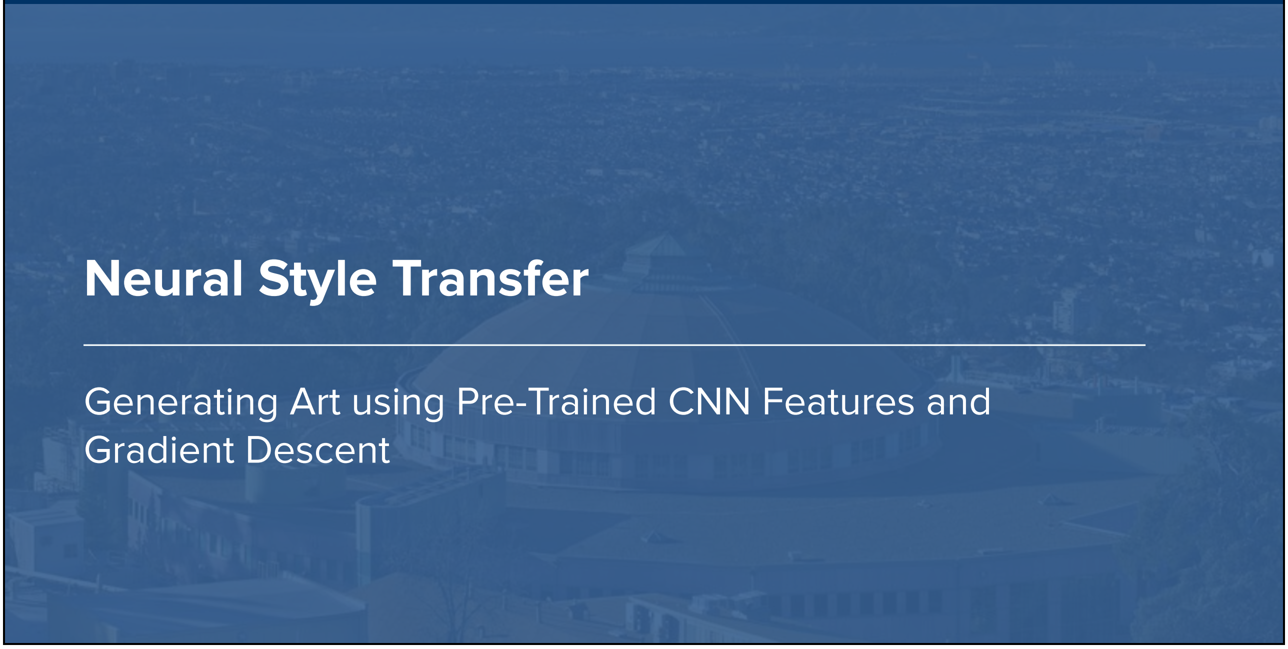
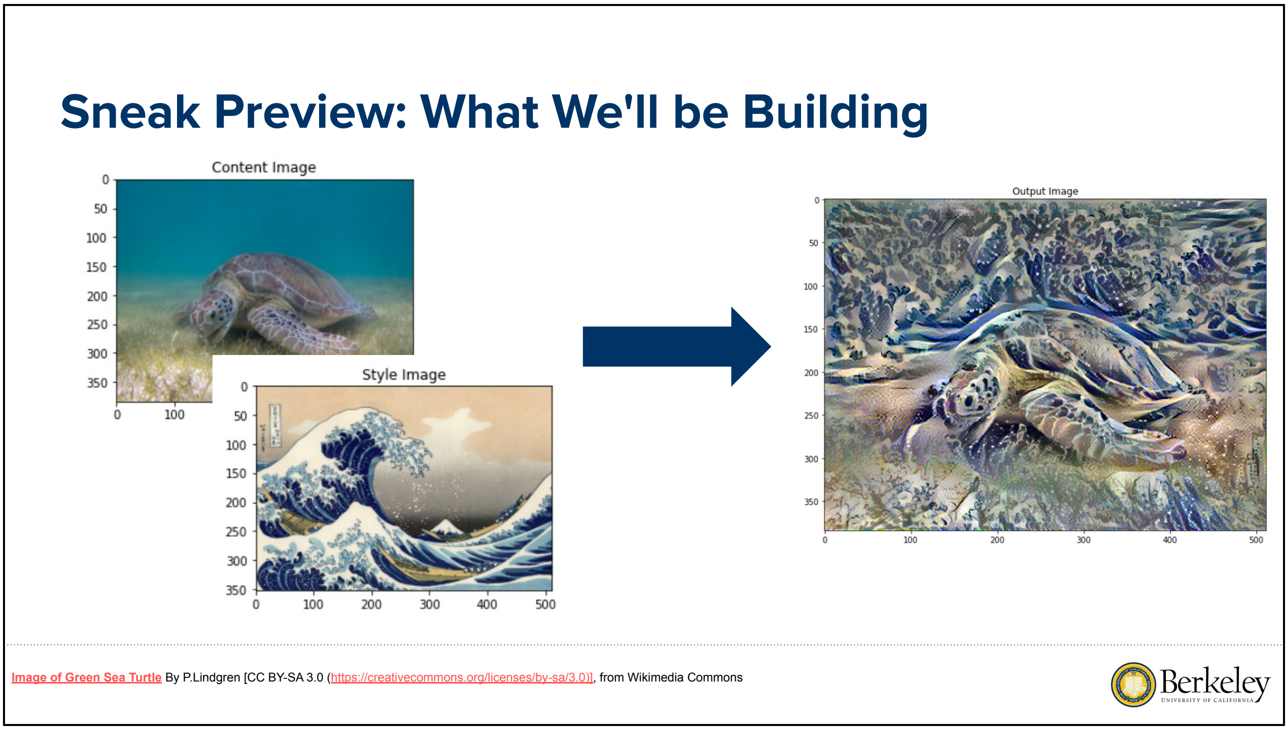


Neural Style Transfer

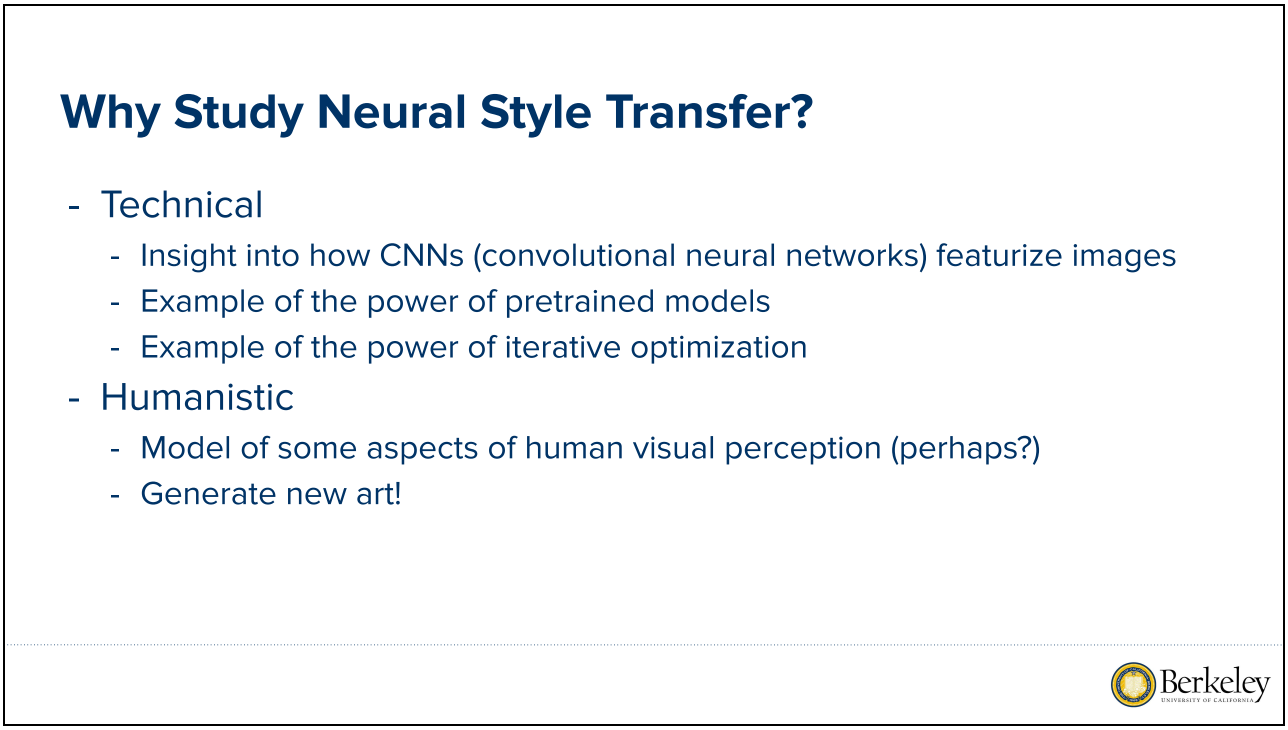
Generating Art using Pre-Trained CNN Features and Gradient Descent



