A practical guide to understanding and applying Generative Adversarial Networks and their improvements
DRAFT

Quentin Garrido, Anthony Lafourcade, Raphaël Lapertot, Josselin Lefevre

June 2019

Contents

1 Introduction

The popularity of GANs is undeniable, giving state of the art results in various domains. Since their first appearance [3] in 2014 they have been improved numerous times and many variants have been developed.

However those improvements are not always easy to use in practice which is why we aim to provide a comprehensive guide on some of the major GAN improvements, studying how to use them in practice and analysing their performances.

We are going to first show how GANs work and what are the challenges that arise in practice, especially gradient vanishing and mode collapse.
We will then look at DCGANs, one of the most important improvement on GANs, and look at how they perform in comparison to traditional GANs.
Afterwards, we will look at different methods that were developed to help with performance and mode collapse, such as label smoothing, minibatch discrimination, virtual batch normalization and others. We will show that some methods indeed help with performance while others produce more mixed results.
Finally, we will study the theoretical shortcomings of GANs and look at a more theoretically robust type of GANs, Wasserstein GANs, which uses the Wasserstein distance as loss function. We will compare them to a DCGAN to see how they both perform in practice, with WGANs not encountering mode collapse and DCGANs producing sharper images.

We also provide implementations for every methods using Tensorflow 2 and Keras with notebooks explaining every methods, available at : https://github.com/garridoq/gan-guide

2 GAN

2.1 What is a GAN

GAN stands for Generative Adversarial Network. This is a very popular neural network architecture for generating data (here, we will mostly focus on images . . .). GANs use two models that will fight against each other :

- The generator, that will try to produce data that looks like it originates from the original distribution as closely as possible
- The discriminator, that will judge the performances of the generator by telling if a sample it sees has been generated or is part of our real data (training set), and giving a prediction for each sample (e.g. 1 for real data and 0 for fake data).

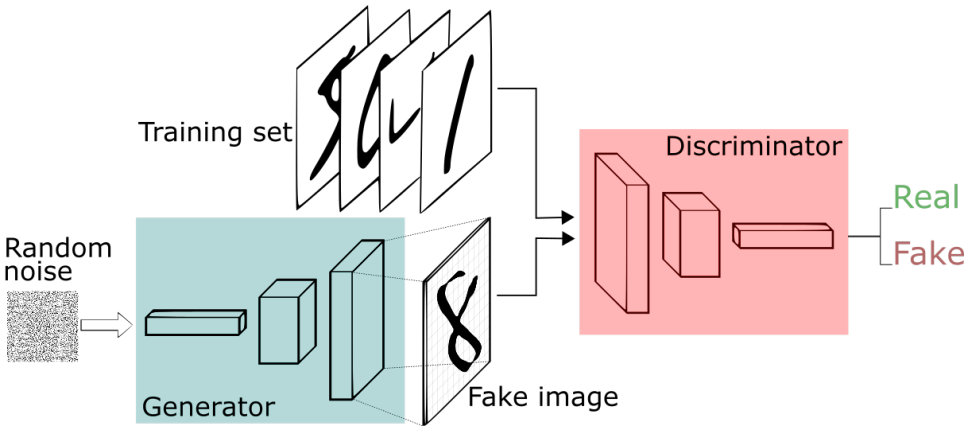

Figure 1: Gan architecture

Since we train two neural nets at once, the training is slightly different than usual. Here, it will be as follows :

- The generator generates some data using data from our latent space (usually a noise vector)
- The discriminator takes randomly either ground truth data, or data generated by our generator, and rates it, tells if it is true data or generated data
- The result provided by the discriminator is compared to the expected result, and then the discriminator is adjusted
- In the case where the given data to the discriminator was generated data, measure if that generated data fooled the discriminator or not. Then, adjust the generator

We can see here that the two models are fighting against each other: the generator tries to fool the discriminator, and the discriminator tries not to get fooled by the generator. This way, they will in theory constantly improve to beat each other, and we will eventually have a very good generator, that will generate similar data to our real data.

2.2 GAN architecture

In order to illustrate ourselves, we will use the CIFAR-10 dataset [1] for most examples, since MNIST [2] is too simple.
We will use the following architectures for our examples.

2.2.1 Generator

Here, we will use a convolutional neural network taking as input a noise vector z of length 100 taken from a normal distribution.

Layer	Kernel	Strides	Features	BN	Activation	Output shape
Input z						(100)
Dense, Reshape	N/A	N/A	$256 * 4 * 4$	Y	ReLU	(4, 4, 256)
Convolution + Upsampling	(5, 5)	(1, 1)	256	Y	ReLU	(8, 8, 256)
Convolution + Upsampling	(5, 5)	(1, 1)	128	Y	ReLU	(16, 16, 128)
Convolution + Upsampling	(5, 5)	(1, 1)	64	Y	ReLU	(32, 32, 64)
Convolution	(5, 5)	(1, 1)	3	N	tanh	(32, 32, 3)

2.2.2 Discriminator

For our discriminator, we will use the following architecture :

Layer	Kernel	Strides	Features	BN	Activation	Output shape
Input						(32, 32, 3)
Convolution	(5, 5)	(1, 1)	64	N		(32, 32, 64)
Max Pooling	(2, 2)	(2, 2)	N/A	Y	ReLU	(16, 16, 64)
Convolution	(5, 5)	(1, 1)	128	N		(16, 16, 128)
Max Pooling	(2, 2)	(2, 2)	N/A	Y	ReLU	(8, 8, 128)
Convolution	(5, 5)	(1, 1)	256	N		(8, 8, 256)
Max Pooling	(2, 2)	(2, 2)	N/A	Y	ReLU	(4, 4, 256)
Convolution	(4, 4)	(1, 1)	1	N	sigmoid	(1,)

2.3 Loss functions

2.3.1 Discriminator loss

Let our real data distribution be $p_{data}(x)$ and our input noise distribution $p_z(z)$. Our generator will be $G(z, \theta_g)$, a multilayer perceptron with parameters θ_g. Our disciminator will be $D(x, \theta_d)$, a multilayer perceptron with parameters θ_d.

The discriminator will try to recognize ground truth data by predicting 1, and to recognize the generated images $G(z)$ by predicting 0. Thus, we will use the binary cross entropy of the difference between the values predicted and the expected values as our loss function.

The loss function will then be [3] :

$$L_D = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

As shown in Theorem 1 from [3], minimizing this loss function is equivalent to minimizing the Jensen Shannon Divergence between our real data distribution and our fake one. This will become important when studying the shortcomings of this loss function, and when looking at Wasserstein GANs that we will study later.

2.3.2 Generator loss

The generator's goal is to have all of its produced images guessed as True by our discriminator. A way to achieve this goal is to get the distance from 1 (value predicted for real data $x \in p_{data}$), which results in the following loss :

$$L_G = \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

However, this loss, while useful theoretically, suffers from problems, most notably gradient vanishing. When our discriminator is optimal, $D(G(z)) \to 0$, which means that $\log(1 - D(G(z)) \to 0$ and we suffer from gradient vanishing as well as saturation as we can see in figure 2.

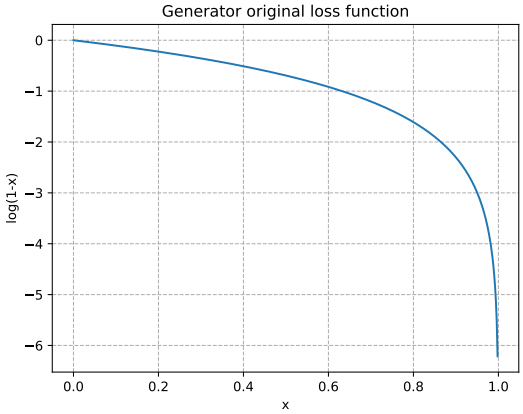

Figure 2: Original loss function

The vanishing gradient problem present using the aforementioned loss is problematic in practice since we want our generator to be able to improve when it performs poorly (at the beginning of training) which is complicated in this scenario.
Instead, we will use the following loss:

$$L_G = \mathbb{E}_{z \sim p_z(z)}[-\log(D(G(z)))]$$

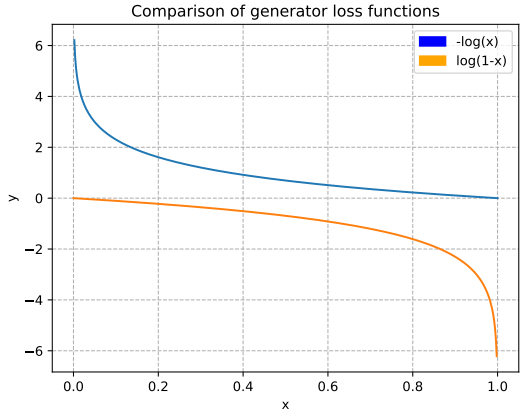

Figure 3: Comparison of the two suggested loss functions

As we can see, when using $-log(D(G(z))$ as our loss function, we will not suffer from vanishing gradients since our loss will increase as we go towards 0. As such, we will more easily create good images, even though when our loss goes towards 1 our gradient will also decrease.

The gradient vanishing problem was not entirely removed but mostly moved, and we will encounter it when our generator is nearly optimal, which is a much better scenario than before.

However, this loss is not motivated by theory but only by experiments, yet it seems more intuitive and gives better results in practice. We will use this loss in all of our experiments.

2.4 Training a GAN

Now that we have everything that is needed, we can define our training loop.

