keep clear of the report, some outputs of code snippets would not be shown, to check the complete result please kindly check the `.ipynb` files.

TASK 1

In this task we will build a Linear model and abstract a Linear regression problem and try to solve it, all the modules we will use in this assessment are open sources and free to use. Therefore, we will use some python libraries working with us. Firstly, we will import all the essential modules we will use in below tasks.

```python
import pandas as pd
import numpy as np
import torch
```

For more details check the documentation of these modules:

- Pandas Docs
- Numpy Docs
- PyTorch Docs

Subtask-A: Data Import

Data Importation

In this step we can use pandas and NumPy module to import and process our data. pandas can read data from the `.csv` file and visualize the data on the Jupyter notebook in a good way. Here we use below code snippet for data importation and visualization.

```
# Reading the .csv data and store it into variable df
df = pd.read_csv('./data/soybean_tabular.csv')

# Visulize df in Jupyter notebook
df
```

	Variety	S_1	S_2	S_3	S_4	M_1	M_2	M_3	W_1	W_2	W_3	W_4	Yield
0	1	4.0900	15.3	396.90	4.98	0.00632	18.0	6.575	2.381979	0.475522	65.2	296.350195	24.0
1	2	4.9671	17.8	396.90	9.14	0.02731	0.0	6.421	7.071148	0.509165	78.9	241.620198	21.6
2	2	4.9671	17.8	392.83	4.03	0.02729	0.0	7.185	6.896941	0.580673	61.1	241.551476	34.7
3	3	6.0622	18.7	394.63	2.94	0.03237	0.0	6.998	2.237817	0.491539	45.8	222.023994	33.4
4	3	6.0622	18.7	396.90	5.33	0.06905	0.0	7.147	1.979327	0.103660	54.2	221.723972	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	1	2.4786	21.0	391.99	9.67	0.06263	0.0	6.593	11.774062	0.565469	69.1	272.952382	22.4
502	1	2.2875	21.0	396.90	9.08	0.04527	0.0	6.120	11.785249	0.562102	76.7	273.264306	20.6
503	1	2.1675	21.0	396.90	5.64	0.06076	0.0	6.976	11.854205	0.777569	91.0	273.421726	23.9
504	1	2.3889	21.0	393.45	6.48	0.10959	0.0	6.794	11.765196	0.679057	89.3	273.913622	22.0
505	1	2.5050	21.0	396.90	7.88	0.04741	0.0	6.030	12.007563	0.615699	80.8	273.623418	11.9

506 rows × 13 columns

(Figure. T1-A-1)

Print Statistical Information

From the above table, we can tell that the column variety is just a number representing corp variety so we don't have to count its statistical information. We will print the statistical summary information of the data via the below code snippet:

```
# delete the column variety
df1 = df.iloc[:, 1:]
# shown describe info
df1.describe()
```

	S_1	S_2	S_3	S_4	M_1	M_2	M_3	W_1	W_2	W_3	W_4	Yield
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.795043	18.455534	356.674032	12.653063	3.613524	11.363636	6.284634	11.131488	0.551258	68.574901	408.252839	22.532806
std	2.105710	2.164946	91.294864	7.141062	8.601545	23.322453	0.702617	6.868654	0.181693	28.148861	168.530578	9.197104
min	1.129600	12.600000	0.320000	1.730000	0.006320	0.000000	3.561000	0.472905	0.103660	2.900000	186.765075	5.000000
25%	2.100175	17.400000	375.377500	6.950000	0.082045	0.000000	5.885500	5.149096	0.422286	45.025000	278.745884	17.025000
50%	3.207450	19.050000	391.440000	11.360000	0.256510	0.000000	6.208500	9.588040	0.528133	77.500000	330.467783	21.200000
75%	5.188425	20.200000	396.225000	16.955000	3.677083	12.500000	6.623500	18.094315	0.668573	94.075000	665.354401	25.000000
max	12.126500	22.000000	396.900000	37.970000	88.976200	100.000000	8.780000	28.074197	1.158538	100.000000	711.210992	50.000000

(Figure. T1-A-2)

From the `Figure. T1-A-1` and `Figure. T1-A-2`, we can tell that there are many values that are missed because many values are represented by 0, the Min of `M_2` is 0, Max of `M_2` is 100 instead, and the 75 percentile is 12.5.

Moreover, we assume to a certain variety of crops, there may be a certain result corresponding to its yield, therefore, we will get the statistical description of different varieties of corps separately.

Variety - Yield Function

We will write below code snippets to implement this function:

```python
def show_describe(dataframe, variety_num):
    """"The variety_num can only be one of [24, 5, 4, 3, 6, 2, 8, 1, 7]"""
    df_v = dataframe.loc[dataframe['Variety'] == variety_num]
    return  df_v.describe()


def DiffYield(dataframe):
    """
    split the table according to the varieties and re-combine them with
    variety - yield corresponding table
    """
    varieties = [24, 5, 4, 3, 6, 2, 8, 1, 7]
    index = ['mean', 'std', 'min', '25%', '50%', '75%', 'max']
    v_names = []
    data = {}

    for i in varieties:
        df_v = show_describe(dataframe, i)  # get the describe info of a single variety
        corp_yield = df_v.iloc[1:, -1].values
        exec(f"data['Variety_{i}'] = corp_yield")
    yield_df = pd.DataFrame(data, index=index)

    return yield_df
```

```python
DiffYield(df)
```

	Variety_24	Variety_5	Variety_4	Variety_3	Variety_6	Variety_2	Variety_8	Variety_1	Variety_7
mean	16.403788	25.706957	21.387273	27.928947	20.976923	26.833333	30.358333	24.365000	27.105882
std	8.539745	9.328401	6.957883	8.324692	2.312801	7.874376	9.727724	8.024454	6.493215
min	5.000000	11.800000	7.000000	14.400000	16.800000	15.700000	16.000000	11.900000	17.600000
25%	11.225000	19.500000	17.575000	21.125000	18.900000	21.400000	23.825000	20.475000	24.300000
50%	14.400000	23.000000	20.450000	26.500000	21.200000	23.850000	28.250000	22.200000	26.200000
75%	19.900000	30.000000	23.650000	34.525000	23.025000	33.225000	33.175000	27.225000	29.600000
max	50.000000	50.000000	50.000000	50.000000	24.800000	43.800000	50.000000	50.000000	42.800000

(Figure-T1-A-3)

From the above table, we can tell that to different varieties of corps, the yield is also different. For example, The Max yield of `Variety_6` is just 24.8 which is much lower than other varieties. According to the mean of `variety_24`, we can tell its average yield is much lower than other varieties. Therefore, we can conclude that the variety of corps affects its yield.

Subtask B: Data Pre-Processing

In this section, we will split the dataset to train, validation, test with the rate of 6:2:2. At the same time, we will try our best to ensure the fairness and uniformity of data. According to the conclusion of `Task 1 - A`, we know that the variety of corps affects its yield. Therefore, we will take samples from each variety at a ratio of $6:2:2$ (train:validation: test), and then compose the datasets. In this case, we can assure as much as possible that in each set of data, the various proportions of corps are approximately equal.

