Deep dream network

With training artificial neural network, people adjust the network parameters to gradually get their classification they want. The image is fed into the input layer, then talks to the next layer, layer by layer, and eventually get the answer of the output layer.

Therefore, people becomes curious about what exactly goes on at each layer, what does each layer see and do. The theory is that each layer parse and process different level features of the original image, and the final layer decide what image to show. For instance, the first layer in network may just see edges and dots, the intermediate layer begins to see simple feature like doors or leaves, and some final layers are able to see complete and complex images such as buildings or trees.

One important way to know what goes on in each layer is to feed the network with some images and photos firsts, and then pick a layer and tell the network to enhance what this layer detects. As we said above each layer of the network deal with features at different levels. Therefore, the complexity of the final features we generate depends on which layer we choose to enhance.

We will first start with the codes, then the images, and parameter experiments at last.

```
parser = argparse.ArgumentParser(description='Deep Dreams with Keras.')#create an object which includes the interpretation of commands
parser.add_argument('base_image_path', metavar='base', type=str,
                    help='Path to the image to transform.')# add argument base_image_path in ArgumentParser
parser.add_argument('result_prefix', metavar='res_prefix', type=str,
                    help='Prefix for the saved results.')# add argument result_prefix
args = parser.parse_args()# was used to parse the parameters
base_image_path = args.base_image_path
result_prefix = args.result_prefix
```

This segment of codes perform the function that regulate the format of the commands to input the trivial picture，then set up the parameter of the input and output

```
settings = {
    'features': {
        'mixed2': 0.2,#layer name and its weights
        'mixed3': 0.5,
        'mixed4': 2.,
        'mixed5': 1.5,
    },
}
```

This This segment of codes establishes the layers and their weights, we will then use them to compute the loss.

```
K.set_learning_phase(0)
# Build the InceptionV3 network with our placeholder.
# The model will be loaded with pre-trained ImageNet weights.
model = inception_v3.InceptionV3(weights='imagenet',
                                 include_top=False)
dream = model.input
print('Model loaded.')
```

This segment of codes first initiate the learning phase as a fixed value, then use the input image to do complex inception operation. Note that here we also use the weights that the keras provided. After inception function, we can get back a keras model.(The operation in inceptionv3 will talk later.)

Set up dream variable and assign it the input of model(The image).

```
layer_dict = dict([(layer.name, layer) for layer in model.layers])#use dict format
(key:value) to fetch the elements from the array model.layers(use the format of
for to get element) layers is a keras' class
#create a new dict by using the existed dict model.layers   get all layers used
in the inception_v3

# Define the loss.
loss = K.variable(0.)#Instantiates a variable and returns it.
```

This segment of codes use the format of for to get layer name information from the model, and then assign them to a new dict object. The element in the new dict object will be the layers used in the inception_v3.
Initiate the loss as a variable.

```
for layer_name in settings['features']:
    # Add the L2 norm of the features of a layer to the loss.
    assert layer_name in layer_dict.keys(), 'Layer ' + layer_name + ' not found
in model.' #expression is layer_name in layer_dict.keys()      if layer_name is
the 'key' to the dict layer-dict
    coeff = settings['features'][layer_name]
    #fectch and define respective values in setting array
    x = layer_dict[layer_name].output#get the output of the layer.
    # We avoid border artifacts by only involving non-border pixels in the loss.
    scaling = K.prod(K.cast(K.shape(x), 'float32'))
    if K.image_data_format() == 'channels_first':#k.image_data_format(),返回默
认的图像的维度顺序（‘channels_last’或‘channels_first’）
        loss += coeff * K.sum(K.square(x[:, :, 2: -2, 2: -2])) / scaling
#coeff refers to each layer’s weight,   scaling is the multiplication of tensors
    else:
        loss += coeff * K.sum(K.square(x[:, 2: -2, 2: -2, :])) / scaling
```

This segment of codes use layer_name as key to extract weight and output information from the keras model which was the return of the inception_v3 function.

Then use prod function to compute multiplication of elements in the tensors(tensors are out of each model layer. Cast function was used to change the type of the data).

At last use the formula to compute loss. The elements of the formula are values we extracted from the model and computation in former steps.

```
# Compute the gradients of the dream weights the loss.
grads = K.gradients(loss, dream)[0]#gradients(loss, variables),return is a
gradient of variable.
# Normalize gradients.
grads /= K.maximum(K.mean(K.abs(grads)), 1e-7)#k.abs compute absolute value,
k.mean()is to compute the average of data, grads is the computation of gradient

# Set up function to retrieve the value
# of the loss and gradients given an input image.
outputs = [loss, grads]
fetch_loss_and_grads = K.function([dream], outputs)#dream Inceptionv3 input,
output is a two dimensional data
#function(inputs, outputs, updates=[]), , , make a Keras function
#inputs:: input list, could be tensors
#outputs: output list
#updates: a tuple, the format of it is like (old_tensor, new_tensor)
```

This segment of codes first compute the gradients using loss value(It is a summed value computed by adding loss from each layer) and dream(the input image). The return will be a gradient of variables.

K.function is a function that regulate the format and input of a new function.

From the operations above, we computed the gradient and loss(summed value of each layer used in inception_v3) of a inception_v3 model, the input of it is our trivial image.

Subsequent codes are the main operations.

```
step = 0.01  # Gradient ascent step size,0.01
num_octave = 3  # Number of scales at which to run gradient ascent 3
octave_scale = 1.4  # Size ratio between scales 1.4
iterations = 20  # Number of ascent steps per scale 20
max_loss = 10.#10
```

These parameter will be used to change the shape or effect of the images, and they are the main factors that we will change to do experiments later.

```
img = preprocess_image(base_image_path)#get an inception_v3 image model
if K.image_data_format() == 'channels_first':
    original_shape = img.shape[2:]
else:
    original_shape = img.shape[1:3]
successive_shapes = [original_shape]

def preprocess_image(image_path):
    # Util function to open, resize and format pictures
    # into appropriate tensors.
    img = load_img(image_path) #Loads an image into PIL format. PIL Python Imaging 
Library  is a free library for the Python programming language that adds support 
for opening, manipulating, and saving many different image file formats.
    #Some of the file formats supported include PPM, PNG, JPEG, GIF, TIFF, and 
BMP.
    img = img_to_array(img)#Converts a PIL Image instance to a Numpy 
array.,inputing parameter 'img' is PIL image instance, the result would be a 3D 
numpy array
    img = np.expand_dims(img, axis=0)#expand the dimension
    img = inception_v3.preprocess_input(img)
    return img
```

This segment of codes loads the trivial picture, change format of the picture, and then use inception_v3 to process it. At last, get a processed picture back.

```
for i in range(1, num_octave):
    shape = tuple([int(dim / (octave_scale ** i)) for dim in original_shape])
    successive_shapes.append(shape)#get different shapes size during each scale
successive_shapes = successive_shapes[::-1]
original_img = np.copy(img)#get a copy of original image
shrunk_original_img = resize_img(img, successive_shapes[0])#resize the original 
image
```

Get different shape size of the image to compute the successive shape, then resize the original image.

```
for shape in successive_shapes:
    print('Processing image shape', shape)
    img = resize_img(img, shape)#resize image using each shape we got during every 
scale
    img = gradient_ascent(img,
                          iterations=iterations,
                          step=step,
                          max_loss=max_loss)#return the result of img + grad value 
at each iteration
    upscaled_shrunk_original_img = resize_img(shrunk_original_img, shape)#this 
shape is the final successive shape size
    same_size_original = resize_img(original_img, shape)
    lost_detail = same_size_original - upscaled_shrunk_original_img#used the 
shrinked and upscaled images and original image to get the loss during the upscaling

    img += lost_detail#result
    shrunk_original_img = resize_img(original_img, shape)

save_img(img, fname=result_prefix + '.png')
```

Use gradient_ascent function to compute an new image, the parameters are set before. This function will return the result of image + grad value at each iteration. Because we have changed several shapes, they are store in successive_shapes, when we upscale the image to the next scale, the image lost some data, so we have to add these data back. Complete the whole loop.

```
def eval_loss_and_grads(x):
    outs = fetch_loss_and_grads([x])
    loss_value = outs[0]
    grad_values = outs[1]
    return loss_value, grad_values


def resize_img(img, size):
    img = np.copy(img)
    if K.image_data_format() == 'channels_first':
        factors = (1, 1,
                   float(size[0]) / img.shape[2],
                   float(size[1]) / img.shape[3])
    else:
        factors = (1,
                   float(size[0]) / img.shape[1],
                   float(size[1]) / img.shape[2],
                   1)
    return scipy.ndimage.zoom(img, factors, order=1)



def gradient_ascent(x, iterations, step, max_loss=None):
    for i in range(iterations):
        loss_value, grad_values = eval_loss_and_grads(x)
        if max_loss is not None and loss_value > max_loss:
            break
        print('..Loss value at', i, ':', loss_value)
        x += step * grad_values
    return x
```

These functions are used to change the size of data, and computing the gradient ascent of the image.

