HW 3

Total points: 80

Name(s) and EID(s):

Sushanth Ravichandran (sr56925)

Sankeerth Viswanadhuni (vps386)

If you prefer, you can work in groups of two. **Please note that only one student per team needs to submit the assignment but make sure to include both students' names and UT EIDs.**

For any question that requires a handwritten solution, you may upload scanned images of your solution in the notebook or attach them to the assignment . You may write your solution using markdown as well.

Please make sure your code runs and the graphs and figures are displayed in your notebook and PDF before submitting.

```
pip install torch torchvision torchaudio
```

```
Collecting torch
  Downloading torch-2.4.1-cp312-none-macosx_11_0_arm64.whl.metadata (26 kB)
Collecting torchvision
  Downloading torchvision-0.19.1-cp312-cp312-macosx_11_0_arm64.whl.metadata (6.0 kB)
Collecting torchaudio
  Downloading torchaudio-2.4.1-cp312-cp312-macosx_11_0_arm64.whl.metadata (6.4 kB)
Requirement already satisfied: filelock in /opt/anaconda3/lib/python3.12/site-packages (from torch) (3.13.1)
Requirement already satisfied: typing-extensions>=4.8.0 in /opt/anaconda3/lib/python3.12/site-packages (from torch) (4.1
Requirement already satisfied: sympy in /opt/anaconda3/lib/python3.12/site-packages (from torch) (1.12)
Requirement already satisfied: networkx in /opt/anaconda3/lib/python3.12/site-packages (from torch) (3.2.1)
Requirement already satisfied: jinja2 in /opt/anaconda3/lib/python3.12/site-packages (from torch) (3.1.4)
Requirement already satisfied: fsspec in /opt/anaconda3/lib/python3.12/site-packages (from torch) (2024.3.1)
Requirement already satisfied: setuptools in /opt/anaconda3/lib/python3.12/site-packages (from torch) (69.5.1)
Requirement already satisfied: numpy in /opt/anaconda3/lib/python3.12/site-packages (from torchvision) (1.26.4)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /opt/anaconda3/lib/python3.12/site-packages (from torchvision) (
Requirement already satisfied: MarkupSafe>=2.0 in /opt/anaconda3/lib/python3.12/site-packages (from jinja2->torch) (2.1.
Requirement already satisfied: mpmath>=0.19 in /opt/anaconda3/lib/python3.12/site-packages (from sympy->torch) (1.3.0)
Downloading torch-2.4.1-cp312-none-macosx_11_0_arm64.whl (62.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.1/62.1 MB 9.1 MB/s eta 0:00:0000:0100:01m
Downloading torchvision-0.19.1-cp312-cp312-macosx_11_0_arm64.whl (1.7 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.7/1.7 MB 7.2 MB/s eta 0:00:0000:0100:01m
Downloading torchaudio-2.4.1-cp312-cp312-macosx_11_0_arm64.whl (1.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.8/1.8 MB 9.9 MB/s eta 0:00:00ta 0:00:01
Installing collected packages: torch, torchvision, torchaudio
Successfully installed torch-2.4.1 torchaudio-2.4.1 torchvision-0.19.1
Note: you may need to restart the kernel to use updated packages.
```

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

Choose files 2 files

- **AML_HW3_1.jpeg**(image/jpeg) - 138433 bytes, last modified: 16/10/2024 - 100% done
- **AML_HW3_2.jpeg**(image/jpeg) - 151850 bytes, last modified: 16/10/2024 - 100% done

```
Saving AML_HW3_1.jpeg to AML_HW3_1.jpeg
Saving AML_HW3_2.jpeg to AML_HW3_2.jpeg
```

Q1. (15 points) Tensorflow Playground

In this question, you will be playing with Tensorflow Playground.

Select **Classification** as the Problem Type. Among the four datasets shown in DATA, please select the top left dataset.

Use the following settings as the DEFAULT settings for all subquestions:

- Learning rate = 0.01
- Activation = ReLU
- Regularization = None
- Ratio of training to test data = 50%
- Noise = 0
- Batch Size = 30

- Input features as X_1 and X_2
- One hidden layer with 4 neurons

For all questions below, it is ok if you did not run to the exact number of epochs specified as long as it is within ± 20 epochs. For example, if you are asked to run 1000 epochs and you ran 1012 epochs, it is fine because it's within 980 and 1020.

Part 1. (4 pts) Effect of activation function (keep other settings the same as in DEFAULT)

- Using ReLU as the activation function
- Using the Linear activation function

(a) Report the train and test losses for both at the end of 1000 epochs.
(b) What qualitative difference do you observe in the decision boundaries obtained and what do you think is the reason for this?

Answer

(a)

Activation	ReLU	Linear
Train Loss	0.014	0.524
Test Loss	0.020	0.481

(b) ReLU produced a clear and effective boundary that nicely separates the orange dots from the blue dots. Linear activation function produced a boundary that doesn't effectively separate the classes. Instead, it looks like a straight line that fails to capture the complex patterns in the data.

ReLU allows the model to create more complex relationships. It outputs zero for negative inputs but passes positive inputs unchanged. This means it can create nonlinear decision boundaries, which are essential for complex datasets.

Linear activation, on the other hand, treats inputs too simply, creating straight lines in decision boundaries. This is great for very simple problems, but for data like this, where the classes are intermingled in a nonlinear fashion, it fails to capture the necessary complexity

Part 2. (4 pts) Effect of number of hidden units (keep other settings the same as in DEFAULT)

- Use 2 neurons in the hidden layer
- Use 8 neurons in the hidden layer

(a) Report the train and test losses at the end of 1000 epochs.
(b) What do you observe in terms of the decision boundary obtained as the number of neurons increases and what do you think is the reason for this?

Answer

(a)

# Hidden Units	2	8
Train Loss	0.256	0.008
Test Loss	0.270	0.008

(b) Analysis of the Results ***Train Loss:*** With 2 hidden units, the training loss is 0.256. With 8 hidden units, the training loss significantly drops to 0.008. This indicates that the model with 8 hidden units is much better at fitting the training data. ***Test Loss:*** The test loss remains at 0.270 for the model with 2 hidden units, but it drops to 0.008 for the model with 8 hidden units. A low test loss in the model with 8 hidden units suggests that it generalizes well to unseen data.

With 2 Hidden Units:

The decision boundary tends to be simpler and may struggle to capture the underlying complexity of the data. This can lead to underfitting, where the model is too simplistic to learn the patterns present in the dataset.

With 8 Hidden Units:

The decision boundary becomes more complex and can effectively separate different classes. A well-fitted model with more neurons can create more intricate shapes, which allows it to capture the underlying distribution of the data better.

Part 3. (4 pts) Effect of Learning rate and number of epochs (keep other settings the same as in DEFAULT)

Set the learning rate to the following numbers

- 10
- 0.1
- 0.0001

(a) Report the train and test losses at the end of 100 epochs, 500 epochs and 1000 epochs respectively.
(b) What do you observe from the change of loss vs learning rate, and the change of loss vs epoch numbers? Also report your observations on the training and test loss curve (observe if you see noise for certain learning rates and reason why this is happening).

Answer

(a)

Learning rate: 10

Epoch	100	500	1000
Train Loss	0.566	0.566	0.566
Test Loss	0.695	0.693	0.693

(Copy and fill the tables for other learning rates below)

Learning rate: 0.1

Epoch	100	500	1000
Train Loss	0.008	0.001	0.000
Test Loss	0.011	0.004	0.003

Learning rate: 0.0001

Epoch	100	500	1000
Train Loss	0.498	0.444	0.407
Test Loss	0.548	0.483	0.436

(b) Change of Loss vs. Learning Rate

Learning Rate: 10

Losses are high and stagnant: The model does not learn effectively with such a high learning rate. The train and test losses remain constant around 0.566 and 0.693, indicating that the model is not updating weights appropriately. This suggests that the learning rate is too high, causing the model to diverge instead of converge.

Learning Rate: 0.1

Significant improvement: Both train and test losses drop significantly as the epochs increase. By 1000 epochs, the train loss is near 0.000 and test loss around 0.003. This shows that the model can learn effectively and generalizes well to the test set, indicating a good balance.

Learning Rate: 0.0001

Slow progress: The losses decrease but at a much slower rate. After 1000 epochs, train loss is 0.407 and test loss is 0.436. This indicates that while the model is learning, the small learning rate results in very slow convergence. It may require many more epochs to reach a better performance.