Algorithm 1 GAN training using the $-\log$ trick. We will use the adam optimizer and k=1 in all of our experiments, which is the less expensive method

for number of training iterations **do**
  for k steps **do**
    Sample a batch of m noise samples $\{z^i\}_{i=1}^{i=m}$ from $p_g(z)$
    Sample a batch of m examples $\{x^i\}_{i=1}^{i=m}$ from $p_{data}(x)$
    Update our discriminator with gradient:

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

  end for
  Sample a batch of m noise samples $\{z^i\}_{i=1}^{i=m}$ from $p_g(z)$
  Update our generator with gradient:

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

end for

In experiments, in order to be more efficient, we will reuse the last noise samples used to update the discriminator to update our geenerator.

2.5 Mode collapse

Mode collapse is the main issue that we will have to deal with while training GAN.
Mode collapse happens when the generator always generates the same results from different noise vectors, as isslustrated in 4.

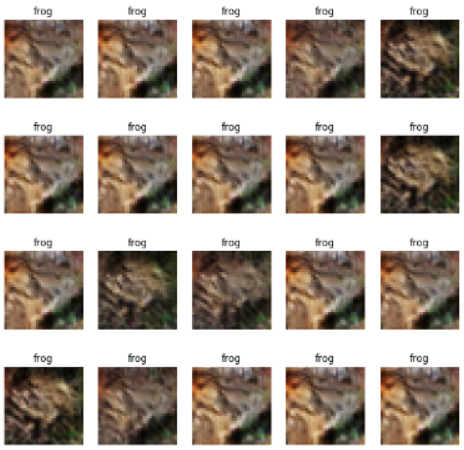

Figure 4: Mode collapse illustration on CIFAR-10

This happens because for a fixed discriminator, the optimal generator only produces the data that has the highest values assigned by our discriminator.
Indeed, when those points are found, the generator has no incentive to produce other, different data points.
A solution would be to train our discriminator to optimality, which is not feasible in practice since it can lead to vanishing gradients for our discriminator due to the Jensen Shannon divergence's properties (which we will detail when talkin about Wasserstein GANs).

This scenario is not desirable as most of the time we want to eventually generate a large variety of data. We will describe different methods to avoid it in the DCGAN improvement methods section and the WGAN section.

2.6 Balancing our generator and discriminator

2.6.1 Imbalance detection

In theory, both our generator and discriminator losses should converge after a certain time. However, in some situations, the losses don't converge. This is due to an imbalance between the discriminator and the generator : one of them is too strong compared to the other one.
In practice, this however doesn't mean that our samples won't be of good quality: they can still be with a discriminator that has a loss of 0.
On the other hand, having converged doesn't mean that the networks won't learn anymore, since as long as there are oscillations, both networks are still learning.

Still, studying those imbalances and detecting them can be useful to evaluate our training.

The first way to detect them is to plot the evolution of the losses at each epoch. This way, we can easily detect whether the losses converge or not. The discriminator's loss has to not converge to 0 too quickly, as the generator will be stuck because of gradient vanishing, and will not improve anymore. The generator's loss should increase and converge. When the generator is too strong, both losses will stay constant and it won't be able to learn any useful data representation as it fools the discriminator too easily.
Sadly, this method is not always good since lack of convergence doesn't mean that our images won't be good and convergence doesn't always mean that the images will be good.

Another way is to look at images genreated by the generator at each epoch. They are supposed to get better and better at each epoch, nevertheless if the networks are imbalanced, there will be consequences on the images: if the discriminator is too strong, the generator will be stuck and will not improve anymore, then the images will stay bad even after several epochs. If our generator is too strong, we probably won't learn any good data representation.

The best way to look as this problem is from our images and not our losses first.
If we produce bad images, our losses can point us in the right direction to investigate the issue. The most important thing is that non converging losses doesn't imply that our images will be of bad quality.

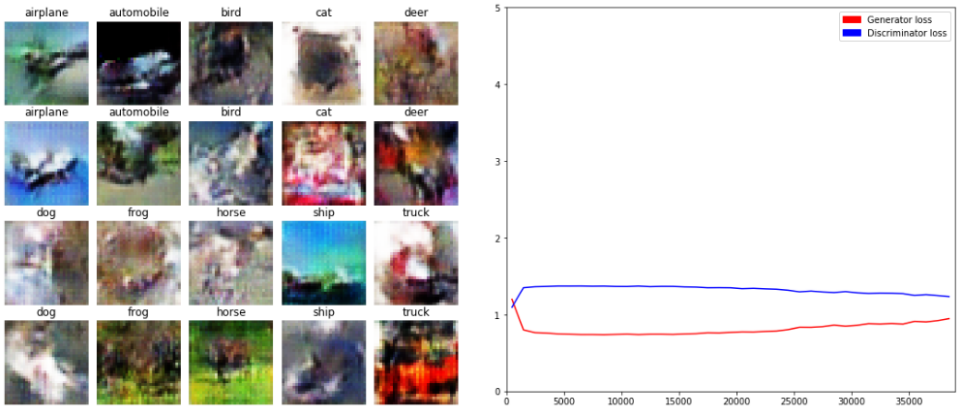

Figure 5: Early loss convergence influence on image quality

For example, in figure 5 the images look terrible. Then, by looking at the losses, we notice that the two curves converge without any oscillation. We can conclude that there is an imbalance, and that the generator might be too "strong".

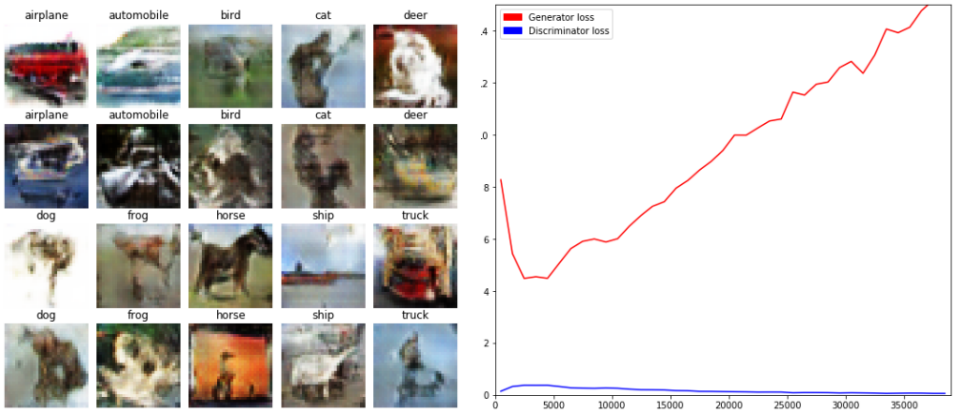

Figure 6: Diverging losses and image quality

Another example in figure 6, the images look a bit better, but are still not that great. By looking at the losses, we see that the generator's loss diverges in a linear way. We can conclude that there is an imbalance, and that the discriminator might be too "strong".

The lack of interpretability of our losses is one of the main problems with GANs as ultimately only our images can tell us if our generator performs well.
This can be resolved by using other loss functions, all of which won't always perform well or better, for example the Wasserstein distance (cf WGAN chapter).

Now that we have talked about what can happen, let's see potential ways to fix those issues.

2.6.2 Adapting the architecture

One of the most straight-forward way to solve critical imbalances is to alter our network's architecture. Our generator being too "good" can actually be that our discriminator is unable to learn properly, and increasing its capacity might help, and vice-versa.

Trying different architectures may help to solve the problem, or at the very least can help you understand better where the problem comes from.

2.6.3 Hyperparameters tweaking

Hyperparameter tweaking is fundamental in all neural networks, but with GANs some tweaks will provide really good results to specific problems.
Indeed, avoiding aforementioned problems is not only a matter of architecture but also of hyperparameters, some tweaks that are specific to the GAN architecture, and the fact that we have two neural networks instead of one.

If one of our networks is learning too "fast" compared to the other, one way to fix this would be to either slow it down or speed up the other.
This translates in different learning rates for our networks, obviously this isn't a solutions to every problem but helps on certain problems, especially when even architectural tweaking didn't give us good results.
Most of the time, we use the adam optimizer with a learning rate of 2×10^{-4}, and in some case we can use a lower of a bit bigger one on one network to get better results.

The other hyperparameter that could be interesting to change is k, however this did not give us good results so we wouldn't recommend doing it. Using $k = 1$ seems at the same time the best and most efficient way for the training.

2.6.4 Should we weaken our discriminator?

If our discriminator is too good, it would seem logical to weaken it a bit, in order to help our generator learn better. This can lead to bad results in practice because by weakening our discriminator, we will also hinder its ability to distinguish real and fake data. This means that our generator won't be able to learn properly how to generate realistic data as its measure of success depends on the discriminator's output and it will not be able to get adequate feedback, which is vital in order to learn properly.

Indeed, the real problem is not that our discriminator is too "good" (which would translates as it being optimal), but rather the consequence of such optimality that we actually want in theory.
Using the -log trick presented previously is one way to help us to avoid vanishing gradients, and other ways, that we will detail with DCGANs, exist to improve our training.

2.7 Conditioning a GAN

Most of the datasets bring different classes of images: for example, MNIST provides different labeled digits. Something that would be nice to have is to instead of trying to generate random data, try to generate a specific class of data. This is achievable by conditioning the GAN as shown in [4].

