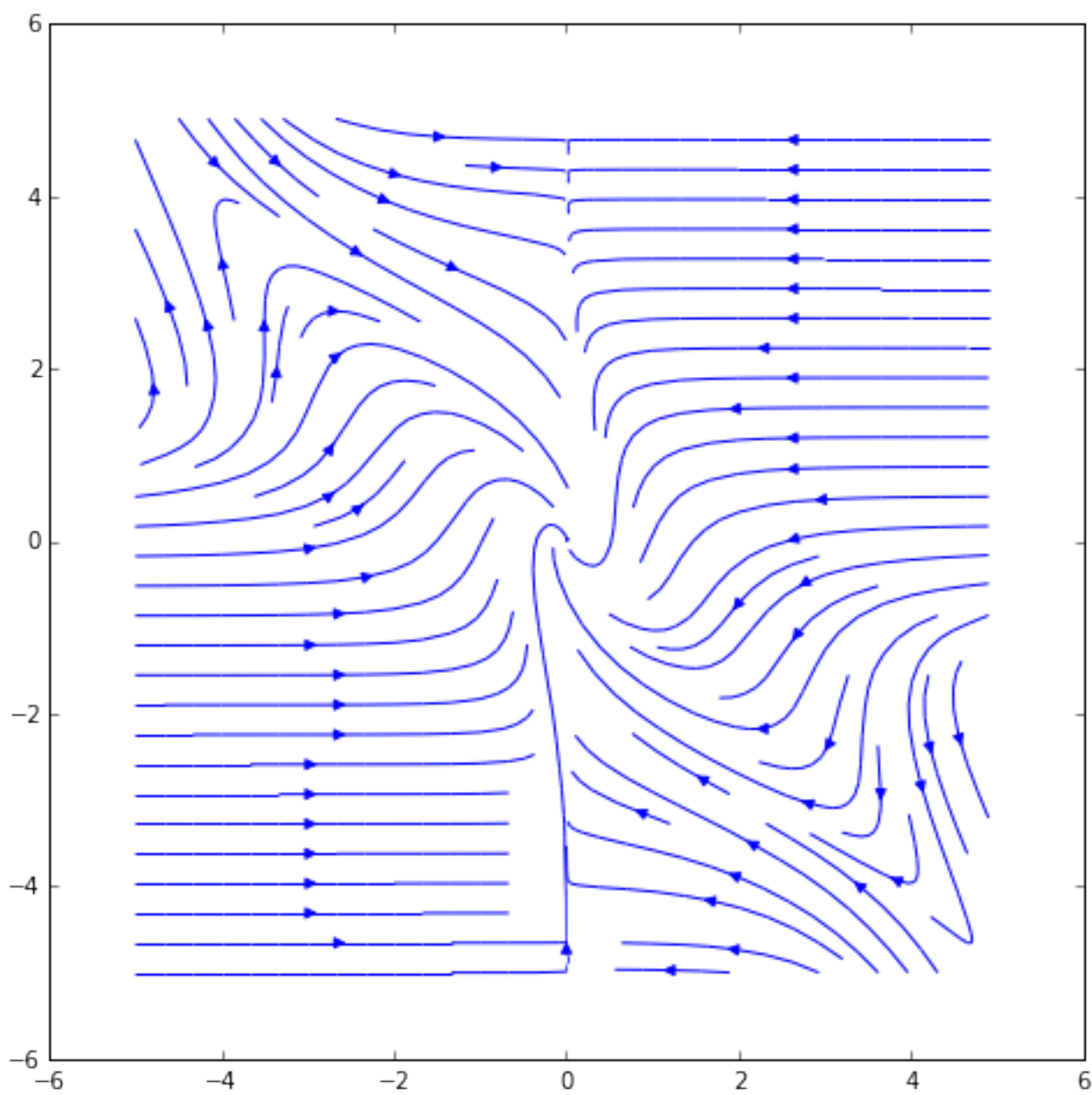


05  
OCT



# Stability of Generative Adversarial Networks

By Nicholas Guttenberg

**Generative Adversarial Networks** (and the related **DCGAN**) are a relatively new type of neural network architecture which pits two sub-networks against each-other in order to learn very realistic generative models of high-dimensional data (mostly used for **image synthesis**, though extensions to **sound**, **text**, and other media have been constructed). When one generates something like an image, the pixels themselves are not individually so responsible for the realism of the image as the correlations and relatively arrangements of pixels, and so hand-crafted loss functions to measure reconstruction accuracy have a tendency towards blurriness or other perceptual artefacts. But when you allow two networks to compete with each-other: one to make a plausible fake, the other to distinguish forgeries from reality, then the networks (in principle) eliminate the noticeable differences from reality one-by-one. A blurry image can be detected, so it eliminates blurriness, and so on. The results of this approach are visually quite impressive, but because the two sub-networks are trained against different, opposed target functions, there is a wealth of new instabilities and problems that can crop up compared to training traditional neural networks which have a single unified loss function.

In this post, I'm going to look at extremely simple adversarial networks – ones with only one parameter for each sub-network – so that we can exhaustively explore the parameter space, and hopefully get some intuition for how these adversarial networks behave when trained.


