Chapter – 3 (implementation – Part 3 – Train & Output)

**Computer Vision**

**GANs: Image Generation**

Training intro

Train Discriminator (update weights)

Calculate the Error

Train Generator (update weights)

Printing losses and Real Images

**3.8. Training intro**

This training implementation will be broken down into two big steps:

1. First step: Update the weight of the discriminator NN. We want to train the discriminator to understand what's real and what's fake.This first step can be broken down into two smaller steps.

* We'll first train it with a real image and we'll set the ***target*** to ***1*** because ***one*** means that the image is ***accepted***.
* Then we'll do another training by giving it a ***fake image*** (a fake image created by the ***generator***) setting the ***target*** to ***0*** because ***zero*** means that the image is ***not accepted***.

1. Second Step: Update the weight of the generator NN. We'll take the fake image again which will be in one step of the FOR-loop, we'll feed this fake image into the discriminator, to get the output (i.e. the discriminating value between 0 and 1).

* But then we'll set a **new** **target** to **1**. The **target** will be equal to **1** **always**.
* We'll compute the loss between the output (the value between 0 and 1) of the discriminator and the *fixed target =* ***1***.
* Note:
* This **target** always equal to **1**.
* We'll back-propagate this ERROR not back inside the discriminator but back inside the generator.
* The key thing to understand is:

1. The *error* is calculated between the *prediction* of the *discriminator* (value between 0 and 1) and the *fixed* *target* **= 1** and
2. The ERROR will back-propagate inside the neural network of the generator.

* We'll apply the Stochastic Gradient Descent (SGD) to update the weight of the neural network of the Generator.

|  |  |
| --- | --- |
| * Criterion: This criterion will measure the error of prediction. * The **prediction** will be a value between 0 and 1 which is a discriminating number. i.e. * The **ground-truth** will only be **0** or **1**. **0** means **fake** and **1** means **real**. * The criterion is an object of the **BCELoss()**. * In PyTorch documentation we can see that **BCELoss** given by following formula: this is a special kind of loss, it is perfect to train at the adversarial networks. BCE means Binary Cross Entropy. |  |

* It is also used for auto-encoders.
* The important thing is: The targets will either be
* **0**: when we want to try to discriminate or to recognize a fake image or
* **1**: when we train the discriminator to recognize a real image.
* Optimizers: We need two optimizers, one for the generator and another one for the discriminator.
* Optimizer Of The Discriminator: We'll get it from **torch.optim** library, we'll get the Adam-optimizer which is a highly advanced optimizer for SGD.
* Arguments:
* **netD.parameters()**: We'll use our NN objects of the discriminator that we called **netD**.
* **lr**: Learning rate, we set **0.0002**.
* **betas**: It's a touple of two values which are actually coefficients for computing running averages of gradient and it's square. We'll choose **0.5** and **0.999** and it's based on experimentation.
* Optimizer of the generator: We just need to change the name **netD** to **netG**, and all parameters are the same.

optimizerD = **optim.Adam**(**netD.parameters**(), lr = 0.0002, betas = (0.5, 0.999))

optimizerG = **optim.Adam**(**netG.parameters**(), lr = 0.0002, betas = (0.5, 0.999))

* FOR-loop: Now we start loop over the different epochs. We're going to have 25 epochs. In each of these 25 epochs we'll go through all the images of the dataset.

**for** epoch **in** **range**(25): # We iterate over 25 epochs.

**for** i, data **in** **enumerate**(dataloader, 0): # We iterate over the images of the dataset.

* Nested FOR-loop: **i** is the **index** of the loop and **data** contains the mini batch of images.
* The whole data set of all the images is breaks into mini batches and **data** will be each of these different mini batches.
* We're going to use **enumerate()** function to get these separate mini batches of the dataset. Inside **enumerate()** we have to put our **dataloader** which will get us the mini batches.
* We also have to specify where the ***index*** of the loop is ***going to start*** from and we'll start from zero.

Next we'll update the weights of the Neural Network the Discriminator.