The goal is to make the generator take into account the condition (e.g. the label) before generating the image. Thus, we can instead of only inputting the noise to the generator, input both the noise and the condition, using one-hot encoding.
The main idea is to give the condition to our network as early as possible, so that it can truly take it into account.
For the discriminator, we don't need it to guess from which class the image is. Thus, we can make it analyse the image through the convolutions, and then specify it the condition at the end before making the prediction. We specify the condition by concatenating it to a fully connected layer right after the flattened result of the convolutions. Be sure to concatenate the condition to a poor neural layer, with a few neurons, so that the condition stays visible and is taken into account. Else, we will face what is

called class leakage, which means that the condition will not be taken into account.

For our generator, we will concatenate our noise vector and our labels before feeding them to our generator. It is worth noting that sometimes, it can help to pass our noise vector through a dense layer before concatenation.

3 DCGAN

DCGAN [5] stands for Deep Convolutional Generative Adversarial Network. The only difference with regular GANs is the use of a specific architecture of the generator and the discriminator. There are no differences in the training loop.

3.1 Architectural differences with GAN

DCGAN are a type of GAN that uses convolutional layers, and as such, can only be used where convolutions make sense to be used. In our case, it will be used to generate images. The architecture of DCGAN complies with a few guidelines. According to [5], the guidelines are the following.

3.1.1 Strided convolutions

Pooling layers, such as max pooling, are determinists. It means there is no randomness: for a given entry, we always have the same output. This is why it can be better to use strided convolutions in order to learn how to downsample our image. This is obviously more costly since it adds weights to train, but helps to improve our results.

3.1.2 Transpose convolutions

The reasoning for using transpose convolutions [6] (also called deconvolutions of fractionally strided convolutions) is the same as with strided convolutions : instead of pooling, we can learn how to upsample instead of using a deterministic upsampling method (using bilinear interpolation for example).

3.1.3 Batch normalization

It is known that normalizing the processed data avoids scale problems, thus making the training easier. Batch normalization [7] uses data from our whole batch to normalize our inputs and even if reasons for it success are unclear, evidence shows that is performs really well.
This is done by calculating two normalization parameters : the mean, and the variance, on each batch. Those two parameters will then be used to reparametrize the feature of this batch. We will simply subtract the feature by the mean, and divide it by the standard deviation.
We do not use batch normalization on the output layer, because this can produce instabilities in the model.

3.1.4 No fully connected layers

Fully connected or dense layers are only used at the beginning of the generator in order to process noise and reshape it in a more useful shape for the upscaling.

Dense layers can be used in the discriminator if we want our DCGAN to be conditional, we will see in a bit how exactly.

ReLU works well in practice, but we still got results of similar quality using leaky ReLU.
Other than those examples we will only use convolutional layers in our models.

3.1.5 Leaky ReLU in the discriminator

Leaky ReLU activation function allows us to avoid the “dying ReLU” problem, as illustrated in figure 7. This problem happens when ReLU receives values under zero. Because the derivative of ReLU is equal to zero for the negative part, the learning process can be completely stuck. Leaky ReLU has a very small slope in negative value, so we can avoid this problem.

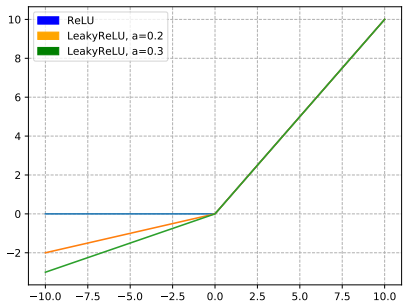

Figure 7: ReLU and Leaky ReLU comparison

3.1.6 ReLU for the generator

In the generator, we will use ReLU for all layers except our output which will use tanh to bound our pixel values in the interval $[-1, 1]$.

ReLU works well in practice, but we still got results of similar quality using leaky ReLU.

3.1.7 Generator and discriminator architecture

Here is the architecture that we will use in all our experiments with DCGANs and on which our future improvements will be based.
The architecture for our generator will be:

Layer	Kernel	Strides	Features	BN	Activation	Output shape
Input z						(100)
Dense, Reshape	N/A	N/A	$256 * 4 * 4$	Y	ReLU	(4, 4, 256)
Transposed Convolution	(5, 5)	(2, 2)	256	Y	ReLU	(8, 8, 256)
Transposed Convolution	(5, 5)	(2, 2)	128	Y	ReLU	(16, 16, 128)
Transposed Convolution	(5, 5)	(2, 2)	64	Y	ReLU	(32, 32, 64)
Transposed Convolution	(5, 5)	(2, 2)	3	N	tanh	(32, 32, 3)

As we can see this is really similar to what we used before, however at the time when GANs were developed, Batch Normalization did not exist yet.

For our discriminator, we will use this architecture :

Layer	Kernel	Strides	Features	BN	Activation	Output shape
Input z						(32, 32, 3)
Convolution	(5, 5)	(2, 2)	64	Y	LeakyReLU($\alpha = 0.2$)	(16, 16, 64)
Convolution	(5, 5)	(2, 2)	128	Y	LeakyReLU($\alpha = 0.2$)	(8, 8, 128)
Convolution	(5, 5)	(2, 2)	256	Y	LeakyReLU($\alpha = 0.2$)	(4, 4, 256)
Convolution	(4, 4)	(1, 1)	1	N	sigmoid	(1,)

Again, this is really similar as the previously used architecture, but the results will be significantly better.

3.2 Kernel size influence on image quality

By changing the convolutions' kernel size in both our generator and our discriminator we will alter training time and image quality.

By using a large kernel size we will process a bigger neighbourhood both when creating and processing the image, which should help both our networks learn more global patterns.
On the other hand, using a small kernel size, we will not be as aware of our surroundings, which should result in a quality loss, however the training time will be reduced.

The question is then, how much does this affects our image quality in practice?
We trained all networks for 100 epochs on the CIFAR-10 dataset using the architectures previously mentioned.

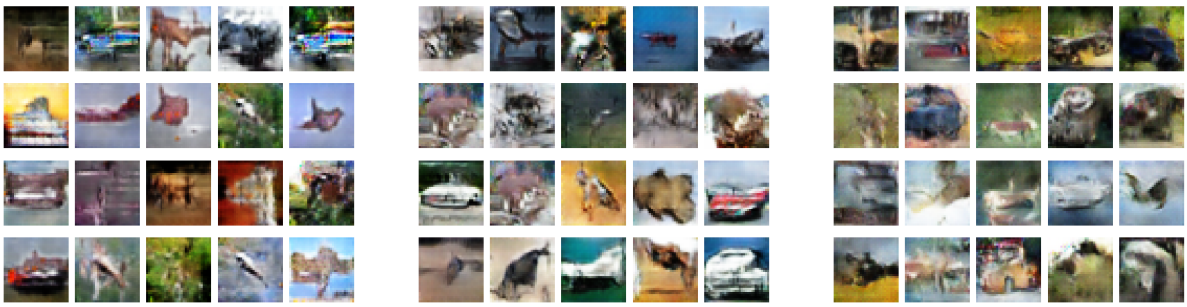

Figure 8: Comparison of image quality using a kernel size of 3x3 (left), 4x4 (middle), 5x5 (right)

In figure 8 we can see a clear improvement already from a kernel size of 3 to a kernel size of 4: there are no repeated image, and there are no apparent grids in the images anymore.
A kernel size of 5 yielded even better results, with images that look more realistic and a bit sharper.
Most of the time, a kernel size of 5 is a good compromise, and going above 7 is too much. The time increase is quadratic as the kernel is 2 dimensional.

Using a Tesla T4 we got the following training time:

- 3x3 took 30 minutes for 100 epochs
- 4x4 took 37 minutes for 100 epochs
- 5x5 took 50 minutes for 100 epochs

3.3 Conditioning a DCGAN

3.3.1 Feeding the condition

In order to condition a DCGAN, we will do it the same way as with a more traditional GAN: we will concatenate our label to our noise in the generator, and we will concatenate our label to a dense layer in our discriminator (usually the first dense layer).
This will result in the following example architecture with our one hot encoded condition c and $n = 10$ classes.
Our generator will be :

Layer	Kernel	Strides	Features	BN	Activation	Output shape
Input z						(100)
Condition c						(10)
Concatenate z with c						(110)
Dense, Reshape	N/A	N/A	$256 * 4 * 4$	Y	ReLU	(4, 4, 256)
Transposed Convolution	(5, 5)	(2, 2)	256	Y	ReLU	(8, 8, 256)
Transposed Convolution	(5, 5)	(2, 2)	128	Y	ReLU	(16, 16, 128)
Transposed Convolution	(5, 5)	(2, 2)	64	Y	ReLU	(32, 32, 64)
Transposed Convolution	(5, 5)	(2, 2)	3	N	tanh	(32, 32, 3)

As we can see this is really similar to what we used before, however at the time when GANs were developed, Batch Normalization did not exist yet.

