Deep Dream and Neural Style Transfer

Overview

Neural style transfer is an optimization technique used to take three images, a **content** image, a **style reference** image (such as an artwork by a famous painter), and the **input** image you want to style -- and blend them together such that the input image is transformed to look like the content image, but “painted” in the style of the style image.

We will follow the general steps to perform style transfer:

1. Visualize data
2. Basic Preprocessing/preparing our data
3. Set up loss functions
4. Create model
5. Optimize for loss function

Setup

Download Images

```
import os
img_dir = '/tmp/nst'
if not os.path.exists(img_dir):
    os.makedirs(img_dir)
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/commons/d/d7/Gr
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/commons/0/0a/Th
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/commons/b/b4/Va
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/commons/0/00/Tu
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/commons/6/68/Pi
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/commons/thumb/e
```

Import modules

```
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['figure.figsize'] = (10,10)
mpl.rcParams['axes.grid'] = False

import numpy as np
from PIL import Image
import time
import functools
```

```
%tensorflow_version 2.x
import tensorflow as tf

from tensorflow.keras.utils import image_dataset_from_directory as kp_image
from tensorflow.python.keras import models
from tensorflow.python.keras import losses
from tensorflow.python.keras import layers
from tensorflow.python.keras import backend as K
```

```
Colab only includes TensorFlow 2.x; %tensorflow_version has no effect.
```

```
# Set up some global values here
content_path = '/tmp/nst/Green_Sea_Turtle_grazing_seagrass.jpg'
style_path = '/tmp/nst/The_Great_Wave_off_Kanagawa.jpg'
```

Visualize the input

```
from keras.preprocessing.image import img_to_array
from PIL import Image
import numpy as np

def load_img(path_to_img):
    max_dim = 512
    img = Image.open(path_to_img)
    long = max(img.size)
    scale = max_dim / long
    img = img.resize((round(img.size[0] * scale), round(img.size[1] * scale)), Im

    img_array = img_to_array(img)

    # We need to broadcast the image array such that it has a batch dimension
    img_array = np.expand_dims(img_array, axis=0)
    return img_array
```

```
def imshow(img, title=None):
  # Remove the batch dimension
  out = np.squeeze(img, axis=0)
  # Normalize for display
  out = out.astype('uint8')
  plt.imshow(out)
  if title is not None:
    plt.title(title)
  plt.imshow(out)
```

These are input content and style images. We hope to "create" an image with the content of our content image, but with the style of the style image.

```
from PIL import Image
import numpy as np

def load_img_as_uint8(path_to_img):
    # Open the image file using PIL
    with Image.open(path_to_img) as img:
        # Convert the image to RGB if it's not already
        img = img.convert('RGB')
        # Convert the image to a NumPy array
        img_array = np.asarray(img)
        # Ensure the datatype is uint8
        img_array_uint8 = img_array.astype('uint8')
    return img_array_uint8

# Use the function to load your images
content = load_img_as_uint8(content_path)
style = load_img_as_uint8(style_path)
```

```
import matplotlib.pyplot as plt

# Create a figure with two subplots
plt.figure(figsize=(10, 5))  # Width, height in inches

# Display the content image
plt.subplot(1, 2, 1)  # 1 row, 2 columns, 1st subplot
plt.imshow(content)
plt.title('Content Image')
plt.axis('off')  # Hide the axis ticks and labels

# Display the style image
plt.subplot(1, 2, 2)  # 1 row, 2 columns, 2nd subplot
plt.imshow(style)
plt.title('Style Image')
plt.axis('off')  # Hide the axis ticks and labels

# Show the plot
plt.show()
```

Prepare the data

Methods that will allow us to load and preprocess our images easily. We perform the same preprocessing process as are expected according to the VGG training process. VGG networks are trained on image with each channel normalized by `mean = [103.939, 116.779, 123.68]` and with channels BGR.

```
def load_and_process_img(path_to_img):
  img = load_img(path_to_img)
  img = tf.keras.applications.vgg19.preprocess_input(img)
  return img
```

In order to view the outputs of our optimization, we are required to perform the inverse preprocessing step. Furthermore, since our optimized image may take its values anywhere between $-\infty$ and ∞, we must clip to maintain our values from within the 0-255 range.

```
def deprocess_img(processed_img):
  x = processed_img.copy()
  if len(x.shape) == 4:
    x = np.squeeze(x, 0)
  assert len(x.shape) == 3, ("Input to deprocess image must be an image of "
                             "dimension [1, height, width, channel] or [height, w
  if len(x.shape) != 3:
    raise ValueError("Invalid input to deprocessing image")

  # perform the inverse of the preprocessing step
  x[:, :, 0] += 103.939
  x[:, :, 1] += 116.779
  x[:, :, 2] += 123.68
  x = x[:, :, ::-1]

  x = np.clip(x, 0, 255).astype('uint8')
  return x
```

Define content and style representations

For an input image, we will try to match the corresponding style and content target representations at the intermediate layers.

```
# Content layer where will pull our feature maps
content_layers = ['block5_conv2']

# Style layer we are interested in
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1'
               ]

num_content_layers = len(content_layers)
num_style_layers = len(style_layers)
```

Build the Model

In this case, we load VGG19, and feed in our input tensor to the model. This will allow us to extract the feature maps (and subsequently the content and style representations) of the content, style, and generated images.

```
def get_model():
  # Load our model. We load pretrained VGG, trained on imagenet data
  vgg = tf.keras.applications.vgg19.VGG19(include_top=False, weights='imagenet')
  vgg.trainable = False
  # Get output layers corresponding to style and content layers
  style_outputs = [vgg.get_layer(name).output for name in style_layers]
  content_outputs = [vgg.get_layer(name).output for name in content_layers]
  model_outputs = style_outputs + content_outputs
  # Build model
  return models.Model(vgg.input, model_outputs)
```

Content Loss

```
def get_content_loss(base_content, target):
  return tf.reduce_mean(tf.square(base_content - target))
```

Style Loss

```
def gram_matrix(input_tensor):
  # We make the image channels first
  channels = int(input_tensor.shape[-1])
  a = tf.reshape(input_tensor, [-1, channels])
  n = tf.shape(a)[0]
  gram = tf.matmul(a, a, transpose_a=True)
  return gram / tf.cast(n, tf.float32)

def get_style_loss(base_style, gram_target):
  """Expects two images of dimension h, w, c"""
  # height, width, num filters of each layer
  # We scale the loss at a given layer by the size of the feature map and the num
  height, width, channels = base_style.get_shape().as_list()
  gram_style = gram_matrix(base_style)

  return tf.reduce_mean(tf.square(gram_style - gram_target))# / (4. * (channels *
```

Apply style transfer to our images

Run Gradient Descent

```
def get_feature_representations(model, content_path, style_path):

  # Load our images in
  content_image = load_and_process_img(content_path)
  style_image = load_and_process_img(style_path)

  # batch compute content and style features
  style_outputs = model(style_image)
  content_outputs = model(content_image)


  # Get the style and content feature representations from our model
  style_features = [style_layer[0] for style_layer in style_outputs[:num_style_la
  content_features = [content_layer[0] for content_layer in content_outputs[num_s
  return style_features, content_features
```

```
def compute_loss(model, loss_weights, init_image, gram_style_features, content_fe

  style_weight, content_weight = loss_weights

  # Feed our init image through our model. This will give us the content and
  # style representations at our desired layers. Since we're using eager
  # our model is callable just like any other function!
  model_outputs = model(init_image)

  style_output_features = model_outputs[:num_style_layers]
  content_output_features = model_outputs[num_style_layers:]

  style_score = 0
  content_score = 0

  # Accumulate style losses from all layers
  # Here, we equally weight each contribution of each loss layer
  weight_per_style_layer = 1.0 / float(num_style_layers)
  for target_style, comb_style in zip(gram_style_features, style_output_features)
    style_score += weight_per_style_layer * get_style_loss(comb_style[0], target_

  # Accumulate content losses from all layers
  weight_per_content_layer = 1.0 / float(num_content_layers)
  for target_content, comb_content in zip(content_features, content_output_featur
    content_score += weight_per_content_layer* get_content_loss(comb_content[0],

  style_score *= style_weight
  content_score *= content_weight

  # Get total loss
  loss = style_score + content_score
  return loss, style_score, content_score
```

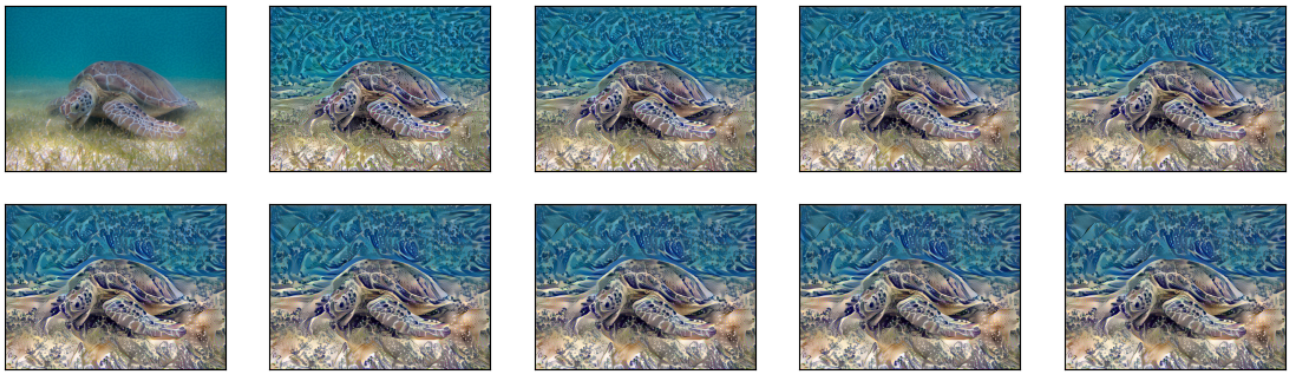
```
def compute_grads(cfg):
  with tf.GradientTape() as tape:
    all_loss = compute_loss(**cfg)
  # Compute gradients wrt input image
  total_loss = all_loss[0]
  return tape.gradient(total_loss, cfg['init_image']), all_loss
```

Optimization loop

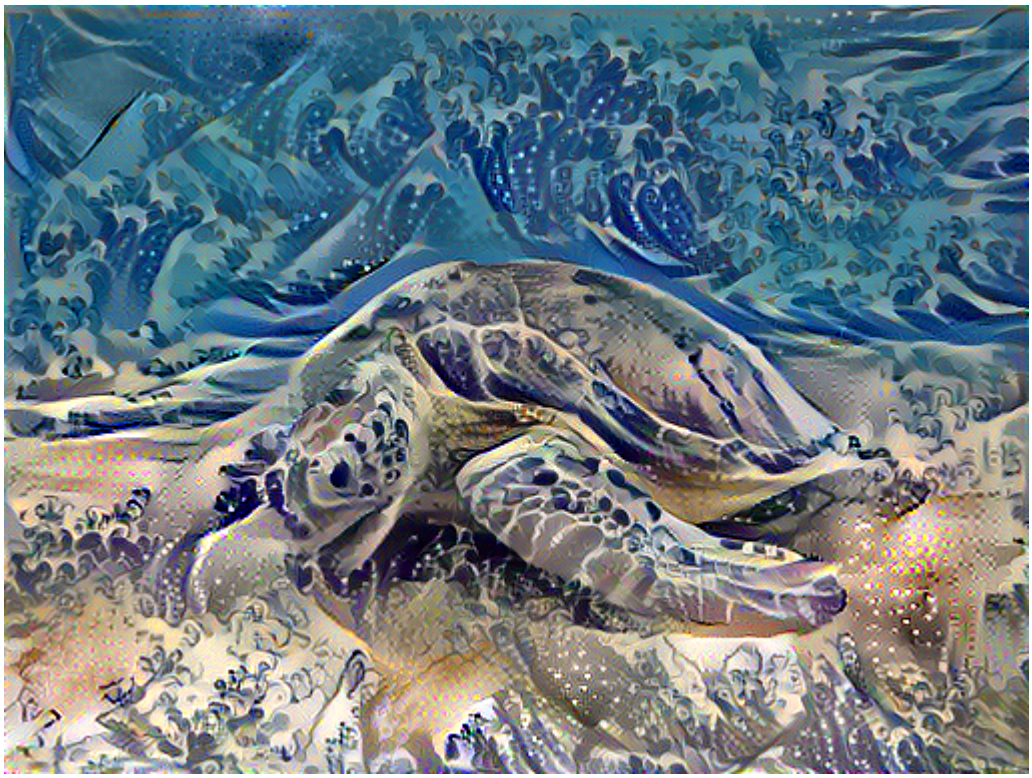
```
import IPython.display

def run_style_transfer(content_path,
                       style_path,
                       num_iterations=1000,
                       content_weight=1e3,
                       style_weight=1e-2):
  # We don't need to (or want to) train any layers of our model, so we set their
  # trainable to false.
  model = get_model()
  for layer in model.layers:
    layer.trainable = False

  # Get the style and content feature representations (from our specified interme
  style_features, content_features = get_feature_representations(model, content_p
  gram_style_features = [gram_matrix(style_feature) for style_feature in style_fe

  # Set initial image
  init_image = load_and_process_img(content_path)
  init_image = tf.Variable(init_image, dtype=tf.float32)
  # Create our optimizer
  opt = tf.optimizers.Adam(learning_rate=5, epsilon=1e-1)

  # For displaying intermediate images
  iter_count = 1

  # Store our best result
  best_loss, best_img = float('inf'), None

  # Create a nice config
  loss_weights = (style_weight, content_weight)
  cfg = {
      'model': model,
      'loss_weights': loss_weights,
      'init_image': init_image,
      'gram_style_features': gram_style_features,
      'content_features': content_features
  }

  # For displaying
  num_rows = 2
  num_cols = 5
  display_interval = num_iterations/(num_rows*num_cols)
  start_time = time.time()
  global_start = time.time()

  norm_means = np.array([103.939, 116.779, 123.68])
  min_vals = -norm_means
  max_vals = 255 - norm_means

  imgs = []
  for i in range(num_iterations):
    grads, all_loss = compute_grads(cfg)
    loss, style_score, content_score = all_loss
    opt.apply_gradients([(grads, init_image)])
```

```
    clipped = tf.clip_by_value(init_image, min_vals, max_vals)
    init_image.assign(clipped)
    end_time = time.time()

    if loss < best_loss:
      # Update best loss and best image from total loss.
      best_loss = loss
      best_img = deprocess_img(init_image.numpy())

    if i % display_interval== 0:
      start_time = time.time()

      # Use the .numpy() method to get the concrete numpy array
      plot_img = init_image.numpy()
      plot_img = deprocess_img(plot_img)
      imgs.append(plot_img)
      IPython.display.clear_output(wait=True)
      IPython.display.display_png(Image.fromarray(plot_img))
      print('Iteration: {}'.format(i))
      print('Total loss: {:.4e}, '
            'style loss: {:.4e}, '
            'content loss: {:.4e}, '
            'time: {:.4f}s'.format(loss, style_score, content_score, time.time()
  print('Total time: {:.4f}s'.format(time.time() - global_start))
  IPython.display.clear_output(wait=True)
  plt.figure(figsize=(14,4))
  for i,img in enumerate(imgs):
      plt.subplot(num_rows,num_cols,i+1)
      plt.imshow(img)
      plt.xticks([])
      plt.yticks([])

  return best_img, best_loss
```

```
# Assign the paths to variables for better readability
content_image_path = content_path
style_image_path = style_path

# Define the number of iterations as a variable
iterations = 1000

# Call the style transfer function with the variables
best, best_loss = run_style_transfer(content_image_path, style_image_path, num_it
```



