National College of Ireland

Configuration Manual

MSc Research Project
MSCAI

David Oluwatimilehin Bamikole
Student ID: X22179640

School of Computing
National College of Ireland

Supervisor: Dr. Devanshu Anand

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	David Oluwatimilehin Bamikole
Student ID:	X22179640
Programme:	MSCAI
Year:	2024
Module:	MSc Research Project
Supervisor:	Dr. Devanshu Anand
Submission Due Date:	12/08/2024
Project Title:	Configuration Manual
Word Count:	509
Page Count:	10

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	David Oluwatimilehin Bamikole
Date:	16th September 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

David Oluwatimilehin Bamikole
X22179640

1 Introduction

This manual provides information on the environment where the project was implemented. It also gives an outline of how to reproduce the project titled "Evaluation of Multimodal Transformer Data Fusion Techniques".

2 System Configuration

This project was executed on Google Colab Pro, a web-based interactive Jupyter notebook compatible with Python programming language. The platform gives access to virtual system resources such as CPU, Disk space, RAM and GPU. Also, the platform comes with pre-installed libraries, with a provision to install other libraries as well. Figure 1 shows the system information and Figure 2 shows the GPU information used. The available disk space was 78GB.

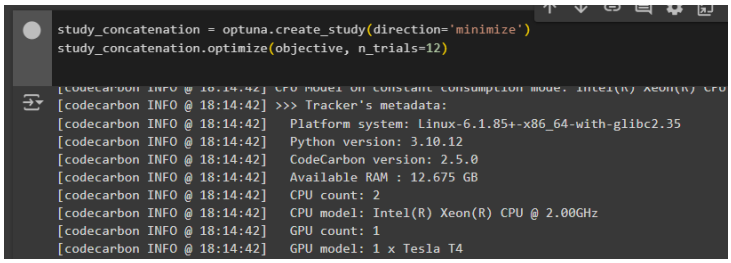


Figure 1: System Information

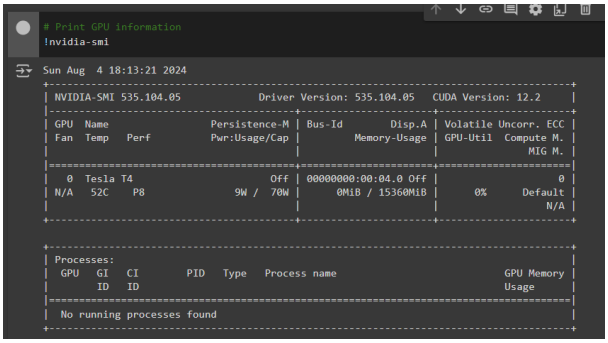


Figure 2: GPU Information

3 Python Libraries

The following libraries were used for the implementation of this project.

- Os
- Pickle
- H5py
- Pandas
- Torch
- Json
- Scipy
- Math
- PIL
- Numpy
- Time
- Seaborn
- Optuna
- Logging
- Matplotlib
- Transformers
- Codecarbon
- Scikit-learn

Most of the libraries come pre-installed on Colab while the other ones need to be installed. Figure 3 shows the snippet for the missing libraries installation and Figure 4 shows the importation of necessary libraries.

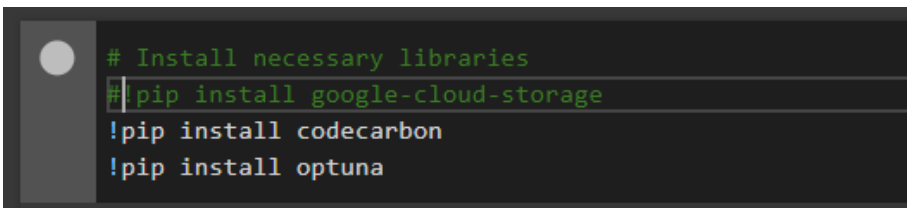


Figure 3: Libraries Installation

```python
# Import other necessary libraries
import pickle
import os
import h5py
import pandas as pd
import torch
import json
import scipy.stats
import math
from PIL import Image
import numpy as np
import time
import seaborn as sns
import optuna
import logging
import matplotlib.pyplot as plt

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as data
from transformers import BertTokenizer
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import Dataset, DataLoader
import torch.optim.lr_scheduler as lr_scheduler
from codecarbon import EmissionsTracker
from sklearn.metrics import classification_report, confusion_matrix
```

Figure 4: Importation of Libraries

4 Environment Setup

To run the experiment, the preprocessed data could be obtained on Google Drive [1] which is the storage for the official dataset GitHub account [2].