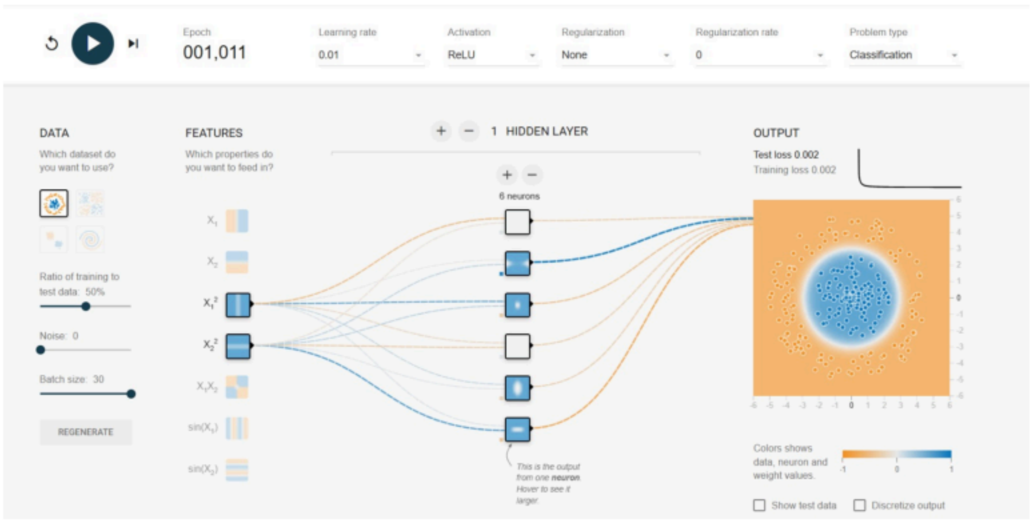
Part 4. (3 pts) Use the DEFAULT setting but choose a configuration to modify (e.g. network depth, input features, noise, etc.)

(a) State the change you made and attach the screenshot showing your full network, output and the parameters.
(b) Report the best train and test loss you obtain.
(c) Briefly justify why the modification you chose lead to those results.

(c) By increasing the number of input features and neurons in the hidden layer, I enhanced the model's capacity to learn complex patterns from the data. The additional input features (x2 and y2) provided more information for the model, which likely improved its ability to generalize to the test set. Increasing the number of neurons from 4 to 6 allowed the model to capture more intricate relationships within the data, leading to improved training and testing performance.

g'))

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Load the image
img1 = mpimg.imread('AML_HW3_1.jpeg')

# Create a figure to display the image
plt.figure(figsize=(10, 5))

# Display the image
plt.imshow(img1)
plt.axis('off')  # Turn off axis if you don't want to display it

# Show the figure
plt.show()
```


Epoch 001,011
Learning rate 0.01
Activation ReLU
Regularization None
Regularization rate 0
Problem type Classification

Ratio of training to test data: 50%
Noise: 0
Batch size: 30

1 HIDDEN LAYER
6 neurons

Test loss 0.002
Training loss 0.002

DATA
Which dataset do you want to use?
REGENERATE
FEATURES
Which properties do you want to feed in?
X_1
X_2
X_1^2
X_2^2
X_1X_2
$\sin(X_1)$
$\sin(X_2)$
This is the output from one neuron. Hover to see it larger.
OUTPUT
Colors shows data, neuron and weight values.
-1
0
1
Show test data
Discretize output

Q2. (30 Points) - Coding Neural Networks with PyTorch

In this question we will code a simple MLP network with PyTorch and train it on the house cost dataset (provided in the supplementary files as *house_cost_data.csv*) that we used in HW1. We will then compare the results with linear regression. If you want to run it locally, please check out this link to install PyTorch. Otherwise, you can just use Google Colab.

Here is a tutorial for you to quickly to get familiar with PyTorch and finish the problems below.

This is a programming question. Please read through each subpart of this question carefully. You are required to add lines of code as specified in the code cells. Please carefully read through the comments in the code cells to identify what code is to be written, where it is to be written and how many lines of code are required. Code is to be added between the ## START CODE ## and ## END CODE ## comments and in place of the keyword None. In certain cases, the number of lines of code that are to be written will be specified. For example, ## START CODE ## (1 line of code) specifies that only 1 line of code is to be added between the ## START CODE ## and ## END CODE ## comments. In case there is no information on the required number of lines, you are allowed to add any number of lines of code.

The following question covers a dataset for house cost and linear models in python. The categorical variables and rows with missing variables are removed to make it easier to run the models.

NOTE

- Only use the following code block if you are using Google Colab. If you are using Jupyter Notebook, please ignore this code block. You can directly upload the file to your Jupyter Notebook file systems.
- It will prompt you to select a local file. Click on “Choose Files” then select and upload the file. Wait for the file to be 100% uploaded. You should see the name of the file once Colab has uploaded it.

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

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import tqdm
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score
import copy

%matplotlib inline
pd.options.mode.chained_assignment = None

df = pd.read_csv('house_cost_data.csv')
X = df.drop(['house_cost'],axis=1).to_numpy()
Y = df['house_cost'].to_numpy()
```

Part 1 (2pts) - Preprocessing

Split the dataset into train(75% of data) and test(25% data) using the train_test_split function with random_state = 50.

```
##  START CODE  ## (1 line of code)
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25, random_state=50)
##  END CODE  ##
```

Scale the data (not including target) so that each of the independent variables would have zero mean and unit variance. You can use the StandardScaler() class for this.

Not Scaling y data (FYR); just renaming the variable

```
##  START CODE  ##
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Keep the output variable as is, no scaling
y_train_scaled = y_train  # Keep the original y_train
y_test_scaled = y_test    # Keep the original y_test
##  END CODE  ##
```

Split the train dataset into train(85% of train data) and validation(15% of train data) using the train_test_split function with random_state = 50.

```
##  START CODE  ## (1 line of code)
X_train_new, X_val, y_train_new, y_val = train_test_split(X_train_scaled, y_train_scaled, test_size=0.15, random_state=50)
##  END CODE  ##
```

Convert the numpy arrays to torch tensors. Make sure that the tensors are of dtype torch.float32. Also make sure that the target tensors (y) are 2-dimensional by using .reshape(-1,1) on the tensors.

```
# Convert to 2D PyTorch tensors
##  START CODE  ##


X_train_tensor = torch.tensor(X_train_new, dtype=torch.float32)
X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test_scaled, dtype=torch.float32)

# Convert y variables to tensors without scaling
y_train_tensor = torch.tensor(y_train_new.reshape(-1, 1), dtype=torch.float32)
y_val_tensor = torch.tensor(y_val.reshape(-1, 1), dtype=torch.float32)
y_test_tensor = torch.tensor(y_test.reshape(-1, 1), dtype=torch.float32)


##  END CODE  ##
```

Part 2 (13 pts) - Train and Eval functions

(3 pts) First we will write and eval function which takes a model, input(X) and target(y) and evaluates the MSE and R2.

```
def eval(model, X, y):
    model.eval()
    ##  START CODE  ##
    with torch.no_grad():
        # Forward pass: compute the model's predictions
        y_pred = model(X)

        # Convert the predictions and target to NumPy arrays for metric calculations
        y_pred_np = y_pred.detach().cpu().numpy()
        y_np = y.detach().cpu().numpy()

        # Calculate Mean Squared Error (MSE)
        mse = mean_squared_error(y_np, y_pred_np)

        # Calculate R² score
        r2 = r2_score(y_np, y_pred_np)
    ##  END CODE  ##
    return mse, r2
```

(10 pts) Now we will write the train function. Refer here for the tutorial. Make sure to understand what the optimizer does.

```
def train(model, X_train, y_train, X_valid, y_valid, n_epochs, batch_size, optimizer, loss_fn):
    # parameters to store the best model on validation MSE
    best_mse = np.inf   # init to infinity
    best_weights = None
    train_history = []
    valid_history = []
    batch_start = torch.arange(0, len(X_train), batch_size)

    for epoch in range(n_epochs):
        model.train()
        with tqdm.tqdm(batch_start, unit="batch", mininterval=0, disable=False) as bar:
            bar.set_description(f"Epoch {epoch}")
            train_losses = []
            for start in bar:
                # take a batch with the help of 'start' variable
                ##  START CODE  ## (2 lines of code)
                end = start + batch_size
                X_batch = X_train[start:end]  # Select batch of input features
                y_batch = y_train[start:end]  # Select batch of target values
                ##  END CODE  ##

                # forward pass on the batch
                ##  START CODE  ## (1 line of code)
                y_pred = model(X_batch)

                ##  END CODE  ##

                # calculate loss using loss_fn
                ##  START CODE  ## (1 line of code)
                loss = loss_fn(y_pred, y_batch)
                ##  END CODE  ##

                # backward pass
                ##  START CODE  ## (2 lines of code)
                optimizer.zero_grad()  # Clear previous gradients
                loss.backward()  # Backpropagation to compute gradients
                ##  END CODE  ##

                # update weights (1 line of code)
                ##  START CODE  ##
                optimizer.step()
                ##  END CODE  ##

                # print progress
                bar.set_postfix(mse=float(loss))
                train_losses.append(float(loss))
        train_history.append(sum(train_losses)/len(train_losses))
        # evaluate validation MSE and R2 at end of each epoch
        ##  START CODE  ## (1 line of code)
        mse, _ = eval(model, X_valid, y_valid)
        ##  END CODE  ##

        # store the best model depending on MSE
        valid_history.append(mse)
        if mse < best_mse:
            ##  START CODE  ## (when storing best model weights make sure to use copy.deepcopy())
            best_mse = mse  # Update the best MSE
            best_weights = copy.deepcopy(model.state_dict())
            ##  END CODE  ##

    print("Validation MSE: %.2f" % best_mse)
    print("Validation RMSE: %.2f" % np.sqrt(best_mse))
    return best_weights, train_history, valid_history, best_mse
```