Inception_v3

```
input_shape = _obtain_input_shape(
    input_shape,
    default_size=299,
    min_size=139,
    data_format=K.image_data_format(),
    require_flatten=False,
    weights=weights) #this is the input shape
```

Get the input shape of the image.

```
if input_tensor is None:
    img_input = Input(shape=input_shape) #make an input tensor in terms of the input
shape,  a placeholder
else:
    if not K.is_keras_tensor(input_tensor):
        img_input = Input(tensor=input_tensor, shape=input_shape) #Optional
existing tensor to wrap into the `Input` layer. If set, the layer will not create
a placeholder tensor.
    else:
        img_input = input_tensor
```

If the input does not have a tensor, create a tensor now, and using input_shape as parameter. Note that the tensor now is just a placeholder.

If the input already has a tensor, make a new tensor with the input tensor as parameter, and the tensor now has value.

```
if K.image_data_format() == 'channels_first':
    channel_axis = 1 #image_data_format is channel first
else:
    channel_axis = 3
```

Set different axis value for different image data format.

```
x = conv2d_bn(img_input, 32, 3, 3, strides=(2, 2), padding='valid')#this img_input
should be a tensor, either a esisting tensor or a placeholder
x = conv2d_bn(x, 32, 3, 3, padding='valid')#the number filter is 32, the height
of kernel is 3 the width of kernel is 3
x = conv2d_bn(x, 64, 3, 3)#64 filters ,3*3
x = MaxPooling2D((3, 3), strides=(2, 2))(x)#Max pooling operation for spatial
data.,,pool size is 2*2

x = conv2d_bn(x, 80, 1, 1, padding='valid')#80 filters 1*1
x = conv2d_bn(x, 192, 3, 3, padding='valid')#192 filters 3*3
x = MaxPooling2D((3, 3), strides=(2, 2))(x)
```

Use conv2d function(used as performing convolutional layer) to process input image with different filters ,height and width, applying pooling layer after convolutional layers.

```
# mixed 0: 35 x 35 x 256
branch1x1 = conv2d_bn(x, 64, 1, 1)#64 filters 1*1

branch5x5 = conv2d_bn(x, 48, 1, 1)
branch5x5 = conv2d_bn(branch5x5, 64, 5, 5)#64 filters 5*5

branch3x3dbl = conv2d_bn(x, 64, 1, 1)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)

branch_pool = AveragePooling2D((3, 3), strides=(1, 1), padding='same')(x)# use
average pooling operation
branch_pool = conv2d_bn(branch_pool, 32, 1, 1)
x = layers.concatenate(# concatenate layers branch1x1(64 fiklters 1*1),
branch5x5(64 filters 5*5), branch3x3dbl(96 filyters 3*3), branch_pool(use
pooling operation)
    [branch1x1, branch5x5, branch3x3dbl, branch_pool],
    axis=channel_axis,
    name='mixed0')
#name this as mixed0
```

Construct mixed layer 0, the dimension will be 35*35*256. These mixed layer consists of three major convolutional layers:branch1x1 with 64 filters and 1*1 kernel size, branch5x5 with 64 filters and 5*5 kernel size, Branch 3x3db1with 96 filters and 3*3 kernel size, and an average pooling layer.

```
# mixed 1: 35 x 35 x 256
branch1x1 = conv2d_bn(x, 64, 1, 1)

branch5x5 = conv2d_bn(x, 48, 1, 1)
branch5x5 = conv2d_bn(branch5x5, 64, 5, 5)

branch3x3dbl = conv2d_bn(x, 64, 1, 1)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)

branch_pool = AveragePooling2D((3, 3), strides=(1, 1), padding='same')(x)
branch_pool = conv2d_bn(branch_pool, 64, 1, 1)
x = layers.concatenate(#concatenate these layers: branch1x1(64 filters 1*1),
branch5x5(through two layers,fitrst a 48 filters 1*1 and then 64 filters 5*5),
branch3x3dbl(through 3 layers first 64 filters 1*1,then 96 filters 3*3, then 96
filters 3*3), branch_pool
    [branch1x1, branch5x5, branch3x3dbl, branch_pool],
    axis=channel_axis,
    name='mixed1')
#name these mixed layers as mixed1
```

Construct mixed layer 1, the dimension will be 35*35*256. These mixed layer consists of three major convolutional layers:branch1x1 with 64 filters and 1*1 kernel size, branch5x5 with 64 filters and 5*5 kernel size, Branch 3x3db1with 96 filters and 3*3 kernel size, and an average pooling layer.

```
# mixed 2: 35 x 35 x 256
branch1x1 = conv2d_bn(x, 64, 1, 1)

branch5x5 = conv2d_bn(x, 48, 1, 1)
branch5x5 = conv2d_bn(branch5x5, 64, 5, 5)

branch3x3dbl = conv2d_bn(x, 64, 1, 1)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)

branch_pool = AveragePooling2D((3, 3), strides=(1, 1), padding='same')(x)
branch_pool = conv2d_bn(branch_pool, 64, 1, 1)
x = layers.concatenate(
    [branch1x1, branch5x5, branch3x3dbl, branch_pool],
    axis=channel_axis,
    name='mixed2')
```

Construct mixed layer 2, the dimension will be 35*35*256. These mixed layer consists of three major convolutional layers:branch1x1 with 64 filters and 1*1 kernel size, branch5x5 with 64 filters and 5*5 kernel size, Branch 3x3db1with 96 filters and 3*3 kernel size, and an average pooling layer.

```
# mixed 3: 17 x 17 x 768
branch3x3 = conv2d_bn(x, 384, 3, 3, strides=(2, 2), padding='valid')

branch3x3dbl = conv2d_bn(x, 64, 1, 1)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)
branch3x3dbl = conv2d_bn(
    branch3x3dbl, 96, 3, 3, strides=(2, 2), padding='valid')

branch_pool = MaxPooling2D((3, 3), strides=(2, 2))(x)
x = layers.concatenate(
    [branch3x3, branch3x3dbl, branch_pool], axis=channel_axis, name='mixed3')
```

Construct mixed layer 3, the dimension will be 17*17*768(the height and width decreased, but enlarge the number of channel). These mixed layer consists of one major convolutional layers:branch1x1 with 96 filters and 3*3 kernel size, and an average pooling layer.

```
# mixed 4: 17 x 17 x 768
branch1x1 = conv2d_bn(x, 192, 1, 1)

branch7x7 = conv2d_bn(x, 128, 1, 1)
branch7x7 = conv2d_bn(branch7x7, 128, 1, 7)
branch7x7 = conv2d_bn(branch7x7, 192, 7, 1)

branch7x7dbl = conv2d_bn(x, 128, 1, 1)
branch7x7dbl = conv2d_bn(branch7x7dbl, 128, 7, 1)
branch7x7dbl = conv2d_bn(branch7x7dbl, 128, 1, 7)
branch7x7dbl = conv2d_bn(branch7x7dbl, 128, 7, 1)
branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 1, 7)

branch_pool = AveragePooling2D((3, 3), strides=(1, 1), padding='same')(x)
branch_pool = conv2d_bn(branch_pool, 192, 1, 1)
x = layers.concatenate(
    [branch1x1, branch7x7, branch7x7dbl, branch_pool],
    axis=channel_axis,
    name='mixed4')
```

