Objective:

The objective of this project is to design and implement a system capable of generating realistic images from sketches utilizing a Conditional Generative Adversarial Network (CGAN). The system will take a sketch as input along with a corresponding label and produce high-quality images that accurately represent the characteristics associated with the provided label.

1. Key Components:

- **Conditional Generative Adversarial Network (CGAN):** Implementation of a CGAN architecture that can effectively leverage both the sketch input and label information to generate realistic images. The generator network will transform input sketches into detailed images, while the discriminator network will distinguish between real and generated images, conditioned on the input label.

- **Label Conditioning:** Ensure that the generated images capture the specific attributes and features corresponding to the provided label. The label information will guide the generation process, enabling the model to produce images with desired characteristics such as object categories, shapes, textures, and colors.

- **Sketch-to-Image Translation:** Develop mechanisms within the generator network to translate input sketches into visually coherent images. Explore techniques such as conditional image generation, feature alignment, and attention mechanisms to enhance the fidelity and realism of generated images.

2. Dataset Description:

The dataset required for the development of the conditional Generative Adversarial Network (CGAN) for realistic image generation from sketches is sourced from the Provided link:
"https://drive.google.com/drive/folders/1vYv5SmA6nu4PKB_5PIk6FoTCtyEWztKP?usp=sharing"

Below is a detailed breakdown of the dataset components and their respective roles in the project.

2.1. Dataset Components:

2.1.1. Training Dataset:

a) Train Data (Training Images):

- **Description:** This subset includes actual images that are used during the training phase of the CGAN model.

- **Purpose:** These images serve as the ground truth or reference for the model, helping it learn to generate photorealistic images from sketches.

b) Train Labels (Training Labels):

- **Description:** Accompanying labels for the training images.
- **Purpose:** These labels provide categorical information associated with each training image, which is crucial for conditioning the GAN, ensuring that the generated images align with specified attributes.

c) Train Sketch (Unpaired Training Sketches):

- **Description:** This subset consists of sketches that correspond to the training images but are not paired directly with any specific training image.
- **Purpose:** These sketches are used to train the generator component of the CGAN. The challenge for the model is to fill in details and textures to convert these sketches into realistic images that correspond to the categorical information provided by the train labels.

2.1.2. Testing Dataset:

a) Test Data (Test Images):

- **Description:** This set includes images that are used for evaluating the performance of the trained CGAN model.
- **Purpose:** These images are used as a benchmark to assess the quality and accuracy of the images generated by the model during the testing phase.

b) Test Labels (Test Labels):

- **Description:** Labels corresponding to the test images.
- **Purpose:** These labels are used to verify if the generated images during the test phase correctly represent the characteristics associated with these labels, ensuring label consistency and accuracy in generation.

c) Test Sketch (Unpaired Test Sketches):

- **Description:** Sketches that are intended for use during the testing phase.
- **Purpose:** These sketches are used to test the model's ability to generate images from new, unseen sketches, assessing the model's generalization capabilities over unpaired input.

3. Experiments:

Experiment 3.1:

- Implement a conditional GAN architecture conditioned on class labels to synthesize realistic images by taking unpaired sketches as input.

- Evaluate the model's performance in terms of image quality, diversity, and realism. Analyze the effectiveness of the approach and identify areas for improvement. You need to report Frechet Inception Distance (FID) and Inception Score (IS).

3.1. Methodology:

3.1.1. Importing the Packages:

This initial setup is crucial as it prepares the environment for all subsequent data processing and analysis tasks. Begin by importing necessary packages like panadas, numpy, torch, torch.nn, sklearn etc.

3.1.2. Data Loading and Pre-processing:

The data loading and pre-processing step is crucial for preparing the dataset for input into our machine learning models. This section outlines the processes involved in handling training and testing images and labels, as well as applying necessary transformations to standardize the dataset.

3.1.2.1. Data Loading:

- **Data Paths Definition:** Paths for training and testing data, sketches, and labels are defined to organize and streamline the access process during loading:

- **Label Loading:** Labels for both training and testing data are loaded from CSV files into pandas DataFrames.