Stratifield_Sampling

Firstly, we will define functions below for stratified sampling:

```python
from sklearn.preprocessing import StandardScaler

def extract_df_by_variety(dataframe):
    """Split dataframe based on varieties, return all dataframes with dictionary"""
    varieties = [24, 5, 4, 3, 6, 2, 8, 1, 7]
    extracted_df = {}
    for variety in varieties:
        df_s = dataframe.loc[dataframe['Variety'] == variety]
        # exec(f"extracted_df['Variety_{variety}'] = df_s")
        extracted_df[variety] = df_s
    return extracted_df

def split_single_dataset(dataframe):
    """split particular dataset to train:val:test = 6:2:2"""
    train_set = dataframe.sample(frac=0.6, random_state=0, axis=0)
    rest_set = dataframe[~dataframe.index.isin(train_set.index)]
    test_set = rest_set.sample(frac=0.5, random_state=0, axis=0)
    val_set = rest_set[~rest_set.index.isin(test_set.index)]

    return train_set, test_set, val_set

def stratified_sampling(dataframe):
    """"Combine above functions, input the dataframe, return the stratified sampled datasets"""
    extracted_dfs = extract_df_by_variety(dataframe)
    train_sets, test_sets, val_sets = [], [], []
    for _df in extracted_dfs.values():
        train_set, test_set, val_set = split_single_dataset(_df)
        train_sets.append(train_set)
        test_sets.append(test_set)
        val_sets.append(val_set)
    p_train = np.array(pd.concat(train_sets).sample(frac=1), dtype='float32')
    p_test = np.array(pd.concat(test_sets).sample(frac=1), dtype='float32')
    p_val = np.array(pd.concat(val_sets).sample(frac=1), dtype='float32')

    return p_train, p_test, p_val
```

```
train_set, test_set, val_set = stratified_samplig(df)
```

Tensor Convertion:

Then we will convert the datasets from `numpy.ndarray` to `torch.Tensor`, which is the data type we can input into the linear model later.

```
def tensor_generator(dataset):
    """input stratified sampled dataset, return variable x and result y"""
    x = torch.tensor(dataset[...,:12])
    y = torch.tensor(dataset[...,12:])
    return x, y
```

```
# get train and test data and corresponding labels
x_train, y_train = tensor_generator(train_set)
x_val, y_val = tensor_generator(val_set)
x_test, y_test = tensor_generator(test_set)
```

Normalization

According to the table `T1-A-1`, we can tell that the values of different features have the observable difference, for example, in the first example, the value of feature `s_3` is 369.90, the value of feature `w_2` is 0.475522. To eliminate the gradient descent that majorly depends on some features, we will do z-score normalization to the variable X. As we know, the formula of z-score is: $\hat{X} = \frac{X-\mu}{\epsilon}$, where $\hat{X}$ is the normalized variable X, μ is the mean of X, ϵ is the standard deviation of X. Therefore, we can define a function like the following.

```
def normalization_x(x, y):
    """Input a tensor, return the z-score normalized tensor"""
    mean = torch.mean(x)
    std = torch.std(x)
    normed_x = (x - mean) / std
    normed_y = (y - mean) / std
    return normed_x, normed_y
```

```
# data normalization
normed_x_train, normed_y_train = normalization_x(x_train, y_train)
normed_x_val, normed_y_val = normalization_x(x_val, y_val)
normed_x_test, normed_y_test = normalization_x(x_test, y_test)
```

Subtask C: Linear Regression Training

In this section, we will define the Linear models and fit them with PyTorch. According to the requirements of the task. We will create two linear models. One is Ridge regression, the other one is Lasso regression. During the training process, we will count the mean square error for the training set and validation set. Then we choose the best performance model according to these observed values.

In fact, Edge regression and Lasso regression is L2 and L1 optimization in Linear regression.

The essence of L1 optimization is adding a $\frac{1}{2}\lambda\omega^2$ to every ω of the function($\frac{1}{2}\lambda||W||^2 = \frac{1}{2}\sum_j \omega_j^2$). Therefore, to define L1 optimization, we will re-write the calculation process of loss function during train process.

To define the Edge regression, L2 regularization means that all ω decrease linearly toward 0 with $\omega + = -\lambda * W$. Fortunately, in PyTorch, the optimizer has the parameter `weight_decay(float, optional)`, when this parameter is not equal to 0, it is L2 regularization.

Therefore, we can define `edge_linear` and `lasso_linear` to train our model as below.

After several times of trials, we will use the following hyperparameters:

- lr(learning rate/step size) = 0.001
- epoch = 10000
- L1/L2 lambda = 0.001

Model Building

We will build our two models with below code snippets:

```
import matplotlib.pyplot as plt
%matplotlib inline

def edge_linear(x_train, y_train, x_val, y_val, epoch, p_step=10, lr=0.01, save_model=False, gpu=False, vis=False, lambda_L=0):
    """Edge regression"""

    model = torch.nn.Linear(12, 1, bias=True)  # define Linear model
    loss_func = torch.nn.MSELoss()  # define loss function
    optim = torch.optim.SGD(model.parameters(), lr=lr, weight_decay=lambda_L)  # define optimizer
    loss_history = []
    loss_val_history = []
    if gpu:
        model = model.cuda(0)
        x_train = x_train.cuda(0)
        y_train = y_train.cuda(0)
        x_val = x_val.cuda(0)
        y_val = y_val.cuda(0)

    print('iter,\ttrain_loss,\tval_loss')
```

```python

    for i in range(epoch):
        """Train process"""
        y_hat = model(x_train)
        loss = loss_func(y_hat, y_train)
        loss_history.append(loss)
        optim.zero_grad()
        loss.backward()
        optim.step()
        # print(model.weight.detach().numpy())
        """validation process"""
        y_val_hat = model(x_val)
        loss_val = loss_func(y_val_hat, y_val)
        loss_val_history.append(loss_val)
        # print(model.weight.detach().numpy())
        """print the train loss and validation loss"""
        if i % p_step == 0 or i==epoch-1:
            print(f'{i}\t{loss.item():.4f}\t\t{loss_val.item():.4f}')
    if save_model:
        torch.save(model, './EdgeLinear.pth')
    if vis:
        x_1 = loss_history
        x_2 = loss_val_history
        y = range(epoch)
        plt.plot(y, x_1, label="train loss")
        plt.plot(y, x_2, label="val loss")
        plt.xlabel("epoch")
        plt.ylabel("loss value")
        plt.title("Loss Values")
        plt.legend()
```

```python
def lasso_linear(x_train, y_train, x_val, y_val, epoch, p_step=10, lr=0.001, save_model=False, gpu=False, vis=False, lambda_L=0):
    """Lasso regression"""
    model = torch.nn.Linear(12, 1, bias=True)  # define Linear model
    loss_func = torch.nn.MSELoss()  # define loss function
    optim = torch.optim.SGD(model.parameters(), lr=lr)  # define optimizer
    loss_history = []
    loss_val_history = []
    if gpu:
        model = model.cuda(0)
        x_train = x_train.cuda(0)
        y_train = y_train.cuda(0)
        x_val = x_val.cuda(0)
        y_val = y_val.cuda(0)
    print('iter,\ttrain_loss,\tval_loss')

    for i in range(epoch):
        """Train process"""
```

```
        w = 0
        for param in model.parameters():
            w += torch.sum(abs(param))
        y_hat = model(x_train)
        loss = loss_func(y_hat, y_train) + lambda_L * w
        loss_history.append(loss.item())
        optim.zero_grad()
        loss.backward()
        optim.step()
        # print(model.weight.detach().numpy())
        """validation process"""
        y_val_hat = model(x_val)
        loss_val = loss_func(y_val_hat, y_val)
        loss_val_history.append(loss_val.item())
        # print(model.weight.detach().numpy())
        """print the train loss and validation loss"""
        if i % p_step == 0 or i==epoch-1:
            print(f'{i}\t{loss.item():.4f}\t\t{loss_val.item():.4f}')
    if save_model:
        torch.save(model, './LassoLinear.pth')
    if vis:
        x_1 = loss_history
        x_2 = loss_val_history
        y = range(epoch)
        plt.plot(y, x_1, label="train loss")
        plt.plot(y, x_2, label="val loss")
        plt.xlabel("epoch")
        plt.ylabel("loss value")
        plt.title("Loss Values")
        plt.legend()
```

