In [2]:

```
%matplotlib inline
from PIL import Image
from io import BytesIO
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.optim as optim
import requests
from torchvision import transforms, models
```

In [3]:

```
print(torch.cuda.is_available())
```

```
False
```

In [4]:

```
# get the "features" portion of VGG19 (we will not need the "classifier" portion)
vgg = models.vgg19(pretrained=True).features
```

In [5]:

```
# freeze all VGG parameters since we're only optimizing the target image
for param in vgg.parameters():
    param.requires_grad_(False)


# move the model to GPU, if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
vgg.to(device)
```

Out[5]:

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace=True)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace=True)
  (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace=True)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace=True)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace=True)
  (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (17): ReLU(inplace=True)
  (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace=True)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace=True)
  (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (24): ReLU(inplace=True)
  (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (26): ReLU(inplace=True)
  (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace=True)
  (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (31): ReLU(inplace=True)
  (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (33): ReLU(inplace=True)
  (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (35): ReLU(inplace=True)
  (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
```

In [6]:

```
def load_image(img_path, max_size=400, shape=None):
    ''' Load in and transform an image, making sure the image
        is <= 400 pixels in the x-y dims.'''
    if "http" in img_path:
        response = requests.get(img_path)
        image = Image.open(BytesIO(response.content)).convert('RGB')
    else:
        image = Image.open(img_path).convert('RGB')
# large images will slow down processing
    if max(image.size) > max_size:
        size = max_size
    else:
        size = max(image.size)
    if shape is not None:
        size = shape
    in_transform = transforms.Compose([transforms.Resize(size),transforms.ToTensor(),transforms.Normalize((0.485,
0.456, 0.406),(0.229, 0.224, 0.225))])


# discard the transparent, alpha channel (that's the :3) and add the batch dimension
    image = in_transform(image)[:3,:,:].unsqueeze(0)
    print("after",image.shape)
    return image
```

In [7]:

```
# load in content and style image
content = load_image('/home/anudeep/Desktop/lion.jpg').to(device)
# Resize style to match content, makes code easier
style = load_image('/home/anudeep/Desktop/style.jpeg', shape=content.shape[-
2:]).to(device)
```

```
after torch.Size([1, 3, 366, 478])
after torch.Size([1, 3, 366, 478])
```

In [8]:

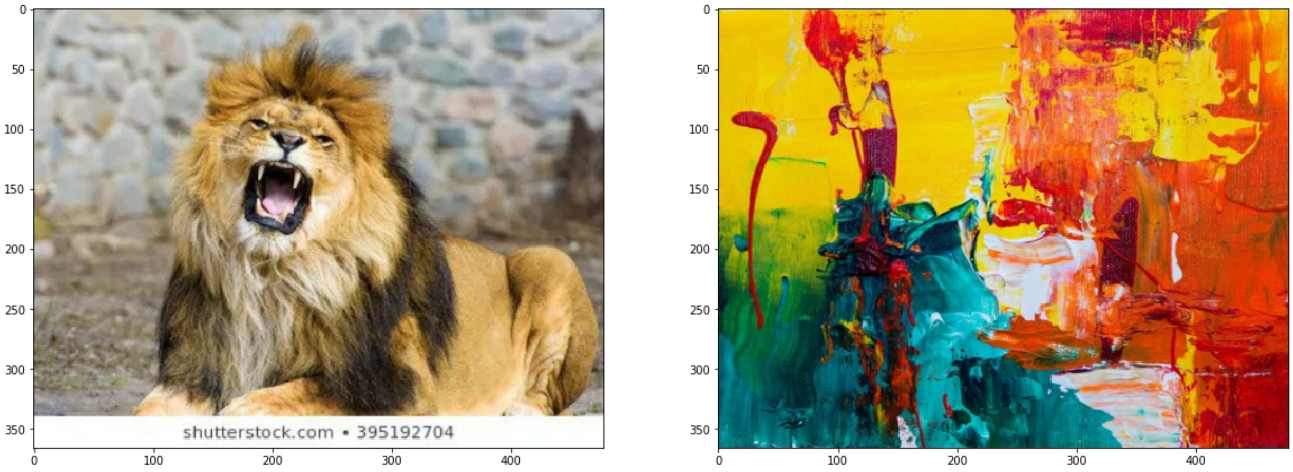
```
# helper function for un-normalizing an image
# and converting it from a Tensor image to a NumPy image for display
def im_convert(tensor):
#""" Display a tensor as an image. """
    image = tensor.to("cpu").clone().detach()
    image = image.numpy().squeeze()
#print(image.shape)
    image = image.transpose(1,2,0)#the new image say image' will have dimension x'-->y && y'-->z && z'-->x

