Neural Style Transfer: Feature extractor, Training Loop

paper (https://arxiv.org/pdf/1508.06576.pdf)

The objective of *Neural Style Transfer*:

- Given Content Image C
- Given Style Image S
- Create Generated Image G that is the Content image re-drawn in the "style" of the Style image

Style image S **Content image C** **Generated image G**

Neural Style Transfer highlights several themes we will encounter in the course

- The essential element of Deep Learning is
    - defining a Loss function that captures the semantics of the task
    - architecture is less important: just a tool
- The intermediate representations of a Deep Network has meaning, that can be leveraged
- Re-using existing models in novel ways leads to powerful results

We will explain ideas with reference to code.

here (https://www.tensorflow.org/tutorials/generative/style_transfer) is a tutorial view of the notebook.

Loss function: sum of Content Loss and Style Loss

We create a Loss Function $\mathcal{L}(C, G, S)$ and solve for the optimal generated image G^*

$$G^* = \underset{G}{\operatorname{argmin}}\, \mathcal{L}(C, G, S)$$

where $\mathcal{L}(C, G, S)$ is the sum of

- "content loss" $\mathcal{L}_{\text{content}}$: dissimilarity of "content" of G and C
- "style loss" $\mathcal{L}_{\text{style}}$ dissimilarity of "style" of G and S

We solve for G^* using $\frac{\partial \mathcal{L}}{\partial G}$

- depending on how we write $\mathcal{L}$
    - minimize dissimilarity: Gradient Descent
    - maximize similarity; Gradient Ascent

That is: the "weights" we are optimizing are the *pixels of image* I.

How do we measure the dissimilarity of the "content" ?

We can't just use plain MSE of the pixel-wise differences

- G is different than C, by definition (the "styles" are different)

And how do we define what the "style" of an image is ?

- And how do we measure dissimilarity of the "style" ?

We will use an *alternate representation* of each image

- such that we can compare *alternate representations* in a useful sense

The goal of using an alternate representation of an image

- is to capture the "semantics" (deeper, non-surface meaning) of an image
- rather than "syntax" (superficial surfrace meaning, literal pixels) of an image

-

Representation of intermediate layers as a measure of Style and Content

Recall that each layer in a multi-layer Neural Network is creating an *alternate representation* of the input.

Rather than directly comparing G with C (and G with S) our dissimilarity will be measured

- Not on raw images as seen by the human eye
- But on their alternate representations as created at some layer of a multi-layer Neural Network

That is: we will *re-use a model* $\mathbb{C}$ (e.g., VGG19)

- originally designed for Image Classification
- as a means to create two alternate representations of an image
- one alternate representation of image I will encode the "content" of I
- the other alternate representation of image I will encode the "style" of I

Suppose $\mathbb{C}$ consists of a sequence of CNN Layers

Let $\mathbb{C}_{(l)}$ denote the set of $n_{(l)}$ feature maps produced at layer l

- Feature map: value of one feature, at each spatial location
- $\mathbb{C}_{(l),j}$: feature map j

We choose

- One layer l_c of $\mathbb{C}$ and call it the "content representation" layer
    - Will tend to be shallow: closer to the input
    - Features of shallow layers will be more "syntax" than "semantics"
- One layer l_s of $\mathbb{C}$ and call it the "style representation" layer
    - Will tend to be deep: closer to the output
        - Features of deep layers will be more "semantics" than "syntax"

For arbitrary image I, let

- $\mathbb{C}_{(l_c)}(I)$
  - denote the feature maps of the Classifier $\mathbb{C}$, on image I, at the "content representation" layer
- $\mathbb{C}_{(l_s)}(I)$
  - denote the feature maps of the Classifier $\mathbb{C}$, on image I, at the "style representation" layer

Using the alternate representations derived from $\mathbb{C}$ we can define

- $\mathcal{L}_{\text{content}}$ as the dissimilarity of $\mathbb{C}_{(l_c)}(C)$ and $\mathbb{C}_{(l_c)}(G)$
- $\mathcal{L}_{\text{style}}$ as the dissimilarity of $\mathbb{C}_{(l_s)}(S)$ and $\mathbb{C}_{(l_c)}(G)$

Content Loss $\mathcal{L}_{\text{content}}$

We can now define the similarity of the "content" of Content Image C and "content" of Generated Image G

- by comparing $\mathbb{C}_{(l_c)}(C)$ and $\mathbb{C}_{(l_c)}(G)$
- using sum of pixel-wise squared difference (MSE) of $\mathbb{C}_{(l_c)}(C)$ and $\mathbb{C}_{(l_c)}(G)$

Here is the code for content loss $\mathcal{L}_{\text{content}}$

- `base` is $\mathbb{C}_{(l_c)}(C)$: the alternate representation of content image C
- `combination` is $\mathbb{C}_{(l_c)}(G)$ the alternate representation of generated image G

```
def content_loss(base, combination):
    return tf.reduce_sum(tf.square(combination - base))
```

Style Loss $\mathcal{L}_{\mathbf{style}}$

Similarly, we can define the similarity of the "style" of Content Image C and "style" of Generated Image G

- by comparing $\mathbb{C}_{(l_s)}(S)$ and $\mathbb{C}_{(l_s)}(G)$

For any image I: $\mathbb{C}_{(l)}(I)$ consists of $n_{(l)}$ feature maps.

