neural_art

July 27, 2017

```
In [1]: from IPython.core.display import HTML
        HTML("""
        <style>

        div.text_cell_render { /* Customize text cells */
        font-family: 'Arial';
        font-size:1.5em;
        line-height:1.4em;
        padding-left:3em;
        padding-right:3em;
        }
        </style>
        """)
```

```
Out[1]: <IPython.core.display.HTML object>
```

A Neural Algorithm of Artistic Style

Code here follows the TensorFlow implementation by Mark Chang found here.

NOTE: this notebook was tested with Python 3, but it *should* also work with Python 2 as long as you have TensorFlow, six, and other required libraries. However, it hasn't been *tested* with Python 2, so expect problems, and if you find one, open a pull request or file an issue, etc.

0.1 Introduction to Deep Learning

What is deep learning? In short it is the use of **neural networks with many hidden layers**.

What is a neural network? It is a (non-linear) mathematical model. We need to a short digression for context. Let's rush through the ideas of:

- mathematical modeling
- regression
- classification

0.1.1 Mathematical Modeling

A *mathemaical model* is a system of equations or numerical recipes used as a proxy for another system. Examples of mathematical models include:

- stock market and weather simulation programs,

- budgeting spreadsheets
- physics equations

There is a lot of flexibility in the idea of a mathematical model, so don't get hung up on specific semantics.

0.1.2 Regression

Suppose you had some data relating an independent variable to a dependent variable. In mathematical modeling which variable is dependent and which is independent can sometimes be difficult to say, but suppose we decided to look for a relationship between age and height. In this case it seems reasonable to assert that height is dependent on age, and not the other way around.

So, let's take some data by finding four individuals and plot their height versus their age - in other words, let's put age on the x-axis of a plot and height on the y-axis. How can we use this data to build a mathematical model that takes age as an input and produces a predicted height?

This is an example of a *regression* problem - we want to predict a numerical outcome. In this case, we might choose to write down a simple linear model like

```
height = [some factor] times age + [some other factor]
```

How do we determine what those factors are? There are numerical recipes to we can follow. The basic idea is to start with a "reasonable guess", look at the *error* in the guess, and adjust our guesses to make the error smaller. We do this, generically, by taking the *derivative* of the error function and moving our parameters in the direction of the gradient.

0.1.3 Classification

Another sort of problem we might like to model is *classifiction* - given some attributes, we'd like to say what "category" an example belongs to. In this sort of problem we think less about dependent and independent variables, and more about just attributes.

For example, we might want to categorize dogs based on their length from nose to tail and height from ground to shoulder. It might be possible to again write down a linear model relating the two:

```
height = [some factor] times length + [some other factor]
```

We again fit for these parameters, but now the goal is not to use one to predict the other, bu rather to say everything on one side of this line is in one class, and everything on the other side of the line is in the other.

0.1.4 Problems

Sometimes (all the time), we get data that doesn't fit our simplest models. Returning to the height vs age model, suppose we find a 5th person and add them to the model:

What do we do here? We could consider adding more independent variables (e.g., gender, weight, eye color) - some will have more explanatory power than others! Now our model looks like

```
height = [factor_1] * age + [factor_2] * gender + [factor_3] * weight + [factor_4]
```

How would we include gender here? Isn't it a binary option? There are tricks we can use to get around this issue that I won't go into now - the bottom line is we can use it here.

We could also make the model non-linear. For example, we could write:

```
height = [factor_1] * age^2 + [factor_2]
```

This may or may not work better. In this particular case, we would expect a quadractic model to break down except, perhaps, over very specific age ranges (the *range of validity* of a model is always an important concern). But non-linear models in general are very powerful.

For example, no linear model could effectively separate these classes:

But with a non-linear model, separating them is trivial.

What if we wanted a model that could separate dogs by breed by looking at pictures? Could we show these pictures to a mathematical model and ask which is the husky and which is the eskimo dog?

There is a computer vision "challenge" organized around exactly this question - ImageNet. Sadly this year was ImageNet's last year. It has played a key role in the evolution of deep learning.

For many years, people managed to beat the previous year's best ImageNet scores (where scores are computed based on how many pictures a computer algorithm could correctly identify). Then, in 2012, there was a *huge* leap forward. This leap was based on a deep neural network.

Neural networks are non-linear models that operate by taking a set of inputs, applying a weight factor to each of them, adding the values up, and then applying a function to that sum to produce an output:

In principle, neural networks can approximate any function with enough nodes in their "hidden layer" (the nodes between the inputs and the output) - there is a theorem one may prove to sow this is true. However, we may have to make our network very, very wide to accomodate complex behavior. It can actually be more efficient to make a network "deep" by chaining together many hidden layers - if we use the output from one layer as the inputs to another, we can, in principle, make an arbitrarily "deep" network.

To fit, or *train* a neural network, we use a trick called back-propagation. Essentially, this is the same as what we discussed before - one looks at the error on the predictions and takes the derivative of the objective function with respect to the parameters in the network. We call this "training a network" because typically need many, many examples to fit our full model. The reason we need so many events is because we often have a huge number of parameters in our network.

One important problem when using a neural network on images is that a "naive" network that just connects neurons to all the pixels and assigns weights to everything regards these two images are completely distinct:

But this the same image, only translated inside the "viewing window"!

It turns out we can manage this problem with an idea that is more than 20 years old - we can use a convolutional neural net. The central idea is rather than giving each input its own unique weight, we will build a small "patch" and map it over the image so all the inputs *share the weights*. We will use multiple *feature maps* (map many different patches at once) so we can find a variety of different structures:

If we build a stack of these layers, we actually "shrink" the input image and move from a visual space into another space that we can think of as a semantic space. This is core to the network that won ImageNet in 2012 and ushered in the popularity of deep learning:

The action of the "convolution" of these filters, or kernels, with the image depends on the values in the kernels. A variety of image processing techniques rely on this. For example, a kernel with a negative value in the center surrounded by positive values becomes an *edge detector*:

The clever idea with deep convolutional neural networks for image analysis is we let the neural network *learn* which filters are effective for the problem we're trying to solve, rather than guessing them.

One last trick we will need to mention here is that to contain the number of parameters in a network we sometimes use *pooling* - this is a form of downsampling that we can use to reduce the dimensionality of a problem (at the price of some information loss).

In a sign of how fast things are moving with computer hardware, pooling is becoming less necessary as new GPUs with a lot of memory hit the market - but we use it here.

0.2 Introduction to Style Transfer

Let's explore artistic style transfer. Why?

- Sort of a cool demonstration.
- It turns out to be very simple, technically - so we get a chance to see some of the machinery of a deep learning library work without needing a huge variety of functions from the library. In particular, we use pooling (so we need to see some tensor operations) and stochastic graident descent (so we need to see an optimizer), but we aren't building convolutions or making recurrent connections, etc.