Construct mixed layer 4, the dimension will be 17*17*768. These mixed layer consists of three major convolutional layers:branch1x1 with 192 filters and 1*1 kernel size, branch7x7 with 192 filters(at first 128 filters) and 7*1 kernel size, Branch 7x7db1with 192 filters(at first 128 filters) and 1*7 kernel size, and an average pooling layer.

```python
# mixed 5, 6: 17 x 17 x 768
for i in range(2):
    branch1x1 = conv2d_bn(x, 192, 1, 1)

    branch7x7 = conv2d_bn(x, 160, 1, 1)
    branch7x7 = conv2d_bn(branch7x7, 160, 1, 7)
    branch7x7 = conv2d_bn(branch7x7, 192, 7, 1)

    branch7x7dbl = conv2d_bn(x, 160, 1, 1)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 160, 7, 1)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 160, 1, 7)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 160, 7, 1)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 1, 7)

    branch_pool = AveragePooling2D(
        (3, 3), strides=(1, 1), padding='same')(x)
    branch_pool = conv2d_bn(branch_pool, 192, 1, 1)
    x = layers.concatenate(
        [branch1x1, branch7x7, branch7x7dbl, branch_pool],
        axis=channel_axis,
        name='mixed' + str(5 + i))
```

Construct mixed layer 5 and 6, both the dimension will be 17*17*768. These mixed layer consists of three major convolutional layers:branch1x1 with 192 filters and 1*1 kernel size, branch7x7 with 192 filters(at first 160 filters) and 7*1 kernel size, Branch 7x7db1with 192 filters (at first 160 filters)and 1*7 kernel size, and an average pooling layer.

```python
# mixed 7: 17 x 17 x 768
branch1x1 = conv2d_bn(x, 192, 1, 1)

branch7x7 = conv2d_bn(x, 192, 1, 1)
branch7x7 = conv2d_bn(branch7x7, 192, 1, 7)
branch7x7 = conv2d_bn(branch7x7, 192, 7, 1)

branch7x7dbl = conv2d_bn(x, 192, 1, 1)
branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 7, 1)
branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 1, 7)
branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 7, 1)
branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 1, 7)

branch_pool = AveragePooling2D((3, 3), strides=(1, 1), padding='same')(x)
branch_pool = conv2d_bn(branch_pool, 192, 1, 1)
x = layers.concatenate(
    [branch1x1, branch7x7, branch7x7dbl, branch_pool],
    axis=channel_axis,
    name='mixed7')
```

Construct mixed layer 7, the dimension will be 17*17*768. These mixed layer consists of three major convolutional layers:branch1x1 with 192 filters and 1*1 kernel size, branch7x7 with 192 filtersand 7*1 kernel size, Branch 7x7db1with 192 filters and 1*7 kernel size, and an average pooling layer.

```
# mixed 8: 8 x 8 x 1280
branch3x3 = conv2d_bn(x, 192, 1, 1)
branch3x3 = conv2d_bn(branch3x3, 320, 3, 3,
                      strides=(2, 2), padding='valid')

branch7x7x3 = conv2d_bn(x, 192, 1, 1)
branch7x7x3 = conv2d_bn(branch7x7x3, 192, 1, 7)
branch7x7x3 = conv2d_bn(branch7x7x3, 192, 7, 1)
branch7x7x3 = conv2d_bn(
    branch7x7x3, 192, 3, 3, strides=(2, 2), padding='valid')

branch_pool = MaxPooling2D((3, 3), strides=(2, 2))(x)
x = layers.concatenate(
    [branch3x3, branch7x7x3, branch_pool], axis=channel_axis, name='mixed8')
```

Construct mixed layer 8, the dimension will be 8*8*1280. These mixed layer consists of two major convolutional layers:branch3x3 with 320 filters(at first 192 filters) and 3*3 kernel size, branch7x7x3 with 192 filters and 7*1 kernel size, Branch 7x7db1with 192 filters and 1*7 kernel size, and an average pooling layer.

```
# mixed 9: 8 x 8 x 2048
for i in range(2):
    branch1x1 = conv2d_bn(x, 320, 1, 1)

    branch3x3 = conv2d_bn(x, 384, 1, 1)
    branch3x3_1 = conv2d_bn(branch3x3, 384, 1, 3)
    branch3x3_2 = conv2d_bn(branch3x3, 384, 3, 1)
    branch3x3 = layers.concatenate(
        [branch3x3_1, branch3x3_2], axis=channel_axis, name='mixed9_' + str(i))

    branch3x3dbl = conv2d_bn(x, 448, 1, 1)
    branch3x3dbl = conv2d_bn(branch3x3dbl, 384, 3, 3)
    branch3x3dbl_1 = conv2d_bn(branch3x3dbl, 384, 1, 3)
    branch3x3dbl_2 = conv2d_bn(branch3x3dbl, 384, 3, 1)
    branch3x3dbl = layers.concatenate(
        [branch3x3dbl_1, branch3x3dbl_2], axis=channel_axis)

    branch_pool = AveragePooling2D(
        (3, 3), strides=(1, 1), padding='same')(x)
    branch_pool = conv2d_bn(branch_pool, 192, 1, 1)#branch_pool is an after
pooling tensor
    x = layers.concatenate(
        [branch1x1, branch3x3, branch3x3dbl, branch_pool],
        axis=channel_axis,
        name='mixed' + str(9 + i))#create a new string object
```

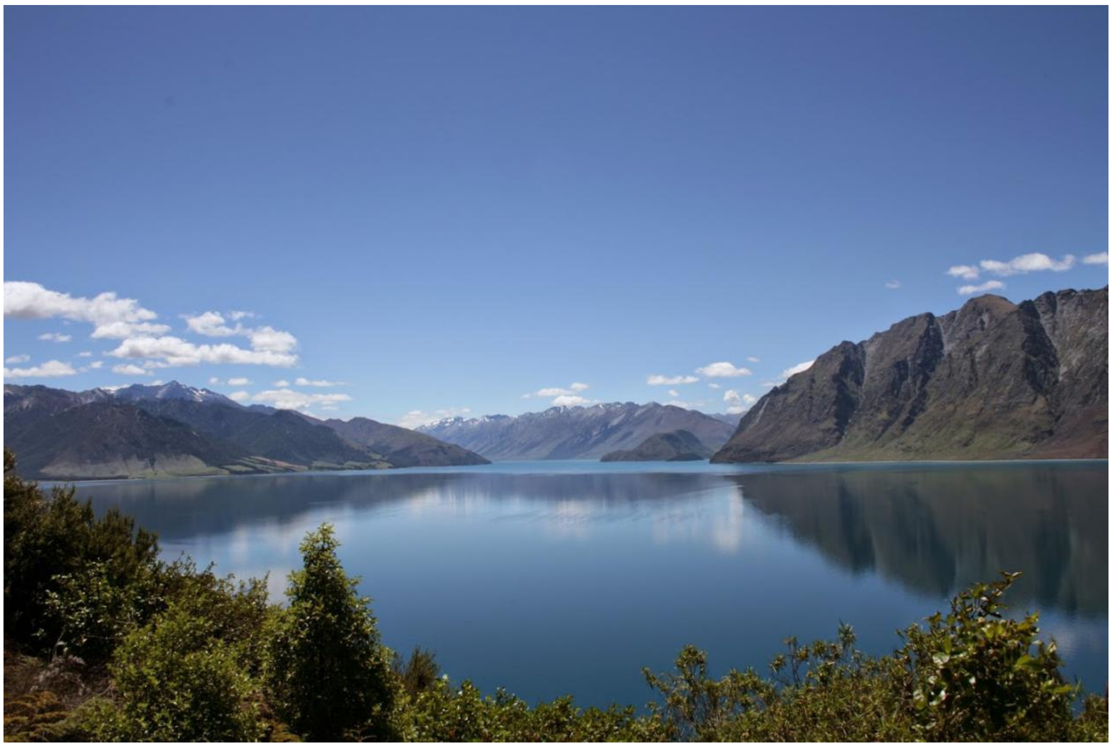
Construct mixed layer 9, the dimension will be 8*8*2048. These mixed layer consists of two major convolutional layers:branch1x1 with 320 filters and 1*1 kernel size, branch3x3db1 with 384 filters and 3*1 kernel size, Branch 3x3 with 384 filters and 3*1 kernel size, and an average pooling layer.