# *Training the DCGANs*

criterion = **nn.BCELoss**() # *the criterion object will measure the error between the prediction and the target.*

# *We create the optimizer object of the discriminator.*

optimizerD = **optim.Adam**(**netD.parameters**(), lr = 0.0002, betas = (0.5, 0.999))

# *We create the optimizer object of the generator.*

optimizerG = **optim.Adam**(**netG.parameters**(), lr = 0.0002, betas = (0.5, 0.999))

**for** epoch **in** **range**(25): # *We iterate over 25 epochs.*

**for** i, data **in** **enumerate**(dataloader, 0): # *We iterate over the images of the dataset.*

**3.9. Train Discriminator (update weights)**

* We're going to tackle this into three subsets.

1. The first step is to train the discriminator with a real image of the data set.
2. The second step is to train the discriminator with a fake image generated by the generator.
3. Finally we'll back-propagate the total error which will be the *sum of the errors* of these two previous trainings.

* We train the discriminator with both a real image (from dataset) and a fake image generated by the generator, because we want to train the discriminator to see and understand what's real and what's fake.
* To do that, we need to give the discriminator the two different ground truths: the ground truth of what's real (real image from dataset) and the ground truth of what's fake (image generated by the generator).
* Initialize Gradient: Before we start the training of the discriminator, we need to initialize the GRADIENT of the discriminator with respect to the weight to zero.

**netD.zero\_grad**()

* Sub-step 1 (Training the discriminator with a REAL image) Get the real images: We're going to get a mini batch of real images and that's because a NN actually accept as inputs some mini batches of single inputs like single images.
* Since in this FOR-loop we already iterated through some mini batches using enumerate(dataloader, 0), we already have these mini batches.

real, \_ = data

* We use the first element of our mini-batch ***data***.
* Right now we're dealing with a specific mini batch which is ***data*** and ***data*** is *composed of two elements*.
* The first elements are the real images themselves and the second element are the labels. For now, we *don't really care* about the *labels*.
* Wrapping: But it's not ready yet because PyTorch NN only accept the input in Torch variable that contains both a ***tensor*** and a ***gradient***.
* Right now we have a *tensor of images* but we need to wrap it into a Torch variable to associate it with a gradient. This will be the *input* of the *neural network*. To do this we use **Variable()** from **'torch.autograd'** and this input is going to be an object of the Variable class.

**input** = **Variable**(real)

Now are input images are in to mini-batch and also a torch-variable.

* Target: We're going to have 2 targets for the different training.
* At first we are train the discriminator with a real image of the data-set, for each of the *real image* of the *mini-batch* we need to set the ***target*** to ***1***. We use ***1***, because we need to specify the ground truth to the discriminator that the image is real (and 0 means fake).
* So we create a tensor of the *size* of the *mini-batch* that is going to be *composed* *of only 1's*. That's how we will have a 1 for each of the input image of the mini-batch.

target = **Variable**(**torch.ones**(**input.size**()[0]))

* we use **torch.ones()** to create the tensor.
* **input.size**()[0] is the size of the mini-batch.
* Also we apply **Variable** class to make the tensor a torch-variable.

Indeed we're going to compute some gradients of the target as well and therefore we need to attach this target tensor to a gradient inside a torch variable.

        # *--------    1st Step: Updating the weights of the discriminator NN    --------*

**netD.zero\_grad**() # *We initialize to 0 the gradients of the discriminator with respect to the weights.*

        # *Training the discriminator with a real image of the dataset*

        real, \_ = data      # *get a real image of the dataset which will be used to train the discriminator.*

**input** = **Variable**(real) # *We wrap it in a variable, used it from 'torch.autograd'.*

        target = **Variable**(**torch.ones**(**input.size**()[0])) # *We get the target.*

        # *forward propagate 'real image' into the discriminator NN to get the prediction (a value between 0 and 1).*

        output = **netD**(**input**)

        # *We compute the loss between the predictions (output) and the target (equal to 1).*

        errD\_real = **criterion**(output, target)

* Outputs (forward propagate 'real image' into the discriminator NN to get the prediction): We take our Discriminator Neural network **netD** and apply **input** the **Torch-Variable** as an argument (we feed the netD with our input). The **input** is a Torch Variable of a mini batch of real images.

output = **netD**(**input**)

