Q1. **(9 Marks)** Principal Component Analysis

Download the `PADL-Q1.csv` file from the Assessment section of the PADL VLE site. The file contains 200 rows of numerical data for five variables, x1 to x5, as listed in the first row of the file.

(a) **[3 marks]** Apply principal component analysis (PCA) with a number of principal components (PCs) equal to the number of original variables, i.e., p = 5, and study the result in order to decide whether you can reduce the dimensionality of the dataset with little to no information loss. Report the minimum number of dimensions `Dmin` the new representation will require and briefly explain your choice.

First, we load the data:

```
import pandas as pd

# Loading data
data_q1 = pd.read_csv('PADL-Q1.csv')
```

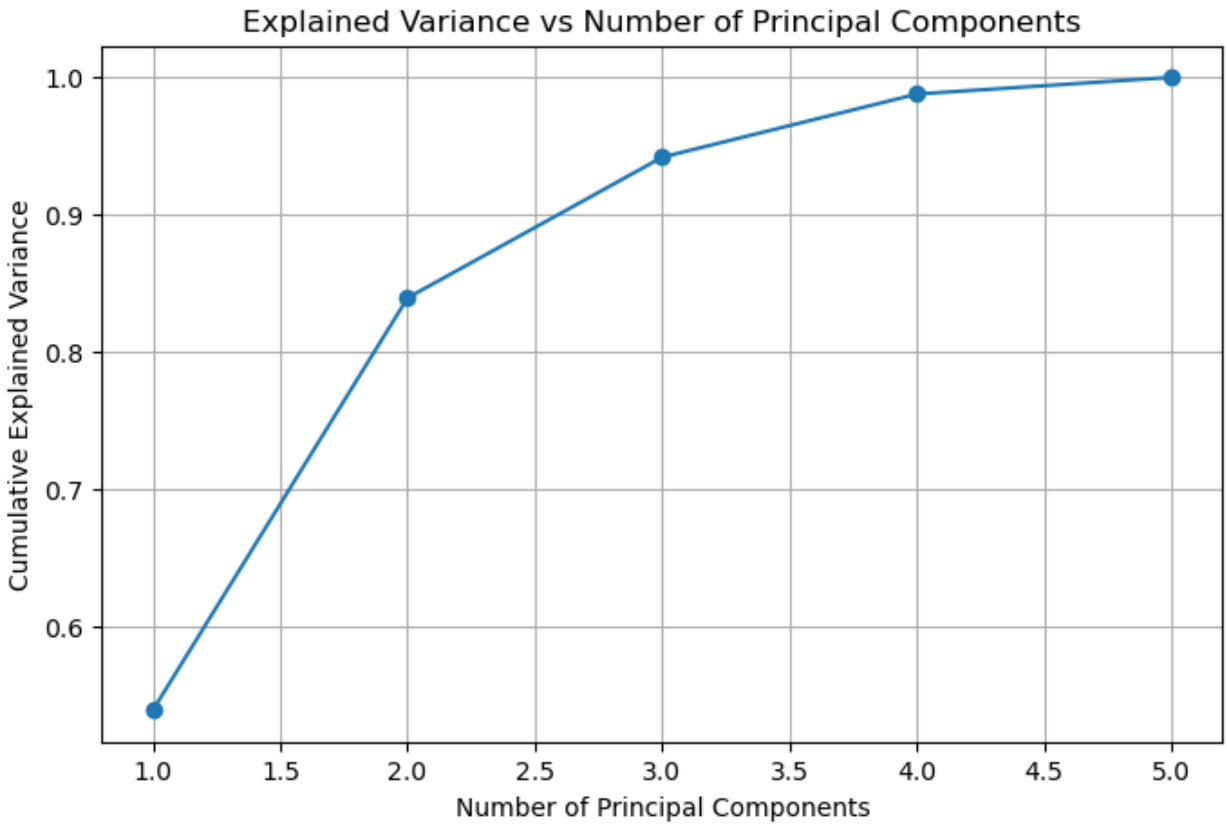
Then we apply principal component analysis (PCA) with a number of principal components (PCs) equal to the number of original variables (5):

```
from sklearn.decomposition import PCA

# Performing PCA
pca = PCA(n_components=5)
pca.fit(data_q1)

# Explained variance ratio
explained_variance = pca.explained_variance_ratio_
cumulative_variance = explained_variance.cumsum()
```

Finally, we plot the cumulative explained variance against the number of Principal Components, and print the explained variance to prepare for analysis:

```
import matplotlib.pyplot as plt

# Plot cumulative variance
plt.figure(figsize=(8, 5))
plt.plot(range(1, 6), cumulative_variance, marker='o')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Explained Variance vs Number of Principal Components')
plt.grid(True)
plt.show()

# Print explained variance and cumulative variance
print("Explained Variance for each Principal Component:")
for i, ev in enumerate(explained_variance, start=1):
```

```
    print(f"PC{i}: {ev:.4f}")

print("\nCumulative Explained Variance:")
for i, cv in enumerate(cumulative_variance, start=1):
    print(f"After PC{i}: {cv:.4f}")
```




```
Explained Variance for each Principal Component:
PC1: 0.5385
PC2: 0.3007
PC3: 0.1026
PC4: 0.0461
PC5: 0.0122

Cumulative Explained Variance:
After PC1: 0.5385
After PC2: 0.8391
After PC3: 0.9418
After PC4: 0.9878
After PC5: 1.0000
```

Analysis and Determination of $Dmin$

The cumulative explained variance shows that the first two principal components (PC1 and PC2) together explain approximately 83.91% of the total variance. Adding PC3 brings it up to 94.18%.

To minimize information loss, a common threshold is to retain components that explain at least 95% (+/- 1) of the variance. In this case, we need at least the first three principal components to achieve this, making $Dmin = 3$

(b) **[6 marks]** Repeat the PCA analysis using `Dmin` number of dimensions (principal components) and show the equations used to compute each principal component from the original variables x1, ..., x5. List these equations in decreasing order of the variance of the principal component they define.

First, we perform PCA on the data again, but using Dmin=3 as the number of components:

```
# Performing PCA with Dmin Components
pca_min = PCA(n_components=3)
principal_components = pca_min.fit_transform(data_q1)
```

Then, we get the PCA components in order to create the equations for each Principal Component and print them out in decreasing order of variance explained:

```
# Geting the PCA components (equations)
components = pca_min.components_

# Create equations for each principal component
equations = []
for i, component in enumerate(components):
    equation = f"PC{i+1} = " + " + ".join([f"({coeff:.2f} * x{j+1})"
for j, coeff in enumerate(component)])
    equations.append(equation)

# Print the equations
print("Equations for each Principal Component in Decreasing Order of
Variance Explained:")
for equation in equations:
    print(equation)
```

```
Equations for each Principal Component in Decreasing Order of Variance
Explained:
PC1 = (0.09 * x1) + (-0.03 * x2) + (0.35 * x3) + (-0.15 * x4) + (0.92
* x5)
PC2 = (-0.01 * x1) + (0.30 * x2) + (0.11 * x3) + (0.94 * x4) + (0.11 *
x5)
PC3 = (0.11 * x1) + (-0.13 * x2) + (0.91 * x3) + (-0.02 * x4) + (-0.37
* x5)
```

Marking guidance For each of the parts: one third of the marks for working code, another third for good, informed design choices, and one third for explaining it well.

Q2. **(27 marks)** Regression Models

Download the file PADL-Q2-train.csv from the Assessment section of the PADL VLE site. The comma-separated file contains data on four variables, x , y, z, w, and out. The label for each column is given in the first, header row of the file. The remaining rows contain 80 data points:

x	y	z	w	out
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...

Your ultimate goal here is to train and submit a single regression model whose performance will be tested by the exam markers on unseen data not available to you. Throughout this question you need to use scikit-learn – no marks will be given for the use of PyTorch! Regression should always be evaluated in terms of the R2 value on out-of-sample data.

(a) **[9 marks]** You need to demonstrate that you have considered and evaluated a suitable range of basis functions, alongside the possible use of data scaling and/or normalisation.

First, the data is loaded from the CSV file and split into training and testing sets:

```
# Load required libraries
import pandas as pd
from sklearn.model_selection import train_test_split

# Load the data
data = pd.read_csv('PADL-Q2-train.csv')

# Separate features and target
X = data[['x', 'y', 'z', 'w']]
y = data['out']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

Then, we scale the features using StandardScaler, which will help standardize the range of the features:

```
from sklearn.preprocessing import StandardScaler

# Apply scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Display scaled data
print("Scaled Training Data:\n", X_train_scaled[:5])
print("Scaled Testing Data:\n", X_test_scaled[:5])
```

```
Scaled Training Data:
 [[-0.53666347 -0.42556545 -0.09092072  1.06458129]
 [-1.23301285 -1.54628805 -1.53403699 -0.93933644]
 [-1.10704208  0.24201299  1.24624968 -0.93933644]
 [ 1.275974    0.89897087  1.66193756  1.06458129]
```

```
 [ 1.49064492 -0.60864378  1.1167212  -0.93933644]]
Scaled Testing Data:
 [[ 0.63474177 -0.50597831  0.19101492 -0.93933644]
 [ 0.49624228 -1.77538884  1.1012341   1.06458129]
 [ 0.04088661  1.46894125  0.80838709 -0.93933644]
 [-0.25105019  0.33810383  0.31279985 -0.93933644]
 [-0.29576837 -0.31076219  1.24132197 -0.93933644]]
```

Next, we create polynomial features to capture more complex relationships in the data:

```
from sklearn.preprocessing import PolynomialFeatures

# Apply polynomial features
poly = PolynomialFeatures(degree=2)
X_train_poly = poly.fit_transform(X_train_scaled)
X_test_poly = poly.transform(X_test_scaled)
```

We then train a linear regression model on both the scaled features and the polynomial features, and evaluate their performance using the R2 score:

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

# Train a linear regression model on scaled data
model_scaled = LinearRegression()
model_scaled.fit(X_train_scaled, y_train)

# Train a linear regression model on polynomial features
model_poly = LinearRegression()
model_poly.fit(X_train_poly, y_train)

# Predict and evaluate on scaled data
y_pred_scaled = model_scaled.predict(X_test_scaled)
r2_scaled = r2_score(y_test, y_pred_scaled)

# Predict and evaluate on polynomial features
y_pred_poly = model_poly.predict(X_test_poly)
r2_poly = r2_score(y_test, y_pred_poly)

print(f"R2 score for scaled features: {r2_scaled}")
print(f"R2 score for polynomial features (degree 2): {r2_poly}")

R2 score for scaled features: -0.48025325984432077
R2 score for polynomial features (degree 2): -0.0769266542933742
```

(b) **[9 marks]** You need to demonstrate that you have considered, tuned and evaluated a suitable range of linear regression models with respect to the possible use of regularisation and piecewise regression.