## Landscapes and Vector Fields

When training the usual neural network architecture, there is some kind of objective which the network is tuned to optimize (the loss function), and training proceeds by performing gradient descent on the parameters of the network to try to increase or decrease this function. Since it's just a scalar function, it defines a kind of landscape  $F(x, y, \dots)$  – a given set of network parameters maps to a certain value of the function – potentially expensive to compute, but very well-defined. If that function changes in the right direction, you know that training is proceeding correctly, and if it goes in the wrong direction you know that you should stop, reduce the learning rate, etc. If you make the learning rate small enough, be careful enough, you have a pretty strong guarantee that you can make it so that the loss function is always going in the right direction up until the point where it finds some local optimum. You can ultimately be sure that the network can't possibly get any worse at the task than it already is (at least, on the training data), so the worst thing that can happen is for it to get stuck somewhere. Even in that case, if you have two networks trained completely separately from different initial conditions and so on, you can always compare them and see which is better just by comparing their loss functions.

Search

Search ...

Language Switcher

 日本語

Recent Posts

- Collaboration with Prof. Amari
- Article published 4/27/2017
- Paper published 3/16/2017
- Paper published 3/14/2017
- Paper published 3/2/2017

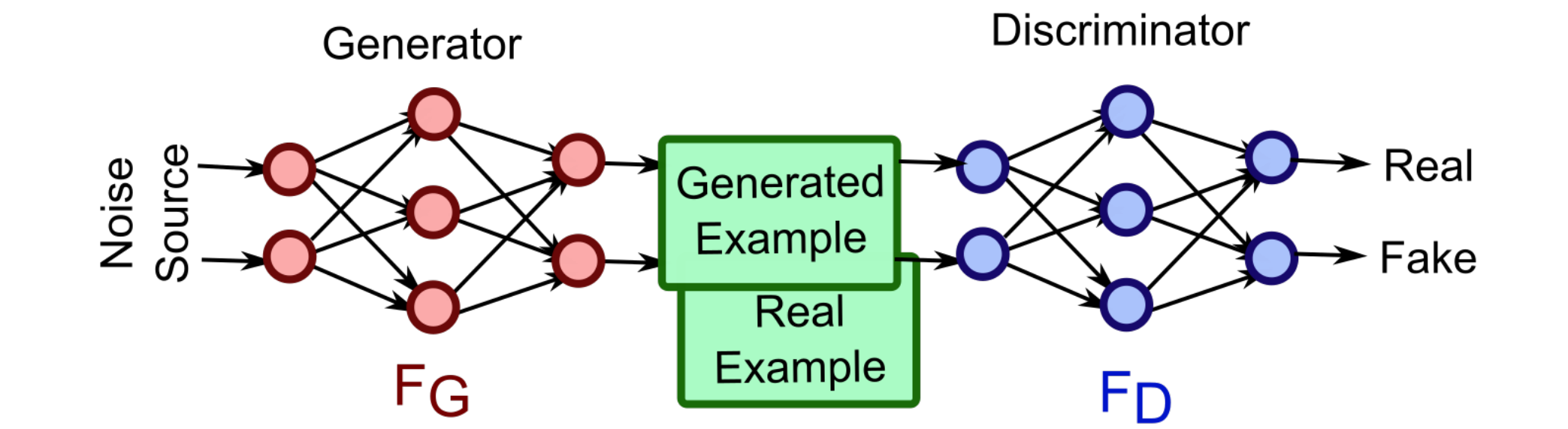
Recent Comments

- Kaih Chen on Stability of Generative Adversarial Networks
- Nicholas Guttenberg on Stability of Generative Adversarial Networks
- Kaihu Chen on Stability of Generative Adversarial Networks
- Nicholas Guttenberg on Stability of Generative Adversarial Networks
- Kaihu Chen on Stability of Generative Adversarial Networks

Archives

- May 2017
- March 2017
- February 2017
- January 2017
- December 2016
- November 2016
- October 2016
- September 2016
- August 2016
- July 2016
- June 2016
- January 2016
- December 2015
- November 2015
- September 2015
- August 2015
- July 2015
- April 2015
- March 2015





For adversarial networks, the structure is a bit different. The first sub-network, the Generator, takes a random variable and outputs what it thinks an example from the data set that can fool the other network looks like. The second network, the Discriminator, takes as inputs both real examples and fake examples, and is tasked to classify them. So the Discriminator has a loss function  $F_D(W_D, W_G, \hat{y})$ , but the Generator has a different loss function  $F_G(W_D, W_G)$ . These loss functions are chosen to guarantee the theoretical result that if the generator can output samples from an arbitrary probability distribution  $g(x)$  and the discriminator can assign an arbitrary probability to a sample  $d(x)$ , then for data distributed according to a true distribution  $p(x)$  there should be a fixed point when  $g(x) = d(x) = p(x)$  – that is, the correct answer to the problem is a point at which neither the discriminator nor generator can improve their individual loss functions.

However, this is a much weaker guarantee than you have in gradient descent on a landscape – you don’t know whether this fixed point is stable, or if one even exists if the generator and discriminator are finite networks. When training both networks in parallel, there isn’t guaranteed to be some objective way to measure the improvement of the network as a whole – you can’t read out improvements or convergence by examining a training curve. As this is a generalized dynamical system, you can get limit cycles and strange attractors, meaning that training can go in circles or become chaotic.

**That said, GANs work shockingly well.**

Even if they can be hard to tune and to get working, there are [many papers](#) and open-source [implementations](#) showing [impressive](#), stable results from GANs. When GANs work, they learn *fast* – producing realistic images even after a single epoch of training. So even if these kinds of troublesome behaviors are possible, they are either not guaranteed to be a problem or often they simply don’t matter when it comes to GANs producing reasonable outputs. At the same time, tuning GANs is currently somewhat of an art form, and requires significant human intervention and inspection of the results, making it hard to extend to tasks where a human cannot readily judge gradations in quality of different sets of outputs. There was a [recent paper](#) which tried to impose a scalar landscape onto the GAN training, essentially treating the adversarial part as a perturbation on top of a more traditional, well-defined training problem. So even if GANs are working, there’s reason to try to figure out what’s going on, objectively evaluate their performance, and move back towards scalar optimization if possible or beneficial.