The '`mosi_raw.pkl`' will be downloaded from Google Drive and uploaded to the Colab root directory. At this stage, all the cells can be run sequentially.

5 Execution Stages Explained

The code snippet in Figure 5 shows the dataset loading phrase.

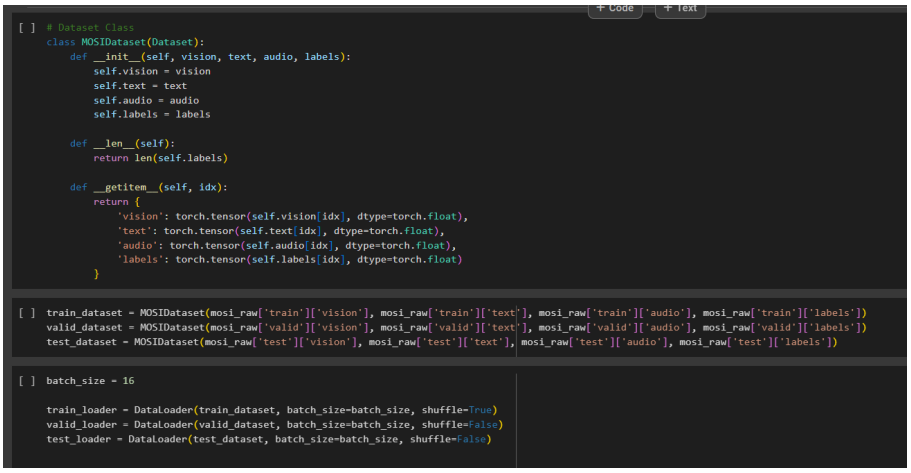
```python
[ ] # Set device to GPU if it exist
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

[ ] # Set data name
    MOSI_RAW = '//mosi_raw.pkl'
    data_dir = destination_file_path + MOSI_RAW

    # Define the source and destination paths
    source = './mosi_raw.pkl'
    destination = './data/mosi_raw.pkl'

    # Move the file
    os.rename(source, destination)

[ ] # Import data
    with open(data_dir, 'rb') as file:
        mosi_raw = pickle.load(file)
        print(f'mosi_raw loaded from {data_dir}')

    mosi_raw loaded from ./data///mosi_raw.pkl
```

Figure 5: Loading of Dataset

[1] `https://drive.google.com/drive/folders/1uEK737LXB9jAlf9kyqRs6B9N6cDncodq?usp=sharing`

[2] `https://github.com/pliang279/MultiBench?tab=readme-ov-file`

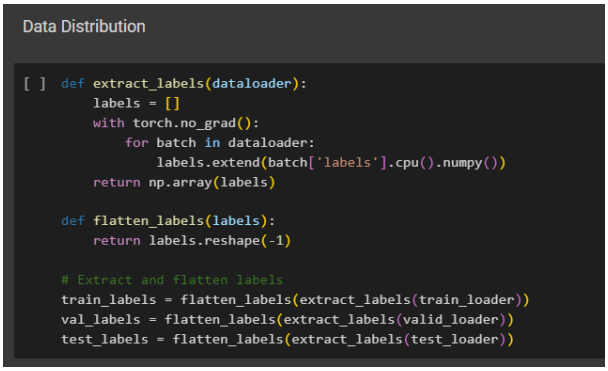
The code snippet in Figure 6 shows the alignment and creation of dataloader for all dataset groups.



```
[ ]  # Dataset Class
     class MOSIDataset(Dataset):
         def __init__(self, vision, text, audio, labels):
             self.vision = vision
             self.text = text
             self.audio = audio
             self.labels = labels

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

         def __getitem__(self, idx):
             return {
                 'vision': torch.tensor(self.vision[idx], dtype=torch.float),
                 'text': torch.tensor(self.text[idx], dtype=torch.float),
                 'audio': torch.tensor(self.audio[idx], dtype=torch.float),
                 'labels': torch.tensor(self.labels[idx], dtype=torch.float)
             }

[ ]  train_dataset = MOSIDataset(mosi_raw['train']['vision'], mosi_raw['train']['text'], mosi_raw['train']['audio'], mosi_raw['train']['labels'])
     valid_dataset = MOSIDataset(mosi_raw['valid']['vision'], mosi_raw['valid']['text'], mosi_raw['valid']['audio'], mosi_raw['valid']['labels'])
     test_dataset = MOSIDataset(mosi_raw['test']['vision'], mosi_raw['test']['text'], mosi_raw['test']['audio'], mosi_raw['test']['labels'])

[ ]  batch_size = 16

     train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
     valid_loader = DataLoader(valid_dataset, batch_size=batch_size, shuffle=False)
     test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Figure 6: Dataloader Implementation

The code snippet in Figure 7 generates the dataset distribution plot in Figure 8.