- **Image Transformations:** To ensure consistency and improve model performance, all images undergo a series of transformations:

  - Resizing to a uniform dimension (256x256 pixels)
  - Conversion of image data into PyTorch tensors
  - Normalization to scale the pixel values to the range [-1, 1]

- To normalize the labels to the range of 0 to 6, the following line of code was applied: `train_labels = torch.argmax(train_labels, dim=1)`
  This line converts the one-hot encoded labels into their corresponding class indices, effectively transforming them into the range of 0 to 6.

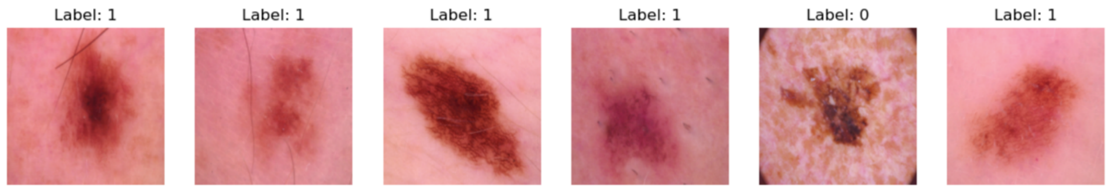
- Same process for test dataset.

3.1.2.2. Data Pre-processing:

- **Pre-processing Images:** A custom function **preprocess_isic_images** is employed to load and transform the images based on the paths and labels provided. Processed images and labels are then loaded into tensors.

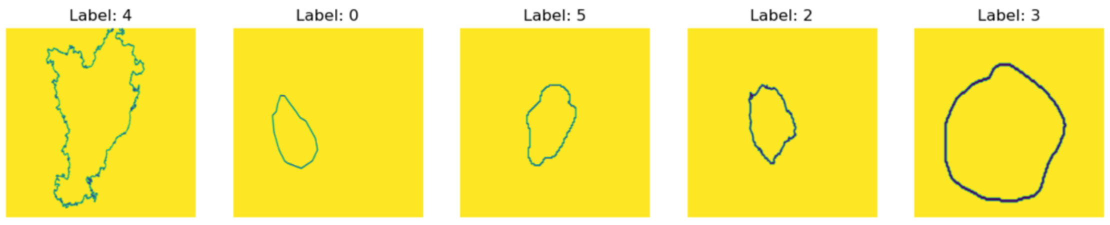
- This data is then printed to verify dimensions and used for visualization to confirm the successful loading and processing of images.
  - `torch.Size([10015, 3, 256, 256])`
  - `torch.Size([10015, 7])`

```
# Plot some of the images
plot_images(train_images, train_labels)
```



- **Sketch Image Processing:** Similar to the ISIC images, sketches are processed using a slightly modified transformation pipeline suitable for grayscale images. The transformations and the pre-processing function are outlined, which load and transform sketches into tensors.

```
plot_images(train_sketches, sketch_labels, num_images=5)
```



3.1.3. Conditional Generator and Discriminator Architecture

3.1.3.1. Conditional Generator Implementation:

The conditional generator is a key component of conditional generative models such as Conditional Generative Adversarial Networks (CGANs). This section details the implementation of the conditional generator, which takes both image data and class labels as input to generate realistic images corresponding to the specified classes.

Components of the Conditional Generator:

1. **Label Embedding Module (LabelEmbedding):**

   - This module embeds the class labels into a continuous representation.

