Data Preparation

```
ab_path = "/kaggle/input/image-colorization/ab/ab/ab1.npy"

l_path = "/kaggle/input/image-colorization/l/gray_scale.npy"

ab_df = np.load(ab_path)[0:5000]

L_df = np.load(l_path)[0:5000]

dataset = (L_df,ab_df )

gc.collect()
```

Data Exploration and Visualiaztion

```
def lab_to_rgb(L, ab):

    """

    Takes an image or a batch of images and converts from LAB space to RGB

    """

    L = L  * 100

    ab = (ab - 0.5) * 128 * 2

    Lab = torch.cat([L, ab], dim=2).numpy()

    rgb_imgs = []

    for img in Lab:

        img_rgb = lab2rgb(img)

        rgb_imgs.append(img_rgb)

    return np.stack(rgb_imgs, axis=0)

plt.figure(figsize=(30,30))

for i in range(1,16,2):

    plt.subplot(4,4,i)

    img = np.zeros((224,224,3))

    img[:,:,0] = L_df[i]

    plt.title('B&W')

    plt.imshow(lab2rgb(img))


    plt.subplot(4,4,i+1)
```

```
    img[:,:,1:] = ab_df[i]
    img = img.astype('uint8')
    img = cv2.cvtColor(img, cv2.COLOR_LAB2RGB)
    plt.title('Colored')
    plt.imshow(img)
gc.collect()
```

Data Loader

```
class ImageColorizationDataset(Dataset):
    ''' Black and White (L) Images and corresponding A&B Colors'''
    def __init__(self, dataset, transform=None):
        '''
        :param dataset: Dataset name.
        :param data_dir: Directory with all the images.
        :param transform: Optional transform to be applied on sample
        '''
        self.dataset = dataset
        self.transform = transform
    def __len__(self):
        return len(self.dataset[0])
    def __getitem__(self, idx):
        L = np.array(dataset[0][idx]).reshape((224,224,1))
        L = transforms.ToTensor()(L)
        ab = np.array(dataset[1][idx])
        ab = transforms.ToTensor()(ab)
        return ab, L
batch_size = 1
# Prepare the Datasets
train_dataset = ImageColorizationDataset(dataset = (L_df, ab_df))
test_dataset = ImageColorizationDataset(dataset = (L_df, ab_df))
```

Build DataLoaders

train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle = True, pin_memory = True)

test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle = False, pin_memory = True)

Data Modelling

```
class ResBlock(nn.Module):
  def __init__(self, in_channels, out_channels, stride=1):
    super().__init__()
    self.layer = nn.Sequential(
      nn.Conv2d(in_channels,out_channels,kernel_size=3, padding=1, stride=stride, bias=False),
      nn.BatchNorm2d(out_channels),
      nn.ReLU(inplace=True),
      nn.Conv2d(out_channels, out_channels,kernel_size=3,padding=1, stride=1, bias=False),
      nn.BatchNorm2d(out_channels),
      nn.ReLU(inplace=True)
    )
    self.identity_map = nn.Conv2d(in_channels, out_channels,kernel_size=1,stride=stride)
    self.relu = nn.ReLU(inplace=True)
  def forward(self, inputs):
    x = inputs.clone().detach()
    out = self.layer(x)
    residual  = self.identity_map(inputs)
    skip = out + residual
    return self.relu(skip)
class DownSampleConv(nn.Module):
  def __init__(self, in_channels, out_channels, stride=1):
```

```
        super().__init__()
        self.layer = nn.Sequential(
            nn.MaxPool2d(2),
            ResBlock(in_channels, out_channels)
        )
    def forward(self, inputs):
        return self.layer(inputs)
class UpSampleConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.upsample = nn.Upsample(scale_factor=2, mode="bilinear", align_corners=True)
        self.res_block = ResBlock(in_channels + out_channels, out_channels)
    def forward(self, inputs, skip):
        x = self.upsample(inputs)
        x = torch.cat([x, skip], dim=1)
        x = self.res_block(x)
        return x
class Generator(nn.Module):
    def __init__(self, input_channel, output_channel, dropout_rate = 0.2):
        super().__init__()
        self.encoding_layer1_ = ResBlock(input_channel,64)
        self.encoding_layer2_ = DownSampleConv(64, 128)
        self.encoding_layer3_ = DownSampleConv(128, 256)
        self.bridge = DownSampleConv(256, 512)
        self.decoding_layer3_ = UpSampleConv(512, 256)
        self.decoding_layer2_ = UpSampleConv(256, 128)
        self.decoding_layer1_ = UpSampleConv(128, 64)
        self.output = nn.Conv2d(64, output_channel, kernel_size=1)
        self.dropout = nn.Dropout2d(dropout_rate)
```

```
  def forward(self, inputs):
    ###################### Enocoder #########################
    e1 = self.encoding_layer1_(inputs)
    e1 = self.dropout(e1)
    e2 = self.encoding_layer2_(e1)
    e2 = self.dropout(e2)
    e3 = self.encoding_layer3_(e2)
    e3 = self.dropout(e3)
    ###################### Bridge #########################
    bridge = self.bridge(e3)
    bridge = self.dropout(bridge)
    ###################### Decoder #########################
    d3 = self.decoding_layer3_(bridge, e3)
    d2 = self.decoding_layer2_(d3, e2)
    d1 = self.decoding_layer1_(d2, e1)
    ###################### Output #########################
    output = self.output(d1)
    return output
model = Generator(1,2).to(device)
summary(model, (1, 224, 224), batch_size = 1)
```