Part 3 (15 pts) - Train the model and analyze results

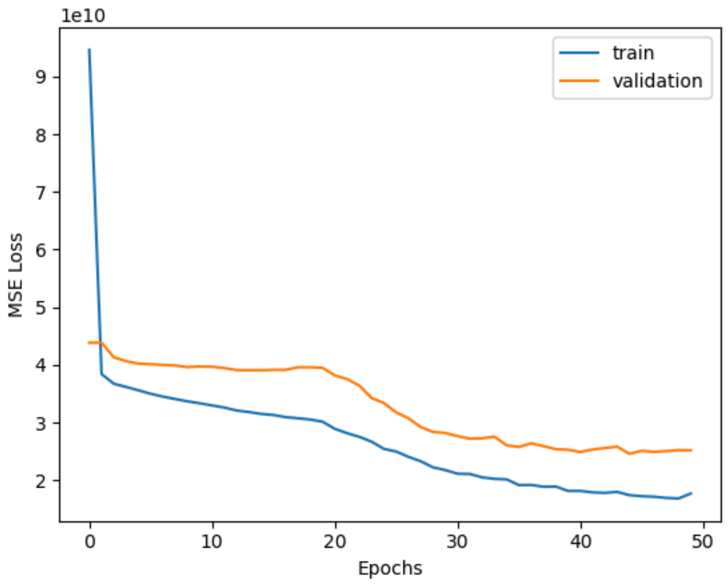
(5 pts) Initialize a model with 2 hidden layers of size 64 and 16 and ReLU activation function. Use the MSELoss and Adam optimizer from PyTorch to train the model. Sweep learning rates in [10, 1, 0.1, 0.01] and find the best learning rate with the smaller validation set MSE. For the best model, evaluate the performance on the test set and plot training curves(loss vs epochs).

```
best_valid_mse = np.inf
best_weights = None
best_train_history = None
best_valid_history = None
best_lr = None
for lr in [10, 1, 0.1, 0.01]:
    print("Running with LR: ", lr)
    # Define the model using nn.Sequential
```

```
        ##  START CODE  ##
        model = nn.Sequential(
            nn.Linear(X_train_tensor.shape[1], 64),  # Input layer to first hidden layer
            nn.ReLU(),  # Activation function
            nn.Linear(64, 16),  # First hidden layer to second hidden layer
            nn.ReLU(),  # Activation function
            nn.Linear(16, 1)  # Second hidden layer to output layer
        )
        ##  END CODE  ##

        #  Define loss function and optimizer
        ##  START CODE  ##
        loss_fn = nn.MSELoss()  # Mean Squared Error Loss
        optimizer = optim.Adam(model.parameters(), lr=lr)  # Adam optimizer with learning rate
        ##  END CODE  ##

        n_epochs = 50   # number of epochs to run
        batch_size = 256  # size of each batch

        # train the model
        weights, train_history, valid_history, valid_mse = train(model, X_train_tensor, y_train_tensor, X_val_tensor, y_val_tens
        # store values for best LR
        if valid_mse < best_valid_mse:
            best_valid_mse = valid_mse
            best_weights = weights
            best_train_history = train_history
            best_valid_history = valid_history
            best_lr = lr

    print("Best LR: ", best_lr)
    print("Best Validation MSE: ", best_valid_mse)
    Validation RMSE: 202910.12
    Best LR:  1
    Best Validation MSE:  24582470000.0
```

```
# restore model using the .load_state_dict() function
##  START CODE  ## (1 line of code)
model.load_state_dict(best_weights)

##  END CODE  ##

# calculate test performance using the eval function we defined above (MSE and R2)
##  START CODE  ##
test_mse, test_r2 = eval(model, X_test_tensor, y_test_tensor)
##  END CODE  ##

print("Test MSE: %.2f" % test_mse)
print("Test R2: %.2f" % test_r2)
plt.plot(best_train_history, label='train')
plt.plot(best_valid_history, label='validation')
plt.legend(loc="upper right")
plt.xlabel('Epochs')
plt.ylabel('MSE Loss')
plt.show()
```

```
Test MSE: 26794692608.00
Test R2: 0.81
```