```
Data Distribution

[ ]  def extract_labels(dataloader):
         labels = []
         with torch.no_grad():
             for batch in dataloader:
                 labels.extend(batch['labels'].cpu().numpy())
         return np.array(labels)

     def flatten_labels(labels):
         return labels.reshape(-1)

     # Extract and flatten labels
     train_labels = flatten_labels(extract_labels(train_loader))
     val_labels = flatten_labels(extract_labels(valid_loader))
     test_labels = flatten_labels(extract_labels(test_loader))
```

Figure 7: Code for Dataset Distribution

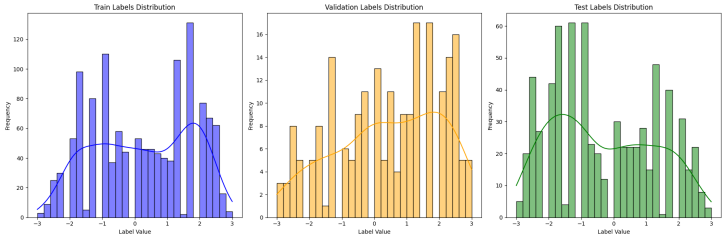


Figure 8: Dataset Distribution

The code snippet in Figure 9 shows the helper functions to normalize the dataset, identify NaN and inf in the data and calculate Pearson correlation

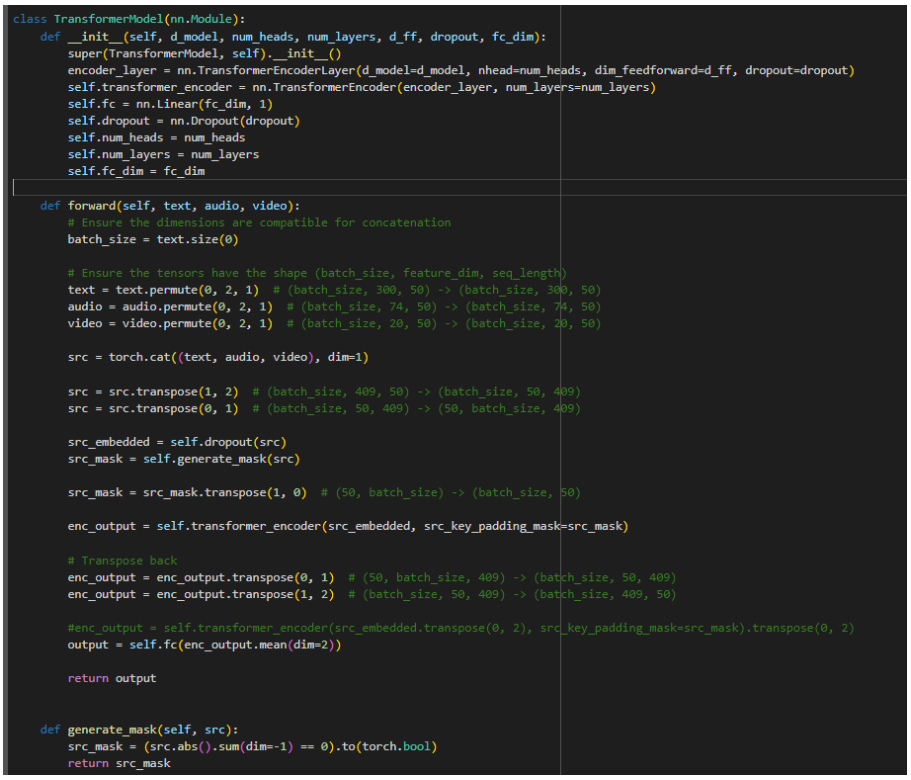
The code snippet for the early concatenation is shown in Figure 10, while the code snippet for training the model and hyperparameter search is shown in Figure 11. Figure 12 shows the code snippet for the best parameter obtained.

The code snippet for the cross-modal attention is shown in Figure 13 and Figure 14, while the code snippet for training the model and hyperparameter search is shown in Figure 15. Figure 16 shows the code snippet for the best parameter obtained.