For our discriminator, we will use this architecture :

Layer	Kernel	Strides	Features	BN	Activation	Output shape
Input z						(32, 32, 3)
Convolution	(5, 5)	(2, 2)	64	Y	LeakyReLU($\alpha = 0.2$)	(16, 16, 64)
Convolution	(5, 5)	(2, 2)	128	Y	LeakyReLU($\alpha = 0.2$)	(8, 8, 128)
Convolution	(5, 5)	(2, 2)	256	Y	LeakyReLU($\alpha = 0.2$)	(4, 4, 256)
Condition c						(10)
Concatenate with c						256*4*4 + 10
Dense	N/A	N/A	256	N	LeakyReLU($\alpha = 0.2$)	(256,)
Dense	N/A	N/A	1	N	sigmoid	(1,)

3.3.2 Influence on training and image quality

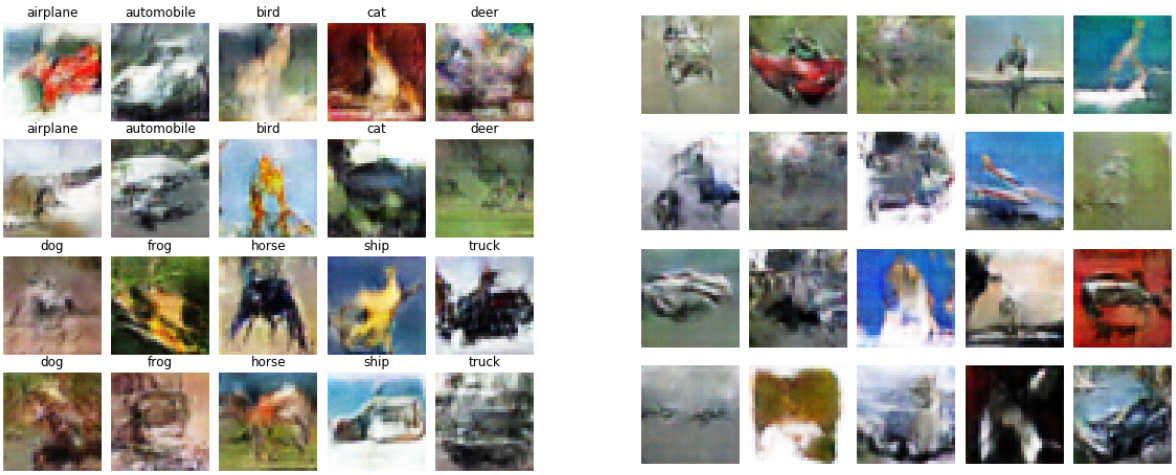

Figure 9: Results using a conditional DCGAN (left) and a non conditional DCGAN (right)

As we can see in figure 9 the condition is taken into consideration by our GAN, even though we never explicitly told our generator to classify our image. The images also look better when we specify a condition. This may be due to the fact that when learning with a condition, the network is able to learn features specific to the class and not to others, reducing the mixing of classes that could create images that are realistic from the discriminator point of view but not when comparing to real images when using

a non conditional DCGAN.

Training time is roughly the same, especially if we included the dense layers from our conditional DCGAN in our non conditional DCGAN.

4 Improving GAN training

As we have seen, GANs suffer from problems such as mode collapse, but methods to improve them have been presented in [8].
We will implement methods present in it and other methods, comparing to our baseline which is the conditional DCGAN we studied previously.

4.1 Label smoothing

When we calculated the discriminator's loss, we calculated the difference between 1 and the prediction for real images. However, it was shown in [8] that smoothing our truth value to $\alpha = 0.9$ improved the results, by preventing the discriminator to be over confident.
However, we should only use it on our "true" label. Using label smoothing on our "false" data value means comparing the difference between the prediction and 0.1 instead of 0. However, this causes a problem, if we use β instead of 0 the optimal discriminator becomes :

$$D(x) = \frac{\alpha p_{data}(x) + \beta p_{model}(x)}{p_{data}(x) + p_{model}(x)}$$

The problem is that p_{model} appears in the numerator, and in situation where p_{data} is low and p_{model} is high (which means that the difference is high, thus we want the discriminator to return a low value), the output of the discriminator will be close to β, and higher than what we expect. This will not incentivize p_{model} to move closer to p_{data}, therefore, we will only smooth our truth label. We can see the results after 200 epochs in figure 10.

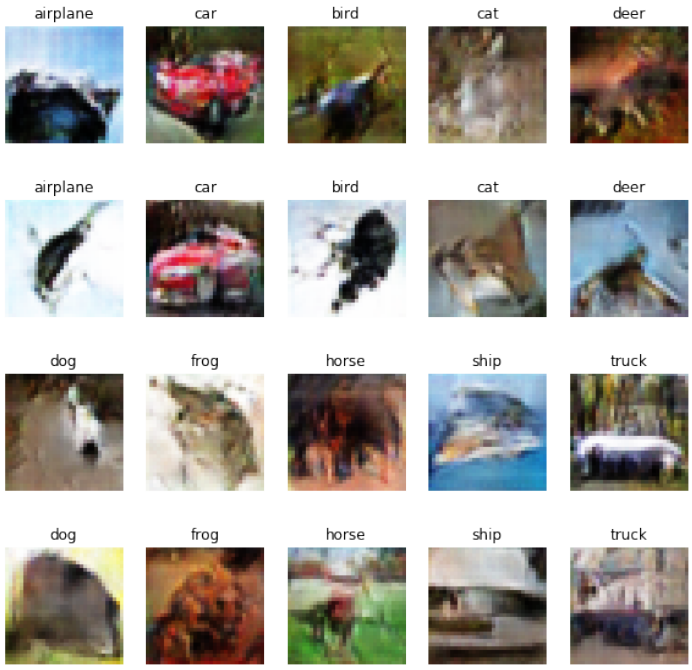

Figure 10: Results of one sided label smoothing on CIFAR-10

We can see that the images already look better than previously.
Furthermore this is really easy to implement in practice, we just need to multiply our truth value when

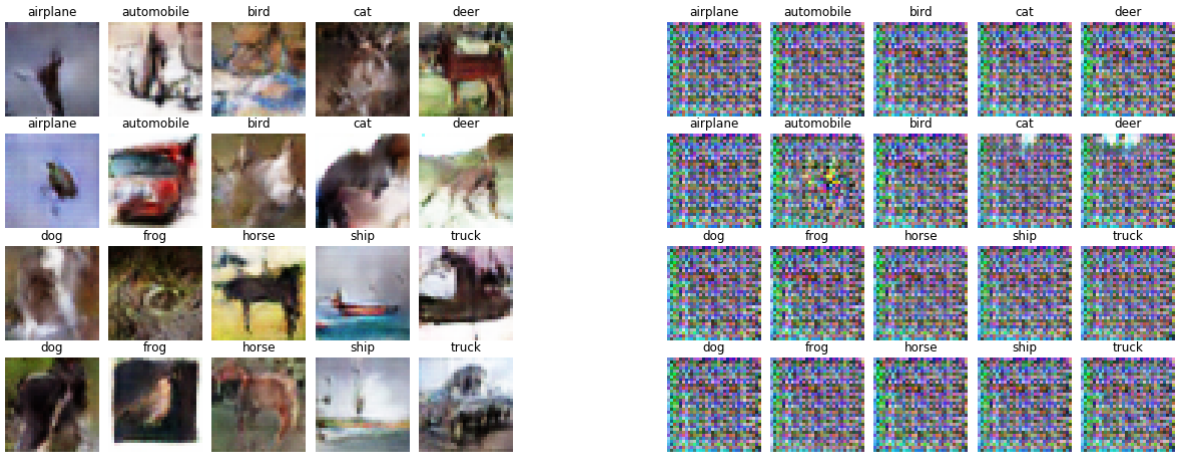

Figure 11: Two sided label smoothing influence over time, at epoch 60 (left) and 61 (right)

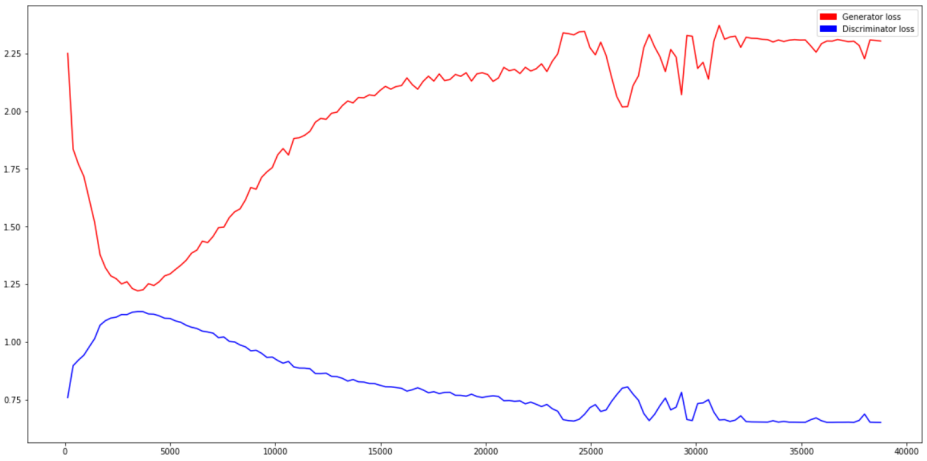

Figure 12: Loss function over time with two sided label smoothing

calculating our loss (α instead of one).

However, even if in theory two-sided labelsmoothing should cause problems, doest it really in practice? We tried with different parameters, and the results were similar to one-sided label smoothings during the first epochs, but the results became suddenly awful after a certain epoch (roughly 60 epochs in our case).
We can see results at epochs 60 and 61 using the same architecture as before in figure 11.
As we can see, we encountered the problem that we mentioned earlier, and it was present in all of our tests, happening around epoch 70.

This can be clearly seen in figure 12 where we can see our losses start oscillating way more than usual at the same time that our images started to look like noise.
We can see them stabilize afterwards but we never got images that looked like they did before this phenomenum appeared. We could argue that even if it came back to normal afterwards this still makes two sided label smoothing unusable as it would still waste a lot of time with no real guarantee of success.