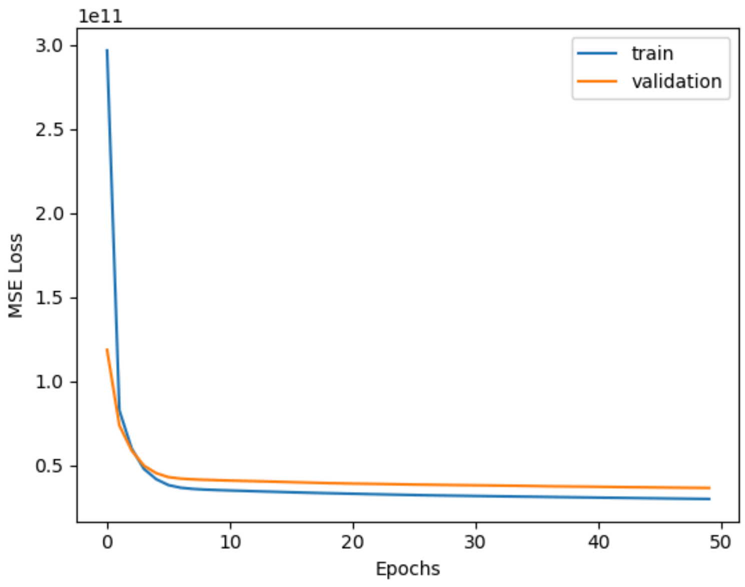
(5 pts) Now we will analyze the performance of a smaller model on the same dataset. Define a smaller model with just 1 hidden layer of size 32 and the ReLU activation function. Use the same loss function and optimizer. Sweep learning rates in [1, 0.1, 0.01, 0.001] to find the best learning rate. Finally, for the best model, evaluate the performance on the test set and plot training curves(loss vs epochs).

```
small_model_best_valid_mse = np.inf
small_model_best_weights = None
small_model_best_train_history = None
small_model_best_valid_history = None
best_lr = None
for lr in [1, 0.1, 0.01, 0.001]:
    print("Running with LR: ", lr)
    # Define the model using nn.Sequential
    ##  START CODE  ##
    small_model = nn.Sequential(
        nn.Linear(X_train.shape[1], 32),  # Input layer to hidden layer 1
        nn.ReLU(),                         # Activation function
        nn.Linear(32, 1),                # Hidden layer 1 to hidden layer 2
    )
    ##  END CODE  ##

    # Define loss function and optimizer
    ##  START CODE  ##
    loss_fn = nn.MSELoss()  # Mean Squared Error Loss
    optimizer = optim.Adam(small_model.parameters(), lr=lr)  # Adam optimizer with current learning rate
    ##  END CODE  ##

    n_epochs = 50   # number of epochs to run
    batch_size = 256  # size of each batch

    # train the model
    weights, train_history, valid_history, valid_mse = train(small_model, X_train_tensor, y_train_tensor, X_val_tensor, y_va

    # store values for best LR
    if valid_mse < small_model_best_valid_mse:
        small_model_best_valid_mse = valid_mse  # Update best validation MSE
        small_model_best_weights = copy.deepcopy(small_model.state_dict())  # Store best model weights
        small_model_best_train_history = train_history  # Store training history
        small_model_best_valid_history = valid_history  # Store validation history
        best_lr = lr  # Update best learning rate

print("Best LR: ", best_lr)
print("Best Validation MSE: ", small_model_best_valid_mse)
```

```
Epoch 49: 100%|██████████| 54/54 [00:00<00:00, 227.94batch/s, mse=2.5e+11]
Validation MSE: 273822384128.00
Validation RMSE: 523280.41
Running with LR:  0.001
Epoch 0: 100%|██████████| 54/54 [00:00<00:00, 215.21batch/s, mse=4.17e+11]
Epoch 1: 100%|██████████| 54/54 [00:00<00:00, 195.11batch/s, mse=4.17e+11]
Epoch 2: 100%|██████████| 54/54 [00:00<00:00, 218.52batch/s, mse=4.17e+11]
Epoch 3: 100%|██████████| 54/54 [00:00<00:00, 216.49batch/s, mse=4.17e+11]
Epoch 4: 100%|██████████| 54/54 [00:00<00:00, 197.15batch/s, mse=4.17e+11]
Epoch 5: 100%|██████████| 54/54 [00:00<00:00, 248.48batch/s, mse=4.17e+11]
Epoch 6: 100%|██████████| 54/54 [00:00<00:00, 250.20batch/s, mse=4.17e+11]
Epoch 7: 100%|██████████| 54/54 [00:00<00:00, 249.00batch/s, mse=4.17e+11]
Epoch 8: 100%|██████████| 54/54 [00:00<00:00, 232.17batch/s, mse=4.17e+11]
Epoch 9: 100%|██████████| 54/54 [00:00<00:00, 276.21batch/s, mse=4.17e+11]
Epoch 10: 100%|██████████| 54/54 [00:00<00:00, 242.73batch/s, mse=4.17e+11]
Epoch 11: 100%|██████████| 54/54 [00:00<00:00, 265.15batch/s, mse=4.17e+11]
Epoch 12: 100%|██████████| 54/54 [00:00<00:00, 268.71batch/s, mse=4.17e+11]
Epoch 13: 100%|██████████| 54/54 [00:00<00:00, 249.16batch/s, mse=4.17e+11]
Epoch 14: 100%|██████████| 54/54 [00:00<00:00, 300.74batch/s, mse=4.17e+11]
Epoch 15: 100%|██████████| 54/54 [00:00<00:00, 255.95batch/s, mse=4.17e+11]
Epoch 16: 100%|██████████| 54/54 [00:00<00:00, 270.09batch/s, mse=4.17e+11]
Epoch 17: 100%|██████████| 54/54 [00:00<00:00, 230.65batch/s, mse=4.17e+11]
Epoch 18: 100%|██████████| 54/54 [00:00<00:00, 313.90batch/s, mse=4.17e+11]
Epoch 19: 100%|██████████| 54/54 [00:00<00:00, 328.42batch/s, mse=4.16e+11]
Epoch 20: 100%|██████████| 54/54 [00:00<00:00, 285.03batch/s, mse=4.16e+11]
Epoch 21: 100%|██████████| 54/54 [00:00<00:00, 220.37batch/s, mse=4.16e+11]
Epoch 22: 100%|██████████| 54/54 [00:00<00:00, 297.37batch/s, mse=4.16e+11]
Epoch 23: 100%|██████████| 54/54 [00:00<00:00, 274.98batch/s, mse=4.16e+11]
Epoch 24: 100%|██████████| 54/54 [00:00<00:00, 285.52batch/s, mse=4.16e+11]
Epoch 25: 100%|██████████| 54/54 [00:00<00:00, 277.53batch/s, mse=4.16e+11]
Epoch 26: 100%|██████████| 54/54 [00:00<00:00, 248.52batch/s, mse=4.16e+11]
Epoch 27: 100%|██████████| 54/54 [00:00<00:00, 320.02batch/s, mse=4.16e+11]
Epoch 28: 100%|██████████| 54/54 [00:00<00:00, 327.00batch/s, mse=4.16e+11]
Epoch 29: 100%|██████████| 54/54 [00:00<00:00, 295.24batch/s, mse=4.16e+11]
Epoch 30: 100%|██████████| 54/54 [00:00<00:00, 284.76batch/s, mse=4.16e+11]
Epoch 31: 100%|██████████| 54/54 [00:00<00:00, 200.43batch/s, mse=4.16e+11]
Epoch 32: 100%|██████████| 54/54 [00:00<00:00, 289.20batch/s, mse=4.16e+11]
Epoch 33: 100%|██████████| 54/54 [00:00<00:00, 300.38batch/s, mse=4.16e+11]
Epoch 34: 100%|██████████| 54/54 [00:00<00:00, 292.34batch/s, mse=4.16e+11]
Epoch 35: 100%|██████████| 54/54 [00:00<00:00, 232.33batch/s, mse=4.16e+11]
Epoch 36: 100%|██████████| 54/54 [00:00<00:00, 191.78batch/s, mse=4.15e+11]
Epoch 37: 100%|██████████| 54/54 [00:00<00:00, 312.64batch/s, mse=4.15e+11]
Epoch 38: 100%|██████████| 54/54 [00:00<00:00, 334.09batch/s, mse=4.15e+11]
Epoch 39: 100%|██████████| 54/54 [00:00<00:00, 322.35batch/s, mse=4.15e+11]
Epoch 40: 100%|██████████| 54/54 [00:00<00:00, 252.48batch/s, mse=4.15e+11]
Epoch 41: 100%|██████████| 54/54 [00:00<00:00, 278.01batch/s, mse=4.15e+11]
Epoch 42: 100%|██████████| 54/54 [00:00<00:00, 313.07batch/s, mse=4.15e+11]
Epoch 43: 100%|██████████| 54/54 [00:00<00:00, 308.47batch/s, mse=4.15e+11]
Epoch 44: 100%|██████████| 54/54 [00:00<00:00, 248.23batch/s, mse=4.15e+11]
Epoch 45: 100%|██████████| 54/54 [00:00<00:00, 223.12batch/s, mse=4.15e+11]
Epoch 46: 100%|██████████| 54/54 [00:00<00:00, 306.63batch/s, mse=4.15e+11]
Epoch 47: 100%|██████████| 54/54 [00:00<00:00, 291.60batch/s, mse=4.15e+11]
Epoch 48: 100%|██████████| 54/54 [00:00<00:00, 288.82batch/s, mse=4.14e+11]
Epoch 49: 100%|██████████| 54/54 [00:00<00:00, 305.46batch/s, mse=4.14e+11]
Validation MSE: 445799890944.00
Validation RMSE: 667682.50
Best LR:  1
Best Validation MSE:  36593545000.0
```

```
# restore model using the .load_state_dict() function
##  START CODE  ## (1 line of code)
small_model.load_state_dict(small_model_best_weights)

##  END CODE  ##

# calculate test performance using the eval function we defined above (MSE and R2)
##  START CODE  ##
test_mse, test_r2 = eval(small_model, X_test_tensor, y_test_tensor)
##  END CODE  ##

print("Test MSE: %.2f" % test_mse)
print("Test R2: %.2f" % test_r2)
plt.plot(small_model_best_train_history, label='train')
plt.plot(small_model_best_valid_history, label='validation')
plt.legend(loc="upper right")
plt.xlabel('Epochs')
plt.ylabel('MSE Loss')
plt.show()
```

```
Test MSE: 36559396864.00
Test R2: 0.74
```



- Compared to the linear model used in Assignment 1, the MLPs exhibit a significantly higher degree of fit, achieving an (R^2) value of 0.74 with a simple model that includes just one hidden layer, while the regularized linear model reached approximately 0.65 (R^2).
- The more complex MLP not only converges to a model with a better fit, as indicated by a hig her (R^2 of 0.84) and lower MSE loss, but it also achieves this in fewer epochs.
- However, a potential drawback of the more complex MLP is the noticeable overfitting, as illustrated by the gap between the training and test loss curvtails!