First, we perform Ridge and Lasso regression, which are forms of linear regression that include regularization terms to prevent overfitting. Then both methods are evaluated using cross-validation to find the optimal regularization strength.

```
# Ridge and Lasso Regression
from sklearn.linear_model import RidgeCV, LassoCV

# Ridge regression with cross-validation
ridge = RidgeCV(alphas=[0.1, 1.0, 10.0], cv=5)
ridge.fit(X_train_scaled, y_train)
ridge_r2 = r2_score(y_test, ridge.predict(X_test_scaled))

# Lasso regression with cross-validation
lasso = LassoCV(alphas=[0.1, 1.0, 10.0], cv=5)
lasso.fit(X_train_scaled, y_train)
lasso_r2 = r2_score(y_test, lasso.predict(X_test_scaled))
```

Next, we evaluate a simple linear regression model on the scaled features to serve as a baseline for comparison.

```
# Linear regression
linear_reg = LinearRegression()
linear_reg.fit(X_train_scaled, y_train)
linear_reg_r2 = r2_score(y_test, linear_reg.predict(X_test_scaled))
```

Additionally, we evaluate a linear regression model on polynomial features (degree 2) to capture more complex relationships in the data.

```
# Linear regression with polynomial features
linear_reg_poly = LinearRegression()
linear_reg_poly.fit(X_train_poly, y_train)
linear_reg_poly_r2 = r2_score(y_test,
linear_reg_poly.predict(X_test_poly))
```

Finally, we print the R2 scores for all the models to compare their performance.

```
print(f"R2 score for Ridge regression: {ridge_r2}")
print(f"R2 score for Lasso regression: {lasso_r2}")
print(f"R2 score for Linear regression: {linear_reg_r2}")
print(f"R2 score for Linear regression w polynomial features (degree
2): {linear_reg_poly_r2}")
```

```
R2 score for Ridge regression: -0.44766716852335064
R2 score for Lasso regression: -0.6592145820991382
R2 score for Linear regression: -0.48025325984432077
R2 score for Linear regression w polynomial features (degree 2): -
0.0769266542933742
```

(c) **[9 marks]** You need to implement an appropriate automated procedure that will train all of the above models and select the model expected to perform best on unseen data of the same kind as your training data. Include a code tile at the end of this section of your Jupyter notebook that attempts to test your final choice of model on a data set stored in a file `PADL-Q2-unseen.csv` and compute R2 for it. The file will have exactly the same format as file `PADL-Q2-train.csv`, including the header, but possibly a different overall number of rows. This means you can use a renamed copy of `PADL-Q2-train.csv` to debug that part of your code, and to produce the corresponding content for your PDF file (in order to demonstrate that this part of the code is in working order).

First, we load the dataset and split it into training and testing sets.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

# Load the dataset
df = pd.read_csv('PADL-Q2-train.csv')

# Define the features (X) and the target (y)
X = df[['x', 'y', 'z', 'w']]
y = df['out']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

Next, we define the models and parameters for Linear Regression, Ridge Regression using CV, and Lasso Regression using CV.

```
from sklearn.linear_model import RidgeCV, LassoCV, LinearRegression

# Define models and parameters
models = {
    'Linear Regression': LinearRegression(),
    'Ridge Regression using CV': RidgeCV(alphas=np.logspace(-6, 6,
13), cv=5),
    'Lasso Regression using CV': LassoCV(alphas=np.logspace(-6, 6,
13), cv=5)
}
```

We define the pipeline for each model, including scaling and the respective regression model.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# Define the pipeline with different polynomial degrees
pipelines = {}
for name, model in models.items():
```

```
    for degree in [1, 2]:  # Trying polynomial degrees 1 and 2
        pipelines[f'{name} with poly degree {degree}'] = Pipeline([
            ('poly', PolynomialFeatures(degree=degree)),
            ('scaler', StandardScaler()),
            ('regressor', model)
        ])
```

We train each model using cross-validation, print the R2 scores for each, and select the best model.

```
from sklearn.metrics import r2_score
import warnings

# Suppress convergence warnings
warnings.filterwarnings(action='ignore', category=UserWarning)

# Train and evaluate models
best_model = None
best_score = -np.inf

for name, pipeline in pipelines.items():
    pipeline.fit(X_train, y_train)
    if 'Ridge' in name or 'Lasso' in name:
        print(f'Complexity parameter for {name}:
{pipeline.named_steps["regressor"].alpha_}')
    score = pipeline.score(X_test, y_test)
    print(f'R2 score on the test set for {name}: {score}\n')
    if score > best_score:
        best_score = score
        best_model = pipeline

print(f"\nBest model selected: \n{best_model}")
print(f"R2 score on the test set: {best_score}")
```

```
R2 score on the test set for Linear Regression with poly degree 1: -
0.48025325984432077

R2 score on the test set for Linear Regression with poly degree 2: -
0.07692665429337442

Complexity parameter for Ridge Regression using CV with poly degree 1:
10.0
R2 score on the test set for Ridge Regression using CV with poly
degree 1: -0.44766716852335064

Complexity parameter for Ridge Regression using CV with poly degree 2:
100.0
R2 score on the test set for Ridge Regression using CV with poly
degree 2: -0.5346755323292165
```

```
Complexity parameter for Lasso Regression using CV with poly degree 1:
0.001
R2 score on the test set for Lasso Regression using CV with poly
degree 1: -0.47728130464229523

Complexity parameter for Lasso Regression using CV with poly degree 2:
0.01
R2 score on the test set for Lasso Regression using CV with poly
degree 2: -0.48038886885184273


Best model selected:
Pipeline(steps=[('poly', PolynomialFeatures()), ('scaler',
StandardScaler()),
                ('regressor', LinearRegression())])
R2 score on the test set: -0.07692665429337442
```

```python
# Load the unseen data (use a renamed copy of PADL-Q2-train.csv for
debugging)
unseen_data = pd.read_csv('PADL-Q2-unseen.csv')
X_unseen = unseen_data[['x', 'y', 'z', 'w']]
y_unseen = unseen_data['out']

# Make predictions
y_unseen_pred = best_model.predict(X_unseen)

unseen_r2 = r2_score(y_unseen, y_unseen_pred)

print(f"R2 score on the unseen data: {unseen_r2}")
```

```
R2 score on the unseen data: 0.47528013871527297
```

RUN THE FOLLOWING BLOCK OF CODE TO ATTEMPT ON UNSEEN DATA

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.linear_model import RidgeCV, LassoCV, LinearRegression
from sklearn.metrics import r2_score
import warnings

# Load the dataset
df = pd.read_csv('PADL-Q2-train.csv')

# Define the features (X) and the target (y)
X = df[['x', 'y', 'z', 'w']]
y = df['out']
```

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Define models and parameters
models = {
    'Linear Regression': LinearRegression(),
    'Ridge Regression using CV': RidgeCV(alphas=np.logspace(-6, 6,
13), cv=5),
    'Lasso Regression using CV': LassoCV(alphas=np.logspace(-6, 6,
13), cv=5, max_iter=100000)
}

# Containers for polynomial features and scalers
poly_features = {}
scalers = {}

# Containers for transformed data
X_train_poly = {}
X_test_poly = {}
X_train_scaled = {}
X_test_scaled = {}

# Apply polynomial features and scaling
degrees = [1, 2]
for degree in degrees:
    poly = PolynomialFeatures(degree=degree)
    X_train_poly[degree] = poly.fit_transform(X_train)
    X_test_poly[degree] = poly.transform(X_test)

    scaler = StandardScaler()
    X_train_scaled[degree] =
scaler.fit_transform(X_train_poly[degree])
    X_test_scaled[degree] = scaler.transform(X_test_poly[degree])

    poly_features[degree] = poly
    scalers[degree] = scaler

# Debug prints for transformed shapes
for degree in degrees:
    print(f"Degree {degree}:")
    print(f"  X_train_poly shape: {X_train_poly[degree].shape}")
    print(f"  X_test_poly shape: {X_test_poly[degree].shape}")
    print(f"  X_train_scaled shape: {X_train_scaled[degree].shape}")
    print(f"  X_test_scaled shape: {X_test_scaled[degree].shape}")

# Train and evaluate models
model_scores = {}
for degree in degrees:
    for name, model in models.items():
```

```
        model.fit(X_train_scaled[degree], y_train)
        score = model.score(X_test_scaled[degree], y_test)
        model_scores[(name, degree)] = (model, score)

# Select the best model
best_model = None
best_score = -np.inf
best_degree = None
best_name = None
for (name, degree), (model, score) in model_scores.items():
    print(f'R2 score on the test set for {name} with poly degree
{degree}: {score}')
    if score > best_score:
        best_score = score
        best_model = model
        best_degree = degree
        best_name = name

print(f"\nBest model selected: {best_name} with poly degree
{best_degree}")
print(f"R2 score on the test set: {best_score}")

# Load the unseen data
unseen_data = pd.read_csv('PADL-Q2-unseen.csv')
X_unseen = unseen_data[['x', 'y', 'z', 'w']]
y_unseen = unseen_data['out']

# Apply the best polynomial transformation and scaling to the unseen
data
poly = PolynomialFeatures(degree=best_degree)
X_unseen_poly = poly.fit_transform(X_unseen)

scaler = StandardScaler()
X_unseen_scaled = scaler.fit_transform(X_unseen_poly)

# Debug prints for unseen data
print(f"X_unseen shape: {X_unseen.shape}")
print(f"X_unseen_poly shape: {X_unseen_poly.shape}")
print(f"X_unseen_scaled shape: {X_unseen_scaled.shape}")

# Make predictions
y_unseen_pred = best_model.predict(X_unseen_scaled)

# Evaluate predictions
unseen_r2 = r2_score(y_unseen, y_unseen_pred)
print(f"R2 score on the unseen data: {unseen_r2}")
```