```python
# Data normalization
def normalize(tensor, eps=1e-8):
    tensor = torch.clamp(tensor, -1e4, 1e4)
    mean = tensor.mean(dim=-1, keepdim=True)
    std = tensor.std(dim=-1, keepdim=True)
    normalized_tensor = (tensor - mean) / (std + eps)
    # Replace any NaNs with zeros
    normalized_tensor = torch.nan_to_num(normalized_tensor, nan=0.0)
    return normalized_tensor

# Check for NaN in the data
def check_for_nans(tensor, name):
    if torch.isnan(tensor).any():
        print(f"NaN values found in {name}")
    if torch.isinf(tensor).any():
        print(f"Infinite values found in {name}")

# Check for inf in the data
def count_nans_infs(tensor):
    nan_count = torch.isnan(tensor).sum().item()
    inf_count = torch.isinf(tensor).sum().item()
    return nan_count, inf_count

def calculate_pearson_corr(y_true, y_pred):
    return scipy.stats.pearsonr(y_true, y_pred)[0]
```

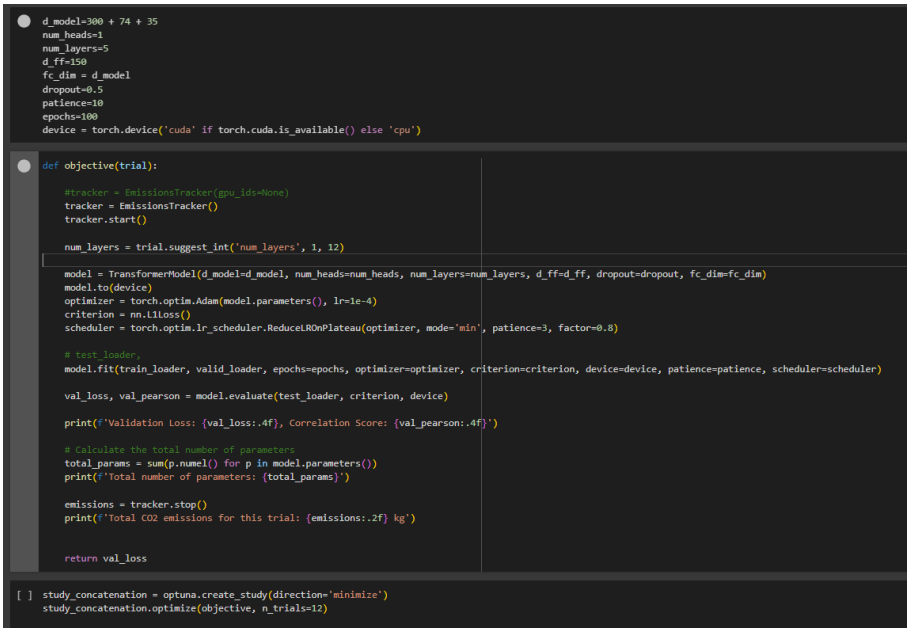
Figure 9: Utility Function

The code snippet for the hierarchy modal attention is shown in Figure 17, while the code snippet for training the model and hyperparameter search is shown in Figure 18. Figure 19 shows the code snippet for the best parameter obtained.



```python
class TransformerModel(nn.Module):
    def __init__(self, d_model, num_heads, num_layers, d_ff, dropout, fc_dim):
        super(TransformerModel, self).__init__()
        encoder_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=num_heads, dim_feedforward=d_ff, dropout=dropout)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_layers)
        self.fc = nn.Linear(fc_dim, 1)
        self.dropout = nn.Dropout(dropout)
        self.num_heads = num_heads
        self.num_layers = num_layers
        self.fc_dim = fc_dim

    def forward(self, text, audio, video):
        # Ensure the dimensions are compatible for concatenation
        batch_size = text.size(0)

        # Ensure the tensors have the shape (batch_size, feature_dim, seq_length)
        text = text.permute(0, 2, 1)  # (batch_size, 300, 50) -> (batch_size, 300, 50)
        audio = audio.permute(0, 2, 1)  # (batch_size, 74, 50) -> (batch_size, 74, 50)
        video = video.permute(0, 2, 1)  # (batch_size, 20, 50) -> (batch_size, 20, 50)

        src = torch.cat((text, audio, video), dim=1)

        src = src.transpose(1, 2)  # (batch_size, 409, 50) -> (batch_size, 50, 409)
        src = src.transpose(0, 1)  # (batch_size, 50, 409) -> (50, batch_size, 409)

        src_embedded = self.dropout(src)
        src_mask = self.generate_mask(src)

        src_mask = src_mask.transpose(1, 0)  # (50, batch_size) -> (batch_size, 50)

        enc_output = self.transformer_encoder(src_embedded, src_key_padding_mask=src_mask)

