Comparative Analysis of Multimodal and Single Entity Models

An Effort to Learn Something Amazing

Abdur Razzak
Department of Computer Science
New Mexico State University, Las Cruces, New Mexico, USA
arazzak@nmsu.edu

Javed Akhtar
Department of Mechanical Engineering
New Mexico State University, Las Cruces, New Mexico, USA
akhtarj@nmsu.edu

ABSTRACT

This work presents a comparative analysis of Multimodal Machine Learning models with single entity models, focusing on their performance, challenges, and pitfalls. Specifically, the study compares the performance of two data entities (textual and image) in Multimodal architectures such as LSTM, with the current state-of-the-art single entity models such as SVM for textual data and CNN for image data. The research aims to deeply investigate the outputs of these models.

The objective of this project was to gain knowledge and insight into the emerging and highly innovative field of multimodal machine learning by creating a model that integrates multiple modalities, specifically text and image data, for making predictions. In addition, the aim was to acquire expertise in current cutting-edge single entity models such as SVM for textual and CNN for image datasets.

KEYWORDS

Single entry model, Multimodality model, SVM, CNN, LSTM

1 Problem Statement

In this work, we will analyze Multimodal Machine Learning models along with single entity models and compare their performances, challenges, and pitfalls. More concretely, two data entities (textual and image) will be trained and tested in Multimodal architectures like Long Short-Term Memory (LSTM) model and in contrast, will also be fitted to the current state of the art single entity model like Support Vector Machine (SVM) for textual data and Convolutional Neural Networks (CNN) for image data and, finally, we will deeply investigate their outputs.

2 Motivation

The human experience of the world is complex and involves various modes of perception. These modes of perception include visual, auditory, tactile, olfactory, and gustatory senses. The term modality refers to the way in which something happens or is experienced, and a research problem is characterized as multimodal when it includes multiple such modalities. To improve the ability of artificial intelligence (AI) to serve humanity, it needs to understand the world in the same way that humans do, using multiple senses at once. This is where multimodal machine learning comes into play.

Multimodal machine learning refers to the integration of information from various modalities within a research problem. For instance, in computer vision, an image may be accompanied by text, audio, or other data sources. Similarly, in natural language processing, a sentence may be complemented by an image or video. By combining information from multiple modalities, AI can potentially achieve better performance and make more robust predictions.

Multimodal machine learning has applications in several areas such as speech recognition, emotion recognition, recommendation systems, and autonomous driving. In speech recognition, multimodal learning can help to improve accuracy by integrating information from both auditory and visual modalities. In emotion recognition, multimodal learning can help to better identify emotions by integrating information from both facial expressions and speech. Recommendation systems can also benefit from multimodal learning by incorporating information from both text and visual modalities. In autonomous driving, multimodal learning can improve safety by combining information from different sensors, such as cameras and lidar, to make more accurate predictions.

Finally, multimodal machine learning is an emerging field that shows enormous potential to revolutionize the artificial intelligence landscape. With continued efforts and investment, this area of research represents a promising and exciting next stage in the evolution of AI.

3 Preparation before Fitting the Models

This project involves the use of three models, namely SVM, CNN, and LSTM, to fit a dataset consisting of two data entities, namely textual and image data. The SVM model will be utilized to make predictions using textual data, while the CNN model will be used to fit image data.
The preparations we had to make before training and testing our models are as follows:

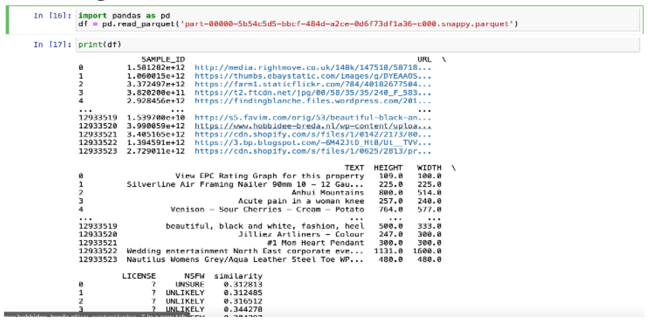
- ❑ Data Selection
- ❑ Data Acquisition and Data Separation
- ❑ Data Preprocessing for SVM model
- ❑ Data Preprocessing for CNN model
- ❑ Data Preprocessing for LSTM model

3.1 Data Selection

Our project aims to examine the benefits and challenges of utilizing multimodality models over single entity models. To achieve this, we require a dataset with at least two distinct data entities referring to the same object. Initially, we began working with the MS-COCO dataset for image classification and the fake and real news dataset to analyze text data. However, using these datasets separately did not yield desired results, as they lacked the shared context required to align text and image data. Multimodal machine learning involves co-learning between entities, which necessitated a dataset that met these requirements.
After extensive research, we chose the LAION-400M dataset, which comprises 400 million image-text pairs that have been filtered with CLIP. The textual data provides descriptions of the image, making it an ideal dataset for conducting a comparative analysis of single entity and multimodality models. Given the massive size of the dataset and the limitations of our computational resources, we focused solely on the first subset of the dataset, part-00000-5b54c5d5-bbcf-484d-a2ce-0d6f73df1a36-c000.parquet.
This allowed us to evaluate the model's effectiveness in a shorter timeframe.

3.2 Data Acquisition and Data Separation

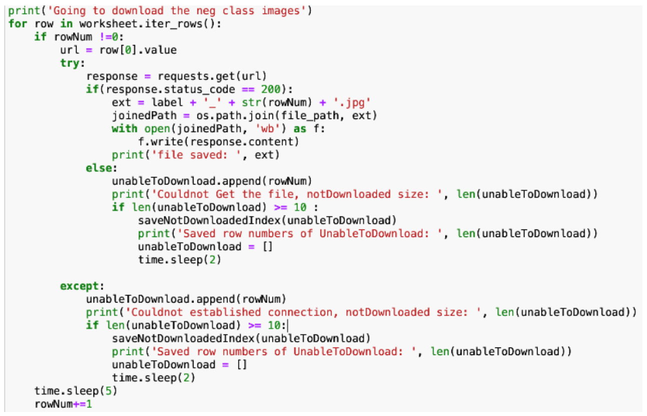
Data Acquisition: We extracted the parquet file using the following code.



```
In [16]: import pandas as pd
         df = pd.read_parquet('part-00000-5b54c5d5-bbcf-484d-a2ce-0d6f73df1a36-c000.snappy.parquet')

In [17]: print(df)

                SAMPLE_ID                                                URL  \
0            1.581282e+12  http://media.rightmove.co.uk/148k/147518/58718...
1            1.060015e+12  https://thumbs.ebaystatic.com/images/g/DYEAAOS...
2            3.372497e+12  https://farm1.staticflickr.com/784/40182677504...
3            3.820200e+11  https://t2.ftcdn.net/jpg/00/58/35/35/240_F_583...
4            2.928456e+12  https://findingblanche.files.wordpress.com/201...
...                   ...                                                ...
12933519     1.539700e+10  http://s5.favim.com/orig/53/beautiful-black-an...
12933520     3.990059e+12  https://www.hobbidee-breda.nl/wp-content/uploa...
12933521     3.405165e+12  https://cdn.shopify.com/s/files/1/0142/2173/80...
12933522     1.394591e+12  https://3.bp.blogspot.com/-6M42JtD_Ht8/Ut__TVV...
12933523     2.729011e+12  https://cdn.shopify.com/s/files/1/0625/2813/pr...

                                                       TEXT  HEIGHT   WIDTH  \
0                     View EPC Rating Graph for this property   109.0   100.0
1          Silverline Air Framing Nailer 90mm 10 - 12 Gau...   225.0   225.0
2                                            Anhui Mountains   800.0   514.0
3                                   Acute pain in a woman knee   257.0   240.0
4                    Venison - Sour Cherries - Cream - Potato   764.0   577.0
...                                                       ...     ...     ...
12933519               beautiful, black and white, fashion, heel   500.0   333.0
12933520                            Jilliez Artliners - Colour   247.0   300.0
12933521                                   #1 Mom Heart Pendant   300.0   300.0
12933522  Wedding entertainment North East corporate eve...  1131.0  1600.0
12933523  Nautilus Womens Grey/Aqua Leather Steel Toe WP...   480.0   480.0

            LICENSE      NSFW  similarity
0                 ?    UNSURE    0.312813
1                 ?  UNLIKELY    0.312485
2                 ?  UNLIKELY    0.316512
3                 ?  UNLIKELY    0.344278
```

After analyzing the various columns in the dataset, we determined that the TEXT and URL columns were the most effective. These columns revealed the relationship between the text and the image URL. However, the dataset did not contain raw images; instead, it provided image links. We stored the URL list in a separate excel file and the text in a text file. To download the images, we created a downloader that gradually fetched the URLs since direct images were not available.