## The Really Simple GAN

If I want to plot how a GAN would train in a way that we can evaluate by eye, it really needs to be in 2D (or 3D at most). So that leaves me with two trainable parameters, across two sub-networks. This is going to be a very simple GAN.

Given how few variables I have, lets try to make a GAN which takes random numbers from a 1D Gaussian as input, and generates samples from a 1D Gaussian with the same standard deviation but potentially with a shifted mean as output. And actually, lets make it a trivial problem – the mean is the same too, so the best generator should just pass its input through without modification. Keeping it as simple as possible, my generator ‘network’ is going to be:

$$g(z) = W_g z + b_g$$

where  $z$  is the latent variable input, a unit-standard deviation zero-mean Gaussian. So this network just scales its input by  $W_g$ , applies a constant shift  $b_g$ , and has no nonlinearity. But this is going to be too many parameters to visualize, so lets just restrict  $W_g = 1$  for now and let  $b_g$  vary.

My discriminator ‘network’ is going to be:

$$d(x) = \sigma(W_d x + b_d)$$

where  $\sigma(\dots)$  is a sigmoid function and  $x$  is a sample either from the generator, or from data (which is a unit Gaussian with zero mean). Again, too many parameters to visualize, so lets just fix  $W_d = 1$  for now.



The loss function for the discriminator is the log loss:

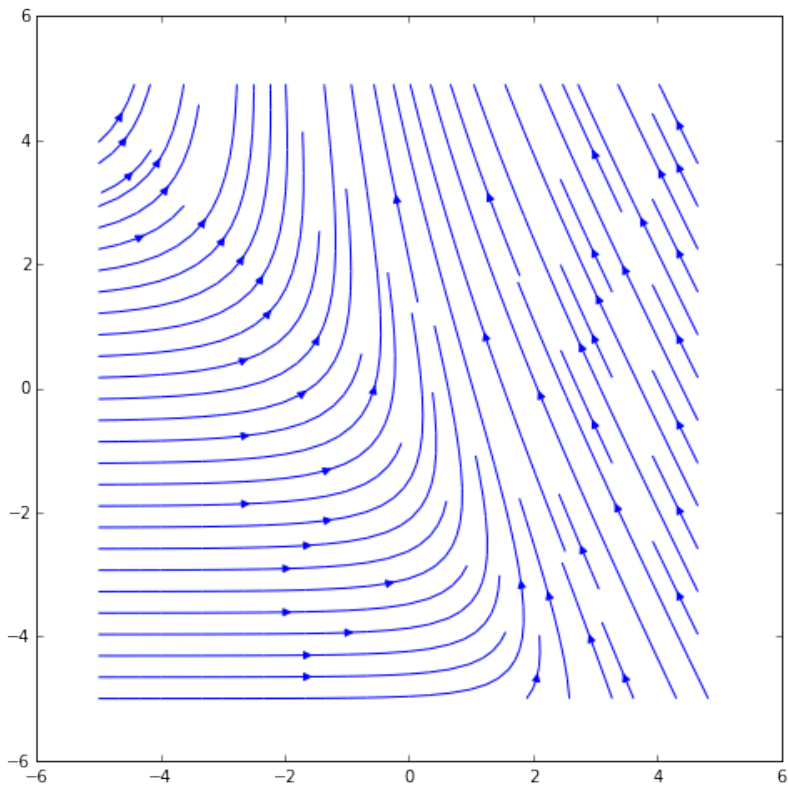
$$L_d = -\hat{y} \log(d(x)) - (1 - \hat{y}) \log(1 - d(x)),$$

where  $\hat{y} = 1$  if  $x$  is a real sample, and  $\hat{y} = 0$  if  $x$  is a fake. The loss function for the generator is just the negative of the loss for the discriminator (it wants to output a sample that fools the discriminator), but for the generator all examples are going to be fake, so this is just  $L_g = \log(1 - d(g(z)))$ .

Given this set of networks, we can calculate the gradient vector, if we were just training these two networks together via gradient descent against their respective loss functions  $-(\frac{\partial L_g}{\partial b_g}, \frac{\partial L_d}{\partial b_d})$ , and plot that as a function of  $b_g$  and  $b_d$ . The results are a bit worrying: there doesn't appear to be any kind of finite fixed point, so this training will never converge. Instead, it looks as though the generator parameter  $b_g$  will become positive infinity, and the discriminator parameter  $b_d$  goes to negative infinity. What gives?