        # Transpose back
        enc_output = enc_output.transpose(0, 1)  # (50, batch_size, 409) -> (batch_size, 50, 409)
        enc_output = enc_output.transpose(1, 2)  # (batch_size, 50, 409) -> (batch_size, 409, 50)

        #enc_output = self.transformer_encoder(src_embedded.transpose(0, 2), src_key_padding_mask=src_mask).transpose(0, 2)
        output = self.fc(enc_output.mean(dim=2))

        return output


    def generate_mask(self, src):
        src_mask = (src.abs().sum(dim=-1) == 0).to(torch.bool)
        return src_mask
```

Figure 10: Early Concatenation Design Code

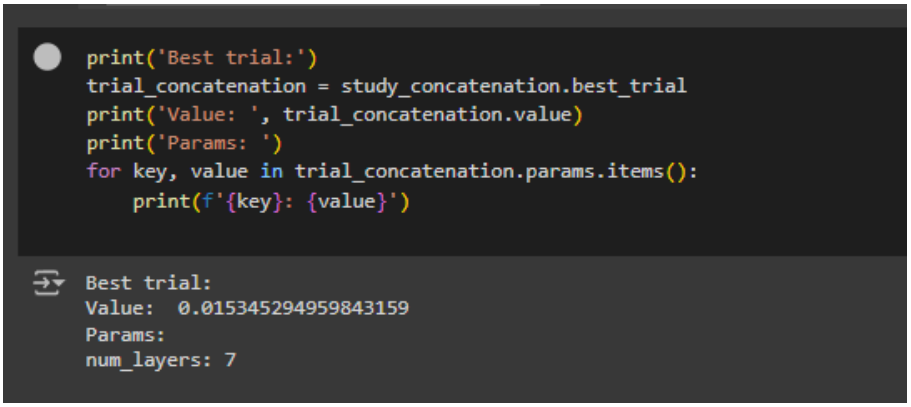


```python
d_model=300 + 74 + 35
num_heads=1
num_layers=5
d_ff=150
fc_dim = d_model
dropout=0.5
patience=10
epochs=100
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```python
def objective(trial):

    #tracker = EmissionsTracker(gpu_ids=None)
    tracker = EmissionsTracker()
    tracker.start()

    num_layers = trial.suggest_int('num_layers', 1, 12)

    model = TransformerModel(d_model=d_model, num_heads=num_heads, num_layers=num_layers, d_ff=d_ff, dropout=dropout, fc_dim=fc_dim)
    model.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
    criterion = nn.L1Loss()
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', patience=3, factor=0.8)

    # test_loader,
    model.fit(train_loader, valid_loader, epochs=epochs, optimizer=optimizer, criterion=criterion, device=device, patience=patience, scheduler=scheduler)

    val_loss, val_pearson = model.evaluate(test_loader, criterion, device)

    print(f'Validation Loss: {val_loss:.4f}, Correlation Score: {val_pearson:.4f}')

    # Calculate the total number of parameters
    total_params = sum(p.numel() for p in model.parameters())
    print(f'Total number of parameters: {total_params}')

    emissions = tracker.stop()
    print(f'Total CO2 emissions for this trial: {emissions:.2f} kg')


    return val_loss
```

```python
[ ] study_concatenation = optuna.create_study(direction='minimize')
    study_concatenation.optimize(objective, n_trials=12)
```

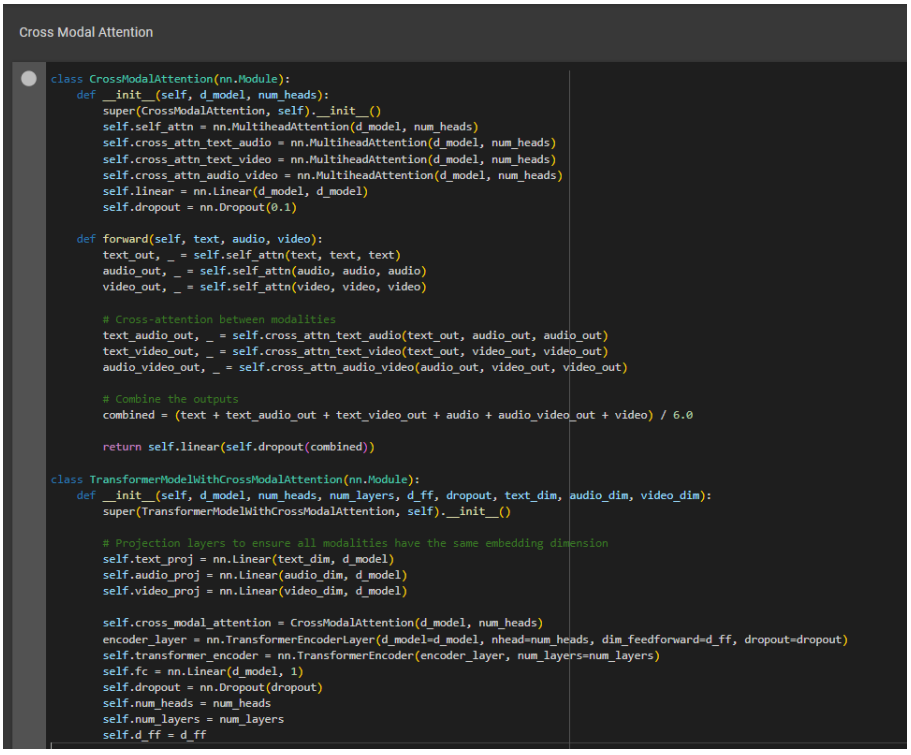
Figure 11: Early Concatenation Implementation Code