- ❑ Used response '**requests.get(url)**' to download images.
- ❑ Managed different exception cases like **ConnectionError**, **FileNotFound**, **Forbidden** etc.
- ❑ Kept track of images that were corrupted or not being downloaded and excluded those correspondent textual data to keep the data consistent.
- ❑ Renamed as 'class_imageNumber.jpg' in order to help understand the data better.
- ❑ For positive images we kept all the positive images in pos directory and the negative images on the neg directory

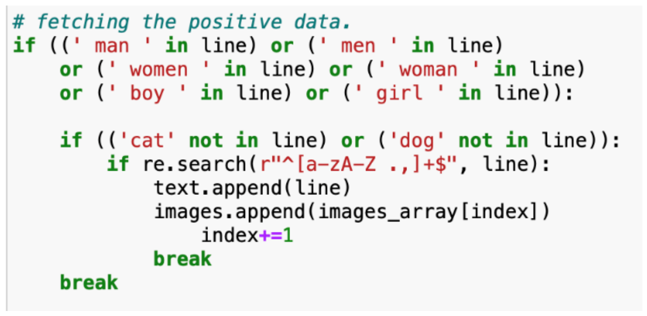


```
print('Going to download the neg class images')
for row in worksheet.iter_rows():
    if rowNum !=0:
        url = row[0].value
        try:
            response = requests.get(url)
            if(response.status_code == 200):
                ext = label + '_' + str(rowNum) + '.jpg'
                joinedPath = os.path.join(file_path, ext)
                with open(joinedPath, 'wb') as f:
                    f.write(response.content)
                print('file saved: ', ext)
            else:
                unableToDownload.append(rowNum)
                print('Couldnot Get the file, notDownloaded size: ', len(unableToDownload))
                if len(unableToDownload) >= 10 :
                    saveNotDownloadedIndex(unableToDownload)
                    print('Saved row numbers of UnableToDownload: ', len(unableToDownload))
                    unableToDownload = []
                    time.sleep(2)

        except:
            unableToDownload.append(rowNum)
            print('Couldnot established connection, notDownloaded size: ', len(unableToDownload))
            if len(unableToDownload) >= 10:
                saveNotDownloadedIndex(unableToDownload)
                print('Saved row numbers of UnableToDownload: ', len(unableToDownload))
                unableToDownload = []
                time.sleep(2)
    time.sleep(5)
    rowNum+=1
```

Data Separation: Once the downloading is finished, we analyzed the data and found that some of the data is not consistent. We removed the text descriptions if it is any one of the following.

- ❑ Image is wrongly/partially downloaded.
- ❑ Text area is empty.
- ❑ Text has unusual characters except alpha numeric.

While downloading, we labeled as positive or 1 if the datapoint has person in it and otherwise negative or 0. We applied the following condition to filter out for example position class.



```
# fetching the positive data.
if ((' man ' in line) or (' men ' in line)
    or (' women ' in line) or (' woman ' in line)
    or (' boy ' in line) or (' girl ' in line)):

    if (('cat' not in line) or ('dog' not in line)):
        if re.search(r"^[a-zA-Z .,]+$", line):
            text.append(line)
            images.append(images_array[index])
                index+=1
            break
    break
```

After completing all the above mentioned procedures, we were able to download total of 12314 instances, the distribution (between positive and negative class) of which is shown in the chat below.

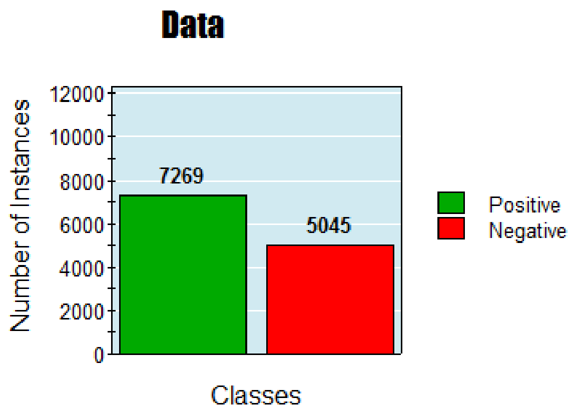


Figure 1: Data distribution of positive and negative class

The images depicted in the following figure consist of a positive (top) and a negative (bottom) instance, each labeled with their respective image name and description. The positive instance's image name is 1_7, indicating that it belongs to the positive class (as denoted by the first letter) and that its corresponding textual data has a sequence number of 7. Although a few images were not downloaded for reasons mentioned earlier, we were able to successfully download all instances of textual data. This naming convention helps to ensure that the textual descriptions of the images are accurately matched.

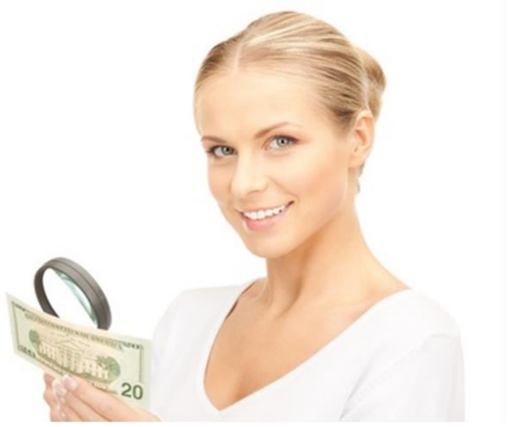


Image Name: 1_7
Image Description: lovely woman with magnifying glass and money photo

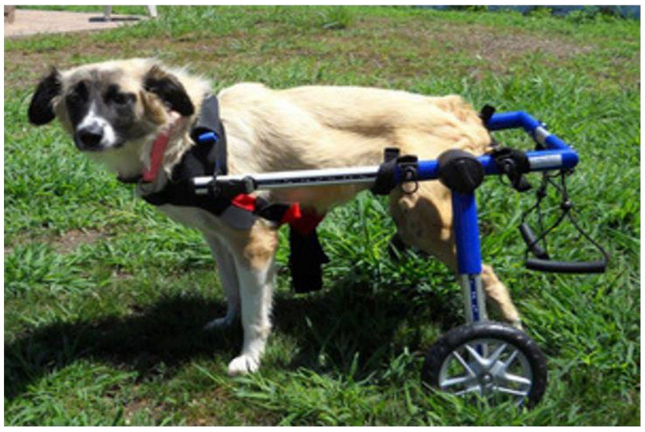
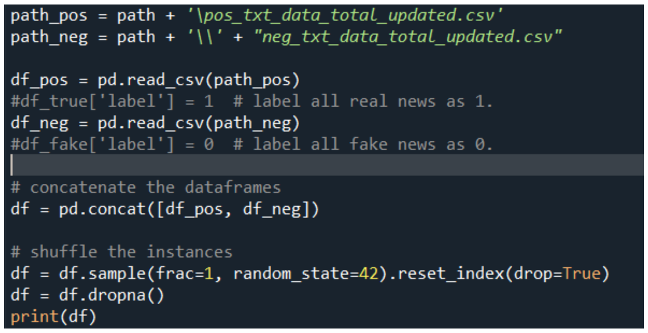


Image Name: 0_14
Image Description: Walkin Wheels dog wheelchair

Figure 1: Positive (top) and negative (bottom) class with image name and description.

