

# Lecture 12

## Generative Adversarial Networks, GANs

CSCI E-89 Deep Learning , Fall 2024

Zoran B. Djordjević

# References

This lecture follows material presented in:

- *Image Style Transfer Using CNNs*, Leon A. Gatys, Alexander S. Ecker and Matthias Bethge, 2015, *IEEE Xplore*, (<https://arxiv.org/abs/1508.06576>)
- Chapter 12, *Deep Learning with Python 2<sup>nd</sup> Ed.* by François Chollet, O'Reilly 2021 , and Jupyter notebooks associated with that chapter.
- *Chapter 4, Generative Deep Learning, 2<sup>nd</sup> Ed.* by David Foster, O'Reilly, 2023

*There are excellent materials on the subject in the following texts:*

- *NIPS 2016 Tutorial: Generative Adversarial Networks*, Ian Goodfellow, 2017 (<https://arxiv.org/abs/1701.00160>)
- Chapter 8, *Deep Learning with Python 1<sup>st</sup> Ed.* by François Chollet, O'Reilly 2017 , and Jupyter notebooks associated with that chapter.
- Chapter 17, *Hands on Machine Learning with Scikit-learn, Keras & TensorFlow, 2<sup>nd</sup> Edition*, by Aurélien Géron, O'Reilly 2019
- *Generative Adversarial Networks*, Alec Radford, Luke Metz, Soumith Chintala, 2016 (<https://arxiv.org/pdf/1511.06434.pdf>)
- *Deep Learning* by Ian Goodfellow, Yoshua Bengio & Aaron Courville, MIT Press, 2017. This book has a public web site with the entire text available: [https://www.deeplearningbook.org/lecture\\_slides.html](https://www.deeplearningbook.org/lecture_slides.html)
- Experiments described in this lecture were done using Tensorflow-GPU 2.10 and Python 3.9.7

# Difference from what we did so far

- Generative models are distinct from either supervised or unsupervised models.
- These are the characteristic of those two traditional types of models:
- **Supervised Learning**
  - **Data:**  $(x, y)$
  - $x$  is data,  $y$  is label
  - **Goal:** Learn a *function* to map  $x \rightarrow y$
  - **Examples:** Classification, regression,
    - object detection, semantic
    - segmentation, image captioning, etc.
- **Unsupervised Learning**
  - **Data:**  $x$
  - Just data, no labels!
  - **Goal:** Learn some underlying hidden *structure* of the data
  - **Examples:** Clustering, dimensionality reduction, feature learning, density estimation, etc.
- The generative models create  $y$ -s (labels) like unsupervised models do. That is perhaps a reason some people categorize generative models under the unsupervised type. Generative networks train models that can produce multiple different correct answers and we can not minimize network parameters with respect to one label.
- Generative Models are many
  - PixelRNN, PixelCNN, StyleGANa, DCGANs, CycleGANs, etc
  - Variational Autoencoders (VAE)

# Neural Style Transfer

# Neural Style Transfer

- Neural Style Transfer is generative technique different from GANs.
- We would like to have ability to apply style of one image or painting to another image or painting.
- Transferring the style from one image onto another can be considered a problem of texture transfer. In texture transfer, the goal is to extract the texture from a source image, constrain the texture regeneration in the target image, and preserve the semantic content of the target image.
- Texture synthesis is a task older than Deep Learning. Deep learning version is called *Neural style transfer* and was introduced by *Leon Gatys et al.* in 2015.  
(<https://arxiv.org/abs/1508.06576>)
- Neural style transfer consists of applying the style of a reference image to a target image while conserving the content of the target image. Figure below shows an example.



# Separation of Content from Style

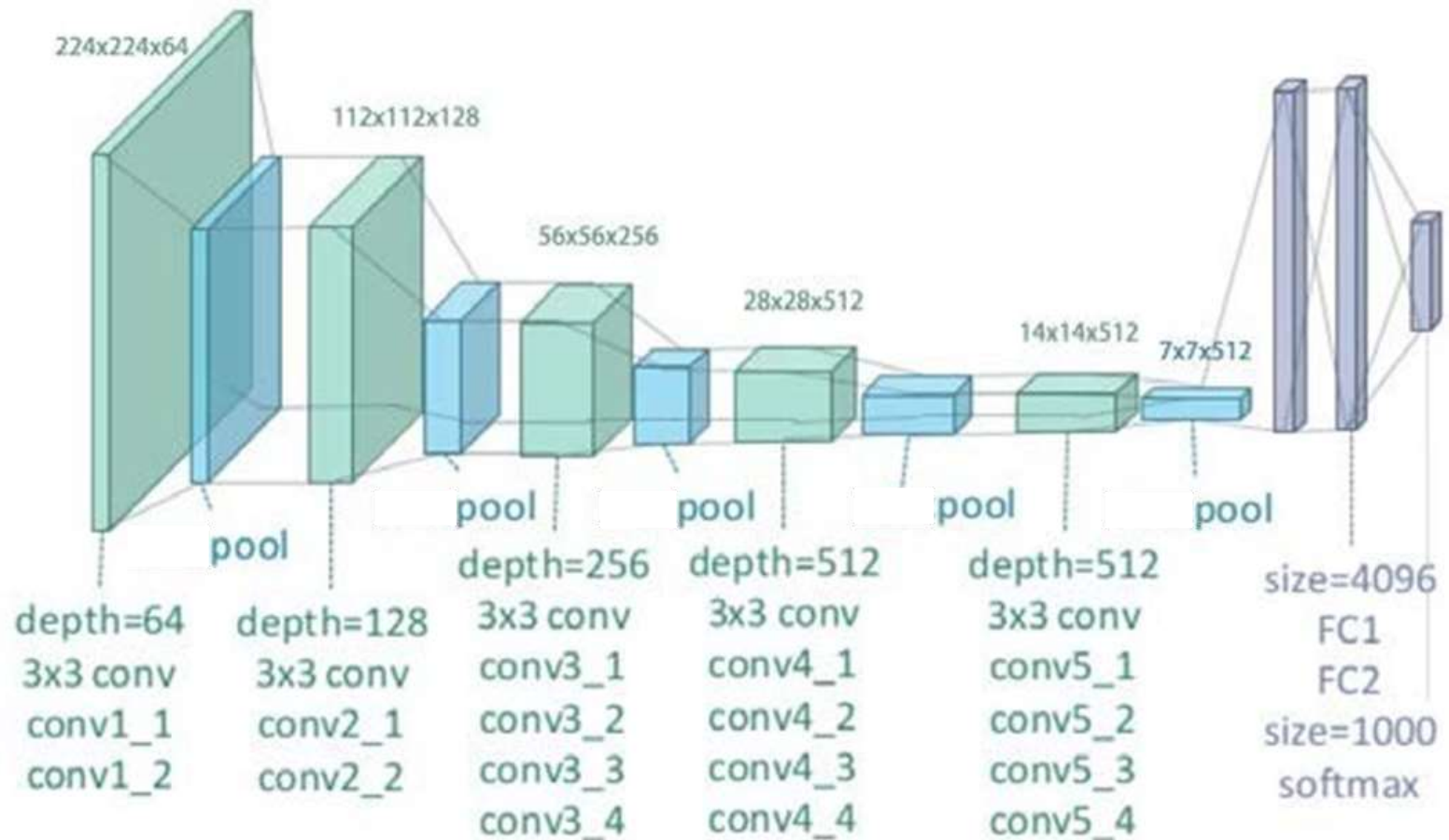
- Separation of content from style in natural images is a difficult problem.
- Deep CNNs could extract high-level semantic information from natural images.
- CNNs trained with sufficient labeled data on specific tasks such as object recognition learn to extract high-level generic features representing image contents. Those features apparently generalize across datasets and even to other visual information processing tasks, including texture recognition and artistic style classification.
- Gatys et al. used the generic feature representations learned by CNNs to independently process and manipulate the content and the style of natural images.
- The key finding of this paper is that the representations of content and style in the Convolutional Neural Network are separable. That is, one can manipulate both representations independently to produce new, perceptually meaningful images.
- To demonstrate this finding, they generate images that mix the content and style representation from two different source images.
- In particular, they match the content representation of a photograph depicting the “Neckarfront” in Tübingen, Germany and the style representations of several well-known artworks taken from different periods of art.
- The images are synthesized by finding an image that simultaneously matches the content representation of the photograph and the style representation of the respective piece of art.
- Effectively, this renders the photograph in the style of the artwork, such that the appearance of the synthesized image resembles the work of art, even though it shows the same content as the photograph.

## Network Used: VGG-19

- They normalized the network by scaling the weights so that the mean activation of each convolutional filter over images and positions is equal to one. Such re-scaling can be done for the VGG network without changing its output, because it contains only rectifying linear activation functions and no normalization or pooling over feature maps. They did not use any of the fully connected layers.
- For image synthesis they found that replacing the maximum pooling operation by average pooling yields slightly more appealing results, which is why the images shown were generated with average pooling.

# Architecture of VGG-19

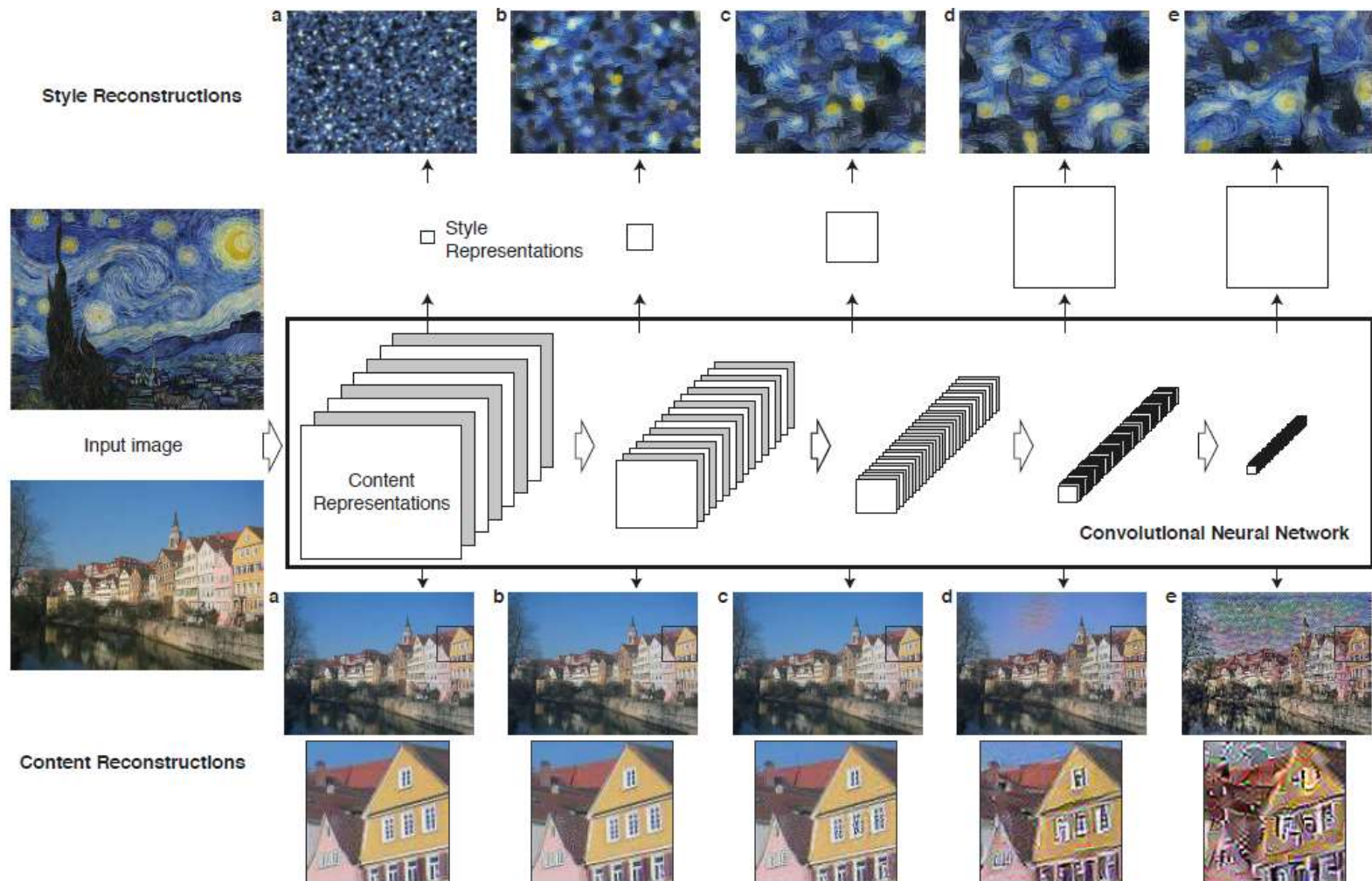
- Gatys et al. used the feature space provided by a normalized version of the 16 convolutional and 5 pooling layers of the 19-layer VGG network.





# Image Representation in CNN

- A given input image is represented as a set of filtered images at each processing stage in the CNN. While the number of different filters increases along the processing hierarchy, the size of the filtered images is reduced by some down-sampling mechanism (e.g., avg-pooling) leading to a decrease in the total number of units per layer of the network.



# The Content Loss

- The target image is the image which provides the content.
- The reference image is the image that supplies the style.
- The generated image is the one which combines the content and the style.
- Let  $A_i(\vec{X}; \vec{W})$  be the outputs of the  $i$ -th layer in the network.
- Also, let  $\vec{X}_{target}$  be the target image, and  $\vec{X}_{ref}$  the reference image.
- The deeper layers of neural networks capture the content rather well. Therefore, it is natural to enforce content similarity by making the outputs of one of deeper layers of the base image and to be close to the same layer of the generated image. This defines the content part of the loss function:

$$L_{content}(\vec{X}) = \|A_i(\vec{X}; \vec{W}) - A_i(\vec{X}_{target}; \vec{W})\|^2$$

# The style loss function

- Style is a bit more complicated to capture. We may still make advantage of the layer outputs, but style corresponds less to content; more to correlation between parts of the content.
- Correlations can be reasonably approximated using a Gram matrix (an outer product of the features with themselves). For features  $x$  and  $y$  of the  $i$ -th layer:

$$G_i(\vec{X}; \vec{W})_{xy} = A_i(\vec{X}; \vec{W})_x \otimes A_i(\vec{X}; \vec{W})_y$$

- Typically, we want to match style at multiple scales  $j$ . Therefore:

$$L_{style}(\vec{X}) = \|G_j(\vec{X}; \vec{W}) - G_j(\vec{X}_{ref}; \vec{W})\|^2$$

# The continuity loss function

- There is another desirable property of the input—we do not want the generated image to be too “jumpy” (i.e., changing color intensities too much with adjacent pixels).
- This can be enforced with a continuity loss defined as:

- $$L_{continuity}(\vec{X}) = \sum_{i=1}^n \sum_{j=2}^m (X_{ij} - X_{i,j-1})^2 + \sum_{i=2}^n \sum_{j=1}^m (X_{ij} - X_{i-1,j})^2$$

# Combining the losses

- As expected, we can combine the losses:

$$L_{styletrans}(\vec{X}) = \beta L_{content}(\vec{X}) + \alpha L_{style}(\vec{X}) + L_{continuity}(\vec{X})$$

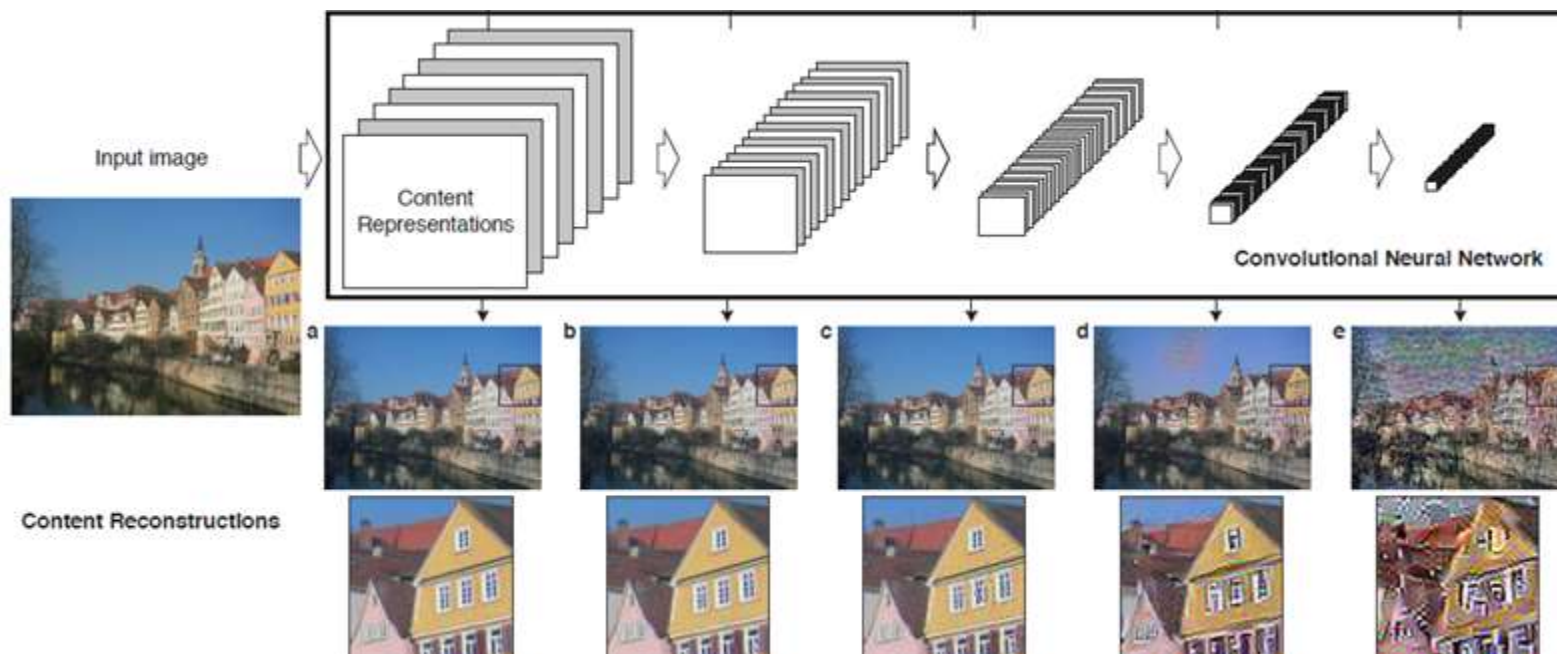
- Correctly choosing  $\alpha$  and  $\beta$  is important and requires quite some experimentation. Different values give different results, many of which are useless.

# Content representation

- Generally, each layer in the network defines a non-linear filter bank whose complexity increases with the position of the layer in the network. Hence a given input image  $\vec{x}$  is encoded in each layer of the CNN by the filter responses to that image. A layer with  $N_l$  distinct filters has  $N_l$  feature maps each of size  $M_l$ .  $M_l$  is equal to the product of the height and the width of the feature map.
- To visualize the image information encoded at different layers of the hierarchy, one can perform gradient descent on a white noise image to find another image that matches the feature responses of the original image.
- When CNNs are trained for object recognition, they develop a representation of the image that makes object information increasingly explicit along the processing hierarchy. Therefore, along the processing hierarchy of the network, the input image is transformed into representations that are increasingly sensitive to the actual *content* of the image but become relatively invariant to its precise appearance.
- Thus, *higher layers in the network capture the high-level content in terms of objects and their arrangement in the input image* but do not constrain the exact pixel values of the reconstruction very much.
- In contrast, reconstructions from the lower layers simply reproduce the exact pixel values of the original image.
- Therefore, *we refer to the feature responses in higher layers of the network as the content representation*. (Gatys et al. 2015)

# Content Reconstruction in CNN

- **Content Reconstructions:** We can visualize the information at different processing stages of a CNN by reconstructing the input image from only knowing the network's responses in a particular layer.
- We consider reconstructing the input image from layers: 'conv1 2' (a), 'conv2 2' (b), 'conv3 2' (c), 'conv4 2' (d) and 'conv5 2' (e) of the original VGG-Network. We find that reconstruction from lower layers is almost perfect (a–c). In higher layers of the network, detailed pixel information is lost while the high-level content of the image is preserved (d,e). (from Gatys et al. 2015)



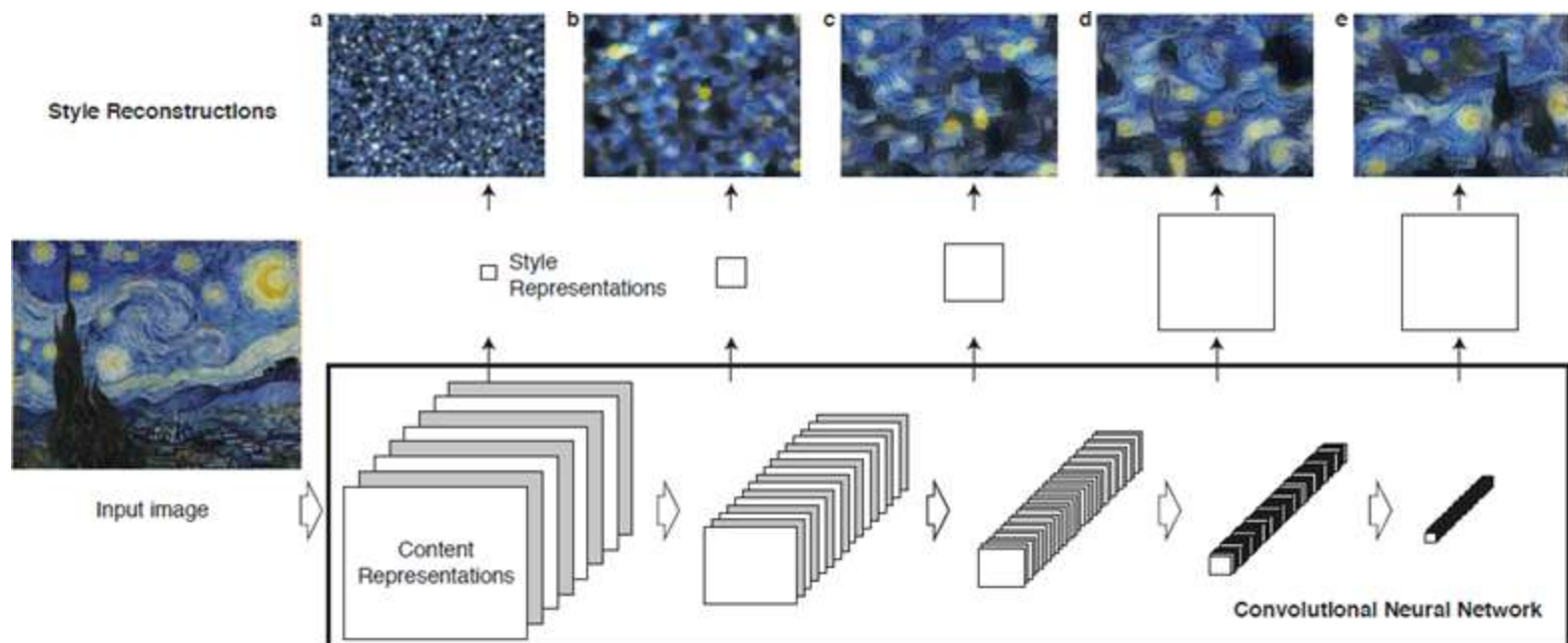
# Style representation

- To obtain a representation of the *style* of an input image, Gatys et al. used a feature space designed to capture texture information.
- This feature space can be built on top of the filter responses in any layer of the network. It consists of the correlations between the different filter responses, where the expectation is taken over the spatial extent of the feature maps.
- *The feature correlations of multiple layers is a stationary, multi-scale representation of the input image, which captures its texture information but not the global arrangement.*
- Again, they could visualize the information captured by these style feature spaces built on different layers of the network by constructing an image that matches the style representation of a given input image.



# Style Reconstruction, from Gatys et al.

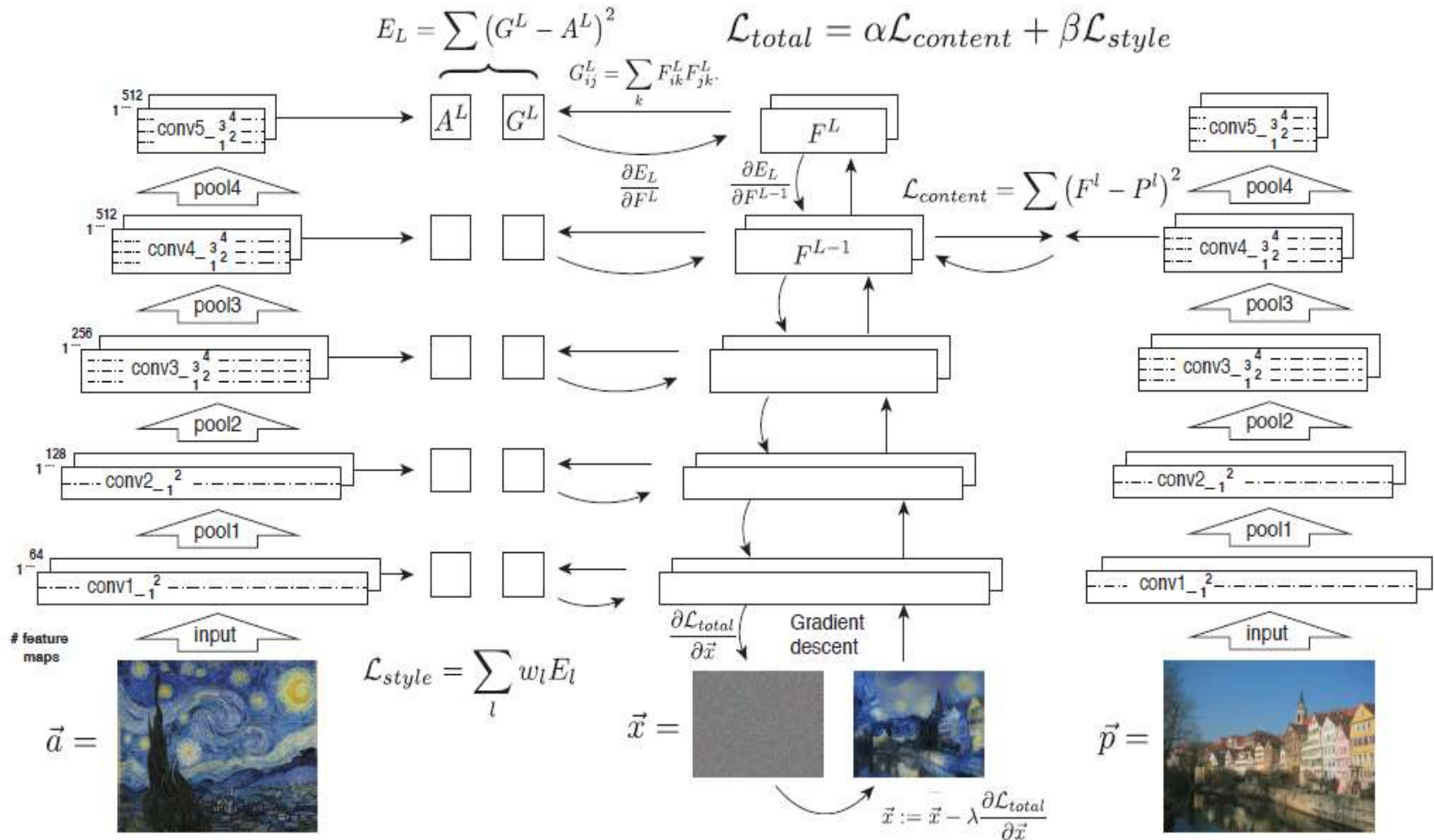
- **Style Reconstructions:** On top of the original CNN activations, Gatys et al. 2015 use a feature space that captures the texture information of an input image. *The style representation computes correlations between the different features in different layers of the CNN.* They reconstruct the style of the input image from a style representation built on different subsets of CNN layers ( 'conv1 1' (a), 'conv1 1' and 'conv2 1' (b), 'conv1 1', 'conv2 1' and 'conv3 1' (c), 'conv1 1', 'conv2 1', 'conv3 1' and 'conv4 1' (d), 'conv1 1', 'conv2 1', 'conv3 1', 'conv4 1' and 'conv5 1' (e).
- This process creates images that match the style of a given image on an increasing scale while discarding information of the global arrangement of the scene.



# Style transfer algorithm

- To transfer the style of an artwork  $\vec{a}$  onto a photograph or another image  $\vec{p}$ , Gatys et al. synthesized a new image that simultaneously matches the content representation of  $\vec{p}$  and the style representation of  $\vec{a}$ .
- They jointly minimized the distance of the feature representations of a white noise image from the content representation of the photograph in one layer and the style representation of the painting on a number of layers of the CNN.
- First, the content and style features are extracted and stored. The style image  $\vec{a}$  is passed through the network and its style representation  $A^l$  on all included layers are computed and stored (left path on next slide).
- The content image  $\vec{p}$  is passed through the network and the content representation  $P^l$  in one layer is stored (right path on next slide). Then a random white noise image  $\vec{x}$  is passed through the network and its style features  $G^l$  and content features  $F^l$  are computed.
- On each layer included in the style representation, the element-wise mean squared difference between  $G^l$  and  $A^l$  is computed to give the style loss  $\mathcal{L}_{\text{style}}$  (left). Also the mean squared difference between  $F^l$  and  $P^l$  is computed to give the content loss  $\mathcal{L}_{\text{content}}$  (right).
- The total loss  $\mathcal{L}_{\text{total}}$  is then a linear combination between the content and the style loss.
- The derivatives of  $\mathcal{L}_{\text{total}}$  with respect to the pixel values can be computed using error back-propagation (middle). This gradient is used to iteratively update the image  $\vec{x}$  until it simultaneously matches the style features of the style image  $\vec{a}$  and the content features of the content image  $\vec{p}$  (middle, bottom).

# Style transfer algorithm



(Gatys et al. 2015)

# Styles of Artwork applied to Photograph of Tübingen, Germany

- The images were created by finding an image that simultaneously matches the content representation of the photograph and the style representation of the artwork.
- The original photograph depicting the bank of river Neckar, the Neckarfront, in Tübingen, Germany, is shown in A)
- The painting that provided the style for the generated image is shown in the bottom middle panel: C). *The Shipwreck of the Minotaur* by J.M.W. Turner, 1805.
- Resulting image with the content of A) and style of C) is image B)





# Images of Tübingen

- Below are two more images of Tübingen with styles borrowed from
- D. *Femme nue assise* by Pablo Picasso, 1910.
- E. *Composition VII* by Wassily Kandinsky, 1913.



- In what follows, we will apply above ideas in somewhat simplified but still working fashion.

# Loss Function

- Just like in all deep-learning algorithms: we define a loss function to specify what we want to achieve, and we minimize that loss function.
- We want to conserve the content of the original image while adopting the style of the reference image. If we were able to mathematically define *content* and *style*, then an appropriate loss function to minimize both would read:

```
loss = distance(style(style_reference_image) -  
                style(generated_image)) +  
distance(content(original__content_image) -  
         content(generated_image))
```

- Here, `distance()` is a norm function such as the L2 norm, `content()` is a function that takes an image and computes a representation of its content, and `style()` is a function that takes an image and computes a representation of its style.
- Minimizing this loss causes `style(generated_image)` to be close to `style(reference_image)`, and `content(generated_image)` is close to `content(original_image)`, thus achieving style transfer as we defined it.
- A fundamental observation made by *Gatys et al.* was that deep CNNs offer a way to mathematically define the `style` and `content` functions.

# The content loss

- Activations from earlier layers in a network contain *local* information about the image, whereas activations from higher layers contain increasingly *global*, *abstract* information.
- Formulated in a different way, the activations of the different layers of a CNN provide a decomposition of the contents of an image over different spatial scales. Therefore, *you'd expect the content of an image, which is more global and abstract, to be captured by the representations of the upper layers in a CNN.*
- A good candidate for content loss is thus the L2 norm between the activations of an upper layer in a pretrained CNN, computed over the target image, and the activations of the same layer computed over the generated image. This guarantees that, as seen from the upper layer, the generated image will look similar to the original target image.
- Assuming that what the upper layers of a CNN see is really the content of the input images, then this works as a way to preserve image content.

# Gram Matrix, Wikipedia.org

- For finite-dimensional real vectors in  $R^n$  with the usual Euclidean dot product, the Gram matrix is  $G = V^T V$ , where  $V$  is a matrix whose columns are the vectors  $v_k$  and  $V^T$ , its transpose, has as rows the vectors  $v_k^T$ .
- If the vectors are centered random variables, the Gramian is approximately proportional to the covariance matrix, with the scaling determined by the number of elements in the vector.
- In machine learning, [kernel functions](#) are often represented as Gram matrices. (Also see [kernel PCA](#))
- Since the Gram matrix over the reals is a symmetric matrix, it is diagonalizable and its eigenvalues are non-negative. The diagonalization of the Gram matrix is the singular value decomposition.



# The Style loss

- The content loss only uses a single upper layer, but the style loss, as defined by Gatys et al., uses multiple layers of a CNN. You try to capture the appearance of the style reference image at all spatial scales extracted by the CNN, not just a single scale.
- For the style loss, Gatys et al. use the *Gram matrix* of a layer's activations: that is the inner product of the feature maps of a given layer. Gram matrix  $G^l \in R^{N_l \times N_l}$ , where  $G_{ij}^l$  is the inner product between the vectorized feature maps  $i$  and  $j$  in layer  $l$ :

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l$$

- This inner product can be understood as representing a map of the correlations between the layer's features. These feature correlations capture the statistics of the patterns of a particular spatial scale, which empirically correspond to the appearance of the textures found at this scale.
- Hence, the style loss aims to preserve similar internal correlations within the activations of different layers, across the style-reference image and the generated image. In turn, this guarantees that the textures found at different spatial scales look similar across the style-reference image and the generated image.
- We can use a pretrained CNN to define a loss that will do the following:
  - Preserve content by maintaining similar high-level layer activations between the target content image and the generated image. The CNN should “see” both the target image and the generated image as containing the same things.
  - Preserve style by maintaining similar *correlations* within activations for both low level layers and high-level layers. Feature correlations capture *textures* : the generated image and the style-reference image should share the same textures at different spatial scales.

# Neural Style Transfer in Keras

# Neural Style Transfer in Keras

- Neural style transfer can be implemented using any pretrained CNN. Here, we will use the VGG19 network used by Gatys et al.
- VGG19 is a simple variant of the VGG16 CNN network, with three more convolutional layers.

This is the process we will follow:

1. Set up a network that computes VGG19 layer activations for the style-reference image, the target image, and the generated image at the same time.
  2. Use the layer activations computed over these three images to define the loss function, which we will minimize in order to achieve style transfer.
  3. Set up a gradient-descent process to minimize this loss function.
- We will start by defining the paths to the style-reference image and the target image. To make sure that the processed images are a similar size (widely different sizes make style transfer more difficult), we will resize them all to a shared height of 400 px.

# Defining initial variables

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.applications import vgg19

base_image_path = keras.utils.get_file("paris.jpg",
    "https://i.imgur.com/F28w3Ac.jpg")
style_reference_image_path = keras.utils.get_file(
    "starry_night.jpg", "https://i.imgur.com/9ooB60I.jpg"
)
result_prefix = "paris_generated"

# Weights of the different loss components
total_variation_weight = 1e-6
style_weight = 1e-6
content_weight = 2.5e-8

# Dimensions of the generated picture.
width, height =
keras.preprocessing.image.load_img(base_image_path).size
img_nrows = 400
img_ncols = int(width * img_nrows / height)
```

# Base (content) and Style Reference Image

```
from IPython.display import Image, display
display(Image(base_image_path))
display(Image(style_reference_image_path))
```



# Image preprocessing / deprocessing utilities

- We will need some auxiliary functions for loading, pre-processing and post-processing the images that will go in and out of the VGG19 convent:

```
import numpy as np
from keras.applications import vgg19
def preprocess_image(image_path):
    img = load_img(image_path, target_size=(img_height, img_width))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = vgg19.preprocess_input(img)
    return img

def deprocess_image(x):
    # Remove zero-center by mean pixel
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    # 'BGR' -> 'RGB'
    x = x[:, :, ::-1]
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

# Compute the style transfer loss

- We need to define 4 utility functions:
  - **gram\_matrix** (used to compute the style loss)
  - **The style\_loss** function, which keeps the generated image close to the local textures of the style reference image

```
# The gram matrix of an image tensor (feature-wise outer product)
```

```
def gram_matrix(x):  
    x = tf.transpose(x, (2, 0, 1))  
    features = tf.reshape(x, (tf.shape(x)[0], -1))  
    gram = tf.matmul(features, tf.transpose(features))  
    return gram
```

```
# The "style loss" is designed to maintain  
# the style of the reference image in the generated image.  
# It is based on the gram matrices (which capture style) of  
# feature maps from the style reference image  
# and from the generated image
```

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

# Compute the style transfer loss

- We need to define 2 additional utility functions:
  - The `content_loss` function, which keeps the high-level representation of the generated image close to that of the base image
  - The `total_variation_loss` function, a regularization loss which keeps the generated image locally-coherent

```
# An auxiliary loss function
```

```
# designed to maintain the "content" of the
```

```
# base image in the generated image
```

```
def content_loss(base, combination):
```

```
    return tf.reduce_sum(tf.square(combination - base))
```

```
# The 3rd loss function, total variation loss,
```

```
# designed to keep the generated image locally coherent
```

```
def total_variation_loss(x):
```

```
    a = tf.square(
```

```
        x[:, : img_nrows - 1, : img_ncols - 1, :] - x[:, 1:, : img_ncols - 1, :]
```

```
)
```

```
    b = tf.square(
```

```
        x[:, : img_nrows - 1, : img_ncols - 1, :] - x[:, : img_nrows - 1, 1:, :]
```

```
)
```

```
    return tf.reduce_sum(tf.pow(a + b, 1.25))
```



# Load Network, three Images

- Next, let's create a feature extraction model that retrieves the intermediate activations of VGG19 (as a dict, by name).

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

```
# Get the symbolic outputs of each "key" layer (we gave them unique
names).
```

```
outputs_dict = dict([(layer.name, layer.output) for layer in
model.layers])
```

```
# Set up a model that returns the activation values for every layer in
# VGG19 (as a dict).
```

```
feature_extractor = keras.Model(inputs=model.inputs,
outputs=outputs_dict)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19\_weights\_tf\_dim\_ordering\_tf\_kernels\_notop.h5
80142336/80134624 [=====] - 2s 0us/step
```

# Compute Content and Style Transfer Loss

- Finally, here's the code that computes the style transfer loss.

```
# List of layers to use for the style loss.
style_layer_names = [
    "block1_conv1",
    "block2_conv1",
    "block3_conv1",
    "block4_conv1",
    "block5_conv1",
]
# The layer to use for the content loss.
content_layer_name = "block5_conv2"

def compute_loss(combination_image, base_image, style_reference_image):
    input_tensor = tf.concat(
        [base_image, style_reference_image, combination_image], axis=0
    )
    features = feature_extractor(input_tensor)
    # Initialize the loss
    loss = tf.zeros(shape=())
    # Add content loss
    layer_features = features[content_layer_name]
    base_image_features = layer_features[0, :, :, :]
    combination_features = layer_features[2, :, :, :]
    loss = loss + content_weight * content_loss(
        base_image_features, combination_features
    )
    # Add style loss
    for layer_name in style_layer_names:
        layer_features = features[layer_name]
        style_reference_features = layer_features[1, :, :, :]
        combination_features = layer_features[2, :, :, :]
        sl = style_loss(style_reference_features, combination_features)
        loss += (style_weight / len(style_layer_names)) * sl

    # Add total variation loss
    loss += total_variation_weight * total_variation_loss(combination_image)
    return loss
```

# Convert loss function to TF function

- We will add a `@tf.function` decorator to loss & gradient computation to compile it and thus make it much faster.

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

# The Training Loop

- Repeatedly run vanilla gradient descent steps to minimize the loss, and save the resulting image every 100 iterations.
- We decay the learning rate by 0.96 every 100 steps.

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

```
base_image = preprocess_image(base_image_path)  
style_reference_image = preprocess_image(style_reference_image_path)  
combination_image = tf.Variable(preprocess_image(base_image_path))
```

```
iterations = 4000  
for i in range(1, iterations + 1):  
    loss, grads = compute_loss_and_grads(  
        combination_image, base_image, style_reference_image  
    )  
    optimizer.apply_gradients([(grads, combination_image)])  
    if i % 100 == 0:  
        print("Iteration %d: loss=%.2f" % (i, loss))  
        img = deprocess_image(combination_image.numpy())  
        fname = result_prefix + "_at_iteration_%d.png" % i  
        keras.preprocessing.image.save_img(fname, img)
```

# Output

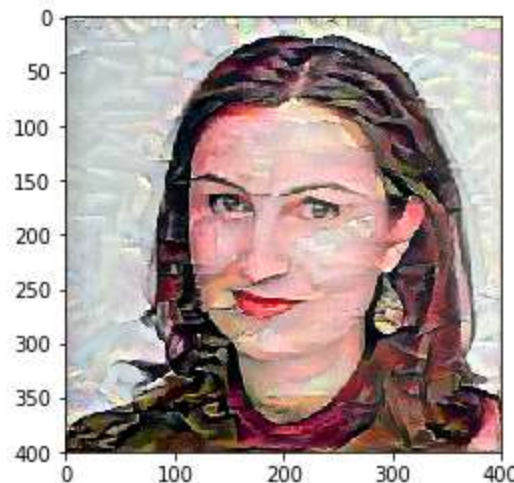
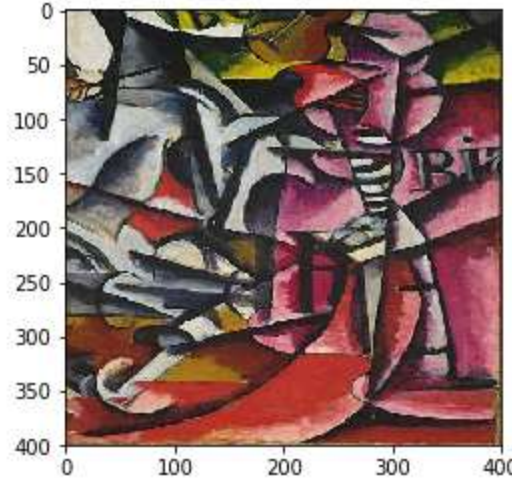
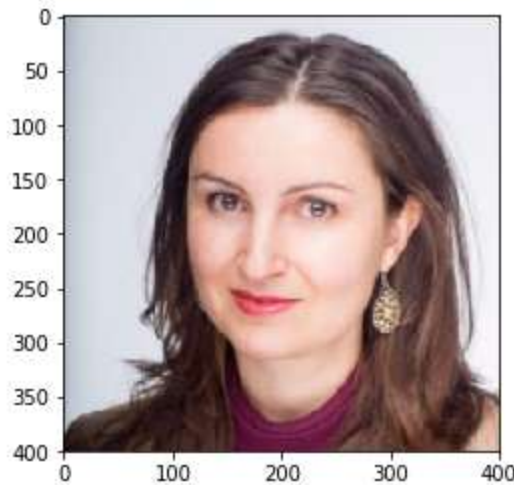
```
Iteration 100: loss=11024.46  
Iteration 200: loss=8519.11  
...  
Iteration 3900: loss=5199.79  
Iteration 4000: loss=5193.09
```

- Then we plot the result:

```
display(Image(result_prefix + "_at_iteration_4000.png"))
```



## Similar Results

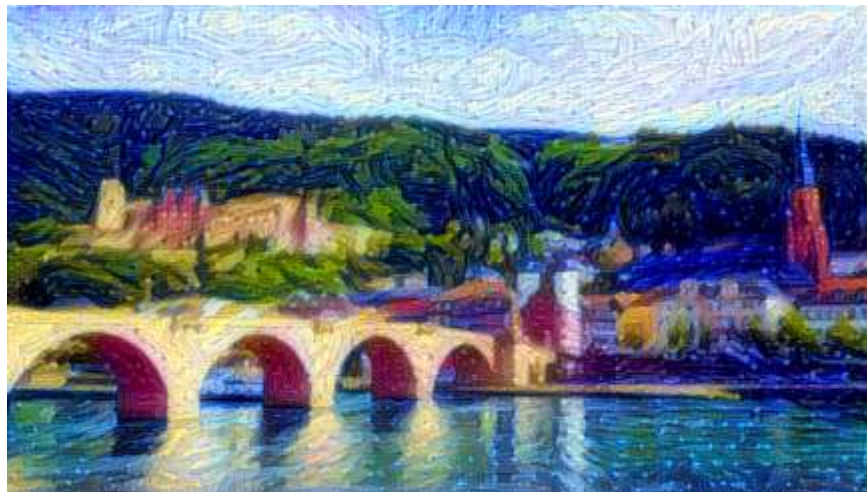


- Image on the left is a face from Google Images.
- The painting is by: Liubov Popova (1889-1924), a cubist painter.
- The image on the bottom is the previous portrait with some cubists style added to it. One could perhaps vary hyper-parameters of the model, those weights associated with different losses to strengthen or weaken the style.



