Published in Towards Data Science · Follow

Matthew Stewart, PhD Researcher · Follow
May 6, 2019 · 20 min read

Introduction to Turing Learning and 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**

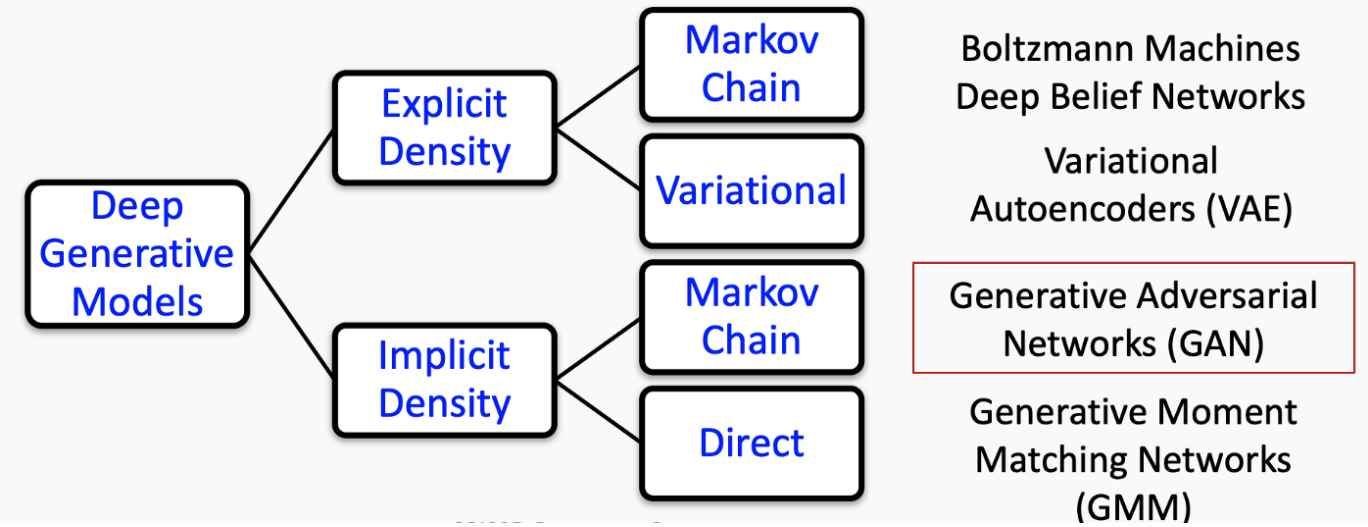
Part 2 is up and can be found here:

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

Part 3 is up and can be found 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

This three-part tutorial continues my series on deep generative models. This topic on Turing learning and GANs 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:

- Network Training
- Network Construction
- GAN Challenges
- GAN rules of thumb (GANHACKs)

There will be no coding in part 1 of the tutorial (otherwise this tutorial would be extremely long), part 2 will act as a continuation to the current tutorial and will go into the more advanced aspects of GANs, with a simple coding implementation used to generate celebrity faces. The third part of the tutorial will be a coding tutorial for applying VAEs, GANs, and VAE-GANs to generate celebrity faces, as well as anime images. Part two and part three will be published in the next week.

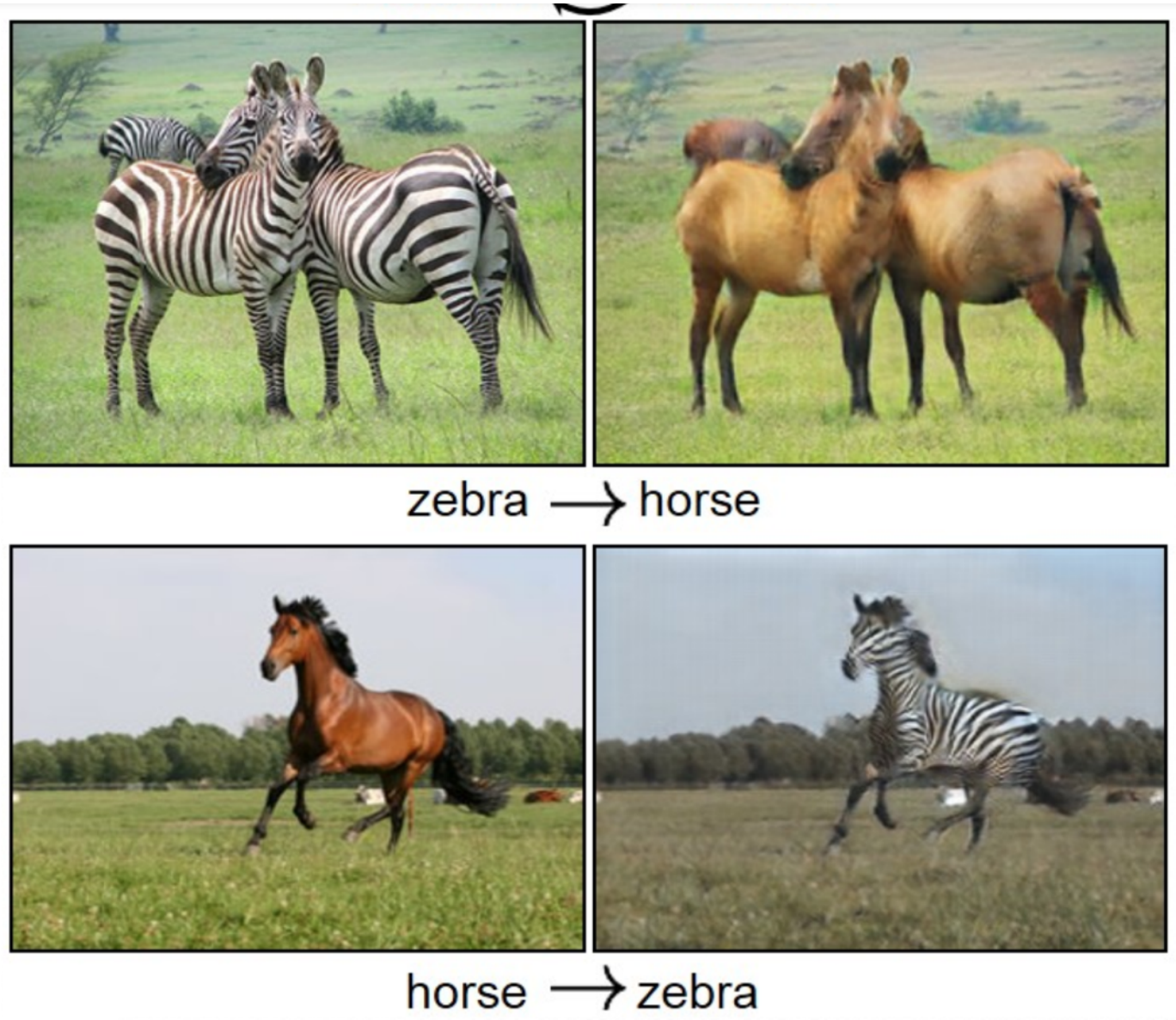
GANs is a fast-moving topic, this tutorial covers the state-of-the-art advances in GANs as of April 2019. If you are reading after this date then beware, there have likely been developments in the field and changes to the rules of thumb.

My aim is for this to be the most comprehensive and accessible tutorial on GANs available, if you have any recommendations for improving this article, please let me know.

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

mrdragonbear/GAN-Tutorial
GitHub is home to over 50 million developers working together to host and review code, manage projects, and build...
github.com

Let us begin!



Horse/zebra image translation using a pre-trained DC-GAN.

The motivation for Turing learning and GANs

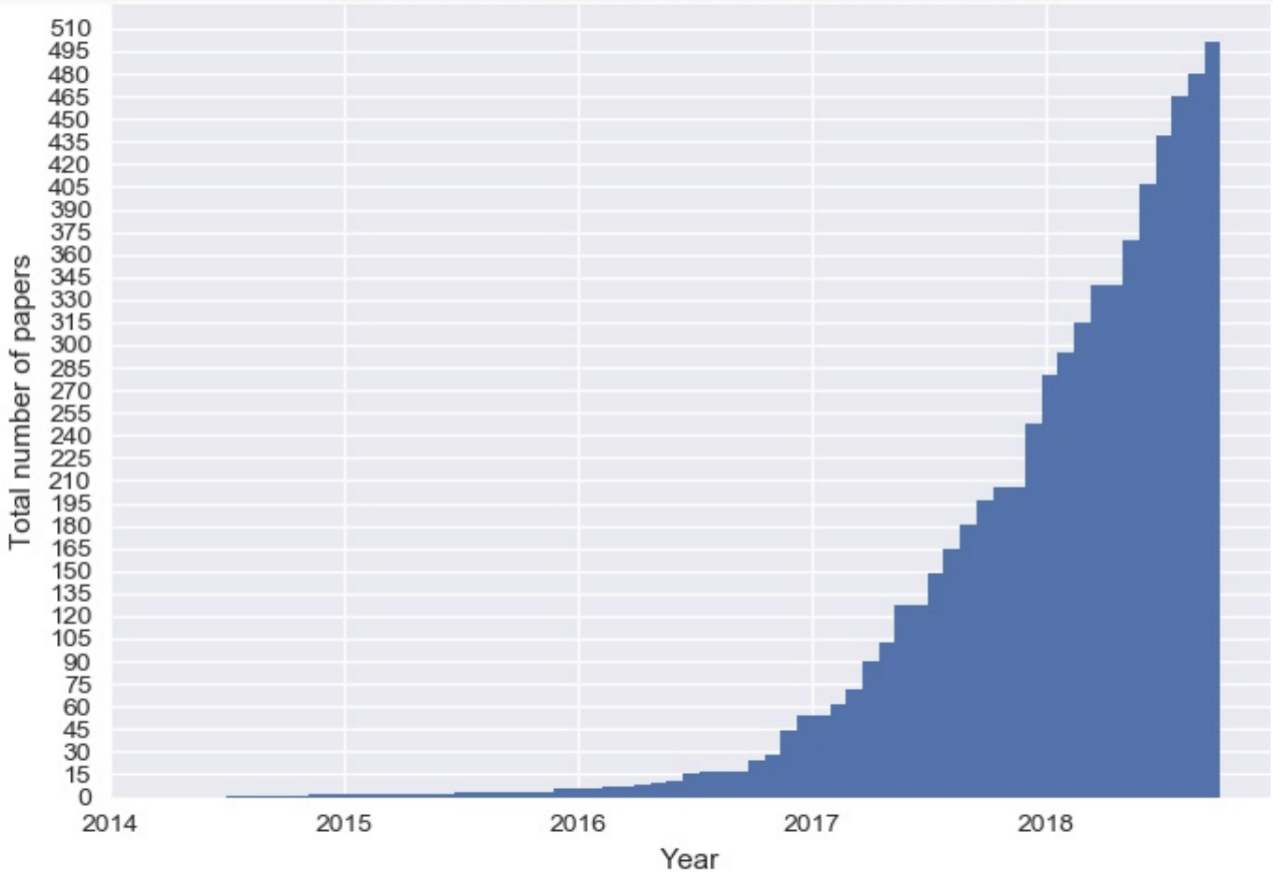
Hopefully, you are reading this because you know nothing (or relatively little) about GANs and how they work. In this section, I hope to get you excited about the potential of GANs and how they can be used to solve real-world problems, as well as to have a lot of fun generating fake celebrities, anime characters, etc.

In previous articles, we focused on generating data using autoencoders. However, the images produced by this procedure were not of very high resolution. In this article, we look at a completely different approach to generating data that is like the training data.

This technique lets us generate types of data that go far beyond what a VAE offers. The GAN is based on a smart idea where two different networks are pitted against one another, with the goal of getting one network to create new samples that are different from the training data, but still close enough that the other network cannot differentiate which are synthetic and which belong to the original training set.

As before, we would like to construct our generative model, which we would like to train to generate lightcurves like these from **scratch**. A generative model, in this case, could be one large neural network that outputs lightcurves: **samples from the model**.

If you are unfamiliar with the idea of generative models or variational autoencoders, you may first want to read my previous article, Comprehensive Introduction to Autoencoders.



The basics of GANs

It is highly likely that you have heard of generative adversarial networks before, but may not have heard of Turing learning. Essentially, Turing learning is the generalization of the procedure underlying a GAN.

The word 'Turing' comes from the similarities to the Turing test, in which a computer tries to fool the system into thinking that it is a human. As we will see, this is analogous to the goals of the generator in a GAN, which tries to fool its 'adversary', the discriminator. The need for having a generalization of GANs stems from the fact that Turing learning can be performed with any form of generator or discriminator, not necessarily a neural network.

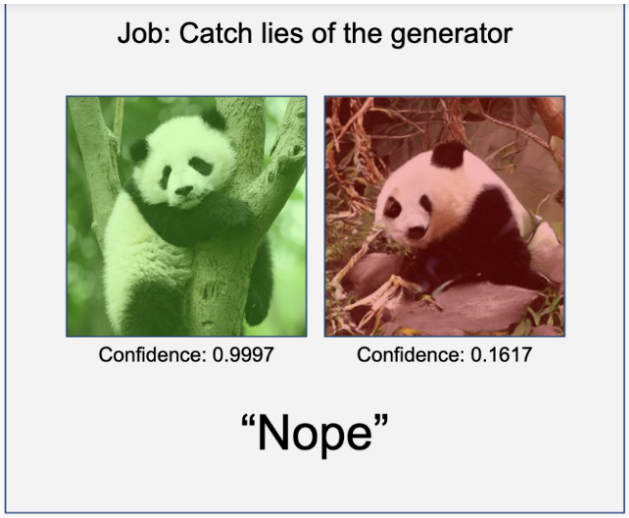
The main reason that using neural networks is commonplace within Turing learning is the fact that a neural network is a **universal function approximator**. That is, we are able to use a neural network (assuming it has sufficient capacity, i.e., a large number of nodes) to 'learn' a non-linear mapping between the input and the output. This gives neural networks much more freedom than most methods, as they are guaranteed to converge for any non-linear function (given an infinite network capacity and infinite training data — see the universal approximation theorem for more information).

There are no real constraints on what form the generator or discriminator takes, they do not even need to be of the same form. However, using anything other than a neural network may increase the bias of the model. As an example, one could use support vector machines for both the generator and discriminator; similarly, a support vector machine for the generator and a neural network for the discriminator.