3.3 Data Preparation for SVM

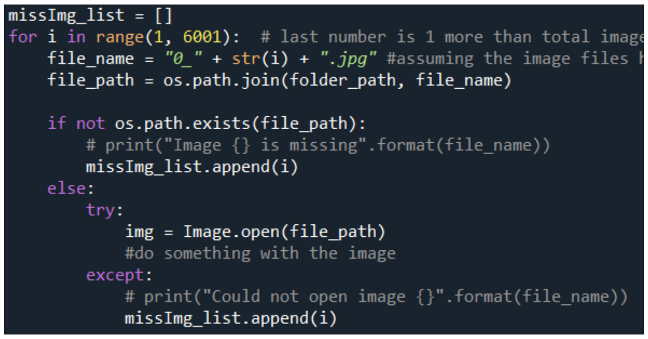
As mentioned in the previous section, the positive and negative classes of textual data were labeled 1 and 0, respectively. To assign these labels, we imported two CSV files, one containing positive instances and the other containing negative instances, as dataframes. Using pandas, we added a column named "class" to the positive class dataframe and assigned it a value of 1 for all instances. We repeated this process for the negative class dataframe, assigning a value of 0 to the "class" column for all instances. Finally, we concatenated the two dataframes (df_pos and df_neg) and shuffled the instances to create a new dataframe with positive and negative instances randomly distributed. The corresponding code is provided below.



```
path_pos = path + '\pos_txt_data_total_updated.csv'
path_neg = path + '\\' + "neg_txt_data_total_updated.csv"

df_pos = pd.read_csv(path_pos)
#df_true['label'] = 1  # label all real news as 1.
df_neg = pd.read_csv(path_neg)
#df_fake['label'] = 0  # label all fake news as 0.

# concatenate the dataframes
df = pd.concat([df_pos, df_neg])

# shuffle the instances
df = df.sample(frac=1, random_state=42).reset_index(drop=True)
df = df.dropna()
print(df)
```

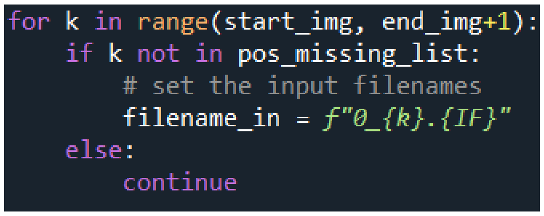
3.4 Data Preparation for CNN

The downloaded images needed to be preprocessed before fitting them into the CNN model. However, due to the issues mentioned in section 3.2, we were unable to download every single image listed in the URL file. Consequently, the image names were not in a uniform, sequential order (e.g., 1_1, 1_2, 1_3, etc.). Moreover, some of the downloaded images were corrupted and unfit for use in the model. To account for these issues, we created a list (missImg_list) to store the sequential number of missing and corrupted images (e.g., if image 1_55 was missing or corrupted, we stored the value 55 in the list). The corresponding code is provided below.



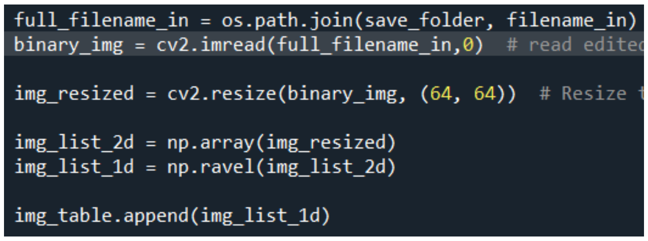
```
missImg_list = []
for i in range(1, 6001):  # last number is 1 more than total image
    file_name = "0_" + str(i) + ".jpg" #assuming the image files h
    file_path = os.path.join(folder_path, file_name)

    if not os.path.exists(file_path):
        # print("Image {} is missing".format(file_name))
        missImg_list.append(i)
    else:
        try:
            img = Image.open(file_path)
            #do something with the image
        except:
            # print("Could not open image {}".format(file_name))
            missImg_list.append(i)
```

The missImg_list was saved as a CSV file, and it is essential when storing image pixel information in a CSV file. We used the missImg_list to access the non-uniform and non-corrupted images based on their names. This was done by looping through the missImg_list using the image names (code provided below).



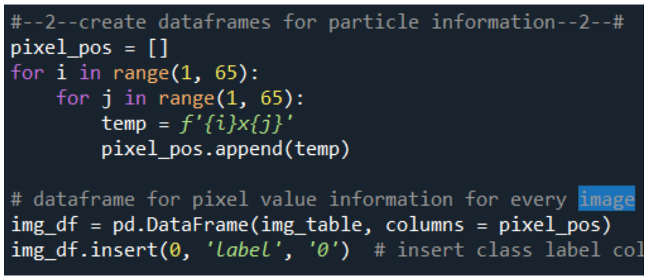
```
for k in range(start_img, end_img+1):
    if k not in pos_missing_list:
        # set the input filenames
        filename_in = f"0_{k}.{IF}"
    else:
        continue
```

In order to utilize the image data for the CNN model, we saved the pixel value of each image in a csv file. Each row of the csv file represented an image, with the first column dedicated to the class label of the image (1 for positive class and 0 for negative class), and the remaining columns reserved for the pixel values. Prior to storage, the images were converted to grayscale and resized to a dimension of 64x64. This resulted in a total of 4096 (64x64) columns being used to store the pixel values of the resized images. The corresponding code for this process can be found below.



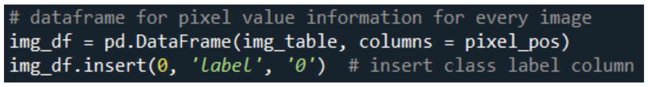
```
full_filename_in = os.path.join(save_folder, filename_in)
binary_img = cv2.imread(full_filename_in,0)  # read edite

img_resized = cv2.resize(binary_img, (64, 64))  # Resize t

img_list_2d = np.array(img_resized)
img_list_1d = np.ravel(img_list_2d)

img_table.append(img_list_1d)
```

The pixel value at position (1, 1) was stored in 1×1 column and other pixel values follow the same pattern. Since we downloaded the data for positive and negative class separately and stored the images in separate folders, therefore, we created the separate csv file for positive and negative class of images (Code below).



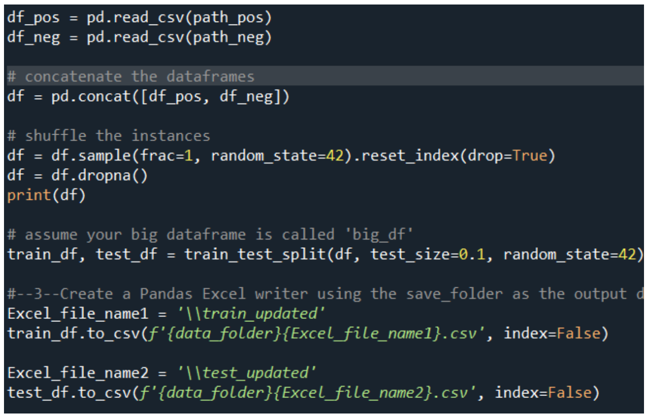
```
#--2--create dataframes for particle information--2--#
pixel_pos = []
for i in range(1, 65):
    for j in range(1, 65):
        temp = f'{i}x{j}'
        pixel_pos.append(temp)

# dataframe for pixel value information for every image
img_df = pd.DataFrame(img_table, columns = pixel_pos)
img_df.insert(0, 'Label', '0')  # insert class label col
```

The first column of the csv file was added with all 0 values for negative class of images (and 1 for positive class of images).



```
# dataframe for pixel value information for every image
img_df = pd.DataFrame(img_table, columns = pixel_pos)
img_df.insert(0, 'Label', '0')  # insert class label column
```

Then, positive and negative class csv files were imported as dataframe and merged and shuffled to create new dataframe with uniformly distributed dataset for positive and negative class. **'train-test-split'** from scikit-learn was used to divide the dataset into train and test dataset in ratio of 90:10. The resulting train and test dataset was then saved as train and test csv file (Code below).