We will use a pre-trained convolutional network built for object classification (in this case, the 2014 ImageNet competition) and we will study activations in the model weights. Rather than varying the weights to achieve a task, we're actually going to vary the *input image*.

The key idea of A Neural Algorithm of Artistic Style is that **representations of image style and content may both be found in the activations of layers in a convolutional neural network and these representations are *separable*** (we may manipulate them independently).

The authors speculate about biological analogs to the mechanisms they employ here and argue that the ability to see and appreciate style is a *necessary consequence* of being able to visually process images into high-level representations of *content* (as opposed to collections of unordered colors and edges).

An important claim the paper makes is that as one progresses through the convolutional pyramid the input image is converted into representations that are increasingly concerned with the content of an image and less with the specific location of each pixel. Content representation is encoded in the activations of different filters in the network.

Image style is also built from the filter responses in each layer of the network. It uses the "*correlations* between the different filter responses over the spatial extent of the feature maps" (emphasis added). One may define a stlye based on a single layer or any arbitrary number of layers. The authors argue that using the maximum number of layers produces the best stylistic image.

Because the direct values in a filter and the correlations between them are separable (to an extent), we can separate content and style, and manipulate them separately.

As evidence they offer a figure that shows image reconstruction based on the activations in a given layer. The image is suggestive of their conlcusions, if not definitive:

Image taken from https://arxiv.org/abs/1508.06576

Here, as we go from (a) to (e) in the style reconstruction we are including first one, then two, then three, etc. layers when computing the correlations between different filter responses.

Note: we can recreate images like this (sort of) by turning off the parts of the cost function that affect style or content, and by choosing which layers from the pre-trained classifer we want to retain.

Image taken from https://arxiv.org/abs/1508.06576

Here, as we traverse columns left to right, we are increasing the weight on the content when computing the loss function.

As we traverse rows from top to bottom, we are increasing the number of layers used in computing the style representation.

0.3 Code

```
In [2]: from __future__ import print_function
        from __future__ import division
        import tensorflow as tf
        import numpy as np
        import scipy.io
        import scipy.misc
        from skimage.transform import resize
        from tensorflow.python.framework import ops  # used to be for reset graph - can just do
        import os

        import matplotlib.pyplot as plt
        %matplotlib inline
```

We use a pre-trained 'VGG' (Visual Geometry Group) model from Oxford to compute content and style representations. We will download the weights for the model (unless we have already downloaded them) and use them to initialize a network built in TensorFlow.

(It might be neat to try a different model - e.g., Google's Inception, etc.)

In the paper, the authors change the pooling from max pooling to average pooling. It is somewhat amazing this works and doesn't require re-training the network!

```
In [3]: vggmodelfile = 'imagenet-vgg-verydeep-19.mat'
        if not os.path.isfile(os.path.join('.', vggmodelfile)):
            import six.moves.urllib.request as request
            # use `dl=1` as part of query to force a download
            origin = ('https://www.dropbox.com/s/qy9gikdpuq95w8n/%s?dl=1' % vggmodelfile)
            print('Downloading VGG16 model from: %s' % origin)
            request.urlretrieve(origin, vggmodelfile)
        else:
            print('VGG19 model', vggmodelfile, 'is available.')
```

```
VGG19 model imagenet-vgg-verydeep-19.mat is available.
```

0.4 Image Preparation

We will pull content and style images to use for the demo from an online URL. In principle, you can use anything (although the image manipulaiton functions that prep the 600x800x3 images we need have not been *exhaustively* tested, so if you have an image with too different a shape, it might not work).

```
In [4]: image_dir = './images'
        if os.path.isdir(image_dir):
            print('Image directory is ready...')
        else:
            os.makedirs(image_dir)

        source_image = 'fermilab-scenes.jpg'
        source = os.path.join(image_dir, source_image)

        if os.path.exists(source):
            print('Source image exists...')
        else:
            import six.moves.urllib.request as request
            if source_image == 'fermilab-scenes.jpg':
                origin = ('https://www.dropbox.com/s/g5lnnm83fetngok/fermilab-scenes.jpg?dl=1')
            print('Downloading source image from: %s' % origin)
            request.urlretrieve(origin, source)

        style_image = 'cezanne1.jpg'
        style_image = 'Picasso.jpg'
        style_image = 'StarryNight.jpg'
        style = os.path.join(image_dir, style_image)

        if os.path.exists(style):
            print('Style image exists...')
        else:
            import six.moves.urllib.request as request
            if style_image == 'cezanne1.jpg':
                origin = ('https://www.dropbox.com/s/rv6ijc229b4t2d4/cezanne1.jpg?dl=1')
            elif style_image == 'StarryNight.jpg':
                origin = ('https://www.dropbox.com/s/2hya8bqfoq3f75l/StarryNight.jpg?dl=1')
            print('Downloading style image from: %s' % origin)
            request.urlretrieve(origin, style)
```

```
Image directory is ready...
Source image exists...
Style image exists...
```

Now, let's make an output directory to store trained/translated images. We'll use the filename to build the directory, so if the files you're using have very complex names, you might choose to rename them first.

```
In [5]: source_root = source_image.split('.')[0]
        style_root = style_image.split('.')[0]

        out_dir = './results_' + source_root + '_' + style_root
        if not os.path.exists(out_dir):
            print('Making directory %s' % out_dir)
            os.mkdir(out_dir)
        print('Results directory %s is available.' % out_dir)
```

```
Making directory ./results_fermilab-scenes_StarryNight
Results directory ./results_fermilab-scenes_StarryNight is available.
```

VGG expects 600x800x3 images, and it expects the images to be 'normalized' (we must subtract the RGB means of the training set used to train the VGG model itself (which was ImageNet 2014) - we just need these 'arbitrary' numbers, or we need to retrain our own version of VGG, etc.).

```
In [6]: IMAGE_W = 800    # 1.333 aspect ratio
        IMAGE_H = 600
        # TF tensors are (batch, H, W, D) - here we need the D={RGB}
        VGG_MEAN_VALUES = np.array([123, 117, 104]).reshape((1,1,1,3))
```

Now we'll define some plotting functions for convenience:

```
In [7]: def imshow_clean(img, interpolation=None):
            """
            show an image with no axes (looks nicer for looking at pictures, etc.)
            """
            fig = plt.figure()
            ax = plt.gca()
            ax.axis('off')
            im = ax.imshow(img, interpolation=interpolation)
            return im
```

```
In [8]: def show_img_pair(raw, normed,
                          raw_title='Raw Image', normed_title='Normalized Image', interpolation=
                          raw_type='uint8', normed_type='int8'):
            """
            use this method to display raw and normalized images by default, but can just show p
            we have a base title for each image and additionally attach the dimensions (height a
            """
            show_norm = normed
            if len(normed.shape) == 4:
                show_norm = normed[0]

            fig = plt.figure(figsize=(22,12))
            gs = plt.GridSpec(1, 2)
            ax = plt.subplot(gs[0])
            ax.axis('off')
            im = ax.imshow(raw.astype(raw_type), interpolation=interpolation)
            plt.title(raw_title + ': {} x {}'.format(raw.shape[0],raw.shape[1]))
            ax = plt.subplot(gs[1])
            ax.axis('off')
            im = ax.imshow(show_norm.astype(normed_type), interpolation=interpolation)
            plt.title(normed_title + ': {} x {}'.format(show_norm.shape[0],show_norm.shape[1]))
```