Answer

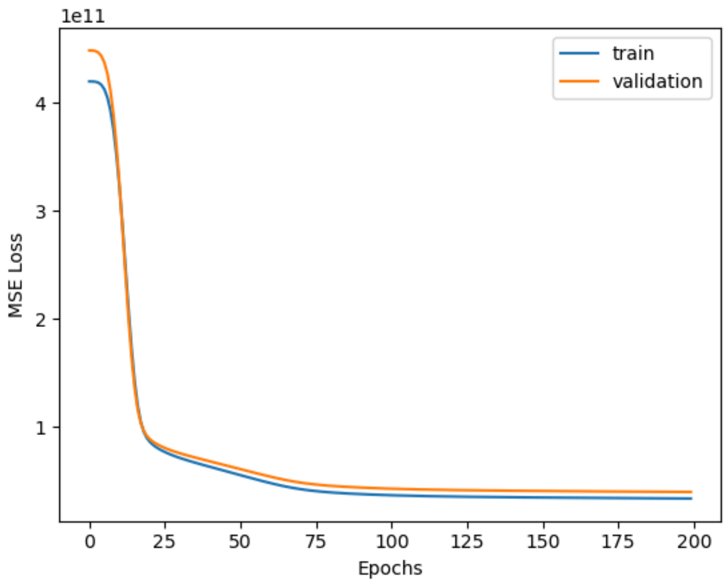
(5 pts) Finally, we will train a bigger MLP model on the same dataset. Define a bigger model with 3 hidden layers with sizes (256, 32, 8) and the ReLU activation function. Use the same loss function and optimizer. For the best model on the validation data, evaluate the performance on the test set and plot training curves(loss vs epochs).

```
# Define the model using nn.Sequential
##  START CODE  ##
big_model = nn.Sequential(
    nn.Linear(X_train_tensor.shape[1], 256),  # Input layer to first hidden layer
    nn.ReLU(),
    nn.Linear(256, 32),                       # First hidden layer to second hidden layer
    nn.ReLU(),
    nn.Linear(32, 8),                         # Second hidden layer to third hidden layer
    nn.ReLU(),
    nn.Linear(8, 1)                           # Output layer
)
##  END CODE  ##

# Define loss function and optimizer
##  START CODE  ##
loss_fn = nn.MSELoss()                      # Mean Squared Error Loss
optimizer = optim.Adam(big_model.parameters(), lr=0.001)  # Adam optimizer
##  END CODE  ##

n_epochs = 200   # number of epochs to run
batch_size = 256  # size of each batch

# train the model
big_model_best_weights, big_model_train_history, big_model_valid_history, big_model_best_valid_mse \
= train(big_model, X_train_tensor, y_train_tensor, X_val_tensor, y_val_tensor, n_epochs, batch_size, optimizer, loss_fn)
```

```
Epoch 0: 100%|████████████████| 54/54 [00:00<00:00, 278.02batch/s, mse=4.17e+11]
Epoch 1: 100%|████████████████| 54/54 [00:00<00:00, 228.81batch/s, mse=4.17e+11]
Epoch 2: 100%|████████████████| 54/54 [00:00<00:00, 168.51batch/s, mse=4.16e+11]
Epoch 3: 100%|████████████████| 54/54 [00:00<00:00, 217.86batch/s, mse=4.15e+11]
Epoch 4: 100%|████████████████| 54/54 [00:00<00:00, 213.64batch/s, mse=4.12e+11]
Epoch 5: 100%|████████████████| 54/54 [00:00<00:00, 194.40batch/s, mse=4.06e+11]
Epoch 6: 100%|████████████████| 54/54 [00:00<00:00, 178.77batch/s, mse=3.97e+11]
Epoch 7: 100%|████████████████| 54/54 [00:00<00:00, 215.95batch/s, mse=3.82e+11]
Epoch 8: 100%|████████████████| 54/54 [00:00<00:00, 219.16batch/s, mse=3.61e+11]
Epoch 9: 100%|████████████████| 54/54 [00:00<00:00, 190.59batch/s, mse=3.33e+11]
Epoch 10: 100%|███████████████| 54/54 [00:00<00:00, 223.01batch/s, mse=2.99e+11]
Epoch 11: 100%|████████████████| 54/54 [00:00<00:00, 248.60batch/s, mse=2.6e+11]
Epoch 12: 100%|████████████████| 54/54 [00:00<00:00, 230.05batch/s, mse=2.2e+11]
Epoch 13: 100%|███████████████| 54/54 [00:00<00:00, 185.86batch/s, mse=1.81e+11]
Epoch 14: 100%|███████████████| 54/54 [00:00<00:00, 224.42batch/s, mse=1.47e+11]
Epoch 15: 100%|████████████████| 54/54 [00:00<00:00, 239.33batch/s, mse=1.2e+11]
Epoch 16: 100%|███████████████| 54/54 [00:00<00:00, 204.50batch/s, mse=1.01e+11]
Epoch 17: 100%|███████████████| 54/54 [00:00<00:00, 150.33batch/s, mse=8.86e+10]
Epoch 18: 100%|███████████████| 54/54 [00:00<00:00, 192.42batch/s, mse=8.13e+10]
Epoch 19: 100%|███████████████| 54/54 [00:00<00:00, 168.75batch/s, mse=7.69e+10]
Epoch 20: 100%|███████████████| 54/54 [00:00<00:00, 140.57batch/s, mse=7.39e+10]
Epoch 21: 100%|███████████████| 54/54 [00:00<00:00, 161.88batch/s, mse=7.18e+10]
Epoch 22: 100%|██████████████████| 54/54 [00:00<00:00, 202.93batch/s, mse=7e+10]
Epoch 23: 100%|███████████████| 54/54 [00:00<00:00, 213.96batch/s, mse=6.86e+10]
Epoch 24: 100%|███████████████| 54/54 [00:00<00:00, 183.90batch/s, mse=6.73e+10]
Epoch 25: 100%|███████████████| 54/54 [00:00<00:00, 229.47batch/s, mse=6.61e+10]
Epoch 26: 100%|███████████████| 54/54 [00:00<00:00, 177.31batch/s, mse=6.51e+10]
Epoch 27: 100%|███████████████| 54/54 [00:00<00:00, 228.45batch/s, mse=6.42e+10]
Epoch 28: 100%|███████████████| 54/54 [00:00<00:00, 247.55batch/s, mse=6.33e+10]
Epoch 29: 100%|███████████████| 54/54 [00:00<00:00, 232.19batch/s, mse=6.25e+10]
Epoch 30: 100%|███████████████| 54/54 [00:00<00:00, 210.99batch/s, mse=6.17e+10]
Epoch 31: 100%|███████████████| 54/54 [00:00<00:00, 184.93batch/s, mse=6.09e+10]
Epoch 32: 100%|███████████████| 54/54 [00:00<00:00, 224.98batch/s, mse=6.01e+10]
Epoch 33: 100%|███████████████| 54/54 [00:00<00:00, 216.57batch/s, mse=5.94e+10]
Epoch 34: 100%|███████████████| 54/54 [00:00<00:00, 232.45batch/s, mse=5.86e+10]
Epoch 35: 100%|███████████████| 54/54 [00:00<00:00, 170.07batch/s, mse=5.79e+10]
Epoch 36: 100%|███████████████| 54/54 [00:00<00:00, 214.24batch/s, mse=5.71e+10]
Epoch 37: 100%|███████████████| 54/54 [00:00<00:00, 170.37batch/s, mse=5.63e+10]
Epoch 38: 100%|███████████████| 54/54 [00:00<00:00, 234.49batch/s, mse=5.55e+10]
Epoch 39: 100%|███████████████| 54/54 [00:00<00:00, 195.34batch/s, mse=5.47e+10]
Epoch 40: 100%|███████████████| 54/54 [00:00<00:00, 144.77batch/s, mse=5.39e+10]
Epoch 41: 100%|████████████████| 54/54 [00:00<00:00, 215.39batch/s, mse=5.3e+10]
Epoch 42: 100%|███████████████| 54/54 [00:00<00:00, 227.03batch/s, mse=5.22e+10]
Epoch 43: 100%|███████████████| 54/54 [00:00<00:00, 224.95batch/s, mse=5.13e+10]
Epoch 44: 100%|███████████████| 54/54 [00:00<00:00, 226.91batch/s, mse=5.05e+10]
Epoch 45: 100%|███████████████| 54/54 [00:00<00:00, 174.06batch/s, mse=4.96e+10]
Epoch 46: 100%|███████████████| 54/54 [00:00<00:00, 244.08batch/s, mse=4.87e+10]
Epoch 47: 100%|███████████████| 54/54 [00:00<00:00, 249.58batch/s, mse=4.79e+10]
Epoch 48: 100%|████████████████| 54/54 [00:00<00:00, 227.40batch/s, mse=4.7e+10]
Epoch 49: 100%|███████████████| 54/54 [00:00<00:00, 164.39batch/s, mse=4.61e+10]
Epoch 50: 100%|███████████████| 54/54 [00:00<00:00, 193.61batch/s, mse=4.53e+10]
Epoch 51: 100%|███████████████| 54/54 [00:00<00:00, 192.16batch/s, mse=4.44e+10]
Epoch 52: 100%|███████████████| 54/54 [00:00<00:00, 205.46batch/s, mse=4.35e+10]
Epoch 53: 100%|███████████████| 54/54 [00:00<00:00, 219.13batch/s, mse=4.27e+10]
Epoch 54: 100%|███████████████| 54/54 [00:00<00:00, 174.92batch/s, mse=4.18e+10]
Epoch 55: 100%|████████████████| 54/54 [00:00<00:00, 224.52batch/s, mse=4.1e+10]
Epoch 56: 100%|███████████████| 54/54 [00:00<00:00, 175.96batch/s, mse=4.02e+10]
```

```python
# restore model using the .load_state_dict() function
##  START CODE  ## (1 line of code)
big_model.load_state_dict(big_model_best_weights)  # Load the best weights

##  END CODE  ##

# calculate test performance using the eval function we defined above (MSE and R2)
##  START CODE  ##
test_mse, test_r2 = eval(big_model, X_test_tensor, y_test_tensor)  # Evaluate on the test set
##  END CODE  ##

print("Test MSE: %.2f" % test_mse)
print("Test R2: %.2f" % test_r2)
plt.plot(big_model_train_history, label='train')
plt.plot(big_model_valid_history, label='validation')
plt.legend(loc="upper right")
plt.xlabel('Epochs')
plt.ylabel('MSE Loss')
plt.show()
```

```
Test MSE: 39172497408.00
Test R2: 0.72
```



What do you notice in the curve above? In particular, what happens to the train loss vs validation loss as you keep training the model? Why does this happen and what technique(s) can you use to avoid this?

Answer

Training Loss: The training loss decreases consistently as the number of epochs increases, indicating that the model is learning from the training data.

Validation Loss: The validation loss initially decreases but then starts to fluctuate and eventually increases, indicating that the model begins to overfit the training data after a certain point.

To mitigate overfitting, we can consider the following techniques:

Early Stopping: Monitor the validation loss during training and stop when it starts to increase. This helps in selecting the best model before overfitting occurs.

Regularization: Techniques like L1 (Lasso) and L2 (Ridge) regularization add a penalty for larger weights, which can help to prevent overfitting.

Dropout: Introduce dropout layers in your model to randomly drop a fraction of neurons during training, which helps prevent co-adaptation of features.