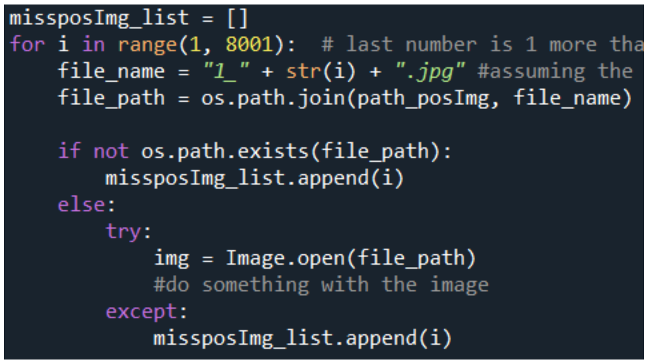


```
df_pos = pd.read_csv(path_pos)
df_neg = pd.read_csv(path_neg)

# concatenate the dataframes
df = pd.concat([df_pos, df_neg])

# shuffle the instances
df = df.sample(frac=1, random_state=42).reset_index(drop=True)
df = df.dropna()
print(df)

# assume your big dataframe is called 'big_df'
train_df, test_df = train_test_split(df, test_size=0.1, random_state=42)

#--3--Create a Pandas Excel writer using the save_folder as the output d
Excel_file_name1 = '\\train_updated'
train_df.to_csv(f'{data_folder}{Excel_file_name1}.csv', index=False)

Excel_file_name2 = '\\test_updated'
test_df.to_csv(f'{data_folder}{Excel_file_name2}.csv', index=False)
```

3.5 Data Preparation for LSTM

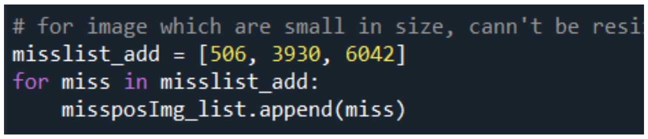
In order to prepare the downloaded data for the LSTM model, we had to meet certain requirements. Although the images and their descriptions were downloaded separately, as described in section 3.2, we encountered several issues while attempting to download all of the images. Consequently, we were not able to retrieve all of them. However, accessing the textual descriptions of the images was relatively easy, which allowed us to obtain all of them. As a result, we assigned each image a sequence number based on its corresponding text description. Additionally, we assigned a letter to indicate whether the image belonged to the positive or negative class, with "1" indicating the positive class and "0" indicating the negative class. We also included an underscore and sequence number after the letter to differentiate between images. Therefore, if an image related to the nth positive class text description (which is stored at the nth position in the positive text CSV file) was missing, the images with a sequence number of 1_n would be missing from the image folder.

As with the preparation of CNN data, we obtained access to the positive image folder and retrieved all the available images. However, any images that were either unavailable or corrupted were identified and compiled into a list based on their sequence number in the image name.



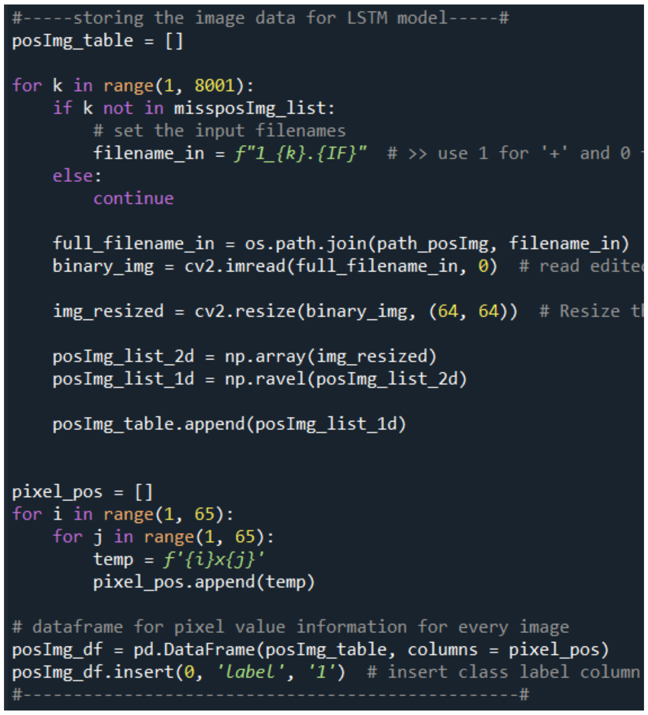
```
missposImg_list = []
for i in range(1, 8001):  # last number is 1 more tha
    file_name = "1_" + str(i) + ".jpg" #assuming the
    file_path = os.path.join(path_posImg, file_name)

    if not os.path.exists(file_path):
        missposImg_list.append(i)
    else:
        try:
            img = Image.open(file_path)
            #do something with the image
        except:
            missposImg_list.append(i)
```

A few of the images had inappropriate sizes, hence were unable to resize to 64×64, therefore, we removed them manually one by one.



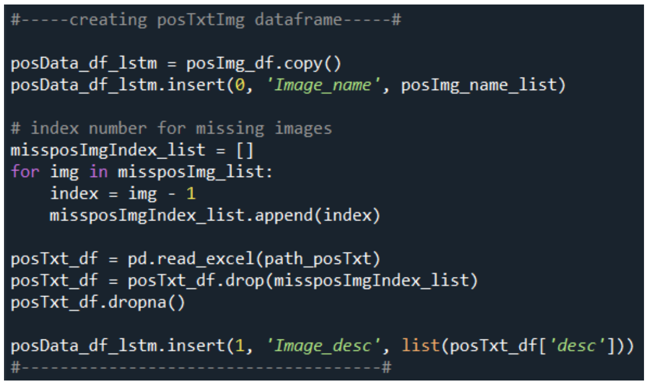
```
# for image which are small in size, cann't be resi
misslist_add = [506, 3930, 6042]
for miss in misslist_add:
    missposImg_list.append(miss)
```

Once we obtained the list of missing or unsuitable images for the LSTM model, we proceeded to execute a loop that would store the pixel values of each image in the positive class as a row in a dataframe 'posImg_df'. In this dataframe, 4096 columns (64x64) were allocated for the pixel values of each image. Additionally, we added another column to the dataframe to indicate the class of each image, with a value of 1 for positive class. This process was similar to that of the CNN model's data preparation, as outlined in section 3.4. The corresponding codes for this process can be found below.