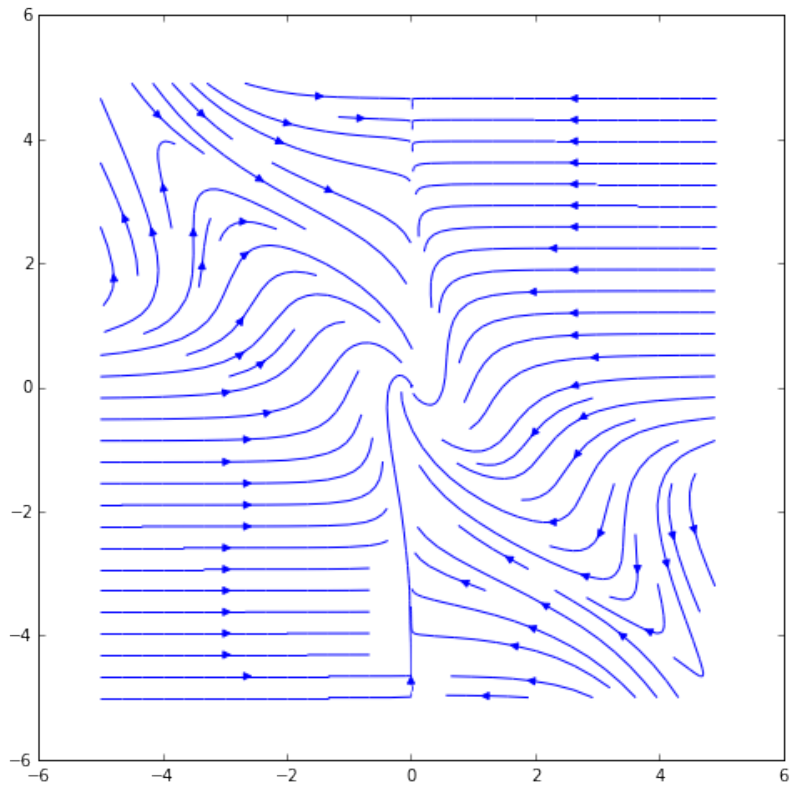
It turns out that this discriminator has a critical blind spot. It's using a sigmoid as the output nonlinearity, which is pretty normal for making a network with a probabilistic output on a single binary question. But in this case, the discriminator is too weak to properly classify both  $x \rightarrow \infty$  and  $x \rightarrow -\infty$  as unrealistic samples. It has to be one or the other. If the discriminator decides that  $x = -\infty$  is unrealistic, then  $x = +\infty$  has to be the 'most realistic sample', and vice versa. Even worse, in this case because  $W_d$  is fixed, the discriminator can never declare a larger value of  $x$  less realistic than a smaller one!

So in order to always win, the generator just has to hide out at  $+\infty$ , and there's nothing the discriminator can do about it. Because the discriminator lacks the capacity to properly model the probability distribution that the network is being asked to generate, the result is a fundamental instability of the training process in this case – no matter how slow you go or how careful you are, you won't converge to the data distribution.



Gradient descent flow in the  $b_d$  vs  $b_g$  space for the sigmoid classifier.

Fortunately, this kind of problem may only exist in these artificially tiny networks. It's known that if you give a neural network enough nodes, they can act as universal function approximators. That is to say, if a given network can't model a particular function well enough, you can in principle always just make the network bigger. So presumably this kind of problem is one that goes away when you're working with large enough networks. But, correspondingly, it suggests that a discriminator network which is too small may actually be **unstable** for a GAN, whereas it would simply underfit in a standard neural network)



Gradient descent flows for a Gaussian classifier, now with stable fixed point.

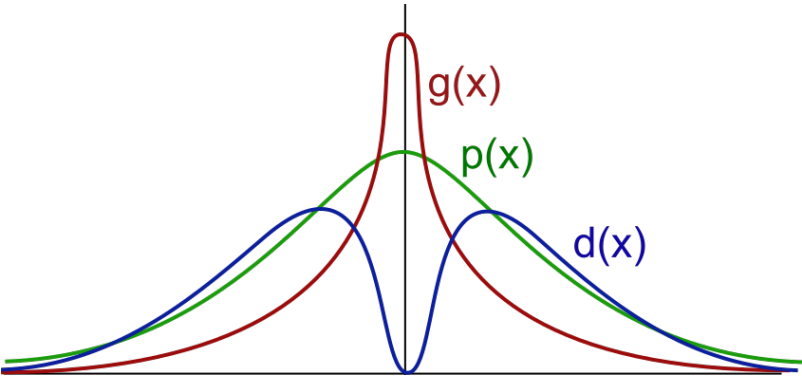
For our tiny network, the asymptotic behavior of the sigmoid function is the source of the problem, but we can replace the sigmoid nonlinearity with a Gaussian. So that way we know the discriminator can exactly, accurately model the true distribution:  $d(x) = \exp(-(W_d x + b_d)^2)$ . This appears to work much better. There's now a clear stable fixed point at (0,0), though the approach to the fixed point is oscillatory. It turns out that even if we force  $W_d$  to the wrong value, e.g.  $W_d = 2$  or  $W_d = 0.5$ , we still get the stable fixed point in the middle – so not all errors or limitations of the discriminator lead to a fundamental training instability.

It appears that the biases are doing the right thing now – this network correctly discerns the mean of the data Gaussian, albeit in a somewhat round-about way. So what if we do the same thing with fixed biases ( $b_d = b_g = 0$ ), but now look at training the weights  $W_g$  and  $W_d$ ? In principle, this should have a fixed point at (1,1) and at (-1,-1) (because the Gaussian with zero mean is symmetric under reflection). What actually happens?