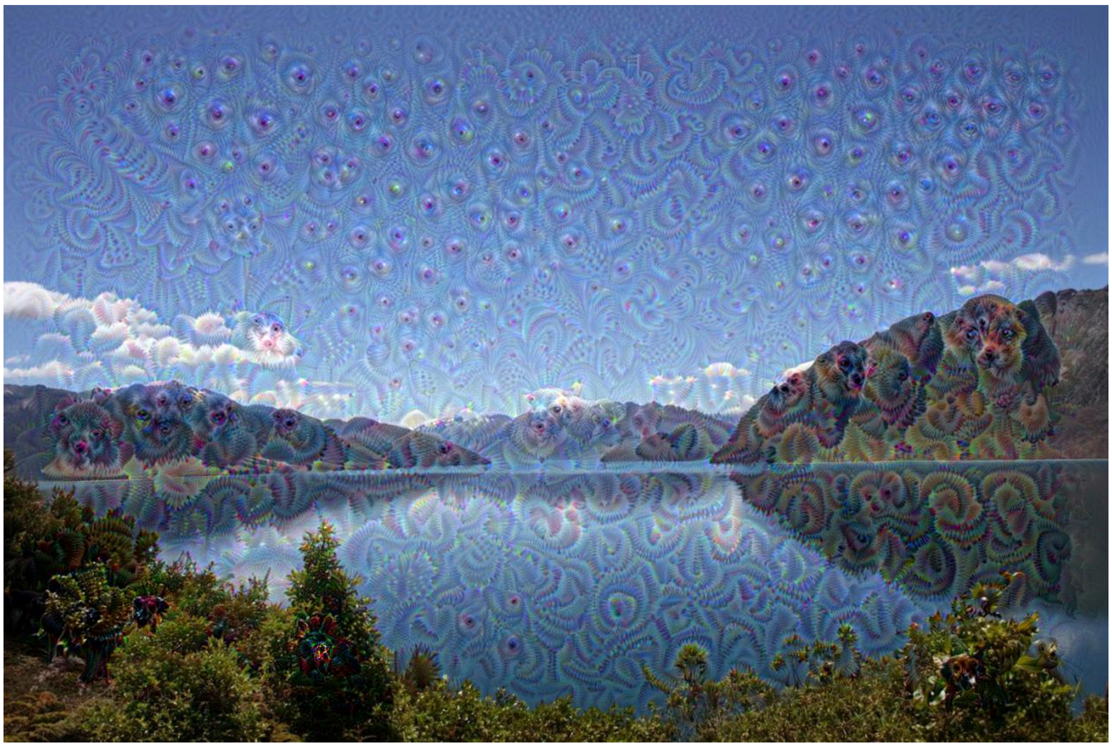
```
if include_top:
    # Classification block
    x = GlobalAveragePooling2D(name='avg_pool')(x)
    x = Dense(classes, activation='softmax', name='predictions')(x)
else:
    if pooling == 'avg':
        x = GlobalAveragePooling2D()(x)
    elif pooling == 'max':
        x = GlobalMaxPooling2D()(x)

# Ensure that the model takes into account
# any potential predecessors of `input_tensor`.
if input_tensor is not None:
    inputs = get_source_inputs(input_tensor)
else:
    inputs = img_input
# Create model.
model = Model(inputs, x, name='inception_v3')
```

The last part, create the model and return this model.