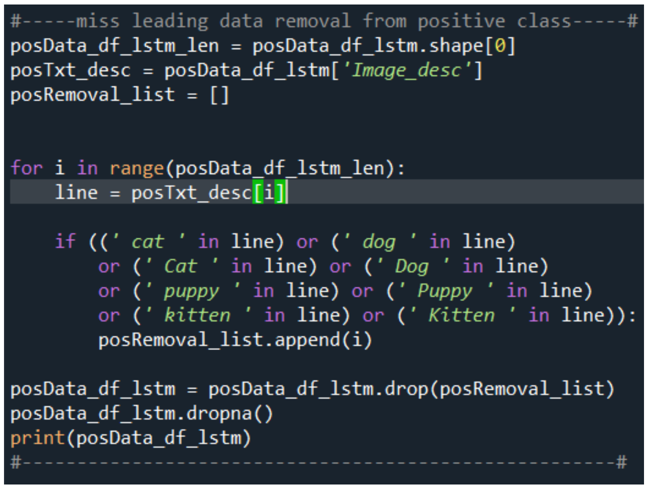
```
#-----storing the image data for LSTM model-----#
posImg_table = []

for k in range(1, 8001):
    if k not in missposImg_list:
        # set the input filenames
        filename_in = f"1_{k}.{IF}"  # >> use 1 for '+' and 0
    else:
        continue

    full_filename_in = os.path.join(path_posImg, filename_in)
    binary_img = cv2.imread(full_filename_in, 0)  # read edite

    img_resized = cv2.resize(binary_img, (64, 64))  # Resize t

    posImg_list_2d = np.array(img_resized)
    posImg_list_1d = np.ravel(posImg_list_2d)

    posImg_table.append(posImg_list_1d)


pixel_pos = []
for i in range(1, 65):
    for j in range(1, 65):
        temp = f'{i}x{j}'
        pixel_pos.append(temp)

# dataframe for pixel value information for every image
posImg_df = pd.DataFrame(posImg_table, columns = pixel_pos)
posImg_df.insert(0, 'label', '1')  # insert class label column
#----------------------------------------------#
```

To continue with the LSTM model, we first created a new dataframe named 'posData_df_lstm' by making a copy of the existing images dataframe, 'posImg_df'. We also accessed the textual data in the form of another dataframe called 'posTxt_df', and removed the instances which corresponding to missing image data of dataframe ‘posImg_df’. we access the textual data (images description) as another dataframe ‘posTxt_df’ and dropped all instances that corresponded to missing image data in the 'posImg_df' dataframe. Following this, we added the 'desc' column of the 'posTxt_df' dataframe (which contained the image descriptions) to the final dataframe 'posData_df_lstm'.



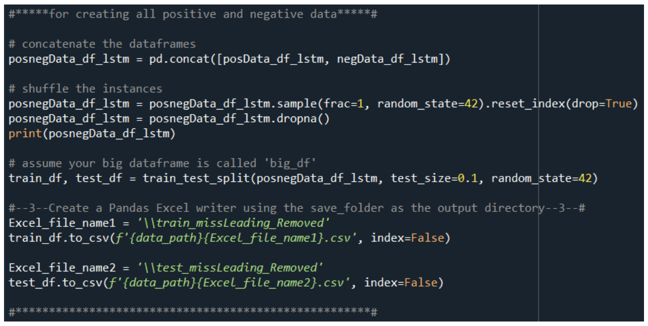
```
#-----creating posTxtImg dataframe-----#

posData_df_lstm = posImg_df.copy()
posData_df_lstm.insert(0, 'Image_name', posImg_name_list)

# index number for missing images
missposImgIndex_list = []
for img in missposImg_list:
    index = img - 1
    missposImgIndex_list.append(index)

posTxt_df = pd.read_excel(path_posTxt)
posTxt_df = posTxt_df.drop(missposImgIndex_list)
posTxt_df.dropna()

posData_df_lstm.insert(1, 'Image_desc', list(posTxt_df['desc']))
#--------------------------------------#
```

At the outset, we made a mistake while downloading image data by failing to use the appropriate keywords to distinguish between positive and negative class images. As a result, we ended up with a few images containing both positive and negative class labels, which could potentially compromise the accuracy of our model. In order to mitigate this issue, we removed these problematic images. We did so by accessing the text descriptions of each image from the ‘posData_df_lstm’ dataframe and searching for keywords such as "dog" or "cat," or any other synonymous terms that could indicate the presence of both positive and negative class images within a single image. Whenever we found such instances, we removed them along with the corresponding image information (refer to the code below).



```
#-----miss leading data removal from positive class-----#
posData_df_lstm_len = posData_df_lstm.shape[0]
posTxt_desc = posData_df_lstm['Image_desc']
posRemoval_list = []


for i in range(posData_df_lstm_len):
    line = posTxt_desc[i]

    if ((' cat ' in line) or (' dog ' in line)
        or (' Cat ' in line) or (' Dog ' in line)
        or (' puppy ' in line) or (' Puppy ' in line)
        or (' kitten ' in line) or (' Kitten ' in line)):
        posRemoval_list.append(i)

posData_df_lstm = posData_df_lstm.drop(posRemoval_list)
posData_df_lstm.dropna()
print(posData_df_lstm)
#-------------------------------------------------------#
```

We followed the same procedure to obtain the 'negData_df_lstm' dataframe for negative class data instances. Once we had dataframes for both positive and negative classes, we concatenated them to create the final 'posnegData_df_lstm' dataframe. We then shuffled the instances in the dataframe to ensure a homogeneous distribution of positive and negative classes. To split the data into training and testing datasets in a 90:10 ratio, we used the **'train_test_split'** function from scikit-learn. Finally, we saved the resulting training and testing datasets as a CSV file, which would be used for training and testing the LSTM model.

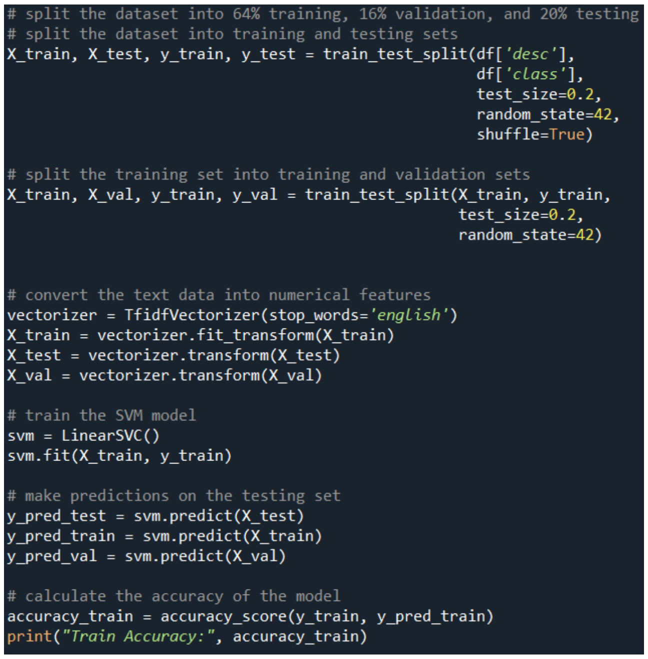


```
#*****for creating all positive and negative data*****#

# concatenate the dataframes
posnegData_df_lstm = pd.concat([posData_df_lstm, negData_df_lstm])

# shuffle the instances
posnegData_df_lstm = posnegData_df_lstm.sample(frac=1, random_state=42).reset_index(drop=True)
posnegData_df_lstm = posnegData_df_lstm.dropna()
print(posnegData_df_lstm)

# assume your big dataframe is called 'big_df'
train_df, test_df = train_test_split(posnegData_df_lstm, test_size=0.1, random_state=42)

#--3--Create a Pandas Excel writer using the save_folder as the output directory--3--#
Excel_file_name1 = '\\train_missLeading_Removed'
train_df.to_csv(f'{data_path}{Excel_file_name1}.csv', index=False)

Excel_file_name2 = '\\test_missLeading_Removed'
test_df.to_csv(f'{data_path}{Excel_file_name2}.csv', index=False)

#****************************************************#
```

4 Fitting SVM Model on Textual Data

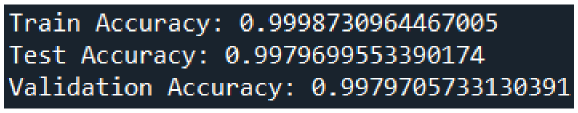
In this project, the Support Vector Machine (SVM) model was used for making predictions based on the textual data entities of the LAION-400M dataset. To prepare the dataset for fitting the SVM model, it was first collected in accordance with the model's requirements, as outlined in section 3.2. The data was then preprocessed as described in section 3.3.