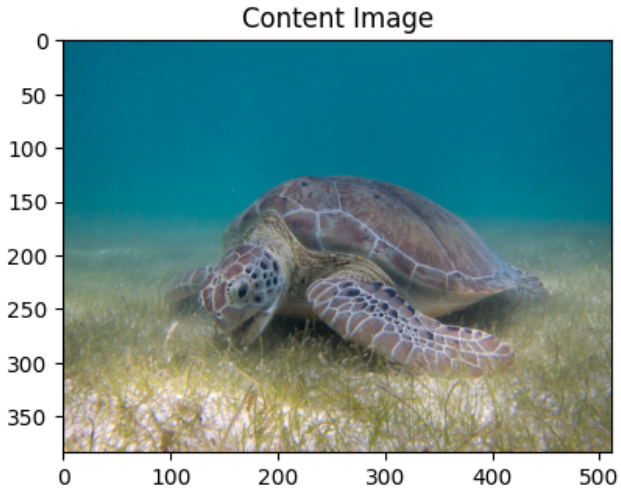
```
Image.fromarray(best)
```

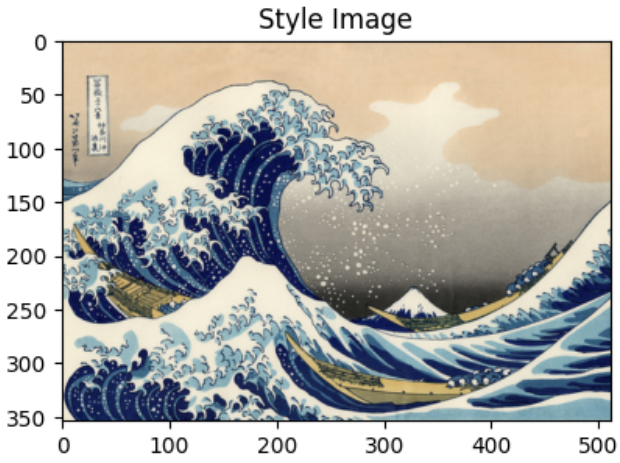


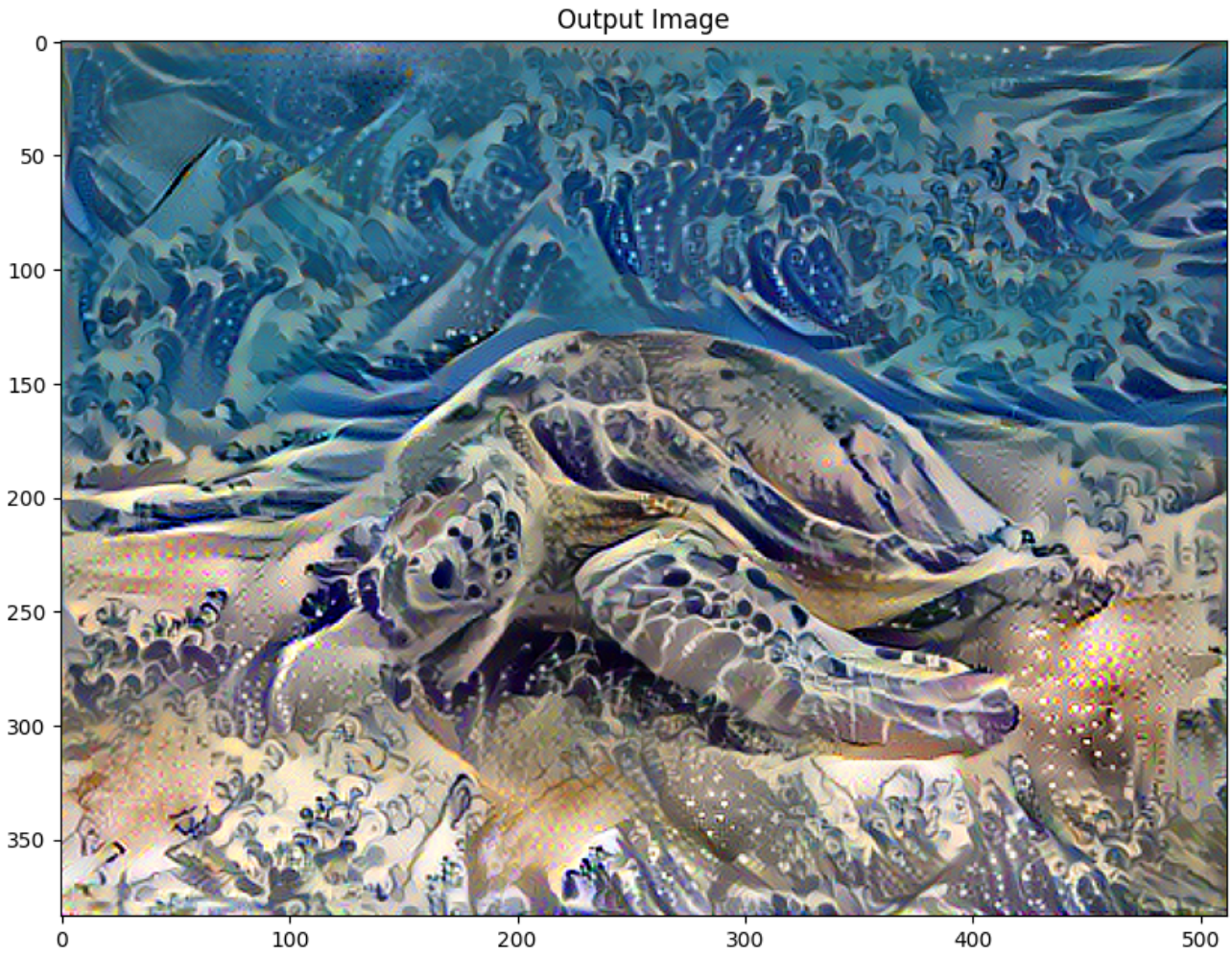
Visualize outputs

```
def show_results(best_img, content_path, style_path, show_large_final=True):
  plt.figure(figsize=(10, 5))
  content = load_img(content_path)
  style = load_img(style_path)

  plt.subplot(1, 2, 1)
  imshow(content, 'Content Image')

  plt.subplot(1, 2, 2)
  imshow(style, 'Style Image')

  if show_large_final:
    plt.figure(figsize=(10, 10))

    plt.imshow(best_img)
    plt.title('Output Image')
    plt.show()
```

```
show_results(best, content_path, style_path)
```


Content Image
0
50
100
150
200
250
300
350
0
100
200
300
400
500


Style Image
0
50
100
150
200
250
300
350
0
100
200
300
400
500


Output Image
0
50
100
150
200
250
300
350
0
100
200
300
400
500

Deep Dream

Objective :

1. Experiment with different models(VGG16, Inception V3)
2. Tweak image
3. Exposing different layers in model to visualise different types of image
4. Try image space gradient ascent for visualisation
5. Vary pyramid size
6. Vary pyramid ratio
7. Vary noise
8. Vary lr

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

```
Mounted at /content/drive
```

```
import os
import enum
from collections import namedtuple
import argparse
import numbers
import math

import torch
import torch.nn as nn
from torchvision import models
from torchvision import transforms
import torch.nn.functional as F

import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt  # visualizations
```

Setting up the environment and configurations

```
# The 2 datasets we'll be using
class SupportedPretrainedWeights(enum.Enum):
    IMAGENET = 0
    PLACES_365 = 1


# The 2 models we'll be using
class SupportedModels(enum.Enum):
    VGG16_EXPERIMENTAL = 0,
    RESNET50 = 1


# Commonly used paths, let's define them here as constants
DATA_DIR_PATH = os.path.join(os.getcwd(), 'data')
INPUT_DATA_PATH = os.path.join(DATA_DIR_PATH, 'input')
BINARIES_PATH = os.path.join(os.getcwd(), 'models', 'binaries')
OUT_IMAGES_PATH = os.path.join(DATA_DIR_PATH, 'out-images')

# Make sure these exist as the rest of the code relies on it
os.makedirs(BINARIES_PATH, exist_ok=True)
os.makedirs(OUT_IMAGES_PATH, exist_ok=True)

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")  # checking

# Images will be normalized using these, because the CNNs were trained with norma
IMAGENET_MEAN_1 = np.array([0.485, 0.456, 0.406], dtype=np.float32)
IMAGENET_STD_1 = np.array([0.229, 0.224, 0.225], dtype=np.float32)
```

Experimentation using VGG16

VGG16 Activations : fetching and preparing the desired model with the specified pretrained weights, exposing intermediate feature maps from specific layers.

```
class Vgg16Experimental(torch.nn.Module):

    def __init__(self, pretrained_weights, requires_grad=False, show_progress=Fal
        super().__init__()

        # Only ImageNet weights are supported for now for this model
        if pretrained_weights == SupportedPretrainedWeights.IMAGENET.name:
            vgg16 = models.vgg16(pretrained=True, progress=show_progress).eval()
        else:
            raise Exception(f'Pretrained weights {pretrained_weights} not yet sup

        vgg_pretrained_features = vgg16.features

        # Exposed the best/most interesting layers
        self.layer_names = ['relu3_3', 'relu4_1', 'relu4_2', 'relu4_3', 'relu5_1'

        # 31 layers in total for the VGG16
        self.conv1_1 = vgg_pretrained_features[0]
        self.relu1_1 = vgg_pretrained_features[1]
        self.conv1_2 = vgg_pretrained_features[2]
        self.relu1_2 = vgg_pretrained_features[3]
        self.max_pooling1 = vgg_pretrained_features[4]
        self.conv2_1 = vgg_pretrained_features[5]
        self.relu2_1 = vgg_pretrained_features[6]
        self.conv2_2 = vgg_pretrained_features[7]
        self.relu2_2 = vgg_pretrained_features[8]
        self.max_pooling2 = vgg_pretrained_features[9]
        self.conv3_1 = vgg_pretrained_features[10]
        self.relu3_1 = vgg_pretrained_features[11]
        self.conv3_2 = vgg_pretrained_features[12]
        self.relu3_2 = vgg_pretrained_features[13]
        self.conv3_3 = vgg_pretrained_features[14]
        self.relu3_3 = vgg_pretrained_features[15]
        self.max_pooling3 = vgg_pretrained_features[16]
        self.conv4_1 = vgg_pretrained_features[17]
        self.relu4_1 = vgg_pretrained_features[18]
        self.conv4_2 = vgg_pretrained_features[19]
        self.relu4_2 = vgg_pretrained_features[20]
        self.conv4_3 = vgg_pretrained_features[21]
        self.relu4_3 = vgg_pretrained_features[22]
        self.max_pooling4 = vgg_pretrained_features[23]
        self.conv5_1 = vgg_pretrained_features[24]
        self.relu5_1 = vgg_pretrained_features[25]
        self.conv5_2 = vgg_pretrained_features[26]
        self.relu5_2 = vgg_pretrained_features[27]
        self.conv5_3 = vgg_pretrained_features[28]
        self.relu5_3 = vgg_pretrained_features[29]
        self.max_pooling5 = vgg_pretrained_features[30]

        # Turning these off because we'll be using a pretrained network
        #not updating weights during optimization process, retain knowledge learn
        if not requires_grad:
            for param in self.parameters():
                param.requires_grad = False
```

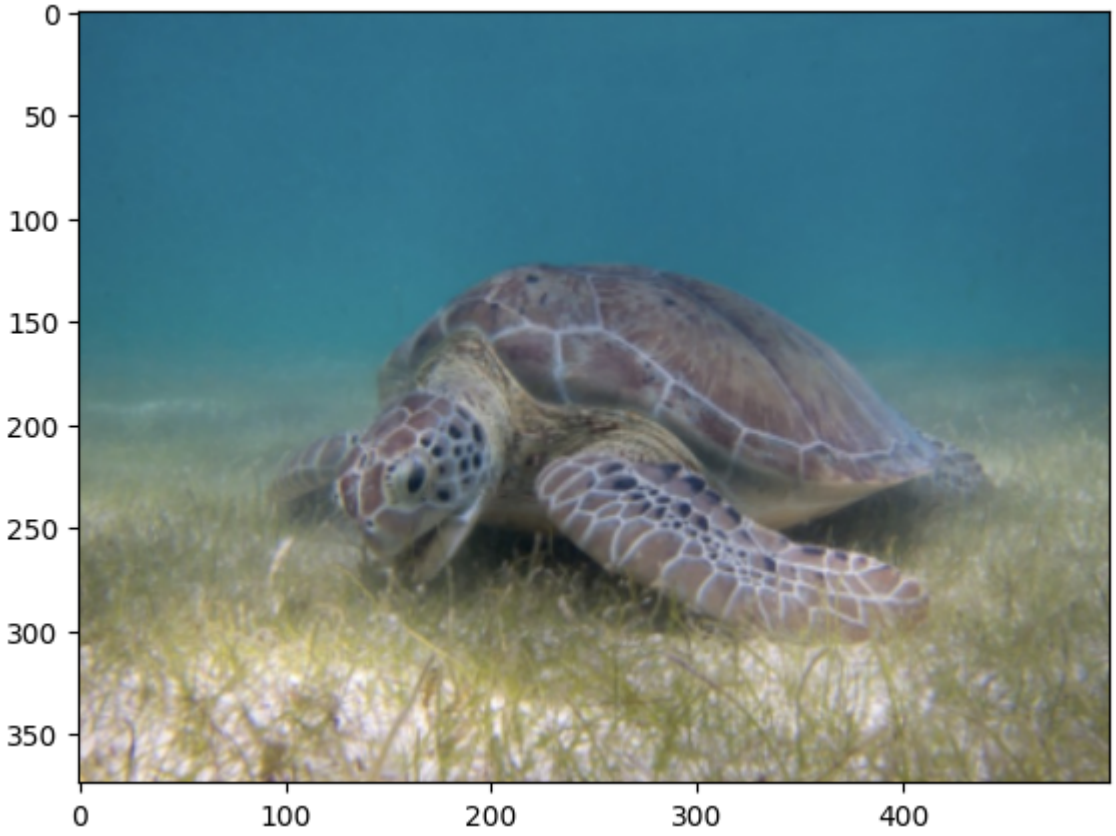
```
    # Expose every single layer during the forward pass
    def forward(self, x):
        x = self.conv1_1(x)
        conv1_1 = x
        x = self.relu1_1(x)
        relu1_1 = x
        x = self.conv1_2(x)
        conv1_2 = x
        x = self.relu1_2(x)
        relu1_2 = x
        x = self.max_pooling1(x)
        x = self.conv2_1(x)
        conv2_1 = x
        x = self.relu2_1(x)
        relu2_1 = x
        x = self.conv2_2(x)
        conv2_2 = x
        x = self.relu2_2(x)
        relu2_2 = x
        x = self.max_pooling2(x)
        x = self.conv3_1(x)
        conv3_1 = x
        x = self.relu3_1(x)
        relu3_1 = x
        x = self.conv3_2(x)
        conv3_2 = x
        x = self.relu3_2(x)
        relu3_2 = x
        x = self.conv3_3(x)
        conv3_3 = x
        x = self.relu3_3(x)
        relu3_3 = x
        x = self.max_pooling3(x)
        x = self.conv4_1(x)
        conv4_1 = x
        x = self.relu4_1(x)
        relu4_1 = x
        x = self.conv4_2(x)
        conv4_2 = x
        x = self.relu4_2(x)
        relu4_2 = x
        x = self.conv4_3(x)
        conv4_3 = x
        x = self.relu4_3(x)
        relu4_3 = x
        x = self.max_pooling4(x)
        x = self.conv5_1(x)
        conv5_1 = x
        x = self.relu5_1(x)
        relu5_1 = x
        x = self.conv5_2(x)
        conv5_2 = x
        x = self.relu5_2(x)
        relu5_2 = x
        x = self.conv5_3(x)
```