```
Degree 1:
  X_train_poly shape: (64, 5)
  X_test_poly shape: (16, 5)
```

```
  X_train_scaled shape: (64, 5)
  X_test_scaled shape: (16, 5)
Degree 2:
  X_train_poly shape: (64, 15)
  X_test_poly shape: (16, 15)
  X_train_scaled shape: (64, 15)
  X_test_scaled shape: (16, 15)
R2 score on the test set for Linear Regression with poly degree 1: -
0.48025325984432077
R2 score on the test set for Ridge Regression using CV with poly
degree 1: -0.44766716852335064
R2 score on the test set for Lasso Regression using CV with poly
degree 1: -0.47728130464229523
R2 score on the test set for Linear Regression with poly degree 2: -
0.07692665429337442
R2 score on the test set for Ridge Regression using CV with poly
degree 2: -0.5346755323292165
R2 score on the test set for Lasso Regression using CV with poly
degree 2: -0.48038886885184273

Best model selected: Linear Regression with poly degree 2
R2 score on the test set: -0.07692665429337442
X_unseen shape: (80, 4)
X_unseen_poly shape: (80, 15)
X_unseen_scaled shape: (80, 15)
R2 score on the unseen data: 0.4566762010378016
```

Breakdown of marks For each of the parts: one third of the marks for working code, another third for good, informed design choices, and one third for explaining it well.

Q3. **(14 marks)** Embeddings

Cockney rhyming slang is a social and linguistic phenomenon where, in order to disguise the topic of a conversation from prying ears, one word is replaced with another word or a phrase. One of the possible patterns used is as follows: to disguise a given word X, one finds a pair of semantically related words Y and Z, such that the latter word Z rhymes with the word X. Then in the conversation, the word X is replaced by either the entire phrase 'Y and Z' or by Y alone. For instance, replacing 'stairs' with 'apples and pears' or just 'apples' would give sentences, such as: "I went up the apples." Similarly, "He is on the dog and bone" (on the phone).

Download the plain text of Sir Arthur Conan Doyle's book 'Adventures of Sherlock Holmes' from this URL: Adventures of Sherlock Holmes. Your aim will be to design a fully automated procedure that will make use of this text to ultimately generate candidate rhyming slang phrases of the above mentioned 'A and B' pattern for each word in the following list L = ['gold', 'diamond', 'robbery', 'bank', 'police']. To achieve that, take the following steps:

(a) **[3 marks]** Design and implement a procedure that takes the text of the entire book and extracts all triplets of words where the word in the middle is 'and'.

First, we read the text from the URL:

```
# Step 0: Download the text file from the URL using wget
file_url = "https://www.gutenberg.org/cache/epub/48320/pg48320.txt"
file_path = "pg48320.txt"

!wget -O {file_path} {file_url}

# Step 1: Read the text from the file
with open(file_path, 'r') as file:
    text = file.read()
```

```
--2024-05-26 09:11:40--
https://www.gutenberg.org/cache/epub/48320/pg48320.txt
Resolving www.gutenberg.org (www.gutenberg.org)... 152.19.134.47,
2610:28:3090:3000:0:bad:cafe:47
Connecting to www.gutenberg.org (www.gutenberg.org)|
152.19.134.47|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 622286 (608K) [text/plain]
Saving to: ‘pg48320.txt’

pg48320.txt         100%[===================>] 607.70K  1.09MB/s    in
0.5s

2024-05-26 09:11:41 (1.09 MB/s) - ‘pg48320.txt’ saved [622286/622286]
```

We then convert it all to lowercase and remove punctuation so we'll only have words / letters remaining:

```
import re

# Step 2: Convert to lowercase and remove punctuation
words = re.findall(r'\b[a-zA-Z]+\b', text.lower())
# print(len(words))
```

Now that the text is processed, we search for all 'and' instances within the text, take the word before and after, then add it to a list:

```
# Step 3: Extract triplets
triplets = []
for i in range(1, len(words) - 1):
    if words[i] == 'and':
        triplets.append((words[i-1], words[i], words[i+1]))

print(f"Total triplets found: {len(triplets)}")
```

```
Total triplets found: 3149
```

We now have a list of all triplets of words where the word in the middle is 'and'. Here's a preview of the first 10 triplets:

```
print(triplets[:10])
```

```
[('states', 'and', 'most'), ('cost', 'and', 'with'), ('shut', 'and',
'locked'), ('bye', 'and', 'be'), ('eclipses', 'and', 'predominates'),
('emotions', 'and', 'that'), ('reasoning', 'and', 'observing'),
('gibe', 'and', 'a'), ('motives', 'and', 'actions'), ('delicate',
'and', 'finely')]
```

(b) **[3 marks]** Design and implement a procedure that takes a word W from the list L and selects all triplets T_i produced in the previous step, such that the last of the three words in the triplet, and the word W share a suffix of length at least 3, e.g. 'old' – 'gold', 'bold' – 'gold', 'snobbery' – 'robbery'.

First, we make 2 functions; one to check wether two words share a suffix of length 3 minimum, and another to filter triplets based on the suffix condition:

```
# Function to check if two words share a suffix of length at least 3
def has_common_suffix(word1, word2, min_length=3):
    for i in range(min_length, len(word2) + 1):
        if word1.endswith(word2[-i:]):
            #print(f"Matching suffix found: '{word2[-i:]}' in words
'{word1}' and '{word2}'")
            return True
    return False

# Function to check if two words share a suffix of length at least 2
(for the word 'diamond')
def has_common_suffix_dia(word1, word2, min_length=2):
    for i in range(min_length, len(word2) + 1):
        if word1.endswith(word2[-i:]):
            #print(f"Matching suffix found: '{word2[-i:]}' in words
'{word1}' and '{word2}'")
            return True
    return False

# Function to filter triplets based on the suffix condition
def filter_triplets(triplets, target_words):
    filtered_triplets = []
    for triplet in triplets:
        for word in target_words:
            if word == 'diamond':
                if has_common_suffix_dia(triplet[2], word):
                    print(f"Triplet '{triplet}' matches with word
'{word}'")
                    filtered_triplets.append(triplet)
                    break
            if has_common_suffix(triplet[2], word):
                print(f"Triplet '{triplet}' matches with word
'{word}'")
                filtered_triplets.append(triplet)
```

```
                break
    return filtered_triplets
```

Then, we run that function on all words in the provided list and the 3rd word in the triplets we found earlier and save to a list.

```
# List L
L = ['gold', 'diamond', 'robbery', 'bank', 'police']

# Apply the filter function to the triplets and list L
filtered_triplets = filter_triplets(triplets, L)

# Display the filtered triplets
print(f'\nTotal filtered triplets: {len(filtered_triplets)}')
print(filtered_triplets[:10])
```

```
Triplet '('delicacy', 'and', 'every')' matches with word 'robbery'
Triplet '('street', 'and', 'found')' matches with word 'diamond'
Triplet '('year', 'and', 'found')' matches with word 'diamond'
Triplet '('body', 'and', 'mind')' matches with word 'diamond'
Triplet '('pay', 'and', 'very')' matches with word 'robbery'
Triplet '('ten', 'and', 'every')' matches with word 'robbery'
Triplet '('abbots', 'and', 'archery')' matches with word 'robbery'
Triplet '('assistant', 'and', 'found')' matches with word 'diamond'
Triplet '('police', 'and', 'every')' matches with word 'robbery'
Triplet '('good', 'and', 'kind')' matches with word 'diamond'
Triplet '('air', 'and', 'scenery')' matches with word 'robbery'
Triplet '('away', 'and', 'told')' matches with word 'gold'
Triplet '('forward', 'and', 'found')' matches with word 'diamond'
Triplet '('cry', 'and', 'found')' matches with word 'diamond'
Triplet '('yet', 'and', 'and')' matches with word 'diamond'
Triplet '('day', 'and', 'send')' matches with word 'diamond'
Triplet '('dock', 'and', 'found')' matches with word 'diamond'
Triplet '('long', 'and', 'very')' matches with word 'robbery'
Triplet '('chair', 'and', 'cheery')' matches with word 'robbery'
Triplet '('packet', 'and', 'found')' matches with word 'diamond'
Triplet '('right', 'and', 'found')' matches with word 'diamond'
Triplet '('yard', 'and', 'behind')' matches with word 'diamond'
Triplet '('help', 'and', 'advice')' matches with word 'police'
Triplet '('round', 'and', 'round')' matches with word 'diamond'
Triplet '('rope', 'and', 'land')' matches with word 'diamond'
Triplet '('safe', 'and', 'sound')' matches with word 'diamond'
Triplet '('room', 'and', 'found')' matches with word 'diamond'
Triplet '('police', 'and', 'very')' matches with word 'robbery'
Triplet '('day', 'and', 'very')' matches with word 'robbery'
Triplet '('do', 'and', 'frank')' matches with word 'bank'
Triplet '('mercifully', 'and', 'thank')' matches with word 'bank'
Triplet '('up', 'and', 'hand')' matches with word 'diamond'
Triplet '('rapidly', 'and', 'told')' matches with word 'gold'
```

```
Triplet '('up', 'and', 'found')' matches with word 'diamond'
Triplet '('lodgings', 'and', 'found')' matches with word 'diamond'
Triplet '('night', 'and', 'find')' matches with word 'diamond'
Triplet '('passage', 'and', 'found')' matches with word 'diamond'
Triplet '('out', 'and', 'round')' matches with word 'diamond'
Triplet '('silk', 'and', 'gold')' matches with word 'gold'
Triplet '('indemnify', 'and', 'hold')' matches with word 'gold'
Triplet '('sections', 'and', 'and')' matches with word 'diamond'

Total filtered triplets: 41
[('delicacy', 'and', 'every'), ('street', 'and', 'found'), ('year',
'and', 'found'), ('body', 'and', 'mind'), ('pay', 'and', 'very'),
('ten', 'and', 'every'), ('abbots', 'and', 'archery'), ('assistant',
'and', 'found'), ('police', 'and', 'every'), ('good', 'and', 'kind')]
```

The resulting list is all triplets produced in the previous step, such that the last of the three words in the triplet, and the words in L share a suffix of length at least 3.

(c) **[8 marks]** For each word W and corresponding list of triplets [T_1, T_2, ...] selected in the previous step, use word2vec to compute the semantic similarity between the first and third word in the triplet, and sort the triplets in order of decreasing similarity, then return the top 5 triplets. You can use a pretrained word2vec model or you can train it yourself from scratch.

First, we load a pre-trained Word2Vec model:

```
from gensim.models import Word2Vec
from gensim import downloader as api

''' for training the model from scratch (not as efficient as the pre-
trained model)
# Prepare sentences for training Word2Vec model
sentences = [words[i:i+5] for i in range(len(words) - 5)]

# Train Word2Vec model
w2v_model = Word2Vec(sentences, vector_size=100, window=5,
min_count=1, workers=4)

# Save the model
w2v_model.save("q3_w2v.model")
'''

w2v = api.load("glove-wiki-gigaword-50")
```

Then we compute the semantic similarity between the first and third word in each triplet:

```
from scipy.spatial.distance import cosine

# Initialize a dictionary to store the top triplets for each word in L
top_triplets_for_L = {word: [] for word in L}
```

```
# Function to compute cosine similarity between two words using the
Word2Vec model
def compute_similarity(word1, word2, model):
    # Check if both words exist in the Word2Vec model's vocabulary
    if word1 in model.w2v and word2 in model.w2v:
        # Compute cosine similarity and return it
        return 1 - cosine(model.w2v[word1], model.w2v[word2])
    else:
        # If either word is not in the vocabulary, return a similarity
of 0
        return 0

# Iterate over each triplet in the filtered triplets
for triplet in filtered_triplets:
    # Iterate over each word in the list L
    for word in L:
        if word == 'diamond':
            if has_common_suffix_dia(triplet[2], word):
                # Compute the semantic similarity between the first
and third words in the triplet
                similarity = w2v.similarity(triplet[0], triplet[2])
                # Append the triplet and its similarity score to the
list for the current word in L
                top_triplets_for_L[word].append((triplet, similarity))

        # Check if the third word in the triplet has a common suffix
with the current word in L
        if has_common_suffix(triplet[2], word):
            # Compute the semantic similarity between the first and
third words in the triplet
            #similarity = compute_similarity(triplet[0], triplet[2],
w2v_model)
            similarity = w2v.similarity(triplet[0], triplet[2])
            # Append the triplet and its similarity score to the list
for the current word in L
            top_triplets_for_L[word].append((triplet, similarity))

# Sort each list of triplets by similarity in descending order and
keep only the top 5 triplets
for word in top_triplets_for_L:
    top_triplets_for_L[word] = sorted(top_triplets_for_L[word],
key=lambda x: x[1], reverse=True)[:5]

# Function to display the top triplets for each word in L in an
organized way
def display_top_triplets(top_triplets):
    # Iterate over each word and its list of top triplets
    for word, triplets in top_triplets.items():
        print(f"Top triplets for '{word}':")
        # Iterate over each triplet and its similarity score
```

```
        for triplet, similarity in triplets:
            print(f"Triplet: {triplet} - Similarity:
{similarity:.4f}")
        print()

# Call the function to display the results
display_top_triplets(top_triplets_for_L)
```

```
Top triplets for 'gold':
Triplet: ('silk', 'and', 'gold') - Similarity: 0.5162
Triplet: ('away', 'and', 'told') - Similarity: 0.4344
Triplet: ('rapidly', 'and', 'told') - Similarity: 0.1344
Triplet: ('indemnify', 'and', 'hold') - Similarity: -0.1362

Top triplets for 'diamond':
Triplet: ('round', 'and', 'round') - Similarity: 1.0000
Triplet: ('good', 'and', 'kind') - Similarity: 0.8917
Triplet: ('up', 'and', 'hand') - Similarity: 0.8242
Triplet: ('yet', 'and', 'and') - Similarity: 0.7186
Triplet: ('out', 'and', 'round') - Similarity: 0.6340

Top triplets for 'robbery':
Triplet: ('long', 'and', 'very') - Similarity: 0.7726
Triplet: ('ten', 'and', 'every') - Similarity: 0.6716
Triplet: ('day', 'and', 'very') - Similarity: 0.6441
Triplet: ('police', 'and', 'every') - Similarity: 0.4976
Triplet: ('pay', 'and', 'very') - Similarity: 0.4755

Top triplets for 'bank':
Triplet: ('do', 'and', 'frank') - Similarity: 0.3390
Triplet: ('mercifully', 'and', 'thank') - Similarity: 0.0573

Top triplets for 'police':
Triplet: ('help', 'and', 'advice') - Similarity: 0.6427
```

NOTE: the word 'diamond' had no matching suffix with length = 3 and therefore had no matching triplets. So it looks for matching suffixes with length = 2 instead.

Marking guidance For each of the parts: one third of the marks (rounded up) for working code, another third (rounded up) for good, informed design choices, and one third (rounded down) for explaining it well.

Q4. **(10 marks)** Basic MLPs

Consider a multilayer perceptron (MLP) with two inputs and one output, `f : R^2 → R`, that seeks to approximate multiplication of two real numbers, i.e., `f(x, y) ≈ x · y`.

(a) **[4 marks]** Using only linear (fully connected) and ReLU layers, implement an MLP for this task in PyTorch.

First, we implement a MLP with two inputs, two hidden layers with ReLU activations, and one output:

```
import torch
import torch.nn as nn
import torch.optim as optim

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(2, 128)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(128, 64)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(64, 1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        return x

# Initialize the model
model = MLP()
print(model)
```

```
MLP(
  (fc1): Linear(in_features=2, out_features=128, bias=True)
  (relu1): ReLU()
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (relu2): ReLU()
  (fc3): Linear(in_features=64, out_features=1, bias=True)
)
```

(b) **[3 marks]** Create an appropriate training loop to train the network on random data using a loss function of your choice. You should explicitly comment on what assumptions you are making about the range of the training data.

Then, we create a training loop to train the network on random data within the range -100 to 100 and prints the loss.

The range contains both negative and positive numbers, and provides 200 diffent numbers to train on, this should make the network generalize better on numbers outside its training range.

If the range were too small, it would result in a really low loss within the training range, but result in high losses outside that range.

```python
# Generating random training data
def generate_data(n_samples, range_min, range_max):
    x = torch.FloatTensor(n_samples, 2).uniform_(range_min, range_max)
    y = (x[:, 0] * x[:, 1]).view(-1, 1)
    return x, y

# Training loop
def train(model, criterion, optimizer, n_epochs=20000, n_samples=1000,
range_min=-100, range_max=100):
    model.train()
    for epoch in range(n_epochs):
        optimizer.zero_grad()
        x_train, y_train = generate_data(n_samples, range_min,
range_max)
        outputs = model(x_train)
        loss = criterion(outputs, y_train)
        loss.backward()
        optimizer.step()
        if (epoch+1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{n_epochs}], Loss:
{loss.item():.4f}')

# Loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the model
train(model, criterion, optimizer)
```

```
Epoch [100/20000], Loss: 7156821.0000
Epoch [200/20000], Loss: 639096.0625
Epoch [300/20000], Loss: 438366.8125
Epoch [400/20000], Loss: 404129.3438
Epoch [500/20000], Loss: 452223.7500
Epoch [600/20000], Loss: 460500.0625
Epoch [700/20000], Loss: 431043.4375
Epoch [800/20000], Loss: 452684.2812
Epoch [900/20000], Loss: 471160.4688
Epoch [1000/20000], Loss: 431193.0625
Epoch [1100/20000], Loss: 447197.1875
Epoch [1200/20000], Loss: 425870.9375
Epoch [1300/20000], Loss: 460596.0938
Epoch [1400/20000], Loss: 447500.9688
Epoch [1500/20000], Loss: 432251.6562
Epoch [1600/20000], Loss: 439442.3125
Epoch [1700/20000], Loss: 462014.7812
Epoch [1800/20000], Loss: 415280.5000
Epoch [1900/20000], Loss: 459586.1875
Epoch [2000/20000], Loss: 475219.9375
Epoch [2100/20000], Loss: 444056.4375
```

```
Epoch [2200/20000], Loss: 428450.2500
Epoch [2300/20000], Loss: 447239.4062
Epoch [2400/20000], Loss: 442061.4062
Epoch [2500/20000], Loss: 446100.7812
Epoch [2600/20000], Loss: 401601.6250
Epoch [2700/20000], Loss: 411560.8750
Epoch [2800/20000], Loss: 391147.7812
Epoch [2900/20000], Loss: 396324.0938
Epoch [3000/20000], Loss: 399991.0625
Epoch [3100/20000], Loss: 374376.0938
Epoch [3200/20000], Loss: 345308.3750
Epoch [3300/20000], Loss: 329884.0312
Epoch [3400/20000], Loss: 353048.2812
Epoch [3500/20000], Loss: 306807.7500
Epoch [3600/20000], Loss: 285469.2500
Epoch [3700/20000], Loss: 252159.5312
Epoch [3800/20000], Loss: 237458.0156
Epoch [3900/20000], Loss: 227865.1250
Epoch [4000/20000], Loss: 202900.7812
Epoch [4100/20000], Loss: 199581.6250
Epoch [4200/20000], Loss: 186900.0625
Epoch [4300/20000], Loss: 157446.4688
Epoch [4400/20000], Loss: 138844.7344
Epoch [4500/20000], Loss: 127012.0625
Epoch [4600/20000], Loss: 123292.5938
Epoch [4700/20000], Loss: 117435.5703
Epoch [4800/20000], Loss: 93580.6172
Epoch [4900/20000], Loss: 89785.2734
Epoch [5000/20000], Loss: 77059.4141
Epoch [5100/20000], Loss: 67843.2578
Epoch [5200/20000], Loss: 61710.7148
Epoch [5300/20000], Loss: 57258.9922
Epoch [5400/20000], Loss: 44439.5742
Epoch [5500/20000], Loss: 44399.9258
Epoch [5600/20000], Loss: 45099.0508
Epoch [5700/20000], Loss: 36442.0391
Epoch [5800/20000], Loss: 40747.8086
Epoch [5900/20000], Loss: 28733.9277
Epoch [6000/20000], Loss: 32271.1562
Epoch [6100/20000], Loss: 24970.5684
Epoch [6200/20000], Loss: 19860.5664
Epoch [6300/20000], Loss: 21910.7012
Epoch [6400/20000], Loss: 21191.2383
Epoch [6500/20000], Loss: 19438.2656
Epoch [6600/20000], Loss: 18464.2012
Epoch [6700/20000], Loss: 17370.9844
Epoch [6800/20000], Loss: 15825.1406
Epoch [6900/20000], Loss: 15090.9170
Epoch [7000/20000], Loss: 14189.2070
```