Next, the shuffled dataframe from section 3.3 was split into training, testing, and validation sets in a ratio of 64:20:16. The text data was converted into numerical features using the 'TfidfVectorizer' method. The SVM model was then trained and validated using the 'LinearSVC' class from the scikit-learn library. Finally, the trained model was used to make predictions on the test dataset and the accuracy of the model was evaluated using the 'accuracy_score' function from scikit-learn (Code below).



```
# split the dataset into 64% training, 16% validation, and 20% testing
# split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(df['desc'],
                                                    df['class'],
                                                    test_size=0.2,
                                                    random_state=42,
                                                    shuffle=True)

# split the training set into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
                                                  test_size=0.2,
                                                  random_state=42)


# convert the text data into numerical features
vectorizer = TfidfVectorizer(stop_words='english')
X_train = vectorizer.fit_transform(X_train)
X_test = vectorizer.transform(X_test)
X_val = vectorizer.transform(X_val)

# train the SVM model
svm = LinearSVC()
svm.fit(X_train, y_train)

# make predictions on the testing set
y_pred_test = svm.predict(X_test)
y_pred_train = svm.predict(X_train)
y_pred_val = svm.predict(X_val)

# calculate the accuracy of the model
accuracy_train = accuracy_score(y_train, y_pred_train)
print("Train Accuracy:", accuracy_train)
```

The training, testing and validation accuracy of the model was more than 99%.

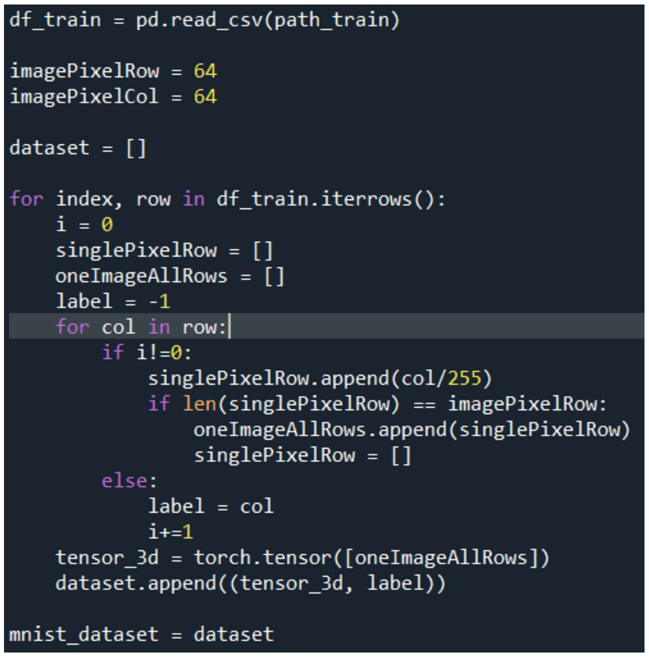


```
Train Accuracy: 0.9998730964467005
Test Accuracy: 0.9979699553390174
Validation Accuracy: 0.9979705733130391
```

5 Fitting CNN Model on Image Data

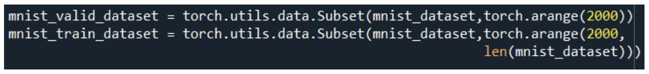
The CNN model was utilized to predict using image data entities of the LAION-400M dataset. To prepare the data for the CNN model, the requirements were identified and implemented as outlined in section 3.4.

The train and test csv files were imported as dataframes, and each instance in the dataframe was converted into a tuple with two items. The first item of the tuple was a tensor of 64x64 containing the pixel values of resized gray images, while the second item was the label of the instances from the dataframe. These tuples were then stored in a list. The corresponding code for this process can be found below.



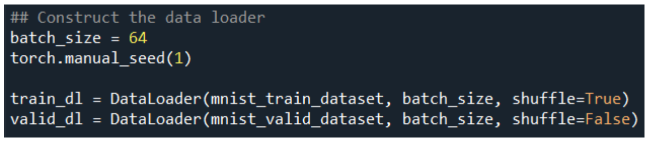
```
df_train = pd.read_csv(path_train)

imagePixelRow = 64
imagePixelCol = 64

dataset = []

for index, row in df_train.iterrows():
    i = 0
    singlePixelRow = []
    oneImageAllRows = []
    label = -1
    for col in row:
        if i!=0:
            singlePixelRow.append(col/255)
            if len(singlePixelRow) == imagePixelRow:
                oneImageAllRows.append(singlePixelRow)
                singlePixelRow = []
        else:
            label = col
        i+=1
    tensor_3d = torch.tensor([oneImageAllRows])
    dataset.append((tensor_3d, label))

mnist_dataset = dataset
```

The converted training dataset was again divided into train and validation dataset using **'torch.utils.data.Subset'** from torch library.



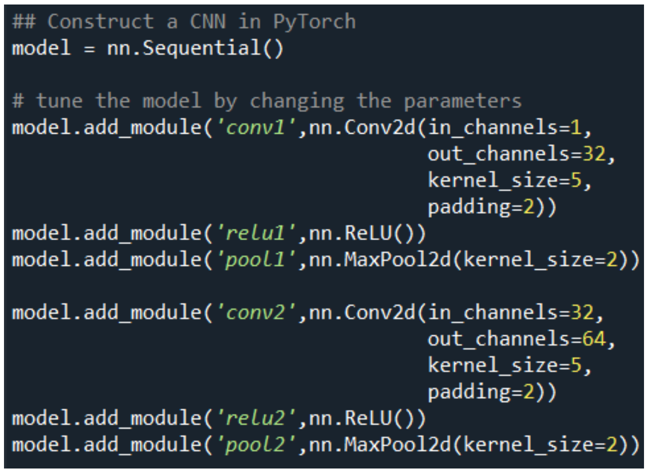
```
mnist_valid_dataset = torch.utils.data.Subset(mnist_dataset,torch.arange(2000))
mnist_train_dataset = torch.utils.data.Subset(mnist_dataset,torch.arange(2000,
                                                                 len(mnist_dataset)))
```

Then, we constructed the data loader with batches of 64 images for the training set and validation set, respectively using **'DataLoader'** from torch.utils.data.



```
## Construct the data loader
batch_size = 64
torch.manual_seed(1)

train_dl = DataLoader(mnist_train_dataset, batch_size, shuffle=True)
valid_dl = DataLoader(mnist_valid_dataset, batch_size, shuffle=False)
```

We use the **'torch.nn'** Sequential class to stack different layers, such as convolution, pooling, and dropout, as well as the fully connected layers.



```
## Construct a CNN in PyTorch
model = nn.Sequential()

# tune the model by changing the parameters
model.add_module('conv1',nn.Conv2d(in_channels=1,
                                   out_channels=32,
                                   kernel_size=5,
                                   padding=2))
model.add_module('relu1',nn.ReLU())
model.add_module('pool1',nn.MaxPool2d(kernel_size=2))

model.add_module('conv2',nn.Conv2d(in_channels=32,
                                   out_channels=64,
                                   kernel_size=5,
                                   padding=2))
model.add_module('relu2',nn.ReLU())
model.add_module('pool2',nn.MaxPool2d(kernel_size=2))
```

We read the images and the default dimension for the channels is the first dimension of the tensor array. The size of the convolutional output o depends on the given input vector of size n, filter of size (or kernel size) m, padding parameter p, and stride parameter s. $o = \left\lfloor \frac{n+2p-m}{s} \right\rfloor + 1$.
The CNN model that we utilized in this project has a multilayer architecture, as depicted in the figure below. The parameters of the architecture are listed as follows.
First Convolution Layer (conv1): kernel size 5×5, padding 2×2, output channel size 32.

- ❑ First Pooling (pool1): kernel size 2×2
- ❑ Second Convolution Layer (conv2): kernel size 5×5, padding 2×2, output channel size 64.
- ❑ Second Pooling (pool2): kernel size 2×2
- ❑ First Fully Connected Layer (FC1): output size 1024
- ❑ Second Fully Connected Layer (FC2): output size 2

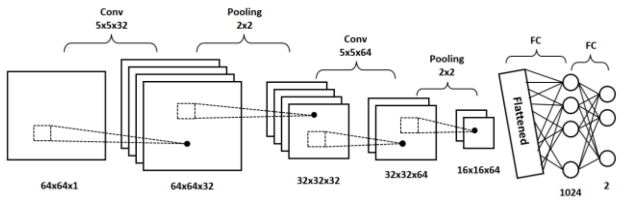
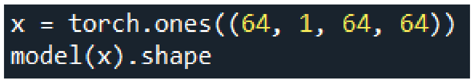


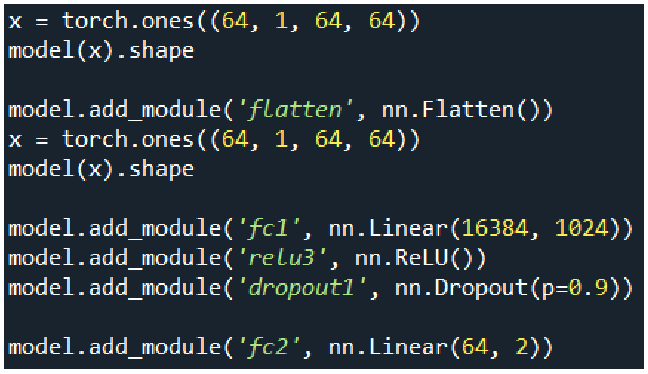
Figure 2: Multilayer CNN architecture.

PyTorch provides a convenient method to compute the size of the feature maps at this stage by providing the input shape as a tuple (64, 1, 64, 64) (64 images within the batch, 1 channel, and image size 64×64). The output is a shape (64, 64, 16, 16), indicating feature maps with 64 channels and a spatial size of 64×64.



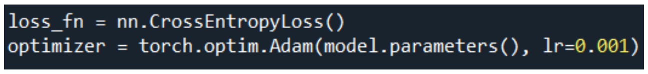
```
x = torch.ones((64, 1, 64, 64))
model(x).shape
```

For fully connected (FC) layer, the input of first FC layer must have rank 2, that is, shape [batch-size×input_units]. To meet this requirement for the FC layer, the output of the previous layer must be flattened. We added two fully connected layers named 'fc1' and 'fc2' which have 1024 and 2 output units respectfully.



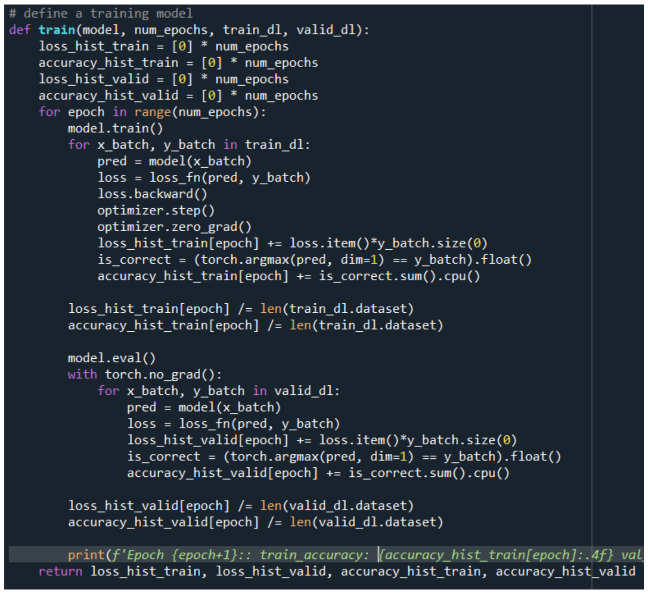
```
x = torch.ones((64, 1, 64, 64))
model(x).shape

model.add_module('flatten', nn.Flatten())
x = torch.ones((64, 1, 64, 64))
model(x).shape

model.add_module('fc1', nn.Linear(16384, 1024))
model.add_module('relu3', nn.ReLU())
model.add_module('dropout1', nn.Dropout(p=0.9))

model.add_module('fc2', nn.Linear(64, 2))
```

We tried four fully connected layers as well, but it did not affect the accuracy.
Then, we created a loss function and an optimizer for the model using **'CrossEntropyLoss'** from torch.nn and **'Adam optimizer'** from torch.



```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Finally, we define the training model using the algorithm mentioned in the class.



```
# define a training model
def train(model, num_epochs, train_dl, valid_dl):
    loss_hist_train = [0] * num_epochs
    accuracy_hist_train = [0] * num_epochs
    loss_hist_valid = [0] * num_epochs
    accuracy_hist_valid = [0] * num_epochs
    for epoch in range(num_epochs):
        model.train()
        for x_batch, y_batch in train_dl:
            pred = model(x_batch)
            loss = loss_fn(pred, y_batch)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            loss_hist_train[epoch] += loss.item()*y_batch.size(0)
            is_correct = (torch.argmax(pred, dim=1) == y_batch).float()
            accuracy_hist_train[epoch] += is_correct.sum().cpu()

        loss_hist_train[epoch] /= len(train_dl.dataset)
        accuracy_hist_train[epoch] /= len(train_dl.dataset)

        model.eval()
        with torch.no_grad():
            for x_batch, y_batch in valid_dl:
                pred = model(x_batch)
                loss = loss_fn(pred, y_batch)
                loss_hist_valid[epoch] += loss.item()*y_batch.size(0)
                is_correct = (torch.argmax(pred, dim=1) == y_batch).float()
                accuracy_hist_valid[epoch] += is_correct.sum().cpu()

        loss_hist_valid[epoch] /= len(valid_dl.dataset)
        accuracy_hist_valid[epoch] /= len(valid_dl.dataset)

