# Data sources:

# **Style Data**

Kaggle: <https://www.kaggle.com/competitions/painter-by-numbers/data>

!ls -lha /home/ec2-user/SageMaker/kaggle.json

!pip install -q kaggle

!mkdir -p ~/.kaggle #Create the directory

!cp kaggle.json ~/.kaggle/

!chmod 600 /home/ec2-user/SageMaker/kaggle.json

!kaggle competitions download -f train.zip -p '/home/ec2-user/SageMaker' -o painter-by-numbers

local\_zip = '/home/ec2-user/SageMaker/train.zip'

zip\_ref = zipfile.ZipFile(local\_zip, 'r')

!mkdir /home/ec2-user/SageMaker/style-data

zip\_ref.extractall('/home/ec2-user/SageMaker/style-data')

zip\_ref.close()

os.remove(local\_zip)

print('The number of images present in WikiArt dataset are:',len(os.listdir('/home/ec2-user/SageMaker/train')))

# **Content Data**

CocoDataset: <https://cocodataset.org/#download> the train2014 dataset from coco was used

!wget --no-check-certificate \

"http://images.cocodataset.org/zips/train2014.zip" \

-O "/home/ec2-user/SageMaker/coco.zip"

local\_zip = '/home/ec2-user/SageMaker/coco.zip'

zip\_ref = zipfile.ZipFile(local\_zip, 'r')

!mkdir /home/ec2-user/SageMaker/content-data

zip\_ref.extractall('/home/ec2-user/SageMaker/content-data')

zip\_ref.close()

os.remove(local\_zip)

print('The number of images present in COCO dataset are:',len(os.listdir('/home/ec2-user/SageMaker/content-data/train2014')))

Data preprocessing:

**Data Transformation and Normalization Pipeline for Image Processing**

# Define the statistics used for normalization

stats = ((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))

# Define the normalization transform

normalize = T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

# Define the batch size

batch\_size = 8

# Define the data transformations for training

tfms = T.Compose([

    T.Resize((512, 512)),  # Resize the image to (512, 512)

    T.RandomCrop((256, 256)),  # Randomly crop the image to (256, 256)

    T.ToTensor(),  # Convert the image to a tensor

    T.Normalize(\*stats, inplace=True)  # Normalize the image with the defined statistics

])

# Define the data transformations for testing

test\_tfms = T.Compose([

    T.Resize((512, 512)),  # Resize the image to (512, 512)

    T.ToTensor(),  # Convert the image to a tensor

    T.Normalize(\*stats, inplace=True)  # Normalize the image with the defined statistics

])

def denormalize(images, means, stds):

    """

    Denormalize the images using the provided mean and standard deviation values.

    Args:

        images (Tensor): Input tensor of images.

        means (tuple): Mean values used for normalization.

        stds (tuple): Standard deviation values used for normalization.

    Returns:

        Tensor: Denormalized images.

    """

    means = torch.tensor(means).reshape(1, 3, 1, 1)  # Reshape means to match the shape of the input images

    stds = torch.tensor(stds).reshape(1, 3, 1, 1)  # Reshape stds to match the shape of the input images

    return images \* stds + means  # Denormalize the images using element-wise multiplication and addition

**Checking for GPU**

def get\_default\_device():

    """Pick GPU if available, else CPU"""

    if torch.cuda.is\_available():

        print("GPU")

        return torch.device('cuda')

    else:

        return torch.device('cpu')

def to\_device(data, device):

    """Move tensor(s) to chosen device"""

    if isinstance(data, (list,tuple)):

        return [to\_device(x, device) for x in data]

    return data.to(device, non\_blocking=True)

class DeviceDataLoader():

    """Wrap a dataloader to move data to a device"""

    def \_\_init\_\_(self, dl, device):

        self.dl = dl

        self.device = device

    def \_\_iter\_\_(self):

        """Yield a batch of data after moving it to device"""

        for b in self.dl:

            yield to\_device(b, self.device)

    def \_\_len\_\_(self):

        """Number of batches"""

        return len(self.dl)

device = get\_default\_device()

device

DL model:

The VGG19 model is a convolutional neural network (CNN) architecture that was introduced by the Visual Geometry Group (VGG) at the University of Oxford. It is a variant of the VGG family of models, which are known for their simplicity and effectiveness in image classification tasks.

Here's a breakdown of the key characteristics of the VGG19 model:

**Architecture:** The VGG19 model consists of 19 layers, including 16 convolutional layers and 3 fully connected layers. The convolutional layers are organized into blocks, with each block containing multiple convolutional layers followed by max-pooling layers for downsampling.

**Deep Features:** VGG19 has a deep architecture, allowing it to learn hierarchical features from input images. The deeper layers of the network capture high-level semantic features, while the shallow layers capture low-level features such as edges and textures.

**Pre-Trained Weights**: Pre-trained weights for the VGG19 model are available in PyTorch, trained on the ImageNet dataset. These pre-trained weights capture the learned representations from a large dataset of images and can be fine-tuned or used as feature extractors for various computer vision tasks.

**Transfer Learning:** The VGG19 model is often used in transfer learning scenarios, where the pre-trained model is adapted to a new task with a different dataset. By leveraging the learned features from the ImageNet dataset, transfer learning with VGG19 can lead to faster convergence and improved performance on downstream tasks, even with limited training data.

**Usage in PyTorch**: In PyTorch, the VGG19 model is available as part of the torchvision library, which provides popular datasets, model architectures, and image transformation utilities for computer vision tasks. By importing the torchvision.models module, users can easily instantiate a VGG19 model and load pre-trained weights using the torchvision.models.vgg19 function.

import torch

import torchvision.models as models

vg19 = vgg19(True)

print(vg19)

Encoder:

**Forward pass of the VGGEncoder.**

**Args:**

**images (Tensor): Input images to be encoded.**

**output\_last\_feature (bool): If True, only the last feature is returned. Otherwise, all intermediate features are returned.**

**Returns:**

**Tensor or Tuple[Tensor]: Encoded features from the VGG encoder. If output\_last\_feature is True, returns the last feature tensor. Otherwise, returns a tuple of feature tensors from each slice.**

class VGGEncoder(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        # Load the VGG19 model with default weights

        vgg = vgg19(weights='DEFAULT').features

        # Define different slices of the VGG model for feature extraction

        self.slice1 = vgg[:2]

        self.slice2 = vgg[2:7]

        self.slice3 = vgg[7:12]

        self.slice4 = vgg[12:21]

        # Set requires\_grad=False for all parameters to freeze the pre-trained weights

        for p in self.parameters():

            p.requires\_grad = False

    def forward(self, images, output\_last\_feature=False):

        # Pass the input images through each slice of the VGG encoder

        h1 = self.slice1(images)

        h2 = self.slice2(h1)

        h3 = self.slice3(h2)

        h4 = self.slice4(h3)

        if output\_last\_feature:

            # Return the last feature tensor

            return h4

        else:

            # Return a tuple of feature tensors from each slice

            return h1, h2, h3, h4

Decoder:

A wrapper of ReflectionPad2d and Conv2d

This class represents a combination of reflection padding and a convolutional layer.

It applies reflection padding to the input and then performs convolution on the padded input.

Optionally, it applies ReLU activation to the output of the convolution.

Args:

in\_channels (int): Number of input channels.

out\_channels (int): Number of output channels.

kernel\_size (int): Size of the convolution kernel. Default is 3.

pad\_size (int): Size of the reflection padding. Default is 1.

activated (bool): Whether to apply activation (ReLU) after convolution. Default is True.

FORWARD PASS OF THE RC MODULE.

ARGS:

X: INPUT TENSOR OF SHAPE (BATCH\_SIZE, IN\_CHANNELS, HEIGHT, WIDTH).

RETURNS:

OUTPUT TENSOR AFTER APPLYING REFLECTION PADDING, CONVOLUTION,

AND ACTIVATION (IF ENABLED) OF SHAPE (BATCH\_SIZE, OUT\_CHANNELS, HEIGHT, WIDTH)

DECODER NETWORK FOR IMAGE RECONSTRUCTION.

THIS NETWORK TAKES FEATURES EXTRACTED BY AN ENCODER NETWORK WITH

ADAPTIVE INSTANCE NORMALIZATION APPLIED USING STYLE FEATURES AND GENERATES A RECONSTRUCTED IMAGE.

THIS MODULE CONSISTS OF A SERIES OF RC (REFLECTIONPAD2D AND CONV2D) LAYERS FOR UPSAMPLING AND IMAGE RECONSTRUCTION.

FORWARD PASS OF THE DECODER MODULE.

ARGS:

FEATURES (TORCH.TENSOR): INPUT FEATURES FROM THE ENCODER MODULE.