4.2 Experience replay

Experience replay is inspired by a biological mechanism and is usually used in Q-learning. The principle is simple : show previously generated images to the generator to make sure that they are rightfully classified.
This could intuitively be useful to alleviate mode collapse since this would help tell the discriminator that previously generated images were fake and if we collapsed, this could help us get out of the collapse.

To implement experience replay, we maintain a list of generated images by picking randomly an image for each batch. Here we introduce an integer parameter *exp_replay*. When the length of the list is equal to *exp_replay* we make a discriminator training step using these images.

After many experimentations, we realized that the *exp_replay* value must be carefully selected. Indeed if *exp_replay* is equal to the batch size, we can encounter some problems:

- If the batch size is small we will process a lot of experience replay step in an epoch. As a result, this lead to mode collapse earlier than if there was no replay experience.
- If the batch size is high, the frequency of experience replay is too low, and the experience replay does not have a strong enough impact to make it useful.

In conclusion, the best way to do this is to calculate *exp_replay* by using both our batch size and the size of our dataset, so that it can be as representative of the dataset as possible while still controlling the discriminator's updates frequency.

4.3 Minibatch discrimination

Minibatch discrimination is a way to reduce mode collapse, and to create more diversity in the images created by the generator. It was first described in [8]
The idea is pretty simple : when there is mode collapse, almost all the images are the same, so the difference between the images is very low. So we can detect if there is mode collapse by determining the difference in a minibatch of generated images, and comparing it to the difference in a minibatch of real images. So we can implement this idea by adding a new layer in the discriminator. The role of this layer is to calculate the difference in a batch of images generated by the generator or in a batch of real images. We use as input $f(x_i) \in \mathbb{R}^A$, the output from a previous layer. We then multiply it with a tensor $T \in \mathbb{R}^{A \times B \times C}$, which gives us a matrix $M_i \in \mathbb{R}^{B \times C}$.
We then take the L_1-distance between rows of our matrix for every sample i.
We then compute the activation as follows:

$$o(x_i)_b = \sum_{j=1}^{n} \exp\left(-|M_{i,b} - M_{j,b}|\right)$$
$$o(x_i) = [o(x_i)_1, o(x_i)_2, \ldots, o(x_i)_B]$$
$$o(X) \in \mathbb{R}^{n \times B}$$

With n our number of samples in a batch.
We can finally concatenate this result to the initial features.
If our previous layer produces an output in $\mathbb{R}^{n \times A}$ after the minibatch discrimination layer we will have features in $\mathbb{R}^{n \times (A+B)}$.

We can then add this layer at the end of the discriminator. To see the best way of using the layer, we tried to put a dense layer before, then a dense layer after, and finally a dense layer before and after. Here are the results with 100 epochs, a batch size of 128, and a dense layer before. The architecture used for our discriminator is as follows:

Layer	Kernel	Strides	Features	BN	Activation	Output shape
Input z						(32, 32, 3)
Convolution	(5, 5)	(2, 2)	64	Y	LeakyReLU($\alpha = 0.2$)	(16, 16, 64)
Convolution	(5, 5)	(2, 2)	128	Y	LeakyReLU($\alpha = 0.2$)	(8, 8, 128)
Convolution	(5, 5)	(2, 2)	256	Y	LeakyReLU($\alpha = 0.2$)	(4, 4, 256)
Condition c						(10)
Concatenate with c						256*4*4 + 10
Dense(optional)	N/A	N/A	256	N	LeakyReLU($\alpha = 0.2$)	(256,)
Minibatch Discrimination	5	N/A	128	N	N/A	(256 + 128,)
Dense(optional)	N/A	N/A	256	N	LeakyReLU($\alpha = 0.2$)	(256,)
Dense	N/A	N/A	1	N	sigmoid	(1,)

Here 5 corresponds to our dimension C in our tensor and 128 corresponds to B.

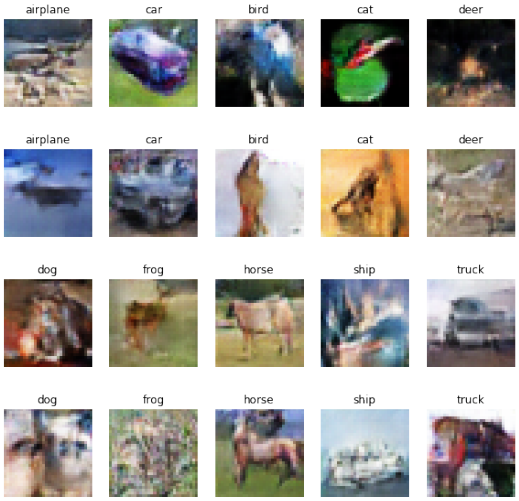

Figure 13: Minibatch discrimination using only the dense layer right before

As we can see in figures 13,14,15, we got the best results when using both dense layers (before and after our minibatch discrimination layer), with using only one, be it before or after, giving worse results but not with a dramatic difference.

When using minibatch discrimination, choosing the numbers of features you want to add is important to control the influence of diversity, for a dense layer of 256 we got the best results with 128 features. This will depend on the dataset and is another hyperparameter that will have to be tweaked.

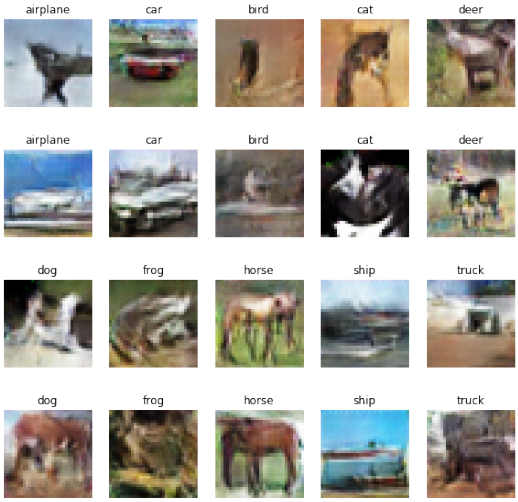

Figure 14: Minibatch discrimination using only the dense layer right after

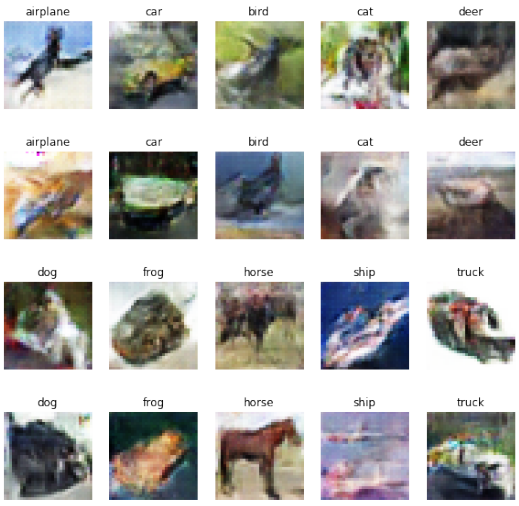

Figure 15: Minibatch discrimination using both dense layers

4.4 Virtual batch normalization

We have seen that batch normalization is a great way to improve the optimization of a model, though, it can have issues that are highlighted in [9].
The main one is that each batch of data can be very different from another batch, causing a large fluctuation of the batch normalization parameters. This is especially true when using small batches, which is very common, because it uses less memory. This fluctuation can lead to a decrease in image quality when those fluctuations are too important.
A way to solve this problem is to use a technique called reference batch normalization. With this technique, we use a reference batch, that is sampled at the beginning of the training, and is never changed throughout the training. We then train the discriminator on both the reference batch, and the current batch. We then use the μ and σ parameters of the reference batch, and apply them on both batches. This method also has limits, since it is possible for the model to overfit the reference batch.
Virtual batch normalization is an improvement of the aforementioned method. We still use a reference batch, but we calculate the parameters using the union of the reference batch and the current batch. The main issue caused by this method is that it is computationally expensive since two batches instead of one have to go through the network.

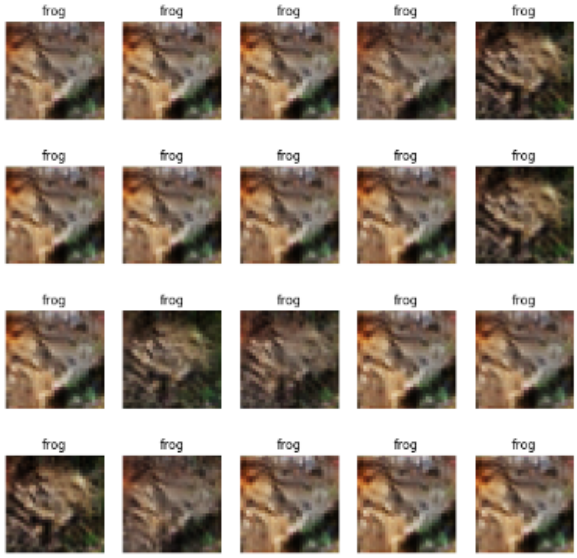

Figure 16: Consequence of using different classes for our reference batch and current batch using virtual batch normalization

If we are using a conditional GAN and use random labels in our reference batch, we will encounter problems regarding the classes of our output images as we can see in figure 16 where classes are completely ignored.
Since we don't have the same labels in our reference batch used to measure μ and σ as in our real batch, our normalization won't be working properly. This can easily be seen if our classes are "dogs" and "cars", we obviously have very different data in both classes and if we try to normalize dog images using car images as reference, the result will obviously not be right.

By using the current labels with the same noise we were able to get the results in figure 17. As we can see, we got much better results using virtual batch normalization than by using batch normalization. However in our experiments with VBN we encountered saturation in some genreated batches of images. We believe that it comes from implementation details more so than from problems with VBN in itself.

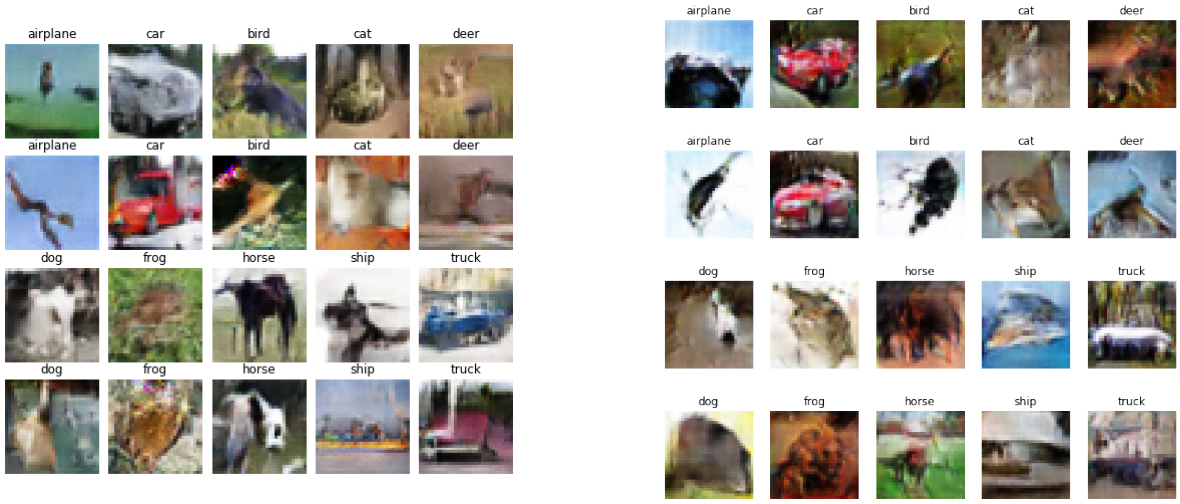

Figure 17: Comparison of output images using virtual batch normalization (left) and batch normalization (right)

4.5 Noisy labels

The idea behind noisy labels is to flip the labels for a small percentage (about 5%) of images. We change labels so that some real images are considered fake and some generated images are considered real.
This should help avoid overconfidence from our discriminator by forcing it to learn even when we could say that it has converged.

We did not find this method backed up by experiments but it was mentioned in some articles online so we decided to implement it to see if can be useful.
To do so, we tried some noisy labels rate over 100 epochs with a classic conditional DCGAN.

At first, we tested the suggested flipping rate of 5%. As we can see in figure 18 results were pretty bad, we got the same problem as we got on the two-sided label smoothing ; at a certain epoch, the generated images turned into noise.
We then tested with a rate of 10% and got the same results, as illustrated by figure 19.
The reason why this failed is that flipping labels is similar to forcing the erroneous scenarios caused by two sided label smoothing.

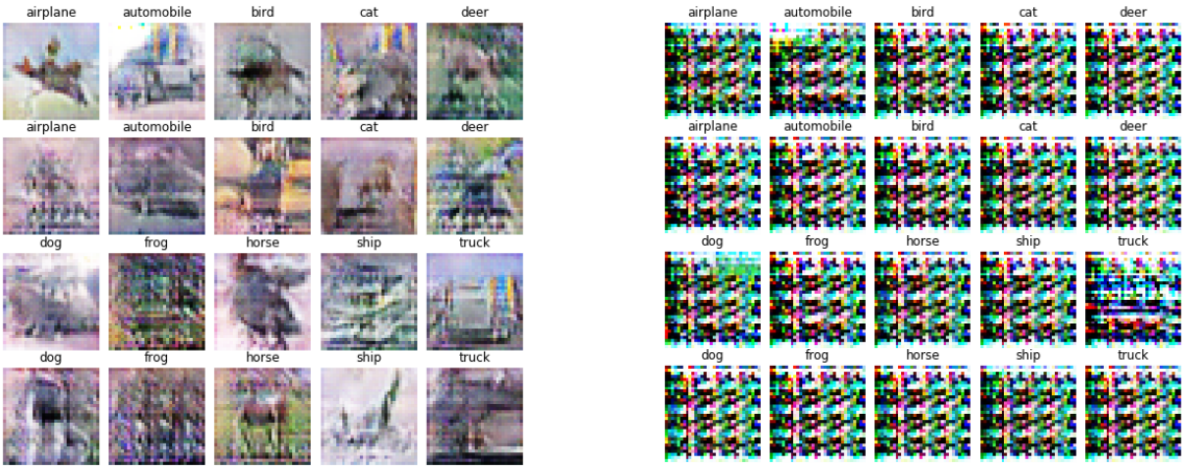

Figure 18: Noisy labels influence on image quality, at epoch 78 (left) and 79 (right)

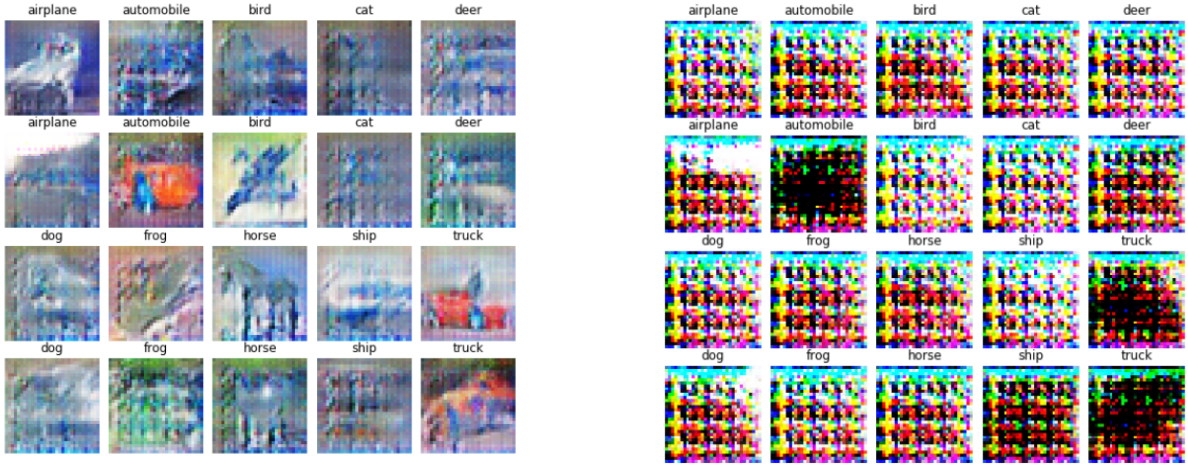

Figure 19: Two sided label smoothing influence over time, at epoch 75 (left) and 75 (right)

5 Wasserstein GAN

Wasserstein GANs [10] are a type of GAN that aims to solve most of the traditional GAN's problems with a substantial theoretical evidence to support it. One of the most important features of WGANs is the absence of mode collapse, which we will detail in a bit.

5.1 Differences with a traditional GAN

We know that the GAN framework is able to optimize problems using all of f-divergences but the most used is the Jensen-Shannon divergence, as used in [3].

Instead, WGANs will use the Earth-Mover distance (or Wasserstein 1) that is defined as follows:

$$W(p_{data}, p_g) = \inf_{\gamma \in \Pi(p_{data}, p_g)} \mathbb{E}_{(x,y)\sim\gamma}[||x - y||]$$

With p_{data} our real data distribution, p_g our generated data distribution and $\Pi(p_{data}, p_g)$ the set of all joint distributions $\gamma(x, y)$ with marginals p_{data} and p_g.
If our distributions were piles of dirt, $\gamma(x, y)$ would measure how much mass has to be moved from x to y to tranform p_{data} into p_g.
The EM distance would then be the cost of the optimal way to do this transformation.

Sadly it is impossible to compute, but using the Kantorovich-Rubinstein duality it can be rewritten as (refer to [10] for more details and calculations) :

$$W(p_{data}, p_g) = \sup_{||f||_L \leq 1} \mathbb{E}_{x\sim p_{data}}[f(x)] - \mathbb{E}_{y\sim p_g}[f(y)]$$

With the supremum being calculated over all 1-Lipschitz functions.

This is still very hard to compute, as we can't iterate over all 1-Lipschitz functions, but is much more easier. In order to use this equation we must make sure our function in Lipschitz continuous, our function being a neural network in the case of WGANs. There are two ways to do this that we will detail a bit further on : weight clipping and gradient penalty. The training loop is slightly different for both methods but stays the same fundamentally.

5.1.1 Loss function

A legitimate question to ask is whether or not the Wasserstein distance is better than the JS-Divergence that is more commonly used.

First of all, we need to understand what makes JS-Divergence a poor fit for our problem. The main pitfalls of it being mode collapse and lack of continuity.

Mode collapse is mainly due to our impossibility to train the discriminator to optimality, since it would result in gradient vanishing or unstable training [11].
Since we cannot train an optimal discriminator, we are going to suffer from mode collapse because we will wrongly estimate the JS-Divergence, resulting in bad results fooling the discriminator. We have seen a lot of good methods to alleviate mode collapse but none is perfect.

Also, as stated in [10], the JSD is not derivable everywhere nor is it continuous everywhere. This makes it harder to train our model since it cannot always converge easily to the solution. And when we add

our vanishing gradient problem to this, it is really hard even in toy examples to obtain good results.