```
print('Best trial:')
trial_concatenation = study_concatenation.best_trial
print('Value: ', trial_concatenation.value)
print('Params: ')
for key, value in trial_concatenation.params.items():
    print(f'{key}: {value}')
```

```
Best trial:
Value:  0.015345294959843159
Params:
num_layers: 7
```

Figure 12: Early Concatenation Best Parameter Code



Cross Modal Attention

```
class CrossModalAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(CrossModalAttention, self).__init__()
        self.self_attn = nn.MultiheadAttention(d_model, num_heads)
        self.cross_attn_text_audio = nn.MultiheadAttention(d_model, num_heads)
        self.cross_attn_text_video = nn.MultiheadAttention(d_model, num_heads)
        self.cross_attn_audio_video = nn.MultiheadAttention(d_model, num_heads)
        self.linear = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(0.1)

    def forward(self, text, audio, video):
        text_out, _ = self.self_attn(text, text, text)
        audio_out, _ = self.self_attn(audio, audio, audio)
        video_out, _ = self.self_attn(video, video, video)

        # Cross-attention between modalities
        text_audio_out, _ = self.cross_attn_text_audio(text_out, audio_out, audio_out)
        text_video_out, _ = self.cross_attn_text_video(text_out, video_out, video_out)
        audio_video_out, _ = self.cross_attn_audio_video(audio_out, video_out, video_out)

        # Combine the outputs
        combined = (text + text_audio_out + text_video_out + audio + audio_video_out + video) / 6.0

        return self.linear(self.dropout(combined))

class TransformerModelWithCrossModalAttention(nn.Module):
    def __init__(self, d_model, num_heads, num_layers, d_ff, dropout, text_dim, audio_dim, video_dim):
        super(TransformerModelWithCrossModalAttention, self).__init__()

        # Projection layers to ensure all modalities have the same embedding dimension
        self.text_proj = nn.Linear(text_dim, d_model)
        self.audio_proj = nn.Linear(audio_dim, d_model)
        self.video_proj = nn.Linear(video_dim, d_model)

        self.cross_modal_attention = CrossModalAttention(d_model, num_heads)
        encoder_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=num_heads, dim_feedforward=d_ff, dropout=dropout)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_layers)
        self.fc = nn.Linear(d_model, 1)
        self.dropout = nn.Dropout(dropout)
        self.num_heads = num_heads
        self.num_layers = num_layers
        self.d_ff = d_ff
```

Figure 13: Cross-Modal Attention Design Code

```python
def forward(self, text, audio, video):
    batch_size = text.size(0)

    # Project the input features to the same dimension
    text = self.text_proj(text)  # (batch_size, seq_length, d_model)
    audio = self.audio_proj(audio)
    video = self.video_proj(video)

    # Ensure the dimensions match before applying attention
    text = text.permute(1, 0, 2)  # (seq_length, batch_size, d_model)
    audio = audio.permute(1, 0, 2)
    video = video.permute(1, 0, 2)

    # Apply cross-modal attention
    combined = self.cross_modal_attention(text, audio, video)

    # Prepare input for Transformer Encoder
    #src = combined.transpose(1, 2)#***
    src = combined.transpose(0, 1)  # (seq_length, batch_size, d_model)

    src_embedded = self.dropout(src)
    src_mask = self.generate_mask(src)

    src_mask = src_mask.transpose(1, 0)#***

    enc_output = self.transformer_encoder(src_embedded, src_key_padding_mask=src_mask)