Now, we will define a series of functions that transform roughly arbitrary color images into the shape needed to use with VGG. **Be a bit careful with arbitrary images, you may need to tweak these functions, etc.** For example, right now things are set up to work only with images that are *wider than they are tall...*

```
In [9]: def crop_by_aspectr(img, aspect_ratio=1.3333, tolerance=0.01):
            """
            currently works ONLY FOR IMAGES THAT ARE WIDER THAN THEY ARE TALL
            * if too tall, remove segments from top and bottom
            * if too wide, remove segments from left and right
            """
            if aspect_ratio <= 1 or img.shape[0] == img.shape[1]:
                print('this function is set up for images wider than they are tall')
                cropped_img = img[:, :, :]
                return cropped_img
            img_shape = img.shape
            taller_than_wide = img_shape[0] > img_shape[1]
            if taller_than_wide:
                new_h = int(img_shape[1] / aspect_ratio)
                extra = int((img_shape[0] - new_h) / 2)
                cropped_img = img[extra: new_h + extra, :, :]
            else:
                new_w = int(img_shape[0] * aspect_ratio)
                extra = int((img_shape[1] - new_w) / 2)
                cropped_img = img[:, extra: new_w + extra, :]
            return cropped_img
```

```
In [10]: def resizer(img, imgh=600, imgw=800, scale=255):
             """
             here, the default scale assumes we are rescaling for uint8 (8 bits expected by VGG)
             `resize` an image, the elements are re-scaled into the range $[0, 1]$. Be careful t
             0 to 255.
             """
             img_shape = img.shape
             if len(img_shape) == 2:
                 img_resized = scale * resize(img, (imgh, imgw))
             elif len(img_shape) == 3:
                 img_resized = scale * resize(img, (imgh, imgw, 3))
             else:
                 print('invalid image dimensions, image shape =', img_shape)

             return img_resized
```

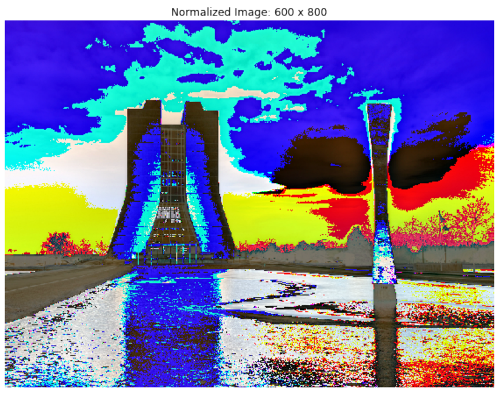
In TensorFlow, *image* tensor shapes should be : `Batch-size x H x W x NChannels` (where `NChannels == 3` at input for RGB images, and equals the number of filter maps as we progress through the network). Somewhat confusingly, *convolutional kernel* tensor shapes are different, `K_H x K_W x NF_in x NF_out`, where `K_H x K_W` gives the kernel's height and width, while `NF_in x NF_out` gives the number of in and output filters/channels. Pooling kernel shapes are `Batch-size x H x W x NChannels` (if first dimension is larger than 1, I believe we are combining images).

We will be using a pre-trained VGG16 model here, so we need to resize our images appropriately. In order to use the VGG model, we will also need to subtract the mean of the training dataset for that model.

```
In [11]: def tensorfy_and_normalize(img, mean_values=VGG_MEAN_VALUES):
             """
             take `H x W x NChannels` images and prepend the 'batch size' dimension;
             subtract a set of mean values (computed from VGG training set)
             """
             img_shape = img.shape
             assert(len(img_shape) == 3)    # we need rgb images to work with VGG anyway
             img_resized = img[np.newaxis, :, :, :]
             img_normalized = img_resized - mean_values
             return img_normalized
```

Now, let's load and prepare our content ('source') image, and save a copy of the re-sized image for later comparison:

```
In [12]: raw_source = plt.imread(source)
         cropped_source = crop_by_aspectr(raw_source)
         resized_source = resizer(cropped_source)

         path = out_dir + '/' + source_root + '_resized.png'
         scipy.misc.imsave(path, resized_source.astype('uint8'))

         normed_source = tensorfy_and_normalize(resized_source)
         show_img_pair(raw_source, normed_source)
```

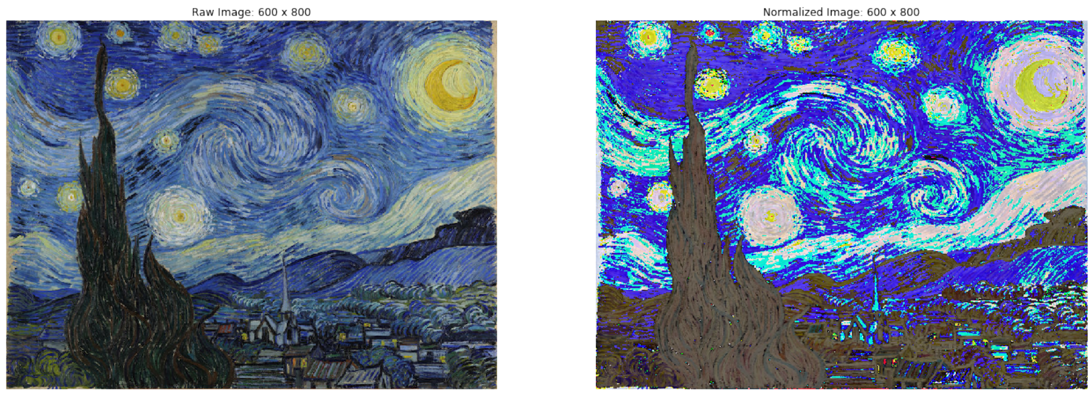


As a sanity check, let's be sure that if we add the `VGG_MEAN_VALUES` back, we get the image we expect:

```
In [13]: imshow_clean((normed_source + VGG_MEAN_VALUES)[0].astype('uint8'))

Out[13]: <matplotlib.image.AxesImage at 0x11f4b8160>
```

Now, let's do the same for the style image:

```
In [14]: raw_style = plt.imread(style)
         cropped_style = crop_by_aspectr(raw_style)
         resized_style = resizer(cropped_style)

         path = out_dir + '/' + style_root + '_resized.png'
         scipy.misc.imsave(path, resized_style.astype('uint8'))

         normed_style = tensorfy_and_normalize(resized_style)
         show_img_pair(raw_style, normed_style)
```



```
In [15]: imshow_clean((normed_style + VGG_MEAN_VALUES)[0].astype('uint8'))
```

```
Out[15]: <matplotlib.image.AxesImage at 0x12e763c18>
```

Let's rename the images for convenience / simplicity...

```
In [16]: source_img = normed_source
         style_img = normed_style
```

0.5 Network Definition

Now, let's define a utility function for building layers and chaining them together:

```
In [17]: def build_net(ntype, nin, rwb=None):
             """
             utility function for building network layers
             * ntype == network layer type (string, 'conv' or 'pool')
             * nin == network input layer (the object itself)
             * rwb == weights and biases
             """
             # `strides` basically describe how far the kernel steps in each tensor dimension
             # `ksize` describes the size of the pooling region
             # If we use `SAME` padding, the filter is allowed to go off the edge by half the fi
             # so we would expect an unchanged image size with stride=1 and a (1/2)x(1/2) for st
             if ntype == 'conv':
                 return tf.nn.relu(tf.nn.conv2d(nin, rwb[0], strides=[1, 1, 1, 1], padding='SAME
             elif ntype == 'pool':
                 return tf.nn.avg_pool(nin, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='S
```

0.5.1 Detour on kernels, strides, and padding

`ksize` tells us about kernel size (4-tensor), and strides tells us how far we move the the kernel with each "step" when mapping it across the image. Padding is explained in the TF documentation for convolution.

Let's take a detour and demonstrate it with the pooling operation:

```
In [18]: # super-simple "graph" to demonstrate pooling
         # X shape of None x 6 x 6 x 1 -> any number of 6x6 tensors that are "1-deep" (e.g., gra
         X = tf.placeholder(tf.float32, shape=(None, 6, 6, 1), name='X')
         Y_same = tf.nn.avg_pool(X, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
         Y_valid = tf.nn.avg_pool(X, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')
```

TensorFlow separates graph definition and computation into two steps - we must execute operations within a *Session*.

```
In [19]: sess = tf.Session()
         sess.run(tf.global_variables_initializer())
```

We need to pass a tensor with actual data into our "graph" to see how it works.

```
In [20]: X_vals = np.arange(6*6).reshape(1, 6, 6, 1)
         # print(X_vals)
```

To get a value we must run the computation inside the `Session` we created earlier. `feed_dict` tells TF what values to plug into the placeholder we used to start the graph:

```
In [21]: Y_same_vals = sess.run(Y_same, feed_dict={X: X_vals})
         print(Y_same_vals.shape)
         print(Y_same_vals)
```

```
(1, 3, 3, 1)
[[[[  3.5]
   [  5.5]
   [  7.5]]

  [[ 15.5]
   [ 17.5]
   [ 19.5]]

  [[ 27.5]
   [ 29.5]
   [ 31.5]]]]
```

```
In [22]: Y_valid_vals = sess.run(Y_valid, feed_dict={X: X_vals})
         print(Y_valid_vals.shape)
         print(Y_valid_vals)
```

```
(1, 3, 3, 1)
[[[[  3.5]
   [  5.5]
   [  7.5]]

  [[ 15.5]
   [ 17.5]
   [ 19.5]]

  [[ 27.5]
   [ 29.5]
   [ 31.5]]]]
```

In this case, the tensors came out the same because the kernel and stride sizes conspired with the image size to keep the filter mapping "inside" the tensor. What does this mapping look like (with a lazy figure)?

Let's try again with a different stride to see if we can get a better sense for the difference.

First, let's 'reset' the graph, just to keep things simple:

```
In [23]: tf.reset_default_graph()
```

Let's look at a different stride to see if the pooling ops behave differently:

```
In [24]: X = tf.placeholder(tf.float32, shape=(None, 6, 6, 1), name='X')
         Y_same = tf.nn.avg_pool(X, ksize=[1, 2, 2, 1], strides=[1, 1, 1, 1], padding='SAME')
         Y_valid = tf.nn.avg_pool(X, ksize=[1, 2, 2, 1], strides=[1, 1, 1, 1], padding='VALID')
```

```
In [25]: # we are using the default graph, so we have to re-initialize
         sess = tf.Session()
         sess.run(tf.global_variables_initializer())
```

```
In [26]: Y_same_vals = sess.run(Y_same, feed_dict={X: X_vals})
         print(Y_same_vals.shape)
         print(Y_same_vals)
```

```
(1, 6, 6, 1)
[[[[  3.5]
   [  4.5]
   [  5.5]
   [  6.5]
   [  7.5]
   [  8. ]]

  [[  9.5]
   [ 10.5]
   [ 11.5]
   [ 12.5]
   [ 13.5]
   [ 14. ]]

  [[ 15.5]
   [ 16.5]
   [ 17.5]
   [ 18.5]
   [ 19.5]
   [ 20. ]]

  [[ 21.5]
   [ 22.5]
   [ 23.5]
   [ 24.5]
   [ 25.5]
   [ 26. ]]

  [[ 27.5]
   [ 28.5]
   [ 29.5]
   [ 30.5]
   [ 31.5]
   [ 32. ]]

  [[ 30.5]
   [ 31.5]
   [ 32.5]
   [ 33.5]
   [ 34.5]
   [ 35. ]]]]
```

So here, `SAME` goes "off the edge" on the right to keep the overall tensor the same size...

```
In [27]: Y_valid_vals = sess.run(Y_valid, feed_dict={X: X_vals})
         print(Y_valid_vals.shape)
         print(Y_valid_vals)
```

```
(1, 5, 5, 1)
[[[[  3.5]
   [  4.5]
   [  5.5]
   [  6.5]
   [  7.5]]

  [[  9.5]
   [ 10.5]
   [ 11.5]
   [ 12.5]
   [ 13.5]]

  [[ 15.5]
   [ 16.5]
   [ 17.5]
   [ 18.5]
   [ 19.5]]

  [[ 21.5]
   [ 22.5]
   [ 23.5]
   [ 24.5]
   [ 25.5]]

  [[ 27.5]
   [ 28.5]
   [ 29.5]
   [ 30.5]
   [ 31.5]]]]
```

... but `VALID` requires the kernel to stay entirely in-bounds. This means the output tensor must shrink.

Ok, with that demo over, let's reset the graph. Before we do, we can inspect it:

```
In [28]: [op.name for op in tf.get_default_graph().get_operations()]

Out[28]: ['X', 'AvgPool', 'AvgPool_1', 'init']

In [29]: tf.reset_default_graph()

In [30]: [op.name for op in tf.get_default_graph().get_operations()]

Out[30]: []
```

0.5.2 VGG

Now, let's define a funciton we can use to extract the model parameters from the pre-trained VGG file we downloaded:

```
In [31]: def get_weight_bias(vgg_layers, i):
             """
             utility function to get weights and bias values from the layers of our pre-built vg
             we set the weights as `constant` to indicate they should not be trainable.
             """
             weights = vgg_layers[i][0][0][0][0][0]
             weights = tf.constant(weights)
             bias = vgg_layers[i][0][0][0][0][1]
             bias = tf.constant(np.reshape(bias, (bias.size)))
             return weights, bias
```

And, now for the network:

```
In [32]: def build_vgg19(path):
             """
             build the VGG DNN - note that the **input** is a `tf.Variable` - this is what we wi
             """
             net = {}
             vgg_rawnet = scipy.io.loadmat(path)
             vgg_layers = vgg_rawnet['layers'][0]
             net['input'] = tf.Variable(np.zeros((1, IMAGE_H, IMAGE_W, 3)).astype('float32'))
             net['conv1_1'] = build_net('conv', net['input'], get_weight_bias(vgg_layers, 0))
             net['conv1_2'] = build_net('conv', net['conv1_1'], get_weight_bias(vgg_layers, 2))
             net['pool1'] = build_net('pool', net['conv1_2'])
             net['conv2_1'] = build_net('conv', net['pool1'], get_weight_bias(vgg_layers, 5))
             net['conv2_2'] = build_net('conv', net['conv2_1'], get_weight_bias(vgg_layers, 7))
             net['pool2'] = build_net('pool', net['conv2_2'])
             net['conv3_1'] = build_net('conv', net['pool2'], get_weight_bias(vgg_layers, 10))
             net['conv3_2'] = build_net('conv', net['conv3_1'], get_weight_bias(vgg_layers, 12))
             net['conv3_3'] = build_net('conv', net['conv3_2'], get_weight_bias(vgg_layers, 14))
             net['conv3_4'] = build_net('conv', net['conv3_3'], get_weight_bias(vgg_layers, 16))
             net['pool3'] = build_net('pool', net['conv3_4'])
             net['conv4_1'] = build_net('conv', net['pool3'], get_weight_bias(vgg_layers, 19))
             net['conv4_2'] = build_net('conv', net['conv4_1'], get_weight_bias(vgg_layers, 21))
             net['conv4_3'] = build_net('conv', net['conv4_2'], get_weight_bias(vgg_layers, 23))
             net['conv4_4'] = build_net('conv', net['conv4_3'], get_weight_bias(vgg_layers, 25))
             net['pool4'] = build_net('pool', net['conv4_4'])
             net['conv5_1'] = build_net('conv', net['pool4'], get_weight_bias(vgg_layers, 28))
             net['conv5_2'] = build_net('conv', net['conv5_1'], get_weight_bias(vgg_layers, 30))
             net['conv5_3'] = build_net('conv', net['conv5_2'], get_weight_bias(vgg_layers, 32))
             net['conv5_4'] = build_net('conv', net['conv5_3'], get_weight_bias(vgg_layers, 34))
             net['pool5'] = build_net('pool', net['conv5_4'])
             return net
```

Now we may compute loss functions.

In order to visualize the encoded image information, we use gradient descent on the pixel values in a white-noise image to find a new image that matches the in-network responses of the original image. So, for the image content we define the error as the squared difference between the feature representations of the 'source' and 'target' images:

```
In [33]: def build_content_loss(p, x):
             """
             here, p has shape (batch, h, w, n-channels)

             * p is an evaluated TF tensor or a ndarray
             * x is a TF placeholder
             """
             M = p.shape[1] * p.shape[2]    # h x w
             N = p.shape[3]                 # n-channels
```

```
        loss = (1. / (2 * N**0.5 * M**0.5)) * tf.reduce_sum(tf.pow((x - p), 2))
        return loss
```

Note: the paper defines the content loss slightly differently (no 'normalizing' factor that is a function of image size and depth). However, we include one here to scale down the absolute value of the loss and keep it closer to the style loss.

In order to define the style loss, we need to first define a matrix $F^l \in \mathcal{R}^{N_l \times M_l}$, where N_l counts the number of filter maps and M_l is the height times the width of the filter map (basically, the 1-d re-arrangment of the 2-d information). F^l_{ij} is the activation of the ith filter at position j in layer l.

Next we will use F^l to define computations for the Gram matrix, $G^l \in \mathcal{R}^{N_l \times N_l}$, with N_l equal to the number of filter maps in layer l, and G^l_{ij} equal to the inner product between the vectorized feature maps i and j in layer l,

$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk} \tag{1}$$

If the vectors are centered random variables, the Gram matrix is roughly proportional to the covariance matrix, with additional scaling proporitonal to the number of elements in the vectors. Here, we are computing the Gram using the activations of a layer with themselves - so **we are looking at the covariance between activations in different filter maps.**

We define two versions of the matrix (computing function). First, we define a "TensorFlow version," that expects TF variables that constructs a graph element. Second, we define a "NumPy version" that expects NumPy arrays and computes a fixed value (based on the initial network weights).

```
In [34]: def gram_matrix(x, area, depth):
             """
             input/evaluate with TF tensors

             here we take a tensor of shape (1, h, w, n) and reshape it to (h * w, n),
             and then compute the inner product of the new tensor with itself
             """
             x1 = tf.reshape(x, (area, depth))
             g = tf.matmul(tf.transpose(x1), x1)
             return g
```

```
In [35]: def gram_matrix_val(x, area, depth):
             """
             numpy version of the `gram_matrix` method (may be evaluated outside a TF `Session`)

             here we take a ndarray of shape (1, h, w, n) and reshape it to (h * w, n),
             and then compute the inner product of the new ndarray with itself
             """
             x1 = x.reshape(area, depth)
             g = np.dot(x1.T, x1)
             return g
```

Let's just look at that with a few examples:

```
In [36]: x_test = np.array([1] * 18).reshape(1, 3, 3, 2)
         print(x_test)
```

```
[[[[1 1]
   [1 1]
   [1 1]]

  [[1 1]
   [1 1]
   [1 1]]

  [[1 1]
   [1 1]
   [1 1]]]]
```

```
In [37]: x_test_gram = gram_matrix_val(x_test, 3*3, 2)
         print(x_test_gram)
```

```
[[9 9]
 [9 9]]
```

```
In [38]: x_test = np.array([1, 2] * 9).reshape(1, 3, 3, 2)
         print(x_test)
```

```
[[[[1 2]
   [1 2]
   [1 2]]

  [[1 2]
   [1 2]
   [1 2]]

  [[1 2]
   [1 2]
   [1 2]]]]
```

```
In [39]: x_test_gram = gram_matrix_val(x_test, 3*3, 2)
         print(x_test_gram)
```

```
[[ 9 18]
 [18 36]]
```

```
In [40]: x_test = np.array([1, -1] * 9).reshape(1, 3, 3, 2)
         print(x_test)
```

```
[[[[ 1 -1]
   [ 1 -1]
   [ 1 -1]]

  [[ 1 -1]
   [ 1 -1]
   [ 1 -1]]

  [[ 1 -1]
   [ 1 -1]
   [ 1 -1]]]]
```

```
In [41]: x_test_gram = gram_matrix_val(x_test, 3*3, 2)
         print(x_test_gram)
```

```
[[ 9 -9]
 [-9  9]]
```

```
In [42]: x_test = np.random.normal(loc=0.0, scale=1.0, size=(1, 3, 3, 2))
         print(x_test)
```

```
[[[[ 0.01112503  0.83821159]
   [ 0.98208945 -1.85440403]
   [ 1.13437189 -0.4133171 ]]

  [[ 1.22646323  0.67066412]
   [-0.25629429 -1.76345124]
   [ 1.45313605  2.02801639]]

  [[-0.12623371 -1.15925216]
   [ 1.41841123  0.65704513]
   [ 0.75632214  0.19312161]]]]
```

```
In [43]: x_test_gram = gram_matrix_val(x_test, 3*3, 2)
         print(x_test_gram)
```

```
[[  8.53277479   3.16512933]
 [  3.16512933  13.79751495]]
```

```
In [44]: x_test = np.random.normal(loc=0.0, scale=1.0, size=(1, 10, 10, 2))
         x_test_gram = gram_matrix_val(x_test, 10 * 10, 2)
         print(x_test_gram)
```

```
[[ 68.96467605  -5.14521512]
 [ -5.14521512  92.4676978 ]]
```

Okay, now we may compute the style loss:

```
In [45]: def build_style_loss(a, x):
             """
             here, a has shape (batch, h, w, n-channels)

             * a is an evaluated TF tensor or a ndarray
             * x is a TF placeholder
             """
             M = a.shape[1] * a.shape[2]    # h x w
             N = a.shape[3]                 # n-channels
             A = gram_matrix_val(a, M, N)
             G = gram_matrix(x, M, N)
             loss = (1. / (4 * N**2 * M**2)) * tf.reduce_sum(tf.pow((G - A), 2))
             return loss
```

```
In [47]: tf.reduce_sum(x_test_gram).eval(session=tf.Session())  # "throwaway" session
```

```
Out[47]: 151.14194360370436
```

Now, with helper functions set, let's perform the style transfer.

```
In [48]: tf.reset_default_graph()
```

```
In [49]: net = build_vgg19(vggmodelfile)
```

```
In [50]: [op.name for op in tf.get_default_graph().get_operations()]
```

```
Out[50]: ['Variable/initial_value',
          'Variable',
          'Variable/Assign',
          'Variable/read',
          'Const',
          'Const_1',
          'Conv2D',
          'add',
          'Relu',
          'Const_2',
          'Const_3',
          'Conv2D_1',
          'add_1',
          'Relu_1',
          'AvgPool',
          'Const_4',
          'Const_5',
          'Conv2D_2',
          'add_2',
          'Relu_2',
          'Const_6',
'Const_7',
'Conv2D_3',
'add_3',
'Relu_3',
'AvgPool_1',
'Const_8',
'Const_9',
'Conv2D_4',
'add_4',
'Relu_4',
'Const_10',
'Const_11',
'Conv2D_5',
'add_5',
'Relu_5',
'Const_12',
'Const_13',
'Conv2D_6',
'add_6',
'Relu_6',
'Const_14',
'Const_15',
'Conv2D_7',
'add_7',
'Relu_7',
'AvgPool_2',
'Const_16',
'Const_17',
'Conv2D_8',
'add_8',
'Relu_8',
'Const_18',
'Const_19',
'Conv2D_9',
'add_9',
'Relu_9',
'Const_20',
'Const_21',
'Conv2D_10',
'add_10',
'Relu_10',
'Const_22',
'Const_23',
'Conv2D_11',
'add_11',
'Relu_11',
'AvgPool_3',
'Const_24',
 'Const_25',
 'Conv2D_12',
 'add_12',
 'Relu_12',
 'Const_26',
 'Const_27',
 'Conv2D_13',
 'add_13',
 'Relu_13',
 'Const_28',
 'Const_29',
 'Conv2D_14',
 'add_14',
 'Relu_14',
 'Const_30',
 'Const_31',
 'Conv2D_15',
 'add_15',
 'Relu_15',
 'AvgPool_4']
```

Let's make a new session to go with the reset and rebuilt graph:

```
In [51]: sess = tf.Session()
         sess.run(tf.global_variables_initializer())
```

We define a white noise image to use as a starting point (we will actually cheat and 'mix' some of the starting image into this one to help the network out):

```
In [52]: noise_img = np.random.uniform(-20, 20, (1, IMAGE_H, IMAGE_W, 3)).astype('float32')
```

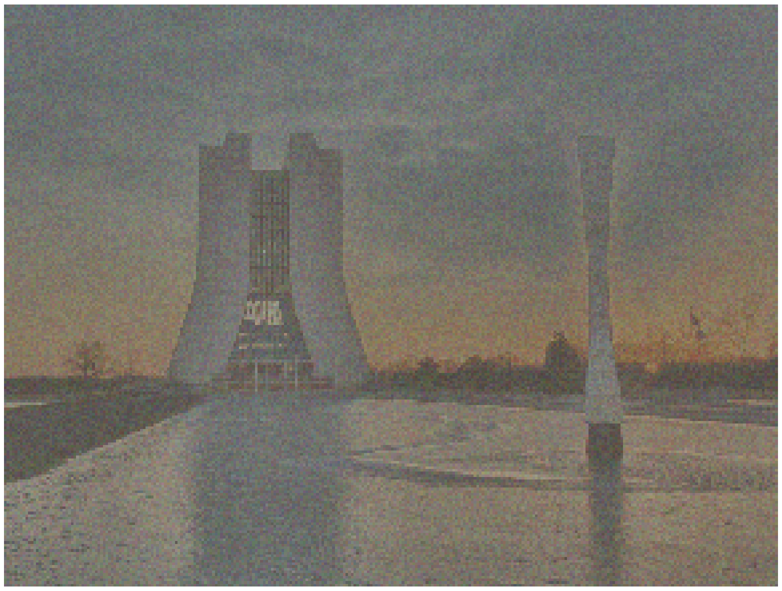
We use a high level in the network for content and multuple levels for style. We could switch to a different (probably higher) level for the content if we wished. We can also edit the style list to get different results... perhaps experiment with these?

```
In [53]: # choice in the paper (paper noise ratio and style strengths not really known)
         CONTENT_LAYERS = [('conv4_2', 1.)]
         STYLE_LAYERS = [('conv1_1', 1.), ('conv2_1', 1.), ('conv3_1', 1.), ('conv4_1', 1.), ('c
         STYLE_STRENGTH = 500
         INI_NOISE_RATIO = 0.7

         # choice in the paper, but with lower style strength
         # CONTENT_LAYERS = [('conv4_2', 1.)]
         # STYLE_LAYERS = [('conv1_1', 1.), ('conv2_1', 1.), ('conv3_1', 1.), ('conv4_1', 1.), (
         # STYLE_STRENGTH = 50
         # INI_NOISE_RATIO = 0.7

         # experiment 1
         # CONTENT_LAYERS = [('conv5_1', 1.)]
```

```
    # STYLE_LAYERS = [('conv1_1', 1.), ('conv2_1', 1.), ('conv3_1', 1.)]
    # STYLE_STRENGTH = 500
    # INI_NOISE_RATIO = 0.7

    # experiment 2
    # CONTENT_LAYERS = [('conv5_1', 1.)]
    # STYLE_LAYERS = [('conv1_1', 0.5), ('conv2_1', 1.), ('conv3_1', 1.5), ('conv4_1', 2.)]
    # STYLE_STRENGTH = 500
    # INI_NOISE_RATIO = 0.7

    # experiment 3 - deep content "only"
    # CONTENT_LAYERS = [('conv5_1', 1.)]
    # STYLE_LAYERS = [('conv1_1', 1.)]
    # STYLE_STRENGTH = 0
    # INI_NOISE_RATIO = 0.7

    # experiment 4 - deep style "only"
    # CONTENT_LAYERS = [('conv4_2', 0.)]
    # STYLE_LAYERS = [('conv1_1', 1.), ('conv2_1', 1.), ('conv3_1', 1.), ('conv4_1', 1.), (
    # STYLE_STRENGTH = 500
    # INI_NOISE_RATIO = 0.7

    # experiment 5 - deep style "only"
    # CONTENT_LAYERS = [('conv4_2', 0.)]
    # STYLE_LAYERS = [('conv1_1', 1.), ('conv2_1', 1.), ('conv3_1', 1.), ('conv4_1', 1.), (
    # STYLE_STRENGTH = 500
    # INI_NOISE_RATIO = 0.99

    # experiment 6 - paper-like, but higher noise & style strength
    # CONTENT_LAYERS = [('conv4_2', 1.)]
    # STYLE_LAYERS = [('conv1_1', 1.), ('conv2_1', 1.), ('conv3_1', 1.), ('conv4_1', 1.), (
    # STYLE_STRENGTH = 1000
    # INI_NOISE_RATIO = 0.85


    initial_img = INI_NOISE_RATIO * noise_img + (1. - INI_NOISE_RATIO) * source_img
    print('noise image : ', np.min(noise_img), np.max(noise_img))
    print('source image: ', np.min(source_img), np.max(source_img))
    print('src+vgg img : ', np.min(source_img + VGG_MEAN_VALUES), np.max(source_img + VGG_M
    print('combo image : ', np.min(initial_img[0]), np.max(initial_img[0]))
    print('cmb+vgg img : ', np.min((initial_img + VGG_MEAN_VALUES)[0]), np.max((initial_img
    imshow_clean((initial_img + VGG_MEAN_VALUES)[0].astype('uint8'))

    path = out_dir + '/initial.png'
    scipy.misc.imsave(path, np.clip((initial_img + VGG_MEAN_VALUES)[0], 0, 255).astype('uin
```

```
noise image :  -20.0 20.0
source image:  -123.0 125.941768229
```

```
src+vgg img :  0.0 246.032977604
combo image :  -50.8825124741 49.8108793915
cmb+vgg img :  58.8320657657 170.812313936
```



Now, we build the cost function for the source image's contributon to the total cost. We will map our loss function over the set of layers we want to use (just one, typically, for the content). We want to build a TF 'function' that operates on a given input image, but we want to go ahead and compute the values for the real target source image. Therefore, we will `run()` the network *on* the source to actually define the function.

First, let's just look at the shape of one of our loss function components to make sure it is a scalar:

```
In [54]: build_content_loss(sess.run(net[CONTENT_LAYERS[0][0]]), net[CONTENT_LAYERS[0][0]]).eval
```

```
Out[54]: ()
```

Now, we will (temporarily) assign the *source image* as the input to the network so we can build a graph that computes differences between layer activations for that image and the image we are manipulating; so - we fix one set of numbers in the graph by `run()`ing the network to evaluate it, and we let the other be an empty graph element that here is a `tf.Variable`. This means we are actually training the input image pixel values to match the source image:

```
In [55]: print(CONTENT_LAYERS)
         sess.run([net['input'].assign(source_img)])
         cost_source = sum(
             map(lambda l,: l[1] * build_content_loss(sess.run(net[l[0]]), net[l[0]]), CONTENT_L
         )
```

```
[('conv4_2', 1.0)]
```

Now, we (temporarily again) switch the input to be the *style image* so we can compute the covariances between the filter maps in the network corresponding to that image. The style component of the loss function will penalize input images that create activations that differ from these values and our loss function will ultimately use graident descent to push the input image to create more similar activations:

```
In [56]: print(STYLE_LAYERS)
         sess.run([net['input'].assign(style_img)])
         cost_style = sum(
             map(lambda l,: l[1] * build_style_loss(sess.run(net[l[0]]), net[l[0]]), STYLE_LAYER
         )
```

```
[('conv1_1', 1.0), ('conv2_1', 1.0), ('conv3_1', 1.0), ('conv4_1', 1.0), ('conv5_1', 1.0)]
```

Our cost function is just a linear sum of these source and style components. We will use the "Adam" Optimizer by Kingma and Ba:

```
In [57]: cost_total = cost_source + STYLE_STRENGTH * cost_style
         optimizer = tf.train.AdamOptimizer(learning_rate=2.0)
         train = optimizer.minimize(cost_total)
```

Now, we initialize the network again to clear values and set the (*variable*) input to our initial image:

```
In [58]: sess.run(tf.global_variables_initializer())
         sess.run(net['input'].assign(initial_img))
```