- It utilizes an embedding layer to map each class label to a dense vector representation.
- The embedded vectors are reshaped to match the input image size.

```
            # Shape: [batch_size, 1, 256, 256]
```

2. **Generator Architecture Overview:** The generator follows a modified U-Net architecture, which comprises both an encoder (down-sampling pathway) and a decoder (up-sampling pathway) to transform input data into high-resolution output images.

   a) **Initial Processing:**

      - **Input Size:** The input size for each image is 256x256 pixels.
      - **Input Channels:** Initially, the input consists of the image data (1 channel) concatenated with the label embeddings (1 channel).
      - **Processing:** This concatenated input undergoes initial processing through a convolutional layer (conv1) followed by batch normalization and ReLU activation.
      - **Output Size:** After initial processing, the output feature maps have 64 channels with dimensions reduced due to convolutional operations.

   b) **Down-sampling Pathway (Encoder):**

      - **Layers:** The down-sampling pathway consists of 7 down-sampling layers (down_stack) to progressively reduce the spatial dimensions of the feature maps while increasing the number of channels to capture abstract features.

      - **Each Down-sampling Layer:**

        - **Convolutional Layer:** Each layer performs a 2D convolution operation with a kernel size of 4x4, a stride of 2, and a padding of 1 to down-sample the feature maps.
        - **Activation Function:** Leaky ReLU activation function (with a negative slope of 0.2) introduces non-linearity.
        - **Batch Normalization:** Optional batch normalization is applied to stabilize and accelerate training.

      - **Output Size:** After each down-sampling operation, the spatial dimensions of the feature maps are halved while the number of channels increases up to 512.

   c) **Up-sampling Pathway (Decoder):**

      - **Layers:** The up-sampling pathway consists of 7 up-sampling layers (up_stack) to progressively up-sample the low-resolution feature maps back to the original input size.

      - **Each Up-sampling Layer:**

- **Transposed Convolutional Layer:** Each layer performs a transposed convolution operation with a kernel size of 4x4, a stride of 2, and a padding of 1 to up-sample the feature maps.
- **Activation Function:** ReLU activation function introduces non-linearity.
- **Optional Dropout:** Some layers include dropout regularization with a dropout rate of 0.5 to prevent overfitting.

- **Output Size:** After each up-sampling operation, the spatial dimensions of the feature maps are doubled while the number of channels decreases gradually back to 3 channels (for RGB images).

d) Final Output:

- **Output Layer:** The final output consists of two consecutive transposed convolutional layers (last_1 and last_2) to generate the output image.

- **Output Activation:** The output is passed through a hyperbolic tangent (tanh) activation function to ensure pixel values are in the range [-1, 1], suitable for real images.

Summary:

- The generator consists of a total of 14 convolutional layers (7 down-sampling and 7 up-sampling layers).

- It takes input images of size 1x256x256 pixels and class labels as input.

- The output consists of high-resolution RGB images (3x256x256) with pixel values in the range [-1, 1].

3.1.3.2. Conditional Discriminator Implementation:

This section outlines the implementation of the conditional discriminator, which evaluates the authenticity of generated images while considering class labels.

Components of the Conditional Discriminator:

1. **Label Embedding Module (LabelEmbedding):** Similar to the generator, the discriminator utilizes an embedding layer to map class labels to continuous representations.

2. **Discriminator Architecture Overview:** The discriminator is responsible for distinguishing between real and generated images. It analyses input images and class labels to produce a probability score indicating the authenticity of each image.

   a) Initial Processing:

- **Input Size:** The input size for each image is 256x256 pixels.
- **Input Channels:** Initially, the input consists of the image data (3 channel) concatenated with the label embeddings (1 channel).
- **Processing:** The concatenated input undergoes initial processing through a down-sampling layer (down1) to reduce the spatial dimensions of the feature maps and extract basic features.

b) Down-sampling Layers:

- **Layers:** The discriminator utilizes three down-sampling layers (down1, down2, down3) to progressively reduce the spatial dimensions of the input feature maps and extract hierarchical features.
- **Each Down-sampling Layer:** Same as generator
- **Output Size:** After the final down-sampling layer (down3), the spatial dimensions of the feature maps are reduced, while the number of channels increases up to 256.

c) Convolutional Layers:

- **Layers:** Following the down-sampling layers, the discriminator applies a convolutional layer (conv) to further process the feature maps and make authenticity predictions.
- **Processing:** The convolutional layer performs a 2D convolution operation with a kernel size of 4x4 to aggregate features across spatial dimensions.
- **Normalization:** Batch normalization is applied to stabilize and accelerate training.
- **Activation Function:** Leaky ReLU activation function introduces non-linearity.

d) Output Layer:

- **Output Size:** The output of the discriminator is a single scalar value representing the probability that the input image is real.
- **Convolutional Layer:** The output layer consists of a convolutional layer (last) with a kernel size of 4x4 to aggregate features.
- **Output Activation:** No activation function is applied to the output, as the discriminator aims to produce a raw probability score.