B. Discriminator (Critic)

```
class Critic(nn.Module):
  def __init__(self, in_channels=3):
    super(Critic, self).__init__()
    def critic_block(in_filters, out_filters, normalization=True):
      """Returns layers of each critic block"""
      layers = [nn.Conv2d(in_filters, out_filters, 4, stride=2, padding=1)]
      if normalization:
```

```
                layers.append(nn.InstanceNorm2d(out_filters))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers
        self.model = nn.Sequential(
            *critic_block(in_channels, 64, normalization=False),
            *critic_block(64, 128),
            *critic_block(128, 256),
            *critic_block(256, 512),
            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
            nn.Linear(512, 1)
        )
    def forward(self, ab, l):
        # Concatenate image and condition image by channels to produce input
        img_input = torch.cat((ab, l), 1)
        output = self.model(img_input)
        return output
model = Critic(3).to(device)
summary(model, [(2, 224, 224), (1, 224, 224)], batch_size = 1)
```

C. Generative Adversarial Network

```
def _weights_init(m):
    if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d)):
        torch.nn.init.normal_(m.weight, 0.0, 0.02)
    if isinstance(m, nn.BatchNorm2d):
        torch.nn.init.normal_(m.weight, 0.0, 0.02)
        torch.nn.init.constant_(m.bias, 0)
def display_progress(cond, real, fake, current_epoch = 0, figsize=(20,15)):
    """
    Save cond, real (original) and generated (fake)
```

```
    images in one panel
    """
    cond = cond.detach().cpu().permute(1, 2, 0)
    real = real.detach().cpu().permute(1, 2, 0)
    fake = fake.detach().cpu().permute(1, 2, 0)
    images = [cond, real, fake]
    titles = ['input','real','generated']
    print(f'Epoch: {current_epoch}')
    fig, ax = plt.subplots(1, 3, figsize=figsize)
    for idx,img in enumerate(images):
        if idx == 0:
            ab = torch.zeros((224,224,2))
            img = torch.cat([images[0]* 100, ab], dim=2).numpy()
            imgan = lab2rgb(img)
        else:
            imgan = lab_to_rgb(images[0],img)
        ax[idx].imshow(imgan)
        ax[idx].axis("off")
    for idx, title in enumerate(titles):
        ax[idx].set_title('{}'.format(title))
    plt.show()
class CWGAN(pl.LightningModule):
    def __init__(self, in_channels, out_channels, learning_rate=0.0002, lambda_recon=100, display_step=10, lambda_gp=10, lambda_r1=10,):
        super().__init__()
        self.save_hyperparameters()
        self.display_step = display_step
        self.generator = Generator(in_channels, out_channels)
        self.critic = Critic(in_channels + out_channels)
```

```
        self.optimizer_G = optim.Adam(self.generator.parameters(), lr=learning_rate, betas=(0.5,
0.9))
        self.optimizer_C = optim.Adam(self.critic.parameters(), lr=learning_rate, betas=(0.5, 0.9))
        self.lambda_recon = lambda_recon
        self.lambda_gp = lambda_gp
        self.lambda_r1 = lambda_r1
        self.recon_criterion = nn.L1Loss()
        self.generator_losses, self.critic_losses  =[],[]
    def configure_optimizers(self):
        return [self.optimizer_C, self.optimizer_G]
    def generator_step(self, real_images, conditioned_images):
        # WGAN has only a reconstruction loss
        self.optimizer_G.zero_grad()
        fake_images = self.generator(conditioned_images)
        recon_loss = self.recon_criterion(fake_images, real_images)
        recon_loss.backward()
        self.optimizer_G.step()
        # Keep track of the average generator loss
        self.generator_losses += [recon_loss.item()]
    def critic_step(self, real_images, conditioned_images):
        self.optimizer_C.zero_grad()
        fake_images = self.generator(conditioned_images)
        fake_logits = self.critic(fake_images, conditioned_images)
        real_logits = self.critic(real_images, conditioned_images)
        # Compute the loss for the critic
        loss_C = real_logits.mean() - fake_logits.mean()
        # Compute the gradient penalty
        alpha = torch.rand(real_images.size(0), 1, 1, 1, requires_grad=True)
        alpha = alpha.to(device)
        interpolated = (alpha * real_images + (1 - alpha) * fake_images.detach()).requires_grad_(True)
```

```
        interpolated_logits = self.critic(interpolated, conditioned_images)
        grad_outputs = torch.ones_like(interpolated_logits, dtype=torch.float32, requires_grad=True)