As such WGANs aims to solve those problems by giving a loss function that is continuous everywhere and derivable almost everywhere, which will enable us to train our critic to optimality. The losses for our discriminator (called a critic) and generator are as follows:

$$L_D(p_{data}, p_z) = \mathbb{E}_{x \sim p_{data}}[D(x)] - \mathbb{E}_{z \sim p_z}[D(G(z))]$$
$$L_G(p_z) = -\mathbb{E}_{z \sim p_z}[D(G(z))]$$

With p_z our noise distribution, D our critic and G our generator.

The difficulty here is making sure our critic is Lipschitz-continuous, for which methods have been developed such as weight clipping, and gradient penalty (which is the better of the two).

The critic loss (representing our Wasserstein distance) also has the benefit that it is a good indicator of image quality and as such training success/failure will not rely solely on us looking at images or using the inception score but we will also be able to use this loss as a good indicator of success or failure.

5.1.2 Architecture

Some architectural changes are necessary in order to use the Wasserstein loss, the most notable one being that we have to remove Batch Normalization because it interferes with the gradient penalty.

Instead, we will use layer normalization [12] which will still provide us normalization without causing any issues.

5.2 Enforcing lipschitz continuity

As we have seen before, we must have our critic be Lipschitz continuous, let's look at both ways to ensure that it is the case.

5.2.1 Weight clipping

The first way to enforce the Lipschitz constraint is to constraint weights to a certain interval, usually [-0.01, 0.01].
This was already pointed out as a bad way to enforce the Lipschitz constraint in [10], but it was working and still helped to prove that WGANs work in practice.

However, a consequence of this clipping is pointed out in [13], the weights will accumulate on both extrema of the clipping interval and almost no weights will take intermediate value.
An other problem is shown on the Swiss Roll dataset, which is that weight clipping can either lead to gradient vanishing or explosion, and requires fine tuning of the clipping threshold.

Even though these problems won't always happen in practice they still can happen, which is certainly a problem.

Due to these potential issues, weight clipping is not a recommended way to enforce the constraint and it is instead recommended to use Gradient Penalty which gives much better results.

5.2.2 Gradient penalty

Gradient penalty enforces Lipschitz continuity by penalising the critic if its gradient is far from 1, which is the maximum value expected for 1-Lipschitz continuity.

The loss function is then :

$$L_C(p_{data}, p_z) = \mathbb{E}_{x\sim p_{data}}[D(x)] - \mathbb{E}_{z\sim p_z}[D(G(z))] + \lambda \mathbb{E}_{\hat{x}\sim p_{\hat{x}}}[(||\nabla_{\hat{x}} D(\hat{x})||_2 - 1)^2]$$

The lambda factor is used to weigh the penalty so that is doesn't overpower our initial loss or so that is does not become too weak to have an impact. Using $\lambda = 10$ yields good results in practice.

When calculating the gradient penalty, we will use the predictions over $\hat{x}$, which is a weighted average between real and fake samples.

5.3 Training a WGAN

5.3.1 Algorithm

The algorithm used to train a WGAN with gradient penalty is as follows :

Algorithm 2 WGAN training using gradient penalty. We will use the adam optimizer with $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$ and use $k = 5$

for number of training iterations **do**
  for k steps **do**
    Sample a batch of m noise samples $\{z^i\}_{i=1}^{i=m}$ from $p_g(z)$
    Sample a batch of m examples $\{x^i\}_{i=1}^{i=m}$ from $p_{data}(x)$
    Sample a batch of m weights $\{\epsilon^i\}_{i=1}^{i=m}$ from $U[0, 1]$
    $\forall i \in [1, m], \quad \hat{x}^i = \epsilon^i x^i + (1 - \epsilon^i) G(z^i)$
    Update our discriminator with gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} D(x^i) - D(G(z^i)) + \lambda[(||\nabla_{\hat{x}^i} D(\hat{x}^i)||_2 - 1)^2]$$

  end for
  Sample a batch of m noise samples $\{z^i\}_{i=1}^{i=m}$ from $p_g(z)$
  Update our generator with gradient:

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

end for

If you wanted to adapt this algorithm to use weight clipping, just ignore the $\hat{x}$ computation and the gradient penalty. You will also need to clip all your network weights to an interval [-c, c].

Overall this is quite similar to a traditional GAN, only the loss function changed and k is now 5 instead of 1 (in most scenarios).

Since we want our critic to be optimal at any given time, we will adapt k over time, as follows:

- for the first 100 iterations, $k = 100$
- every 500th iterations, $k = 100$

This will help our critic improve at the beginning of training and make sure it never gets too far away from optimality.

5.3.2 Stability of training and architecture importance

A nice feature of WGANs is that they are really robust and way less architecture dependent than GANs. In a GAN if you do not tweak your architectures and hyperparameters carefully you won't get any good results (especially regarding the generator-discriminator "balance"), however on a WGAN you will still get results, although convergence might be slower.

This does not mean that a generic architecture will always produce good results given enough time, but that the tweaking that you will have to do will not need to be as crucial as it is on traditional GANs.

WGANs remain neural nets and tweaking hyperparameters and the architecture will still be something that will have to be done to achieve good results.

5.3.3 Interpretability of the critics's loss function

One of the best aspects of using the Wasserstein distance as a loss function is that it is a good indicator of training success, and even correlates with image quality.
In traditional GANs, since the generator and discriminator are "fighting" against each other, their losses (ideally) converge to a constant value, oscillating around it as training continues.
The value is not an indicator of success and neither is the convergence, even if the loss stagnates learning can still occur.

On the other hand, if we look at the Wasserstein distance, which will be our loss minus the gradient penalty, we can see it converge towards zero as training progresses and gives us an insight to our image quality, to a certain degree.

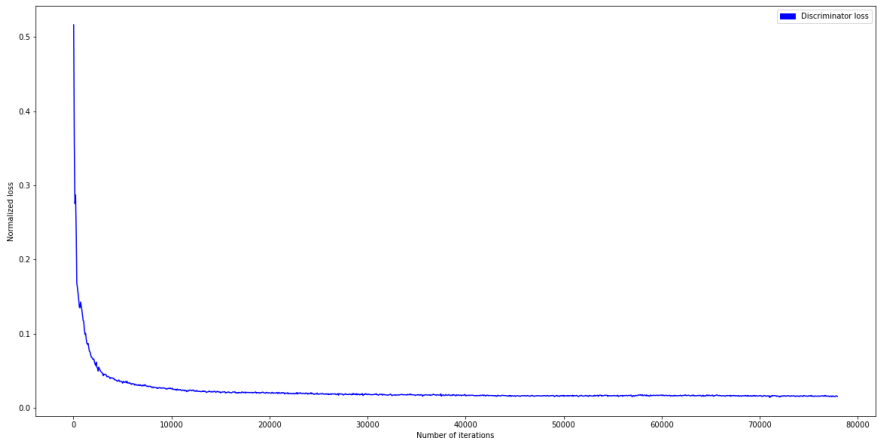

Figure 20: Critic's loss when training on CIFAR-10 dataset

In figure 20, we trained a conditional WGAN on CIFAR-10 for 500 epochs, and as we can see the loss decreases steadily over time, converging towards 0. In most case, the progression won't be as smooth as it is here, but you will still be able to draw the same conclusions overall : if the loss goes down, the

image quality improves.

From what we have seen in our experiments, once convergence is reached, or nearly reached, the image quality will not improve as much as it can with a traditional GAN that converged. Let's look at an example to illustrate.

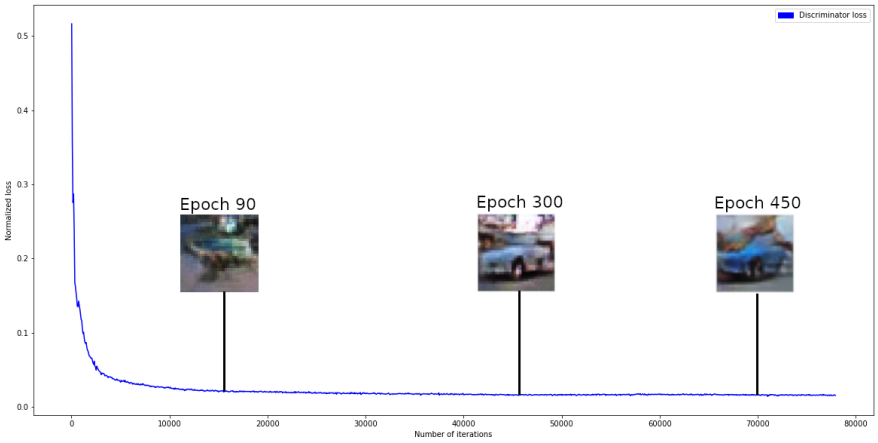

Figure 21: Inage quality improvement with a WGAN on CIFAR-10 dataset

As we can see in figure 21, from epoch 90 to 300, even if the loss doesn't decrease as much as it did in the beginning, image quality increased drastically, however afterwards, from 300 to 450 it didn't really increase, we could even argue that it got worse.
All images were generated using the same noise vector and the same label "car" with a WGAN trained on CIFAR-10.

5.4 Absence of mode collapse

Since we are training the critic to optimality at each step, mode collapse is absent in theory. However, since in practice we will never be able to train to optimality for obvious reasons, mode collapse may still appear on certain datasets, but by increasing k it should disappear. It is worth noting that in practice, mode collapse almost never happens.

Since every 500 iterations we train the critic for 100 steps instead of 5, this should also help alleviate potential mode collapse by making sure the critic stays always as close to optimality as possible.

