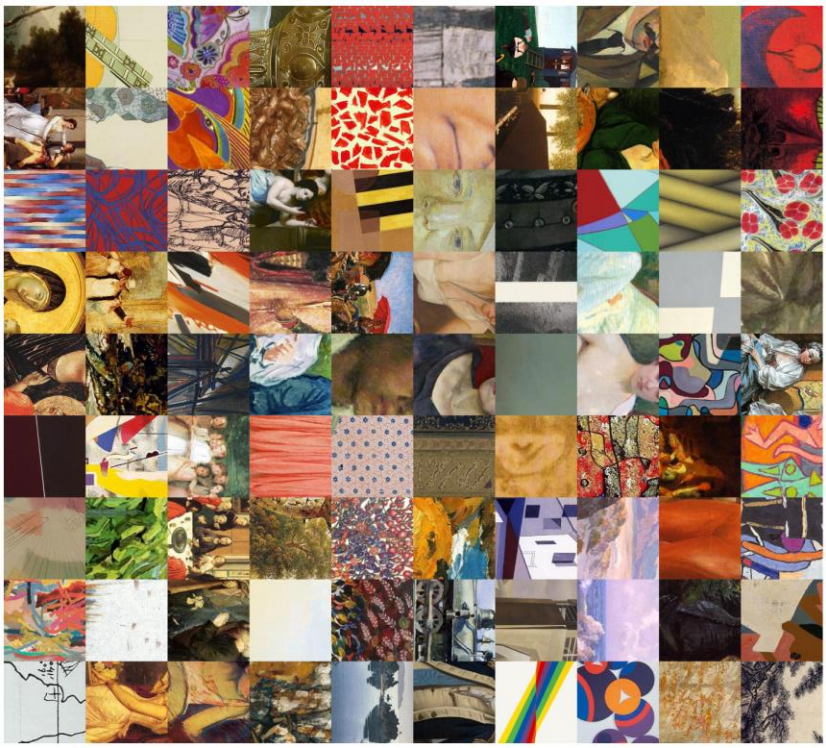
Painter by Numbers

- Murad Khoury: 206538928

- Ram Khoury: 211912993



Instructions for running the code:

- Press on this link: https://drive.google.com/drive/folders/1v5RscLgRjsTAgHna-JE2LOcouZ8rnoVc?usp=drive_link to get the dataset of images on Drive: train_cropped.zip test_cropped.zip, and the best model we got, save them on your personal Drive.
- Download the csv files from the zip file we submitted and save them together.
- Go to the second cell in the project notebook.
- Specify the mode variable to be “pc” if you’re running on PC, and “Drive” if you’re running on Google Colab.
- Change the “path” variable to be the path of your main dir where you saved both of the zipped files of data on Drive - “/content/drive/MyDrive” if the zipped data is saved on MyDrive.
- Change the “csv_data” variable to be the path of the downloaded csv’s from step 2.
- Change the “model_save_path” to be the path where you want to save the model each epoch and load it from in the test phase.
- That’s it, you’re good to go, Run all.

Part 1 – Data Preprocessing:

1) Download train, test images from Kaggle -

First, we downloaded all the data from the website, we made sure to buy a Google Drive cloud so that we could upload the data there.

After uploading all the images to the drive, we wrote a code where we go through all the images, unzip them, and then perform a CenterCrop for all the images to save space, we did this once at the beginning and saved all the images that we CenterCropped (1.5GB instead of 90GB).

Also, when going through the images, we made sure to throw out all the images that had problems (corrupted images).

```
from PIL import Image
from torchvision import transforms
# Path to the folder containing extracted images
images_folder = '/content/extracted_images/test'

# Create a folder for cropped images
cropped_folder = '/content/cropped_images_test'
os.makedirs(cropped_folder, exist_ok=True)

# Define the center crop transformation
center_crop = transforms.CenterCrop((224, 224))  # Adjust dimensions as needed

# Center crop images and save to new folder
for filename in os.listdir(images_folder):
    if filename.endswith('.jpg') or filename.endswith('.png'):
        try:
            image_path = os.path.join(images_folder, filename)
            image = Image.open(image_path)
            # Convert RGBA to RGB if the image has an alpha channel
            if image.mode == 'RGBA' or image.mode == 'P' or image.mode =='LA':
                image = image.convert('RGB')
            cropped_image = center_crop(image)  # Adjust dimensions as needed
            cropped_image.save(os.path.join(cropped_folder, filename))
        except Exception as e:
            print(f"Error processing {filename}: {str(e)}")
            # error_images_train.append(filename)
            error_images_test.append(filename)
            continue  # Skip to the next image
```

2) Building the CSV files of the input –

We built another code, which goes through the CSV file given in Kaggle - all_data_info.csv which contains all the information about the data, for each image: file's name, artist's name, a boolean column of in_train.