The picture I run using the default parameter

Original lake and sky



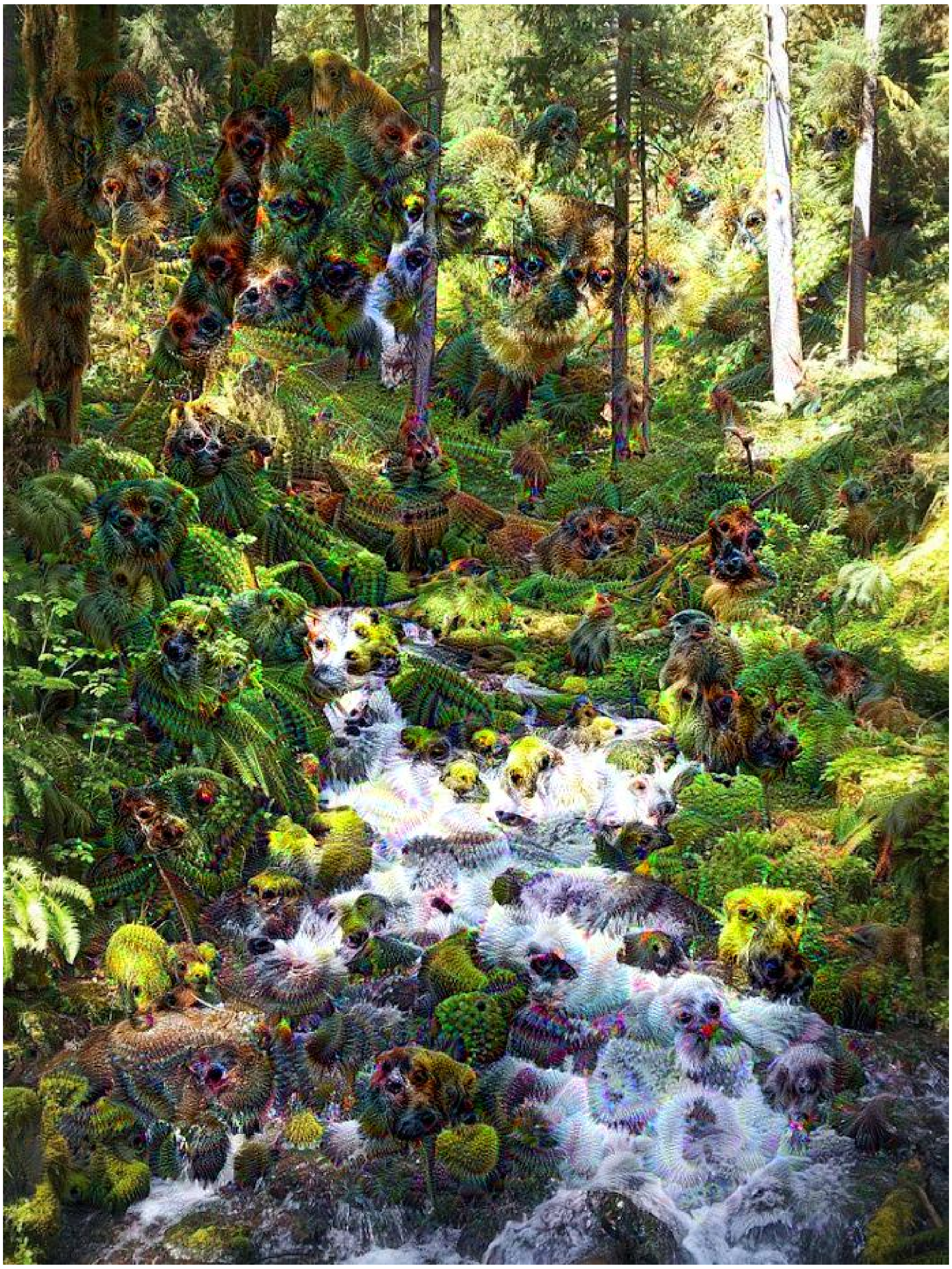


After deep dream network

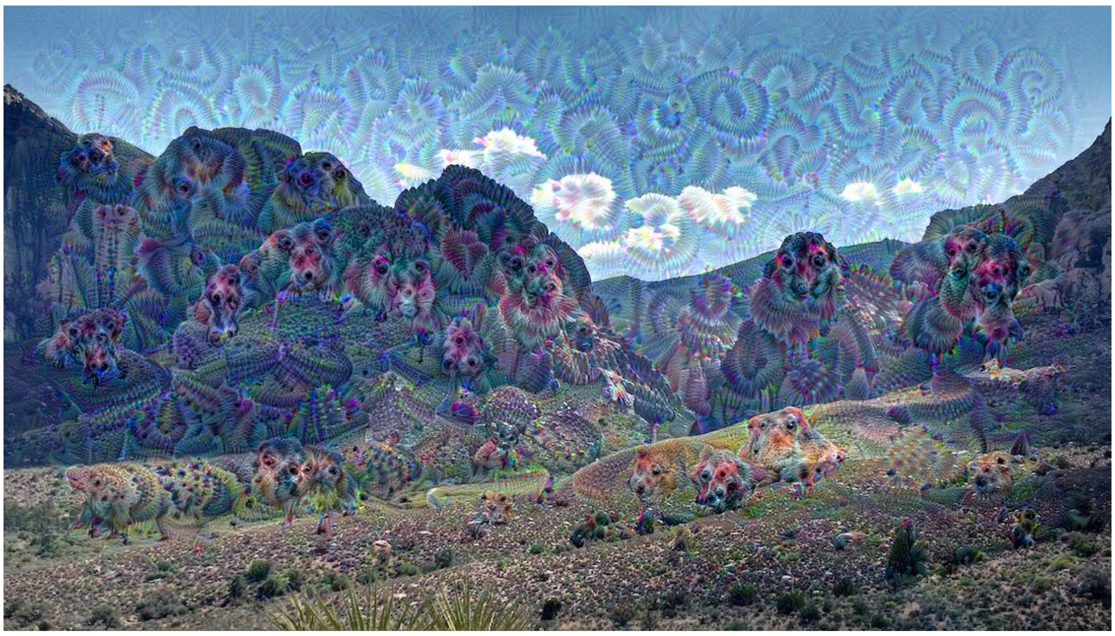
We can see that the cloud were inception with dog's head, the sky is full of eyes and ripples, the mountain is like squirrel. This is because we showed the network the library about these animals and features. The network detects the features in the network in different layers, and we choose the high level layer to enhance and told the network that “whatever you see there, show me more”(achieve this by setting different layers in model with different weights). This will cause the network to identify more sophisticated features in the image, complex features or even whole object tend to emerge. In addition, our choice of layers and enhancement will cause the network to step into a loop that if the mountain looks like a squirrel, the network will make it more like it. At last, a highly detailed squirrel will appear in the image.

