Model description :

- Conditional Generative Adversarial Networks (cGAN) is a variant of Generative Adversarial Networks (GANs) where both the generator and the discriminator are conditioned on additional information. In a cGAN, the generator tries to generate data samples based on both a random noise vector and a conditioning input.

Main Required Steps :

1. Choose a Dataset and write the Dataset class to transform the images to grayscale.
2. Implement the UNET Autoencoder.(generator)
3. Implement the patch GAN.(discriminator)
4. Setup the losses and train the system.
5. Improve the model with some of the suggestions or propose something yourself.

The paper uses the whole ImageNet dataset (with 1.3 million images!) but here we are using only 5000 images.

1.1- Loading Image Paths:

```
!pip install fastai==2.4
```

```
Collecting fastai==2.4
  Downloading fastai-2.4-py3-none-any.whl (187 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 187.9/187.9 kB 4.1 MB/s eta 0:00:00
Requirement already satisfied: pip in /usr/local/lib/python3.10/dist-packages (from fastai==2.4) (23.1.2)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from fastai==2.4) (23.2)
Collecting fastcore<1.4,>=1.3.8 (from fastai==2.4)
  Downloading fastcore-1.3.29-py3-none-any.whl (55 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.0/56.0 kB 5.1 MB/s eta 0:00:00
Requirement already satisfied: torchvision>=0.8.2 in /usr/local/lib/python3.10/dist-packages (from fastai==2.4) (0.16.0+cu118)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (from fastai==2.4) (3.7.1)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (from fastai==2.4) (1.5.3)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from fastai==2.4) (2.31.0)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.10/dist-packages (from fastai==2.4) (6.0.1)
Requirement already satisfied: fastprogress>=0.2.4 in /usr/local/lib/python3.10/dist-packages (from fastai==2.4) (1.0.3)
Requirement already satisfied: pillow>6.0.0 in /usr/local/lib/python3.10/dist-packages (from fastai==2.4) (9.4.0)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packages (from fastai==2.4) (1.2.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from fastai==2.4) (1.11.3)
Requirement already satisfied: spacy<4 in /usr/local/lib/python3.10/dist-packages (from fastai==2.4) (3.6.1)
INFO: pip is looking at multiple versions of fastai to determine which version is compatible with other requirements. This could tak
ERROR: Could not find a version that satisfies the requirement torch<1.10,>=1.7.0 (from fastai) (from versions: 1.11.0, 1.12.0, 1.12
ERROR: No matching distribution found for torch<1.10,>=1.7.0
```

```
import os
import glob
import time
import numpy as np
from PIL import Image
from pathlib import Path
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt
from skimage.color import rgb2lab, lab2rgb
import pandas as pd

import torch
from torch import nn, optim
from torchvision import transforms
from torchvision.utils import make_grid
from torch.utils.data import Dataset, DataLoader
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
use_colab = None
```

```
device
```

```
device(type='cpu')
```

```
import fastai
fastai.__version__
```

```
'2.7.13'
```

```
import torch
torch.__version__
```

```
'2.1.0+cu118'
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
!cp -R /content/drive/MyDrive/DNN_final_project/5k /content/
```

```
^C
```

Downloading data directly if we want to have more data with kaggle API Imagenetmini.In this project we are using 5000 data becuase our recources are limited.

```
#! pip install kaggle
! mkdir ~/.kaggle
! cp kaggle.json ~/.kaggle/
! chmod 600 ~/.kaggle/kaggle.json
```

```
! kaggle datasets download ifigotin/imagenetmini-1000
```

```
Downloading imagenetmini-1000.zip to /content
100% 3.92G/3.92G [02:11<00:00, 33.6MB/s]
100% 3.92G/3.92G [02:11<00:00, 32.0MB/s]
```

```
! unzip imagenetmini-1000
```

```
Streaming output truncated to the last 5000 lines.
  inflating: imagenet-mini/train/n07875152/n07875152_2632.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_2763.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_3015.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_3378.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_3497.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_3745.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_4157.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_461.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_4704.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_4789.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_4848.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_4940.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_546.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_5607.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_5711.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_634.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_7316.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_756.JPEG
  inflating: imagenet-mini/train/n07875152/n07875152_931.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_10933.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_1200.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_1967.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_202.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_2288.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_2666.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_281.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_289.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_3065.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_3393.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_3461.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_3511.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_3693.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_3703.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_3717.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_374.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_3848.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_3874.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_4354.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_4880.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_496.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_5406.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_5634.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_6269.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_681.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_8081.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_814.JPEG
  inflating: imagenet-mini/train/n07880968/n07880968_8804.JPEG
  inflating: imagenet-mini/train/n07892512/n07892512_10102.JPEG
  inflating: imagenet-mini/train/n07892512/n07892512_11784.JPEG
  inflating: imagenet-mini/train/n07892512/n07892512_12331.JPEG
  inflating: imagenet-mini/train/n07892512/n07892512_12922.JPEG
  inflating: imagenet-mini/train/n07892512/n07892512_1380.JPEG
```

```
  inflating: imagenet-mini/train/n07892512/n07892512_14036.JPEG
  inflating: imagenet-mini/train/n07892512/n07892512_14353.JPEG
  inflating: imagenet-mini/train/n07892512/n07892512_14741.JPEG
  inflating: imagenet-mini/train/n07892512/n07892512_14779.JPEG
  inflating: imagenet-mini/train/n07892512/n07892512_15094.JPEG
```

loading Dataset :

- collect 5000 data and devide them for training set and test set.

splitting Data :

- Data splitting is when data is divided into two or more subsets. Typically, with a two-part split, one part is used to evaluate or test the data and the other to train the model. Data splitting is an important aspect of data science, particularly for creating models based on data.
- We assume 80% (4000 images) of data as the train set and 20% (1000 images) as the validation set.

```
path = "/content/imagenet-mini"

import glob
import numpy as np

paths = glob.glob(path + "/*.JPEG") # Grabbing all the image file names
np.random.seed(123)
paths_subset = np.random.choice(paths, 1000, replace = True) # choosing 1000 images randomly
rand_idxs = np.random.permutation(5000)
train_idxs = rand_idxs[:4000] # choosing the first 4000 as training set
val_idxs = rand_idxs[4000:] # choosing last 1000 as validation set
train_paths = paths_subset[train_idxs]
val_paths = paths_subset[val_idxs]
print(len(train_paths), len(val_paths))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-14-06832687aa1b> in <cell line: 9>()
      7 paths = glob.glob(path + "/*.JPEG") # Grabbing all the image file names
      8 np.random.seed(123)
----> 9 paths_subset = np.random.choice(paths, 1000, replace = True) # choosing 1000 images randomly
     10 rand_idxs = np.random.permutation(5000)
     11 train_idxs = rand_idxs[:4000] # choosing the first 4000 as training set

mtrand.pyx in numpy.random.mtrand.RandomState.choice()

ValueError: 'a' cannot be empty unless no samples are taken
```