Q3. (10 points) - Principal Component Analysis

Consider a set of data points $\{x_1, x_2, ... x_N\}$ where each $x_i \in \mathbb{R}^d$, given to you after centering, i.e., $\frac{1}{N}\sum_{i=1}^{N} x_i = 0$.

Now suppose you want to project this data on a single unit vector given by u by learning an appropriate u. Show that, for learning u, minimizing the residual of the projections computed by the squared error between the projected data and the original data is equivalent to maximizing the variance of the projections. Hint : the projection of an x on a unit vector u is given by $(x^T u)u$.

\section*{Principal Component Analysis}

Consider a set of data points $\{x_1, x_2, \ldots, x_N\}$ where each $x_i \in \mathbb{R}^d$, given to you after centering, i.e.,

$$\frac{1}{N}\sum_{i=1}^{N} x_i = 0.$$

We want to project this data onto a single unit vector u by learning an appropriate u. The projection of a data point x_i onto the unit vector u is given by:

$$\text{proj}_u(x_i) = (x_i^T u)u.$$

The residual error for each projection can be defined as the squared distance between the original data point and its projection:

$$\text{Residual}(x_i) = \|x_i - \text{proj}_u(x_i)\|^2 = \|x_i - (x_i^T u)u\|^2.$$

To minimize the residual, we compute:

$$\text{Residual}(x_i) = \|x_i - (x_i^T u)u\|^2 = \|x_i\|^2 - 2(x_i^T u)(u^T x_i) + (x_i^T u)^2\|u\|^2.$$

Since u is a unit vector, $\|u\|^2 = 1$:

$$\text{Residual}(x_i) = \|x_i\|^2 - 2(x_i^T u)(u^T x_i) + (x_i^T u)^2.$$

Now, summing over all data points and averaging gives:

$$\text{Total Residual} = \frac{1}{N}\sum_{i=1}^{N}\text{Residual}(x_i) = \frac{1}{N}\sum_{i=1}^{N}\left(\|x_i\|^2 - 2(x_i^T u)(u^T x_i) + (x_i^T u)^2\right).$$

We can denote the first term as $C = \frac{1}{N}\sum_{i=1}^{N}\|x_i\|^2$, which is a constant with respect to u. Therefore, minimizing the total residual simplifies to maximizing the term:

$$\frac{1}{N}\sum_{i=1}^{N}(x_i^T u)^2.$$

Next, we can express the variance of the projections $z_i = x_i^T u$:

$$\text{Var}(z) = \frac{1}{N}\sum_{i=1}^{N} z_i^2 - \left(\frac{1}{N}\sum_{i=1}^{N} z_i\right)^2.$$

Since the data $\{x_i\}$ is centered, we have:

$$\frac{1}{N}\sum_{i=1}^{N} z_i = 0.$$

Thus, the variance simplifies to:

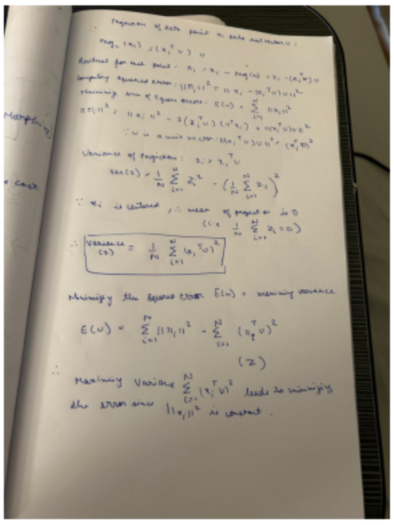
$$\text{Var}(z) = \frac{1}{N}\sum_{i=1}^{N}(x_i^T u)^2.$$

This shows that minimizing the residual of the projections is equivalent to maximizing the variance of the projections.

\textbf{Conclusion:} Minimizing the squared error between the projected data and the original data is equivalent to maximizing the variance of the projections. ta is equivalent to maximizing the variance of the projections. hus, the variance simpl

ng the residual of the projections.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Load the image
img1 = mpimg.imread('AML_HW3_2.jpeg')

# Create a figure to display the image
plt.figure(figsize=(10, 5))

# Display the image
plt.imshow(img1)
plt.axis('off')  # Turn off axis if you don't want to display it

# Show the figure
plt.show()
```



Projection of data point x_i onto unit vector u:

$\text{Proj}_u(x_i) = (x_i^T u)\, u$

Residual for each point: $r_i = x_i - \text{Proj}(x_i) = x_i - (x_i^T u)u$

Computing squared error: $\|r_i\|^2 = \|x_i - (x_i^T u)u\|^2$

Minimizing sum of squared errors: $E(u) = \sum_{i=1}^{N}\|r_i\|^2$

$\|r_i\|^2 = \|x_i\|^2 - 2(x_i^T u)(u^T x_i) + \|(x_i^T u)u\|^2$

$\because u$ is a unit vector: $\|(x_i^T u)u\|^2 = (x_i^T u)^2$

Variance of projection: $z_i = x_i^T u$

$\text{Var}(z) = \frac{1}{N}\sum_{i=1}^{N} z_i^2 - \left(\frac{1}{N}\sum_{i=1}^{N} z_i\right)^2$

$\because x_i$ is centered, $\therefore$ mean of projection is 0 (i.e. $\frac{1}{N}\sum_{i=1}^{N} z_i = 0$)

$\therefore \boxed{\text{Variance}(z) = \frac{1}{N}\sum_{i=1}^{N}(x_i^T u)^2}$

Minimizing the squared error $E(u)$ = maximizing variance

$E(u) = \sum_{i=1}^{N}\|x_i\|^2 - \sum_{i=1}^{N}(x_i^T u)^2$

(2)

$\therefore$ Maximizing variance $\sum_{i=1}^{N}(x_i^T u)^2$ leads to minimizing the error since $\|x_i\|^2$ is constant.

Q4. (25 points) - Principal Component Analysis

In this problem we will be applying PCA and T-SNE on the Superconductivity Dataset. More details on the dataset is present here. The goal here is to predict the critical temperature of a superconductor based on the features extracted.

First use Principal Component Analysis (PCA) to solve this problem.

- **Part 1. (5 points)** Perform the following steps to prepare the dataset:
  - Load the dataset from the "Q4data.csv" file provided as a dataframe df.
  - Select the '**critical_temp**' column as the target column and the rest of the columns from the dataframe df as X.
  - Split the dataset into train and test set with 35% data in test set and random_state = 42
  - Perform Standard Scaling on the dataset. Remember that when we have training and testing data, we fit preprocessing parameters on training data and apply them to all testing data. You should scale only the features (independent variables), not the target variable y.

    `Note: X should have 81 features.`

```
# Only use this code block if you are using Google Colab.
# If you are using Jupyter Notebook, please ignore this code block. You can directly upload the file to your Jupyter Noteboc
from google.colab import files

## It will prompt you to select a local file. Click on “Choose Files” then select and upload the file.
## Wait for the file to be 100% uploaded. You should see the name of the file once Colab has uploaded it.
uploaded = files.upload()
```

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SequentialFeatureSelector
import os, sys, re
import time
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler,StandardScaler
import matplotlib.pyplot as plt
df = pd.read_csv("Q4data.csv")
```

```
y = df["critical_temp"]
X = df.drop(columns=["critical_temp"])

X_train, X_test, Y_train, Y_test = train_test_split(X, y, test_size=0.35, random_state=42)

scalar = StandardScaler()

### START CODE ###
### Scale the dataset
#TODO
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
### END CODE ###
```

- **Part 2a (3 points)** Use PCA and reduce the dimension of X_train to the following number of components: `[3,20,40,60,81]`. For each of the five datasets, print the cumulative variance explained by the principal components `N = [3,20,40,60,81]`.(i.e. what percentage of variance in the original dataset is explained if we transform the dataset to have 3,20,40,60 and 81 principal components respectively).