Some other images are as follows

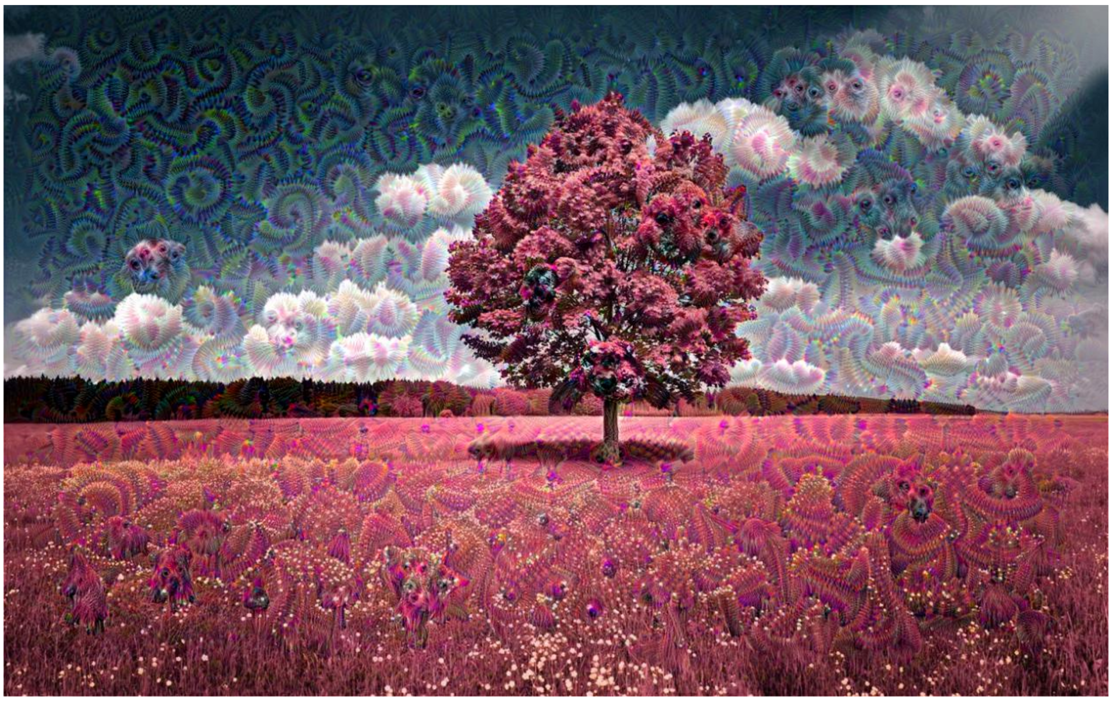
Waterfall



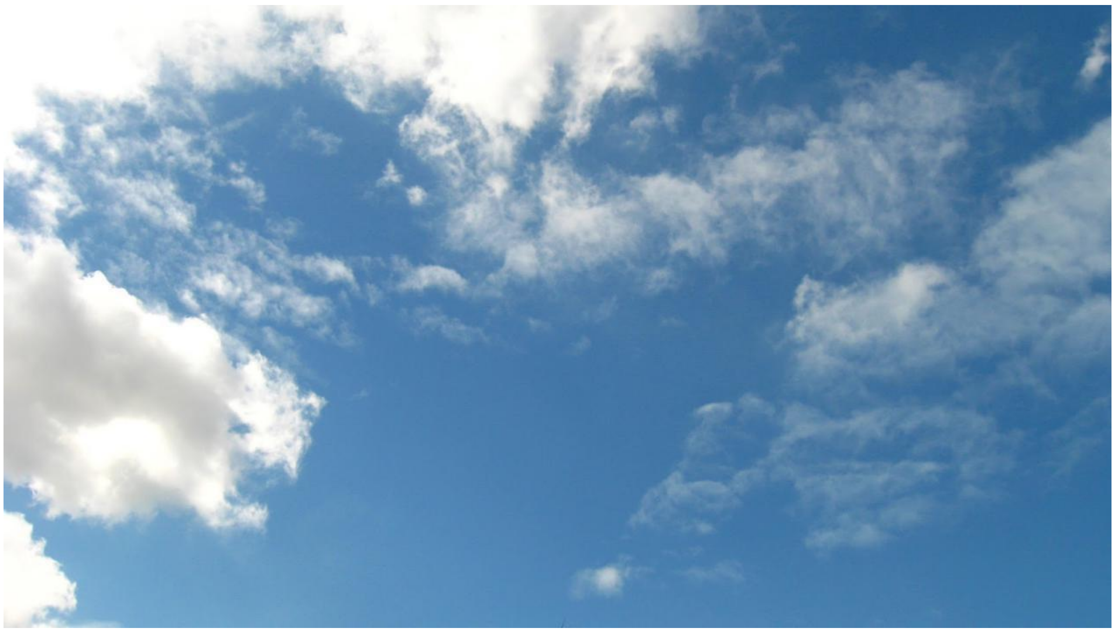
Mountain

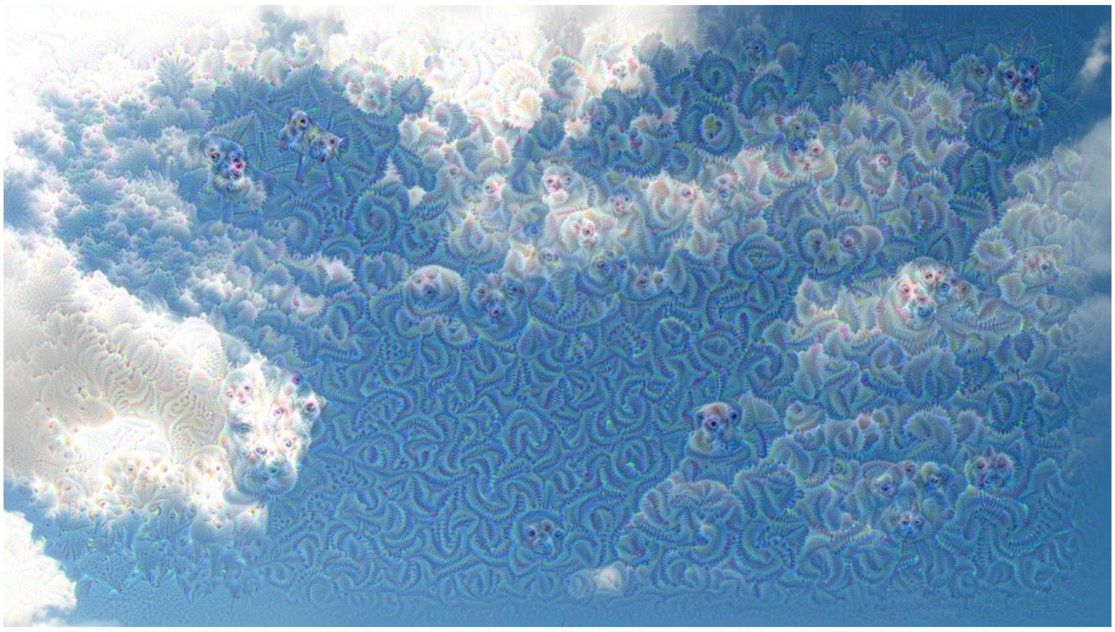


Redtree



Sky



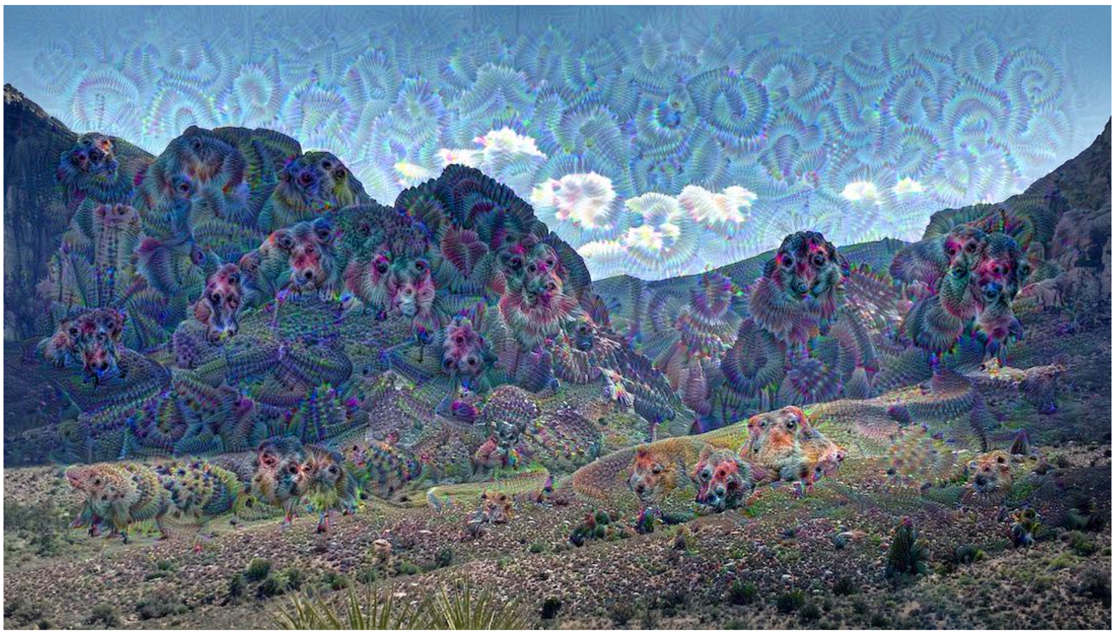


Parameter experiment

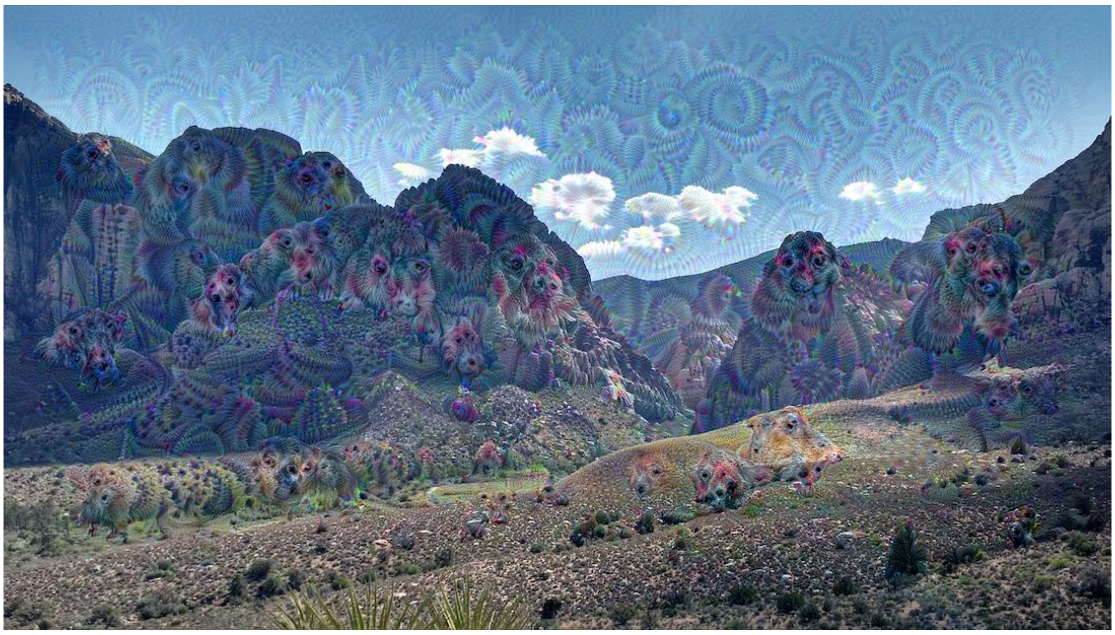
```
step = 0.01
num_octave = 3
octave_scale = 1.4
iterations = 20
max_loss = 10
```