# Heidelberg Experiment

- We ran a few more experiments with the same code. Below is the bridge in Heidelberg, Van Gogh's Starry Night and the modified image of Heidelberg with Van Gogh's style added to it. This also proves that the technique works with other university towns in Germany, and not only Tübingen.



# Generated Art

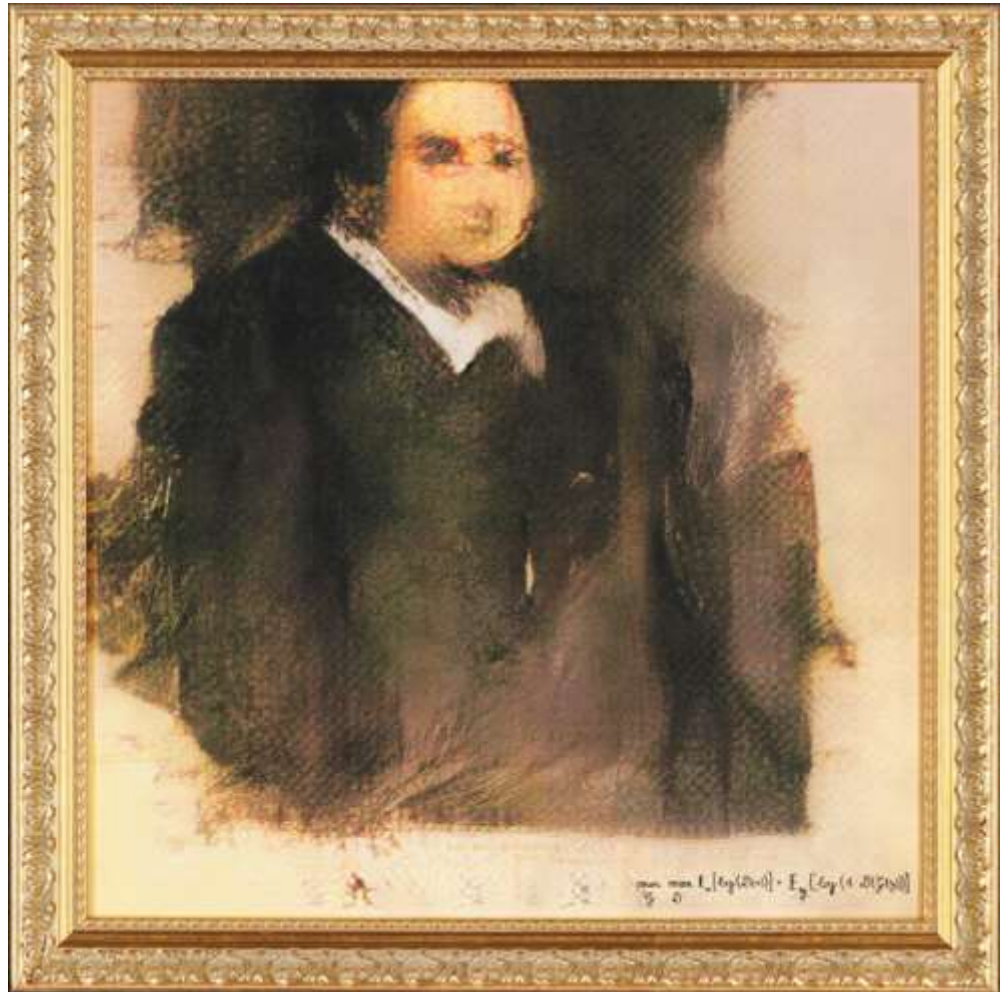


# AI Generated Art Sold at Christie's

- *Portrait of Edmond de Belamy, from La Famille de Belamy*, Price realized US \$432,500

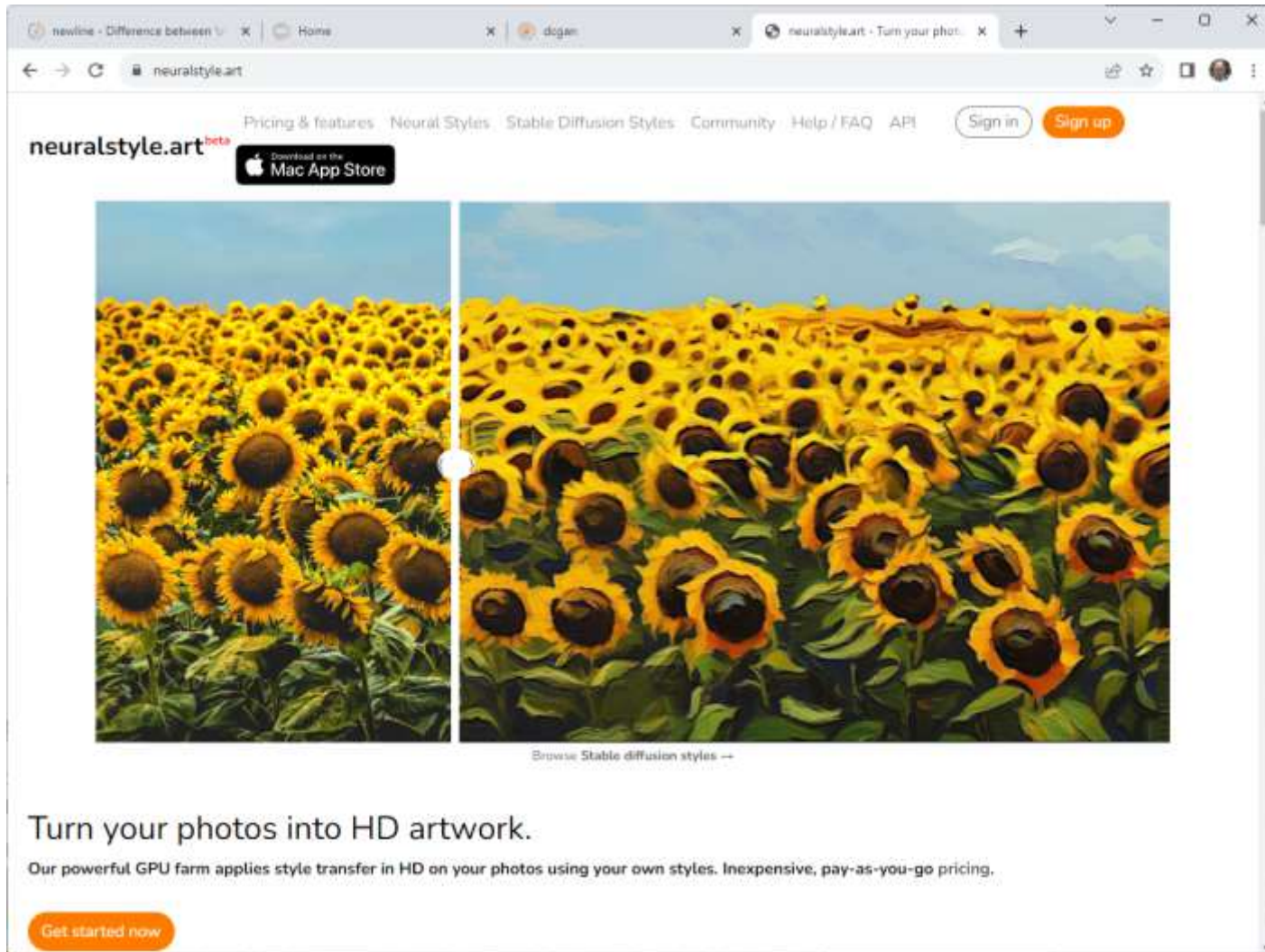
<https://www.christies.com/features/A-collaboration-between-two-artists-one-human-one-a-machine-9332-1.aspx>

Artist's signature:  $\min_G \max_D \mathbb{E}_x [\log(D(x))] + \mathbb{E}_z [\log(1 - D(G(z)))]$



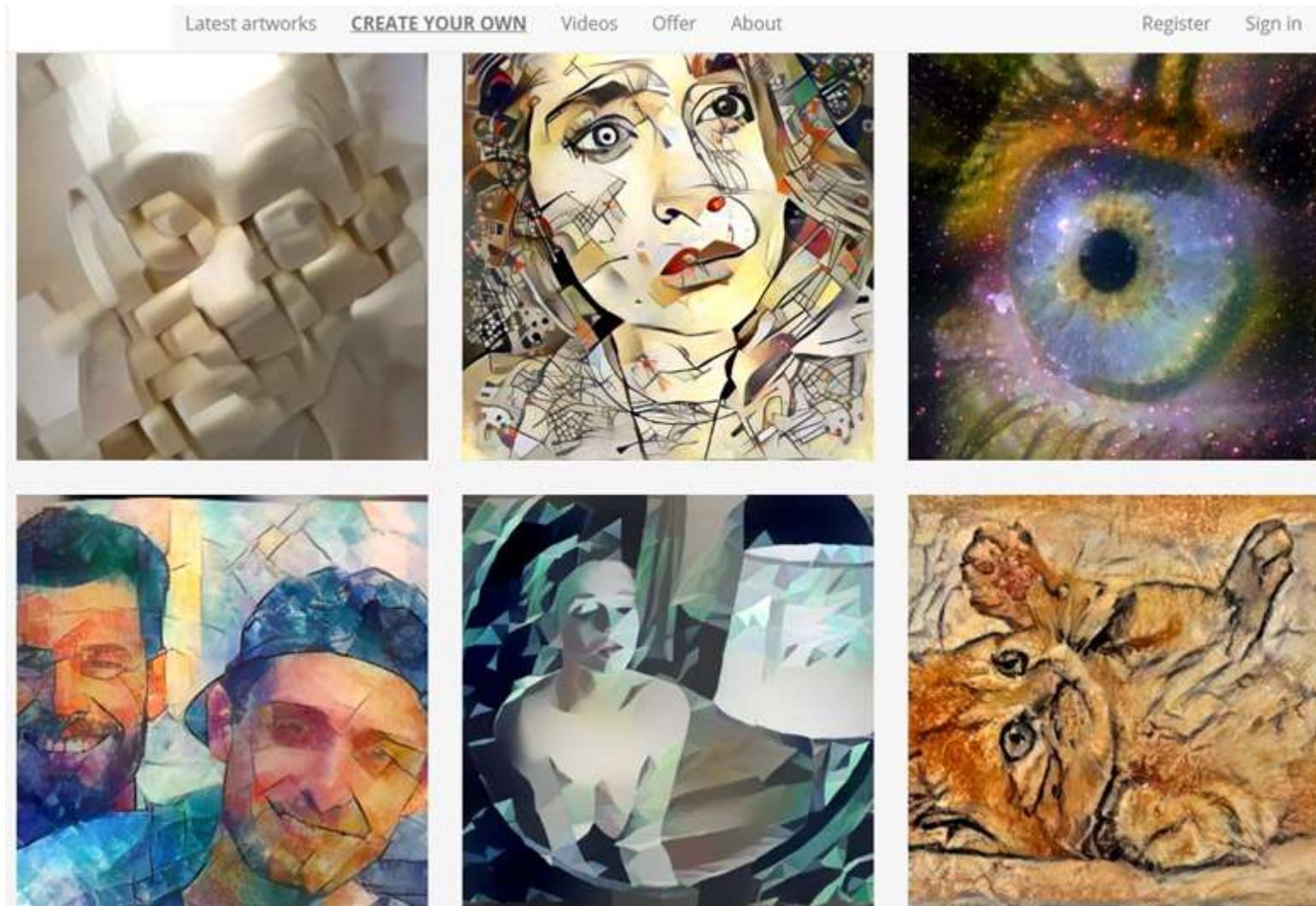
# Site where you can pay & practice Neural Style Transfer

- <https://neuralstyle.art/>





# Site where you can practice Neural Style Transfer



# Generative Models

# Generative Models

## *Single vs. Multi-output*

- In training traditional machine learning models, we are minimizing the difference between the single desired output and the model's predicted output.
- Generative models, and GANs in particular, usually work with multi-modal outputs.
- In many tasks, a single input may correspond to many different correct answers, each of which is acceptable.
- GANs train models that can produce multiple different correct answers and we can not minimize network parameters with respect to one label. One example of such scenario is predicting the next frame in a video.

## *Probabilistic interpretation*

- The term Generative Model refers to any model that takes a training set, consisting of samples drawn from a distribution  $p_{data}$ , and somehow learns to present an estimate of that distribution. The result is a probability distribution  $p_{model}$ .
- In some cases, the model estimates  $p_{model}$  explicitly. In other cases, the model is only able to generate samples from  $p_{model}$ .
- The most popular generative models, Generative Adversarial Networks (GANs) focus on sample generation.

# Examples of generative objectives: “Create new images”

“Given training data, generate new samples from the same distribution”



**Training data  $\sim p_{data}(x)$**



**Generated samples  $\sim p_{model}(x)$**

- Want to learn or generate  $p_{model}(x)$  similar to  $p_{data}(x)$
- This is density estimation, a core effort in unsupervised learning

## **Several flavors:**

- Explicit density estimation: explicitly define and solve for  $p_{model}(x)$
- Implicit density estimation: learn model that can sample from  $p_{model}(x)$  w/o explicitly defining it

## Small Gallery of GANs




# GAN Architectures

- Generative Models are generating realistic images.
- Generative Adversarial Networks (GANs) have shown great success in Computer Vision.
- GANs are showing promising results in Audio and Text as well.
- Some of the most popular GAN application are:
  - Transforming an image from one domain to another (CycleGAN),
  - Generating an image from a textual description (text-to-image),
  - Generating very high-resolution images (ProgressiveGAN) and many more.
- The following review articles have many examples of various GANs:  
<https://www.theclickreader.com/list-of-generative-adversarial-networks-applications/>  
<https://towardsdatascience.com/image-generation-in-10-minutes-with-generative-adversarial-networks-c2afc56bfa3b>  
<https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/>
- A few years ago, Hindu Puravinsah started collecting all “new” GAN architectures. The gentleman created a site called The-Gan-Zoo. He eventually got bored and gave up.
- The site is still there but unfortunately has no new incarnations of GANs
  - <https://github.com/hindupuravinash/the-gan-zoo>
- In what follows we will describe basic features of several popular GAN architectures.

# The GAN Zoo

- On this site: <https://github.com/hindupuravinash/the-gan-zoo> you can find a long list of GAN implementations in many area of IT, science and art. What follows is a small portion of the list:

 | <https://github.com/hindupuravinash/the-gan-zoo>

Contributions are welcome. Add links through pull requests in gans.tsv file in the same format or create an issue to lemme know something I missed or to start a discussion.

Check out [Deep Hunt](#) - my weekly AI newsletter for this repo as [blogpost](#) and follow me on [Twitter](#).

- 3D-ED-GAN - [Shape Inpainting using 3D Generative Adversarial Network and Recurrent Convolutional Networks](#)
- 3D-GAN - [Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling \(github\)](#)
- 3D-IWGAN - [Improved Adversarial Systems for 3D Object Generation and Reconstruction \(github\)](#)
- 3D-RecGAN - [3D Object Reconstruction from a Single Depth View with Adversarial Learning \(github\)](#)
- ABC-GAN - [ABC-GAN: Adaptive Blur and Control for improved training stability of Generative Adversarial Networks \(github\)](#)
- ABC-GAN - [GANs for LIFE: Generative Adversarial Networks for Likelihood Free Inference](#)
- AC-GAN - [Conditional Image Synthesis With Auxiliary Classifier GANs](#)
- acGAN - [Face Aging With Conditional Generative Adversarial Networks](#)
- ACGAN - [Coverless Information Hiding Based on Generative adversarial networks](#)
- ACtuAL - [ACtuAL: Actor-Critic Under Adversarial Learning](#)
- AdaGAN - [AdaGAN: Boosting Generative Models](#)
- AdvGAN - [Generating adversarial examples with adversarial networks](#)
- AE-GAN - [AE-GAN: adversarial eliminating with GAN](#)
- AEGAN - [Learning Inverse Mapping by Autoencoder based Generative Adversarial Nets](#)
- AF-DCGAN - [AF-DCGAN: Amplitude Feature Deep Convolutional GAN for Fingerprint Construction in Indoor Localization System](#)
- AffGAN - [Amortised MAP Inference for Image Super-resolution](#)
- AL-CGAN - [Learning to Generate Images of Outdoor Scenes from Attributes and Semantic Layouts](#)
- ALI - [Adversarially Learned Inference \(github\)](#)

# CycleGAN

- CycleGAN is a very popular GAN architecture used to learn transformation between images of different styles. CycleGANs can create
  - a map between artistic and realistic images,
  - a transformation between images of horse and zebra,
  - a transformation between winter image and summer image, and so on
- <http://FaceApp.com> is one of the most popular examples of CycleGAN where human faces are transformed into different age groups or the facial expression is changed



- Paper: <https://arxiv.org/pdf/1703.10593.pdf>
- Code: <https://www.tensorflow.org/tutorials/generative/cyclegan>

# StyleGAN

- StyleGAN is a GAN formulation which is capable of generating very high-resolution images even of 1024\*1024 resolution.
- StyleGAN learn features regarding human face and generates a new image of the human face that does not exist in reality.

Visit <https://thispersondoesnotexist.com/> . Each visit to this URL will generate a new image of a human face who doesn't exist in the universe.



- Paper: <https://arxiv.org/pdf/1812.04948.pdf>
- Github: <https://github.com/NVLabs/stylegan>

# Text2image, StackGAN

- Text2image GAN architecture made significant progress in generating meaningful images based on an explicit textual description.
- This GAN formulation takes a textual description as input and generates an RGB image that was described in the textual description.
- As an example, given *“this flower has a lot of small round pink petals”* as input, it will generate an image of a flower having round pink petals.



- Research Paper: <https://arxiv.org/pdf/1605.05396.pdf>
- Github: <https://github.com/paarthneekhara/text-to-image>

# DiscoGAN, Deep Convolutional GANs (DCGANs)

- DiscoGAN became very popular because of its ability to learn cross-domain relations given unsupervised data.
- For humans, cross-domain relations are very natural. Given images of two different domains, a human can figure out how they are related to each other. As an example, in the following figure, we have images from 2 different domains and just by one glance at these images, we can figure out very easily that they are related by the nature of their exterior color.



- Research Paper: <https://arxiv.org/pdf/1703.05192.pdf>
- Github: <https://github.com/SKTBrain/DiscoGAN>



# SRGAN

- Many tasks intrinsically require generation of realistic samples from some distribution. Generative modeling is required because this task requires the model to impute more information into the image than was originally there in the input.



- Ledig et al. (2016) demonstrated excellent single-image super-resolution results.
- The leftmost image is an original high-resolution image. It is down-sampled to make a low-resolution image, and different methods are used in attempts to recover the high-resolution image.
- The bicubic method is simply an interpolation method that does not use the statistics of the training set at all.
- Paper: <https://arxiv.org/abs/1609.04802>)
- Code: <https://github.com/deepak112/Keras-SRGAN>



# pix2pix

- pix2pix uses a *conditional generative adversarial network* (cGAN) to learn a mapping from an input image to an output image.
- The network is composed of two main pieces, the **Generator** and the **Discriminator**. The **Generator** applies some transform to the input image to get the output image. The **Discriminator** compares the input image to an unknown image (either a target image from the dataset or an output image from the generator) and tries to guess if this was produced by the generator.

- You can test the power of the technique at:  
<https://affinelayer.com/pixsrv/>
- Paper: <https://arxiv.org/abs/1611.07004>
- Code: <https://github.com/affinelayer/pix2pix-tensorflow>



# Objectives: “Samples for Reinforcement Learning”

- Reinforcement learning algorithms used in robotics, typically require many labeled examples to be able to generalize well.
- Those learning algorithm can be made more affordable or improve their generalization ability by studying many unlabeled examples created by generative models, and specially GANs.

# Additional Objectives for Generative Models

- Realistic samples for artwork, architecture, engineering, super-resolution image enhancement, movie colorization, etc.



- Generative models of time-series data can be used for simulation and planning (reinforcement learning applications!)
- Training generative models can also enable inference of the latent representations what can be useful as general features generation technique.

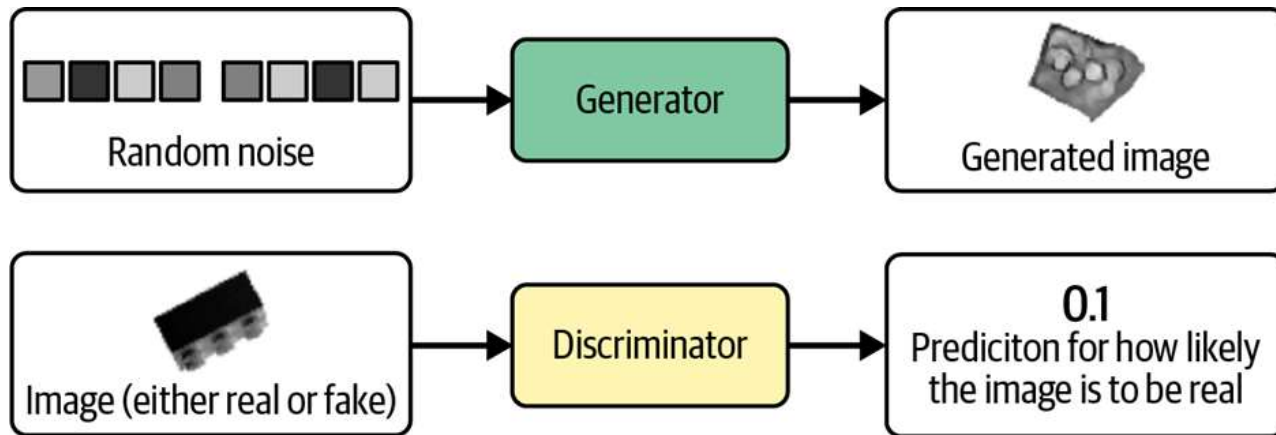
# Generative Adversarial Networks - GANs

# Generative Adversarial Networks

- Generative adversarial networks (GANs), introduced by Ian Goodfellow et al., “Generative Adversarial Networks,” arXiv (2014), <https://arxiv.org/abs/1406.2661> are an alternative to VAEs for learning latent spaces of images. GANs enable the generation of fairly realistic synthetic images almost indistinguishable from real ones.
- An intuitive way to understand GANs is to imagine a forger trying to create a fake Picasso painting. At first, the forger is pretty bad at the task. He mixes some of his fakes with authentic Picassos and shows them all to an art dealer. The art dealer makes an authenticity assessment for each painting and gives the forger feedback about what makes a Picasso look like a Picasso. The forger goes back to his studio to prepare some new fakes. As time goes on, the forger becomes increasingly competent at imitating the style of Picasso, and the art dealer becomes increasingly expert at spotting fakes. In the end, they have on their hands some excellent fake Picassos.
- That’s what a GAN is: a forger network and an expert network, each being trained to best the other. As such, a GAN is made of two parts: the generator and the discriminator.
  - **Generator network**—Takes as input a random vector (a random point in the latent space), and decodes it into a synthetic image
  - **Discriminator network** (or adversary)—Takes as input an image (real or synthetic), and predicts whether the image came from the training set or was created by the generator network

# Adversaries

- A GAN is a battle between two adversaries, the generator and the discriminator.
- The generator tries to convert **random latent space noise** into observations that look as if they are sampled from the original dataset, and the discriminator tries to predict whether an observation comes from the original dataset or is one of the generator's forgeries. Examples of the inputs and outputs to the two networks are shown below.



- At the start of the process, the generator outputs noisy images and the discriminator predicts readily. With time both networks improve in what they do.
- The key to GANs is alternative training of the two networks. The generator becomes more adept at fooling the discriminator. The discriminator adapts in order to maintain its ability to correctly identify which observations are fake. This drives the generator to find new ways to fool the discriminator, and so the cycle continues.
- The optimization process seeks not a minimum of a loss function but an equilibrium between two forces.
- The above image is from *Chapter 4, Generative Deep Learning, 2<sup>nd</sup> Ed.* by David Foster, O-Reilly, 2023

# The Nature of Discriminator vs. Generator

- The goal of the discriminator is to predict if an image is real or fake. This is a supervised image classification problem, so we can use a similar architecture to those we worked with earlier like stacked convolutional layers, with a single output node giving us the probability between 0 (fake) and 1 (real).
- The input to the generator will be a vector drawn from a multivariate standard normal distribution. The output is an image of the same size as an image in the original training data.
- This description may remind you of the decoder in a variational autoencoder. In fact, the generator of a GAN fulfills exactly the same purpose as the decoder of a VAE: converting a vector in the latent space to an image.
- The concept of mapping from a latent space back to the original domain is very common in generative modeling, as it gives us the ability to manipulate vectors in the latent space to change high-level features of images in the original domain.
- In the following slides we will present Deep Convolutional GAN (DCGAN).



# Training DCGANs

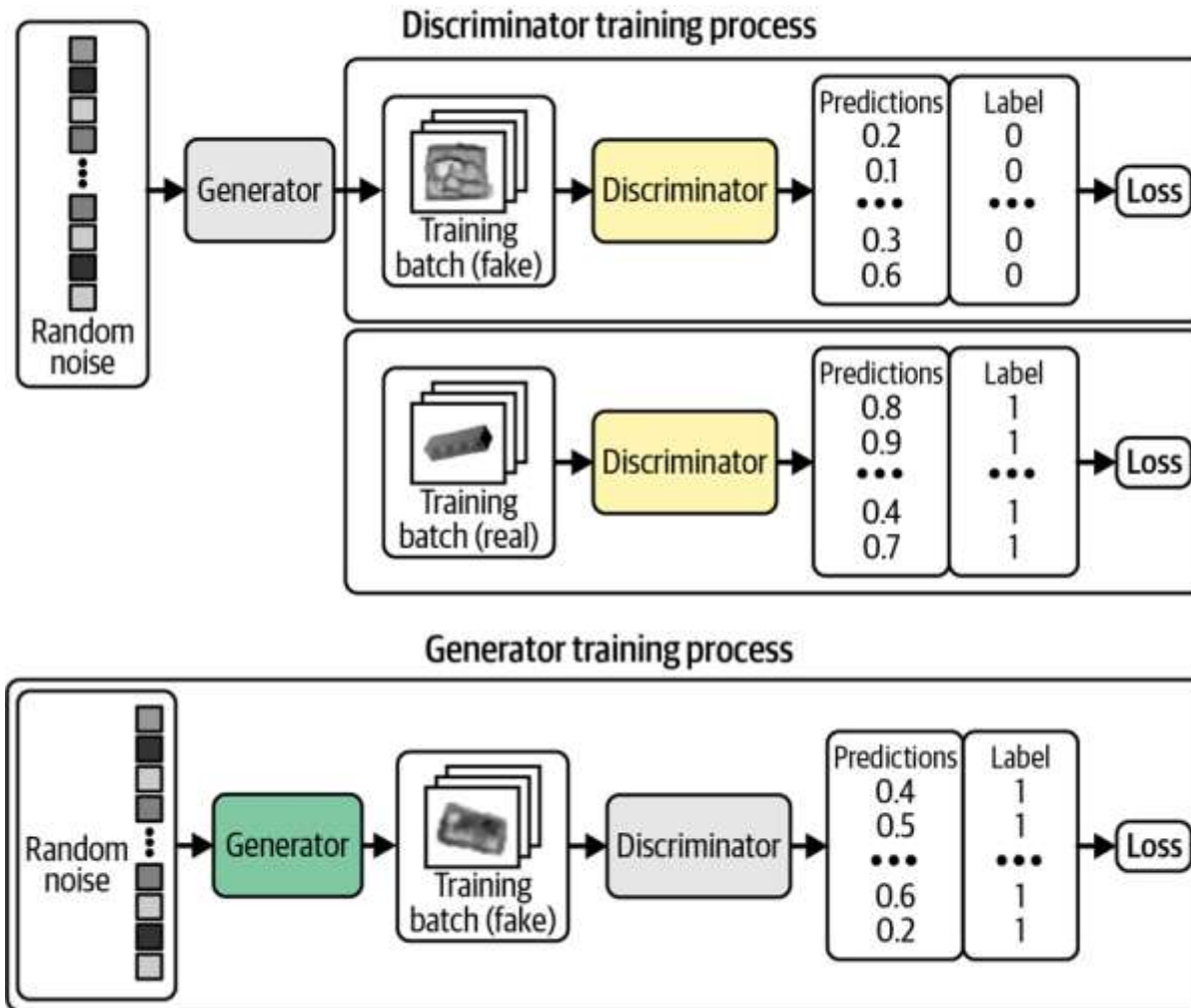
- The key to understanding GANs lies in understanding the training process for the generator and discriminator.
- We train the discriminator by creating a training set where some of the images are real observations from the training set and some are fake outputs from the generator. We treat this as a supervised learning problem, where the labels are 1 for the real images and 0 for the fake images, with binary cross-entropy as the loss function.
- To train the generator, we need to find a way of scoring each generated image so that the generator can produce more high-scoring images. The discriminator can help! We can generate a batch of images and pass these through the discriminator to get a score for each image.
- The loss function for the generator is then simply the binary cross-entropy between these probabilities and a vector of ones, because we want to train the generator to produce images that the discriminator thinks are real.

# Training GANs, Update one network at a time

- Crucially, we must alternate the training of these two networks, making sure that we only update the weights of one network at a time.
- During the generator training process, only the generator's weights are updated.
- If we allowed the discriminator's weights to change as well, the discriminator would just adjust so that it is more likely to predict the generated images to be real, which is not the desired outcome.
- We want generated images to be predicted close to 1 (real) because the generator is strong, not because the discriminator is weak.

# Training of the Discriminator and the Generator

- A diagram of the training process for the discriminator and generator is shown below:



- The above image is from *Chapter 4, Generative Deep Learning, 2<sup>nd</sup> Ed.* by David Foster, O'Reilly, 2023

# Deep Convolutional GAN (DcGAN)

# DCGANs

- In the paper “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks” by Alec Radford, Luke Metz, Soumith Chintala, <https://arxiv.org/abs/1511.06434>, proposed Deep Convolutional GANs.

```
codings_size = 100
```

```
generator = tf.keras.Sequential([
    tf.keras.layers.Dense(7 * 7 * 128),
    tf.keras.layers.Reshape([7, 7, 128]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2,
                                     padding="same", activation="relu"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2,
                                     padding="same", activation="tanh"),
])
discriminator = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="same",
                           activation=tf.keras.layers.LeakyReLU(0.2)),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="same",
                           activation=tf.keras.layers.LeakyReLU(0.2)),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
gan = tf.keras.Sequential([generator, discriminator])
```

# DCGAN Architecture

- Most GANs today are at least loosely based on the DcGAN architecture (Radford et al., 2015).
- Though GANs were both deep and convolutional prior to DCGANs, the name DcGAN refers to a specific style of architecture. Some of the key recommendations when implementing the DcGAN architecture are:
  - Use batch normalization (Ioffe and Szegedy, 2015) layers in most layers of both the discriminator and the generator, with the two mini-batches for the discriminator normalized separately.
  - The last layer of the generator and first layer of the discriminator are not batch normalized, so that the model can learn the correct mean and scale of the data distribution.
  - The overall network structure is mostly borrowed from the all-convolutional net (Springenberg et al., 2015). This architecture contains neither pooling nor unpooling" layers.
  - When the generator needs to increase the spatial dimension of the representation it uses transposed convolution with a stride greater than 1.
  - Use of the Adam optimizer rather than SGD with momentum.

# Deep Convolutional Generative Adversarial Network (DcGAN)

- The following slides demonstrates how to generate images of handwritten digits using a Deep Convolutional Generative Adversarial Network (DCGAN). The code is written using the Keras Sequential API with a `tf.GradientTape` training loop.

```
import tensorflow as tf
# To generate GIFs
!pip install -q imageio
!pip install -q git+https://github.com/tensorflow/docs
import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time

from IPython import display
```



# Load and prepare the dataset

- We will use the MNIST dataset to train the generator and the discriminator. The generator will generate handwritten digits resembling the MNIST data.

```
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1)
                    .astype('float32')
train_images = (train_images - 127.5) / 127.5 # Normalize images to [-1, 1]
BUFFER_SIZE = 60000
BATCH_SIZE = 256
# Batch and shuffle the data
train_dataset =
tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE)
                    .batch(BATCH_SIZE)
```

# Create the models, Generator

- Both the generator and discriminator are defined using the Keras Sequential API.
- The generator uses `tf.keras.layers.Conv2DTranspose` (upsampling) layers to produce an image from a seed (random noise in the latent space).
- Start with a Dense layer that takes this seed as input, then upsample several times until you reach the desired image size of 28x28x1. Notice the `tf.keras.layers.LeakyReLU` activation for each layer, except the output layer which uses `tanh()`.

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False,
                                     activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

# Test the generator

- Use the (as yet untrained) generator to create an image.

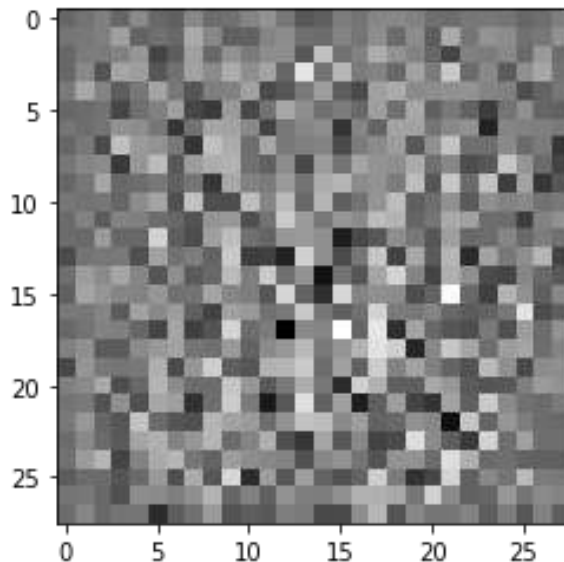
```
generator = make_generator_model()
```

```
noise = tf.random.normal([1, 100])
```

```
generated_image = generator(noise, training=False)
```

```
plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

```
Out[10] <matplotlib.image.AxesImage at 0x21bae83ffd0>
```



# The Discriminator

- The discriminator is a CNN-based image classifier.

```
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                            input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3)) # usually, only this Dropout is used.

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model
```

# Test the discriminator

- Use the (as yet untrained) discriminator to classify the generated images as real or fake.
- The model will be trained to output positive values for real images, and negative values for fake images.

```
discriminator = make_discriminator_model()  
decision = discriminator(generated_image)  
print (decision)
```

```
tf.Tensor([[0.00262085]], shape=(1, 1), dtype=float32)
```

# The loss and optimizers, Discriminator loss

- Define loss functions and optimizers for both models.

```
# This method returns a helper function to compute cross entropy loss  
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

## Discriminator loss

- This method distinguishes real images from fakes. It compares the discriminator's predictions on real images to an array of 1s, and the discriminator's predictions on fake (generated) images to an array of 0s.

```
def discriminator_loss(real_output, fake_output):  
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)  
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)  
    total_loss = real_loss + fake_loss  
    return total_loss
```

# Generator loss, Optimizers

- The generator's loss quantifies how well it was able to trick the discriminator. Intuitively, if the generator is performing well, the discriminator will classify the fake images as real (or 1). Here, we will compare the discriminators decisions on the generated images to an array of 1s.

```
def generator_loss(fake_output):  
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

- The discriminator and the generator optimizers are different objects since we will train two networks separately.

```
generator_optimizer = tf.keras.optimizers.Adam(1e-4)  
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```



# Save checkpoints, Environment

- We will also demonstrate how to save and restore models, which can be helpful in case a long running training task is interrupted.

```
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
discriminator_optimizer=discriminator_optimizer,
                           generator=generator,
                           discriminator=discriminator)
```

- Next we will set the training environment:

```
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16
# We will reuse this seed overtime (so it's easier)
# to visualize progress in the animated GIF)
seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

- The training loop begins with generator receiving a random seed as input. That seed is used to produce an image.

# Training Steps

- This is what the training loop looks like schematically. For each epoch, you do the following:
  1. Draw random points in the latent space (random noise).
  2. Generate images with generator using this random noise.
  3. Mix the generated images with real ones.
  4. Train discriminator using these mixed images, with corresponding targets: either “real” (for the real images) or “fake” (for the generated images).
  5. Draw new random points in the latent space.
  6. Train generator using these random vectors, with targets that all say “these are real images.” This updates the weights of the generator to move them toward getting the discriminator to predict “these are real images” for generated images: this trains the generator to fool the discriminator.

# The Training Step

- The discriminator is then used to classify real images (drawn from the training set) and fakes images (produced by the generator). The loss is calculated for each of these models, and the gradients are used to update the generator and discriminator.

```
# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss,
        generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
        discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
        generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
        discriminator.trainable_variables))
```

# The Training Loop

```
def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()

        for image_batch in dataset:
            train_step(image_batch)

        # Produce images for the GIF as we go
        display.clear_output(wait=True)
        generate_and_save_images(generator,
                                epoch + 1,
                                seed)

        # Save the model every 15 epochs
        if (epoch + 1) % 15 == 0:
            checkpoint.save(file_prefix = checkpoint_prefix)

        print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

    # Generate after the final epoch
    display.clear_output(wait=True)
    generate_and_save_images(generator,
                            epochs,
                            seed)
```

# Generate and save images

```
def generate_and_save_images(model, epoch, test_input):
    # Notice `training` is set to False.
    # This is so all layers run in inference mode (batchnorm).
    predictions = model(test_input, training=False)

    fig = plt.figure(figsize=(4,4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis('off')

    plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
    plt.show()
```

# Train the model

- We call the `train()` method defined above to train the generator and discriminator simultaneously. Note, training GANs can be tricky. It's important that the generator and discriminator do not overpower each other (e.g., that they train at a similar rate).
- At the beginning of the training, the generated images look like random noise. As training progresses, the generated digits look increasingly real. After about 50 epochs, they resemble MNIST digits. This may take about one minute / epoch with the default settings on Colab.

```
train(train_dataset, EPOCHS)
```



- Restore the latest checkpoint.

```
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))  
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at  
0x21c9e1a5208>
```

# Create a GIF

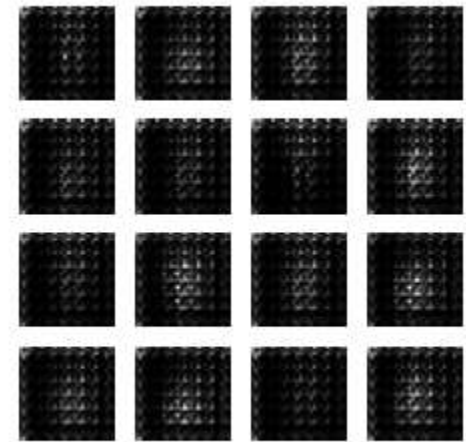
```
# Display a single image using the epoch number
def display_image(epoch_no):
    return PIL.Image.open(
        'image_at_epoch_{:04d}.png'.format(epoch_no))
display_image(EPOCHS)
```



- Use `imageio` to create an animated gif using the
- images saved during training.

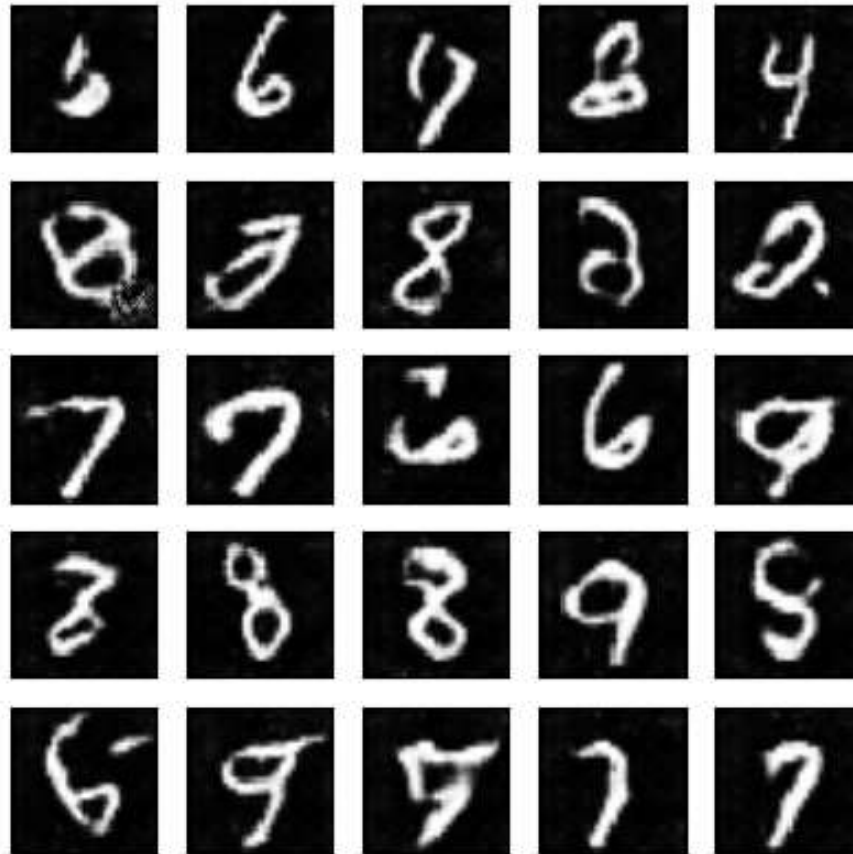
```
anim_file = 'dcgan.gif'
with imageio.get_writer(anim_file, mode='I') as writer:
    filenames = glob.glob('image*.png')
    filenames = sorted(filenames)
    for filename in filenames:
        image = imageio.imread(filename)
        writer.append_data(image)
        image = imageio.imread(filename)
        writer.append_data(image)

import tensorflow_docs.vis.embed as embed
embed.embed_file(anim_file)
```





# DcGAN Fixed Results



Epoch 19

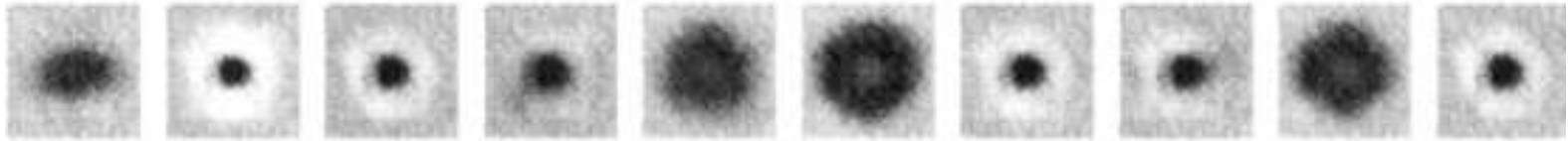
# Bag of Tricks for DcGANs

# The Bag of Tricks

- Training GANs and tuning GAN implementations is difficult. There are a number of known "tricks" that are backed by some level of intuitive understanding of the phenomenon at hand, and they are known to work well empirically.
- Use `tanh()` as the last activation in the generator, instead of sigmoid, which is more commonly found in other types of models.
- Sample points from the latent space using a normal distribution (Gaussian distribution), not a uniform distribution.
- Adding stochasticity is a good way to enhance robustness. Since GAN training results in a dynamic equilibrium, GANs are likely to get "stuck" in all sorts of ways. Introducing randomness during training helps prevent this. Introduce randomness in two ways:
  1. use dropout in the discriminator,
  2. add some random noise to the labels for the discriminator.
- Sparse gradients can hinder GAN training. In deep learning, sparsity is often a desirable property, but not in GANs. There are two things that can induce gradient sparsity:
  1. max pooling operations,
  2. ReLU activations.
- Instead of max pooling, use strided convolutions for down-sampling, use LeakyReLU layer instead of a ReLU activation. LeakyReLU is similar to ReLU but it relaxes sparsity constraints by allowing small negative activation values.
- In generated images, it is common to see "checkerboard artifacts" caused by unequal coverage of the pixel space in the generator. To fix this, when using strided Conv2DTranspose or Conv2D in both the generator and discriminator, use a kernel size that is divisible by the stride size.

# Discriminator overpowers the generator

- If the discriminator becomes too strong, the signal from the loss function becomes too weak to drive any meaningful improvements in the generator. In the worst-case scenario, the discriminator perfectly learns to separate real images from fake images and the gradients vanish completely, leading to no training whatsoever.
- Typical generator output then looks like the following:

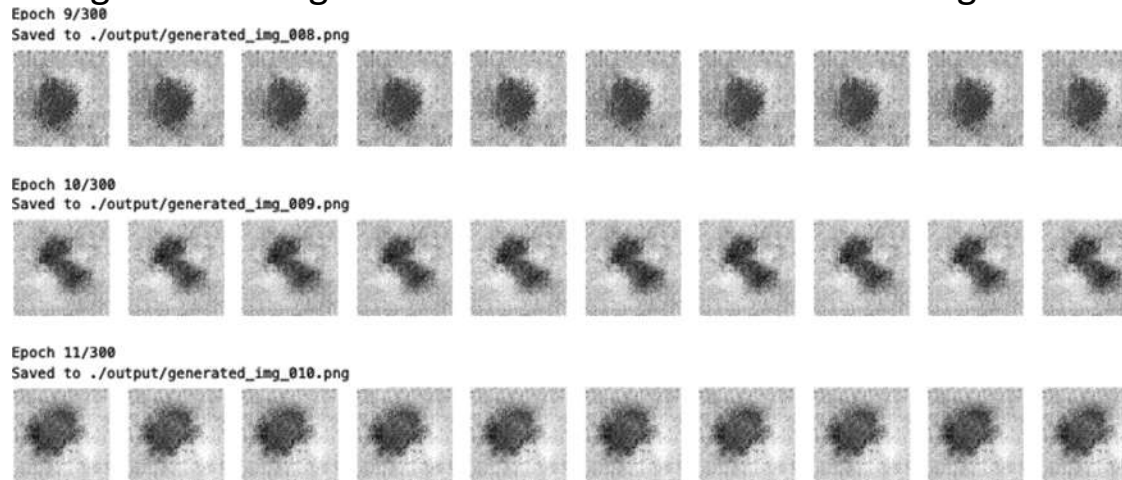


If you find your discriminator loss function collapsing, you need to find ways to weaken the discriminator. Try the following suggestions:

- Increase the rate parameter of the Dropout layers in the discriminator to dampen the amount of information that flows through the network.
- Reduce the learning rate of the discriminator.
- Reduce the number of convolutional filters in the discriminator.
- Add noise to the labels when training the discriminator.
- Flip the labels of some images at random when training the discriminator.

# Generator overpowers the discriminator

- If the discriminator is not powerful enough, the generator will find ways to easily trick the discriminator with a small sample of nearly identical images. This is known as mode collapse.
- For example, suppose we were to train the generator over several batches without updating the discriminator in between. The generator would be inclined to find a single observation (also known as a mode) that always fools the discriminator and would start to map every point in the latent input space to this image. Moreover, the gradients of the loss function would collapse to near 0, so it wouldn't be able to recover from this state.
- Even if we then tried to retrain the discriminator to stop it being fooled by this one point, the generator would simply find another mode that fools the discriminator, since it has already become numb to its input and therefore has no incentive to diversify its output.
- If you find that your generator is suffering from mode collapse, you can try strengthening the discriminator using the opposite suggestions to those listed in the previous slide. Also, you can try reducing the learning rate of both networks and increasing the batch size.



# The Bag of Tricks Site

- On the following page, you can find a large collection of excellent recommendations on how to train GANs: <https://github.com/soumith/ganhacks>

] | <https://github.com/soumith/ganhacks>

---

## How to Train a GAN? Tips and tricks to make GANs work

While research in Generative Adversarial Networks (GANs) continues to improve the fundamental stability of these models, we use a bunch of tricks to train them and make them stable day to day.

Here are a summary of some of the tricks.

[Here's a link to the authors of this document](#)

If you find a trick that is particularly useful in practice, please open a Pull Request to add it to the document. If we find it to be reasonable and verified, we will merge it in.

### 1. Normalize the inputs

- normalize the images between -1 and 1
- Tanh as the last layer of the generator output

### 2: A modified loss function

In GAN papers, the loss function to optimize G is  $\min (\log 1-D)$ , but in practice folks practically use  $\max \log D$

- because the first formulation has vanishing gradients early on
- Goodfellow et. al (2014)

In practice, works well:

- Flip labels when training generator: real = fake, fake = real

### 3: Use a spherical Z

## Some General Feature of DcGANs

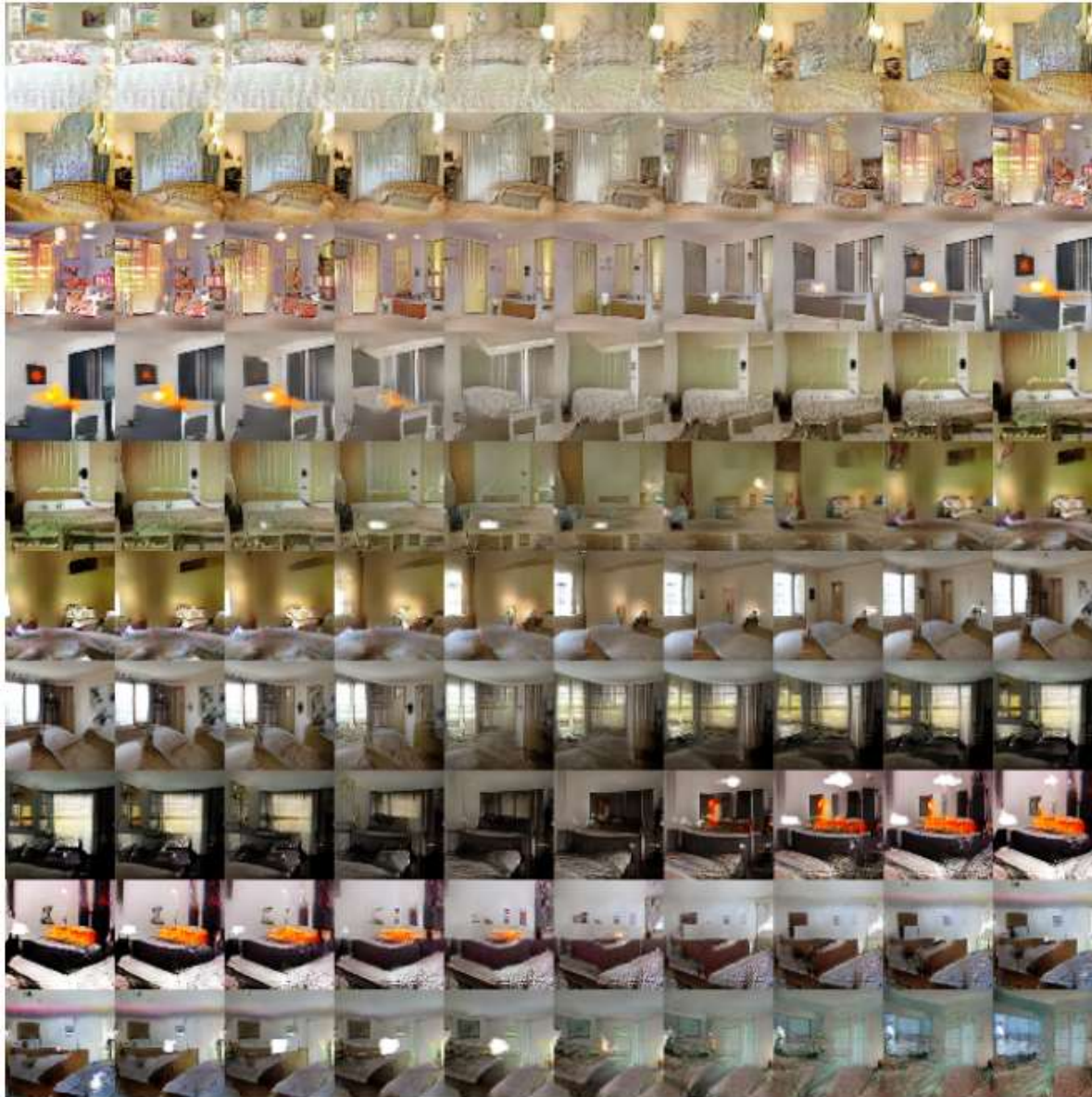


# What could DCGANs generate

- If you train them on images from furniture catalogs, DcGANs can generate impressive realistic images of furnished rooms. To demonstrate how DcGAN model scales with more data and higher resolution generation, Radford et al 2015, trained a model on the LSUN bedrooms dataset containing a little over 3 million training examples of images of furnished bedrooms.



# Continuous Evolution, Interpolation of Images



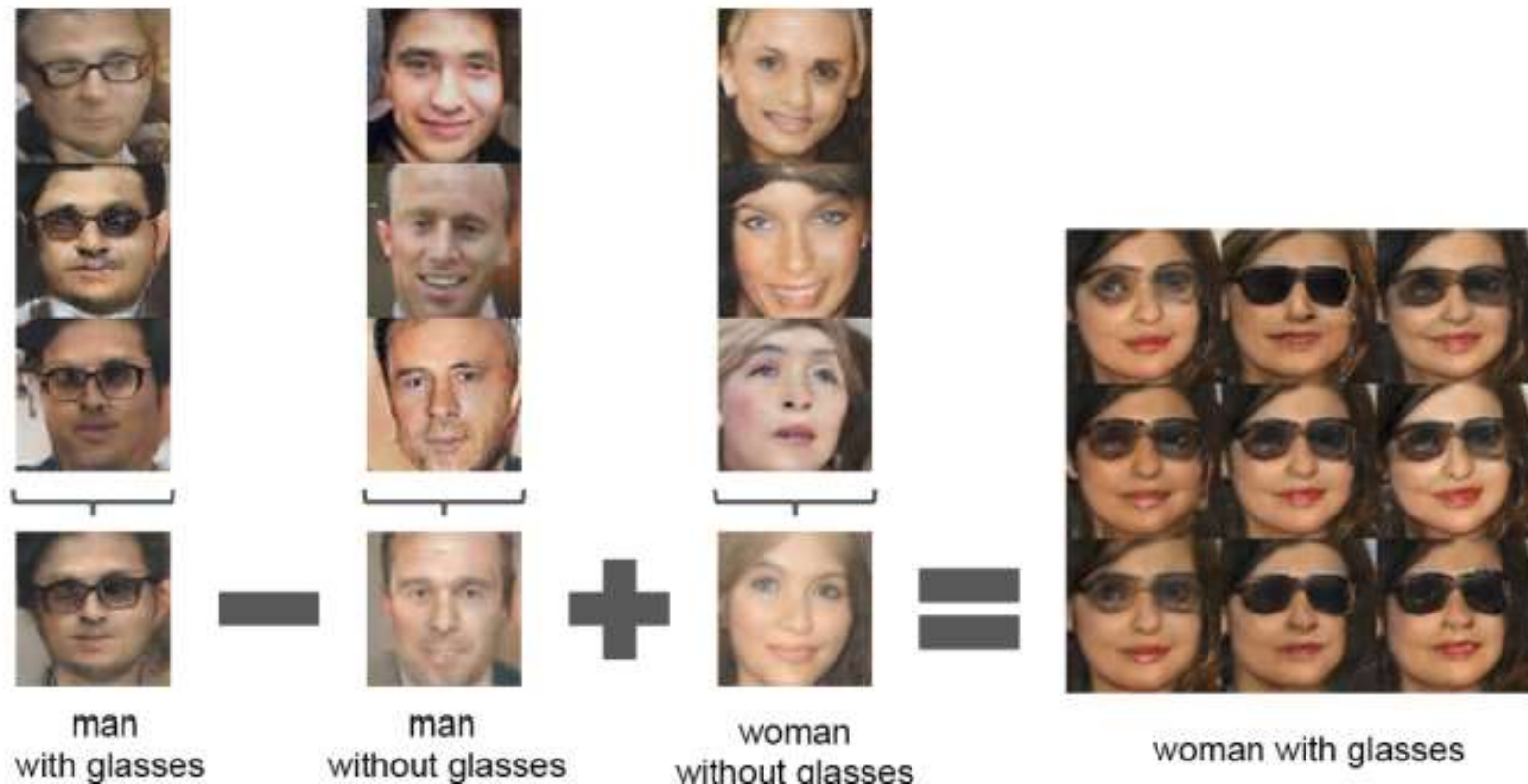
# Generation of Faces

- If you scale up DCGAN architecture and train it on a large dataset of faces, you can get fairly realistic images. DCGANs can learn quite meaningful latent representations, as you can see on the next slide.
- Many images were generated, and nine of them were picked manually (top left), including three representing men with glasses, three men without glasses, and three women without glasses. For each of these categories, the codings that were used to generate the images were averaged, and an image was generated based on the resulting mean codings (lower left).
- In short, each of the three lower-left images represents the mean of the three images located above it. But this is not a simple mean computed at the pixel level (this would result in three overlapping faces), it is a mean computed in the latent space, so the images still look like normal faces.
- You compute men with glasses, minus men without glasses, plus women without glasses—where each term corresponds to one of the mean codings—and you generate the image that corresponds to this coding, you get the image at the center of the  $3 \times 3$  grid of faces on the right: a woman with glasses!
- The eight other images around it were generated based on the same vector plus a bit of noise, to illustrate the semantic interpolation capabilities of DCGANs. It appears that DCGANs are able to do arithmetic on faces, like we did with kings and queens in Word2Vec space.



# Embedded Spaces of Images

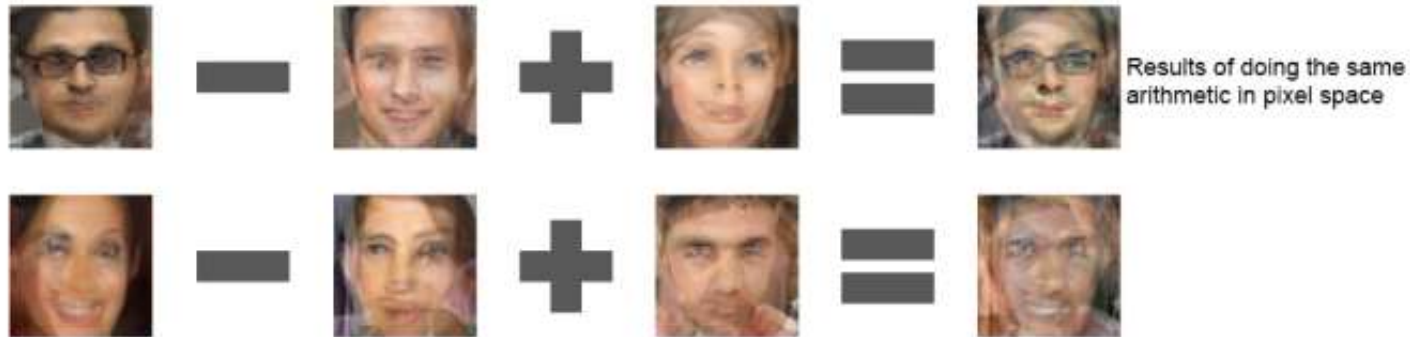
- You recall how word2vec was able to perform arithmetic on words:  
 $\text{king} - \text{man} + \text{woman} \Rightarrow \text{queen}$
- With DcGANs Radford et al. 2015 were able to demonstrate Image Arithmetic.



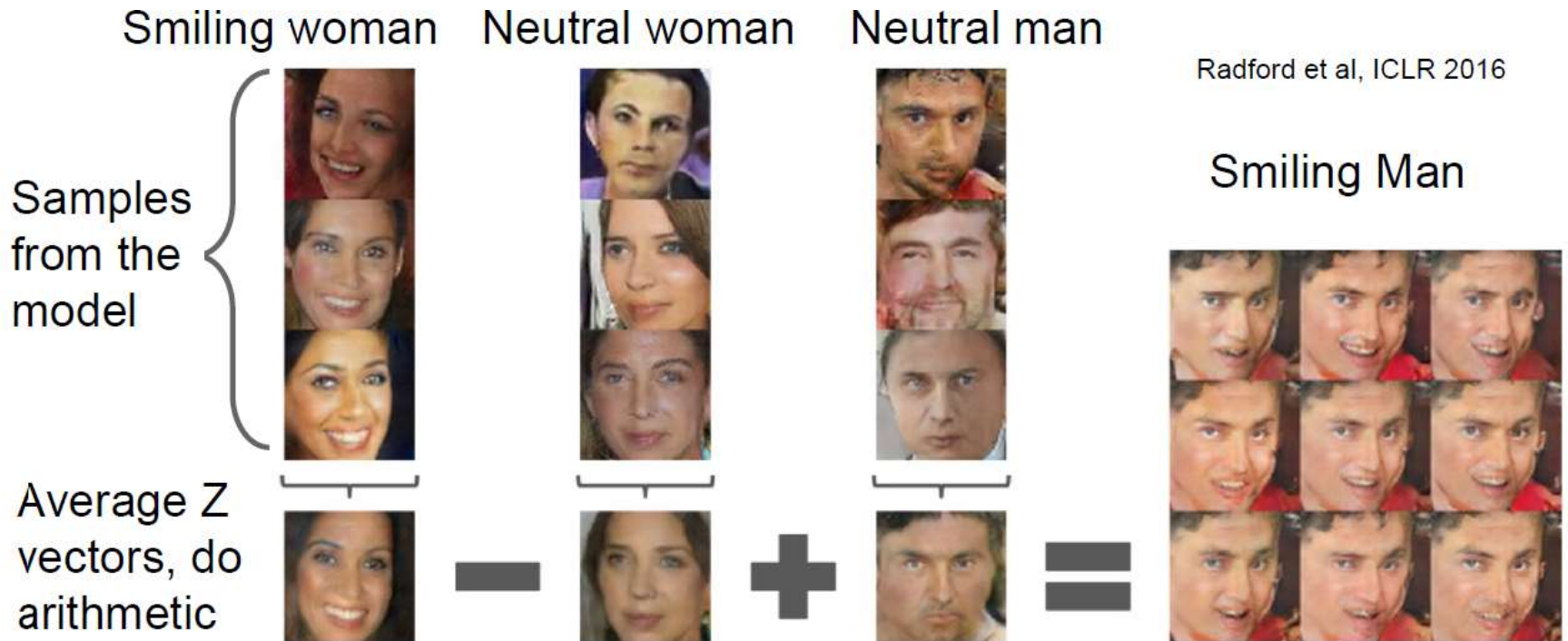
- For each column, the Z vectors of samples are averaged. Arithmetic was then performed on the mean vectors creating a new vector Y. The center sample on the right-hand side is produced by feeding Y as input to the generator. To demonstrate the interpolation capabilities of the generator, uniform noise sampled with scale +/-0.25 was added to Y to produce the 8 other samples

# Attempt at Arithmetic in Pixel Space

- If you would try to do similar calculation in pixel space, results would not be nearly as good. Applying arithmetic in the input space (bottom two examples) results in noisy overlap due to misalignment.



# Embedded Space of Images, Smile



# Instead of Summary

- Image generation with deep learning is done by learning latent spaces that capture statistical information about a dataset of images. By sampling and decoding points from the latent space, you can generate never-before-seen images.
- There are two major tools to do this: VAEs and GANs.
  - VAEs result in highly structured, continuous latent representations. For this reason, they work well for doing all sorts of image editing in latent space: face swapping, turning a frowning face into a smiling face, and so on. They also work nicely for doing latent-space-based animations, such as animating a walk along a cross section of the latent space, showing a starting image slowly morphing into different images in a continuous way.