* that forward propagates the *real input images* of the mini match inside the *Neural Network of the Discriminator* to get the prediction of (for each of these real inputs images) whether they should be accepted or not.
* The prediction is a number between **0** and **1**.
* Close to zero means reject and close to 1 means accept.
* Error: Since we have **target** full of **1**'s and the outputs. Well of course, we're going to get the error.
* The first error coming from this first training of the discriminator with the real image ground trouth. We call this **errD\_real**.

errD\_real = **criterion**(output, target)

* To get this error, we'll use our **criterion** object (from **nn.BCELoss**()). It will calculate the "loss-error" for us. It will compute the loss error between the output and the target.
* We're also going to have another error for the generator in the second big step of the training.

We have our loss-error of the discriminator the corresponding to the training of the discriminator with the real images, to train it to understand/recognize a real image.

* Sub-step 2 (Training the discriminator with a FAKE image):
* In the second step of the training of the discriminator, we'll train it with a fake image generated by the *generator*. We want the discriminator to recognize fake images.
* We'll do the same process: Get the input, then target & then the output. After these we'll generate a ***loss*** for the ***fake*** image (**errD\_fake**).
* Finally we'll get the total error (errD\_real + errD\_fake).
* And then we'll do the back propagation of this total error back inside the Neural Network of the Discriminator.

# *sub-step 2 : Training the discriminator with a fake image generated by the generator*

noise = **Variable**(**torch.randn**(**input.size**()[0], 100, 1, 1))   # *We make a random input vector (noise) of the generator.*

fake = **netG**(noise)  # *We forward propagate random input vector into the GENERATOR NN to get some fake generated images.*

target = **Variable**(**torch.zeros**(**input.size**()[0]))     # *We get the target.*

# *forward propagate the fake generated images into the DISCRIMINATOR NN to get the prediction (a value between 0 and 1).*

output = **netD**(**fake.detach**())

errD\_fake = **criterion**(output, target) # *We compute the loss between the prediction (output) and the target (equal to 0).*

* Getting the inputs: Previously we used a mini batch of real images, but this time we won't use a mini batch of fake images.
* Because the generator is like an inverted CNN, in the architecture of the generator we used inverted convolution that takes as input a random vector of size **100**.
* And since *CNN* takes some images as input and returns a *flattened* vector of *one dimension*.
* The Inverted CNN of the Generator will do exactly the opposite it will take *a vector of one dimension* as *input* and we'll return the fake images.
* Therefore to get a fake image from the Generator, we need to create a vector of size **100**. This will be a random vector and that will represent some noise. After feeding the generator with this random vector, it will return some fake images.
* At the *beginning*, the generator will *return some images that look like nothing* but over the epochs we will ***update*** the ***weights*** so that the ***generated images*** look like some ***real images***.
* But before doing that we need to train the discriminator to recognize what's fake.
* Getting the noise: To create random values of a specific size we'll use **torch.randn**().

noise = **Variable**(**torch.randn**(**input.size**()[0], 100, 1, 1))

* Arguments:
* **input.size**()[0] is the batch-size. It creates a mini-batch of random vectors of size 100. So that we can get a mini-batch of fake-images.
* **100:** the number of elements we want in this vector, because we specify in the architecture of the generator that the input vector should be of size ***100***.
* **1, 1:** To give these random vectors some fake dimensions that will correspond to a feature map we use **1, 1**. Instead of having **100** values in the vector, we'll have 100 feature maps of size . i.e. each of the 100 feature maps will be a matrix of size .
* Wrap inside a Variable: Since, above noise is going to be the input of the generator NN and PyTorch only accept **torch Variables**. So, applied **Variable()**.
* New ground truth: It's a mini batch of fake images. We're going feed above **noise** to our **netG()** object. It will get a mini batch of the same size containing some fake images.

fake = **netG**(noise)

* Target: This target is going to be a *new kind of target*. Before we have **target** full of **1**'s. Now since we're dealing with fake-images, we want to train the Discriminator to recognize the fake images. Hence, the target should be the rejection of the images. And the rejection of the images correspond to ***0***. So, we'll use the **target** full of **0**'s using **zeros()** function.

target = **Variable**(**torch.zeros**(**input.size**()[0]))