    # Pooling and Linear layer
    enc_output = enc_output.mean(dim=1)  # (batch_size, d_model)
    output = self.fc(enc_output)

    return output

def generate_mask(self, src):
    src_mask = (src.abs().sum(dim=-1) == 0).to(torch.bool)
    return src_mask
```

Figure 14: Cross-Modal Attention Design Code

```
[ ] print('Best trial:')
    trial_cross_atn = study_cross_atn.best_trial
    print('Value: ', trial_cross_atn.value)
    print('Params: ')
    for key, value in trial_cross_atn.params.items():
        print(f'{key}: {value}')
```

```
Best trial:
Value:  0.015166140161454678
Params:
num_heads: 8
num_layers: 8
```

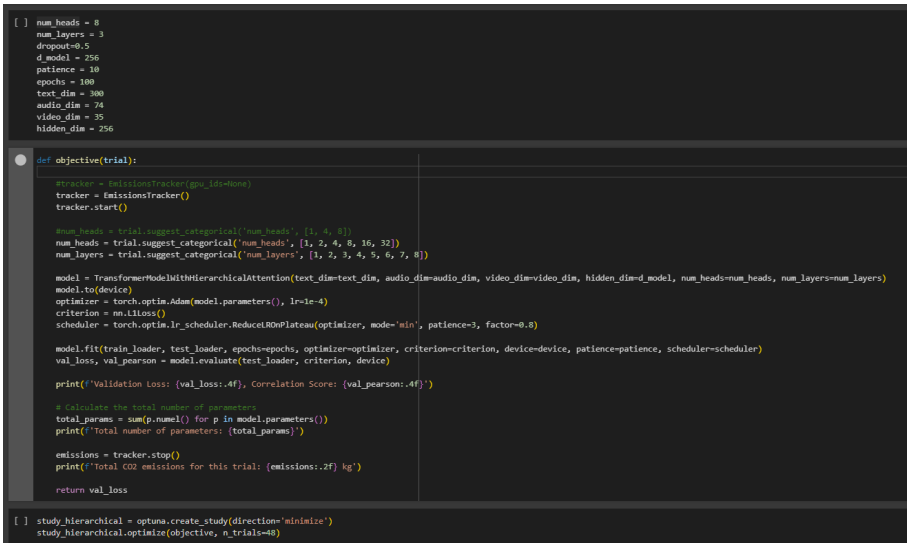
Figure 15: Cross-Modal Attention Implementation Code

```
[ ]  print('Best trial:')
     trial_cross_atn = study_cross_atn.best_trial
     print('Value: ', trial_cross_atn.value)
     print('Params: ')
     for key, value in trial_cross_atn.params.items():
         print(f'{key}: {value}')

     Best trial:
     Value:  0.015166140161454678
     Params:
     num_heads: 8
     num_layers: 8
```

Figure 16: Cross-Modal Attention Best Parameter Code

```
class TransformerModelWithHierarchicalAttention(nn.Module):
    def __init__(self, text_dim, audio_dim, video_dim, hidden_dim, num_heads, num_layers):
        super(TransformerModelWithHierarchicalAttention, self).__init__()

        # Transformer encoders for each modality
        self.text_encoder = nn.TransformerEncoder(nn.TransformerEncoderLayer(d_model=hidden_dim, nhead=num_heads), num_layers=num_layers)
        self.audio_encoder = nn.TransformerEncoder(nn.TransformerEncoderLayer(d_model=hidden_dim, nhead=num_heads), num_layers=num_layers)
        self.video_encoder = nn.TransformerEncoder(nn.TransformerEncoderLayer(d_model=hidden_dim, nhead=num_heads), num_layers=num_layers)

        # Projection layers for each modality
        self.text_proj = nn.Linear(text_dim, hidden_dim)
        self.audio_proj = nn.Linear(audio_dim, hidden_dim)
        self.video_proj = nn.Linear(video_dim, hidden_dim)

        # Transformer encoder for fused output
        self.fusion_encoder = nn.TransformerEncoder(nn.TransformerEncoderLayer(d_model=hidden_dim * 3, nhead=num_heads), num_layers=num_layers)
        # Final fusion layer
        self.fusion_layer = nn.Linear(hidden_dim * 3, hidden_dim)
        # Final output layer
        self.output_layer = nn.Linear(hidden_dim, 1)  # Adjust output size as per requirement

        self.num_heads = num_heads
        self.num_layers = num_layers

    def forward(self, text, audio, video):
        # Project modalities to common space
        text_emb = self.text_proj(text)  # (batch_size, seq_length, hidden_dim)
        audio_emb = self.audio_proj(audio)
        video_emb = self.video_proj(video)