  - GANs enable the generation of realistic single-frame images but may not induce latent spaces with solid structure and high continuity.
- Most successful practical applications I have seen with images rely on VAEs, but GANs were extremely popular in the world of academic research, until the appearance of Stable Diffusion.



# Generative Adversarial Networks as Nash Game Theory Exercise

# Generative Adversarial Networks (GANs)

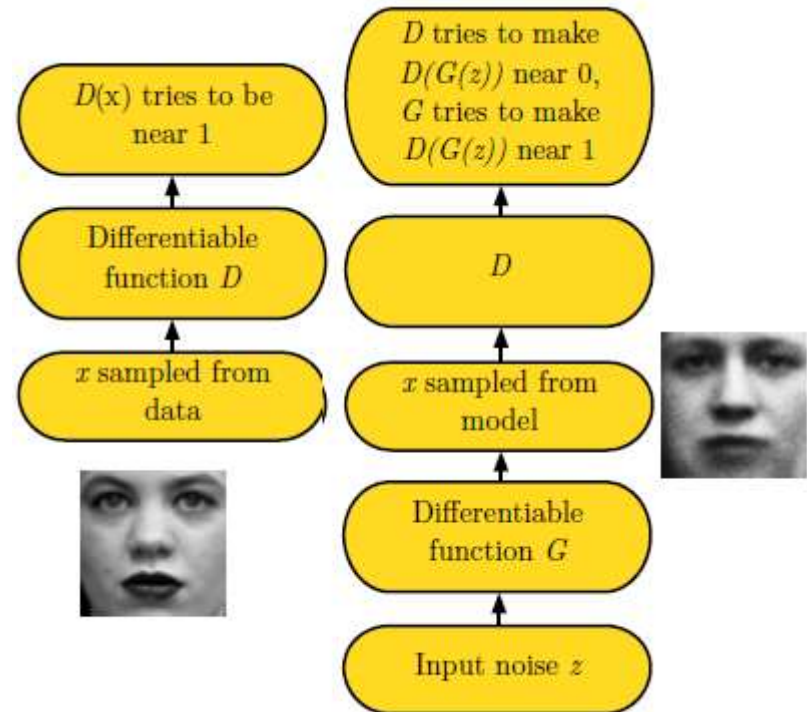
- The basic idea of GANs is to set up a game between two players.
- One of the players is called the **generator**. The generator creates samples that are intended to look as if they come from the same distribution as the training data.
- The other player is the **discriminator**. The discriminator examines samples to determine whether they are real or fake. The discriminator learns using traditional supervised learning techniques, dividing inputs into two classes (real or fake).
- The generator is being trained to fool the discriminator. We can think of the generator as a counterfeiter, trying to make fake money, and the discriminator as police detective, trying to allow legitimate money and catch counterfeit money.
- To succeed in this game, the counterfeiter tries to learn to make money that is indistinguishable from genuine money. In other words, the generator must learn to create samples that look as if they are drawn from the same distribution as the training data.
- The two players in the game are represented by two functions, each of which is differentiable both with respect to its inputs and with respect to its parameters.
- The discriminator is a function  $D$  that takes  $x$  as input and uses *parameters*  $W^{(D)}$ .
- The generator is defined by a function  $G$  that takes  $z$  as input and uses *parameters*  $W^{(G)}$ .

# Nash Equilibrium

- In the game theory, the **Nash equilibrium**, named after American mathematician John Forbes Nash Jr., is a solution concept of a non-cooperative game involving two or more players in which each player is assumed to know the equilibrium strategies of the other players, and no player has anything to gain by changing only their own strategy.
- If each player has chosen a strategy and no player can benefit by changing strategies while the other players keep theirs unchanged, then the current set of strategy choices and the corresponding payoffs constitutes a Nash equilibrium. The Nash equilibrium is one of the foundational concepts in game theory. The reality of the Nash equilibrium of a game can be tested using experimental economics methods.
- Two players Alice and Bob are in Nash equilibrium if Alice is making the best decision she can, taking into account Bob's decision while Bob's decision remains unchanged, and Bob is making the best decision he can, taking into account Alice's decision while Alice's decision remains unchanged.

# The Game

- The GAN framework pits two adversaries against each other in a game.
- Each player is represented by a differentiable function controlled by a set of parameters. These functions are implemented as deep neural networks. The game plays out in two steps.
- In the first step, training examples  $x$  are randomly sampled from the training set and used as input for the first player, the discriminator, represented by network  $D$ . The discriminator produces the probability that its input is real rather than fake, under the assumption that half of its inputs are real and half are fake. In this first step, the goal of the discriminator is for  $D(x)$  to be near 1.
- In the second step, inputs  $z$  to the generator are randomly sampled from the latent variable space. The discriminator then receives as input a fake sample  $G(z)$  created by the generator.
- In this step, both players participate. The discriminator strives to make  $D(G(z))$  approach 0 while the generative strives to make the same quantity approach 1.
- If both models have sufficient capacity, then the Nash equilibrium of this game corresponds to the  $G(z)$  being drawn from the same distribution as the training data, and  $D(x) = 1/2$  for all  $x$ .



# Cost Functions

- Both players have cost functions that are defined in terms of respective parameters.
- The discriminator wishes to minimize  $J^{(D)}(W^{(D)}, W^{(G)})$  and must do so while controlling only  $W^{(D)}$ .
- The generator wishes to minimize  $J^{(G)}(W^{(D)}, W^{(G)})$  and must do so while controlling only  $W^{(G)}$ .
- Because each player's cost depends on the other player's parameters, but each player cannot control the other player's parameters, this scenario is most straightforward to describe as a game rather than as an optimization problem.
- The solution to an optimization problem is a (local) minimum, a point in parameter space where all neighboring points have greater or equal cost. The solution to a game is a Nash equilibrium.
- Nash equilibrium is a tuple  $(W^{(D)}, W^{(G)})$  that is a local minimum of  $J^{(D)}$  with respect to  $W^{(D)}$  and a local minimum of  $J^{(G)}$  with respect to  $W^{(G)}$ .

## The discriminator's cost, $J^{(D)}$

- Most of the different games designed for GANs use the same cost for the discriminator,  $J^{(D)}$ . They differ only in terms of the cost used for the generator,  $J^{(G)}$ .
- The cost used for the discriminator is:

$$J^{(D)}(W^{(D)}, W^{(G)}) = \frac{1}{2} E_{x \sim p_{data}} \log(D(x)) - \frac{1}{2} E_z \log(1 - D(G(z)))$$

- Vector  $x$  labels a point in the space of real images and  $z$  labels a point in the latent space. This is just the standard cross-entropy cost that is minimized when training a standard binary classifier with a sigmoid output. The only difference is that the classifier is trained on two mini-batches of data; one coming from the dataset  $\{x\}$ , where the label is 1 for all examples, and one coming from the generator ( $D(G(z))$ ), where the label is 0 for all examples.
- All versions of the GAN game encourage the discriminator to minimize the above equation. In all cases, the discriminator has the same optimal strategy.
- By training the discriminator, one obtain an estimate of the ratio

$$p_{data}(x)/p_{model}(x)$$

- at every point  $x$ . Estimating this ratio enables us to compute a wide variety of divergences and their gradients. This is the key approximation technique that sets GANs apart from variational autoencoders and Boltzmann machines.
- Other deep generative models make approximations based on lower bounds of Markov chains; GANs make approximations based on using supervised learning to estimate a ratio of two densities.
- The GAN approximation is subject to the failures of supervised learning: overfitting and underfitting. In principle, with perfect optimization and enough training data, these failures can be overcome.

# The Discriminator's Cost

- A complete specification of the game requires that we specify a cost function also for the generator. The simplest version of the game is a zero-sum game, in which the sum of all player's costs is always zero. In this version of the game,

$$J^{(G)} = -J^{(D)}$$

- Because  $J^{(G)}$  is tied directly to  $J^{(D)}$ , we can summarize the entire game with a value function specifying the discriminator's payoff:

$$V(W^{(D)}, W^{(G)}) = -J^{(D)}(W^{(D)}, W^{(G)})$$

- Zero-sum games are also called minimax games because their solution involves minimization in an outer loop and maximization in an inner loop:

$$W^{(G)*} = \arg \min_{W^{(G)}} \max_{W^{(D)}} V(W^{(D)}, W^{(G)})$$

- The minimax game is mostly of interest because it is easily amenable to theoretical analysis. One approach is to define the cost function for the discriminator as:

$$J^{(G)}(W^{(D)}, W^{(G)}) = -\frac{1}{2} E_z \log(D(G(z)))$$

# The training process

- The training process consists of simultaneous Stochastic Gradient Descent.
- On each step, two mini-batches are sampled: a mini-batch of  $x$  values from the dataset and a mini-batch of  $z$  values drawn from the model's prior over latent variables.
- Then two gradient steps are made simultaneously: one updating  $W^{(D)}$  to reduce  $J^{(D)}$  and one updating  $W^{(G)}$  to reduce  $J^{(G)}$ . In both cases, it is possible to use the gradient-based optimization algorithm of your choice. Adam optimizer (Kingma and Ba, 2014) is usually a good choice.
- Many authors recommend running more steps of one player than the other, but as of late 2016, prevailing opinion is that the protocol that works the best in practice is simultaneous gradient descent, with one step for each player.
- Remarkably, a GAN is a system where the optimization minimum isn't fixed, unlike in any other training we encountered so far. Normally, gradient descent consists of rolling down hills in a static loss landscape.
- With a GAN, every step taken down the hill changes the entire landscape a little. It's a dynamic system where the optimization process is seeking not a minimum, but an equilibrium between two forces.
- GANs are notoriously difficult to train—getting a GAN to work requires lots of careful tuning of the model architecture and training parameters