Summary:

- The discriminator comprises a total of 4 convolutional layers (3 down-sampling layers and 1 output layer).
- It takes input images of size 256x256 pixels and class labels as input.
- The output is a single scalar value representing the probability that the input image is real.

3.1.4. Loss Function:

The training process relies on optimizing two key loss functions: the generator loss and the discriminator loss. These loss functions play a crucial role in guiding the training process and ensuring the convergence of the model.

3.1.4.1. Generator Loss:

- The generator loss is a sigmoid cross-entropy loss of the generated images and an **array of ones**.
- L1 loss, which is a MAE (mean absolute error) between the generated image and the target image.
- This allows the generated image to become structurally similar to the target image.
- The formula to calculate the total generator loss is:

  ```
  gan_loss + LAMBDA * l1_loss, where LAMBDA = 100.
  ```

  This Labmda value was decided by the authors of the paper. (The pix2pix paper)

3.1.4.2. Discriminator Loss:

- The discriminator_loss function takes 2 inputs: real images and generated images.
- real_loss is a sigmoid cross-entropy loss of the real images and an array of ones(since these are the real images).
- generated_loss is a sigmoid cross-entropy loss of the generated images and an array of zeros (since these are the fake images).
- The total_loss is the sum of real_loss and generated_loss.

3.1.5. Training the Conditional GAN:

The training of the Conditional Generative Adversarial Network (cGAN) involves optimizing the generator and discriminator networks iteratively to generate realistic images conditioned on input sketches. The training loop consists of several key steps, including data loading, network forward passes, loss computation, and gradient updates.

3.1.5.1. Initialization and Setup:

- **WandB Integration:** The training process begins with initializing the WandB run for experiment tracking and visualization.

- **Hyperparameters:** Hyperparameters such as the
  # epochs =100,
  lambda value =100, and
  batch size- 64 are defined.
- **Data Preparation:** The training data, including images and corresponding labels and sketches and corresponding labels, is loaded into a TensorDataset and DataLoader for efficient batch processing.

3.1.5.2. Discriminator Training:

- **Clearing Gradients:** Gradients of the discriminator model (discriminator) are reset to zero using zero_grad() to avoid accumulating gradients from previous iterations.

- **Generating Fake Images:** The generator (generator) is employed to produce fake images from input sketches (sketch) and their corresponding labels (skech_lab). These generated images are stored in the fake_images tensor.

- **Discriminator Predictions:** The discriminator evaluates both real and fake images. For real images (real_images), their labels (real_labels) are used for conditioning, and the predictions are computed using the discriminator model.

- **Calculating Discriminator Loss:** The discriminator loss (disc_loss) is calculated based on the predictions for real and fake images. This loss quantifies the ability of the discriminator to distinguish between real and fake images.

- **Updating Discriminator Weights:** Gradients are computed for the discriminator loss, and the optimizer (optimizer_D) updates the discriminator's weights (optimizer_D.step()).

3.1.5.3. Generator Training:

- **Clearing Gradients:** Like the discriminator, gradients of the generator model (generator) are cleared to prevent accumulation.

- **Generating Fake Images:** The generator produces fake images (fake_images) from the input sketches and labels, like the discriminator training step.

- **Discriminator Predictions:** The discriminator evaluates the generated fake images to assess their realism.

- **Calculating Generator Loss:** The generator loss is computed, which comprises the total_gen_loss, gen_gan_loss, and the gen_l1_loss. Throughout training, the gen_l1_loss should decrease steadily.