First, we made sure to remove all of the corrupted images:

```
all_info = pd.read_csv('all_data_info.csv')
# Removing corrupted images or images that have too large of a size
filenames_to_remove = ['81823.jpg', '91033.jpg', '101947.jpg', '41945.jpg', '95010.jpg', '72255.jpg', '50420.jpg',
                       '95347.jpg', '3917.jpg', '98873.jpg', '79499.jpg', '92899.jpg', '33557.jpg', '82594.jpg',
                       '18649.jpg', '24000.jpg', '100532.jpg', '9989.jpg', '20153.jpg']

all_info = all_info[~all_info['new_filename'].isin(filenames_to_remove)]
```

Second, we removed all the artists that had only 1 image:

```
# Filter artists with at least 2 images
artists_with_2_or_more_images = artist_counts[artist_counts >= 2].index.tolist()
print('Number of artists after cleaning: ', len(artists_with_2_or_more_images))
cleaned_data = all_info[all_info['artist'].isin(artists_with_2_or_more_images)]
```

The next step was to make sure that there was no intersection of artists between train and test, so first we divided the data frame into 2 parts according to the in_train column by True / False and we got the test_DF, train_DF.

Then we found the intersection group of the artists:

```
# Find intersection of artists
common_artists = set(train_artists).intersection(test_artists)
```

We went through the entire intersection group of the artists and for each artist we decided whether to keep him in the train group or test group, receiving the following ratio:

```
Train artists before removing intersections:  1608
Test artists before removing intersections:  1099
Train artists after removing intersections:  1101
Test artists after removing intersections:  597
```

After that, what we did was to limit the number of images for each artist to a maximum of 8, and this is in order not to cause overfit in training for the model that can learn certain patterns only for a certain artist.

We did this using this function:

```
2 usages
def get_n_images_from_artists(df, n=8):
    # Group DataFrame by artist
    grouped_data = df.groupby('artist')
    # Initialize DataFrame to store sampled data
    sampled_data = pd.DataFrame(columns=df.columns)
    for artist, group in grouped_data:
        if len(group) > n:
            sampled_group = group.sample(n=n, random_state=42)  # Randomly sample n images
            sampled_data = pd.concat([sampled_data, sampled_group])
        else:
            sampled_data = pd.concat([sampled_data, group])
    return sampled_data
```

Getting the clean data frames as csv's:

```
train_data_DF.to_csv( path_or_buf: 'df_train_clean.csv', index=False)
test_data_DF.to_csv( path_or_buf: 'df_test_clean.csv', index=False)
```

The last step was building triplets because we used the triplet loss function, we built a function that with (df, unique_artists) arguments and builds the triplets in the following way:

Go through each artist, build all of the possible pairs of images in the order of (anchor, positive) and then choose another artist randomly from the unique_artist group and randomly choose on of his images as the negative sample, so we can get triplets of (anchor, positive, negative).

We did this for both train_df and test_df and got 2 csv files of triplets (train, test).

The last step we wanted to do was to ensure that there would also be no intersection between the artists of the Train and the Validation, therefore we built a function that got the train_triplets.csv as an input file and split it into 2 csv s, one for train and one for validation without intersection.

And at the end of this step, we received 5 csv files which are the input of our project:

- **df_train_clean.csv**
- **df_test_clean.csv**
- **train_triplets_with_labels_new.csv:** the training triplets
- **val_triplets_with_labels.csv:** the validation triplets
- **test_triplets_with_labels.csv:** the test triplets

And this is how we basically built all our input files for the project of classifying image pairs by artists.

Part 2 – Dataset Class:

Using the torch.utils.data's Dataset class we built a class named PainterByNumbersDataset, we used this class to access to our data in the best most efficient way possible in terms of convenience, resource utilization, code execution time, and various other factors.

This class works as follows:

Upon initialization, the class receives the CSV file containing the triplets and the path to the location of the images in memory.

The class is designed to load images into memory (RAM) only when the `__getitem__(self, idx)` method is called, or in other words trying to get an item for the dataset using indexing This lazy loading approach ensures that we don't unnecessarily occupy memory with images that we might not use during training or evaluation. Instead, images are loaded dynamically as needed, which can be particularly beneficial when dealing with large datasets like ours that cannot fit entirely into memory.

Furthermore, this approach allows us to apply transformations to the data on-the-fly. By defining transformations within the dataset class and applying them directly within the `__getitem__(self, idx)`method, we can perform data augmentation and normalization seamlessly without needing to pre-process the entire dataset beforehand. This not only saves memory but also allows for greater flexibility in experimenting with different transformation techniques and parameters during training and also getting different results from applying Random transformations on the data each time the data is accessed.