        gradients = torch.autograd.grad(outputs=interpolated_logits, inputs=interpolated,
grad_outputs=grad_outputs,create_graph=True, retain_graph=True)[0]
        gradients = gradients.view(len(gradients), -1)
        gradients_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean()
        loss_C += self.lambda_gp * gradients_penalty
        # Compute the R1 regularization loss
        r1_reg = gradients.pow(2).sum(1).mean()
        loss_C += self.lambda_r1 * r1_reg
        # Backpropagation
        loss_C.backward()
        self.optimizer_C.step()
        self.critic_losses += [loss_C.item()]
    def training_step(self, batch, batch_idx, optimizer_idx):
        real, condition = batch
        if optimizer_idx == 0:
            self.critic_step(real, condition)
        elif optimizer_idx == 1:
            self.generator_step(real, condition)
        gen_mean = sum(self.generator_losses[-self.display_step:]) / self.display_step
        crit_mean = sum(self.critic_losses[-self.display_step:]) / self.display_step
        if self.current_epoch%self.display_step==0 and batch_idx==0 and optimizer_idx==1:
            fake = self.generator(condition).detach()
            torch.save(cwgan.generator.state_dict(), "ResUnet_"+ str(self.current_epoch) +".pt")
            torch.save(cwgan.critic.state_dict(), "PatchGAN_"+ str(self.current_epoch) +".pt")
            print(f"Epoch {self.current_epoch} : Generator loss: {gen_mean}, Critic loss: {crit_mean}")
            display_progress(condition[0], real[0], fake[0], self.current_epoch)
```

```
gc.collect()

cwgan = CWGAN(in_channels = 1, out_channels = 2 ,learning_rate=2e-4, lambda_recon=100, display_step=10)

trainer = pl.Trainer(max_epochs=150, gpus=-1)

trainer.fit(cwgan, train_loader)

plt.figure(figsize=(30,60))

idx =1

for batch_idx, batch in enumerate(test_loader):

  real, condition = batch

  pred = cwgan.generator(condition).detach().squeeze().permute(1, 2, 0)

  condition  = condition.detach().squeeze(0).permute(1, 2, 0)

  real  = real.detach().squeeze(0).permute(1, 2, 0)

  plt.subplots_adjust(wspace=0, hspace=0)

  plt.subplot(6,3,idx)

  plt.grid(False)

  ab = torch.zeros((224,224,2))

  img = torch.cat([condition * 100, ab], dim=2).numpy()

  imgan = lab2rgb(img)

  plt.imshow(imgan)

  plt.title('Input')

  plt.subplot(6,3,idx + 1)

  ab = torch.zeros((224,224,2))

  imgan = lab_to_rgb(condition,real)

  plt.imshow(imgan)

  plt.title('Real')

  plt.subplot(6,3,idx + 2)

  imgan = lab_to_rgb(condition,pred)

  plt.title('Generated')

  plt.imshow(imgan)

  idx += 3
```

```
    if idx >= 18:
        break
torch.set_grad_enabled(False)
cwgan.generator.eval()
all_preds = []
all_real = []
for batch_idx, batch in enumerate(test_loader):
    real, condition = batch
    pred = cwgan.generator(condition).detach()
    Lab = torch.cat([condition, pred], dim=1).numpy()
    Lab_real = torch.cat([condition, real], dim=1).numpy()
    all_preds.append(Lab.squeeze())
    all_real.append(Lab_real.squeeze())
    if batch_idx == 500:
  break
class InceptionScore:
    def __init__(self, device):
        self.device = device
        self.inception = inception_v3(pretrained=True, transform_input=False).to(self.device)
        self.inception.eval()
    def calculate_is(self, generated_images):
        generated_images = generated_images.to(self.device)
        with torch.no_grad():
            generated_features = self.inception(generated_images.view(-1,3,224,224))
        generated_features = generated_features.view(generated_features.size(0), -1)
        p = F.softmax(generated_features, dim=1)
        kl = p * (torch.log(p) - torch.log(torch.tensor(1.0/generated_features.size(1)).to(self.device)))
        kl = kl.sum(dim=1)
```

```
        return kl.mean().item(), kl.std().item()
device = "cuda" # or "cpu" if you don't have a GPU
is_calculator = InceptionScore(device)
all_preds = np.concatenate(all_preds, axis=0)
all_preds = torch.tensor(all_preds).float()
all_real = np.concatenate(all_real, axis=0)
all_real = torch.tensor(all_real).float()
is_model = InceptionScore(device)
# Calculate the Inception Score
mean_real, std_real = is_model.calculate_is(all_real)
print("Inception Score of real images: mean: {:.4f}, std: {:.4f}".format(mean_real, std_real))
mean_is, std_is = is_model.calculate_is(all_preds)
print("Inception Score of fake images: mean: {:.4f}, std: {:.4f}".format(mean_is, std_is))
```