The original parameter

The original after processed picture

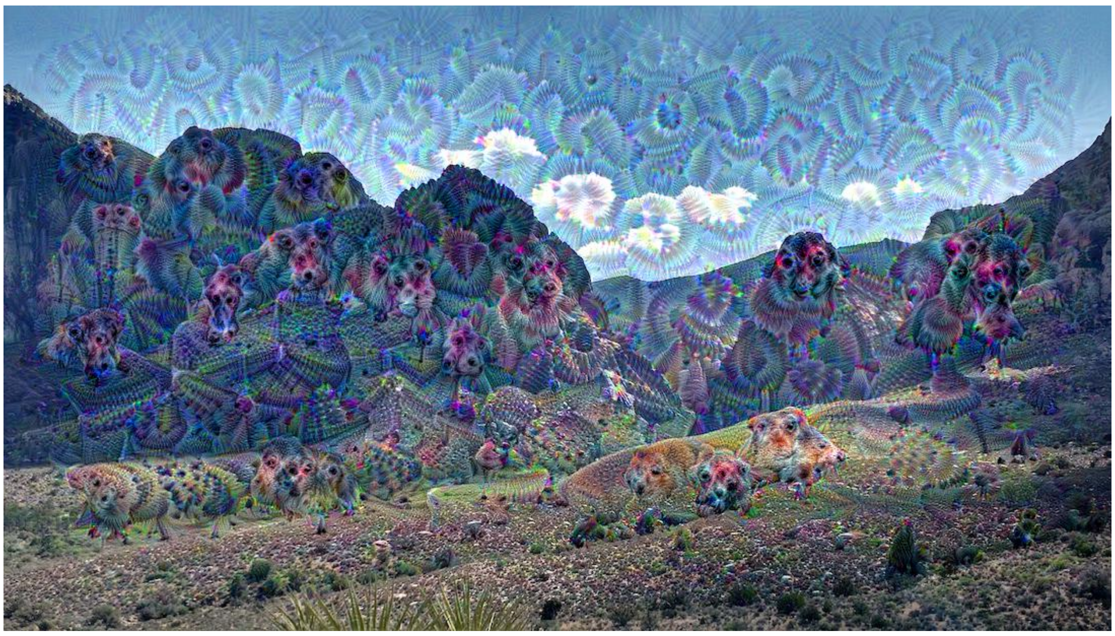


Set iteration as 10



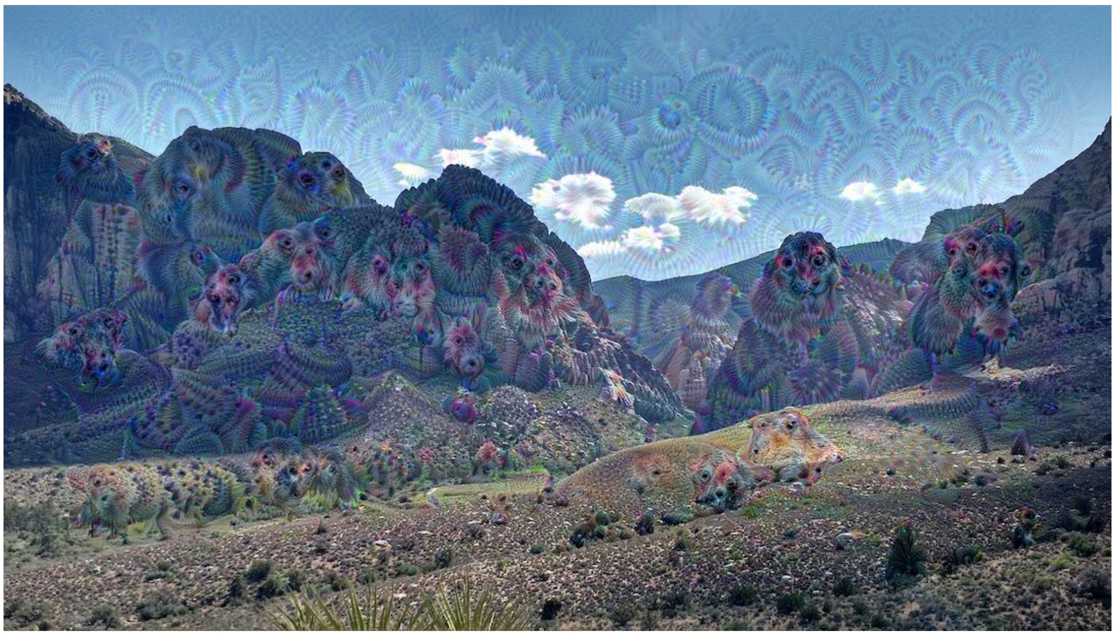
With the iteration cut as half, we can see that the ripple and crinkle in the figures become sparsely, the squirrel's head and other features in the figure become more natural.

Set iteration as 100



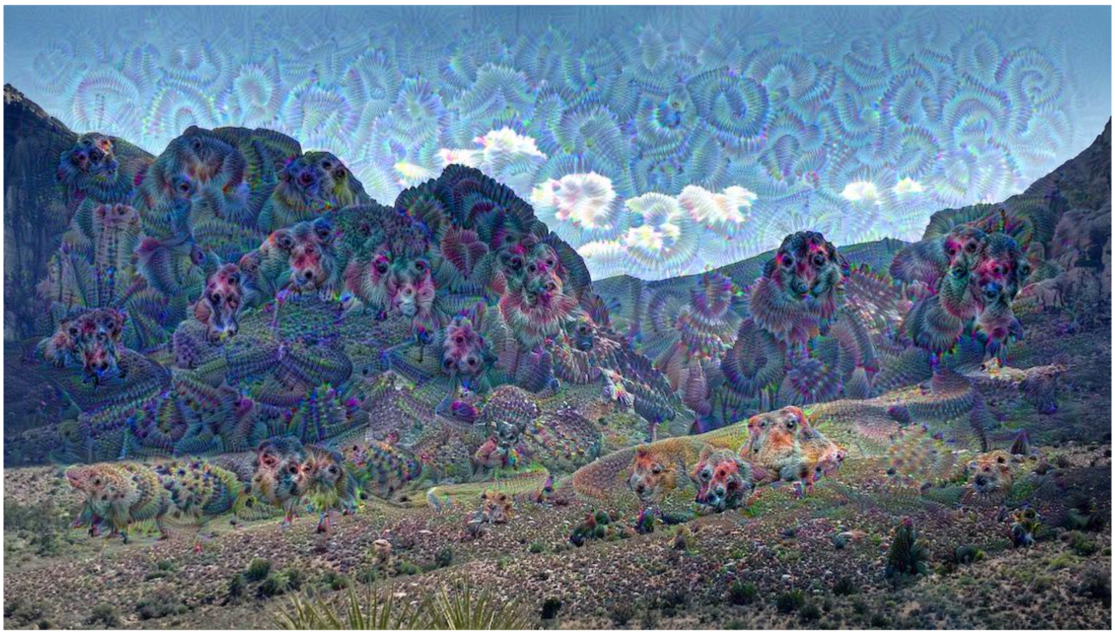
we can see from above figure that ripple and crinkle increase a lot, and the figure becomes wired. This is because we apply more iterations in the network, the original image was processed further.

Set loss as 5



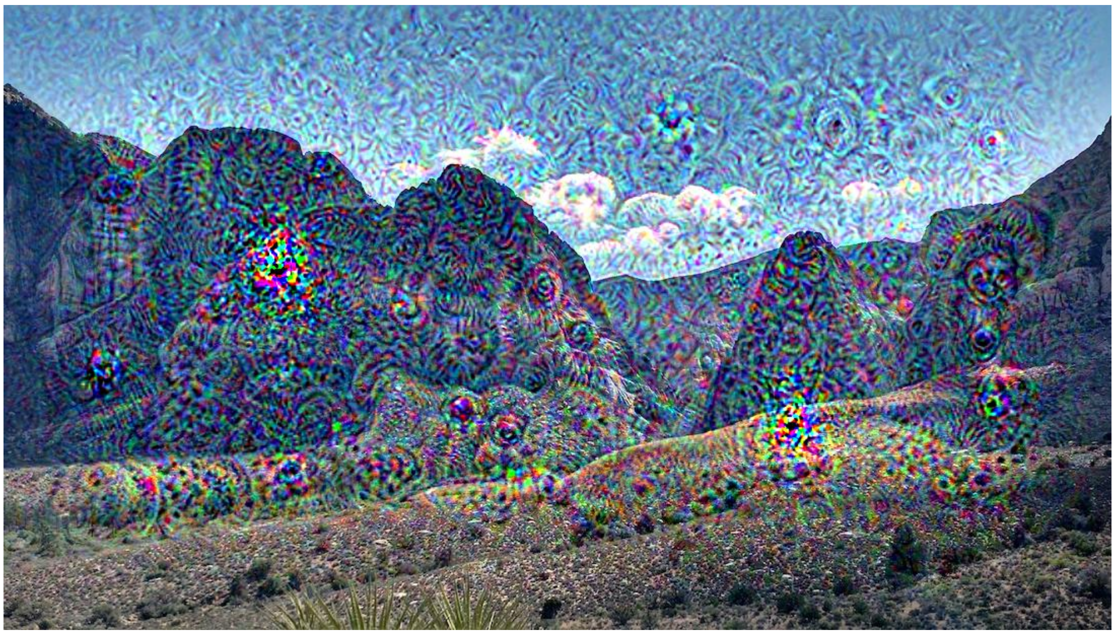
This change of loss parameter seems just achieve the same output as halving the iteration, the image of squirrel, eyes and other features become natural and real, ripple and crinkle appear sparsely compared to original figure.