Part 3 – Data Augmentation:

Data augmentation is a technique helpful in a lot of scenarios, one of them is when the model's generalization ability needs improvement and that's why we choose to use it.

It involves generating new training samples by applying a variety of transformations to the existing dataset. These transformations modify the appearance of the data while preserving its content, this way it increases the diversity of the training samples and by that improves the model's performance and prevents overfitting.

The first Data augmentation technique is **using the Dataloader class**:

The data loader class allows you to split your data into batches and by setting the argument `shuffle=True` the data is randomly shuffled before splitting into batches before every epoch this way we avoid the model learning to memorize the order in which the training data is being read.

The second Data augmentation technique is **using transformations**:

Transformations in the training data work to prevent overfitting and help the model's accuracy. by introducing variations in orientation, color, perspective, and texture, the model becomes more robust and capable of generalizing to unseen data (this particularly helps in our case since there is not intersection between the artist in the training, validation or test datasets).

We additionally apply normalization to all of the datasets including validation and test since normalization ensures that the model's embedding production process is stable and efficient.

Here are the transformations that we used or at least used a combination of:

```
RandomVerticalFlip(0.5),
RandomHorizontalFlip(0.5),
RandomRotation(15),
ColorJitter(.5, .3),
RandomPerspective(0.6, 0.2),
GaussianBlur((5, 9), (0.1, 5.)),
RandomAffine((-15,15), (0,0),(0.8, 1.2), 10),
Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
```

(in the notebook there is a plot with each transformation's effect on a random data sample)

Overall, the combination of these transformations enhances the model's ability to learn meaningful features from the data and achieve better performance on unseen images **and most importantly prevent Overfitting.**

Part 4 – Siamese Network:

Siamese Network Architecture:

1. **Base Model:**
    - The Siamese Network uses a pre-trained ResNet34 model from torchvision as its base model. ResNet34 is a convolutional neural network architecture known for its effectiveness in image classification tasks.
2. **Modification of Last Layer:**
    - The last fully connected layer of the ResNet34 model is modified to output 256-dimensional embeddings instead of the original number of output features. This modification is done by replacing the last fully connected layer (self.base_model.fc) with a new linear layer (nn.Linear) with an output size of 256.
3. **Forward Pass:**
    - In the forward method, the Siamese Network takes triplets of images (anchor, positive, negative) as input.
    - Each image in the triplet is passed through the base model to extract embeddings.
    - The embeddings for the anchor, positive, and negative images are obtained as out1, out2, and out3, respectively.
    - These embeddings are then returned as the output of the forward pass.

Triplet Loss Function:

1. **Definition**:
    - The TripletLoss class implements the triplet loss function, which is used to train the Siamese Network.
    - Triplet loss aims to minimize the distance between the anchor and positive examples while maximizing the distance between the anchor and negative examples by a margin.
2. **Forward Method**:
    - In the forward method, the distances between the anchor and positive embeddings (distance_positive) and between the anchor and negative embeddings (distance_negative) are computed using the pairwise Euclidean distance (F.pairwise_distance).
    - The loss is then calculated as the mean of the differences between distance_positive and distance_negative, adjusted by a margin and clamped at a minimum of zero.

Initialization and Training Setup:

Initialization:

- The ResNet34 model is initialized with pre-trained weights (**pretrained=True**).
- The Siamese Network is initialized with modified ResNet34 base model.
- The Triplet Loss function is initialized with a specified margin value.
- The Adam optimizer is used to optimize the parameters of the Siamese Network, with a specified learning rate.

This Siamese Network architecture and training setup are designed for learning embeddings that capture the similarity between images in a triplet-based fashion, where each triplet consists of an anchor, positive, and negative image. The network aims to minimize the distance between similar images (anchor and positive) while maximizing the distance between dissimilar images (anchor and negative) by learning discriminative embeddings.

Part 5 – Parameters:

- Number of epochs: 14.
- Train dataset size: 23,755.
- Validation dataset size: 5,996.
- Test dataset size: 6,135.
- Batch size: 64.
- Margin: 5.
- Learning rate: 0.000006.