Training the model

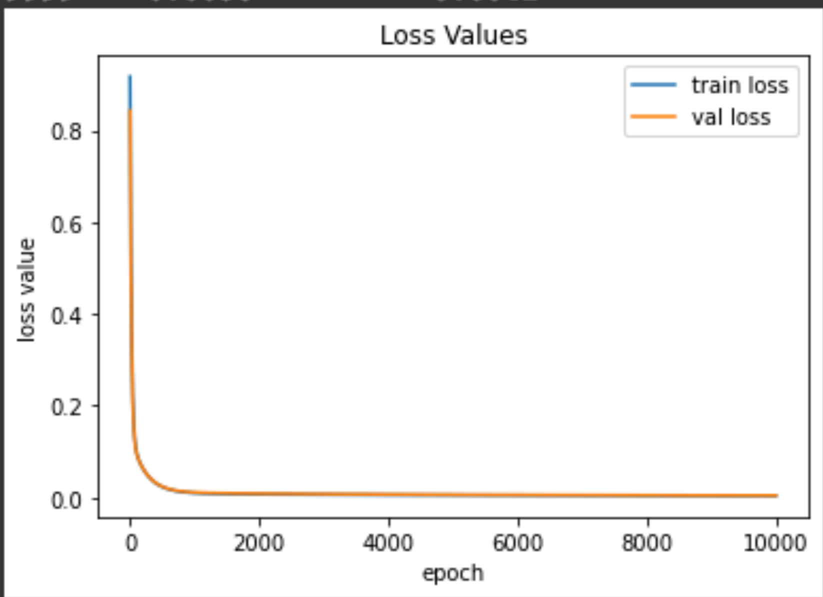
After model the function as above, we will input the model and start the traninig process:

```
# train edge linear model
edge_linear(normed_x_train, normed_y_train, normed_x_val, normed_y_val, epoch=10000, p_step=500, lr=0.001, save_model=True, gpu=True, vis=True, lambda_L=0.001)
```

```
# train lasso linear model
lasso_linear(normed_x_train, normed_y_train, normed_x_val, normed_y_val, epoch=10000, p_step=500, lr=0.001, save_model=True, gpu=True, vis=True, lambda_L=0.001)
```

The training process is shown as the following:

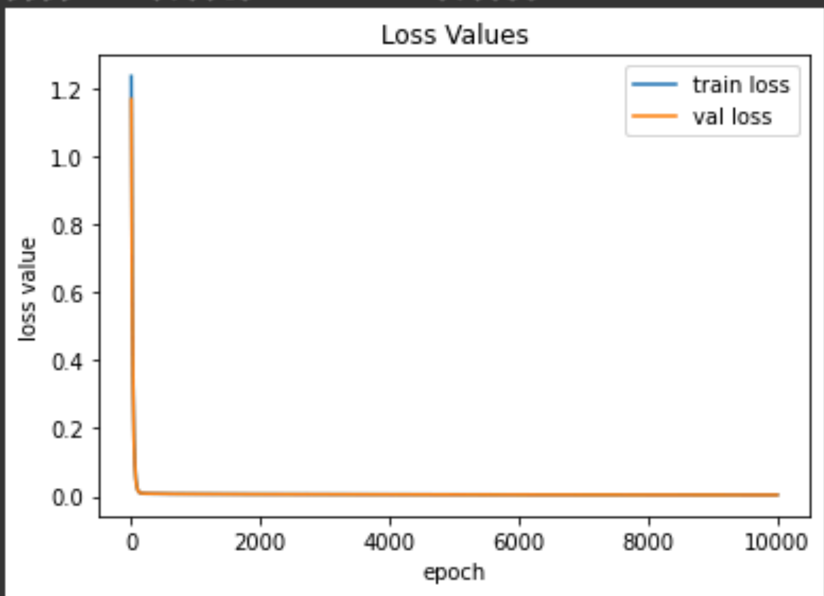
iter,	train_loss,	val_loss
0	0.9173	0.8430
500	0.0229	0.0235
1000	0.0095	0.0108
1500	0.0075	0.0090
2000	0.0069	0.0083
2500	0.0064	0.0078
3000	0.0060	0.0073
3500	0.0057	0.0068
4000	0.0054	0.0065
4500	0.0051	0.0061
5000	0.0049	0.0058
5500	0.0047	0.0056
6000	0.0045	0.0053
6500	0.0043	0.0051
7000	0.0042	0.0050
7500	0.0041	0.0048
8000	0.0040	0.0047
8500	0.0039	0.0045
9000	0.0038	0.0044
9500	0.0037	0.0043
9999	0.0036	0.0042



(Figure. T1-C-1)

The training process is shown as the following:

```
iter,   train_loss,      val_loss
0       1.2349           1.1668
500     0.0070           0.0072
1000    0.0065           0.0066
1500    0.0063           0.0062
2000    0.0060           0.0059
2500    0.0058           0.0056
3000    0.0056           0.0054
3500    0.0054           0.0051
4000    0.0053           0.0050
4500    0.0052           0.0048
5000    0.0050           0.0047
5500    0.0049           0.0045
6000    0.0048           0.0044
6500    0.0047           0.0043
7000    0.0047           0.0042
7500    0.0046           0.0041
8000    0.0045           0.0040
8500    0.0044           0.0040
9000    0.0044           0.0039
9500    0.0043           0.0038
9999    0.0043           0.0038
```



(Figure. T1-C-2)

Subtask D: Inference

We have defined, trained, and saved two models in `Task 1 - C`, as we can tell that the performance of Edge regression is better than Lasso regression. In this section, we will define a test function to calculate the MSE, RMSE, MAE based on the chosen model.

As we know the formulas of the mentioned loss functions are:

- $MAE = \frac{1}{m}\sum_{i=1}^{m}|y_i - \hat{y}_i|$
- $MSE = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$
- $RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2}$

In PyTorch, we can define MSE by `torch.nn.MSEloss()`, define MAE by `torch.nn.L1Loss()`, according to the above formulas, we can write RMSE manually.

```python
def test_data(model_path, x_test, y_test, visualize=False):
    """
    input the path of model, x_test, y_test, the function will print the mse, rmse, mae
    based on the loaded model
    """
    model = torch.load(model_path)
    mse_loss = torch.nn.MSELoss()
    mae_loss = torch.nn.L1Loss()
    # rmse_loss = torch.sqrt(mse_loss)
    predict = model(x_test)
    mse = mse_loss(y_test, predict)     # calculate mse
    mae = mae_loss(y_test, predict)     # calculate mae
    rmse = torch.sqrt(mse_loss(y_test, predict))    # calculate rmse

    print(f'mae:{mae.item():.4f}\tmse: {mse.item():.4f}\trmse:{rmse.item():.4f}')
    return round(mae.item(), 4), round(mse.item(), 4), round(rmse.item(), 4)
```

```python
test_data('./EdgeLinear.pth', normed_x_test.cuda(0), normed_y_test.cuda(0))
```

```
mae:0.0424      mse: 0.0035     rmse:0.0593
(0.0424, 0.0035, 0.0593)
```

(Figure. T1-C-3)

By here we can see in the EdgeLiear model, the error are:

- $MAE = 0.0431$
- $MSE = 0.0036$
- $RMSE = 0.0597$

Comparing with the MSE in the training dataset, we can tell that the MSE_{test} is not much larger than MSE_{train}, inversely, MSE_{test} is smaller than MSE_*train, therefore, we can say the generalization of the trained model is relatively eligible.