It's a bit hard to see from the plot, but something is going wrong. The horizontal axis ( $W_d$ ) seems to be doing alright – there are fixed points close to  $\pm 1$ , so that's okay. But the vertical axis ( $W_g$ ) appears to be collapsing to

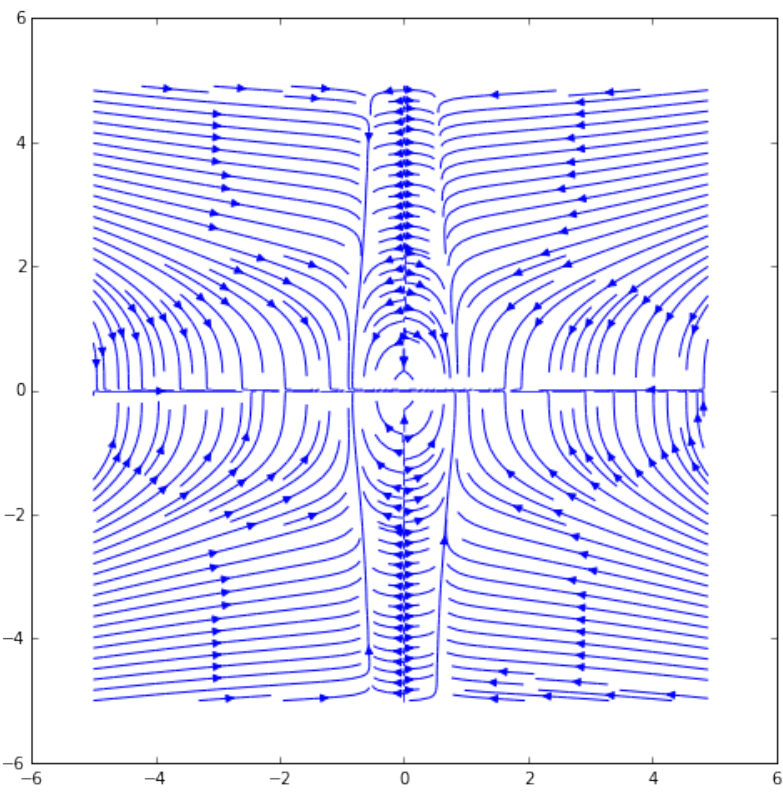


zero! That means that the generator is outputting a delta function at  $x = 0$ , not a Gaussian with the same standard deviation as the data distribution! Why?



Again, this is a problem with the discriminator not being powerful enough. In the theoretical picture

on GANs, if the generator deviates at all from the true  $p(x)$ , there's always some function  $d(x)$  the discriminator can come up with to get some extra advantage. But that  $d(x)$  function in this case would not be a Gaussian, it would be something bi-modal with a dip in the center where the generator has decided to focus. So even if the discriminator is capable of modeling the true distribution exactly, that may not be sufficient – it needs to be able to cover some set of fluctuations around the true distribution in order to prevent the GAN from exploiting those vulnerabilities.



Gradient descent flows for the weight matrices.

Essentially, the discriminator needs to be able to model the true distribution plus **every possible perturbation around it that the generator could come up with.**

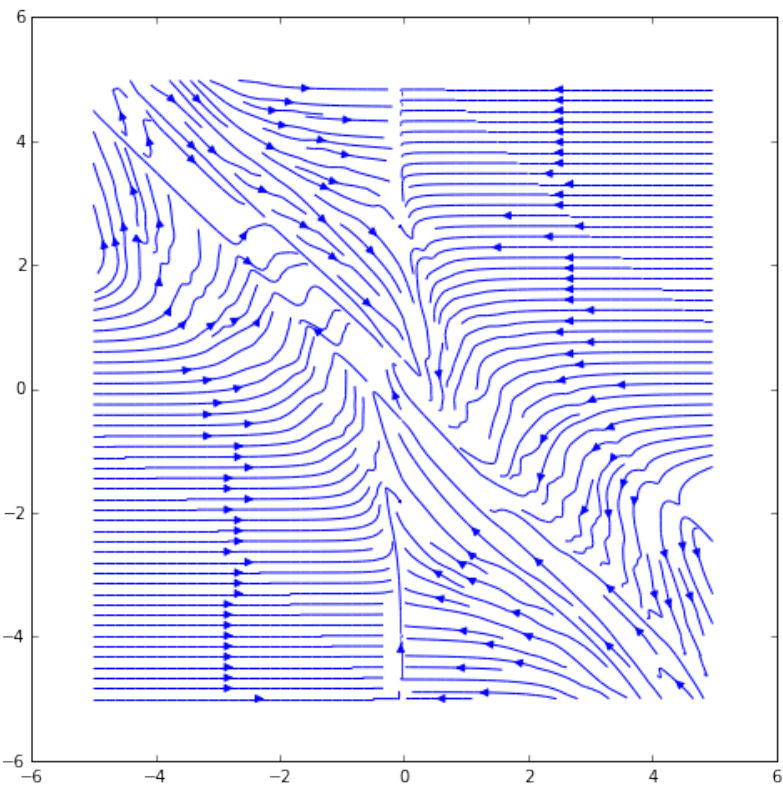
The universal function approximation properties of neural networks can actually work against things here – if the generator is too powerful compared to the discriminator, it could just find some function the discriminator can never answer.

The outcome mirrors a problem that does actually occur in higher-dimensional GANs. If we look at what happens to the data generated here during this malfunction, the generator is only outputting copies of the single most likely data point – the diversity of the generated set has collapsed. This kind of collapse happens in image-generation GANs as well, where if the generator gets too far ahead it just memorizes one example from the training set and only outputs that regardless of the variations in the noise source. Once the network has fallen into that hole, it's usually very hard to recover the entropy of the outputs with further training. The usual protocol is to adjust the learning rates to keep the error of both the generator and discriminator from getting too small, but perhaps the reason why this kind of careful management is necessary is that there are true fixed points in the parameter space corresponding to this collapse, and unless those fixed points are somehow destabilized then the networks will always be at risk of going there if you continue training indefinitely.

## Data Dependency

Another question we can ask with these visualizations is, how much data is needed and how does the vector field respond to having insufficient data? In a standard single network case, the usual result is overfitting, and the signature in the loss function landscape is that the local and global minima become very sharp. So far, these plots have been generated using 2500 samples. What if we go down to 100 samples? We'll examine the  $b_g, b_d$  plane  $W_g = W_d = \mathbf{1}$  for this one, since training is ostensibly stable and correct in that set of parameters.

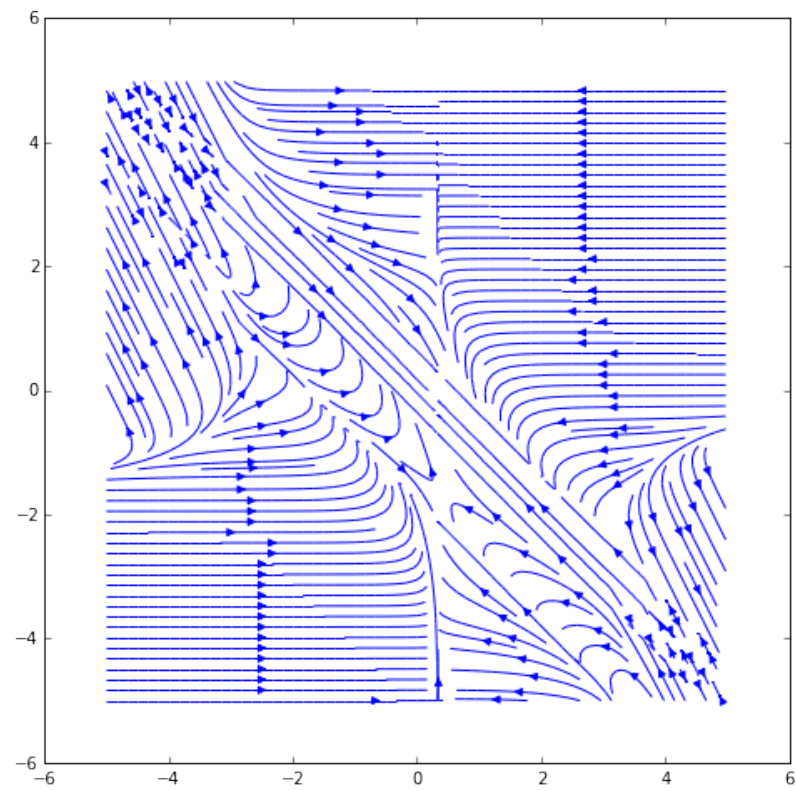
With only 100 samples, we get a much more ragged set of contours. On a 'landscape' like this, the wrong step-size could easily end up running off to somewhere weird, getting further away from the fixed point, getting trapped in parasitic cycles, or accidentally escaping the approach to the fixed point and having a 'collapse'. 100 samples is tiny for a data-set, but keep in mind that this is a 1D data set, so what constitutes small enough to create snags might be a much larger number of samples in higher dimensions where you can't possibly cover the entire space.



With only 100 samples, the gradient descent becomes a bit messy.

We can reduce the amount of data further, down to 5 samples, and this becomes a real nightmare. It's unclear whether gradient descent will find the fixed point anymore, no matter how careful you are. The regions at (-4,4) and (4,-4) become essentially noise patterns without any clear escape trajectory. An additional saddle point has appeared around (1,-2), and it looks as though there may be a number of local minima. Of course, asking anything from 5 samples is unreasonable, but this gives an idea of how things look when they break.





Gradient descent in the bias space with only 5 data samples.

To do this, we sample each gradient 30 times, but adding Gaussian random noise with standard deviation 0.2 to the parameters ( $b_d, b_g$ ). We keep the same data and latent vectors (100 samples for this plot), but basically just average over the local neighborhood in the parameter space. The results are much smoother (of course), and also much more directly approach the fixed point. So the lesson for larger GANs is:

Yes, **regularization matters for GANs!**

Sample a lot of random variations of the parameters (e.g. through dropout), apply  $L_2$  normalization to the weights and biases to shut down fixed points at infinity, etc – all of those things should help with the instabilities and problems we found here.

To do this, we

sample each gradient 30 times, but adding Gaussian random noise with standard deviation 0.2 to the parameters ( $b_d, b_g$ ). We keep the same data and latent vectors (100 samples for this plot), but basically just average over the local neighborhood in the parameter space. The results are much smoother (of course), and also much more directly approach the fixed point. So the lesson for larger GANs is:

Yes, **regularization matters for GANs!**

Sample a lot of random variations of the parameters (e.g. through dropout), apply  $L_2$  normalization to the weights and biases to shut down fixed points at infinity, etc – all of those things should help with the instabilities and problems we found here.

Sample a lot of random variations of the parameters (e.g. through dropout), apply  $L_2$  normalization to the weights and biases to shut down fixed points at infinity, etc – all of those things should help with the instabilities and problems we found here.