Set loss as 20



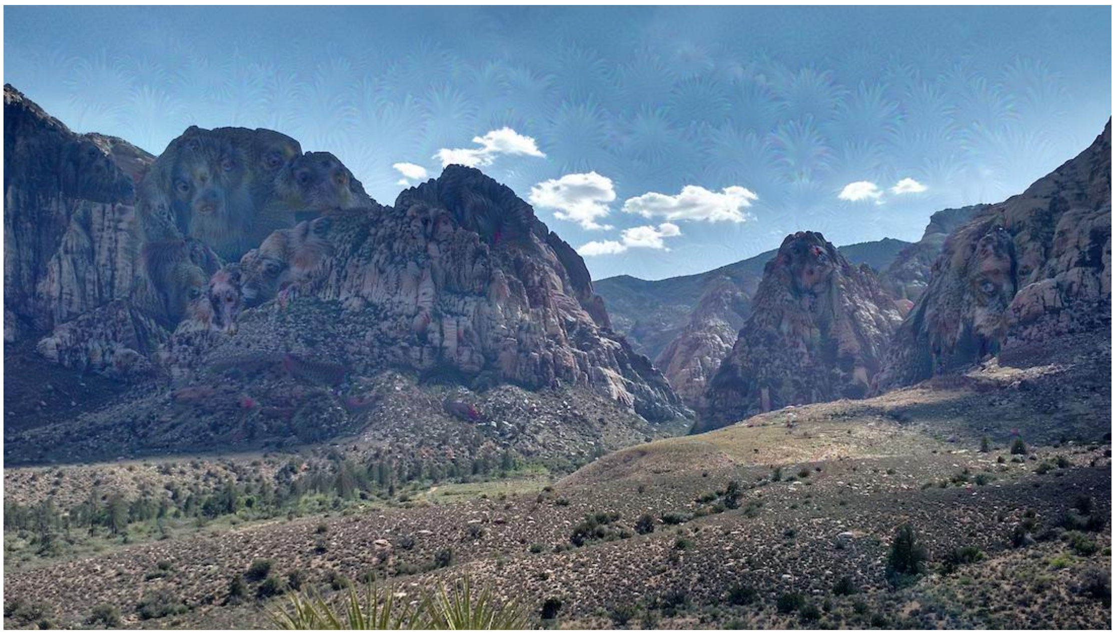
The output of changing loos is like the change of increasing iteration, we at last get a more wired image full of ripple and crinkle.

Set step as 0.1



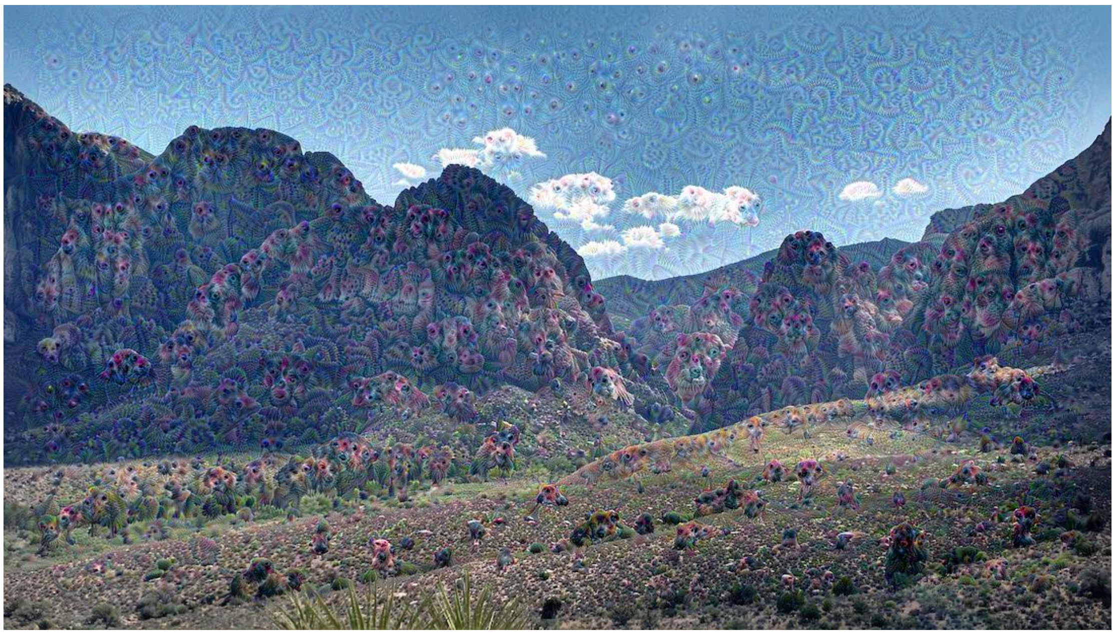
The processed figure becomes highly disturbed and fuzzy, we cannot see detailed animals and eyes emerged in the figure. Maybe the step is too large so that we passed some detailed features in the original images.

Set step as 0.001



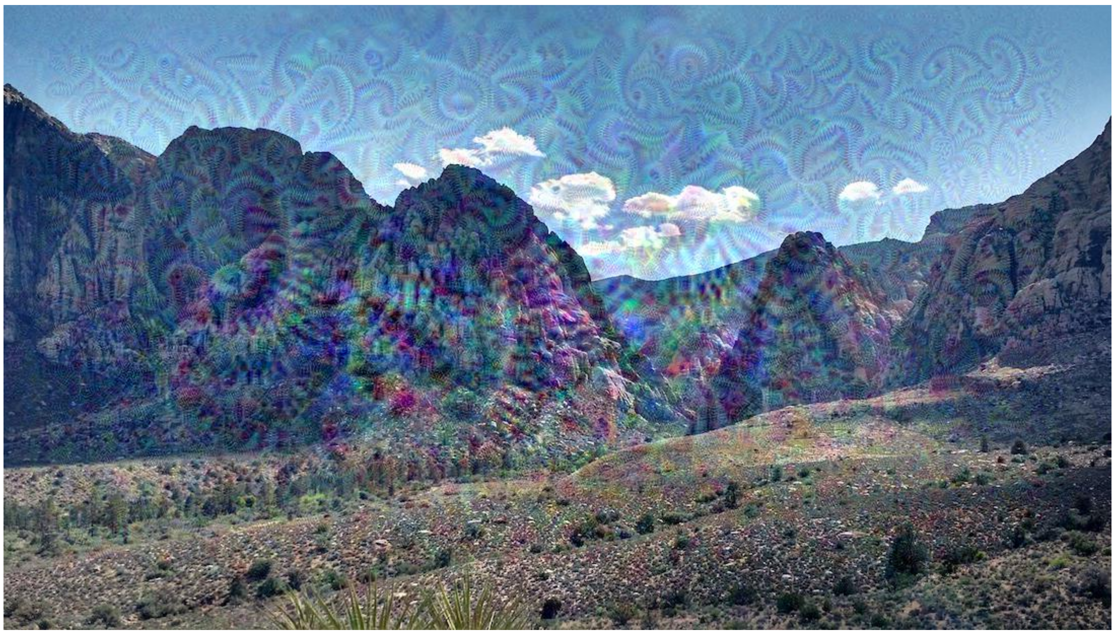
Applying a much smaller step in this time, but nothing happened. The inference is that if you set the step too small, the gradient and loss in each layer and iteration could only affect little in the image. Therefore, the processed image is just like original image before entering deep dream network.

Set scale as 0.7



We can see that the inception images(dogs, squirrels and eyes) becomes smaller in size, and one mountain contains hundreds of this inception images. This is because when we decrease the scale, we parsed and processed a smaller shape(size) in the original image, and we at last get the figure above.

Set scale as 5



If we increase the scale too much, we will parse the picture in a larger shape of original picture, this results in the lost of many detail features in image. Therefore, at last, we get this figure that inception images distribute sparsely.

Conclusion: Using the deep dream, we can figure out and understand the question that what does one particular layer see and deal with by allocating different weights to different layers. And we can also have a view about parameters(step, scale, iterations...) has which kind of effect on processing the images. In addition, it also help us to understand and visualize how neural network are able to carry out difficult classification tasks and check what the network learned during training.