Published in Towards Data Science · Follow

Matthew Stewart, PhD Researcher · Follow
May 9, 2019 · 22 min read

Advanced Topics in GANs

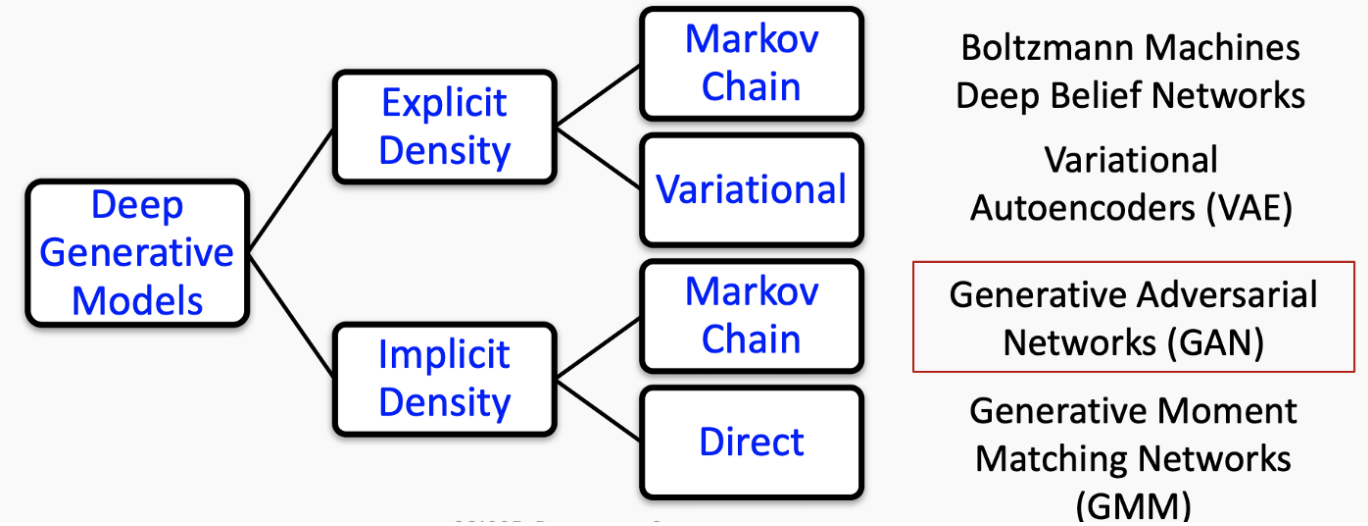
Want to turn horses into zebras? Make DIY anime characters or celebrities? Generative adversarial networks (GANs) are your new best friend.

"Generative Adversarial Networks is the most interesting idea in the last 10 years in Machine Learning." — **Yann LeCun, Director of AI Research at Facebook AI**

Part 1 of this tutorial can be found here:

Introduction to Turing Learning and GANs
Want to turn horses into zebras? Make DIY anime characters or celebrities? Generative adversarial networks (GANs) are…
towardsdatascience.co

This is the second part of a 3 part tutorial on creating deep generative models specifically using generative adversarial networks. This is a natural extension to the previous topic on variational autoencoders (found here). We will see that GANs are largely superior to variational autoencoders, but are notoriously difficult to work with.



Taxonomy of deep generative models. This article's focus is on GANs.

Throughout this tutorial, we will tackle the following topics:

- Short Review of GANs
- GAN Applications
- Problems with GANs
- Other Types of GANs
- Building an Image GAN
- Coding Tutorial

All related code can now be found in my GitHub repository:

Short Review of GANs

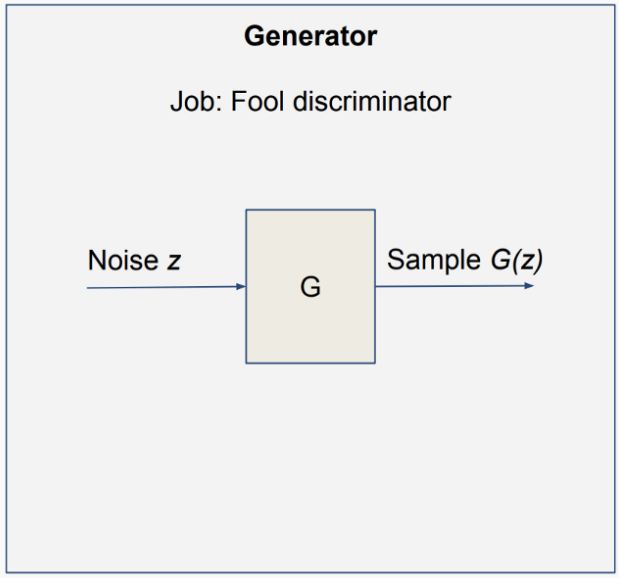
For those of you who have read the first part, you can probably skip over this section as I will just be reiterating how GANs work mechanistically using my previous example with pandas.

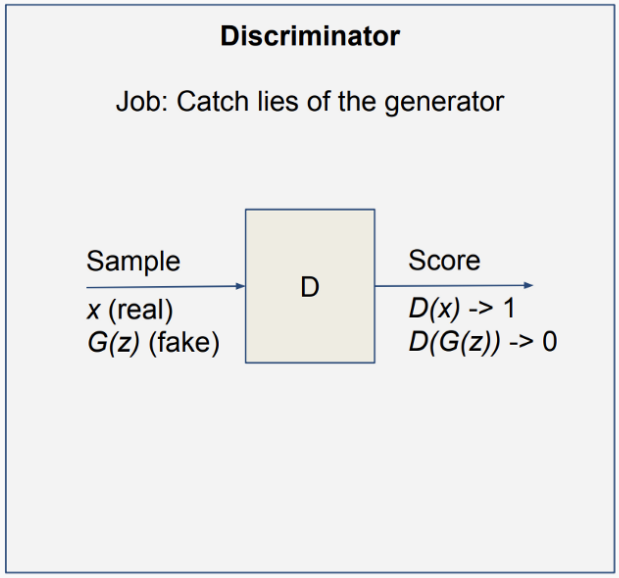
Generative adversarial nets were recently introduced (in the past 5 years or so) as a novel way to train a generative model, i.e. create a model that is able to generate data. They consist of two 'adversarial' models: a generative model *G* that captures the data distribution, and a discriminative model *D* that estimates the probability that a sample came from the training data rather than *G*. Both *G* and *D* could be a non-linear mapping function, such as a multi-layer perceptron

In a generative adversarial network (GAN) we have two neural networks that are pitted against each other in a zero-sum game, whereby the first network, the generator, is tasked with fooling the second network, the discriminator. The generator creates 'fake' pieces of data, in this case, images of pandas, to try and fool the discriminator into classifying the image as a real panda. We can iteratively improve these two networks in order to produce photorealistic images as well as performing some other very cool applications, some of which we will discuss later in this tutorial.

Initially, the images may be fairly obviously faked, but as the networks get better, it becomes harder to distinguish between real and fake images, even for humans!

The two networks can be considered black boxes representing some arbitrarily complicated function that is applied to either noise or real data. The input to the generator is some random noise, which produces a fake image, and the input to the discriminator is both fake samples as well as samples from our real data set. The discriminator then makes a binary decision about whether the image supplied, z, is a real image $D(z)=1$, or a fake image, $D(z)=0$.





To train the two networks we must have a loss function, and the loss function for each network depends on the second network. To train the networks we perform backpropagation whilst freezing the other network’s neuron weights.

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$$

m: Number of samples
z: Random noise samples
x: Real samples

How realistic are the generated samples?

G wants to maximize this.

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right]$$

Make sure real samples are classified as being real.

D wants to maximize this.

Make sure generated samples are classified as unreal.

D wants to minimize this.

If you are having trouble understanding what is going on, I recommend you go back and read part 1 to get a better intuition of how this training procedure works.

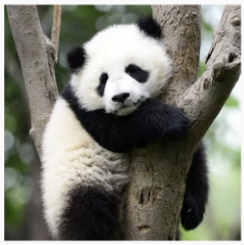
Generator

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$$

m: Number of samples
z: Random noise samples
x: Real samples

Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right]$$



x G(z)

D(x) = 0.9997 D(G(z)) = 0.1617

Generator	-	1.0
Discriminator	1.0	0.0

Targets

It is important to note that in general, the discriminator and generator networks can be any form of mapping function, such as a support vector machine. It is this generalization of GANs which is often sometimes referred to as Turing learning. In practice, however, neural networks are the most common as they are generalized function approximators for arbitrary non-linear functions.

I will now discuss some of the coolest applications of GANs before jumping into some of the more advanced topics and also a code walkthrough of GANs aimed at generating celebrity faces as well as anime characters.

GAN Applications

In this section, I will briefly talk about some of the most interesting applications of GANs that I have found during the course of my data science research. The most common topics are:

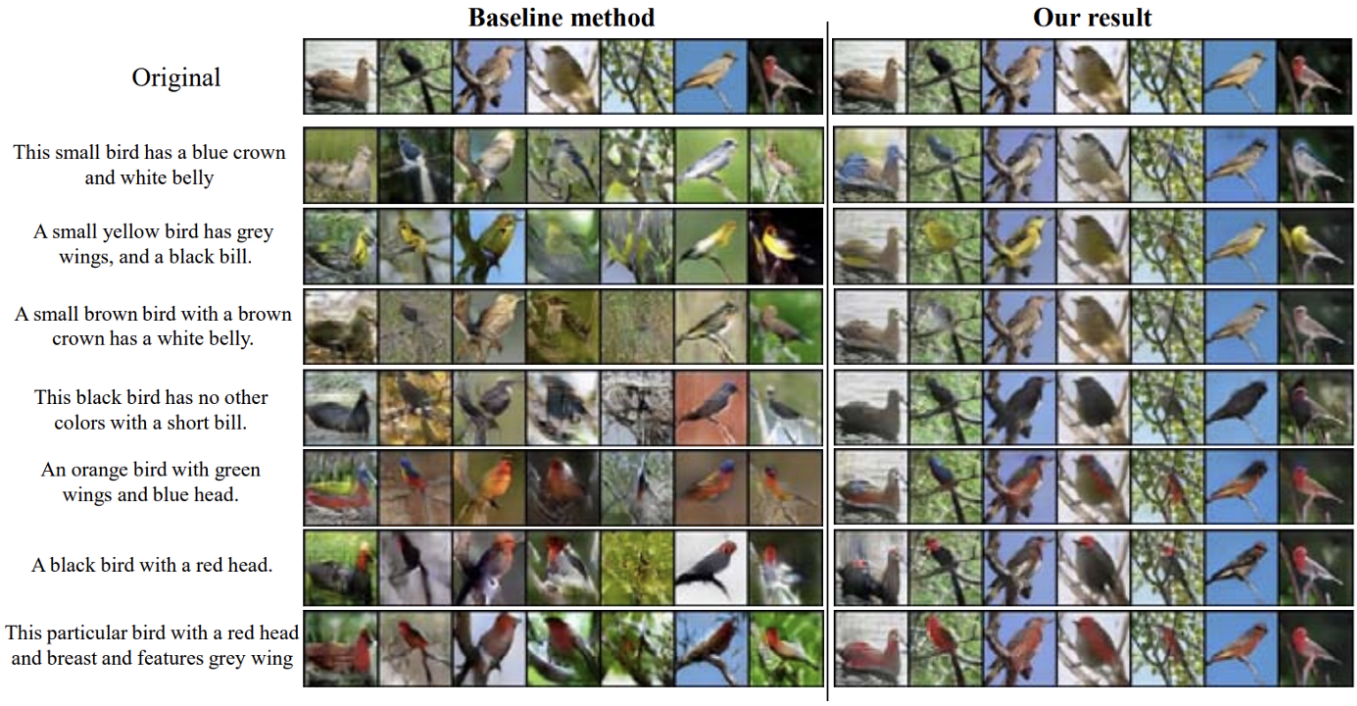
- **(Conditional) Synthesis** — This includes font generation, Text2Image, as well as 3D Object generation.
- **Data Augmentation** — Aiming to reduce the need for labeled data (GAN is only used as a tool for enhancing the training process of another model).

Conditional Synthesis

Conditional synthesis is fairly incredible in my opinion. Arguably the most fascinating application of conditional synthesis is Image2Text and Text2Image, which is able to translate a picture into words (the whole picture says a thousand words malarky..), or vice versa.

The applications of this are far-reaching, if not only for the analysis of medical images to describe the features of the image, thus removing the subjective analysis of images by doctors (this is actually something I am currently trying looking into myself with mammograms as part of a side project).

This also works the other way (assuming we give our networks words that it understands), and images can be generated purely from words. Here is an example of a Multi-Condition GAN (MC-GAN) used to perform this text-to-image conditional synthesis:



Implementation of MC-GAN for translating words into images. Source: https://arxiv.org/pdf/1902.06068.pdf

Data Augmentation

This one is fairly self-explanatory. GANs learn a data generating distribution like we did when looking at VAEs. Because of this, we are able to sample from our generator and produce additional samples which we can use to augment our training set. Thus, GANs provide an additional method to perform data augmentation (besides rotating and distorting images).

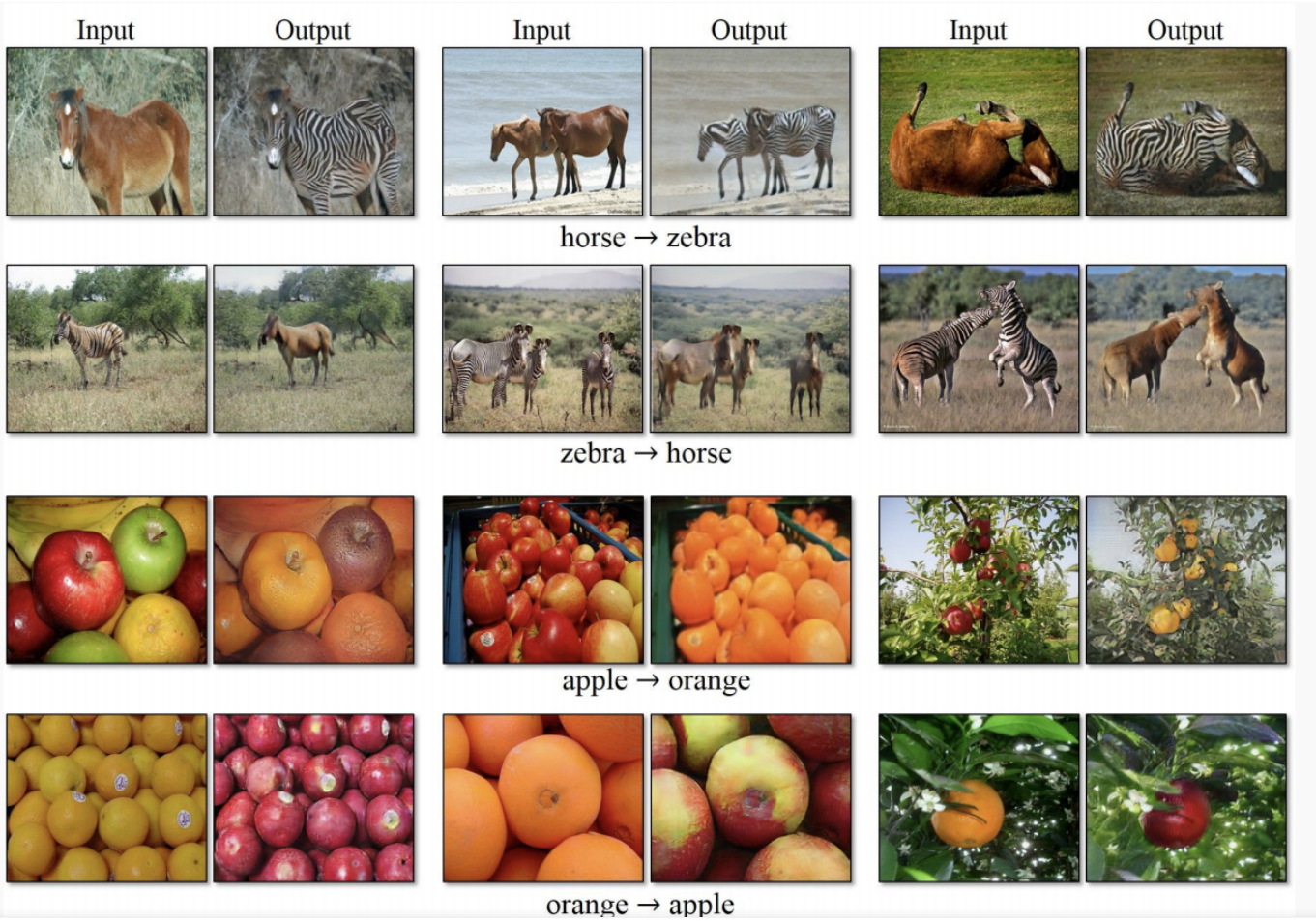
Style Transfer and Manipulation

Style transfer is quite self-explanatory. It involves the transfer of the 'style' of one image onto another image. This is very similar to neural style transfer, which I will be discussing in a future article.

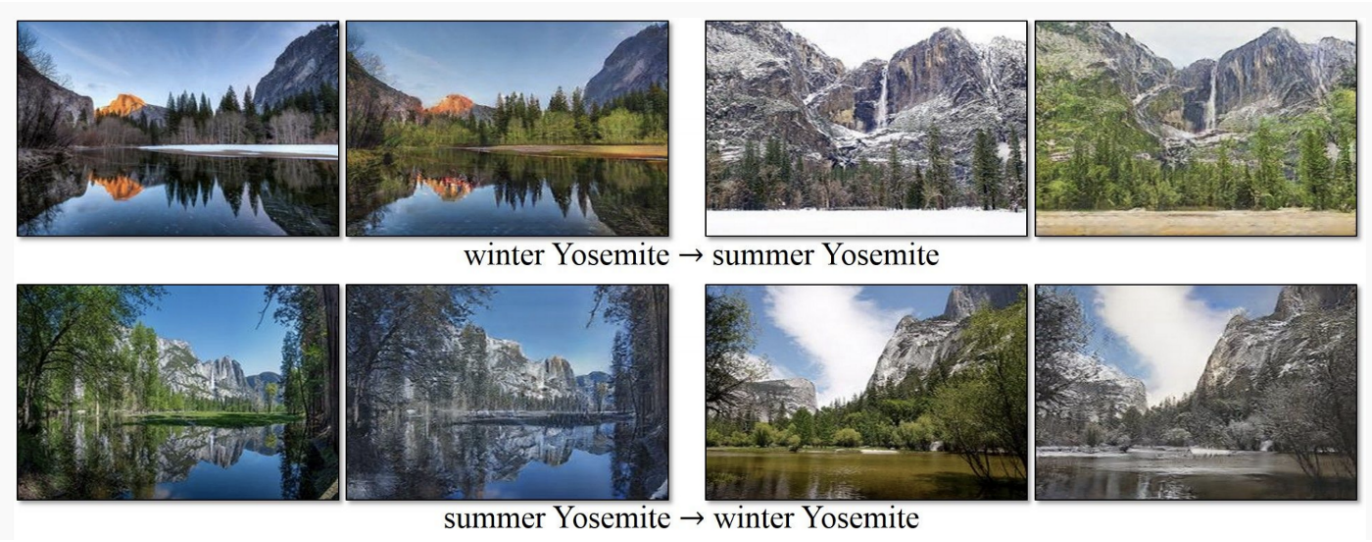
Example of style transfer using a GAN.

This works very nicely for background scenery and is similar to image filtering except that we can manipulate aspects of the actual image (compare the clouds in the above images for the input and output).