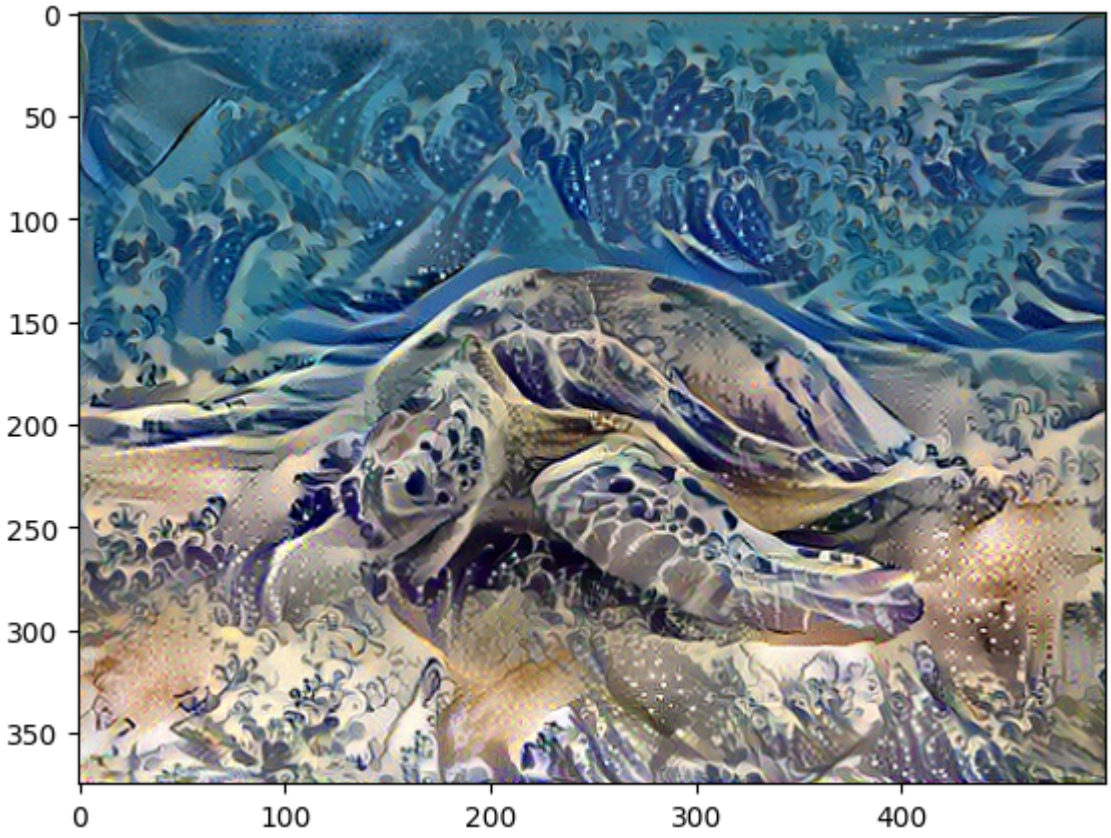
```
        conv5_3 = x
        x = self.relu5_3(x)
        relu5_3 = x
        mp5 = self.max_pooling5(x)

        # Expose only the layers that we want to experiment with here
        vgg_outputs = namedtuple("VggOutputs", self.layer_names)
        out = vgg_outputs(relu3_3, relu4_1, relu4_2, relu4_3, relu5_1, relu5_2, r

        return out
#tch a neural network model based on the specified type and pretrained weights. I

def fetch_and_prepare_model(model_type, pretrained_weights):
    if model_type == SupportedModels.VGG16_EXPERIMENTAL.name:
        model = Vgg16Experimental(pretrained_weights, requires_grad=False, show_p
    elif model_type == SupportedModels.RESNET50.name:
        # We'll define the ResNet50 later
        model = ResNet50(pretrained_weights, requires_grad=False, show_progress=T
    else:
        raise Exception('Model not yet supported.')
    return model
```

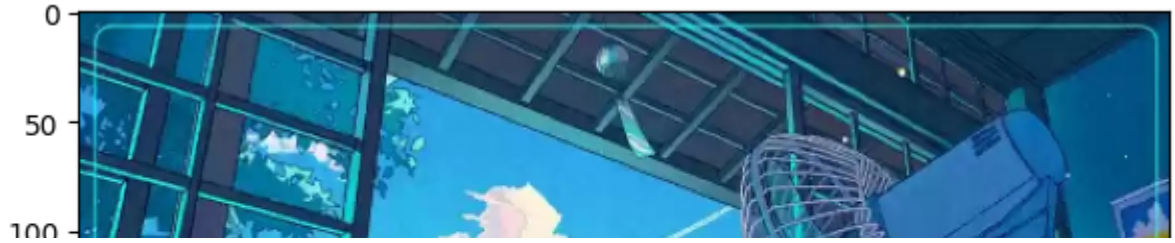
```
#def load_image(img_path, target_shape=None):
    if not os.path.exists(img_path):
        raise Exception(f'Path does not exist: {img_path}')
    img = cv.imread(img_path)[:, :, ::-1]  # [:, :, ::-1] converts BGR (opencv fo

    if target_shape is not None:  # resizing
        if isinstance(target_shape, int) and target_shape != -1:  # scalar -> imp
            current_height, current_width = img.shape[:2]
            new_width = target_shape
            new_height = int(current_height * (new_width / current_width))
            img = cv.resize(img, (new_width, new_height), interpolation=cv.INTER_
        else:  # set both dimensions to target shape
            img = cv.resize(img, (target_shape[1], target_shape[0]), interpolatio

    # This need to go after resizing - otherwise cv.resize will push values outsi
    img = img.astype(np.float32)  # convert from uint8 to float32
    img /= 255.0  # get to [0, 1] range
    return img


# config is just a shared dictionary that you'll be seeing used everywhere, but w
# For the time being think of it as an oracle - whatever the function needs - con
def save_and_maybe_display_image(config, dump_img, name_modifier=None):
    assert isinstance(dump_img, np.ndarray), f'Expected numpy array got {type(dum

    # Step 1: figure out the dump dir location
    dump_dir = config['dump_dir']
    os.makedirs(dump_dir, exist_ok=True)

    # Step 2: define the output image name
    if name_modifier is not None:
        dump_img_name = str(name_modifier).zfill(6) + '.jpg'
    else:
        dump_img_name = build_image_name(config)

    if dump_img.dtype != np.uint8:
        dump_img = (dump_img*255).astype(np.uint8)

    # Step 3: write image to the file system
    # ::-1 because opencv expects BGR (and not RGB) format...
    dump_path = os.path.join(dump_dir, dump_img_name)
    cv.imwrite(dump_path, dump_img[:, :, ::-1])

    # Step 4: potentially display/plot the image
    if config['should_display']:
        fig = plt.figure(figsize=(7.5,5), dpi=100)  # otherwise plots are really
        plt.imshow(dump_img)
        plt.show()

    return dump_path


# This function makes sure we can later reconstruct the image using the informati
# Again don't worry about all the arguments we'll define them later
def build_image_name(config):
```

```
    input_name = 'rand_noise' if config['use_noise'] else config['input'].split('
    layers = '_'.join(config['layers_to_use'])
    img_name = f'{input_name}_width_{config["img_width"]}_model_{config["model_na
    return img_name
```

```
def load_image(img_path, target_shape=None):
    if not os.path.exists(img_path):
        raise Exception(f'Path does not exist: {img_path}')
    img = cv.imread(img_path)[:, :, ::-1]

    if target_shape is not None:
        if isinstance(target_shape, int) and target_shape != -1:
            current_height, current_width = img.shape[:2]
            new_width = target_shape
            new_height = int(current_height * (new_width / current_width))
            img = cv.resize(img, (new_width, new_height), interpolation=cv.INTER_
        else:
            img = cv.resize(img, (target_shape[1], target_shape[0]), interpolatio

    img = img.astype(np.float32)
    img /= 255.0
    return img

def visualize_image(img):
    fig = plt.figure(figsize=(7.5, 5), dpi=100)
    plt.imshow(img)
    plt.show()

# Define img_width before using it
img_width = 500


# Load and visualize the second image
input_img_name3 = '/content/deepdreamold.jpeg'  # Provide the path to your second
img_path3 = os.path.join(INPUT_DATA_PATH, input_img_name3)
img3 = load_image(img_path3, target_shape=img_width)
visualize_image(img3)

# Load and visualize the first image
input_img_name1 = '/content/deepdream1.jpeg'
img1 = load_image(input_img_name1, target_shape=img_width)
visualize_image(img1)

# Load and visualize the second image
input_img_name2 = '/content/deepdreamtry2.jpeg'  # Provide the path to your secon
img_path2 = os.path.join(INPUT_DATA_PATH, input_img_name2)
img2 = load_image(img_path2, target_shape=img_width)
visualize_image(img2)
```


0
50
100
150
200
250
300
350
0
100
200
300
400


0
50
100
150
200
250
300
350
0
100
200
300
400


0
50
100

Preprocessing and postprocessing of images

Introducing random circular spatial shifts to add randomness to the algorithm

1. Normalise the mean and standard deviation values from the ImageNet dataset.
2. De-normalise the image using the mean and standard deviation values from the ImageNet dataset.
3. Clip the values to ensure they are within the [0, 1] range.
4. Expand the dimensions to match the expected shape (1, 3, H, W) and set requires_grad to True for gradient tracking.

```
# ImageNet's mean and std capture the statistics of natural images.

# Function to pre-process a numpy image by normalizing it
def pre_process_numpy_img(img):
    assert isinstance(img, np.ndarray), f'Expected numpy image got {type(img)}'

    # Normalize image using ImageNet mean and standard deviation
    img = (img - IMAGENET_MEAN_1) / IMAGENET_STD_1  # normalize image
    return img


# Function to post-process a numpy image by de-normalizing it
def post_process_numpy_img(img):
    assert isinstance(img, np.ndarray), f'Expected numpy image got {type(img)}'

    # If channel-first format, move to channel-last (CHW -> HWC)
    if img.shape[0] == 3:
        img = np.moveaxis(img, 0, 2)

    mean = IMAGENET_MEAN_1.reshape(1, 1, -1)
    std = IMAGENET_STD_1.reshape(1, 1, -1)
    img = (img * std) + mean  # de-normalize
    img = np.clip(img, 0., 1.)  # make sure it's in the [0, 1] range

    return img


# Transform a numpy image into a PyTorch tensor
def pytorch_input_adapter(img):
    # shape = (1, 3, H, W)
    tensor = transforms.ToTensor()(img).to(DEVICE).unsqueeze(0)
    tensor.requires_grad = True  # collect gradients for the input image
    return tensor

# Transform a PyTorch tensor into a numpy image
def pytorch_output_adapter(tensor):
    # convert from (1, 3, H, W) tensor into (H, W, 3) numpy image
    return np.moveaxis(tensor.to('cpu').detach().numpy()[0], 0, 2)


# Adds randomness to the algorithm and makes the results more diverse
def random_circular_spatial_shift(tensor, h_shift, w_shift, should_undo=False):
    if should_undo:
        h_shift = -h_shift
        w_shift = -w_shift
    with torch.no_grad():
        rolled = torch.roll(tensor, shifts=(h_shift, w_shift), dims=(2, 3))
        rolled.requires_grad = True
        return rolled
```

Image Pyramid

Image is progressively downsampled to create a pyramid of different resolutions.

The purpose of an image pyramid is to create a multi-scale representation of an image, with different levels of resolution. Each level in the pyramid represents the image at a different scale, where higher levels have lower resolution and cover larger areas, while lower levels have higher resolution and cover smaller areas.

```
# Function to calculate the new dimensions (shape) of an image in a pyramid struc
def get_new_shape(config, original_shape, current_pyramid_level):
    SHAPE_MARGIN = 10

    # Extract configuration parameters
    pyramid_ratio = config['pyramid_ratio']
    pyramid_size = config['pyramid_size']

    # Calculate the exponent for scaling the original shape based on the current
    exponent = current_pyramid_level - pyramid_size + 1

    # Compute the new shape by scaling the original shape using the pyramid_ratio
    new_shape = np.round(np.float32(original_shape) * (pyramid_ratio**exponent)).

    if new_shape[0] < SHAPE_MARGIN or new_shape[1] < SHAPE_MARGIN:
        print(f'Pyramid size {config["pyramid_size"]} with pyramid ratio {config[
        print(f'Please change the parameters.')
        exit(0)

    return new_shape
```

The function appears to be designed for a single static image. For video sequences, additional modifications may be needed.

