Understanding Generative Adversarial Networks (GANs)

Building, step by step, the reasoning that leads to GANs.

Joseph Rocca
Jan 7, 2019

Introduction

Yann LeCun described it as "the most interesting idea in the last 10 years in Machine Learning". Of course, such a compliment coming from such a prominent researcher in the deep learning area is always a great advertisement for the subject we are talking about! To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy. And, indeed, Generative Adversarial Networks (GANs for short) have had a huge success since they were introduced in 2014 by Ian J. Goodfellow and co-authors in the article Generative Adversarial Nets.

So what are Generative Adversarial Networks ? What makes them so "interesting" ? In this post, we will see that adversarial training is an enlightening idea, beautiful by its simplicity, that represents a real conceptual progress for Machine Learning and more especially for generative models (in the same way as backpropagation is a simple but really smart trick that made the ground idea of neural networks became so popular and eRcient).

Before going into the details, let's give a quick overview of what GANs are made for. Generative Adversarial Networks belong to the set of generative models. It means that they are able to produce /to generate (we'll see how) new content. To illustrate this notion of "generative models", we can take a look at some well known examples of results obtained with GANs.

Illustration of GANs abilities by Ian Goodfellow and co-authors. These are samples generated by Generative Adversarial Networks after training on two datasets: MNIST and TFD. For both, the rightmost column contains true data that are the nearest from the direct neighboring generated samples. This shows us that the produced data are really generated and not only memorised by the network. (source: "Generative Adversarial Nets" paper)

Naturally, this ability to generate new content makes GANs look a little bit "magic", at least at Yrst sight. In the following parts, we will overcome the apparent magic of GANs in order to dive into ideas, maths and modelling behind these models. Not only we will discuss the fundamental notions Generative Adversarial Networks rely on but, more, we will build step by step and starting from the very beginning the reasoning that leads to these ideas.

Without further ado, let's re-discover GANs together!

> ***Note:*** *Although we tried to make this article as self-contained as possible, a basic prior knowledge in Machine Learning is still required. Nevertheless, most of the notions will be remained when needed and some references will be given otherwise. We really tried to make this article as smooth to read as possible. Do not hesitate to mention in the comment section what you would have liked to read more about (for possible further articles on the subject).*

Outline

In the first following section we will discuss the process of generating random variables from a given distribution. Then, in section 2 we will show, through an example, that the problems GANs try to tackle can be expressed as random variable generation problems. In section 3 we will discuss matching based generative networks and show how they answer problems described in section 2. Finally in section 4 we will introduce GANs. More especially, we will present the general architecture with its loss function and we will make the link with all the previous parts.

Generating random variables

In this section, we discuss the process of generating random variables: we remind some existing methods and more especially the inverse transform method that allows to generate complex random variables from simple uniform random variables. Although all this could seem a little bit far from our subject of matter, GANs, we will see in the next section the deep link that exists with generative models.

Uniform random variables can be pseudo-randomly generated

Computers are fundamentally deterministic. So, it is, in theory, impossible to generate numbers that are really random (even if we could say that the question "what really is randomness?" is a difficult one). However, it is possible to define algorithms that generate sequences of numbers whose properties are very close to the properties of theoretical random numbers sequences. In particular, a computer is able, using a pseudorandom number generator, to generate a sequence of numbers that approximatively follows a uniform random distribution between 0 and 1. The uniform case is a very simple one upon which more complex random variables can be built in different ways.

Random variables expressed as the result of an operation or a process

There exist different techniques that are aimed at generating more complex random variables. Among them we can find, for example, inverse transform method, rejection sampling, Metropolis-Hasting algorithm and others. All these methods rely on different mathematical tricks that mainly consist in representing the random variable we want to generate as the result of an operation (over simpler random variables) or a process.

Rejection sampling expresses the random variable as the result of a process that consist in sampling not from the complex distribution but from a well known simple distribution and to accept or reject the sampled value depending on some condition. Repeating this process until the sampled value is accepted, we can show that with the right condition of acceptance the value that will be effectively sampled will follow the right distribution.

In the Metropolis-Hasting algorithm, the idea is to find a Markov Chain (MC) such that the stationary distribution of this MC corresponds to the distribution from which we would like to sample our random variable. Once this MC found, we can simulate a long enough trajectory over this MC to consider that we have reach a steady state and then the last value we obtain this way can be considered as having been drawn from the distribution of interest.

We won't go any further into the details of rejection sampling and Metropolis-Hasting because these methods are not the ones that will lead us to the notion behind GANs (nevertheless, the interested reader can refer to the pointed Wikipedia articles and links therein). However, let's focus a little bit more on the inverse transform method.

The inverse transform method

The idea of the inverse transform method is simply to represent our complex — in this article "complex" should always be understood in the sense of "not simple" and not in the mathematical sense — random variable as the result of a function applied to a uniform random variable we know how to generate.

We consider in what follows a one-dimensional example. Let X be a complex random variable we want to sample from, and U be a uniform random variable over [0,1] we know how to sample from. We remind that a random variable is fully defined by its Cumulative Distribution Function (CDF). The CDF of a random variable is a function from the domain of definition of the random variable to the interval [0,1] and defined, in one dimension, such that

$$CDF_X(x) = \mathbb{P}(X \leq x) \qquad \in [0,1]$$

In the particular case of our uniform random variable U, we have

$$CDF_U(u) = \mathbb{P}(U \leq u) = u \qquad \forall u \in [0,1]$$

For simplicity, we will suppose here that the function CDF_X is invertible, and its inverse is denoted

$$CDF_X^{-1}$$

(the method can easily be extended to the non-invertible case by using the generalised inverse of the function, but it is really not the main point we want to focus on here). Then if we define

$$Y = CDF_X^{-1}(U)$$

we have

$$CDF_Y(y) = \mathbb{P}(Y \leq y) = \mathbb{P}(CDF_X^{-1}(U) \leq y) = \mathbb{P}(U \leq CDF_X(y)) = CDF_X(y)$$

As we can see, Y and X have the same CDF and then define the same random variable. So, by defining Y as above (as a function of a uniform random variable) we have managed to define a random variable with the targeted distribution.

To summarise, inverse transform method is a way to generate a random variable that follows a given distribution by making a uniform random variable goes through a well-designed "transform function" (inverse CDF). This notion of "inverse transform method" can, in fact, be extended to the notion of "transform method" that consists, more generally, in generating random variables as function of some simpler random variables (not necessarily uniform and then the transform function is no longer the inverse CDF). Conceptually, the purpose of the "transform function" is to deform/reshape the initial probability distribution: the transform function takes from where the initial distribution is too high compared to the targeted distribution and puts it where it is too low.

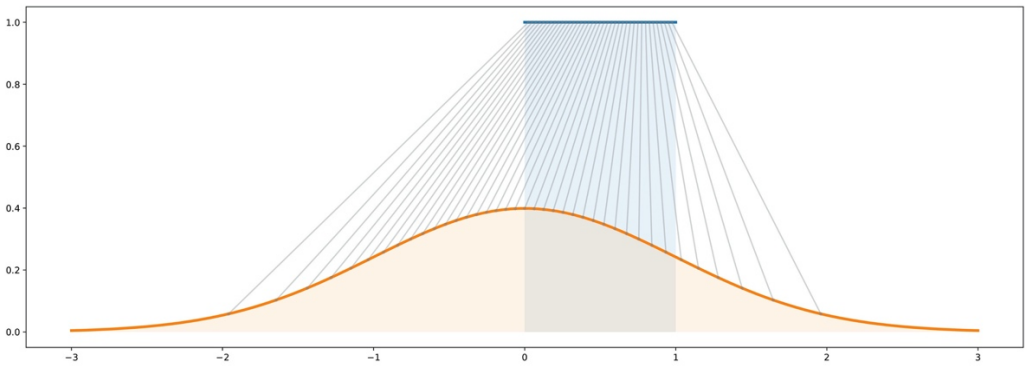


Illustration of the inverse transform method. In blue: the uniform distribution over [0,1]. In orange: the standard Gaussian distribution. In grey: the mapping from the uniform to the Gaussian distribution (inverse CDF)

Generative models

We try to generate very complex random variables…

Suppose that we are interested in generating black and white square images of dogs with a size of n by n pixels. We can reshape each data as a N = n x n dimensional vector (by stacking columns on top of each other) such that an image of dog can then be represented by a vector. However, it

doesn't mean that all vectors represent a dog once shaped back to a square! So, we can say that the N-dimensional vectors that effectively give something that look like a dog are distributed according to a very specific probability distribution over the entire N-dimensional vector space (some points of that space are very likely to represent dogs whereas it is highly unlikely for some others). In the same spirit, there exists, over this N-dimensional vector space, probability distributions for images of cats, birds and so on.

Then, the problem of generating a new image of dog is equivalent to the problem of generating a new vector following the "dog probability distribution" over the N dimensional vector space. So we are, in fact, facing a problem of generating a random variable with respect to a specific probability distribution.

At this point, we can mention two important things. First the "dog probability distribution" we mentioned is a very complex distribution over a very large space. Second, even if we can assume the existence of such underlying distribution (there actually exists images that looks like dog and others that doesn't) we obviously don't know how to express explicitly this distribution. Both previous points make the process of generating random variables from this distribution pretty difficult. Let's then try to tackle these two problems in the following.

… so let's use transform method with a neural network as function!

Our first problem when trying to generate our new image of dog is that the "dog probability distribution" over the N-dimensional vector space is a very complex one and we don't know how to directly generate complex random variables. However, as we know pretty well how to generate N uncorrelated uniform random variables, we could make use of the transform method. To do so, we need to express our N-dimensional random variable as the result of a very complex function applied to a simple N-dimensional random variable!

Here, we can emphasise the fact that finding the transform function is not as straightforward as just taking the closed-form inverse of the Cumulative Distribution Function (that we obviously don't know) as we have done when describing the inverse transform method. The transform function can't be explicitly expressed and, then, we have to learn it from data.

As most of the time in these cases, very complex function naturally implies neural network modelling. Then, the idea is to model the transform function by a neural network that takes as input a simple N-dimensional uniform random variable and that returns as output another N-dimensional random variable that should follow, after training, the right "dog probability distribution". Once the architecture of the network has been designed, we still need to train it. In the next two sections, we will discuss two ways to train these generative networks, including the idea of adversarial training behind GANs!

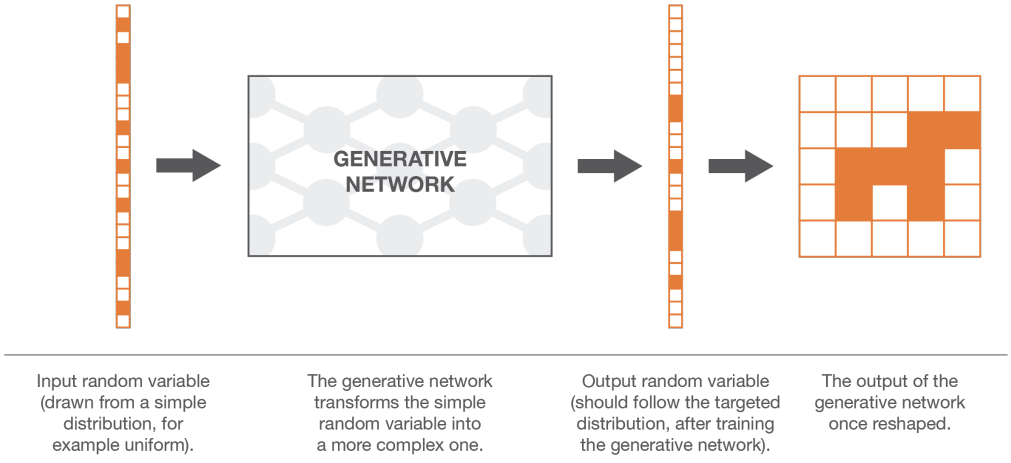


Illustration of the notion of generative models using neural networks. Obviously, the dimensionalities we are really talking about are much higher than represented here.

Generative Matching Networks

> ***Disclaimer:*** *The denomination of "Generative Matching Networks" is not a standard one. However, we can find in the literature, for example, "Generative Moments Matching Networks" or also "Generative Features Matching Networks". We just want here to use a slightly more general denomination for what we describe bellow.*

Training generative models

So far, we have shown that our problem of generating a new image of dog can be rephrased into a problem of generating a random vector in the N-dimensional vector space that follows the "dog probability distribution" and we have suggested to use a transform method, with a neural network to model the transform function.

Now, we still need to train (optimise) the network to express the right transform function. To this purpose, we can suggest two different training methods: a direct one and an indirect one. The direct training method consists in comparing the true and the generated probability distributions and backpropagating the difference (the error) through the network. This is the idea that rules Generative Matching Networks (GMNs). For the indirect training method, we do not directly compare the true and generated distributions. Instead, we train the generative network by making these two distributions go through a downstream task chosen such that the optimisation process of the generative network with respect to the downstream task will enforce the generated distribution to be close to the true distribution. This last idea is the one behind Generative Adversarial Networks (GANs) that we will present in the next section. But for now, let's start with the direct method and GMNs.

Comparing two probability distributions based on samples

As mentioned, the idea of GMNs is to train the generative network by directly comparing the generated distribution to the true one. However, we do not know how to express explicitly the true "dog probability distribution" and we can also say that the generated distribution is far too complex to be expressed explicitly. So, comparisons based on explicit expressions are not possible. However, if we have a way to compare probability distributions based on samples, we can use it to train the network. Indeed, we have a sample of true data and we can, at each iteration of the training process, produce a sample of generated data.

Although, in theory, any distance (or similarity measure) able to compare effectively two distributions based on samples can be used, we can mention in particular the Maximum Mean Discrepancy (MMD) approach. The MMD defines a distance between two probability distributions that can be computed (estimated) based on samples of these distributions. Although it is not fully out of the scope of this article, we have decided not to spend much more time describing the MMD. However, we have the project to publish soon an article that will contains more details about it. The reader that would like to know more about MMD right now can refer to these slides, this article or this article.

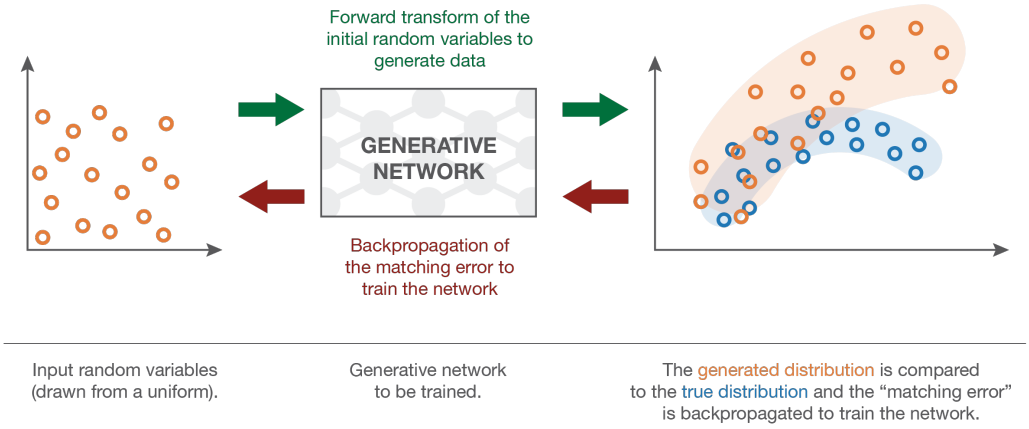
Backpropagation of the distribution matching error

So, once we have defined a way to compare two distributions based on samples, we can define the training process of the generative network in GMNs. Given a random variable with uniform probability distribution as input, we want the probability distribution of the generated output to be the "dog probability distribution". The idea of GMNs is then to optimise the network by repeating the following steps:

- generate some uniform inputs
- make these inputs go through the network and collect the generated outputs

- compare the true "dog probability distribution" and the generated one based on the available samples (for example compute the MMD distance between the sample of true dog images and the sample of generated ones)
- use backpropagation to make one step of gradient descent to lower the distance (for example MMD) between true and generated distributions

As written above, when following these steps we are applying a gradient descent over the network with a loss function that is the distance between the true and the generated distributions at the current iteration.



Generative Matching Networks take simple random inputs, generate new data, directly compare the distribution of the generated data to the distribution of the true data and backpropagate the matching error to train the network.

Generative Adversarial Networks

The "indirect" training method

The "direct" approach presented above compare directly the generated distribution to the true one when training the generative network. The brilliant idea that rules GANs consists in replacing this direct comparison by an indirect one that takes the form of a downstream task over these two distributions. The training of the generative network is then done with respect to this task such that it forces the generated distribution to get closer and closer to the true distribution.

The downstream task of GANs is a discrimination task between true and generated samples. Or we could say a "non-discrimination" task as we want the discrimination to fail as much as possible. So, in a GAN architecture, we have a discriminator, that takes samples of true and generated data and that try to classify them as well as possible, and a generator that is trained to fool the discriminator as much as possible. Let's see on a simple example why the direct and indirect approaches we mentioned should, in theory, lead to the same optimal generator.

The ideal case: perfect generator and discriminator

In order to better understand why training a generator to fool a discriminator will lead to the same result as training directly the generator to match the target distribution, let's take a simple one dimensional example. We forget, for the moment, how both generator and discriminator are represented and consider them as abstract notions (that will be specified in the next subsection). Moreover, both are supposed "perfect" (with infinite capacities) in the sense that they are not constrained by any kind of (parametrised) model.

Suppose that we have a true distribution, for example a one-dimensional gaussian, and that we want a generator that samples from this probability distribution. What we called "direct" training method would then consist in adjusting iteratively the generator (gradient descent iterations) to correct the measured difference/error between true and generated distributions. Finally, assuming the optimisation process perfect, we should end up with the generated distribution that matches exactly the true distribution.

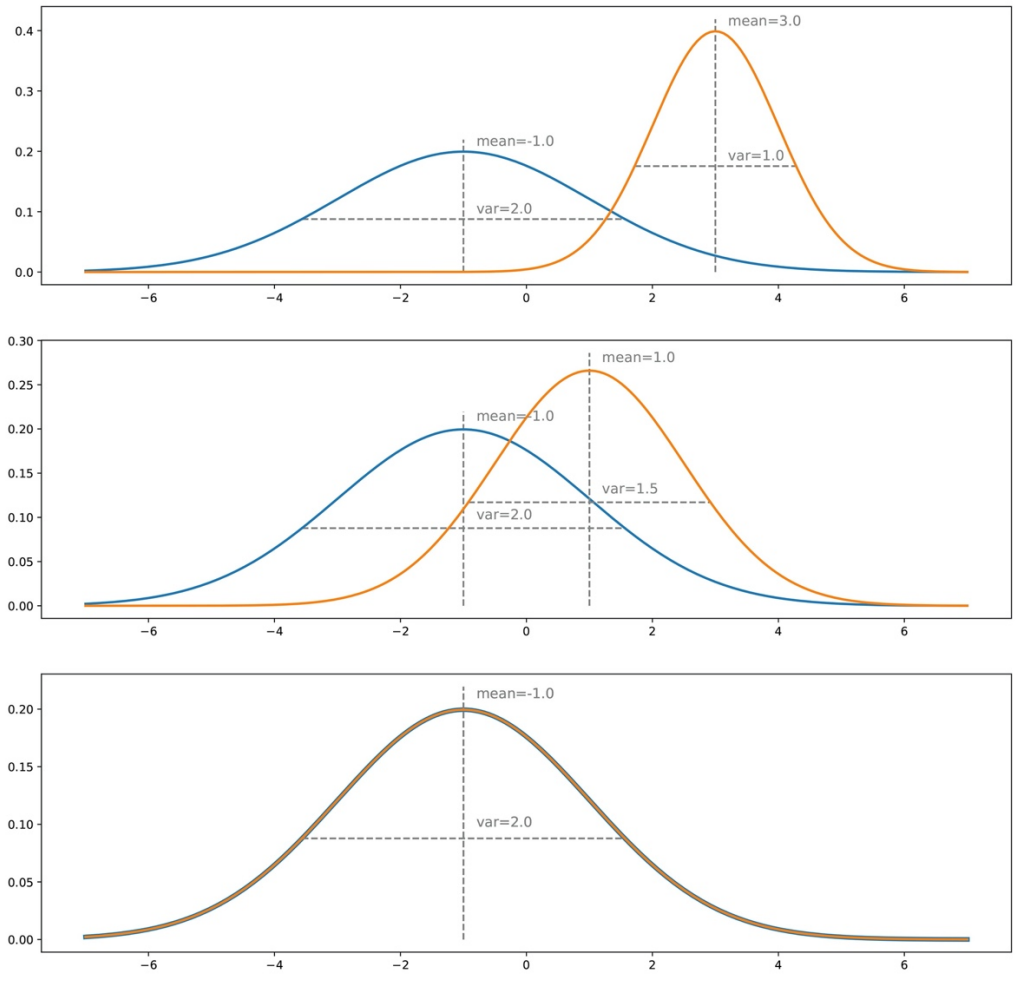
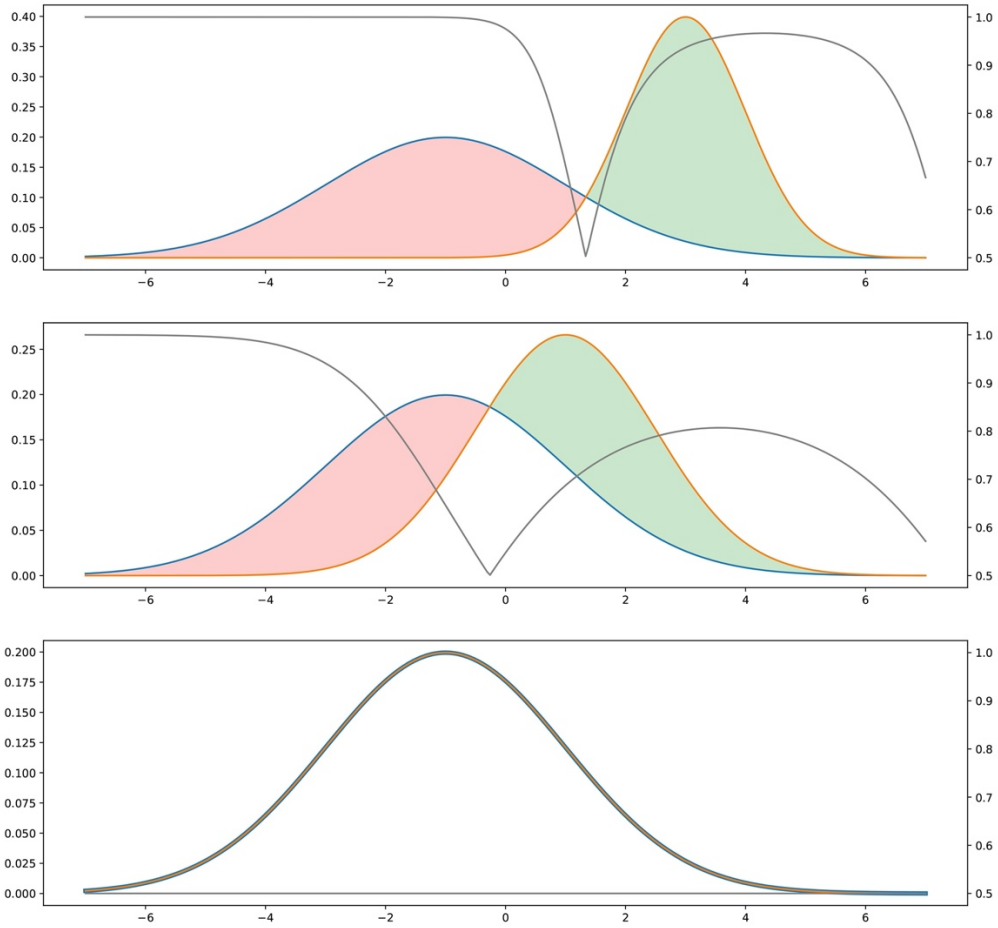


Illustration of the concept of direct matching method. The distribution in blue is the true one while the generated distribution is depicted in orange. Iteration by iteration, we compare the two distributions and adjust the networks weights through gradient descent steps. Here the comparison is done over the mean and the variance (similar to a truncated-moments matching method). Notice that (obviously) this example is so simple that it doesn't require an iterative approach: the purpose is only to illustrate the intuition given above.

For the "indirect" approach, we have to consider also a discriminator. We assume for now that this discriminator is a kind of oracle that knows exactly what the true and generated distributions are, and that is able, based on this information, to predict a class ("true" or "generated") for any given point. If the two distributions are far apart, the discriminator will be able to classify easily and with a high level of confidence most of the points we present to it. If we want to fool the discriminator, we have to bring the generated distribution close to the true one. The discriminator will have the most difficulty to predict the class when the two distributions will be equal in all points: in this case, for each point there are equal chances for it to be "true" or "generated" and then the discriminator can't do better than being true in one case out of two in average.



Intuition for the adversarial method. The blue distribution is the true one, the orange is the generated one. In grey, with corresponding y-axis on the right, we displayed the probability to be true for the discriminator if it chooses the class with the higher density in each point (assuming "true" and "generated" data are in equal proportions). The closer the two distributions are, the more often the discriminator is wrong. When training, the goal is to "move the green area" (generated distribution is too high) towards the red area (generated distribution is too low).

At this point, it seems legit to wonder whether this indirect method is really a good idea. Indeed, it seems to be more complicated (we have to optimise the generator based on a downstream task instead of directly based on the distributions) and it requires a discriminator that we consider here as a given oracle but that is, in reality, neither known nor perfect. For the first point, the difficulty of directly comparing two probability distributions based on samples counterbalances the apparent higher complexity of indirect method. For the second point, it is obvious that the discriminator is not known. However, it can be learned!

The approximation: adversarial neural networks

Let's now describe the specific form that take the generator and the discriminator in the GANs architecture. The generator is a neural network that models a transform function. It takes as input a simple random variable and must return, once trained, a random variable that follows the targeted distribution. As it is very complicated and unknown, we decide to model the discriminator with another neural network. This neural network models a discriminative function. It takes as input a point (in our dog example a N-dimensional vector) and returns as output the probability of this point to be a "true" one.

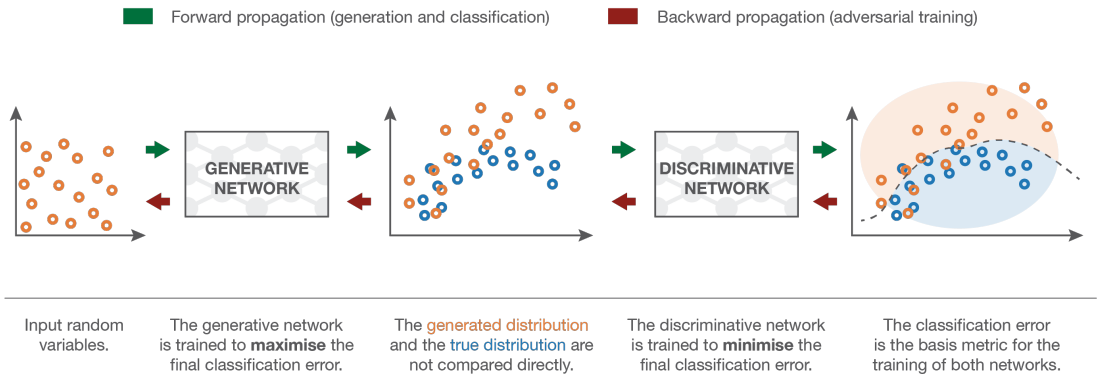
Notice that the fact that we impose now a parametrised model to express both the generator and the discriminator (instead of the idealised versions in the previous subsection) has, in practice, not a huge impact on the theoretical argument/intuition given above: we just then work in some para-

metrised spaces instead of ideal full spaces and, so, the optimal points that we should reach in the ideal case can then be seen as "rounded" by the precision capacity of then parametrised models.

Once defined, the two networks can then be trained jointly (at the same time) with opposite goals:

- the goal of the generator is to fool the discriminator, so then generative neural network is trained to maximise the final classification error (between true and generated data)
- the goal of the discriminator is to detect fake generated data, so the discriminative neural network is trained to minimise the final classification error.

So, at each iteration of the training process, the weights of the generative network are updated in order to increase the classification error (error gradient ascent over the generator's parameters) whereas the weights of the discriminative network are updated so that to decrease this error (error gradient descent over the discriminator's parameters).



Generative Adversarial Networks representation. The generator takes simple random variables as inputs and generate new data. The discriminator takes "true" and "generated" data and try to discriminate them, building a classifier. The goal of the generator is to fool the discriminator (increase the classification error by mixing up as much as possible generated data with true data) and the goal of the discriminator is to distinguish between true and generated data.

These opposite goals and the implied notion of adversarial training of the two networks explains the name of "adversarial networks": both networks try to beat each other and, doing so, they are both becoming better and better. The competition between them makes these two networks "progress" with respect to their respective goals. From a game theory point of view, we can think of this setting as a minimax two-players game where the equilibrium state corresponds to the situation where the generator produces data from the exact targeted distribution and where the discriminator predicts "true" or "generated" with probability 1/2 for any point it receives.

Mathematical details about GANs

Note: *This section is a little bit more technical and not absolutely necessary for the overall understanding of GANs. So, the readers that don't want to read some mathematics right now can skip this section for the moment. For the others, let's see how the intuitions given above are mathematically formalised.*

Disclaimer: *The equations in the following are not the ones of the article of Ian Goodfellow. We propose here another mathematical formalisation for two reasons: First, to stay a little bit closer to the intuitions given above and, second, because the equations of the original paper are already so clear that it would not have been useful to just rewrite them. Notice also that we absolutely do not enter into the practical considerations (vanishing gradient or other) related to the different possible loss functions. We highly encourage the reader to also take a look at the equations of the original paper: the main difference is that Ian Goodfellow and co-authors have worked with cross-entropy error instead of absolute error*

(as we do below). Moreover, in the following we assume a generator and a discriminator with unlimited capacity.

Neural networks modelling essentially requires to define two things: an architecture and a loss function. We have already described the architecture of Generative Adversarial Networks. It consists in two networks:

- a generative network G(.) that takes a random input z with density p_z and returns an output x_g = G(z) that should follow (after training) the targeted probability distribution
- a discriminative network D(.) that takes an input x that can be a "true" one (x_t, whose density is denoted p_t) or a "generated" one (x_g, whose density p_g is the density induced by the density p_z going through G) and that returns the probability D(x) of x to be a "true" data

Let's take now a closer look at the "theoretical" loss function of GANs. If we send to the discriminator "true" and "generated" data in the same proportions, the expected absolute error of the discriminator can then be expressed as

$$\begin{aligned} E(G,D) &= \frac{1}{2}\mathbb{E}_{x\sim p_t}[1-D(x)] + \frac{1}{2}\mathbb{E}_{z\sim p_z}[D(G(z))] \\ &= \frac{1}{2}\left(\mathbb{E}_{x\sim p_t}[1-D(x)] + \mathbb{E}_{x\sim p_g}[D(x)]\right) \end{aligned}$$

The goal of the generator is to fool the discriminator whose goal is to be able to distinguish between true and generated data. So, when training the generator, we want to maximise this error while we try to minimise it for the discriminator. It gives us

$$\max_G \left(\min_D E(G,D)\right)$$

For any given generator G (along with the induced probability density p_g), the best possible discriminator is the one that minimises

$$\mathbb{E}_{x\sim p_t}[1-D(x)] + \mathbb{E}_{x\sim p_g}[D(x)] = \int_{\mathbb{R}} (1-D(x))p_t(x) + D(x)p_g(x)dx$$

In order to minimise (with respect to D) this integral, we can minimise the function inside the integral for each value of x. It then defines the best possible discriminator for a given generator

$$\mathbb{1}_{(p_t(x)\geq p_g(x))}$$

(in fact, one of the best because x values such that p_t(x)=p_g(x) could be handled in another way but it doesn't matter for what follows). We then search G that maximises

$$\int_{\mathbb{R}} (1-D_G^*(x))p_t(x) + D_G^*(x)p_g(x)dx = \int_{\mathbb{R}} \min(p_t(x), p_g(x))dx$$

Again, in order to maximise (with respect to G) this integral, we can maximise the function inside the integral for each value of x. As the density p_t is independent of the generator G, we can't do better than setting G such that

$$p_g(x) \geq p_t(x)$$

Of course, as p_g is a probability density that should integrate to 1, we necessarily have for the best G

$$p_g(x) = p_t(x)$$

So, we have shown that, in an ideal case with unlimited capacities generator and discriminator, the optimal point of the adversarial setting is such that the generator produces the same density as the true density and the discriminator can't do better than being true in one case out of two, just like the intuition told us. Finally, notice also that G maximises

$$\frac{1}{2}\int_{\mathbb{R}} \min(p_t(x), p_g(x))dx = \int_{\mathbb{R}} \frac{\min(p_t(x), p_g(x))}{p_t(x) + p_g(x)} \frac{p_t(x) + p_g(x)}{2} dx$$

Under this form, we better see that G wants to maximise the expected probability of the discriminator to be wrong.

Takeaways

The main takeaways of this article are:

- computers can basically generate simple pseudo-random variables (for example they can generate variables that follow very closely a uniform distribution)
- there exist different ways to generate more complex random variables including the notion of "transform method" that consists in expressing a random variable as a function of some simpler random variable(s)
- in machine learning, the generative models try to generate data from a given (complex) probability distribution
- deep learning generative models are modelled as neural networks (very complex functions) that take as input a simple random variable and that return a random variable that follows the targeted distribution ("transform method" like)
- these generative networks can be trained "directly" (by comparing the distribution of generated data to the true distribution): this is the idea of Generative Matching Networks
- these generative networks can also be trained "indirectly" (by trying to fool another network that is trained at the same time to distinguish "generated" data from "true" data): this is the idea of Generative Adversarial Networks

Even if the "hype" that surrounds GANs is maybe a little bit exaggerated, we can say that the idea of adversarial training suggested by Ian Goodfellow and its co-authors is really a great one. This way to twist the loss function to go from a direct comparison to an indirect one is really something that can be very inspiring for further works in the deep learning area. To conclude, let's say that we don't know if the idea of GANs is really "the most interesting idea in the last 10 years in Machine Learning"… but it's pretty obvious that it is, at least, one of the most interesting!

Thanks for reading!

> ***Note:*** *We highly recommend the interested readers to both read the initial paper "Adversarial Neural Nets", that is really a model of clarity for a scientific paper, and watch the lecture video about GANs of Ali Ghodsi, who is truly an amazing lecturer/teacher. Additional explanation can be found in the tutorial about GANs written by Ian Goodfellow.*