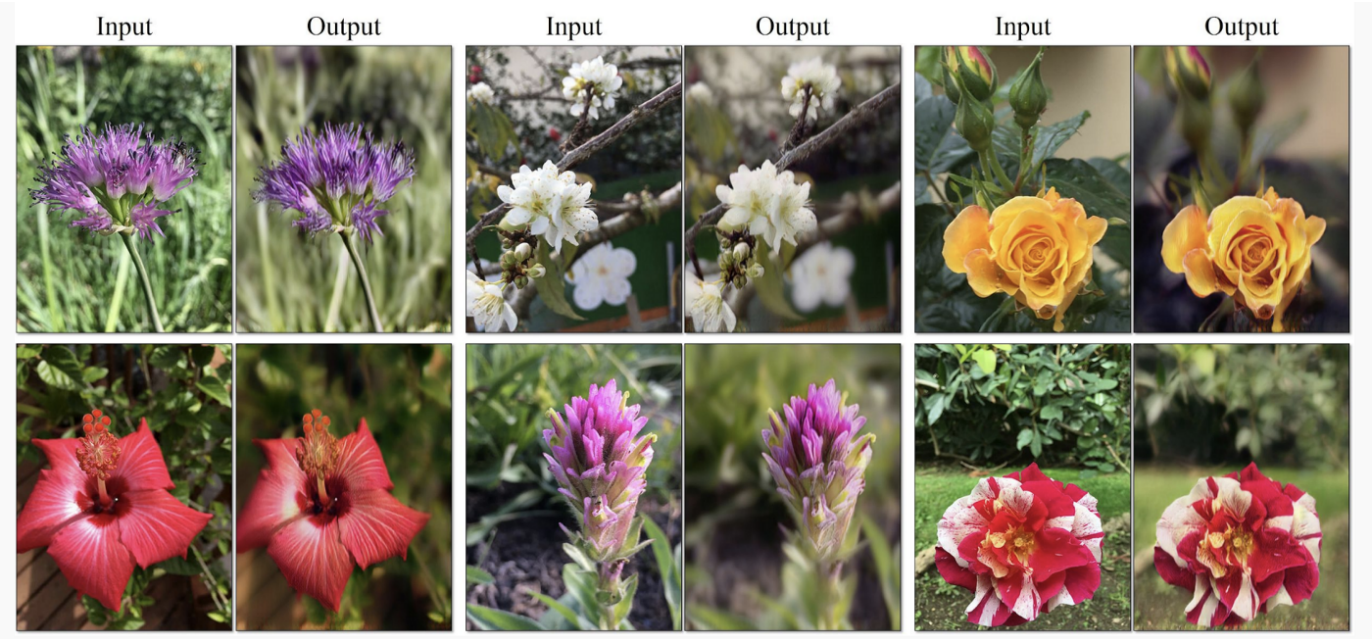
How do GANs perform on other objects such as animals or fruit?



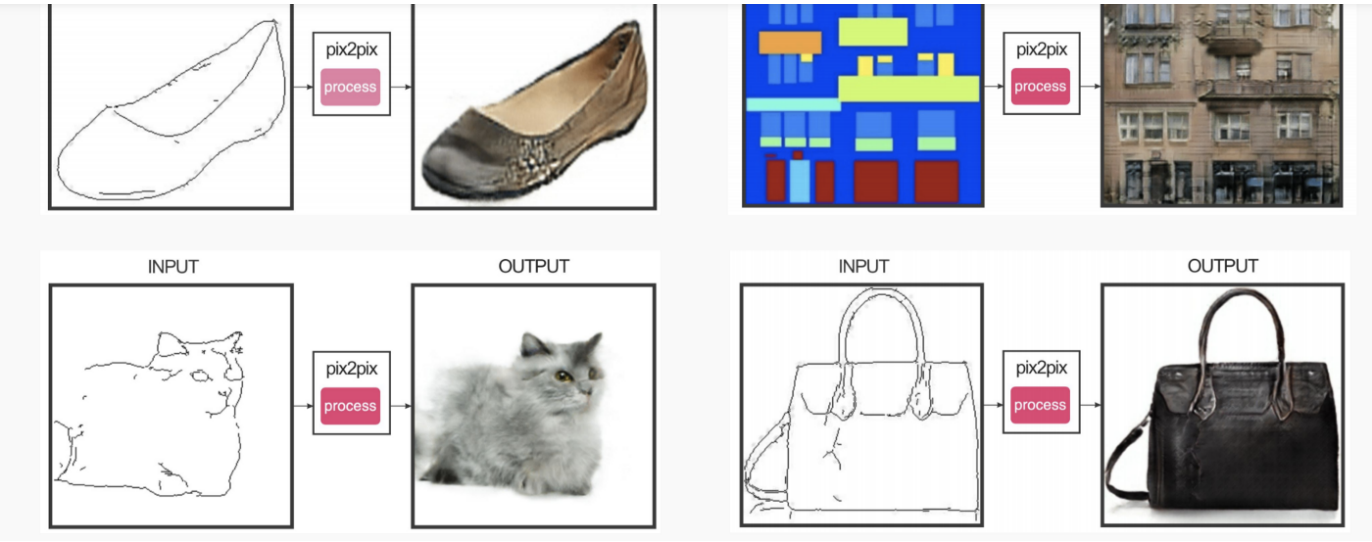
We can also change scenery to manipulate the season, a potentially useful manipulation for things like video games and virtual reality simulators.

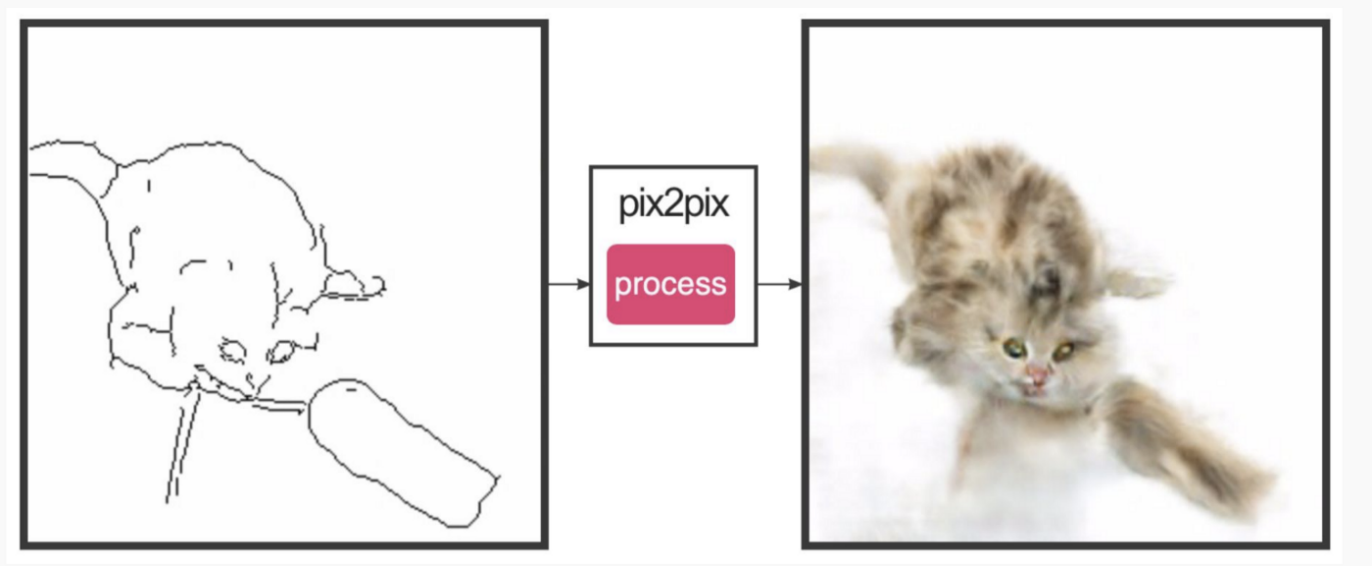


We can also change the depth of field using GANs.

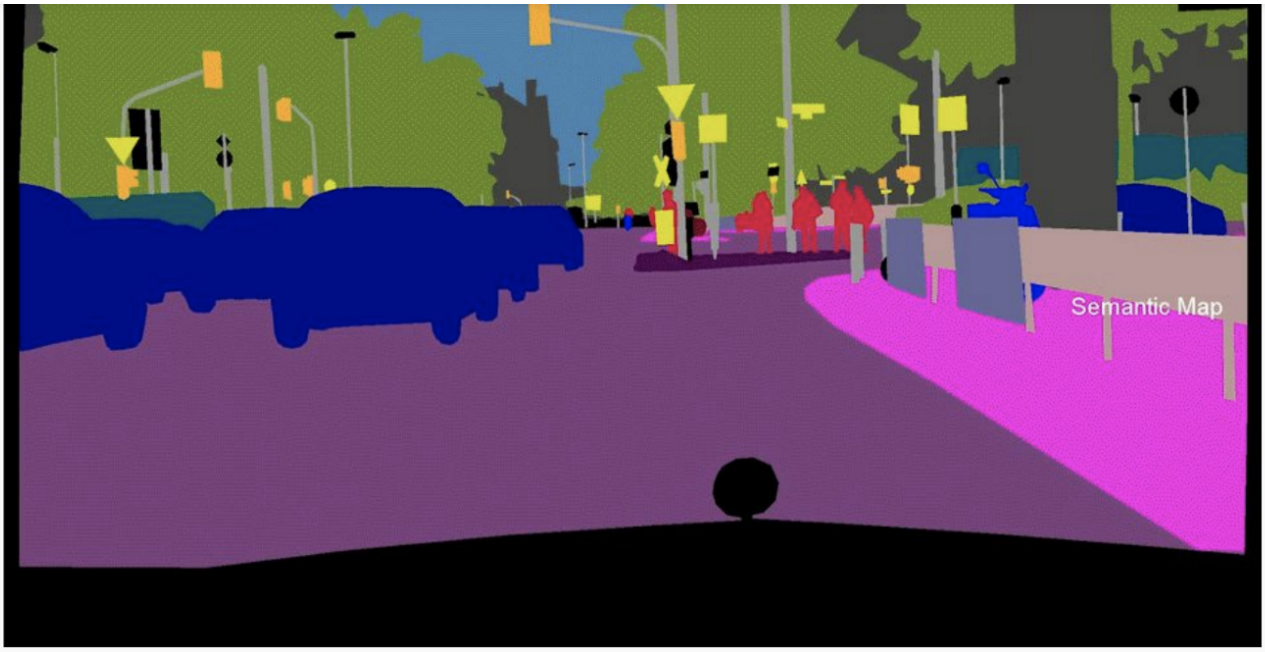


We can also manipulate drawings to turn them into real objects with relative ease (however, this probably requires substantially more drawing skills than I currently possess).





Self-driving cars see the world in a similar view to the image below, which allows objects to be viewed in a more contrasting manner (typically called a semantic map).



We can even do style transfer to render images like the environment of Grand Theft Auto (for any fans out there).

We can also transfer day into night in the same fashion.

That is enough about style transfer and image manipulation. There are many good applications of this but we have already seen several malicious uses of the technology, with people impersonating political figures and creating fake phone conversations, emails, etc.

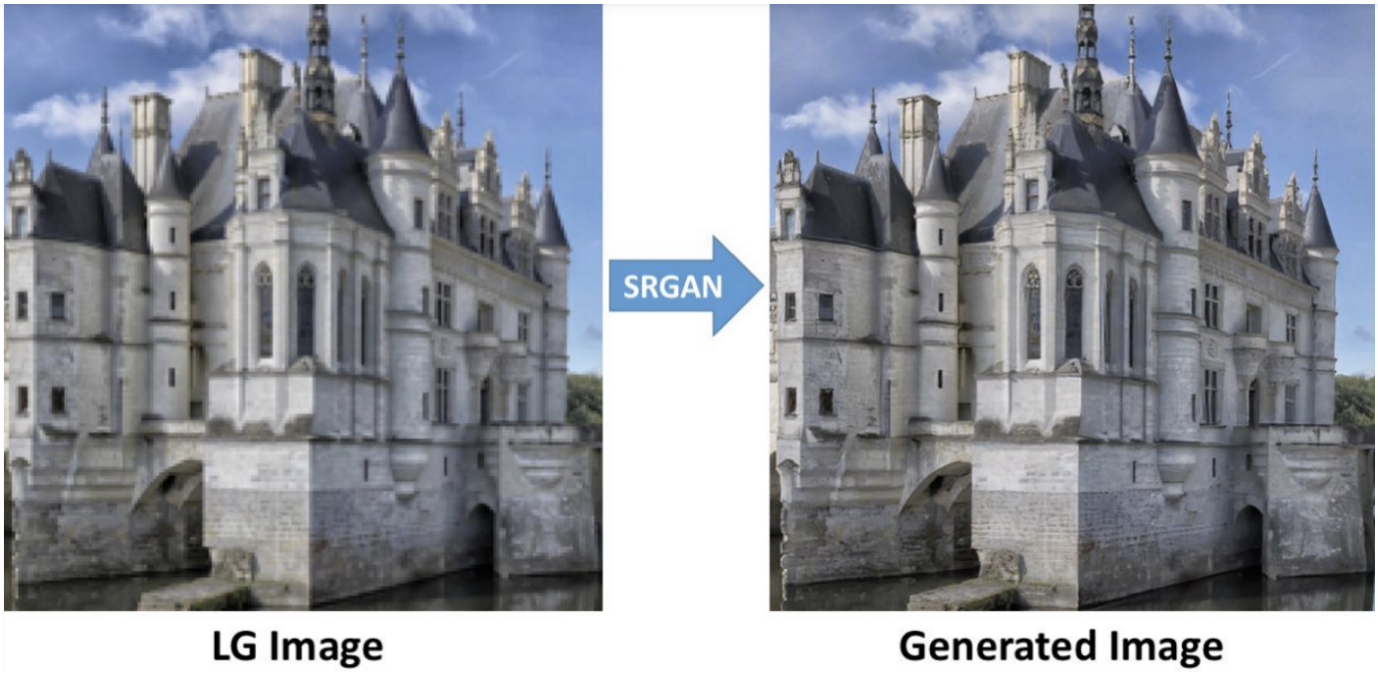
This is actually such a problem that the U.S. military is developing a new field of forensics to study videos and similar examples of media in order to determine whether they were generated by GANs (oh my, what a world we live in..).

Image Super-Resolution

Image super-resolution (SR), which refers to the process of recovering high-resolution (HR) images from low-resolution (LR) images, is an important class of image processing techniques in computer vision and image processing.

In general, this problem is very challenging and inherently ill-posed since there are always multiple HR images corresponding to a single LR image.

A variety of deep learning methods have been applied to tackle SR tasks, ranging from the early Convolutional Neural Networks (CNN) based method (e.g., SRCNN) to recent promising SR approaches using GANs (e.g., SRGAN). In general, the family of SR algorithms using deep learning techniques differs from each other in the following major aspects: different types of network architectures, different types of loss functions, different types of learning principles and strategies, etc.



The procedure to do this is actually quite involved, so I will not go into too much detail in this article (although if readers are interested I will cover this in the future).

A comprehensive overview of style transfer with GANs can be found here.

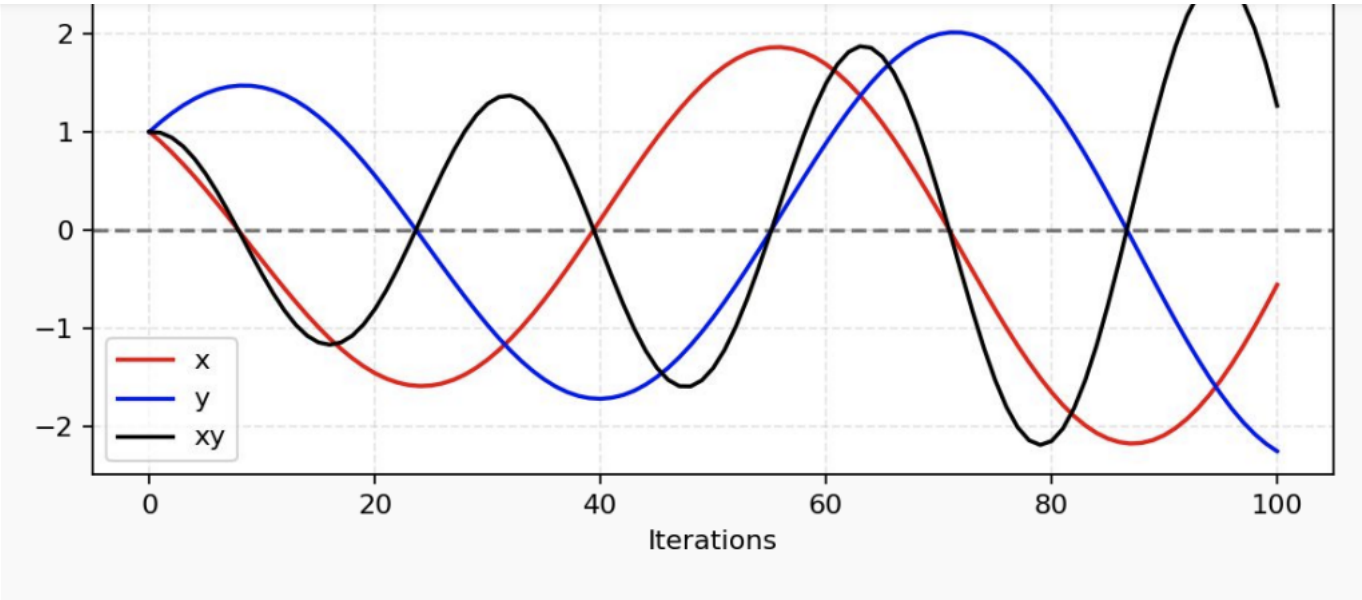
Problems with GANs

We discussed some of the most fundamental issues with GANs in the previous article, mainly the high amount of computational power necessary, the difficulty of training on large images, sensitivity, as well as modal collapse.

I wanted to reiterate these problems because training a GAN is very difficult and time-consuming. There are so many other problems with GANs that in academia (at least in Harvard), there is a running joke that if you want to train a GAN, you pick an unsuspecting and naive graduate student to do it for you (I have been on the receiving end of such a joke).

Oscillations

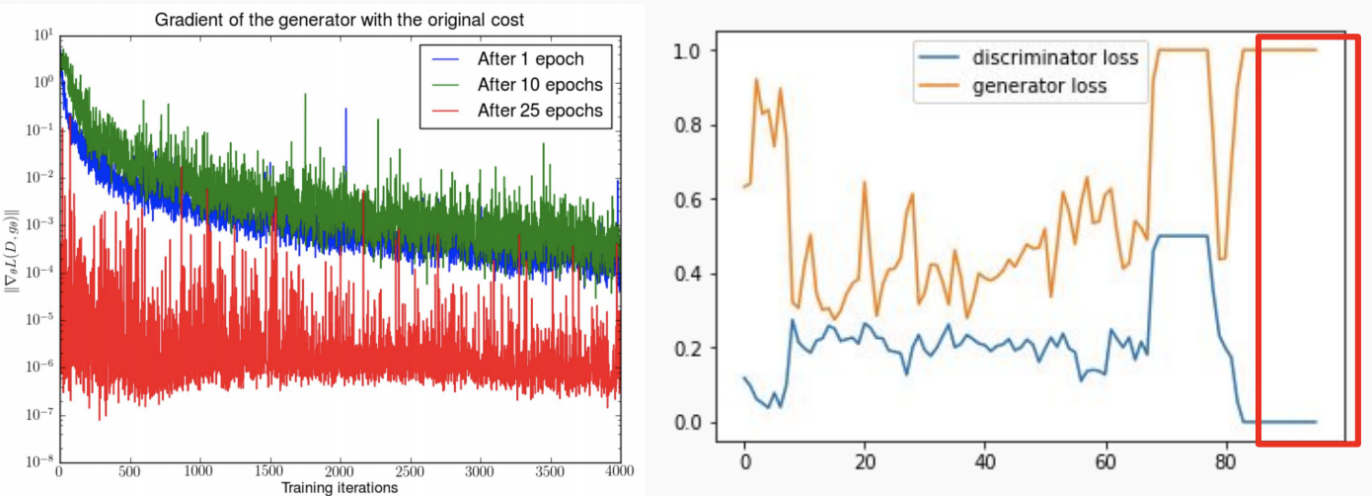
Oscillations can occur when both the generator and discriminator are jointly searching for equilibrium, but the model updates are independent. There is no theoretical guarantee of convergence and in fact, the outcome can oscillate.



The solution to this is an extensive hyperparameter-search, which sometimes may require manual intervention. An extensive hyperparameter-search on something that already takes 10 hours to run is not something I recommend lightly.

Vanishing Gradient

It is possible for the discriminator to become too strong to provide a signal for the generator. What do I mean by this? If the generator gets too good too fast (it rarely occurs the other way around), the generator can learn to fool the discriminator consistently and will stop needing to learn anything.



A GAN exhibiting the vanishing gradient problem.

The solution is not to pre-train the discriminator, or lower its learning rate compared to that of the generator. One can also change the number of updates for generator/discriminator per iteration (as I recommended in part 1).

It is fairly easy to see when a GAN has converged as a stabilization of the two networks will occur somewhere in the middle ground (i.e. one of the networks is not dominating).

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

The minimax expression for Nash equilibrium in a GAN.

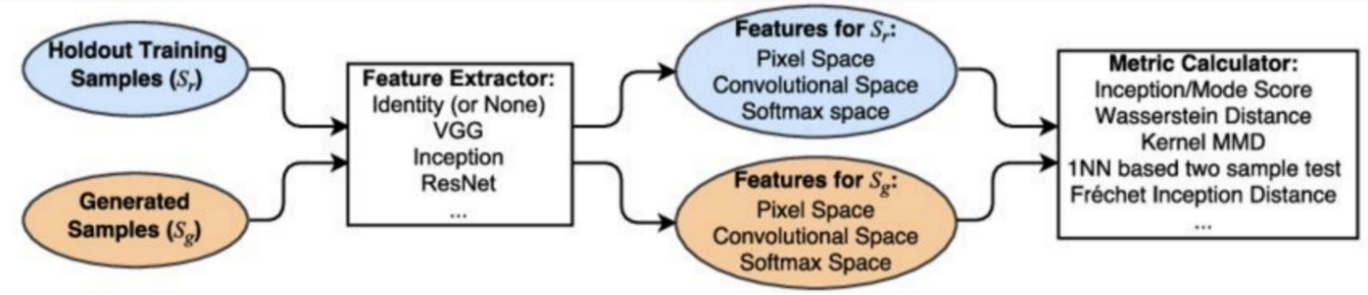
Modal collapse

Five of the above generated images look identical, and several others seem to occur in pairs.

The solution to this is to encourage diversity through mini-batch discrimination (presenting the whole batch to the discriminator for review) or by feature matching (i.e. adding a generator penalty for low diversity) or to use multiple GANs.

Evaluation Metrics

This is one I did not mention in the previous article. GANs are still evaluated on a very qualitative basis — essentially, does this image look good? Defining proper metrics that are somewhat objective is surprisingly challenging. How does a "good" generator look?



There is no definitive solution for this and it is still an active research field and quite domain specific. Strong classification models are commonly used to judge the quality of generated samples. Two common scores that are used are the **inception score**, and the **TSTR score** (Train on Synthetic, Test on Real).

Jonathan Hui has a fairly comprehensive article explaining the most common metrics used to evaluate GAN performance, which you can find here:

GAN — How to measure GAN performance?
In GANs, the objective function for the generator and the discriminator usually measures how well they are doing…
medium.com

In the next section, I will outline some of the most important types of GANs that are surfacing from the academic field.

Other Types of GANs

There are many other types of GAN that have surfaced for tackling domain-specific problems as well as different types of data (for example, time series, images, or normal csv-style data).

The types of GAN I will discuss in this section are:

- Wasserstein GAN
- CycleGAN
- Conditional GAN (briefly discussed previously)

Using the standard GAN formulation, we have already observed that training is extremely unstable. The discriminator often improves too quickly for the generator to catch up, which is why we need to regulate the learning rates or perform multiple epochs on one of the two networks. To get a decent output we need careful balancing and even then, modal collapse is quite frequent.

Source of information: Arjovsky, M., Chintala, S. and Bottou, L., 2017. Wasserstein GAN. arXiv preprint arXiv:1701.07875.

In general, generative models seek to minimize the distance between real and learned distribution (distance is everything!). The Wasserstein (also EM, Earth-Mover) distance, in informal terms, refers to when distributions are interpreted as two different ways of piling up a certain amount of dirt over a region *D*. The Wasserstein distance is the minimum cost of turning one pile into the other; where the cost is assumed to be the amount of dirt moved times the distance by which it is moved.

Unfortunately, the exact computation is intractable in this case. However, we can use CNNs to approximate the Wasserstein distance. Here, we reuse the discriminator, whose outputs are now unbounded. We define a custom loss function corresponding to the Wasserstein loss:

```
def wasserstein_loss(y_true, y_pred):
    return K.mean(y_true * y_pred)
```

What is the idea here? We can make predictions for one type as large as possible, and predictions for other types as small as possible.

The authors of the Wasserstein paper claim that:

- Higher stability during training, less need for carefully balancing generator and discriminator.
- Meaningful loss metric, correlating well with sample quality.
- Modal collapse is rare.

Tips for implementing Wasserstein GAN in Keras.

- Leave the discriminator output unbounded, i.e. apply linear activation.
- Initialize with small weights to not run into clipping issues from the start.
- Remember to run sufficient discriminator updates. This is crucial in the WGAN setup.
- You can use the Wasserstein surrogate loss implementation.
- Clip discriminator weights by implementing your own Keras constraint.

```
class WeightClip(keras.constraints.Constraint):
    def __init__(self, c):
        self.c = c

    def __call__(self, p):
        return K.clip(p, -self.c, self.c)

    def get_config(self):
        return {'name': self.__class__.__name__, 'c': self.c}
```

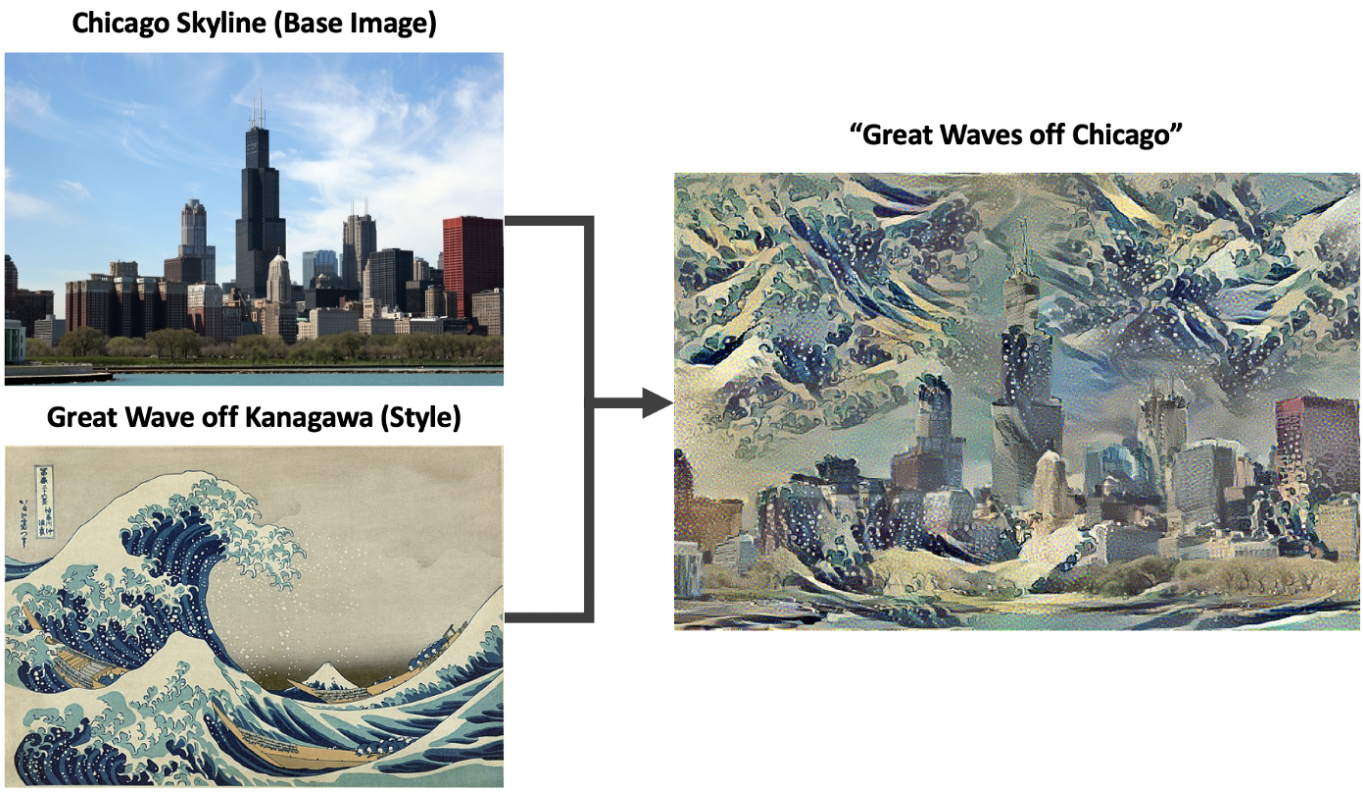
This is a complicated subject, and implementing a Wasserstein GAN is not a trivial task. If you are interested in pursuing one of these for a personal project I recommend reading the original paper that I referenced previously.

CycleGAN

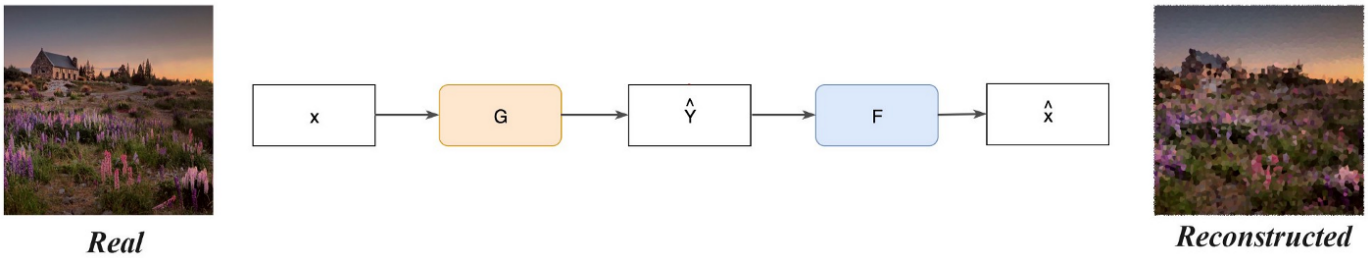
Remember the first image we saw of a horse being interchanged with a zebra? That was a CycleGAN. CycleGANs transfer styles to images. This is essentially the same idea as performing neural style transfer, which I will cover in a future article.

As an example, imagine taking a famous, such as a picture of the Golden Gate Bridge, and then extracting the style from another image, which could be a famous painting, and redrawing the picture of the bridge in the style of the said famous painting.

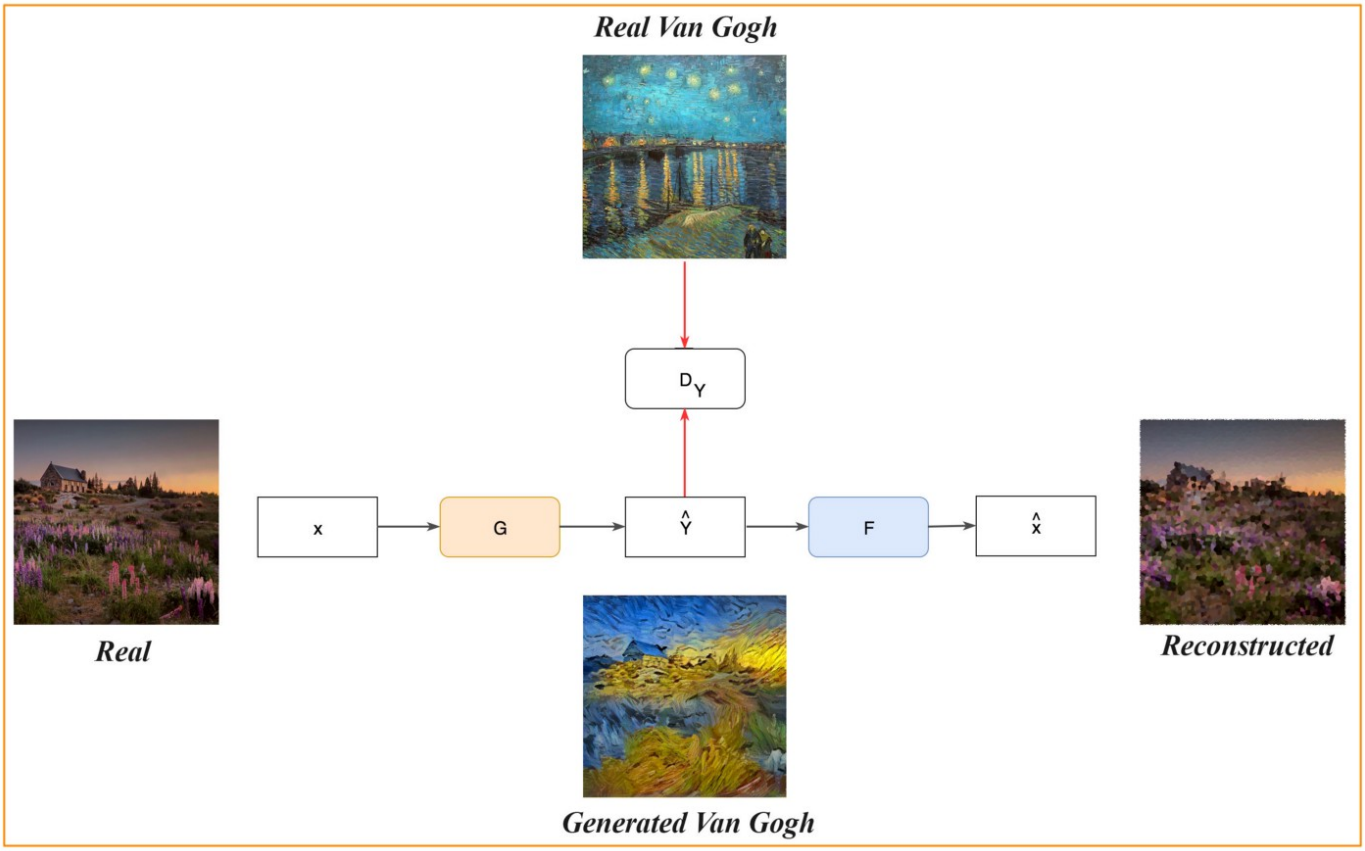
As a teaser for my neural transfer learning article, here is an example of one I did previously.



Combining the style of the famed “Great Wave off Kanagawa” with the Chicago skyline.



The difference here is that we do not actually care about reconstructing the images, we are trying to mix the styles of the two images. In the GAN implementation, a discriminator ***D*** is added to an existing design to guide the generator network to perform better. ***D*** acts as a critic between the training samples and the generated images. Through this criticism, we use backpropagation to modify the generator to produce images that address the shortcoming identified by the discriminator. In this problem, we introduce a discriminator ***D*** to make sure ***Y*** resemble Van Gogh's paintings.

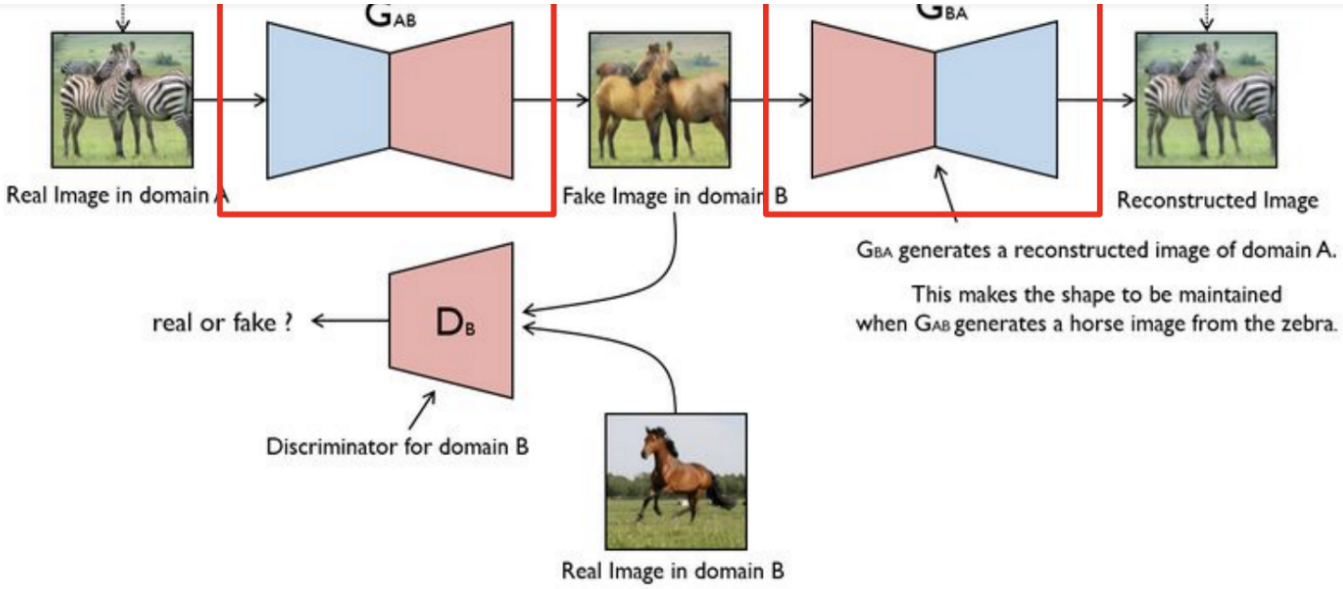


CycleGANs transfer pictures from one domain to another. To transform pictures between real images and Van Gogh paintings. We build three networks.

- A generator ***G*** to convert a real image to a Van Gogh style picture.
- A generator ***F*** to convert a Van Gogh style picture to a real image.
- A discriminator ***D*** to identify real or generated Van Gogh pictures.

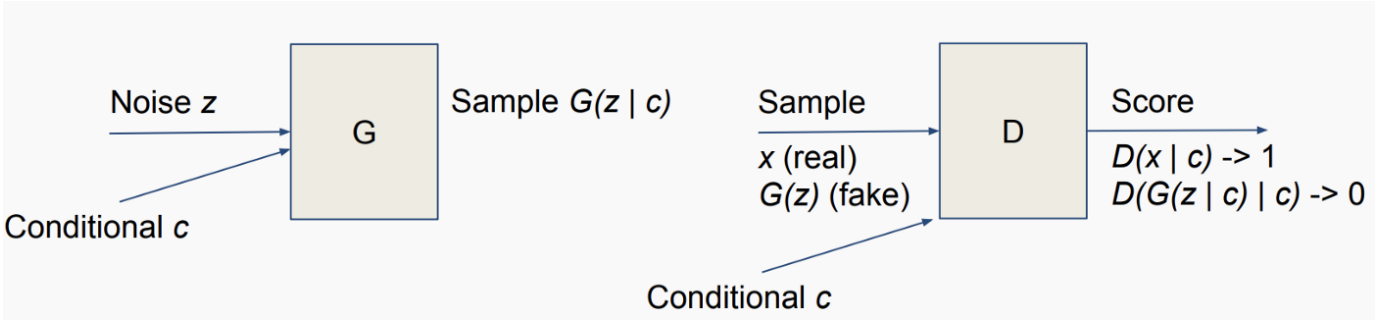
For the reverse direction, we just reverse the data flow and build an additional discriminator to identify real images.

An example of this implementation in creating the zebra/horse images is shown below.



Conditional GAN

As in VAEs, GANs can simply be conditioned to generate a certain mode of data. A great reference paper for conditional GANs can be found here.



We can extend the generative model of a GAN to a conditional model if both the generator and discriminator are conditioned on some extra information *c*. This *c* could be any kind of auxiliary information, such as class labels or data from other modalities. We can perform the conditioning by feeding *c* into both the discriminator and generator as an additional input layer.

In the generator, the prior input noise $p(z)$, and *c* are combined in joint hidden representation, and the adversarial training framework allows for considerable flexibility in how this hidden representation is composed. In the discriminator *x* and *c* are presented as inputs and to a discriminative function.

Troubleshooting GANs

Just a brief overview of all the methods of troubleshooting that we have discussed up to now, for those of you who like summaries.

[1] Models. Make sure models are correctly defined. You can debug the discriminator alone by training on a vanilla image-classification task.

[2] Data. Normalize inputs properly to [-1, 1]. Make sure to use tanh as final activation for the generator in this case.

[3] Noise. Try sampling the noise vector from a normal distribution (not uniform).

[6] Smoothing. Apply label smoothing to avoid overconfidence when updating the discriminator, i.e. set targets for real images to less than 1.

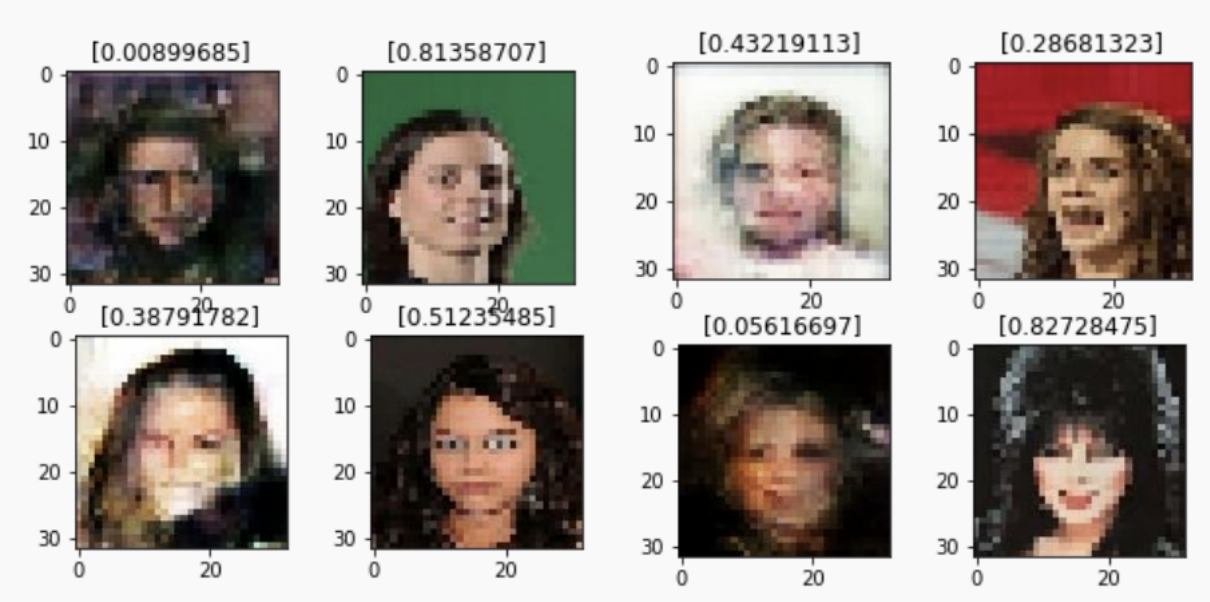
[7] Diagnostics. Monitor the magnitude of gradients constantly.

[8] Vanishing gradients. If the discriminator becomes too strong (discriminator loss = 0), try decreasing its learning rate or update the generator more often.

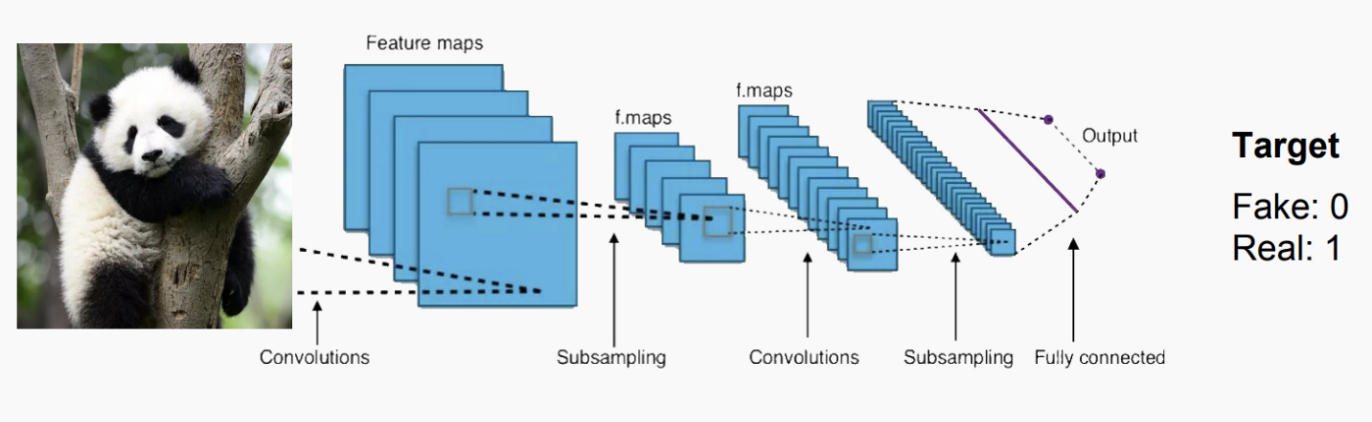
Now let's get into the fun part, actually building a GAN.

Building an Image GAN

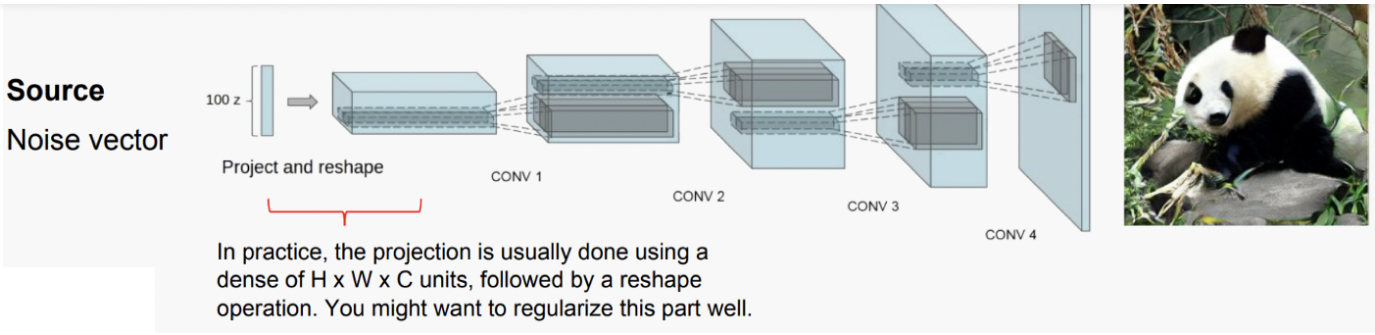
As we have already discussed several times, training a GAN can be frustrating and time-intensive. We will walk through a clean minimal example in Keras. The results are only on the proof-of-concept level to enhance understanding. In the code example, if you don't tune parameters carefully, you won't surpass this level (see below) of image generation by much:



The network takes an **image *[H, W, C]*** and outputs a **vector of *[M]***, either class scores (classification) or single score quantifying photorealism. Can be any image classification network, e.g. ResNet or DenseNet. We use minimalistic custom architecture.



Takes a vector of **noise *[N]*** and outputs an **image of *[H, W, C]***. The network has to perform synthesis. Again, we use a very minimalistic custom architecture.



It is important to define the models properly in Keras so that the weights of the respective models are fixed at the right time.

[1] Define the discriminator model, and compile it.

[2] Define the generator model, no need to compile.

[3] Define an overall model comprised of these two, setting the discriminator to not trainable before the compilation:

```
model = keras.Sequential()
model.add(generator)
model.add(discriminator)
discriminator.trainable = False
model.compile(...)
```

In the simplest form, this is all that you need to do to train a GAN.

The training loop has to be executed manually:

[1] Select R real images from the training set.

[2] Generate *F* fake images by sampling random vectors of size *N*, and predicting images from them using the generator.

[3] Train the discriminator using train_on_batch: call it separately for the batch of *R* real images and *F* fake images, with the ground truth being 1 and 0, respectively.

[4] Sample new random vectors of size *N*.

[5] Train the full model on the new vectors using train_on_batch with targets of 1. This will update the generator.

Coding Implementation

Now we will go through the above minimalistic implementation in code format using Keras on the well-known CelebA dataset. You may need to reference the above procedure if you are confused about the way the code is structured, although I will do my best to walk you through this.

The full code implementation is freely available on my corresponding GitHub repository for this 3-part tutorial.

Note that in this code I have not optimized this in any way, and I have ignored several of the rules of thumb. I will discuss these more

First, we import the necessary packages.

```
import keras
from keras.layers import *
from keras.datasets import cifar10
import glob, cv2, os
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import clear_output
```

Global Parameters

It is always best (in my opinion) to specify these parameters at the start to avoid later confusion, as these networks can get fairly messy and involved.

```
SPATIAL_DIM = 64 # Spatial dimensions of the images.
LATENT_DIM = 100 # Dimensionality of the noise vector.
BATCH_SIZE = 32 # Batchsize to use for training.
DISC_UPDATES = 1  # Number of discriminator updates per training iteration.
GEN_UPDATES = 1 # Nmber of generator updates per training iteration.

FILTER_SIZE = 5 # Filter size to be applied throughout all convolutional layers.
NUM_LOAD = 10000 # Number of images to load from CelebA. Fit also according to the available memory on
your machine.
NET_CAPACITY = 16 # General factor to globally change the number of convolutional filters.

PROGRESS_INTERVAL = 80 # Number of iterations after which current samples will be plotted.
ROOT_DIR = 'visualization' # Directory where generated samples should be saved to.

if not os.path.isdir(ROOT_DIR):
    os.mkdir(ROOT_DIR)
```

Prepare Data

We now do some image preprocessing in order to normalize the images, and also plot an image to make sure our implementation is working correctly.

```
def plot_image(x):
    plt.imshow(x * 0.5 + 0.5)

X = []
# Reference to CelebA dataset here. I recommend downloading from the Harvard 2019 ComputeFest GitHub page
(there is also some good coding tutorials here)

faces = glob.glob('../Harvard/ComputeFest 2019/celeba/img_align_celeba/*.jpg')

for i, f in enumerate(faces):
    img = cv2.imread(f)
    img = cv2.resize(img, (SPATIAL_DIM, SPATIAL_DIM))
    img = np.flip(img, axis=2)
    img = img.astype(np.float32) / 127.5 - 1.0
    X.append(img)
    if i >= NUM_LOAD - 1:
        break
X = np.array(X)
plot_image(X[4])
X.shape, X.min(), X.max()
```

Define Architecture

The architecture is fairly simple in the Keras format. It is good to write the code in blocks to keep things as simple as possible.

```
def add_encoder_block(x, filters, filter_size):
    x = Conv2D(filters, filter_size, padding='same')(x)
    x = BatchNormalization()(x)
    x = Conv2D(filters, filter_size, padding='same', strides=2)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(0.3)(x)
    return x
```

Followed by the discriminator itself — notice how we have recycled the encoder block segment and are gradually increasing the filter size to solve the problem we previously discussed for training on (large) images (best practice to do it for all images).

```
def build_discriminator(start_filters, spatial_dim, filter_size):
    inp = Input(shape=(spatial_dim, spatial_dim, 3))

    # Encoding blocks downsample the image.
    x = add_encoder_block(inp, start_filters, filter_size)
    x = add_encoder_block(x, start_filters * 2, filter_size)
    x = add_encoder_block(x, start_filters * 4, filter_size)
    x = add_encoder_block(x, start_filters * 8, filter_size)

    x = GlobalAveragePooling2D()(x)
    x = Dense(1, activation='sigmoid')(x)
    return keras.Model(inputs=inp, outputs=x)
```