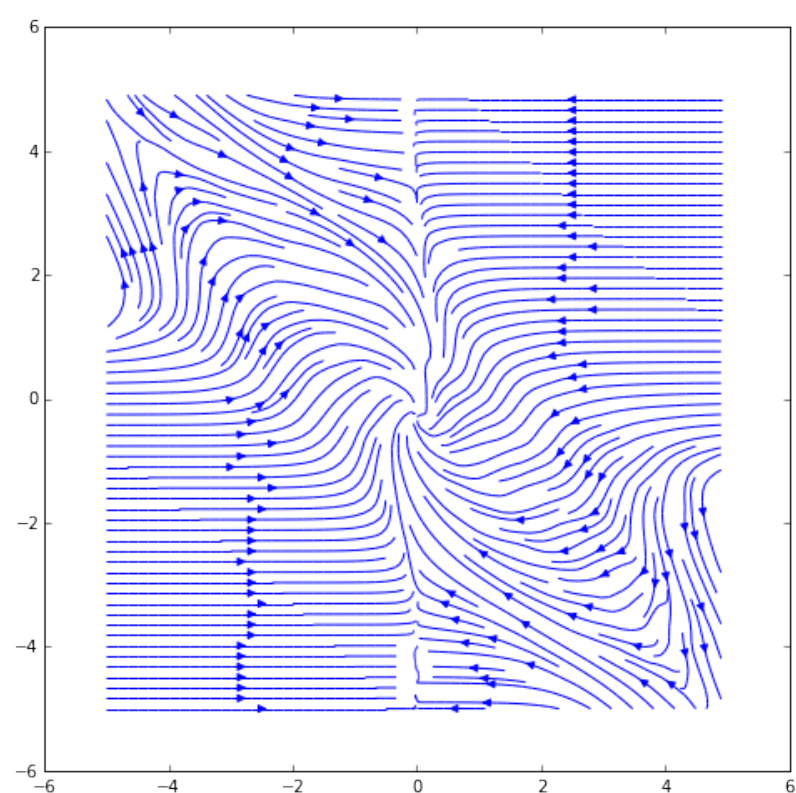
Sample a lot of random variations of the parameters (e.g. through dropout), apply  $L_2$  normalization to the weights and biases to shut down fixed points at infinity, etc – all of those things should help with the instabilities and problems we found here.

Sample a lot of random variations of the parameters (e.g. through dropout), apply  $L_2$  normalization to the weights and biases to shut down fixed points at infinity, etc – all of those things should help with the instabilities and problems we found here.

Sample a lot of random variations of the parameters (e.g. through dropout), apply  $L_2$  normalization to the weights and biases to shut down fixed points at infinity, etc – all of those things should help with the instabilities and problems we found here.

Sample a lot of random variations of the parameters (e.g. through dropout), apply  $L_2$  normalization to the weights and biases to shut down fixed points at infinity, etc – all of those things should help with the instabilities and problems we found here.

Is there a reasonable way to fix this kind of mess? In classic neural networks, one would use regularization – usually **dropout**. We can't do that here (we only have one link to turn off!), but what we can do instead is add noise to the parameters. Even as we're looking at these very low-data vector fields, its mostly the local structure that has become messy due to the small amount of data. The overall global structure remains quite similar in all cases, even the 5 sample case. So if we could somehow take a local average of the gradient direction over some radius, it might actually fix these local snags and make the training behavior more stable.



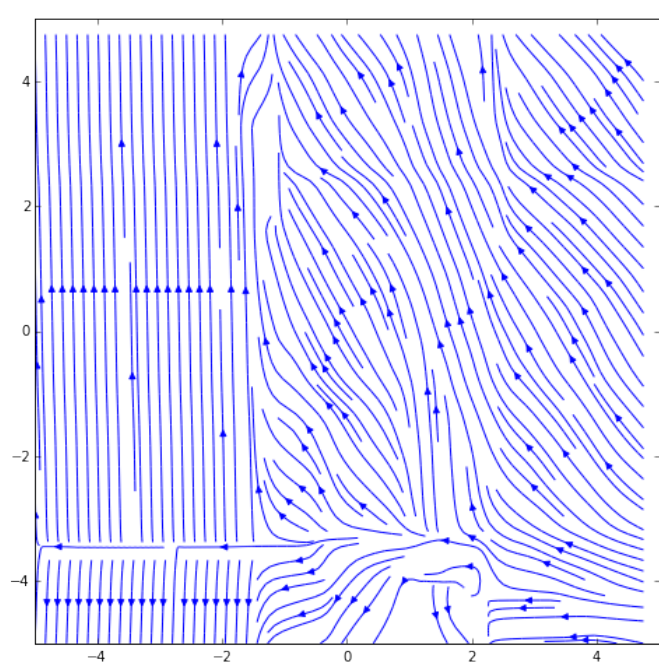
Gradient descent with parameter noise – much smoother, even though only 100 samples!

## Speculation on an Extension to Large Networks

Can we do anything like this analysis in full-sized GANs? After all, everything here could just be an artefact of the tiny, extremely weak 'networks' we're forced to use to make 2D visualizations. Well, its hard to say.

One thing we can in principle do is to take 2D slices out of the parameter space of the larger network, and look for fixed points, saddles, and run-off-to-infinity behaviors on those slices. On the left is a random cut taken out of the bias space of a randomly initialized GAN with two 100 neuron hidden layers for both the generator and discriminator, and it seems to show some of the same characteristics as our tiny network – in this case, it looks like the first bias plot with things appearing to run off to infinity. However, unlike our 2-parameter network, there is significant out-of-plane motion here which is not visualized – that means that something appearing to run off to infinity could just be on an oscillatory approach to a fixed point somewhere out-of-plane (which is probably the case here, as training this kind of 1D GAN tends to be relatively well-behaved for larger networks). So some caution in directly applying the visualization technique is probably necessary.

On the other hand, perhaps the **features** of these vector field landscapes as seen from the point of view of a network-in-training will be more robust and reliable. In dynamical systems theory, one metric which can be used to evaluate the overall behavior of a dynamical system is the **Lyapunov exponent** – it measures the rate at which nearby trajectories diverge from each-other, and detects the presence (and structure) of chaos in the system.




Random slice of the bias space for a large network.