RETURNS:

TORCH.TENSOR: OUTPUT TENSOR REPRESENTING THE RECONSTRUCTED IMAGE.

class RC(torch.nn.Module):

    def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=3, pad\_size=1, activated=True):

        super().\_\_init\_\_()

        self.pad = nn.ReflectionPad2d((pad\_size, pad\_size, pad\_size, pad\_size))

        self.conv = nn.Conv2d(in\_channels, out\_channels, kernel\_size)

        self.activated = activated

    def forward(self, x):

        h = self.pad(x)     # Apply reflection padding to the input tensor

        h = self.conv(h)    # Perform convolution on the padded input

        if self.activated:  # Apply ReLU activation if activated is True

            return F.relu(h)

        else:

            return h         # Otherwise, return the output without activation

class Decoder(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.rc1 = RC(512, 256, 3, 1)

        self.rc2 = RC(256, 256, 3, 1)

        self.rc3 = RC(256, 256, 3, 1)

        self.rc4 = RC(256, 256, 3, 1)

        self.rc5 = RC(256, 128, 3, 1)

        self.rc6 = RC(128, 128, 3, 1)

        self.rc7 = RC(128, 64, 3, 1)

        self.rc8 = RC(64, 64, 3, 1)

        self.rc9 = RC(64, 3, 3, 1, False)

    def forward(self, features):

        # Forward pass of the Decoder module for image upsampling and reconstruction

        h = self.rc1(features)

        h = F.interpolate(h, scale\_factor=2)      # Perform upsampling using F.interpolate with scale factor 2

        h = self.rc2(h)

        h = self.rc3(h)

        h = self.rc4(h)

        h = self.rc5(h)

        h = F.interpolate(h, scale\_factor=2)      # Perform another upsampling using F.interpolate with scale factor 2

        h = self.rc6(h)

        h = self.rc7(h)

        h = F.interpolate(h, scale\_factor=2)      # Perform another upsampling using F.interpolate with scale factor 2

        h = self.rc8(h)

        h = self.rc9(h)

        return h

Computing Loss:

The `generate` method generates a stylized image by blending content and style features extracted from input content and style images using Adaptive Instance Normalization (AdaIN) and the decoder. Static methods `calc\_content\_loss` and `calc\_style\_loss` compute the content and style losses, respectively, using mean squared error (MSE) loss between features. In the `forward` method, the model computes the total loss by combining content and style losses with specified weights. The loss is used to optimize the model during training, facilitating the generation of stylized images that combine the content of one image with the style of another.

class Model(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.vgg\_encoder = VGGEncoder()  # Initialize the VGGEncoder to extract content and style features

        self.decoder = Decoder()        # Initialize the Decoder for image reconstruction

    def generate(self, content\_images, style\_images, alpha=1.0):

        content\_features = self.vgg\_encoder(content\_images, output\_last\_feature=True)  # Extract content features

        style\_features = self.vgg\_encoder(style\_images, output\_last\_feature=True)      # Extract style features

        t = adain(content\_features, style\_features)    # Apply Adaptive Instance Normalization (AdaIN) to normalize features

        t = alpha \* t + (1 - alpha) \* content\_features  # Blend content and style features based on alpha

        out = self.decoder(t)                          # Generate the stylized output using the decoder

        return out

    @staticmethod

    def calc\_content\_loss(out\_features, t):

        return F.mse\_loss(out\_features, t)  # Calculate Mean Squared Error (MSE) loss

    @staticmethod

    def calc\_style\_loss(content\_middle\_features, style\_middle\_features):

        loss = 0

        for c, s in zip(content\_middle\_features, style\_middle\_features):

            c\_mean, c\_std = calc\_mean\_std(c)   # Calculate mean and standard deviation of content features

            s\_mean, s\_std = calc\_mean\_std(s)   # Calculate mean and standard deviation of style features

            loss += F.mse\_loss(c\_mean, s\_mean) + F.mse\_loss(c\_std, s\_std)  # Calculate MSE loss between means and stds

        return loss

    def forward(self, content\_images, style\_images, alpha=1.0, lam=10):

        content\_features = self.vgg\_encoder(content\_images, output\_last\_feature=True)  # Extract features from the content image

        style\_features = self.vgg\_encoder(style\_images, output\_last\_feature=True)      # Extract features from the style image.

        t = adain(content\_features, style\_features)    # Apply Adaptive Instance Normalization (AdaIN) to combine features

        t = alpha \* t + (1 - alpha) \* content\_features  # Blend content and style features based on alpha

        out = self.decoder(t)                          # Generate the stylized output using the decoder

        output\_features = self.vgg\_encoder(out, output\_last\_feature=True)    # Extract features from the stylized output

        output\_middle\_features = self.vgg\_encoder(out, output\_last\_feature=False)  # Extract middle-level features from the stylized output

        style\_middle\_features = self.vgg\_encoder(style\_images, output\_last\_feature=False)  # Extract style middle features from the style image

        loss\_c = self.calc\_content\_loss(output\_features, t)     # Calculate content loss

        loss\_s = self.calc\_style\_loss(output\_middle\_features, style\_middle\_features)  # Calculate style loss from the middle-level features from the stylized output and the style image

        loss = loss\_c + lam \* loss\_s  # Combine content and style loss with the specified lambda weight

        return loss