#print(image.shape)
#print(image)
    image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
    image = image.clip(0, 1) #values smaller than zero will be rounded to zeros and values greater than one will
be rounded to one

    return image
```

In [9]:

```
# display the images
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
# content and style ims side-by-side
ax1.imshow(im_convert(content))

ax2.imshow(im_convert(style))
```

Out[9]:

```
<matplotlib.image.AxesImage at 0x7fc7c349e710>
```



In [10]:

```
def get_features(image, model, layers=None):
#""" Run an image forward through a model and get the features for
#a set of layers. Default layers are for VGGNet matching Gatys et al (2016)
#"""
## TODO: Complete mapping layer names of PyTorch's VGGNet to names from the paper
## Need the layers for the content and style representations of an image
    if layers is None:
        layers = {'0': 'conv1_1',
                  '5': 'conv2_1',
                  '10': 'conv3_1',
                  '19': 'conv4_1',
                  '21': 'conv4_2', ## content representation
                  '28': 'conv5_1'}
    features = {}
    x = image
# model._modules is a dictionary holding each module in the model
    for name, layer in model._modules.items():
        x = layer(x)
        if name in layers:
            features[layers[name]] = x

    return features
```

In [11]:

```
def gram_matrix(tensor):
    #""" Calculate the Gram Matrix of a given tensor Gram Matrix: https://en.wikipedia.org/wiki/Gramian_matrix
    # get the batch_size, depth, height, and width of the Tensor
    _, d, h, w = tensor.size()
    # reshape so we're multiplying the features for each channel
    tensor = tensor.view(d, h * w)
    # reshape so we're multiplying the features for each channel
    gram = torch.mm(tensor, tensor.t())
    return gram
```

In [12]:

```
# get content and style features only once before training
content_features = get_features(content, vgg)
style_features = get_features(style, vgg)
# calculate the gram matrices for each layer of our style representation
style_grams = {layer: gram_matrix(style_features[layer]) for layer in style_features}
# create a third "target" image and prep it for change
# it is a good idea to start off with the target as a copy of our *content* image
# then iteratively change its style
target = content.clone().requires_grad_(True).to(device)
# weights for each style layer
# weighting earlier layers more will result in *larger* style artifacts
# notice we are excluding `conv4_2` our content representation
style_weights = {'conv1_1': 1.,'conv2_1': 0.75,'conv3_1': 0.2,'conv4_1': 0.2,'conv5_1': 0.2}
content_weight = 1 # alpha
style_weight = 1e6 # beta
print(style_weight)
```

```
1000000.0
```

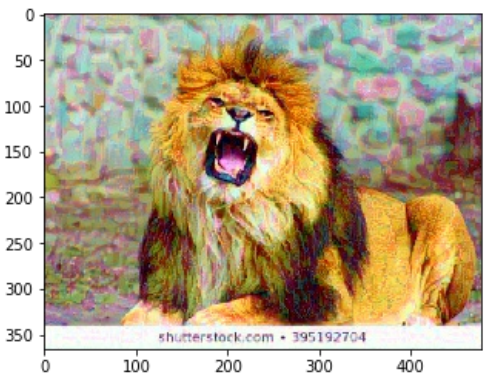
In [13]:

```
# for displaying the target image, intermittently
show_every = 400
# iteration hyperparameters
optimizer = optim.Adam([target], lr=0.003)
steps = 2000 # decide how many iterations to update your image (5000)
```

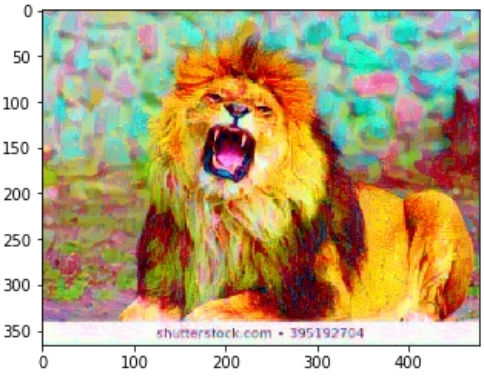
In [14]:

```
for ii in range(1, steps+1):
    # get the features from your target image
    target_features = get_features(target, vgg)
    # the content loss
    content_loss = torch.mean((target_features['conv4_2'] - content_features['conv4_2'])**2)
    # the style losshttps://nationalzoo.si.edu/sites/default/files/newsroom/20190226-bridgetisrael08.jpg

    # initialize the style loss to 0
    style_loss = 0
    # then add to it for each layer's gram matrix loss
    for layer in style_weights:
        # get the "target" style representation for the layer
        target_feature = target_features[layer]
        target_gram = gram_matrix(target_feature)
        _, d, h, w = target_feature.shape
        # get the "style" style representation

        style_gram = style_grams[layer]
        # the style loss for one layer, weighted appropriately
        layer_style_loss = style_weights[layer] * torch.mean((target_gram - style_gram)**2)
        # add to the style loss
        style_loss += layer_style_loss / (d * h

                                          * w)
        # calculate the *total* loss
    total_loss = content_weight * content_loss + style_weight * style_loss
    # update your target image
    optimizer.zero_grad()
    total_loss.backward()
    optimizer.step()
    if ii % show_every == 0:
        print('Total loss: ', total_loss.item())
        plt.imshow(im_convert(target))
        plt.show()
```

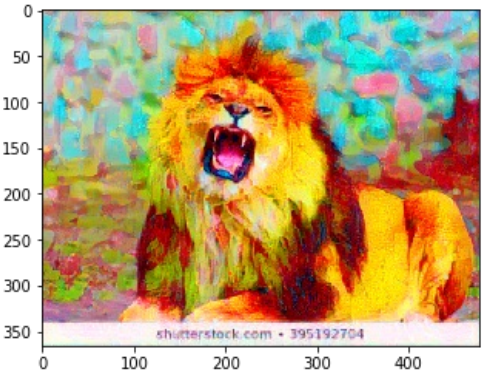
```
Total loss:  158464816.0
```



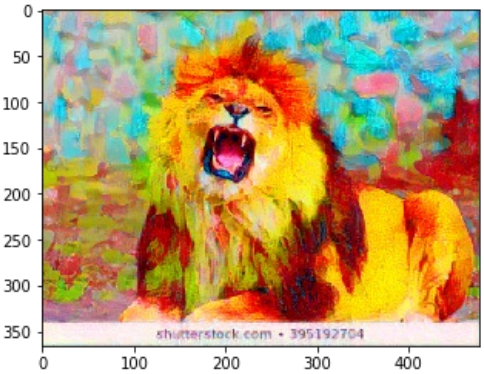
```
Total loss:  45037716.0
```



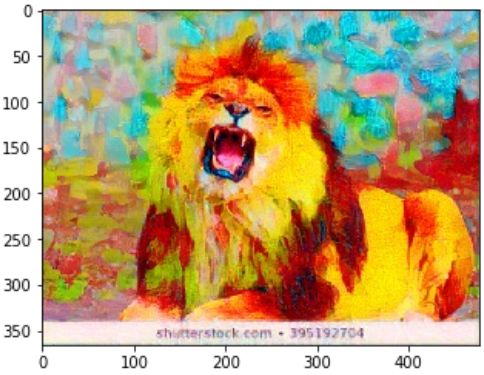

```
Total loss:  14411135.0
```




```
Total loss:  8105792.5
```




```
Total loss:  5410263.5
```



In [ ]: