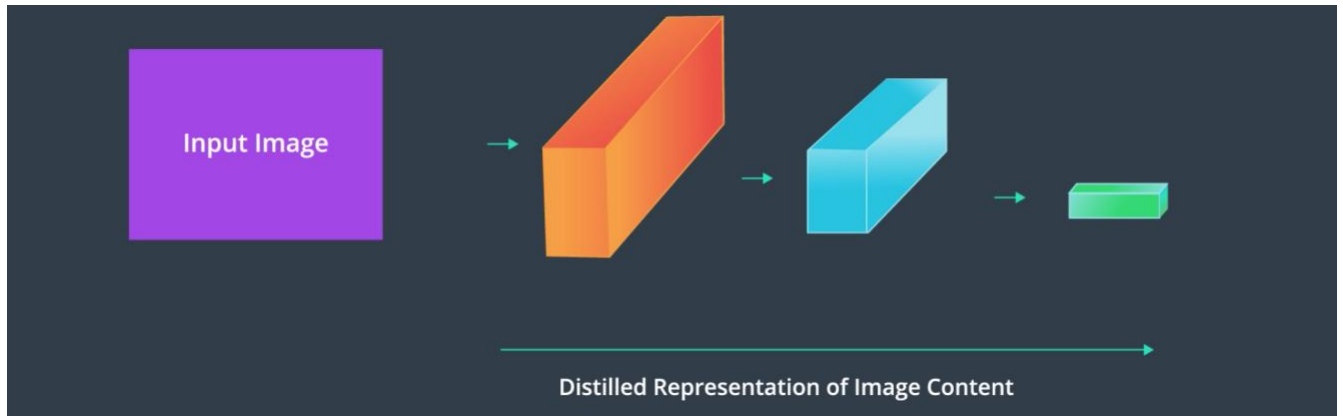
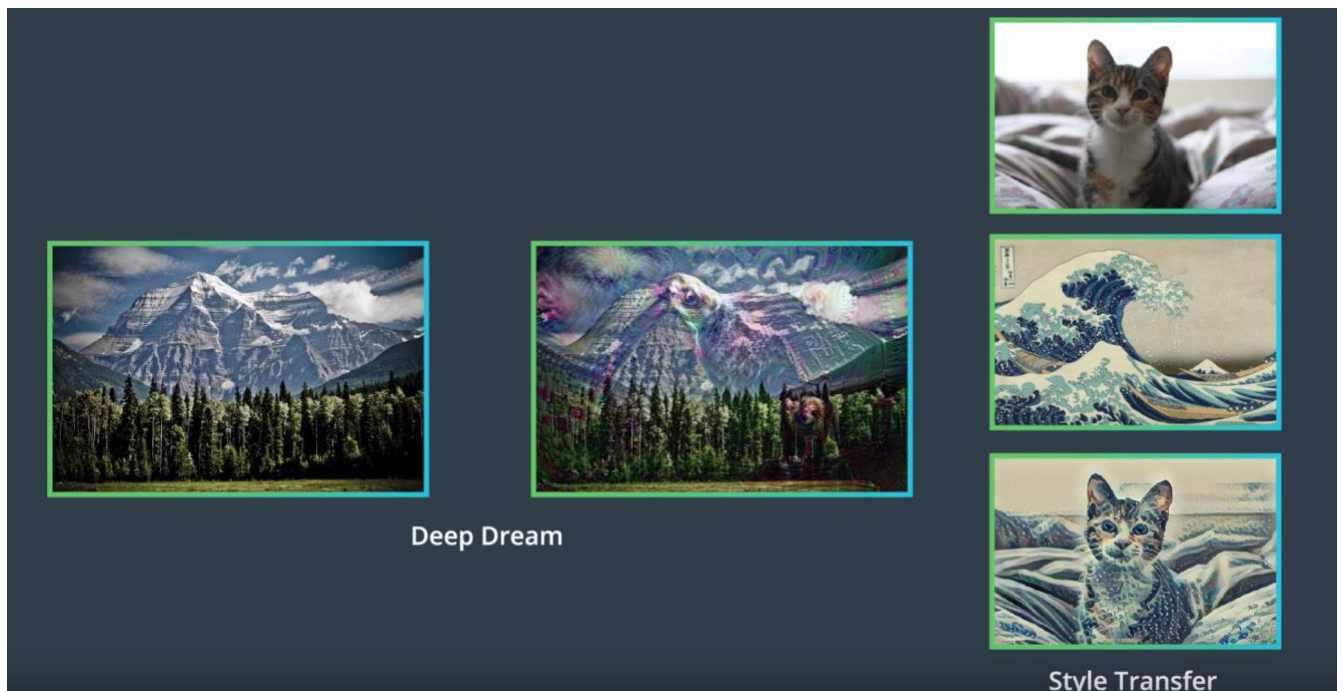


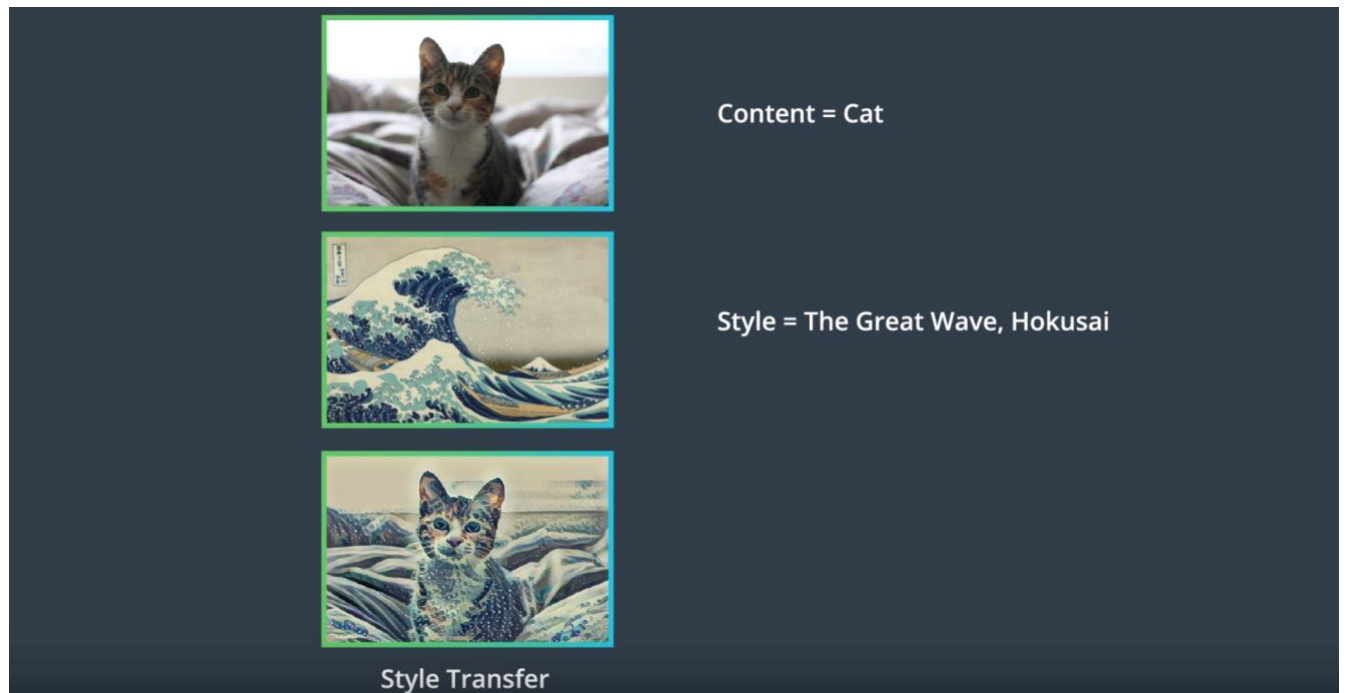
## Lesson 6: Style Transfer



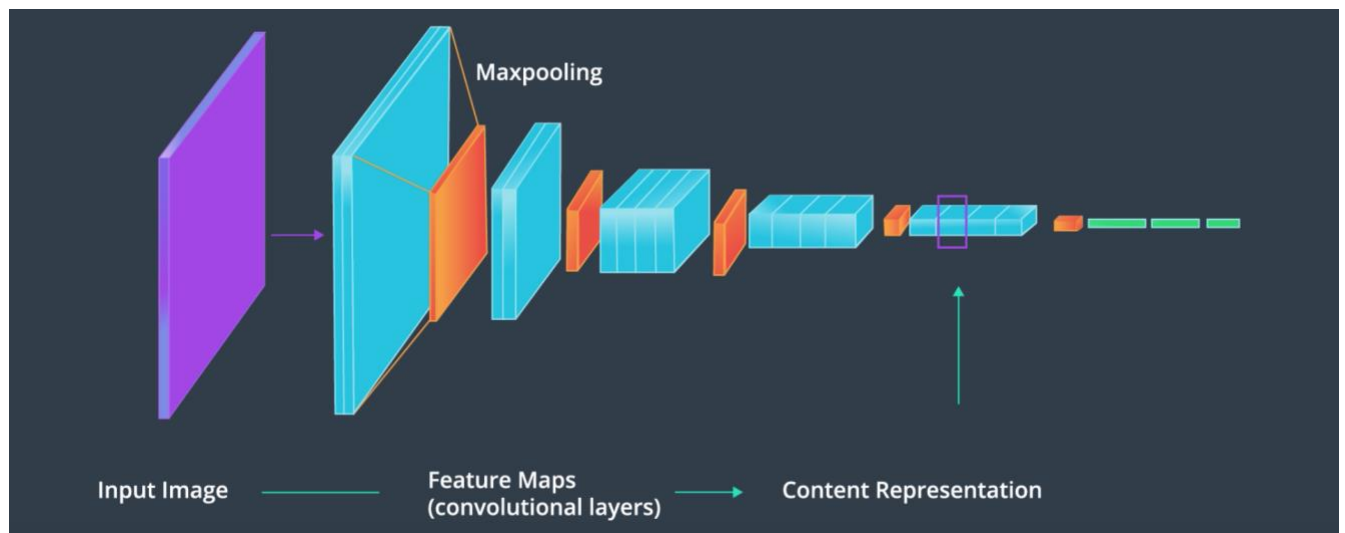
125. You've seen that CNN's or some of the most powerful networks for image classification and analysis. CNN's process visual information in a feed forward manner, passing an input image through a collection of image filters which extract certain features from the input image. It turns out that these feature level representations are not only useful for classification, but for image construction as well.



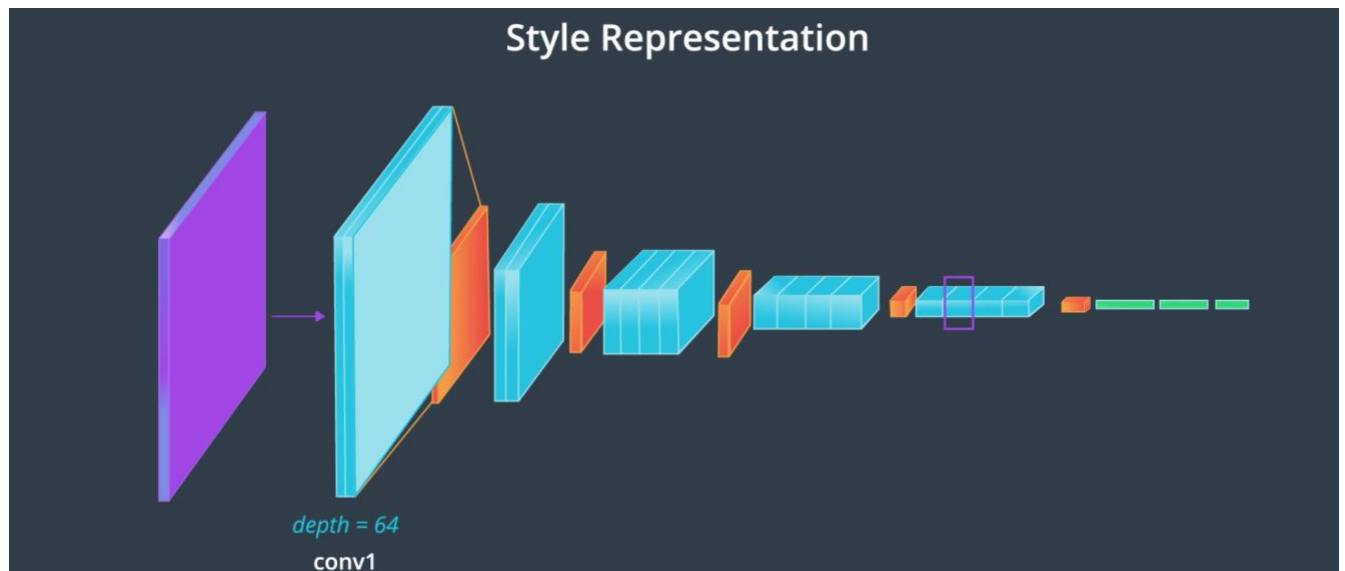
These representations are the basis for applications like Style Transfer and Deep Dream. Which compose images based on CNN layer activations and extracted features.



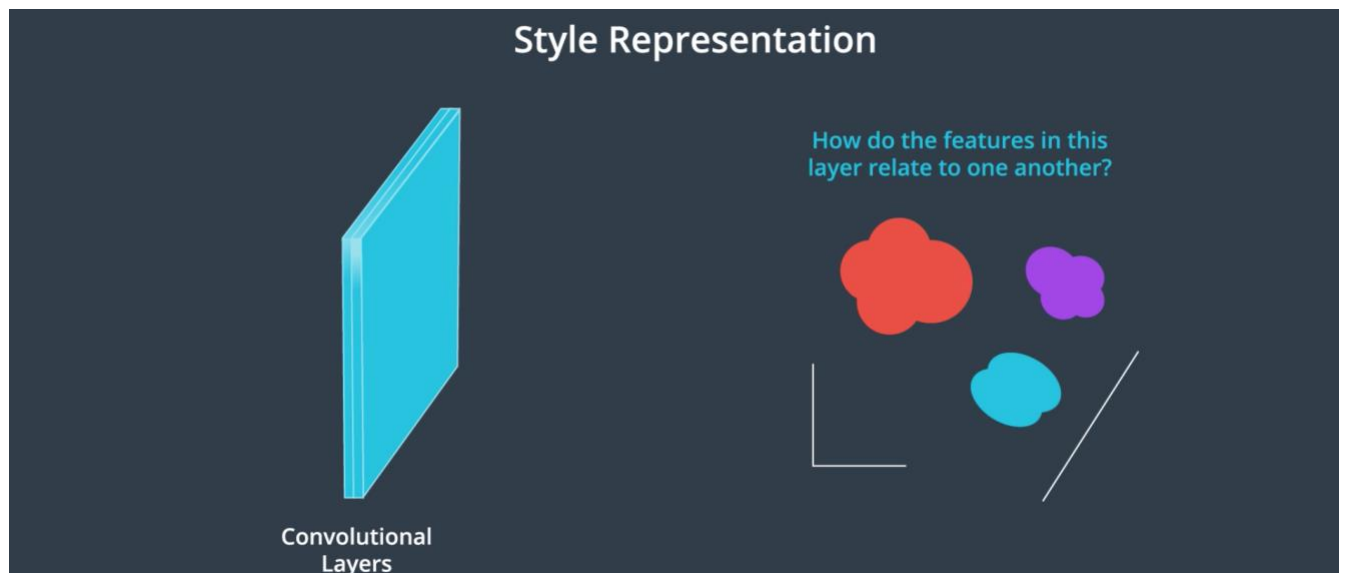
In this lesson we'll focus on learning about and implementing the style transfer algorithm. Style transfer allows you to apply the style of one image to another image of your choice. For example, here we've applied the style of Hokusai, The Great Wave [inaudible] print to a picture of a cat. The key to this technique is using a trained CNN to separate the content from the style of an image. If you can do this then you can merge the content of one image with the style of another and create something entirely different. Next, we'll talk about how style and content can be separated and by the end of this lesson, you'll have all the knowledge you need to generate a stylized image of your own design.



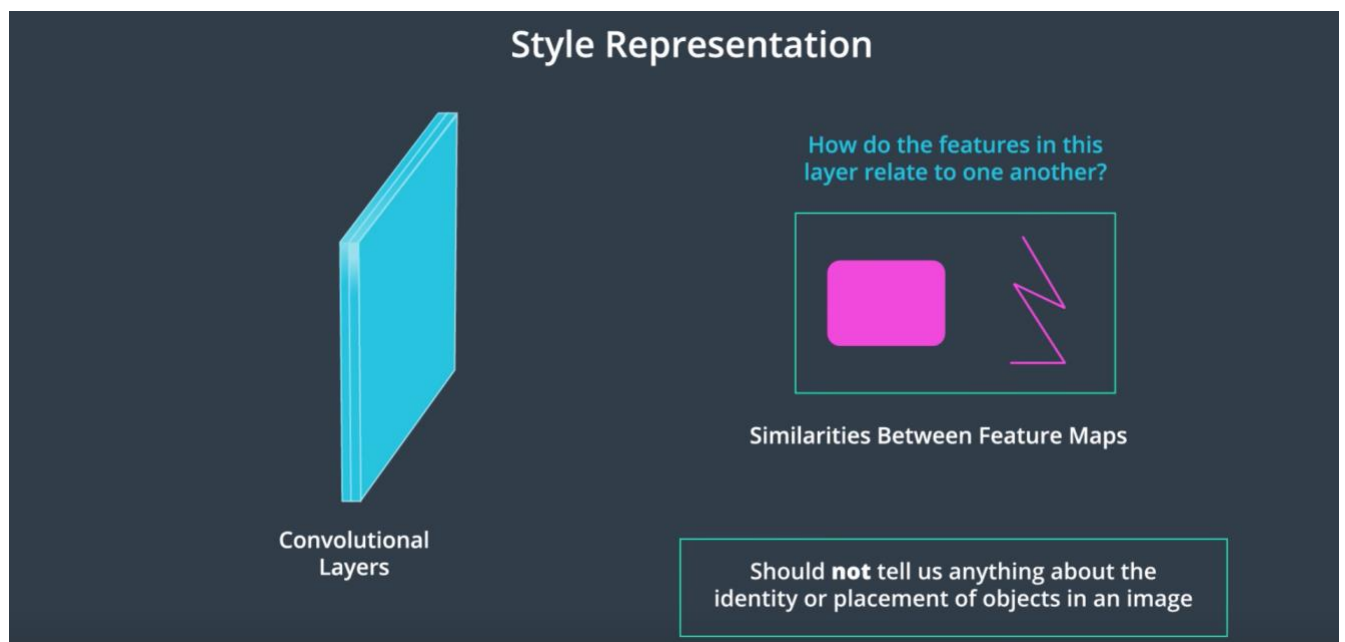
126. When a CNN is trained to classify images, its convolutional layers learn to extract more and more complex features from a given image. Intermittently, max pooling layers will discard detailed spatial information, information that's increasingly irrelevant to the task of classification. The effect of this is that as we go deeper into a CNN, the input image is transformed into feature maps that increasingly care about the content of the image rather than any detail about the texture and color of pixels. Later layers of a network are even sometimes referred to as a content representation of an image. In this way, a trained CNN has already learned to represent the content of an image, but what about style? Style can be thought of as traits that might be found in the brush strokes of a painting, its texture, colors, curvature, and so on. To perform style transfer, we need to combine the content of one image with the style of another. So, how can we isolate only the style of an image? To represent the style of an input image, a feature space designed to capture texture and color information is used. This space essentially looks at spatial correlations within a layer of a network. A correlation is a measure of the relationship between two or more variables.



For example, you could look at the features extracted in the first convolutional layer which has some depth. The depth corresponds to the number of feature maps in that layer.



For each feature map, we can measure how strongly its detected features relate to the other feature maps in that layer. Is a certain color detected in one map similar to a color in another map? What about the differences between detected edges and corners, and so on? See which colors and shapes in a set of feature maps are related and which are not.



Say, we detect that mini-feature maps in the first convolutional layer have similar pink edge features. If there are common colors and shapes among the feature maps, then this can be thought of as part of that image's style. So, the similarities and differences between features in a layer should give us some information about the texture and color information found in an image. But at the same time, it should leave out information about the actual arrangement and identity of different objects in that image.



**Content Image**

*Object and shape arrangement*



**Style Image**

*Colors and textures*

Now, we've seen that content and style can be separate components of an image. Let's think about this in a complete style transfer example. Style transfer will look at two different images. We often call these the style image and the content image. Using a trained CNN, style transfer finds the style of one image and the content of the other. Finally, it tries to merge the two to create a new third image. In this newly created image, the objects and their arrangement are taken from the content image, and the colors and textures are taken from the style image.

## Style Transfer



Content Image

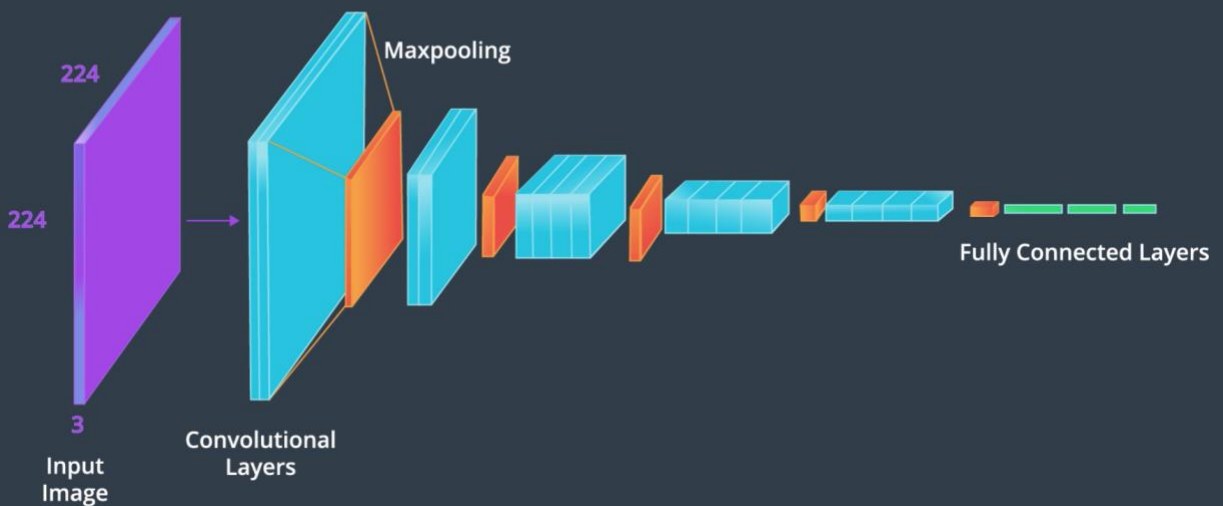


Style Image



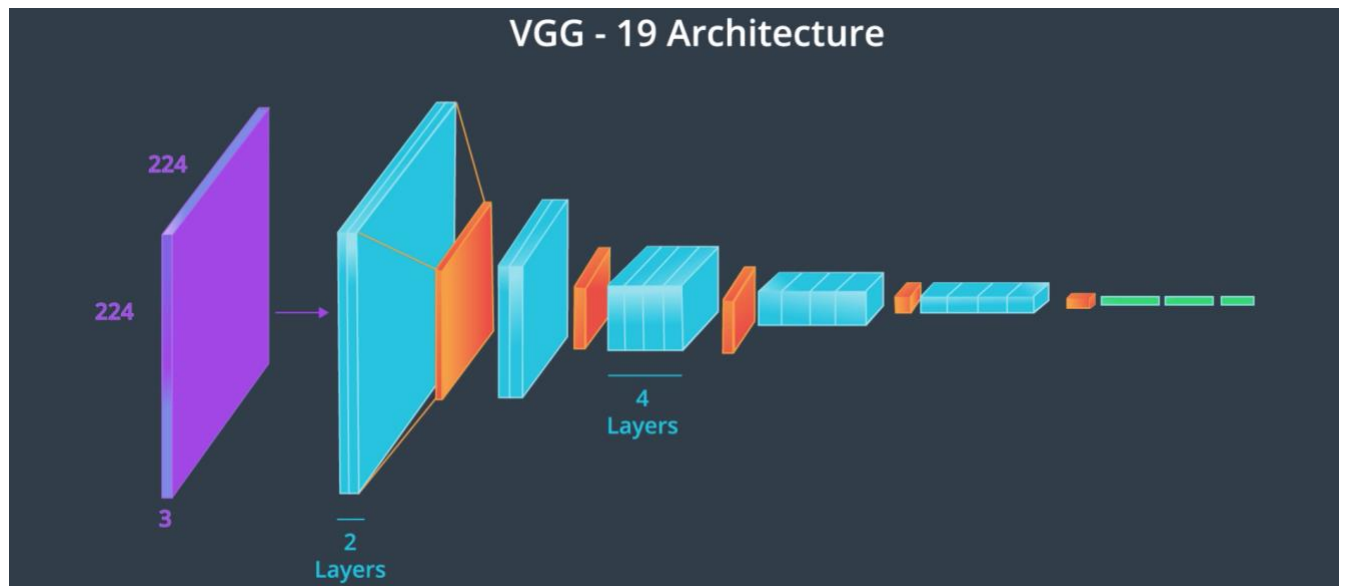
Here's our example of an image of a cat, the content image, being combined with a Hokusai-style image of waves. Effectively, style transfer creates a new image that keeps the cat content, but renders it with the colors, the print texture, the style of the wave artwork. This is the theory behind how style transfer works. Next, let's talk more about how we can actually extract features from different layers of a trained model and use them to combine the style and content of two different images.

## VGG - 19 Architecture

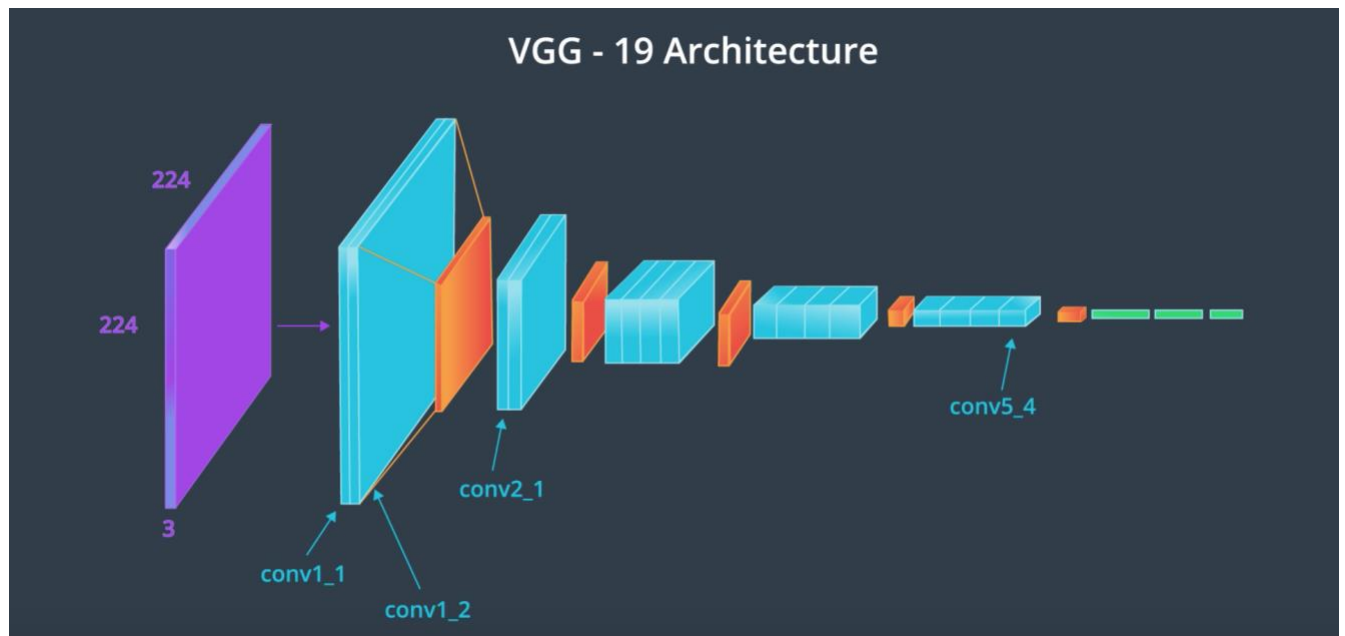


127. In the code example that I'll go through, we'll recreate a style transfer method that's outlined in the paper, image style transfer using convolutional neural networks. In this paper, style transfer uses the features found in the 19 layer VGG network, which I'll call VGG 19. This network accepts a color image as input and passes it through a series of convolutional and pooling layers. Followed finally by three fully connected layers but classify the past in image.

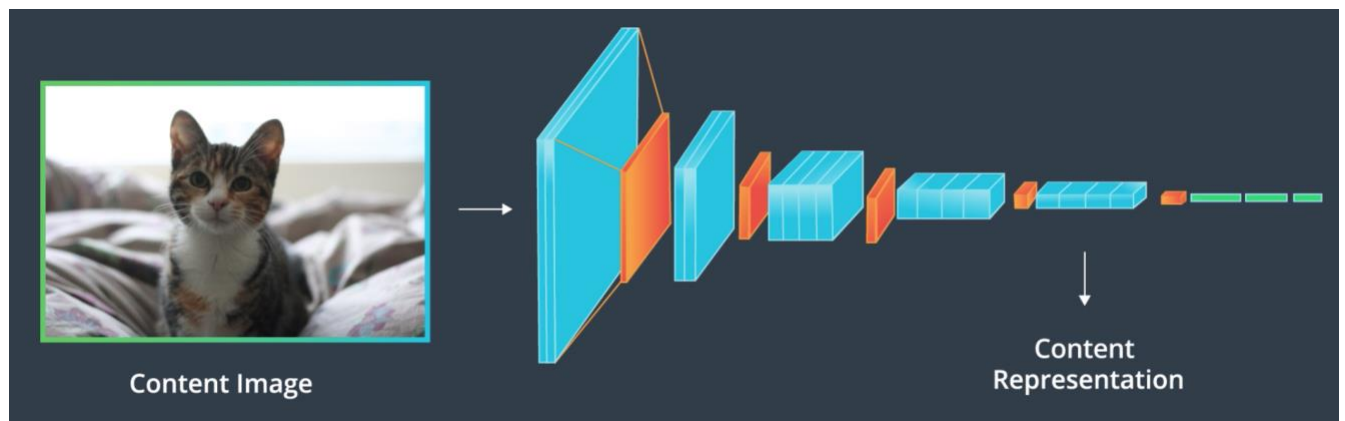




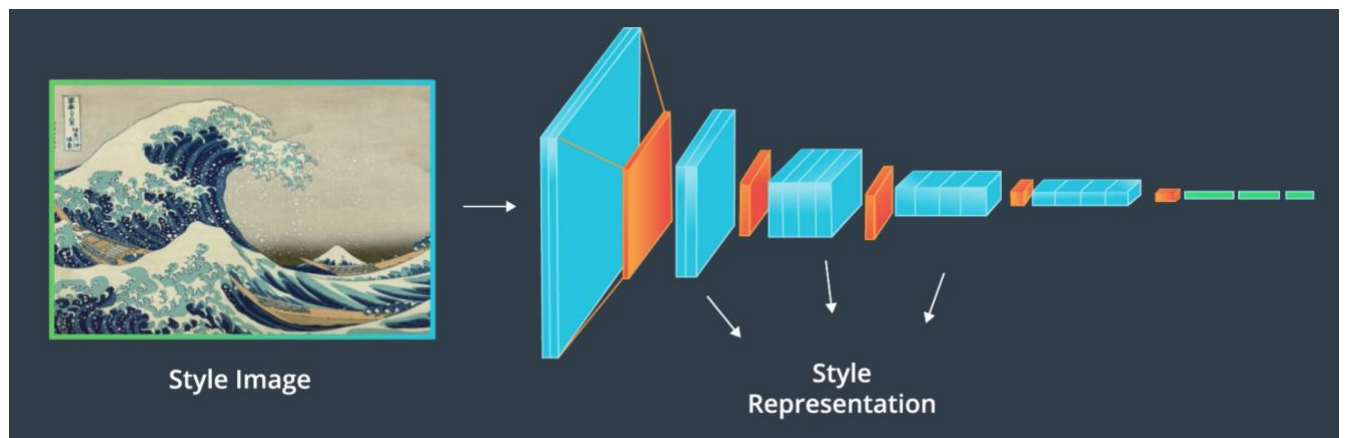
In-between the five pooling layers, there are stacks of two or four convolutional layers. The depth of these layers is standard within each stack, but increases after each pooling layer.



Here they're named by stack and their order in the stack. Conv one, one, is the first convolutional layer that an image is passed through in the first stack. Conv two, one, is the first convolutional layer in the second stack. The deepest convolutional layer in the network is conv five, four.

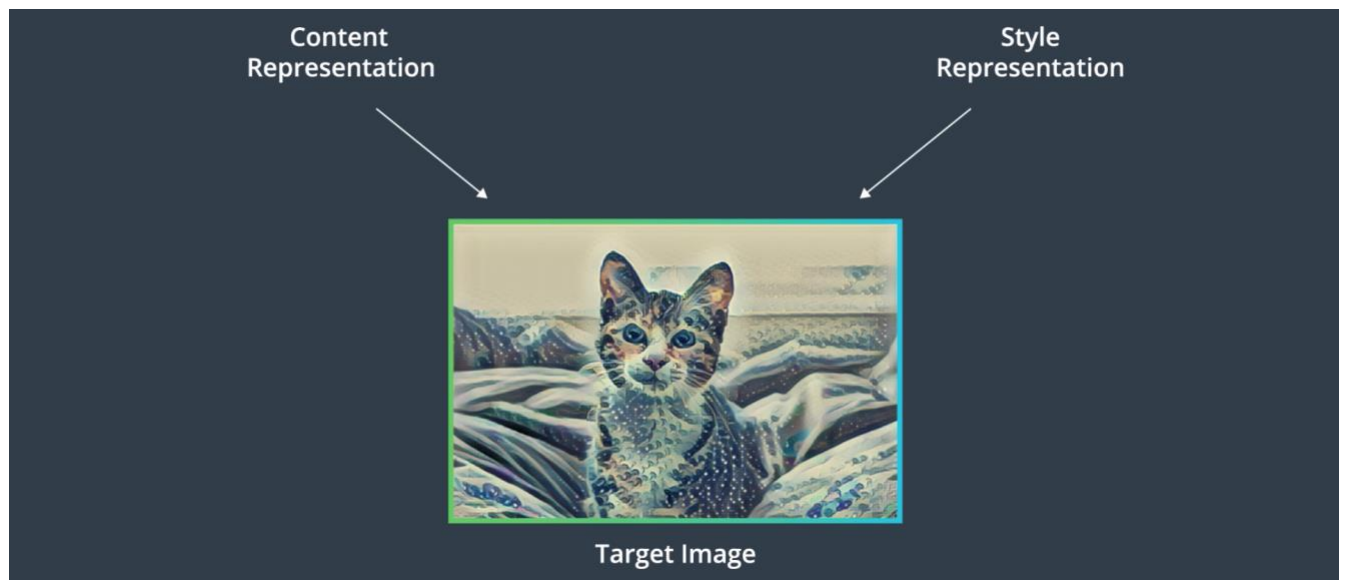


Now, we know that style transfer wants to create an image that has the content of one image and the style of another. To create this image, which I'll call our target image, it will first pass both the content and style images through this VGG 19 network. First, when the network sees the content image, it will go through the feed-forward process until it gets to a convolutional layer that is deep in the network. The output of this layer will be the content representation of the input image.

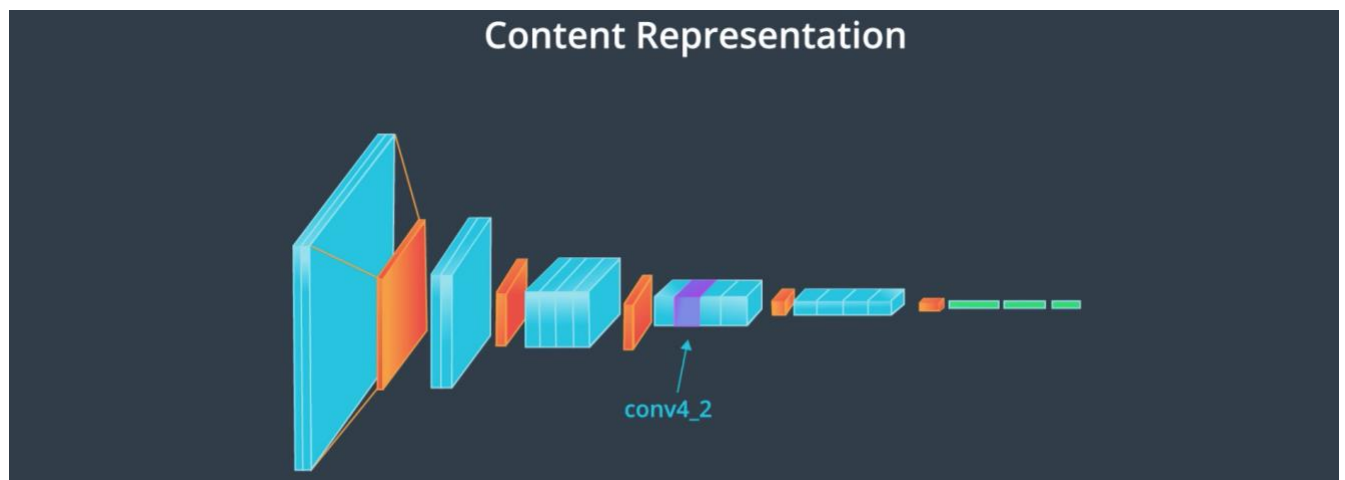


Next, when it sees the style image, it will extract different features from multiple layers that represent the style of that image.

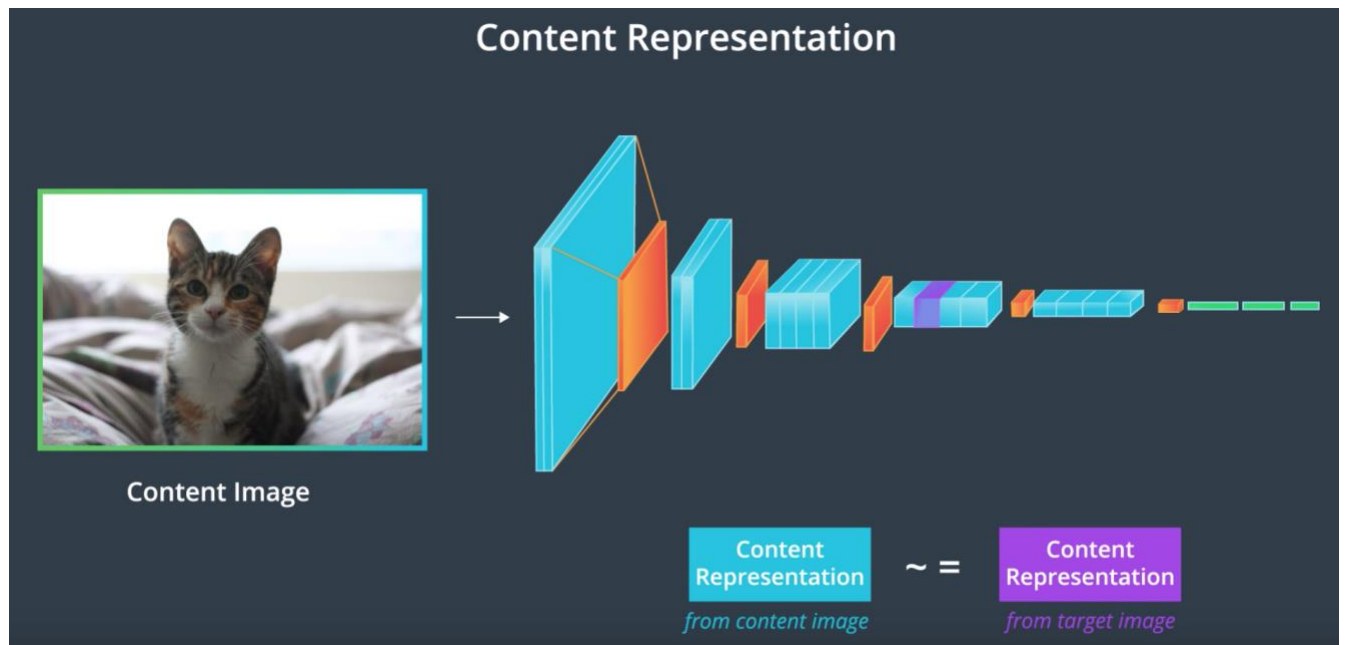




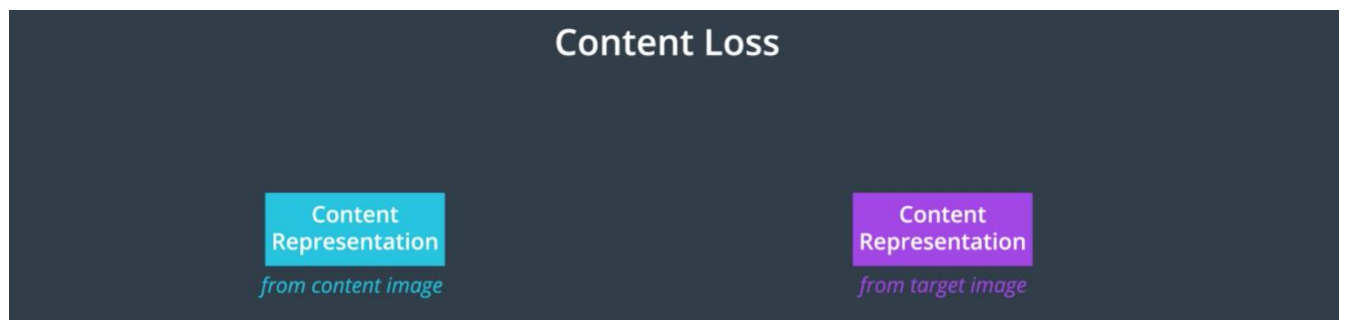
Finally, it will use both the content and style representations to inform the creation of the target image. The challenge is how to create the target image. How can we take a target image which often starts as either a blank canvas or as a copy of our content image, and manipulate it so that it's content is close to that of our content image, and it's style is close to that of our style image?



Let's start by discussing in the content. In the paper, the content representation for an image is taken as the output from the fourth convolutional stack, conv four, two.



As we form our new target image, we'll compare it's content representation with that of our content image. These two representations should be close to the same even as our target image changes it's style.



To formalize this comparison, we'll define a content loss, a loss that calculates the difference between the content and target image representations, which I'll call  $C_c$  (Content content) and  $C_t$  (Target content) respectively. Tao: later we will have  $S_s$  (Style style) and  $S_t$  (Target style).

## Content Loss

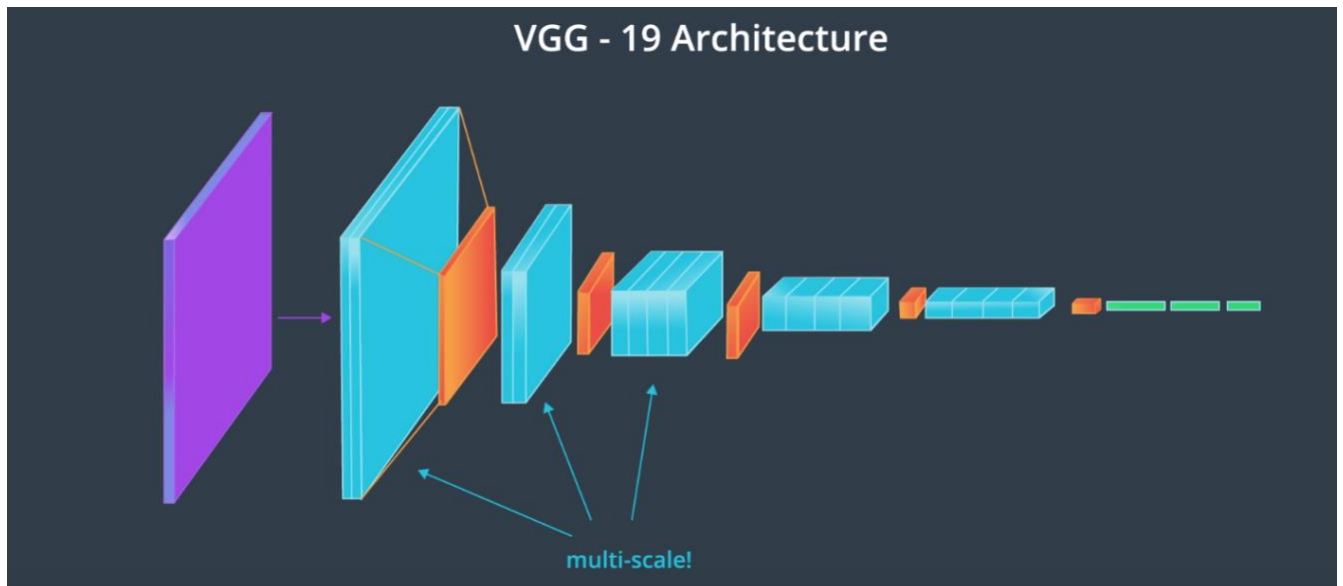
$C_c$

The only value that changes!

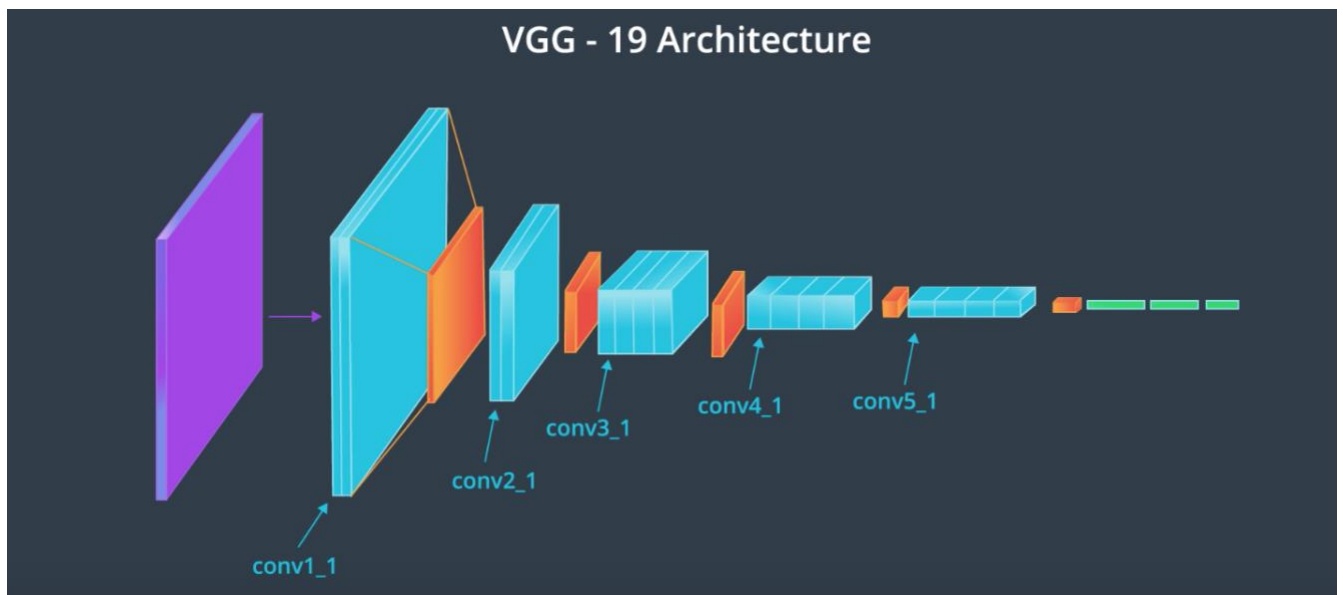
$T_c$

$$\mathcal{L}_{content} = \frac{1}{2} \sum (T_c - C_c)^2$$

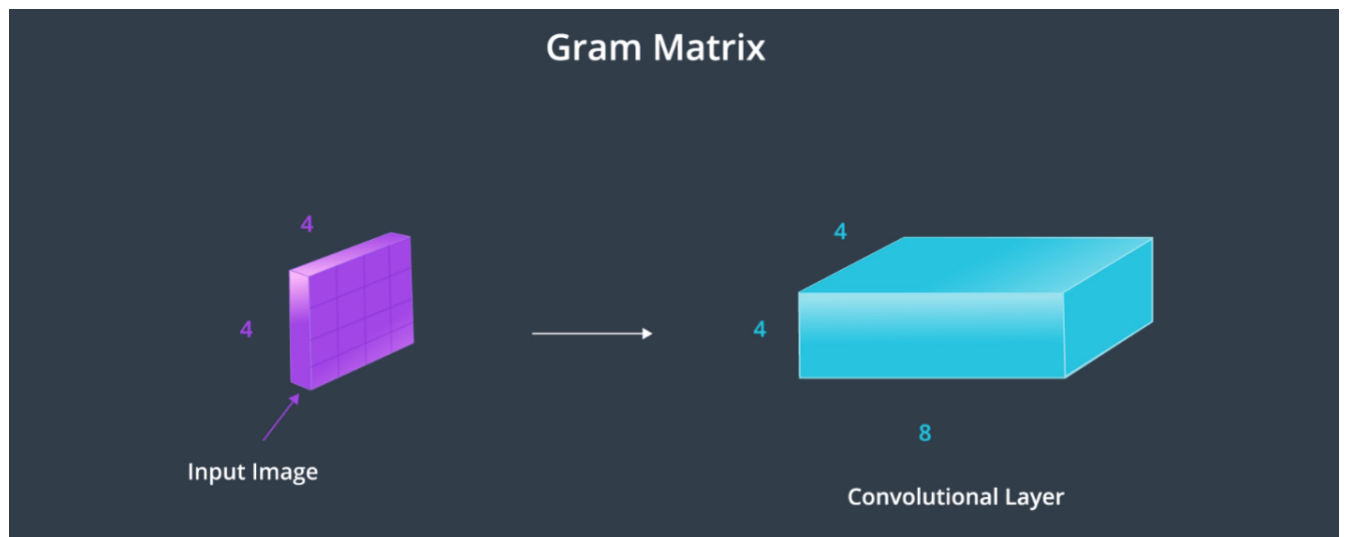
In this case, we calculate the mean squared difference between the two representations. This is our content loss, and it measures how far away these two representations are from one another. As we try to create the best target image, our aim will be to minimize this loss. This is similar to how we used loss and optimization to determine the weights of a CNN during training. But this time, our aim is not to minimize classification error. In fact, we're not training the CNN at all. Rather, our goal is to change only the target image, updating its appearance until its content representation matches that of our content image. So, we're not using the VGG 19 network in a traditional sense, we're not training it to produce a specific output. But we are using it as a feature extractor, and using back propagation to minimize a defined loss function between our target and content images. In fact, we'll have to define a loss function between our target and style images, in order to produce an image with our desired style. Next, let's learn more about how to represent the style of an image.



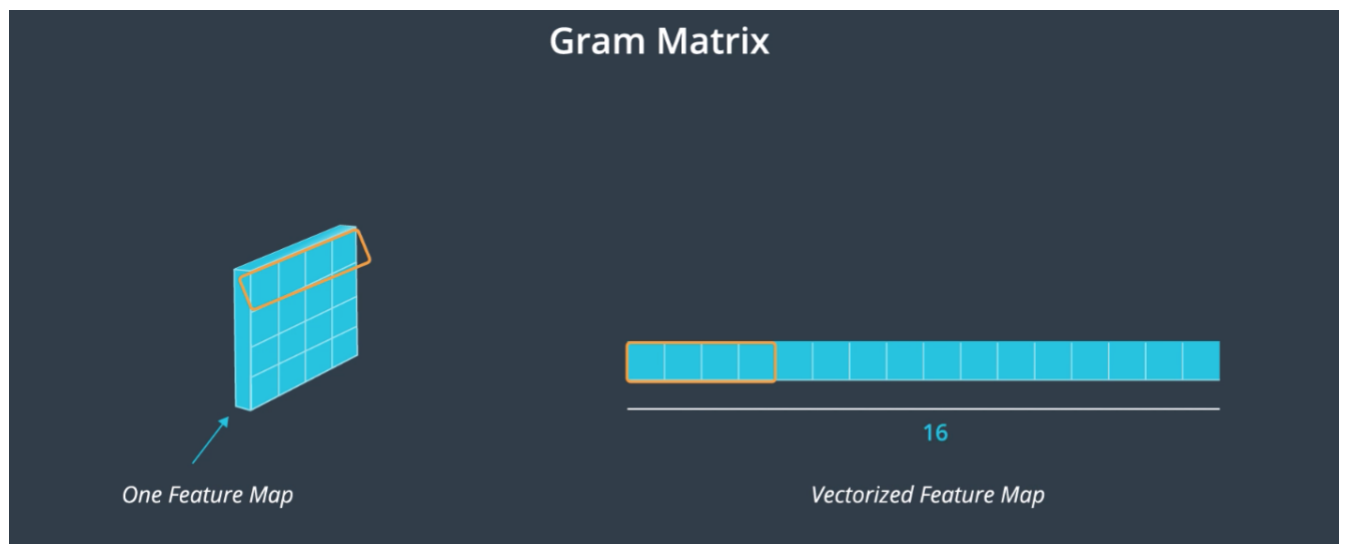
128. To make sure that our target image has the same content as our content image, we formalize the idea of a content loss, that compares the content representations of the two images. Next, we want to do the same thing for the style representations of our target image and style image. The style representation of an image relies on looking at correlations between the features in individual layers of the VGG-19 network, in other words looking at how similar the features in a single layer are. Similarities will include the general colors and textures found in that layer. We typically find the similarities between features in multiple layers in the network. By including the correlations between multiple layers of different sizes, we can obtain a multiscale style representation of the input image, one that captures large and small style features.