```
Epoch [7100/20000], Loss: 10716.5020
Epoch [7200/20000], Loss: 12155.6240
Epoch [7300/20000], Loss: 11410.3184
Epoch [7400/20000], Loss: 10041.0439
Epoch [7500/20000], Loss: 8745.4004
Epoch [7600/20000], Loss: 7293.2451
Epoch [7700/20000], Loss: 8233.2920
Epoch [7800/20000], Loss: 7068.0293
Epoch [7900/20000], Loss: 6753.8286
Epoch [8000/20000], Loss: 6092.5278
Epoch [8100/20000], Loss: 6299.7559
Epoch [8200/20000], Loss: 5364.7690
Epoch [8300/20000], Loss: 4647.4629
Epoch [8400/20000], Loss: 4703.3618
Epoch [8500/20000], Loss: 4829.2759
Epoch [8600/20000], Loss: 5069.5146
Epoch [8700/20000], Loss: 4750.1021
Epoch [8800/20000], Loss: 3528.2896
Epoch [8900/20000], Loss: 3104.3384
Epoch [9000/20000], Loss: 3247.5774
Epoch [9100/20000], Loss: 2770.2644
Epoch [9200/20000], Loss: 3486.0374
Epoch [9300/20000], Loss: 3038.2615
Epoch [9400/20000], Loss: 2683.2043
Epoch [9500/20000], Loss: 2197.9822
Epoch [9600/20000], Loss: 2906.9543
Epoch [9700/20000], Loss: 2583.4221
Epoch [9800/20000], Loss: 2046.1545
Epoch [9900/20000], Loss: 2057.4954
Epoch [10000/20000], Loss: 2340.8960
Epoch [10100/20000], Loss: 1731.9542
Epoch [10200/20000], Loss: 1810.4336
Epoch [10300/20000], Loss: 2098.4941
Epoch [10400/20000], Loss: 1496.6040
Epoch [10500/20000], Loss: 1534.4536
Epoch [10600/20000], Loss: 1573.9126
Epoch [10700/20000], Loss: 1428.0295
Epoch [10800/20000], Loss: 1422.1102
Epoch [10900/20000], Loss: 1296.4858
Epoch [11000/20000], Loss: 1533.7338
Epoch [11100/20000], Loss: 1339.2461
Epoch [11200/20000], Loss: 1371.2722
Epoch [11300/20000], Loss: 1197.8131
Epoch [11400/20000], Loss: 1081.0253
Epoch [11500/20000], Loss: 1360.3275
Epoch [11600/20000], Loss: 1544.1526
Epoch [11700/20000], Loss: 1202.8156
Epoch [11800/20000], Loss: 1192.8662
Epoch [11900/20000], Loss: 918.4973
```

```
Epoch [12000/20000], Loss: 803.5720
Epoch [12100/20000], Loss: 992.5989
Epoch [12200/20000], Loss: 886.8444
Epoch [12300/20000], Loss: 920.3054
Epoch [12400/20000], Loss: 1098.0723
Epoch [12500/20000], Loss: 938.2054
Epoch [12600/20000], Loss: 914.9908
Epoch [12700/20000], Loss: 887.6133
Epoch [12800/20000], Loss: 888.4598
Epoch [12900/20000], Loss: 842.8616
Epoch [13000/20000], Loss: 755.6763
Epoch [13100/20000], Loss: 771.9789
Epoch [13200/20000], Loss: 872.5573
Epoch [13300/20000], Loss: 939.0565
Epoch [13400/20000], Loss: 710.9609
Epoch [13500/20000], Loss: 791.3531
Epoch [13600/20000], Loss: 816.7366
Epoch [13700/20000], Loss: 548.7240
Epoch [13800/20000], Loss: 736.3877
Epoch [13900/20000], Loss: 786.9243
Epoch [14000/20000], Loss: 580.7491
Epoch [14100/20000], Loss: 636.1669
Epoch [14200/20000], Loss: 623.8946
Epoch [14300/20000], Loss: 572.1484
Epoch [14400/20000], Loss: 601.1545
Epoch [14500/20000], Loss: 727.5382
Epoch [14600/20000], Loss: 579.9039
Epoch [14700/20000], Loss: 522.7258
Epoch [14800/20000], Loss: 438.7659
Epoch [14900/20000], Loss: 597.9713
Epoch [15000/20000], Loss: 603.1180
Epoch [15100/20000], Loss: 490.5750
Epoch [15200/20000], Loss: 522.6911
Epoch [15300/20000], Loss: 419.6772
Epoch [15400/20000], Loss: 647.5373
Epoch [15500/20000], Loss: 617.6136
Epoch [15600/20000], Loss: 530.0731
Epoch [15700/20000], Loss: 453.1344
Epoch [15800/20000], Loss: 476.1889
Epoch [15900/20000], Loss: 623.5402
Epoch [16000/20000], Loss: 432.0012
Epoch [16100/20000], Loss: 389.2590
Epoch [16200/20000], Loss: 471.9448
Epoch [16300/20000], Loss: 523.2708
Epoch [16400/20000], Loss: 471.2187
Epoch [16500/20000], Loss: 537.0206
Epoch [16600/20000], Loss: 528.2941
Epoch [16700/20000], Loss: 498.3112
Epoch [16800/20000], Loss: 415.9265
```

```
Epoch [16900/20000], Loss: 359.2828
Epoch [17000/20000], Loss: 480.6928
Epoch [17100/20000], Loss: 410.0601
Epoch [17200/20000], Loss: 373.1820
Epoch [17300/20000], Loss: 401.1017
Epoch [17400/20000], Loss: 366.6898
Epoch [17500/20000], Loss: 410.8686
Epoch [17600/20000], Loss: 438.8560
Epoch [17700/20000], Loss: 359.7556
Epoch [17800/20000], Loss: 374.0771
Epoch [17900/20000], Loss: 493.5708
Epoch [18000/20000], Loss: 318.8110
Epoch [18100/20000], Loss: 326.9509
Epoch [18200/20000], Loss: 434.0184
Epoch [18300/20000], Loss: 383.8690
Epoch [18400/20000], Loss: 405.6537
Epoch [18500/20000], Loss: 376.2532
Epoch [18600/20000], Loss: 407.4207
Epoch [18700/20000], Loss: 304.2929
Epoch [18800/20000], Loss: 321.5668
Epoch [18900/20000], Loss: 297.4747
Epoch [19000/20000], Loss: 304.4978
Epoch [19100/20000], Loss: 383.9733
Epoch [19200/20000], Loss: 381.5843
Epoch [19300/20000], Loss: 398.5869
Epoch [19400/20000], Loss: 348.4291
Epoch [19500/20000], Loss: 268.3399
Epoch [19600/20000], Loss: 280.6915
Epoch [19700/20000], Loss: 432.6684
Epoch [19800/20000], Loss: 313.6793
Epoch [19900/20000], Loss: 280.0261
Epoch [20000/20000], Loss: 294.6757
```

(c) **[3 marks]** Evaluate the network in terms of the absolute error in its prediction, i.e., $|f(x, y) - (x \cdot y)|$. You should report a mean error over random samples within the range of the training data but also a generalization error when tested on inputs outside the range of the training data.

```python
# Function to evaluate the model
def evaluate(model, range_min, range_max, n_samples=1000):
    model.eval()
    with torch.no_grad():
        x_test, y_test = generate_data(n_samples, range_min,
range_max)
        predictions = model(x_test)
        errors = torch.abs(predictions - y_test)
        mean_error = torch.mean(errors).item()
    return mean_error
```

```python
# Evaluate on training data range
train_error = evaluate(model, -100, 100)
print(f'Mean Absolute Error on Training Data Range (-100 to 100):
{train_error:.4f}')

# Evaluate on the range -200 to -100
generalization_error_1 = evaluate(model, -200, -100)
print(f'Mean Absolute Error outside Training Data Range (-200 to -
100): {generalization_error_1:.4f}')

# Evaluate on the range 100 to 200
generalization_error_2 = evaluate(model, 100, 200)
print(f'Mean Absolute Error outside Training Data Range (100 to 200):
{generalization_error_2:.4f}')
```

```
Mean Absolute Error on Training Data Range (-100 to 100): 12.8182
Mean Absolute Error outside Training Data Range (-200 to -100):
5169.0864
Mean Absolute Error outside Training Data Range (100 to 200):
5404.0220
```

Q5. **(20 marks)** Telling the time

Consider the task of telling the time from an image of an analogue clock face. The objective is to estimate the hour, H (an integer in the range 0...11, so H ∈ {0, 1, ..., 11}) and the minute, M (an integer in the range 0...59, so M ∈ {0, 1, ..., 59}), from a given image. You are to tackle this as an end-to-end deep learning problem, i.e., train a network that takes an image as input and directly outputs the estimated time.

We have provided you with a training dataset of cartoon clocks (`clocks_dataset.zip` on the VLE). Once unzipped, you will find 10k training images in PNG format, named `0000.png` to `9999.png`. Each image is of size H = 448, W = 448 and they are RGB colour. Examples from the dataset are shown in Figure 1. The images are procedurally generated with the hour and minute drawn randomly from uniform distributions. Each image is accompanied by a text file containing the corresponding label, named `0000.txt` to `9999.txt`. Each text file contains a single line with the ground truth time in the format `HH:MM`.

Figure 1: Example images from the training set. Note the wide variation in clock shapes, colours, text and hand appearance. Clocks may or may not have second and/or alarm hands.

The ground truth labels associated with each clock are `(H = 0, M = 0)`, `(H = 3, M = 23)`, `(H = 10, M = 24)`, `(H = 1, M = 52)` and `(H = 2, M = 1)` respectively.

Your task in this question is to implement and train a network in PyTorch that outputs the time given an image. You may tackle this problem as either classification or regression. Your network will be evaluated on an unseen test set. The performance metric will be the absolute difference in minutes between the estimated and ground truth time. For example, if the ground truth is 11:59 and you predict 00:01, then the error will be 2 minutes. The test images will be generated by exactly the same procedure as the training images and the overall performance will be measured by the median of the absolute differences over the whole test set.

Your goal is for this median error to be as close to zero as possible. You do not have access to the test set, but you need to include a python script `predict_time.py` that I can import which contains a function `predict(images)` as explained in the assessment section of the VLE. This function should load your pretrained network from the weights you supply, pass the input images through the network and return times. The input images is a B × 3 × 448 × 448 PyTorch tensor containing a batch of B images, each with an RGB image of size 448 × 448 - i.e. the same format as the training data. Intensity values will be in the range (0, 1). Any preprocessing or normalisation that you need to apply to the images must be inside the `predict` function. The output `times` is a B × 2 tensor, where `times[:,0]` contains the estimated hours for each image (an integer between 0 and 11) and `times[:,1]` contains the estimated minutes for each image (an integer between 0 and 59).

Your notebook must include your training and validation code along with discussion and justification for all design decisions. You are not allowed to use transfer learning on a pretrained network (i.e. you must train your network from scratch) and your saved network weights must not exceed 20MiB.

(a) **[5 marks]** Create a dataloader for the clocks dataset. You are free to preprocess or augment the images in any way you deem suitable but should explain your decisions in comments or text blocks.

First, I created a dataloader for the clocks dataset by defining a custom ClockDataset class that reads images and corresponding labels. The labels are assumed to be stored in text files with the same base name as the images but with a .txt extension. The dataset applies transformations to the images and prepares batches using a dataloader. It resizes the pictures to 224x224 to use less computing power and to make the Convoluted Neural Network smaller in size.