Sneak Preview: What We'll be Building

Image of Green Sea Turtle By P.Lindgren [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], from Wikimedia Commons

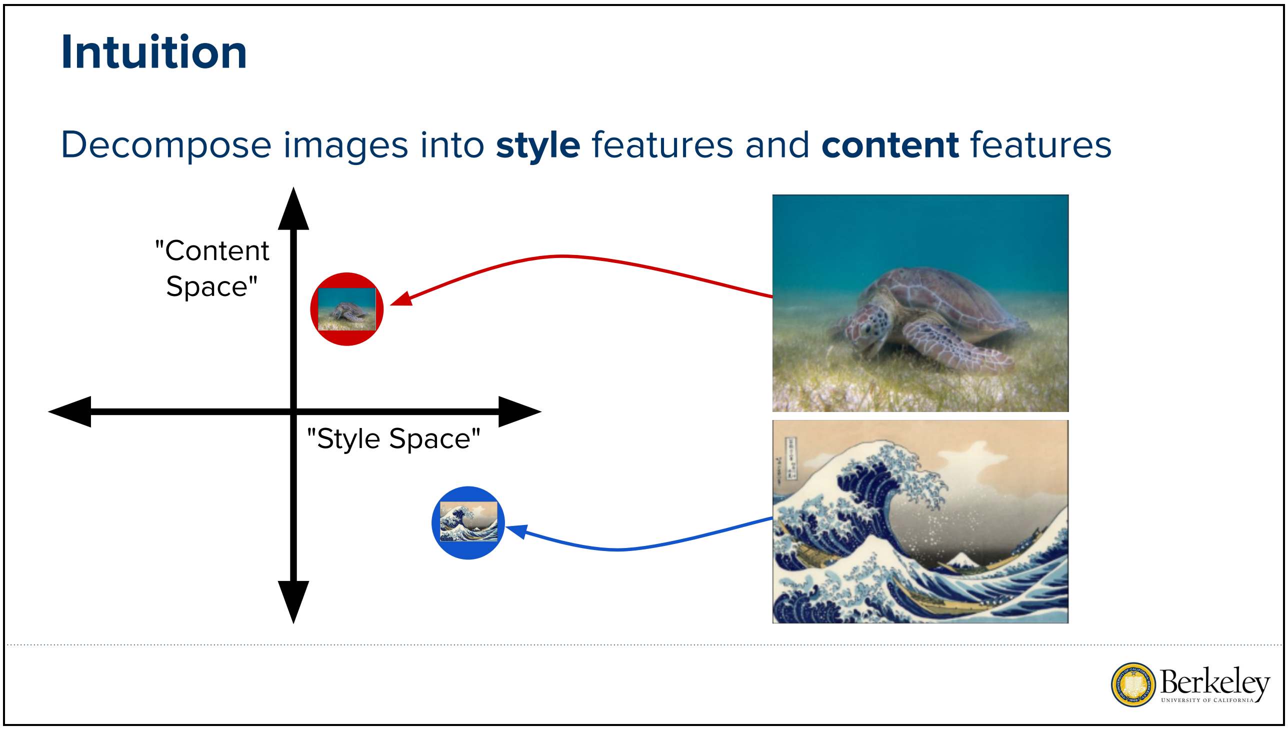
In this lecture we'll be talking about the design and implementation of an algorithm called "Neural Style Transfer," which can combine the "style" of one image with the "content" of another to produce new images, like this wave-turtle on the right. This might seem like an incredibly daunting task, but it's actually quite accessible. The key ingredient is that we don't have to start from scratch; we can build this style transfer algorithm on top of existing image classification model. We don't actually have to know much about how this pre-trained model works to use it for style transfer; we'll just be using its internal latent features as a black box.



Why Study Neural Style Transfer?

- Technical
  - Insight into how CNNs (convolutional neural networks) featurize images
  - Example of the power of pretrained models
  - Example of the power of iterative optimization
- Humanistic
  - Model of some aspects of human visual perception (perhaps?)
  - Generate new art!

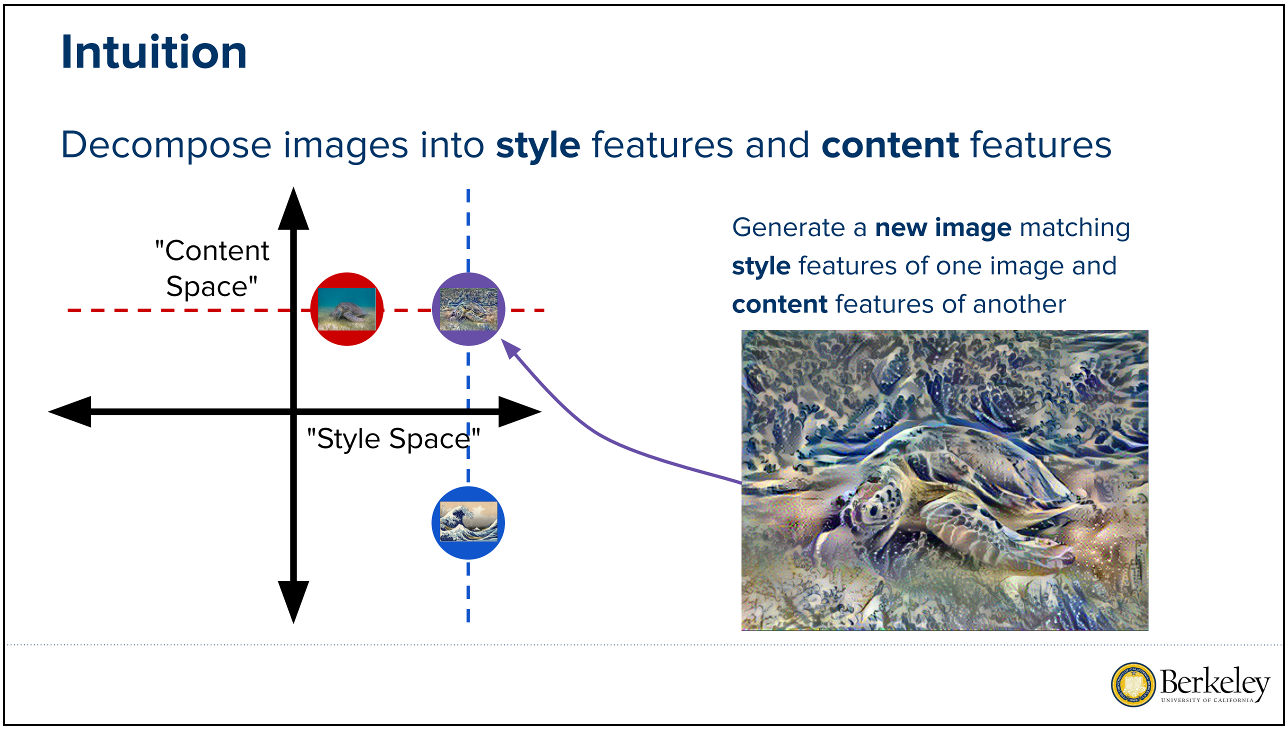
Why are we talking about this algorithm in the first place? There are some technical reasons this algorithm is a useful example to study: it gives us some qualitative understanding of what image classification models based on convolutional neural networks "look like" internally, and it's a nice example of how pretrained models and iterative optimization algorithms can get you a long way towards implementing novel ML applications. But we're also studying style transfer because you might argue that the images it generates, and the techniques it uses to generate them, reveal something about what the intuitive human / aesthetic concepts of "style" and "content" actually *are*. It's also just a nice example of the fun things you can do with modern ML — you can generate very compelling images and artworks using this algorithm and similar techniques.



Intuition

Decompose images into **style** features and **content** features

So, how might we go about actually designing an algorithm for style transfer? Well, imagine for the sake of argument that we had some magic black box that can decompose any input image into two vectors of features — "style" features and "content" features — and that these features actually correspond to our intuitive notions of "content" and "style". For simplicity, we can visualize the style feature space and the content feature space as both being one-dimensional, although of course in reality these spaces are very high-dimensional. In this simplified view, we can think of each image as being mapped to some point on a plane, where the "x coordinate" is the style of the image and "y coordinate" is the content.



Intuition

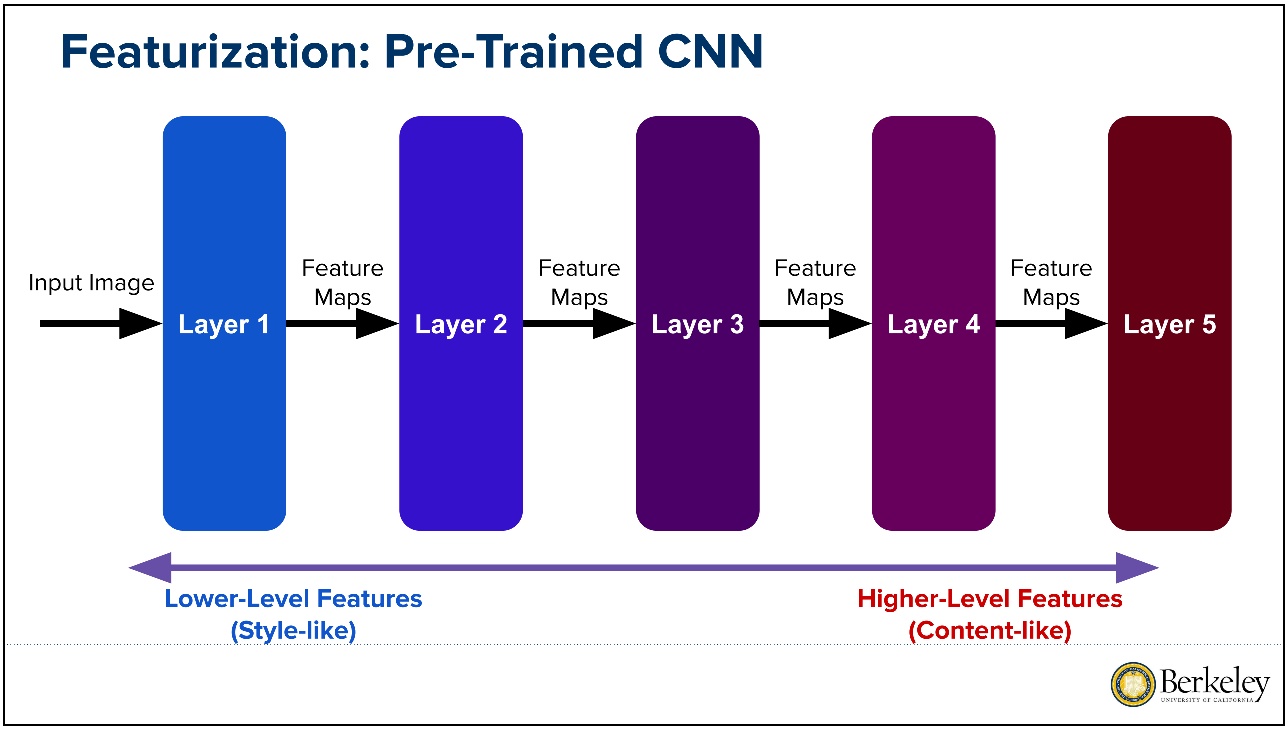
Decompose images into **style** features and **content** features

"Content Space"

"Style Space"

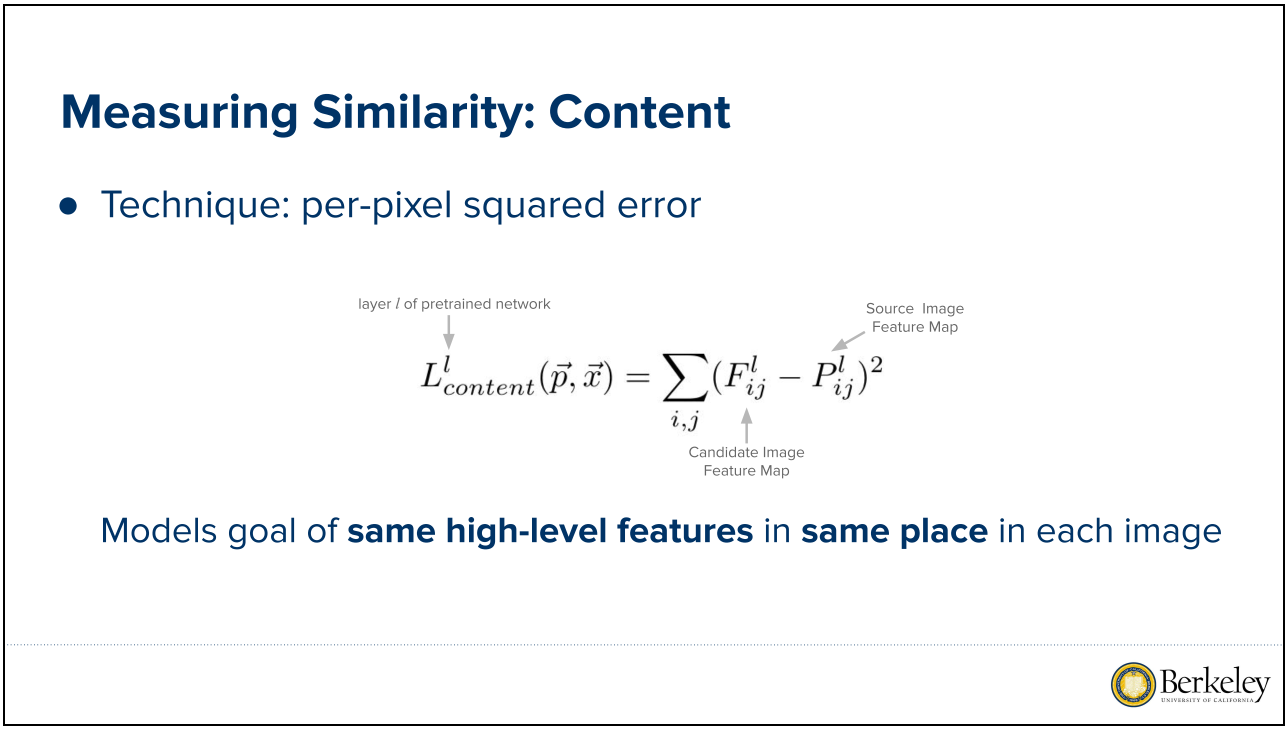
Generate a **new image** matching **style** features of one image and **content** features of another

Equipped with the ability to map images into this feature space, performing style transfer is actually quite straightforward. What we need to do is construct an image whose style features match those of the style source image as closely as possible, and whose content features match those of the content source image as closely as possible. In our simplified visualization, this is like finding an image whose "x coordinate" matches the featurization of the style image, and whose "y coordinate" matches the featurization of the content image.



Featurization: Pre-Trained CNN

So, can we actually build this magic black box that decomposes images into style features and content features? The answer is yes! The trick is to use a convolutional neural network model that's already been trained to perform an image-processing task, such as image classification, and extract the *internal* features into which it decomposes images at different layers in the model. We don't actually need to know much about how convolutional nets work to do this; all we need to know is that a convolutional neural net processes an image via a sequence of transformations, each of which transforms a representation of the image in one internal feature-space into another internal feature-space. In a trained model, we expect the feature spaces of early layers in the network to represent fairly "low-level" features like texture and color, which intuitively correspond to "style." Moreover, in e.g. a classification model which has been trained to recognize semantic content in images, we expect the later layers in the network to represent fairly "high-level" features like object types and composite shapes, which intuitively correspond to "content." Thus, to decompose an image into style and content features, we can just run a classification model on it and record the internal feature maps it generates in the process. We then take the early feature maps to represent style, and take the later feature maps represent content.



Measuring Similarity: Content

- Technique: per-pixel squared error

layer *l* of pretrained network

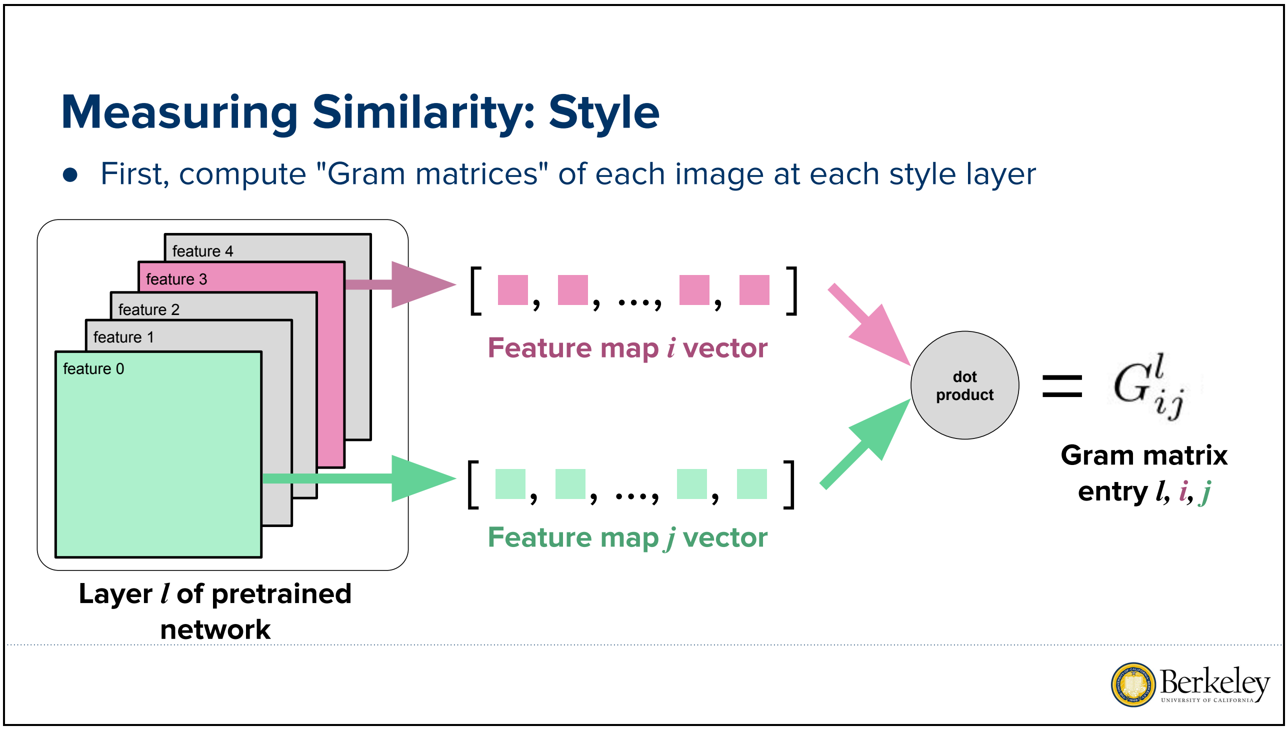
Source Image Feature Map

$$L^l_{content}(\vec{p},\vec{x}) = \sum_{i,j}(F^l_{ij} - P^l_{ij})^2$$

Candidate Image Feature Map

Models goal of **same high-level features** in **same place** in each image

Now suppose we have some candidate image, and we want to determine how similar its features are to the style and content features we want. How should we do this? Let's start with the content features. We want the generated image to contain the same "objects" — as modeled by high-level features — as the content source image, and we want it to have those objects in the same *place*. This means that to get a useful measure of content similarity, it suffices to extract content feature maps for the candidate image, content feature maps for the target image, and just measure the squared Euclidean distance between the two feature maps — in other words, the sum of the squared distances at each pixel.

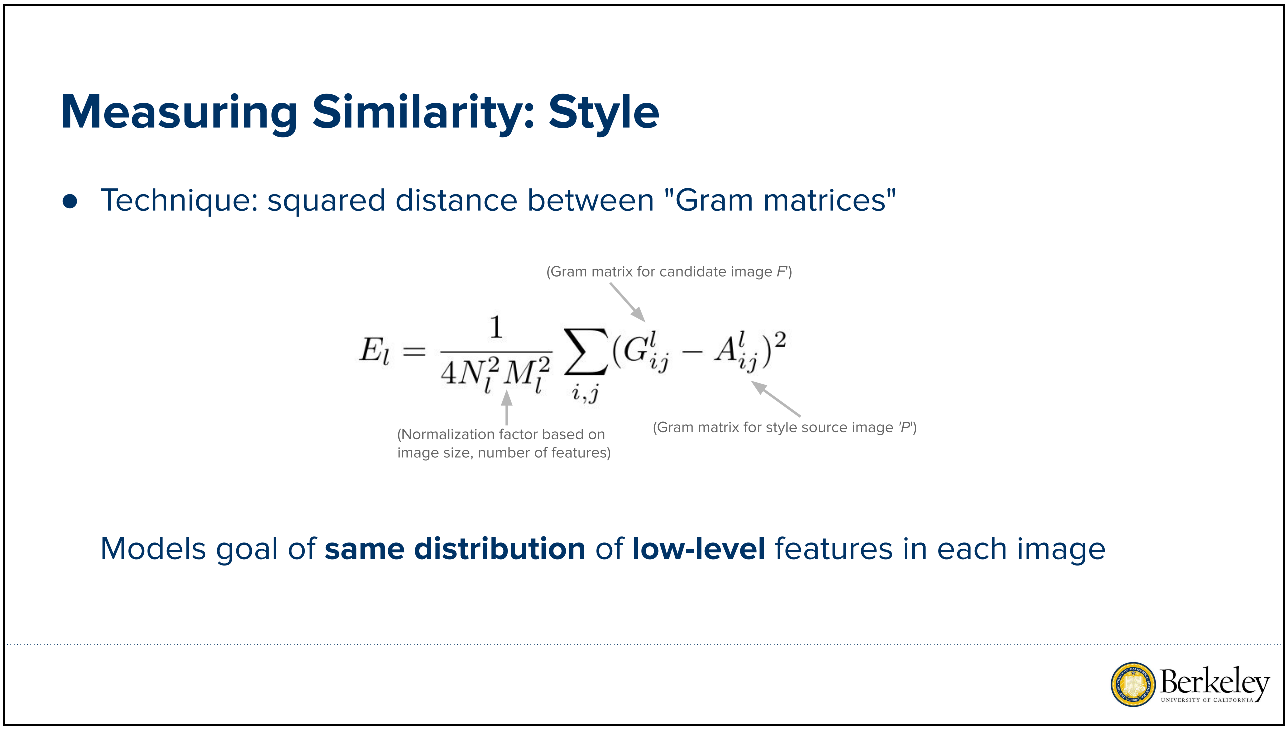


Measuring Similarity: Style

- First, compute "Gram matrices" of each image at each style layer

Now, how we measure similarity of style features? Unlike the content features, the style feature maps don't need to agree *at each position* in the image. Instead, we just want the *distribution* of all style features across both images to agree. To capture this, instead of directly comparing feature maps pixel-by-pixel, we compute something called a *Gram matrix* for the feature map of each image, which measures the prevalence of and relationship between different features in the feature map, but not where in the image those features occur. The Gram matrix thus gives kind of a statistical summary of all the features in an image, and which features tend to occur together with which other features.

Concretely, we implement this by taking pairwise dot products between feature maps: if we think of each feature map in a layer as a vector whose elements are its pixels values, laid out in some standard order, then each entry in the Gram matrix is the dot product between two feature maps in a layer.



Measuring Similarity: Style

- Technique: squared distance between "Gram matrices"

(Gram matrix for candidate image *F*')

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2$$

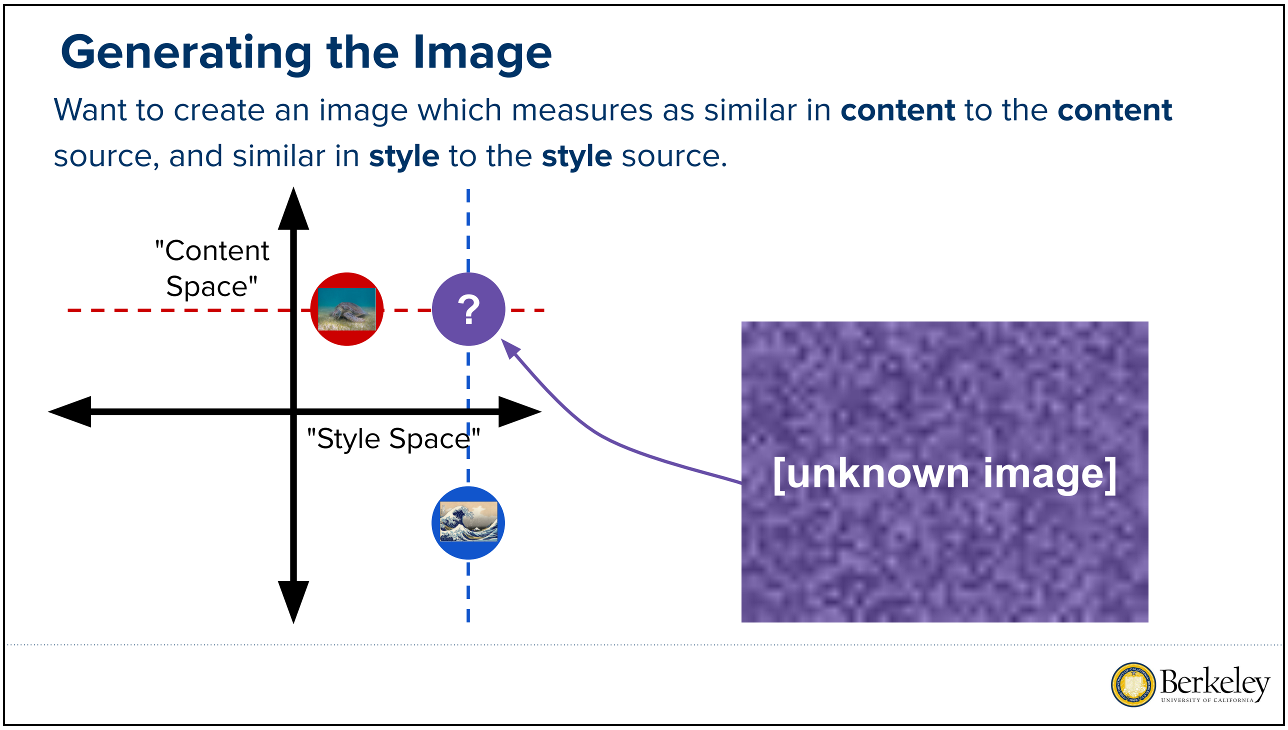
(Normalization factor based on image size, number of features)

(Gram matrix for style source image *'P'*)

Models goal of **same distribution** of **low-level** features in each image

Berkeley
UNIVERSITY OF CALIFORNIA

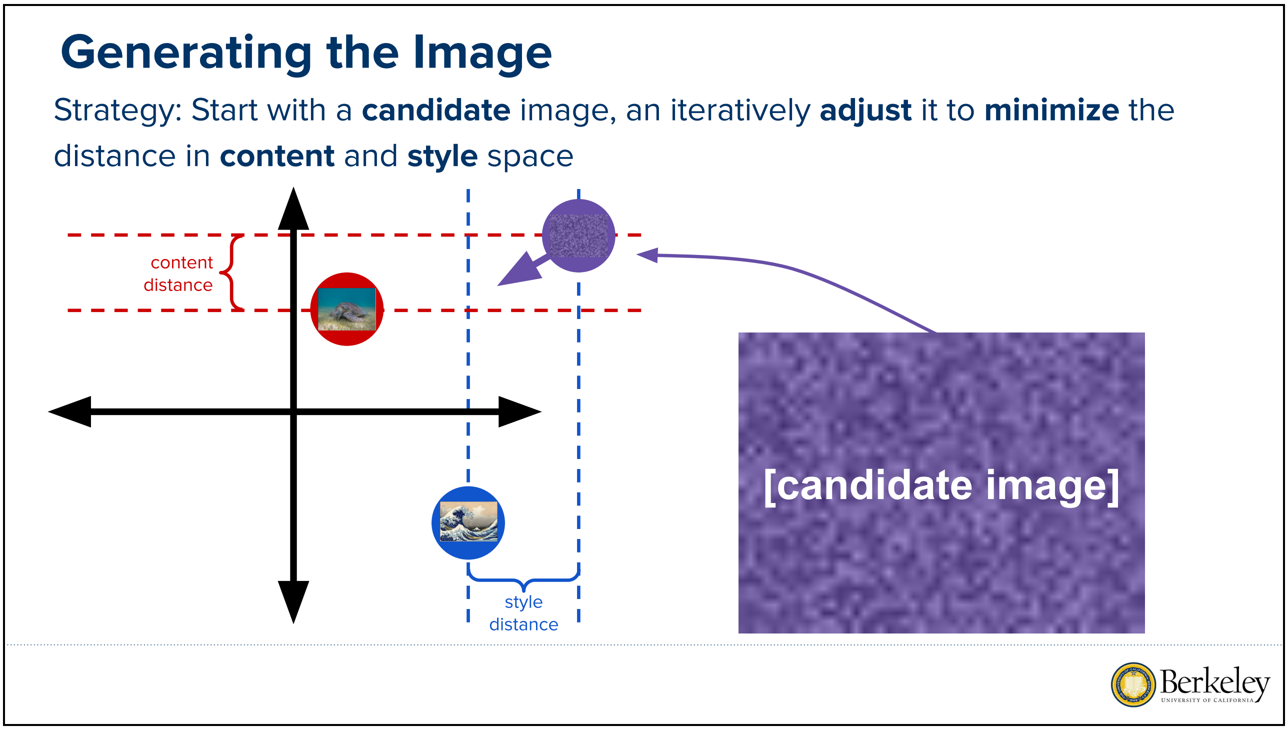
Once we've computed the Gram matrix for each image at each style layer — remember, we choose the "style layers" to be the early layers of the network, which we hope capture low-level visual information — we then measure style similarity by taking the squared Euclidean distance between the two images' Gram matrices at that layer. Again, what's useful about this measure of distance is that it doesn't depend directly on *where* different features appear in the image — it just measures the extent to which each image has a similar *distribution* of style features across all its pixels.



Generating the Image

Want to create an image which measures as similar in **content** to the **content** source, and similar in **style** to the **style** source.

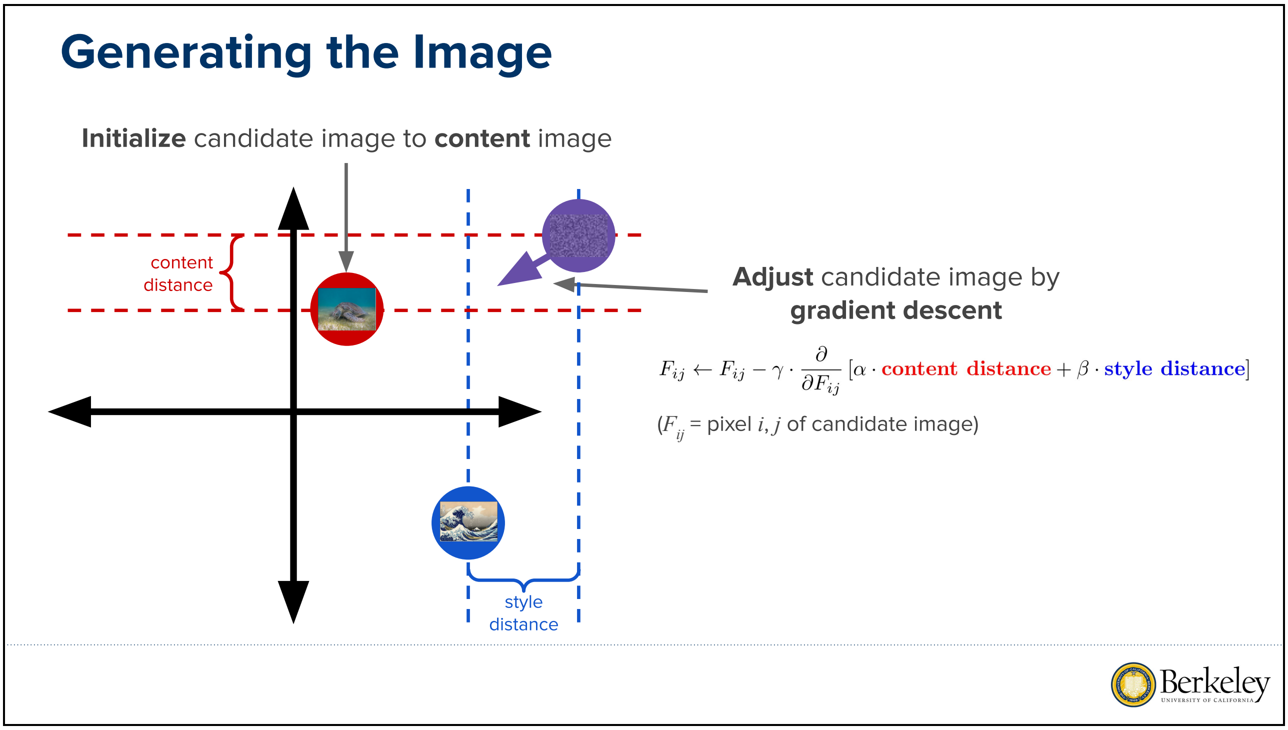
So, we now have a way to extract the content and style features an image, and we have a way of measuring the style or content similarity (or equivalently, style or content distance) to each image. We want an image which achieves low content-distance to the content source, and low style-distance to the style source. But we can't just pull such an image out of thin air — we need to somehow find some pixel values which *map* to the right point in style space, and the right point in content space. How do we find these pixels?



Generating the Image

Strategy: Start with a **candidate** image, an iteratively **adjust** it to **minimize** the distance in **content** and **style** space

The trick is iterative optimization. Suppose we have some candidate image which isn't very good yet — it's too far from the content image in content space, or from the style image in style space. If we can measure the content- and style-distances of this candidate image, then we can make small *adjustments* to each of the candidate image's pixel values to make the content and style distances smaller, basically moving in the direction of the content source image in content space and the style source image in style space. If we iterate this process, gradually moving in the right direction, the final version of the candidate image should achieve low content and style distance.



Generating the Image

Initialize candidate image to **content** image

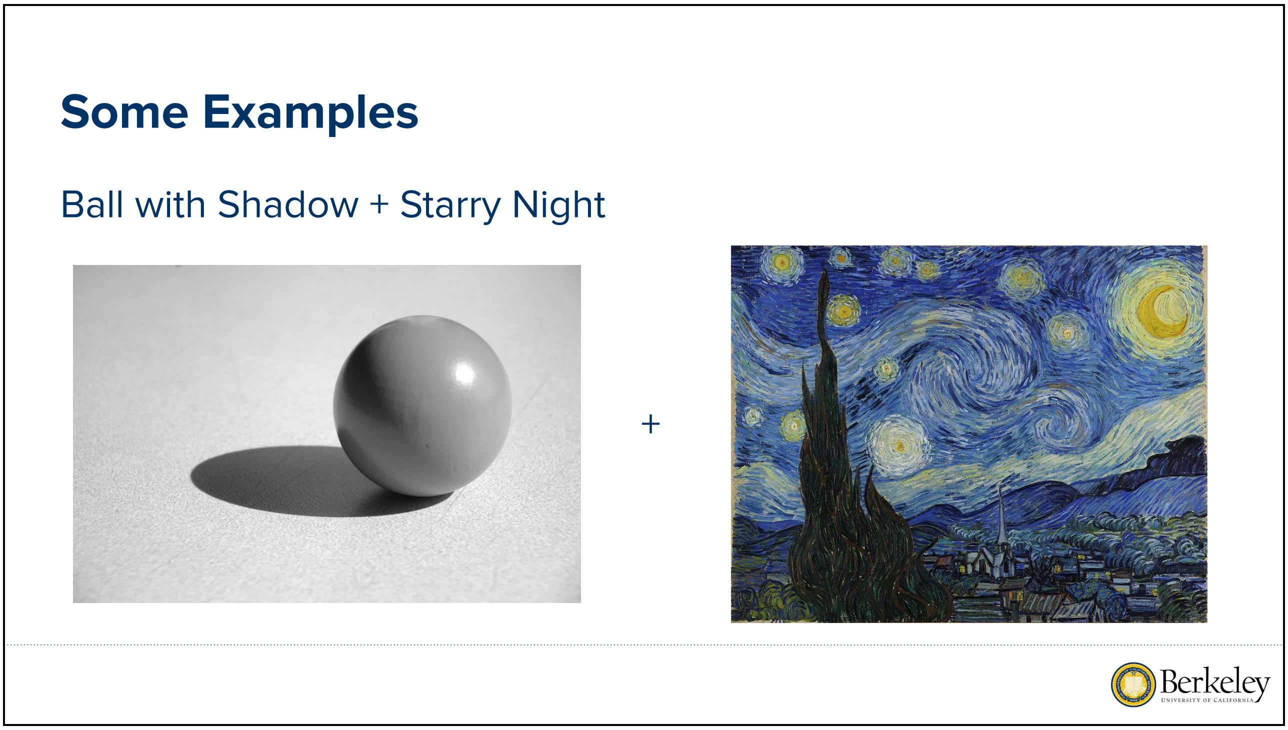
Adjust candidate image by
gradient descent

$$F_{ij} \leftarrow F_{ij} - \gamma \cdot \frac{\partial}{\partial F_{ij}} [\alpha \cdot \textbf{content distance} + \beta \cdot \textbf{style distance}]$$

(F_{ij} = pixel i, j of candidate image)

Concretely, the iterative optimization algorithm we just described works as follows. At the beginning, we initialize our candidate image to just our content image, without any style information applied yet. This will have zero distance in content-space to the content image, but potentially a large distance in style space to the style image. Then, we iteratively update the pixels of the style image using *gradient descent*: you take the gradient of some linear combination of the content and style distances with respect to the pixels of the candidate image, and then on each time step you subtract some multiple of this gradient from the candidate image. We subtract because we want to move in a direction where the distances will *decrease*. This moves us gradually closer in style-space to the style source, while ensuring we don't stray too far in content-space from the content source.

With a modern machine learning framework, we don't need to worry too much about how to compute the gradients; we can just specify the distance functions, and the framework does the rest. In practice, a machine learning framework might also provide a slightly more advanced gradient-based optimization method than the "naive" gradient descent algorithm described here, but the concept is the same.