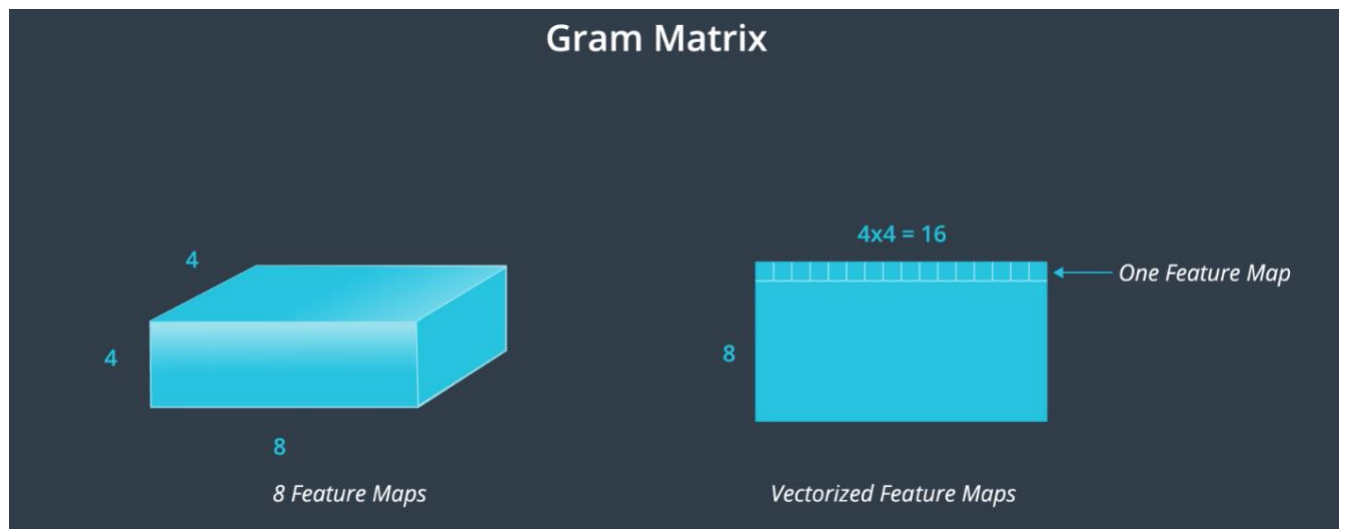
The style representation is calculated as an image passes through the network at the first convolutional layer in all five stacks, conv1\_1, conv2\_1, up to conv5\_1. The correlations at each layer are given by a Gram matrix.



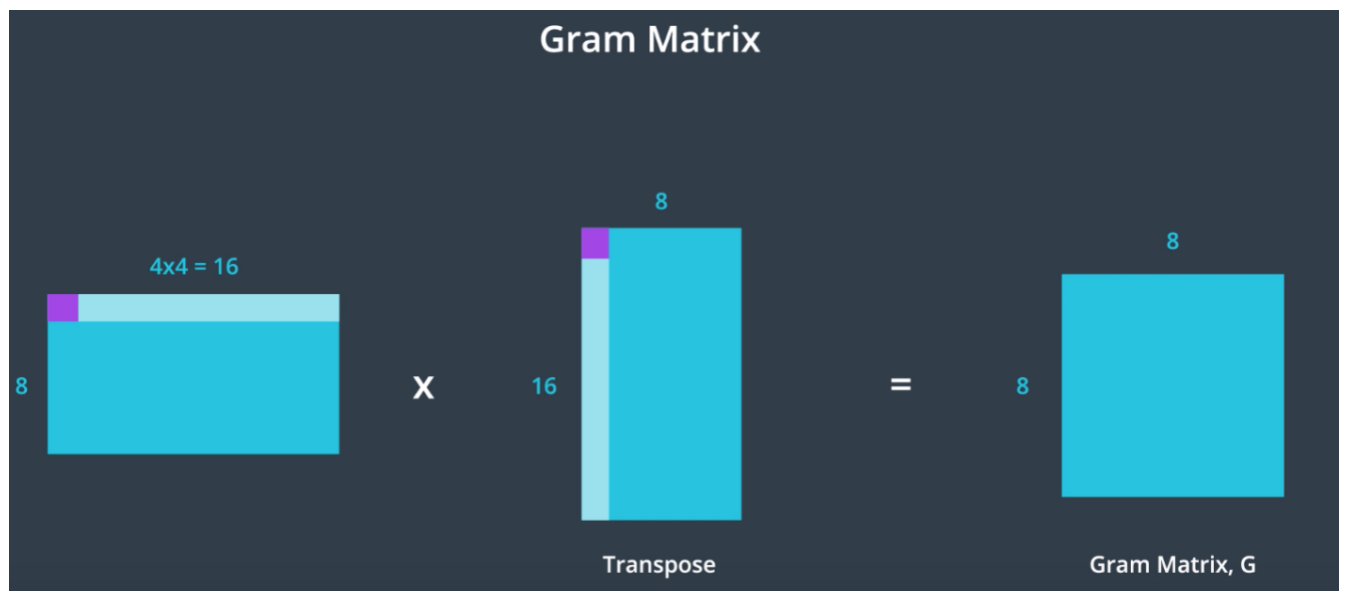
The matrix is a result of a couple of operations, and it's easiest to see in a simple example. Say, we start off with a four by four image, and we convolve it with eight different image filters to create a convolutional layer. This layer will be four by four in height and width, and eight in depth. Thinking about the style representation for this layer, we can say that this layer has eight feature maps that we want to find the relationships between.



The first step in calculating the Gram matrix, will be to vectorize the values in this layer. This is very similar to what you've seen before, in the case of vectorizing an image so that it can be seen by an NLP. The first row of four values in the feature map, will become the first four values in a vector with length 16. The last row will be the last four values in that vector.



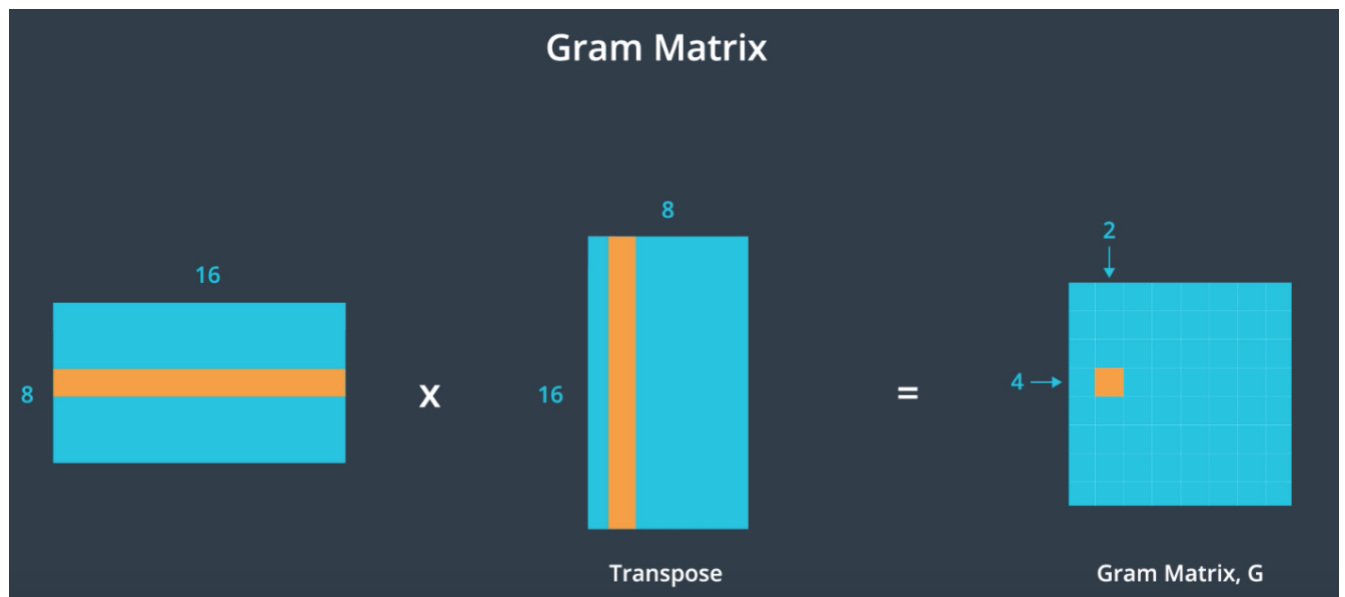
By flattening the XY dimensions of the feature maps, we're converting a 3D convolutional layer into a 2D matrix of values.