Subtask E: Feature Importance

In this section, we will infer the most important features based on the given data and trained model. Our idea is that we will eliminate one of the features in the data set. Then we will run this feature eliminated feature on our trained model to check the error. Then we will count the difference between this error with the error which wasn't eliminated. The bigger difference is, the feature is more important.

```
def feature_eliminate(test_data, feature_num):
    """
    replace the value of one of the features in the test dataset
    by given the feature number, then return a new test dataset without this feature.
    Feature number representation:
        0 -- Variety
        1 -- S1
        2 -- S2
        3 -- S3
        4 -- S4
        5 -- M1
        6 -- M2
        7 -- M3
        8 -- W1
        9 -- W2
        10 -- W3
        11 -- W4
    """
    new_test = deepcopy(test_data)
    new_test[..., feature_num] = 0
    return new_test

def importance_prs(original_err, other_errors):
    """count the difference based on mae, mse, rmse"""
    importance_mae = {}
    importance_mse = {}
    importance_rmse = {}
    for k, v in other_errors.items():
        mae_dff = round(abs(original_error[0] - v[0]), 4)
        importance_mae[k] = mae_dff

        mse_dff = round(abs(original_error[1] - v[1]), 4)
        importance_mse[k] = mse_dff

        rmse_dff = round(abs(original_error[2] - v[2]), 4)
```

```
        importance_rmse[k] = rmse_dff
    return importance_mae,importance_mse,importance_rmse

def importance_evl(importances, matrix = 'mse'):
    """print the importance from unimportant to important sequence based on the pointed matrix"""
    if matrix == 'mae':
        print(sorted(importances[0].items(), key=lambda item:item[1]))
    elif matrix == 'mse':
        print(sorted(importances[1].items(), key=lambda item:item[1]))
    else:
        print(sorted(importances[2].items(), key=lambda item:item[1]))
```

```
# eliminate the each features

variety_x_test = feature_eliminate(normed_x_test, 0)

s1_x_test = feature_eliminate(normed_x_test, 1)
s2_x_test = feature_eliminate(normed_x_test, 2)
s3_x_test = feature_eliminate(normed_x_test, 3)
s4_x_test = feature_eliminate(normed_x_test, 4)

m1_x_test = feature_eliminate(normed_x_test, 5)
m2_x_test = feature_eliminate(normed_x_test, 6)
m3_x_test = feature_eliminate(normed_x_test, 7)

w1_x_test = feature_eliminate(normed_x_test, 8)
w2_x_test = feature_eliminate(normed_x_test, 9)
w3_x_test = feature_eliminate(normed_x_test, 10)
w4_x_test = feature_eliminate(normed_x_test, 11)
```

```
# using the eliminated data, run on the trained model to count mae, mse, rmse.

original_error = test_data('./EdgeLinear.pth', normed_x_test.cuda(0), normed_y_test.cuda(0))

errors = {}
errors['Variety'] = test_data('EdgeLinear.pth', variety_x_test.cuda(0), normed_y_test.cuda(0))
errors['S_1'] = test_data('EdgeLinear.pth', s1_x_test.cuda(0), normed_y_test.cuda(0))
errors['S_2'] = test_data('EdgeLinear.pth', s2_x_test.cuda(0), normed_y_test.cuda(0))
errors['S_3'] = test_data('EdgeLinear.pth', s3_x_test.cuda(0), normed_y_test.cuda(0))
errors['S_4'] = test_data('EdgeLinear.pth', s4_x_test.cuda(0), normed_y_test.cuda(0))

```

```
errors['M_1'] = test_data('EdgeLinear.pth', m1_x_test.cuda(0), normed_y_test.cuda(0))
errors['M_2'] = test_data('EdgeLinear.pth', m2_x_test.cuda(0), normed_y_test.cuda(0))
errors['M_3'] = test_data('EdgeLinear.pth', m3_x_test.cuda(0), normed_y_test.cuda(0))

errors['W_1'] = test_data('EdgeLinear.pth', w1_x_test.cuda(0), normed_y_test.cuda(0))
errors['W_2'] = test_data('EdgeLinear.pth', w2_x_test.cuda(0), normed_y_test.cuda(0))
errors['W_3'] = test_data('EdgeLinear.pth', w3_x_test.cuda(0), normed_y_test.cuda(0))
errors['W_4'] = test_data('EdgeLinear.pth', w4_x_test.cuda(0), normed_y_test.cuda(0))
```

```
# store the importances
importances = importance_prs(original_error, errors)
```

```
# get the importance sequences by different metric
importance_evl(importances, 'mae')
importance_evl(importances, 'mse')
importance_evl(importances, 'rmse')
```

```
[('W_1', 0.0001), ('W_3', 0.0005), ('W_2', 0.0011), ('S_3', 0.0024), ('M_2', 0.0032), ('Variety', 0.0085), ('S_4', 0.009), ('S_2', 0.0361), ('S_1', 0.0404), ('W_4', 0.0563), ('M_3', 0.0854), ('M_1', 0.1157)]
[('S_3', 0.0), ('M_2', 0.0), ('W_1', 0.0), ('W_3', 0.0002), ('W_2', 0.0003), ('Variety', 0.0016), ('S_4', 0.0018), ('S_2', 0.0043), ('S_1', 0.0051), ('W_4', 0.0097), ('M_3', 0.0154), ('M_1', 0.024)]
[('W_1', 0.0), ('S_3', 0.0002), ('M_2', 0.0004), ('W_2', 0.0021), ('W_3', 0.0021), ('Variety', 0.0124), ('S_4', 0.0137), ('S_2', 0.0292), ('S_1', 0.0335), ('W_4', 0.0555), ('M_3', 0.0783), ('M_1', 0.1066)]
```

(Figure. T1-E-1)

From above result, we can tell that to different matrix, the feature importance is different. If we choose MAE as our matrix, then the importance of data features is:

$$W_1 < W_3 < W_2 < S_3 < M_2 < Variety < S_4 < S_2 < S_1 < W_4 < M_3 < M_1$$

If we choose MSE as our matrix, then the feature importance is:

$$S_3 < M_2 < W_1 < W_3 < W_2 < Variety < S_4 < S_2 < S_1 < W_4 < M_3 < M_1$$

If we choose RMSE as our matrix, then the feature immportance is:

$$W_1 < S_3 < M_2 < W_2 < W_3 < Variety < S_4 < S_2 < S_1 < W_4 < M_3 < M_1$$

Especially, we can tell no matter what matrix we choose, the most 7 important features are exactly same, as well as the sequence. The rest features have barely affection to the result so they are not important features to the data. Therefore, we can conclude that these 7 features are important for the data. The importance from high to low is $M_1 > M_3 > W_4 > S_1 > S_2 > S_4 > Variety$.

TASK 2

In this task we will build a MLP and CNN neural network to predict the average yield based on the given data of pictures.

Subtask A: Data Import/Pre-processing

In this part, we will first import the data and do a brief analysis, then we will decide if we are going to do normalization of the data, finally, we will split the data to train, validation, test sets with rate of 6:2:2.

Data Importation and Extraction

Firstly, we will import the modules we will use in this section.

```
# import essential modules
import os
import random
from tqdm import tqdm
import numpy as np
```

Then we will unzip the data from `.zip` file.

```
# first unzip the .zip file. '-q' keep verbose, not print the process
!unzip -q /content/drive/MyDrive/Machine_Learning_UoA/soybean_images.zip
```

After unzipped, we can see there is a new folder in our current work directory named `syobean_images`. The images are stored in this directory. Each `npz` file is a picture containing the data of image and its label (average yield). Therefore, we will write a function to extract the data from `.npz` file and store them into images and label variable.

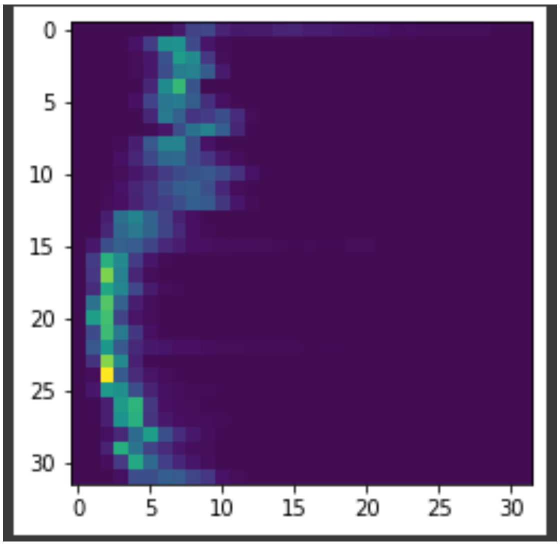
```
def extract_data(directory):
    """
    Input the path of directory where containing `.npz` files,
    then extract the data of images and labels, return two `numpy.ndarray`  type variables
    which containing of. the image data corresponding to their labels
    """
    images = [np.load(directory+str(f))['image'] for f in tqdm(os.listdir(directory))]
    labels = [np.load(directory+str(f))['y'] for f in tqdm(os.listdir(directory))]
    return np.asarray(images, dtype='float32'), np.asarray(labels, dtype='float32')
```

```python
# extract the images and labels
sample_dir = './soybean_images/'
images, labels = extract_data(sample_dir)

# reshape labels as matrix
labels = labels.reshape(-1, 1)

# check the shape of variable image and labels
print(images.shape)
print(labels.shape)
```

After execution above snippets, we can tell that we have 22986 arrays in both images and labels, each array contains 9 infrared images scanned by different infrared intensity. We can plot one of the images to see how's it look like.

Note: the image are generated by scanning of infrared ray, the pixels of the image is very low (from 0 to 1), which means we can't observe the image if we just simply convert NumPy to RGB image. However, we can use Pycharm sciview or matplotlib to plot the image.

```python
import matplotlib.pyplot as plt

# choose the first infrared image of first array to plot. An array contains 9 plots.
plt.imshow(images[0][0])
```



(Figure. T2-A-1)

Normalization

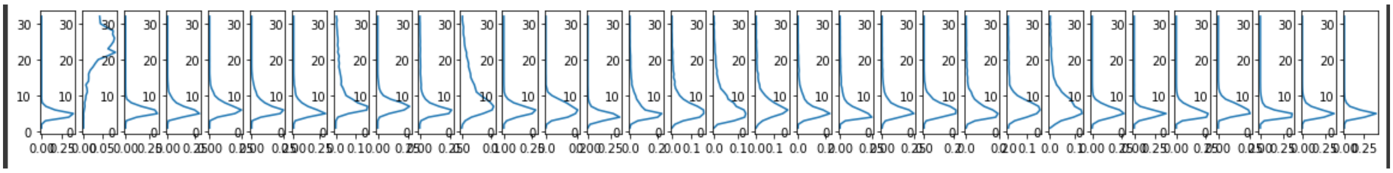
The purpose of normalization is to scale the data features with a standard to accelerate convergence, accuracy, and prevent the gradient explosion of the model. The preliminary idea is to scale all the values in the range of `[0-1]`. We will check the value range of images to decide if we have to do data normalization.

```python
# show max value of images
np.max(images)
```

```
# show min value of images
np.min(images)
```

```
def plot_distribution(arrs):
    """
    Randomly pick one of the pictures, then pick one of 9 layers, then plot the data distribution
    of this layer.
    """
    n = random.randint(1, 20000)
    for imgs in tqdm(arrs[n]):  # images[0]
        k = 1
        plt.figure(figsize=(20, 20))
        for img in imgs:    # images[0][0]
            plt.subplot(9, 32, k)
            plt.plot(img, range(1, 33))
            k+=1
```

```
plot_distribution(images)
```



(Figure. T2-A-2)

From the above results, we can tell that the maximum value of images is 1, the minimum value of data is 0. Moreover, the data distribution is relatively equal to the Gaussian distribution. Therefore, we don't have to do the data normalization.

Split the dataset

Normally, if the number of samples in the dataset isn't beyond a million, we split the data to train, validate, and test with the rate of 6:2:2, if we have a million level dataset, then we normally split the datasets with the rate of 98:1:1. In this step, we will split the data as a train set, validation set, and test set where the rate is 6:2:2, the training set containing contains 13792 samples, validation and testing set are both containing 4597 samples. In this case, we have enough samples for training and validation, at the same time, we have enough samples to verify the generalization of the model on the testing set.

In this step, we will define an `ADataset` which is succeeded to `torch.utils.data.Dataset`, it is an abstract class representing a dataset. Then we will write a method called `random_split` in this class, this method takes a rate and split the dataset based on the rate. (torch.utils.data.Dataset Docs).

```
class AsDataset(torch.utils.data.Dataset):
    def __init__(self, x, y):
        self.x = torch.from_numpy(x)
        self.y = torch.from_numpy(y)
```

```python

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

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]

    def random_split(self, rate):
        """
        take the rate(type:list) as input, then return same type split subsets based on the rate, the number of subsets
        depends on the length of rate. The sum of all numbers in rate must be 10.
        i.g list = [2, 2, 2, 2, 2], the function will return 5 subsets, length of subset_1: subset_2 : ... : subset_5 = 2: 2: 2: 2: 2.
            list = [8, 2], the function will return 2 subsets, length of subset_1 : subset_2 = 8: 2
        """
        lengths = [int(round(len(self) * i / 10)) for i in rate]
        sets =  torch.utils.data.random_split(self, lengths)
        # return [s.dataset  for s in sets]
        return sets
```

```python
img_dataset = AsDataset(images, labels)
train_set, val_set, test_set = img_dataset.random_split([6, 2, 2])

# print the length of split dataset
print(f'Length of tranining set: {len(train_set)}')
print(f'Length of validation set: {len(val_set)}')
print(f'Length of testing set: {len(test_set)}')
```

After execution of the above code snippets, we can see the image dataset was split to `train_set`, `val_set`, and `test_set` by the rate of $6:2:2$.

Subtask B: Training and Justification

In this task, we will predict the average yield based on the split datasets. The prediction is a value representing the average yield. Therefore, we can abstract it as a regression problem.

In MLP, each neuron on the first layer takes a feature of the input data that share the same dimensionality, the first layer will calculate the weight and bias according to the features, then the subsequent layer will calculate the weight and bias based on the output of its upper layer, then the next layer will do the same thing...., finally, we will get more accurate weight and bais which can represent prediction based on the input data. In PyTorch, this layer is defined as `nn.Linear()`.

In CNNs, compare with MLPs, MLPs learn the global features of input (eg. to an image, MLP learns all the modes of the image about the pixel. However, CNNs learn only learn partial features of the image according to its "kernel", the kernel is like a 2D window with a fixed height and width, the window will side on the image, it is like a scanner scanning the image to get the features, the feature CNNs get is learnt from the window. Because of the translation invariant and spatial hierarchies of patterns in the visual world, if the CNN learnt the features somewhere of the image when the feature appears at other places of the image, CNN can also recognize it, but for MLP, if the feature appears on a new position of the image, then the network has to learn this mode over again. Moreover, the CNN layer can learn a bigger model from the output of its prior layer so CNN can learn more and more complicated and abstracted modes. In PyTorch, the CNN layer for the 2D image is defined as `nn.Conv2D()`.

Overall, we will build up to two models as `class` in Python to predict the average yield. One of the models is Multilayer Perception(MLP), the other one is Convolutional Network(ConvNet). Both of them are feasible for this task. Before we define our models, we will install an open-source module called `torchinfo`, this module can help us check the summary information of our models (In Tensorflow and Keras, they have the same function called `model.summary()`)

```
# intall a third-part module so that we can check the summary information of model
!pip install torchinfo
```

Because each sample in our dataset is a 9-dimensional image, so when we send the sample to our model, we have to flatten it to a 1-dimensional array as we know each layer in our MLP model is a linear model. Our MLP model is succeeded to `torch.nn.Module`, therefore, it has all features of general modules in PyTorch. We will define our MLP model as the following:

```
from torch import nn

# build our MLP model
class MLP(nn.Module):
    '''MLP model to solve our task based on the data shape'''
    def __init__(self, model_name):
        super().__init__()
        self.model_name = model_name
        self.layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(32 * 32 * 9, 576),
```

```
            nn.ReLU(),
            # nn.Linear(64, 32),
            # nn.ReLU(),
            nn.Linear(576, 288),
            nn.ReLU(),
            nn.Linear(288, 144),
            nn.ReLU(),
            nn.Linear(144, 72),
            nn.ReLU(),
            nn.Linear(72, 1),
            # nn.ReLU(),
            # nn.Linear(36, 1)
        )

    def forward(self, x):
        """forward pass"""
        return self.layers(x)
```

In CNNs, because every time the kernel slides on a 2-D surface of the image, it will get the features of the image with size [kernal_height, kernal_width, channel], therefore, we don't have to flatten the image. However, as it is a regression task and we still have to calculate the weight and bias of features, we have to flatten the CNN layer to a 1-D layer, then sending to the Linear layer.

```
# build out ConvNet model
class ConvNet(nn.Module):
    '''
    ConvNet model to solve our task based on the processed dataset.
    We define the Conv2D layers and Dense layers separately for the convenience of code refactor
    '''
    def __init__(self, model_name):
        super().__init__()
        self.model_name = model_name
        self.conv_layers = nn.Sequential(
            nn.Conv2d(9, 18, kernel_size=3),
            nn.ReLU(),
            nn.Conv2d(18, 9, kernel_size=3),
            nn.ReLU(),

            nn.Flatten(),
        )
        self.fc = nn.Sequential(
            nn.Linear(28 * 28 * 9, 576),
            nn.ReLU(),
            nn.Linear(576, 288),
            nn.ReLU(),
            nn.Linear(288, 144),
            nn.ReLU(),
            nn.Linear(144, 72),
```

```python
                nn.ReLU(),
                nn.Linear(72, 1),
            )

    def forward(self, x):
        '''forward pass'''
        out = self.conv_layers(x)
        return self.fc(out)
```

Let's initialize our model defined above and check the summary information. We assume that we will send 100 samples (the shape is [9, 32, 32]) to the model in each batch.

```python
from torchinfo import summary
import torch

mlp = MLP('mlp_model')
summary(mlp, input_size=(100, 9, 32, 32))

convnet = ConvNet('cnn_model')
summary(convnet, input_size=(100, 9, 32, 32))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
MLP                                      --                        --
├─Sequential: 1-1                        [100, 1]                  --
│    └─Flatten: 2-1                      [100, 9216]               --
│    └─Linear: 2-2                       [100, 576]                5,308,992
│    └─ReLU: 2-3                         [100, 576]                --
│    └─Linear: 2-4                       [100, 288]                166,176
│    └─ReLU: 2-5                         [100, 288]                --
│    └─Linear: 2-6                       [100, 144]                41,616
│    └─ReLU: 2-7                         [100, 144]                --
│    └─Linear: 2-8                       [100, 72]                 10,440
│    └─ReLU: 2-9                         [100, 72]                 --
│    └─Linear: 2-10                      [100, 1]                  73
==========================================================================================
Total params: 5,527,297
Trainable params: 5,527,297
Non-trainable params: 0
Total mult-adds (M): 552.73
==========================================================================================
Input size (MB): 3.69
Forward/backward pass size (MB): 0.86
Params size (MB): 22.11
Estimated Total Size (MB): 26.66
==========================================================================================
```

(Figure. T2-B-1)

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
ConvNet                                  --                        --
├─Sequential: 1-1                        [100, 7056]               --
│    └─Conv2d: 2-1                       [100, 18, 30, 30]         1,476
│    └─ReLU: 2-2                         [100, 18, 30, 30]         --
│    └─Conv2d: 2-3                       [100, 9, 28, 28]          1,467
│    └─ReLU: 2-4                         [100, 9, 28, 28]          --
│    └─Flatten: 2-5                      [100, 7056]               --
├─Sequential: 1-2                        [100, 1]                  --
│    └─Linear: 2-6                       [100, 576]                4,064,832
│    └─ReLU: 2-7                         [100, 576]                --
│    └─Linear: 2-8                       [100, 288]                166,176
│    └─ReLU: 2-9                         [100, 288]                --
│    └─Linear: 2-10                      [100, 144]                41,616
│    └─ReLU: 2-11                        [100, 144]                --
│    └─Linear: 2-12                      [100, 72]                 10,440
│    └─ReLU: 2-13                        [100, 72]                 --
│    └─Linear: 2-14                      [100, 1]                  73
==========================================================================================
Total params: 4,286,080
Trainable params: 4,286,080
Non-trainable params: 0
Total mult-adds (M): 676.17
==========================================================================================
Input size (MB): 3.69
Forward/backward pass size (MB): 19.47
Params size (MB): 17.14
Estimated Total Size (MB): 40.30
==========================================================================================
```

(Figure. T2-B-2)

In the next five code snippets, we will define below functions:

```python
def fit(model, trainloader, loss_func, optim, device):
    """fit the model for one epoch step"""
    device = device
    fit_trainloader = trainloader
    model.train()

    train_loss = 0.0
    counter = 0

    # Iterate over the DataLoader for training data
    for i, data in enumerate(fit_trainloader, 0):
        counter += 1
        inputs, targets = data[0].to(device), data[1].to(device)  # get inputs and targets

        optim.zero_grad()   # zero the optimizer
        outputs = model(inputs)   # perform forward pass
        loss = loss_func(outputs, targets)  # compute loss

        train_loss += loss.item()

```

```
        loss.backward()     # perform backward pass
        optim.step()    # perform optimization

    train_avg_loss = train_loss / counter

    # return train_avg_loss, train_avg_acc
    return train_avg_loss
```

```
def validate(model, val_loader, loss_func, device):
    """evaluate the model for one time"""
    device = device
    val_loader = val_loader
    counter = 0
    val_loss = 0
    model.eval()

    # Iterate over the DataLoader for validation data
    with torch.no_grad():
        for _, data in enumerate(val_loader, 0):
            counter += 1
            inputs, targets = data[0].to(device), data[1].to(device)

            outputs = model(inputs)

            loss = loss_func(outputs, targets)
            val_loss += loss.item()

        val_avg_loss = val_loss / counter

        return val_avg_loss
```

```
def save_model(model):
    """save the model into current work directory"""
    save_path = os.path.join(os.getcwd(), model.model_name+'.pth')
    torch.save(model, save_path)
```

```
class LRScheduler:
    """
    Learning rate scheduler.
    If the validation loss does not decrease for the given number of 'patience' epochs,
    then the learning rate will decrease by given 'factor'.
    """

    def __init__(self, optimizer, patience=5, min_lr=1e-6, factor=0.5):
        """
        new_lr = old_lr * factor
```

```
        :param optimizer: the optimizer we are using
        :param patience: how many epochs to wait before updating the lr
        :param min_lr: least lr value to reduce to while updating
        :param factor: factor by which the lr should be updated
        """
        self.optimizer = optimizer
        self.patience = patience
        self.min_lr = min_lr
        self.factor = factor

        self.lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
            self.optimizer,
            mode='min',
            patience=self.patience,
            factor=self.factor,
            min_lr=self.min_lr,
            verbose=True
        )

    def __call__(self, val_loss):
        self.lr_scheduler.step(val_loss)
```

```
class EarlyStopping:
    """
    Early stopping to stop the training when the loss does not improve after
certain epochs.
    """

    def __init__(self, patience=5, min_delta=0):
        """
        :param patience: how many epochs to wait before stopping when loss is not
improving
        :param min_delta: minimum difference between new loss and old loss for new
loss
                          to be considered as an improvement
        """
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss):
        if self.best_loss is None:
            self.best_loss = val_loss
        elif self.best_loss - val_loss > self.min_delta:
            self.best_loss = val_loss
            self.counter = 0  # reset the counter if validation loss improves
        elif self.best_loss - val_loss < self.min_delta:
```

```
            self.counter += 1
            print(f"INFO: Early stopping counter {self.counter} of {self.patience}")
            if self.counter >= self.patience:
                print('INFO: Early Stopping')
                self.early_stop = True
```

```
import time

def train(model, epochs, train_set, val_set, batch_size, loss_func, optim, device, save=False, opt=None):
    """train the model with required epochs and other required parameters"""
    train_loss, val_loss = [], []
    model = model

    train_loader = torch.utils.data.DataLoader(val_set, batch_size=batch_size, shuffle=True, num_workers=1, pin_memory=True)
    val_loader = torch.utils.data.DataLoader(val_set, batch_size=batch_size, shuffle=True, num_workers=1, pin_memory=True)

    start = time.time()
    print(summary(model, input_size=(batch_size, 9, 32, 32)))

    for epoch in range(epochs):
        print(f"Epoch: {epoch+1} of {epochs}")
        train_epoch_loss = fit(
            model=model,
            trainloader = train_loader,
            loss_func = mae_loss,
            optim = optim,
            device = device
        )

        val_epoch_loss = validate(
            model=model,
            val_loader=val_loader,
            loss_func = mae_loss,
            device=device
        )
        train_loss.append(train_epoch_loss)
        val_loss.append(val_epoch_loss)

        # lr_scheduler
        if opt == "lr_scheduler":
            lr_scheduler = LRScheduler(optim)
            lr_scheduler(val_epoch_loss)

        # early stopping
        elif opt == "early_stopping":
```

```
            early_stopping(val_epoch_loss)
            if early_stopping.early_stop:
                break

        print(f"\tTrain loss: {train_epoch_loss:.8f}\tVal loss: {val_epoch_loss:.8f}")
    if save:
        save_model(model)

    end = time.time()
    print(f"Training time: {(end-start)/60:.3f} minutes")
    return train_loss, val_loss
```

Then we will define the loss function, optimizer, the device where we are going to train our model (We always train the model on GPU if the machine has as it's much faster than CPU).