The core of the DeepDream algorithm, the gradient ascent step, is performed in a loop for a specified number of iterations. The gradient_ascent function, which contains the details of this step, is not provided in the code snippet.

The spatial shift introduced during iterations adds stochasticity to the algorithm, creating more diverse and interesting visual patterns.

The resizing of the image at different pyramid levels contributes to the multiscale approach of the DeepDream algorithm.

The final deep-dreamed image is returned after post-processing.

```
def deep_dream_static_image(config, img=None):
    # Initialize the neural network model
    model = fetch_and_prepare_model(config['model_name'], config['pretrained_weig

    try:
        # Extract indices of the layers to be used for Deep Dream
        layer_ids_to_use = [model.layer_names.index(layer_name) for layer_name in
    except Exception as e:
        # Handle invalid layer names
        print(f'Invalid layer names {[layer_name for layer_name in config["layers
        print(f'Available layers for model {config["model_name"]} are {model.laye
        return

    # Load or initialize the input image
    if img is None:
        img_path = os.path.join(INPUT_DATA_PATH, config['input'])
        # Load a numpy, [0, 1] range, channel-last, RGB image
        img = load_image(img_path, target_shape=config['img_width'])
        if config['use_noise']:
            # Generate a random noise image if specified
            shape = img.shape
            img = np.random.uniform(low=0.0, high=1.0, size=shape).astype(np.floa

    # Preprocess the input image
    img = pre_process_numpy_img(img)
    original_shape = img.shape[:-1]  # Save initial height and width

    # Loop through different pyramid levels
    for pyramid_level in range(config['pyramid_size']):
        # Calculate the new shape for the current pyramid level
        new_shape = get_new_shape(config, original_shape, pyramid_level)
        # Resize the image based on the new shape
        img = cv.resize(img, (new_shape[1], new_shape[0]))
        # Convert the resized image to a trainable tensor
        input_tensor = pytorch_input_adapter(img)

        # Iterate through gradient ascent steps
        for iteration in range(config['num_gradient_ascent_iterations']):
            # Introduce spatial randomness for diversity
            h_shift, w_shift = np.random.randint(-config['spatial_shift_size'], c
            input_tensor = random_circular_spatial_shift(input_tensor, h_shift, w

            # Apply the Deep Dream algorithm using gradient ascent
            gradient_ascent(config, model, input_tensor, layer_ids_to_use, iterat

            # Undo the spatial shift to maintain consistency
            input_tensor = random_circular_spatial_shift(input_tensor, h_shift, w

        # Convert the tensor back to a numpy image for the next pyramid level
        img = pytorch_output_adapter(input_tensor)

    # Postprocess the final image
    return post_process_numpy_img(img)
```

Gradient Ascent Implementation for Deep Dream:

```
LOWER_IMAGE_BOUND = torch.tensor((-IMAGENET_MEAN_1 / IMAGENET_STD_1).reshape(1, -
UPPER_IMAGE_BOUND = torch.tensor(((1 - IMAGENET_MEAN_1) / IMAGENET_STD_1).reshape

def gradient_ascent(config, model, input_tensor, layer_ids_to_use, iteration):
    # Step 0: Forward pass to get model output
    out = model(input_tensor)

    # Step 1: Extract activations/feature maps of interest from specified layers
    activations = [out[layer_id_to_use] for layer_id_to_use in layer_ids_to_use]

    # Step 2: Calculate loss over activations
    losses = []
    for layer_activation in activations:
        # Calculate mean squared error (MSE) loss
        loss_component = torch.nn.MSELoss(reduction='mean')(layer_activation, tor
        losses.append(loss_component)

    # Aggregate losses and calculate mean
    loss = torch.mean(torch.stack(losses))
    # Backward pass to compute gradients
    loss.backward()

    # Step 3: Process image gradients (smoothing + normalization)
    grad = input_tensor.grad.data

    # Apply Gaussian smoothing to gradients for visual enhancement
    sigma = ((iteration + 1) / config['num_gradient_ascent_iterations']) * 2.0 +
    # Use a CascadeGaussianSmoothing function with a kernel size of 9 for visual
    smooth_grad = CascadeGaussianSmoothing(kernel_size=9, sigma=sigma)(grad)

    # Normalize the gradients (make them have mean = 0 and std = 1)
    g_std = torch.std(smooth_grad)
    g_mean = torch.mean(smooth_grad)
    smooth_grad = smooth_grad - g_mean
    smooth_grad = smooth_grad / g_std

    # Step 4: Update image using the calculated gradients (gradient ascent step)
    input_tensor.data += config['lr'] * smooth_grad

    # Step 5: Clear gradients and clamp the data to a valid range
    input_tensor.grad.data.zero_()
    input_tensor.data = torch.clamp(input_tensor.data, LOWER_IMAGE_BOUND, UPPER_I
```

Cascaded Gaussian Smoothing

- Cascaded Gaussian smoothing allows for the extraction of features at multiple scales. Fine details are captured by the filters with smaller standard deviations, while larger standard deviations capture broader features. '

reduce noise and create a more blurred or smoothed version of an image.

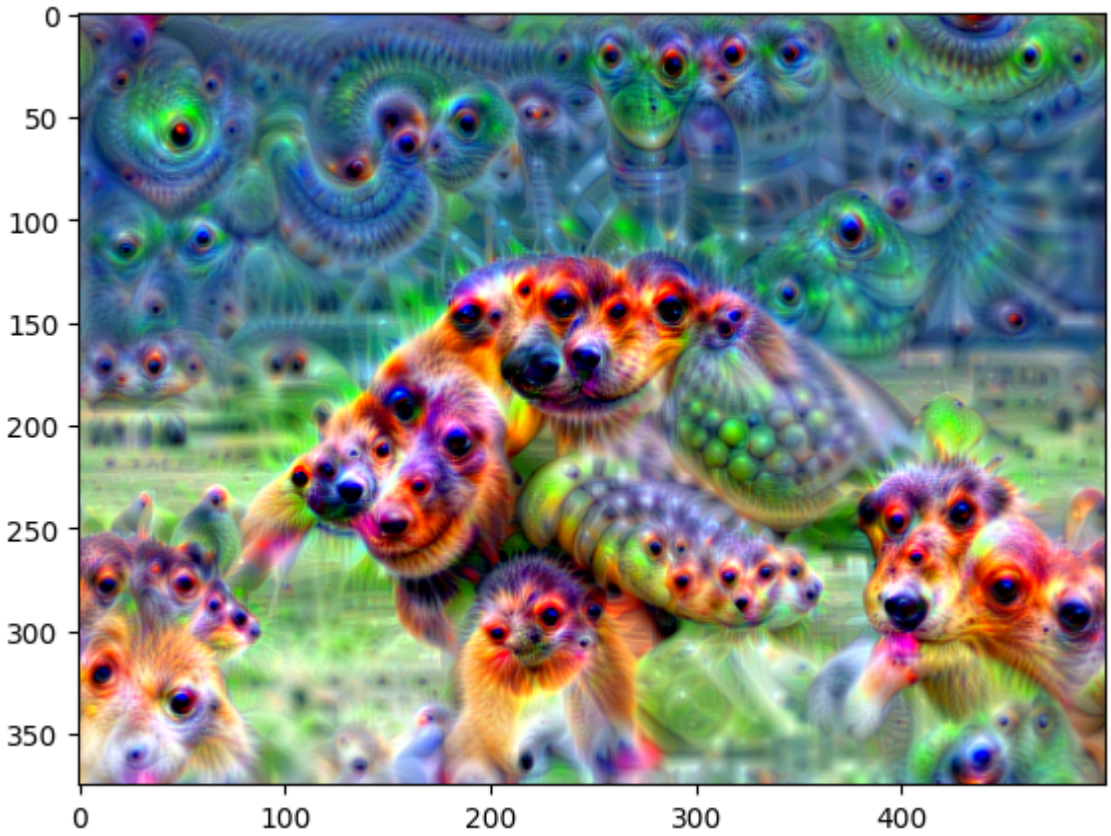
```
class CascadeGaussianSmoothing(nn.Module):

    def __init__(self, kernel_size, sigma):
        super().__init__()

        if isinstance(kernel_size, numbers.Number):
            kernel_size = [kernel_size, kernel_size]

        cascade_coefficients = [0.5, 1.0, 2.0]  # Use 3 different Gaussian kernel
        sigmas = [[coeff * sigma, coeff * sigma] for coeff in cascade_coefficient

        self.pad = int(kernel_size[0] / 2)  # assure we have the same spatial res

        # The gaussian kernel is the product of the gaussian function of each dim
        kernels = []
        meshgrids = torch.meshgrid([torch.arange(size, dtype=torch.float32) for s
        for sigma in sigmas:
            kernel = torch.ones_like(meshgrids[0])
            for size_1d, std_1d, grid in zip(kernel_size, sigma, meshgrids):
                mean = (size_1d - 1) / 2
                kernel *= 1 / (std_1d * math.sqrt(2 * math.pi)) * torch.exp(-((gr
            kernels.append(kernel)

        gaussian_kernels = []
        for kernel in kernels:
            # Normalize - make sure sum of values in gaussian kernel equals 1.
            kernel = kernel / torch.sum(kernel)
            # Reshape to depthwise convolutional weight
            kernel = kernel.view(1, 1, *kernel.shape)
            kernel = kernel.repeat(3, 1, 1, 1)
            kernel = kernel.to(DEVICE)

            gaussian_kernels.append(kernel)

        self.weight1 = gaussian_kernels[0]
        self.weight2 = gaussian_kernels[1]
        self.weight3 = gaussian_kernels[2]
        self.conv = F.conv2d

    def forward(self, input):
        input = F.pad(input, [self.pad, self.pad, self.pad, self.pad], mode='refl

        # Apply Gaussian kernels depthwise over the input
        # shape = (1, 3, H, W) -> (1, 3, H, W)
        num_in_channels = input.shape[1]
        grad1 = self.conv(input, weight=self.weight1, groups=num_in_channels)
        grad2 = self.conv(input, weight=self.weight2, groups=num_in_channels)
        grad3 = self.conv(input, weight=self.weight3, groups=num_in_channels)

        return (grad1 + grad2 + grad3) / 3
```

Argument Parsing

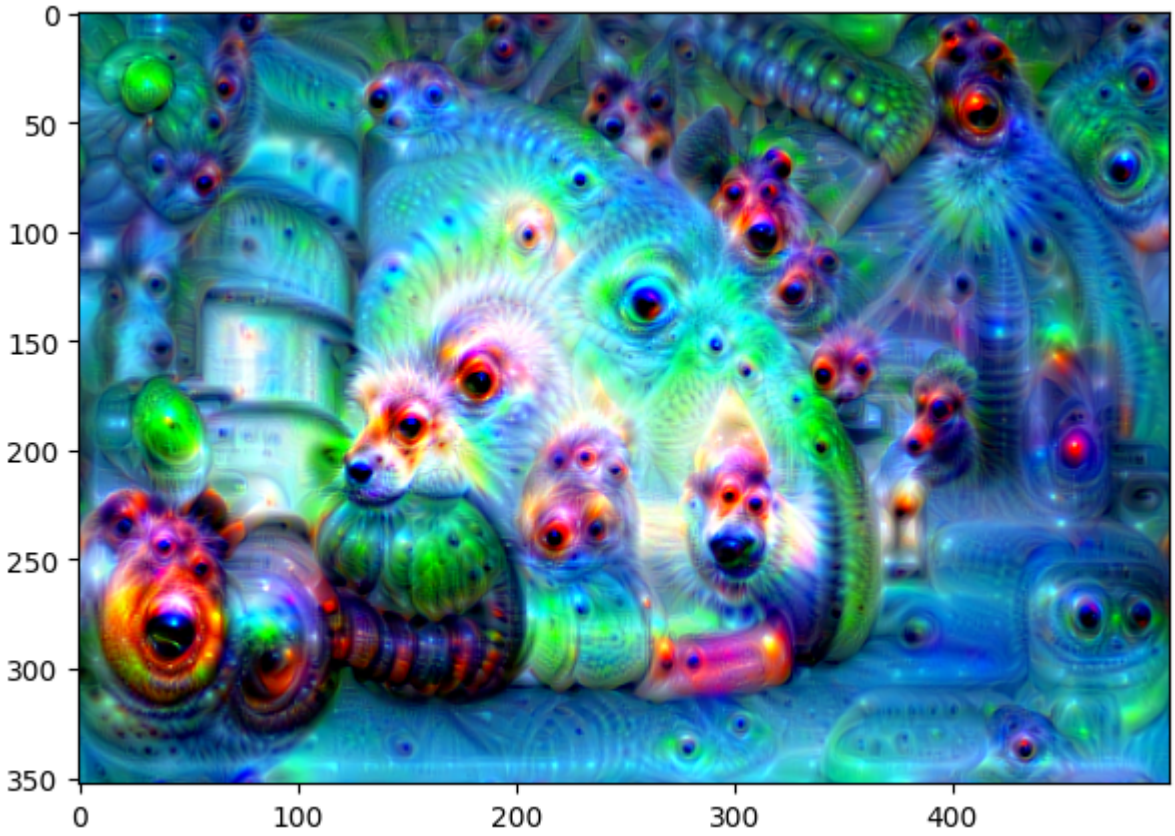
```
# Only a small subset is exposed by design to avoid cluttering
parser = argparse.ArgumentParser()

# Common params
parser.add_argument("--input", type=str, help="Input IMAGE or VIDEO name that wil
parser.add_argument("--img_width", type=int, help="Resize input image to this wid
parser.add_argument("--layers_to_use", type=str, nargs='+', help="Layer whose act
parser.add_argument("--model_name", choices=[m.name for m in SupportedModels],
                    help="Neural network (model) to use for dreaming", default=Su
parser.add_argument("--pretrained_weights", choices=[pw.name for pw in SupportedP
                    help="Pretrained weights to use for the above model", default

# Main params for experimentation (especially pyramid_size and pyramid_ratio)
parser.add_argument("--pyramid_size", type=int, help="Number of images in an imag
parser.add_argument("--pyramid_ratio", type=float, help="Ratio of image sizes in
parser.add_argument("--num_gradient_ascent_iterations", type=int, help="Number of
parser.add_argument("--lr", type=float, help="Learning rate i.e. step size in gra

# You usually won't need to change these as often
parser.add_argument("--should_display", type=bool, help="Display intermediate dre
parser.add_argument("--spatial_shift_size", type=int, help='Number of pixels to r
parser.add_argument("--smoothing_coefficient", type=float, help='Directly control
parser.add_argument("--use_noise", type=bool, help="Use noise as a starting point
args = parser.parse_args('')

# Wrapping configuration into a dictionary
config = dict()
for arg in vars(args):
    config[arg] = getattr(args, arg)
config['dump_dir'] = os.path.join(OUT_IMAGES_PATH, f'{config["model_name"]}_{conf
config['input'] = os.path.basename(config['input'])  # handle absolute and relati
```

```
# Apply deep dream to the second image
config['input'] = input_img_name3
result_img3 = deep_dream_static_image(config, img3)
config['should_display'] = True
dump_path3 = save_and_maybe_display_image(config, result_img3)
print(f'Saved DeepDream static image for {input_img_name3} to: {os.path.relpath(d

# Apply deep dream to the first image
config['input'] = input_img_name1
result_img1 = deep_dream_static_image(config, img1)
config['should_display'] = True
dump_path1 = save_and_maybe_display_image(config, result_img1)
print(f'Saved DeepDream static image for {input_img_name1} to: {os.path.relpath(d

# Apply deep dream to the second image
config['input'] = input_img_name2
result_img2 = deep_dream_static_image(config, img2)
config['should_display'] = True
dump_path2 = save_and_maybe_display_image(config, result_img2)
print(f'Saved DeepDream static image for {input_img_name2} to: {os.path.relpath(d
```


```
Saved DeepDream static image for /content/deepdreamold.jpeg to: deepdreamold_
```




```
Saved DeepDream static image for /content/deepdream1.jpeg to: deepdream1_widtl
```

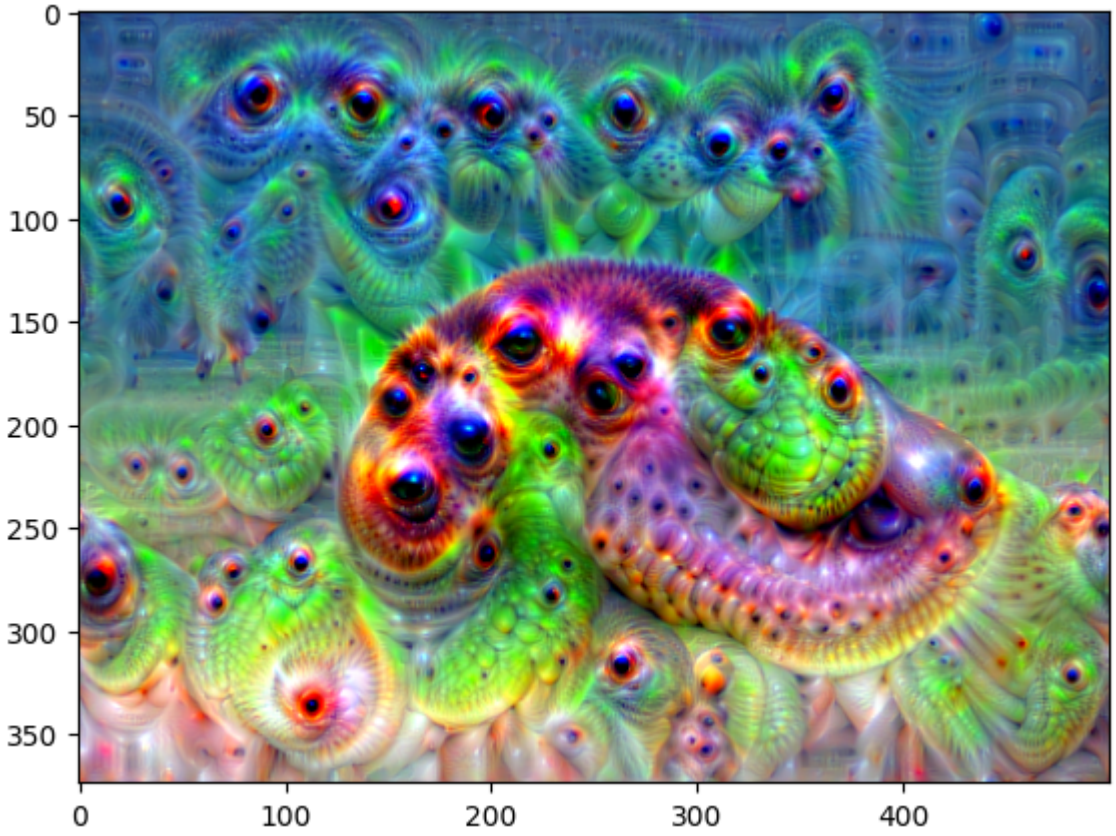



```
Saved DeepDream static image for /content/deepdreamtry2.jpeg to: deepdreamtry
```

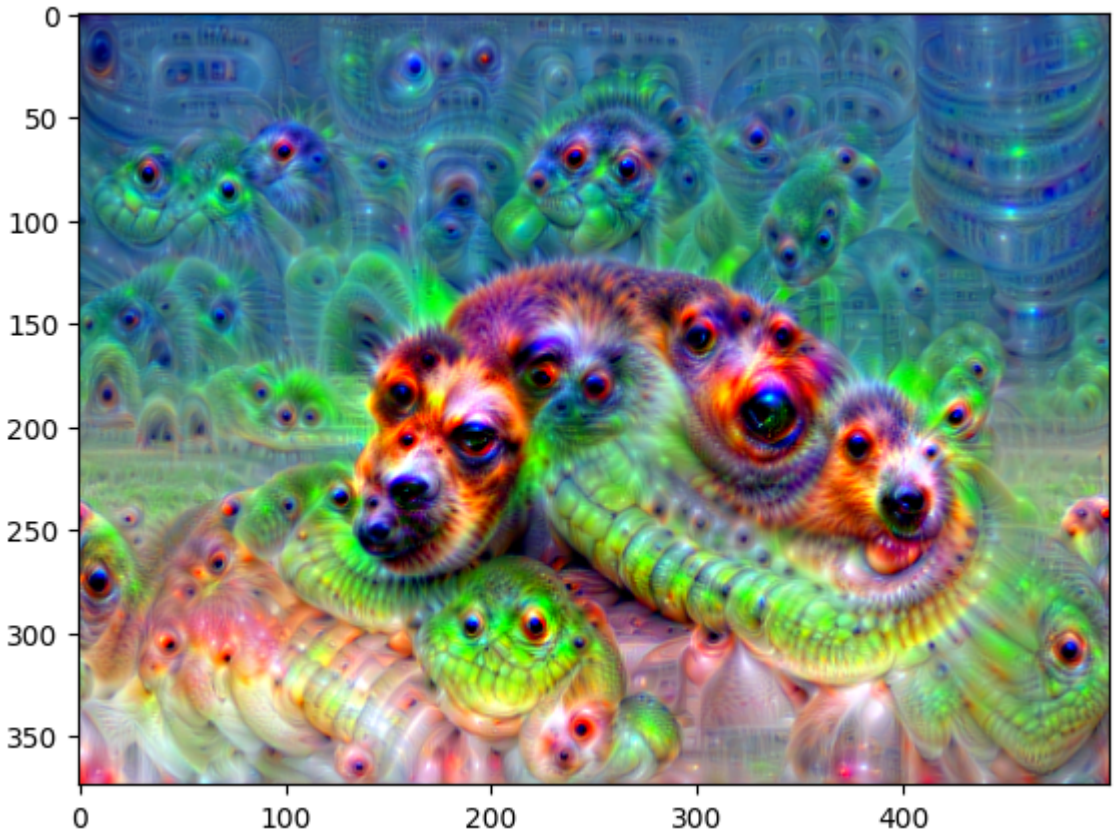
Tweaking the image

Adding noise

```
# Define configurations
config1 = config.copy()  # Assuming config is defined somewhere in your code
config1['use_noise'] = True

config2 = config.copy()
config2['use_noise'] = False
# Apply deep dream to img2 with noise
result_img3_noise = deep_dream_static_image(config1, img3)
dump_path3_noise = save_and_maybe_display_image(config1, result_img3_noise)
print(f'Saved DeepDream static image for before style transfer image with noise t

# Apply deep dream to img2 without noise
result_img3_no_noise = deep_dream_static_image(config2, img3)
dump_path3_no_noise = save_and_maybe_display_image(config2, result_img3_no_noise)
print(f'Saved DeepDream static image for before style transfer image without nois


# Apply deep dream to img1 with noise
result_img1_noise = deep_dream_static_image(config1, img1)
dump_path1_noise = save_and_maybe_display_image(config1, result_img1_noise)
print(f'Saved DeepDream static image for after style transfer image with noise to

# Apply deep dream to img1 without noise
result_img1_no_noise = deep_dream_static_image(config2, img1)
dump_path1_no_noise = save_and_maybe_display_image(config2, result_img1_no_noise)
print(f'Saved DeepDream static image for after style transfer image without noise

# Apply deep dream to img2 with noise
result_img2_noise = deep_dream_static_image(config1, img2)
dump_path2_noise = save_and_maybe_display_image(config1, result_img2_noise)
print(f'Saved DeepDream static image for img3 with noise to: {os.path.relpath(dum

# Apply deep dream to img2 without noise
result_img2_no_noise = deep_dream_static_image(config2, img2)
dump_path2_no_noise = save_and_maybe_display_image(config2, result_img2_no_noise)
print(f'Saved DeepDream static image for img2 without noise to: {os.path.relpath(
```




```
Saved DeepDream static image for before style transfer image with noise to: da
```



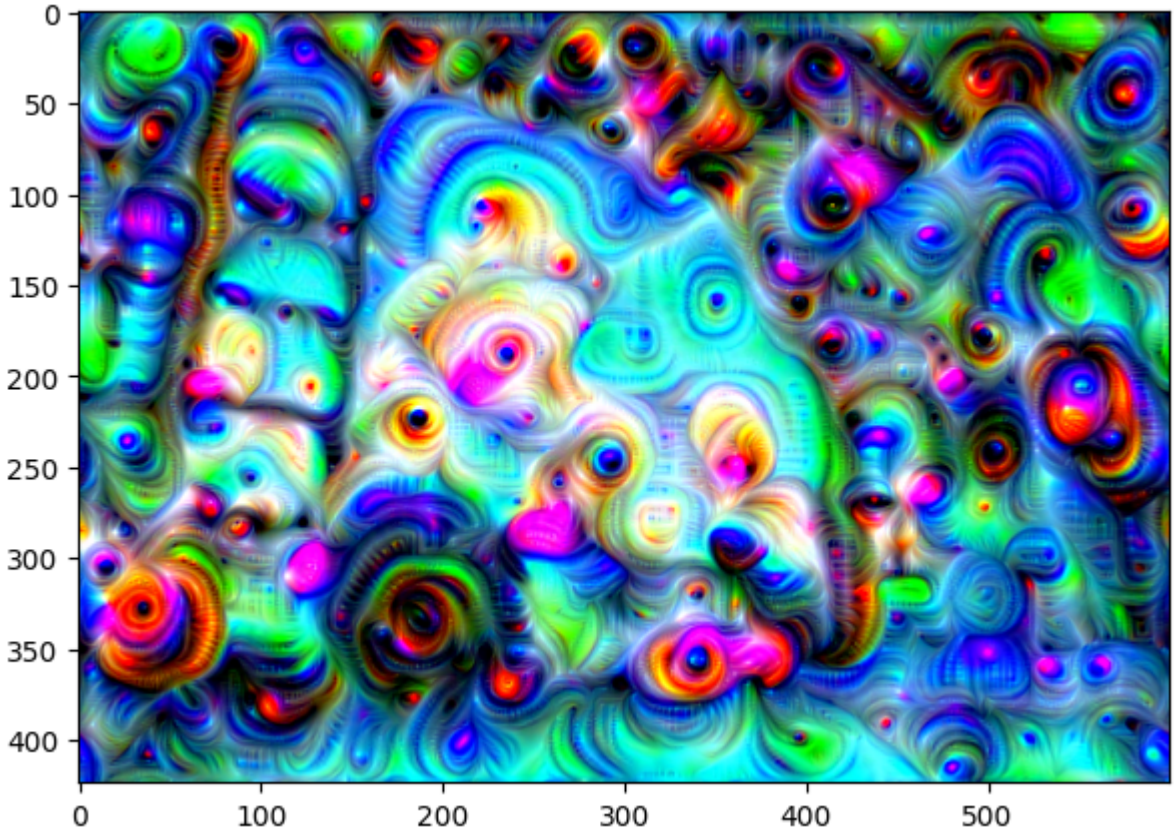

```
Saved DeepDream static image for before style transfer image without noise to
```

Visualize the DeepDream effect on different layers of the VGG16 model.

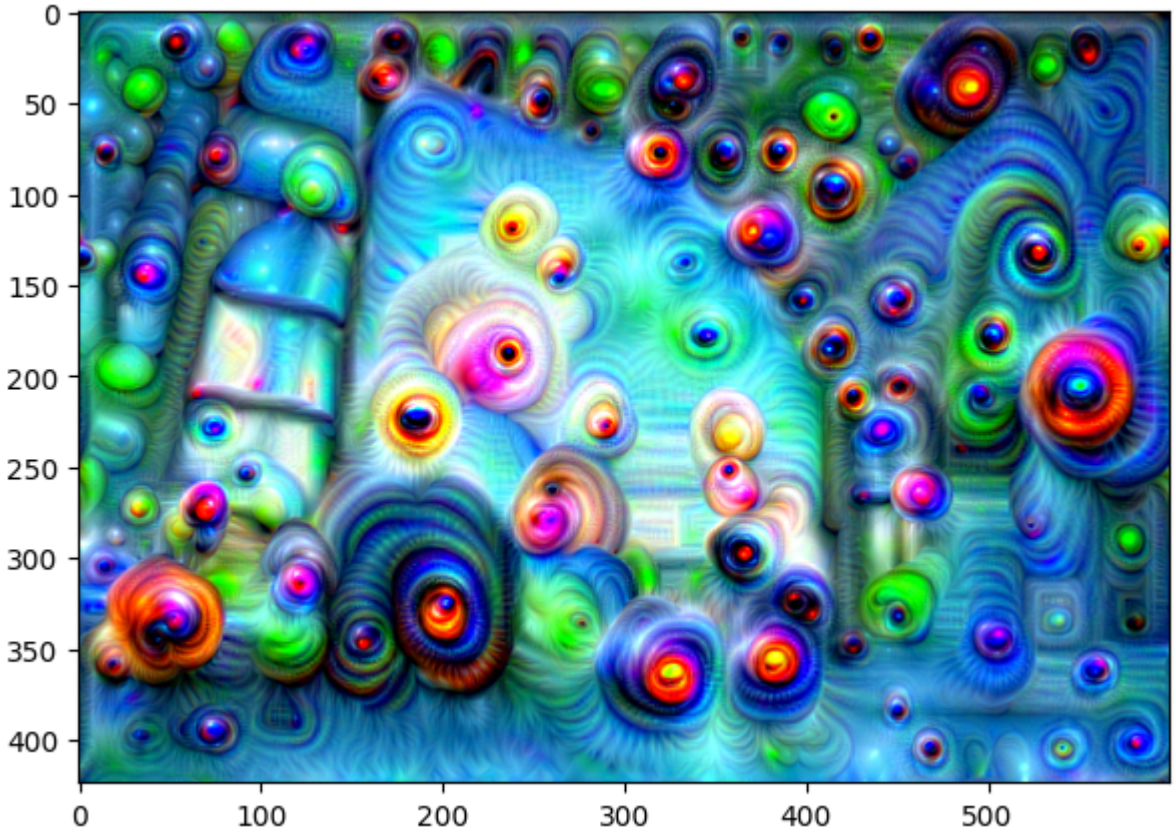
During the dreaming process, optimizing activations in these layers can lead to the generation of images that emphasize or enhance features at different levels of abstraction, providing insights into what patterns the neural network finds interesting or important in the input image.

```
exposed_layers = Vgg16Experimental(pretrained_weights=SupportedPretrainedWeights.
exposed_layers = exposed_layers[:-3]  # truncating last 3 layers
print(f'Exposed layers = {exposed_layers}')

for layer in exposed_layers:
    config['layers_to_use'] = [layer]

    img = deep_dream_static_image(config)
    dump_path = save_and_maybe_display_image(config, img)
    print(f'Saved DeepDream static image to: {os.path.relpath(dump_path)}\n')

config['layers_to_use'] = ['relu4_3']  # keep the config consistent fr
```

```
Exposed layers = ['relu3_3', 'relu4_1', 'relu4_2', 'relu4_3', 'relu5_1']
```



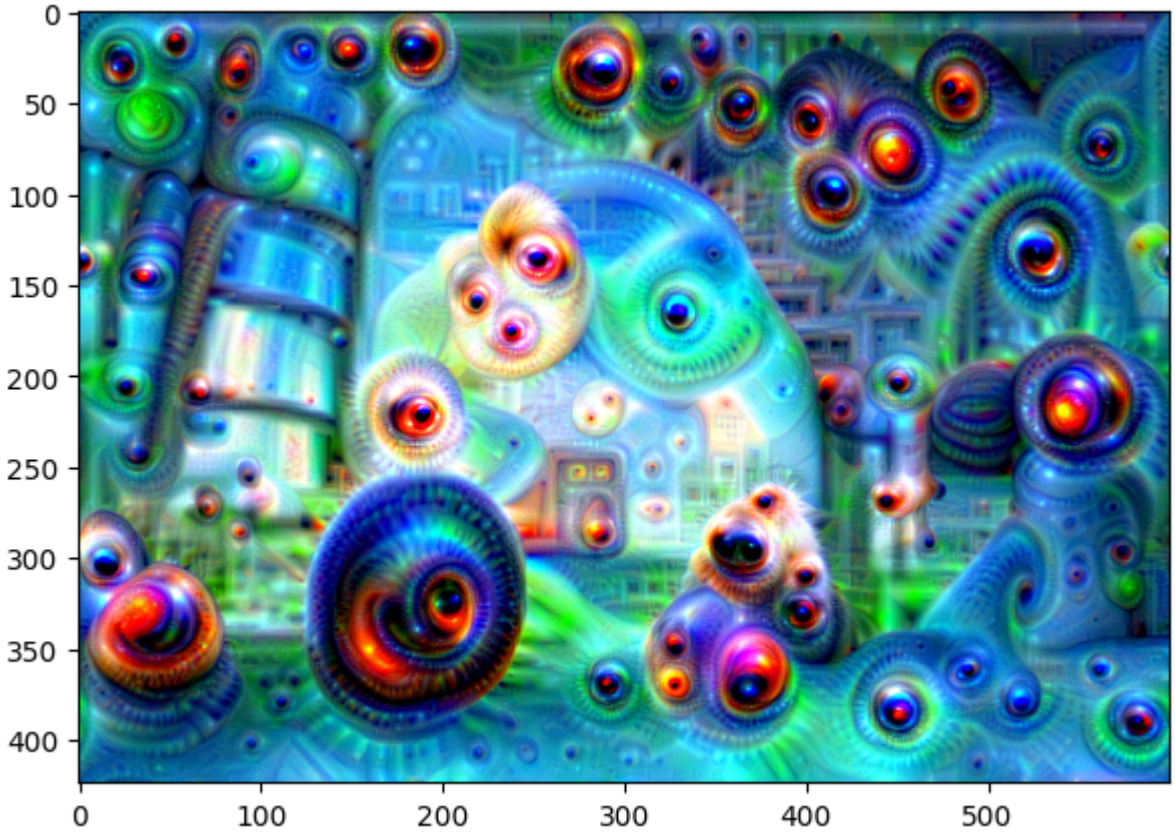

```
Saved DeepDream static image to: deepdreamtry2_width_600_model_VGG16_EXPERIMEN
```



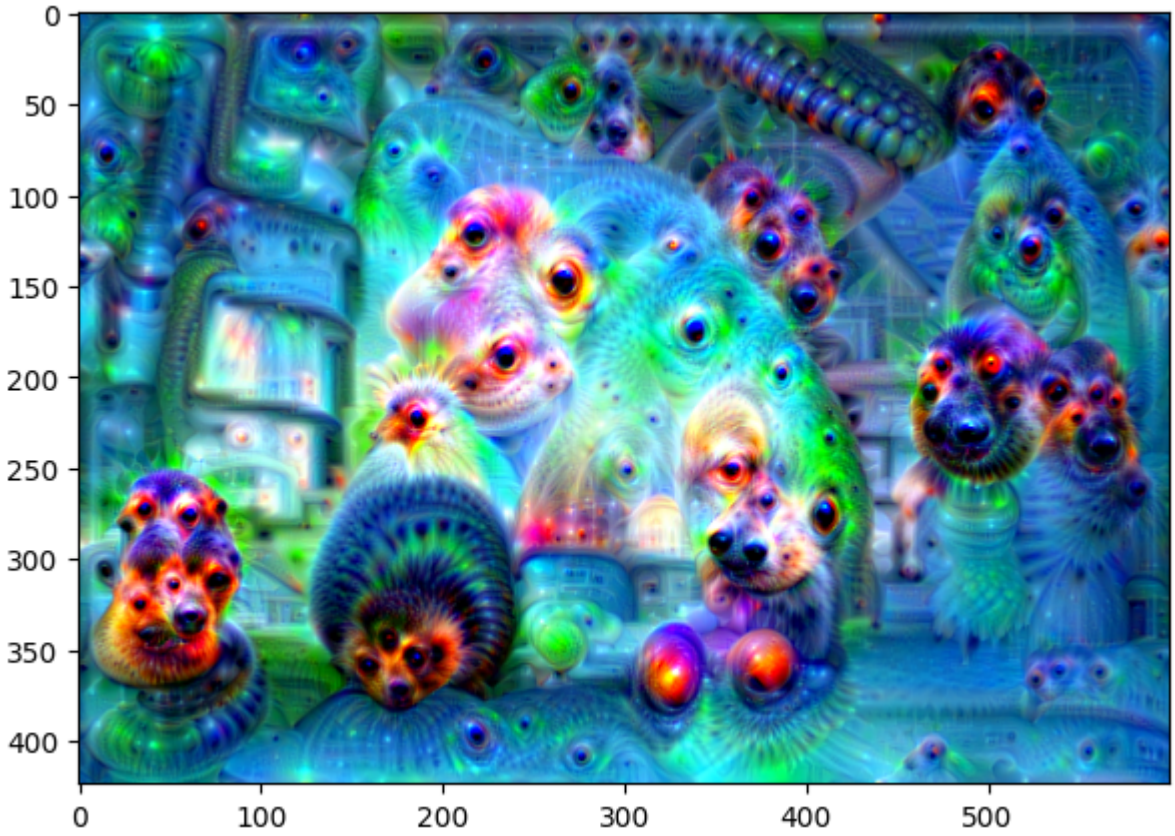

```
Saved DeepDream static image to: deepdreamtry2_width_600_model_VGG16_EXPERIMEN
```

```
exposed_layers = Vgg16Experimental(pretrained_weights=SupportedPretrainedWeights.
exposed_layers = exposed_layers[:-3]
print(f'Exposed layers = {exposed_layers}')

for layer in exposed_layers:
    config['layers_to_use'] = [layer]

    img1 = deep_dream_static_image(config)
    dump_path = save_and_maybe_display_image(config, img1)
    print(f'Saved DeepDream static image to: {os.path.relpath(dump_path)}\n')

config['layers_to_use'] = ['relu4_3']  # keep the config consistent fr
```


```
Saved DeepDream static image to: deepdreamtry2_width_600_model_VGG16_EXPERIMEN
```

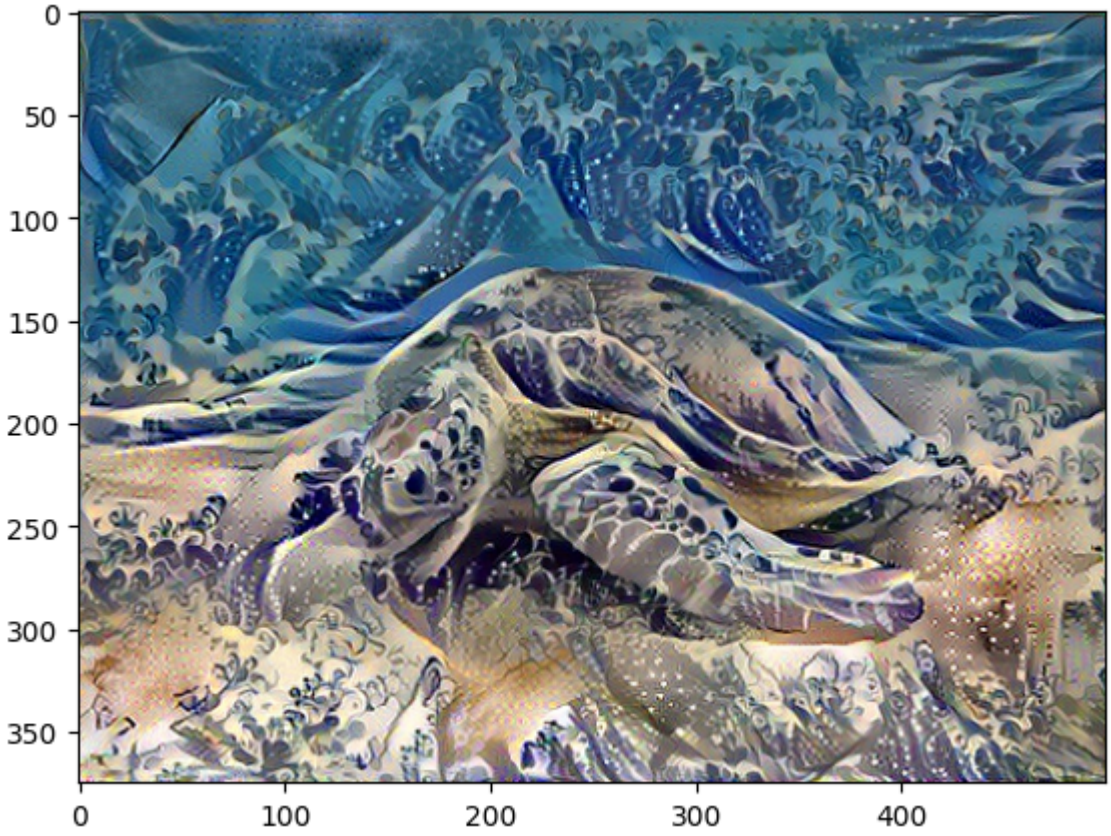


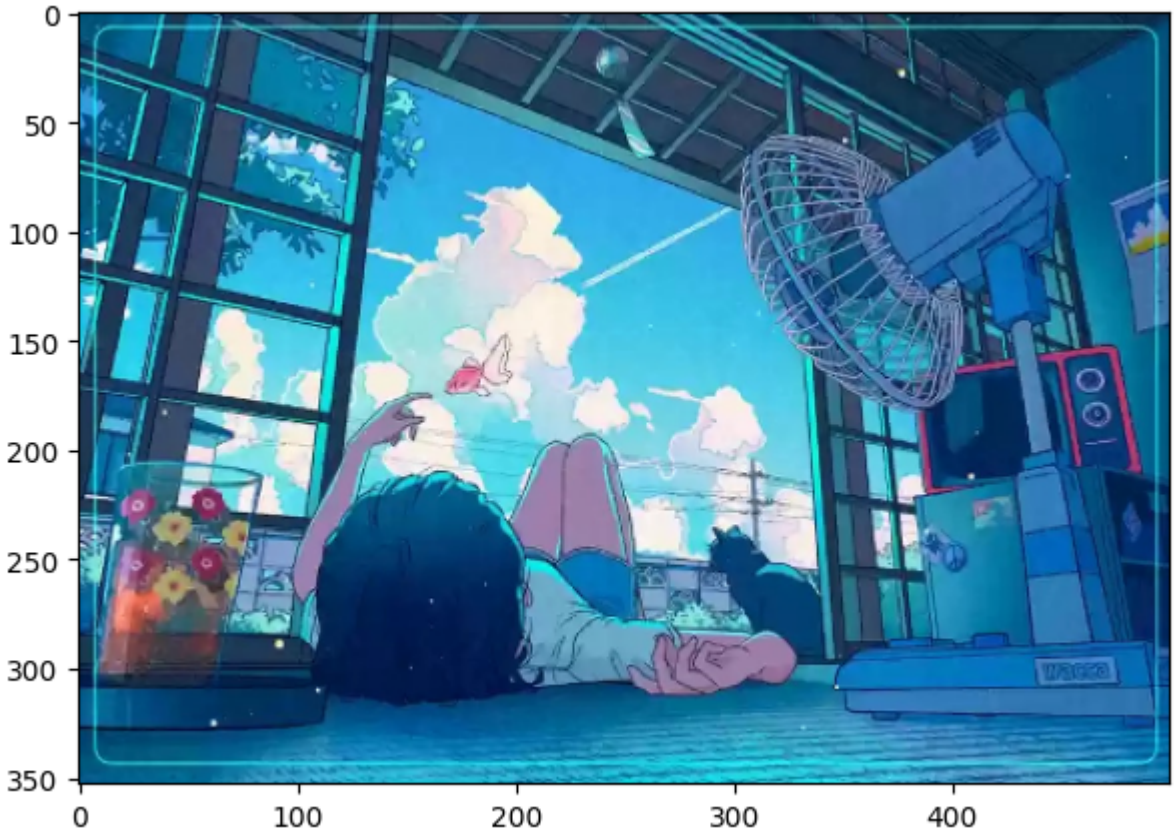

```
Saved DeepDream static image to: deepdreamtry2_width_600_model_VGG16_EXPERIMEN
```