```
import os
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image

# Define the transformations
# Transformations help in preprocessing the data and augmenting it for
better training
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
])
```

```
# Custom Dataset class for the clock images for loading the data
class ClockDataset(Dataset):
    def __init__(self, image_dir, indices, transform=None):
        self.image_dir = image_dir
        self.transform = transform
        self.image_files = sorted([f for f in os.listdir(image_dir) if
f.endswith('.png')])
        self.image_files = [self.image_files[i] for i in indices]

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

    # Load and preprocess the image and label
    def __getitem__(self, idx):
        image_file = self.image_files[idx]
        image_path = os.path.join(self.image_dir, image_file)
        label_file = image_file.replace('.png', '.txt')
        label_path = os.path.join(self.image_dir, label_file)

        image = Image.open(image_path).convert('RGB')
        with open(label_path, 'r') as f:
            label = f.readline().strip()
            hour, minute = map(int, label.split(':'))

        if self.transform:
            image = self.transform(image)

        return image, torch.tensor([hour, minute], dtype=torch.long)

# Create train and validation datasets
image_dir = 'train/'
indices = list(range(10000))
train_indices = indices[:8000]
val_indices = indices[8000:]

train_dataset = ClockDataset(image_dir=image_dir,
indices=train_indices, transform=transform)
val_dataset = ClockDataset(image_dir=image_dir, indices=val_indices,
transform=transform)

# Create dataloaders
batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True, num_workers=4)
val_loader = DataLoader(val_dataset, batch_size=batch_size,
shuffle=False, num_workers=4)

# Display the structure of a batch for verification
```

```
for images, labels in train_loader:
    print(images.shape, labels.shape)
    break
```

```
torch.Size([32, 3, 224, 224]) torch.Size([32, 2])
```

(b) **[4 marks]** Design and implement an appropriate network architecture.

Then, I designed a convolutional neural network (CNN) for predicting the time from clock images. This network uses multiple convolutional layers followed by fully connected layers.

```
import torch.nn as nn
import torch.nn.functional as F

class ClockNet(nn.Module):
    def __init__(self):
        super(ClockNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1,
padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1,
padding=1)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1,
padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.fc1 = nn.Linear(64 * 28 * 28, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 2)  # Output layer with 2 neurons for
hour and minute

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 64 * 28 * 28)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Check if CUDA is available and move model to GPU if possible
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = ClockNet().to(device)
print(model)

# Calculate the size in bytes
model_size_bytes = sum(p.numel() * p.element_size() for p in
model.parameters())

# Convert bytes to megabytes
model_size_mb = model_size_bytes / (1024 ** 2)
```

```
print(f"Model size: {model_size_mb:.2f} MB")

# Test the model with a random input to verify shapes
input_tensor = torch.randn(1, 3, 224, 224).to(device)
output_tensor = model(input_tensor)
print(f"Output shape: {output_tensor.shape}")
```

```
ClockNet(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (fc1): Linear(in_features=50176, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=32, bias=True)
  (fc3): Linear(in_features=32, out_features=2, bias=True)
)
Model size: 12.35 MB
Output shape: torch.Size([1, 2])
```

```
/home/aa-101/.local/lib/python3.10/site-packages/torch/nn/modules/
conv.py:456: UserWarning: Applied workaround for CuDNN issue, install
nvrtc.so (Triggered internally at
../aten/src/ATen/native/cudnn/Conv_v8.cpp:84.)
  return F.conv2d(input, weight, bias, self.stride,
```

(c) **[3 marks]** Choose and justify an appropriate loss function.

For predicting the hour and minute from images, we can treat this as a regression problem where we want to minimize the difference between the predicted and actual values. Mean Squared Error (MSE) is a common loss function for regression tasks because it penalizes larger errors more heavily. This will be suitable for our task since we want to minimize the difference in both hours and minutes.

```
import torch.optim as optim

# Define the loss function
criterion = nn.MSELoss()

# Define the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Justification:
# We use Mean Squared Error (MSE) loss because it is appropriate for
regression tasks where we want to minimize the difference between
predicted and actual continuous values.
# The Adam optimizer is chosen for its efficiency and adaptive
```

```
learning rate capabilities, which generally result in faster
convergence.
```

(d) **[3 marks]** Plot training and validation losses and use this to justify hyperparameter choices.

```
import matplotlib.pyplot as plt

num_epochs = 100
train_losses = []
val_losses = []

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs.float(), labels.float())
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    train_losses.append(running_loss / len(train_loader))

    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs.float(), labels.float())
            val_loss += loss.item()

    val_losses.append(val_loss / len(val_loader))

    print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {running_loss /
len(train_loader)}, Validation Loss: {val_loss / len(val_loader)}")


torch.save(model.state_dict(), 'weights.pkl')

# Plotting the training and validation losses
plt.figure(figsize=(10, 5))
plt.plot(range(1, num_epochs + 1), train_losses, label='Train Loss')
plt.plot(range(1, num_epochs + 1), val_losses, label='Validation
Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
```

```
plt.legend()
plt.title('Training and Validation Losses')
plt.show()
```

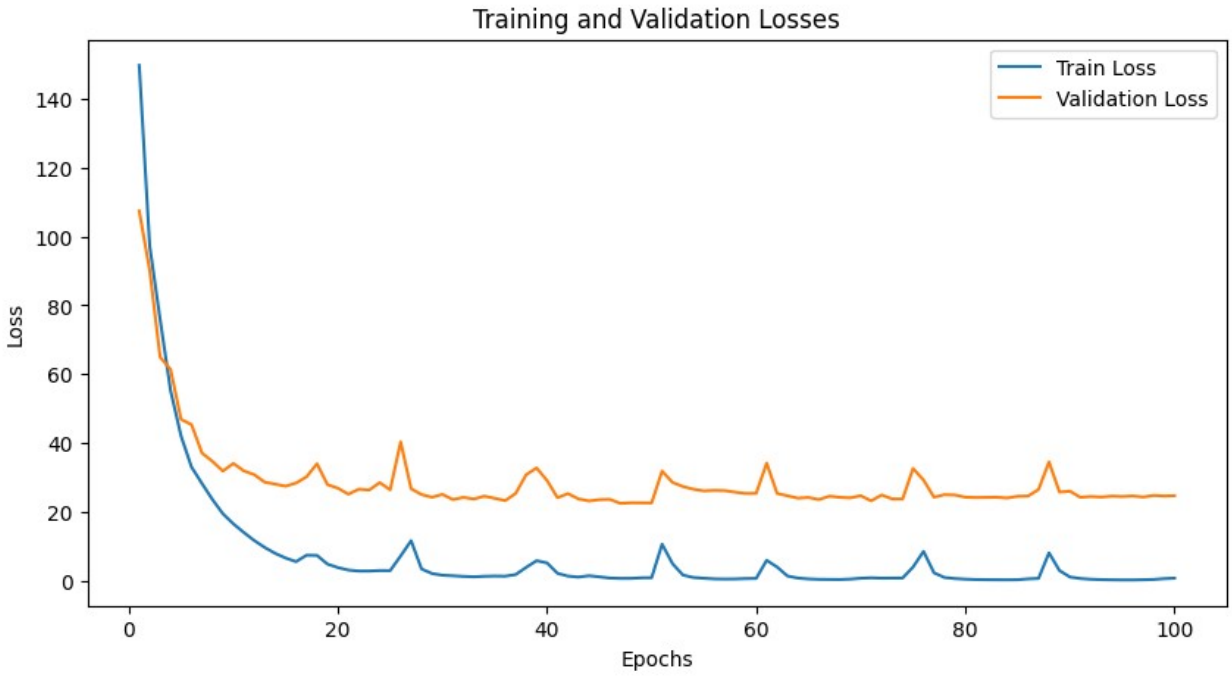
```
Epoch 1/100, Train Loss: 149.7893373260498, Validation Loss:
107.45615611000666
Epoch 2/100, Train Loss: 97.31934423828125, Validation Loss:
90.23136514330668
Epoch 3/100, Train Loss: 75.87244702148438, Validation Loss:
64.9432610405816
Epoch 4/100, Train Loss: 55.171560455322265, Validation Loss:
61.34505293104384
Epoch 5/100, Train Loss: 41.990273864746094, Validation Loss:
46.8299504537431
Epoch 6/100, Train Loss: 32.96635829925537, Validation Loss:
45.31429511781723
Epoch 7/100, Train Loss: 28.179439817428587, Validation Loss:
37.05034077356732
Epoch 8/100, Train Loss: 23.53338720703125, Validation Loss:
34.62596676841615
Epoch 9/100, Train Loss: 19.37849861907959, Validation Loss:
31.799542684403676
Epoch 10/100, Train Loss: 16.45397525215149, Validation Loss:
34.01136732858325
Epoch 11/100, Train Loss: 13.943631755828857, Validation Loss:
31.858750941261412
Epoch 12/100, Train Loss: 11.593285581588745, Validation Loss:
30.75009342980763
Epoch 13/100, Train Loss: 9.580223581314087, Validation Loss:
28.56842915217082
Epoch 14/100, Train Loss: 7.86133874130249, Validation Loss:
27.990291209447953
Epoch 15/100, Train Loss: 6.506401420593262, Validation Loss:
27.39997399042523
Epoch 16/100, Train Loss: 5.452574820041656, Validation Loss:
28.320991319323344
Epoch 17/100, Train Loss: 7.327305699348449, Validation Loss:
30.127800199720596
Epoch 18/100, Train Loss: 7.266927416801453, Validation Loss:
33.924737479951645
Epoch 19/100, Train Loss: 4.74253731584549, Validation Loss:
27.843892279125395
Epoch 20/100, Train Loss: 3.7306636571884155, Validation Loss:
26.810751786307684
Epoch 21/100, Train Loss: 3.0037265324592592, Validation Loss:
25.078763999636212
Epoch 22/100, Train Loss: 2.718762231826782, Validation Loss:
26.514758927481516
Epoch 23/100, Train Loss: 2.7261263875961306, Validation Loss:
26.30109034644233
```