* Outputs: To get the output, we feed the Discriminator NN with fake images. But to save some memory we use **detach**(). Since the variable ***fake*** is a torch tensor and it contains not only the fake-images but also the gradients, **detach**() only selects the fake-images, not the gradients (detach the gradient of this fake torch variable to speed up the computation).

output = **netD**(**fake.detach**())

* We're not going to use this *gradient* after *back propagating* the *error* back inside the *Discriminator NN* applying *SGD*.
* We *absolutely don't care* of the *gradient* of the output with *respect* to the *weight* of the *generator*.
* It will not be part of the computation in SGD.
* Error for the fake: We have the *target* and the *output*, now we calculate the *new ERROR* corresponding to the training of the discriminator with the fake images.

errD\_fake = **criterion**(output, target)

Now we are done with the *two trainings* that we had to do with the *discriminator*. We trained the discriminator to recognize *real images* and *fake images*.

Next sub-step is the back propagating the total error back into the discriminator NN.

**3.10. Calculate the Total-Error & back-propagate**

* The total Error is:

errD = errD\_real + errD\_fake

* We'll back-propagate this total error back into the Discriminator NN and then ***update*** the ***weights*** through ***SGD*** according to how much they're responsible for the error.
* Back propagate: We'll use the PyTorch method **backward**() to our total error **errD**.

**errD.backward**()

* Apply SGD: To update the weight we need to apply SGD, to do that we take our optimizer of the discriminator **optimizerD** (which we defined at the beginning of the training phase) and apply the **step()** method.
* **step()** function applies the optimizer on the Discriminator NN to update the weights of the *Discriminator* according to how much they're responsible for the total loss error (errD\_real + errD\_fake).

# *Backpropagating the total error*

errD = errD\_real + errD\_fake # *We compute the total error of the discriminator.*

# *backpropagate the loss error by computing the gradients of the total error with respect to the weights of the discriminator.*

**errD.backward**()

# *We apply the optimizer to update the weights according to how much they are responsible for the loss error of the discriminator.*

**optimizerD.step**()

**3.11. Train Generator (update weights)**

We'll update the weights of the Generator NN. This time there will only be ***one error*** the ***loss error*** between the *prediction* of the *discriminator* whether or not the generated image should be accepted (yes/no).

* The key point to understand is that the target will be equal to **1**. Even if this time the input of the discriminator will be the fake image of the generator.
* The **target** will be **1**, because this time we want the *generator to have some weights* that produce some images that look like real images and therefore we want to ***push the production*** close to a ***target*** of ***one (1)***.
* This time we're *training the brain* of the *generator* to be able to *generate* some images that look like *real images*.
* Initializing the gradient of the generator: We're going to initialize the gradient of the generator with respect to the weights to zero. The same thing we did for the gradient of the discriminator.

**netG.zero\_grad**()

* Input: We already have the inputs from the *previous step of training the discriminator*. The input is going to be the fake image (mini batch of fake images) that we generated to train the discriminator, we'll use those same mini-batches.
* We already have the fake images of the mini batch from the previous step (training the discriminator)
* Targets: This time we want to push the predictions to one because we want to generate realistic images so that the discriminator will accept that those fake images as real images.
* So the target for all the input fake images of the mini batch should be all 1's.

target = **Variable**(**torch.ones**(**input.size**()[0])

It's wrapped into a **torch** Variable.

* Output: The next step is to get the output. It's the *output of the discriminator* when the inputs are ***fake images***.

output = **netD**(fake)

* We take our *Discriminator NN* ***netD()*** and feed with the *mini-batch* of *fake input images*.
* For each of these *fake images* we'll get a *discriminating number* between ***0*** and ***1***. If this number is close to zero, the image will be rejected. And if this number is close to 1 the image will be accepted.
* Remember that in the ***previous output*** (previous step *of training the discriminator*) we ***detached*** the gradient of ***fake***.
* This time we're not going to do it because using those gradients we're going to update the weights of the generator NN.
* Error Of Prediction: This error of prediction is going to be the error related to the generator because we will back propagate this error back into the neural network of the generator.
* In the previous step, we back propagated the total error (**errD**) back to the discriminator NN. But in this case we'll back-propagate ***only*** the ***error of prediction***. That’s why we'll call this error **errG** instead of **errG\_fake**.

errG = **criterion**(output, target)

We'll use our ***criterion*** and ***target*** & ***output*** as arguments.

* Back propagation: Now we back propagate errG in the neural network of the generator.

**errG.backward**()

It only computes the gradients.

* Applying optimizer : We apply the optimizer of the generator to make sure that the weight of the generator will be updated.

**optimizerG.step**()

# *--------    2nd Step: Updating the weights of the neural network of the generator    --------*

# *We initialize to 0 the gradients of the generator with respect to the weights.*

**netG.zero\_grad**()

# *We get the target.*

target = **Variable**(**torch.ones**(**input.size**()[0]))

# *forward propagate the generated images into discriminator NN to get the prediction (a value between 0 and 1).*

output = **netD**(fake)

# *We compute the loss between the prediction (output between 0 and 1) and the target (equal to 1).*

errG = **criterion**(output, target)

# *We backpropagate the loss error by computing the gradients of the total error with respect to the weights of the generator.*

**errG.backward**()

# *We apply the optimizer to update the weights according to how much they are responsible for the loss error of the generator.*

**optimizerG.step**()

Next, we're gonna print the losses inside the loop, save the real images and also the fake images.

**3.12. Printing losses and Real Images**

In this step we're going to save the real and generated images in a folder called "result". We're going to check if generated image looks like something or not. We'll also print the no. of epochs and steps.

# *--------    3rd Step: Printing losses, saving real & generated images of the minibatch every 100 steps    --------*

# *We print les losses of the discriminator (Loss\_D) and the generator (Loss\_G).*

**print**('[%d/%d][%d/%d] Loss\_D: %.4f Loss\_G: %.4f' % (epoch, 25, i, **len**(dataloader), errD.data[0], errG.data[0]))

**if** i % 100 **==** 0: # *Every 100 steps:*

    # *We save the real images of the minibatch.*

**vutils.save\_image**(real, '%s/real\_samples.png' % "./results", normalize = **True**)

    fake = **netG**(noise) # *We get our fake generated images.*

    # *We also save the fake generated images of the minibatch.*

**vutils.save\_image**(fake.data, '%s/fake\_samples\_epoch\_%03d.png' % ("./results", epoch), normalize = **True**)

* Epoch, Steps and Errors: We print epoch, steps and errors in *epoch/25 step/total\_data\_length Loss\_D: Loss\_G* pattern. Losses are in 4 decimals precision.

**print**('[%d/%d][%d/%d] Loss\_D: %.4f Loss\_G: %.4f' % (epoch, 25, i, **len**(dataloader), errD.data[0], errG.data[0]))

* Saving the images: We want to save the real images and the generated images of the mini-batch after every 100 steps. We use an **IF** condition for that.
* Save real images: To save the images we use **torchvision.utils**, which we renamed as **vutils**, and we use the **save\_image** function.

**vutils.save\_image**(real, '%s/real\_samples.png' % "./results", normalize = **True**)

Arguments:

**real:** batch of the real images

**path:** %s is for the root-directory which is "./results".

**normalize:** we normalize the images, so we used True.

* Save Fake images: First we generate the fake image from the generator by feeding it the **noise**.

fake = **netG**(noise) # *We get our fake generated images.*

To save the fake image we use **torchvision.utils** as **vutils** again.

**vutils.save\_image**(fake.data, '%s/fake\_samples\_epoch\_%03d.png' % ("./results", epoch), normalize = **True**)

Arguments:

**fake.data:** containd the fake images.

**path** '%s/fake\_samples\_epoch\_%03d.png': **%s** for the path , **%03d** for the epoch, to know from which epoch they are coming.

**normalize:** we normalize the images, so we used True.

**All code at once**

# *---------------    DC GANs    ---------------*

# *Deep Convolutional GANs*

# *Importing the libraries*

**from** \_\_future\_\_ **import** print\_function

**import** torch

**import** torch.nn **as** nn

**import** torch.nn.parallel

**import** torch.optim **as** optim

**import** torch.utils.data

**import** torchvision.datasets **as** dset

**import** torchvision.transforms **as** transforms

**import** torchvision.utils **as** vutils

**from** torch.autograd **import** Variable

# *Setting some hyperparameters*

batchSize = 64 # *We set the size of the batch.*

imageSize = 64 # *We set the size of the generated images (64x64).*

# *Creating the transformations*

# *We create a list of transformations (scaling, tensor conversion, normalization) to apply to the input images.*

transform = **transforms.Compose**([**transforms.Scale**(imageSize), **transforms.ToTensor**(), **transforms.Normalize**((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),])

# *Loading the dataset*

# *We download the training set in the ./data folder and we apply the previous transformations on each image.*

dataset = **dset.CIFAR10**(root = './data', download = **True**, transform = transform)

# *We use dataLoader to get the images of the training set batch by batch.*

dataloader = **torch.utils.data.DataLoader**(dataset, batch\_size = batchSize, shuffle = **True**, num\_workers = 2)

# *Defining the weights\_init function that takes as input a neural network m and that will initialize all its weights.*

**def** **weights\_init**(m):

    classname = m.\_\_class\_\_.\_\_name\_\_

**if** **classname.find**('Conv') **!=** -1:

**m.weight.data.normal\_**(0.0, 0.02)

**elif** **classname.find**('BatchNorm') **!=** -1:

**m.weight.data.normal\_**(1.0, 0.02)

**m.bias.data.fill\_**(0)

**class** G(nn.Module): # *We introduce a class to define the generator.*

**def** **\_\_init\_\_**(self): # *\_\_init\_\_() will define the architecture of the generator.*

**super**(G, self).**\_\_init\_\_**() # *We inherit from the nn.Module tools.*

        # *We create a meta module of a neural network that will contain a sequence of modules (convolutions, full connections, etc.).*

        self.main = **nn.Sequential**(

**nn.ConvTranspose2d**(100, 512, 4, 1, 0, bias = **False**), # *We start with an inversed convolution.*

**nn.BatchNorm2d**(512), # *We normalize all the features along the dimension of the batch.*

**nn.ReLU**(**True**), # *We apply a ReLU rectification to break the linearity.*

**nn.ConvTranspose2d**(512, 256, 4, 2, 1, bias = **False**), # *We add another inversed convolution.*

**nn.BatchNorm2d**(256), # *We normalize again.*

**nn.ReLU**(**True**), # *We apply another ReLU.*

**nn.ConvTranspose2d**(256, 128, 4, 2, 1, bias = **False**), # *We add another inversed convolution.*

**nn.BatchNorm2d**(128), # *We normalize again.*

**nn.ReLU**(**True**), # *We apply another ReLU.*

**nn.ConvTranspose2d**(128, 64, 4, 2, 1, bias = **False**), # *We add another inversed convolution.*

**nn.BatchNorm2d**(64), # *We normalize again.*

**nn.ReLU**(**True**), # *We apply another ReLU.*

**nn.ConvTranspose2d**(64, 3, 4, 2, 1, bias = **False**), # *We add another inversed convolution.*

**nn.Tanh**() # *We apply a Tanh rectification to break the linearity and stay between -1 and +1.*

        )

**def** **forward**(self, input): # *We define the forward function that takes as argument an input that will be fed to the neural network, and that will return the output containing the generated images.*

        output = self**.main**(**input**) # *We forward propagate the signal through the whole neural network of the generator defined by self.main.*

**return** output # *We return the output containing the generated images.*

# *Creating the generator*

netG = **G**() # *We create the generator object.*

**netG.apply**(weights\_init) # *We initialize all the weights of its neural network.*

# *Defining the discriminator*

**class** D(nn.Module): # *the discriminator class*

**def** **\_\_init\_\_**(self):     # *\_\_init\_\_() will define the architecture of the discriminator.*

**super**(D, self).**\_\_init\_\_**()   # *We inherit from the nn.Module tools.*

        # *meta module of a NN containing a sequence of modules (convolutions, full connections, etc.).*

        self.main = **nn.Sequential**(

**nn.Conv2d**(3, 64, 4, 2, 1, bias = **False**),    # *We start with a convolution.*

**nn.LeakyReLU**(0.2, inplace = **True**),  # *We apply a LeakyReLU.*

**nn.Conv2d**(64, 128, 4, 2, 1, bias = **False**),  # *We add another convolution.*

**nn.BatchNorm2d**(128),    # *We normalize all the features along the dimension of the batch.*

**nn.LeakyReLU**(0.2, inplace = **True**),  # *We apply another LeakyReLU.*

**nn.Conv2d**(128, 256, 4, 2, 1, bias = **False**), # *We add another convolution.*

**nn.BatchNorm2d**(256),    # *We normalize again.*

**nn.LeakyReLU**(0.2, inplace = **True**),  # *We apply another LeakyReLU.*

**nn.Conv2d**(256, 512, 4, 2, 1, bias = **False**), # *We add another convolution.*

**nn.BatchNorm2d**(512),    # *We normalize again.*

**nn.LeakyReLU**(0.2, inplace = **True**),  # *We apply another LeakyReLU.*

**nn.Conv2d**(512, 1, 4, 1, 0, bias = **False**),   # *We add another convolution.*

**nn.Sigmoid**()    # *We apply a Sigmoid rectification to break the linearity and stay between 0 and 1.*

        )

**def** **forward**(self, input): # *it will take an input that will be fed to the NN, and will return the output between 0 and 1.*

        output = self**.main**(**input**) # *We forward propagate the signal through the whole neural network of the discriminator defined by self.main.*

**return** **output.view**(-1) # *We return the output which will be a value between 0 and 1.*

# *Creating the discriminator*

netD = **D**() # *We create the discriminator object.*

**netD.apply**(weights\_init) # *We initialize all the weights of its neural network.*

# *Training the DCGANs*

criterion = **nn.BCELoss**() # *the criterion object will measure the error between the prediction and the target.*

optimizerD = **optim.Adam**(**netD.parameters**(), lr = 0.0002, betas = (0.5, 0.999)) # *We create the optimizer object of the discriminator.*

optimizerG = **optim.Adam**(**netG.parameters**(), lr = 0.0002, betas = (0.5, 0.999)) # *We create the optimizer object of the generator.*

**for** epoch **in** **range**(25): # *We iterate over 25 epochs.*

**for** i, data **in** **enumerate**(dataloader, 0): # *We iterate over the images of the dataset.*

        # *--------    1st Step: Updating the weights of the discriminator NN    --------*

**netD.zero\_grad**() # *We initialize to 0 the gradients of the discriminator with respect to the weights.*

        # *sub-step 1 : Training the discriminator with a real image of the dataset*

        real, \_ = data      # *get a real image of the dataset which will be used to train the discriminator.*

**input** = **Variable**(real) # *We wrap it in a variable, used it from 'torch.autograd'.*

        target = **Variable**(**torch.ones**(**input.size**()[0])) # *We get the target.*

        output = **netD**(**input**)    # *forward propagate 'real image' into the discriminator NN to get the prediction (a value between 0 and 1).*

        errD\_real = **criterion**(output, target)   # *We compute the loss between the predictions (output) and the target (equal to 1).*

        # *sub-step 2 : Training the discriminator with a fake image generated by the generator*

        noise = **Variable**(**torch.randn**(**input.size**()[0], 100, 1, 1))   # *We make a random input vector (noise) of the generator.*

        fake = **netG**(noise)  # *We forward propagate random input vector into the GENERATOR NN to get some fake generated images.*

        target = **Variable**(**torch.zeros**(**input.size**()[0]))     # *We get the target.*

        output = **netD**(**fake.detach**()) # *forward propagate the fake generated images into the DISCRIMINATOR NN to get the prediction (a value between 0 and 1).*

        errD\_fake = **criterion**(output, target) # *We compute the loss between the prediction (output) and the target (equal to 0).*

        # *sub-step 3 : Backpropagating the total error*

        errD = errD\_real + errD\_fake # *We compute the total error of the discriminator.*

**errD.backward**() # *We backpropagate the loss error by computing the gradients of the total error with respect to the weights of the discriminator.*

**optimizerD.step**() # *We apply the optimizer to update the weights according to how much they are responsible for the loss error of the discriminator.*