SEARCH STACK OVERFLOW

```
import glob
import numpy as np

# Your path to the image directory
path = "/content/drive/MyDrive/MergedImages"

# Grabbing all the image file names
paths = glob.glob(path + "/*.JPEG")

# Checking the length of paths to avoid sampling more items than available
total_paths = len(paths)

# Choosing the number of images to sample (adjust as needed)
num_images_to_sample = 1000

if total_paths >= num_images_to_sample:
    np.random.seed(123)
    paths_subset = np.random.choice(paths, num_images_to_sample, replace=False)  # Sample without replacement
    rand_idxs = np.random.permutation(num_images_to_sample)
    train_idxs = rand_idxs[:800]  # Choosing the first 800 as the training set
    val_idxs = rand_idxs[800:]   # Choosing the last 200 as the validation set
    train_paths = paths_subset[train_idxs]
    val_paths = paths_subset[val_idxs]
    print(len(train_paths), len(val_paths))
else:
    print("Not enough paths available to sample the specified number of images.")
    print("Total paths available:", total_paths)
```

```
Not enough paths available to sample the specified number of images.
Total paths available: 0
```

this is just simple visualization of dataset :

```
_, axes = plt.subplots(4, 4, figsize=(10, 10))
for ax, img_path in zip(axes.flatten(), train_paths):
    ax.imshow(Image.open(img_path))
    ax.axis("off")
```

loading images in machine :

- we use Lab color space instead of RGB to train the models because To train a model for colorization,we should give it a grayscale image and hope that it will make it colorful. When using Lab, we can give the L channel to the model (which is the grayscale image) and want it to predict the other two channels (a, b) and after its prediction, we concatenate all the channels and we get our colorful image.

Making Datasets and DataLoaders :

We resize the images and flipping horizontally (flipping only if it is training set), and read the RGB image. Then convert it to Lab color space and separate the first (grayscale) channel and the color channels. Then define the data loaders.

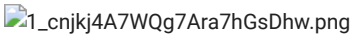
```
SIZE = 256
class ColorizationDataset(Dataset):
    def __init__(self, paths, split='train'):
        if split == 'train':
            self.transforms = transforms.Compose([
                transforms.Resize((SIZE, SIZE),  Image.BICUBIC),
                transforms.RandomHorizontalFlip(), # A little data augmentation!
            ])
        elif split == 'val':
            self.transforms = transforms.Resize((SIZE, SIZE),  Image.BICUBIC)

        self.split = split
        self.size = SIZE
        self.paths = paths

    def __getitem__(self, idx):
        img = Image.open(self.paths[idx]).convert("RGB")
        img = self.transforms(img)
        img = np.array(img)
        img_lab = rgb2lab(img).astype("float32") # Converting RGB to L*a*b
        img_lab = transforms.ToTensor()(img_lab)
        L = img_lab[[0], ...] / 50. - 1. # Between -1 and 1
        ab = img_lab[[1, 2], ...] / 110. # Between -1 and 1

        return {'L': L, 'ab': ab}

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

def make_dataloaders(batch_size=16, n_workers=0, pin_memory=True, **kwargs): # A handy function to make our dataloaders
    dataset = ColorizationDataset(**kwargs)
    dataloader = DataLoader(dataset, batch_size=batch_size, num_workers=n_workers,
                            pin_memory=pin_memory)
    return dataloader
```

```
train_dl = make_dataloaders(paths=train_paths, split='train')
val_dl = make_dataloaders(paths=val_paths, split='val')

data = next(iter(train_dl))
Ls, abs_ = data['L'], data['ab']
print(Ls.shape, abs_.shape)
print(len(train_dl), len(val_dl))
```

```
torch.Size([16, 1, 256, 256]) torch.Size([16, 2, 256, 256])
250 63
```

Generator proposed by the paper : (U-NET)

- A generator in a Generative Adversarial Network (GAN) is a deep learning model designed to generate new data samples that are similar to a target dataset. A UNet is a specific type of generator architecture that is used for image-to-image translation tasks, like our project goal (Image Colorization).



- UNet is composed of an encoder and a decoder, which are mirror images of each other. The encoder compresses the input image into a lower-dimensional feature representation, while the decoder expands the feature representation back into an output image. The encoder and decoder are connected by skip connections, which help to preserve spatial information and improve the quality of the generated image.
- This UNet which we build has 8 layers down, so if we start with a 256 by 256 image, in the middle of the U-Net we will get a 1 by 1 ($256 / 2^8$) image and then it gets up-sampled to produce a 256 by 256 image (with two channels).

Generator architectures:

The encoder-decoder architecture consists of:

encoder: C64-C128-C256-C512-C512-C512-C512-C512

decoder: CD512-CD512-CD512-C512-C256-C128-C64

After the last layer in the decoder, a convolution is applied to map to the number of output channels (2 chanels in colorization task), followed by a Tanh function. As an exception to the above notation, Batch- Norm is not applied to the first C64 layer in the encoder. All ReLUs in the encoder are leaky, with slope 0.2, while ReLUs in the decoder are not leaky. The U-Net architecture is identical except with skip connections between

each layer i in the encoder and layer n-i in the decoder, where n is the total number of layers. The skip connections concatenate activations from layer i to layer n - i. This changes the number of channels in the decoder:

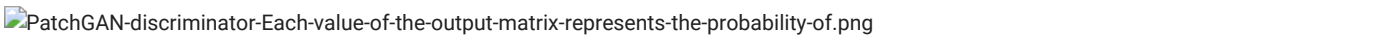
U-Net decoder:

CD512-CD1024-CD1024-C1024-C1024-C512-C256-C128

```
class UnetBlock(nn.Module):
    def __init__(self, nf, ni, submodule=None, input_c=None, dropout=False,
                 innermost=False, outermost=False):
        super().__init__()
        self.outermost = outermost
        if input_c is None: input_c = nf
        downconv = nn.Conv2d(input_c, ni, kernel_size=4,
                             stride=2, padding=1, bias=False)
        downrelu = nn.LeakyReLU(0.2, True)
        downnorm = nn.BatchNorm2d(ni)
        uprelu = nn.ReLU(True)
        upnorm = nn.BatchNorm2d(nf)

        if outermost:
            upconv = nn.ConvTranspose2d(ni * 2, nf, kernel_size=4,
                                        stride=2, padding=1)
            down = [downconv]
            up = [uprelu, upconv, nn.Tanh()]
            model = down + [submodule] + up
        elif innermost:
            upconv = nn.ConvTranspose2d(ni, nf, kernel_size=4,
                                        stride=2, padding=1, bias=False)
            down = [downrelu, downconv]
            up = [uprelu, upconv, upnorm]
            model = down + up
        else:
            upconv = nn.ConvTranspose2d(ni * 2, nf, kernel_size=4,
                                        stride=2, padding=1, bias=False)
            down = [downrelu, downconv, downnorm]
            up = [uprelu, upconv, upnorm]
            if dropout: up += [nn.Dropout(0.5)]
            model = down + [submodule] + up
        self.model = nn.Sequential(*model)

    def forward(self, x):
        if self.outermost:
            return self.model(x)
        else:
            return torch.cat([x, self.model(x)], 1)

class Unet(nn.Module):
    def __init__(self, input_c=1, output_c=2, n_down=8, num_filters=64):
        super().__init__()
        unet_block = UnetBlock(num_filters * 8, num_filters * 8, innermost=True)
        for _ in range(n_down - 5):
            unet_block = UnetBlock(num_filters * 8, num_filters * 8, submodule=unet_block, dropout=True)
        out_filters = num_filters * 8
        for _ in range(3):
            unet_block = UnetBlock(out_filters // 2, out_filters, submodule=unet_block)
            out_filters //= 2
        self.model = UnetBlock(output_c, out_filters, input_c=input_c, submodule=unet_block, outermost=True)

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

Discriminator : (patchGAN)

- PatchGANs are a useful technique for image generation tasks, as they can produce high-quality, realistic images while improving stability and efficiency during training.



- In a patch discriminator, the model outputs one number for every patch of say 70 by 70 pixels of the input image and for each of them decides whether it is fake or not separately.
- The architecture of our discriminator is rather straight forward. Here we implements a model by stacking blocks of Conv-BatchNorm-LeackyReLU to decide whether the input image is fake or real. Notice that the first and last blocks there is nonormalization and the last block has no activation function (it is embedded in the loss function we will use).

- Discriminator architectures:

  The discriminator structure will follow the pattern build of C64-C128-C256-C512. We implement a model by stacking blocks of Conv-BatchNorm-LeackyReLU to decide whether the input image is fake or real. Besides, first and last blocks do not use normalization and the last block has no activation function(It is embedded in the loss function)

```
class PatchDiscriminator(nn.Module):
    def __init__(self, input_c, num_filters=64, n_down=3):
        super().__init__()
        model = [self.get_layers(input_c, num_filters, norm=False)]
        model += [self.get_layers(num_filters * 2 ** i, num_filters * 2 ** (i + 1), s=1 if i == (n_down-1) else 2)
                          for i in range(n_down)] # the 'if' statement is taking care of not using
                                                  # stride of 2 for the last block in this loop
        model += [self.get_layers(num_filters * 2 ** n_down, 1, s=1, norm=False, act=False)] # Make sure to not use normalization or
                                                                                             # activation for the last layer of the mode
        self.model = nn.Sequential(*model)

    def get_layers(self, ni, nf, k=4, s=2, p=1, norm=True, act=True): # when needing to make some repeatitive blocks of layers,
        layers = [nn.Conv2d(ni, nf, k, s, p, bias=not norm)]          # it's always helpful to make a separate method for that purpose
        if norm: layers += [nn.BatchNorm2d(nf)]
        if act: layers += [nn.LeakyReLU(0.2, True)]
        return nn.Sequential(*layers)

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

```
PatchDiscriminator(3)
```

```
PatchDiscriminator(
  (model): Sequential(
    (0): Sequential(
      (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
      (1): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (1): Sequential(
      (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (2): Sequential(
      (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (3): Sequential(
      (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (4): Sequential(
      (0): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
    )
  )
)
```

And its output shape:

- Here, the model's output shape is 30 by 30 but it does not mean that our patches are 30 by 30. The actual patch size is obtained when you compute the receptive field of each of these 900 (30 multiplied by 30) output numbers which in our case will be 70 by 70.**

```
discriminator = PatchDiscriminator(3)
dummy_input = torch.randn(16, 3, 256, 256) # batch_size, channels, size, size
out = discriminator(dummy_input)
out.shape
```

```
torch.Size([16, 1, 30, 30])
```

GAN Loss:

- The loss function in a cGAN typically consists of two components: the generator loss and the discriminator loss. The generator loss measures how well the generator is able to fool the discriminator, while the discriminator loss measures how well the discriminator is able to distinguish between real and generated samples.

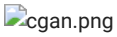
Generator loss :

- The generator loss is usually defined as the negative log-likelihood of the discriminator's output, given the generated sample and the conditioning input. The generator tries to minimize this loss, which encourages it to generate samples that are classified as real by the

discriminator.

Discriminator loss :

- The discriminator loss measures the binary cross-entropy between the true labels (real or fake) and the discriminator's predictions. The discriminator tries to maximize this loss, which encourages it to produce accurate predictions.



- This is a handy class we can use to calculate the GAN loss of our final model. In the **init** we decide which kind of loss we're going to use (which will be "vanilla" in our project) and register some constant tensors as the "real" and "fake" labels. Then when we call this module, it makes an appropriate tensor full of zeros or ones (according to what we need at the stage) and computes the loss.
- we use L1 loss as well and compute the distance between the predicted two channels and the target two channels and multiply this loss by a coefficient (which is 100 in our case) to balance the two losses and then add this loss to the adversarial loss. Then we call the backward method of the loss.

```
class GANLoss(nn.Module):
    def __init__(self, gan_mode='vanilla', real_label=1.0, fake_label=0.0):
        super().__init__()
        self.register_buffer('real_label', torch.tensor(real_label))
        self.register_buffer('fake_label', torch.tensor(fake_label))
        if gan_mode == 'vanilla':
            self.loss = nn.BCEWithLogitsLoss()
        elif gan_mode == 'lsgan':
            self.loss = nn.MSELoss()

    def get_labels(self, preds, target_is_real):
        if target_is_real:
            labels = self.real_label
        else:
            labels = self.fake_label
        return labels.expand_as(preds)

    def __call__(self, preds, target_is_real):
        labels = self.get_labels(preds, target_is_real)
        loss = self.loss(preds, labels)
        return loss
```

Model Initialization :

- Here is our logic to initialize our models. We are going to initialize the weights of our modelwhich are the proposed hyperparameters in the article, with:

1. mean= 0.0
2. standard deviation = 0.02

```
def init_weights(net, init='norm', gain=0.02):

    def init_func(m):
        classname = m.__class__.__name__
        if hasattr(m, 'weight') and 'Conv' in classname:
            if init == 'norm':
                nn.init.normal_(m.weight.data, mean=0.0, std=gain)
            elif init == 'xavier':
                nn.init.xavier_normal_(m.weight.data, gain=gain)
            elif init == 'kaiming':
                nn.init.kaiming_normal_(m.weight.data, a=0, mode='fan_in')

            if hasattr(m, 'bias') and m.bias is not None:
                nn.init.constant_(m.bias.data, 0.0)
        elif 'BatchNorm2d' in classname:
            nn.init.normal_(m.weight.data, 1., gain)
            nn.init.constant_(m.bias.data, 0.)

    net.apply(init_func)
    print(f"model initialized with {init} initialization")
    return net

def init_model(model, device):
    model = model.to(device)
    model = init_weights(model)
    return model
```

Main Model :

- step1: This class brings together all the previous parts and implements a few methods to take care of training our complete model.
- step2: In the **init** our generator and discriminator using the previous functions and classes which defined and also initialize them with init_model function.
- step 3: using module's forward method (only once per iteration (batch of training set)) and store the outputs in fake_color variable of the class.
- step4: first train the discriminator by using backward_D method in which we feed the fake images produced by generator to the discriminator and label them as fake.
- step5: Then feed a batch of real images from training set to the discriminator and label them as real. also add up the two losses for fake and real and take the average and then call the backward on the final loss.
- step6: train the generator. In backward_G method feed the discriminator the fake image and try to fool it by assigning real labels to them and calculating the adversarial loss.