```
Epoch 24/100, Train Loss: 2.8676485357284545, Validation Loss:
28.43619638019138
Epoch 25/100, Train Loss: 2.8404768776893614, Validation Loss:
26.340838553413512
Epoch 26/100, Train Loss: 7.106076881408692, Validation Loss:
40.30116844177246
Epoch 27/100, Train Loss: 11.545632331371307, Validation Loss:
26.648197537376767
Epoch 28/100, Train Loss: 3.312406532764435, Validation Loss:
24.9583714651683
Epoch 29/100, Train Loss: 1.995234478712082, Validation Loss:
24.189571002173047
Epoch 30/100, Train Loss: 1.5024596560001373, Validation Loss:
25.012673862396724
Epoch 31/100, Train Loss: 1.347879724264145, Validation Loss:
23.508651915050688
Epoch 32/100, Train Loss: 1.16609037733078, Validation Loss:
24.14693214022924
Epoch 33/100, Train Loss: 1.0544448882341384, Validation Loss:
23.671721208663214
Epoch 34/100, Train Loss: 1.1897140733003617, Validation Loss:
24.487287339710054
Epoch 35/100, Train Loss: 1.2627785435914993, Validation Loss:
23.871092334626212
Epoch 36/100, Train Loss: 1.2220017001628876, Validation Loss:
23.1902821177528
Epoch 37/100, Train Loss: 1.698657734632492, Validation Loss:
25.306322309705948
Epoch 38/100, Train Loss: 3.781745800495148, Validation Loss:
30.72271363697355
Epoch 39/100, Train Loss: 5.734650583744049, Validation Loss:
32.69827073717874
Epoch 40/100, Train Loss: 5.081531156539917, Validation Loss:
29.041191857958598
Epoch 41/100, Train Loss: 2.055908180236816, Validation Loss:
24.054180114988295
Epoch 42/100, Train Loss: 1.249484334230423, Validation Loss:
25.24041441508702
Epoch 43/100, Train Loss: 0.973251032114029, Validation Loss:
23.701960253337074
Epoch 44/100, Train Loss: 1.3564512799978257, Validation Loss:
23.109531886993892
Epoch 45/100, Train Loss: 1.0405248572826387, Validation Loss:
23.46982621389722
Epoch 46/100, Train Loss: 0.6748175356388092, Validation Loss:
23.563660538385786
Epoch 47/100, Train Loss: 0.5846573459506035, Validation Loss:
22.409260295686266
Epoch 48/100, Train Loss: 0.5846962831020355, Validation Loss:
```

```
22.566278707413446
Epoch 49/100, Train Loss: 0.7468712439537049, Validation Loss:
22.55852496434772
Epoch 50/100, Train Loss: 0.7890562885999679, Validation Loss:
22.52237783916413
Epoch 51/100, Train Loss: 10.525413354039193, Validation Loss:
31.796196120125906
Epoch 52/100, Train Loss: 4.857352998256683, Validation Loss:
28.488391240437824
Epoch 53/100, Train Loss: 1.5379269943237304, Validation Loss:
27.2908485881866
Epoch 54/100, Train Loss: 0.8459160294532776, Validation Loss:
26.507950646536692
Epoch 55/100, Train Loss: 0.6208776684999466, Validation Loss:
26.035799541170636
Epoch 56/100, Train Loss: 0.44212654465436935, Validation Loss:
26.178968982091025
Epoch 57/100, Train Loss: 0.39403230518102644, Validation Loss:
26.097412722451345
Epoch 58/100, Train Loss: 0.4226047292351723, Validation Loss:
25.647882681044322
Epoch 59/100, Train Loss: 0.5316963786482811, Validation Loss:
25.282459766145738
Epoch 60/100, Train Loss: 0.5833554196357728, Validation Loss:
25.32740406762986
Epoch 61/100, Train Loss: 5.8443961651325225, Validation Loss:
34.09944697788784
Epoch 62/100, Train Loss: 3.8795205101966856, Validation Loss:
25.274770373389835
Epoch 63/100, Train Loss: 1.263124227285385, Validation Loss:
24.599980081830704
Epoch 64/100, Train Loss: 0.6729106550216675, Validation Loss:
23.919832767002166
Epoch 65/100, Train Loss: 0.42519450205564496, Validation Loss:
24.113373858588083
Epoch 66/100, Train Loss: 0.3075543438196182, Validation Loss:
23.51084473776439
Epoch 67/100, Train Loss: 0.2762944698929787, Validation Loss:
24.485540442996555
Epoch 68/100, Train Loss: 0.25746663892269134, Validation Loss:
24.151377564384823
Epoch 69/100, Train Loss: 0.37477443850040437, Validation Loss:
24.010448629893954
Epoch 70/100, Train Loss: 0.6267831265330315, Validation Loss:
24.623607771737234
Epoch 71/100, Train Loss: 0.7459134390950203, Validation Loss:
23.137400884476918
Epoch 72/100, Train Loss: 0.6490462719202041, Validation Loss:
24.851614520663308
```

```
Epoch 73/100, Train Loss: 0.6701784001588822, Validation Loss:
23.688238552638463
Epoch 74/100, Train Loss: 0.6893286908864975, Validation Loss:
23.678537891024636
Epoch 75/100, Train Loss: 3.884453835606575, Validation Loss:
32.54002910190158
Epoch 76/100, Train Loss: 8.44109558916092, Validation Loss:
29.195872851780482
Epoch 77/100, Train Loss: 2.2168499011993408, Validation Loss:
24.20806520704239
Epoch 78/100, Train Loss: 0.8420802701711655, Validation Loss:
24.926504316784087
Epoch 79/100, Train Loss: 0.5382850283384323, Validation Loss:
24.83861181471083
Epoch 80/100, Train Loss: 0.3589651216864586, Validation Loss:
24.197251940530442
Epoch 81/100, Train Loss: 0.2454140758216381, Validation Loss:
24.09986608747452
Epoch 82/100, Train Loss: 0.20770412608981131, Validation Loss:
24.136979012262252
Epoch 83/100, Train Loss: 0.18300836169719695, Validation Loss:
24.178474456544908
Epoch 84/100, Train Loss: 0.17227292582392692, Validation Loss:
23.964053381057013
Epoch 85/100, Train Loss: 0.19559960147738456, Validation Loss:
24.463889205266558
Epoch 86/100, Train Loss: 0.44550782984495163, Validation Loss:
24.520212271856884
Epoch 87/100, Train Loss: 0.5781644005179405, Validation Loss:
26.434455152541872
Epoch 88/100, Train Loss: 8.02035721874237, Validation Loss:
34.45615411940075
Epoch 89/100, Train Loss: 2.8559438586235046, Validation Loss:
25.688793174804204
Epoch 90/100, Train Loss: 0.987567153453827, Validation Loss:
25.93989350303771
Epoch 91/100, Train Loss: 0.549012631893158, Validation Loss:
24.16193711568439
Epoch 92/100, Train Loss: 0.33218859320878985, Validation Loss:
24.391188515557182
Epoch 93/100, Train Loss: 0.220017636179924, Validation Loss:
24.236350301712278
Epoch 94/100, Train Loss: 0.1683308789730072, Validation Loss:
24.505999671088325
Epoch 95/100, Train Loss: 0.1343469721376896, Validation Loss:
24.397584339929004
Epoch 96/100, Train Loss: 0.1328686377108097, Validation Loss:
24.552717731112526
Epoch 97/100, Train Loss: 0.18841465392708778, Validation Loss:
```

```
24.25095244059487
Epoch 98/100, Train Loss: 0.26812483522295955, Validation Loss:
24.702809863620335
Epoch 99/100, Train Loss: 0.5056202661991119, Validation Loss:
24.539420271676683
Epoch 100/100, Train Loss: 0.6437234287858009, Validation Loss:
24.634067883567205
```



(e) **[5 marks]** Up to 5 marks are available for performance on the unseen test set. A median error <5 minutes will score 5 marks. Partial marks will be awarded for larger errors.

In case *predict_time.py* doesn't work and you still want to give me a chance, run the following code block. Be sure to update *test_image_dir* to the unseen data directory.

```
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
import numpy as np

# Define the ClockNet model class
class ClockNet(nn.Module):
    def __init__(self):
        super(ClockNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1,
```

```
padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1,
padding=1)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1,
padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.fc1 = nn.Linear(64 * 28 * 28, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 2)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 64 * 28 * 28)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Define the dataset class for loading images and labels
class ClockDataset(Dataset):
    def __init__(self, image_dir, transform=None):
        self.image_dir = image_dir
        self.transform = transform
        self.image_files = sorted([f for f in os.listdir(image_dir) if
f.endswith('.png')])

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

    def __getitem__(self, idx):
        image_file = self.image_files[idx]
        image_path = os.path.join(self.image_dir, image_file)
        label_file = image_file.replace('.png', '.txt') # CHOOSE THE
CORRECT IMAGE EXTENSION IF NEED BE (e.g., .jpg, .jpeg, etc.)
        label_path = os.path.join(self.image_dir, label_file)

        image = Image.open(image_path).convert('RGB')
        with open(label_path, 'r') as f:
            label = f.readline().strip()
            hour, minute = map(int, label.split(':'))

        if self.transform:
            image = self.transform(image)

        return image, torch.tensor([hour, minute],
dtype=torch.float32)

# Set device to GPU if available
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Load the model and weights
model = ClockNet().to(device)
model.load_state_dict(torch.load('weights.pkl', map_location=device))
model.eval()

# Define the transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
])

# Load the test data
test_image_dir = 'trainn/'  # CHANGE TO THE UNSEEN TEST DATA
DIRECTORY!!!
test_dataset = ClockDataset(image_dir=test_image_dir,
transform=transform)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False,
num_workers=4)

# Evaluate the model on the test set
errors = []
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        errors.extend(torch.abs(outputs - labels).cpu().numpy())

# Calculate the median error
median_error = np.median(errors)
print(f'Median Error on the test set: {median_error:.2f} minutes')

# NOTE: This doesn't seem to work if you don't have a cuda enabled
GPU. I have tested this on my local machine and it works fine.
```

```
Median Error on the test set: 1.23 minutes
```

Q6. **(20 marks)** Generative models

The final task uses the same dataset as Question 5. However, this time you will only use the images, not the labels. The goal is to train a generative image model such as a Generative Adversarial Network, Variational Autoencoder, or Diffusion Model to create realistic clock images. Again, you may preprocess the images however you like and can augment to provide more images for training if you wish.

You should include a code tile that generates 8 random samples from your generative model. i.e., randomly sample from your latent space 8 times, pass these through your generator

network and display the resulting images. Ensure that your PDF file properly displays these images.

You should also include a code tile that performs latent space interpolation between two samples. Generate two random samples from the latent space as before, then linearly interpolate 5 intermediate latent vectors and display all 7 resulting images in order (i.e., the first randomly sampled clock image, then the 5 interpolated samples, then finally the second randomly sampled clock image). You would expect that the middle image looks something like an average between the start and end images, while all should be plausible. Ensure that your PDF file contains these images and choose an example where the two random samples are visually different so that the effect of the interpolation is clear.

(a) **[5 marks]** Appropriate network architecture chosen with justification.

I'll be using a Generative Adversarial Network (GAN) to solve this task. let's start by defining the Generator and Discriminator:

```
import torch
import torch.nn as nn

class Generator(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 256),
            nn.ReLU(True),
            nn.Linear(256, 512),
            nn.ReLU(True),
            nn.Linear(512, 1024),
            nn.ReLU(True),
            nn.Linear(1024, output_dim),
            nn.Tanh()
        )

    def forward(self, x):
        return self.model(x)

class Discriminator(nn.Module):
    def __init__(self, input_dim):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )
```

```
    def forward(self, x):
        return self.model(x)

# Dimensions
input_dim = 100  # Latent vector size
output_dim = 224 * 224  # Image size (224x224)
```

The Generator uses an input dimension of 100, 3 hidden layers and 3 ReLU activation layers.

Whereas the Discriminator uses an input the same size as the image (flattened), 3 hidden layers with decreasing size (1024, 512, 256) to progressively reduce the dimensionality, and LeakyReLU layers for non-linearity.

(b) **[5 marks]** Appropriate preprocessing applied, training loop implementation and choice of training hyperparameters.

Loading the images and preprocessing:

```
import os
from PIL import Image
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms

class ClockDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.image_files = [f for f in os.listdir(root_dir) if
f.endswith('.png')]

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

    def __getitem__(self, idx):
        img_name = os.path.join(self.root_dir, self.image_files[idx])
        image = Image.open(img_name).convert('L')
        if self.transform:
            image = self.transform(image)
        return image

# Define image transformations
transform = transforms.Compose([
    transforms.Resize(224), # Resize to 224x224 because 448x448 is too
large
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])
])

# Load the dataset
```

```
dataset = ClockDataset(root_dir='trainn/', transform=transform)
dataloader = DataLoader(dataset, batch_size=64, shuffle=True)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

import torch
import torch.optim as optim
import torch.nn as nn

# Initialize the models
generator = Generator(input_dim=100, output_dim=224*224).to(device)
discriminator = Discriminator(input_dim=224*224).to(device)

# Optimizers
lr = 0.0002
beta1 = 0.5
optimizer_G = optim.Adam(generator.parameters(), lr=lr, betas=(beta1,
0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr,
betas=(beta1, 0.999))

# Loss function
criterion = nn.BCELoss()

# Training Loop
num_epochs = 25
for epoch in range(num_epochs):
    for i, imgs in enumerate(dataloader):
        imgs = imgs.to(device)
        # Prepare real and fake labels
        real_labels = torch.ones(imgs.size(0), 1).to(device)
        fake_labels = torch.zeros(imgs.size(0), 1).to(device)

        # Train Discriminator
        optimizer_D.zero_grad()
        real_imgs = imgs.view(imgs.size(0), -1)
        outputs = discriminator(real_imgs)
        d_loss_real = criterion(outputs, real_labels)
        z = torch.randn(imgs.size(0), 100).to(device)
        fake_imgs = generator(z)
        outputs = discriminator(fake_imgs.detach())
        d_loss_fake = criterion(outputs, fake_labels)
        d_loss = d_loss_real + d_loss_fake
        d_loss.backward()
        optimizer_D.step()

        # Train Generator
        optimizer_G.zero_grad()
        z = torch.randn(imgs.size(0), 100).to(device)
        fake_imgs = generator(z).to(device)
```

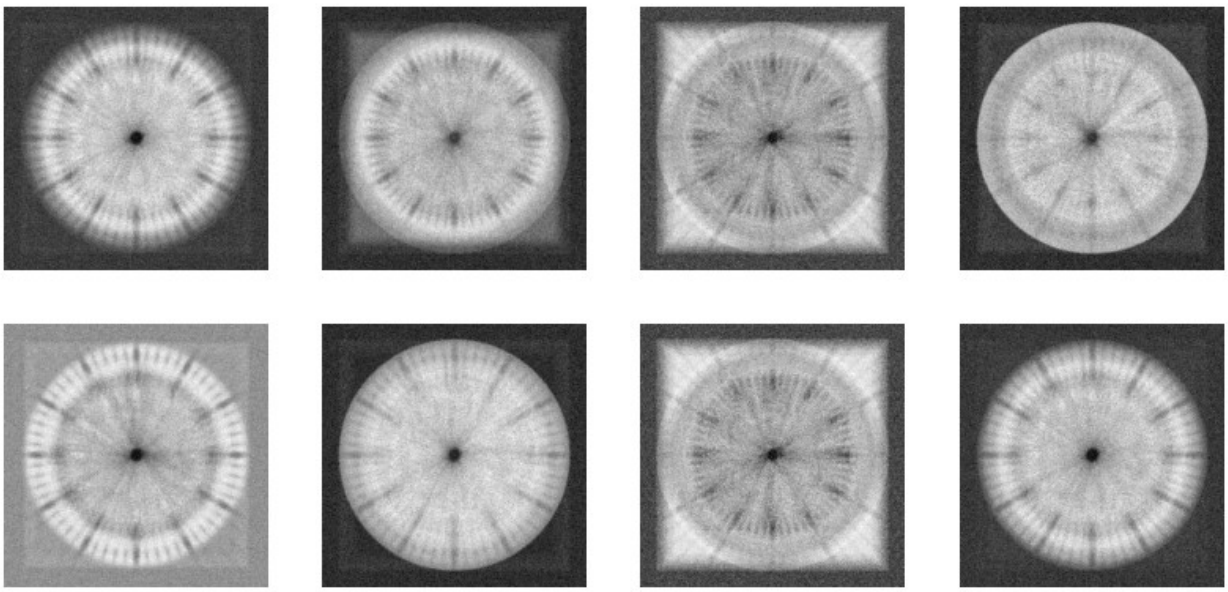
```
        outputs = discriminator(fake_imgs)
        g_loss = criterion(outputs, real_labels)
        g_loss.backward()
        optimizer_G.step()

    print(f'Epoch [{epoch}/{num_epochs}], d_loss: {d_loss.item()},
g_loss: {g_loss.item()}')
```

```
Epoch [0/25], d_loss: 0.6407524943351746, g_loss: 0.8230777978897095
Epoch [1/25], d_loss: 0.4898959994316101, g_loss: 1.0960900783538818
Epoch [2/25], d_loss: 1.3220152854919434, g_loss: 0.6460625529289246
Epoch [3/25], d_loss: 0.7588850259780884, g_loss: 0.6791025996208191
Epoch [4/25], d_loss: 0.5298561453819275, g_loss: 1.0751430988311768
Epoch [5/25], d_loss: 0.5774203538894653, g_loss: 1.1172550916671753
Epoch [6/25], d_loss: 0.5838243961334229, g_loss: 0.9120639562606812
Epoch [7/25], d_loss: 0.6613993048667908, g_loss: 0.906516432762146
Epoch [8/25], d_loss: 0.6807405352592468, g_loss: 0.8948749303817749
Epoch [9/25], d_loss: 0.786944568157196, g_loss: 1.0374425649642944
Epoch [10/25], d_loss: 0.6065340042114258, g_loss: 0.805011510848999
Epoch [11/25], d_loss: 0.6631759405136108, g_loss: 1.3471847772598267
Epoch [12/25], d_loss: 0.8572636246681213, g_loss: 1.1937320232391357
Epoch [13/25], d_loss: 0.5217522382736206, g_loss: 1.3324397802352905
Epoch [14/25], d_loss: 0.7532292008399963, g_loss: 1.1288541555404663
Epoch [15/25], d_loss: 0.9713233709335327, g_loss: 1.100295066833496
Epoch [16/25], d_loss: 0.6789244413375854, g_loss: 1.098163366317749
Epoch [17/25], d_loss: 0.962302029132843, g_loss: 1.471487045288086
Epoch [18/25], d_loss: 0.5623961687088013, g_loss: 1.3641471862792969
Epoch [19/25], d_loss: 0.8816056251525879, g_loss: 0.506594181060791
Epoch [20/25], d_loss: 0.8794061541557312, g_loss: 1.0328303575515747
Epoch [21/25], d_loss: 0.6669563055038452, g_loss: 1.1288020610809326
Epoch [22/25], d_loss: 0.6103840470314026, g_loss: 1.3319991827011108
Epoch [23/25], d_loss: 0.5665942430496216, g_loss: 1.598853349685669
Epoch [24/25], d_loss: 0.41996920108795166, g_loss: 2.0590436458587646
```

(c) **[5 marks]** Successfully generate 8 random clock images (marks awarded depending on quality of synthesised images).

```
import matplotlib.pyplot as plt

def generate_and_save_images(generator, num_images=8):
    # Ensure the generator is on the correct device
    device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
    generator.to(device)

    # Generate random noise and move it to the same device as the
generator
    z = torch.randn(num_images, 100).to(device)

    # Generate images and move them to CPU for visualization
```

```
    fake_images = generator(z).view(num_images, 224,
224).detach().cpu().numpy()

    # Plot the generated images
    plt.figure(figsize=(10, 10))
    for i in range(num_images):
        plt.subplot(4, 4, i+1)
        plt.imshow(fake_images[i], cmap='gray')
        plt.axis('off')
    plt.show()

generate_and_save_images(generator, num_images=8)
```



(d) **[5 marks]** Successfully interpolate between two random clock images in latent space.

```
def interpolate_images(generator, num_interpolations=5):
    # Ensure the generator is on the correct device
    device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
    generator.to(device)

    # Generate random noise and move it to the same device as the
generator
    z1 = torch.randn(1, 100).to(device)
    z2 = torch.randn(1, 100).to(device)
    interpolation_steps = torch.linspace(0, 1, num_interpolations +
2).to(device)

    interpolated_images = []
    for alpha in interpolation_steps:
        z = (1 - alpha) * z1 + alpha * z2
```

```
        interpolated_images.append(generator(z).view(224,
224).detach().cpu().numpy())

    # Plot the interpolated images
    plt.figure(figsize=(15, 5))
    for i, img in enumerate(interpolated_images):
        plt.subplot(1, num_interpolations + 2, i + 1)
        plt.imshow(img, cmap='gray')
        plt.axis('off')
    plt.show()

interpolate_images(generator)
```