We need to define what it means to compare $\mathbb{C}_{(l)}(I)$ and $\mathbb{C}_{(l)}(I')$.

The *Gramm Matrix* $\mathbb{G}$ of $\mathbb{C}_{(l)}(I)$

- Has shape ($n_{(l)} \times n_{(l)}$)
- $\mathbb{G}_{j,j'}(I) = \mathrm{correlation}(\mathrm{flatten}(\mathbb{C}_{(l),j}(I)),$
  $\mathrm{flatten}(\mathbb{C}_{(l),j'}(I)))$
    - the correlation of the feature map j of $\mathbb{C}_{(l)}(I)$ with feature map j' of $\mathbb{C}_{(l)}(I')$

Intuitively, the Gramm Matrix

- measures the correlation of the values across pixel locations (flattened feature maps) of two feature maps of image I

Here is the code computing the "style loss" $\mathcal{L}_{\text{style}}$

- `style` is $\mathbb{C}_{(l_s)}(S)$: the alternate representation of style image S
- `combination` is $\mathbb{C}_{(l_s)}(G)$ the alternate representation of generated image G

```
def style_loss(style, combination):
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_nrows * img_ncols
    return tf.reduce_sum(tf.square(S - C)) / (4.0 * (channels**2) * (size**2))
```

Gradient ascent: generating G

We can find image G via Gradient Ascent

- Ascent versus Descent: we have measured *similarity* (correlation) rather than *dissimilarity*
- Initialize G to noise
- Update pixel $G_{i,i',k}$ by $\frac{\partial \mathcal{L}}{G_{i,i',k}}$

Feature extractor

One key coding trick that we will illustrate

- Obtaining the feature maps of the Classifier $\mathbb{C}$, on image I, at an arbitrary layer

We will call this tool the *feature extractor*

```
# Build a VGG19 model loaded with pre-trained ImageNet weights
model = vgg19.VGG19(weights="imagenet", include_top=False)

# Get the symbolic outputs of each "key" layer (we gave them unique names).
outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])

# Set up a model that returns the activation values for every layer in
# VGG19 (as a dict).
feature_extractor = keras.Model(inputs=model.inputs, outputs=outputs_dict)
```

The `feature_extractor` code returns a dictionary

- mapping layer name to alternate representation at that layer

It is used within the `compute_loss` function

- to simultaneously compute (through threading on the first dimension) the alternate representations
- of the three images C, S and G
- `base_image` is the content image C
- `style_reference_image` is the style image S
- `combination_image` is the generated image G

```
def compute_loss(combination_image, base_image, style_reference_image):
    input_tensor = tf.concat(
        [base_image, style_reference_image, combination_image], axis=0
    )
    features = feature_extractor(input_tensor)
    ....
```

The alternate representations (at all layers) of each image

- is extracted from `features` later in the code
- here is the code using the alternate representations to eventually compute `style_loss`
    - n.b., $\mathcal{L}_{\mathrm{sty[e}}$ is computed here over *several layers*
    - here is the code for one layer named `layer_name`

```
layer_features = features[layer_name]
style_reference_features = layer_features[1, :, :, :]
combination_features = layer_features[2, :, :, :]
```

Computing gradients

Here is the code to *enable* gradients to be computed

- `tf.GradientTape` records the forward pass
- to facilitate computation of gradients in the backward pass

You may want to review (from the Intro course)

- Back propagation: forward and backward pass (Training_Neural_Network_Backprop.ipynb)
- Computing analytic gradients in Keras (Training_Neural_Network_Operation_Forward_and_Backward_Pass.ipynb)

```
@tf.function
def compute_loss_and_grads(combination_image, base_image, style_reference_imag
e):
    with tf.GradientTape() as tape:
        loss = compute_loss(combination_image, base_image, style_reference_imag
e)
    grads = tape.gradient(loss, combination_image)
    return loss, grads
```

Recall the variables

- `base_image` is the content image C
- `style_reference_image` is the style image S
- `combination_image` is the generated image G

Training loop

Here is the code for the training loop

- `optimizer.apply_gradients([(grads, combination_image)])`
- updates the "weights" `combination_image` using gradients `grads`

```
```

```
optimizer = keras.optimizers.SGD(
    keras.optimizers.schedules.ExponentialDecay(
        initial_learning_rate=100.0, decay_steps=100, decay_rate=0.96
    )
)

...

iterations = 4000
for i in range(1, iterations + 1):
    loss, grads = compute_loss_and_grads(
        combination_image, base_image, style_reference_image
    )
    optimizer.apply_gradients([(grads, combination_image)])
```

Notebooks links

Full notebook (https://www.tensorflow.org/tutorials/generative/style_transfer) is a tutorial view of the notebook we used for the code snippets.

In [2]:
```
print("Done")
```

```
Done
```