- **Updating Generator Weights:** Gradients of the generator loss are computed and used to update the generator's weights (optimizer_G.step()).

3.1.5.4. Things to look for:

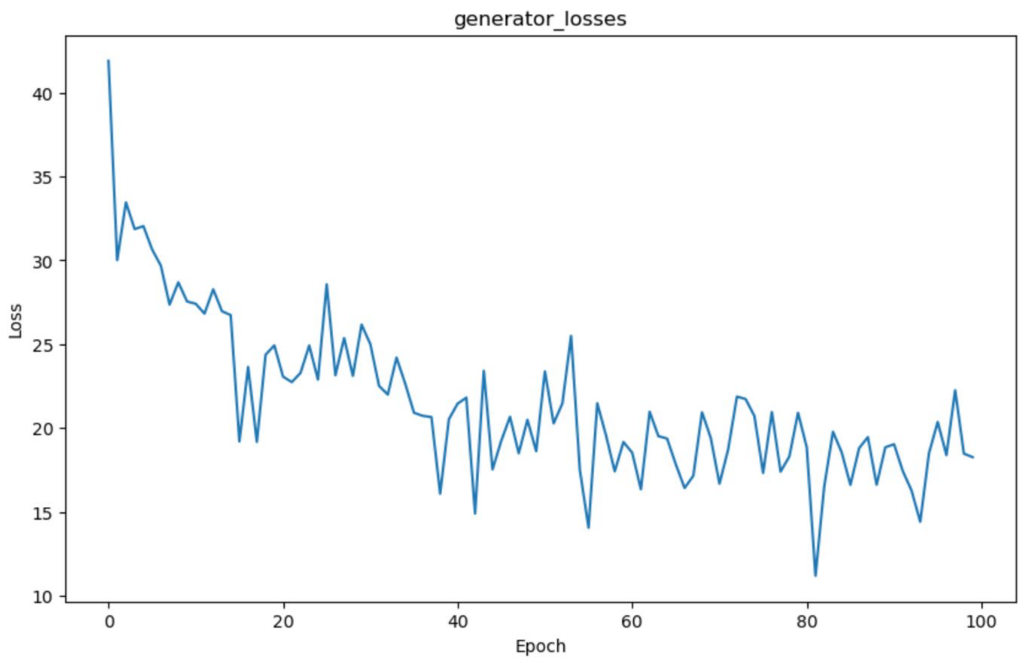
- If either the gen_gan_loss or the disc_loss gets very low, it's an indicator that this model is dominating the other, and you are not successfully training the combined model.

- The ideal reference point for these losses (gen_gan_loss or the disc_loss) is the value of log(2), approximately 0.69, indicating a perplexity of 2. This signifies that the discriminator is equally uncertain about both real and generated images.

- A disc_loss below 0.69 indicates the discriminator performs better than random on the combined dataset of real and generated images.

- A gen_gan_loss below 0.69 suggests the generator effectively fools the discriminator better than random.

- Throughout training, the gen_l1_loss should decrease steadily.



Generator Loss After 100 epoch Training: `11.265947341918945`

3.1.6. Model Testing:

In the testing phase of the (cGAN), we evaluate the trained generator model on unseen test data to assess its performance in generating realistic images from sketches. The following steps outline the testing procedure:

- **Set Model to Evaluation Mode:** Before testing, we switch the generator model to evaluation mode using generator.eval(). This ensures that any layers like batch normalization or dropout operate in evaluation mode rather than training mode.

- **Data Loading and Testing Loop:**

    - Test data, consisting of sketches, images and corresponding labels, is loaded using a data loader (test_loader).

        - `test_images: torch.Size([1512, 3, 256, 256])`
        - `test_sketch: torch.Size([1000, 1, 256, 256])`

    - The testing loop iterates over batches of test data.

    - We have generated the Images corresponding to test images dataset using its label and unpaired sketches dataset.

    - `Shape of generated images: torch.Size([1512, 3, 256, 256])`

- **Compute Metrics:**

    - After generating fake images, we compute evaluation metrics to assess the quality of the generated images.

    - Two evaluation metrics are computed: the Fréchet Inception Distance (FID) and the Inception Score (IS).

    - **FID measures** the similarity between the distribution of real images and generated images in feature space.

    - **Inception Score (IS) Evaluation:**

        - Utilized a pre-trained Inception V3 model to extract activations from generated images.
        - Computed the Inception Score by measuring the KL divergence between conditional and marginal distributions of class labels.
        - Calculated the mean and standard deviation of KL divergences over multiple splits.
        - Generated images obtained a mean Inception Score with standard deviation.
        - Higher scores indicate better quality and diversity in generated images.

- **Visualize Generated Images:**
    - The generated images are visualized to provide a qualitative assessment of the generator's performance.
    - A subset of generated images along with their corresponding labels is displayed for inspection.
- **Reporting Results:**
    - The computed FID and IS scores are reported as quantitative measures of the generator's performance.
    - These scores provide insights into the fidelity and diversity of the generated images compared to the real data distribution.

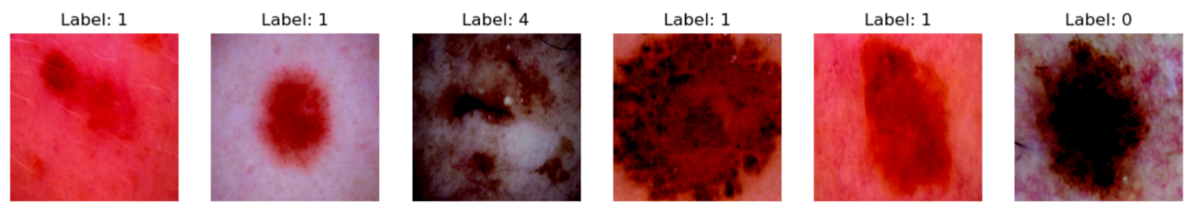
3.1.7. **Results of Conditional GAN:**

After testing the Conditional Generative Adversarial Network (cGAN), we obtained the following results:

Generated Images Visualization:
(Example of generated images along with their corresponding labels)

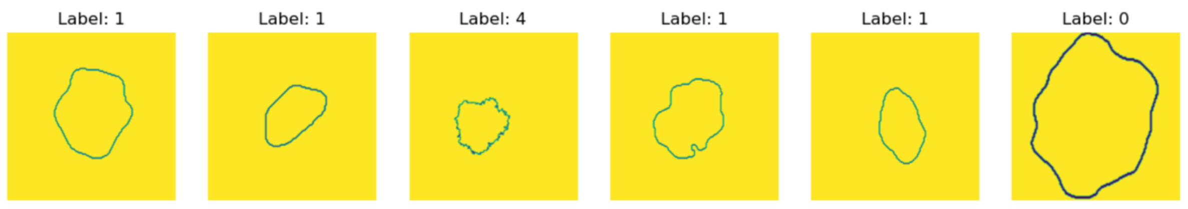
1. Test Images with its labels:

```
Test ISIC images Loaded
```



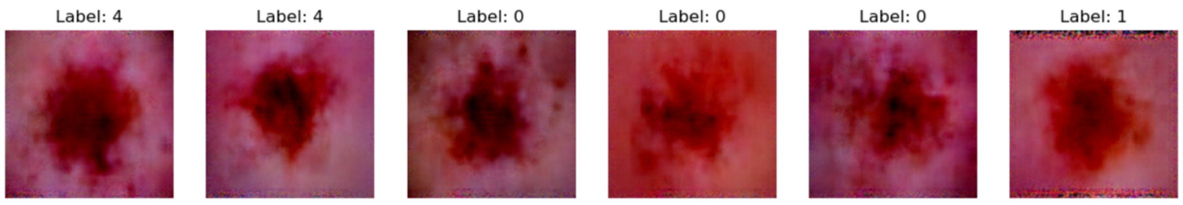
2. Test Sketches corresponding to test images and its label: (Given Input to model):

```
plot_images(test_sketches, test_labels, num_images=6)
```



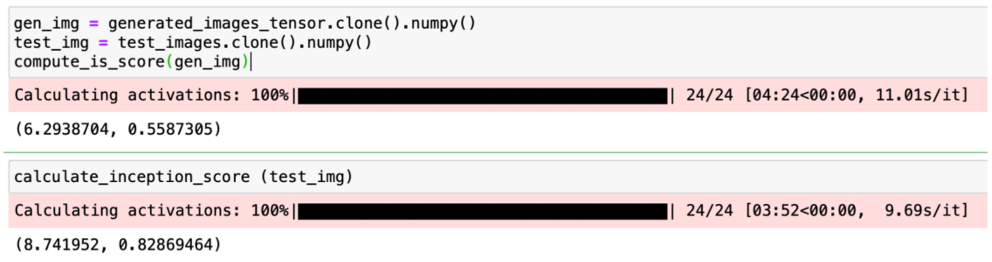
3. Generated Images Corresponding to Image, sketches its corresponding labels:

```
visualize_images(generated_images_tensor, test_labels)
```



4. **Scores:**

- Fréchet Inception Distance (FID) Score: `0.0294`
- Inception Score (IS):
    - For Generated Images: `6.2938704`
    - For Test Images: `8.741952`



```
gen_img = generated_images_tensor.clone().numpy()
test_img = test_images.clone().numpy()
compute_is_score(gen_img)
Calculating activations: 100%|████████████████████████████████████████| 24/24 [04:24<00:00, 11.01s/it]
(6.2938704, 0.5587305)
```

```
calculate_inception_score (test_img)
Calculating activations: 100%|████████████████████████████████████████| 24/24 [03:52<00:00,  9.69s/it]
(8.741952, 0.82869464)
```

Experiment 3.2:

- Evaluate the generated images (at least 10 for each class) by considering them as test images and report the accuracy. Compare the accuracy with original test set images of the dataset. For this, you can train any classifier of your choice with the training data available to you.

3.2. Methodology:

We evaluated the performance of a pre-trained ResNet18 classifier on both the original test set images and the generated images produced by a Conditional Generative Adversarial Network (CGAN). The objective is to compare the classification accuracy between the two sets of images and analyze any differences observed.

3.2.1. Data Preparation

- **Original Test Set:** We start by loading the original test set images and their corresponding labels from the dataset. These images represent real-world examples from various classes.

- **Generated Images:** We generate synthetic images using a Conditional Generative Adversarial Network (CGAN). These images are produced from sketches using a trained generator model. Each generated image is associated with a label indicating its class.

3.2.2. Classifier Model:

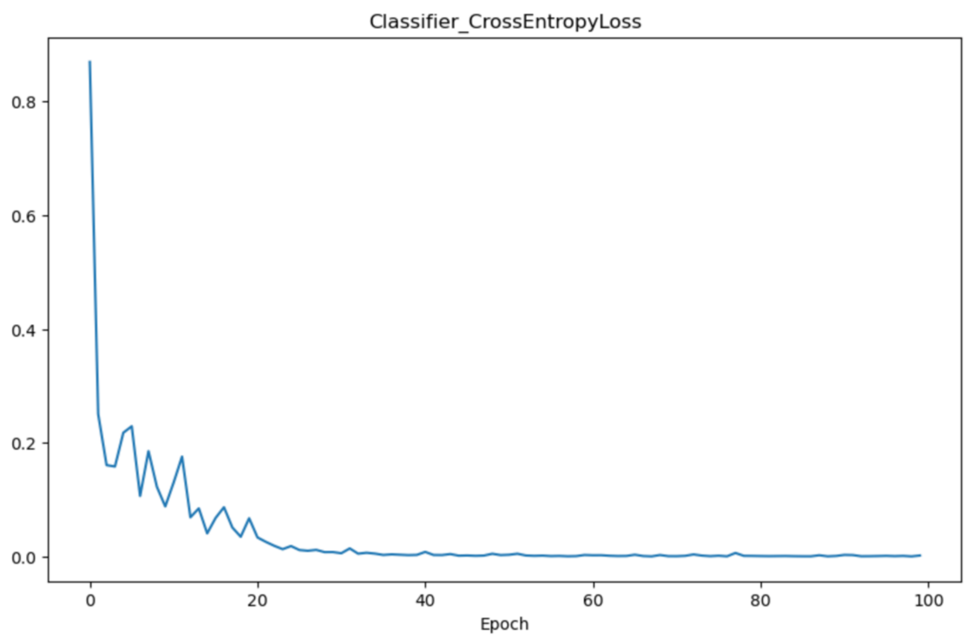
- **Selection:** We choose a pre-trained ResNet18 classifier as the model for image classification. The ResNet18 architecture is known for its effectiveness in computer vision tasks and has been pretrained on the ImageNet dataset.