```
# declare which will hold the device we're training on (CPU or GPU)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"computation device: {device}\n")

# Set fixed random number seed
torch.manual_seed(42)

# Put our models into the device.
convnet.to(device)
mlp.to(device)

# Define the loss function and optimizer
mae_loss = nn.L1Loss()
mse_loss = nn.MSELoss()

def rmse_loss(y, y_hat):
    mse = nn.MSELoss()
    return torch.sqrt(mse(y, y_hat))
rmse_loss = rmse_loss

# Define the optimizer
lr = 0.01
weight_decay = 0.001
conv_optimizer = torch.optim.Adam(convnet.parameters(), lr=lr, weight_decay=weight_decay)
mlp_optimizer = torch.optim.Adam(mlp.parameters(), lr=lr, weight_decay=weight_decay)
```

In the next, we will start to fit the model of MLP and CNN. In the draft version of this experiment, we found out it would be so long to overfit the model, so we adopt the training trick of `lr_schedular` instead of `early_stopping`.

```
# train mlp
mlp_history = train(
    model=mlp,
    epochs=200,
    train_set = train_set,
    val_set = val_set,
    batch_size = 1500,
    loss_func = mae_loss,
    optim = mlp_optimizer,
    device = device,
    save = True,
    opt = 'lr_scheduler'
)
```

```
# train cnn
conv_history = train(
    model=convnet,
    epochs=200,
    train_set = train_set,
    val_set = val_set,
    batch_size = 1500,
    loss_func = mae_loss,
    optim = conv_optimizer,
    device = device,
    save = True,
    opt='lr_scheduler'
)
```

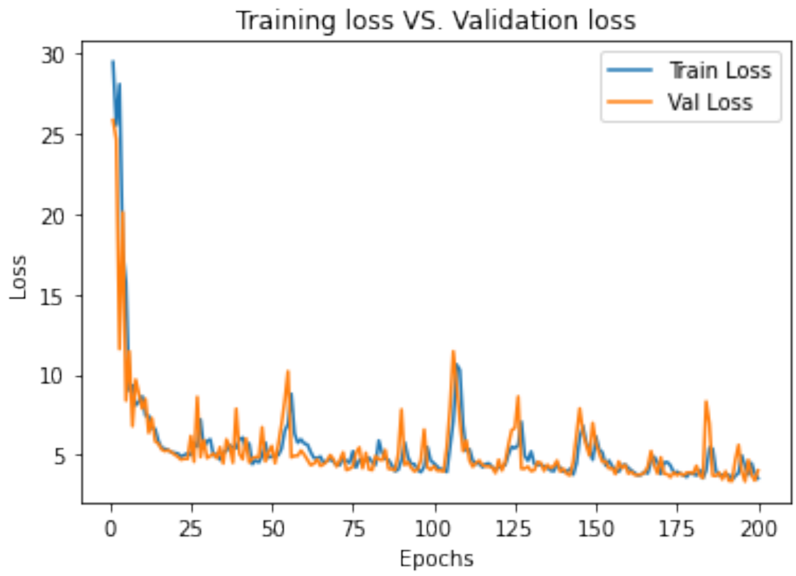
After the training process, we will write a function to plot the loss values during the training process.

```
def loss_plot(history, epochs):
    """plot the val_loss and train_loss"""
    train_loss = history[0]
    val_loss = history[1]
    plt.plot(range(1, epochs+1), train_loss, label="Train Loss")
    plt.plot(range(1, epochs+1), val_loss, label="Val Loss")
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training loss VS. Validation loss')
    plt.legend()
```

```
loss_plot(mlp_history, 200)
```

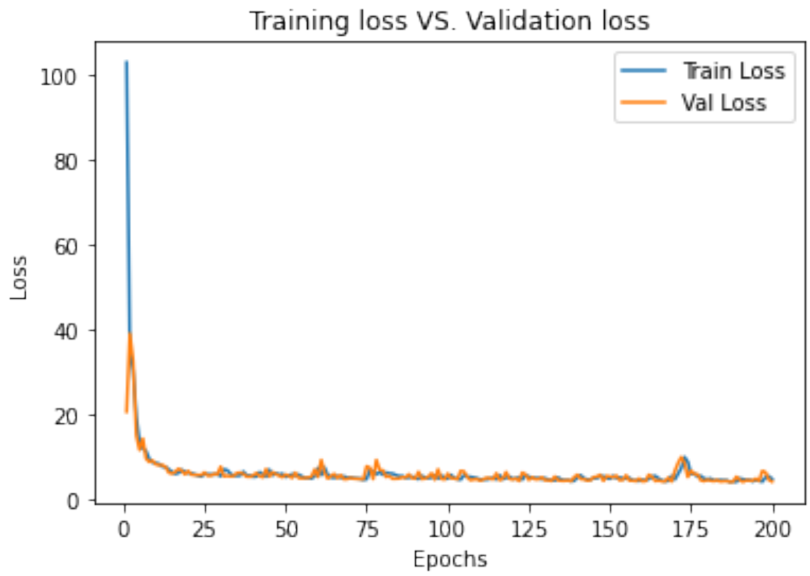
```
loss_plot(conv_history, 200)
```

The plot of MLP is shown as following:



(Figure. T2-B_3)

The plot of CNN is hown as following:



(Figure. T2-B-4)

From the above plots, we can see that the amplitude of MLP is larger than that of CNN. In CNN, the curve of validation fits the loss curve more closely compared with MLP. Moreover, in the same number of epochs, CNN converges faster than MLP.

We will also define a function to evaluate the loss of the model:

```python
def model_test(model_path, test_set, loss_func, batch_size, device):
    """Evaluate the model by given model path, test dataset, and loss function"""
    model = torch.load(model_path)
    model.eval()

    test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, shuffle=True, num_workers=1, pin_memory=True)
    avg_test_loss = validate(model, test_loader, loss_func, device)

    print(f'The test loss is: {avg_test_loss:.4f}')
```

```python
# test mlp
model_test('mlp_model.pth', test_set, mae_loss, 1000, device)
```

```python
# test cnn
model_test('cnn_model.pth', test_set, mae_loss, 1000, device)
```

We will get the result as the following:

```
The test loss is: 5.0598
```

```
The test loss is: 4.9000
```

Subtask C: Cross Validation

In this subtask, we will train the model with K-fold cross-validation. K-fold cross-validation is splitting the whole dataset to K subsets, then we will train the model for K times, each time, we will use the ith ($i \in \{1, 2, \dots K\}$) dataset as the test set, and the rest $K - 1$ subsets as the training set. The final result is the mean of each training result in K training. We use this way to ensure that each sample is given the opportunity to be used in the test set 1 time and used to train the model $K - 1$ time. Therefore, when the data distribution is not even, we can still evaluate whether the model is under-fitting, over-fitting, or well-generalized.

Function Re-Define

To implement K-fold cross validation, we will define functions as the following:

```python
from torch.utils.data import DataLoader, ConcatDataset
from sklearn.model_selection import KFold
```

```python
def reset_weights(m):
    """
    Try resetting model weights to aovid weight leakage.
    """
    for layer in m.children():
        if hasattr(layer, 'reset_parameters'):
            print(f'Reset trainable parameters of layer = {layer}')
            layer.reset_parameters()
```