Some Examples

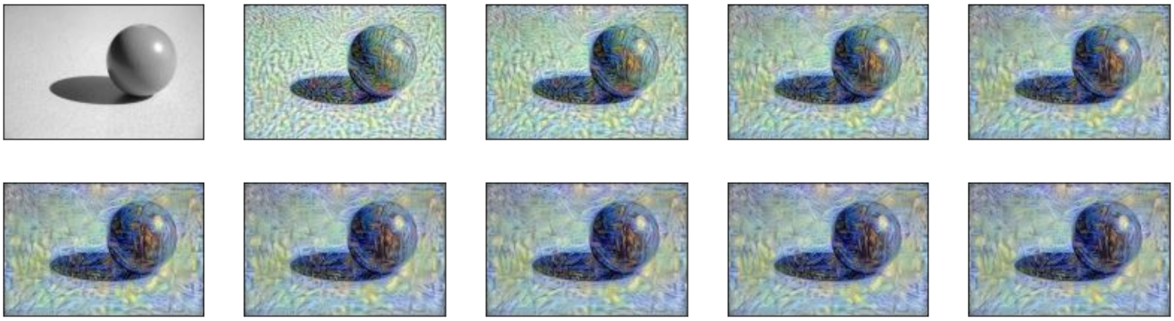
Ball with Shadow + Starry Night

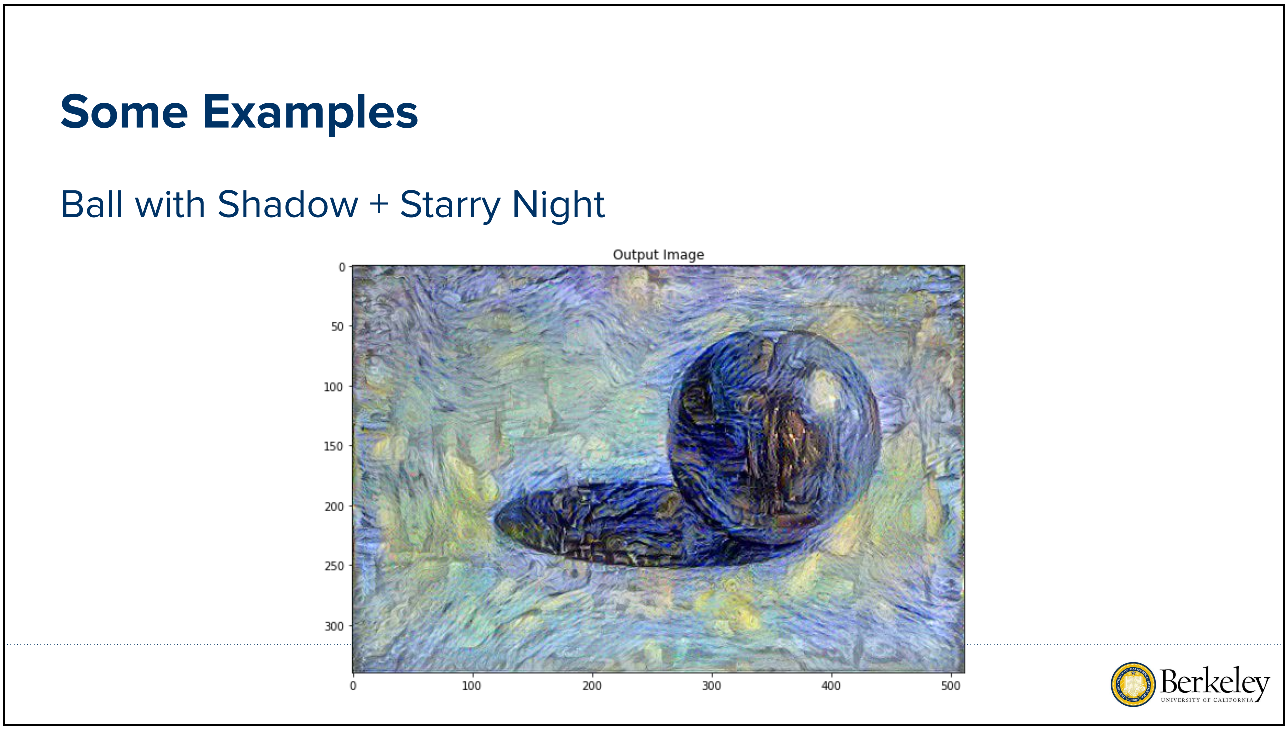
+

Now we will look at some examples! Why don’t we try something simple like a ball as our content image

Some Examples

Ball with Shadow + Starry Night

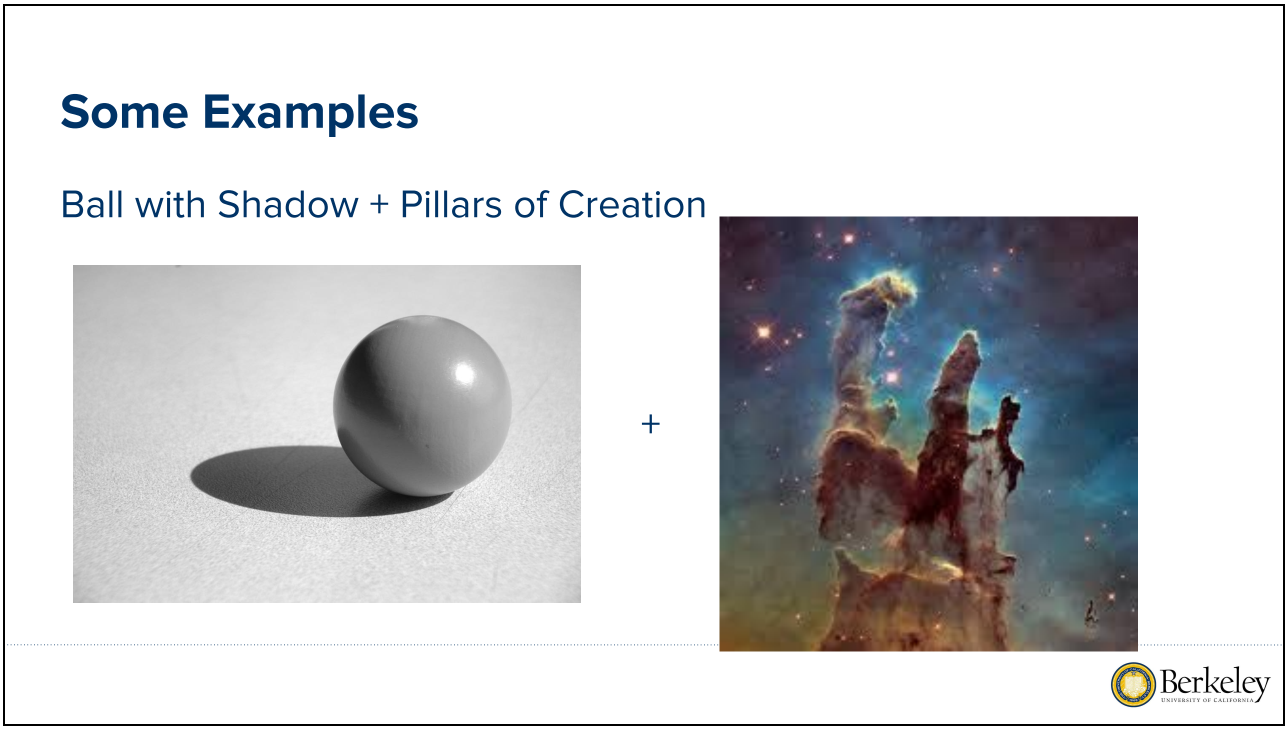




Some Examples

Ball with Shadow + Starry Night

Looks pretty cool!