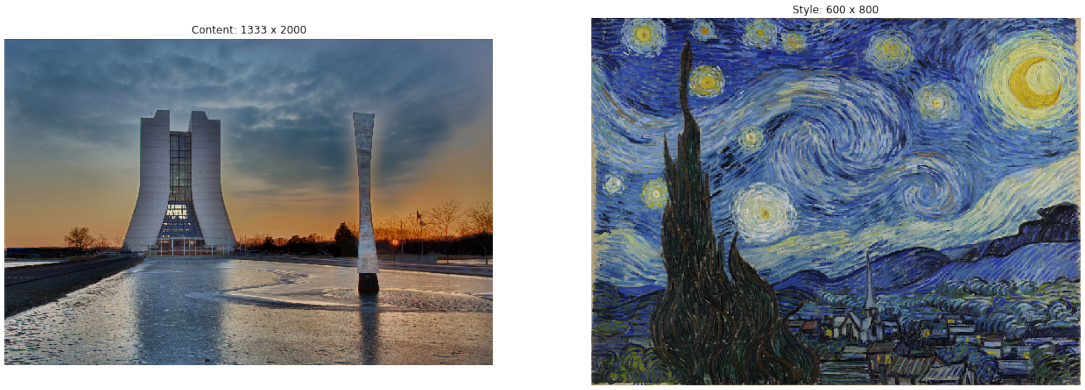
```
Out[58]: array([[[[-18.40870094,  -6.01660967,   3.13375258],
                  [ -6.34923744,   2.61424136,   3.45290422],
                  [ -9.27506161,  -5.95323563,  23.1446209 ],
                  ...,
                  [-21.00718307,   1.84606016,  18.70490456],
                  [ -5.32743168,   0.12734549,   7.1924448 ],
                  [ -1.74223328, -10.5707922 ,  15.40859795]],

                 [[ -6.67753983,   3.62718081,  20.14824486],
                  [-28.74234772,  -5.2949729 ,   1.33851838],
                  [-16.55991745, -18.71420479,   5.90929031],
                  ...,
                  [ -1.03649008,   2.34314656,   1.84428227],
                  [  0.1724063 ,   1.24243581,  23.09021378],
                  [-20.50514603,   8.05275345,  23.46661949]],

                 [[-18.8158474 ,   3.96411562,  -3.18941259],
                  [-21.72597122,  -9.12056637,   6.02023983],
                  [-15.10590458,  -3.862638  ,   6.02378035],
                  ...,
                  [-10.93319893,  -5.15579462,   5.41856241],
                  [-10.12177849,  10.88141918,  -2.3950398 ],
                  [  1.89030051,   1.20697165,   8.86113262]],

                 ...,
                 [[  1.32304192,   0.73107201,  21.62789536],
                  [ 16.42001343,  -2.60263038,  20.29132271],
                  [ 16.22038269,  15.54796219,  -3.25903082],
                  ...,
                  [  6.91192293,   0.56925505,  -2.29145575],
                  [  9.73430443,   3.83390379,   9.37287045],
                  [ -1.43473339,  11.44698811,  -8.64361286]],

                 [[ -6.88900375,  12.26208496,   4.63737154],
                  [ -2.76293349, -15.18908024,  12.60201168],
                  [-12.94264221, -13.64317608,  -0.13379064],
                  ...,
                  [ -6.94406271, -16.67651558, -17.72475243],
                  [-13.46937561, -15.74014854,   2.76372004],
                  [ 12.81842232,  -2.18550634,   6.08937359]],

                 [[ -1.36749077,   4.04563522,   5.96214199],
                  [-10.97194672,  15.99391747,  15.22669888],
                  [  0.65895414,   7.23044252,  17.97756195],
                  ...,
                  [-15.94617748,  -2.71174836, -12.8913784 ],
                  [  7.08524084, -13.33631897,  -9.91191673],
                  [-10.27324581,   3.19253397, -18.43449783]]]], dtype=float32)
```

```
In [59]: NITERATIONS = 3000   # long runs are best, but take time...
         NITERATIONS = 500    # 500 iters seem to be good enough to get a "decent" image
         NITERATIONS = 100
         NITERATIONS = 1      # demo that the code runs...


         PRINT_IMG_FREQ = 100
         PRINT_IMG_FREQ = 25
         PRINT_IMG_FREQ = 1

         START = 0
         STOP = START + NITERATIONS

         for i in range(START, STOP):
             sess.run(train)
             result_image = sess.run(net['input'])
             print(i, sess.run(cost_total))
```

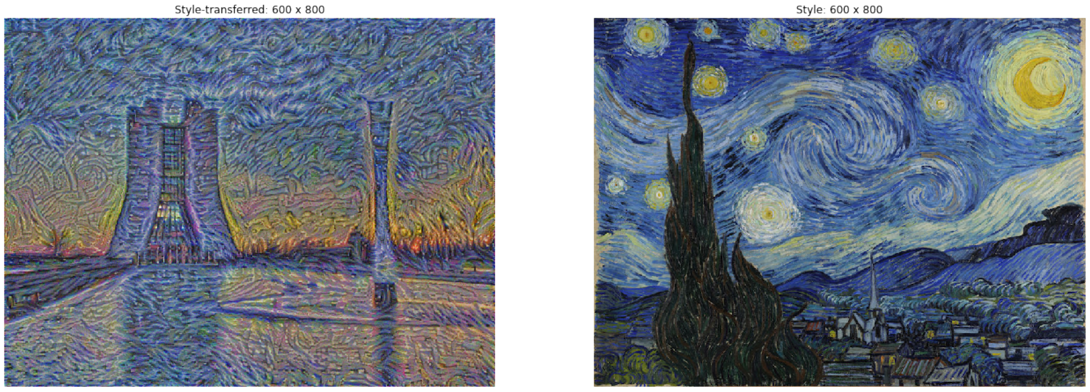
```
            if (i + 1) % PRINT_IMG_FREQ == 0:
                path = out_dir + '/styled_%04d.png' % i
                scipy.misc.imsave(path, np.clip((result_image + VGG_MEAN_VALUES)[0], 0, 255).as
```

```
0 2.9441e+11
```

```
In [62]: show_img_pair(raw=raw_source, normed=raw_style,
                       raw_title='Content', normed_title='Style', interpolation=None,
                       raw_type='uint8', normed_type='uint8')
```



```
In [63]: show_img_pair(raw=np.clip((result_image + VGG_MEAN_VALUES)[0], 0, 255), normed=raw_styl
                       raw_title='Style-transferred', normed_title='Style', interpolation=None,
                       raw_type='uint8', normed_type='uint8')
```



This is only after 100 iterations - the more you run, the better it gets! (Up to a point, of course - tuning hyperparameters is important also.)

```
In [64]: fig = plt.figure(figsize=(22,38))
         gs = plt.GridSpec(5, 2)

         def make_plot(img_path, title_text, grid_value):
             img = plt.imread(img_path)
             ax = plt.subplot(grid_value)
             ax.axis('off')
             im = ax.imshow(img, interpolation=None)
             plt.title(title_text)

         make_plot('./example_results/finals/apaper_0499.png', 'Paper settings, 500 iters', gs[0
         make_plot('./example_results/finals/experiment1_0499.png', 'Level-5 content; 1-3 style,
         make_plot('./example_results/finals/experiment2_0499.png', 'Level-5 content; 1-4 style
         make_plot('./example_results/finals/experiment3_0399.png', 'Level-5 content; no style,
         make_plot('./example_results/finals/experiment4_0499.png', 'No content; 1-5 style, 500
         make_plot('./example_results/finals/experiment5_0499.png', 'No content; 1-5 style; pure
         make_plot('./example_results/finals/experiment6_0499.png', 'Level-4 content; 1-5 style;
         make_plot('./example_results/finals/fermi_cezanne_papersettings_0499.png', 'Paper setti
         make_plot('./example_results/finals/fermi_monet_papersettings_0149.png', 'Paper setting

IOPub data rate exceeded.
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_data_rate_limit`.


In [ ]:
```