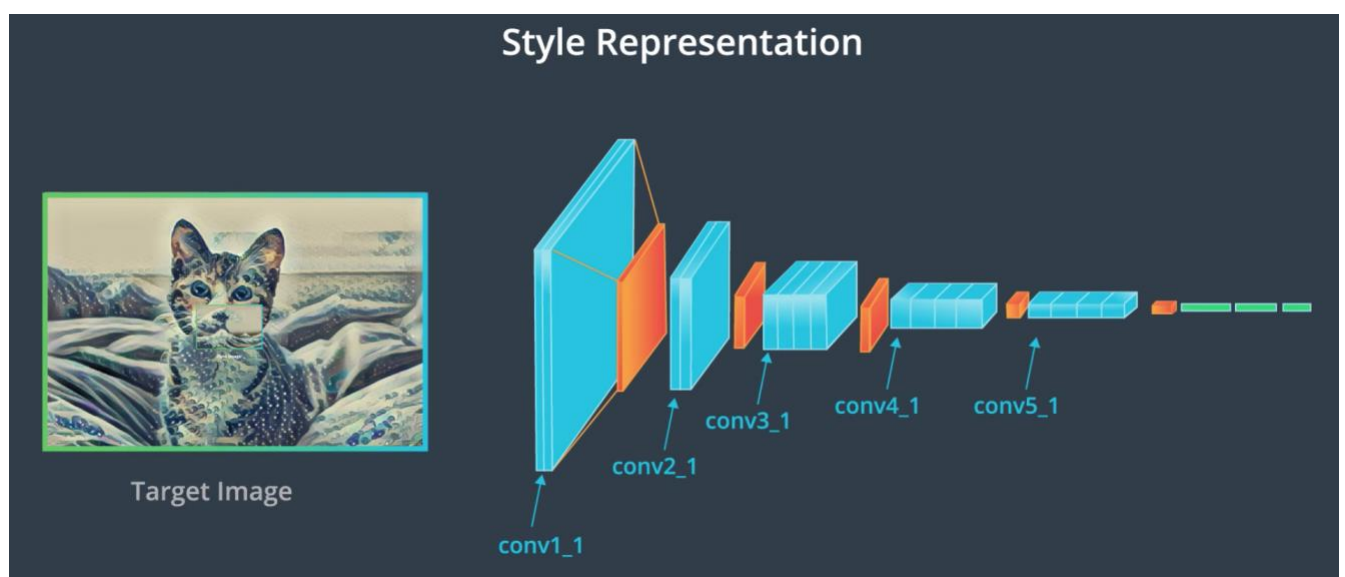
The next step is to multiply this matrix by its transpose. Essentially, multiplying the features in each map to get the gram matrix. This operation treats each value in the feature map as an individual sample, unrelated in space to other values. So, the resultant Gram matrix contains non-localized information about the layer. Non-localized information, is information that would still be there even if an image was shuffled around in space. For example, even if the content of a filtered image is not identifiable, you should still be able to see prominent colors and textures the

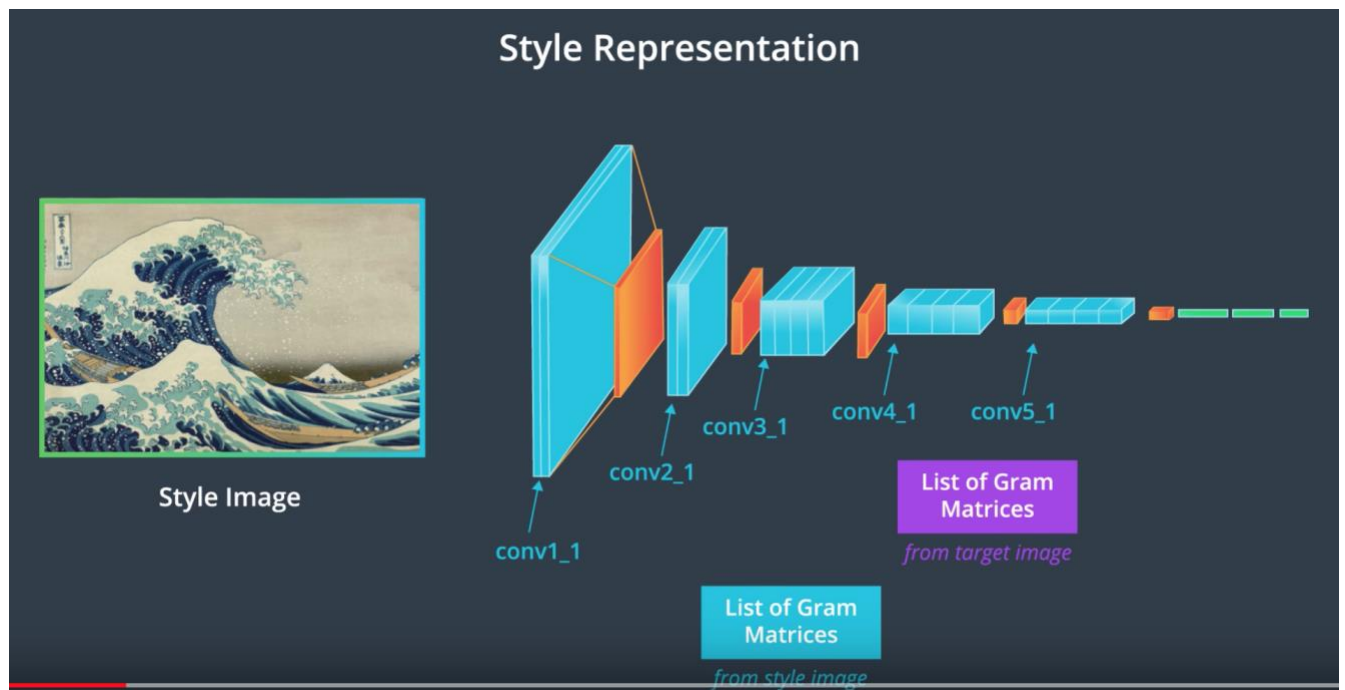


style. Finally, we're left with the square eight by eight Gram matrix, whose values indicate the similarities between the the layers (Tao: see below).



So, row four column two, will hold a value that indicates the similarity between the fourth and second feature maps in a layer. Importantly, the dimensions of this matrix are related only to the number of feature maps in the convolutional layer, it doesn't depend on the dimensions of the input image. I should note that the Gram matrix is just one mathematical way of representing the idea of shared in prominent styles. Style itself is an abstract idea but the Gram matrix, is the most widely used in practice. Now that we've defined the Gram matrix as having information about the style of a given layer, next we can calculate a style loss that compares the style of our target image and our style image.





129. To calculate the style loss between a target and style image, we find the mean squared distance between the style and target image gram matrices, all five pairs that are computed at each layer in our predefined list, conv1\_1 up to conv5\_1.

## Style Loss

The only value that changes!

$S_s$

$T_s$

$$\mathcal{L}_{style} = a \sum_i w_i (T_{s,i} - S_{s,i})^2$$

Tao: i is the index of layer.

These lists, I'll call  $S_s$  and  $T_s$ , and  $A$  is a constant that accounts for the number of values in each layer. We'll multiply these five calculated distances by some style weights  $W$  that we specify, and then add them up. The style weights are values that will give more or less weight to the calculated style loss at each of the five layers, thereby changing how much effect each layer style representation will have on our final target image. Again, we'll only be changing the target image's style representations as we minimize this loss over some number of iterations.

## Total Loss

$$\mathcal{L}_{content} = \frac{1}{2} \sum (T_c - C_c)^2$$

+

$$\mathcal{L}_{style} = a \sum_i w_i (T_{s,i} - S_{s,i})^2$$


So, now we have the content loss, which tells us how close the content of our target image is to that of our content image, and the style loss, which tells us how close our target is in style to our style image. We can now add these losses together to get the total loss, and then use typical back propagation and optimization to reduce this loss by iteratively changing the target image to match our desired content and style.

**Total Loss**

$$\boxed{\alpha} \mathcal{L}_{content} + \boxed{\beta} \mathcal{L}_{style} \quad \boxed{\frac{\alpha}{\beta}}$$


Content Weight                      &                      Style Weight  
Often Much Larger


130. Before we move on to coding, I should mention one more detail about the total style transfer loss. We have values for the content and style loss, but because they're calculated differently, these values will be pretty different, and we want our target image to take both into account fairly equally. So, it's necessary to apply constant weights, alpha and beta, to the content and style losses, such that the total loss reflects an equal balance. In practice, this means multiplying the style loss by a much larger weight value than the content loss. You'll often see this expressed as a ratio of the content and style weights, alpha over beta.



Content Image

$$\frac{\alpha}{\beta} = \frac{1}{10}$$







Style Image

[Image Style Transfer Using Convolutional Neural Networks, L. Gatys, A. Ecker, M. Bethge, 2016]


In the paper, we see the effects of a bigger or smaller ratio. Here is an example of a content and style image. We can imagine that the content weight alpha is one, and that the style weight beta is 10.





Content Image




Style Image

$10^{-4}$   


$10^{-3}$   


$10^{-2}$   


$10^{-1}$   


[Image Style Transfer Using Convolutional Neural Networks, L. Gatys, A. Ecker, M. Bethge, 2016]

You can see that this target image is mostly content without much style, but as beta increases to 100, then 1,000, and alpha stays at one, we can see more and more style in the generated image.

Finally, we see that this can go too far, and at a ratio of 10 to the negative four, we see that most of the content is gone, and only style remains. So, in general, the smaller the alpha-beta ratio, the more stylistic effect you will see. This makes intuitive sense because a smaller ratio corresponds to a larger value for beta, the style weight. You may find that certain ratios work well for one image, but not another. These weights will be good values to change to get the exact kind of stylized effect that you want. All right. Now that you know the theory and math behind using a pre-trained CNN to separate the content and style of an image, next, you'll see how to implement style transfer in PyTorch.

Not\_read\_begin

Notebook-style\_transfer-exercise\_begin

localhost:8888/notebooks/deep-learning-v2-pytorch/style-transfer/Style\_Transfer\_Exercise.ipynb

131. In this notebook, I'm going to go over an implementation of style transfer, following the details outlined in this paper. Image style transfer using convolutional neural networks. We're going to use a pre-trained VGG 19 net as a feature extractor. We can put individual images through this network, then at specific layers get the output, and calculate the content and style representations for an image. Basically, style transfer aims to create a new target image that tries to match the content of a given content image and the style of a given style image. Here's our example of a cat and a Hokusai wave image as an example. But with the code in this notebook, which is in our public GitHub, you'll be able to upload images of your own and really customize your own target image. Okay. So, first things first, I'm loading in our usual libraries including a new one, the PIL image library. This will help me load in any kind of image I want to. Next, I want to load in the pre-trained VGG 19 network that this implementation relies on. Using `torchvision` models, I can load this network in by name and ask for it to be pretrained. I actually just want to load in all the convolutional and pooling layers, which in this case are named features, and this is unique to the VGG network. You may remember doing something similar in the transfer learning lesson. We load in a model and we freeze any weights or parameters that we don't want to change. So, I'm saving this pre-trained model, then for every weight in this network, I'm setting `requires_grad` to false. This means that none of these weights will change. So now, VGG becomes a kind of fixed feature extractor, which is just what we want for getting content and style features later. Next, I'm going to check if a GPU device is available, and if it is, I'm moving my model to it. I do recommend running this example on a GPU just to speed up the target image creation process. Then, this is going to print out the VGG model and all its layers. We can see the sequence of layers is all numbered. Here's the first convolutional layer, and the first stack, `conv11`, and its number zero. You have the second in that first stack, `conv12`, and it's labeled two. Then, we have a max pooling layer and `conv21` in our second stack. We can keep going until the very last max pooling layer. Next, I'm going to continue loading and the resources I need to implement style transfer. So, I have my trained VGG model, and now I need to load in my content and style images. Here I have a function, which is going to transform any image into a normalized tensor. This will deal with jpegs or PNGs, and it will make sure that the size is reasonable for our purposes. Then, I'm going to actually load in style and content images from my images directory. I'm also reshaping my style image into the same shape as the content image. This reshaping step is just going to make the math nicely lined up later on. Then here, I also have a function to help me convert a normalized tensor



image back into a numpy image for display, and I can show you the images that I chose. I chose an octopus for my content image and a David Hockney painting for my style image. I really like to do people or animals for content images and bright artistic paintings for style images. But it's really going to be up to you in this case. Okay. So now, we have all the elements we need for style transfer. Next, I'm going to give you your first task. We know that we have to eventually pass our content and style images through our VGG network, and extract content and style features from particular layers. So, your job is going to be to complete this get features function. This function takes in an image and returns the outputs from layers that correspond to our content and style representations. This is going to be a list of features that are taken at particular layers in our VGG 19 model. This function is almost complete. It just needs a descriptive dictionary that maps our VGG 19 layers, that are currently numbered 0 through 36, into names like conv1\_1, conv2\_1, and so on. I've given you the first layer that we're interested in to start. If you need a reminder for which layers make up the content and style representations of an image, take a look at the original paper, and identify which layers we'll need, then list them all here. If you get stuck or don't know where to start, I'll show you my solution in the next video.

any some differences in the image synthesis are expected due to the different network architecture and optimisation algorithm we use.