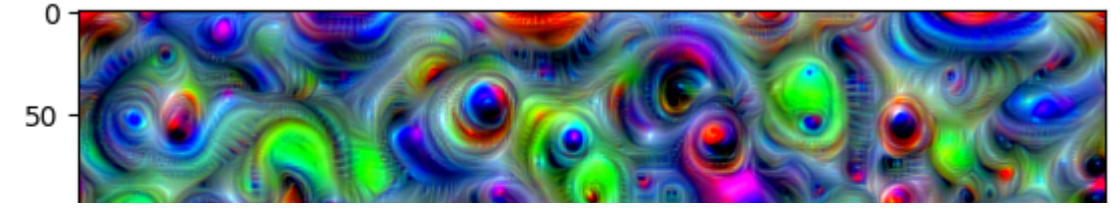

```
Saved DeepDream static image to: deepdreamtry2_width_600_model_VGG16_EXPERIMEN
```

```
# Load and visualize the first image
input_img_name1 = '/content/deepdream1.jpeg'
img1 = load_image(input_img_name1, target_shape=img_width)
visualize_image(img1)

# Load and visualize the second image
input_img_name2 = '/content/deepdreamtry2.jpeg'
img2 = load_image(input_img_name2, target_shape=img_width)
visualize_image(img2)

# Store the images and their corresponding names in a list
images = [img1, img2]
image_names = [input_img_name1, input_img_name2]

# Get exposed layers
exposed_layers = Vgg16Experimental(pretrained_weights=SupportedPretrainedWeights.
print(f'Exposed layers = {exposed_layers}')

# Apply deep dream for each layer on both images
for image, image_name in zip(images, image_names):
    for layer in exposed_layers:
        config['layers_to_use'] = [layer]

        img = deep_dream_static_image(config, image)
        dump_path = save_and_maybe_display_image(config, img, f'{layer}_{os.path.
        print(f'Saved DeepDream static image for {layer} on {os.path.basename(ima

# Keep the config consistent from cell to cell
config['layers_to_use'] = ['relu4_3']
```



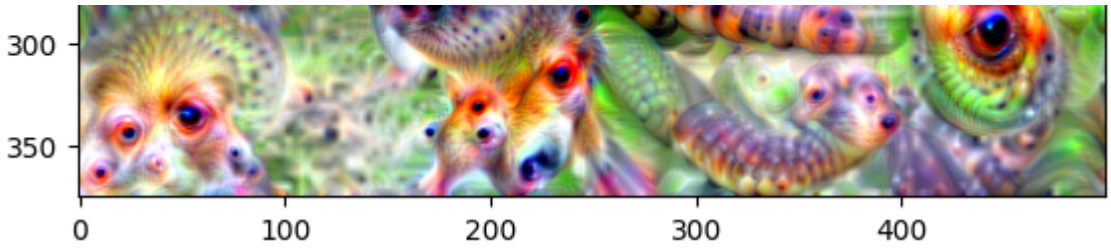



```
Exposed layers = ['relu3_3', 'relu4_1', 'relu4_2', 'relu4_3', 'relu5_1']
```

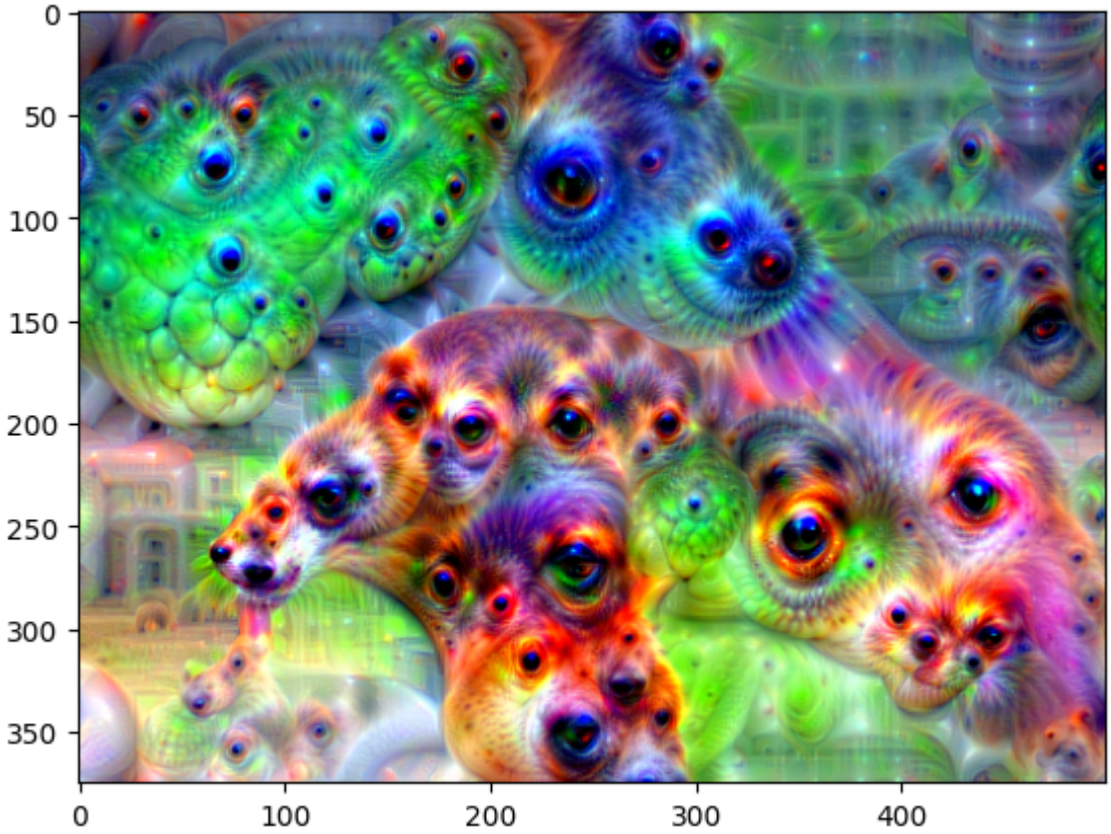


Experimenting pyramid size and pyramid ratio

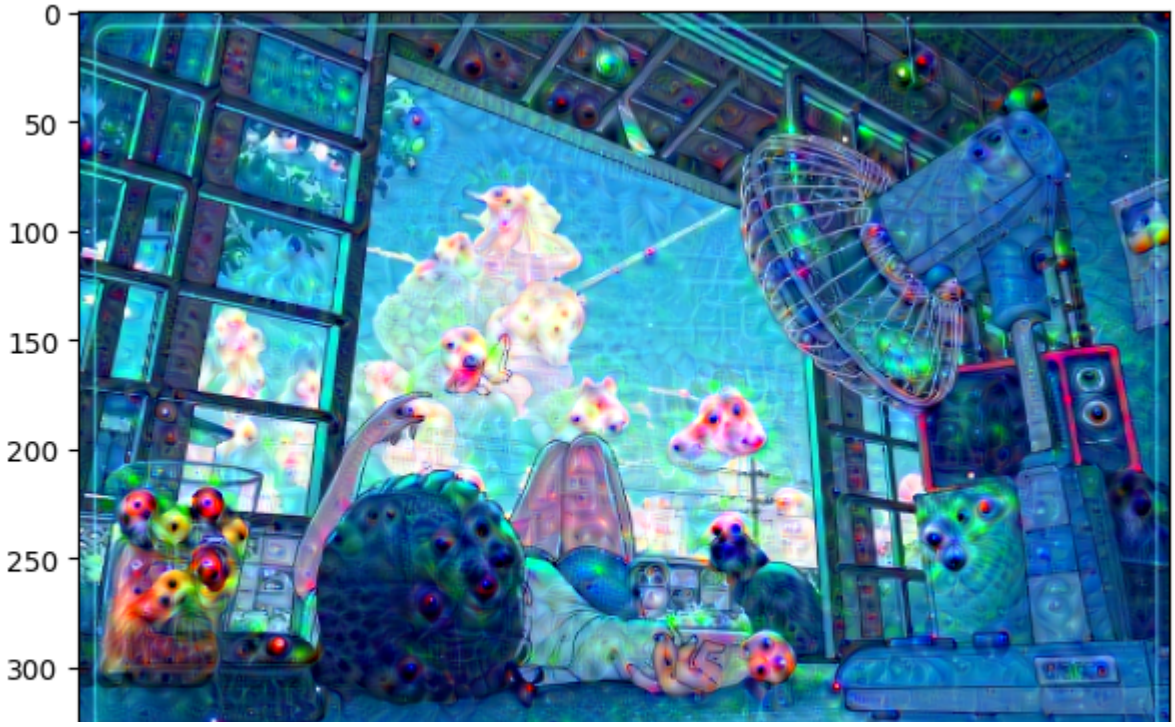
```
# Load and visualize the first image
input_img_name1 = '/content/deepdream1.jpeg'
img1 = load_image(input_img_name1, target_shape=img_width)
visualize_image(img1)

# Load and visualize the second image
input_img_name2 = '/content/deepdreamtry2.jpeg'
img2 = load_image(input_img_name2, target_shape=img_width)
visualize_image(img2)

# Store the images and their corresponding names in a list
images = [img1, img2]
image_names = [input_img_name1, input_img_name2]

# Define pyramid sizes
pyramid_sizes = [1, 3, 5]
config['pyramid_ratio'] = 1.8  # feel free to play with this one as well

# Apply deep dream for each pyramid size on both images
for image, image_name in zip(images, image_names):
    for pyramid_size in pyramid_sizes:
        config['pyramid_size'] = pyramid_size

        img = deep_dream_static_image(config, image)
        dump_path = save_and_maybe_display_image(config, img, f'pyramid_size_{pyr
        print(f'Saved DeepDream static image for pyramid size {pyramid_size} on {

# Keep the config consistent from cell to cell
config['pyramid_size'] = 4
```




```
Saved DeepDream static image for pyramid size 3 on deepdream1.jpeg to: data/o
```

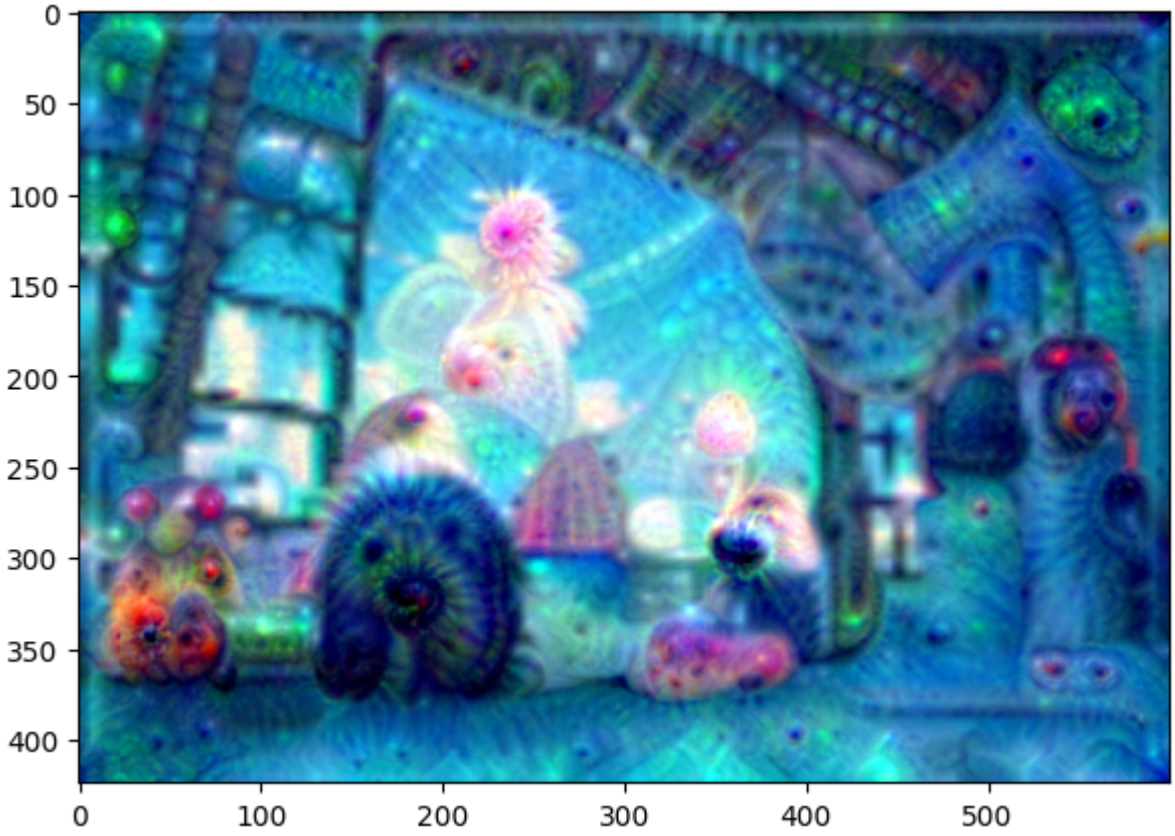



```
Saved DeepDream static image for pyramid size 5 on deepdream1.jpeg to: data/o
```

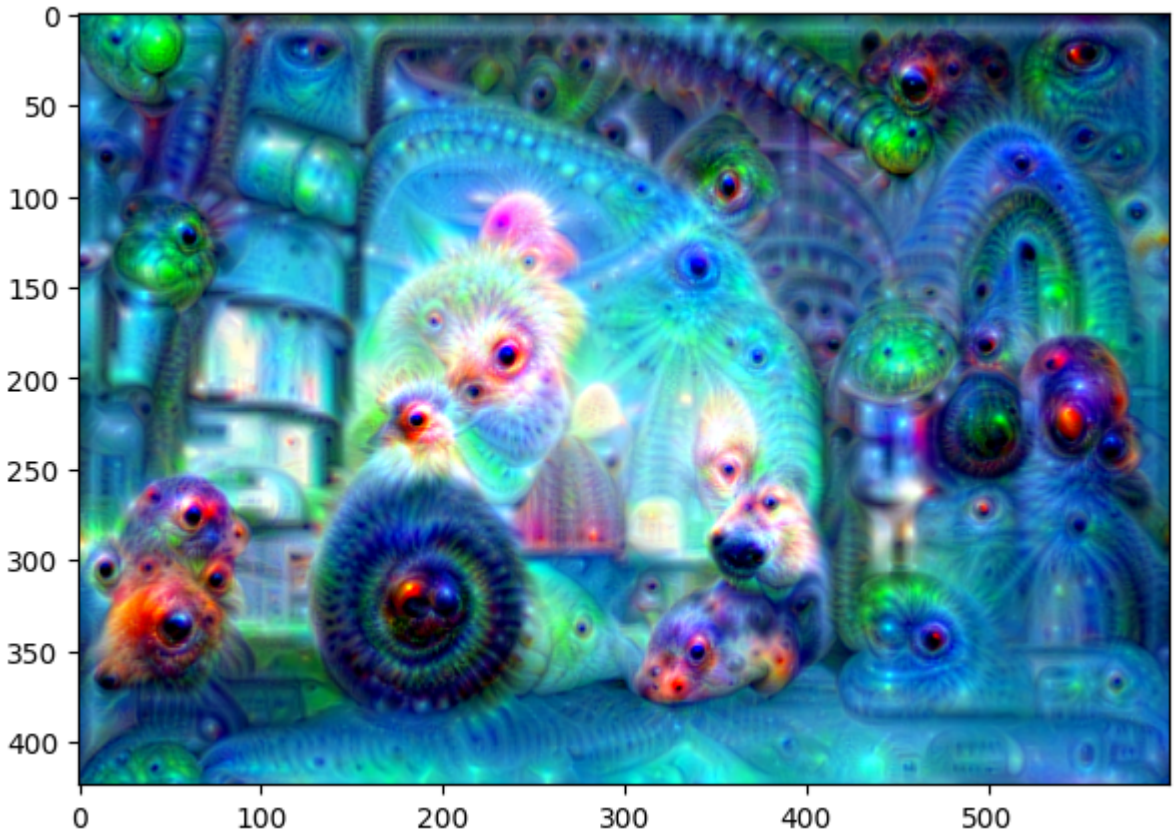


tuning the num_gradient_ascent_iterations and lr

```
num_gradient_ascent_iterations = [2, 5, 20]
config['lr'] = 0.09  # feel free to play with this one as well

for num_iter in num_gradient_ascent_iterations:
    config['num_gradient_ascent_iterations'] = num_iter

    img = deep_dream_static_image(config)
    dump_path = save_and_maybe_display_image(config, img)
    print(f'Saved DeepDream static image to: {os.path.relpath(dump_path)}\n')

config['num_gradient_ascent_iterations'] = 10  # keep the config consistent from
```




```
Saved DeepDream static image to: deepdreamtry2_width_600_model_VGG16_EXPERIMEN
```




```
Saved DeepDream static image to: deepdreamtry2_width_600_model_VGG16_EXPERIMEN
```

Eperimentation using inception V3

"neuron visualization" through a process called "gradient ascent"

```
# boilerplate code
from __future__ import print_function
import os
from io import BytesIO
import numpy as np
from functools import partial
import PIL.Image
from IPython.display import clear_output, Image, display, HTML

import tensorflow as tf
```

```
#!wget https://storage.googleapis.com/download.tensorflow.org/models/inception5h.
```

```
import tensorflow as tf
import numpy as np
from PIL import Image

# Disable eager execution
tf.compat.v1.disable_eager_execution()

# Define necessary functions and placeholders
def tffunc(*argtypes):
    placeholders = list(map(tf.compat.v1.placeholder, argtypes))
    def wrap(f):
        out = f(*placeholders)
        def wrapper(*args, **kw):
            return out.eval(dict(zip(placeholders, args)), session=kw.get('sessio
        return wrapper
    return wrap

def resize(img, size):
    img = tf.expand_dims(img, 0)
    return tf.compat.v1.image.resize_bilinear(img, size)[0,:,:,:]
resize = tffunc(np.float32, np.int32)(resize)


def calc_grad_tiled(img, t_grad, tile_size=512):
    sz = tile_size
    h, w = img.shape[:2]
    sx, sy = np.random.randint(sz, size=2)
    img_shift = np.roll(np.roll(img, sx, 1), sy, 0)
    grad = np.zeros_like(img)
    for y in range(0, max(h-sz//2, sz),sz):
        for x in range(0, max(w-sz//2, sz),sz):
            sub = img_shift[y:y+sz,x:x+sz]
            g = sess.run(t_grad, {t_input:sub})
            grad[y:y+sz,x:x+sz] = g
    return np.roll(np.roll(grad, -sx, 1), -sy, 0)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow/py
Instructions for updating:
Use `tf.image.resize(...method=ResizeMethod.BILINEAR...)` instead.
```

```
# boilerplate code
from __future__ import print_function
import os
from io import BytesIO
import numpy as np
from functools import partial
import PIL.Image
from IPython.display import clear_output, Image, display, HTML

import tensorflow as tf
```

```
!wget https://storage.googleapis.com/download.tensorflow.org/models/inception5h.z
!unzip inception5h.zip
```

```
--2023-11-14 15:52:53--  https://storage.googleapis.com/download.tensorflow.o
Resolving storage.googleapis.com (storage.googleapis.com)... 172.253.114.207,
Connecting to storage.googleapis.com (storage.googleapis.com)|172.253.114.207
HTTP request sent, awaiting response... 200 OK
Length: 49937555 (48M) [application/zip]
Saving to: ‘inception5h.zip’

inception5h.zip     100%[===================>]  47.62M  94.5MB/s    in 0.5s

2023-11-14 15:52:53 (94.5 MB/s) - ‘inception5h.zip’ saved [49937555/49937555]

Archive:  inception5h.zip
  inflating: imagenet_comp_graph_label_strings.txt
  inflating: tensorflow_inception_graph.pb
  inflating: LICENSE
```

```
model_fn = 'tensorflow_inception_graph.pb'

# creating TensorFlow session and loading the model from the model_fn file
graph = tf.Graph()
sess = tf.compat.v1.InteractiveSession(graph=graph)
with tf.io.gfile.GFile(model_fn, 'rb') as f:
    graph_def = tf.compat.v1.GraphDef()
    graph_def.ParseFromString(f.read())
t_input = tf.compat.v1.placeholder(np.float32, name='input') # define the input t
imagenet_mean = 117.0
t_preprocessed = tf.expand_dims(t_input - imagenet_mean, 0)
tf.import_graph_def(graph_def, {'input': t_preprocessed})
```

```
layers = [op.name for op in graph.get_operations() if op.type=='Conv2D' and 'impo
feature_nums = [int(graph.get_tensor_by_name(name+':0').get_shape()[-1]) for name

print('Number of layers', len(layers))
print('Total number of feature channels:', sum(feature_nums))
```

```
Number of layers 59
Total number of feature channels: 7548
```

```
def T(layer):
    '''Helper for getting layer output tensor'''
    return graph.get_tensor_by_name("import/%s:0"%layer)
```

```
layer=layers[4]
print(layer)
layer = layer.split("/")[1]
print(layer)
```

```
import/mixed3a_3x3_bottleneck_pre_relu/conv
mixed3a_3x3_bottleneck_pre_relu
```

```
T(layer)
```

```
<tf.Tensor 'import/mixed3a_3x3_bottleneck_pre_relu:0' shape=(None, None,
None, 96) dtype=float32>
```

```
for l, layer in enumerate(layers):
    layer = layer.split("/")[1]
    num_channels = T(layer).shape[3]
    print(layer, num_channels)
```

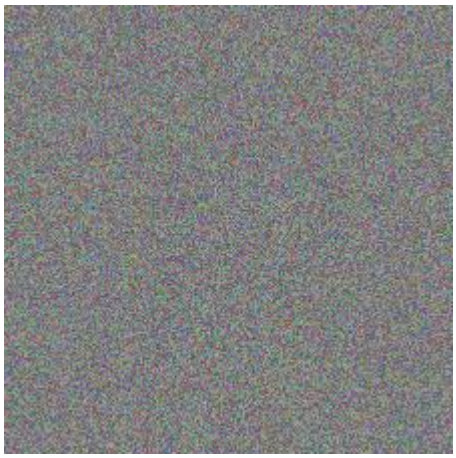
```
conv2d1_pre_relu 64
conv2d2_pre_relu 192
mixed3a_1x1_pre_relu 64
mixed3a_3x3_bottleneck_pre_relu 96
mixed3a_3x3_pre_relu 128
mixed3a_5x5_bottleneck_pre_relu 16
mixed3a_5x5_pre_relu 32
mixed3a_pool_reduce_pre_relu 32
mixed3b_1x1_pre_relu 128
mixed3b_3x3_bottleneck_pre_relu 128
mixed3b_3x3_pre_relu 192
mixed3b_5x5_bottleneck_pre_relu 32
mixed3b_5x5_pre_relu 96
mixed3b_pool_reduce_pre_relu 64
mixed4a_1x1_pre_relu 192
mixed4a_3x3_bottleneck_pre_relu 96
mixed4a_3x3_pre_relu 204
mixed4a_5x5_bottleneck_pre_relu 16
mixed4a_5x5_pre_relu 48
mixed4a_pool_reduce_pre_relu 64
mixed4b_1x1_pre_relu 160
mixed4b_3x3_bottleneck_pre_relu 112
mixed4b_3x3_pre_relu 224
mixed4b_5x5_bottleneck_pre_relu 24
mixed4b_5x5_pre_relu 64
mixed4b_pool_reduce_pre_relu 64
mixed4c_1x1_pre_relu 128
mixed4c_3x3_bottleneck_pre_relu 128
mixed4c_3x3_pre_relu 256
mixed4c_5x5_bottleneck_pre_relu 24
mixed4c_5x5_pre_relu 64
mixed4c_pool_reduce_pre_relu 64
mixed4d_1x1_pre_relu 112
mixed4d_3x3_bottleneck_pre_relu 144
mixed4d_3x3_pre_relu 288
mixed4d_5x5_bottleneck_pre_relu 32
mixed4d_5x5_pre_relu 64
mixed4d_pool_reduce_pre_relu 64
mixed4e_1x1_pre_relu 256
mixed4e_3x3_bottleneck_pre_relu 160
mixed4e_3x3_pre_relu 320
mixed4e_5x5_bottleneck_pre_relu 32
mixed4e_5x5_pre_relu 128
mixed4e_pool_reduce_pre_relu 128
mixed5a_1x1_pre_relu 256
```

```
mixed5a_3x3_pre_relu 320
mixed5a_5x5_bottleneck_pre_relu 48
mixed5a_5x5_pre_relu 128
mixed5a_pool_reduce_pre_relu 128
mixed5b_1x1_pre_relu 384
mixed5b_3x3_bottleneck_pre_relu 192
mixed5b_3x3_pre_relu 384
mixed5b_5x5_bottleneck_pre_relu 48
mixed5b_5x5_pre_relu 128
mixed5b_pool_reduce_pre_relu 128
head0_bottleneck_pre_relu 128
head1_bottleneck_pre_relu 128
```

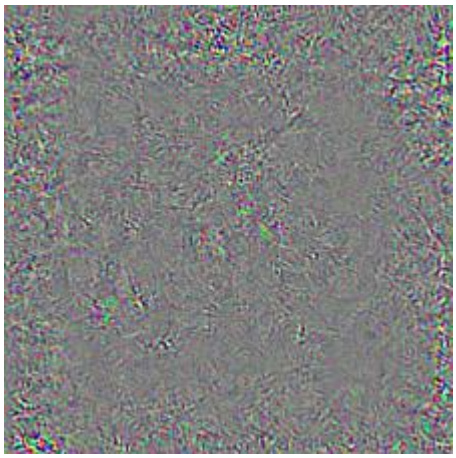
```
# Picking some internal layer. Note that we use outputs before applying the ReLU
# to have non-zero gradients for features with negative initial activations.
layer = 'mixed4d_3x3_bottleneck_pre_relu'
channel = 139 # picking some feature channel to visualize

# start with a gray image with a little noise
img_noise = np.random.uniform(size=(224,224,3)) + 100.0
```

The outputs during the visualization represent the optimization score (mean of the neuron activities in the specified layer and channel) at each iteration of the gradient ascent process.

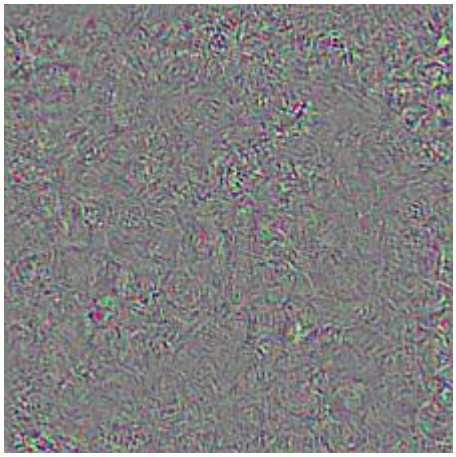
```
def showarray(a, fmt='jpeg'):
    '''create a jpeg file from an array a and visualize it'''
    # clip the values to be between 0 and 255
    a = np.uint8(np.clip(a, 0, 1)*255)
    f = BytesIO()
    PIL.Image.fromarray(a).save(f, fmt)
    display(Image(data=f.getvalue()))

def visstd(a, s=0.1):
    '''Normalize the image range for visualization'''
    return (a-a.mean())/max(a.std(), 1e-4)*s + 0.5


def render_naive(t_obj, img0=img_noise, iter_n=20, step=1.0):


    t_score = tf.reduce_mean(t_obj) # defining the optimization objective (mean o
    t_grad = tf.gradients(t_score, t_input)[0] # calculate the gradient of the ob

    img = img0.copy()
    showarray(visstd(img)) # show the input image

    for i in range(iter_n): # for iter_n iterations keep updating the image
        g, score = sess.run([t_grad, t_score], {t_input:img})
        # normalizing the gradient, so the same step size should work
        g /= g.std()+1e-8         # for different layers and networks
        img += g*step
        print(i, score, end = ' ') # show the current objective value
        showarray(visstd(img)) # show the actual image
    #clear_output()


render_naive(T(layer)[:,:,:,channel]) # run the render_naive function and start h
```



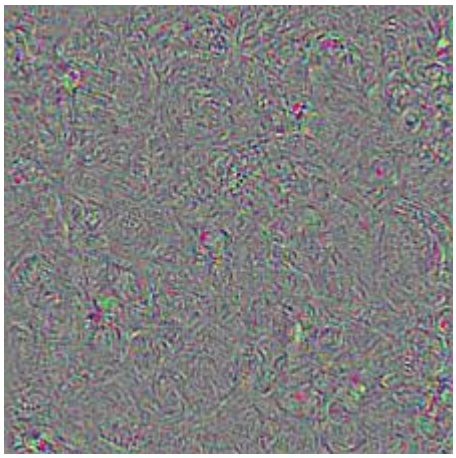
0 -19.853647



1 -30.510702



2 9.637701



3 87.54462