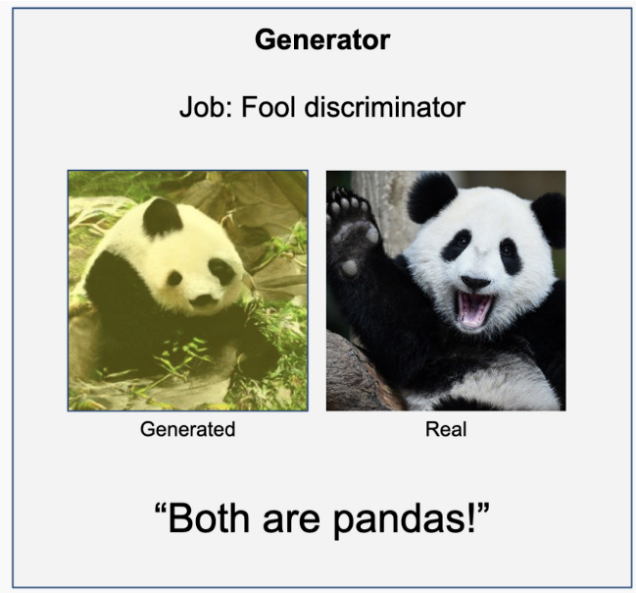
A large part of this tutorial (mostly in part 2) will look at generating anime images similar to those below, using a VAE, followed by a GAN, followed by a VAE-GAN (more on this one later).

Anime images from our ‘GANIME’ training set.

Now we will dive into the structure of the GAN more specifically. As we have discussed, the job of the generator is to make fake images (in the case of image analysis, at least) that look reminiscent of the training set. The discriminator looks at this fake image and tries to identify whether it is real or not. The loss function of both the generator and discriminator is highly dependent on how well the discriminator performs its job.

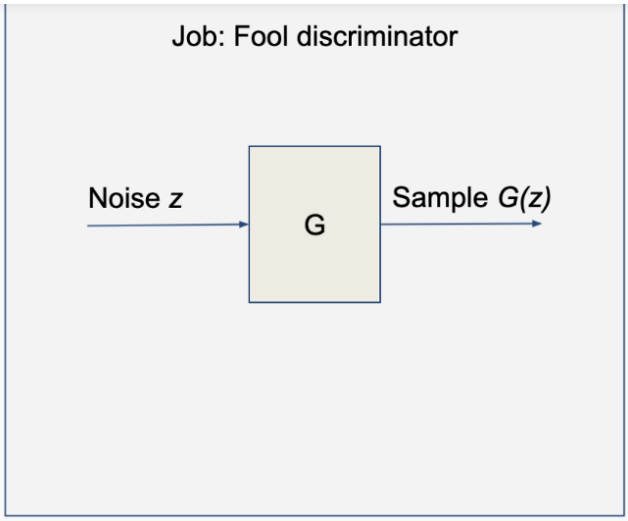


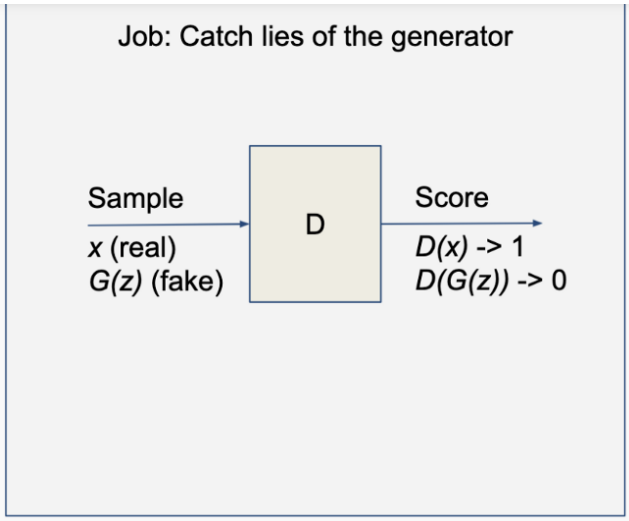
After sufficient training, the generator will become better, and the images will begin to look more photorealistic.



Schematically, we can represent the generator and discriminator as black box models, which are abstractions of some form of function. This function can (as always in machine learning) be approximated using our catch-all function approximators, neural networks.

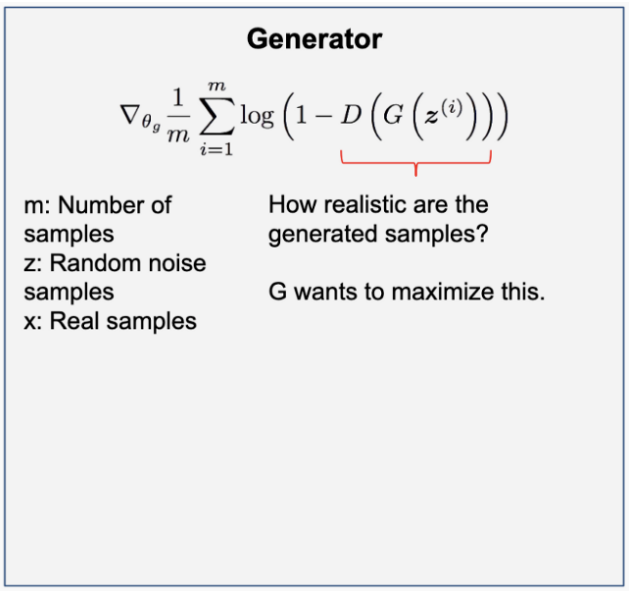
The input to the generator is noize z, and the generated sample will be the output of our generator function, $G(z)$. This generated image is then added arbitrarily to our input data to the discriminator, which then performs a binary classification (i.e. fake or not fake) and is assigned a score based on whether the image was, in fact, fake or not.

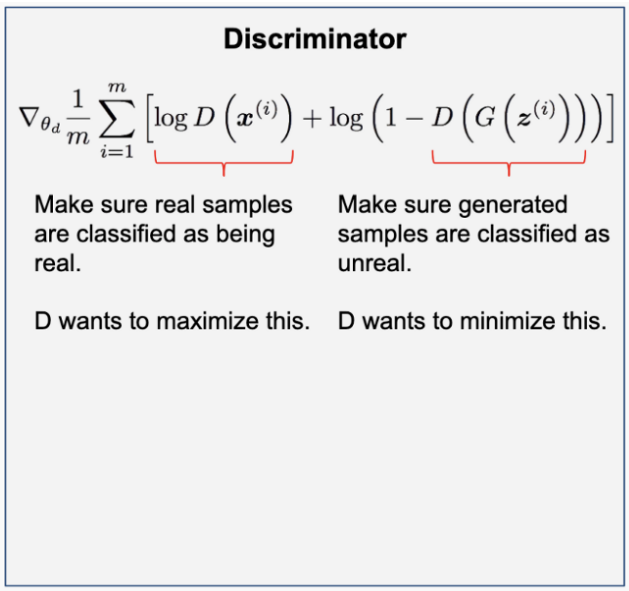




The loss functions for the generator and the discriminator look a little bit intimidating at first, but they are actually very simple. *G(z)* is the output of our generator, i.e. the fake image, *D(G(z))* is the prediction from the discriminator on our fake data and *m* is the number of samples. We use the logarithm because it is more numerically stable as a loss function and we take the gradient of the loss function with respect to the parameters so that we can apply stochastic gradient descent.

If this sounds like gibberish to you right now then fear not, a large part of this tutorial will go into explaining the updating and refinement of the generator and discriminator.





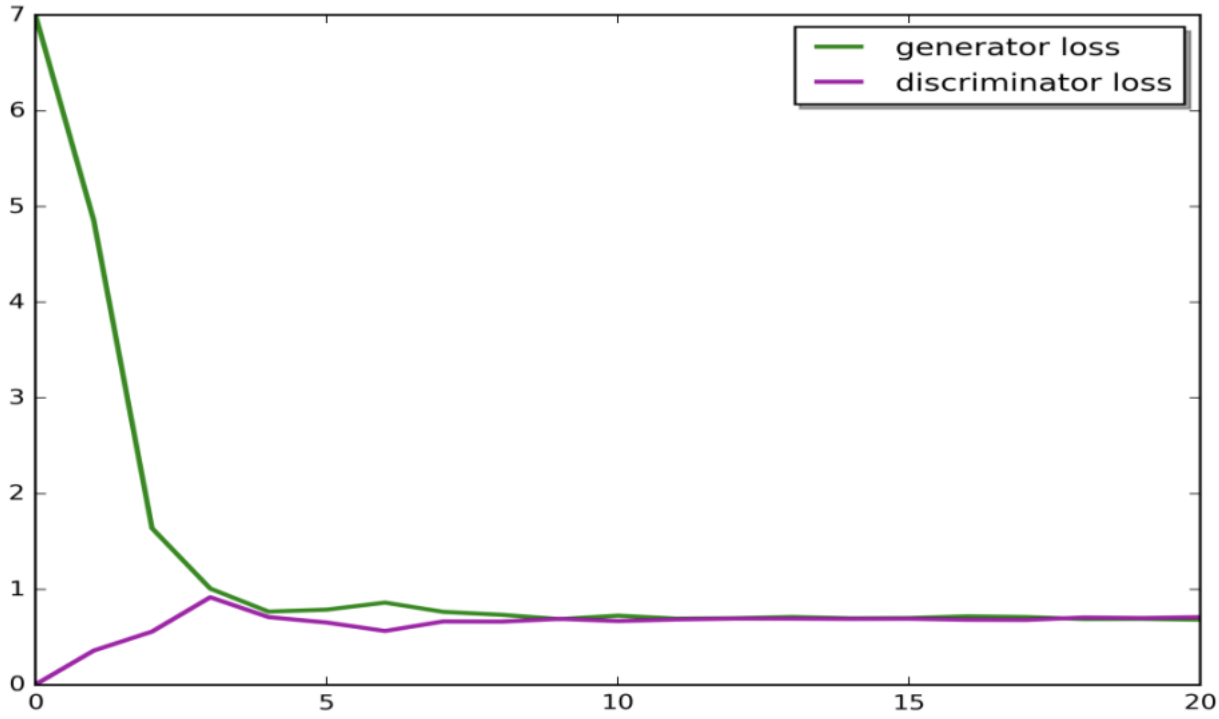
Game Theory

The entire idea of GANs is predicated on game theory. For those of you unaware, game theory analyzes games in order to come up with ideal strategies for how to win. It has become relatively intertwined with artificial learning and is how computers were able to beat world champions in pretty much every board game that exists. The most recent and impressive of which is probably Go, where the AI AlphaGo was able to beat world champion Ke Jie.

In some games, there are unbounded resources. For example, in a game of poker, the pot can theoretically get larger and larger without limit. For many games, the resources are bounded, meaning that a player can only win at another player's expense; this is known as a zero-sum game.

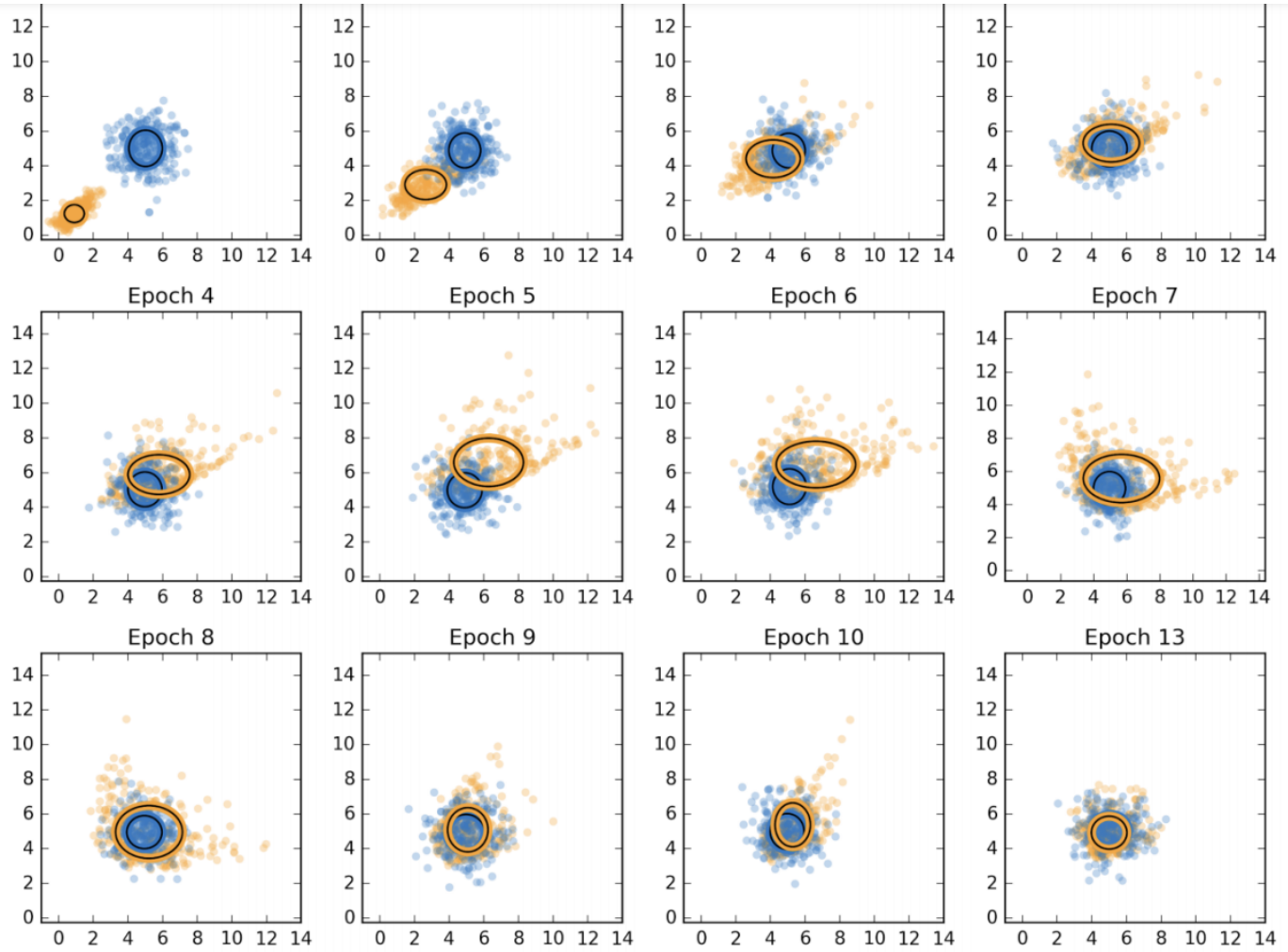
Our goal in training the GAN is to produce two networks that are each as good as they can be. In other words, we don't end up with a "winner."

Instead, both networks have reached their peak ability given the other network's abilities to thwart it. Game theorists call this state a **Nash equilibrium**, where each network is at its best configuration with respect to the other. This idea is illustrated below.



Loss function vs. number of epochs for discriminator and generator networks — flat line is Nash equilibrium.