### 3. Results

The key finding of this paper is that the representations of content and style in the Convolutional Neural Network are well separable. That is, we can manipulate both representations independently to produce new, perceptually meaningful images. To demonstrate this finding, we generate images that mix the content and style representation from two different source images. In particular, we match the content representation of a photograph depicting the riverfront of the Neckar river in Tübingen, Germany and the style representations of several well-known artworks taken from different periods of art (Fig 3). The images shown in Fig 3 were synthesised by matching the content representation on layer 'conv4\_2' and the style representation on layers 'conv1\_1', 'conv2\_1', 'conv3\_1', 'conv4\_1' and 'conv5\_1' ( $w_l = 1/5$  in those layers,  $w_l = 0$  in all other layers). The ratio  $\alpha/\beta$  was either  $1 \times 10^{-3}$  (Fig 3 B),  $8 \times 10^{-4}$  (Fig 3 C),  $5 \times 10^{-3}$  (Fig 3 D), or  $5 \times 10^{-4}$  (Fig 3 E, F).

#### 3.1. Trade-off between content and style matching

Figure 4. Relative weighting of matching content and style of the respective source images. The ratio  $\alpha/\beta$  between matching the content and matching the style increases from top left to bottom right. A high emphasis on the style effectively produces a texturised version of the style image (top left). A high emphasis on the content produces an image with only little stylisation (bottom right). In practice one can smoothly interpolate between the two extremes.

emphasis on content, one can clearly identify the photograph, but the style of the painting is not as well-matched ( $\alpha/\beta = 1 \times 10^{-1}$ , Fig 4, bottom right). For a specific pair of content and style images one can adjust the trade-off between content and style to create visually appealing images.

#### 3.2. Effect of different layers of the Convolutional Neural Network

Another important factor in the image synthesis process is the choice of layers to match the content and style representation on. As outlined above, the style representation is a multi-scale representation that includes multiple layers of the neural network. The number and position of these layers determines the local scale on which the style is matched, leading to different visual experiences (Fig 1, style representation).

132. Okay. If I look at the original paper and I scroll down close to the bottom, I can see exactly which layers make up our style and content representations. I see here that the content representation is taken as the output of layer conv4\_2, and the style representations are going to be made of features from the first convolutional layer in all five stacks, conv1\_1 up to conv5\_1.

Continue to talk about this notebook:

localhost:8888/notebooks/deep-learning-v2-pytorch/style-transfer/Style\_Transfer\_Exercise.ipynb

Back to our notebook, I basically want to map these numbers that point to these VGG 19 layers to their more descriptive name. So, 0 is Conv1\_1, which is easy enough. Next after this maxpooling layer, I can see I have Conv2\_1 at layer 5. I'll look for the next maxpooling layer and then Conv3\_1

at layer 10. So now I want to grab Conv4\_1 and Conv4\_2 for our content representation. Lastly, I also want to grab Conv5\_1. So, it might get features function, I've listed out all the layers whose outputs I'll need to form my style and content representations. I find it easiest to keep all these in one list and access them by name later, but you could also choose to separate out this content representation layer. Okay. Great. So this completes our get features function. Eventually, one of our tasks will be to pass a style image through the VGG model and extract the style features at the right layers which we've just specified. Then, once we get these style features at a specific layer, we'll have to compute the gram matrix. Your next exercise will be to complete this function, `gram_matrix`. This function takes in a tensor, the output of the convolutional layer, and returns the gram matrix, the correlations of all the features in that layer. So to complete this function, you'll want to take a look at the shape of the past in tensor. This tensor should be four-dimensional with a batch size a depth and a height and width. You'll need to reshape this so that the height and width are flattened, and then you can calculate the gram matrix by doing matrix multiplication. Try this out and then I'll go through one implementation next.

133. So, here is a complete Gram matrix function. This takes inner tensor which will be the output of some convolutional layer. Then the first thing I do is take a look at its size. Each tensor is going to be four-dimensional with a batch size, a depth, a height, and a width. I can ignore the batch size at this point because I'm really interested in the depth or number of feature maps, and the height and width. These dimensions then tell me all I need to know to then factorize this tensor. Next, I'm reshaping this tensor so that it's now a 2-D shape that has its spatial dimensions flattened. It retains the number of feature maps as the number of rows. So, it's D rows by H times W columns. Finally, I calculate the Gram matrix by matrix multiplying this tensor times its transpose. This effectively multiplies all the features and gets the correlations. Finally, I make sure to return that calculated matrix. Then, I can put all these pieces together, I have my Get features function and my Gram matrix function. Before I even start to form my target image, I know I want to get the features from my content and style image. Those are going to remain the same throughout this process. So, right here I'm calling Get features on our content image passing in our content image and the VGG model, and I do the same thing for our style features, passing on our style image and the VGG model. Here, I'm calculating all the gram matrices for each of my style layers conv 11 up to conv 51. This looks at all of the layers in our style features and computes the gram matrix. Then it returns a dictionary where I can call style grams with a given layer name and get the gram matrix for that layer. Then I'm going to create a target image. I could start with a blank slate, but it turns out to be easier to just start with a clone of the content image. This way, my image will not divert too far from my octopus content and my plan will be to iterate and change this image to stylize it more and more later. So, in preparation for changing this target image, I'm going to set `requires_grad` to true, and I'll move it to GPU if available. All right, the next part is the most involved part, and next we'll talk about how to set and calculate our style and content losses for creating interesting target images.

134. So after we have our content and style features and gram matrices, next we need to define style and content losses. Alongside these, we need to define loss weights. This way, we can iteratively update our target image. First up, we're defining our style weights for each of our individual style layers. Notice that conv4\_2 is excluded here. Now, these are just weights that are going to give one set of style features more importance than another's. For example, I prefer to weigh earlier layers a little bit more. These features are often larger due to the spatial size of these

feature maps. Whereas weighting later layers might emphasize more fine-grain features. But again, this is really a preference and up to you to customize. I'd recommend keeping the values within the zero to one range. Then we have  $\alpha$  and  $\beta$ , which I'm going to descriptively name our content weight and our style weight. Earlier, we discussed this as a ratio that makes sure that style and content are equally important in the target image creation process. Because of how style loss is calculated, we basically want to give our style loss a much larger weight than the content loss. Here, it's 1 times 10 to the 6th and content loss is just one. Now, if  $\beta$  is too large, you may see too much of a stylized effect, but these values are good starting points. Next, we enter the iteration loop, and here's where you're actually going to be changing your target image. Now, this is not a training process, so it's arbitrary where you stop updating the target image. I'd recommend at least 2,000 iterations, but you may want to do more or less depending on your computing resources and desired effect. So, in this iteration loop, I'm going to ask you to calculate the content loss first. This will just be the mean square difference between the target and content representations. I've given you some example code up here. We can get those representations by getting the features from our target image, and then comparing those features at a particular layer, in this case conv4\_2, to the features at that layer for our content image. We're going to subtract these two representations and then square that difference and calculate the mean. This will give us our content loss. Next in this loop, you're going to do something similar for the style loss. Only this time, you have to go through multiple layers for our multiple representations for style. Recall that each of our relevant layers was listed in our style\_weights dictionary above. Here, I'm assuming you've calculated our target features so you can access a single target feature by layer. You'll get some style features from the target image and calculate the gram matrix for that layer. Then you have to compare with this style\_gram for the style image at that layer weighting it with our specified style weight for this layer. Here, these are going to be added up and normalized by the number of values in that layer. Finally, you should be able to add up everything and calculate a total loss. This is what will be used to update the target image using typical back propagation. So, try out completing this loss code on your own. This is the last piece of code you'll need to implement style transfer. Then, you'll be able to test this method on target images of your own design.

135. All right, here's my final solution. I played around with the values for style\_weights giving more weight to earlier layers. I left the content and style\_weights at one and one times 10<sup>6</sup>. Then I got into the iteration loop. I first got a list of target features using our get\_features function. Then I defined our content loss by looking at the target\_features at the layer conv4\_2 and the content\_features at conv4\_2. So, I'm comparing my content image and my target image content representations. I found the distance between the two and calculated the mean squared difference. Then the style\_loss. For this one, I looked at every layer in our style\_weights dictionary. For each of these layers, I got the target\_feature at that layer. For example, this would be what happens if our target image goes through our VGG 19 Network and hits conv1\_1. The output of the convolutional layer is then fed into our gram\_matrix function. This gives us our target\_gram\_matrix. Earlier, I calculated a dictionary of gram matrices for our style image. So, I get the gram\_matrix for our style image by accessing that by layer. Then here, I'm calculating the mean squared difference between our style\_gram and target\_gram matrix. Again, this is for a particular layer and I weighted by the weights that I specified in our style\_weights dictionary. So, for example, for the first layer conv1\_1, I'm going to multiply the difference between the target and style\_gram matrices by one. Then I'm adding that layer style\_loss to our accumulated style\_loss and just dividing it by the size of that layer. This effectively normalizes our layer style\_loss. So, by the end

of this for loop, I have the value for this `style_loss` at all five of my convolutional layers added up. Finally, I can compute the `total_loss`, which is just my `content_loss` and `style_loss` summed up and multiplied by their respective weights. Our `content_loss` is multiplied by one and style is multiplied by one times `10_6`, and that's basically it. I run this loop for 2,000 iterations, but I showed intermittently images every 400. I printed out the loss, which was quite large, and I could see a difference in my octopus image right away. Then at the end of my 2,000 iterations, I displayed my content and my target image side-by-side. You can see that the target image still looks a lot like an octopus. In fact, I think I could have stylized this even more. But it also has the colors and some brushstroke texture from the Hockney painting that I used as a style image. Now, using this notebook, you should be able to choose any content and style image combo that you want. Some example images have been provided in this notebook folder for you, but if you run this in a local environment, you can combine any images. If you do end up making your own images, I'd encourage you to share it on social media. I know that I would love to see it.

Notebook-style\_transfer-exercise\_end

Not\_read\_end