We use minibatch SGD and apply the Adam solver, with a learning rate of 0.0002, and momentum parameters beta1 = 0.5, beta2 = 0.999 . These parameteres obtained by the refrance paper.

```
class MainModel(nn.Module):
    def __init__(self, net_G=None, lr_G=2e-4, lr_D=2e-4,
                 beta1=0.5, beta2=0.999, lambda_L1=100.):
        super().__init__()

        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.lambda_L1 = lambda_L1

        if net_G is None:
            self.net_G = init_model(Unet(input_c=1, output_c=2, n_down=8, num_filters=64), self.device)
        else:
            self.net_G = net_G.to(self.device)
        self.net_D = init_model(PatchDiscriminator(input_c=3, n_down=3, num_filters=64), self.device)
        self.GANcriterion = GANLoss(gan_mode='vanilla').to(self.device)
        self.L1criterion = nn.L1Loss()
        self.opt_G = optim.Adam(self.net_G.parameters(), lr=lr_G, betas=(beta1, beta2))
        self.opt_D = optim.Adam(self.net_D.parameters(), lr=lr_D, betas=(beta1, beta2))

    def set_requires_grad(self, model, requires_grad=True):
        for p in model.parameters():
            p.requires_grad = requires_grad

    def setup_input(self, data):
        self.L = data['L'].to(self.device)
        self.ab = data['ab'].to(self.device)

    def forward(self):
        self.fake_color = self.net_G(self.L)

    def backward_D(self):
        fake_image = torch.cat([self.L, self.fake_color], dim=1)
        fake_preds = self.net_D(fake_image.detach())
        self.loss_D_fake = self.GANcriterion(fake_preds, False)
        real_image = torch.cat([self.L, self.ab], dim=1)
        real_preds = self.net_D(real_image)
        self.loss_D_real = self.GANcriterion(real_preds, True)
        self.loss_D = (self.loss_D_fake + self.loss_D_real) * 0.5
        self.loss_D.backward()

    def backward_G(self):
        fake_image = torch.cat([self.L, self.fake_color], dim=1)
        fake_preds = self.net_D(fake_image)
        self.loss_G_GAN = self.GANcriterion(fake_preds, True)
        self.loss_G_L1 = self.L1criterion(self.fake_color, self.ab) * self.lambda_L1
        self.loss_G = self.loss_G_GAN + self.loss_G_L1
        self.loss_G.backward()

    def optimize(self):
        self.forward()
        self.net_D.train()
        self.set_requires_grad(self.net_D, True)
        self.opt_D.zero_grad()
        self.backward_D()
        self.opt_D.step()

        self.net_G.train()
        self.set_requires_grad(self.net_D, False)
        self.opt_G.zero_grad()
        self.backward_G()
        self.opt_G.step()
```

Utility functions :

- some utility functions to log the losses of our network and also visualize the results during training.

```
class AverageMeter:
    def __init__(self):
        self.reset()

    def reset(self):
        self.count, self.avg, self.sum = [0.] * 3

    def update(self, val, count=1):
        self.count += count
        self.sum += count * val
        self.avg = self.sum / self.count

def create_loss_meters():
    loss_D_fake = AverageMeter()
    loss_D_real = AverageMeter()
    loss_D = AverageMeter()
    loss_G_GAN = AverageMeter()
    loss_G_L1 = AverageMeter()
    loss_G = AverageMeter()

    return {'loss_D_fake': loss_D_fake,
            'loss_D_real': loss_D_real,
            'loss_D': loss_D,
            'loss_G_GAN': loss_G_GAN,
            'loss_G_L1': loss_G_L1,
            'loss_G': loss_G}

def update_losses(model, loss_meter_dict, count):
    for loss_name, loss_meter in loss_meter_dict.items():
        loss = getattr(model, loss_name)
        loss_meter.update(loss.item(), count=count)

def lab_to_rgb(L, ab):
    """
    Takes a batch of images
    """

    L = (L + 1.) * 50.
    ab = ab * 110.
    Lab = torch.cat([L, ab], dim=1).permute(0, 2, 3, 1).cpu().numpy()
    rgb_imgs = []
    for img in Lab:
        img_rgb = lab2rgb(img)
        rgb_imgs.append(img_rgb)
    return np.stack(rgb_imgs, axis=0)

def visualize(model, data, save=True):
    model.net_G.eval()
    with torch.no_grad():
        model.setup_input(data)
        model.forward()
    model.net_G.train()
    fake_color = model.fake_color.detach()
    real_color = model.ab
    L = model.L
    fake_imgs = lab_to_rgb(L, fake_color)
    real_imgs = lab_to_rgb(L, real_color)
    fig = plt.figure(figsize=(15, 8))
    for i in range(5):
        ax = plt.subplot(3, 5, i + 1)
        ax.imshow(L[i][0].cpu(), cmap='gray')
        ax.axis("off")
        ax = plt.subplot(3, 5, i + 1 + 5)
        ax.imshow(fake_imgs[i])
        ax.axis("off")
        ax = plt.subplot(3, 5, i + 1 + 10)
        ax.imshow(real_imgs[i])
        ax.axis("off")
    plt.show()
    if save:
        fig.savefig(f"colorization_{time.time()}.png")

def log_results(loss_meter_dict,show=True):
    res = {}
    for loss_name, loss_meter in loss_meter_dict.items():
        if show:
          print(f"{loss_name}: {loss_meter.avg:.5f}")
        res[loss_name] = loss_meter.avg

    return res
```