Some Examples

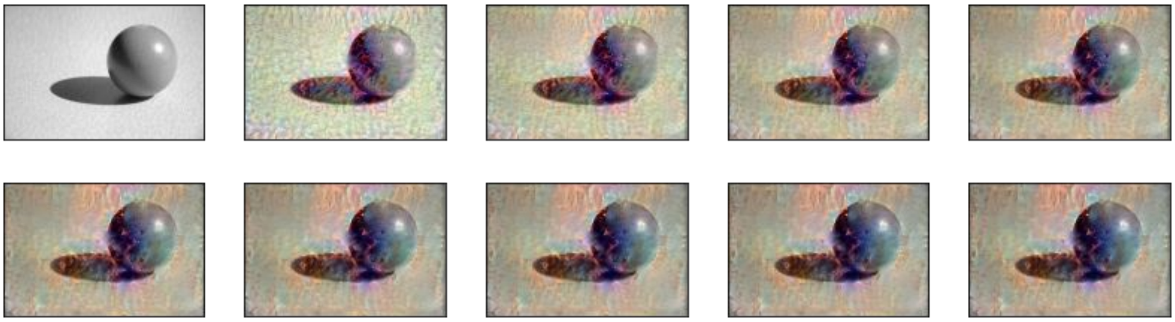
Ball with Shadow + Pillars of Creation

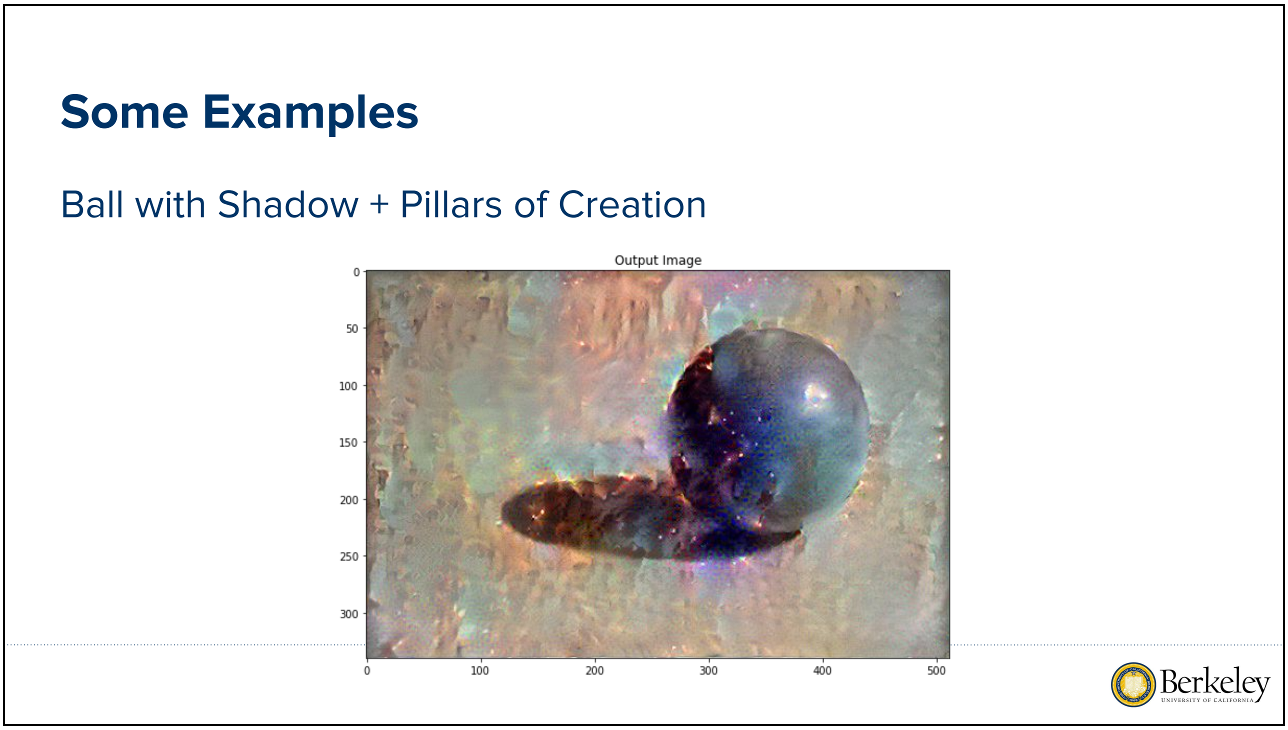
+

Now lets try the transfer with a picture of the Eagle Nebula

Some Examples

Ball with Shadow + Pillars of Creation



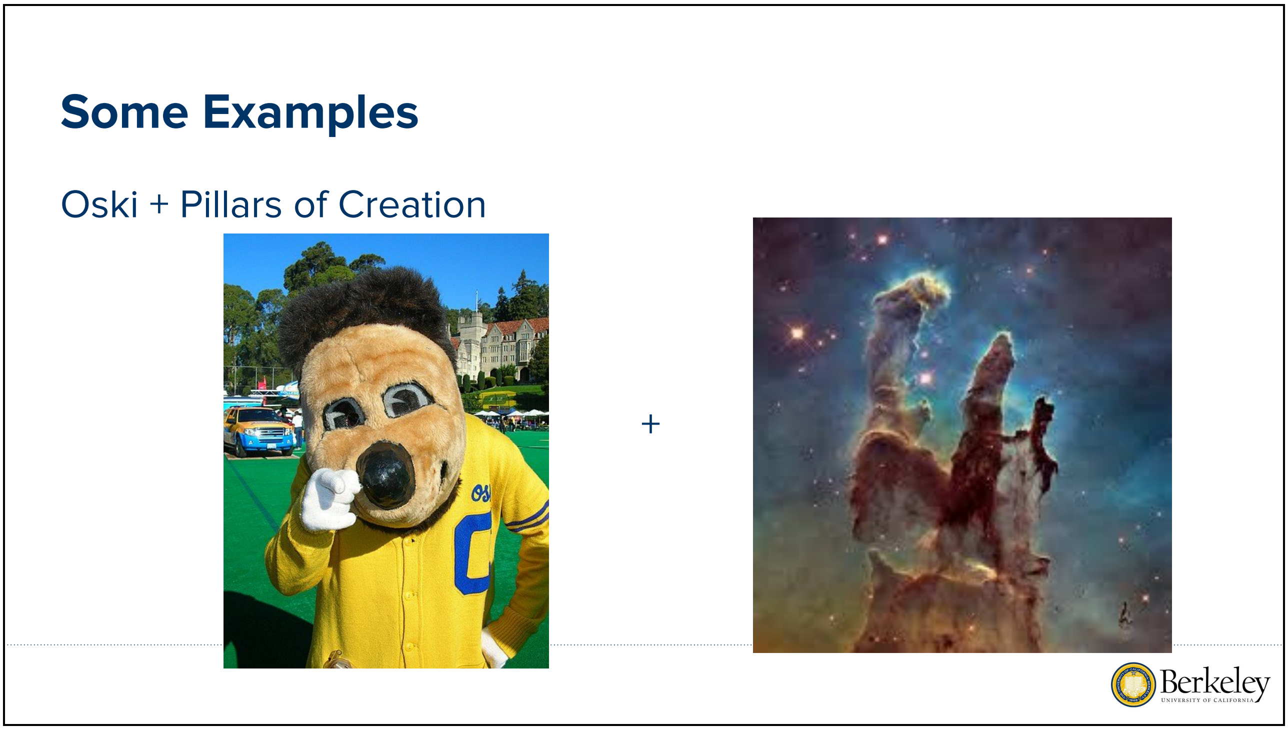


Some Examples

Ball with Shadow + Pillars of Creation

Neat! This worked pretty well.

Can anyone guess when the algorithm doesn’t work so well?



Some Examples

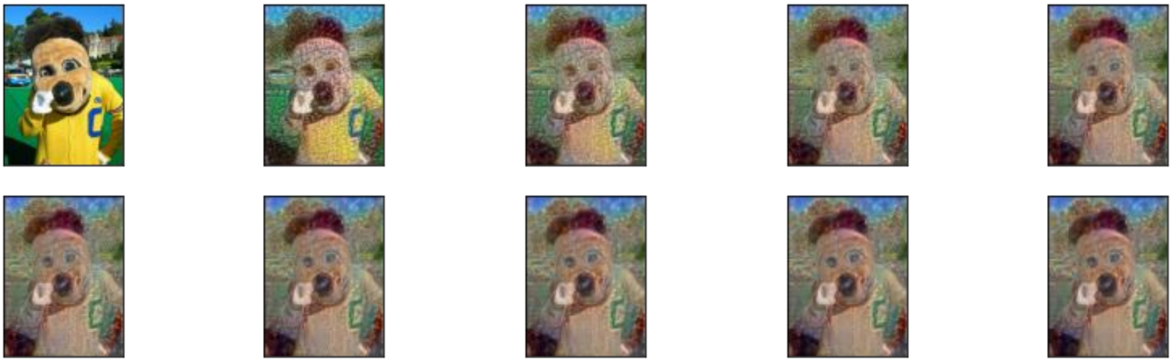
Oski + Pillars of Creation

+

Now lets try the transfer with a picture of Oksi and the Eagle Nebula

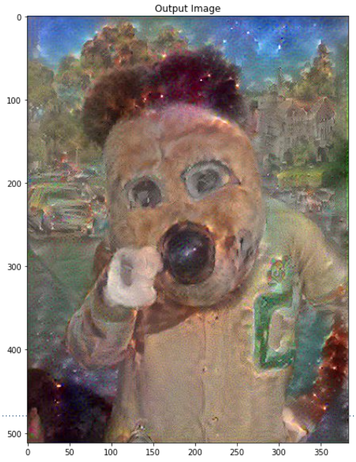
Some Examples

Oski + Pillars of Creation



Some Examples

Oski + Pillars of Creation



This did not work as well.

Can anyone guess why the algorithm didn’t work too well?

It might be because the style of the Pillars image was not easy to pick up or that the VGG-Network is not good at separating a background and the subject in the content.

References

1. Gatys, L. A., Ecker, A. S., Bethge, M. et al. "A neural algorithm of artistic style." arXiv preprint, 2015, https://arxiv.org/abs/1508.06576.
2. Simonyan, K., Zisserman, A. "Very Deep Convolutional Networks for Large-Scale Image Recognition." arXiv preprint, 2015, https://arxiv.org/pdf/1409.1556.
3. Yuan, R. "Neural Style Transfer: Creating Art with Deep Learning using tf.keras and eager execution." Medium, 3, Aug. 2018, https://medium.com/tensorflow/neural-style-transfer-creating-art-with-deep-learning-using-tf-keras-and-eager-execution-7d541ac31398.
4. Neural style transfer." TensorFlow.org, 2020, https://www.tensorflow.org/tutorials/generative/style_transfer.