        print(f'Epoch {epoch+1}:: train_accuracy: {accuracy_hist_train[epoch]:.4f} val
    return loss_hist_train, loss_hist_valid, accuracy_hist_train, accuracy_hist_valid
```

The model was tuned by changing the value of different parameters of convolution and pooling layers. The best performance we obtained with little deliberation (we didn't spend enough time to tune the model as the goal of this project is the learn multimodality model and compare its performance with single entity models) is shown in the figure below.

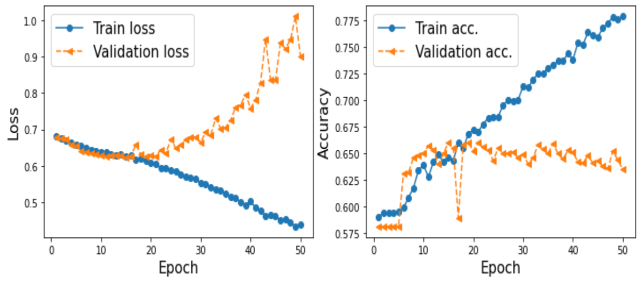


Figure 3: Plot of loss (left) and accuracy (right) of train and validation dataset for 50 epochs.

The loss function plot on the left indicates that the training loss decreases consistently across 50 epochs. However, the validation loss initially decreases for about 20 epochs before starting to rise, indicating overfitting of the model. Similarly, the accuracy plot on the right shows that while the training accuracy keeps increasing, the validation accuracy first rises to around 65% and then fluctuates as the number of epochs increases, confirming the overfitting issue.
Based on the plot above, it is recommended to run the model for 25 epochs. The plots for loss function and accuracy for 25 epochs are shown in the figure below.

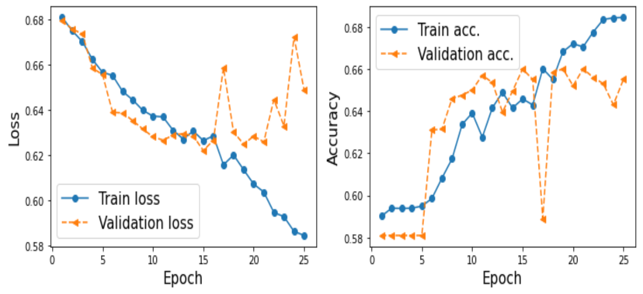


Figure 4: Plot of loss (left) and accuracy (right) of train and validation dataset for 25 epochs.

The train accuracy goes up to 68% whereas validation accuracy fluctuates around 65% after 25 epochs.

6 Multimodal Co-Learning using LSTM and CNN

The Long Short-Term Memory (LSTM) model was used in this study to implement co-learning based on combined text-image data, providing a comparative analysis with SVM and CNN models that rely solely on a single data modality. LSTM is a specialized form of recurrent neural network (RNN) that addresses the challenge of vanishing and exploding gradients that can occur when training conventional RNNs. The LSTM architecture is composed of a memory cell capable of storing and retrieving information over extended time periods, a series of gates regulating the flow of information into and out of the memory cell, and an output layer that produces the final predictions.
The input data for the LSTM consists of both textual and image entities, which are fed into separate models. The textual data is processed by a basic LSTM, while the image data is processed by a CNN. The output of the two models is concatenated, and the output layer is designed for binary classification. Prior to fitting the model, both the text and image data must be preprocessed. The following are the steps for preprocessing and defining the model.

1. **Data Processing:**
    a. Text data: We take **Tokenizer** from keras.preprocessing.text and fit it with text data to produce sequences to produce some numerical score. As each sentence are various level of words, we made them maximum length of 100 by adding 0 as padding. Those padded sequences are the text input of our LSTM model.
    b. Image Data: Just like the CNN models in the previous section, we also used here the same processing. All images are converted with 64*64

gray scale with float value where every pixel values are divided with 255.

c. Label Data: Label data is 0 for negative class and 1 for positive class. We used **to_categorical** to make a 2d array of output values. Each row will have 1 in the class level position. Such as if a class is positive, it's categorical value will be [0, 1].

2. **The model:**
   a. The text data will first pass through embedded layer where input dimension is 2000 and output dimension is 32. Then this output is used for the input of 128 neurons LSTM model. This model will be passed to concertante layer as the text_input model.
   b. The input image shape will be 64×64×1. The first convultion layer will take the first input with 32 filters and kernel size is 3. Then the first max pooling layer is set to 2×2 size. Then it will be used as the input of 2nd convolution layer with increased filters of 64. Pooling layer 2×2 size is also used here. Finally output of the 2nd pooling year is then used as input in the flattened layer where 128 fully connected neurons.
   Then the concatenation layer took place to concatenate two different data models and produce the output layer. All the activations used here are SoftMax.
3. **Model.fit:**
   a. We compile our multimodal that take both the image and text dataset as two input arrays as a tuple and then attached the label with it.
   b. The validation dataset is also attached to the model to validate.

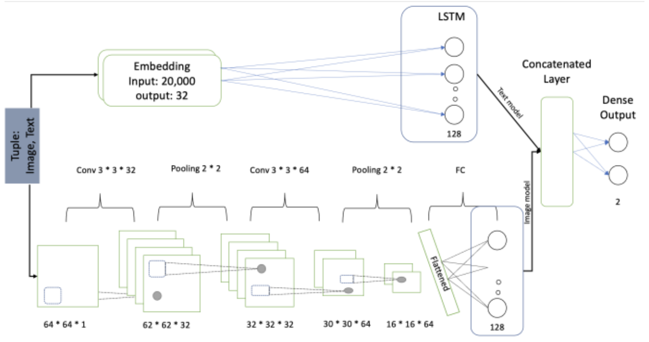


Figure 5: The Architecture of the LSTM model.

Training: The model was set upto 15 epochs, however, it was not converging. We didn't have enough time to tune the parameters. The plots for loss function and accuracy are mentioned below.

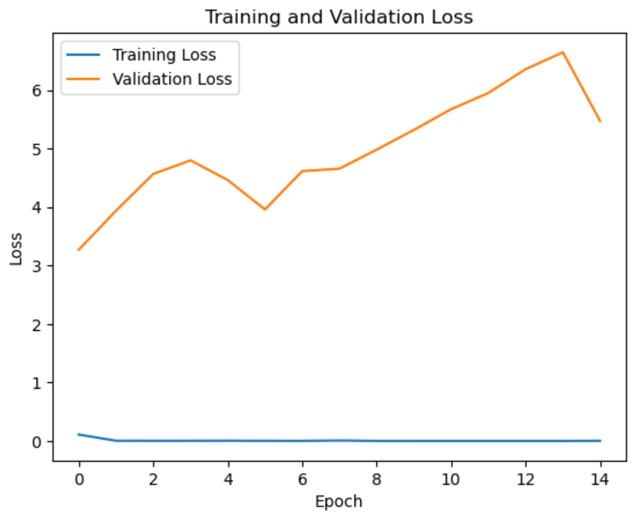


Figure 6: Plot of training and validation loss for 15 epochs.

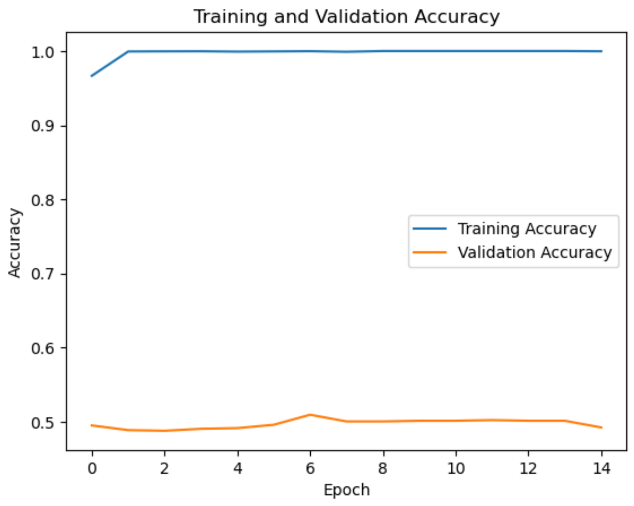


Figure 7: Plot of training and validation accuracy for 15 epochs.

Based on the plots for loss function and accuracy, we can say that the model is overfitting as the training accuracy achieved near 100% but validation accuracy hovers around 50%.

7 Analysis of the Models

The SVM model performed well on the text data since it is a well-suited model for such data. We also tested a single-handed LSTM model on the same test data and it achieved a high accuracy of 99.9%. This could be attributed to the fact that certain keywords such as "boy", "girl", "man", and "woman" could be used to identify whether an image contains a person or not. By keeping track of these keywords, the model was able to perform well.

However, working with image data proved to be challenging, as both the CNN and Multimodal models struggled to achieve good accuracy. The CNN model was designed to process images but could not achieve a good accuracy due to the difficulty in reading the images. The Multimodal model, which combined both text and image data, resulted in lower accuracy than the single-entity CNN model. This could be due to the increased complexity of combining multiple entities in the Multimodal model, making it difficult to learn from the data and achieve good results.

8 Limitations of the Models

1. Due to lack of computation liberty, we worked with a very small amount of data which is not sufficient to conclude a decision. In future more datasets can be added and tested here to observe the models more.

2. As the time was too short, we could not manage enough time to models to tune them and come up with better loss and accuracy. Changing some parameters of the models based on data analysis would be helpful.

9 References

[1] Laion Dataset from Kaggle. Likk: https://www.kaggle.com/datasets/romainbeaumont/laion400m