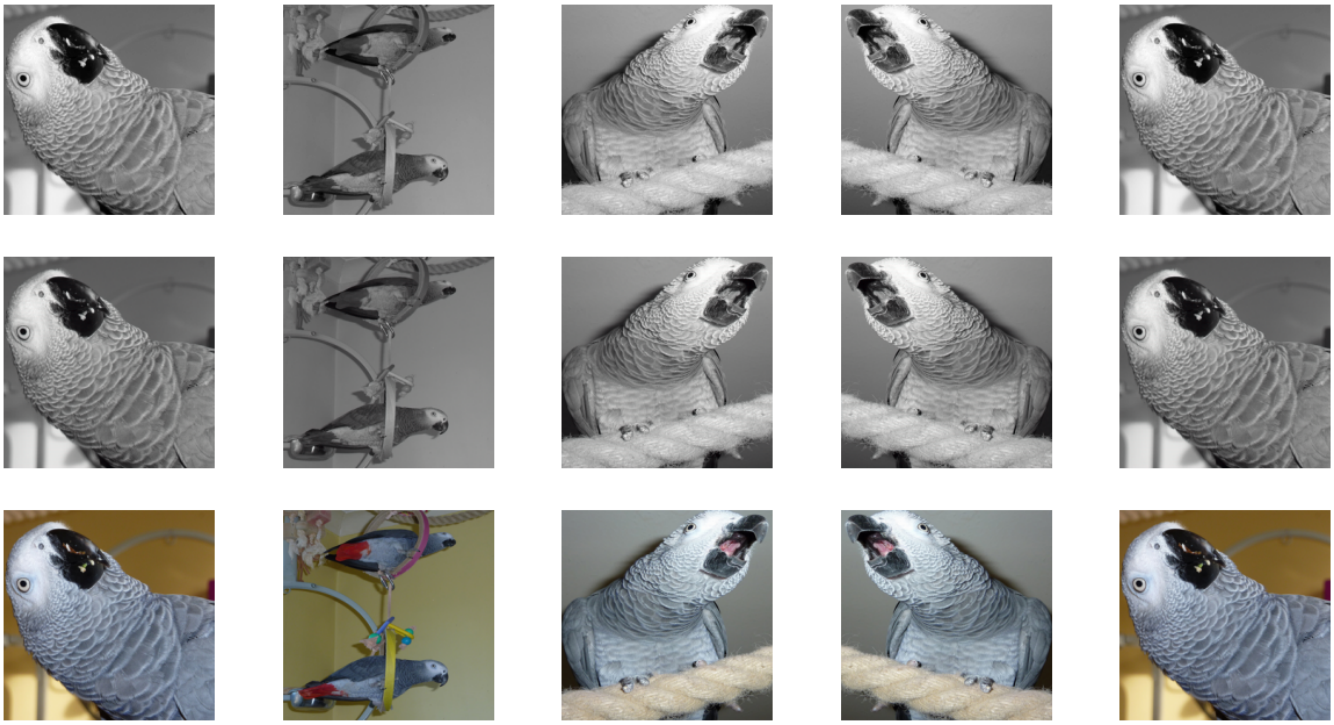
Train Model :

- We train the model for 6 different group of images.
- Every epoch takes about 3 to 4 minutes on Colab.
- In this step we train it for 30 epoch as it took a long time to train.
- But in each step you can see the changes.
- By training this model for more than 50 epoch we can see step by step reasonable results.
- Also for the perfect result we have to train it in 100 epochs.

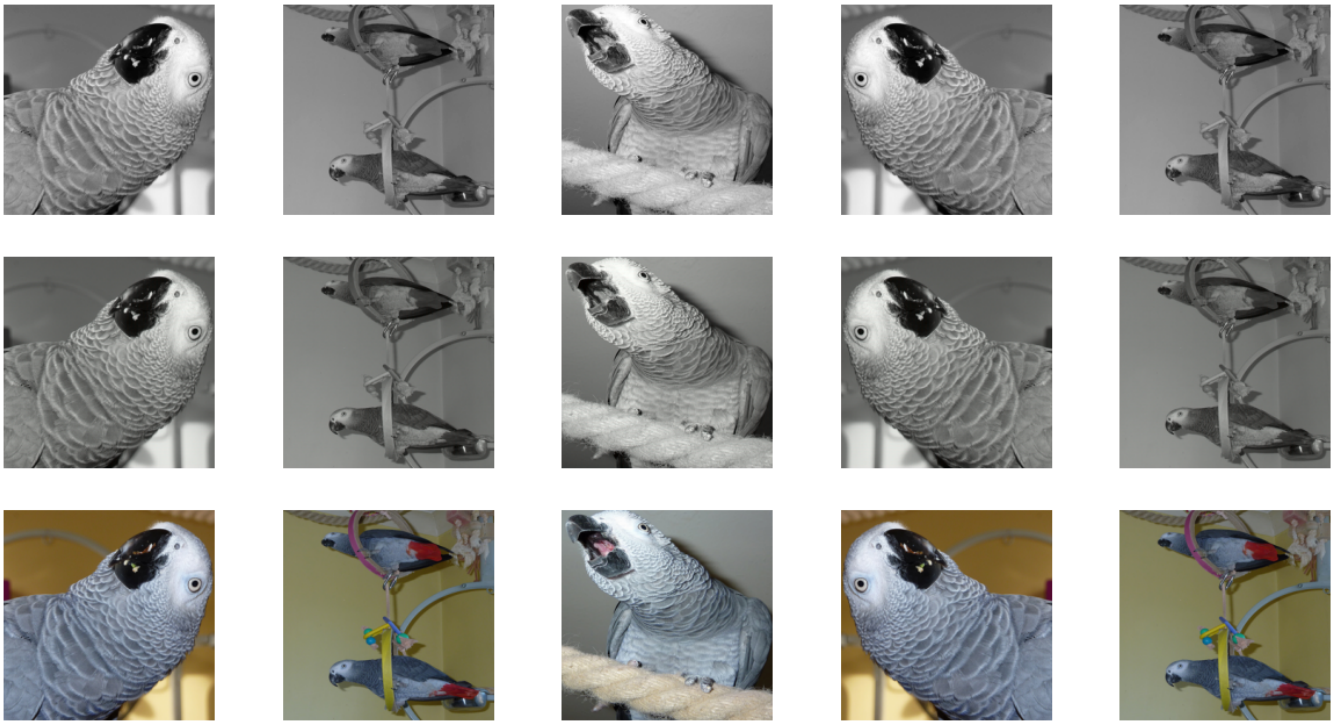
```
results = []
show=False
def train_model(model, train_dl, epochs, display_every=1):
    data = next(iter(val_dl)) # getting a batch for visualizing the model output after fixed intrvals
    for e in range(epochs):
        loss_meter_dict = create_loss_meters() # function returing a dictionary of objects to
        i = 0                                  # log the losses of the complete network
        for data in tqdm(train_dl):
            model.setup_input(data)
            model.optimize()
            update_losses(model, loss_meter_dict, count=data['L'].size(0)) # function updating the log objects
            i += 1
            if i in [1,5,10,50,100,249]:
                visualize(model, data, save=False) # function displaying the model's outputs
                show=True
            else:
                show=False
            if show:
                print(f"\nEpoch {e+1}/{epochs}")
                print(f"Iteration {i}/{len(train_dl)}")
            results.append(log_results(loss_meter_dict,show=show)) # function to print out the losses


model = MainModel()
train_model(model, train_dl, 30)
```

```
model initialized with norm initialization
model initialized with norm initialization
```

