INF473V PROJECT

ARTGAN: ARTWORK SYNTHESIS WITH CONDITIONAL CATEGORICAL GANS

June 4, 2020

Bianca Marin Moreno

1 WHAT IS ARTGAN?

This project is based on the implementation of the algorithme proposed by the paper [1]. The article proposes some improvements to the method of Generative Adversarial Networks (GANs) in order to generate more complexe images like abstract artworks. Usually GANs work pretty well with datasets like CIFAR-10 or MNIST, sets that usually have only one object in the image, a distinguishable background and shapes well defined. What ArtGAN proposes is to create more challenging images, with abstract shapes and objects, and non defined background. Artworks usually have these caracteristics, for example Abstract paintings usually don 't represent a real object, Cubisme sometimes don't have a distinguishable background, or Impressionisme is all about using non natural shapes.

To full fill its goal, the key innovation of the algorithme proposed is to feed an additional information (the label), to the GAN network. The objective is to make back-propagation of the loss function with respect to the labels from the discriminator to the generator, in order to better train the generator allowing it to create more complex images. This idea comes from a study about how human beings learn how to paint: they need to focus in a particular type of skill at a time. Normal GANs usually don't learn by focusing in a particular subject, but if we use the labels, the algorithme will learn by concentrating in one thing at a time, and by that, it will follow the same patterns that humans do when learning to paint.

The method basically consists of feeding the Generator (G) not only a noise vector, but also a random label (that will be the real label for the generated image), and after use this information to back propagate the errors to G. Also, the algorithme feeds the Discriminator (D) an image, and instead of D giving a binary output (classifying the image as real or generated), it will return a probability distribution of the image being in $K+1$ different classes (K being the number of classes of the dataset, and one is representing a fake class). Finally, the article also proposes to add the $L2$ pixel-wise reconstruction loss when training G to help improving the quality of the images.

The ArtGAN method proposes, in a sort of way, a combination of the algorithmes of CondGAN and CatGAN, but with slightly differences. CondGAN also feeds an additional vector to G, however, it does the same to D, and it doesn't allow the feedback from this additional information to the intermediate layers. As for CatGAN, it also proposes to let D returns a vector of size $K+1$, but it doesn't use this information while training G.

2 IMPLEMENTATION

2.1 Generator and Discriminator

As excepected, the first two main blocks of the implementation consists of a Generator and a Discriminator. The Discriminator takes as input an image of size 3@64x64 and give as an output a vector of the size of the number of classes more one fake class. It is made of five convolutional layers, each one except the first followed by a batch normalisation layer, and in the end it has one fully connected layer. For the activation function I use LeakyReLU (a variation of ReLU that allows values less than zero). The final activation is a Sigmoid, to transform the output into results between 0 and 1.

We can think about the Discriminator in two main parts. First it is an encoder made by the first four convolutional layers, and is responsible to transform the image into a latent space where it will have the size of 512@8x8, and after that it is a classifier, that will classify the image as being in one of the classes of the dataset, or a fake image.

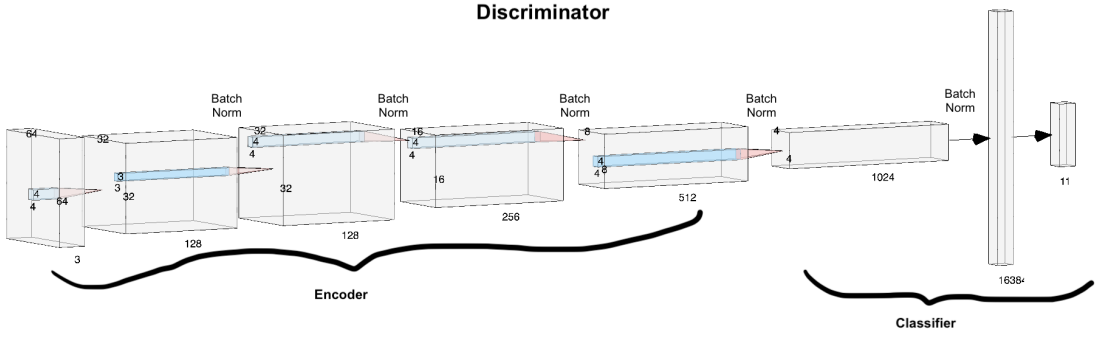


Figure 1: Discriminator architecture for CIFAR-10

As for the generator, it takes as input a noise vector that follows a normal distribution with mean value 0, and variation 1, that I set the size to be 100, concatenated with a random label vector (a one-hot vector, that for a value k of label values 1 in the kth element, and 0 on the others). Both the noise and the random label are created by distinct functions in the code. The generator output is an image of size 3@64x64. It consists of six deconvolutional layers, all except the last one, followed by a batch normalisation. The activation function is ReLU, and the final activation is a Sigmoid.

We can also divide the generator in two parts. The first one is a *zNet* that transforms the input vector into a latent space of size 512@8x8, and finally a decoder that transforms the latent space into an image (it does the opposite that the encoder does). The decoder goes from the $3rd$ deconvolutional layer, until the last one.

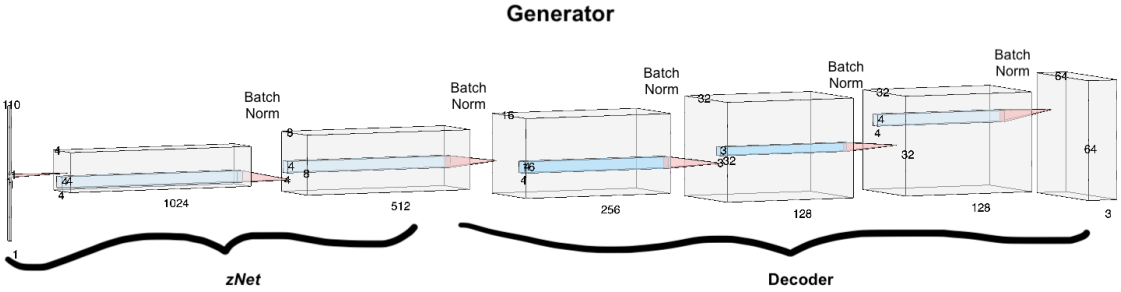


Figure 2: Generator architecture for CIFAR-10

2.2 Loss Function

Before specifying the details of the train implementation, I would like to discuss about the losses I used and some difficulties I had in understanding them.

At first sight the losses were kind of tricky. It wasn't clear how they were really maximising the chance of D to classify the image, and how they were making G generate better images. Also, it wasn't clear how to implement them. But, after some good time starring at them, I ended up seeing the binary cross entropy function behind the expressions, and that settled everything. The target vectors are label vectors in the format of one hot vectors (with an 1 in the element that correspond to the label, and 0 everywhere else), which applies to the type of target needed for the BCELoss.

When feeding D with a real image, we expect it to predict its real label. So we wanna maximise the probability of the classification of D to be equal the real label, which is the same as minimising the first part of the D loss expression below, that is the same as minimising the binary cross entropy loss using as target the real image label

$$\begin{aligned} = - \mathbb{E}_{(\mathbf{x}_r,k)\sim p_{\text{data}}} [&\log p\left(y_i|\mathbf{x}_r, i=k\right) \\ &+ \log\left(1 - p\left(y_i|\mathbf{x}_r, i\neq k\right)\right)] \end{aligned}$$

And when feeding D with a generated image, we expect it to predict that it is a fake image. So we want to maximise the chance of D to give as an output the label of a fake image, one of zeros everywhere except in the last element - the Fake Class element. Which is the same as minimising the second part of the D loss expression below, that is the same as minimising the binary cross entropy loss using as target the fake image label.

$$-\mathbb{E}_{\hat{\mathbf{z}}\sim p_{\text{noise}},\hat{k}\sim\mathbf{K}}\left[\log\left(1-p\left(y_i|G\left(\hat{\mathbf{z}},\hat{\mathbf{y}}_{\hat{k}}\right),i<K+1\right)\right)\right.$$
$$\left.+\log p\left(y_i|G\left(\hat{\mathbf{z}},\hat{\mathbf{y}}_{\hat{k}}\right),i=K+1\right)\right]$$

The G loss is actually the trickiest of them all, because it is trying to maximise the chance that D is wrong! It's here that all the magic happens. What it will be doing is to compare the output given by D when fed with a fake image, with the real label of this image, the one we give to G as an input along with the noise vector! The generator will be updated in a way that minimises the loss between these two vectors, that is, it will try to make D more and more willing to classify a fake image as being part of a real class. This gives us the same as minimising the G loss expression below, that is the same as minimising the binary cross entropy loss using as target the real label of the fake image.

$$\mathcal{L}_{adv}=-\mathbb{E}_{\hat{\mathbf{z}}\sim p_{\text{noise}},\hat{k}\sim\mathbf{K}}\left[\log p\left(y_i|G\left(\hat{\mathbf{z}},\hat{\mathbf{y}}_{\hat{k}}\right),i=\hat{k}\right)\right.$$
$$\left.+\log\left(1-p\left(yi|G\left(\hat{\mathbf{z}},\hat{\mathbf{y}}_{\hat{k}}\right),i\neq\hat{k}\right)\right)\right]$$

Finally, to implement the $L2$ loss proposed by the article to improve the training stability, we use the encoders and decoders to reconstruct the real image, and we compare both real and reconstructed images using the function torch.mean. The final G loss is the sum of the $L2$ loss and the one described above (adversarial loss).

$$\mathcal{L}_{L2}=\mathbb{E}_{\mathbf{x}_r\sim pdata}\left[\|Dec\left(\text{Enc}\left(\mathbf{x}_r\right)\right)-\mathbf{x}_r\|_2^2\right]$$

2.3 Training

First of all, the optimiser proposed by the article is the RmsProp, it takes the square root of the gradient average before adding epsilon. We have also a function dedicated to lowering the learning rate during the training process, which uses torch.optim.lrscheduler.ExponentialLR. The advantages of a learning rate decay is that it can help the algorithme to converge faster and with a higher accuracy.

It's also important to mention that I created the fake one hot vector that is used to calculate the D loss separately, and I also have a fixed noise-label vector to feed G at each epoch to keep track of its progress by printing the images it generates.

The train consists of a larger *for* responsible for realising all epochs, and a inner one responsible for passing through the whole dataset by using a batch of size 128 (size proposed in the article). I encountered a problem every time the *for* passed through the last batch: this batch didn't have size 128 anymore, and the comparisons that we do all takes in consideration the size 128. So my algorithm always skips the last batch to not face this problem (I also thought about changing the batch size for the last iteration, but I figured that as it corresponded only to a small set of images, it would be faster to only skip the iteration).

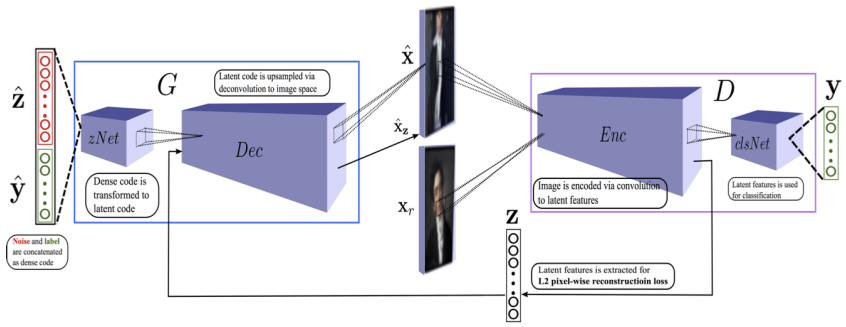


Figure 3: Architecture of ArtGAN. Credits: Wei Ren Tan, Chee Seng Chan, Hernán E. Aguirre, Kiyoshi Tanaka.

For each batch we need to tranform the labels into one hot vector. At first I used torch.scatter, but them I decided to use the functional F.onehot from pytorch that did the job faster. After regularising the labels, we feed D with the real image, create a noise-label input to G and make it generate a fake image, and also feed D with the fake image. Than it is time to compute both real and fake D losses, compute the gradients by applying .backward(), and finally updating D by using .step(). Now it's Gs time. We take a new output of D by feeding it with the generated image again now that it's updated, we reconstruct the real image by using the encoder and decoder, and we use it all to compute Gs loss. Finally, we compute its gradients, and update its weights. We save all losses in a vector in order to make our graphs later, print the losses every 50 times it passes through the inner *for*, and print some images that G is generating at each epoch to keep track of its results.

The only external libraries used were Pytorch, Numpy and Matplotlib.

3 RESULTS

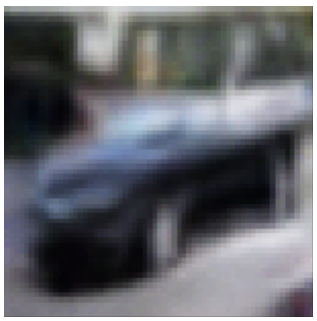
I followed the article suggestion and used a learning rate of 0.001, a decay rate of 0.9 and an $\alpha = 0.2$ in LeakyReLU.

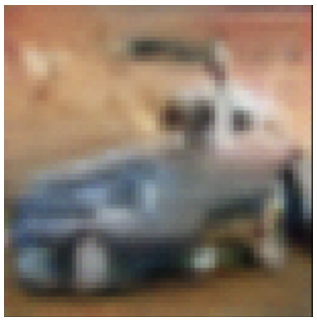
3.1 CIFAR-10

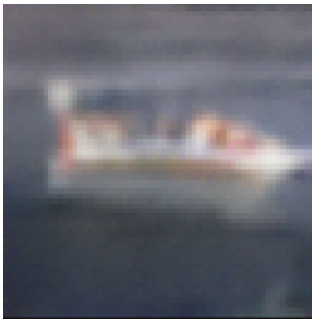
Figure 4: First epochs

Figure 5: Later epochs

(a) Car (b) Car (c) Car (d) Car

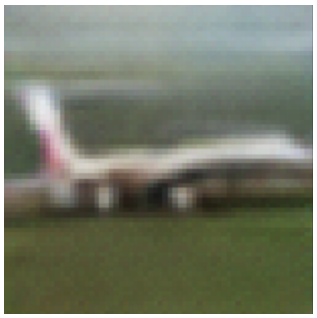

(a) Car

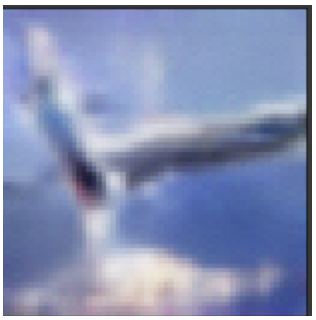

(b) Car

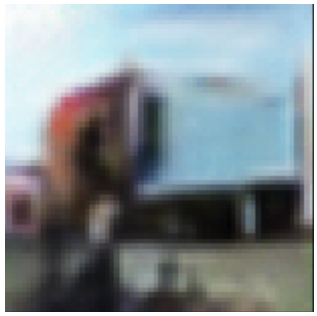

(c) Ship


(d) Ship

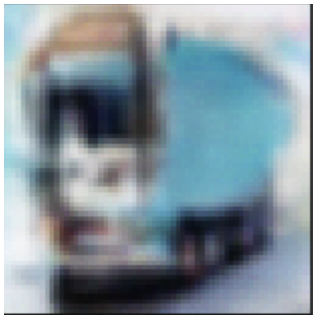

(a) Airplane

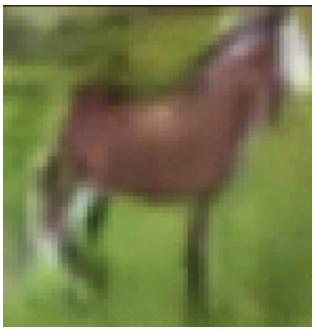

(b) Airplane


(c) Airplane

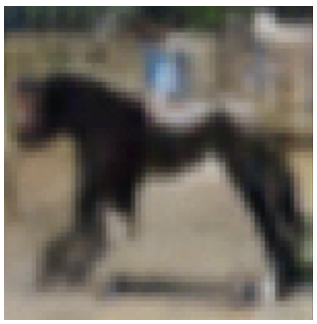

(d) Truck


(a) Truck


(b) Truck


(c) Horse

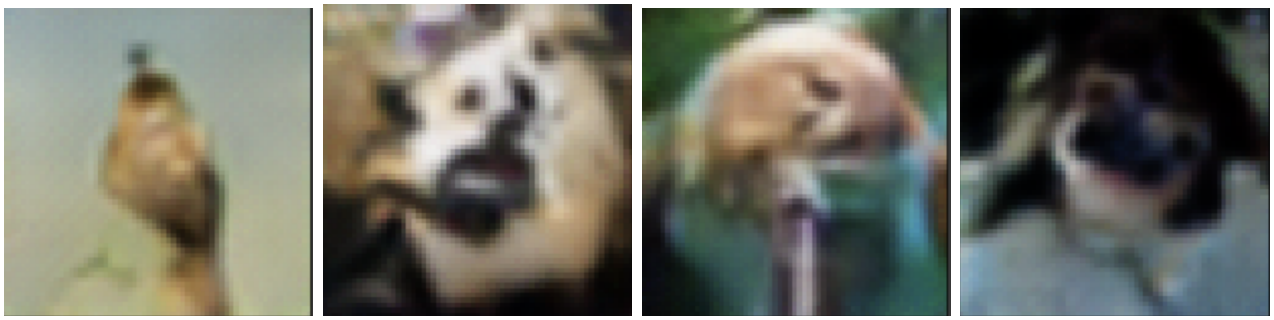

(d) Horse


(a) Horse

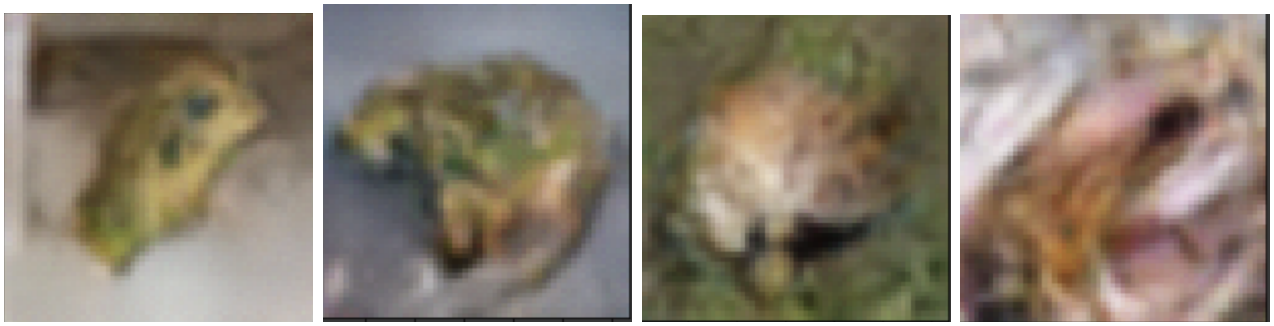

(b) Horse


(c) Bird


(d) Bird



(a) Bird (b) Dog (c) Dog (d) Dog



(a) Frog (b) Frog (c) Frog (d) Frog

(a) Deer (b) Deer (c) Deer (d) Deer

When training with CIFAR-10 I faced a failure mode. After 20 epochs the Discriminator started to suffer an overfitting. Its loss started to go closer to zero, consequently the Generator loss started to get bigger and bigger, and the images started to loose definition. To overcome this problem I applied two tricks. The first one was to introduce some noise into the labels that are used to calculate the D loss. The idea is to, at each batch, flip some labels of the real images into fake images labels, and to change some fake labels of the fake images into random real ones. I thought about it as a way to fool D and preventing it from getting too good too quickly. I flipped a label with a probability of 0.05.

The second trick was to add some noise following a normal distribution with mean value 0 and variation 1 into the real image. Both tricks helped to prevent D from overfitting in 20

epochs, however, I still faced the same problem with approximately $55-65$ epochs and I didn't manage to get better than this.

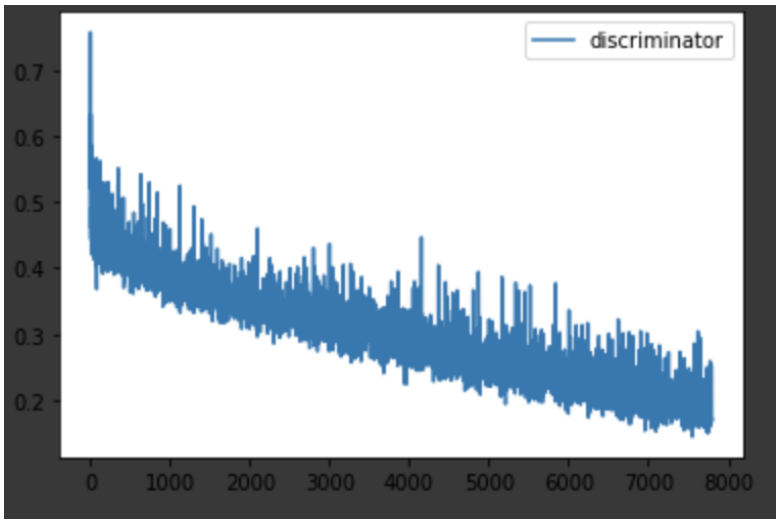


Figure 13: Discriminator loss for CIFAR-10

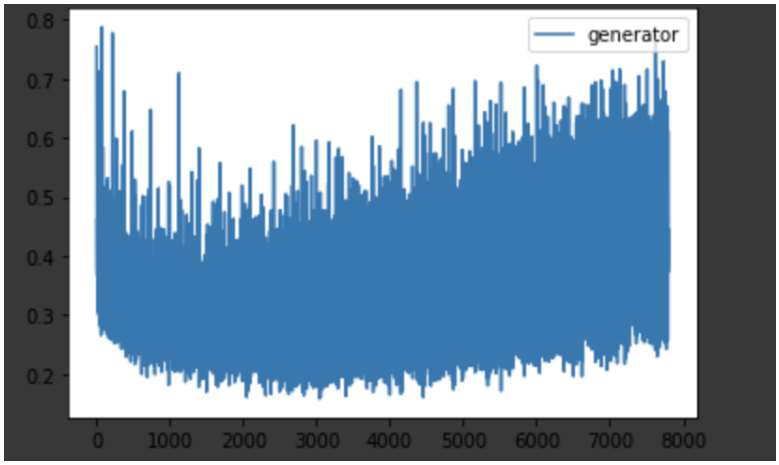


Figure 14: Generator loss for CIFAR-10

3.2 WikiArt

Unfortunately, due to the coronavirus I came back to Brazil and my internet here oscillates a lot and is not powerful enough to download the WikiArt dataset (I tried several times and all went wrong - or it would take more than a day to dowload and my internet would stop working in the middle of the process, or it wouldn't even start downloading). So I didn't manage to do the principal training of the project.

4 CONCLUSION

To summarise, the implementation of ArtGAN consisted finally in an implementation of a GAN that uses the feedback from the label during the back-propagation step. For CIFAR-10 the algorithme succeeded in showing the right shape for some pictures, specially the car. However, some pictures still looked poor in shape and definition, probably because of the huge number of shapes present in CIFAR-10.

In order to overcome the failure modes in CIFAR-10, we need to find more ways to trick the discriminator, so my next tentatives would probably involve adding some dropout layers in the D model, or to mixture some fake images into the real image batch and vice versa. I'm also curious to see how my implementation would work with WikiArt. As it is a dataset without well defined shapes, I imagine that the faillure mode I faced with CIFAR-10 wouldn't happen with WikiArt. Now for the method itself, maybe it could work better with deeper neural networks for the generator and the discriminator.

To conclude, in my view, finding better ways to generate images is useful not only for the challenge of creating pictures made by a machine, but also because this task is in a certain way a study about probability distribution in higher dimensions. If we think about, for example, a dog image as a huge vector that follows a certain probability distribution, knowing how to recreate this image is the same as learning its distribution, and that is an important mathematical skill with a lot of others applications.

REFERENCES

[1] Wei Ren Tan, Chee Seng Chan, Hernán E. Aguirre et Kiyoshi Tanaka : Artgan: Artwork synthesis with conditional categorial gans. *CoRR*, abs/1702.03410, 2017.