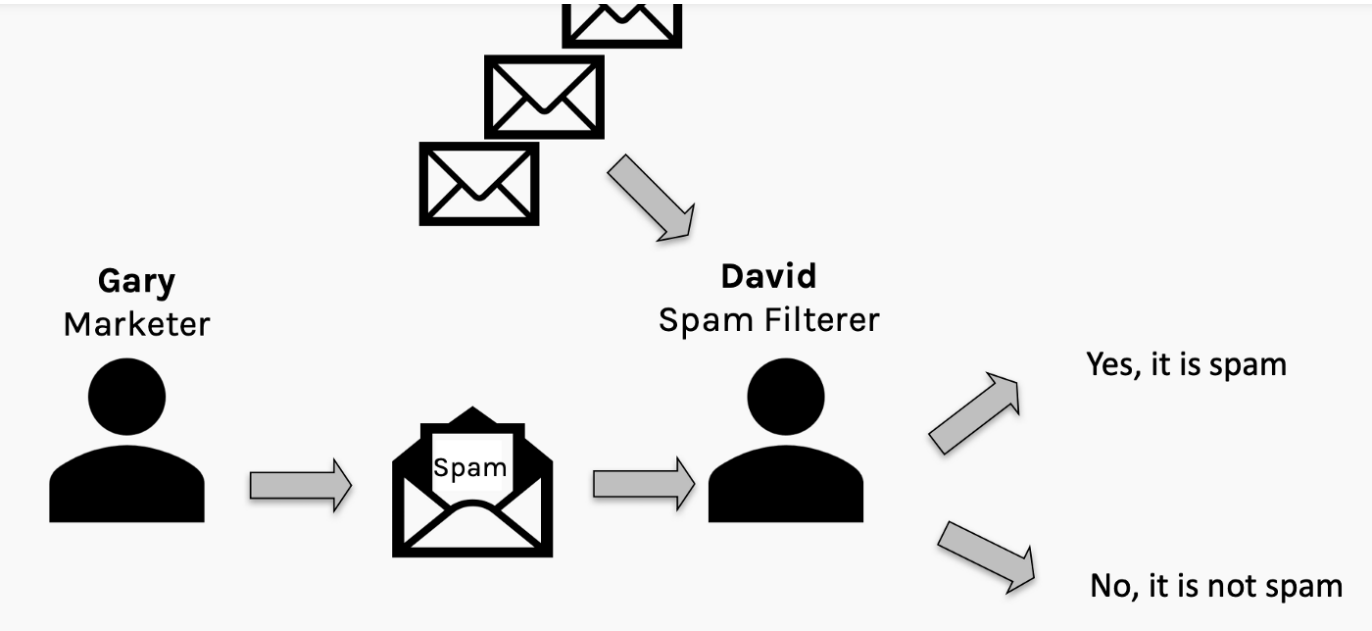
To see this what is happening in terms of the Nash equilibrium in the latent representation, below we have plotted the generator and discriminator, as well as their distributions, as a function of epoch. We see that there is a gradual convergence of the distributions.



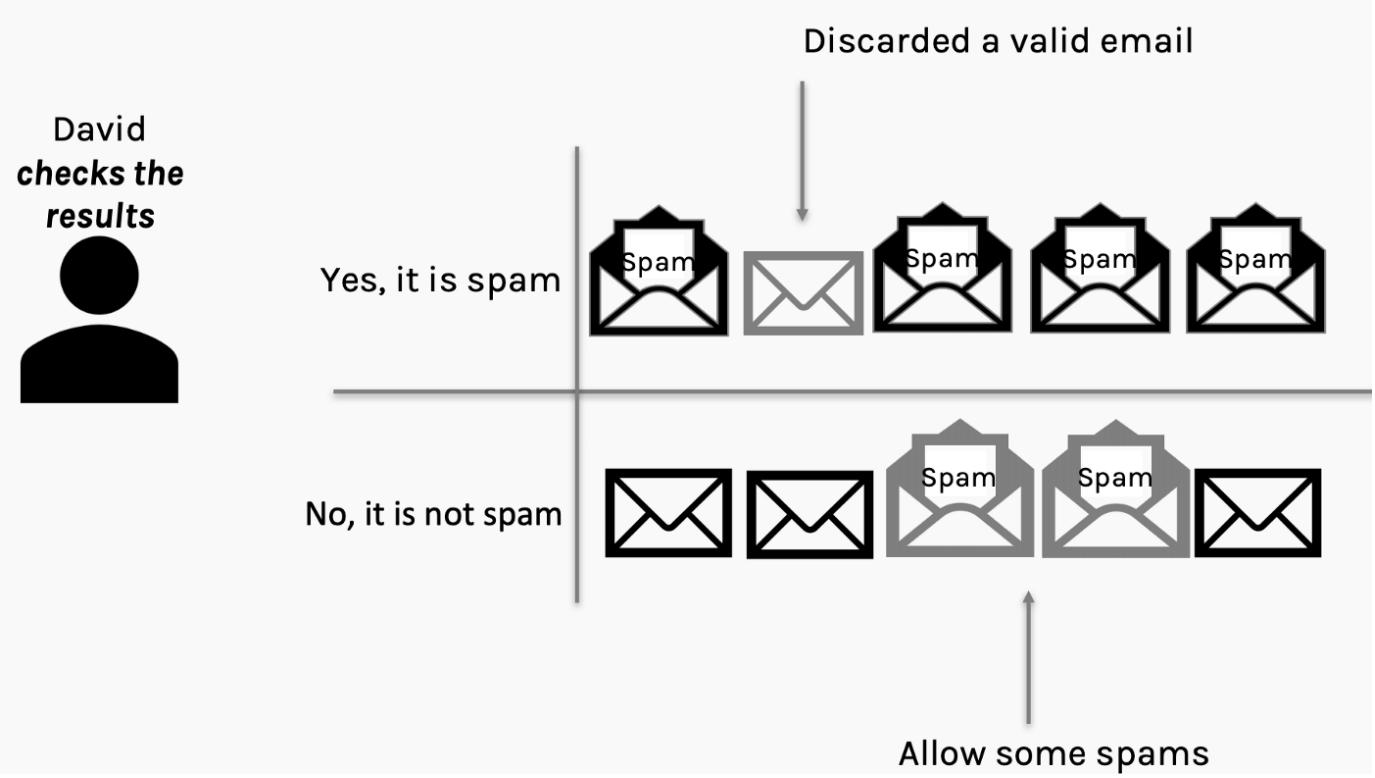
Spam Filter Example

Spam filtering is a great way to think about how the generative adversarial network works. This is similar to how I described the variational autoencoder, but not exactly the same.

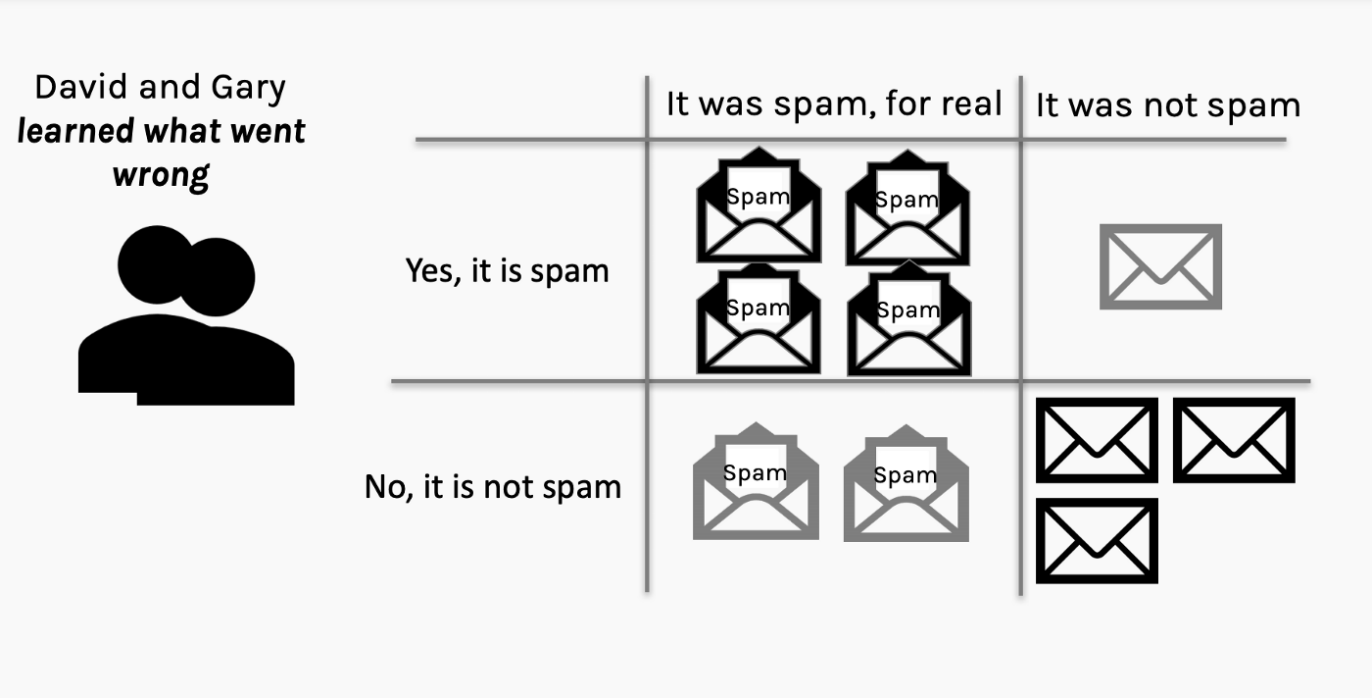
Imagine you have a marketer called Gary who is trying to get spam emails through David's spam filter. David is allowed to classify emails as spam or not after they have been OK'ed by the spam filter. Gary wants to get through as many spam emails as possible, and David would like as few as possible to get through. Ideally, we will eventually reach a Nash equilibrium from such a scenario (although I am sure most people would prefer not spam emails!).



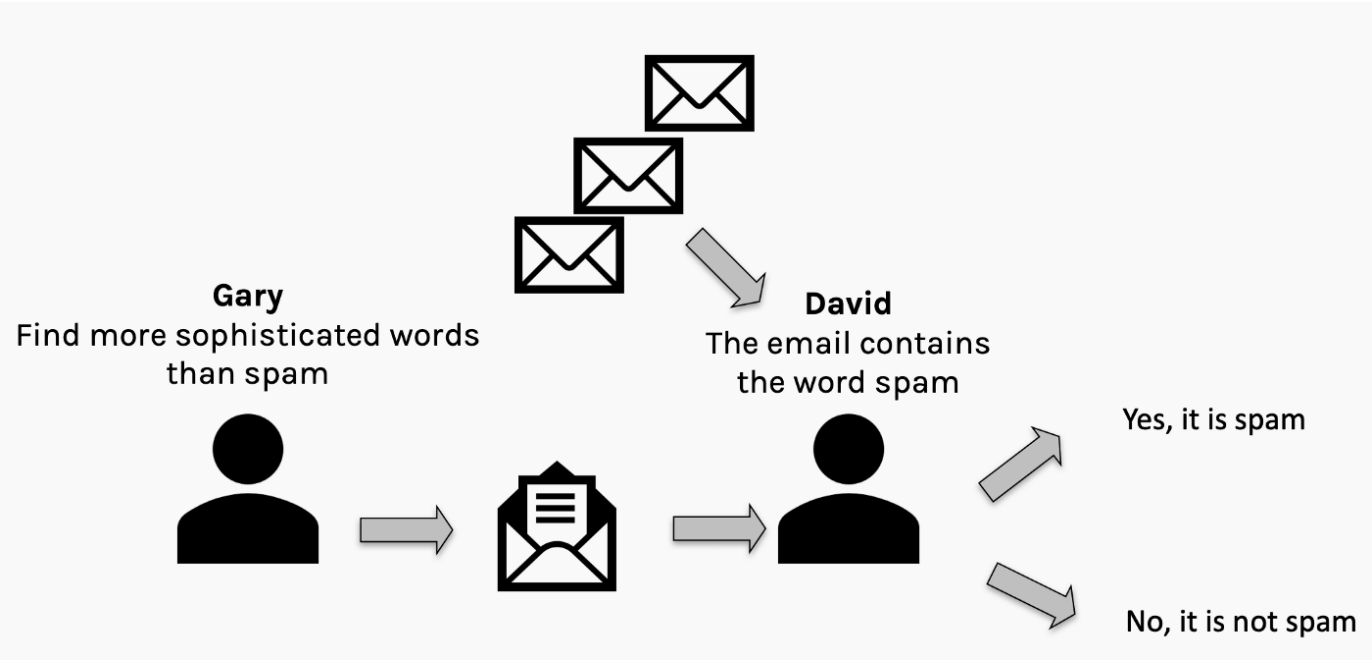
After receiving a bunch of emails, David can check to see how well the spam filter did, and can 'punish' the spam filter by telling it when it got false positives or false negatives.



Assuming that Gary also knows which of his spam emails got through (perhaps he also sends them to himself to validate the success) then both David and Gary can see how well they did at their respective tasks in the form of a confusion matrix (below).

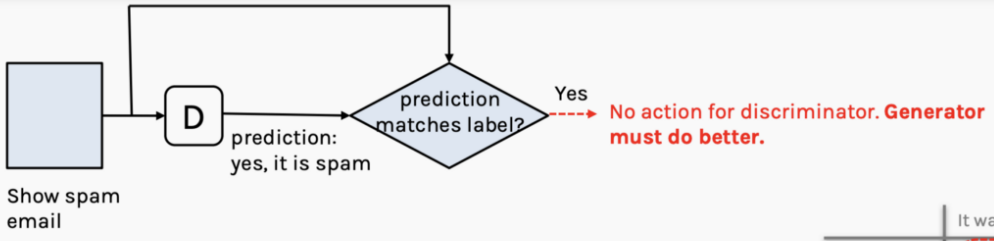


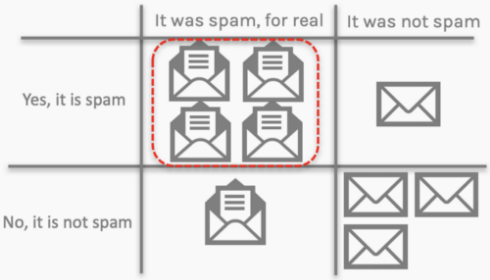
After this, both of them can learn what went wrong and then learn from their mistakes. Gary will try a different approach which makes use of his prior successes, and David will see where the spam filter went wrong to try and improve the filtering mechanism.



We can continuously repeat this procedure until we obtain some form of Nash equilibrium (or one of the two finds out the perfect way to win and 'spams' this method, resulting in modal collapse — more on that later).