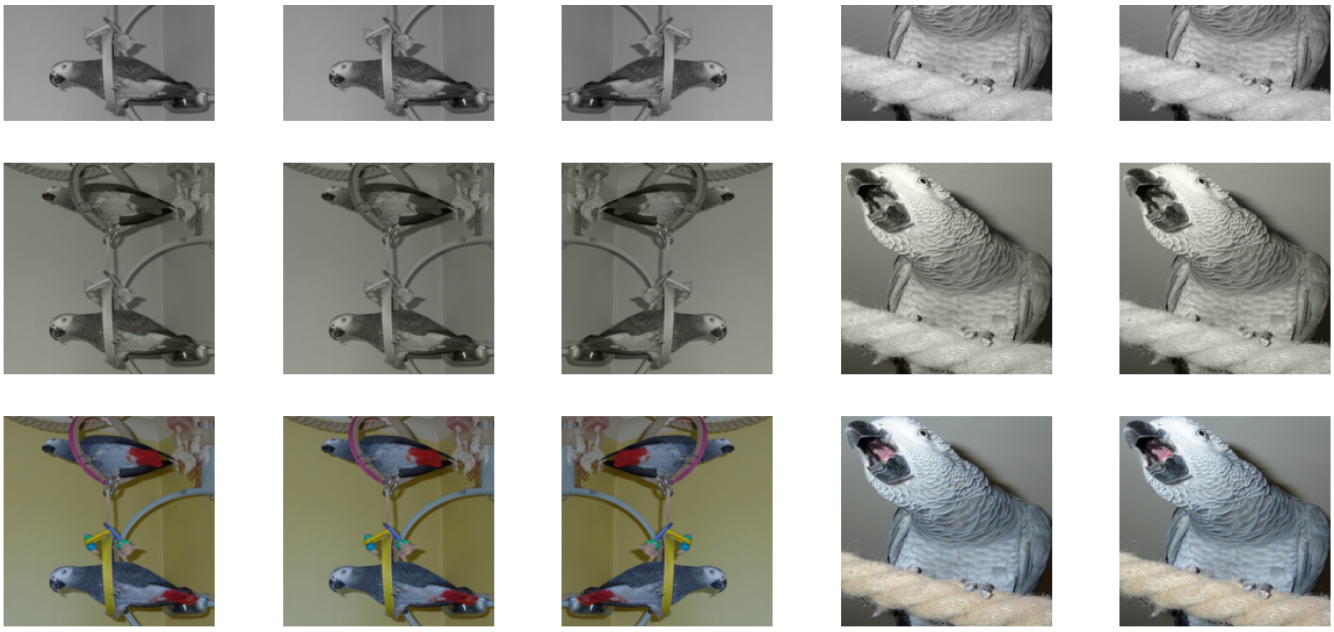
100% 250/250 [03:53<00:00, 1.24s/it]



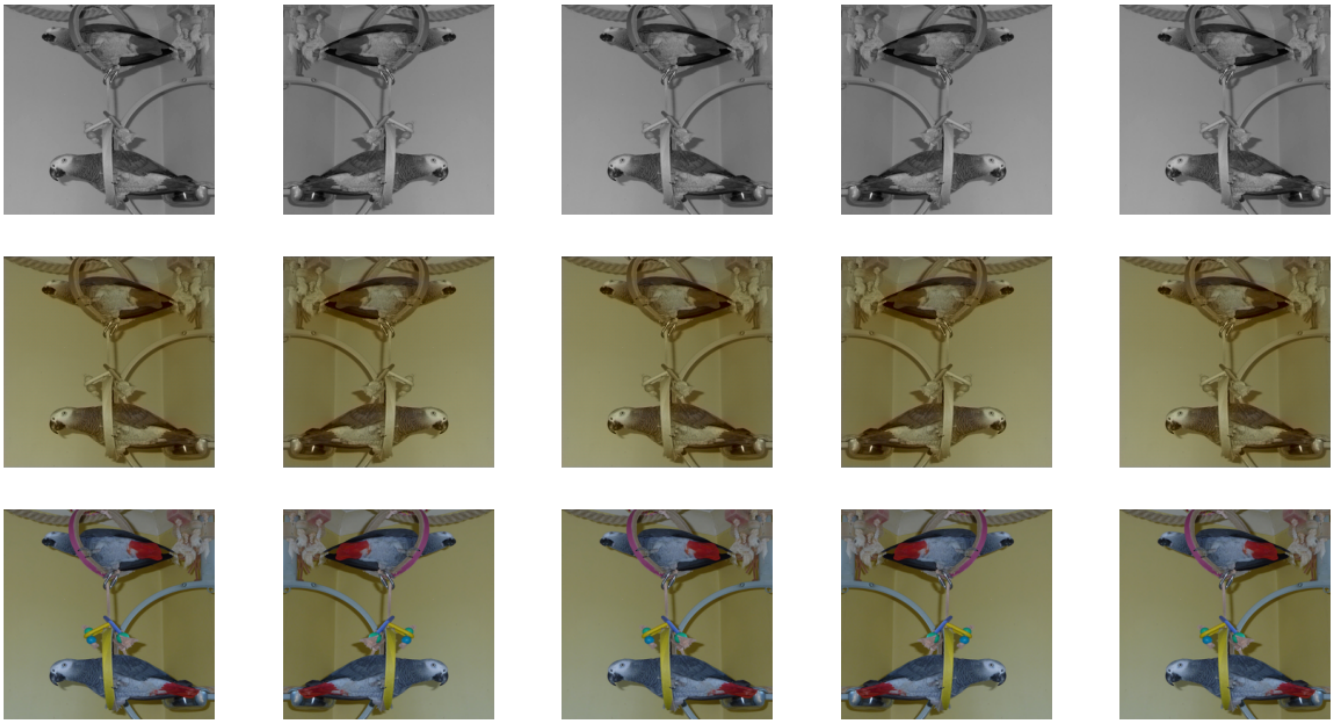
```
Epoch 1/30
Iteration 1/250
loss_D_fake: 1.03298
loss_D_real: 0.65220
loss_D: 0.84259
loss_G_GAN: 1.71544
loss_G_L1: 17.31188
loss_G: 19.02732
```