5.5 Conditioning a WGAN

The way we used previously to condition a GAN (concatenating labels) was ineffective in our experiments with WGANs, the class ended up being ignored.
This problem was present even on the MNIST dataset, which should be one the most simple cases.

In order to specify the condition, we will instead concatenate labels as image "channels" at the beginning of the critic.

If we have an $n \times n$ image with c channels, and our label is encoded in one hot of length m, we will then have our input be of dimension $n \times n \times (c + m)$.

In order to do so, we have to create m $n \times n$ arrays filled either only with zeros or only with ones.

For example, if our label was $[0, 0, 1, 0]$ and our image is of size $2 \times 2 \times c$, we will have :

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

that will be concatenated to our input image.

This has proven to be effective in practice, and also allows us to only use convolutions in our critic instead of using fully connected layers in the end in order to feed our label.

5.6 Training time and results comparison with a DCGAN

We have seen that DCGANs can suffer from mode collapse and that it is a hard problem to solve, and since WGANs do not suffer from it, they can be a really good alternative.

However, the fact that we train our critic to optimality on every step makes WGANs more computationally expensive. As such, we will compare training time and results quality using a DCGAN(with one sided label smoothing and minibatch discrimination) and a WGAN with an architecture and hyperparameters as close as possible to each other.

For our comparison we sill use the following architecture for our DCGAN:

For our discriminator:

Layer	Kernel	Strides	Features	BN	Activation	Output shape
Input z						(32, 32, 3)
Convolution	(5, 5)	(2, 2)	64	Y	LeakyReLU($\alpha = 0.2$)	(16, 16, 64)
Convolution	(5, 5)	(2, 2)	128	Y	LeakyReLU($\alpha = 0.2$)	(8, 8, 128)
Convolution	(5, 5)	(2, 2)	256	Y	LeakyReLU($\alpha = 0.2$)	(4, 4, 256)
Condition c						(10)
Concatenate with c						256*4*4 + 10
Dense	N/A	N/A	256	N	LeakyReLU($\alpha = 0.2$)	(256,)
Minibatch Discrimination	5	N/A	128	N	N/A	(256 + 128,)
Dense	N/A	N/A	256	N	LeakyReLU($\alpha = 0.2$)	(256,)
Dense	N/A	N/A	1	N	sigmoid	(1,)

For our generator:

Layer	Kernel	Strides	Features	BN	Activation	Output shape
Input z						(100)
Condition c						(10)
Concatenate z with c						(110)
Dense, Reshape	N/A	N/A	$256 * 4 * 4$	Y	ReLU	(4, 4, 256)
Transposed Convolution	(5, 5)	(2, 2)	256	Y	ReLU	(8, 8, 256)
Transposed Convolution	(5, 5)	(2, 2)	128	Y	ReLU	(16, 16, 128)
Transposed Convolution	(5, 5)	(2, 2)	64	Y	ReLU	(32, 32, 64)
Transposed Convolution	(5, 5)	(2, 2)	3	N	tanh	(32, 32, 3)

And for our WGAN, we will use :

For our critic (LN being Layer Normalization):

Layer	Kernel	Strides	Features	LN	Activation	Output shape
Input z						(32, 32, 3)
Condition c						(32,32,10)
Concatenate z with c						(32,32,10+3)
Convolution	(5, 5)	(2, 2)	64	Y	LeakyReLU($\alpha = 0.2$)	(16, 16, 64)
Convolution	(5, 5)	(2, 2)	128	Y	LeakyReLU($\alpha = 0.2$)	(8, 8, 128)
Convolution	(5, 5)	(2, 2)	256	Y	LeakyReLU($\alpha = 0.2$)	(4, 4, 256)
Convolution	(4, 4)	(1, 1)	1	N	sigmoid	(4, 4, 256)

For our generator:

Layer	Kernel	Strides	Features	BN	Activation	Output shape
Input z						(100)
Condition c						(10)
Concatenate z with c						(110)
Dense, Reshape	N/A	N/A	$256 * 4 * 4$	Y	ReLU	(4, 4, 256)
Transposed Convolution	(5, 5)	(2, 2)	256	Y	ReLU	(8, 8, 256)
Transposed Convolution	(5, 5)	(2, 2)	128	Y	ReLU	(16, 16, 128)
Transposed Convolution	(5, 5)	(2, 2)	64	Y	ReLU	(32, 32, 64)
Transposed Convolution	(5, 5)	(2, 2)	3	N	tanh	(32, 32, 3)

Figure 22: Comparison between DCGAN (left) and WGAN (right)

We trained both models for 200 epochs (from the discriminator point of view), which means that the WGAN generator was updated five times less than the DCGAN generator.
We also used adam optimizer for both using parameters $\alpha = 2 \times 10^{-4}, \beta_1 = 0.5, \beta_2 = 0.9$ As we can see in figure 22 we got images of lower quality for the WGAN which can be expected due to the difference in number of generator iteration. The images generated by the WGAN also look blurrier
The WGAN was also longer to train, due to the different training loops and the calculation of the gradient penalty.

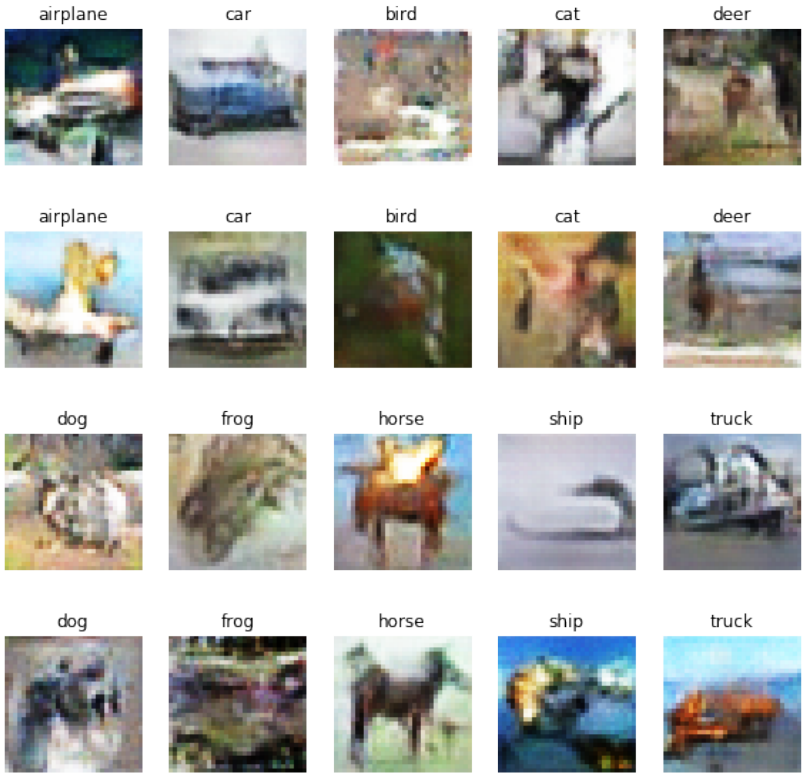

Figure 23: WGAN trained on CIFAR-10 for 500 epochs

In order to get similar results we would have to train our WGAN for longer, we can see in figure 23 the results after continuing the training of our WGAN to 500 epochs, our results did not improve much, they even got worse in some cases. The quality did not really catch up with our DCGAN, even though it trained for 2.5 times longer.

Better results are obtainable with a WGAN but we wanted it to be as close as possible to our DCGAN to compare both of them with the loss being the major difference. You can find more WGAN generated samples in [13] were they used WGANs on various dataset.

Still the absence of mode collapse and interpretable loss make WGANs a compelling alternative to traditional GANs. But with the previously mentioned improvements (minibatch discrimination, one sided label smoothing ...) we manage to reduce mode collapse and close the gap in image quality between a simple DCGAN and a WGAN.

6 Conclusion

We hope that this guide helped you get a better grasp on GANs and their improvements.
We could only analyze a part of all the improvements on GANs and we encourage you to look at others methods that have been developed and that may better suit your needs.

References

[1] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research),"

[2] Y. LeCun and C. Cortes, "MNIST handwritten digit database,"

[3] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," *arXiv e-prints*, p. arXiv:1406.2661, Jun 2014.

[4] M. Mirza and S. Osindero, "Conditional Generative Adversarial Nets," *arXiv e-prints*, p. arXiv:1411.1784, Nov 2014.

[5] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *arXiv e-prints*, p. arXiv:1511.06434, Nov 2015.

[6] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv e-prints*, p. arXiv:1603.07285, Mar 2016.

[7] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *arXiv e-prints*, p. arXiv:1502.03167, Feb 2015.

[8] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved Techniques for Training GANs," *arXiv e-prints*, p. arXiv:1606.03498, Jun 2016.

[9] I. Goodfellow, "NIPS 2016 Tutorial: Generative Adversarial Networks," *arXiv e-prints*, p. arXiv:1701.00160, Dec 2016.

[10] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," *arXiv e-prints*, p. arXiv:1701.07875, Jan 2017.

[11] M. Arjovsky and L. Bottou, "Towards Principled Methods for Training Generative Adversarial Networks," *arXiv e-prints*, p. arXiv:1701.04862, Jan 2017.

[12] J. Lei Ba, J. R. Kiros, and G. E. Hinton, "Layer Normalization," *arXiv e-prints*, p. arXiv:1607.06450, Jul 2016.

[13] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, "Improved Training of Wasserstein GANs," *arXiv e-prints*, p. arXiv:1704.00028, Mar 2017.