B. Implemenation of Fréchet Inception Distance (FID)

```
class FID:
    def __init__(self, device):
        self.device = device
        self.inception = inception_v3(pretrained=True, transform_input=False).to(self.device)
        self.inception.eval()
        self.mu = None
        self.sigma = None
    def calculate_fid(self, real_images, generated_images):
        real_images = real_images.to(self.device)
        generated_images = generated_images.to(self.device)
        with torch.no_grad():
            real_features = self.inception(real_images.view(-1,3,224,224))
            generated_features = self.inception(generated_images.view(-1,3,224,224))
```

```
        real_features = real_features.view(real_features.size(0), -1)
        generated_features = generated_features.view(generated_features.size(0), -1)
        if self.mu is None:
            self.mu = real_features.mean(dim=0)
        if self.sigma is None:
            self.sigma = real_features.std(dim=0)
        real_mu = real_features.mean(dim=0)
        real_sigma = real_features.std(dim=0)
        generated_mu = generated_features.mean(dim=0)
        generated_sigma = generated_features.std(dim=0)
        mu_diff = real_mu - generated_mu
        sigma_diff = real_sigma - generated_sigma
        fid = mu_diff.pow(2).sum() + (self.sigma - generated_sigma).pow(2).sum() + (self.mu - generated_mu).pow(2).sum()
        return fid.item()
# Initialize the FID class
device = "cuda" # or "cpu" if you don't have a GPU
fid_calculator = FID(device)
# Calculate the FID
fid_value = fid_calculator.calculate_fid(all_real, all_preds)
print("FID: {:.4f}".format(fid_value))
```