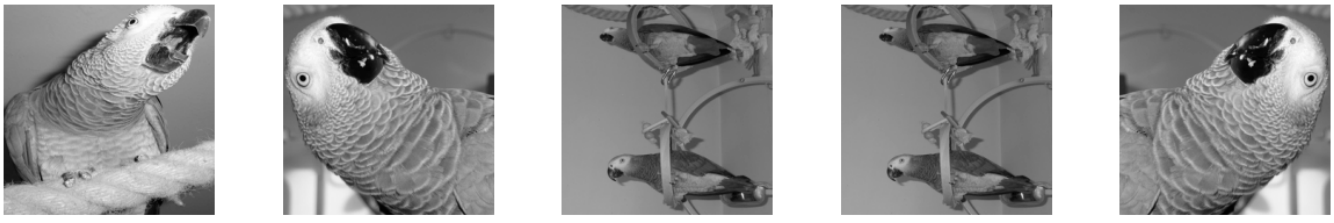
```
Epoch 1/30
Iteration 5/250
loss_D_fake: 0.95037
loss_D_real: 1.00676
loss_D: 0.97856
loss_G_GAN: 1.13093
loss_G_L1: 12.68308
loss_G: 13.81402
```

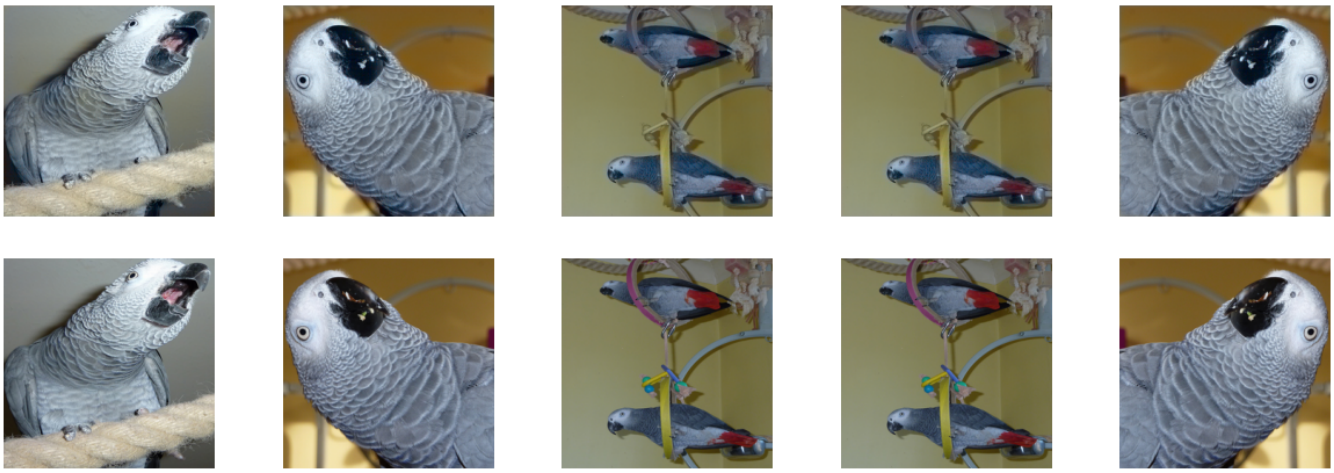


```
Epoch 1/30
Iteration 10/250
loss_D_fake: 0.85160
loss_D_real: 0.85120
loss_D: 0.85140
loss_G_GAN: 0.98162
loss_G_L1: 10.36481
loss_G: 11.34644
/usr/local/lib/python3.10/dist-packages/skimage/_shared/utils.py:394: UserWarning: Color data out of range: Z < 0 in 1 pixels
  return func(*args, **kwargs)
```

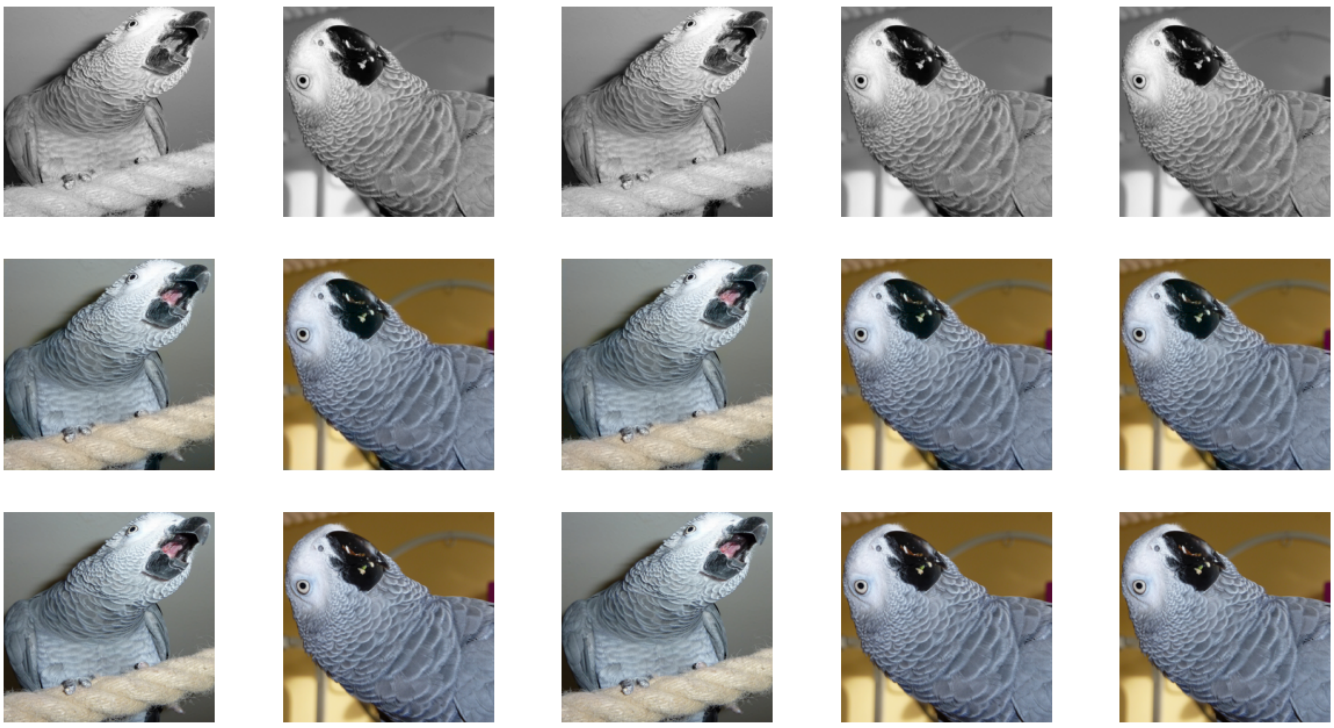


```
Epoch 1/30
Iteration 50/250
loss_D_fake: 0.71280
loss_D_real: 0.71852
loss_D: 0.71566
loss_G_GAN: 0.85204
loss_G_L1: 5.85100
loss_G: 6.70304
```