```python
def k_fold_train(model, dataset, epochs, optimizer, batch_size, loss_func, device, k_folds=5):
    kfold = KFold(n_splits=k_folds, shuffle=True)
    results = {}
    test_loaders = [] # save the hold out test loader for evaluation later
    for fold, (train_ids, test_ids) in enumerate(kfold.split(dataset)):
        print(f'FOLD {fold}')
        print('----------------------------------------')

        # Sample train and test datasets based on the index of dataset.
        tr_sampler = torch.utils.data.SubsetRandomSampler(train_ids)
        ts_sampler = torch.utils.data.SubsetRandomSampler(test_ids)

        train_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, sampler=tr_sampler)
        test_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, sampler=ts_sampler)
        test_loaders.append(test_loader)

        # Initialize model
        model = model
```

```
        model.apply(reset_weights)
        # Initialize optimizer
        optimizer = optimizer
        # Initialize loss function
        loss_func = loss_func

        # Training process
        for epoch in range(epochs):
            avg_epoch_loss = fit(model, train_loader, loss_func, optimizer, device)
            # implement lr_schedular
#             lr_scheduler = LRScheduler(optimizer)
#             lr_scheduler(avg_epoch_loss)
            if epoch % 20 == 19:
                print(f'Epoch <{epoch+1}/{epochs}>, Train_Loss: {avg_epoch_loss:.4f}')
        print('Training process has finished. Saving trained model\n')
        print('Starting testing')
        # Save the model
        save_path = f'./{fold}-fold-'+model.model_name+'.pth'
        torch.save(model, save_path)

        # Evaluate the model
        evl_model = torch.load(save_path)
        avg_evl_loss = validate(evl_model, test_loader, loss_func, device)
        print(f"{fold}-fold evl_loss: {avg_evl_loss:.4f}")
        results[fold] = avg_evl_loss
    print(f'K-fold cross validation results for {k_folds}')
    print('-------------------------')
    sum = 0.0
    for key, value in results.items():
        print(f'fold-{key}: {value}')
        sum+= value
    fold_avg = sum / k_folds
    print(f'Average: {fold_avg:.4f}')
```

Implementation

In this part, we will do the implementation, as required, the $K = 5$, we set the $epochs = 200$, $batch_size = 1500$, we still use MAE as the loss function because we want to see how much difference between the predicted value and the real value.

Firstly, we initialize our loss functions, optimizers, and other related parameters.

```
# declare which will hold the device we're training on (CPU or GPU)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"computation device: {device}\n")

# Put our models into the device.
kf_convnet = ConvNet('kf_convnet').to(device)
```

```python

kf_mlp = MLP('kf_mlp').to(device)

# Define the loss function and optimizer
mae_loss = nn.L1Loss()
mse_loss = nn.MSELoss()

def rmse_loss(y, y_hat):
    mse = nn.MSELoss()
    return torch.sqrt(mse(y, y_hat))
rmse_loss = rmse_loss

# Define the optimizer
lr = 0.001
weight_decay = 0.001
kf_conv_optimizer = torch.optim.Adam(kf_convnet.parameters(), lr=lr, weight_decay=weight_decay)
kf_mlp_optimizer = torch.optim.Adam(kf_mlp.parameters(), lr=lr, weight_decay=weight_decay)
```

Then we will use below snippets to train our MLP and CNN model.

```python
mlp_test_loaders = k_fold_train(kf_mlp, img_dataset, 200, kf_mlp_optimizer, 1500, mae_loss, device, 5)
```

```python
cnn_test_loaders = k_fold_train(kf_convnet, img_dataset, 200, kf_conv_optimizer, 1500, mae_loss, device, 5)
```

After the training execution done, we can see the results in the terminal as the following, the results of MLP is:

```
K-fold cross validation results for 5
-----------------------------
fold-0: 3.708713710308075
fold-1: 3.7527145743370056
fold-2: 3.7509382367134094
fold-3: 3.75701767206192
fold-4: 3.443160057067871
Average: 3.6825
```

(Figure. T2-C-1)

The reults of CNN is:

```
K-fold cross validation results for 5
------------------------
fold-0: 3.6934342980384827
fold-1: 3.470215916633606
fold-2: 3.8090019822120667
fold-3: 3.4993010759353638
fold-4: 3.9616708159446716
Average: 3.6867
```

(Figure. T2-C-2)

Subtask D: Inference

In this part, we will choose the best performance model and evaluate it on our test set. We will define two functions as the following:

- `rmse_loss`: As there is no loss function of $RMSE$ in PyTorch, we have to define it on our own according to the formula we mentioned in Task 1.
- `multi_test`: Evaluate the model by given modle path and test set. Return the MAE, MSE, and RMSE, then plot the distribution scatter of predict value and real value.

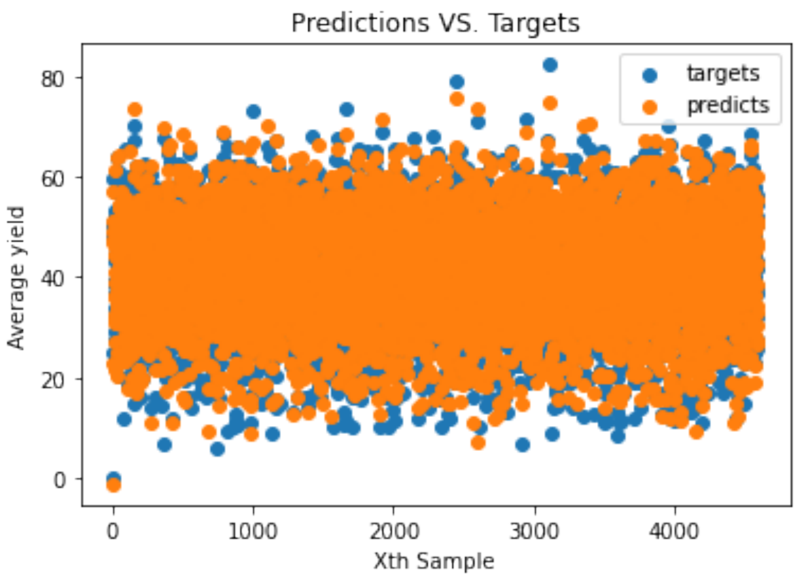
```
def rmse_loss(y, y_hat):
        mse = nn.MSELoss()
        return torch.sqrt(mse(y, y_hat))
```

```
def multi_test(model_path, test_loader, device):
    """evaluate the model, print mae, mse, rmse, and generate a scatter plot."""
    mae_loss = nn.L1Loss()
    mse_loss = nn.MSELoss()

    model = torch.load(model_path)
    model.eval()
    model.to(device)

    counter, mae_total, mse_total, rmse_total = 0, 0.0, 0.0, 0.0

    predicts_a = np.empty((1,1))  # initialize a empty variable to save the predicts
    targets_a = np.empty((1,1))   # initialize a empty variable to save the targets
    for i, data in enumerate(test_loader):
        inputs, targets = data[0].to(device), data[1].to(device)

        predicts = model(inputs)

        mae = mae_loss(predicts, targets)
        mse = mse_loss(predicts, targets)
        rmse = rmse_loss(predicts, targets)

        mae_total += mae.item()
        mse_total += mse.item()
        rmse_total += rmse.item()

        predicts_a = np.append(predicts_a, predicts.cpu().detach().numpy())
        targets_a = np.append(targets_a, targets.cpu().detach().numpy())
    print(f"MAE: {(mae_total/counter):.4f}\tMSE: {(mse_total/counter):.4f}\tRMSE{(rmse_total/counter):.4f}")

    x = range(1, len(targets_a)+1)
    plt.scatter(x, predicts_a, label="predicts")
```

```
    plt.scatter(x, targets_a, label="targets")
    plt.title("Predictions VS. Targets")
    plt.xlabel("Xth Sample")
    plt.ylabel("Average yield")
    plt.legend()
```

After defining above functions, in last subtask, we know the best perform model is cnn model in folder 0. Therefore, we will report this model with below snippet:

```
multi_test(
    model_path='./0-fold-kf_convnet.pth',
    test_set=test_loaders[0],
    device=device
)
```



(Figure. T2-D-1)

From above result we can see, predict values almost cover the real values. Therefore, we can say the model was trained well.