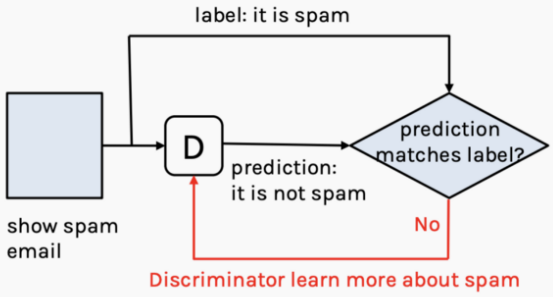
We can consider the confusion matrix and use this as the basis to improve our generator and our discriminator. For example, if the email is, in fact, a spam email and it is classified as fake, the generator is doing a poor job and must do better. The discriminator does not need to do anything in this sense, it has done its job.

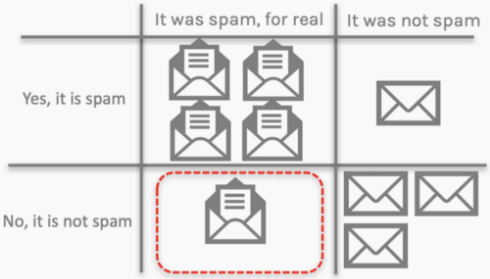




True positive (TP): the discriminator see a spam and predicts correctly. No need for further actions for discriminator. Generator must do a better job.

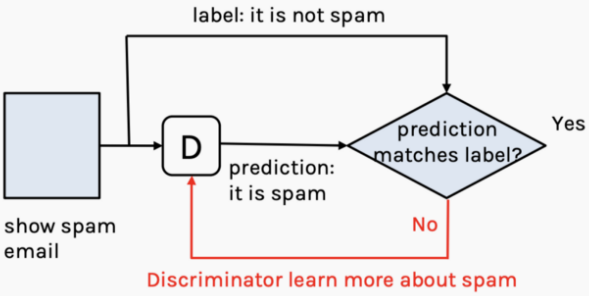
In the case of a false negative (the email was not spam but it was classified as spam), it is the discriminator that has been fooled. The discriminator must do better in this case, whereas the generator has done its job correctly and does not need to be improved.



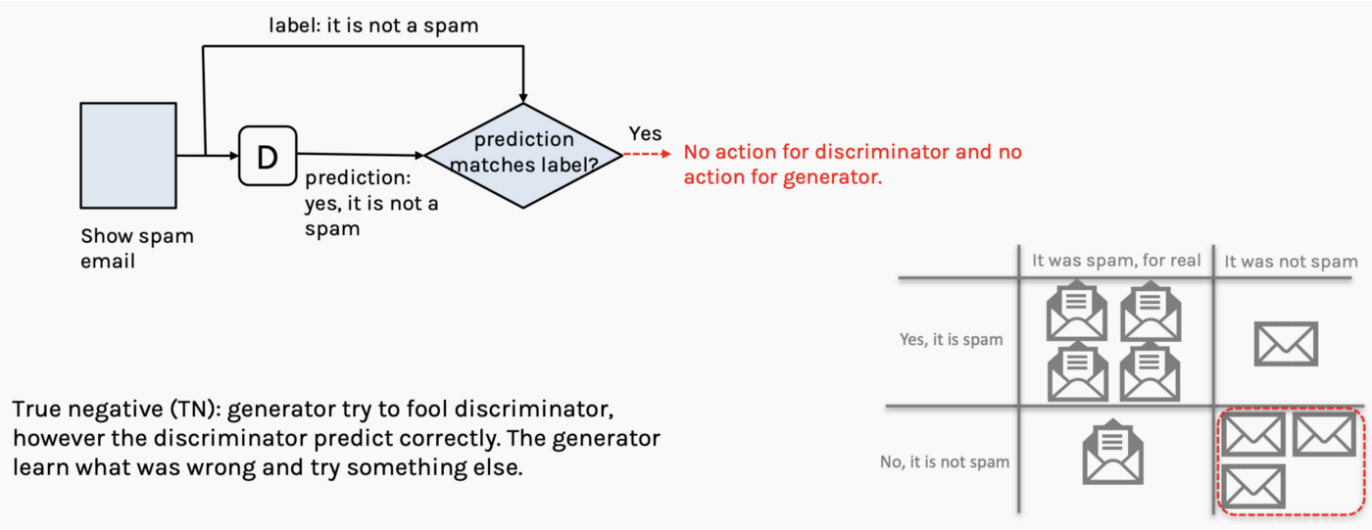


False Negative (FN): the discriminator see an email and predict it as spam even though it is not. The discriminator learn more

In the case of a false positive (classified as real email when it is, in fact, a spam email), it is once again the discriminator that is at fault. The discriminator must then be updated whilst the generator does not do anything.

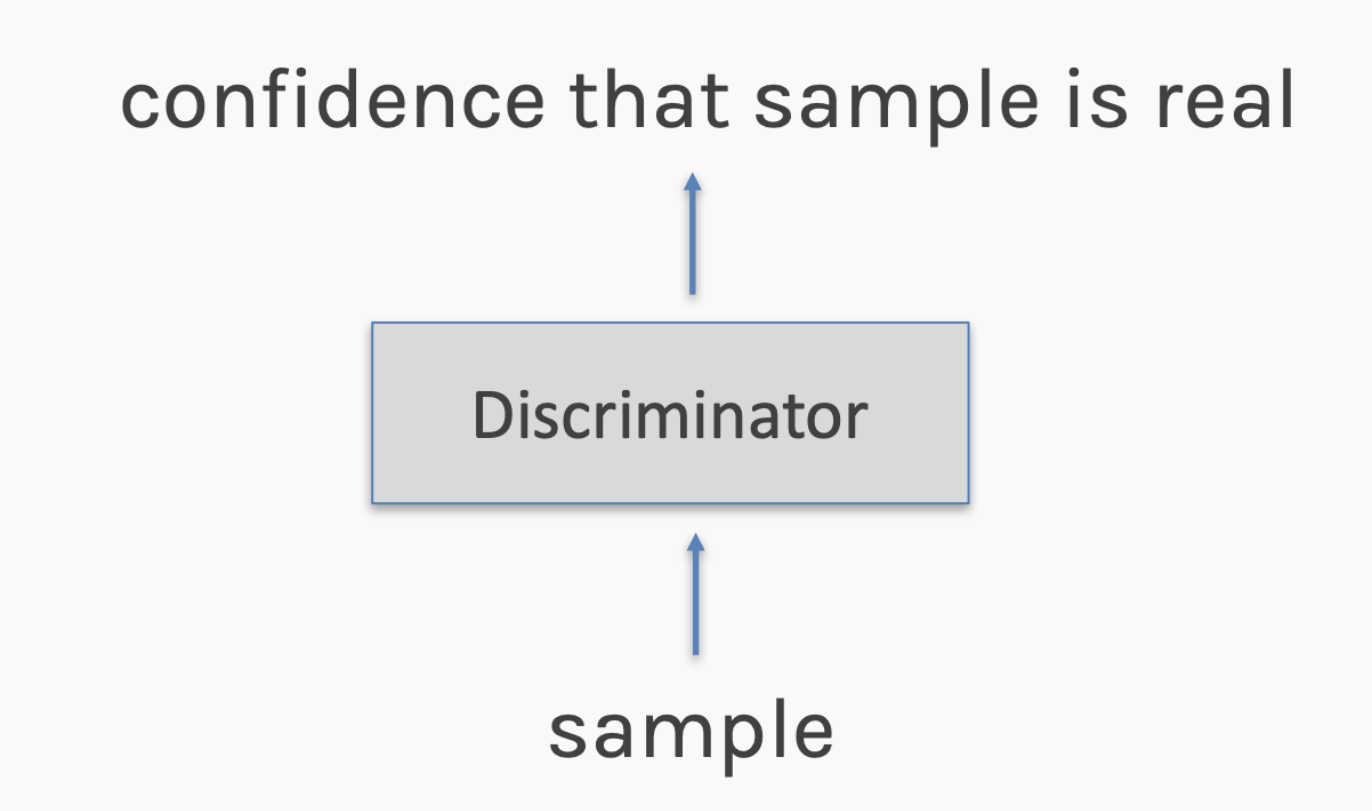


False positive (FP): generator try to fool discriminator and the discriminator fails. The generator succeeded and the generator is forced to improve.

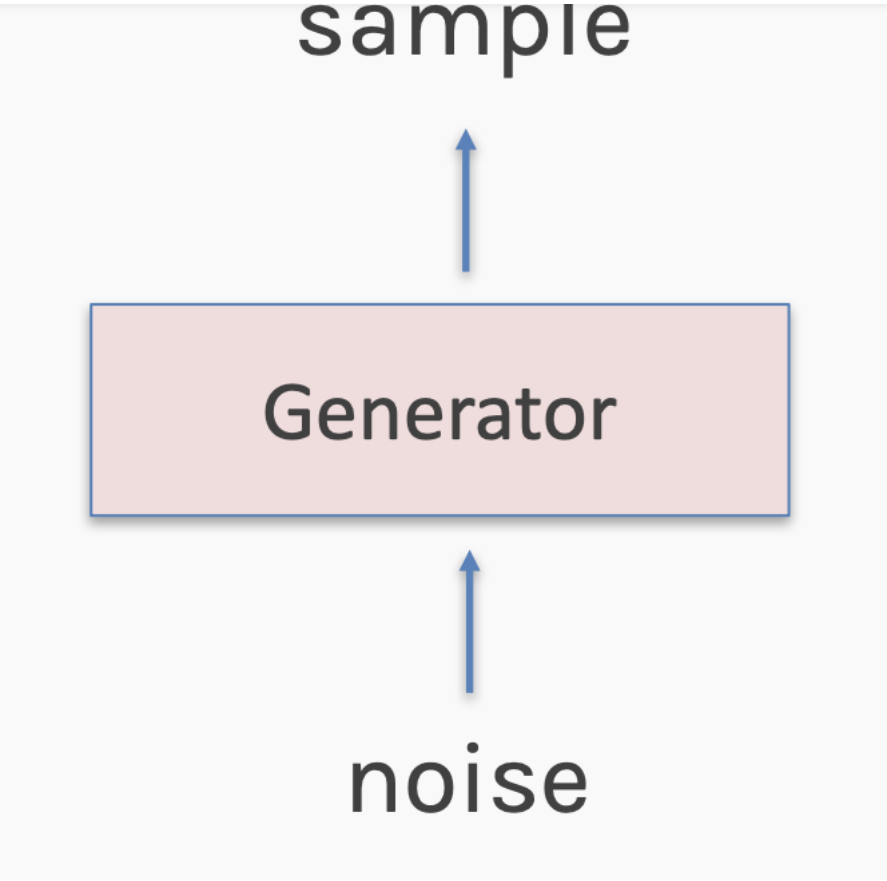


Generator and Discriminator

The discriminator is very simple. It takes a sample as input, and its output is a single value that reports the network's confidence that the input is from the training set, rather than being a fake. There are not many restrictions on what the discriminator is.



The generator takes as input a bunch of random numbers. If we build our generator to be deterministic, then the same input will always produce the same output. In that sense, we can think of the input values as latent variables. But here the latent variables weren't discovered by analyzing the input, as they were for the VAE. The random noise is not "random" but represents (an email in our example) in the "latent" space.



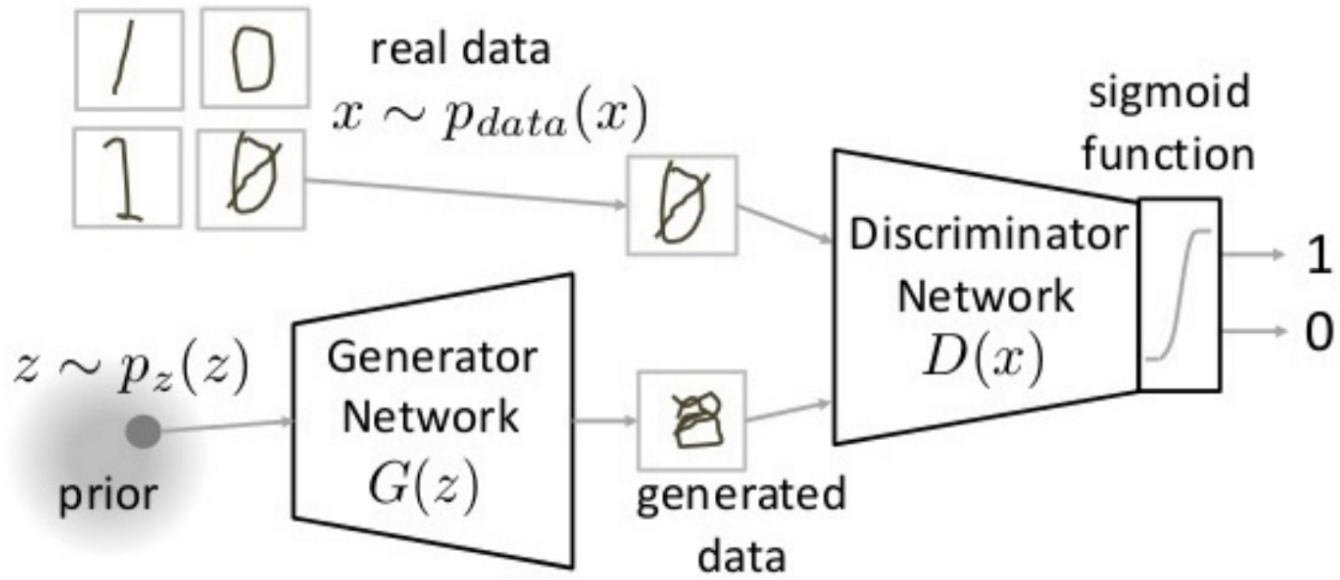
The process — known as a learning round — accomplishes three jobs:

[1] The discriminator learns to identify features that characterize a real sample.

[2] The discriminator learns to identify features that reveal a fake sample.

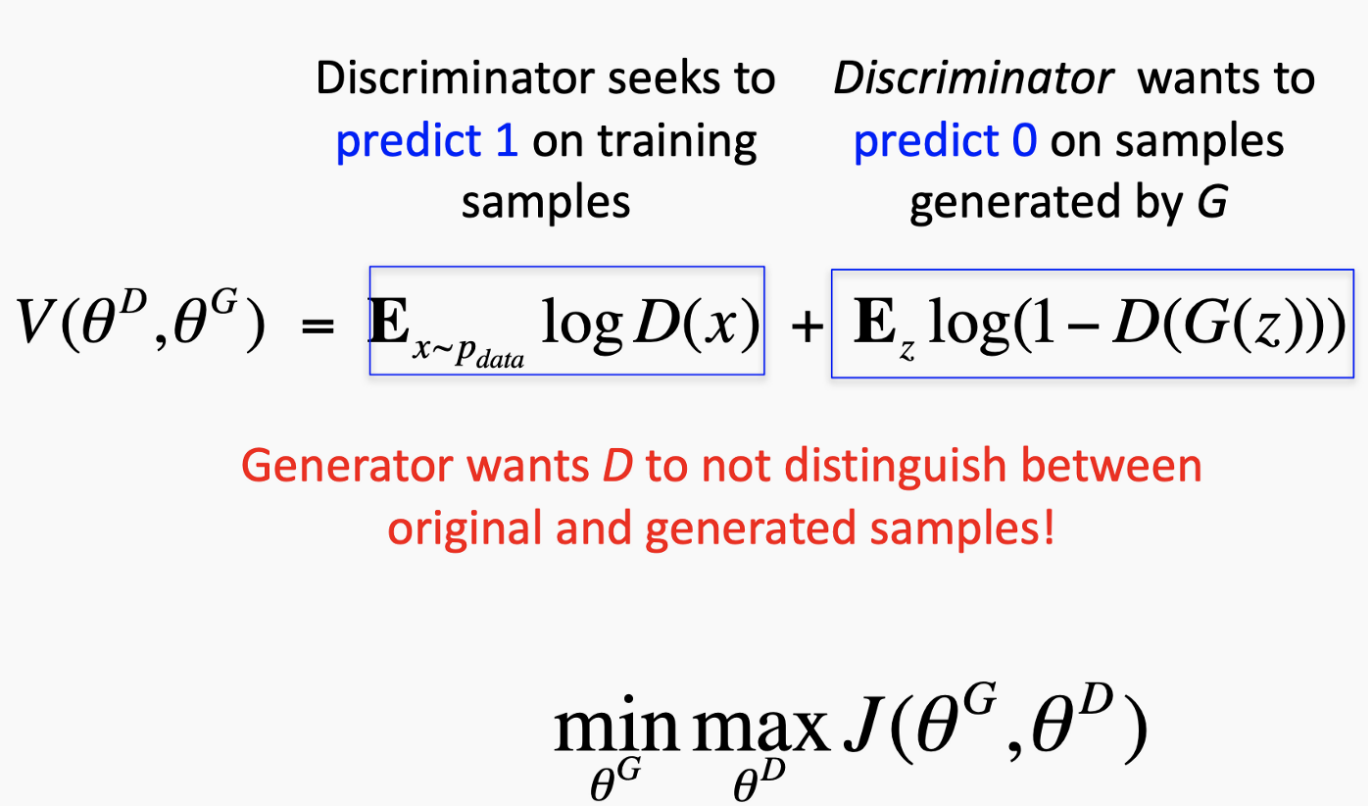
[3] The generator learns how to avoid including the features that the discriminator has learned to spot.

The final network will look something like the one below. To briefly summarise how this works, a random sample is taken from some prior distribution, which is fed into the generator to make some fake image. This fake image, along with the real data, is fed into the discriminator network, which then decides which data comes from the real data set, and which comes from the fake data generated from the prior distribution.



Network Training

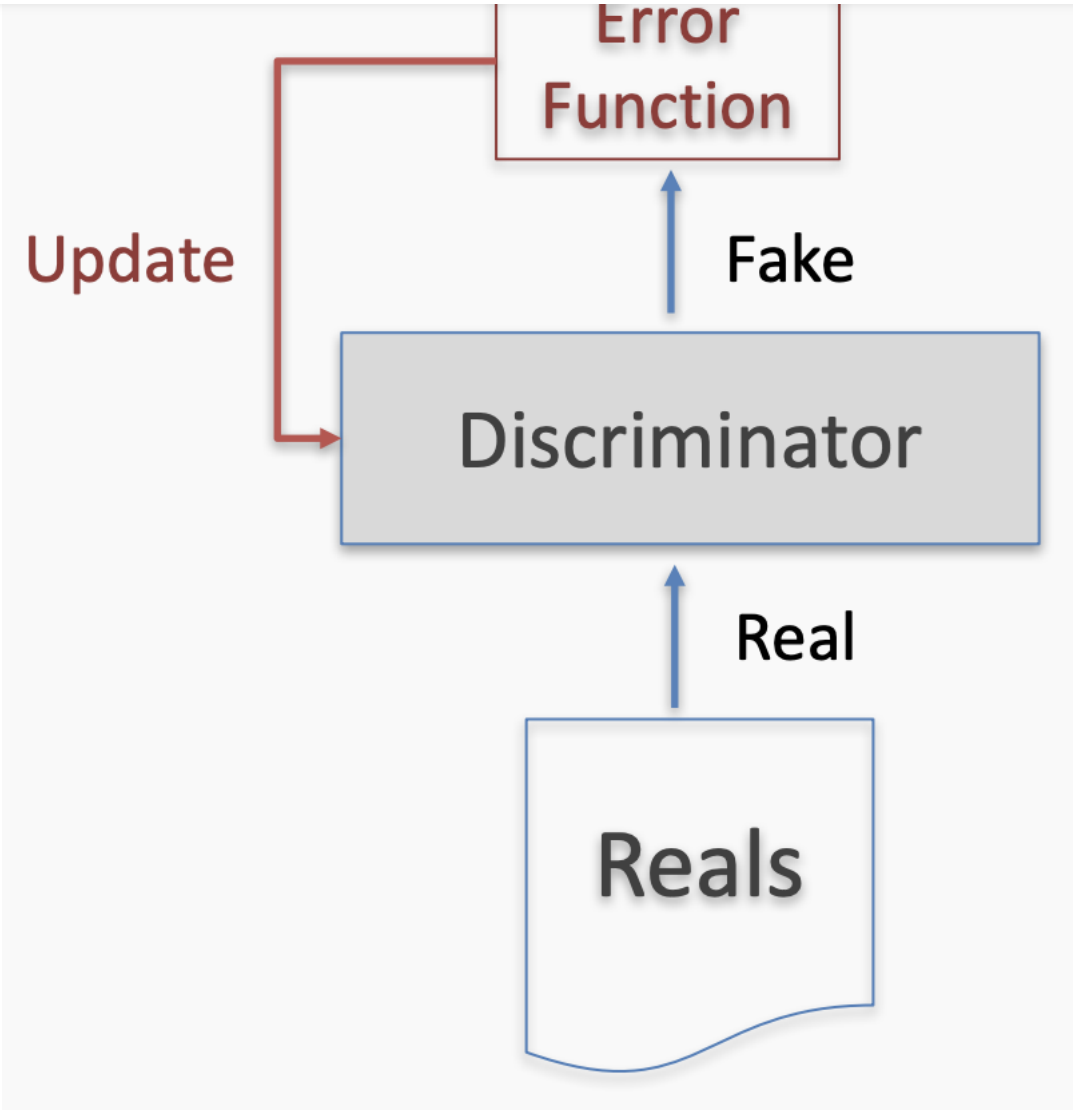
In terms of our networks, there are two networks we need to train. This becomes interesting because both networks have the same overall value function, but slightly different loss functions. The discriminator is trying to maximize the overall value function, whereas the generator seeks to minimize the discriminator's value function.



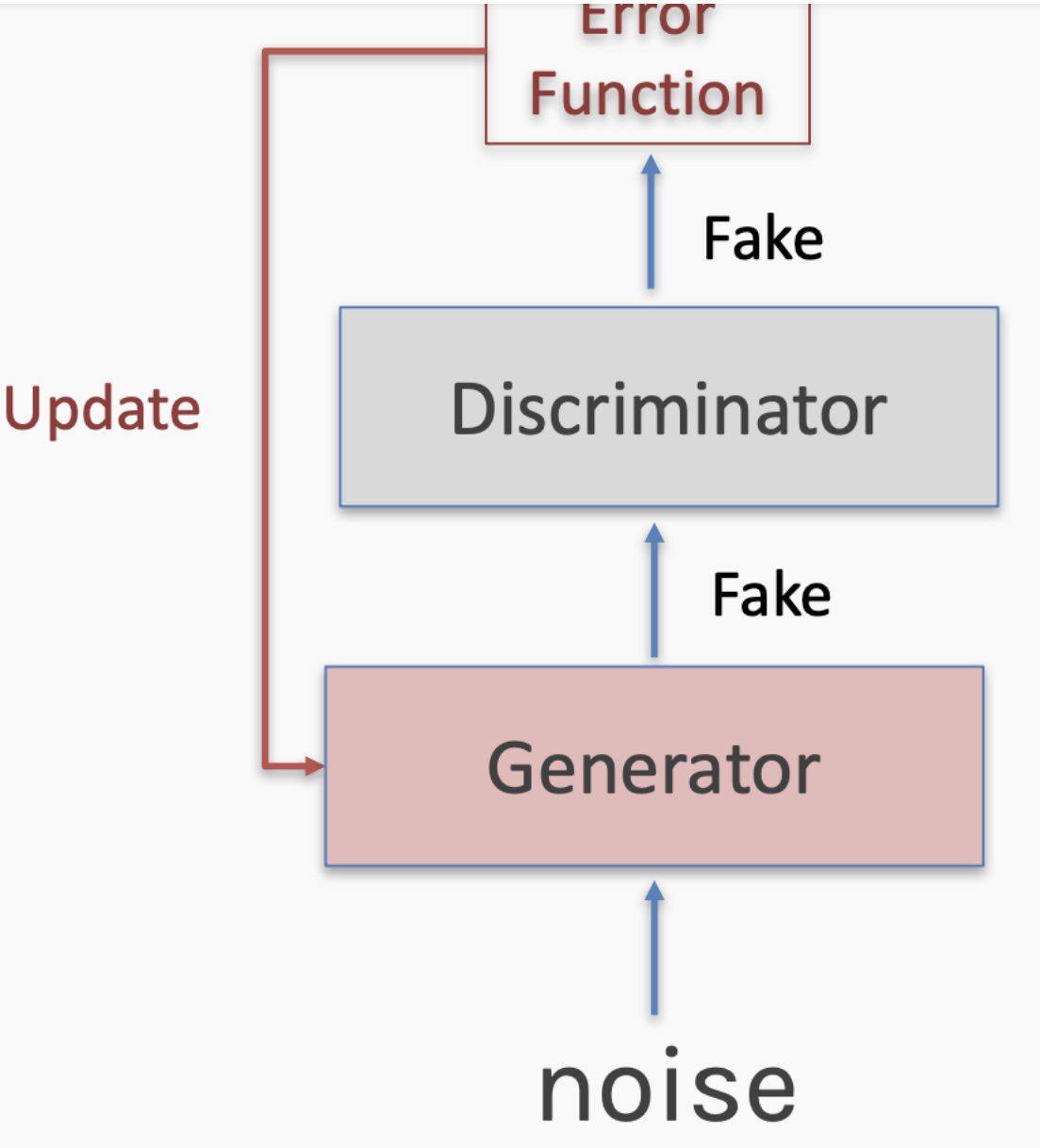
The method of training involves the following:

- Sample a mini-batch of training images x, and generator codes z.
- Updating G and D using backpropagation (optional: run k steps of one player for every step of the other player — typical ratio is D:G of 4:1)

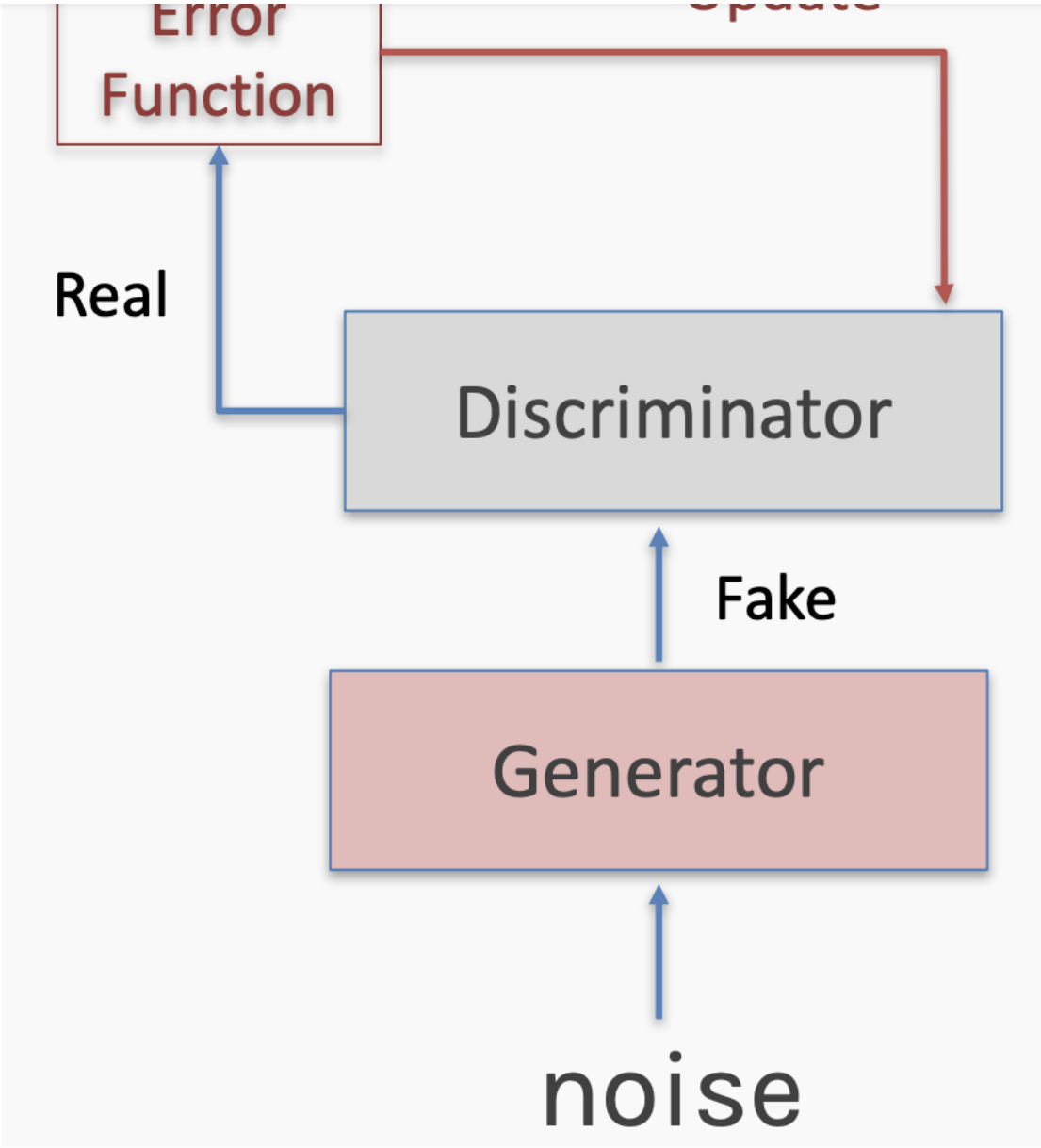
False negative (I: Real/D: Fake): In this case, we feed reals to the discriminator. The generator is not involved in this step at all. The error function here only involves the discriminator and if it makes a mistake the error drives a backpropagation step through the discriminator, updating its weights so that it will get better at recognizing reals.



True negative (I: Fake/D: Fake): We start with random numbers going into the generator. The generator's output is fake. The error function gets a large value if this fake is correctly identified as fake, meaning that the generator got caught. Backpropagation goes through the discriminator (which is frozen) to the generator. The generator is then updated, so it can better learn how to fool the discriminator.

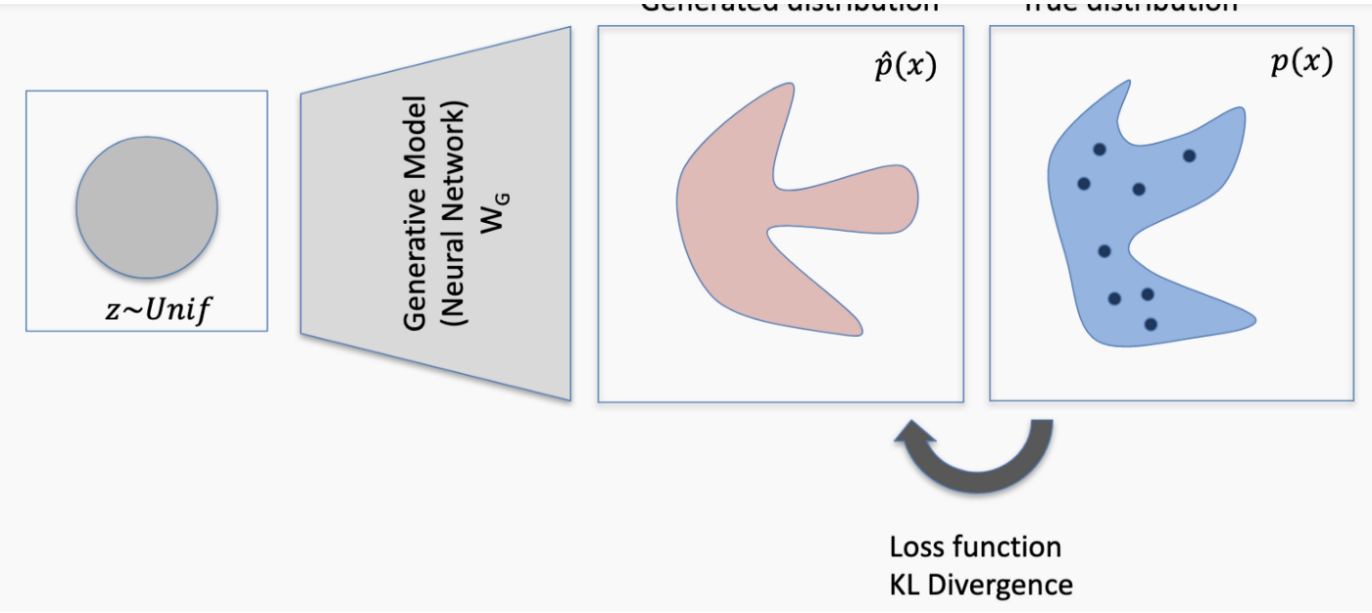


False positives (I:Fake/D:Real): Here we generate a fake and punish the discriminator if it classifies it as real.

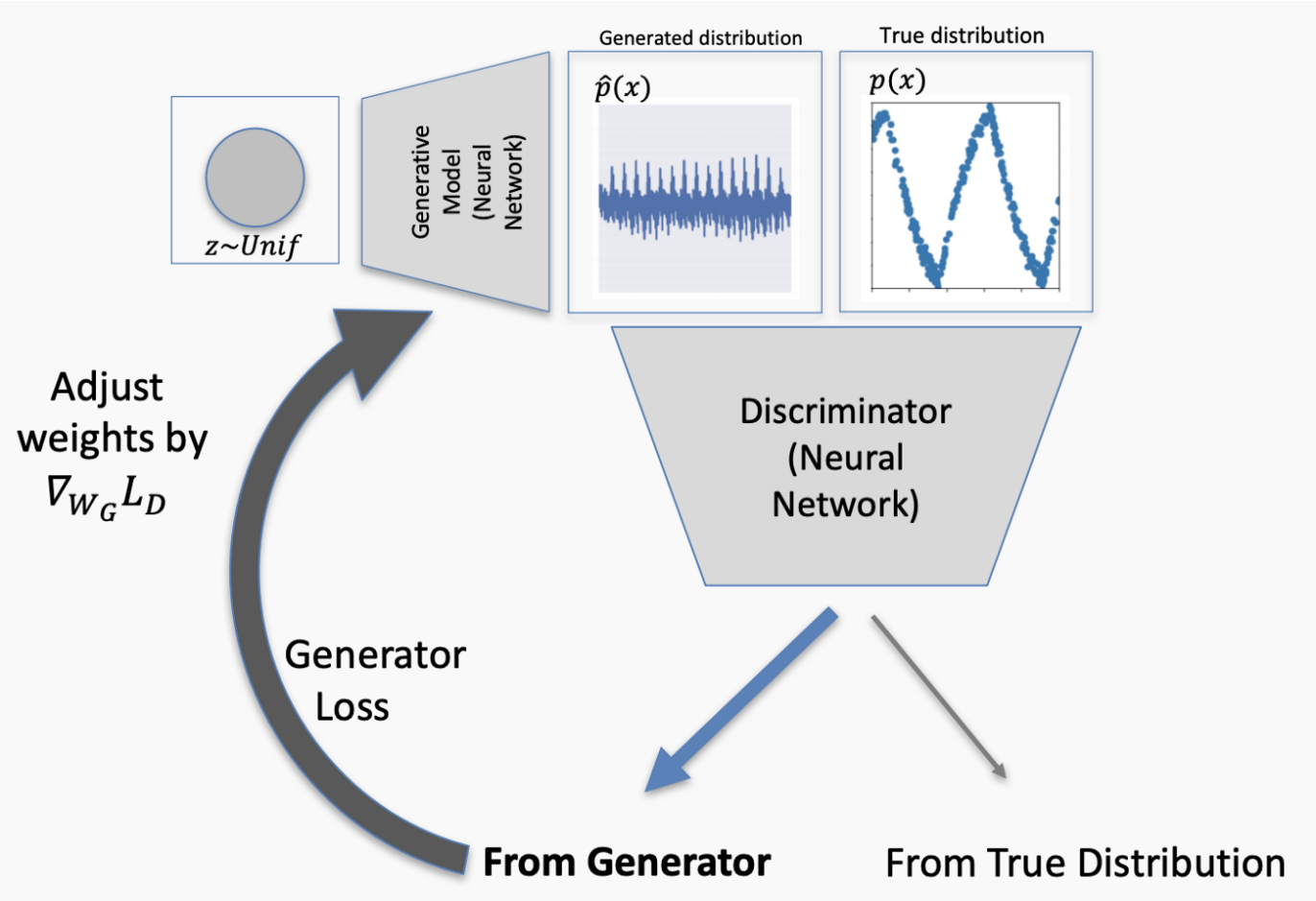


To illustrate the training in a less abstract form, we will go through another example which is slightly more involved than the spam filtering example.

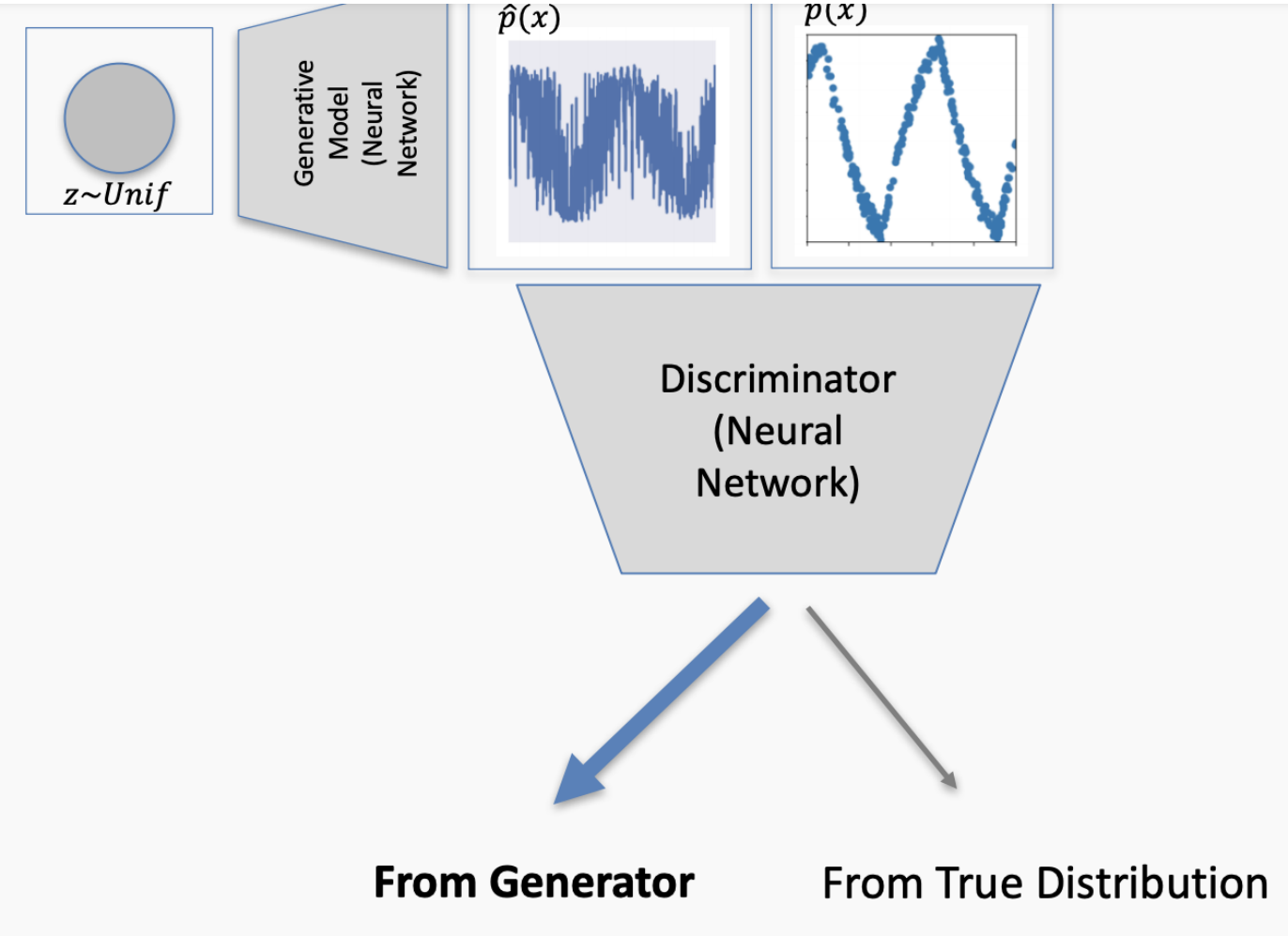
We have our generated (fake) distribution which is produced by our generative model, and we have a known true distribution. There is an associated KL-divergence between the two because they are not identical distributions, meaning that our loss function is non-zero.



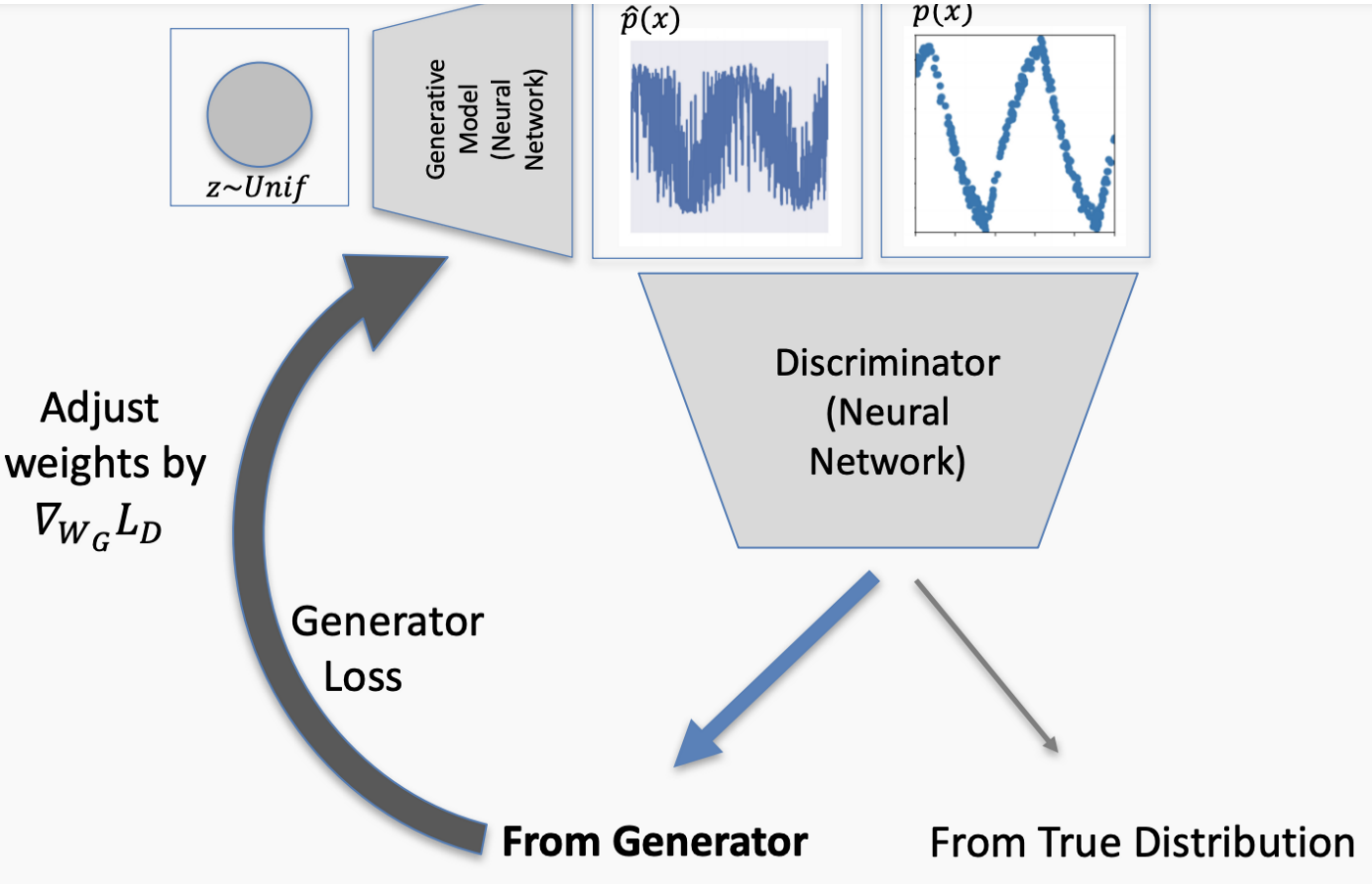
The discriminator then sees the input from the generated and true distributions. If the discriminator decides the data is from the generator, this generates a loss function value which propagates back to the generator and is used to update the weights. **Importantly, only one of the two networks is ever trained at the same time.**



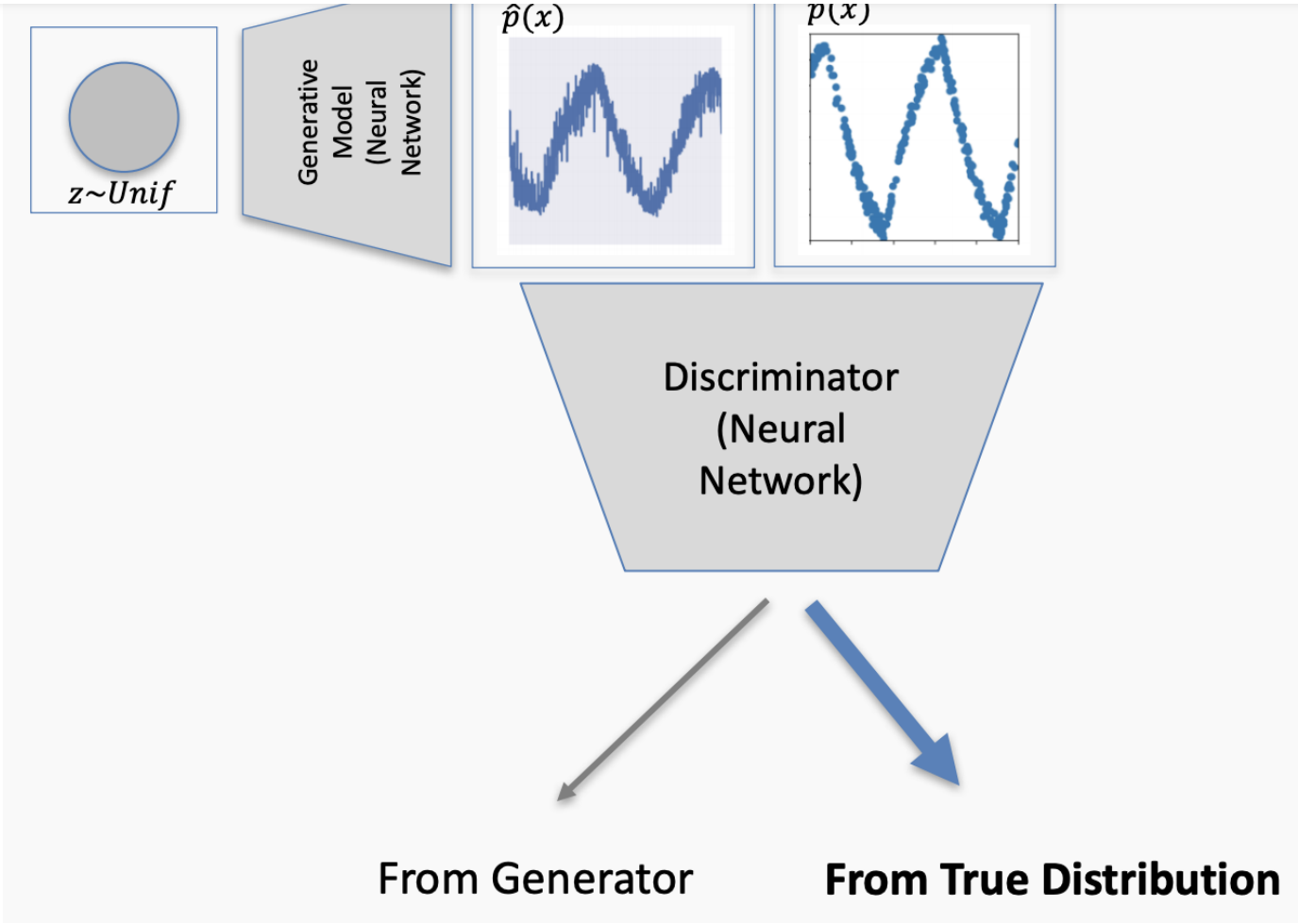
The generator has now improved, and the data looks more reminiscent of the true distribution.



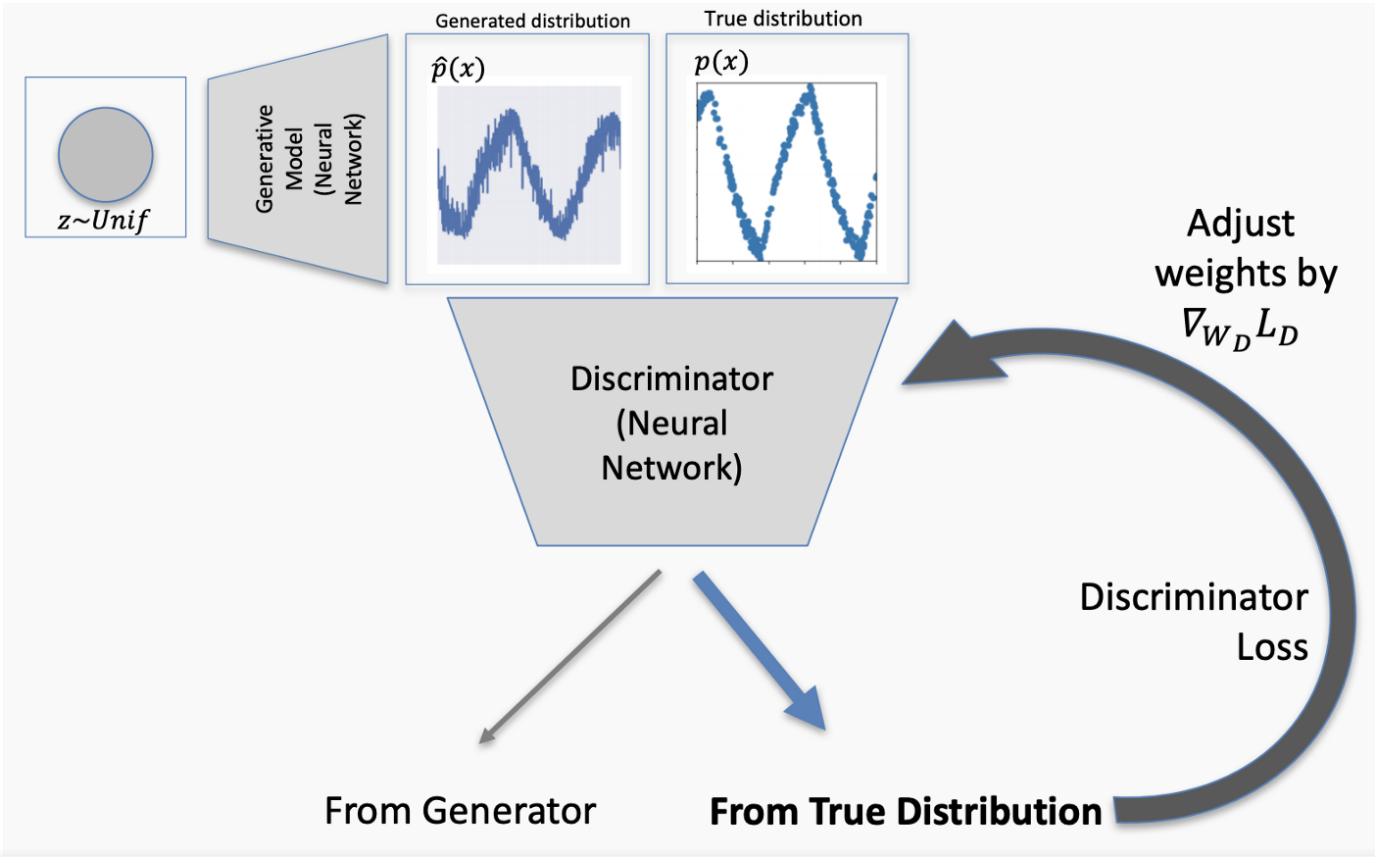
However, the data is still not quite good enough to fool the discriminator, and so the generator weights are updated once again.

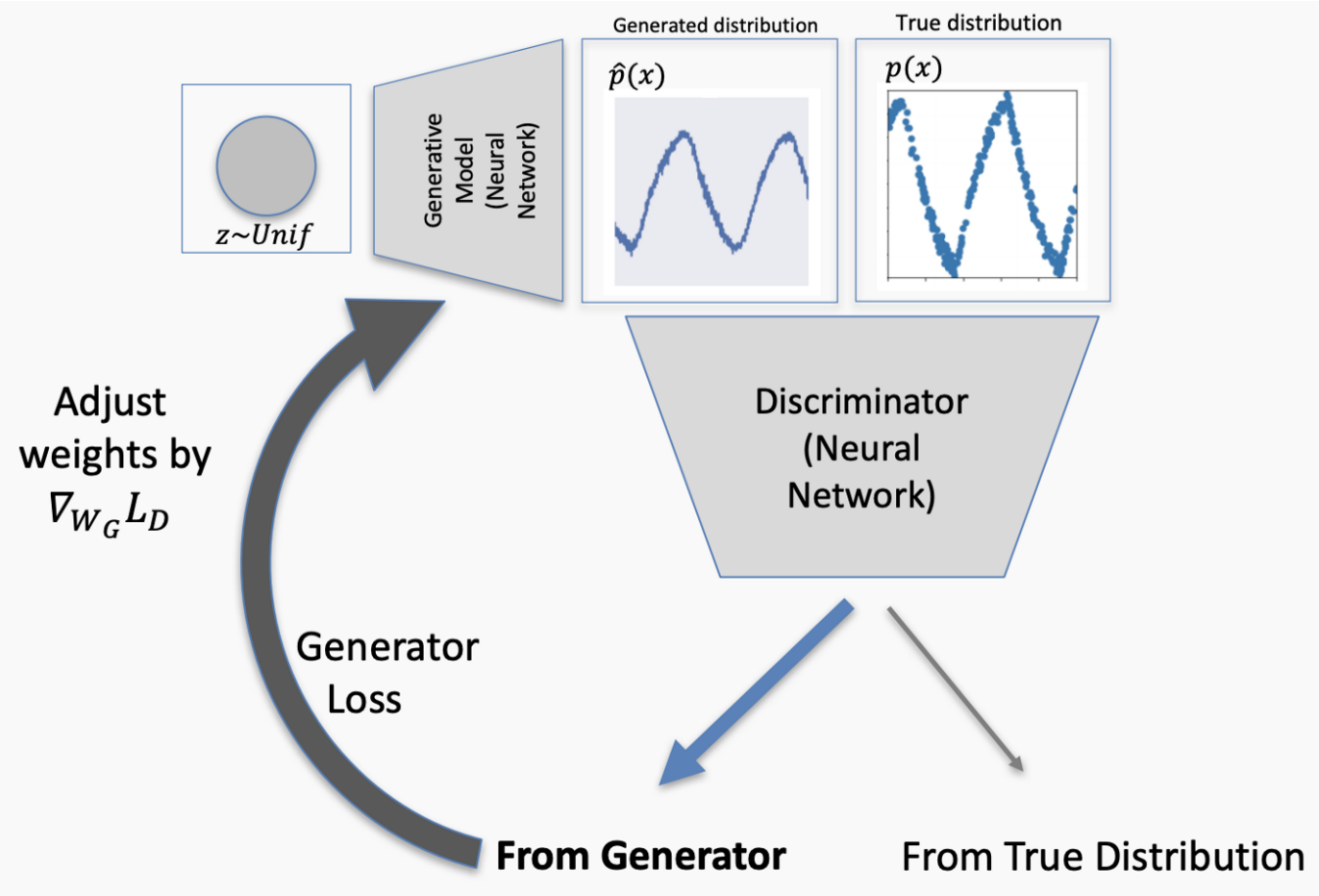


The generated distribution has once again been updated, and now the discriminator has been fooled, it thinks the generated data is from the true distribution. Time to update the discriminator!



The loss function is used to update the discriminator weights through backpropagation.

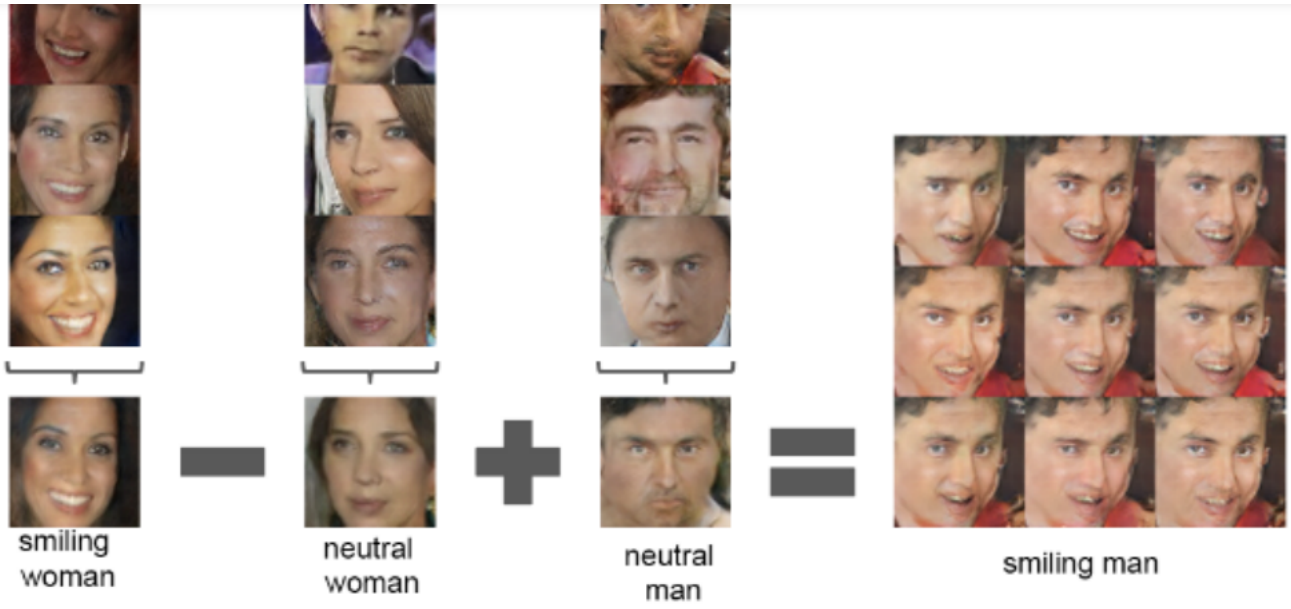




Once our network is built and trained, we can use the generator to produce images that are indistinguishable from the training images, such as the following example of a DC-GAN used on the standard MNIST dataset.

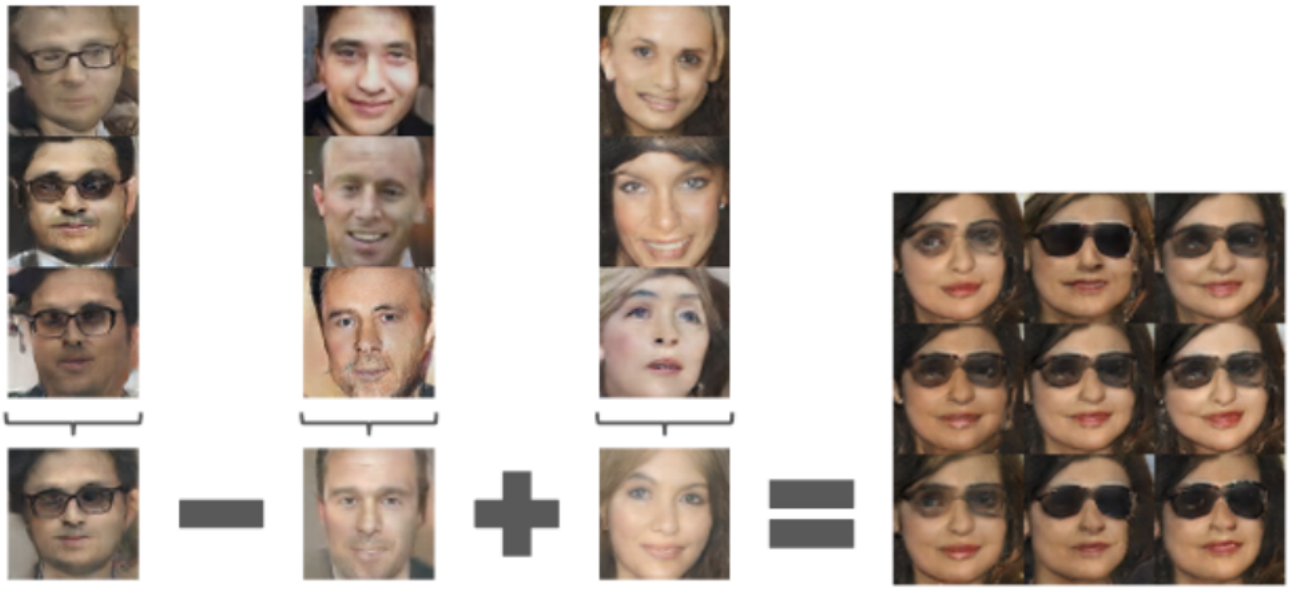
DC-GAN on MNIST

One very interesting application with GANs is the addition or removal of different attributes, illustrated below where smiles are ‘added’ to images without changing other attributes. This can also be done in video editing, and in the future, it may even be possible to post-edit videos to remove or add different actors in a similar manner. Similar things are already being done in the world of DeepFakes (although many applications of this could be considered malicious in nature).



Source: https://arxiv.org/pdf/1511.06434v2.pdf

This can also be done with other traits, such as sunglasses.



Source: https://arxiv.org/pdf/1511.06434v2.pdf

The above idea is essentially how the horses to zebra transition images were obtained at the start of this article.

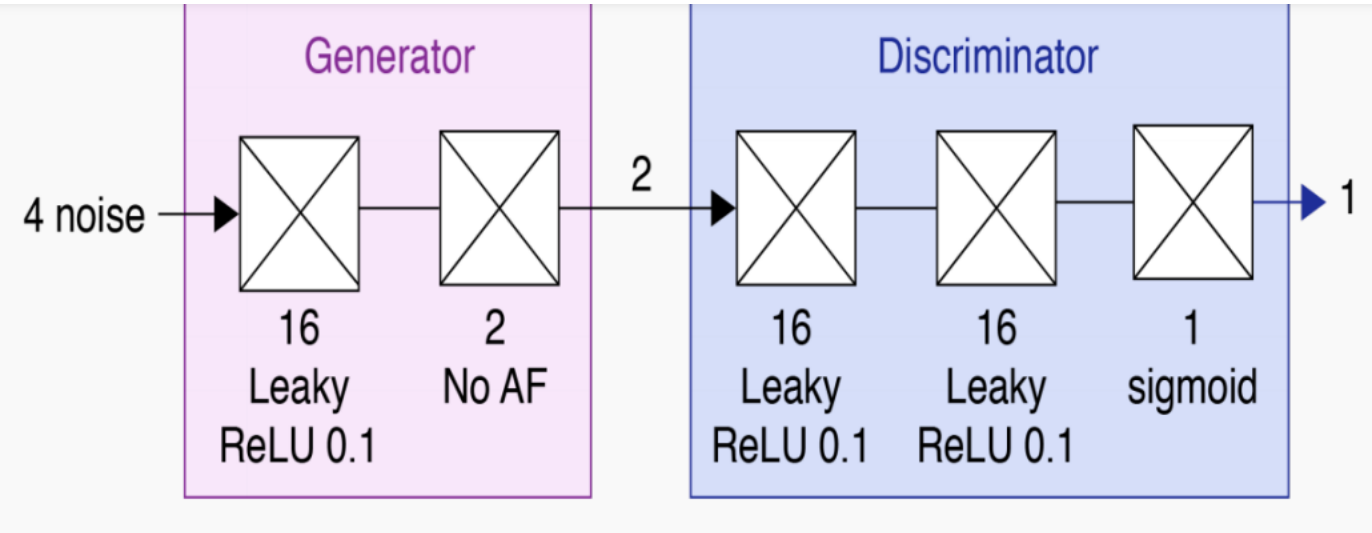
To give you an idea of just how much this area has improved in the past few years, look at the evolution of GANs from 2014 to 2017 from the images below.

Also, to give you an idea for how many 'flavors' on GAN exist, there are a lot. The GAN that we are producing by the end of the tutorial will be a DCGAN, although I will describe the Wasserstein GAN (WGAN) in more detail in part 2. I will also outline how GANs can be used for generating time series, not just images.

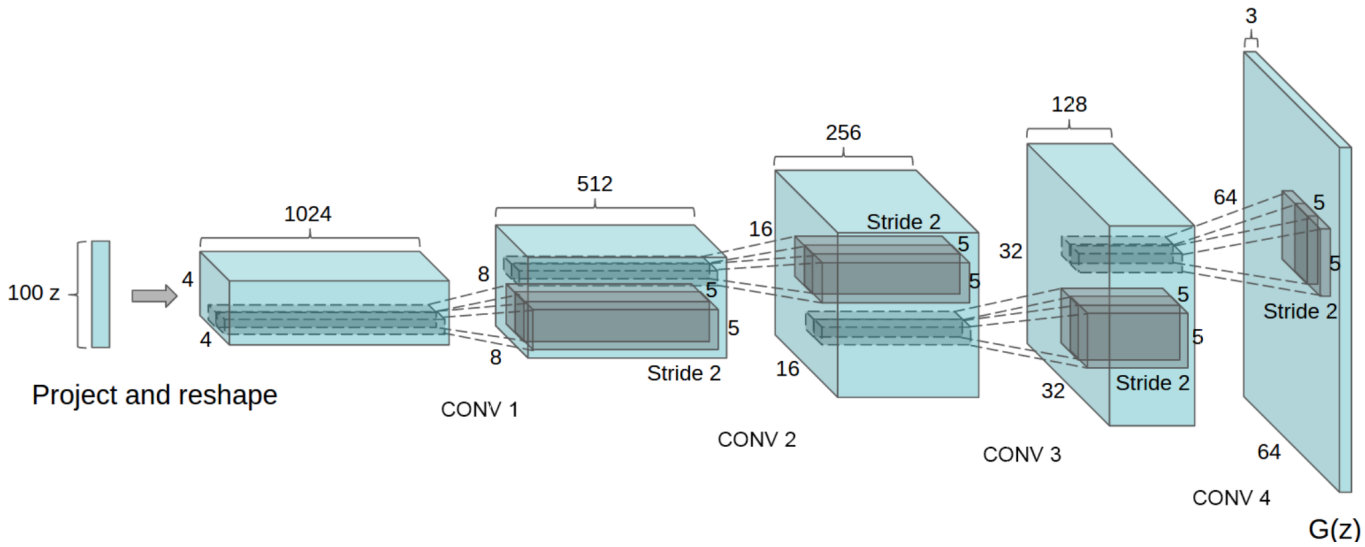
DCGAN	SGAN	AffGAN	RTT-GAN	SL-GAN
WGAN	SimGAN	TP-GAN	GANCS	Context-RNN-GAN
CGAN	VGAN	IcGAN	SSL-GAN	SketchGAN
LAPGAN	iGAN	ID-CGAN	MAD-GAN	GoGAN
SRGAN	3D-GAN	AnoGAN	PrGAN	RWGAN
CycleGAN	CoGAN	LS-GAN	AL-CGAN	MPM-GAN
WGAN-GP	CatGAN	Triple-GAN	ORGAN	MV-BiGAN
EBGAN	MGAN	TGAN	SD-GAN	
VAE-GAN	S^2GAN	BS-GAN	MedGAN	
BiGAN	LSGAN	MalGAN	SGAN	

Network Construction

The two main types of networks to construct are either fully connected (FC) GANs or Deep Convolutional GANs (DC-GANs). Which you use will depend on the training data you are submitting to the network. If you are using single data points, an FC network is more appropriate, and if you are using images, a DC-GAN is more appropriate. The difference architectures for the two networks are shown below.



Fully connected GAN



Deep Convolutional GAN (DC-GAN) — Alex Radford et al. 2016

Some of the rules of thumb to consider when using GANs are:

- Max Pooling is BAD! Replace all max pooling with convolutional stride.
- Use transposed convolution for upsampling.
- Use batch normalization.

We will discuss these more in the following section on GANHACKs.

GAN Rules of Thumb (GANHACKs)

[1] Normalize the inputs — Normalize the images between -1 and 1, and make sure to use tanh as the last layer of the generator output.

[2] Use Spherical Z — Don't sample from a uniform distribution. When doing interpolations, do the interpolation via a great circle, rather than a straight line from point A to point B. I recommend looking at Tom White's Sampling Generative Networks reference code https://github.com/dribnet/plat which has more details about this.

[4] Avoid Sparse Gradients: ReLU, MaxPool — the stability of the GAN game suffers (a lot) if you have sparse gradients. In general, leaky ReLU is good (in both the generator and discriminator).

- **For downsampling, use:** Average Pooling, Conv2d + stride
- **For Upsampling, use:** PixelShuffle, ConvTranspose2d + stride

If you are not familiar with PixelShuffle, there is an entire paper about it that you can read here: https://arxiv.org/abs/1609.05158.

[5] Use Soft and Noisy Labels — Label smoothing, i.e. if you have two target labels: Real=1 and Fake=0, then for each incoming sample, if it is real, then replace the label with a random number between 0.7 and 1.2, and if it is a fake sample, replace it with 0.0 and 0.3 (for example). This is a recommendation from Salimans et. al. 2016. An alternative is to make the labels noisy for the discriminator: occasionally flip the labels when training the discriminator.

See **GANHACKs** (https://github.com/soumith/ganhacks) for more tips.

GAN Challenges

There are a lot of problems with GANs, but I will touch on the main ones in this section and will discuss these more in part 2.

[1] Sensitivity — The biggest challenge to using GANs in practice is their sensitivity to both structure and parameters. If either the discriminator or generator gets better than the other too quickly, **the other will never be able to catch up**. Finding the right combination can be very challenging. Following the rules of thumb we discussed above is generally recommended when we're building a new GAN or DC-GAN.

[2] Convergence — There is no proof that a GAN will converge. GANs do seem to perform very well most of the time when we find the right parameters, but there's no guarantee beyond that. The more complicated the network gets, the more finicky the convergence becomes and the more difficult hyperparameter selection becomes.

[3] Big Samples — Trying to train a GAN generator to produce large images, such as 1000x1000 pixels can be problematic. The problem is that with large images, it's easy for the discriminator to tell the generated fakes from the real images. Many pixels can lead to error gradients that cause the generator's output to move in almost random directions, rather than getting closer to matching the inputs. The best procedure for training GANs on large images is:

- Start by resizing the images: 512x512, 128x128, 64x64, … ,4x4.
- Then build a small generator and discriminator, each with just a few layers of convolution.
- Train with the 4 by 4 images until it does well.
- Add a few more convolution layers to the end network, and now train them with 8 by 8 images.
- Again, when the results are good, add some more convolution layers to the end of each network and train them on 16 by 16 images.

This process takes much less time to complete than if we'd trained with only the full-sized images from the start (and is more likely to converge).

[4] Computation — Compute power, memory, and time to process large numbers is already very high. Running the networks until realistic images are produced can require many hours or days of training, even with high-performance GPUs (this is why our final images are sub-standard compared to those in some papers). This is further exacerbated for more complicated networks, larger training sets, and larger images.

[5] Modal Collapse — This is possibly the most frustrating problem that we encounter in GANs (apart from the 10-hour training times). Let's say I would like to use GAN to produce faces like the ones below from NVIDIA (shown below).

Generated images from NVIDIA GAN.

However, when training our network, the generator somehow finds one image that fools the discriminator.

A generator could then just produce that image every time independently of the input noise. The discriminator will always say it is real, so the generator has accomplished its goal and stops learning. However, the problem is that **every sample made by the generator is identical**.

This problem of producing just one successful output over and over is called **modal collapse**.

This is much more common when the system produces the same few outputs or minor variations of them. This is called partial modal collapse.

The solution is:

- Extend the discriminator's loss function with an additional term to measure the diversity of the outputs produced.
- If the outputs are all the same, or nearly the same, the discriminator can assign a larger error to the result.
- The generator will diversify because that action will reduce the error

Final Comments

This was a very long article but I hope you now have a very good intuition for how these networks work. As a reward for making it this far in the article, here is a Tweet from Ian Goodfellow, the creator of the original GAN, showing you an interesting situation where GANs can fail when trained on cat images, some of which are memes!

@goodfellow_ian

Follow

One of my favorite samples from the Progressive GANs paper is this one from the "cat" category. Apparently some of the cat training photos were memes with text. The GAN doesn't know what text is so it has made up new text-like imagery in the right place for a meme caption.

11:41 AM - 3 Dec 2017

If you are hungry for more, you can continue to part 2 of the tutorial:

Advanced Topics in GANs

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

towardsdatascience.com

Part 3 of the tutorial can also be found 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.

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:

- DCGAN https://arxiv.org/pdf/1511.06434v2.pdf
- Wasserstein GAN (WGAN) https://arxiv.org/pdf/1701.07875.pdf
- 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

- Learning from Simulated and Unsupervised Images through Adversarial Training (SimGAN) https://arxiv.org/pdf/1612.07828v1.pdf