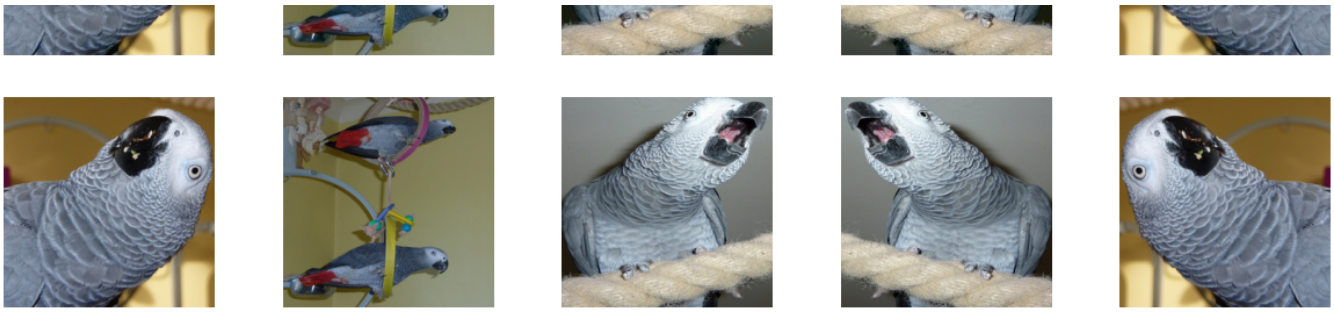


```
Epoch 1/30
Iteration 100/250
loss_D_fake: 0.68754
loss_D_real: 0.69072
loss_D: 0.68913
loss_G_GAN: 0.84801
loss_G_L1: 4.51668
loss_G: 5.36469
```



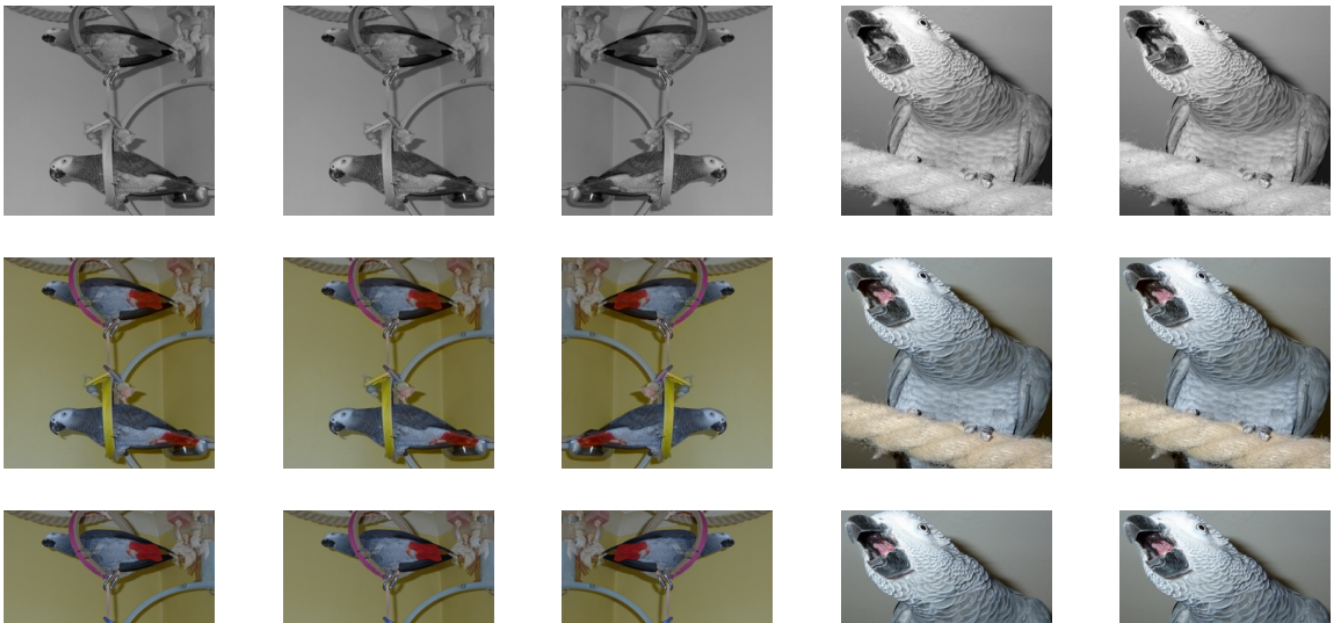
```
Epoch 1/30
Iteration 249/250
loss_D_fake: 0.66964
loss_D_real: 0.67299
loss_D: 0.67131
loss_G_GAN: 0.84148
loss_G_L1: 3.19797
loss_G: 4.03945
```

13% 32/250 [00:34<04:10, 1.15s/it]



Epoch 2/30
Iteration 1/250
loss_D_fake: 0.59616
loss_D_real: 0.68671
loss_D: 0.64143
loss_G_GAN: 0.80020
loss_G_L1: 1.85596
loss_G: 2.65616

Epoch 2/30
Iteration 5/250
loss_D_fake: 0.64511
loss_D_real: 0.65907
loss_D: 0.65209
loss_G_GAN: 0.81778
loss_G_L1: 1.87266
loss_G: 2.69044



```
Epoch 2/30
Iteration 10/250
loss_D_fake: 0.63773
loss_D_real: 0.64851
loss_D: 0.64312
loss_G_GAN: 0.83362
loss_G_L1: 2.00918
loss_G: 2.84280
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-34-99f23642d9e2> in <cell line: 25>()
     23
     24 model = MainModel()
---> 25 train_model(model, train_dl, 30)

8 frames
/usr/local/lib/python3.10/dist-packages/PIL/Image.py in open(fp, mode, formats)
   3225
   3226     if filename:
-> 3227         fp = builtins.open(filename, "rb")
   3228         exclusive_fp = True
   3229

KeyboardInterrupt:
```

SEARCH STACK OVERFLOW

save the model after training:

```
import pandas as pd
torch.save(model, '/content/drive/MyDrive/ DNN_final_project/model.pt')
df=pd.DataFrame(results)
df.to_csv('/content/drive/MyDrive/ DNN_final_project/raw_data.csv')
```

visualization of losses:

- these plots is just for 45 epoches but if you can see even for the 45 epoch we can realize the decreasing of losses.
- if we had the powerfull system, we were able to show these plots for 100 epoches.

model summery :

```
map_location=torch.device('cuda')
model = torch.load('/content/drive/MyDrive/ DNN_final_project/model.pt')
model.eval()
```

In the google colab we can not run the code moLoading and run the model from epoch 30 to 45

```
!cp -R /content/drive/MyDrive/DNN_final_project/5k_model /content/
```

```
map_location=torch.device('cuda')
model = torch.load('/content/5k_model/Copy of model.pt')
```

```
import pandas as pd
df = pd.read_csv('/content/5k_model/Copy of raw_data.csv')
```