Now for the decoder block segment. This time we are performing the opposite of the convolutional layers, i.e. deconvolution. Notice that the strides and padding are the same for ease of implementation, and we again use batch normalization and leaky ReLU.

```
def add_decoder_block(x, filters, filter_size):
    x = Deconvolution2D(filters, filter_size, strides=2, padding='same')(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(0.3)(x)
    return x
```

Now build the generator, notice that this time we are using decoder blocks and gradually decreasing the filter size.

```
def build_generator(start_filters, filter_size, latent_dim):
    inp = Input(shape=(latent_dim,))

    # Projection.
    x = Dense(4 * 4 * (start_filters * 8), input_dim=latent_dim)(inp)
    x = BatchNormalization()(x)
    x = Reshape(target_shape=(4, 4, start_filters * 8))(x)

    # Decoding blocks upsample the image.
    x = add_decoder_block(x, start_filters * 4, filter_size)
    x = add_decoder_block(x, start_filters * 2, filter_size)
    x = add_decoder_block(x, start_filters, filter_size)
    x = add_decoder_block(x, start_filters, filter_size)

    x = Conv2D(3, kernel_size=5, padding='same', activation='tanh')(x)
    return keras.Model(inputs=inp, outputs=x)
```

Training

Now that we have set up the network architectures, we can outline the training procedure, this can be where people get confused easily. I think this is probably because there are functions within functions within more functions.

```
def construct_models(verbose=False):
    # 1. Build discriminator.
    discriminator = build_discriminator(NET_CAPACITY, SPATIAL_DIM, FILTER_SIZE)
```

```
    # 3. Build full GAN setup by stacking generator and discriminator.
    gan = keras.Sequential()
    gan.add(generator)
    gan.add(discriminator)
    discriminator.trainable = False # Fix the discriminator part in the full setup.
    gan.compile(loss='binary_crossentropy', optimizer=keras.optimizers.Adam(lr=0.0002), metrics=['mae'])

    if verbose: # Print model summaries for debugging purposes.
        generator.summary()
        discriminator.summary()
        gan.summary()
    return generator, discriminator, gan
```

Essentially, what we did above was design a function that creates a GAN model based on our global parameters. Notice that we compile the discriminator, but not the generator, but we do compile the GAN as a whole at the end.

Also, notice that we have set the discriminator not to be trainable, this is a very common thing people forget when constructing GANs!

The reason this is so important is that you cannot train both networks at once, it is like trying to calibrate something with multiple variables changing, you will get sporadic results. You require a fixed setup for the rest of the model when training any single network.

Now the full model is constructed, we can get on with the training.

```
def run_training(start_it=0, num_epochs=1000):

# Save configuration file with global parameters

    config_name = 'gan_cap' + str(NET_CAPACITY) + '_batch' + str(BATCH_SIZE) + '_filt' + str(FILTER_SIZE)
+ '_disc' + str(DISC_UPDATES) + '_gen' + str(GEN_UPDATES)
    folder = os.path.join(ROOT_DIR, config_name)

    if not os.path.isdir(folder):
        os.mkdir(folder)

# Initiate loop variables

    avg_loss_discriminator = []
    avg_loss_generator = []
    total_it = start_it

    # Start of training loop
    for epoch in range(num_epochs):
        loss_discriminator = []
        loss_generator = []
        for it in range(200):

            # Update discriminator.
            for i in range(DISC_UPDATES):
                # Fetch real examples (you could sample unique entries, too).
                imgs_real = X[np.random.randint(0, X.shape[0], size=BATCH_SIZE)]

                # Generate fake examples.
                noise = np.random.randn(BATCH_SIZE, LATENT_DIM)
                imgs_fake = generator.predict(noise)

                d_loss_real = discriminator.train_on_batch(imgs_real, np.ones([BATCH_SIZE]))[1]
                d_loss_fake = discriminator.train_on_batch(imgs_fake, np.zeros([BATCH_SIZE]))[1]

            # Progress visualizations.
            if total_it % PROGRESS_INTERVAL == 0:
                plt.figure(figsize=(5,2))
                # We sample separate images.
                num_vis = min(BATCH_SIZE, 8)
                imgs_real = X[np.random.randint(0, X.shape[0], size=num_vis)]
                noise = np.random.randn(num_vis, LATENT_DIM)
                imgs_fake = generator.predict(noise)
                for obj_plot in [imgs_fake, imgs_real]:
                    plt.figure(figsize=(num vis * 3, 3))
```

```
                    if obj_plot is imgs_fake:
                        plt.savefig(os.path.join(folder, str(total_it).zfill(10) + '.jpg'), format='jpg',
bbox_inches='tight')
                    plt.show()

            # Update generator.
            loss = 0
            y = np.ones([BATCH_SIZE, 1])
            for j in range(GEN_UPDATES):
                noise = np.random.randn(BATCH_SIZE, LATENT_DIM)
                loss += gan.train_on_batch(noise, y)[1]

            loss_discriminator.append((d_loss_real + d_loss_fake) / 2.)
            loss_generator.append(loss / GEN_UPDATES)
            total_it += 1

        # Progress visualization.
        clear_output(True)
        print('Epoch', epoch)
        avg_loss_discriminator.append(np.mean(loss_discriminator))
        avg_loss_generator.append(np.mean(loss_generator))
        plt.plot(range(len(avg_loss_discriminator)), avg_loss_discriminator)
        plt.plot(range(len(avg_loss_generator)), avg_loss_generator)
        plt.legend(['discriminator loss', 'generator loss'])
        plt.show()
```

The above code may look very confusing to you. There are several items in the above code that are just helpful for managing the running of the GAN. For example, the first part sets up a configuration file and saves it, so you are able to reference it in the future and know exactly what the architecture and hyperparameters of your network were.

There is also the progress visualization steps, which prints the output of the notebook in real-time so that you can access the current performance of the GAN.

If you ignore the configuration file and progress visualization, the code is relatively simple. First, we update the discriminator, and then we update the generator, and then we iterate between these two scenarios.

Now, due to our smart use of functions, we can run the model in two lines.

```
generator, discriminator, gan = construct_models(verbose=True)
run_training()
```

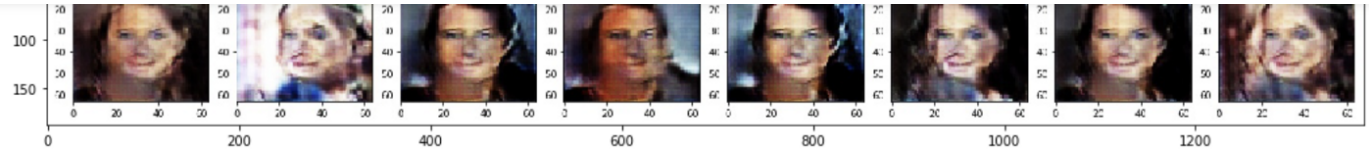
The only thing left to do is wait (possibly for hours or days) and then test the output of the network.

Sample from Trained GAN

Now after waiting for the network to finish training, we are able to take a number of samples from the network.

```
NUM_SAMPLES = 7
plt.figure(figsize=(NUM_SAMPLES * 3, 3))

for i in range(NUM_SAMPLES):
    noise = np.random.randn(1, LATENT_DIM)
    pred_raw = generator.predict(noise)[0]
    pred = pred_raw * 0.5 + 0.5
    plt.subplot(1, NUM_SAMPLES, i + 1)
    plt.imshow(pred)
plt.show()
```



Samples from trained GAN

Voila! That is our first implementation of a basic GAN. This is probably the simplest way of constructing a fully-functioning GAN.

Final Comments

In this tutorial, I have gone through some more of the advanced topics of GANs, their architectures, and their current applications, as well as covered the coding implementation of a simple GAN. In the final part of this tutorial, we will compare the performance of VAEs, GANs, and the implementation of a VAE-GAN for the purpose of generating anime images.

Thank you for reading! If you want to continue to part 3 of the tutorial, you can find it here:

GANs vs. Autoencoders: Comparison of Deep Generative Models

Want to turn horses into zebras? Make DIY anime characters or celebrities? Generative adversarial networks (GANs) are...

medium.com

Newsletter

For updates on new blog posts and extra content, sign up for my newsletter.

Newsletter Subscription

Enrich your academic journey by joining a community of scientists, researchers, and industry professionals to obtain...

mailchi.mp

Further Reading

Run BigGAN in COLAB:

- https://colab.research.google.com/github/tensorflow/hub/blob/master/examples/colab/biggan_generation_with_tf_hub.ipynb

More code help + examples:

- https://www.jessicayung.com/explaining-tensorflow-code-for-a-convolutional-neural-network/
- https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html
- https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
- https://github.com/tensorlayer/srgan
- https://junyanz.github.io/CycleGAN/ https://affinelayer.com/pixsrv/
- https://tcwang0509.github.io/pix2pixHD/

Influential Papers:

- Conditional Generative Adversarial Nets (CGAN) https://arxiv.org/pdf/1411.1784v1.pdf
- Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks (LAPGAN) https://arxiv.org/pdf/1506.05751.pdf
- Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network (SRGAN) https://arxiv.org/pdf/1609.04802.pdf
- Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks (CycleGAN) https://arxiv.org/pdf/1703.10593.pdf
- InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets https://arxiv.org/pdf/1606.03657
- DCGAN https://arxiv.org/pdf/1704.00028.pdf
- Improved Training of Wasserstein GANs (WGAN-GP) https://arxiv.org/pdf/1701.07875.pdf
- Energy-based Generative Adversarial Network (EBGAN) https://arxiv.org/pdf/1609.03126.pdf
- Autoencoding beyond pixels using a learned similarity metric (VAE-GAN) https://arxiv.org/pdf/1512.09300.pdf
- Adversarial Feature Learning (BiGAN) https://arxiv.org/pdf/1605.09782v6.pdf
- Stacked Generative Adversarial Networks (SGAN) https://arxiv.org/pdf/1612.04357.pdf
- StackGAN++: Realistic Image Synthesis with Stacked Generative Adversarial Networks https://arxiv.org/pdf/1710.10916.pdf
- Learning from Simulated and Unsupervised Images through Adversarial Training (SimGAN) https://arxiv.org/pdf/1612.07828v1.pdf