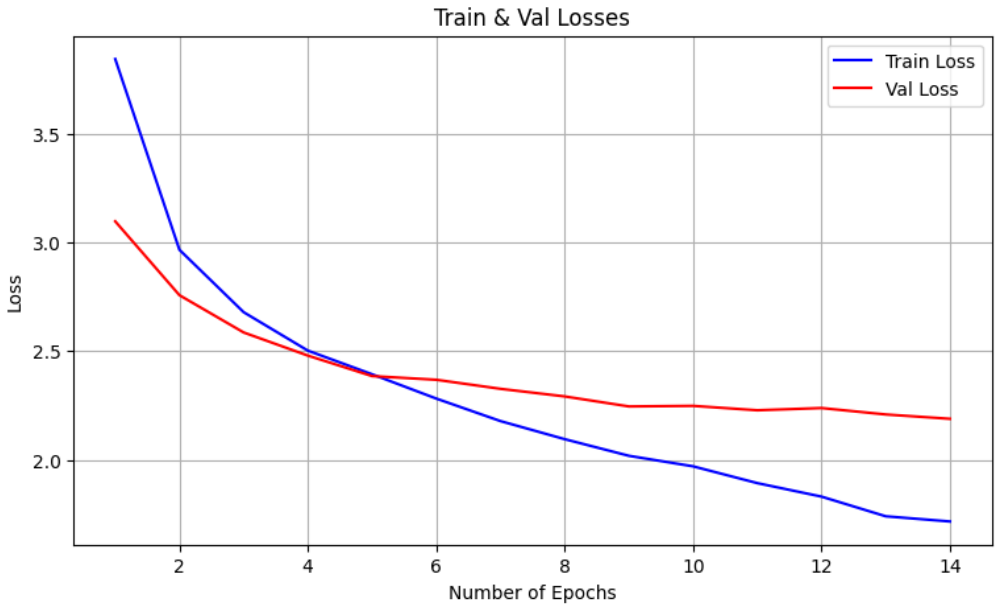
Part 6 – train and validation:

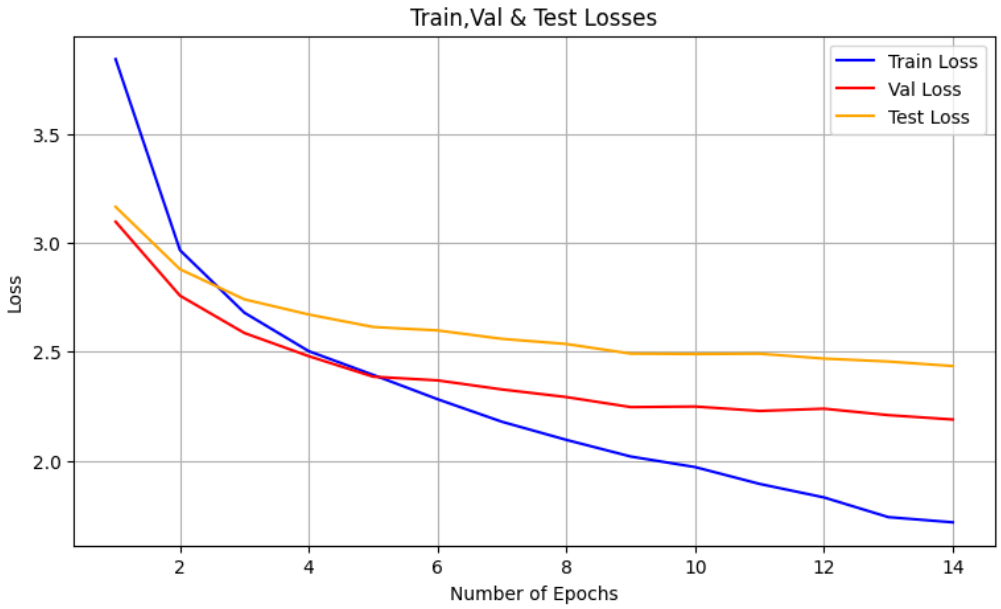
- **Training Loop:**
  - Iterates over epochs.
  - Divides each epoch into training and validation phases.
- **Phase Setup and Data Loading:**
  - Sets model mode: training or validation.
  - Loads batches of data from respective dataloaders.
- **Forward Pass and Backpropagation:**
  - Calculates embeddings for anchor, positive, and negative images.
  - Computes triplet loss and updates model parameters (if in training phase).
- **Loss and Accuracy:**
  - Calculates loss and accuracy for each phase.
- **Model Saving and Testing:**
  - Saves model after each epoch.
  - Tests model on test dataset.
  - Implements early stopping if specified conditions met (loss going up).
- **Best Model Selection:**
  - Tracks best model based on highest test accuracy.
- **Return Values:**
  - Returns best model and latest model after training.

Part 7 – test:

- **Model Evaluation**:
  - Sets model to evaluation mode.
- **Iterating Over Test Data**:
  - Iterates over test data batches.
- **Model Forward Pass**:
  - Computes embeddings for anchor, positive, and negative images.
  - Calculates triplet loss.
- **Loss and Accuracy Calculation**:
  - Computes loss and accuracy for the test phase.
- **Printing Results**:
  - Prints loss and accuracy for the test phase.
- **Return Values**:
  - Returns epoch accuracy and epoch loss for analysis.

Part 8 – Plots:





Part 9 – Accuracy Results:

Train Accuracy - 86.1587%

Validation Accuracy - 81.3209%

Test Accuracy - 79.7555%.