        # *--------    2nd Step: Updating the weights of the neural network of the generator    --------*

**netG.zero\_grad**() # *We initialize to 0 the gradients of the generator with respect to the weights.*

        target = **Variable**(**torch.ones**(**input.size**()[0])) # *We get the target.*

        output = **netD**(fake) # *forward propagate the generated images into discriminator NN to get the prediction (a value between 0 and 1).*

        errG = **criterion**(output, target) # *We compute the loss between the prediction (output between 0 and 1) and the target (equal to 1).*

**errG.backward**() # *We backpropagate the loss error by computing the gradients of the total error with respect to the weights of the generator.*

**optimizerG.step**() # *We apply the optimizer to update the weights according to how much they are responsible for the loss error of the generator.*

        # *--------    3rd Step: Printing losses, saving real & generated images of the minibatch every 100 steps    --------*

        # *We print les losses of the discriminator (Loss\_D) and the generator (Loss\_G).*

**print**('[%d/%d][%d/%d] Loss\_D: %.4f Loss\_G: %.4f' % (epoch, 25, i, **len**(dataloader), errD.data[0], errG.data[0]))

**if** i % 100 **==** 0: # *Every 100 steps:*

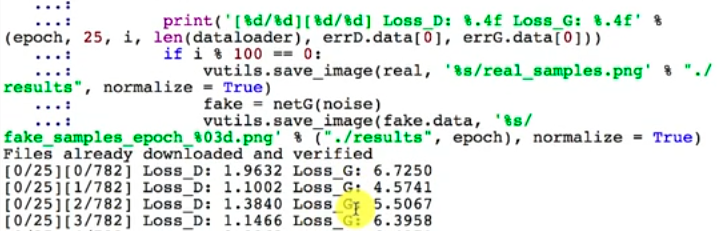
**vutils.save\_image**(real, '%s/real\_samples.png' % "./results", normalize = **True**) # *We save the real images of the minibatch.*

            fake = **netG**(noise) # *We get our fake generated images.*

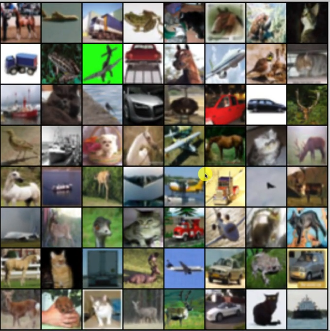
            # *We also save the fake generated images of the minibatch.*

**vutils.save\_image**(fake.data, '%s/fake\_samples\_epoch\_%03d.png' % ("./results", epoch), normalize = **True**)

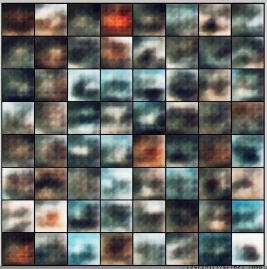
* And finally we run our code!! It can take more or less 8 hours.



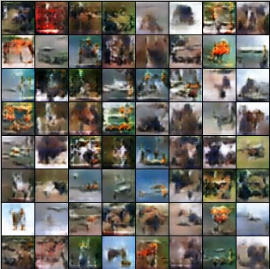
* Real samples:



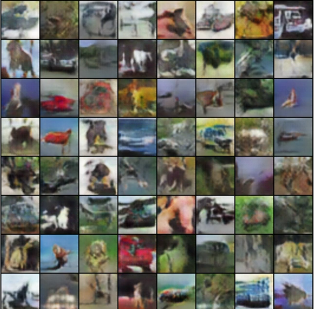
* First fake sample (first epoch): Nothing special.



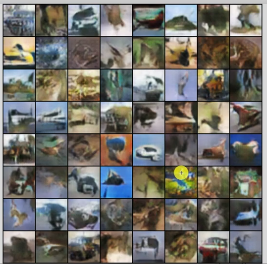
* Third fake sample (third epoch):



* 10 to 15 epoch: Images are recognizable, maybe we don’t need 25 epochs, it could be done with 10-15 epoches.



* 25th epoch: There are some good generated images.



* A special thanks to Alexis Jacq!!

<https://alexis-jacq.github.io/>

<https://github.com/alexis-jacq>