- **Modification:** We modify the classifier's final fully connected layer to output predictions for the number of classes present in our dataset (7 classes).

- **Training Dataset:** For Training this model we have whole training dataset and its corresponding labels. And trained the model for `100 # epoch`.

  - `torch.Size([10015, 3, 256, 256])`
  - `torch.Size([10015, 1])`

- **Loss Function:** We have used Cross-Entropy Loss, this loss function is utilized for both training the CGAN and evaluating the classifier. Cross-entropy loss is commonly employed in multi-class classification tasks as it measures the difference between predicted probabilities and true label distributions.



Classifier Model Loss After 100 epoch of Training: `0.022790934890508650`

3.2.3. Evaluation Procedure

3.2.3.1. Original Test Set Evaluation

- We evaluate the classifier's performance on the original test set images.
- Each image is passed through the classifier, and the predicted class is compared with the ground truth label.
- The classification accuracy is calculated as the percentage of correctly classified images over the total number of images in the test set.

- Accuracy on Original Test Set: **`0.7976190476190477`**

3.2.3.2. Generated Images Evaluation:

- We assess the classifier's ability to classify synthetic images generated by the CGAN.
- For each class, we select at least 10 generated images to ensure a representative evaluation.
- Like the original test set evaluation, each generated image is classified, and the accuracy is calculated.

- Accuracy on Generated Images: **`0.42857142857142855`**

3.2.4. Comparison and Analysis

- The accuracy on the original test set provides a baseline performance of the classifier on real-world images.

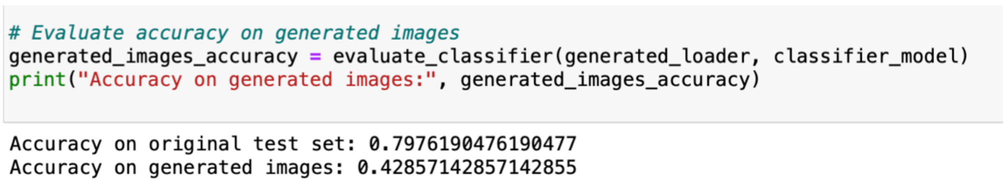
- By comparing this accuracy with the accuracy on the generated images, we can assess the generalization capability of the classifier to synthetic data.

- **Scores:**

  - Fréchet Inception Distance (FID) Score: **`0.0294`**
  - Inception Score (IS):
    - For Generated Images: **`6.2938704`**
    - For Test Images: **`8.741952`**

- **Accuracy:**

  - Accuracy on Original Test Set: **0.79762**
  - Accuracy on Generated Images: **`0.42856`**



```
# Evaluate accuracy on generated images
generated_images_accuracy = evaluate_classifier(generated_loader, classifier_model)
print("Accuracy on generated images:", generated_images_accuracy)
```

```
Accuracy on original test set: 0.7976190476190477
Accuracy on generated images: 0.42857142857142855
```