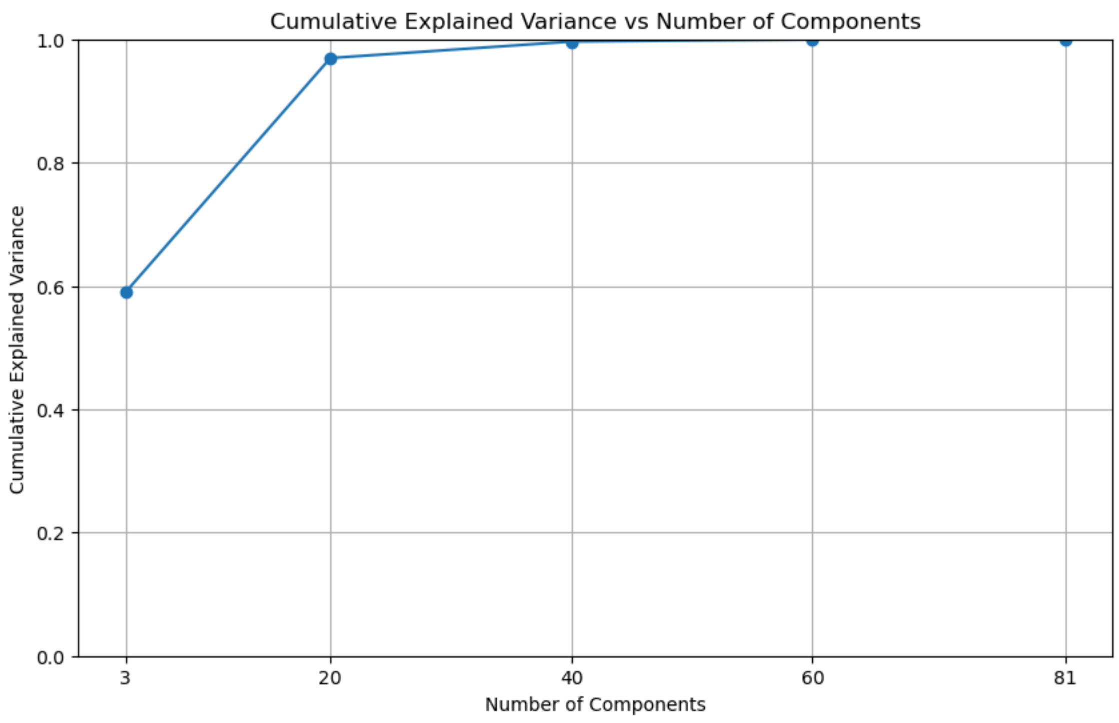
  `Note : PCA should be fit on X_train and the components thus learnt should be later used to transform X_test`

```
from sklearn.decomposition import PCA
nums = [3,20,40,60,81]
res = []
for num in nums:
    ### START CODE ###
    ## Fit PCA
    pca = PCA(n_components=num)  # Initialize PCA with specified number of components
    pca_fitter = pca.fit(X_train_scaled)  # Fit PCA on the scaled training data
    ### END CODE ###

    ### START CODE ###
    ## Transform Data
    X_train_new = pca.transform(X_train_scaled)  # Transform the training data
    X_test_new = pca.transform(X_test_scaled)    # Transform the test data
    ### END CODE ###

    ### START CODE ###
    ## Compute explained variance
    var = pca.explained_variance_ratio_  # Get the explained variance ratio
    cumulative_variance = var.cumsum()    # Calculate the cumulative variance explained
    ### END CODE ###

    print("Cumulative variance explained by {} components is {}".format(num, cumulative_variance[num-1])) #cumulative sum of
```

```
Cumulative variance explained by 3 components is 0.5894367932307177
Cumulative variance explained by 20 components is 0.9694250473503306
Cumulative variance explained by 40 components is 0.9961464440859475
Cumulative variance explained by 60 components is 0.9995333334358952
Cumulative variance explained by 81 components is 1.0000000000000007
```

- **Part 2b. (2 points)** Plot the cumulative variance explained by the principal components using the training data. The plot should display the number of components on X-axis and the cumulative explained variance on the y-axis. What do you understand from the plot obtained?

```
### START CODE ###
## Plot the explained variance vs number of components
# Initialize a list to store cumulative variance for each number of components
cumulative_variance_list = []

# Iterate over the number of components
for num in nums:
    # Fit PCA
    pca = PCA(n_components=num)
    pca_fitter = pca.fit(X_train_scaled)

    # Compute explained variance
    var = pca.explained_variance_ratio_
    cumulative_variance = var.cumsum()

    # Store the cumulative variance for the last component
    cumulative_variance_list.append(cumulative_variance[-1])

# Plot the explained variance vs number of components
plt.figure(figsize=(10, 6))
plt.plot(nums, cumulative_variance_list, marker='o')
plt.title('Cumulative Explained Variance vs Number of Components')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.xticks(nums)  # Set x-ticks to the specified numbers of components
plt.grid()
plt.ylim(0, 1)  # Set y-axis limits from 0 to 1
plt.show()
### END CODE ###

plt.show()
```



- **Part 3. (5 points)** For each of the reduced dataset, obtained in part 2.2, fit a linear regression model on the train data and show how adjusted R^2 varies as a function of # of components.(There will be a total of 5 R^2 score).

```
from sklearn.decomposition import PCA
nums = [3,20,40,60,81]
res = []
for num in nums:

    ### START CODE ###
    ## Fit PCA components
    pca = PCA(n_components=num)
    pca_fitter = pca.fit(X_train_scaled)

    ### END CODE ###

    ### START CODE ###
    ## Transform train and test data
    X_train_new = pca.transform(X_train_scaled)
    X_test_new = pca.transform(X_test_scaled)
    ### END CODE ###

    ### START CODE ###
    ## Compute explained variance
    percent_variance = pca.explained_variance_ratio_  # Get the explained variance ratio for each component
    var = percent_variance.cumsum()  # Calculate cumulative explained variance
    ### END CODE ###

    ### START CODE ###
    ## Fit LR and compute R-square and adjusted R-squared
    lr = LinearRegression()  # Initialize the linear regression model
    lr.fit(X_train_new, Y_train)  # Fit the model on the transformed training data
    Y_pred = lr.predict(X_test_new)  # Predict on the transformed test data
    r_squared = r2_score(Y_test, Y_pred)  # Calculate the R^2 score
    print("R^2 score {} with num_components {}".format(r_squared,num)) #hide
    ### END CODE ###
    adjusted_r_squared = 1 - (1-r_squared)*(len(Y_test)-1)/(len(Y_test)-X_test_new.shape[1]-1)
    print("Adjusted R^2",adjusted_r_squared)
```

```
R^2 score 0.492955970306844 with num_components 3
Adjusted R^2 0.4927514895850965
R^2 score 0.6250445765520567 with num_components 20
Adjusted R^2 0.6240341873754252
R^2 score 0.689952130179291 with num_components 40
Adjusted R^2 0.6882766485806923
R^2 score 0.7178625549139335 with num_components 60
Adjusted R^2 0.715569376005079
R^2 score 0.7307154603807391 with num_components 81
Adjusted R^2 0.7277522695494444
```

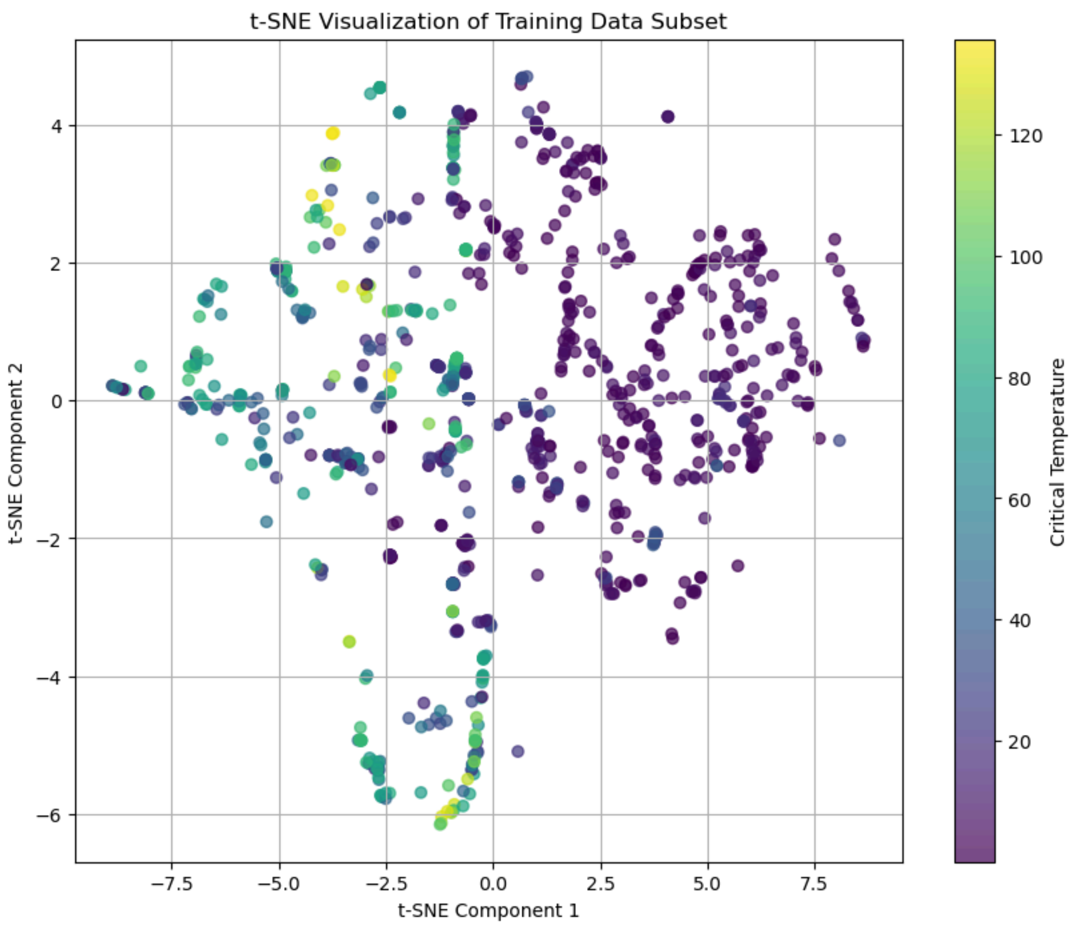
- **Part 4. T-SNE (3 points)** Now apply T-SNE to the dataset given above. You are required to carry out the following tasks:

1. Initialize a t-SNE model with number of dimensions = 3, perplexity = 300, number of iterations = 300 and random state = 42
2. Apply the t-SNE model to the training dataset

```
from sklearn.manifold import TSNE

### START CODE ###
## Initialize t-SNE model
tsne = TSNE(n_components=3, perplexity=300, n_iter = 300, random_state=42)  # Use max_iter instead of n_iter
### END CODE ###

### START CODE ###
## Fit and transform the data
X_tsne = tsne.fit_transform(X_train_scaled)  # Apply t-SNE to the scaled training data
### END CODE ###
```

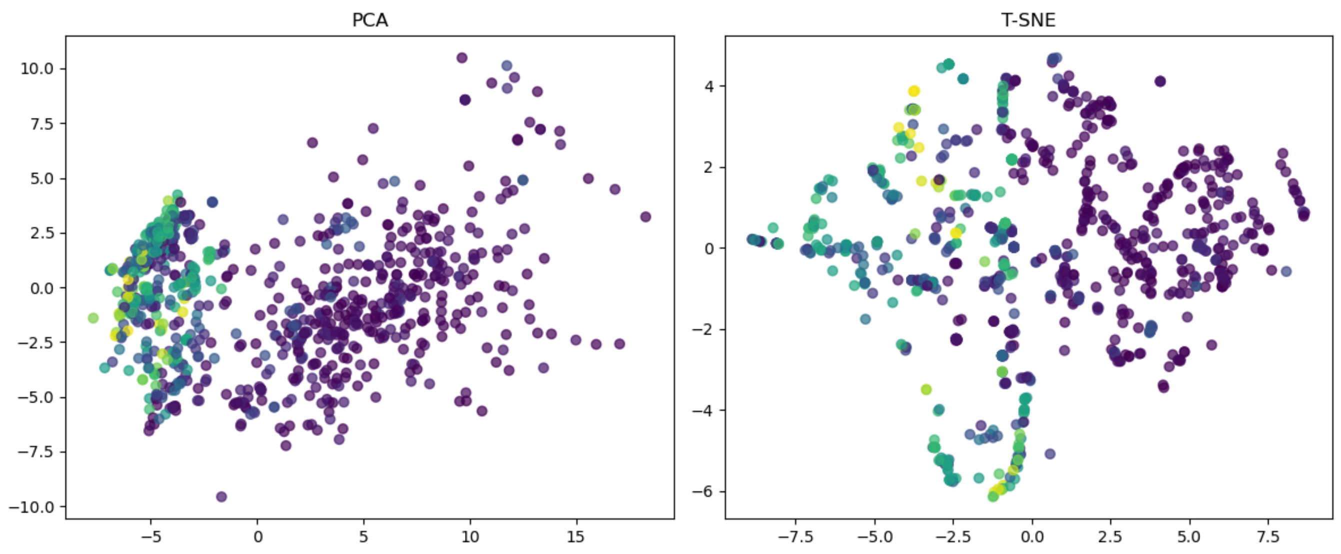
- **Part 5.** (3 points) For this part use a small subset of 1000 samples of the training dataset and plot these data points w.r.t any two t-SNE components, and color of each point depicting the y-value

```
### START CODE
n_samples = 1000

# indices = 1000

if X_tsne.shape[0] > n_samples:
    indices = np.random.choice(X_tsne.shape[0], n_samples, replace=False)
else:
    indices = np.arange(X_tsne.shape[0])  # In case there are fewer than 1000 samples

X_tsne_subset = X_tsne[indices]  # Subset of t-SNE components
y_subset = Y_train.values[indices]  # Corresponding y-values for the subset

### START CODE ###
# Plotting the t-SNE results
plt.figure(figsize=(10, 8))
scatter = plt.scatter(X_tsne_subset[:, 0], X_tsne_subset[:, 1], c=y_subset, cmap='viridis', alpha=0.7)
plt.colorbar(scatter, label='Critical Temperature')  # Color bar to indicate the y-value
plt.title('t-SNE Visualization of Training Data Subset')
plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.grid()
plt.show()
### END CODE
```



- **Part 6. (4 points)** Now we will plot the PCA and t-SNE projections of the data and compare the plots side-by-side to see the difference in scatters created by the two methods. You can use first 1000 data points for this.

```
X_subset = X_train_scaled[indices]

plt.figure(figsize=(12, 5))  # Adjust the figure size as needed

# First subplot (left)

### START CODE ###
### Obtain components from PCA and plot
pca = PCA(n_components=3)  # Initialize PCA with 2 components
pca_fitter = pca.fit(X_subset)  # Fit PCA on the subset of scaled training data
x_pca = pca.transform(X_subset)  # Transform the data to obtain PCA components

### END CODE ###

plt.subplot(1, 2, 1)  # 1 row, 2 columns, select the first subplot
plt.scatter(x_pca[:, 0], x_pca[:, 1], c=y_subset, cmap='viridis', alpha=0.7)  # Scatter plot for PCA
plt.title('PCA')

# Second subplot (right)
plt.subplot(1, 2, 2)  # 1 row, 2 columns, select the second subplot

### START CODE ###
### scatter plot for t-SNE
plt.scatter(X_tsne[indices, 0], X_tsne[indices, 1], c=y_subset, cmap='viridis', alpha=0.7)  # Scatter plot for t-SNE
### END CODE ###

plt.title('T-SNE')

# Adjust spacing between subplots
plt.tight_layout()

# Show the plots
plt.show()
```



Q5. (0 points) Outlier detection using PyOD - UNGRADED

Outlier detection, or anomaly detection is usually an unsupervised learning task where the objective is to identify suspicious observations in data. It has been widely used in military surveillance for enemy activities to prevent attacks, intrusion detection in cyber security, fraud detection for credit cards, etc.

PyOD is a comprehensive and scalable Python library for detecting outlying objects in multivariate data. PyOD includes more than 30 detection algorithms and provides unified APIs which makes it quite handy to use. In this question, you will play with PyOD, explore two different outlier detection algorithms and compare their performances.

First let's install PyOD. Please run the below code cell.

```
# install pyod using pip first
!pip install pyod
```

You can load the data stored in 'Q5_train_dataset.csv' and 'Q5_test_dataset.csv' using the following codes.

NOTE

- Only use the following code block if you are using Google Colab. If you are using Jupyter Notebook, please ignore this code block. You can directly upload the file to your Jupyter Notebook file systems.
- It will prompt you to select a local file. Click on “Choose Files” then select and upload the file. Wait for the file to be 100% uploaded. You should see the name of the file once Colab has uploaded it.

```
# Only use this code block if you are using Google Colab.
# If you are using Jupyter Notebook, please ignore this code block. You can directly upload the file to your Jupyter Notebook
from google.colab import files

## It will prompt you to select a local file. Click on “Choose Files” then select and upload the file.
## Wait for the file to be 100% uploaded. You should see the name of the file once Colab has uploaded it.
uploaded = files.upload()
```

```
import numpy as np
import random
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Load data
train_df = pd.read_csv('Q5_train_dataset.csv')
test_df = pd.read_csv('Q5_test_dataset.csv')

X_train = train_df[['X_train_0', 'X_train_1', 'X_train_2', 'X_train_3', 'X_train_4']].to_numpy()
y_train = train_df[['y_train']].to_numpy()
X_test = test_df[['X_test_0', 'X_test_1', 'X_test_2', 'X_test_3', 'X_test_4']].to_numpy()
y_test = test_df[['y_test']].to_numpy()
```

`X_train` and `X_test` contain the features, with the dimension of 5. `y_train` and `y_test` store the outlier labels, 0 means normal data, 1 means outlier data.

- **Part 1 (0 pts)**
  - Perform Standard Scaling on the dataset. Remember that when we have training and testing data, we fit preprocessing parameters on training data and apply them to all testing data. You should scale only the features (independent variables), not the target variable y.
  - Fit a k Nearest Neighbors (kNN) outlier detection model to `X_train` using `pyod.models.knn.KNN`. You can use this page as a reference.
  - Use the fitted model to predict the outlier labels of `X_test`. Compute the raw outlier scores on `X_test` using `decision_function()`.
  - Run PyOD's `evaluate_print()` function using the test set ground truth outlier labels and the raw outlier scores predicted by the model, to compute the ROC and Precision@n results.

```
# Scale the dataset
### START CODE ###
scalar = #TODO
X_train = #TODO
X_test = #TODO
### END CODE ###
```

```
from pyod.models.knn import KNN

# Initialize kNN model and fit it to X_train
### START CODE ###
clf = #TODO # initialize KNN class using the default parameters
### END CODE ###
```