        # Encode each modality independently
        text_encoded = self.text_encoder(text_emb.permute(1, 0, 2))  # (seq_length, batch_size, hidden_dim)
        audio_encoded = self.audio_encoder(audio_emb.permute(1, 0, 2))
        video_encoded = self.video_encoder(video_emb.permute(1, 0, 2))

        # Concatenate encoded outputs
        concat_emb = torch.cat((text_encoded, audio_encoded, video_encoded), dim=-1)  # (seq_length, batch_size, hidden_dim * 3)
        # Fuse concatenated outputs with another Transformer
        fusion_output = self.fusion_encoder(concat_emb)  # (seq_length, batch_size, hidden_dim * 3)
        # Final fusion layer
        fused_emb = self.fusion_layer(fusion_output.permute(1, 0, 2))  # (batch_size, seq_length, hidden_dim)
        # Output layer
        output = self.output_layer(fused_emb[:, 0, :])  # Use the output of the first token for prediction
        return output
```

Figure 17: Hierarchy Modal Attention Design Code

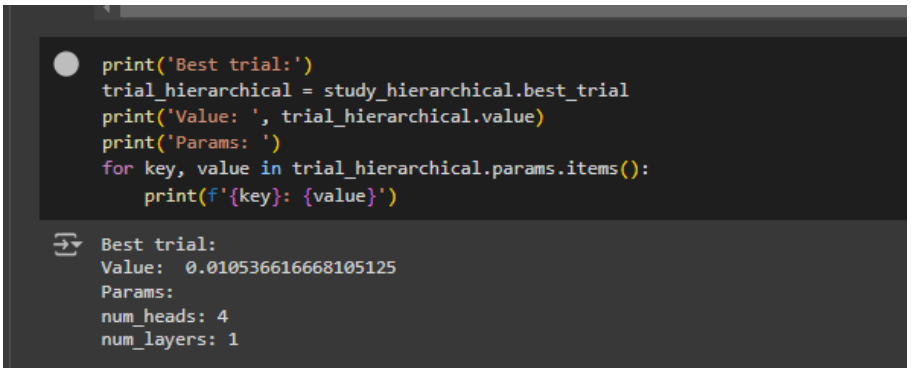


```
num_heads = 8
num_layers = 3
dropout=0.5
d_model = 256
patience = 10
epochs = 100
text_dim = 300
audio_dim = 74
video_dim = 35
hidden_dim = 256
```

```
def objective(trial):

    #tracker = EmissionsTracker(gpu_ids=None)
    tracker = EmissionsTracker()
    tracker.start()

    #num_heads = trial.suggest_categorical('num_heads', [1, 4, 8])
    num_heads = trial.suggest_categorical('num_heads', [1, 2, 4, 8, 16, 32])
    num_layers = trial.suggest_categorical('num_layers', [1, 2, 3, 4, 5, 6, 7, 8])

    model = TransformerModelWithHierarchicalAttention(text_dim=text_dim, audio_dim=audio_dim, video_dim=video_dim, hidden_dim=d_model, num_heads=num_heads, num_layers=num_layers)
    model.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
    criterion = nn.L1Loss()
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', patience=3, factor=0.8)

    model.fit(train_loader, test_loader, epochs=epochs, optimizer=optimizer, criterion=criterion, device=device, patience=patience, scheduler=scheduler)
    val_loss, val_pearson = model.evaluate(test_loader, criterion, device)

    print(f'Validation Loss: {val_loss:.4f}, Correlation Score: {val_pearson:.4f}')

    # Calculate the total number of parameters
    total_params = sum(p.numel() for p in model.parameters())
    print(f'Total number of parameters: {total_params}')

    emissions = tracker.stop()
    print(f'Total CO2 emissions for this trial: {emissions:.2f} kg')

    return val_loss
```

```
study_hierarchical = optuna.create_study(direction='minimize')
study_hierarchical.optimize(objective, n_trials=48)
```

Figure 18: Hierarchy Modal Attention Implementation Code



```
print('Best trial:')
trial_hierarchical = study_hierarchical.best_trial
print('Value: ', trial_hierarchical.value)
print('Params: ')
for key, value in trial_hierarchical.params.items():
    print(f'{key}: {value}')
```

```
Best trial:
Value:  0.010536616668105125
Params:
num_heads: 4
num_layers: 1
```

Figure 19: Hierarchy Modal Attention Best Parameter Code