So the idea is, by training a cluster of nearby networks together, one should be able to detect when the training process passes near saddle points, whether it’s sitting near a fixed point or just drifting off to infinity. I think that this must ultimately be related to Hessian-based update rules – the cluster of nearby trajectories is essentially locally sampling the second derivatives of the loss function along a random set of directions. The interesting aspect of it for GANs is that this higher-order gradient information won’t just determine the stable step size, but also might be able to tell you if you’re stuck in a limit cycle or crashing or things like that, and may be able provide the information needed to bias the training updates in such a way as to damp out those oscillations.

For example, one can generically decompose vector fields into a gradient of a scalar field and a curl of another vector field:  $\vec{v}(\vec{r}) = \vec{\nabla}\phi(\vec{r}) + \vec{\nabla} \times \vec{A}$ . If that decomposition can be done locally (which requires knowledge of the derivatives of  $\vec{v}$ , e.g. second-order derivatives of the loss function), then its possible to only follow the  $\vec{\nabla}\phi$  part of the field, recovering a scalar optimization problem.

So I think it might be interesting to look at efficient ways to approximate the Hessian as a way to get information to stabilize the training updates of GANs. But for now, that extends beyond the scope of this post.


 この記事を書いた人


 最新の記事





Nicholas Guttenberg


Share this:


 Twitter

 Facebook 39


 Reddit


 Tumblr

 Pinterest


 Google

This entry was posted in *BLOG*. Bookmark the *permalink*.

 Lessons from ALife XV – When is evolution smart?

Recurrent generative auto-encoders and novelty search

## 7 thoughts on “Stability of Generative Adversarial Networks”



J


8/10/2016 at 6:59 PM

“here’s always some function q(x) the discriminator can come up with to get some extra advantage. But that q(x) function in this case would not be a Gaussian, it would be something”

Here I think you meant to write `d(x)`, no?

Great article by the way! (I’m still reading but I’m loving it)

Reply




Nicholas Guttenberg

8/10/2016 at 9:59 PM

Thanks! Yes, I mean d(x). The q(x) notation is from some of the GAN papers, but I’ve changed it to all be consistently d(x) for the discriminator’s view of the distribution.

Reply



Kaihu Chen

18/11/2016 at 5:12 AM

Excellent write up. Thanks for sharing your insights!

I am doing research regarding using DCGAN to generate photo-realistic avatar from the facial images of one specific person. As you can imagine, my training dataset is typically very small (say, 64) and lack variety (since all images are of the same person). I manage to get DCGAN to converge and were able to generate usable images out of it. During my experiments I consistently observed the moving average of gLoss/dLoss (ratio of the loss values for the generator against the discriminator) trending up during the training session, which typically goes to 30 or higher within hours.

I wonder if the g/d ratio blowing up like this is typical in your experience, and whether it is a clear indication that the training will not improving much further.

Reply





Nicholas Guttenberg

18/11/2016 at 10:55 AM

Generally speaking, having a very small dLoss doesn't seem to be as bad as having a very small gLoss. If gLoss is very small, the generator has learned how to consistently fool the discriminator into thinking that the fake images are more real than reality – that usually seems to mean that its exploiting a blind spot that the discriminator can't learn to protect, or that it has found a single highly convincing sample and just collapsed on it. I've gotten decent results in some cases where I've maintained dLoss around 0.05 or 0.1 and maintained gLoss around the 0.8-1.4 range (where gLoss first blows up to around 4 or so, then slowly shrinks back down to 1), but it's not an absolute indicator either way – you can be in that range and be improving or getting worse or not changing (part of the difficulty of GANs is that there isn't a simple metric which tells you about convergence).

One thing people have tried is to maintain a population of discriminator networks and then use that population to estimate a more objective sample quality. The idea being that a given discriminator network that the generator is training against might get tricked, but something that fools it might not be the right pattern to fool a discriminator that the generator hasn't been exposed to (the paper is here: <https://arxiv.org/abs/1602.05110>). I haven't tried this technique, but it might give a better indication of whether your network is stagnating.

Reply



Kaihu Chen

20/11/2016 at 12:54 PM

Much appreciated for your pointer to the 'recurrent adversarial networks' paper! One other area that I have been exploring is using video as training dataset to DCGAN for creating the neural model of a specific person and then converting the model into an avatar (see my report here: <http://www.terraai.org/avatars/>). The interesting thing about video is that it is more or less a collection of consecutively similar images. I thought that such a property could be beneficial to convergence or stability during learning when data is batched properly (ref: the 'minibatch discrimination' technique, <https://arxiv.org/pdf/1606.03498v1.pdf>), but my very limited experimentation did not yet give clear indication on this. I wonder if you have any thoughts about it?

Reply



Nicholas Guttenberg

24/11/2016 at 4:01 AM

On the one hand, frame similarity means the entropy is smaller, which should help. On the other hand, generators have a tendency to collapse to just outputting a single example, so having a lot of similarity within-batch while having a lot of diversity between-batch seems like it'd be pretty bad. At the least, I'd shuffle the data before batching, to get the most out of minibatch discrimination.

Reply



Kaih Chen

25/11/2016 at 1:06 AM

Earlier I thought that the "minibatch discrimination" technique suggests putting similar images together. I will now certainly try it both ways to see how it turns out.

Happy Thanksgiving, Guttenberg san!

Reply

Leave a Reply

Your email address will not be published. Required fields are marked \*

Comment

Name \*

Email \*

Website

Post Comment

☐ Notify me of follow-up comments by email.

☐ Notify me of new posts by email.